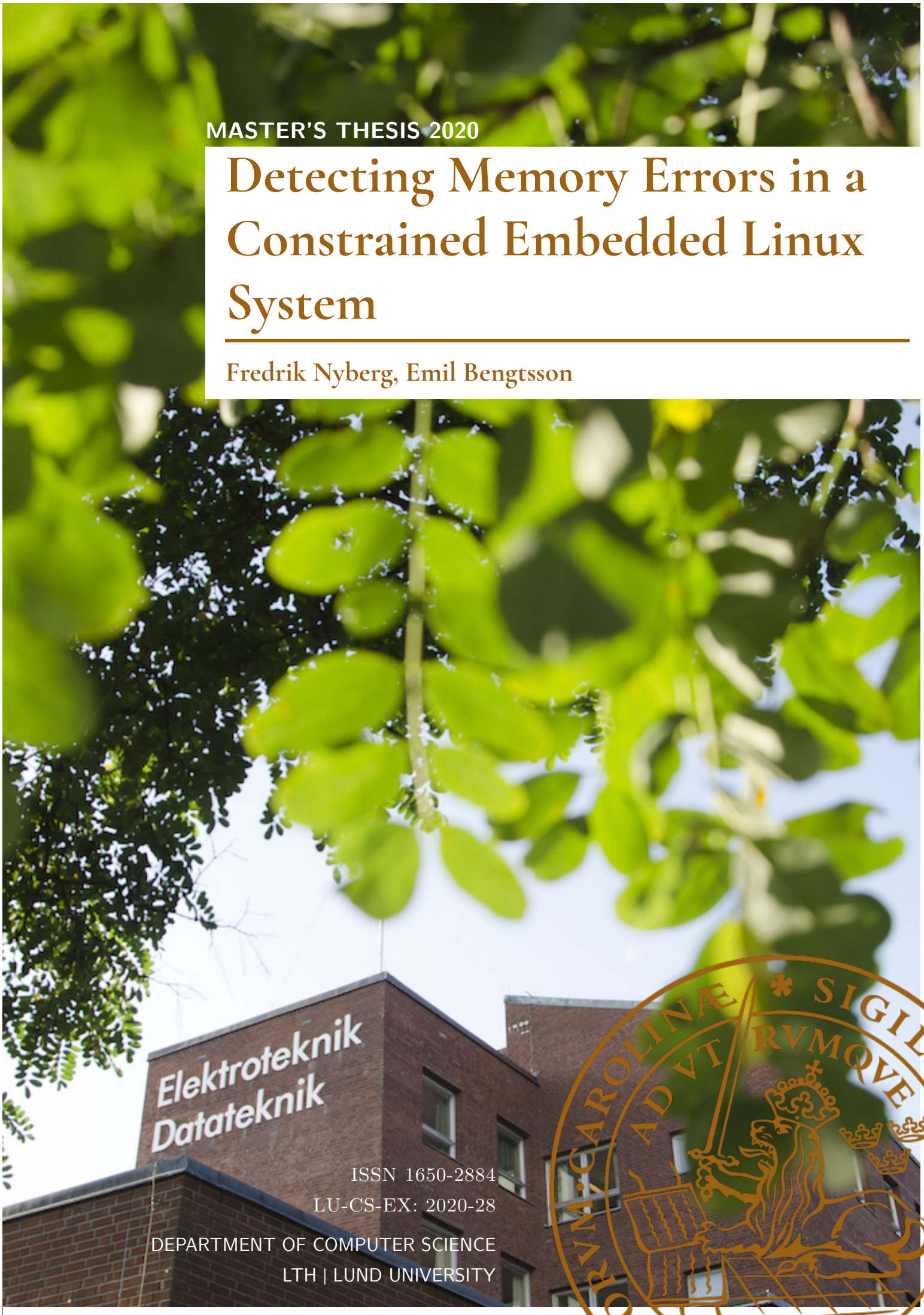


MASTER'S THESIS 2020

Detecting Memory Errors in a Constrained Embedded Linux System

Fredrik Nyberg, Emil Bengtsson

ISSN 1650-2884
LU-CS-EX: 2020-28

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-28

Detecting Memory Errors in a Constrained Embedded Linux System

Fredrik Nyberg, Emil Bengtsson

Detecting Memory Errors in a Constrained Embedded Linux System

(Perspective from the Automotive Industry)

Fredrik Nyberg
htx10fny@student.lu.se

Emil Bengtsson
dat15ebe@student.lu.se

June 22, 2020

Master's thesis work carried out at Volvo Cars.

Supervisors: Flavius Gruian, flavius.gruian@cs.lth.se
Paul Asterland, paul.asterland@volvocars.com

Examiner: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Abstract

Memory errors are the cause of many bugs in software written in C and C++. Embedded systems often run software written in these languages, and it is common for these systems to have limited resources. Embedded systems are also susceptible to errors, since they are often supposed to run for long times without crashing. Finding errors is therefore important, which makes analysing the software rewarding, but the low resource availability can make dynamic analysis tools challenging to use. We have evaluated different dynamic memory analysis tools by assessing their error finding capabilities, and their impact on the analysed programs' performance. We have carried out our work for Volvo Cars, and performed our evaluations on one of their embedded systems. As part of the evaluation, we also used the most promising tools to analyse some of Volvo's programs in a real setting. We managed to find two memory leaks in one of Volvo's programs by analysing them with the tools. The leaks were confirmed to be real by a developer, and Volvo could fix them. Based on our findings, we recommend a selection of tools that we consider useful for analysing software on embedded Linux. We conclude that using 2-3 less advanced tools in separate analysis runs works better than using a more advanced tool on Volvo's embedded system. This approach introduces less overhead while still being able to find as many errors as the more advanced tools.

Keywords: Memory Errors, Limited Resources, Embedded Systems, Automotive, Memory Analysis

Acknowledgements

We would like to thank our LTH supervisor Flavius Gruian for his guidance during the entire thesis process, especially for his detailed and constructive feedback on our writing. We would also like to thank Paul Asterland at Volvo Cars for guiding our work in a direction which made the thesis relevant for Volvo Cars and for making sure we felt welcome and had all the support and equipment we needed. Additionally, we want to thank all developers at Volvo who helped us with any technical questions we had, and with giving suggestions for the thesis: Johan Bohlin, Ola Lilja, Piotr Tomaszewski, and Mikael Persson. Lastly, we would like to thank everyone else who helped proof-reading our thesis; your feedback has greatly improved our writing.

Contents

Chapter 1
Introduction

"Since human beings themselves are not fully debugged yet, there will be bugs in your code no matter what you do."

— Chris Mason *[67]*

Embedded systems are present in many objects we use every day. For convenience and safety, it is important to minimise the risk of these systems malfunctioning. One way an embedded system can malfunction is if there is an error in the software running on it. Some of these errors can be found with analysis tools, but since embedded systems often have limited memory and processing power, not all tools are able to run on them. We have evaluated different tools, to see if they can be used to find memory errors in an embedded environment. We have carried out our work at Volvo Cars and performed our evaluations on an embedded Linux system used for telecommunications in future cars. The final product of our evaluations is a suggestion of tools which we consider useful for finding memory errors in embedded software. In this chapter, we define the problem we want to solve and why it is important, present our research questions, list the limitations and delimitations we worked with, and describe how we divided the work.

1.1 Problem Definition

Finding errors related to how memory is used in software is important, since memory errors can cause malfunctions and crashes. There are many tools that can be used for finding these errors: some of them only find a specific subset of errors, and some of them find many different kinds. A few of these tools, especially the ones that find many types of errors, introduce large overheads on the programs they analyse, both in memory consumption and in execution time. One such tool, which is arguably one of the more common memory analysis tools, is the Valgrind skin Memcheck. Since it is able to find multiple types of errors, Memcheck can be a useful asset during software development. However, the usefulness comes at a price; as described in Section 4.2.1, Memcheck imposes considerable overheads on the programs it analyses.

Finding memory errors in embedded software is especially important, since embedded systems are often intended to run for extended periods of time without crashing. Many of these systems have limited amounts of memory and are not as powerful as desktop computers, which can make demanding analysis tools, such as Memcheck, difficult to use on them. This is not a problem if the software can be compiled for a more powerful computer. However, sometimes the software is dependent on the embedded system. For example, it might need some specific hardware component to be present. Because of this, it would be desirable to find other tools that are not just effective at finding errors, but are also efficient in how much resource overhead they introduce on the programs they analyse.

For these reasons, we have evaluated different memory analysis tools with the intention of finding alternatives to tools like Memcheck. Our findings provide insight for which tools are suitable to use in an embedded system. We have also looked at different methods for finding memory errors, which are implemented by the tools we have evaluated. Based on our observations from the evaluations, we provide suggestions of methods that seem prudent for use in an environment with limited resources. This suggestion is of relevance to anyone interested in implementing their own memory analysis tool for an embedded system.

The work on our thesis has been carried out at Volvo Cars, and we did our evaluations on one of their embedded systems: the *Telematics and Connectivity Antenna Module* (TCAM). The TCAM has limited memory and processing power, and, for this reason, it is difficult to use Memcheck on it. Furthermore, some of the software running on the TCAM is dependent on hardware components and third-party deliverable binaries, and can therefore not easily be run on a more powerful computer.

Even though our thesis is helping Volvo Cars with their software development, we think it is also relevant in a wider context. For example, our results are relevant to other companies in similar situations who want to analyse hardware dependent software running on embedded Linux with resource constraints.

1.2 Research Questions

RQ 1. What are suitable tools and methods for finding memory errors in a system running embedded Linux?

(a) What is the overhead introduced by the tools, in memory and execution time, and does it affect their usability on the system?

(b) Which memory faults are relevant for the tools to detect and how do the evaluated tools perform in this regard?

(c) How useful is the tool-generated feedback for pinpointing a possible error?

RQ 2. Could there be advantages to using multiple tools in the development workflow?

1.3 Limitations

We have used the same embedded system, the TCAM, for all our work. It has a CPU with ARMv7 (AArch32, 32-bit) architecture, 256 MB RAM, and is running an embedded Linux operating system. The programming languages used by programs running on the TCAM are C and C++.

1.4 Delimitations

When we started working on our thesis, we had discussions with our supervisors and other stakeholders at Volvo about their expectations and what type of tools they were most interested in. In the beginning we intended to test both static and dynamic analysis tools. Accordingly, in the literature review we also read about static analysis tools and methods. However, to limit the scope of our thesis and manage the work according to our planned time frame, we decided to further narrow our evaluation of tools.

Together with Volvo Cars we decided to focus our work on evaluating tools that perform dynamic memory analysis, without limiting ourselves to any single type of memory error. Thus, we have not evaluated any static analysis tools. After considering the number of available tools, and the time frame of the project, we also decided to limit our evaluation to open source tools available on the internet.

The reason for excluding static analysis was partly because a thorough analysis of this category would not fit into the time frame of our project. However, our main reasons for not evaluating them was that we found scientific articles which had already compared different static tools [82][47][130][106], and Volvo already uses two different static tools in their workflow, whereas they do not have a good selection of dynamic tools.

1.5 Division of Work

We tried to divide the work evenly. We have mostly worked on the same things simultaneously, and discussed ideas with each other. During the testing phase, Emil focused on evaluating tools using the C test suite and benchmarks, and Fredrik focused on the C++ test suite and benchmarks. Towards the end of the project we focused on slightly separate aspects. Emil focused more on Volvo-specific testing and wrote most of the results section, while Fredrik wrote most of the background.

Chapter 2
Background

"Testing is one of the most profitable investments engineers can make to improve the reliability of their product."

— Alex Perry and Max Luebbe [51]

In this section, we give a background to areas related to our work. We begin by describing embedded systems and their features and constraints in Section 2.1. In Section 2.2, we discuss how the software for embedded systems sometimes depends on the hardware. We then discuss memory errors in Section 2.3. In Section 2.4, we describe how errors can be classified and the type of errors have focused on in this project. Then, in Section 2.5, we describe common ways to analyse programs for finding memory errors. We write about software in the automotive industry in Section 2.6 and how such software follows different standards in Section 2.7. In Section 2.8, we describe the specific embedded system we worked with at Volvo. Finally, in Section 2.9, we write about articles we have read that try to solve problems similar to ours.

2.1 Embedded systems

Computers are commonplace in today's society. Whether we are aware of it or not, we often interact with products controlled by computer systems on a daily basis. These systems are not what commonly comes to mind when we think of a computer; a device with a connected screen and other attachments such as a keyboard and a mouse. Instead, they are small machines with little memory and processing power, which control some functional part of a product [101, p. 1-3]. Often, the systems run programs which act upon either physical signals from the natural world, or electronic signals from other computer systems [107, p. 10]. These systems are commonly referred to as embedded systems or embedded computers. An

embedded system is in most cases not meant to be used as a general-purpose computer, but is instead manufactured with a specific use-case in mind [134]. Different factors can put constraints on its hardware, such as cost, size, or energy consumption [107, p. 13]. The systems can also come with specialised hardware components such as *Application-specific integrated circuits* (ASICs) [103, p. 10][49, p. 11]. ASICs are micro-chips customised to perform a specific task with logic implemented in hardware instead of in software. To keep costs and power consumption down, embedded computers are normally designed to be just as powerful and have just as much memory as is needed for their use-case [49, p. 11][107, p. 13]. Furthermore, many embedded systems are built to run for long periods of time without ever being turned off or rebooted, some even for several years. That means it is crucial the system has stable software to avoid failures [134].

Not all embedded systems use an operating system. Less powerful systems can be made with just enough processing power to run a single program. However, if the embedded system has to run multiple programs at the same time, an operating system is needed to schedule the executions and divide the resources [49, p. 173].

2.1.1 Embedded Software

Software meant to run on embedded systems can be referred to as embedded software [101, p. 2]. Embedded software needs to be developed to work with the limited resources available on the system. Since embedded systems are often constrained for both memory and computational resources, embedded software generally needs to be optimised to be efficient both memory and performance-wise [49, p. 11].

The C and C++ programming languages are often used in embedded programming [138, p. 30,52][49, p. 13-14]. The reason being that they give the programmer a lot of direct control of the hardware, similar to the capabilities of assembly languages, while also using a higher level syntax that makes them easier to read and use [49, p. 13]. C and C++ are similar languages. C++ is close to a superset of C, and it adds object-oriented constructs, such as classes, to the language, while still providing the closeness to the hardware [138, p. 179-178]. Some benefits of C/C++ are:

Memory Manipulation – Both languages provide direct access to memory addresses which enables manipulation of their content.

Deterministic Usage of Resources – It is possible to control exactly how long the program holds a resource. The deterministic nature of C and C++ makes time-predictability possible, which is crucial for some systems where the software needs to deliver a result within a set amount of time.

Performance and Code Size – Since C and C++ are compiled languages the machine code can be optimised by the compiler. This can both reduce the size of the binary and reduce the execution time [49, p. 254][88, p. 802-803]. Furthermore, since there is no *virtual machine* layer needed to execute the code, there can be no bugs which can manifest itself in that layer.

Portability – The code is written using high-level language syntax and constructs, which are translated by a compiler into machine code for a specific architecture. This means that much of the code is portable and often not written for a specific CPU [49, p. 14].

The mentioned strengths of C and C++ have made them some of the most commonly used programming languages for embedded systems. However, the ability to control exactly how long a program should hold on to memory comes with responsibility for the developers. Incorrect memory management can lead to errors.

Using C and C++ results in a trade-off where the program is often fast and small, but it comes with the possibility of memory-related bugs caused by mistakes by the programmer, such as memory leaks and memory faults. It is important to avoid memory faults in programs, as they might cause an unpredictable crash. Memory leaks can also cause crashes if the leaks cause the system to run out of memory. We discuss memory leaks and memory faults further in Section 2.3.

Since embedded systems are part of larger devices, a crash can have severe consequences. For example, if the computer in charge of cruise control in a car stops working, the car might act in an unexpected manner, which in the worst case scenario can lead to a severe accident [131, p. 3]. Another example where embedded systems play a critical role can be seen in the medical industry. For example, embedded systems can be used as part of an infusion pump to monitor the vitals of a patient and react to the readings [116].

2.2 Hardware Dependent Software

Software developed for embedded systems can, to a large extent, be dependent on the hardware of the system. Such software is referred to as *hardware-dependent software*, and it is defined as "*software in an embedded system that closely interacts with the underlying hardware platform*" [75, p. 71]. Such software either directly interacts with the hardware, or is significantly affected by the hardware. The automotive industry is specifically mentioned as using complex systems, with communication between different parts. This leads to high coupling of software and hardware, which makes testing and simulation difficult [75, p. 263][125, p. 335]. Because hardware dependent software needs access to hardware functionality, developers often need direct access to the hardware during development and testing [73]. This means that the software is often developed for a specific hardware platform [104]. A common example of such software is I/O communication software [75, p. 70]. Generally speaking, the functionality of hardware dependent software often fits within some of the following categories: "*communication between different tasks, external communication with other processing subsystems, hardware management and control*" [75, p. 70]. Since embedded applications are often developed for a specific system, it can be difficult to test them on another system [97].

The proposed method for improving the development process of hardware-dependent software is often to construct virtual prototypes of the hardware components. The software can then interface with these prototypes instead of the actual components. However, the creation of them needs to be done in close collaboration between the hardware and software teams and should be initiated early in the design process [69, p. 40]. Furthermore, the creation of such prototypes often incur a substantial investment and commitment from the organisation [69, p. 40][100]. Because of this, aspects of the virtual prototypes, such as their accuracy, development cost, debug insight, and turnaround time need to be evaluated [46, p. 13-14][100].

Because of the coupling of hardware and software, specifically in the automotive industry, initiatives such as AUTOSAR are under development [53]. AUTOSAR aims to provide

a standard for the virtualisation of the hardware layer during development. Such a standard would provide an environment for earlier system evaluation and verification of embedded systems. The AUTOSAR standard aims to provide a middle layer which allows the implementation of the software without the hardware [75, p. 290]. The hardware dependent parts can then be developed using the simulated layer instead of the actual hardware, which helps decouple the software from the hardware [53].

However, there is plenty of difficulty in creating a specialised virtualisation platform standard for embedded development. This is because there is no such thing as a single type of embedded system. Each system is unique in the sense that it is created for a specific need, and the development of its intended functionality needs to conform to the hardware constraints present in that specific system [104].

Moreover, even if virtualisation of the hardware exists, testing and analysis at higher integration levels is also needed during the development process [120]. Dynamic analysis on the actual system can help developers detect certain problems which might not be seen in static analysis or analysis on virtual prototypes. Examples of such problems are interface problems and resource consumption problems, both of which are results of the integration of the software with the hardware. Using systematic testing and analysis procedures throughout the entire development process can therefore help achieve sufficient software quality [104].

2.3 Memory Leaks and Memory Faults

In this section, we describe what memory leaks and memory faults are. The specific memory errors investigated in our thesis are listed in Section 2.4.1. Since C and C++ allow direct memory manipulation, many different memory errors can occur if the developer is careless [118, p. 357]. Some examples are:

- *Accessing dynamically allocated memory after it has been freed*
- *Accessing memory addresses out of the bounds of allocated memory*
- *Freeing memory more than once*
- *Not freeing memory that is not needed anymore*

Finding memory faults in software is important given the prevalence and severity of memory related bugs. For example, 70% of the serious security bugs in "*The Chromium Project*" are caused by memory problems related to erroneous usage of C/C++ pointers [3]. The same number is mentioned by *Mozilla* and is part of the reason why they are moving to rewrite parts of their code base in *Rust* [93]. The same numbers are reportedly also shared by *Microsoft*, which claim that 70% of all software vulnerabilities in their products are due to memory safety issues [63].

Finding memory errors is not always a simple task. Even if the errors are noticed it can be difficult to track their visible symptoms back to the source [88, p. 734]. The reason for these difficulties are that memory errors do not always manifest themselves at their origin. Instead, they can cause the program to behave unexpectedly or crash at a later point in the execution.

2.3.1 Memory Leaks

Memory leaks can be said to occur when a program uses more memory than it needs to execute correctly [118, p. 409][105].

In [55], leaks are divided in two broad categories: "*Lost objects*" and "*Useless objects*". In addition to these two categories, one can argue that memory which is not allocated by the program but is still kept around by the allocator is also a sort of leak. A memory allocator requests memory from the operating system, which is then distributed via memory allocation functions such as *malloc* and *new*. Memory which the allocator is holding on to, that is not allocated, can be considered the same as unused memory but from an operating system point of view since it is not currently used and could be returned to the system. We divide memory leaks into three categories, inspired by a delineation made by a software engineer at Oracle [76]:

Unreachable memory – Memory which has been allocated but not deallocated and is not pointed to by any pointer accessible to the program. This memory is completely dead since the operating system thinks it is used by the program and the program does not have any way to access it anymore. This is what is described as *Lost objects* in [55].

Unused memory – Memory that is allocated by the program, but will not be used any more during the program's lifespan. Unused memory can be considered a memory leak since it can be returned to the operating system without any impact on the program currently in possession of it. It can be caused, for example, by data accumulating in buffers or lists without being cleaned out. This is what is described as *Useless objects* in [55].

Free but unused or unusable memory – This increase in memory usage can arise due to the memory allocator used by the program. If memory has been correctly freed by the programmer, but the allocator does not give back the memory to the operating system, the program is still in possession of the memory [50][84, p. 47]. Depending on how allocations are done in a program it could sometimes be better to release this memory to the system instead of holding on to it. It can also arise due to heap fragmentation, which leaves spaces of empty memory that are too small to be used for a new allocation [45, p.16].

Memory Leak Detection and Prevention

As previously mentioned, memory leaks can be divided into three categories. Different methods can be employed for finding leaks in the different categories:

Unreachable memory – Unreachable memory can be detected by keeping track of allocated memory and checking whether all of it is pointed to by the program. All memory that is not pointed to can be considered unreachable. It can also be detected by printing all unfreed memory at program exit. Many tools that perform static or dynamic memory analysis can find these types of memory leaks.

Unused memory – This kind of leak can be hard to find, unlike obvious logical leaks which can be found through simple analysis. Tools that are categorised as "*heap profilers*" typically have to be used [42, p. 165]. These tools keep track of the memory consumption of a program while it is running. Many of them report how much memory a program used at exit as well as the peak memory usage, and they also provide a graph of memory usage over time [62]. The graph can prove a useful asset in finding memory which is allocated but not used since it shows if the memory consumption keeps increasing the longer the program runs.

Free but unused or unusable memory – To solve this kind of problem the allocator from the standard library could possibly be tweaked by setting different parameters, which would result in better management of the memory. Other alternatives are implementing a custom memory allocator and using an alternative open source implementation. Different memory allocators can affect a program's memory usage in different ways [129].

The memory leak found by most tools that can detect leaks is *unreachable memory*. *Unused memory* can also be found by manually analysing the output of some tools that profile the heap. We have not evaluated any method or tool which finds *free but unused or unusable memory*. That is because it is not really a leak that can be found in the same sense as the other two. It occurs either due to fragmentation or is intentionally created by the memory allocator when it reserves memory from the system for use in future allocations.

2.3.2 Memory Faults

We use the term *memory faults* throughout the thesis. By memory faults, we mean memory errors which are caused by the program doing something it is not allowed to do, related to memory. Examples include writing to deallocated memory, deallocating the same memory twice, and writing and reading out of the bounds of arrays that exist either on the stack or allocated on the heap.

2.4 Common Weakness Enumeration

The Common Weakness Enumeration, or *CWE*, is a category system for different software defects and flaws [4]. We have used the CWE to classify the errors we have tested for. The category system is constructed so that codes can relate to each other by belonging to different types/classes, and different types can belong to different levels in a chain of relationships. The types and levels of the classification system help to show how different kinds of errors are related to each other. The relation between codes is described as a parent-child relationship, where a parent code can be the general description of the error and different child codes can correspond to more concrete cases. There can be several levels in the hierarchy.

CWE has a list of the top 25 most dangerous software errors. They are quite general, but their children are more concrete. In Table 2.1, we show the error codes of our test cases and how they relate to the codes in the top-list. We found 6 categories which were related to memory problems in the CWE top 25 list.

Rank in toplist	CWE ID	Name	Tested by Child With Code
1	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	CWE-121, CWE-122, CWE-124, CWE-126, CWE-127
5	CWE-125	Out-of-bounds Read	CWE-126, CWE-127
7	CWE-416	Use After Free	CWE-416
12	CWE-787	Out-of-bounds Write	CWE-121, CWE-122, CWE-124, CWE-129
14	CWE-476	NULL Pointer Dereference	CWE-476
21	CWE-772	Missing Release of Resource after Effective Lifetime	CWE-401

Table 2.1: Our test cases relation to the CWE top 25 list

2.4.1 List of Errors and Their CWE Classification

Here we present the list of errors we arrived at, which we used to test our tools. In Section 3.3.1, we describe how we constructed the list. The different errors are presented next to their CWE code. Some errors have the same CWE code but are still different since CWE did not always specify if the error was in the stack or the heap. We wanted to test whether the error occurred in the stack or the heap separately to see whether the tools could find it in both.

CWE ID	Case Name
CWE-121	Stack-based Buffer Overflow
CWE-122	Heap-based Buffer Overflow
CWE-124	Stack-based Buffer Underflow
CWE-124	Heap-based Buffer Underflow
CWE-126	Stack-based Buffer Overread
CWE-126	Heap-based Buffer Overread
CWE-127	Stack-based Buffer Underread
CWE-127	Heap-based Buffer Underread
CWE-400	Stack Overflow (stack exhaustion)
CWE-401	Memory leak
CWE-415	Double free
CWE-416	Use after free
CWE-457	Use of Uninitialised Variable
CWE-468	Incorrect Pointer Scaling
CWE-476	NULL Pointer Dereference
CWE-562	Use after return
CWE-588	Attempt to Access Child of a Non-structure Pointer
CWE-590	Free of Memory not on the Heap
CWE-665	Improper Initialisation
CWE-762	Mismatched Memory Management Routines

Table 2.2: List of all our error cases and their corresponding CWE code.

CWE-121: Stack-based Buffer Overflow – A buffer allocated on the stack is written to, beyond its allocated address space. Writing to memory outside of the boundary of the buffer is an error which could cause overwrites of memory belonging to some other data structure. It could also cause other kinds of memory errors, such as return addresses being overwritten.

CWE-122: Heap-based Buffer Overflow – A buffer allocated on the heap (dynamic memory) is written to, beyond its allocated address space. This overwrites either unallocated memory or another memory area which is allocated.

CWE-124: Stack-based Buffer Underflow – A buffer allocated on the stack is written to, before its allocated address space. This can corrupt the data stored at that memory location.

CWE-124: Heap-based Buffer Underflow – A value is written to an address which is outside of the lower bound of an allocation on the heap. This can corrupt the data stored at that memory location.

CWE-126: Stack-based Buffer Overread – A buffer allocated on the stack is read above its allocated address space. A read does not corrupt any data, but depending on how the read data is used it might cause errors in the program.

CWE-126: Heap-based Buffer Overread – A buffer allocated on the heap is read above its allocated address space. A read does not corrupt any data, but depending on how the read data is used it might cause errors in the program.

CWE-127: Stack-based Buffer Underread – A buffer allocated on the stack is read before its allocated address space. A read does not corrupt any data, but depending on how the read data is used it might cause errors in the program.

CWE-127: Heap-based Buffer Underread – A buffer allocated on the heap is read below its allocated address space. A read does not corrupt any data, but depending on how the read data is used it might cause errors in the program.

CWE-400: Stack Overflow (stack exhaustion) – C and C++ programs have specific segments in memory reserved for the stack. The stack memory consists of stack frames which are created during function calls. The stack frames contain return addresses, local variables, and parameters used in the different calls. A recursive function call, or trying to allocate a very big buffer on the stack, can cause the stack to run out of memory and "overflow" into other memory segments. This error caused an immediate crash of the program when run on both our host and target system (x86 Linux and ARM Linux).

CWE-401: Memory leak – A memory leak is memory claimed by a program, but not used. This can either be caused by the program losing its reference to the memory, or by the program not "freeing" the memory when it will not use it any more. Memory leaks can be especially dangerous when references to allocated memory are constantly lost, for example in a loop, as this will drain the available memory on the system.

CWE-415: Double free – A double free is a memory fault caused by two calls to the *free* function. In C, the first call will free the memory, but the second call will result in undefined behaviour. This error caused an immediate crash of our test program when run on both our host and target system (x86 Linux and ARM Linux).

CWE-416: Use after free – A pointer to heap memory is used after the memory has been deallocated.

CWE-457: Use of Uninitialised Variable – A memory address which has been allocated, but not had a value placed in the space, is used as if it had value. That means whatever value happens to reside currently at that memory location is used.

CWE-468: Incorrect Pointer Scaling – C and C++ permits using some arithmetic operations on pointers. This could lead to errors if the resulting address refers to memory outside the allocated space. For example, it can lead to buffer overflows/underflows or overreads/underreads.

CWE-476: NULL Pointer Dereference – A pointer which holds null is dereferenced. This error caused an immediate crash of our test program when run on both our host and target system (x86 Linux and ARM Linux).

CWE-562: Use after return – A pointer to memory which was allocated on the stack is returned from a function and used elsewhere. After the function returns, the stack frame is popped from the stack and memory within the stack frame could contain anything.

CWE-588: Attempt to Access Child of a Non-structure Pointer – Error which occurs if casting a non struct pointer to a struct pointer and then trying to access a field inside the struct from the pointer.

CWE-590: Free of Memory not on the Heap – Free is called with an argument which is not a pointer to memory allocated on the heap. This error caused an immediate crash of our test program when run on both our host and target system (x86 Linux and ARM Linux).

CWE-665: Improper Initialisation – Parent category of *CWE-457: Use of Uninitialised Variable*, but refers to more cases than a variable. For example, it can refer to reading data from an uninitialised buffer.

CWE-762: Mismatched Memory Management Routines – This error can happen if a bad combination of memory allocation and deallocation routines is called. For example, if something is allocated with the C function *malloc* and then released by the C++ operator *delete*.

2.5 Memory Analysis

Memory analysis can be used to find memory leaks and other memory errors. There are two commonly used ways for analysing a program's memory:

Static – Static memory analysis is performed on the source code to determine whether there are any memory errors in it. A great benefit of static analysis is that since it analyses the source code it does not need to work on the architecture for which the code is written. Since static analysis is platform-independent, it is easy to use for analysing embedded software.

Dynamic – Dynamic memory analysis is performed on a program at run-time and reports memory errors that occur during program execution. This type of memory analysis can require a lot of system resources, especially memory, which can be problematic for embedded systems as their resource availability generally is limited. Moreover, tools that perform dynamic analysis must support the same architecture as the programs they analyse.

We have focused on dynamic analysis during the project, but we have also read about static analysis during our literature review. Both static and dynamic memory analysis can be useful for identifying errors in the code, and using them in combination can provide an advantage [104].

For an embedded system, dynamic analysis can present some challenges. As mentioned in Section 2.2, software in embedded systems can depend on the hardware of the system. The software can then not be run on another computer without extra virtualisation steps, such as simulation or emulation of hardware. Furthermore, the limited resources available on embedded systems can make it challenging to use dynamic analysis tools on them. These challenges need to be tackled when performing dynamic memory analysis on an embedded system.

2.5.1 Static Analysis

Static analysis is performed on the source code of a program and reasons about different run-time properties of the program without executing the code [78]. Since the code does not need to be executed, static analysis can be performed on an architecture for which the code is not written.

There are different types of static analysis tools which implement different methods and have different capabilities. In the literature review, we read two papers about static analysis [87][78]. They describe different techniques and methods used in the implementation of static analysis tools with the following terms:

Data Flow Analysis – A control flow graph representation of the program is made where the nodes represent different program states. The flow to the next node depends on the input to the current node and what happens at the current state, such as expressions being evaluated, variables being changed, or functions being called. Each node can reason about what values are possible for different variables and expressions [87]. Data flow analysis is further described by two subcategories:

Path-sensitive analysis – Where only valid program paths are taken, which is determined by the value of variables and expressions [78].

Context-sensitive analysis – Takes into account the whole context of the program state, such as global variables and where a function was called [78].

Symbolic Execution – Instead of giving direct values as input to the program and functions, a symbol representing a range of values is used. All input and output values are represented as functions of the input symbol value range. The program can then take many different paths depending on the symbol values. Output values are represented by the range of different kind of values they can take [87].

Abstract Interpretation – Used by some types of tools to prove the absence of errors. An abstract mathematical model of the program domain and semantic operations is created [87]. The absence of errors can be proved for the model and thus the program. The problem is that a detailed model takes a long time to analyse a program, but a less detailed model may reject valid programs because of missing details about the domain [87].

The different techniques can be used to implement different types of analysis tools. In [87], the authors divide static tools into the following categories depending on their capabilities:

Static checking – These tools can find common coding errors and simple bugs. They mainly check for patterns in the code and use simple data flow analysis with limited precision of program state [87][78]. Static checking is usually very lightweight and fast, but makes many assumptions which can result in a lot of false positives [81][78].

Error detection/bug finding – These tools can find more categories of bugs and report them. They use more advanced data flow and symbolic logic evaluation techniques [87][78]. Instead of simply analysing code, a tool of this type sometimes also runs simulations of function and program execution. It can then find more complex errors resulting from different pointer operations [61].

Verification – Verification tools can guarantee the absence of errors by proofs, using an *"abstract Interpretation"* technique. They often require restrictions of the language, such as disallowing dynamic memory [65]. These tools provide no false negatives, i.e. they find all possible errors which could be present during run-time. For example, if the tool can find "division by zero"-problems and allows the expression a/b, that is proof b can never be 0 [91]. However, these tools can still provide false positives and usually require a lot of time to run [65]. These types of tools are called "*sound tools*" in some papers while other tools are referred to as either being "*unsound*" or "*syntax checkers*" [91].

Type inference – Type inference tools specify which different properties hold at different parts of the program. They are often implemented using some sort of constraint-based solution [87], which analyses the program by traversing the execution path and solving for constraints at different parts [99].

The architecture-independence of static analysis tools can be an advantage compared to dynamic analysis tools, especially when analysing programs that have to be run on a system with limited resources. However, there are some drawbacks. Static analysis tools can have

trouble finding subtle problems in the code and they may report a large number of false errors. False errors can be divided into two groups: *false negatives* and *false positives* [140][111][135].

False negatives – Are errors in the code which the tool does not detect [94].

False positives – Are reported as errors but are in fact not real errors. These are often caused by assumptions made by the tool [98]. Assumptions are made because some values are determined at run-time and the tool then over-approximates the program behaviour [102]. The approximation is done because following all paths in the program could lead to a path explosion with too many paths to evaluate [56, p. 2].

Both false negatives and false positives can arise in cases when program behaviour can not be decided by static analysis [122]. The reason can be that values in the program will become available after execution has started or that they are computed during execution [124]. Another reason can be that some part of the code depends on behaviour which is undefined according to the C/C++ standards [77]. Furthermore, many tools have trouble evaluating complex aspects of C/C++, such as multi-level pointers and multi-level structures [74][136].

Moreover, source code will be turned into machine code which runs on a system. The machine code is created by a compiler which could possibly introduce new errors [66].

As discussed in this section, static tools are not completely reliable, either because of limitations in their capabilities or because of assumptions made during analysis. Therefore, static tools can be complemented with dynamic tools to find more bugs during development [44].

2.5.2 Dynamic Analysis

Dynamic analysis is a general term which means that analysis is performed on a program while it is running. Since dynamic analysis can only capture what happens when the program is run, errors can only be found in code that is executed. Because of this, it is important to make the program follow several different execution paths. This can be achieved by, for example, testing the program's functionality while it is being analysed.

There are different methods which can be used to perform dynamic analysis. They are implemented by various dynamic analysis tools. Some tools might implement several methods, and others might only use one. There are different advantages and drawbacks to each method, and the methods that are used by a tool affect what kinds of errors it can find and how much overhead it imposes.

One of the simpler methods for checking for memory leaks during run-time is to wrap or overload the allocation and deallocation functions and keep a log of all calls to said functions. Then, at program exit, any allocations which have not been freed can be reported as leaks. This approach finds both unreachable and unused memory, but can be considered to have a large drawback. In many cases it is not better, or necessary, to free allocated memory before program exit, and every such allocation that is not freed will be reported as a leak by the aforementioned approach. To implement the method, the allocation and deallocation functions can either be wrapped using macros which redefine them at compile-time, or they can be wrapped by functions that are in a dynamic library which is ahead of the standard library in the linking path. This method is used by tools mentioned in older articles [119][79][80]. Since it is a simple way to implement leak checking, many of the open-source tools use it as well.

There are a number of more advanced methods which can be employed to analyse the memory of a program. For example, *red-zones* can be added before and after each heap allocation [90]. If a program tries to read or write to these red-zones, it is considered a memory access violation. To detect whether a program writes to deallocated memory, that memory can be kept for a while after it has been deallocated and treated as a red-zone.

Some dynamic analysis tools also report any memory leaks that are present at a certain point in program execution. One way to achieve this is presented in [121]. The method described in the article keeps track of which memory blocks are allocated, it follows pointers recursively from the stack and data segments, and then it marks memory blocks that are pointed to by the program. After all pointers have been followed, memory blocks that are not marked can be considered leaks as they are unreachable.

This way of keeping track of pointers is *pointer based*, which means that extra information is stored for each pointer. The information can be stored in, for example, a header, and it can be used to keep track of whether the object pointed to is marked. There is another approach, the *object based* one, which means information about allocated memory is tracked in a separate data structure, such as *shadow memory* [112].

Shadow memory is a software construct that keeps track of memory which is used by a program [137]. The size of the shadow memory can vary depending on how much information is to be stored in it. In [114], Nethercote et al. describe how shadow memory is implemented in Memcheck.

Since dynamic analysis is performed at run-time, dynamic analysis tools must be able to run on the same system as the program being analysed. By analysing a program, the tool contributes to an increase in memory usage and a slowdown in execution time of the program. This makes dynamic analysis tools intrusive, since they will take up system resources such as RAM and CPU time, which might affect the execution of the analysed program.

Because it analyses a program that is running, a dynamic analysis tool can only find errors in code that is executed, and errors in corner cases might slip by unnoticed. An advantage with dynamic analysis is that it can be done in the environment where the analysed program is intended to be used, which means it can be tested and evaluated to see if it behaves correctly in a scenario similar to the finished product.

Some dynamic memory analysis tools focus on the heap usage of the program, instead of, or in addition to, reporting errors. These tools are called heap "profilers" and, as discussed in Section 2.3.1, they can be a used to find memory which is allocated but not used.

2.6 The Automotive Industry and Software

Modern cars are becoming increasingly more digital. Estimates state that around 90% of the innovations in the automotive industry are driven by electronics and software [85]. A newly produced car has numerous electronic control units (ECUs). For example, high-end cars can have over 130 different ECUs [132, p. 11]. Volvo's cars contain more than 100 different ECUs. These units provide functions like cruise control, smartphone-like infotainment systems, communication with the cloud, and much more. As described in [101, p. 2], each ECU could be called an embedded system, which is a broad term that can be applied to many different computer systems. Embedded systems are often built into other products [89, p. 30].

As mechanical control units are being replaced with electronic control units, complex embedded systems control more and more functionality in modern cars. This mix of mechanical parts with electronics is often referred to as *mechatronics*. The term *mechatronic part* is used to refer to the merger of a mechanical part with an electronic part which work together to perform a task. This is not a new concept, but a mechatronic part has been designed as such, instead of designing the two parts separately and then merging them [52]. In a mechatronic part which acts on physical input signals, the control system is usually an embedded system of some sort.

As mechatronic and electronic parts make up more and more of newly built cars, there is also an increased demand on verification and testing of these components [83][57]. A new car is one of the most technologically complex systems used in modern society. According to a 2018 report by McKinsey, a modern car in 2010 had roughly 10 million lines of code (*MLOC*), which by 2016 had grown by a factor of 15 to roughly 150 *MLOC* [60]. This can be compared to the estimated lines of code of software in other products. For example, estimates put the code of a Boeing 787 aeroplane at 6.5 *MLOC*, the whole Windows Vista operating system at 50 *MLOC*, and the source-code underlying the Facebook social media platform at roughly 62 *MLOC* [71].

The ECUs used in the industry are often limited in hardware resources such as CPU power and memory size. The limited hardware capabilities are a result of the balance between cost and system resources, which is needed to keep expenses down. For example, if the cost of an ECU increases by 1 U.S. dollar and 500 thousand cars are sold per year, times an average of 6 years of production costs, the result would be a cost increase of 3 million USD [125, p. 337]. The cost of electronics is becoming a larger part of the total cost of a car; Deloitte estimates that in 2030, automotive electronics will make up 45% of the total cost [70]. This fact creates a large incentive to keep the cost of every electronic unit down, which in turn results in more narrow system constraints. Another aspect of cost-saving is to try and find bugs and errors as early in development as possible. A study found that software-related recalls increased from 5% in 2011 to 15% in 2015 [126]. As the amount of software in cars continues to increase, so will probably the number of recalls, especially if proper processes are not put in place to find more bugs earlier during development. In the same study, the authors claim that early detection of software defects can lead to large savings for the whole product development process for the automakers. They write the following:

> *"Increasing the faults detection rate within the development phases will reduce problem fixing effort and the test effort. More precisely, detection of 10% more defects in software design or coding phases can lead to a potential saving of 3% of the total product development cost. The error correction cost can even increase up to 90 times in the post-production phase compared to the concept phase. Price of recalls comprises beside fault fixing costs, also legal costs and image costs."* [126]

As the complexity of car software continues to increase, security concerns and testing needs be at the centre of the software development process to avoid bugs and security flaws in the product. The push towards high safety standards and reliability comes both from the industry, and the public, which see the need for more reliable software as cars take over more and more of the driving responsibility [86]. As noted in [117], Volvo Cars stand before many challenges to change the way they develop software for an increasingly connected car.

As we mentioned in Section 2.2, developing software for ECUs in the automotive industry often leads to a close interleaving between hardware and software. That means that the

developed software often needs hardware parts to be present to run with full capabilities and functionality. That is why testing in a more complete system environment together with the hardware is a desirable step during development. This way of development follows the "*V-model*" [123] which can be visualised as the v-shape in Figure 2.1 and suggests doing testing in multiple stages [131, p. 212-218].

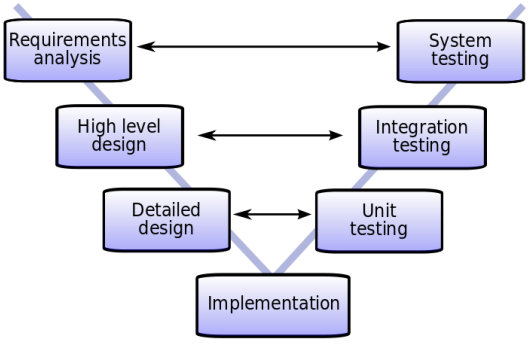


Figure 2.1: V-model for software development [59] [1]

2.7 Automotive Standards and Guidelines

As the complexity of cars continues to rise, more safety standards have been developed to ensure the continued safety of the cars produced by the industry. There are numerous different standards which apply to the different parts of a car. This section describes why there can be memory leaks in some software parts of a car and not other parts. It also highlights that the automotive industry is aware that memory errors can occur when C/C++ is used and have multiple initiatives to combat them.

Requirements and guidelines regarding software components in a modern car are dictated by ISO 26262, which is a multi-part functional safety standard intended for software development in the automotive industry. Some parts of ISO 26262 dictate different regulations and rules which software components should follow [95].

Programs running on various ECUs follow different levels of safety requirements depending on the safety classification level. The ISO 26262 standard measures different safety levels by a system called *Automotive Safety Integrity Level* (ASIL). There are 4 different levels of ASIL: A, the lowest, to D, the highest. The safety classification level of an ECU is determined by how safety-critical the ECU is in the overall vehicle. ISO 26262 recommends following different guidelines of development, depending on the safety level classification of a component. The guidelines prescribe various restrictions on how certain programming language constructs can be used and how there needs to be redundancy in case of failure [68]. For example, since memory leaks are caused by improper handling of dynamic memory they can not be present on more safety-critical ECUs. Dynamic memory is disallowed in ECUs with a higher safety level, but not in the system we have worked with in this project. One of the steps which the ISO standard recommends for the development of embedded software in cars is to use *MISRA-C* [95].

[1] GNU Free Documentation License Version 1.2

MISRA-C are software development guidelines for C, which have been created by the *Motor Industry Software Reliability Association* (MISRA) [108]. The goal of the guidelines is to improve the security, safety, portability, and reliability of the code operating in embedded systems within cars [131, p. 138]. The MISRA-C guidelines restrict which parts of C can be used. For example, MISRA-C completely disallows the usage of dynamically allocated memory. This removes the risk of memory leaks in the software. However, other memory errors, such as buffer overflows, can still occur. MISRA has also created guidelines which apply to C++, with the same restrictions on dynamically allocated memory [48].

Other guidelines also exist, such as *AUTOSAR* C++ guidelines which are aimed towards C++ and designed for the automotive industry [48]. The AUTOSAR guidelines are mostly built on the MISRA C++ guidelines, however, they allow the usage of dynamic memory while still being safety oriented. It was recently announced that MISRA and AUTOSAR will collaborate in creating a new set of guidelines for C++ that can be used in future automotive software [109].

Following the different coding guidelines will, in some cases, eliminate certain types of memory faults and lead to safer code. However, unless the guidelines completely disallow the usage of some memory operations, there is still a risk that leaks and errors will be present in the code. MISRA-C is the only coding standard which is referenced in the ISO 26262, but it does not mandate that it is followed. It is therefore up to the automotive manufacturer to put suitable processes in place, which make sure the software is reliable. The ISO 26262 standard recognises that testing and verification of software needs to be done at multiple steps during development to minimise the risk of faulty software. As a step in the software development process, ISO 26262 explicitly recommends the usage of static tools, and it mentions that other types of tools could be helpful as well [95]. Dynamic analysis tools could, therefore, fit into the development process recommended by the standard. They can help with identifying potential memory errors on systems that are allowed to use all memory operations. This is why the evaluation of such dynamic tools, as done in our thesis, is relevant to the automotive industry.

2.8 Volvo and the *TCAM*

Some of the more powerful ECU's in Volvo's cars are using embedded Linux, and run programs that are developed in-house by Volvo. It is on one of these units, the TCAM, which we have carried out our thesis work.

Volvo has taken over large parts of the development of the TCAM software from a third party supplier, referred to as Company 1. This transition occurred during product development, and the companies are working together to finish the product. Company 1 supplies the hardware and is responsible for the drivers and the software stack that is closest to the hardware. The TCAM is not an over-the-counter device. It has many specialised hardware components which enable its connectivity and telematics functionalities. As the connectivity and antenna module of the car, the TCAM enables communication with the outside world. It supplies a radio antenna, a WiFi chip, an LTE modem, a Bluetooth chip, two 3G/LTE sim card modules, a GNSS module, an *e-Call* antenna (to comply with the European initiative to bring assistance in case of a collision), and a chip and antenna for remote key-less entry

system. Furthermore, the TCAM has connections to enable communication over CAN bus and ethernet bus to other ECU's within the car [131, p. 138].

There is an ARM processor in the TCAM, and it has 256 MB of RAM. In addition to the ARM processor, the TCAM contains a microprocessor called the vehicle-CPU (VuC), which is responsible for reading and reacting to sensor signals related to vehicle start and stop, temperature management (what to turn off when running too hot), and power management (such as what to turn on and off depending on state of the car). The VuC also mediates signals from the BLE chip and sends them to the main CPU. Some of the software running on hardware units in the TCAM, which are all part of the connected system which make up the whole TCAM unit, is only supplied as binaries to Volvo. For example, the firmware for the VuC and the modem.

The software developed by Volvo is running on the TCAM's main ARM processor, and it interacts with many of the hardware components. Parts of the communication between the main CPU and other hardware components in the TCAM, such as the VuC and modem, could to some extent be mocked or stubbed. However, the mocks and stubs would require developer time and resources and they would not fully capture the communication between software and hardware. Furthermore, much of the software developed by Volvo will directly use hardware such as the BLE chip, the VuC, the LTE modem, etc. To faithfully test these components on another computer would require the presence of these hardware units, which are currently only available on the TCAM. The TCAM unit is not classified to the highest ASIL levels, which means it is allowed to run software which uses dynamically allocated memory, and therefore is vulnerable to memory leaks. Some of the software on the TCAM executes only when the car is turned on, but some software is running even when the car is turned off. This is to make sure communication with the car can still be done through different services and applications. Because of this, some software will run for very long periods of time, which means it needs to be stable and robust. Memory errors can make the software unpredictable and maybe even cause a crash. Therefore, it is important to find and eliminate them.

2.9 Related Work

In [110], the challenges of finding memory errors in embedded systems are described. The authors specifically mention that memory errors can cause embedded devices to fail silently, which means the program enters a faulty state but does not crash. This can be dangerous for the functionality of the embedded system, but can be hard to detect immediately. They suggest different approaches for testing embedded software, such as static analysis and dynamic analysis. Furthermore, they describe how emulation can make it easier to test different parts of the system. The problems they lift highlight the same concerns Volvo are facing in their embedded development. Their paper highlights why dynamic analysis of embedded systems is sometimes warranted but problematic due to overhead. It connects with our work in the sense that we evaluate the overhead of different dynamic analysis tools on an embedded system. Our results show which tools might be less problematic to use.

Different suggestions have been made to address the problems of finding memory errors in C/C++ code. Some papers propose to implement a garbage collector solution in the languages [54]. Another suggested solution is to to introduce tracing instructions in the source

code [64][136]. However, most papers describe tools which can be classified as either static or dynamic. These tools can often be used to find memory leaks and sometimes memory access errors as well.

Yu et al. describe how both static and dynamic techniques can be used to find memory leaks in a program [141]. They write that while static tools do not require executing the code, there are types of errors which many of them cannot find. Furthermore, while dynamic tools might catch errors static tools miss, they will only find faults in paths taken during run-time. Their contribution is a dynamic tool which instruments programs before they execute and updates the analyser with information during run-time. The tool is used for finding memory leaks, but it also helps the programmer with fixing them, by keeping track of where the memory was last used and suggesting where a deallocation might be suitable.

The authors argue that the step of suggesting a fix is often omitted in current tools. Furthermore, their tool uses a method called "dynamic symbolic execution" which executes the program with different input values in order to make it take different paths during execution. This is done to alleviate the single-path issue of dynamic analysis. They compare their tool with different static tools as well as dynamic tools. An interesting find in the paper is that their results show that sometimes static analysis tools do not find the same errors as dynamic tools, and vice versa, which showcases that there is an actual benefit to using both approaches when analysing a program. Their paper shows that both static and dynamic analysis have their pros and cons, and, right now, using a combination of them might find the most errors. This is relevant to our work since it shows that dynamic analysis can be a complement to static analysis, rather than an alternative approach. It is therefore valuable to do dynamic analysis too, meaning our work to determine which dynamic tools can be used for embedded systems is relevant.

Zaslavskiy et al. describe that there are some situations when it is warranted to analyse software on embedded systems with dynamic analysis tools [142]. In best-case scenarios, it would be possible to re-compile the code for another system and run the dynamic tool there, which would make resource constraints less of a concern. However, there are situations where re-compiling and re-linking the program is not possible, such as when some libraries are not available for the platform or when software has dependencies on hardware in the embedded system. In these situations, it is suitable to use lightweight dynamic methods. The article describes that Valgrind is not suitable because of its rather large overhead. Furthermore, it is preferable that the dynamic tools do not modify the source code or the machine code of the software. A reason for this concern is that modifications of the code might change how the program executes, for example by changing timings. The article describes how they modify shared library functions to wrap calls to memory management functions and log information in those calls. They do so by hijacking the dynamic linker and dynamic loader to make calls to library functions go to their own implementations instead. This approach does not require recompiling or re-linking the program. They show that their method introduces little overhead in memory and CPU utilisation. Memory overhead is measured in static and dynamic cost where static cost is hard disk space usage and dynamic cost is RAM usage during run-time. The described problem area in the article is similar to the one in our thesis. However, the authors have not published their tool, nor do they supply many implementation details. This article highlights that dynamic testing cannot always be done on another system, and sometimes dynamic analysis has to be done on the embedded system itself. This

work relates to our research, given the common overlap in evaluating dynamic tools on an embedded system.

Joy et al. describe their suggested technique for finding memory leaks in an embedded system with resource constraints [96]. The article is in the same problem domain as our thesis, and the authors describe the same challenges of resource constraints in their system. They have made a custom tool which works by combining a static analysis method with a dynamic method. It performs static analysis on the assembly code of the program to find potential leaks. Then, it analyses the program dynamically, to filter out false positives from the leaks that were reported by the static analysis. In the static phase, their tool creates a control flow graph where the nodes are a collection of assembler instructions, grouped by which instructions are considered to be a "block". Examples of blocks are functions and loops. Their tool then uses an algorithm to analyse the graph and look for patterns involving memory requests for the specific target architecture. The dynamic part involves running the program in a simulated model of the hardware instead of the actual hardware. Using a virtual version of the hardware enables the use of their tool early in development.

Volvo does not have a virtual model of their hardware, but the discussed solution in this article could give insights and inspiration for future development plans. The strength of their method is the combination of both static and dynamic analysis in one tool. The authors argue that their combined approach gives synergies such as multiple path evaluations in the static part and accurately confirming leaks in the dynamic part. They compare their tool to Valgrind, and report that they are able to get more fine grained information about the source of the leak. For example, when memory is allocated in a loop, the authors write that Valgrind only reports the total number of leaked bytes for each allocation in the loop, whereas their tool reports the precise number of bytes leaked at a specific loop count. Furthermore, the authors write that their method is platform independent, since they use a simulated model of the hardware that can be adapted to any type of architecture. It is an interesting approach that shows how future tools could work for finding memory leaks in an embedded environment. No released version of their tool exists at this time. Their work on embedded systems relates to our work since it shows how the constraints of an embedded system leads to the development of different techniques of analysis, as shown in this case for limiting the overhead. Our work evaluates dynamic tools and gives an insight into which methods cause larger overheads compared to others.

Researchers working in cooperation with Hyundai-Kia Motors have written an article, in which they propose a method for finding memory-related errors in embedded systems [127]. Their use-case is very similar to ours as they also test an embedded system running in a car. In their paper, they analyse an "*automotive infotainment system*" and they describe the constraints imposed by such a system, such as the limited amount of free RAM available to run profiling tools. Furthermore, they describe how the creation of simulations of connected hardware signals could be very costly, since such signals often come from devices that are tightly coupled to other ECUs. Their approach to analysis of the system works at a stage they call "*system operational test*", where the requirements for the whole system are verified during testing in an operational environment. That means the system behaves as close as possible to how it would in the released product. Instead of using existing tools, their proposed solution was to create their own tool to accomplish the analysis. Their method consists of hacking and utilising information in the *Process Control Block* (PCB) in the OS kernel. The PCB is a structure which is handled by the operating system and contains information about all

the running processes. In their method, they collect run-time information about executing processes from the PCB. They then wrap memory operation function calls with their own functions. Their code, in turn, calls the original function after performing some logging. The logs can then be analysed to find different memory problems, such as memory leaks. Their solution has several benefits, most notably testing the whole system during operation, including all running programs.

The solution could efficiently find many memory problems with very little time and memory overhead (slowdown factor of about 0.084–0.132x instead of 11-26.5x as in other tools). The tool, which is not publicly available, had been used with success for 3 years in production verification at Hundai-Kia and successfully found hundreds of memory-related bugs. They could also run their tool without recompiling, re-linking or modifying any of the source code of the programs. In the article, they provide a detailed description of the concept of their method. However, they carried out their work on the BlackBerry QNX operating system, whereas we are working with embedded Linux. Implementation of their method would presumably involve modifying code running in kernel space. The article is relevant to our work because it shows how other automotive companies are working with dynamic analysis. Furthermore, it shows that the tools used do not have to be limited to analysing a single program. A system wide approach can be used too.

In [115], the method used in [127] is tested on another system. The authors describe how they used the same PCB hacking method to find memory leaks, but on an Android system. They look at memory leaks which occur in two cases. First, they focus on the garbage collector running in the *Dalvik* virtual machine [2], which collects memory information and frees memory for applications running in this layer. In this case, memory can be leaked if it is not collected. In the second case they investigate memory that is allocated in the kernel layer. They test their method on 35 different android applications running on smartphones. They conclude that their method was efficient in finding memory leaks with very low system overhead. They do not provide a very in-depth technical description of how they manage to find, access, extract, and analyse the correct PCB information from the kernel, which could make reproducing their method difficult. This paper is relevant to our thesis since it shows that system wide analysis might be possible on platforms running embedded Linux. The tools we have evaluated only analyse one program at a time. This article suggests that a tool which analyses the whole system is possible to create.

The related works mentioned in this chapter concern the central problem we are investigating: finding memory errors in an embedded system. The papers that discuss dynamic analysis of embedded software describe similar constraints to our own, such as limited resources. Several of the proposed dynamic solutions involve wrapping the normal allocation and de-allocation functions to keep track of memory accesses. Most of the tools we investigate in this thesis are implemented using similar methods. Many of the papers discuss the problem of overhead of the dynamic tools. We did not, however, find any papers that evaluate existing dynamic tools, like we have done in this thesis.

Chapter 3
Method

"Program testing can be used to show the presence of bugs, but never to show their absence"

— Edsger Dijkstra *[72]*

In this chapter, we describe the method we used for our study. The first thing we did as we started our project was to conduct a literature review in order to gain more knowledge about the subject. After the literature review, we searched for and made a list of memory analysis tools we thought could be relevant. We then tried to use each of these tools on the TCAM and removed the ones we could not get to work from our list. Following this initial evaluation, we were left with 14 tools to study. We evaluated those tools on two test suites containing documented memory errors, to see which and how many errors each tool could detect. Then, we evaluated the tools on 19 different benchmark programs to see how large of an overhead they introduce on the programs they analyse. We measured the overhead both in execution time and in memory consumption. Using the results we got from our evaluations we revised our tool selection to a number of tools we thought could be useful for analysing Volvo's programs. We then tried to analyse Volvo's programs with these tools, as described in Section 3.6. Finally, based on the tools' ability to analyse the programs, we made a final tool selection containing the tools we consider most useful for finding memory errors.

3.1 Literature Review

In the beginning of our project we spent most of our time conducting a literature review. The aim of our literature review was to learn more about the subject, as it would be helpful when we subsequently evaluated the different tools. We also thought this was a good way of finding suitable analysis tools. We looked for relevant papers using Google Scholar [16], and in the

reference lists of other papers. We put the papers we found, but had not yet read, in a list. Then, after we read a paper, we wrote a short summary of it and put it in a list with the other papers we had looked at. We focused on reading articles with descriptions of methods for finding memory leaks and other memory errors, mainly in C and C++. In most of the articles, we also read about tools which the authors had implemented to test their proposed methods. In some cases, we were able to find these tools on the authors' website or in publicly available code repositories on GitHub [13]. In other cases, the tools were available commercially or as free software.

We found many of the tools we read about potentially relevant. We added these tools to a list so that we could find them again when we wanted to evaluate them.

3.2 Initial Tool Selection

During the literature review, we compiled a list of potentially relevant tools. We used this list as a starting point, to which we then added tools that we found by searching online, on websites like StackOverflow [38] and other software developer forums. By searching on forums online, our goal was to find tools which were being used by developers in general, on all systems. Since we did not have a filter for which tools to add to the list, we ended up with just short of 50 following our search.

The next thing we did was to investigate each tool in the list more thoroughly, most often by reading its manual and "README" documents. We then eliminated tools which we did not consider relevant for our cause. For example, we considered tools which were not usable with C or C++, and tools which were not actually meant for detecting memory faults or memory leaks, irrelevant.

After doing the initial investigation, we tried to compile the remaining tools for the TCAM. Compiling a tool was not always straightforward, and sometimes we made small modifications in its code to make it more useful to us. In Appendix B, we describe how we compiled the tools that we managed to run on the TCAM, and outline other modifications we made. Once we managed to compile a tool, we used it to analyse a simple "hello world" program on the TCAM. If we managed to analyse the program without crashes we kept that tool for further testing. Our reason for not keeping more tools was that if we could not compile a tool for and run it on the TCAM, it was not relevant and we eliminated it from the list.

After this initial evaluation, we had removed all but 14 tools. The tools that remained are presented in Table 3.1.

Initial Tool Selection	
Debug_new	[5]
Memwatch	[29]
Heaptrack	[20]
Heapusage	[21]
Valgrind (Memcheck)	[41]
Leaktracer	[23]
Gperftools Heapcheck	[17]
AddressSanitizer	[1]
Duma	[7]
Malloc_count	[28]
Libleak	[25]
Mtrace	[31]
Rmdebug	[35]
Dr. Memory	[6]

Table 3.1: List of remaining tools after the initial tool selection.

3.3 Benchmarks

To differentiate the tools in the initial tool selection, we wanted to know how effective they are at finding memory errors and how much of an overhead they introduce on the programs they analyse. To evaluate this, we decided to use test suites and benchmarks. Our reasons for using test suites and benchmarks were that we then could evaluate our tools on multiple programs and we would not have to spend time writing our own test programs. Another reason for using the benchmarks instead of doing performance testing on Volvo's programs was that we could not measure the execution time on Volvo's programs, since they are meant to be run indefinitely. We chose to use publicly available test suites and benchmarks both because they were easier for us to access and because anyone who wants to reproduce our study can use them.

3.3.1 Memory Errors

To be able to evaluate how useful each tool was for finding different types of memory errors, we first had to identify which problems we wanted to look for. To do this we worked in three stages:

1. We had a meeting with Volvo developers and made notes of the memory errors they thought were of most importance.

2. We complemented the list with errors we ourselves regarded dangerous.

3. We made use of the *Common Weakness Enumeration* (CWE), which is a system for categorising software weaknesses as described in Section 2.4. We also looked at the "CWE top 25", which is a list of the top 25 most dangerous software errors according to the CWE team. In the CWE, we identified entries related to memory errors, compared them with our list, and added the ones we were missing. We also paired each entry in our list with its corresponding CWE code. We did this to specify exactly what kind of error each entry represented and to make sure its definition was not tied to subjective interpretation by different parties.

By creating the list as described above, we listened to Volvo's own concerns and then complemented the list with errors from two other sources. The method gave us confidence that the list would reflect the most important cases. The full list can be seen in Table 2.2 in Section 2.4.1.

Based on our list of memory related errors, we found two benchmark suites which we could use to evaluate which memory errors each tool could find. The first suite, the Toyota ITC static analysis benchmarks [39], contains tests written in C with introduced memory errors. The original benchmarks contained some additional errors which were not supposed to be there, so we used a version of them [11] where the errors had been fixed. We found this version of the benchmarks in [92]. The authors of the article argue that a test for one specific error should not contain other errors.

The Toyota test suite fit our needs very well, except that it contained very few tests written in C++. We wanted to evaluate our tools on code written in C++ too, because the software running on the TCAM is written both in C and C++, so we looked for similar benchmarks written in C++. We found what we wanted in the Juliet test suite [22]. The Juliet test suite was created by the NSA's Center for Assured Software [32]. It is intended for testing static code analysis tools, but some tests had exactly the errors we wanted to test. We selected a subset of the tests to be used for evaluating which memory errors the various tools could find in programs written in C++. Since the test cases were labelled with CWE codes, it was easy for us to select cases relevant to our research.

3.3.2 Overhead

When we started looking for a benchmark to use for measuring the overhead in execution time and memory consumption of each tool, we were considering the SPEC CPU [36] benchmarks, since they were used in many of the articles we had read during the literature review. However, as we did not have access to any SPEC CPU benchmark suite, we decided to look into alternatives. The benchmarks we decided to use instead were a combination of different sources. We used four benchmarks from the MiBench suite [30], which has benchmarks intended for embedded architectures. We also decided to use three common Linux utilities as benchmarks: *grep* [18], *find* [10], and *gzip* [19]. In addition to these, we used three benchmarks from the *llvm test suite* [27]. Two from the *MallocBench* directory and one from the *VersaBench* directory.

The aforementioned benchmarks are all written in C. Since the programs that run on the TCAM contain code written in both C and C++, we wanted to use benchmarks written in C++ as well. We found a suitable collection of these in the *llvm test suite*. We used eight benchmarks from the directory *Prolangs-C++* and one from the directory *tramp3d-v4*. All of the benchmarks we used are listed in Table 3.2, and they are described more in-depth in the next section.

We had various reasons for choosing the different benchmarks. In general we wanted to use a variety of different kinds of programs so we could see if there were any differences in overhead depending on for example how large or how many memory allocations a program made. We also wanted the benchmarks to allocate memory on the heap since many tools only caught errors related to dynamic memory. For example, we chose the *dijkstra*, *espresso*, and *primes* benchmarks because they make many small allocations on the heap. Our reason for choosing *fft* and *family* was because they allocate most of the memory at the start of the

program. We chose to use *cjpeg*, *madplay* and *shapes* because they are quite small programs that run fast. We chose to use *find*, *grep*, and *gzip* because they were commonly used in the articles we read, and they are well-used utility programs which are not just written to be benchmarks.

The C++ programs were selected from the *llvm-test-suite* based on two criteria: they should use dynamic memory allocations and they should not use any random input. Since we could not find any alternative benchmarks in C++ we decided that hose criteria was enough for our selection.

C Benchmarks:	C++ Benchmarks:
dijkstra	city
fft	employ
madplay	family
cjpeg	primes
8b10b	simul
cfrac	deriv2
espresso	life
find	shapes
grep	tramp3d-v4
gzip	

Table 3.2: List of benchmarks used for measuring overhead.

Description of Benchmarking Programs

The programs we used for benchmarking all do some kind of dynamic memory allocation and finish reasonably fast. Since the code for all the benchmarks is publicly available, it is possible to reproduce our testing. The *grep*, *gzip*, and *find* benchmarks require inputs, so we created a folder with books. In this folder we put 15 text files, corresponding to the top 15 public domain books from *Project Gutenberg* [34]. The list of books can be seen in Table C.1 in Appendix C.

We used the following benchmarks written in C:

- **8b10b** ***(VersaBench LLVM)*** – Implementation of IBM 8bit/10bit block encoder. Used with `input.txt 2000`, where `input.txt` is included in the repository.
- **cfrac** ***(MallocBench LLVM)*** – Implementation of "*continued fraction factorisation algorithm*". Factors an integer into two large prime factors. Used with the integer `327905606740421458831903` as input.
- **espresso** ***(MallocBench LLVM)*** – Implementation of the "Espresso algorithm", which is used for reducing boolean expressions to reduce complexity in logical gates. Used with the following input: `-t INPUT/largest.espresso` which is included in the repository.
- **find 4.7.0** ***(GNU software)*** – GNU program used for searching in directories. We used it like this: `./find books/`
- **grep 3.4** ***(GNU software)*** – Grep searches one or more input files for lines containing a match to a specified pattern. We used it like this: `./grep -risn "\n" books/`

- **gzip 1.10** ***(GNU software)*** – A data compression program. We used it like this: `./gzip -rk books/`
- **FFT** ***(MiBench)*** – Fast Fourier Transform implementation.
- **dijkstra** ***(MiBench)*** – Implementation of Dijkstra's algorithm.
- **cjpeg** ***(MiBench)*** – JPEG encoder.
- **madplay** ***(MiBench)*** – MPEG audio decoder.

We found all our C++ benchmarks in the LLVM repository, in the folders *Prolangs-C++* and *tramp3d-v4*. Most of the programs in *Prolangs-C++* are based on various textbook examples and are not for any real world application, the same is true for *Tramp3d-v4*. Their purpose in the *LLVM repository* is benchmarking, which is what we wanted to use them for too.

We used the following benchmarks written in C++:

- **city** ***(Prolangs-C++ LLVM)*** – Simulation of cars and trucks driving on roads with stop lights in a city. Based on a C++ text book example with the source given as "*'An enabling optimization for C++ virtual functions' by Kuhn and Binkley*". No arguments are passed when running this program.
- **employ** ***(Prolangs-C++ LLVM)*** – An example with different kinds of workers (e.g. hourly, salary, by-piece). Calculates different workers earnings after a certain amount of weeks. The program is run with `./employ 400 < inputEmploy.txt`, and *inputEmploy.txt* is included in the repository. The source of the benchmark is listed as "*An enabling optimization for C++ virtual functions' by Kuhn and Binkley*".
- **family** ***(Prolangs-C++ LLVM)*** – The original source is given as "*listings 7.1 and 7.2 from 'The C++ Workbook' by Wiener and Pinson*". It is a simple program for demonstrating classes and inheritance. It prints the name of a person, which calls different functions depending on which class the person belongs to (parent, child, grandparent). No arguments are passed when running this program.
- **primes** ***(Prolangs-C++ LLVM)*** – Original source listed as *"Ravi Sethi's book on Programming Languages"*. Prints all primes smaller than 100001. No arguments are passed when running this program.
- **simul** ***(Prolangs-C++ LLVM)*** – Original source given as "*Pages 396-403 with header files from Ch 7. 'Using C++' by Bruce Eckel, McGraw Hill, 1989*". Simulation of drawing on different grids. No arguments are passed when running this program.
- **deriv2** ***(Prolangs-C++ LLVM)*** – It was written for an undergraduate C++ course project at the Dept. of Computer Science, Rutgers University. Calculates the derivative of a mathematical expression. No arguments are passed when running this program.
- **life** ***(Prolangs-C++ LLVM)*** – Original source given as "*Section 9.6, 'Object-Oriented Programming using C++' by Ira Pohl. Benjamin/Cummings Publishing Company, 1993*". Simulation of different animals and plants living in a world where they can eat each other and grow old. No arguments are passed when running this program.

- **shapes** ***(Prolangs-C++ LLVM)*** – Original source listed as "*Section 6.4 in Bjarne Stroustrup's book 'The C++ Programming Language' 2nd Edition*". Draws shapes on the screen. No arguments are passed when running this program.
- **tramp3d-v4** ***(tramp3d-v4 LLVM)*** – Source listed as "*template-intensive numerical program based on FreePOOMA, written by Richard Guenther*". Further described as "*TraMP3d-v4 is a benchmark application extracted from a 3d astrophysical hydrodynamics simulation code that is based on the FreePOOMA C++ template library. It simulates a sedov-explosion on a uniform Cartesian grid by means of solving the Euler set of partial differential equations with an isothermal equation of state*" [40]. We ran the program like this: `./tramp3d-v4 -cartvis 1.0 0.0 -rhomin 1e-8 -n 1 -domain 16 16 16`

3.4 Evaluation

We evaluated the tools in two steps: first we used them to analyse the test suites with memory faults, then we measured the execution time and memory footprint of each tool on the benchmarks we used for overhead testing. By doing a thorough evaluation of both the types of memory faults each tool was able to find and of the impact they had on the performance of analysed programs, we got a comprehensive idea of which tools might be able to analyse Volvo's programs while also being useful for finding errors.

3.4.1 Memory Faults

We measured which, and how many, memory faults each tool found. We did this by using the tools to analyse the test suites described in Section 3.3.1. We did this evaluation on the TCAM, since that is the system which the tools are needed for.

While we analysed the test suites, we also evaluated how accurate each tool was at reporting where the errors were in the source code. We think this is an important aspect to consider given the difficulty pinpointing the errors yourself if the tool does not give helpful feedback. Some tools reported the source file and line number of the error detected in cleartext. Other tools reported an address instead, which meant we had to use a program such as *addr2line* to pinpoint the error in the source code, as described in Appendix A.

In order to get a rough idea of how informative and useful each tool was, we assigned one or more of the following categories to each tool for each type of error:

1. Found the error and pointed to the correct line.
2. Found the error but printed either an address or the wrong line.
3. Did not find any error.
4. Gave a false positive.
5. Found the error and aborted.

3.4.2 Overhead

By using the tools to analyse the benchmarks listed in Section 3.3.2, we could measure their overhead and compare it with the other tools. We measured both the overhead in execution time and in memory consumption. The memory on the TCAM is limited, and therefore it is of importance that the tools do not introduce a large memory overhead on the programs they analyse. It is also important that the tools do not make the analysed program run too slowly, since some programs on the TCAM have time limits on how slow they are allowed to respond to requests.

Execution Time

We measured execution times using *perf stat* [33], perf stat reports the execution time both in task-clock time, and in how much real time has elapsed. We decided to record both these values since that did not take much more effort and we were not sure which one we wanted to use. In the end we decided to use the real time, since the task clock time only measures the time the program is executing on the CPU. We deemed real time more representative of the actual time the program needed, since it includes any time the program spends waiting for resources.

Memory Consumption

Our original plan for measuring the memory footprint of each tool was to analyse it with Valgrind's Massif skin, but this failed. Some of the tools did not work properly and others complained and aborted when run this way.

Instead, we wrote a shared library which we could preload with the tools when they ran. This library sampled the `smaps` file in `/proc/[PID]/` at two different intervals, 59 and 137 milliseconds. Our reason for sampling at two different intervals was to see whether there was any major difference between the results we got from each. If they were similar we reasoned that our sampling was done fast enough.

Our library also created a copy of the `status` file in `/proc/[PID]/` at program shutdown. We did this so that we could record what the peak RSS of the program was before it exited, by looking at the *VmHWM* entry. Finally, we wrote a python script for calculating the mean, median, and peak RSS, PSS, and USS using the memory data we sampled from the `smaps` file. Descriptions of the terms *RSS, PSS, USS* and *VmHWM* can be found in Appendix A, along with descriptions of the `status` and `smaps` files.

3.5 Revised Tool Selection

Of the tools we evaluated on both the overhead benchmarks and on the test suites with memory errors, we chose four to test on two of Volvo's in-house developed programs that normally run on the TCAM. The tools we chose were AddressSanitizer, Debug_new, Heaptrack, and Dr. Memory.

We chose tools which were efficient at finding errors. We also chose tools that could find different types of memory errors and that performed well in our evaluations. We considered

some other factors too. For example, we valued the ability of the tool to analyse C++ code highly, as most of the programs that run on the TCAM are written in C++.

We chose AddressSanitizer because it found many memory access errors, it gave good feedback as to where the errors occurred in the source code, it did not crash when analysing most of the benchmarks, and it did not introduce as large overheads as Valgrind or Dr. Memory did. The reason we chose it over tools like Memwatch and Rmdebug, which introduced smaller overheads on average, was that it found more types of memory faults and it was able to analyse C++ code.

Another tool that we considered instead of AddressSanitizer was Duma. Duma introduced a considerably lower overhead, both in memory consumption and execution time, but it did not find as many types of errors. Additionally, it could not analyse some of the benchmarks, since it crashed. It is possible that this contributed to the lower overheads introduced by Duma. Perhaps the overhead on the benchmarks that crashed would have raised the average.

A different tool we found interesting was Dr. Memory. Like AddressSanitizer, it could find different memory access errors, but it also found memory leaks. Many of the tools we found did not have this ability, and despite it imposing large overheads on the analysed programs, we thought it would be interesting to see whether it could analyse Volvo's programs. Our reason for choosing Dr. Memory instead of similar tools which introduced a smaller overhead, like Memwatch or Rmdebug, was that it found more types of errors and it could analyse programs written in C++. Valgrind also found some of the errors Dr. Memory found, but it did not find as many different types and it introduced a larger memory overhead on the benchmarks. Furthermore, the feedback received from Valgrind was in many cases hard to follow as it often pointed to locations in the standard library.

We also chose to use Debug_new for testing on Volvo's programs. Our reasons were: it imposed little memory and execution time overhead on the benchmarks, it found memory leaks, and it gave adequate feedback when it found an error. Many of the other tools we chose among also found memory leaks. Our primary reasons for choosing Debug_new instead of another tool was that it had very little overhead while still providing reasonable feedback, and it could analyse programs written in C++. Examples of other programs with a low impact on memory consumption and execution time are Malloc_count, Leaktracer, and Mtrace. We did not choose any of these tools because the feedback they gave was not of the same quality as the feedback given by the tools we ended up choosing, and they were the same or worse in other aspects, such as the errors they detected and their overhead.

The fourth tool we chose, Heaptrack, imposed large overheads. For example, it introduced a median execution time overhead of over 9000% on our C++ benchmarks, on average. We wanted to try Heaptrack on Volvo's programs despite this, because it does not just find memory leaks, but also records when memory is allocated. Using this information, it can display the memory usage over time of a program. This can be used for finding memory that is reachable by the program, but not used.

Of these four tools, we were only able to run two when we did the testing on Volvo's programs: AddressSanitizer and Debug_new. Because of this, and the fact that Debug_new did not give as good feedback as we had hoped, we chose two more tools to test: Gperftools and Heapusage.

Our reason for choosing Gperftools and Heapusage was that they both found all memory leaks in the C++ test suite, gave useful feedback for finding the location of the leak in the code,

and neither introduced a lot of memory overhead on the analysed program. The complete revised tool selection can be seen in Table 3.3.

Revised Tool Selection
Gperftools Heapcheck
AddressSanitizer
Debug_new
Heapusage
Heaptrack
Dr. Memory

Table 3.3: List of the revised tool selection after the testing phase.

3.6 Testing on Volvo's Programs

Using the tools we chose in the revised tool selection, we analysed two programs on the TCAM: FoundationTransportManager (FTM) and SignalingManager.

FoundationTransportManager (FTM) is responsible for transport layer security and IP connection management, and implements protocols such as HTTP and MQTT.

SignalingManager takes care of incoming and ourgoing signals to the TCAM, for example encapsulating/decapsulating payloads and resends messages according to the correct retry policy.

We were recommended to use these programs by our supervisors at Volvo since they do not have many dependencies on other services on the TCAM. This meant we could close other services if the tools needed more memory. Another reason for analysing these specific programs was that our supervisors have written their underlying code and are familiar with them.

We tested the tools by first trying to get each program to run while being analysed by the tool. Then we used different functions of the program, and after that we recorded the peak RSS of the program and compared it with the peak RSS of the program when it was not being analysed. Examples of program functions we used are: using FTM for sending *HTTP Get requests* and *HTTP Post requests*, and using SignalingManager for sending and receiving MQTT messages. After measuring the memory consumption, we introduced some errors in the source code of the program, and then tried to use the analysis tool to find them.

We introduced four errors in FTM: one heap-based buffer overwrite error, two memory leaks, and one error similar to a use after return error. The heap-based buffer overwrite error and one of the memory leaks were simple errors that we came up with ourselves, but the other two errors had previously been in the code. The reintroduced memory leak occurred whenever we made a HTTP Put, Post, or Delete request. The second reintroduced error was caused by writing to an array allocated on the stack, after the stack frame in which it was present had been popped.

We introduced three errors in SignalingManager: one memory leak, one heap-based buffer underflow, and one stack-based buffer overflow. All of these were simple errors that we came up with ourselves and introduced in various places in the source code.

3.7 Final Tool Selection

We made our final tool selection based on the results from testing Volvo's programs. The tools we chose are presented in Table 3.4. The selection was based on how the tools performed when used on Volvo's programs. The tools had to run without crashing and be useful to us for finding memory errors in the programs. Initially, we thought we would be able to pick the best options among multiple tools, but when we performed the testing it became clear that only a few of the tools managed to analyse Volvo's programs.

Final Tool Selection
AddressSanitizer
Debug_new
Heapusage

Table 3.4: List of the final tool selection after the Volvo testing phase.

Chapter 4
Tool Survey

"Tools amplify your talent. The better your tools, and the better you know how to use them, the more productive you can be."

— Andrew Hunt *[133]*

There exist many different memory analysis tools. A majority are available online, some you have to pay for, and some can be used free of charge. Additionally, many of the publicly available tools have an open source license, meaning that their source code is freely available for modification and use. As described in the previous chapter, we were left with 14 tools after our initial tool selection. In Section 4.1 we give a general description of how we used the tools on the TCAM. Then, in Section 4.2, we outline the functionality of these tools. We describe a few tools more in detail, which highlight the different dynamic methods. In Table 4.1, we present a summary of the different implementation methods used by each of the 14 tools.

4.1 General Usage of Tools

For us to be able to use the analysis tools on the TCAM, they had to be compiled for the TCAM's architecture. Some of the tools were available as pre-compiled binaries for ARM, but in most cases they were not. This meant we had to compile them ourselves. All tools we used were either written in C, or in C++. Because of this, we could use the compilers used by Volvo developers for compilation of TCAM software. Some tools required libraries which were not present on the TCAM, which we had to cross-compile too.

For the tools to give reasonable feedback, we compiled the benchmark programs with debug information (the `-g` flag in GCC). The debug information makes it possible to trace a machine instruction back to the line in the C/C++ file which produced it [12].

Furthermore, we compiled the benchmark programs with the GCC flags `-fno-inline` and `-fno-omit-frame-pointer`. This was to avoid optimisations from the the compiler where it inlines some functions and removes frame pointers for some functions, both of which can make the tools less informative.

We compiled many of the tools as shared libraries, which we preloaded when running a program we wanted to analyse. To preload a library, we set the `LD_PRELOAD` environment variable to point to the library file. Preloading a library ensures it will be loaded before other libraries. That means any function name is looked up in the preloaded library before searching elsewhere, such as the standard library. This is useful because many of the tools work by wrapping the allocation and deallocation functions (malloc, calloc, new, etc.). The wrappers can record information and then call the original versions.

Some analysis tools provided functions which had to be called during run-time by the program, usually to start analysing. Most of the time, these tools were intended to be compiled with the program they analysed. To avoid this, we compiled the tools as shared libraries, with a constructor that called any required function for starting the analysis. We added these libraries to the library path, by appending them to the `LD_LIBRARY_PATH` environment variable, when we wanted them to analyse a program.

A detailed description of how we compiled and ran each tool on the TCAM is presented in Appendix B.

4.2 Implementation Details of Tools

All of the tools we evaluated are open source, which let us investigate and modify their source code. Most of them are also well documented, with some being extensively described in scientific articles. In this section we describe different methods used in dynamic memory analysis by using some of the tools we evaluated as examples. At the end of the section, in Table 4.1, we briefly summarise how the tools that could run on the TCAM are implemented.

4.2.1 Valgrind

Valgrind is a framework for dynamic analysis [41]. It consists of a core which provides basic infrastructure for performing program instrumentation and various “skins” which perform different kinds of analysis.

According to an article written by the creators of Valgrind [113], the Valgrind core carries out just-in-time compilation from machine code to an intermediate code (called *UCode*), and then back again. Valgrind uses UCode to reduce complexity. Instructions for instrumentation can be inserted in the UCode representation, which is then compiled back to machine code. The just-in-time compilation and the UCode are useful because they allow for easy instrumentation of a program without recompiling it, but as a consequence of this compilation the program runs slower.

Memcheck is a Valgrind skin for detecting memory faults, as described in [114]. The authors accredit Memcheck for Valgrind’s popularity, and they write that user surveys have

indicated that it accounts for more than 80% of Valgrind tool use. It can detect most errors, such as memory leaks, access-after-free, use of uninitialized memory, and heap-based buffer overread. It can detect all these things by using shadow memory and fine grained instrumentation. It also uses shadow registers, which is like shadow memory, but for registers. This advanced fault detection comes at a price. When analysing a subset of the SPEC2000 [37] benchmarks, Memcheck caused a median slowdown of 28.7x and a median code expansion of 12.6x. Memcheck also introduces an overhead in memory. According to [113], it uses an extra 9 bits of memory per byte of addressable memory.

4.2.2 Dr. Memory

Dr. Memory is a tool which is inspired by Valgrind, and created by a collaboration of two engineers from Google and MIT [58]. Dr. Memory works at run-time and it does not require accesses or modifications to the source code. The tool can find a variety of memory-related errors such as uninitialised memory access, accesses to unaddressable memory (including outside of allocated heap units and heap underflow and overflow), accesses to deallocated memory, double frees, and memory leaks. The tool is supported on a wide variety of CPU-architectures, ARM being one example. However, full support of tool capabilities is not offered on all platforms. For example, on ARM platforms, the tool is unable to check for uninitialised reads. In order for the tool to give useful feedback, the analysed program needs to be compiled with dwarf version 2 debug information [8]. According to [58], Dr. Memory reports few false positives and is on average twice as fast as Valgrind's Memcheck skin.

Dr. Memory uses shadow memory to keep track of memory accesses, and it also adds red zones around memory allocations on the heap.

Similar to Valgrind, Dr. Memory executes the analysed program in a virtual machine. This allows fine grained instrumentation of the analysed program.

In the article, the authors mention that only those reads which impact the program behaviour are reported. Examples of such cases are comparisons for conditional jumps or data passed to system calls. Dr. Memory does not only shadow the memory of the analysed program, but also the registers.

4.2.3 AddressSanitizer

AddressSanitizer is a dynamic memory analysis tool. Like Memcheck and Dr. Memory, it can catch memory errors other than memory leaks. It employs a slightly different approach though. Instead of instrumenting the program instructions at run-time, they are instrumented at compile time. As a result of this instrumentation, programs that are compiled with AddressSanitizer can not run without being analysed. Additionally, any potential overhead caused by instrumentation or just-in-time compilation at run time is avoided.

AddressSanitizer uses shadow memory to represent the real memory. The shadow memory is updated by instrumentation instructions that are inserted at memory accesses in the program code. In addition to instrumenting the code, a part of AddressSanitizer is loaded as a shared library at program start. This library manages the shadow memory and overloads the memory allocation functions. The overloaded allocation functions allocate extra memory for red-zones around the requested memory and store the current call stack to be able to give better feedback if an error is found. AddressSanitizer also creates red-zones around global

arrays and arrays allocated on the stack. The techniques employed by AddressSanitizer are described more in depth in [128].

4.2.4 Debug_new

Debug_new is a dynamic memory analysis tool, mainly designed for finding memory leaks. It can be loaded as a library at run-time, but it can also be compiled together with the program it is intended to analyse. Debug_new can be used in two slightly different ways: we could choose to include *debug_new.h* in the source files of the program we wanted to analyse, or we could refrain from including it and just preload the library.

If we chose to include the header file, *new* and *delete* were replaced by macros. If we chose to only load the library, *new* and *delete* were overloaded by functions defined in the library. Both the macros and the functions wrap the original *new* and *delete*, and they keep track of which memory is allocated and the current call stack at each function call. Debug_new then reports the allocations that were not deallocated at program exit as memory leaks.

4.2.5 Heaptrack

Like Debug_new, Heaptrack can be used to find memory leaks. It is also a memory profiling tool, which can show how much memory the analysed program has allocated on the heap over time. Heaptrack comes with a script that can be used for convenience, but the user can also manually load the Heaptrack library when running a program to be analysed.

Heaptrack wraps the memory allocation and deallocation functions (malloc, realloc, free, etc.), which allows it to keep track of allocated memory and save stack traces. It also keeps track of how much memory is allocated at a certain point in time. If leak detection is enabled, Heaptrack reports memory that is still allocated as leaks.

4.2.6 Heapusage

Heapusage also reports memory leaks. It can be run using either a script which comes with the tool, or by loading the Heapusage run-time library manually.

Heapusage implements its own versions of the allocation functions, which wrap the versions in the standard library. The new allocation functions also log every allocation and deallocation and save stack traces. Just like Heaptrack and Debug_new, Heapusage reports any allocations not deallocated at program exit as memory leaks.

4.2.7 Gperftools

Gperftools is part of *tcmalloc*, which is a custom memory allocator. We evaluated the Gperftools heap checker, which is a memory leak detector. Since using Gperftools requires using tcmalloc, the performance can be affected by this factor too. For example, tcmalloc might be either more or less memory efficient than the allocator in the standard library.

Gperftools heap checker tracks memory allocations and deallocations by being part of the memory allocator. It is able to report all memory that has not been deallocated at program exit, just like Debug_new, Heaptrack, and Heapusage. However, it differs from these

tools in certain aspects, since it can be configured to only report memory allocations that are not reachable via any pointers as leaks. It figures out which memory is unreachable by doing a "liveness flood", which means it searches through global data segments, the stack, and any reachable dynamic memory for pointers to allocations. Any allocations which were not reachable during the liveness flood are then reported as leaks. Another benefit of this way of finding memory leaks is that it can be done while the analysed program is still running.

Tool	**Implementation method**
Valgrind	Instruments the program code at run-time. Uses shadow memory.
Leaktracer	Preload library which wraps the allocation functions, logs allocations and calls the original allocation functions. Tracks allocations in a list and reports still allocated memory at program exit as leaks.
Gperftools Heapcheck	Preload library which overrides allocation functions. Implements its own allocation functions instead of calling standard library. Tracks allocations in a list, can report either still allocated memory, or unreachable memory, as leaks. Can also profile a program by keeping track of how much memory is allocated at different points in time.
AddressSanitizer	Works at compile-time, instruments an intermediate representation of the source code, after optimisation. Uses shadow memory and red-zones.
Duma	Preload library which wraps the allocation functions, logs allocations and calls the original allocation functions. Adds red-zones around allocated memory and finds accesses to them by using the *mprotect* system call.
Dr. Memory	Instruments the binary program code. Uses shadow memory.
Heapusage	Preload library which wraps the allocation functions, logs allocations and calls the original allocation functions. Track allocations in a list and reports leaks.
Heaptrack	Preload library which wraps the allocation functions, logs allocations and calls the original allocation functions. Tracks allocations in a list and reports still allocated memory at program exit as leaks. Can also profile a program by keeping track of how much memory is allocated at different points in time.
Debug_new	Preload library which wraps the allocation functions, logs allocations and calls the original allocation functions. Tracks allocations in a list and reports still allocated memory at program exit as leaks.
Memwatch	Library to be included at compile time, overrides allocations, logs calls and calls the original allocation functions. Tracks allocations in a list and reports still allocated memory at program exit as leaks. Also tracks out-of-bounds accesses.
Malloc_count	Preload library which wraps the allocation functions, logs allocations and calls the original allocation functions. Records memory usage at different points of program execution.
Rmdebug	Preload library which wraps the allocation functions, logs allocations and calls the original allocation functions. Tracks allocations in a list and reports still allocated memory at program exit as leaks.
Libleak	Preload library which wraps the allocation functions, logs allocations and calls the original allocation functions. Tracks allocations in a list and reports still allocated memory at program exit as leaks.
Mtrace	Preload library which wraps the allocation functions, logs allocations and calls the original allocation functions. Tracks allocations in a list and reports still allocated memory at program exit as leaks.

Table 4.1: Summary of tools tested and their implementation method.

Chapter 5
Results

"With good program architecture debugging is a breeze, because bugs will be where they should be."

— David May *[43]*

In this chapter we present the results received from testing and evaluating the analysis tools. In Section 5.1 we write about our observations from the initial tool selection. In Section 5.2 we report the results we got from evaluating the tools on the test suites and benchmarks described in Section 3.3. Using the results from our evaluation, we chose a few tools to proceed with, as described in Section 3.5. Our observations from using these tools to analyse Volvo's programs are presented in Section 5.3.

5.1 Initial Tool Selection

We conducted the initial tool selection early on in the project. We were able to eliminate tools which were irrelevant, but the tools that passed did not have to be effective at finding errors. For example, some of the tools we selected did not seem to give any indication as to whether they were analysing the test program. We decided to keep these tools around anyway and evaluate them using the two test suites which contained known errors, since we would rather do a little more work than miss a tool which could be useful.

5.2 Evaluation

As described in the methods section, we worked in steps when we evaluated the analysis tools on the benchmarks and test suites. In this section, we write about the results in the same order. First, we show which types of and how many errors each tool found. Then, how

useful the feedback it gave was for pinpointing the location of the error in the source code. In the last two subsections, we present our findings from measuring overhead in terms of execution time and memory consumption.

5.2.1 Memory Fault and Leak Detection

In total, we evaluated 14 different tools at this stage: Valgrind, Leaktracer, Gperftools Heapcheck, AddressSanitizer, Duma, Malloc_count, Libleak, Memwatch, Debug_new, Mtrace, Rmdebug, Dr. Memory, Heapusage, and Heaptrack. We were not able to make Libleak find any memory leaks or faults, but all other tools were able to find errors. Additionally, not all tools worked for both C and C++. For example, Rmdebug did not work for C++ and Debug_new did not work for C. The number and types of memory faults and memory leaks we found with Valgrind, Leaktracer, Gperftools Heapcheck, AddressSanitizer, and Duma in C++ are presented in Table 5.1.

	Tools				
Fault	Valgrind	Leaktracer	Gperftools Heapcheck	AddressSanitizer	Duma
Memory leak (lost ref.)	x	11/11	11/11	x	x
Double free	x	x	x	x	x
Heap-based Buffer Overflow	x	x	x	3/4	4/4
Heap-based Buffer Overread	1/2	x	x	2/2	2/2
Heap-based Buffer Underflow	x	x	x	2/2	2/2
Heap-based Buffer Underread	2/2	x	x	2/2	x
Stack Overflow	x	x	x	x	x
Stack-based Buffer Overflow	x	x	x	1/1	x
Stack-based Buffer Overread	x	x	x	2/2	x
Stack-based Buffer Underflow	1/1	x	x	1/1	x
Stack-based Buffer Underread	x	x	x	1/1	x
Use after free	x	x	x	6/6	6/6
Use of Uninitialised Variable	x	x	x	x	x
Use after return	1/1	x	x	x	x
Attempt to Access Child of a Non-structure Pointer	x	x	x	1/1	x
Mismatched Memory Management Routines	x	x	x	10/10	10/10
Improper Initialisation	1/1	x	x	1/1	x
False positives	3/59	2/59	x	x	x

Table 5.1: Number and type of memory faults and memory leaks found by the analysis tools in C++.

As seen in the table, Valgrind, AddressSanitizer, and Duma all found various types of memory faults but no memory leaks. On the other hand, Leaktracer and Gperftools seemingly only found memory leaks. This was not surprising, since Leaktracer and Gperftools are not supposed to find memory faults and AddressSanitizer and Duma do not support finding leaks (at least in our system). The few errors found by Valgrind (Memcheck) was a bit of a surprise, as it claims to find more types of errors. We suspect it might not find as many errors because it ran out of memory.

From our evaluation of the tools, it appears that almost every tool finds either memory leaks or other memory errors. One exception is Dr. Memory. The number and type of memory errors found by Dr. Memory in the C++ test suite can be seen in Table 5.2, along with the results of the evaluation of Heaptrack and Heapusage. As seen in the Table 5.2, Dr. Memory found both memory leaks and other types of errors, although not as many as AddressSanitizer. We found both memory leaks and other memory errors with Memwatch and Rmdebug too, although not as many different types as we found with Dr. Memory or AddressSanitizer.

The number and types of memory errors found by Memwatch and Rmdebug are presented in Table C.2 in the appendix, along with the errors found by the other tools in the C test suite. The full list of the errors found by each tool in the C++ test suite is presented in Table C.3 in Appendix C.

	Tools		
Fault	Dr. Memory	Heapusage	Heaptrack
Memory leak (lost ref.)	11/11	11/11	11/11
Double free	x	x	x
Heap-based Buffer Overflow	3/4	x	x
Heap-based Buffer Overread	2/2	x	x
Heap-based Buffer Underflow	2/2	x	x
Heap-based Buffer Underread	2/2	x	x
Stack Overflow	x	x	x
Stack-based Buffer Overflow	x	x	x
Stack-based Buffer Overread	x	x	x
Stack-based Buffer Underflow	x	x	x
Stack-based Buffer Underread	x	x	x
Use after free	6/6	x	x
Use of Uninitialised Variable	x	x	x
Use after return	x	x	x
Attempt to Access Child of a Non-structure Pointer	x	x	x
Mismatched Memory Management Routines	10/10	x	x
Improper Initialisation	x	x	x
False positives	x	1/59	x

Table 5.2: Number and type of memory faults found by the analysis tools in C++.

5.2.2 Tool Feedback

As we evaluated the ability of each tool to find memory errors, we also evaluated how accurately they reported where the errors occurred in the source code. While many tools indicated that there was an error somewhere, they gave varying amounts of information. For example, Dr. Memory, Heapusage, and Debug_new all found memory leaks but they reported them in different ways. Dr. Memory pointed at the correct line in the source code where the memory was allocated and printed a stack trace which made it easier to understand how the code had been executed before the error occurred. Heapusage also printed a stack trace, and pointed to the correct function, but we were not able to make it point to the correct line in the code.

Debug_new was a bit different, because we could use it in two different ways, and the quality of the feedback it gave depended on how we used it. If we included it as a header file in each source file in the analysed program, it pointed to the correct line numbers and printed a stack trace, but if we only preloaded it as a shared library, it printed a stack trace of memory addresses instead.

Some tools, such as AddressSanitizer and Duma, aborted when they found an error. AddressSanitizer still printed useful information for finding the error. Duma, however, is meant to be run in a debugger that can print the stack trace, and did not print any useful information. In Table 5.3, we present what kind of feedback each tool gave.

Tool	Feedback
Valgrind	Pointed at the correct line in the source code, and printed a stack trace.
Leaktracer	Reported allocated memory at program exit, did not indicate where it had been allocated.
Gperftools Heapcheck	Printed a stack trace of memory addresses.
AddressSanitizer	Pointed at the correct line in the code, and printed a stack trace.
Duma	Aborted when it found an error, meant to be run in a debugger.
Dr. Memory	Pointed at the correct line in the source code, and printed a stack trace.
Heapusage	Pointed at the correct function, not the correct line, in the code. Printed a stack trace.
Heaptrack	Pointed at the correct line in the code, and printed a stack trace.
Debug_new	Depending on how we used it, it either pointed correctly and printed a stack trace, or it printed memory addresses.
Memwatch	Pointed at the correct line in the code, and printed a stack trace.
Malloc_count	Reported allocated memory at program exit, did not indicate where it was allocated.
Rmdebug	Pointed at the correct line in the code, and printed a stack trace.
Libleak	Did not find any errors.
Mtrace	Pointed at the correct line when analysing C code. When analysing C++ code, Mtrace pointed at libstdc++.

Table 5.3: Feedback given by each tool when reporting errors.

5.2.3 Execution Time

After we had evaluated how effective the tools were at finding errors, we moved on to evaluating how much of an impact in execution time they had on the programs they analysed. We did not feel any need to evaluate Libleak since we could not find any errors at all with it. We evaluated all the other tools, and measured their impact. Our measurements of the execution times are presented in Tables C.4 to C.17 in the appendix.

Using these execution times we calculated the overhead introduced by each tool, to make them easier to compare. The median overhead in execution time for each tool, averaged over all of the C++ benchmarks, is presented in Figure 5.2.

As seen in the figure, some tools made the program they analysed run a lot slower. Valgrind, Dr. Memory, and Heaptrack had the most impact on the execution time of the program they analysed, and Duma, Malloc_count, and Debug_new had the least. This impact is for the benchmarks written in C++ though. We were not able to analyse programs written in C++ with two of the tools, Memwatch and Rmdebug. These two tools performed well. As shown in Figure 5.1 they had little impact on the execution time of our C benchmarks, so it is unfortunate we could not get them to analyse the C++ benchmarks.

Graphs of the median overhead in execution time introduced on each benchmark by each tool are presented in Figures D.1 to D.13 in the Appendix.

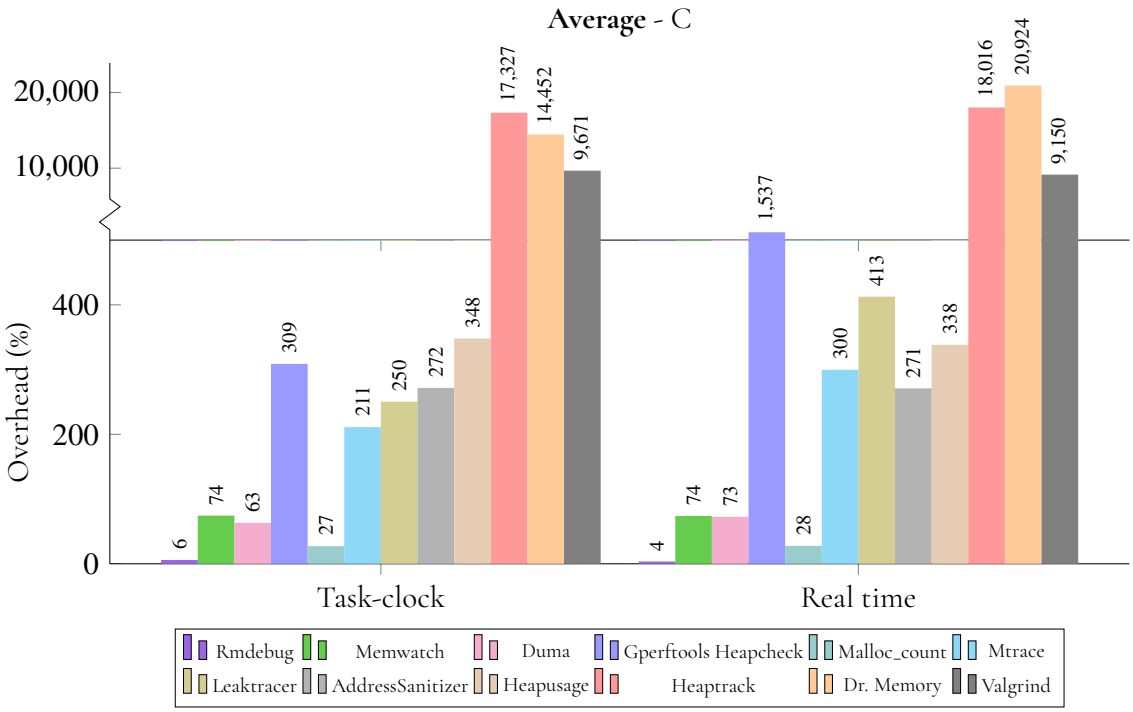


Figure 5.1: Average of the median execution time overhead, for all tools which were able to analyse the benchmarks written in C.

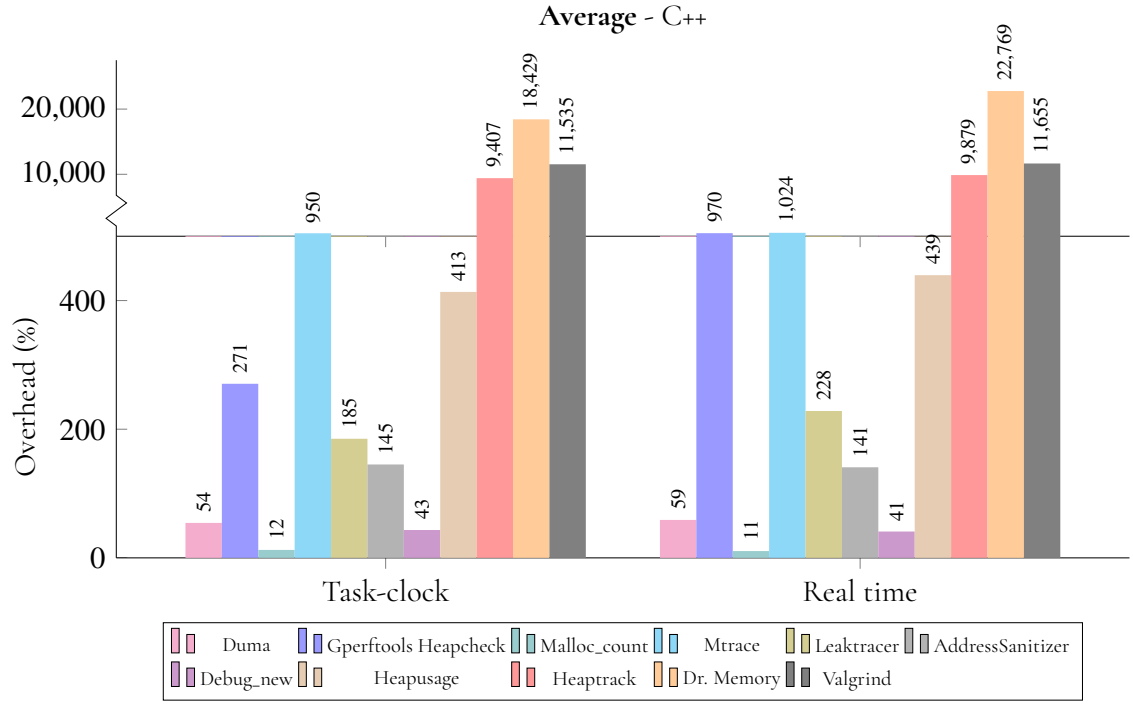


Figure 5.2: Average of the median execution time overhead, for all tools which were able to analyse the benchmarks written in C++.

5.2.4 Memory Consumption

When we had measured and calculated the execution time overhead of the tools, the next step was to find out how much memory overhead each tool introduced. We measured the memory consumption of each program in RSS, PSS, and USS. The results of our measurements are presented in Tables C.18 to C.31 in Appendix C.

Using our measurements, we calculated the average overhead in peak RSS and peak USS for each tool. The overheads introduced on the C++ benchmarks are presented in Figure 5.2, and the overheads introduced on the C benchmarks are presented in Figure 5.3.

As seen in the figures, AddressSanitizer, Dr. Memory, and Valgrind introduced the largest memory overheads. The average overheads introduced by some of the tools were disproportionately affected by a few benchmarks, for example the *dijkstra* benchmark. Valgrind and Heaptrack both introduced large overheads on this benchmark, especially on USS, as seen in Figures D.23 and D.24 in Appendix D. Another thing to note is that the overhead, in peak USS, is displayed as 0 for some tools in the figures. That means we measured the benchmarks as consuming equal to or less than the amount of memory compared to when the benchmarks were not analysed by that tool.

We also noticed a difference in overhead between peak RSS and peak USS, the overhead introduced on peak USS was larger than the one introduced on peak RSS for most of the tools. Perhaps this could be due to RSS being larger than USS and the introduced overhead not being entirely proportional to the memory consumption of the program.

Graphs of the peak overhead in RSS, PSS, and USS, as well as the median overhead in USS, introduced on each benchmark by each tool are presented in Figures D.14 to D.26 in Appendix D.

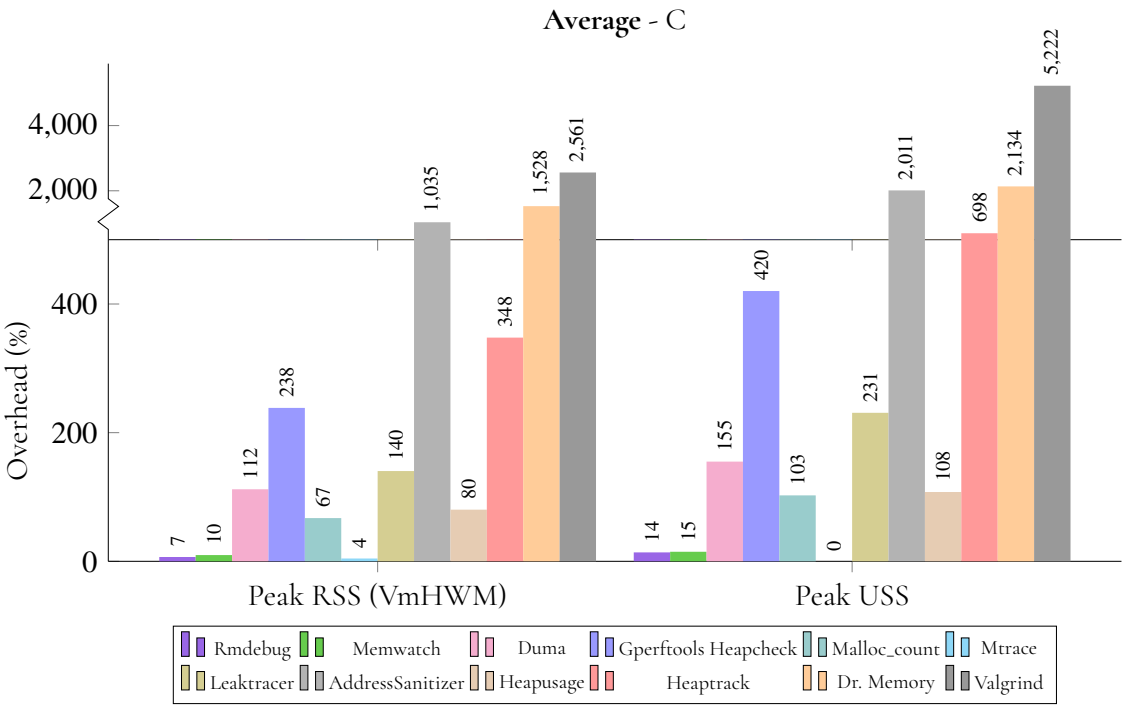


Figure 5.3: Average memory overhead introduced by each tool on the C benchmarks.

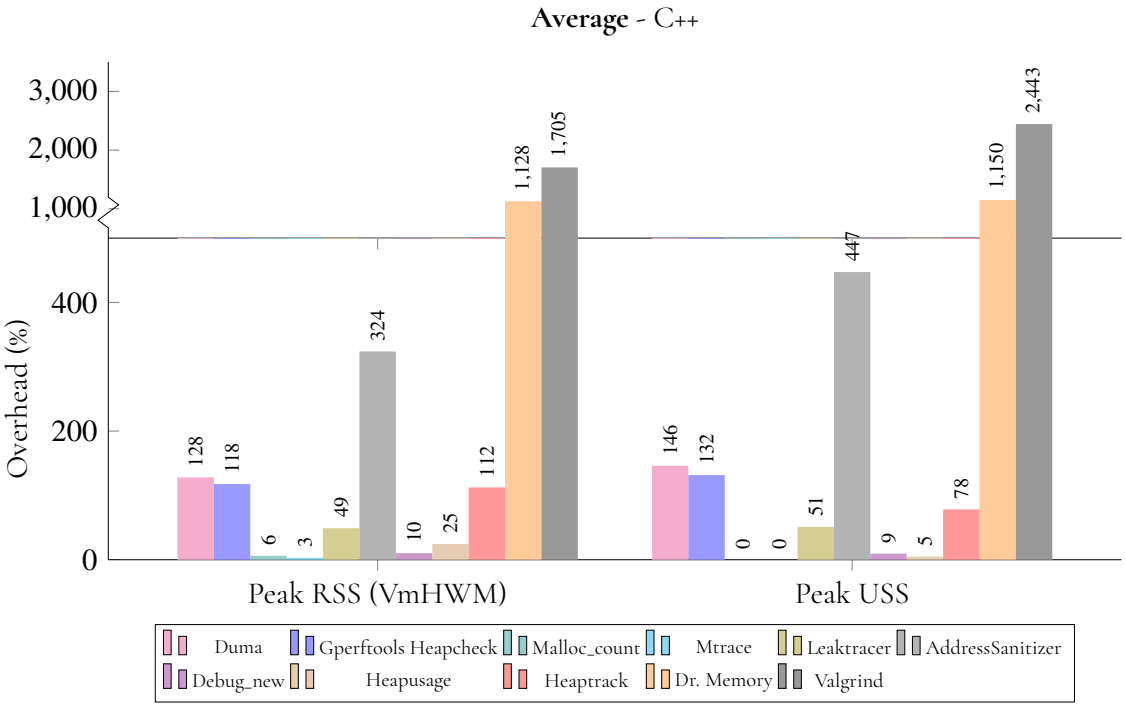


Figure 5.4: Average memory overhead introduced by each tool on the C++ benchmarks.

5.3 Analysing Volvo's Programs

We used AddressSanitizer, Debug_new, Heaptrack, Dr. Memory, Gperftools, and Heapusage to analyse some programs that normally run on the TCAM, to see both whether they would be able to analyse these larger programs and if they would be able to find any errors. The programs we analysed were FoundationTransportManager (FTM) and SignalingManager. We introduced our own errors in these programs, as described in Section 3.6. We measured the peak RSS of the programs when they were not being analysed. The peak RSS of FTM was 8260 kB, and the peak RSS of SignalingManager was 9784 kB.

5.3.1 AddressSanitizer

The first program we tried analysing using AddressSanitizer was FTM. The program managed to run when being analysed, but the TCAM started making a noise and we saw that it consumed more current than usual. The noise stopped and the current consumption went down when we closed some other services, so we continued our evaluation without all services running. We measured the peak RSS of the program to be 27536 kB when it was being analysed. This meant that AddressSanitizer introduced an overhead in peak RSS of about 233%, compared to running FTM without analysing it.

Of the errors we introduced in FTM, we were able to find two using AddressSanitizer: the heap-based buffer overflow and the use-after-free. The feedback printed by AddressSanitizer was clear and understandable, and it led us to the places in the source code where the errors had occurred.

We also used AddressSanitizer for analysing SignalingManager. To be able to run SignalingManager we had to stop a few other services from running, just like we did when we analysed FTM. If we did not do this, the TCAM became exceedingly slow, with some services crashing and never restarting. We managed to run the function tests for SignalingManager while it was being analysed, and they all passed. After running the tests, we measured the peak RSS to be 28472 kB, which means AddressSanitizer introduced an overhead of 191% in peak RSS.

Using AddressSanitizer, we were able to find two of the errors we introduced in SignalingManager: the stack-based buffer overflow, and the heap-based buffer underflow. AddressSanitizer pointed correctly to the location in the source code of both errors, but it labelled the underflow as an overflow, as can be seen in Figure 5.5.

```
=================================================================
==7122==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xb3113c6c at pc 0x7f5c1904 bp 0xbe88b9e4 sp 0xbe88b9dc
WRITE of size 4 at 0xb3113c6c thread T0
    #0 0x7f5c1903 in sigm::SignalingManager::Start() /home/ebengt11/Documents/new_tcam/telematics/framework/SignalingManager/signaling_manager.cpp:344
    #1 0x7f5aab37 in main /home/ebengt11/Documents/new_tcam/telematics/framework/SignalingManager/signaling_manager.cpp:477
    #2 0xb5962cbf in __libc_start_main (/lib/libc.so.6+0x16cbf)
0xb3113c6c is located 4 bytes to the left of 80-byte region [0xb3113c70,0xb3113cc0)
allocated by thread T0 here:
    #0 0xb6ad205f in operator new[](unsigned int) (/usr/lib/libasan.so.3+0xc005f)
    #1 0x7f5c160b in sigm::SignalingManager::Start() /home/ebengt11/Documents/new_tcam/telematics/framework/SignalingManager/signaling_manager.cpp:343
    #2 0x7f5aab37 in main /home/ebengt11/Documents/new_tcam/telematics/framework/SignalingManager/signaling_manager.cpp:477
    #3 0xb5962cbf in __libc_start_main (/lib/libc.so.6+0x16cbf)
SUMMARY: AddressSanitizer: heap-buffer-overflow /home/ebengt11/Documents/new_tcam/telematics/framework/SignalingManager/signaling_manager.cpp:344 in sigm::SignalingManager::Start()
Shadow bytes around the buggy address:
  0x36622730: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x36622740: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x36622750: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x36622760: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x36622770: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x36622780: fa fa fa fa fa fa fa fa fa fa fa fa fa[fa]00 00
  0x36622790: 00 00 00 00 00 00 00 00 fa fa fa fa fd fd fd fd
  0x366227a0: fd fd fd fd fd fa fa fa fa fa fd fd fd fd fd fd
  0x366227b0: fd fd fd fd fa fa fa fa fd fd fd fd fd fd fd fd
  0x366227c0: fd fd fa fa fa fa fa fd fd fd fd fd fd fd fd fa
  0x366227d0: fa fa fa fa fd fd fd fd fd fd fd fd fd fd fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Heap right redzone:      fb
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack partial redzone:   f4
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==7122==ABORTING
```

Figure 5.5: Stack trace printed by AddressSanitizer when it found an introduced heap-based buffer underflow error.

5.3.2 Debug_new

All other services could run as usual on the TCAM while we analysed FTM with Debug_new. We measured the peak RSS to be 8620 kB, which means Debug_new introduced an overhead of about 4%.

Out of the errors we introduced in the FTM program, Debug_new was only able to find the simple memory leak. It pointed to the memory leak correctly, but we had to use addr2line manually to get a human-readable output.

When we analysed SignalingManager using Debug_new, we could run all services as usual on the TCAM. We also got the function tests to pass while analysing the program. After the tests, we measured the peak RSS to be 10172, which corresponds to an overhead of about 4% compared to the peak RSS of SignalingManager when it was not being analysed.

Debug_new was able to find one of the errors we introduced in SignalingManager: the memory leak. It pointed at the correct place in the source code, but we had to use addr2line manually to find it. In addition to the memory leak we introduced, we found two potentially "real" leaks: one in SignalingManager and one in a shared library called *libccm.so*. We got confirmation by a developer at Volvo that the potential leak in SignalingManager was real and that they would fix it. Furthermore, we were also notified that the potential leak in *libccm.so* was most likely authentic, but that another developer was already working on fixing the leaks in that library. Based on our observations of the source code, we also think it looks like a real memory leak.

Both when analysing FTM and when analysing SignalingManager, Debug_new reported many leaks in strange places. For example in the C++ standard library. We think these leaks might be false positives since there were so many of them, and to us it seems likely that the memory was just not deallocated on shutdown. Another reason for why we think some of the leaks are false positives is because their pointers were handled by detached threads, and

it seems possible to us that the tools could struggle with analysing the memory of detached threads.

5.3.3 Heapusage

We were able to analyse FTM with Heapusage without having to close any other services. We measured the peak RSS to 9160 kB, which means Heapusage introduced an overhead of about 11%.

Heapusage was able to find and point to one of the errors that we introudced in FTM: the simple memory leak. Even when accounting for the memory leaked by the simple leak, Heapusage reported a larger amount of memory leaked than it did for the version of FTM without introduced leaks. This additional leaked memory probably comes from the other introduced memory leak, but Heaptrack was not able to produce any useful feedback that let us pinpoint the leak, so we can not be certain.

Just as when we analysed FTM, we could analyse SignalingManager while all the other services were running on the TCAM, without encountering any issues. We measured a peak RSS of 10372 kB, which corresponds to an overhead of about 6%.

Using Heapusage, we were able to find one of the three errors we introduced in SignalingManager: the memory leak. The feedback we got was somewhat useful, it contained the correct function where the memory leak was allocated but did not point at the correct row. We were able to pinpoint the memory leak in the source code using the feedback. We found two other leaks in addition to the memory leak we introduced: the same ones we found with Debug_new. Using the feedback we got from Heapusage, we were able to pinpoint the leaks to their locations in SignalingManager and *libccm.so* respectively.

We also saw that Heapusage reported many leaks in strange places, for both FTM and SignalingManager, just like Debug_new did.

5.3.4 Heaptrack, Dr. Memory, and Gperftools

We were not able to analyse FTM and SignalingManager using Heaptrack, Dr. Memory or Gperftools. Both programs generated segmentation faults and aborted when being analysed by the tools. We lack sufficient evidence to make concrete conclusions, but we hypothesize that it could be due to the programs using too much memory.

Chapter 6
Discussion

"The bitterness of poor quality remains long after the sweetness of meeting the schedule has been forgotten"
— Anonymous [139]

In this chapter, we discuss our findings and comment on our research from the previous chapters. In Sections 6.1 and 6.2, we discuss the results we got from evaluating the tools on the test suites and benchmarks. In Section 6.3, we discuss the differences in maintenance and development of the tools. Then, in Section 6.4, we write about what we observed when trying to analyse Volvo's programs, and in Section 6.5 we discuss how the tools could be used by Volvo. In Section 6.6, we write about some potential pitfalls of our work.

6.1 Memory Leaks and Memory Faults

By looking at the different types of errors found by each analysis tool, it is clear that most of the tools were built to either find leaks or to find memory faults. However, some of the most advanced tools, such as Valgrind and Dr. Memory, were comprehensive enough to detect many variations of both kinds of errors.

We managed to get Dr. Memory to run successfully on our smaller benchmark programs, but when trying to use it on larger programs it did not work. We even got Valgrind to run on the TCAM after closing some running services. Unfortunately, the feedback we received from it was poor. In many cases we only got the message:
"`Program received signal SIGSEGV, Segmentation fault.`" followed by the program aborting. We are not certain why this error message appeared, but we suspect it might have something to do with high memory usage. Furthermore, when Valgrind managed to

analyse a program without causing a crash, the feedback was often not pointing to the correct file and line where the problem occurred. The reason for why Valgrind pointed incorrectly is not completely clear to us. It could be due to the lack of available memory, or possibly due to Valgrind not being completely stable on the TCAM. The conclusion of our Valgrind test was that even though it is possible to make it run, after numerous steps, the feedback it actually gave was not very useful. Even if it would give better feedback, we could not get it to run on larger programs. Our results show that both Valgrind and Dr. Memory imposed a large overhead on both execution time and memory consumption. This makes them practically unusable on an embedded system like the TCAM, unless the analysed program is very small.

We were able to run the less advanced tools, which were simply preloaded as shared libraries, on the TCAM. We also managed to find memory errors with them. Most of these tools could only find memory leaks, but some of them also found other errors, such as missmatched memory management routines. Only one of them, Duma, could find memory access errors, such as buffer overwrite/overread and buffer underwrite/underread, in C++. It is possible simpler tools did not find memory access errors due to the complexity of the task. The most common way to find memory access errors is to monitor every load and store instruction. It can be done by running the code in a virtual machine, like Valgrind and Dr. Memory do, or by instrumenting the code during compilation, like AddressSanitizer does. If a "simpler" tool did this, we would not consider it simple any more.

However, there also exist approaches to finding memory access errors which do not require instrumenting the source code. A tool could insert red-zones around allocations filled with "magic values" that are unlikely to be used by the program. Then, if the program writes to a red-zone, the magic values will be changed, which can be detected by the tool. The drawback with this approach is that the access error is not found immediately when it occurs, but instead when the tool checks the red-zone. This makes it difficult for the tool to give accurate feedback, and it would not notice if the memory had been read but not written to.

Another approach, which is used by Duma, is to use the system call `mprotect`. As mentioned, Duma was the only tool which was able to find memory access errors at the moment they occurred, while not instrumenting the source code. By using `mprotect`, it is possible to mark certain parts of the memory as inaccessible. This makes reads or writes to that memory page cause a segmentation fault, which aborts the program. We think this is why Duma had to be run in a debugger to create any useful feedback. Even if it found the errors when they occurred, the program aborted before the feedback could be reported.

Memwatch and Rmdebug used the red-zone approach and were able to find many buffer overwrites, but they required the inclusion of a header file in each source file of the analysed program. Furthermore, their support for memory access errors was still limited, since they found only writes and not reads. Additionally, they only worked for programs written in C.

While testing the tools on Volvo's programs we received advice from a developer, which made it easier to preload libraries with programs that are run as services. The service files are stored in read-only memory in the TCAM, but they need to be modified to preload a library. In the beginning, we modified a service file by re-flashing the TCAM with new software. However, by using the Linux command "`mount -b`" we could switch out a service file for a new one without needing to re-flash the TCAM. This method could make it easier for developers at Volvo to analyse their program with a preloaded library.

6.2 Overhead

When we started thinking about how to measure the memory overhead, we realised there is no clear "best" way to do it. Initially, we wanted to measure dynamic allocations using Valgrind's Massif skin, but we quickly noticed that it did not work together with the other tools. Massif would have given us a measurement of how the heap usage varied during program execution, but it would have missed the memory occupied by the shared libraries, as well as any increase in code size. Since we could not use Massif, we measured the RSS, PSS, and USS of the analysed program.

Peak RSS was easier to measure than USS or PSS, since we only had to read the `/proc/[PID]/status` file at the end of the program, instead of sampling the `/proc/[PID]/smaps` file during program execution. However, only measuring the peak RSS comes with a disadvantage: it might be misrepresenting the overhead when calculating percentages. A large part of the RSS could be shared memory, and a small increase in unique memory might be insignificant when looking at the total memory because of this.

USS might be more useful for representing the extra overhead, since it only measures the unique memory used by the process. Unfortunately, there is no easy way to get the peak of this value and that is why we resorted to sampling, which might cause us to miss the peak.

PSS represents the USS plus the shared memory divided by processes using it. Because of this, it might be considered a more fair representation of memory per process than RSS. However, it has the same drawback: the shared memory is not relevant to measure since no overhead is imposed on it.

When we tested the tools we chose in the revised tool selection on Volvo's programs, we only measured the peak RSS to see how much of an overhead was introduced by each tool. Initially, we wanted to sample the USS and PSS as well, as we had done for the benchmarks. However, we could not get our sampling library to work in conjunction with some of the tools when they were analysing Volvo's programs. It would have been valuable to see the memory consumption in USS and PSS too. The USS is especially relevant, since it indicates how much extra memory is needed from the system because of the tool. The RSS overhead does not necessarily show this as accurately, since it includes shared memory as well.

For some of the tools, when they analysed certain benchmarks, we measured a peak USS that was lower than the peak USS of the program when it was not analysed. This resulted in a negative overhead in USS. In the same run of the benchmark in which we measured the peak USS, the peak RSS could be larger than what it was when the benchmark was not being analysed. This leads us to believe that the USS overhead was not actually negative, but that the measurement was inaccurate. We can think of a few ways in which this could happen. One possibility is that more programs ran on the TCAM while we were analysing the benchmark. If they used some shared libraries that the benchmark also used, they could possibly reduce the USS of the benchmark. However, we do not think this is very likely, since we took measures to ensure that the same programs were running in the background when we evaluated each tool. Another perhaps more likely possibility is that the benchmark had a spike in USS between the samples that our memory measurement library took.

As can be seen in the results, the tools which imposed the least overhead were most often the tools that had the least complex implementations. Many of these tools only found memory leaks, and they did so by simply wrapping the memory allocation functions and looking for differences in allocated and deallocated memory. However, even if they employ a simple

approach, these tools were able to run on the system without much, if any, impact on its functionality. They also proved to have some value, as they managed to find some leaks in the programs they analysed.

6.3 Development and Maintenance of Tools

We have found that the level of active development and maintenance of the tools we evaluated varies greatly. The more advanced tools such as Valgrind, AddressSanitizer, and Dr. Memory seem to be actively developed by many contributors.

In contrast, many of the less complex tools were not as extensively maintained. Most of these tools had often been developed by a single developer as an open-source project. For example, Debug_new, which proved to be useful for finding leaks, has been developed by a single person, and therefore the implementation details might suffer. We noticed that tools written by a single developer were more likely to be left without maintenance after the initial development phase. Debug_new has been updated sporadically over the last few years, but no real feature development has happened for several years. The effects of only being maintained by one developer, and not being updated, are visible in the functionality. For instance, the tool requires the inclusion of a header, or it needs to use *addr2line*, in order to provide the correct line number and file name when giving feedback.

Tools that are more actively maintained, and have a large contributing community, have implemented this feature in a better way. There are libraries that can be used to read debug information during run time. An example is *libbacktrace* [24], which is used by AddressSanitizer.

6.4 Testing on Volvo's Programs

During testing, we were pleased to find a few real memory leaks that could be fixed. Both Heapusage and Debug_new successfully found the memory leak we had introduced, and they also found other leaks that were later determined to be real. We still had to manually translate addresses when we used Debug_new to get filename and line number, and we only got the function name with Heapusage. However, we think this demonstrates that even the simpler tools can be useful for finding real errors.

When AddressSanitizer was used, it found the two memory access errors we had introduced and gave comprehensive feedback in the form of file names and line numbers. Unfortunately, we could not get any tool which could find both memory leaks and memory access errors, like Dr. Memory, to work. After measuring the overhead, we suspected it might not work, but we nonetheless wanted to try.

We found it quite surprising that we could not get the other two tools, Heaptrack and Gperftools, to work. Their measured memory overheads were not very large, but it is possible that they were large on the specific Volvo programs we analysed. However, it is also possible it was not the memory overhead that made them crash but instead the execution time overhead. Both Heaptrack and Gperftools introduced a substantial execution time overhead which could have led to timing issues in Volvo's programs. Additionally, there is also the possibility that something other than the overheads caused the crashes. For example, Gpertools

uses its own malloc implementation which might be a cause of failure if Volvo's programs depend on the standard implementation.

6.5 Tool Usage in the Workflow

In articles that we read during the literature review, static and dynamic tools were shown to have different strengths and weaknesses. A combination of them seems to be the most efficient for finding the largest number of different errors. Therefore, it could be a good idea to make use of both static and dynamic analysis tools in a software project. Static tools present the smallest hurdle of usability for developers because, in theory, they can analyse the source code directly. We did not evaluate any static tools, but Volvo already uses two common static tools: Clang-tidy and CppCheck.

Dynamic analysis tools can be useful to include in the workflow as well. Some of the tools we evaluated, such as Dr. Memory and Valgrind, are very powerful, but can not be used efficiently on the TCAM because of the resource constraints. Perhaps these tools could be used during the execution of unit-tests instead. This might be possible if unit testing is done on a computer with more resources.

The other, less intrusive, tools that we evaluated, could be used on the TCAM to a larger extent. Perhaps these could be used for testing on the TCAM, although we do not think it would be possible to analyse multiple programs at the same time without causing the memory to run out, at least not with AddressSanitizer. AddressSanitizer could potentially find errors not caught in the previous unit testing stage if these errors are caused by code that was not executed in unit tests or if they are caused by communication between different modules. AddressSanitizer is also the only tool we found useful for finding memory access errors, while still being able to run on the TCAM.

Both Heapusage and Debug_new are implemented as dynamic libraries wrapping the allocation and deallocation functions and operators. Tools using this type of implementation had the least overhead and would function in an environment with memory constraints. They also proved to be effective at finding real leaks. We think that these tools, or another tool using the same preload technique, would prove useful for finding memory leaks in software running on the TCAM.

Unfortunately, no tool which was able to run efficiently on the TCAM could catch all types of errors. Therefore, we conclude that it is advisable to use a combination of 2-3 different tools. However, we recommend using one tool at a time. Otherwise, their interaction could lead to unexpected behaviour and high resource usage.

6.6 Possible Pitfalls

During the entire project, we have strived to be thorough so we could feel confident in our decisions. However, given the limited time and scope of our research, it is possible we did not consider all relevant information. In this section we will discuss the weaknesses of our analysis.

For example, we may not have found every freely available dynamic analysis tool. It is likely we found the most commonly used ones, but there is a chance that there are tools

which we did not evaluate that are better suited for Volvo. Another pitfall is that some of the tools which crashed could have possibly been made useful if we had dedicated more time to finding the root cause of the crash. This may have caused us to write off potentially viable tools due to time constraints.

Before we used the tools, we always looked through the documentation to see how they were supposed to be used. Some tools were simple to use and only had a few configurations, but other tools were more complicated, as they offered many configuration options. It is possible that we could have made the more configurable tools perform better if we had more experience with them.

Since we only considered open source tools, we might have missed some potentially useful tools. We did not find any tool which was both free to use and not open source, but we did find some tools which could be purchased. It is possible that some of these tools would have worked well for us, but given the time frame of the project we decided it was not feasible to invest time and resources on obtaining access and analysing them. One tool which we read a lot about, *Purify*, seemed especially difficult to get access to. Nonetheless, Purify would have been interesting to evaluate since it seems to use an approach similar to AddressSanitizer.

When choosing which tools to test on the TCAM, we chose AddressSanitizer over Duma. Duma could possibly have been a good choice, since it was able to find almost as many errors as AddressSanitizer. The main reason we did not choose Duma was because it did not give useful feedback unless run in a debugger. Still, it remains possible that we would not have had to close any services if using Duma, like we had to do when using AddressSanitizer. Additionally, we also chose not to use Duma because it caused 5 of our benchmarks to crash. This seemed to be caused by the tool itself, since no other tool reported any error in these benchmarks. Furthermore, the source code of the tool has not been updated since 2009, which also contributed to us choosing to use AddressSanitizer.

Another weakness in our research is our method of testing if tools could find double-free errors. We were not able to run any test program with a double-free without it crashing. We think this could be due to the standard implementation of malloc & friends catching these errors and aborting the program. Perhaps another implementation of the allocation functions would let the error through. If a double free is not detected by the allocation functions, we are not confident which tools would catch it.

Chapter 7
Conclusions

"No one in the brief history of computing has ever written a piece of perfect software. It's unlikely that you'll be the first. And unless you accept this as a fact, you'll end up wasting time and energy chasing an impossible dream."

— Andrew Hunt *[133]*

Working on the thesis has been both interesting and educational. The main conclusion we draw from our work is that there is not one perfect tool which catches all errors and can be used in all scenarios. Instead, we think a combination of tools, used at different stages of testing and development, could be accurate and efficient for detecting memory related errors in an embedded system. The usefulness of testing at multiple stages of development has become apparent to us. Using the right tools and processes is helpful for catching many software bugs, and it is something most developers could benefit from doing. In Section 7.1 of this chapter, we answer our research questions, and in Section 7.2 we give some suggestions of topics which could be interesting to explore more in depth.

7.1 Answers to Research Questions

RQ 1. ***What are suitable tools and methods for finding memory errors in a system running embedded Linux?***

Through evaluating different tools in steps, we found a few suitable options: *Debug_new*, *Heapusage* and *AdressSanitizer*. They provide a combination of acceptable overhead and good error finding capabilities. We also found some general methods which seem suitable for use in an environment with limited resources. For example, preloading a shared library that wraps the memory allocation functions seems to work well. Additionally, instrumenting the source

code during compile time, as demonstrated by AddressSanitizer, appears to be more efficient than instrumenting it during run-time, as demonstrated by Valgrind and Dr. Memory.

(a). ***What is the overhead introduced by the tools, in memory and execution time, and does it affect their usability on the system?***

The overhead introduced by the tools varied a lot. Some tools introduced almost no overhead while others made the programs they analysed run several times slower and consume much more memory. We noticed that there seems to be a trade-off between functionality and usability. Less sophisticated tools often introduced small overheads, while more sophisticated tools often introduced large overheads. Tools which were effective in regards to the errors they found, such as Dr. Memory and Valgrind, became unusable in a working scenario in the system due to the large overheads they introduced. Their large overheads in execution time could cause timing issues, and their large memory overheads could cause the system to run out of memory.

When we did our evaluations, we noticed that tools which did not introduce large overheads, especially in memory, were more likely to be able to analyse a program on the TCAM. When we tried to analyse the two Volvo-programs, we discovered that AddressSanitizer was on the edge of consuming too much memory. It was the only tool we found that was really useful for finding memory access errors, but its overhead affected its usefulness since we had to close some services before it could be used to analyse Volvo's programs.

(b). ***Which memory faults are relevant for the tools to detect and how do the evaluated tools perform in this regard?***

We consider all the errors we evaluated relevant to be detected. We deem the CWE top 25 list to be helpful for getting an idea of which errors are the most dangerous.

Memory leaks can be very dangerous since they do not cause the system to malfunction until it starts to run out of memory. This could happen after a long time, when you least expect it. Memory access errors, such as buffer overreads and underreads, are also dangerous. They can cause the program to crash, or, perhaps even worse, change values that affect program behaviour, causing it to act unpredictably.

We consider errors which cause the system to enter a faulty state but continue to run very dangerous. Those errors are "silent" and can make the system unpredictable, making them important to detect. The system entering a faulty state could be caused by both memory leaks and memory access errors.

(c). ***How useful is the tool-generated feedback for pinpointing a possible error?***

The most useful feedback we got from the tools was a direct reference to the source file and line number of the error's occurrence. We also found stack traces to be very useful for finding and fixing errors, since they give insight into what had happened before the error occurred.

The tools did not give equal feedback. Some tools were very informative, while others only pointed to the function where the error occurred, without printing any stack trace. Some tools did not print human readable feedback. Instead, they gave memory addresses which had to be manually translated to line numbers with addr2line.

We found that even the tools which did not give very good feedback could be useful to some extent. While they were more difficult to use, we managed to find memory errors with them as well. We think that for a tool to be really useful it should print a stack trace with file names, function names, and line numbers in cleartext.

RQ 2. ***Could there be advantages to using multiple tools in the development workflow?***

Our testing and results suggest that there is no usable "*silver-bullet*" tool which can catch all errors on the embedded system. Therefore, in order to find different types of errors, multiple tools should be used together in the workflow. In our experience, smaller tools that look for fewer errors seem to have less of an overhead. Their ability to be run separately reduces the peak overhead that a program has to be able to handle without crashing. For example, a program might crash when analysed with a monolithic tool like Dr. Memory, but work without problems when analysed with Heapusage and AddressSanitizer in different program runs. In our experience, multiple tools used in conjunction can find just as many errors in a program compared to a single, more universal, tool, while also having the ability to analyse larger programs without causing the system to crash.

Additionally, we think it could be advantageous to run some of the more heavy duty tools during unit testing, since the unit tests, most likely, are performed on a system with more resources. The less intrusive tools can be run during integration testing, which takes place on the actual embedded system. That way, tools such as Valgrind and Dr. Memory could still be used, just not on the embedded system itself.

7.2 Future Work

We believe there are multiple paths which can be explored in future projects aiming to find and eliminate memory errors in embedded systems

We evaluated many tools that wrapped the memory allocation functions and reported memory that had not been deallocated at program exit. These tools seem pretty straightforward to make, but many of the ones we found suffered from being old and not maintained. One common issue we had with them was that they did not give very good feedback when they found an error. They also reported many false positives if the program that was being analysed did not free all memory at program exit. One possibility is to create a new wrapper library which solves many of the inconveniences in the tools we evaluated. Such a tool could be made to print better feedback and to free memory that was allocated by the standard library.

Another possibility for future work is to create a custom memory analysis tool, similar to Purify. For example, to create a tool that finds memory leaks, you could wrap the C and C++ allocation functions, save information about each allocation in a header, and then check for live allocations recursively. As we mentioned in Section 2.5.2, there is an article about Purify which describes this procedure [121]. The created tool could be compiled as a shared library, and then preloaded when running the program you want to analyse. We think that creating a tool like that could be worthwhile for Volvo to consider. Such a tool would not be able to find memory access errors, and it might not find every single memory leak, but it could probably give better feedback than existing tools and be able to analyse a program while it is running. The tool would be more complex than the tools that only report memory leaks at program exit, but we predict it would be possible for a skilled developer to create it in a few weeks.

Yet another possibility for future work could be to adapt an existing wrapper library. For example Debug_new, to give more informative feedback. The library could be modified to use *libbacktrace*, which is used by AddressSanitizer, to print useful stack traces. If Debug_new was

modified to use libbacktrace, it would be able to print file names and line numbers without having to include header files in a programs source code. That would make it easier to use, but it would also increase the memory overhead because of the added library.

Another possibility could be to work on porting some unsupported tools to the ARM architecture. There is a tool that is similar to AddressSanitizer, called "LeakSanitizer", which is able to find memory leaks according to its documentation. Judging by the capabilities of AddressSanitizer, LeakSanitizer could be useful. Furthermore, AddressSanitizer seems well maintained and widely used. It is open-source, implemented in GCC and Clang, and maintained by Google. If work was done to do the initial porting of LeakSanitizer for Volvo's own purposes, the functionality could possibly continue to be maintained by the open source community. The benefit of this approach is that there is no need to develop a new tool from scratch and maintenance of the initial port might be supported by the "Sanitizer" community.

A larger future project could be to make a tool which does system-wide analysis. A tool like that is described in the related work, but it was not publicly available. It could provide a good next layer in testing where the whole system is monitored, and it might even be able to run in an actual test car with all the hardware present. However, the development of the tool would probably be complex and incur a large cost. Such a tool would fit well into a development workflow of testing in different stages. The tool could be useful for Volvo and possibly to other organisations as well. Since the tool probably would not contain any Volvo specific intellectual property, there is the possibility of first creating it for their specific purpose and then making it open source to enable the open source community to support and maintain it. The argument for that approach is that the tool would be very helpful for development at Volvo. However, since it is not part of the companies core services, their focus should not lie in the continued maintenance and development of a proprietary tool. Making it open-source gives Volvo the benefit of having a maintained tool in the long run, while being able to focus the majority of their own development efforts towards software which generates value for the company. However, there is no guarantee that such a tool would attract an open source community or that it would be general enough in its implementation to be interesting to other parties than Volvo. These are some considerations which would also need to be taken into account before work begins on developing such a tool.

A future project which does not involve any tool creation could be to investigate the memory management routines. If the standard memory allocators (malloc etc.) are used, then it could be possible to tweak their behaviour by setting different parameters. This could for example set a threshold for how much extra memory the allocator will keep around for future calls. There is also the possibility of either writing a custom allocator or using an open source one. There are different allocators which claim to be more efficient in both speed and memory usage than standard implementations. An investigation by Volvo into whether this could help reduce the memory usage of their programs could be worth the effort.

Other non-tool related projects could involve further decoupling the software from the hardware. In Volvo's case, perhaps it could be possible to adapt their development to follow the AUTOSAR standard in all ECU's. This standard aims to make the development ECU software less dependent on specific hardware by using a layered architecture. Another possibility is to work on developing virtual models of hardware to enable testing on more powerful machines. In the literature, frameworks such as SystemC and Simulink are often mentioned when it comes to simulation of hardware.

Yet another possible path to explore for a more long term solution would be to change the development language to something other than C/C++. For example, changing to the Rust programming language, which uses a different memory model, might solve many memory related problems.

References

[1] Addresssanitizer source. https://github.com/google/sanitizers/wiki/AddressSanitizer. (Visited on 23/04/2020).

[2] Android runtime (art) and dalvik. https://source.android.com/devices/tech/dalvik.
(Visited on 18/03/2020).

[3] The chromium projects - memory safety. https://www.chromium.org/Home/chromium-security/memory-safety. (Visited on 25/05/2020).

[4] Common weakness enumeration. https://cwe.mitre.org/.
(Visited on 19/03/2020).

[5] Debug_new source. https://github.com/adah1972/nvwa.
(Visited on 23/04/2020).

[6] Dr. memory source. https://github.com/DynamoRIO/drmemory/wiki/Downloads. (Visited on 23/04/2020).

[7] Duma source. http://duma.sourceforge.net/. (Visited on 23/04/2020).

[8] The dwarf debugging standard. http://dwarfstd.org/. (Visited on 19/03/2020).

[9] Elfutils. https://sourceware.org/elfutils/. (Visited on 19/03/2020).

[10] Find. https://www.gnu.org/software/findutils/. (Visited on 15/05/2020).

[11] Fixed toyota itc benchmarks. https://github.com/AbsInt/itc-benchmarks.
(Visited on 19/03/2020).

[12] Gcc debugging options. https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html. (Visited on 26/05/2020).

[13] Github. https://github.com/. (Visited on 19/03/2020).

[14] Gnu binutils. https://www.gnu.org/software/binutils/. (Visited on 20/05/2020).

[15] The gnu c++ library manual, debugging support. https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug.html. (Visited on 25/05/2020).

[16] Google scholar. https://scholar.google.com/. (Visited on 19/03/2020).

[17] Gperftools source. https://gperftools.github.io/gperftools/heap_checker.html. (Visited on 24/05/2020).

[18] Grep. https://www.gnu.org/software/grep/. (Visited on 15/05/2020).

[19] Gzip. https://www.gnu.org/software/gzip/. (Visited on 15/05/2020).

[20] Heaptrack source. https://github.com/KDE/heaptrack. (Visited on 23/04/2020).

[21] Heapusage source. https://github.com/d99kris/heapusage. (Visited on 23/04/2020).

[22] Juliet test suite for c/c++. https://samate.nist.gov/SARD/around.php#juliet_documents. (Visited on 19/03/2020).

[23] Leaktracer source. https://github.com/fredericgermain/LeakTracer. (Visited on 23/04/2020).

[24] Libbacktrace repository. https://github.com/ianlancetaylor/libbacktrace. (Visited on 17/05/2020).

[25] Libleak source. https://github.com/WuBingzheng/libleak. (Visited on 23/04/2020).

[26] The libunwind project. https://www.nongnu.org/libunwind/. (Visited on 19/03/2020).

[27] Llvm benchmarks. https://github.com/llvm/llvm-test-suite/tree/master/MultiSource/Benchmarks. (Visited on 15/05/2020).

[28] Malloc_count source. https://github.com/bingmann/malloc_count. (Visited on 23/04/2020).

[29] Memwatch source. https://www.linkdata.se/sourcecode/memwatch/. (Visited on 23/04/2020).

[30] Mibench. http://vhosts.eecs.umich.edu/mibench//. (Visited on 22/05/2020).

[31] Mtrace source. http://man7.org/linux/man-pages/man3/mtrace.3.html. (Visited on 23/04/2020).

[32] Nsa center for assured software (cas). https://samate.nist.gov/BF/Enlightenment/CAS.html. (Visited on 19/03/2020).

[33] perf stat manual page. `https://linux.die.net/man/1/perf-stat`. (Visited on 26/05/2020).

[34] Project gutenberg. `https://www.gutenberg.org/`. (Visited on 29/05/2020).

[35] Rmdebug source. `http://www.hexco.de/rmdebug/`. (Visited on 23/04/2020).

[36] Spec cpu benchmarks. `https://www.spec.org/benchmarks.html#cpu`. (Visited on 15/05/2020).

[37] Spec cpu2000. `https://www.spec.org/cpu2000/`. (Visited on 19/03/2020).

[38] Stack overflow. `https://stackoverflow.com/`. (Visited on 19/03/2020).

[39] Toyota itc benchmarks. `https://github.com/regehr/itc-benchmarks`. (Visited on 19/03/2020).

[40] Tramp3d-v4 performance. `https://gcc.opensuse.org/gcc-old/c++bench/tramp3d/`. (Visited on 16/03/2020).

[41] Valgrind. `https://valgrind.org/`. (Visited on 19/03/2020).

[42] Valgrind user manual. `https://www.valgrind.org/docs/manual/valgrind_manual.pdf`. (visited on 13/05/2020).

[43] Wikipedia entry for david may. `https://en.wikipedia.org/wiki/David_May_(computer_scientist)`. (Visited on 19/03/2020).

[44] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[45] David Atienza Alonso, Stylianos Mamagkakis, Christophe Poucet, Miguel Peón-Quirós, Alexandros Bartzas, Francky Catthoor, and Dimitrios Soudris. *Dynamic memory management for embedded systems*. Springer, 2015.

[46] Doug Amos. *FPGA-based prototyping methodology manual : best practices in design-for-prototyping*. Synopsys, Inc, Mountain View, CA, 2011.

[47] A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, and D. Lucanu. A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 161–168, 2017.

[48] AUTOSAR GbR. *Guidelines for the use of the C++14 language in critical and safety-related systems*. AUTOSAR, 2017.

[49] Michael Barr and Anthony Massa. *Programming embedded systems: with C and GNU development tools*. " O'Reilly Media, Inc.", 2006.

[50] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–12, 2002.

[51] B. Beyer, C. Jones, J. Petoff, and N.R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[52] Robert H. Bishop. *The mechatronics handbook*. CRC Press, Boca Raton, Fla., 2002.

[53] H. Bo, D. Hui, W. Dafang, and Z. Guifan. Basic concepts on autosar development. In *2010 International Conference on Intelligent Computation Technology and Automation*, volume 1, pages 871–873, 2010.

[54] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[55] Michael D Bond and Kathryn S McKinley. Bell: Bit-encoding online memory leak detection. *ACM SIGARCH Computer Architecture News*, 34(5):61–72, 2006.

[56] Michael David Bond. Diagnosing and tolerating bugs in deployed systems. 2008.

[57] Manfred Broy, Ingolf H Kruger, Alexander Pretschner, and Christian Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[58] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 213–223. IEEE, 2011.

[59] Herman Bruyninckx. V-model. `https://commons.wikimedia.org/wiki/File:V-model.svg`. (Visited on 23/04/2020).

[60] Ondrej Burkacky, Johannes Deichmann, Georg Doll, and Christian Knochenhauer. Rethinking car software and electronics architecture. *McKinsey & Company*, 2018.

[61] William R Bush, Jonathan D Pincus, and David J Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[62] Stuart Byma and James R Larus. Detailed heap profiling. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, pages 1–13, 2018.

[63] Microsoft Security Response Center. A proactive approach to more secure code. `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/`. (Visited on 25/05/2020).

[64] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. Detecting memory errors at runtime with source-level instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 341–351, 2019.

[65] Patrick Cousot, Radhia Cousot, Jérôme Feret, MINE Antoine, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers: A comparison with astrée. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE'07)*, pages 3–20. IEEE, 2007.

[66] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *NASA Formal Methods Symposium*, pages 120–125. Springer, 2012.

[67] Michael A. Cusumano and Richard W. Selby. *Microsoft secrets : how the world's most powerful software company creates technology, shapes markets, and manages people*. Simon & Schuster, New York, 1998.

[68] Joseph G D'Ambrosio and Rami Debouk. Asil decomposition: the good, the bad, and the ugly. Technical report, SAE Technical Paper, 2013.

[69] Tom De Schutter. *Better Software. Faster!: Best Practices in Virtual Prototyping*. Happy About, 2014.

[70] Deloitte. *Semiconductors – the Next Wave* opportunities and winning strategies for semiconductor companies. `https://www2.deloitte.com/cn/en/pages/technology-media-and-telecommunications/articles/semiconductors-the-next-wave-2019.html`, April 2019. (Visited on 02/06/2020).

[71] Jeff Desjardins. How many millions of lines of code does it take?, Feb 2017. (Visited on 19/04/2020).

[72] Edsger W. Dijkstra. *Notes on structured programming*. Eindhoven, 2 edition, 1970.

[73] Rainer Dömer, Andreas Gerstlauer, and Wolfgang Müller. Introduction to hardware-dependent software design. In *2009 Asia and South Pacific Design Automation Conference*, pages 290–292. IEEE, 2009.

[74] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, 2003.

[75] Wolfgang Ecker, Wolfgang Müller, and Rainer Dömer. *Hardware-dependent Software: Principles and Practice*. Springer Netherlands, 2009.

[76] Nelson Elhage. Three kinds of memory leaks. `https://blog.nelhage.com/post/three-kinds-of-leaks/`. (Visited on 02/06/2020).

[77] Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. *ACM SIGPLAN Notices*, 47(1):533–544, 2012.

[78] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[79] Cal Erickson. Memory leak detection in embedded systems. *Linux Journal*, 2002(101):9, 2002.

[80] Cal Erickson. Memory leak detection in c++. *Linux Journal*, 2003(110):8, 2003.

[81] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51, 2002.

[82] A. Fatima, S. Bibi, and R. Hanif. Comparative study on static code analysis tools for c/c++. In *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 465–469, 2018.

[83] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. Fast and vulnerable: A story of telematic failures. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*, 2015.

[84] Frantisek Franek. *Memory as a Programming Concept in C and C++*. Cambridge University Press, 2004.

[85] Simon Fürst. Challenges in the design of automotive software. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 256–258. IEEE, 2010.

[86] David Gelles, HIROKO Tabuchi, and MATTHEW Dolan. Complex car software becomes the weak spot under the hood. *The New York Times online*, 2015.

[87] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications*, pages 581–591. Springer, 2015.

[88] Marc Gregoire. *Professional C++*. John Wiley & Sons, 2014.

[89] Christopher Hallinan. *Embedded Linux primer: a practical real-world approach*. Pearson Education India, 2011.

[90] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.

[91] Jörg Herter, Daniel Kästner, Christoph Mallon, and Reinhard Wilhelm. Benchmarking static code analyzers. *Reliability Engineering & System Safety*, 188:336–346, 2019.

[92] Jörg Herter, Daniel Kästner, Christoph Mallon, and Reinhard Wilhelm. Benchmarking static code analyzers. *Reliability Engineering & System Safety*, 188:336–346, 2019.

[93] Diane Hosfelt. Implications of rewriting a browser component in rust. `https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/`. (Visited on 25/05/2020).

[94] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19, 2005.

[95] ISO Central Secretary. Road vehicles – Functional safety – Part 6: Product development at the software level. Standard ISO 26262-6:2018, International Organization for Standardization, Geneva, CH, 2018.

[96] Mabel Mary Joy, Wolfgang Mueller, and Franz Josef Rammig. Source code annotated memory leak detection for soft real time embedded systems with resource constraints. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 166–172. IEEE, 2014.

[97] Swapnili P Karmore and Anjali R Mahajan. Universal methodology for embedded system testing. In *2013 8th International Conference on Computer Science & Education*, pages 567–572. IEEE, 2013.

[98] Ugur Koc, Parsa Saadatpanah, Jeffrey S Foster, and Adam A Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 35–42, 2017.

[99] John Kodumal and Alex Aiken. Banshee: A scalable constraint-based analysis toolkit. In *International Static Analysis Symposium*, pages 218–234. Springer, 2005.

[100] Jim A Ledin. Hardware-in-the-loop simulation. *Embedded Systems Programming*, 12:42–62, 1999.

[101] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to embedded systems : a cyber-physical systems approach*. MIT Press, Cambridge, Mass, 2. ed. edition, 2017.

[102] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 391–401. IEEE, 2019.

[103] Rainer Leupers and Peter Marwedel. *Retargetable compiler technology for embedded systems: tools and applications*. Springer Science & Business Media, 2001.

[104] Peter Liggesmeyer and Mario Trapp. Trends in embedded software engineering. *IEEE software*, 26(3):19–25, 2009.

[105] Stanley B. Lippman, Josée. Lajoie, and Barbara E. Moo. *C++ primer*. Addison-Wesley, Upper Saddle River, N.J., 5. ed. edition, 2013.

[106] M. Mantere, I. Uusitalo, and J. Roning. Comparison of static code analysis tools. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 15–22, 2009.

[107] Peter. Marwedel. *Embedded system design : embedded systems, foundations of cyber-physical systems, and the internet of things*. Springer, Cham, Switzerland, third edition edition, 2018.

[108] MIRA Ltd. *MISRA-C:2012 Guidelines for the use of the C language in Critical Systems*. 2012.

[109] Misra consortium announce integration of autosar c coding guidelines into updated industry standard. `https://www.autosar.org/news-events/details/misra-consortium-announce-integration-of-autosar-c-coding-guidelines-into-updated-industry-standar/`, Jan 2019. (Visited on 26/05/2020).

[110] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[111] Tukaram Muske and Alexander Serebrenik. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166. IEEE, 2016.

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

[113] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[114] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74, 2007.

[115] Jihyun Park and Byoungju Choi. Automated memory leakage detection in android based systems. *International Journal of Control and Automation*, 5(2):35–42, 2012.

[116] Nathanael Paul, Tadayoshi Kohno, and David C Klonoff. A review of the security of insulin pump infusion systems. *Journal of diabetes science and technology*, 5(6):1557–1562, 2011.

[117] Patrizio Pelliccione, Eric Knauss, Rogardt Heldal, S Magnus Ågren, Piergiuseppe Mallozzi, Anders Alminger, and Daniel Borgentun. Automotive architecture framework: The experience of volvo cars. *Journal of systems architecture*, 77:83–100, 2017.

[118] Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[119] Cesare Pizzi. Memory access error checkers. *Linux Journal*, 1999(61es):26, 1999.

[120] F. Pospiech and S. Olsen. Embedded software in the soc world. how hds helps to face the hw and sw design challenge [hardware dependent software]. In *Proceedings of the IEEE 2003 Custom Integrated Circuits Conference, 2003.*, pages 653–658, 2003.

[121] Bob Joyce Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. Citeseer, 1991.

[122] Grigore Roşu, Wolfram Schulte, and Traian Florin Şerbănuţă. Runtime verification of c memory safety. In *International Workshop on Runtime Verification*, pages 132–151. Springer, 2009.

[123] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[124] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283, 2003.

[125] Christian Salzmann and Thomas Stauner. Automotive software engineering. In *Languages for system specification*, pages 333–347. Springer, 2004.

[126] Ionut-Andrei Sandu and Alexandru Salceanu. Metrics improvement for phase containment effectiveness in automotive software development process. In *2017 10th International Symposium on Advanced Topics in Electrical Engineering (ATEE)*, pages 661–666. IEEE, 2017.

[127] Jooyoung Seo, Byoungju Choi, and Suengwan Yang. A profiling method by pcb hooking and its application for memory fault detection in embedded system operational test. *Information and Software Technology*, 53(1):106–119, 2011.

[128] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[129] Vatsalkumar H Shah and Apurva Shah. An analysis and review on memory management algorithms for real time operating system. *International Journal of Computer Science and Information Security*, 14(5):236, 2016.

[130] S. Shiraishi, V. Mohan, and H. Marimuthu. Test suites for benchmarks of static analysis tools. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 12–15, 2015.

[131] Miroslaw Staron. Automotive software architectures. *Automot. Softw. Archit*, pages 33–39, 2017.

[132] Miroslaw Staron. Requirements engineering for automotive embedded systems. In *Automotive Systems and Software Engineering*, pages 11–28. Springer, 2019.

[133] David Thomas and Andrew Hunt. *The pragmatic programmer : your journey to mastery*. Addison-Wesley, Boston, 20th anniversary edition edition, 2020.

[134] Rudi van Drunen. Small embedded system. *The magazine of USENIX & SAGE*, 35(3):38–47, 2010.

[135] Antonio Vetro, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele A Shaw. Investigating automatic static analysis results to identify quality problems: An inductive study. In *2012 35th Annual IEEE Software Engineering Workshop*, pages 21–31. IEEE, 2012.

[136] Kostyantyn Vorobyov, Padmanabhan Krishnan, and Phil Stocks. A dynamic approach to locating memory leaks. In *IFIP International Conference on Testing Software and Systems*, pages 255–270. Springer, 2013.

[137] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. Shadow state encoding for efficient monitoring of block-level properties. *ACM SIGPLAN Notices*, 52(9):47–58, 2017.

[138] Colin Walls. *Embedded software: the works*. Elsevier, 2012.

[139] Karl Eugene Wiegers. *Creating a software engineering culture*. Pearson Education, 1996.

[140] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static c analyzer. *Information Processing Letters*, 102(2-3):118–123, 2007.

[141] Bin Yu, Cong Tian, Nan Zhang, Zhenhua Duan, and Hongwei Du. A dynamic approach to detecting, eliminating and fixing memory leaks. *Journal of Combinatorial Optimization*, pages 1–18, 2019.

[142] Mark Zaslavskiy, Edward Ryabikov, and Kirill Krinkin. Lightweight linux dynamic libraries profiling technique for embedded systems. In *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, pages 1–5, 2013.

Appendices

Appendix A
Description of Terms

In this appendix, we describe different terms that we use throughout the thesis.

addr2line – A program that can convert addresses in executables into file names and line numbers, using debugging information [14].

smaps – This is a file created by the Linux kernel in the `/proc/[PID]/` folder. It contains information in cleartext about different values related to the memory consumption of the running process. We used it to sample the RSS,PSS and USS values of a running process.

status – This is a file created by the Linux kernel in the `/proc/[PID]/` folder. It contains information in cleartext about different values related to the memory consumption of the running process. We used it to get the peak RSS (VmHWM) of a running process.

RSS – Resident Set Size, or *RSS*, is how much of the main memory a process is using. This memory may be unique to the process or shared with other processes. For example, the process' stack is unique to it, but a shared library might be used by other processes too.

PSS – Proportional Set Size, or *PSS*, is also how much of the main memory a process is using, but only a proportional amount of shared memory is counted. For example, consider a process that has an RSS of 10 kB, of which 2 kB are shared with one other process, and 3 kB are shared with two other processes. The PSS of the process would be $5 + \frac{2}{2} + \frac{3}{3} = 7$ kB.

USS – Unique Set Size, or *USS*, is how much of the main memory is unique to a process. The USS of a process is equal to its RSS, minus the amount of memory shared with other processes.

VmHWM – VmHWM is the peak RSS, or the "high water mark", so far during a program's execution. It can be found in the `/proc/[PID]/status` file in Linux.

Appendix B
Usage of Tools

In this appendix we describe how we compiled and ran the tools we chose in our initial tool selection. If anyone wants to reproduce our results this should be helpful. Each section except the first one describes a tool.

B.1 Volvo Script

Volvo had made two scripts which changed different Linux environment variables such as *sysroot*, *compilers*, *compile flags* etc. The reason for changing the variables was to make it easier to build different programs for the TCAM. The most relevant environment variables changed are being listed below in list B.1. After we ran the script, the environment variables pointed to the folders and binaries for the TCAM-system and would therefore be used during compilation.

- `LD_LIBRARY_PATH`
- `SYSROOT`
- `PKG_CONFIG_SYSROOT_DIR`
- `PKG_CONFIG_PATH`
- `PKG_CONFIG_LIBDIR`
- `C_INCLUDE_PATH`
- `CFLAGS`
- `CXXFLAGS`
- `LDFLAGS`
- `CXX`
- `CC`

List B.1: Environment variables changed by the scripts to point to TCAM specific folders and binaries.

B.2 Debug_new

Debug_new can only analyse C++ programs in which `new` and `delete` are used to allocate and deallocate memory.

For using Debug_new on the TCAM we modified the build process a little. The *README* file suggests compiling the source code of the tool together with the program to analyse.

However, to adding new source-files to the compilation of a program could be problematic depending on the licence used, and we also considered it more tedious, which could cause developers to avoid using the tool. Instead, we compiled the tool as a shared library which we could preloaded when running the analysed program. The compilation command is presented in Listing B.1.

```
arm-cas-linux-gnueabi-g++ -I./nvwa -fno-omit-frame-pointer -fno-inline -g -shared -fPIC \
debug_new.cpp -o libdebug_new.so -ldl
```

Listing B.1: Compiling Debug_new as a shared library

Furthermore, Debug_new uses macros to overload `new` and `delete`. The macros save line numbers and file names pointing to were they were called from, and then call the standard versions of `new` and `delete`. For this to work, the tool requires a header file to be included in each source file of the analysed program. We could not include the header file with the compilation flag `-include debug_new.h`, because the include had to be below the inclusion of standard library headers, or the program would not compile. We thought that having to manually include the header file in different source files would be tedious and make the tool less usable.

Luckily, it is possible to use the tool without including a header file. It then uses addr2line to get line numbers and file names. We compiled addr2line for the TCAM so Debug_new could use it, but we could not get it to work. We modified Debug_new to print the file name and base address of the library/executable containing the error, using the `dladdr` function, which translates an address to symbolic information. We could then use addr2line manually to get a line number.

We also noticed a lot of false positives in the feedback, pointing to the standard library functions. These functions do not automatically deallocate their resources before program exit, which meant they got reported as leaks. To alleviate that issue we called the function `__libc_freeres(void)` [15], in a destructor in Debug_new, to free those resources manually.

With the mentioned modifications we managed to get feedback on potential leaks where the names and line-numbers were written out in cleartext.

To run the tool, we used the commands in Listing B.2. That generated a text file with the feedback from the tool and the addresses corresponding to the where the tool found the errors. We could then call addr2line to get the file names and line numbers.

```
LD_PRELOAD=libdebug_new.so <program to be analysed>
addr2line -f -e <program that was analysed> <addresses from textfile>
```

Listing B.2: Executing a program with Debug_new

B.3 Memwatch

Memwatch intercepts calls to most memory allocation and deallocation functions in C, but only has experimental support for C++. We found that the intercepted calls to the `strdup(3)` function did not work properly, so we removed that functionality. We compiled Memwatch as a shared library, which we could add to the library path of the analysed program. We compiled Memwatch with the command in Listing B.3.

```
arm-cas-linux-gnueabi-gcc -DMEMWATCH -DMW_STDIO memwatch.c -g -fno-omit-frame-pointer \
-fpic -shared -o libmemwatch.so
```

Listing B.3: Compiling Memwatch as a shared library

For Memwatch to work, we had to include *memwatch.h* in all source files we wanted to analyse. We used the GCC compiler flag `-include` to include memwatch.h in all source files. The variables `-DMEMWATCH` and `-DMW_STDIO` also had to be defined at compile time for Memwatch to work.

When evaluating the tools, we quickly noticed that many of our C++ benchmarks would not compile when we included memwatch.h using the `-include` compiler flag. For it to work we had to include memwatch.h manually.

We ran the command in listing B.4 to analyse a program. Memwatch printed the results to a file named *"memwatch.log"*.

```
LD_LIBRARY_PATH=<directory containing libmemwatch.so> <program to be analysed>
```

Listing B.4: Executing a program with Memwatch analysis

B.4 Heaptrack

In this section we describe modifications we did in order to get Heaptrack to run on the TCAM. Heaptrack was built using *CMake*, and depended on two third-party libraries: *elfutils* [9] and *libunwind* [26]. We compiled these for the TCAM too, by running configure, as shown in Listings B.5 and B.6, and then *make*.

To build Heaptrack we ran CMake with the *cmake toolchain* file shown in Listing B.7. The toolchain file was invoked with the command line parameter `-DCMAKE_TOOLCHAIN_FILE=<path>/<to>/<file>`.

Heaptrack supplied a bash script for convenient usage. Unfortunately, bash is not supported on the TCAM, so we modified the script to use *sh* (Shell Command Language) instead. The script was not necessary, it preloaded a library file with the `LD_PRELOAD` method, which we could do ourselves too.

To analyse programs with Heaptrack, we ran the command in Listing B.8. That created a compressed file which we then needed to analyse on a host machine with the full Heaptrack program.

```
./configure --host=armv7-linux \
CC=/<path>/<to>/arm-cas-linux-gnueabi-gcc \
CXX=/<path>/<to>/arm-cas-linux-gnueabi-g++ \
LD=/<path>/<to>/arm-cas-linux-gnueabi-ld \
CFLAGS=-fno-stack-protector
```

Listing B.5: Configure script for elfutils

```
./configure --host=armv7-linux \
CC=/<path>/<to>/arm-cas-linux-gnueabi-gcc \
CXX=/<path>/<to>/arm-cas-linux-gnueabi-g++ \
LD=/<path>/<to>/arm-cas-linux-gnueabi-ld \
LDFLAGS=-L/<path>/<to>/<libunwind>/src/.libs \
CFLAGS=-fno-stack-protector
```

Listing B.6: Configure script for libunwind

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)
set(CMAKE_SYSROOT /<path>/<to>/<target sysroot>)
set(CMAKE_C_COMPILER /<path>/<to>/arm-cas-linux-gnueabi-gcc)
set(CMAKE_CXX_COMPILER /<path>/<to>/arm-cas-linux-gnueabi-g++)
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
set(CMAKE_FIND_ROOT_PATH /<path>/<to>/<elfutils>)
set(CMAKE_FIND_ROOT_PATH /<path>/<to>/<libunwind>)
```

Listing B.7: Cmake toolchainfile for arm compilation of Heaptrack

```
./heaptrack.sh <program to be analysed>
```

Listing B.8: How we used Heaptrack

B.5 Heapusage

Heapusage consists of a dynamic library and a shell script for convenience.

We compiled the tool ourselves. Heapusage also used CMake as its build system, so we used a toolchain file to compile the library for the TCAM. The toolchain file is presented in Listing B.9. To run CMake with the toolchain file, we added the `-DCMAKE_TOOLCHAIN_FILE=<path>/<to>/<file>` flag.

The script used to preload Heapusage was written in *bash*, so we had to modify it to use the *sh* language instead, like we did with Heaptrack.

After we had done this we could run Heapusage on the TCAM. However, similar to some other tools, Heapusage reported some false positives in the standard library. That issue was once again due to resources not being deallocated at program exit. To alleviate the issue we modified the source code of Heapusage to call `__libc_freeres(void)`. We added the function call to the destructor of Heapusage, which meant the memory was freed just before program exit. We used the command in Listing B.10 to analyse a program with Heapusage.

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)
set(CMAKE_SYSROOT /<path>/<to>/<target sysroot>)
set(CMAKE_C_COMPILER /<path>/<to>/arm-cas-linux-gnueabi-gcc)
set(CMAKE_CXX_COMPILER /<path>/<to>/arm-cas-linux-gnueabi-g++)
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
```

Listing B.9: Cmake toolchainfile for ARM compilation of Heapusage

```
./heapusage <program to be analysed>
```

Listing B.10: How we used Heapusage

B.6 Valgrind (Memcheck)

One motivation for our thesis is the fact that Valgrind's Memcheck skin was unable to properly run on the TCAM system. For completeness, we wanted to test this, to see if it could run at all and, if it did, how it compared with other tools.

There was no pre-compiled binary of Valgrind which could be run on the TCAM system (ARM-architecture), so we compiled it ourselves. Valgrind is built using *GNU Autotools* together with *Make*.

Valgrind required an un-stripped version of *GNU C Library(glibc)* to be present on the TCAM in order to run. An un-stripped version means a version with debug symbols. The un-stripped glibc was not present by default on the TCAM system, which meant we had to rebuild the firmware image for the TCAM to contain the un-stripped versions. Then we had to flash the TCAM with the new image containing the new library files. It took some time and help from the other developers at Volvo to get the process of rebuilding and flashing correct.

Once we had built Valgrind, and installed a glibc with debug symbols on the TCAM, we could start testing. However, when we tried to analyse a program with Memcheck, we got an error message saying there was too little memory available.

We got rid of the error message by closing a number of running processes. Almost all programs on the TCAM are run as services, i.e they are started on boot and never exit. The services are all managed with a system manager tool called *Systemd*. So to free RAM we closed a number of non critical services with the command `systemctl stop <name of service>`. After closing some services we had enough free memory, and managed to get Memcheck to run. To run valgrind the commands in Listing B.12 was used.

```
./configure --host=armv7-linux \
CC=/<path>/<to>/arm-cas-linux-gnueabi-gcc \
CXX=/<path>/<to>/arm-cas-linux-gnueabi-g++ \
LD=/<path>/<to>/arm-cas-linux-gnueabi-ld \
LDFLAGS=-L/<path>/<to>/<libunwind>/src/.libs \
CFLAGS=-fno-stack-protector \
--prefix=/<path>/<to>/<installation base folder>
--exec-prefix=/<path>/<to>/<installation folder for platform specifics>
```

Listing B.11: Configure script for Valgrind

```
VALGRIND_LIB=<Path to library directory> ./valgrind <program to be analysed>
```

Listing B.12: Command to use Valgrind

B.7 Leaktracer

We built Leaktracer ourselves using GNU Make. The commands we used to build Leaktracer are presented in Listing B.13. To analyse a program with Leaktracer we simply ran the command in Listing B.14.

```
source <script1>
source <script2>
export PATH=$PATH:/<path>/<to>/<target binaries>
CROSS_COMPILE=arm-cas-linux-gnueabi- make
```

Listing B.13: Commands used to build Leaktracer

```
LD_PRELOAD=./libleaktracer.so \
LEAKTRACER_AUTO_REPORTFILENAME=leaks.out ./a.out
```

Listing B.14: Command used to run Leaktracer

B.8 Gperftools Heapcheck

We used the commands in Listing B.15 to compile Gperftools. Gperftools is also preloaded when running a program. So to use it on the TCAM we ran the command in Listing B.16.

```
source <script1>
source <script2>
./configure --host arm-linux CXX=$CXX CC=$CC --prefix=/path/to/install/lib
make CXX=$CXX CC=$CC
make install
```

Listing B.15: Commands used for building Gperftools

```
env HEAPCHECK=normal \
LD_PRELOAD=./libtcmalloc.so ./a.out
```

Listing B.16: Command used to analyse a program with Gperftools

B.9 AddressSanitizer

AddressSanitizer, or ASan, comes bundled with two of the most commonly used C/C++ compilers, *Clang* and *GCC*. Because of this we did not have to build ASan ourselves, we could just use the GCC ARM compiler. To compile a program with ASan activated, it needs to be compiled with the flag `-fsanitize=address`.

ASan will then abort the program if it finds any anomaly during execution, and print the result.

B.10 Duma

To build Duma we ran the commands shown in Listing B.17. To analyse a program with Duma, we preloaded the shared library we had compiled, as shown in Listing B.18.

```
source <script1>
source <script2>
export PATH=$PATH:/<path>/<to>/<target binaries>
make CXX=$CXX CC=$CC
make install
```

Listing B.17: Commands to build Duma

```
LD_PRELOAD=./libduma.so <program to be analysed>
```

Listing B.18: Command to use Duma

B.11 Malloc_count

Malloc_count consists of two C-files and two header-files. The instruction for the tool suggests compiling analysed program together with these files. To avoid having to link the source files with the program, we compiled them as a shared library instead, which we could preload at run time. We used the commands in Listing B.19 to compile Malloc_count. To use the tool the commands in Listing B.20 was used.

```
source <script1>
source <script2>
export PATH=$PATH:/<path>/<to>/<target binaries>
arm-cas-linux-gnueabi-gcc -shared *.c *.h -ldl -fPIC \
-fno-omit-frame-pointer -o malloc_count.so
```

Listing B.19: Commands to build Malloc_count

```
LD_PRELOAD=./malloc_count.so <program to be analysed>
```

Listing B.20: Command to use Malloc_count

B.12 Libleak

Libleak is a simple tool which is supposed to be preloaded with the program to analyse. We built Libleak using the commands in Listing B.21, and we ran it using the commands in Listing B.22. Libleak printed the results of the analysis to a file in the `/tmp/` folder.

```
source <script1>
source <script2>
export PATH=$PATH:/<path>/<to>/<target binaries>
make
```

Listing B.21: Commands used to build Libleak

```
LEAK_EXPIRE=0 LD_PRELOAD=libleak.so <program to be analysed>
```

Listing B.22: Command used to run Libleak

B.13 Mtrace

Mtrace is a tool which is included in the GNU C Library. To use it, we had to include the header file *mcheck.h* in each source file we wanted to analyse. We then had to call `void mtrace(void)` to start the analysis. We did not want to modify every program we wanted to analyse to call Mtrace, so we wrote a small shared library with just a constructor in which we called the Mtrace function. We then preloaded this library with any program we wanted to analyse.

We used the command in Listing B.23 to analyse a program with Mtrace. Mtrace created a logfile which had to be analysed on another machine with GNU Mtrace installed to get human-readable feedback. We used the command in Listing B.24 to get feedback from the logfile.

```
LD_PRELOAD=./libmtrace.so MALLOC_TRACE=<logfilename> <program name>
```

Listing B.23: Command used to run Mtrace

```
mtrace <program binary which was analysed> <logfilename>
```

Listing B.24: Command used to analyse Mtrace logfile

B.14 Rmdebug

Rmdebug consisted of a source-file and a header-file. The *README* suggested compiling the program to be analysed with these files, but we made a shared library instead, and added it to the library path of any program we wanted to analyse. The command we used for compiling Rmdebug is shown in B.25. We also had to include `rmalloc.h` in all source files we wanted to analyse, which we did using the GCC `-include` flag. Finally, to analyse a program with rmdebug, we ran the command in Listing B.26.

```
arm-cas-linux-gnueabi-gcc -shared -DMALLOC_DEBUG -fPIC rmalloc.c -o librmalloc.so
```

Listing B.25: Command used to build Rmdebug

```
LD_LIBRARY_PATH=<directory with libary> <program to be analysed>
```

Listing B.26: Command used to run Rmdebug

B.15 Dr. Memory

We found a pre-compiled version of Dr. Memory for ARM-Linux. It was an older version, and although there are newer versions on the website for other platforms, we could not find any newer versions for ARM.

To analyse a program with Dr.Memory, we used the command in Listing B.27

```
./drmemory -- <program to be analysed>
```

Listing B.27: Command used to run Dr.Memory

Appendix C
Tables

In this appendix we present various tables, mostly of our results from evaluating the different tools. The first table is of the top 15 public domain books from Project Gutenberg, the following two tables show how good each tool was at finding memory faults, the next few tables after that show the execution times of each benchmark when analysed by each tool, and the last few tables show the memory consumption of each benchmark when analysed by each tool.

Books		
Rank	**Title**	**Author**
1	A Modest Proposal	Jonathan Swift
2	Pride and Prejudice	Jane Austen
3	Frankenstein; Or, The Modern Prometheus	Mary Wollstonecraft Shelley
4	A Journal of the Plague Year	Daniel Defoe
5	Alice's Adventures in Wonderland	Lewis Carroll
6	The Importance of Being Earnest: A Trivial Comedy for Serious People	Oscar Wilde
7	The Strange Case of Dr. Jekyll and Mr. Hyde	Robert Louis Stevenson
8	Ion	Plato
9	Grimms' Fairy Tales	Jacob Grimm and Wilhelm Grimm
10	Et dukkehjem. English	Henrik Ibsen
11	Treasure Island	Robert Louis Stevenson
12	A Tale of Two Cities	Charles Dickens
13	A Christmas Carol in Prose; Being a Ghost Story of Christmas	Charles Dickens
14	Little Women	Louisa May Alcott
15	The Yellow Wallpaper	Charlotte Perkins Gilman

Table C.1: Top 15 books from *Project Gutenberg* [34]

	Tools													
Fault	Valgrind	Leaktracer	Gperftools	AddressSanitizer	Duma	Malloc_count	Libleak	Memwatch	Debug_new	Mtrace	Rmdebug	Dr. Memory	Heapusage	Heaptrack
Memory leak (lost ref.)	3/17	17/17	2/17	x	x	17/17	x	17/17	x	3/17	17/17	2/17	4/17	4/17
Double free	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Heap-based Buffer Overflow	1/24	x	x	1/24	1/24	x	x	22/24	x	x	23/24	1/24	x	x
Heap-based Buffer Overread	1/1	x	x	1/1	1/1	x	x	x	x	x	x	1/1	x	x
Heap-based Buffer Underflow	x	x	x	x	x	x	x	12/18	x	x	1/18	x	x	x
Heap-based Buffer Underread	1/3	x	x	2/3	x	x	x	x	x	x	x	x	x	x
Stack Overflow	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Stack-based Buffer Underflow	x	x	x	3/3	x	x	x	x	x	x	x	x	x	x
Stack-based Buffer Overflow	x	x	x	2/52	x	x	x	x	x	x	x	x	x	x
Stack-based Buffer Overread	x	x	x	1/3	x	x	x	x	x	x	x	x	x	x
Stack-based Buffer Underread	x	x	x	1/1	x	x	x	x	x	x	x	x	x	x
Use after free	6/14	x	x	4/14	6/14	x	x	x	x	x	x	5/14	x	x
Use of Uninitialised Variable	2/12	x	x	x	x	x	x	x	x	x	x	x	x	x
Use after return	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Attempt to Access Child of a Non-structure Pointer	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Mismatched Memory Management Routines	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Improper Initialisation	x	x	x	x	x	x	x	x	x	x	x	x	x	x
False positives	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Table C.2: Number and type of memory faults found by the analysis tools in C

	Tools													
Fault	Valgrind	Leaktracer	Gperftools	AddressSanitizer	Duma	Malloc_count	Libleak	Memwatch	Debug_new	Mtrace	Rmdebug	Dr. Memory	Heapusage	Heaptrack
Memory leak (lost ref.)	x	11/11	11/11	x	x	11/11	x	11/11	11/11	11/11	x	11/11	11/11	11/11
Double free	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Heap-based Buffer Overflow	x	x	x	3/4	4/4	x	x	2/4	x	x	x	3/4	x	x
Heap-based Buffer Overread	1/2	x	x	2/2	2/2	x	x	x	x	x	x	2/2	x	x
Heap-based Buffer Underflow	x	x	x	2/2	2/2	x	x	2/2	x	x	x	2/2	x	x
Heap-based Buffer Underread	2/2	x	x	2/2	x	x	x	x	x	x	x	2/2	x	x
Stack Overflow	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Stack-based Buffer Underflow	1/1	x	x	1/1	x	x	x	x	x	x	x	x	x	x
Stack-based Buffer Overflow	x	x	x	1/1	x	x	x	x	x	x	x	x	x	x
Stack-based Buffer Overread	x	x	x	2/2	x	x	x	x	x	x	x	x	x	x
Stack-based Buffer Underread	x	x	x	1/1	x	x	x	x	x	x	x	x	x	x
Use after free	x	x	x	6/6	6/6	x	x	x	x	x	x	6/6	x	x
Use of Uninitialised Variable	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Use after return	1/1	x	x	x	x	x	x	x	x	x	x	x	x	x
Attempt to Access Child of a Non-structure Pointer	x	x	x	1/1	x	x	x	x	x	x	x	x	x	x
Mismatched Memory Management Routines	x	x	x	10/10	10/10	x	x	x	10/10	x	x	10/10	x	x
Improper Initialisation	1/1	x	x	1/1	x	x	x	x	x	x	x	x	x	x
False positives	3/59	2/59	x	x	x	x	x	x	x	x	x	x	1/59	x

Table C.3: Number and type of memory faults found by the analysis tools in C++

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	799.5	821.0	1019.7	697.3	697.1	700.3
dijkstra	242.9	251.2	307.8	218.4	218.6	221.0
cjpeg	121.5	172.0	489.1	102.7	102.6	103.9
madplay	405.3	437.5	646.1	279.2	279.3	280.7
8b10b	578.3	654.4	1019.2	491.3	491.3	492.4
cfrac	459.1	469.6	574.6	407.8	407.7	408.8
espresso	6384.9	6487.3	7648.2	3970.6	3970.9	3992.8
find	4.5	4.9	9.1	3.5	3.5	3.8
grep	859.8	922.1	1232.7	390.3	419.8	507.8
gzip	5686.4	5695.7	6159.7	4555.8	4576.6	4843.8
city	185.9	191.8	231.9	155.8	156.0	157.9
employ	55.6	61.1	112.1	51.6	51.7	53.5
family	9.2	12.8	32.5	8.0	8.1	8.8
primes	4096.8	4457.5	6682.5	3435.9	3458.4	3668.3
simul	166.0	173.0	229.0	141.1	141.0	141.8
deriv2	9.9	11.3	20.9	8.5	8.6	9.3
life	2061.0	2074.5	2179.6	1770.0	1770.5	1776.8
shapes	12.8	16.6	31.6	9.0	9.0	9.7
tramp3d-v4	3213.0	3289.4	4088.2	2734.8	2734.3	2738.5

Table C.4: Execution times in milliseconds for the benchmarks when they are not being analysed by any tool.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	1710.7	1640.0	1984.8	710.9	710.9	715.2
dijkstra	x	x	x	x	x	x
cjpeg	130.5	150.1	273.7	116.7	116.7	117.8
madplay	384.0	412.9	493.5	290.0	290.0	293.1
8b10b	600.6	679.5	1060.2	499.4	499.4	501.6
cfrac	x	x	x	x	x	x
espresso	x	x	x	x	x	x
find	21.4	22.9	32.0	17.5	17.8	20.4
grep	973.8	1012.4	1373.5	454.2	476.1	615.8
gzip	5406.9	5547.1	6018.9	4564.2	4575.6	4708.4
city	200.3	203.2	244.1	165.4	165.5	166.4
employ	71.7	80.2	112.6	62.4	62.4	63.9
family	13.2	13.2	13.9	10.8	10.8	11.2
primes	x	x	x	x	x	x
simul	498.3	501.1	550.5	398.7	401.1	416.0
deriv2	14.8	19.4	38.7	11.5	11.5	12.3
life	x	x	x	x	x	x
shapes	15.9	18.7	41.7	13.2	13.8	16.5
tramp3d-v4	x	x	x	x	x	x

Table C.5: Execution times in milliseconds for the benchmarks when they are being analysed by Duma.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	1879.6	1960.9	2478.2	1034.8	1068.0	1350.1
dijkstra	1687.3	1814.0	2953.5	1039.0	1039.2	1044.2
cjpeg	645.0	711.5	1238.0	130.5	130.6	131.5
madplay	999.6	1140.5	2053.5	307.8	308.1	310.6
8b10b	1092.9	1108.4	1159.6	518.8	518.8	519.9
cfrac	7161.9	7425.3	8331.9	5841.8	5841.3	5865.8
espresso	27903.3	28033.4	29475.7	21635.1	21643.2	21755.2
find	539.2	543.8	561.3	31.5	32.1	37.6
grep	2164.0	2227.0	3100.8	528.0	515.1	573.1
gzip	9111.4	8977.0	10094.1	4535.5	4547.4	4781.9
city	240.6	250.3	348.1	185.3	185.5	187.7
employ	107.5	134.7	232.4	84.4	84.5	87.3
family	535.9	544.5	579.9	30.1	30.1	30.7
primes	5452.4	5432.4	6461.5	4234.6	4214.7	4536.7
simul	758.2	758.9	835.5	192.1	192.2	192.8
deriv2	67.1	74.3	150.5	35.9	36.0	37.3
life	31187.7	31269.4	32021.2	26019.5	26017.5	26183.0
shapes	69.3	75.6	131.4	36.3	36.3	37.1
tramp3d-v4	5886.0	6196.5	7688.0	3343.7	3347.4	3374.8

Table C.6: Execution times in milliseconds for the benchmarks when they are being analysed by Gperftools Heapcheck.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	880.1	888.0	1197.9	709.8	710.0	712.5
dijkstra	2093.3	2148.1	3375.3	935.5	935.2	942.1
cjpeg	191.0	192.0	289.9	114.9	115.0	115.8
madplay	415.4	444.8	602.0	291.4	291.8	294.8
8b10b	731.9	752.1	1072.4	502.7	502.8	506.0
cfrac	9660.5	9534.4	10089.5	5820.2	5802.2	5864.5
espresso	36215.3	36239.1	37389.3	22032.9	22030.6	22255.4
find	29.4	36.3	69.4	16.1	16.2	17.0
grep	2118.5	2132.0	2422.0	425.3	425.9	440.4
gzip	11002.9	10817.0	11990.9	4522.9	4535.4	4645.2
city	320.6	328.2	449.3	163.9	163.8	164.8
employ	74.5	108.6	205.9	59.2	59.2	61.6
family	37.2	36.5	81.7	14.1	14.3	15.1
primes	7146.6	7245.3	8341.5	4233.5	4211.2	4494.2
simul	222.6	304.4	547.0	200.2	200.3	201.7
deriv2	17.7	28.9	60.6	14.2	14.2	14.7
life	30818.9	32089.6	42438.7	25615.8	25553.7	25995.8
shapes	17.9	21.6	37.9	15.1	15.1	15.8
tramp3d-v4	3975.2	4024.2	4510.3	3350.4	3348.2	3354.8

Table C.7: Execution times in milliseconds for the benchmarks when they are being analysed by LeakTracer.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	865.5	883.4	1032.2	754.3	754.0	758.2
dijkstra	1151.1	1164.1	1217.3	971.9	973.7	982.4
cjpeg	191.8	214.6	381.4	155.7	156.1	159.5
madplay	507.0	561.7	1072.6	334.4	339.1	385.7
8b10b	633.3	697.0	1256.4	544.7	544.5	546.4
cfrac	6167.2	6124.1	6743.6	4892.3	4887.1	4936.4
espresso	22710.1	22701.4	24782.2	18584.3	18583.8	18672.8
find	66.2	75.5	134.2	57.1	57.3	58.7
grep	1109.4	1122.8	1305.8	501.1	506.0	562.0
gzip	5631.3	5846.0	6641.3	4516.2	4560.3	4713.8
city	241.0	250.3	304.2	208.4	209.4	219.3
employ	135.4	151.2	229.8	107.0	106.9	108.4
family	64.6	77.5	123.2	55.5	55.7	57.1
primes	15360.1	15336.1	15915.3	12350.3	12367.5	12532.8
simul	261.4	259.9	294.6	217.7	217.8	218.4
deriv2	72.1	81.8	142.2	58.8	58.7	59.5
life	32682.9	32922.9	35253.0	27152.7	27131.1	27266.2
shapes	103.5	115.7	176.8	64.9	65.0	67.7
tramp3d-v4	4085.8	4098.4	4274.5	3421.2	3421.3	3432.8

Table C.8: Execution times in milliseconds for the benchmarks when they are being analysed by Heapusage.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	21239.2	21341.6	22035.1	18013.3	18096.2	18774.1
dijkstra	5777.8	5734.7	5892.7	4859.1	4866.3	5014.7
cjpeg	4699.6	4703.6	5221.0	4007.7	3981.5	4258.5
madplay	11379.9	11501.3	12103.0	7594.3	7628.7	7894.0
8b10b	12414.8	12574.3	14022.5	10012.7	10222.0	10739.6
cfrac	17346.1	17304.6	18074.0	13644.9	13704.3	14199.9
espresso	69119.8	69303.0	70946.7	57033.0	56967.2	57395.8
find	3131.2	3217.2	4144.1	2585.9	2619.1	2859.6
grep	22539.7	22435.6	24663.2	14939.4	14876.7	15183.3
gzip	63911.2	64061.3	66017.5	45708.2	45789.8	46718.2
city	8273.6	8607.1	10479.1	7054.0	6991.9	7144.5
employ	5011.6	4961.8	5547.9	4009.6	4020.1	4221.0
family	2486.0	2576.0	3316.3	2010.3	2058.0	2244.8
primes	59535.0	59628.8	61063.8	49002.5	48991.6	49119.8
simul	5910.0	6168.5	7084.6	5017.3	5080.0	5344.1
deriv2	2976.7	3023.4	3480.2	2430.3	2455.5	2538.0
life	45244.7	45438.4	47247.6	37715.9	37829.3	39585.4
shapes	2976.3	3006.4	3625.5	2397.2	2440.3	2655.5
tramp3d-v4	156716.4	155948.7	157255.4	134370.3	134475.6	135098.7

Table C.9: Execution times in milliseconds for the benchmarks when they are being analysed by Valgrind Memcheck.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	812.3	833.9	1036.4	701.9	701.9	705.5
dijkstra	288.9	288.3	326.6	246.3	246.4	250.6
cjpeg	130.8	139.2	189.9	110.6	110.5	111.2
madplay	394.9	402.5	476.6	287.9	293.4	333.3
8b10b	586.8	583.1	625.5	497.9	498.0	499.1
cfrac	594.5	596.1	631.9	525.3	525.9	535.6
espresso	5065.1	5055.3	5156.3	4428.2	4430.4	4457.1
find	11.2	11.4	13.4	9.9	10.0	10.2
grep	1738.7	1740.0	1898.2	477.5	501.9	595.1
gzip	5076.3	5130.0	5540.4	4420.0	4432.8	4511.9
city	170.7	182.5	223.4	155.9	155.9	156.8
employ	55.3	59.1	77.7	51.8	51.8	52.7
family	11.8	16.1	28.2	9.2	9.4	11.5
primes	6216.4	6445.2	8651.8	5113.4	5131.9	5308.3
simul	151.0	191.3	325.9	142.5	142.6	143.9
deriv2	11.3	15.7	43.0	9.6	9.6	9.8
life	2518.9	2687.1	3459.3	2151.1	2152.9	2180.5
shapes	12.2	15.6	32.1	10.0	10.0	10.3
tramp3d-v4	3260.3	3335.4	3883.4	2755.3	2754.3	2759.4

Table C.10: Execution times in milliseconds for the benchmarks when they are being analysed by Malloc_count.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	1109.6	1192.1	1644.7	841.6	842.9	854.6
dijkstra	82000.3	81971.3	83761.6	66477.0	66374.4	66946.0
cjpeg	317.6	311.7	329.8	263.8	263.6	266.1
madplay	752.7	771.7	963.7	438.2	438.0	441.2
8b10b	744.5	903.8	1664.4	628.0	629.5	637.5
cfrac	539782.3	539782.3	539782.3	440491.4	440491.4	440491.4
espresso	1482846.1	1482846.1	1482846.1	1155958.2	1155958.2	1155958.2
find	241.9	258.2	432.9	192.3	192.3	200.4
grep	3116.4	3216.9	3719.6	1554.7	1561.3	1740.9
gzip	5918.4	5967.2	6282.0	4833.6	4845.0	5106.0
city	437.5	438.8	468.5	366.3	366.5	371.0
employ	317.1	319.1	457.5	241.9	249.9	313.1
family	158.1	168.5	226.7	140.0	140.5	143.6
primes	23681.7	23859.1	26250.8	18939.2	18855.6	19379.5
simul	2988.2	3019.1	3457.6	2525.3	2498.2	2553.1
deriv2	174.5	178.6	221.1	142.8	142.8	144.9
life	1597339.0	1597339.0	1597339.0	1300750.2	1300750.2	1300750.2
shapes	182.1	191.9	263.1	164.5	165.5	170.3
tramp3d-v4	135899.3	135573.3	138807.6	103172.5	102986.2	103744.5

Table C.11: Execution times in milliseconds for the benchmarks when they are being analysed by Heaptrack.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	5557.6	5580.0	5925.6	4066.5	4054.0	4132.0
dijkstra	27365.5	27630.9	29870.7	22162.4	22174.6	22385.1
cjpeg	3284.9	3602.2	4735.9	2631.4	2649.5	2788.0
madplay	6952.2	6816.9	7255.7	3752.3	3752.7	3827.2
8b10b	6681.2	6272.7	7812.1	3349.7	3402.1	3662.6
cfrac	193511.3	190330.6	200909.8	135136.8	134945.4	136636.9
espresso	19877.6	19727.9	23737.4	12117.8	12358.6	14771.0
find	6652.3	6423.5	8575.6	3291.7	3321.7	3619.0
grep	10125.4	10163.1	10796.3	5871.9	5862.9	5910.1
gzip	15646.1	16041.0	18261.6	11529.1	11519.7	11653.9
city	4788.5	4917.2	5538.5	3655.4	3704.8	4220.2
employ	14530.6	14689.7	15343.1	11796.6	11756.1	11946.6
family	3530.4	3626.1	4287.8	2563.6	2564.6	2663.2
primes	24041.7	24271.4	25753.9	19662.7	19636.9	19853.3
simul	4944.6	4904.2	5151.9	3731.9	3707.7	3804.3
deriv2	3961.2	4133.6	5041.0	2925.4	2933.9	3025.1
life	x	x	x	x	x	x
shapes	6338.3	6090.2	7539.2	3128.7	3134.5	3290.8
tramp3d-v4	x	x	x	x	x	x

Table C.12: Execution times in milliseconds for the benchmarks when they are being analysed by Dr. Memory.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	724.5	729.7	801.4	600.9	600.7	602.9
dijkstra	1628.1	1639.5	1793.3	997.9	996.4	1008.6
cjpeg	112.2	121.5	181.0	103.8	103.8	104.3
madplay	408.9	416.2	499.3	283.6	288.9	312.6
8b10b	576.1	576.0	665.1	492.8	493.0	494.4
cfrac	9821.9	9784.0	10613.7	5749.8	5749.6	5788.3
espresso	29597.4	29758.6	32031.0	19413.4	19372.4	19534.1
find	6.7	8.6	25.4	5.7	5.7	6.0
grep	852.4	843.7	900.0	407.7	408.0	436.1
gzip	5209.4	5258.6	5548.1	4466.9	4475.5	4589.9
city	171.2	180.5	222.1	162.1	162.0	162.5
employ	63.0	63.1	67.1	58.7	58.7	59.8
family	11.8	18.3	44.0	10.0	10.6	15.8
primes	6603.2	6539.7	6830.3	5500.0	5492.0	5599.9
simul	375.5	379.6	416.9	336.1	336.1	337.9
deriv2	12.0	12.3	14.4	10.8	10.8	11.5
life	182334.7	182630.6	188346.1	144025.0	144170.1	144667.9
shapes	14.8	15.0	17.0	13.0	13.0	13.4
tramp3d-v4	9914.9	10139.2	12668.6	8209.7	8209.4	8251.7

Table C.13: Execution times in milliseconds for the benchmarks when they are being analysed by Mtrace.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	702.9	711.6	865.9	600.8	600.7	602.1
dijkstra	1598.6	1583.5	1653.7	1370.8	1372.1	1393.4
cjpeg	118.7	135.2	231.6	104.0	104.0	104.7
madplay	393.4	405.6	541.4	283.2	283.5	286.3
8b10b	589.0	585.1	622.2	492.7	492.6	494.8
cfrac	469.7	477.1	535.9	424.0	423.6	426.7
espresso	4616.5	4617.1	4717.8	3975.4	3977.0	4019.2
find	x	x	x	x	x	x
grep	x	x	x	x	x	x
gzip	x	x	x	x	x	x
city	168.7	188.6	250.1	158.5	158.5	159.3
employ	x	x	x	x	x	x
family	x	x	x	x	x	x
primes	x	x	x	x	x	x
simul	x	x	x	x	x	x
deriv2	x	x	x	x	x	x
life	x	x	x	x	x	x
shapes	x	x	x	x	x	x
tramp3d-v4	x	x	x	x	x	x

Table C.14: Execution times in milliseconds for the benchmarks when they are being analysed by Memwatch.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	790.6	792.5	872.7	680.7	680.3	683.0
dijkstra	793.1	813.5	990.3	671.8	672.2	680.2
cjpeg	368.9	383.1	449.2	334.7	334.5	335.9
madplay	1462.4	1494.5	1629.5	983.1	989.8	1041.8
8b10b	921.9	926.1	956.7	797.2	797.3	799.1
cfrac	3226.4	3161.9	3369.5	2483.6	2478.8	2515.9
espresso	x	x	x	x	x	x
find	40.9	48.6	96.8	34.4	35.8	45.2
grep	2509.1	2385.8	2951.1	1198.1	1151.8	1496.6
gzip	10199.0	10211.7	10376.4	8690.4	8686.9	8867.3
city	439.3	440.1	472.1	382.4	382.4	384.3
employ	104.0	104.7	116.3	87.2	86.9	88.4
family	27.8	33.0	68.1	23.4	25.3	38.0
primes	10857.7	10801.3	11012.8	9024.5	9035.4	9230.7
simul	414.1	405.7	457.7	351.7	351.8	352.8
deriv2	27.1	31.8	64.7	24.6	24.8	25.7
life	x	x	x	x	x	x
shapes	27.9	34.7	55.3	25.2	25.3	25.8
tramp3d-v4	6298.2	6446.7	8129.4	4723.9	4753.0	4853.2

Table C.15: Execution times in milliseconds for the benchmarks when they are being analysed by AddressSanitizer.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	709.7	708.1	771.7	601.9	602.5	614.7
dijkstra	368.3	374.6	440.5	321.5	321.5	323.1
cjpeg	121.5	138.0	195.4	105.3	105.3	106.3
madplay	428.8	427.7	506.3	285.8	286.1	289.6
8b10b	576.3	573.8	671.7	490.9	491.0	492.2
cfrac	486.8	494.8	558.6	415.9	415.4	417.6
espresso	4651.9	4630.0	4739.3	3994.5	3989.1	4011.5
find	x	x	x	x	x	x
grep	x	x	x	x	x	x
gzip	x	x	x	x	x	x
city	x	x	x	x	x	x
employ	x	x	x	x	x	x
family	x	x	x	x	x	x
primes	x	x	x	x	x	x
simul	x	x	x	x	x	x
deriv2	x	x	x	x	x	x
life	x	x	x	x	x	x
shapes	x	x	x	x	x	x
tramp3d-v4	x	x	x	x	x	x

Table C.16: Execution times in milliseconds for the benchmarks when they are being analysed by Rmalloc.

	Real time elapsed (ms)			Task-clock time elapsed (ms)		
benchmark	median	mean	max	median	mean	max
fft	x	x	x	x	x	x
dijkstra	x	x	x	x	x	x
cjpeg	x	x	x	x	x	x
madplay	x	x	x	x	x	x
8b10b	x	x	x	x	x	x
cfrac	x	x	x	x	x	x
espresso	x	x	x	x	x	x
find	x	x	x	x	x	x
grep	x	x	x	x	x	x
gzip	x	x	x	x	x	x
city	180.7	206.8	298.6	160.6	160.6	161.8
employ	68.4	74.3	108.9	57.2	57.3	58.6
family	9.1	10.3	21.4	7.7	7.8	8.0
primes	11829.5	11744.0	12020.4	9813.5	9812.1	9874.7
simul	211.2	215.7	230.6	199.0	199.1	199.8
deriv2	10.8	13.8	28.6	9.0	9.3	10.5
life	4025.9	4038.7	4228.9	3406.1	3406.2	3435.7
shapes	11.3	13.2	29.8	9.9	10.0	10.6
tramp3d-v4	x	x	x	x	x	x

Table C.17: Execution times in milliseconds for the benchmarks when they are being analysed by Debug_new.

No tools

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	888	978	978	439	508	723	416	483	696	870	962	1208	421	498	723	398	473	696	1208
dijkstra	664	664	664	216	216	218	202	202	204	660	659	664	214	213	218	200	199	204	668
cjpeg	942	942	942	517	517	769	508	508	760	1108	1093	1468	687	671	1045	678	662	1036	1468
madplay	984	996	996	502	508	646	488	492	632	1020	1011	1156	538	526	674	524	511	660	1176
8b10b	620	620	620	324	323	324	316	315	316	620	619	628	324	322	324	316	314	316	628
cfrac	732	732	732	412	410	420	384	382	392	728	726	744	408	406	420	380	378	392	748
espresso	1080	1063	1063	748	731	776	736	719	764	1080	1059	1112	748	727	776	736	715	764	1116
find	868	868	868	497	497	497	480	480	480	864	864	864	493	493	493	476	476	476	872
grep	988	988	988	664	664	668	648	648	652	988	988	992	664	664	668	648	648	652	992
gzip	980	978	978	685	683	685	676	674	676	980	977	980	685	682	685	676	673	676	984
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	1342	1342	1342	931	931	935	916	916	920	1336	1335	1344	927	927	935	912	912	920	1348
employ	1428	1428	1428	947	947	947	928	928	928	1420	1420	1420	943	943	943	924	924	924	1432
family	1156	1156	1156	822	822	822	812	812	812	1144	1144	1144	818	818	818	808	808	808	1160
primes	1452	1436	1436	1130	1114	1204	1120	1104	1192	1452	1433	1532	1130	1111	1204	1120	1101	1192	1536
simul	1188	1188	1188	868	868	870	858	858	860	1188	1183	1188	870	866	870	860	856	860	1192
deriv2	1364	1364	1364	925	925	925	908	908	908	1356	1356	1356	921	921	921	904	904	904	1368
life	1268	1268	1268	946	946	946	936	936	936	1268	1268	1272	946	946	946	936	936	936	1272
shapes	1320	1320	1320	915	915	915	900	900	900	1312	1312	1312	911	911	911	896	896	896	1324
tramp3d-v4	4088	3988	3988	3568	3473	3600	3536	3442	3568	4090	3966	4120	3570	3451	3600	3538	3420	3568	4120

Table C.18: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are not being analysed by any tool.

Mtrace																			
	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	904	989	989	406	475	718	384	449	688	896	996	1248	398	484	718	376	458	688	1248
dijkstra	704	704	704	209	209	209	196	196	196	704	703	704	209	208	209	196	195	196	708
cjpeg	1118	1118	1118	658	658	1046	648	648	1036	1176	1135	1500	718	677	1042	708	667	1032	1508
madplay	992	1035	1035	477	516	729	464	502	716	974	1015	1224	459	498	709	446	484	696	1244
8b10b	660	661	661	213	213	217	168	168	172	660	659	664	213	212	217	168	167	172	668
cfrac	756	741	741	311	297	327	240	226	256	756	741	772	311	297	327	240	226	256	772
espresso	1116	1097	1097	673	654	702	608	589	636	1116	1096	1148	673	653	702	608	588	636	1152
find	912	912	912	411	411	411	348	348	348	908	908	908	407	407	407	344	344	344	916
grep	1028	1028	1028	572	572	576	508	508	512	1028	1028	1032	572	572	576	508	508	512	1032
gzip	1016	1014	1014	589	587	593	528	526	532	1016	1012	1020	589	585	593	528	524	532	1024
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	1382	1382	1382	941	941	945	900	900	904	1372	1375	1384	933	936	945	892	895	904	1388
employ	1472	1472	1472	962	962	962	916	916	916	1464	1464	1464	958	958	958	912	912	912	1480
family	1192	1192	1192	827	827	827	792	792	792	1184	1184	1184	823	823	823	788	788	788	1196
primes	1472	1453	1453	1115	1096	1209	1080	1061	1172	1470	1449	1568	1113	1092	1209	1078	1057	1172	1576
simul	1232	1229	1229	875	871	875	840	836	840	1230	1223	1232	873	866	875	838	831	840	1236
deriv2	1408	1408	1408	935	935	935	892	892	892	1400	1400	1400	931	931	931	888	888	888	1412
life	1316	1316	1316	955	955	955	920	920	920	1316	1316	1316	955	955	955	920	920	920	1320
shapes	1364	1364	1364	925	925	925	884	884	884	1356	1356	1356	921	921	921	880	880	880	1368
tramp3d-v4	4144	4026	4026	3591	3479	3611	3532	3421	3552	4144	4017	4164	3591	3470	3611	3532	3412	3552	4164

Table C.19: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Mtrace.

Memwatch

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	1236	1173	1173	803	724	819	780	697	788	1236	1210	1284	803	765	819	780	739	788	1292
dijkstra	760	760	760	328	328	332	316	316	320	760	760	768	328	327	332	316	315	320	768
cjpeg	1152	1152	1152	747	747	1129	738	738	1120	1520	1336	1536	1119	934	1133	1110	925	1124	1544
madplay	1092	1089	1089	610	603	766	596	588	752	1084	1079	1248	602	595	766	588	581	752	1252
8b10b	692	694	694	341	343	353	308	310	320	692	691	704	341	340	353	308	307	320	708
cfrac	834	833	833	464	461	486	410	407	432	824	823	860	454	453	486	400	399	432	864
espresso	1092	1076	1076	740	724	801	704	688	764	1092	1075	1184	740	723	801	704	687	764	1184
find																			
grep																			
gzip																			
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city																			
employ																			
family																			
primes																			
simul																			
deriv2																			
life																			
shapes																			
tramp3d-v4																			

Table C.20: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Memwatch.

AddressSanitizer

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	5320	5292	5292	4855	4826	5104	4728	4699	4972	5156	5217	5588	4708	4754	5104	4584	4627	4972	5592
dijkstra	6484	6617	6617	6024	6157	7508	5908	6041	7392	6310	6457	7992	5850	5997	7532	5734	5881	7416	7996
cjpeg	8128	7955	7955	7690	7517	7811	7580	7406	7700	8064	7677	8268	7626	7239	7830	7516	7129	7720	8272
madplay	5764	5775	5775	5272	5287	5534	5158	5172	5420	5764	5761	6028	5274	5274	5542	5160	5160	5428	6052
8b10b	5064	5063	5063	4434	4432	4490	4416	4414	4472	5056	5057	5136	4426	4427	4506	4408	4409	4488	5148
cfrac	21152	20517	20517	20520	19923	31525	20490	19894	31500	20732	20059	31912	20098	19459	31537	20068	19430	31512	31916
espresso																			
find	6992	6992	6992	6586	6586	6586	6564	6564	6564	6992	6992	6992	6586	6586	6586	6564	6564	6564	7020
grep	8452	8455	8455	8088	8091	8204	8068	8071	8184	8440	8441	8584	8076	8077	8220	8056	8057	8200	8596
gzip	6588	6566	6566	6248	6226	6676	6232	6210	6660	6584	6556	7028	6244	6216	6688	6228	6200	6672	7040
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	5260	5257	5257	4867	4863	4883	4848	4844	4864	5250	5249	5292	4857	4857	4899	4838	4838	4880	5304
employ	5168	5168	5168	4708	4708	4708	4684	4684	4684	5158	5158	5176	4702	4702	4720	4678	4678	4696	5188
family	4764	4764	4764	4435	4435	4435	4420	4420	4420	4756	4756	4756	4435	4435	4435	4420	4420	4420	4788
primes	5808	5754	5754	5487	5433	6065	5472	5418	6048	5802	5743	6400	5481	5422	6077	5466	5407	6060	6416
simul	4872	4866	4866	4553	4548	4563	4538	4533	4548	4862	4809	4892	4545	4493	4575	4530	4478	4560	4904
deriv2	5244	5244	5244	4817	4817	4817	4796	4796	4796	5240	5240	5240	4817	4817	4817	4796	4796	4796	5268
life																			
shapes	5048	5048	5048	4655	4655	4655	4636	4636	4636	5044	5044	5044	4655	4655	4655	4636	4636	4636	5072
tramp3d-v4	20104	20014	20014	19599	19516	25187	19560	19478	25148	19942	19794	25712	19437	19296	25207	19398	19259	25168	25716

Table C.21: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by AddressSanitizer.

Duma

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	1860	1875	1875	1610	1610	1693	1568	1566	1644	1844	1857	1968	1594	1596	1684	1552	1553	1636	1988
dijkstra																			
cjpeg	2154	2154	2154	1907	1907	1923	1876	1876	1892	2138	2136	2160	1893	1892	1915	1862	1861	1884	2180
madplay	1640	1662	1662	1325	1360	1551	1276	1315	1508	1646	1675	1900	1340	1375	1603	1294	1331	1560	1932
8b10b	1280	1283	1283	978	980	1026	968	970	1016	1280	1279	1344	978	976	1034	968	966	1024	1352
cfrac																			
espresso																			
find	1732	1732	1732	950	950	950	700	700	700	1720	1720	1720	939	939	939	688	688	688	1744
grep	2688	2608	2608	1933	1892	2262	1680	1661	2120	2600	2568	2876	1845	1838	2254	1592	1600	2112	2884
gzip	2318	2311	2311	1832	1825	2447	1706	1699	2320	2296	2294	2924	1810	1808	2438	1684	1682	2312	2944
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	1730	1730	1730	1185	1185	1201	1052	1052	1068	1714	1713	1736	1171	1171	1193	1038	1038	1060	1756
employ	1692	1692	1692	1085	1085	1085	940	940	940	1676	1676	1676	1073	1073	1073	928	928	928	1704
family	1244	1244	1244	767	767	767	644	644	644	1224	1224	1224	755	755	755	632	632	632	1256
primes																			
simul	6962	6897	6897	6490	6425	8876	6360	6295	8740	6436	6423	9368	5968	5952	8884	5842	5823	8748	9376
deriv2	1464	1464	1464	879	879	879	728	728	728	1448	1448	1448	866	866	866	716	716	716	1476
life																			
shapes	1484	1484	1484	929	929	929	784	784	784	1468	1468	1468	917	917	917	772	772	772	1496
tramp3d-v4																			

Table C.22: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Duma.

Gperftools

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	3292	3157	3157	2983	2833	2990	2932	2780	2932	3160	3126	3332	2851	2804	2989	2800	2752	2932	3336
dijkstra	2780	2778	2778	2471	2469	2471	2424	2422	2424	2780	2778	2784	2471	2469	2471	2424	2422	2424	2784
cjpeg	3698	3698	3698	3398	3398	3406	3360	3360	3368	3694	3606	3704	3395	3308	3405	3358	3271	3368	3708
madplay	3048	3086	3086	2703	2737	2923	2660	2692	2880	3080	3084	3272	2735	2738	2927	2692	2695	2884	3288
8b10b	2712	2714	2714	2173	2173	2174	2032	2032	2032	2712	2712	2716	2173	2172	2174	2032	2031	2032	2720
cfrac	2852	2850	2850	2287	2285	2316	2128	2126	2156	2852	2849	2884	2287	2284	2315	2128	2126	2156	2888
espresso	3860	3828	3828	3289	3258	3397	3140	3109	3248	3860	3827	3972	3289	3256	3397	3140	3107	3248	3976
find	3012	3012	3012	2413	2413	2413	2264	2264	2264	3012	3012	3012	2413	2413	2413	2264	2264	2264	3012
grep	3184	3183	3183	2611	2610	2615	2460	2459	2464	3184	3182	3188	2611	2609	2615	2460	2458	2464	3188
gzip	3120	3118	3118	2568	2566	2568	2424	2422	2424	3120	3117	3120	2568	2565	2568	2424	2421	2424	3124
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	2888	2888	2888	2278	2278	2280	2126	2126	2128	2880	2881	2888	2272	2275	2280	2120	2123	2128	2892
employ	2952	2952	2952	2295	2295	2295	2132	2132	2132	2948	2948	2948	2295	2295	2295	2132	2132	2132	2968
family	2704	2704	2704	2158	2158	2158	2016	2016	2016	2696	2696	2696	2157	2157	2157	2016	2016	2016	2704
primes	3104	3076	3076	2565	2537	2672	2424	2396	2528	3096	3070	3212	2557	2531	2671	2416	2390	2528	3216
simul	2762	2762	2762	2225	2225	2225	2084	2084	2084	2756	2751	2760	2221	2217	2225	2080	2076	2084	2764
deriv2	2904	2904	2904	2269	2269	2269	2112	2112	2112	2900	2900	2900	2269	2269	2269	2112	2112	2112	2904
life	2852	2852	2852	2314	2314	2314	2172	2172	2172	2852	2852	2852	2314	2314	2314	2172	2172	2172	2856
shapes	2872	2872	2872	2268	2268	2268	2116	2116	2116	2868	2868	2868	2267	2267	2267	2116	2116	2116	2872
tramp3d-v4	7428	7277	7277	6732	6586	7912	6544	6399	7724	7428	7236	8608	6732	6545	7912	6544	6359	7724	8612

Table C.23: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Gperftools.

Leaktracer

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	2116	2187	2187	1755	1811	2085	1696	1749	2020	2140	2223	2476	1779	1849	2085	1720	1788	2020	2476
dijkstra	1952	1952	1952	1594	1594	1594	1540	1540	1540	1952	1951	1956	1594	1593	1595	1540	1539	1540	1956
cjpeg	2294	2294	2294	1946	1946	2257	1900	1900	2212	2346	2358	2760	1999	2011	2413	1954	1966	2368	2760
madplay	2260	2282	2282	1872	1890	2088	1820	1837	2036	2254	2258	2464	1866	1868	2076	1814	1816	2024	2476
8b10b	1888	1889	1889	1286	1285	1287	1140	1139	1140	1888	1887	1892	1286	1284	1286	1140	1138	1140	1896
cfrac	1992	1994	1994	1367	1369	1383	1200	1202	1216	1992	1993	2012	1367	1368	1383	1200	1201	1216	2016
espresso	2348	2381	2381	1715	1748	1923	1560	1593	1768	2348	2381	2564	1715	1748	1923	1560	1593	1768	2564
find	2136	2136	2136	1603	1603	1603	1512	1512	1512	2132	2132	2132	1599	1599	1599	1508	1508	1508	2140
grep	2272	2257	2257	1768	1753	1768	1676	1661	1676	2272	2256	2272	1768	1752	1768	1676	1660	1676	2272
gzip	2272	2270	2270	1790	1788	1790	1704	1702	1704	2272	2269	2272	1790	1787	1790	1704	1701	1704	2276
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	1900	1900	1900	1416	1416	1418	1330	1330	1332	1892	1893	1900	1410	1411	1418	1324	1325	1332	1904
employ	1984	1984	1984	1442	1442	1442	1344	1344	1344	1976	1976	1976	1438	1438	1438	1340	1340	1340	1988
family	1864	1864	1864	1388	1388	1388	1304	1304	1304	1856	1856	1856	1384	1384	1384	1300	1300	1300	1868
primes	2340	2371	2371	1872	1903	2130	1788	1819	2044	2336	2361	2600	1868	1893	2130	1784	1809	2044	2608
simul	1900	1900	1900	1434	1434	1436	1350	1350	1352	1896	1895	1900	1431	1430	1435	1348	1347	1352	1904
deriv2	1920	1920	1920	1411	1411	1411	1320	1320	1320	1912	1912	1912	1407	1407	1407	1316	1316	1316	1924
life	2156	2156	2156	1691	1691	1692	1608	1608	1608	2156	2156	2156	1691	1691	1691	1608	1608	1608	2160
shapes	1876	1876	1876	1398	1398	1398	1312	1312	1312	1868	1868	1868	1393	1393	1393	1308	1308	1308	1880
tramp3d-v4	4836	4713	4713	4240	4123	4264	4120	4005	4144	4836	4694	4860	4240	4104	4264	4120	3986	4144	4860

Table C.24: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Leaktracer.

Heapusage																			
	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	1516	1583	1583	1109	1159	1427	972	1019	1284	1516	1598	1868	1109	1178	1427	972	1039	1284	1868
dijkstra	1384	1383	1383	981	980	981	852	851	852	1384	1382	1384	981	979	981	852	850	852	1388
cjpeg	1766	1766	1766	1370	1370	1757	1248	1248	1636	1970	1868	2156	1577	1474	1761	1456	1353	1640	2156
madplay	1658	1671	1671	1212	1220	1374	1082	1088	1244	1672	1687	1852	1226	1238	1406	1096	1108	1276	1876
8b10b	1280	1281	1281	763	762	764	656	655	656	1280	1279	1284	763	761	763	656	654	656	1288
cfrac	1474	1476	1476	926	928	1028	794	796	896	1472	1473	1580	924	925	1028	792	793	896	1584
espresso	1860	1844	1844	1314	1298	1702	1200	1184	1588	1860	1842	2248	1314	1296	1702	1200	1182	1588	2248
find	1532	1532	1532	945	945	945	824	824	824	1528	1528	1528	941	941	941	820	820	820	1536
grep	1680	1680	1680	1127	1127	1131	1008	1008	1012	1680	1677	1684	1127	1124	1131	1008	1005	1012	1684
gzip	1668	1666	1666	1138	1136	1138	1028	1026	1028	1668	1665	1668	1138	1135	1138	1028	1025	1028	1672
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	1470	1470	1470	882	882	886	764	764	768	1464	1463	1472	878	878	886	760	760	768	1480
employ	1540	1540	1540	905	905	905	776	776	776	1532	1532	1536	902	902	904	774	774	776	1544
family	1280	1280	1280	753	753	753	644	644	644	1272	1272	1272	748	748	748	640	640	640	1288
primes	2392	2337	2337	1875	1820	2253	1768	1713	2144	2388	2333	2772	1871	1816	2253	1764	1709	2144	2780
simul	1540	1540	1540	1025	1025	1027	918	918	920	1532	1531	1540	1019	1020	1027	912	913	920	1548
deriv2	1484	1484	1484	871	871	871	748	748	748	1472	1472	1472	865	865	865	744	744	744	1492
life	1932	1932	1932	1419	1419	1419	1312	1312	1312	1932	1931	1932	1419	1418	1419	1312	1311	1312	1936
shapes	1448	1448	1448	865	865	865	748	748	748	1436	1436	1436	860	860	860	744	744	744	1452
tramp3d-v4	4516	4393	4393	3831	3714	3851	3680	3564	3700	4516	4369	4536	3831	3691	3851	3680	3542	3700	4536

Table C.25: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Heapusage.

Rmalloc

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	908	989	989	514	578	818	492	553	788	924	995	1244	530	589	818	508	564	788	1244
dijkstra	716	715	715	317	316	321	304	303	308	716	713	724	317	314	321	304	301	308	724
cjpeg	1124	1124	1124	746	746	1132	738	738	1124	1336	1230	1512	960	854	1136	952	846	1128	1512
madplay	1036	1021	1021	609	589	713	596	574	700	1056	1034	1164	629	605	737	616	591	724	1176
8b10b	656	657	657	332	332	337	300	300	304	656	655	660	332	331	336	300	299	304	664
cfrac	796	796	796	449	446	470	396	393	416	788	787	824	441	439	470	388	386	416	828
espresso	1208	1144	1144	856	792	936	820	756	900	1208	1143	1244	856	791	892	820	755	856	1296
find																			
grep																			
gzip																			
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city																			
employ																			
family																			
primes																			
simul																			
deriv2																			
life																			
shapes																			
tramp3d-v4																			

Table C.26: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Rmalloc.

Malloc_count

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	1460	1566	1566	1151	1243	1473	1104	1194	1420	1464	1550	1816	1155	1230	1473	1108	1182	1420	1816
dijkstra	1298	1298	1298	985	985	987	942	942	944	1300	1296	1300	987	983	987	944	940	944	1300
cjpeg	1708	1708	1708	1413	1413	1801	1380	1380	1768	1940	1825	2100	1647	1531	1805	1614	1498	1772	2100
madplay	1560	1582	1582	1235	1251	1415	1196	1210	1376	1596	1595	1776	1271	1267	1451	1232	1227	1412	1792
8b10b	1228	1228	1228	739	738	739	608	607	608	1228	1226	1232	739	737	739	608	606	608	1232
cfrac	1360	1355	1355	848	842	860	696	690	708	1352	1349	1376	840	837	860	688	685	708	1376
espresso	1752	1726	1726	1230	1204	1246	1092	1066	1108	1752	1724	1768	1230	1202	1246	1092	1064	1108	1768
find	1456	1456	1456	920	920	920	780	780	780	1452	1452	1452	916	916	916	776	776	776	1460
grep	1600	1600	1600	1096	1096	1100	956	956	960	1600	1597	1604	1096	1093	1100	956	953	960	1604
gzip	1612	1610	1610	1114	1112	1114	980	978	980	1612	1609	1612	1114	1111	1114	980	977	980	1612
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	1396	1396	1396	844	844	848	704	704	708	1396	1392	1400	844	841	848	704	701	708	1400
employ	1480	1480	1480	868	868	868	716	716	716	1476	1476	1476	864	864	864	712	712	712	1488
family	1212	1212	1212	723	723	723	592	592	592	1204	1204	1204	718	718	718	588	588	588	1216
primes	1624	1602	1602	1142	1120	1256	1012	990	1124	1624	1600	1740	1142	1118	1256	1012	988	1124	1740
simul	1274	1274	1274	796	796	798	666	666	668	1268	1268	1276	790	791	798	660	661	668	1280
deriv2	1416	1416	1416	841	841	841	696	696	696	1412	1412	1412	837	837	837	692	692	692	1420
life	1424	1424	1424	942	942	942	812	812	812	1424	1423	1424	942	942	942	812	812	812	1424
shapes	1372	1372	1372	828	828	828	688	688	688	1368	1368	1368	824	824	824	684	684	684	1376
tramp3d-v4	4164	4026	4026	3498	3366	3526	3324	3194	3352	4164	4002	4192	3498	3342	3526	3324	3170	3352	4192

Table C.27: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Malloc_count.

Valgrind

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	23832	23698	23698	23510	23383	23762	23472	23343	23716	23836	23713	24100	23514	23397	23762	23476	23357	23716	24144
dijkstra	26132	26069	26069	25840	25777	29664	25808	25745	29632	25924	26034	29964	25632	25742	29664	25600	25710	29632	30004
cjpeg	23840	23645	23645	23578	23384	23774	23560	23366	23756	23572	23563	24108	23310	23301	23846	23292	23283	23828	24052
madplay	23728	23721	23721	23404	23397	23916	23376	23369	23888	23722	23710	24148	23398	23386	23824	23370	23358	23796	24176
8b10b	21464	21460	21460	21090	21086	21218	21068	21064	21196	21460	21456	21524	21086	21082	21150	21064	21060	21128	21644
cfrac																			
espresso																			
find	23240	23240	23240	22856	22856	22998	22834	22834	22976	23140	23140	23140	22762	22762	22762	22740	22740	22740	23388
grep	24532	24451	24451	24151	24074	24651	24092	24015	24592	24532	24451	24972	24151	24073	24615	24092	24014	24556	25044
gzip	23728	23674	23674	23423	23349	23774	23374	23293	23712	23732	23679	24112	23428	23353	23778	23376	23297	23716	24112
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	23832	23827	23827	23193	23193	23305	23068	23076	23180	23832	23828	23972	23193	23197	23317	23072	23082	23192	24004
employ	24084	24084	24084	23632	23631	23753	23572	23571	23692	24084	24076	24116	23632	23624	23664	23572	23564	23604	24216
family	23164	23164	23164	22846	22846	22846	22800	22800	22800	23164	23164	23164	22846	22846	22846	22800	22800	22800	23252
primes	24440	24375	24375	24130	24065	24870	24084	24019	24824	24436	24378	25248	24126	24068	24938	24080	24022	24892	25288
simul	23350	23355	23355	23044	23050	23126	22998	23004	23080	23350	23355	23432	23044	23050	23126	22998	23004	23080	23484
deriv2	23812	23812	23812	23361	23361	23361	23284	23284	23284	23860	23860	23860	23409	23409	23409	23332	23332	23332	23912
life																			
shapes	23560	23560	23560	22870	22870	22870	22852	22852	22852	23548	23548	23548	22862	22862	22862	22844	22844	22844	23720
tramp3d-v4																			

Table C.28: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Valgrind.

	Heaptrack																		
	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	3248	3167	3167	2478	2382	2648	2152	2053	2316	3120	3161	3448	2350	2380	2648	2024	2051	2316	3460
dijkstra	5892	5816	5816	5110	5068	7840	4800	4786	7720	5888	5811	8472	5106	5063	7877	4796	4781	7780	8488
cjpeg	3348	3216	3216	2594	2461	2982	2284	2151	2672	3612	3348	3736	2858	2595	2982	2548	2285	2672	3748
madplay	3208	3147	3147	2415	2351	2499	2104	2039	2188	3212	3169	3332	2419	2374	2539	2108	2063	2228	3352
8b10b	2860	2781	2781	2056	1976	2056	1788	1708	1788	2860	2785	2860	2056	1981	2056	1788	1713	1788	2872
cfrac																			
espresso																			
find	2956	2956	2956	2086	2086	2275	1810	1810	2000	2740	2832	3164	1871	1968	2293	1596	1693	2016	3176
grep	3284	3054	3054	2490	2365	2491	2152	2105	2156	3284	3031	3284	2490	2341	2494	2152	2079	2156	3296
gzip	3244	3234	3234	2451	2442	2453	2152	2144	2160	3244	3234	3244	2451	2442	2453	2152	2144	2160	3256
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	3012	2885	2885	2130	2002	2134	1788	1660	1792	3016	2852	3020	2134	1971	2138	1792	1629	1796	3032
employ	3080	3080	3080	2157	2157	2157	1808	1808	1808	2676	2661	2692	1753	1748	1765	1404	1401	1416	3096
family	2444	2444	2444	1660	1660	1660	1368	1368	1368	2428	2428	2428	1651	1651	1651	1360	1360	1360	2460
primes	3048	3033	3033	2262	2247	2406	1964	1949	2108	3048	3035	3196	2262	2249	2408	1964	1951	2108	3208
simul	2864	2865	2865	2081	2082	2097	1784	1785	1800	2864	2843	2880	2081	2061	2097	1784	1764	1800	2892
deriv2	2648	2648	2648	1735	1735	1735	1388	1388	1388	2636	2636	2636	1726	1726	1726	1380	1380	1380	2664
life																			
shapes	2612	2612	2612	1727	1727	1727	1384	1384	1384	2600	2600	2600	1718	1718	1718	1376	1376	1376	2628
tramp3d-v4	7016	6992	6992	6514	6431	7801	6456	6335	7744	7016	6990	8264	6514	6428	7797	6456	6332	7740	8296

Table C.29: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Heaptrack.

Dr. Memory

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft	10992	12189	12189	10625	11758	13940	10580	11694	13844	11020	12240	14448	10645	11807	13944	10600	11742	13848	15688
dijkstra																			
cjpeg	10750	10720	10720	10338	10323	10707	10248	10249	10656	10812	10743	11084	10400	10348	10711	10310	10275	10660	15100
madplay	11588	11517	11517	11113	11043	11325	11020	10949	11232	11600	11528	11820	11125	11054	11345	11032	10961	11252	16048
8b10b	11064	11062	11062	10711	10708	10739	10680	10677	10708	11062	11063	11108	10709	10709	10743	10678	10678	10712	15964
cfrac																			
espresso																			
find	15802	15811	15811	14567	14576	14668	14532	14542	14632	15800	15817	15924	14563	14581	14668	14528	14547	14632	16384
grep	15916	14736	14736	14736	13578	14840	14696	13540	14800	15920	14789	16024	14740	13629	14844	14700	13591	14804	16288
gzip	15636	15626	15626	14527	14517	14791	14496	14486	14760	15640	15630	15904	14531	14522	14787	14500	14491	14756	16180
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	12090	12041	12041	10523	10483	10561	10446	10406	10484	12088	12052	12132	10521	10491	10557	10444	10414	10480	16116
employ	16336	15988	15988	14300	13990	14572	14200	13892	14472	16336	16011	16612	14300	14010	14568	14200	13911	14468	16624
family	11704	11704	11704	10205	10205	10205	10132	10132	10132	11696	11696	11696	10201	10201	10201	10128	10128	10128	16060
primes	16392	16069	16069	14775	14462	15477	14696	14384	15396	16390	16067	17104	14773	14460	15477	14694	14382	15396	17120
simul	11860	11842	11842	10373	10355	10389	10300	10282	10316	11864	11852	11896	10377	10364	10397	10304	10291	10324	16068
deriv2	11944	11939	11939	10382	10380	10432	10304	10302	10352	11932	11942	12024	10377	10381	10424	10300	10303	10344	16084
life																			
shapes	11900	11920	11920	10352	10376	10469	10276	10300	10392	11900	11913	12024	10352	10370	10457	10276	10294	10380	16080
tramp3d-v4																			

Table C.30: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Dr. Memory.

Debug_new

	Sample time 137 (ms)									Sample time 59 (ms)									
Benchmark	**RSS**			**PSS**			**USS**			**RSS**			**PSS**			**USS**			**VmHWM**
C Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
fft																			
dijkstra																			
cjpeg																			
madplay																			
8b10b																			
cfrac																			
espresso																			
find																			
grep																			
gzip																			
C++ Language	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	Median	Mean	Peak	
city	1354	1355	1355	937	937	945	896	896	904	1348	1349	1364	932	932	945	892	892	904	1372
employ	1448	1448	1448	956	956	956	908	908	908	1440	1440	1440	952	952	952	904	904	904	1452
family	1160	1160	1160	818	818	818	784	784	784	1148	1148	1148	814	814	814	780	780	780	1164
primes	1928	1893	1893	1594	1558	1804	1560	1524	1768	1928	1891	2148	1594	1556	1804	1560	1522	1768	2148
simul	1336	1333	1333	998	996	998	964	962	964	1336	1330	1336	998	995	998	964	961	964	1340
deriv2	1384	1384	1384	934	934	934	892	892	892	1376	1376	1376	930	930	930	888	888	888	1388
life	1544	1545	1545	1214	1214	1214	1180	1180	1180	1544	1545	1552	1214	1214	1214	1180	1180	1180	1556
shapes	1344	1344	1344	924	924	924	884	884	884	1336	1336	1336	920	920	920	880	880	880	1352
tramp3d-v4																			

Table C.31: Memory usage measure from the `/proc/[PID]/` folder for the benchmarks when they are being analysed by Debug_new.

Appendix D
Graphs

In this appendix we present graphs of overhead introduced by each tool. The first half of the graphs show the overhead in median execution time, and the last half show the overhead in memory consumption.

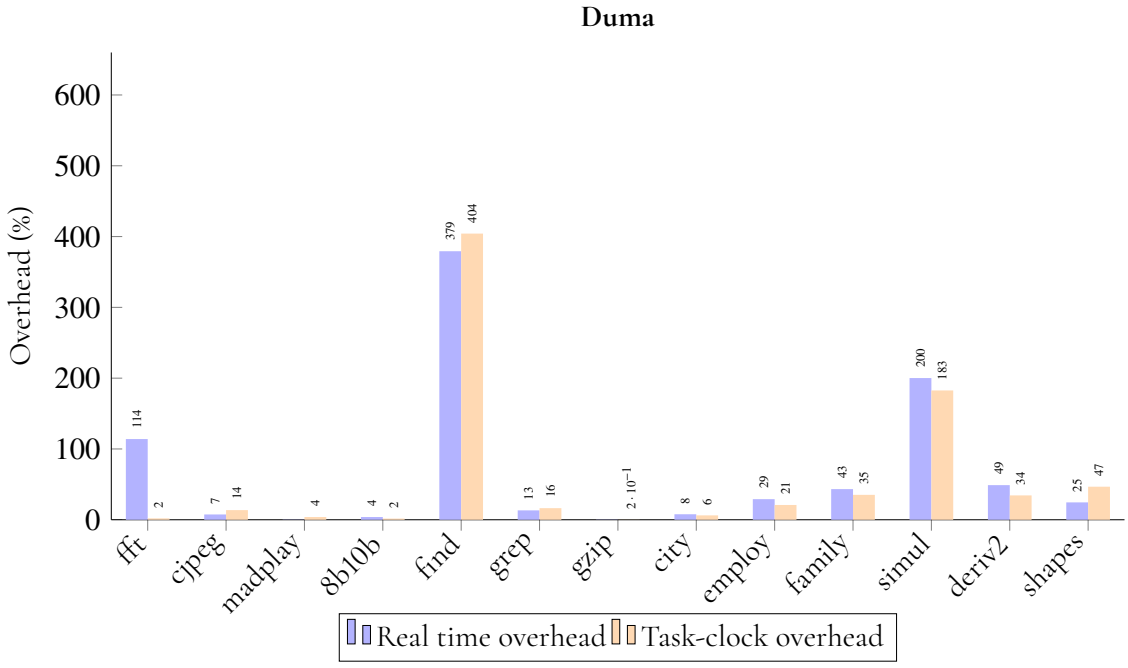


Figure D.1: Median execution time overhead of Duma

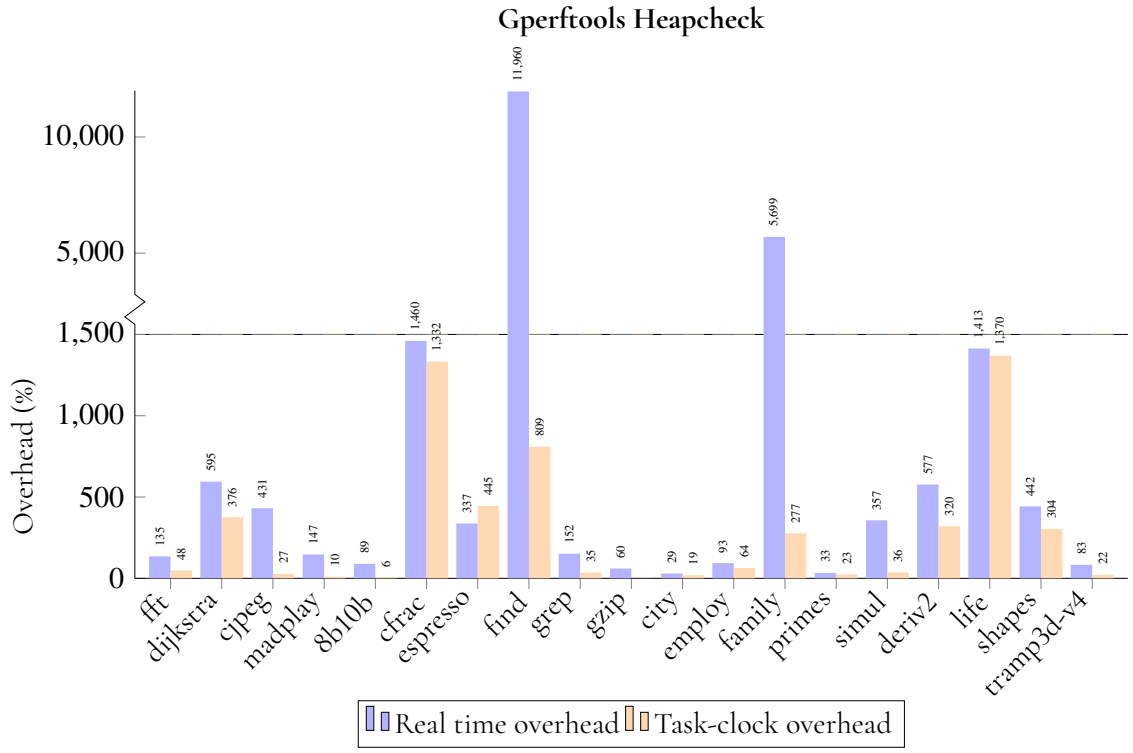


Figure D.2: Median execution time overhead of Gperftools Heapcheck

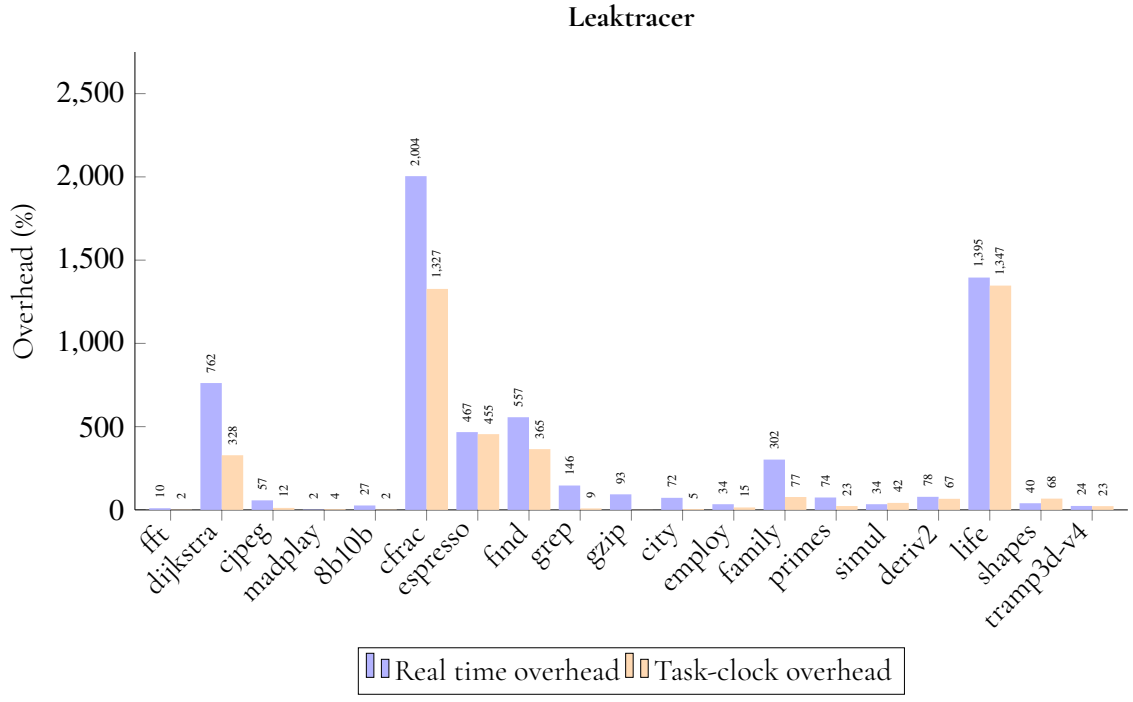


Figure D.3: Median execution time overhead of Leaktracer

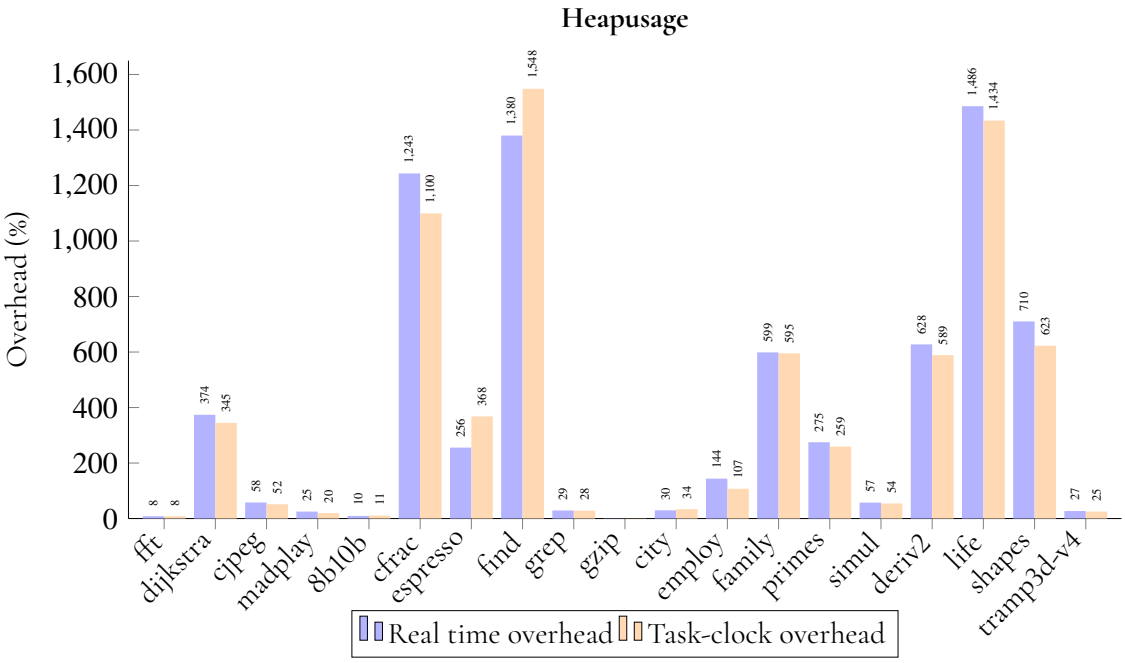


Figure D.4: Median execution time overhead of Heapusage

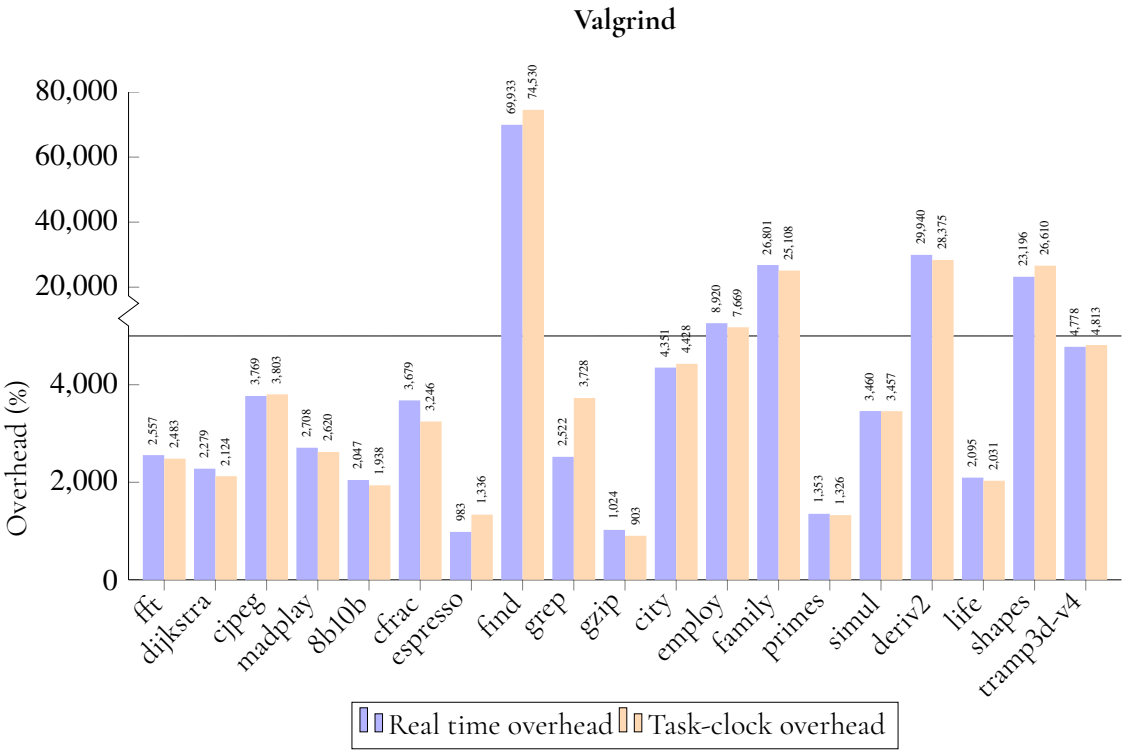


Figure D.5: Median execution time overhead of Valgrind

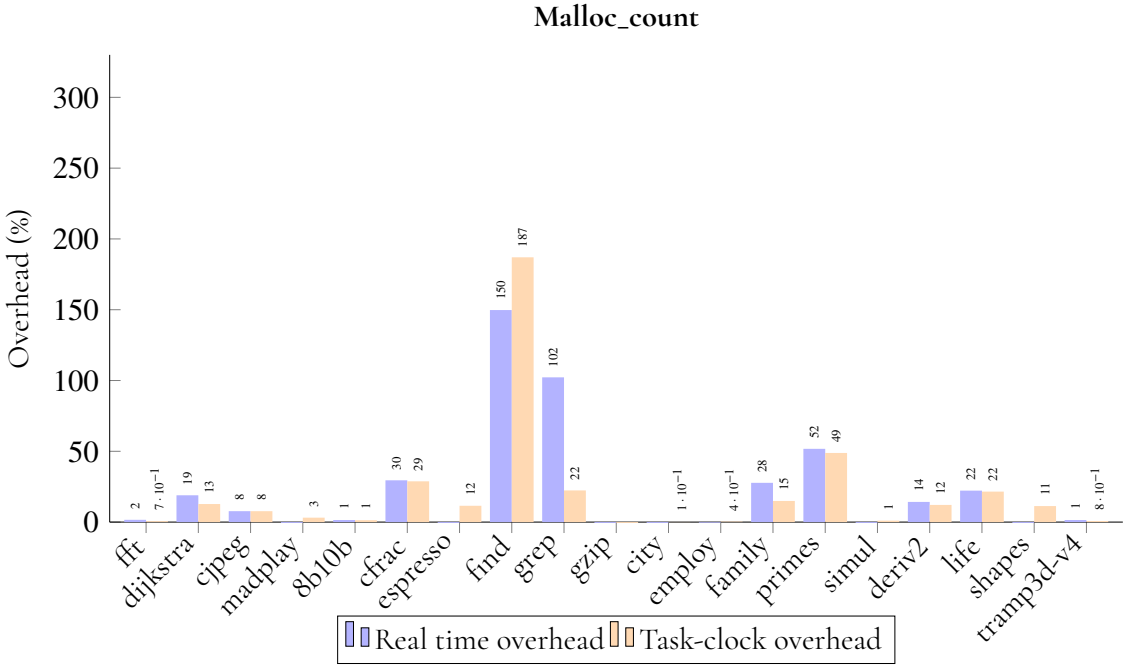


Figure D.6: Median execution time overhead of Malloc_count

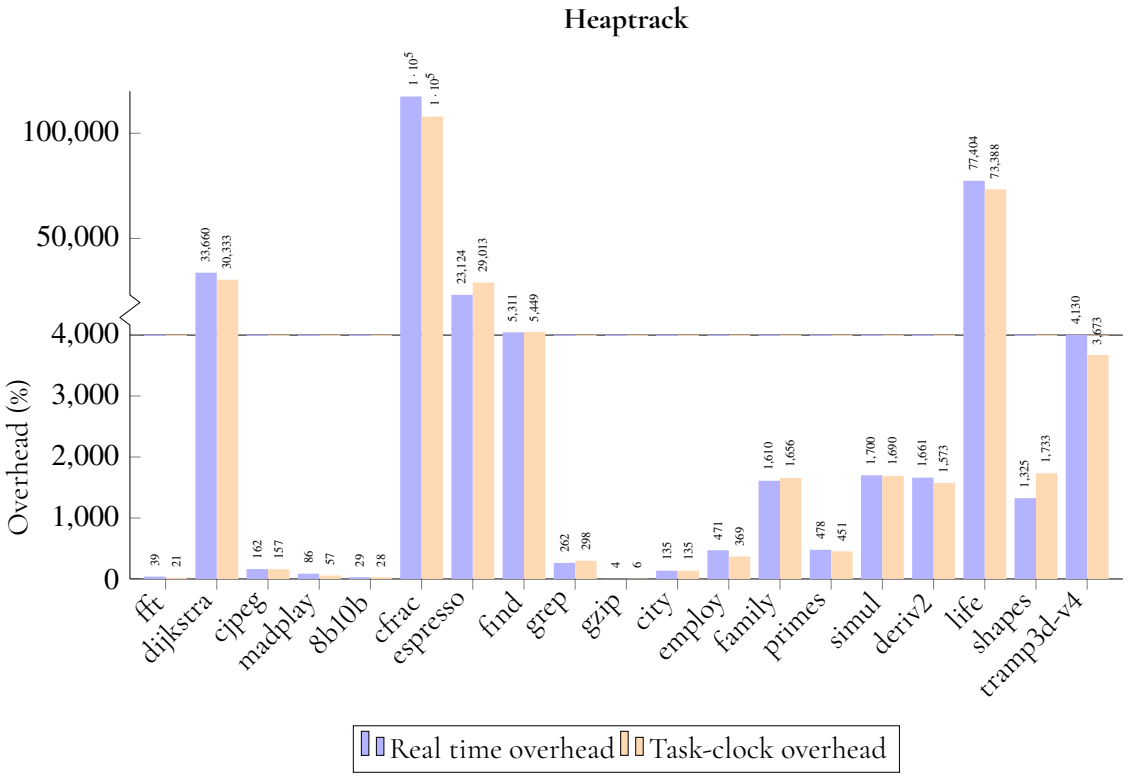


Figure D.7: Median execution time overhead of Heaptrack

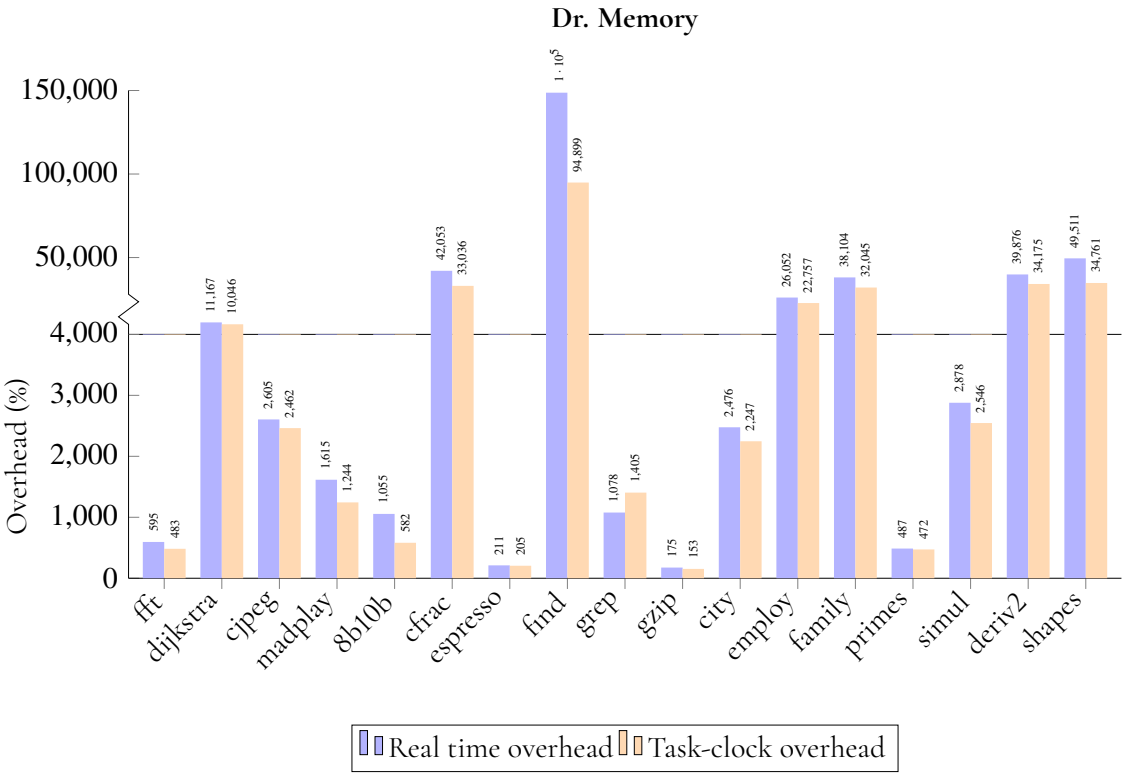


Figure D.8: Median execution time overhead of Dr. Memory

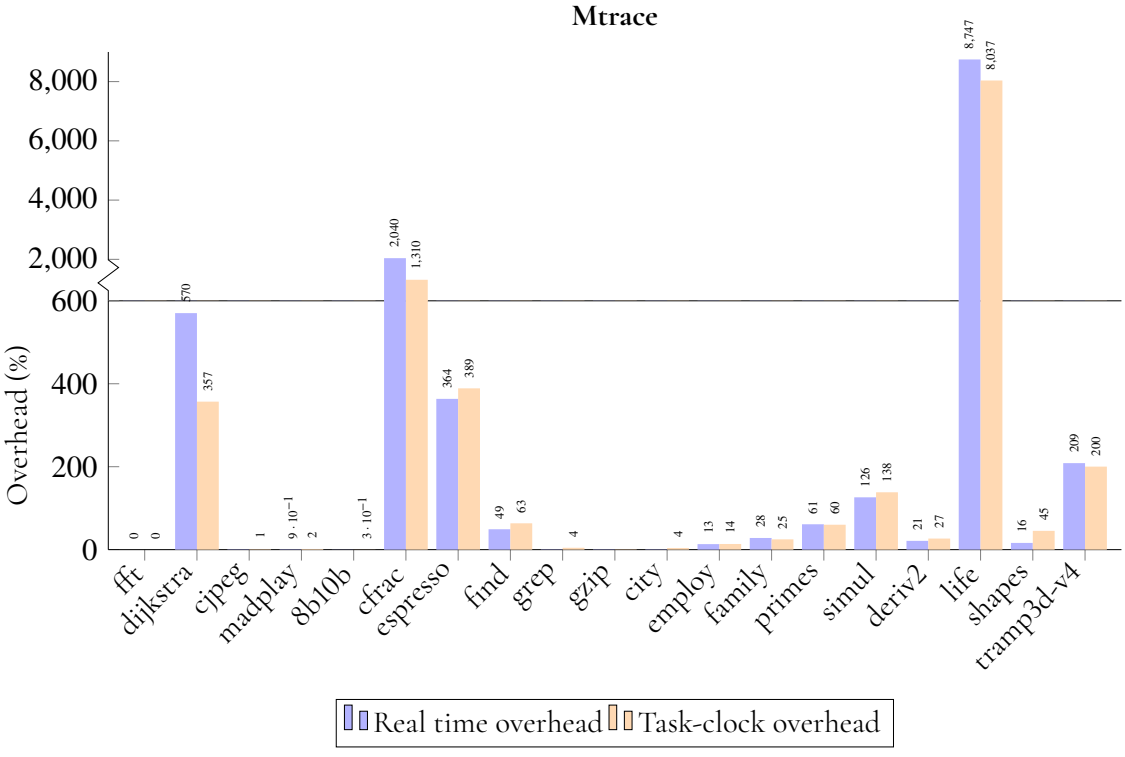


Figure D.9: Median execution time overhead of Mtrace

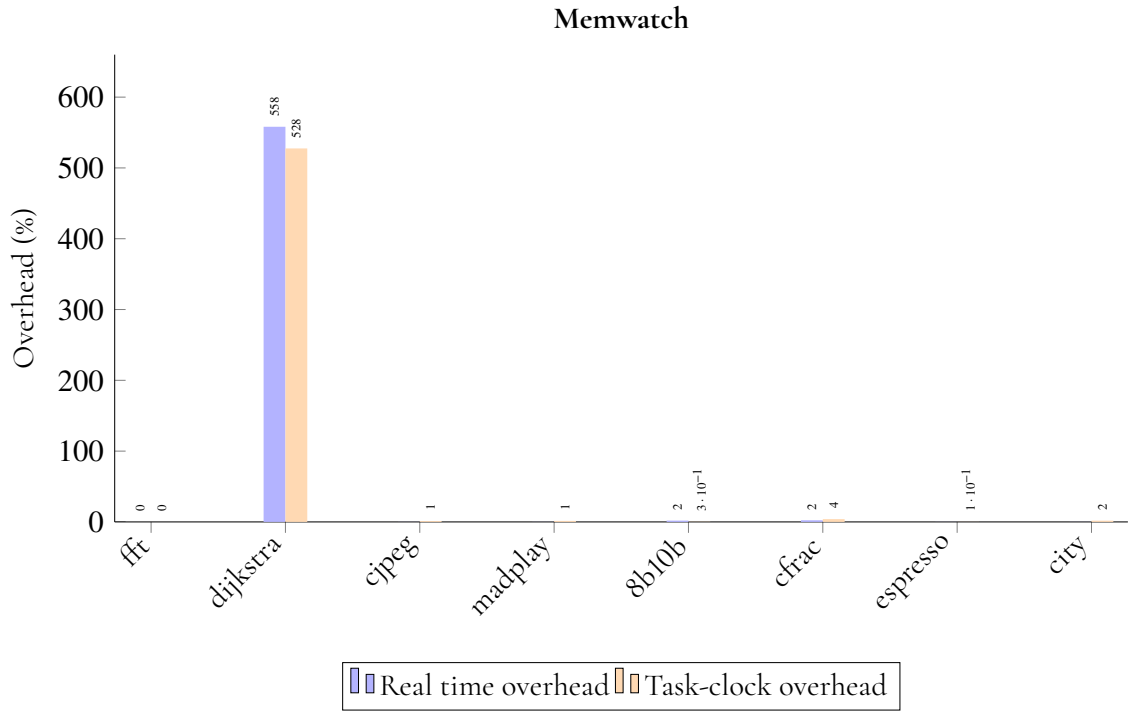


Figure D.10: Median execution time overhead of Memwatch

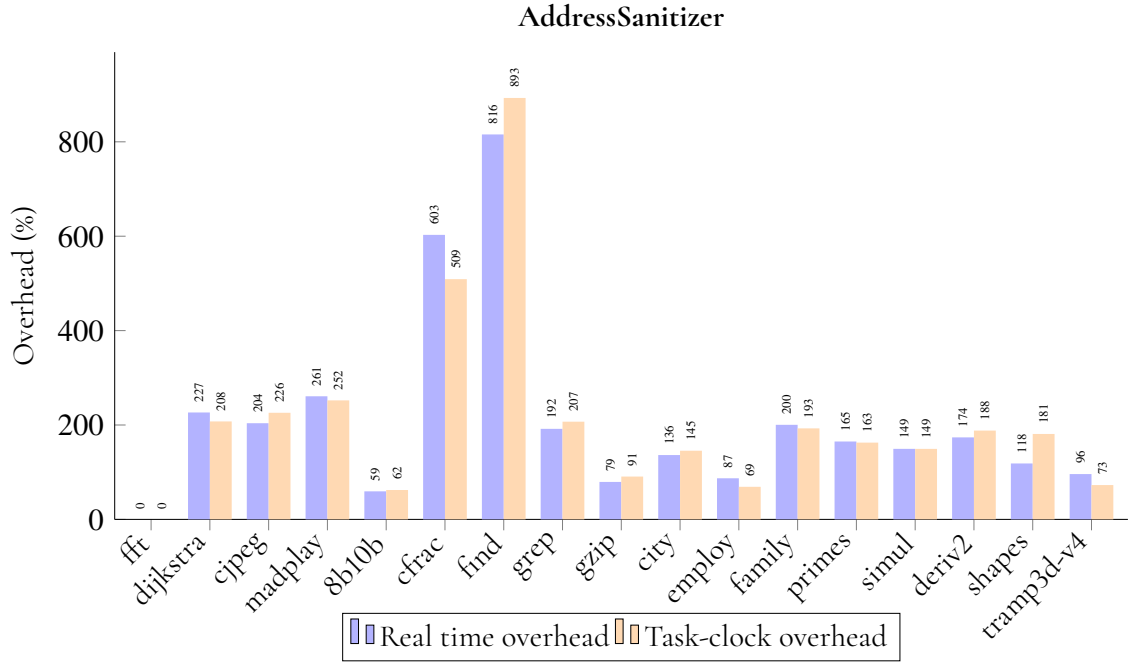


Figure D.11: Median execution time overhead of AddressSanitizer

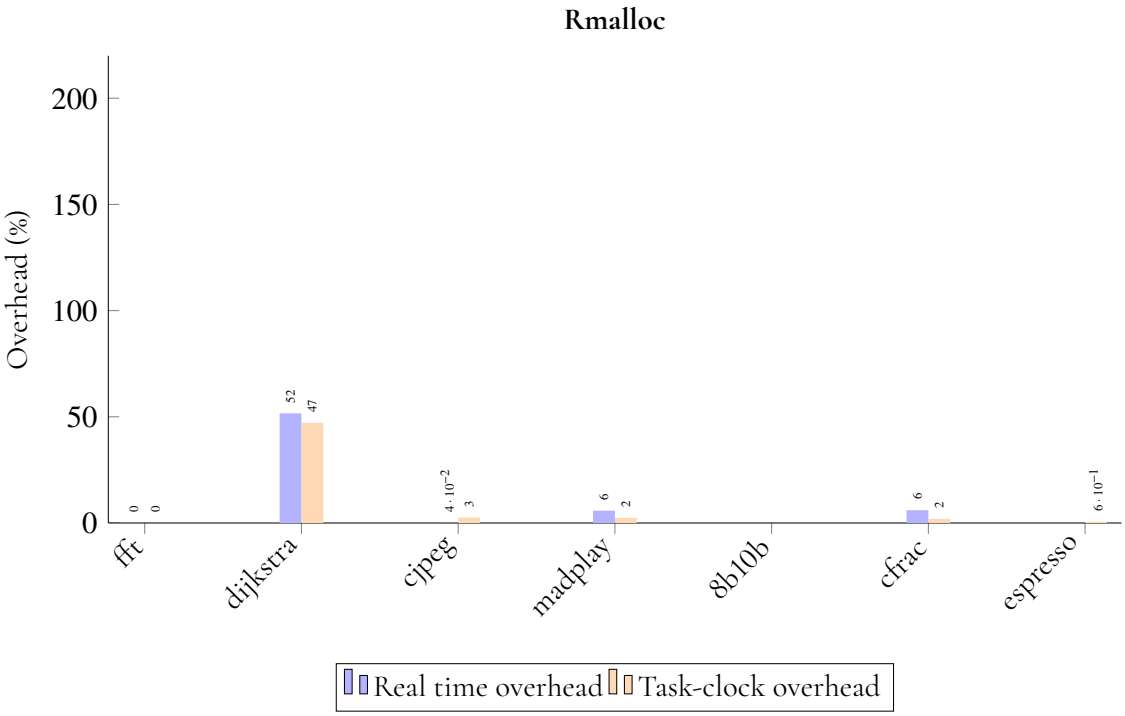


Figure D.12: Median execution time overhead of Rmalloc

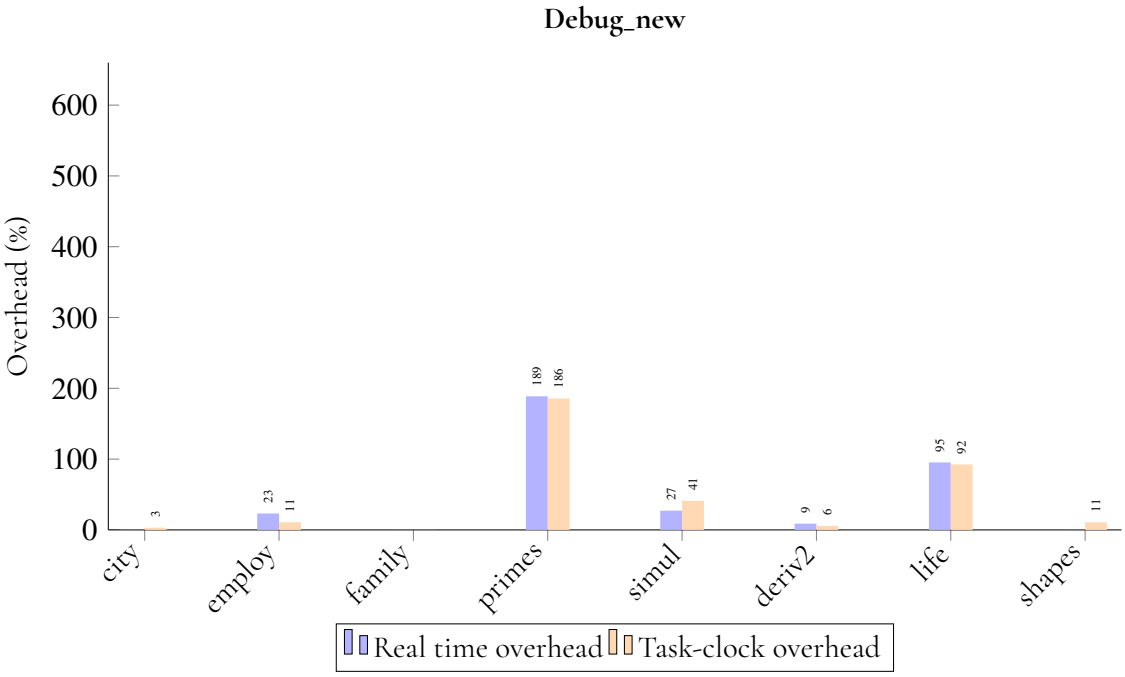


Figure D.13: Median execution time overhead of Debug_new

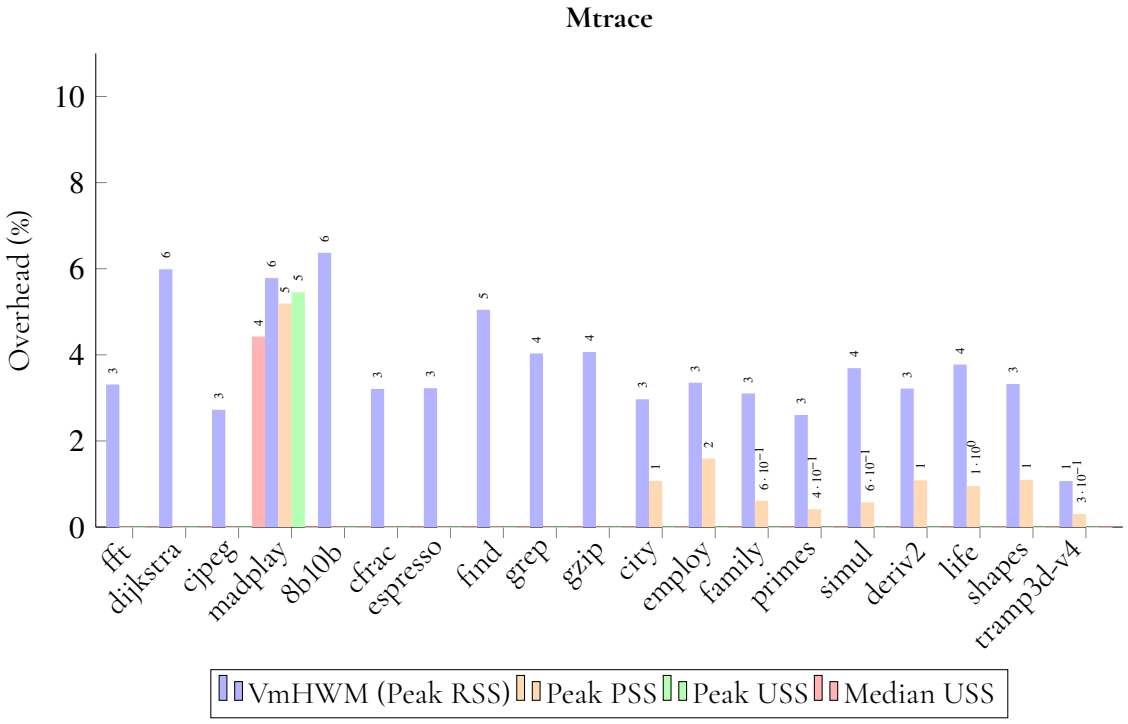


Figure D.14: Memory overhead of Mtrace

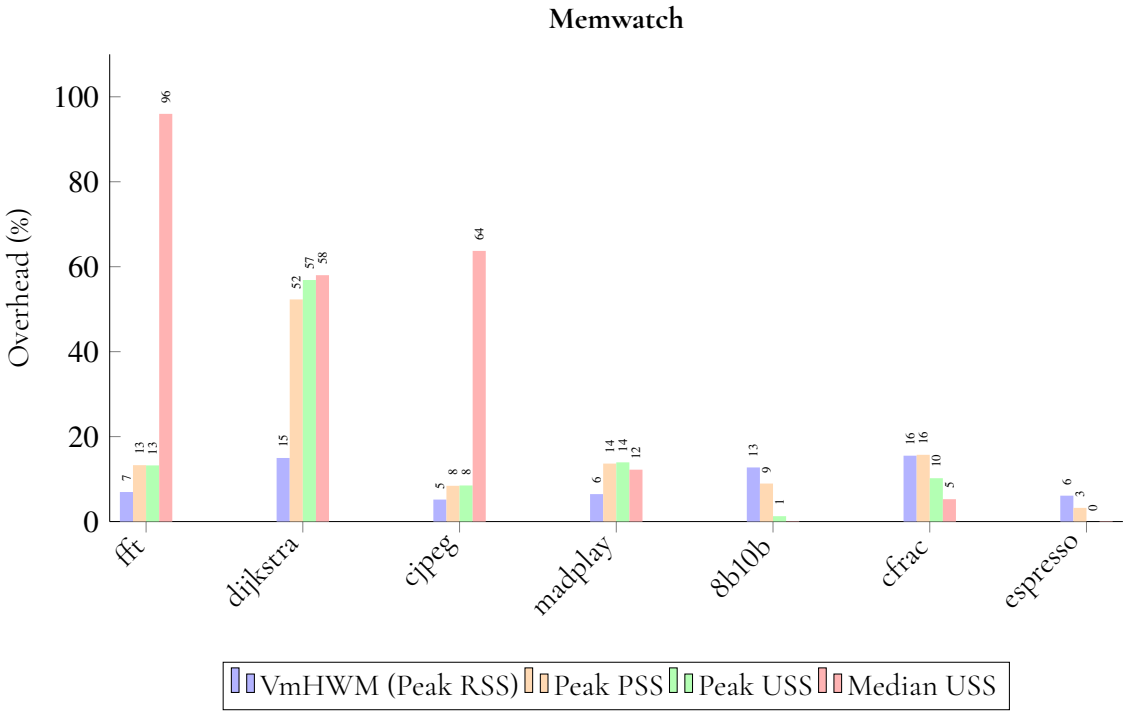


Figure D.15: Memory overhead of Memwatch

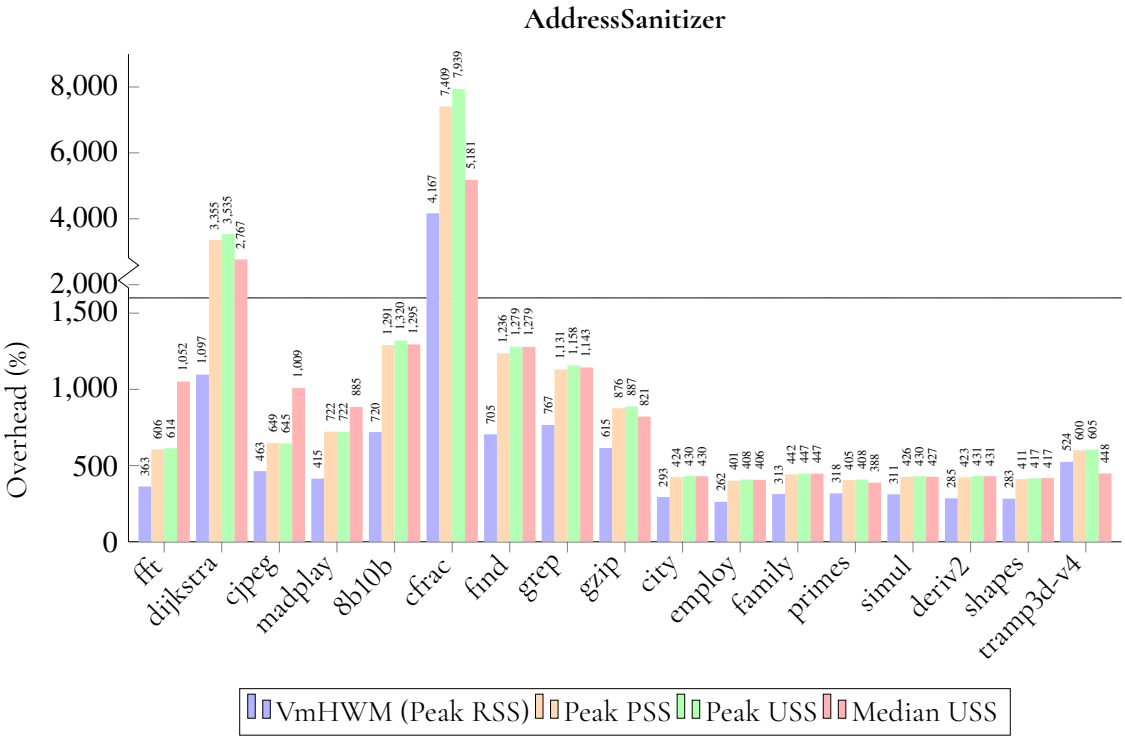


Figure D.16: Memory overhead of AddressSanitizer

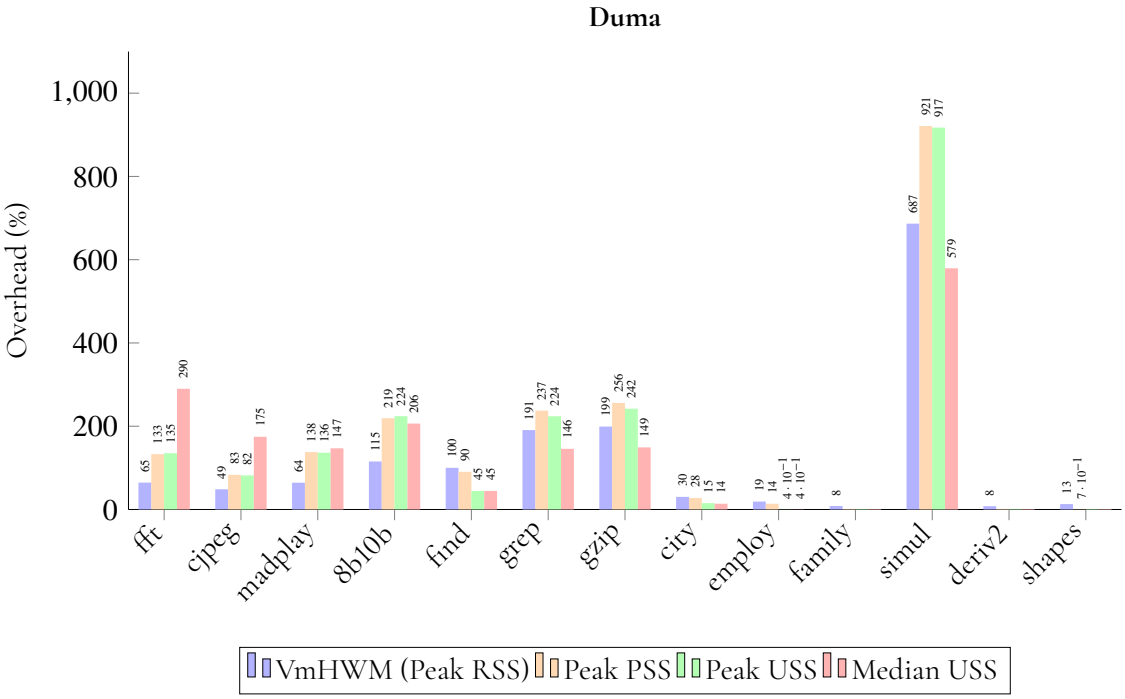


Figure D.17: Memory overhead of Duma

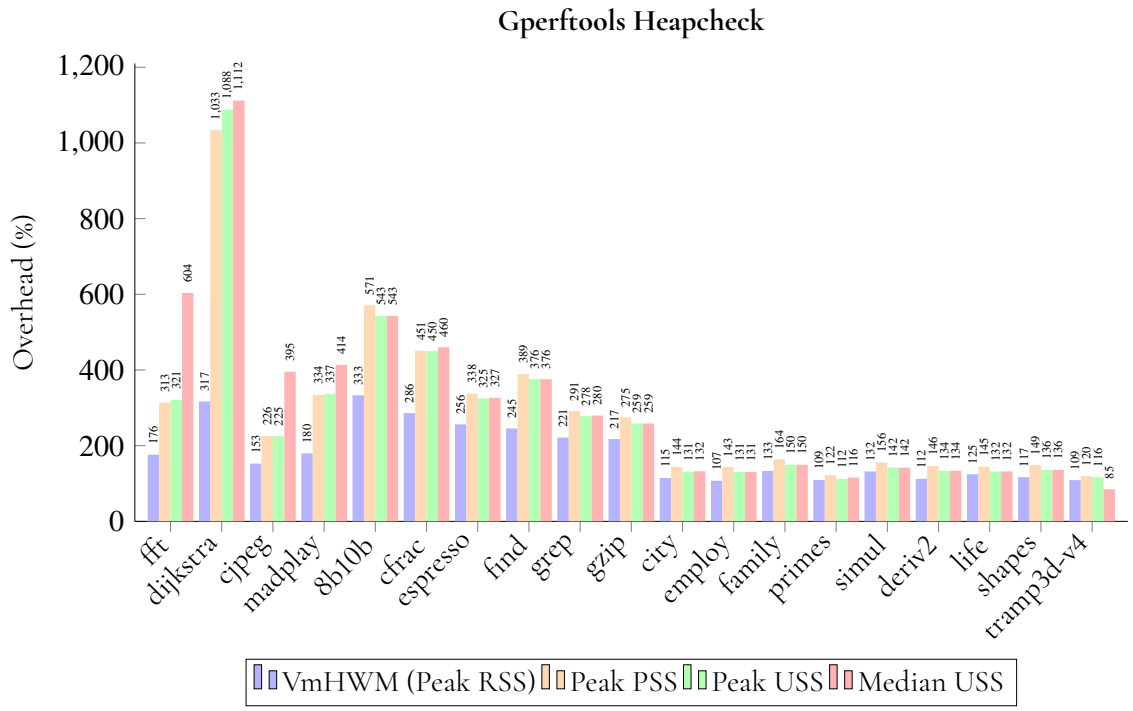


Figure D.18: Memory overhead of Gperftools Heapcheck

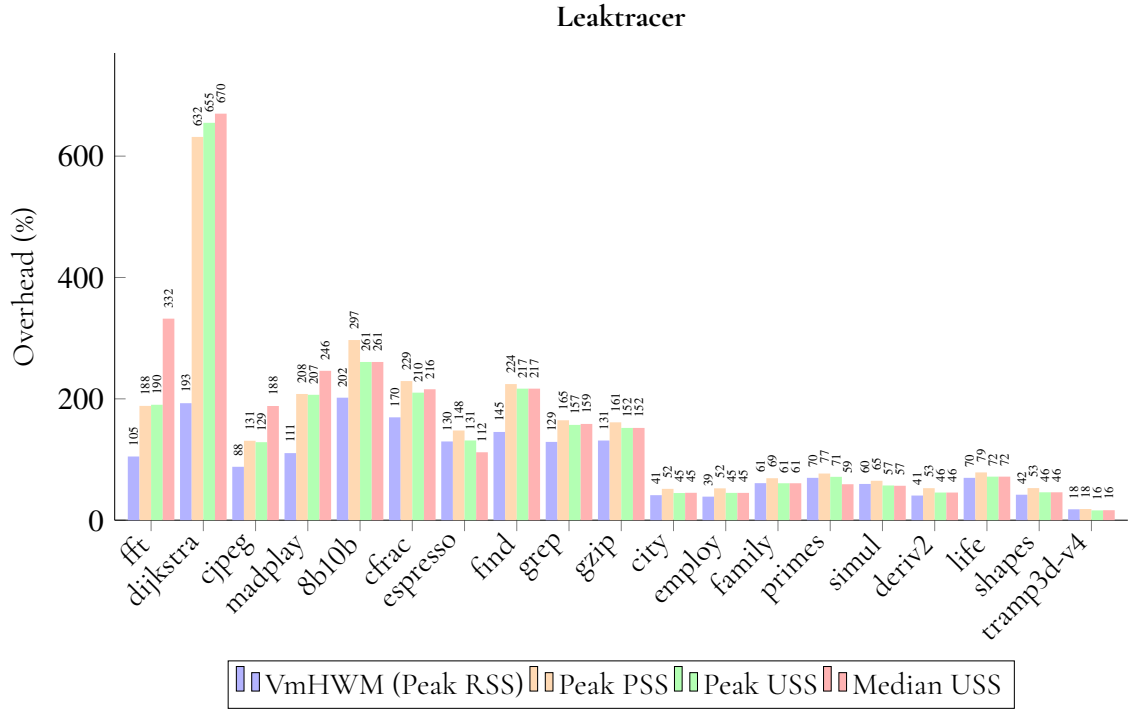


Figure D.19: Memory overhead of Leaktracer

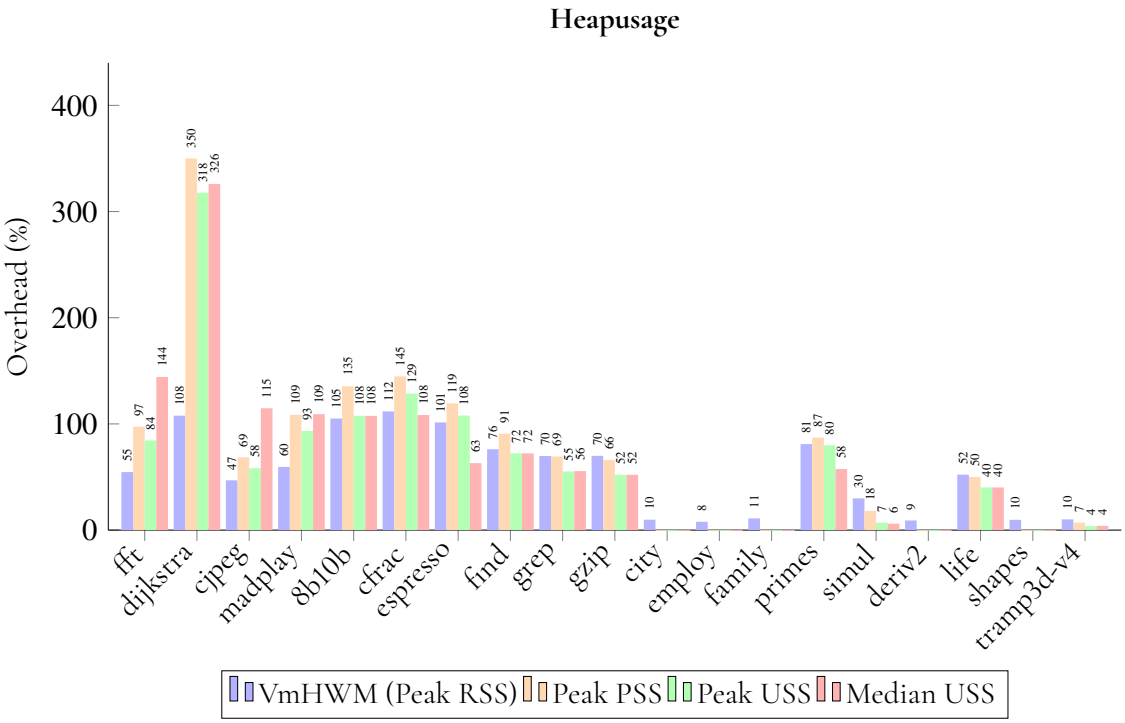


Figure D.20: Memory overhead of Heapusage

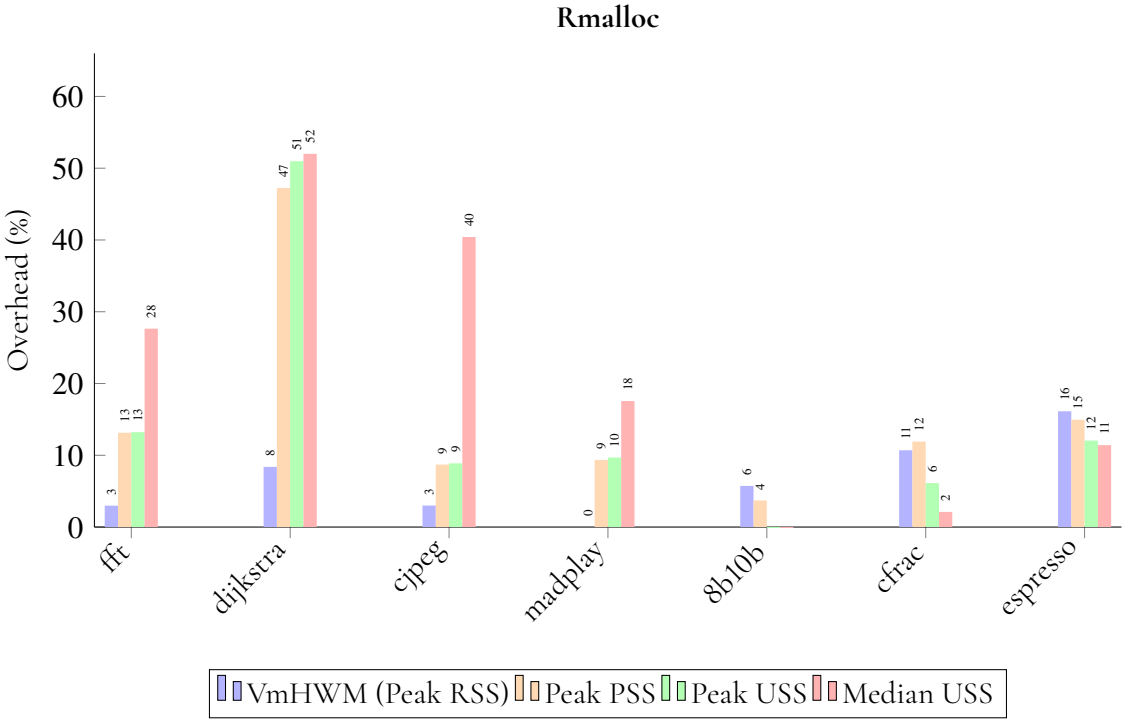


Figure D.21: Memory overhead of Rmalloc

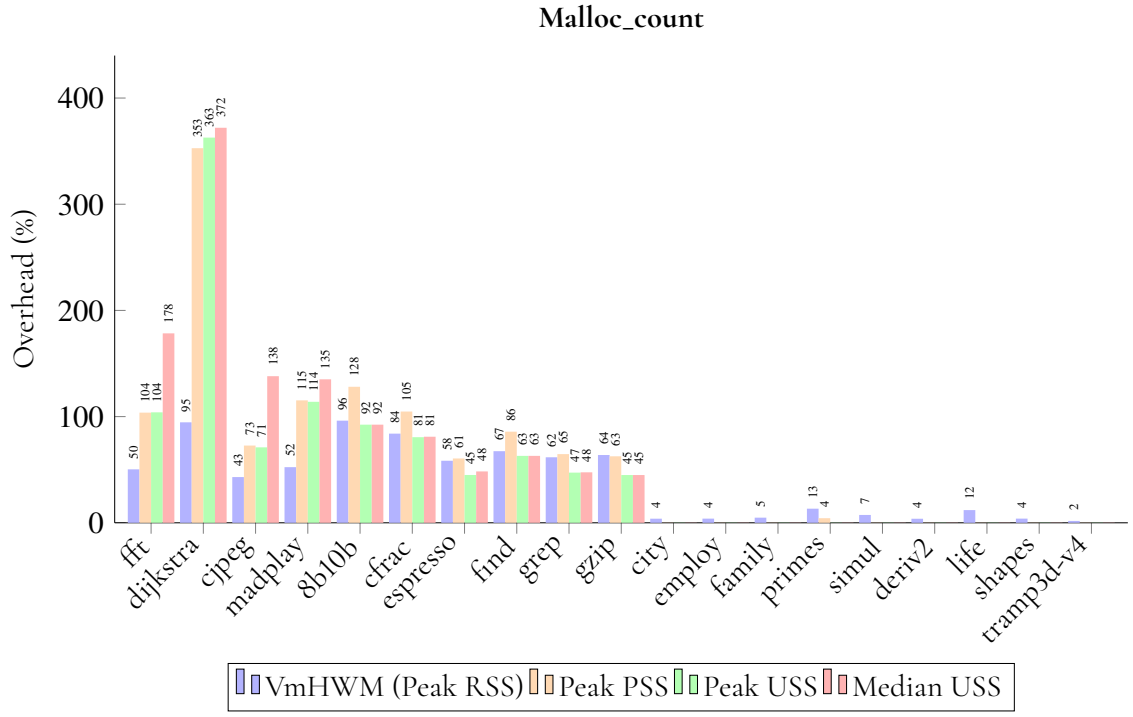


Figure D.22: Memory overhead of Malloc_count

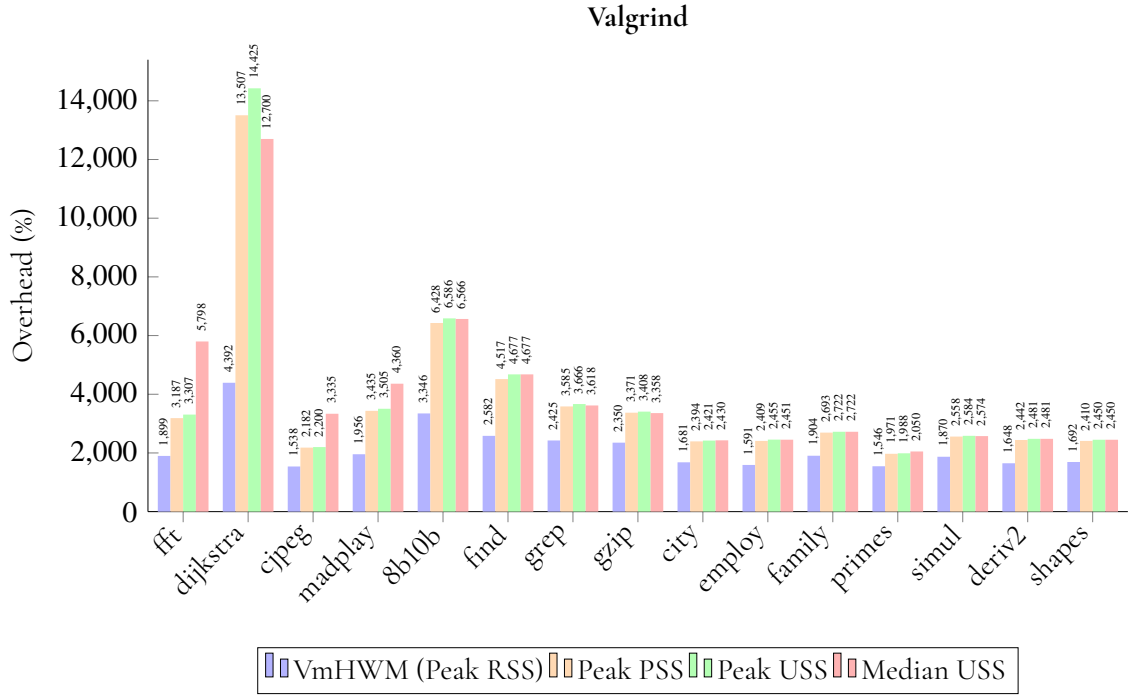


Figure D.23: Memory overhead of Valgrind

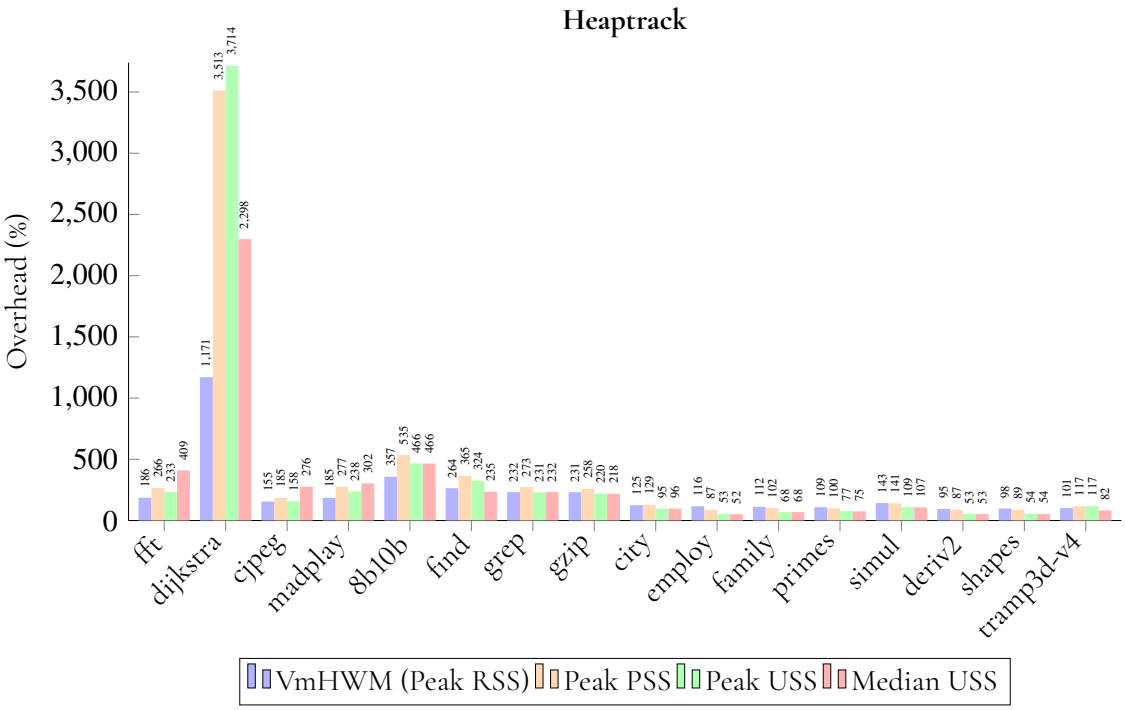


Figure D.24: Memory overhead of Heaptrack

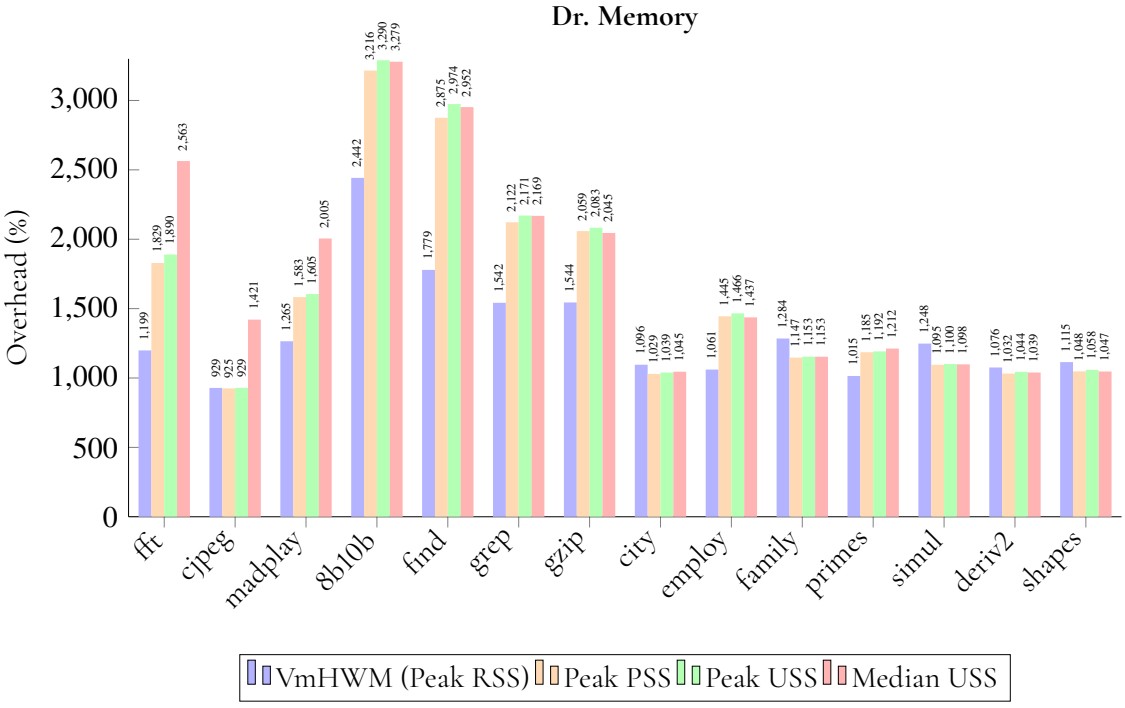


Figure D.25: Memory overhead of Dr. Memory

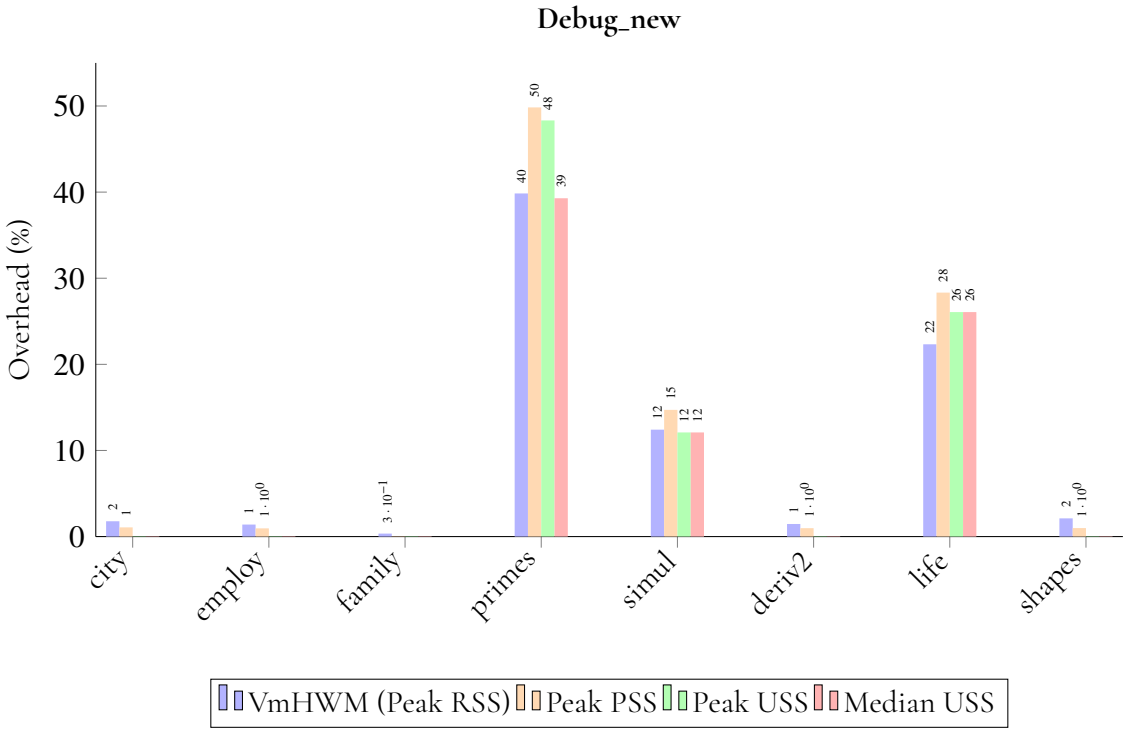


Figure D.26: Memory overhead of Debug_new

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-06-15

EXAMENSARBETE Detecting Memory Errors in a Constrained Embedded Linux System
STUDENTER Fredrik Nyberg, Emil Bengtsson
HANDLEDARE Flavius Gruian (LTH), Paul Asterland (Volvo Cars)
EXAMINATOR Jonas Skeppstedt (LTH)

Att upptäcka minnesfel i ett inbyggt Linux-system med begränsade resurser

POPULÄRVETENSKAPLIG SAMMANFATTNING **Fredrik Nyberg, Emil Bengtsson**

I en allt mer elektronisk värld, är det viktigt att inbyggda system fungerar som de ska. Vi har undersökt olika verktyg och metoder som kan vara till hjälp för att eliminera minnesfel i program som kör på sådana system.

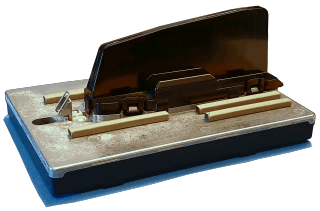
Det sitter inbyggda system i många produkter vi använder dagligen. Ett inbyggt system är en liten dator som är inbyggd i en större apparat eller maskin, som en mikrovågsugn eller ett flygplan. Det inbyggda systemet kontrollerar ofta någon viktig funktion, till exempel autopiloten i flygplanet.

Inbyggda system är ofta gjorda för att vara igång under långa perioder utan avbrott, och det är viktigt att de inte kraschar eller beter sig felaktigt. Något som kan få ett inbyggt system att fungera dåligt är minnesfel i dess programvara. Minnesfel uppstår när ett program brister i sin hantering eller användning av minne. Ett exempel på minnesfel är att programmet inte ger tillbaka oanvänt minne till operativsystemet. Den typen av fel är extra viktiga att hitta i program som kör på inbyggda system, där man inte kan slösa med resurser.

För att hitta minnesfel kan man använda dynamiska analysverktyg. Dessa verktyg analyserar program under körning. Det innebär att verktygen måste kunna användas på samma system som programmen man vill analysera.

Att kunna använda verktyg för att hitta minnesfel kan vara till stor hjälp när man skriver ett program. Men om man skriver program för inbyggda system kan man inte alltid använda vissa dynamiska verktyg, på grund av att de är för resurskrävande. En utvärdering av vilka dynamiska verktyg som fungerar även med begränsad resurstillgång är därför intressant, och kan vara till nytta för alla som skriver program för inbyggda Linux-system med begränsade resurser.

Vi har undersökt olika dynamiska verktyg för att se vilka som kan användas på ett inbyggt system med sådana resursbegränsningar. Vi har utfört vårt arbete åt Volvo Cars, och vi har utvärderat verktygen på ett av deras inbyggda system. Systemet är en antenn-modul som sitter i taket på deras bilar, den syns på bilden nedan.



Vårt slutgiltiga resultat är ett förslag på verktyg som kan användas på Volvos system. Under utvärderingen märkte vi att de verktygen som kunde hitta flest typer av minnesfel krävde mest minne och var långsammast. Inget av dessa mer allsidiga verktygen kunde användas för att analysera Volvos program, eftersom de var för resurskrävande. Vi upptäckte att en kombination av mer specialiserade verktyg som hittar olika sorters minnesfel fungerade bättre, eftersom de behövde mindre resurser var för sig, och kunde användas i separata analyser. Vårt förslag består därför av tre sådana verktyg.

Vårt arbete innehåller förslag på lämpliga verktyg och metoder med låg overhead, vilket kan hjälpa både de som vill använda ett befintligt verktyg och de som vill utveckla ett nytt.