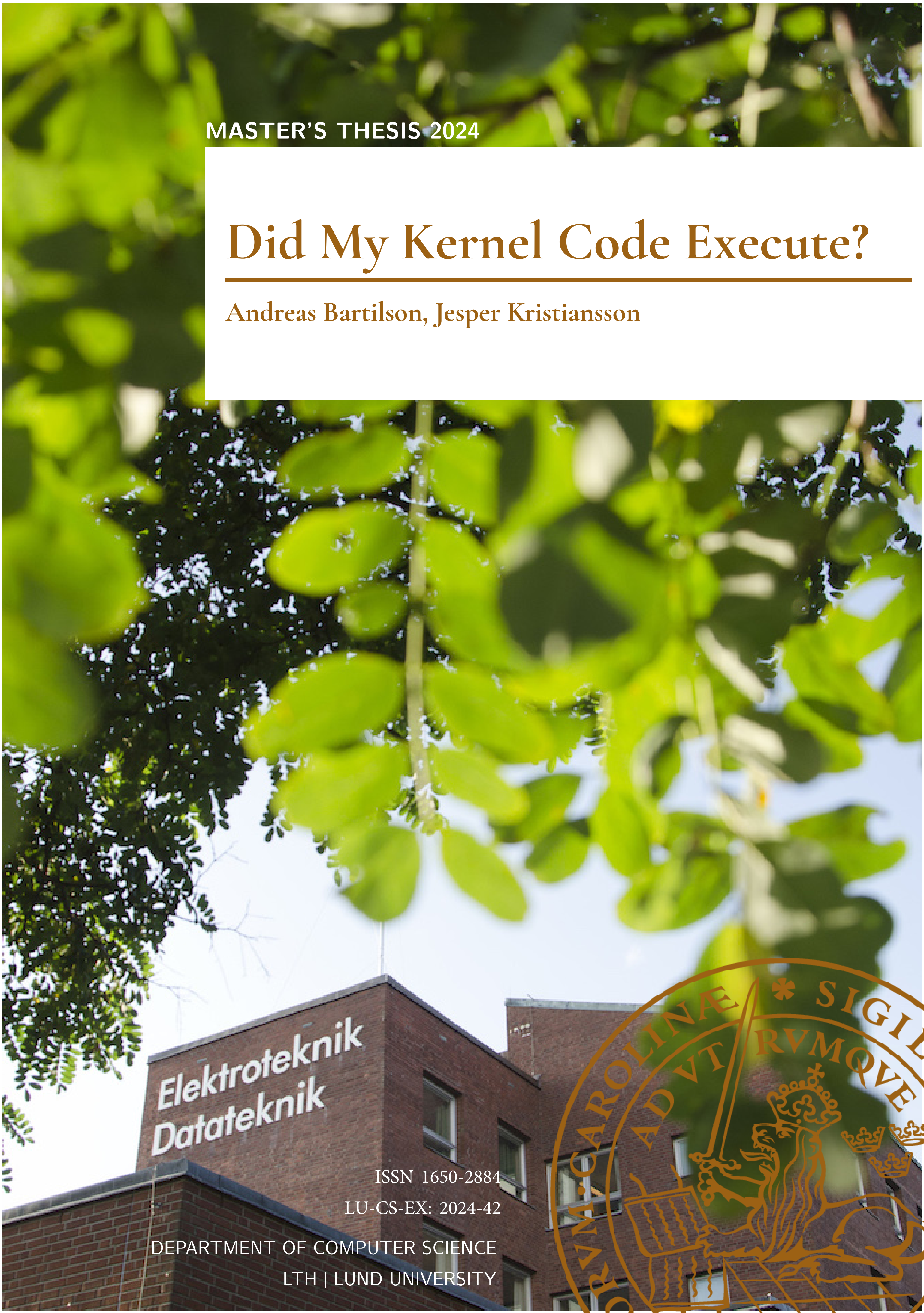


MASTER'S THESIS 2024

Did My Kernel Code Execute?

Andreas Bartilson, Jesper Kristiansson

ISSN 1650-2884
LU-CS-EX: 2024-42

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-42

Did My Kernel Code Execute?

Exekverade min kod i kärnan?

Andreas Bartilson, Jesper Kristiansson

Did My Kernel Code Execute?

Andreas Bartilson
an0257ba-s@student.lu.se

Jesper Kristiansson
je7742kr-s@student.lu.se

June 25, 2024

Master's thesis work carried out at Ericsson.

Supervisors: Patrik Åberg, patrik.aberg@ericsson.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Analysis and visibility of complex software systems is becoming increasingly more important and at the same time difficult to achieve. Comprehensive and extensive analyses of the popular operating system Linux can consequently prove invaluable. There are currently several options to choose from when analysing the Linux kernel, but all have one common denominator; analyses are based on events, system calls or function call stacks.

With this thesis, we propose using a source code level probe-based approach by adapting the Ericsson-developed tool *Did My Code Execute?* for kernel space use, to gain a new way of performing analyses on the kernel. We show that it is possible to use such an approach and provide a framework of considerations to make when writing kernel space probes and how to extract the collected data in an emulated environment. Additionally, we show that it is possible to generate a source code level trace of the kernel and show what performance implications probing different parts of the Linux kernel has.

Keywords: Linux Kernel, Did My Code Execute, Program analysis, Instrumentation

Acknowledgements

We want to thank our supervisor at Ericsson, Patrik Åberg, and Ericsson themselves for the opportunity to dive into such a fascinating topic as well as for all the guidance. Additionally, we would like to thank the service staff at Ericsson for providing amenities such as coffee, tea and fresh fruit every day.

We would also like to thank Jonas Skeppstedt for his many interesting courses and for his support and supervision during our thesis. We also thank Flavius Gruian for taking the time to be our examiner.

Further, we thank the other M.Sc. students also working on their theses at Ericsson for the nice company during lunches and fika-pauses.

Contents

Chapter 1
Introduction

An important aspect of software development is analysis and visibility. As the complexity of computer systems increases, so does the importance of comprehensive analyses, which simultaneously, becomes more challenging to achieve. This is especially true for an operating system such as Linux. Luckily, there are several powerful tools [3, 15, 23, 28, 29] capable of comprehensive analyses.

Unfortunately, these tools mostly let the user listen to events and system calls or trace the call stack. While useful for debugging and finding potential security flaws, they omit some detail, as they are unable to analyse specific lines of code. This is where *Did My Code Execute?* [30] (DMCE) might offer some respite. DMCE use static analysis to insert user-defined software probes on every valid C/C++ expression, which could then be used to collect and output important information about how the program runs and behaves.

1.1 Research questions

To tell whether a probe-based approach is a viable method of profiling the Linux kernel, we aim to answer the following research questions by experimenting and implementing kernel-adapted probes for common analysis use cases.

- **RQ 1:** *To what extent is it possible to statically instrument the kernel code with probes using DMCE? How can problems encountered with the probe analyser/writer be handled?*
- **RQ 2:** *How can we design and write probes for common analysis use cases, e.g. heatmap and trace, in kernel space?*
- **RQ 3:** *Can DMCE's probe-based approach be used to trace the Linux kernel with reasonable accuracy? How can we handle run-time performance issues caused by the overhead introduced by the instrumentation?*

1.2 Contribution

The results of this thesis are meant to provide a new perspective on what analysis can be made when profiling and evaluating the performance of the Linux kernel. More specifically, we provide a framework as an extension of the tool *Did My Code Execute?* (DMCE) to create probes capable of running in the Linux kernel itself, along with a couple of example probes for common analysis use cases. The framework and provided probes (see appendix C) may be further expanded upon or used independently, or alongside existing profiling tools, to gain a deeper understanding of how we can perform analysis on operating systems such as Linux.

DMCE is a tool used and developed by Ericsson, built for instrumenting C/C++ code bases using a probe-based approach. By default, DMCE provides probes e.g. creating a trace (including the multi-core case) of a program or a heatmap of what parts of the code run the most. It offers great scaling with large systems along with source-code level accuracy, providing invaluable feedback about a program's behaviour, what parts may cause issues and where there is room for improvement. The result of our work has provided feedback and minor patches to DMCE which made its way into version 2.0.0, released in March of this year.

1.3 Related work

Linux is perhaps the most widely used operating system in today's industry, and thus, profiling and analysing the kernel is not a new endeavour, but an ongoing and important one nonetheless. Being able to efficiently track down potential flaws that may pose security risks is invaluable. Hung (2023) [16] shows that their runtime fuzzer can effectively be used to expose bugs and vulnerabilities in the Linux kernel. On the topic of fuzzing, Li et. al. (2023) [22] present a way of exposing logical bugs using the *eBPF* [12] *Verifier* with promising results. Developing techniques like Hung's, Ryan et al.'s (2023) [26] probabilistic lockset analysis to detect race conditions and Chen et al.'s (2023) [3] AI approach to analysing execution paths generated with *ftrace* (built-in tool for tracing function calls within the kernel), is therefore an important step for developing safer systems. Furthermore, Waly & Ktari [29] presents a complete framework for a declarative scripting language to work with event-tracing and recognising patterns in execution more easily.

Among the mentioned tools, several other alternatives to profiling the kernel also exist, including *LTTng* [23], *SystemTap* [28], *DTrace* [15] and *eBPF*. LTTng offers a simple command line interface to enable the recording of all or specified kernel events and system calls, while both SystemTap and DTrace provide a scripting-based tool to insert user-defined probes that may execute on specified events and system calls. The tool eBPF works similarly to SystemTap but makes use of the *bpf* system and its own compiler instead of a kernel module [11, 25]. This allows eBPF programs to run inside the kernel safely without a need to modify the kernel's source code. An advantage of using eBPF over SystemTap is that eBPF has access to more trace points than SystemTap [11].

The relation to our work with DMCE is the level of abstraction the different tools operate on. Most tools, including eBPF and SystemTap, execute alongside, within the kernel, i.e. it does not modify the kernel itself but extends it with user-defined event listeners on predefined trace-points and hooks placed by kernel developers [11]. This means that kernel developers are responsible for the quality and quantity of available points in the kernel to instrument. DMCE however, works by instrumenting the source code itself i.e. modifying it without breaking the logical flow of the code (see section 2.5 for more details). Put differently, the user-defined probes with DMCE run in the kernel on a line-of-code level, offering additional insight into the behaviour of the instrumented code and may complement information gathered with other tools.

1.4 Contribution statement

In the beginning, we worked closely together to set up environments and create a proof of concept, that is, build and run a probed kernel, collect data and extract it. When setting up data extraction we wrote our own separate kernel modules to try different ways of extracting the data. Further, we worked individually to implement different probes, i.e. Jesper worked with *trace* and Andreas with *heatmap*. When facing common probe related issues such as probe-recursion (see section 3.3.2) we combined forces to find a solution.

When gathering data for evaluation, Andreas performed the measurements related to the number of probes inserted and the number of executed probes in different configurations. Jesper measured the boot times of the kernel while probed.

As for the report, individual contributions are specified in table A.1 in appendix A, and roughly reflects the division in practical work.

Chapter 2
Background

This chapter serves to give an introduction to the tools used and important concepts for the following chapters.

2.1 The C language

C is a programming language developed in 1972 [27]. It allows low-level manipulation of the computer and serves as an abstraction of assembly, and has been widely used for programs that require high performance and control, such as operating systems and embedded systems, including the Linux kernel. Below we describe some concepts of C that are relevant to both DMCE and how the kernel is structured.

The C preprocessor handles the first step in compiling a C program [14]. Commands to the preprocessor begin with the symbol # and are called directives. There are several different directives, including macros and conditional compilation.

A macro is defined using the #define-directive, as shown below. It associates a name with some content. Later, whenever that name is used, it is replaced by the macro content. There are two types of macros that can be distinguished between, object-like macros and function-like macros. A function-like macro differs from an object-like macro in that after the macro name, there is a list of arguments that may be used in the macro [14]. The code snippet below shows the definition of an object-like macro *FOO*, and a function-like macro *BAR*.

```
#define FOO 15
#define BAR(x, y) (x + y)
```

Conditional compilation can be performed using the #if-, #ifdef- and #ifndef-directives. They are used in conjunction with the #else- and #endif-directives and take a constant expression which is evaluated by the preprocessor to determine whether a block of code is included in the compilation or not [14]. The following block of code shows conditional compilation depending on whether the macro FOO evaluates to 15 or not.

```
#if FOO == 15
/* code is compiled if FOO has the value 15 */
#else
/* code is compiled if FOO does not have the value 15*/
#endif
```

After the preprocessor has resolved directives, it passes a modified version of the source code to the next compilation phase. The rest of the compilation creates an representation of this code known as an Abstract Syntax Tree(AST), which is used to generate machine code. As a result of this, the preprocessor directives, such as macros, cannot be seen in the generated AST, only the code they result in. This has implications for modifying the source code based on the AST, which is done by DMCE, described more in section 2.5.

In addition to what is defined in the C standard, compilers can provide their own extensions to the language. This can e.g. be in the form of built-in functions, or attributes. Attributes can be used to define special properties of variables, which changes how the compiler handles them during compilation [4]. As an example, GCC provides a *section* attribute which allows the programmer to specify which section of the executable file a variable should be placed in [6]. The code fragment below shows the declaration of an integer that will be placed in the section *INITDATA*, using the GCC section attribute.

```
int foo __attribute__((section("INITDATA")));
```

2.2 The Linux kernel

The Linux kernel is the open-source project at the heart of every Linux distribution [1]. It contains the core functionality required to control the underlying hardware as well as manage things like process scheduling and system calls. The kernel code is divided into distinct subsystems, where each subsystem is responsible for one piece of functionality, such as memory management, or networking. The kernel is mainly written using the C programming language and assembly, but since kernel version 6.1, it also contains Rust, to a small degree[19]. A wide variety of computer architectures are supported by Linux[17], which is made possible by the kernel being highly configurable at build time. Using the open source tool *cloc* [10] shows that the 6.8.0-rc1 version of the kernel contains 25.7 million lines of code, of which 17.9 million, almost 70%, is code for drivers.

The kernel runs in a much different environment than user space programs. It is responsible for managing the system's resources and has unrestricted access to all hardware. A user space program on the other hand runs on top of the kernel and may rely on a lot of functionality provided by the kernel via e.g. system calls or a standard library provided by the programming language. Thus, the kernel do not have access to these as itself is the provider of that functionality. Put differently, the kernel must provide all of its own functionality.

It is very important that the generated machine code does exactly what the programmer intends it to do, thus the compiler plays a very important part in making the kernel work. The Linux kernel uses the GNU Compiler Collection, GCC, and is very closely tied to it, as it uses several compiler extensions, such as statement expressions and specifying attributes for symbols.

2.3 Quick Emulator

For our work, we have needed to build and run many kernel images. To reduce the time per iteration cycle we have used the emulator called Quick Emulator [24], or simply QEMU, which is a generic open-source software capable of virtualisation and emulating a processor. It supports several host architectures and an even wider variety of target architectures, such as Arm, x86, and Power. It achieves emulation with good performance through dynamic translation, in which it translates a piece of code in the target binary only the first time it is encountered.

QEMU has two separate modes in which it can run, system and user mode emulation. During system emulation, an entire machine is emulated, including CPU, memory, and peripheral devices. This mode is able to run programs which are designed to run on bare metal, such as an operating system. User mode emulation is available for Linux and BSD and emulates only a CPU, which allows running a program compiled for one CPU on another CPU.

During system emulation, QEMU is capable of booting a Linux kernel without needing to create a full bootable image, by using the *-kernel* option. This makes it a very useful tool for testing during kernel development as it reduces the time per iteration cycle, i.e. instead of loading the kernel to hardware, we can run it directly in QEMU.

QEMU has a monitor, a mode that can be switched to during emulation, which provides commands for manipulating the virtual machine and its environment. Some examples are resetting the system, starting a GDB server, and dumping the virtual memory to a file.

2.4 Program analysis

Program analysis is the process of analysing software to extract information regarding the properties of the program [13]. Program analysis can be divided into two different types, static and dynamic.

Static program analyses are performed purely by analysing the source code. During a static program analysis, it is very useful to have a good representation of the semantic structure of the target program. This is often achieved by constructing an Abstract Syntax Tree (AST) similar to when compiling the program. The AST is a tree representation of the program structure and how expressions and symbols relate to each other. A common application of static program analysis is to detect potential issues or redundancies when writing code. Another, and perhaps less seen, application can be found in optimising compilers. Conversely, dynamic analysis entails gathering and extracting information during the execution of a program.

2.5 Did My Code Execute?

Did My Code Execute? [30], or DMCE for short, is a static and dynamic program analysis tool for C/C++ programs, developed at Ericsson and open-sourced on GitHub. It uses *clang-check*, a tool from LLVM, to generate an AST and performs static program analysis on the generated AST to insert software probes on valid expressions using the C/C++ *comma operator* [14]. The comma operator is a binary operator which evaluates its operands in order and then returns the value and the type of the last operand. The probes are meant to act as instrumentation and thus not modify the logic and control flow of the original program, i.e. so long as the probe code does not intentionally produce side effects directly impacting this. Below follows an example of how a probed line may look:

```
Original:
    int foo = create_some_value_from(bar);
Instrumented:
    int foo = (DMCE_PROBE(0), create_some_value_from(bar));
```

Where *DMCE_PROBE(0)* is the call to the probe's code with probe number zero, and *create_some_value_from(bar)* a call-expression to a user-defined function returning an integer value. In this example, the probe would execute first and gather relevant information followed by the variable *foo* being assigned the result of the function call.

The probes themselves may be tailored to one's needs but have all one thing in common; they collect information of interest to create some sort of report for a given analysis use case. Probes and programs for creating and viewing reports for common analysis use cases [30] are included with DMCE and consist of e.g *heatmap*, *trace* (including multi-core trace) and *racetrack* (race-condition provocation, which does not create data in the same sense as the others). There are two steps to instrument a project or parts of a project. Firstly, one may use the *dmce-set-profile* command to set a profile, specifying which probe to use during the instrumentation. The command also offers several options to include or exclude specific constructs, variables and lines of code. This is useful since one can then adjust the scope of probing or remove instrumentation that causes e.g. compilation errors or large amounts of overhead relative to the other probed lines. Additionally, one may point out specific files or directories to put on the include-path for the analyser, in case it does not find it by itself. After setting a profile, running the command *dmce* inserts all probes. Then, it is only a matter of compiling and running the project as normal.

DMCE will create a reference file as to where in the code each probe is located and after running the instrumented code, produce a report in binary format. These two files are then used together with the appropriate DMCE tool to view the report, e.g *dmce-summary-bin* for heatmap reports and *dmce-trace-viewer* for trace probes.

During this work, we will investigate how well the existing design of DMCE works when probing the kernel, whether it can be improved, as well as creating variants of the existing probes for use in kernel space.

Chapter 3
Method

In this section we will first explain the general method used during this work, and how the research questions were answered. After this, we will describe specific problems that were discovered while using the method, and which need to be solved.

3.1 General method

In order to investigate how well DMCE can insert probes into the kernel and problems related to this, we insert probes using existing DMCE functionality and check what problems arise. To start with, the pre-existing *stub* probe is inserted, which is an empty probe. This probe is chosen as it doesn't cause any compilation problems due to the probe implementation. This allows us to find issues that are purely related to probe insertion. Any issues found this way are investigated to determine what is their cause, and how they potentially can be solved or worked around.

Another part of the work is concerned with constructing probes that work inside the kernel. The probes we wish to construct already have corresponding implementations in user space by DMCE. As a start, these implementations are examined to give an understanding of the fundamental concepts that the probes need to implement to function. After this, an iterative process is followed, where one piece of functionality is implemented at a time. This is done by reading about the relevant parts of the kernel and possibly compiler extensions, and comparing alternatives.

The last part of the work is specifically about tracing the kernel. We will measure the performance of the trace probe by measuring the time it takes to boot the kernel will using the trace probe and compare it to baseline boot time when not probed. If it is determined that the trace probe causes performance degradation, different implementations will be compared in order to determine bottlenecks and improve performance. Additionally, we will look into whether there are any methods of reducing the performance overhead outside of the probe implementation.

3.2 Probing the kernel

Some of the issues that arise are related to the static analysis of the kernel code, and how the probes should be inserted.

3.2.1 Building the kernel

A fundamental prerequisite for probing the source code of a program is to determine the structure of the program. As mentioned before, DMCE uses the tool *clang-check* to produce an AST as a representation of the program. This representation is then analysed in order to find where probes may be inserted. Generating an accurate AST is therefore very important for valid probing. The way the kernel is built causes issues when generating the AST.

Typically, a file of C source code consists of a number of include-directives for external declarations, local declarations within the file, and definitions of functions and variables. As such a file contains all the declarations it needs, it is able to be analysed independently of other source files. In the Linux kernel, there exist files which do not include declarations of external functions it uses. Instead, these files are themselves included in other files which contain the necessary declarations. This often happens for architecture specific code, as it allows subsystems to be divided up into generic code and code that needs to be provided for each supported architecture. The consequence of this is that it is impossible to generate a correct AST for this code if it is analysed independently. In such a case, clang-check generates recovery symbols, that will be ignored.

Using a macro can look identical to calling a function, so in the case where the clang-check finds ambiguity and a declaration or definition can not be found, it treats it as a function call. This can cause a probe to be inserted for a macro, which may cause issues, as described above.

A similar situation occurs when clang-check preprocesses and resolves the directives related to conditional compilation. As explained in 2.1, the conditional compilation depends on a constant expression evaluated at build time. This is commonly used in the kernel when handling configuration, as it allows unused functionality to be completely left out of compilation.

A problem might arise if clang-check is not given enough information to be able to resolve the condition correctly, either due to the issue of missing declarations described above or due to macros being defined by the compiler at build time, such as by using the *-D* option in GCC. This may cause clang-check to discard part of the AST that is in fact used during compilation, leading to fewer probes being inserted than possible. It does, however not lead to any invalid probes. This problem can be alleviated by providing clang-check with build information. This can be done by constructing a file describing the compile command used for each file of source code. Such a file can be generated using a Python script in the Linux kernel repository located, at the time of writing, at *scripts/clang-tools/gen_compile_commands.py*.

3.2.2 Probe notation

Actually inserting probes into the source code is not a trivial challenge. An inserted probe should run immediately before the expression probes, and it must not affect program flow in any way, except for execution time. DMCE solves this by using the comma operator, which requires that both operands are expressions. The comma operator is inserted so that the left operand is a call to the probe, and the right operand is an expression that was found in the AST constructed by clang-check. As mentioned before, a macro is interpreted as a function call by clang-check when the macro definition is not found, which can lead to a macro being used as an operand for the comma operator. As a macro can resolve to some arbitrary content, this is not necessarily a valid expression, which is not allowed in the comma operator. These kinds of macros are common in the kernel, and as such, cause problems when using DMCE to insert probes.

Another way of solving this, which avoids the problem of probing macros that resolve to non-expressions is using compiler-specific syntax. This makes an assumption about the build-environment of the program that is being probed, so it is not the solution for a general probing tool, however in this case we are probing the Linux kernel specifically, which is tightly coupled to using GCC, so GCC extensions is something we can rely on using. GCC provides a special syntax called statement expression [7]. A statement expression consists of a compound statement inside a parenthesis. A statement expression evaluates the compound statement as normal but additionally returns the value of the last statement if the surrounding context expects an expression. Practically, this is very similar to using the comma operator, with the difference being that if the last statement in the statement expression does not return a value, it only generates a build error if the surrounding context expects a value, unlike the comma operator which always results in an error if the right-hand side is not an expression. The code fragment below illustrates using the statement expression syntax.

```
int foo = ({DMCE_PROBE(0); create_value_from(args);});
```

Using this probe notation might increase the number of inserted probe.

3.3 Writing kernel-friendly probes

There are many considerations when designing a probe. Together with the unique environment of the Linux kernel, such as the lack of a standard library, this results in several aspects that need to be considered. Below, we talk about the most prominent issues faced and how one might work around them when adapting DMCE probes to run in kernel space.

3.3.1 The allocation consideration

In order to gather data, we obviously need somewhere to store it. Preferably, we would use a shared or global variable containing or pointing to an array of a suitable type for the information we would like to collect. This could be as simple as e.g. an array of integers but also a custom data structure for potentially more interesting use cases.

The probes available by default in DMCE (located under *probe-examples* in the repository [30]) uses either statically declared arrays (whose data is aggregated at program termination) or pointers. In the probes using pointers, the allocation is done atomically by using *mkdir* as a locking mechanism by having the first probe create a directory before allocating memory and writing a pointer to this memory as an environment variable. The following probes then detects that the directory is already created, and read this pointer from the environment variable. Thus a pointer to memory is shared between all instrumented files. This approach is not quite applicable in kernel space as environment variables do not exist. Similarly, *mkdir* does not exist in the same sense as in user space. Thus we need to make use of kernel space mechanisms to achieve the same.

There are several ways to do this, one of which makes use of the *extern* keyword in C. By declaring our probe's buffer as extern, we tell the probes to look elsewhere for a definition of the symbol. The symbol still needs to be defined somewhere, and a possible location to do this could be in an in-tree kernel module, meaning a module that is compiled together with the rest of the kernel. This is described more in section 3.4. Another way to use *extern* is with a pointer for our data buffer, to which we then allocate memory. Since this buffer will be shared between all instrumented files, it should only be allocated once. In the single-core case this is trivial, but as for the multi-core case one approach would be to use an atomic variable declared extern in the same way the buffer to make sure the allocation only happen once.

DMCE is meant to be agnostic of any build-system [30], however, since the kernel-preferred compiler is GCC we can make use of its compiler attributes for our buffer and condition variable. The attribute relevant to our problem is namely, *weak*. The *weak* attribute is typically used in libraries to mark functions as overrideable and puts the variable in global statically allocated memory. This allows us to see the same symbol everywhere and, in the ISO C standard [14], guaranteed to be zero-initialised.

Using the GCC attribute with our variables is what we ended up going forward with. The reason for this is that we then do not need to take care of atomically allocating memory and zero-initialising. Another advantage with this method is that we do not need a kernel module or any other location to initially declare our variables.

3.3.2 The recursion excursion

As mentioned in section 2.2 we rely on functionality implemented in the kernel. This brings with it one consideration; what happens if we instrument the same functionality as we are using in the probes? Recursion. There are a couple of ways to mitigate this issue, described below, as we will see the *per-kthread* approach exploiting compiler attributes is the more robust and the one we moved forward with.

To illustrate the problem, consider the simple probe seen in listing 3.1 (includes and defines are omitted for brevity) whose only purpose is to tell us when it has been executed. Examining what code executes during a call to *printk* in this way, we will eventually run into a situation like the one illustrated in figure 3.1 where execution happens recursively.

Listing 3.1: Example of a probe using printk.

```
static void dmce_probe_body(unsigned int probenbr) {
    printk("Probe␣number␣%u␣executed!", probenbr);
}
```

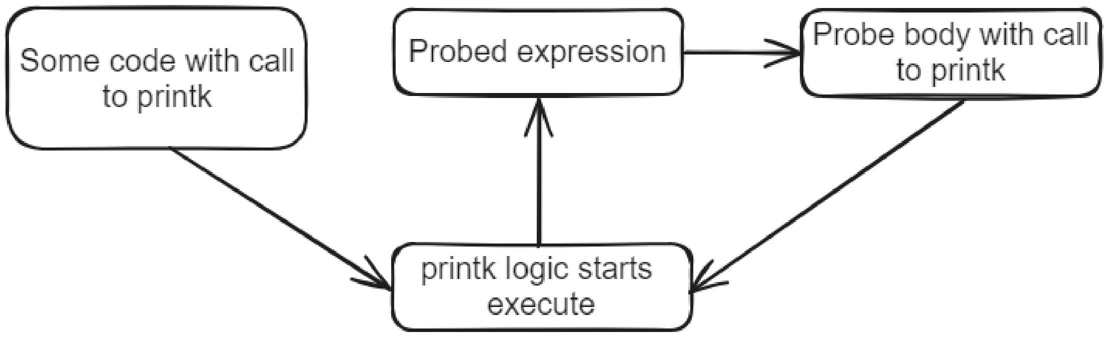

Some code with call to printk
Probed expression
Probe body with call to printk
printk logic starts execute

Figure 3.1: Illustration of logical flow when probing printk while also using printk in the probe.

One way to mitigate these situations is to introduce a static variable (i.e. it exists per-file) to keep track of when we have entered the probe body and conversely, exited (see listing 3.2). This means that we can simply perform a check whether the original probing instance is finished or not, and thus return before we reach the critical section causing recursive probe-calls.

Listing 3.2: Example of a naive recursion prevention.

```
static int is_probing = 0;
static void dmce_probe_body(unsigned int probenbr) {
    if (is_probing)
        return;
    is_probing = 1;
    // Critical section
    is_probing = 0;
}
```

However, since the guard variable exist on a per-file level, it does not take into account the case where more than one core executes code from the same file. That is, a race condition on our guard variable occurs causing one or more of the probe executions to skip its critical section, as illustrated in figure 3.2. The results of this would be incomplete or fragmented data. Similarly, the same issue would occur with just one core, as a single core may have multiple threads of execution. Put differently, if one thread gets preempted just after *"is_probing = 1"* in the aforementioned figure, all other threads will skip the critical section until the first thread is allowed to continue.

An approach that might solve these issues would be a per-kthread or per-process recursion guard. Instead of a variable that several threads of execution may tamper with simultaneously, we optimistically assume an upper bound for the maximum number of kthreads spawned during the kernel's lifetime and set that as the size of a globally shared array of integers where we use the current process id for indexing. In the example seen in listing 3.3, *current* is a macro that expands to inline assembly that returns a pointer to a *task_struct* which in turn contains the process id of the currently executing kthread. This means that if the same kthread tries to recursively enter a probe, it will check against its own, and no other kthread's, recursion guard and return before hitting the critical section of the probe.

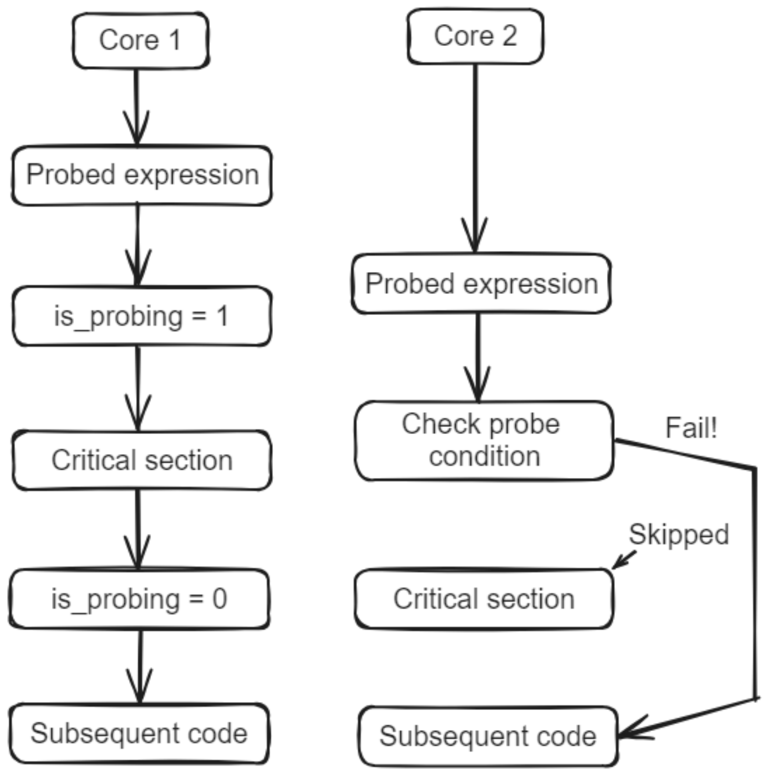


Figure 3.2: Logical flow illustrating a scenario where per-file recursion prevention fails and loses data due to a race-condition on the variable *is_probing*.

Listing 3.3: Example of a probe using a per kthread recursion prevention.

```
#define DMCE_MAX_NUM_KTHREADS 1024 * 16 // Optimistic assumption
int __attribute__((weak)) dmce_recursion_guard[
    DMCE_MAX_NUM_KTHREADS];

static void dmce_probe_body(unsigned int probenbr) {
    pid_t pid = current->pid;
    if (dmce_recursion_guard[pid])
        return;
    dmce_recursion_guard[pid] = 1;
    // Critical section
    dmce_recursion_guard[pid] = 0;
}
```

At first glance the example code seen in listing 3.3 seems to create a relatively large array with uninitialised values for each file (which could result in never reaching the critical section of a probe), however, in C, global variables without an explicit initialisation are implicitly initialised to zero[14]. Additionally, the *weak* attribute tells the compiler (in the kernel case, GCC) that it may overwrite this symbol if we see multiple declarations [5], resulting in everyone seeing the same symbol.

3.3.3 Timestamps

Something that a probe might want to do is to collect timestamps of when it was called, such as when making a trace probe. The kernel provides the *ktime* interface for timekeeping, which provides up to nanosecond resolution. There are several different functions which use different clocks and provide different granularities [18].

There exist so-called *fast* variants of the functions. These are safe to call from any context, which is beneficial for the probes, as the probes may be inserted anywhere in the source code. The downside of these is that the time is allowed to jump under certain conditions, which is not allowed using the base versions. Another fast alternative to read timestamps is to utilise architecture-specific instructions with inline assembly, e.g. *rdtsc* for x86(_64) and *mrc p15* on ARM.

Another variant is the *coarse* variant, which instead of giving nanosecond resolution, is updated every so-called *jiffy*. The frequency of this can be configured when building the kernel and is, for the newest kernel version at the time of writing, limited to either 100**Hz**, 250**Hz**, or 1000**Hz**. As a result, the best resolution of this variant of the functions is a millisecond.

Since the probes might be called very often, it is important that generating timestamps does not take too much time. Depending on the probe, it might also be important that the timestamps are generated with a high enough granularity. Exploring the different variants is therefore important when constructing the probes.

3.4 Extracting the probe data

A part of any analysis is getting a hold of the generated data. Using the amenities of QEMU and its monitor tools, the method we moved forward with was dumping the kernel memory to a file on the host machine, though we did explore several options.

When using DMCE with user space programs, DMCE registers an on-exit callback for the respective terminating signals that ultimately writes all gathered data in binary format to a file with a known location on disk. The kernel equivalent callbacks, e.g.*register_reboot_notifier*, *register_die_notifier* and *register_panic_notifier*, will call all registered functions one at a time when the system is shutting down or rebooting, a process unexpectedly died or a panic has occurred respectively. Writing to a file directly from the kernel is possible but is considered bad practice and is thus discouraged [21]. Additionally, one major issue is that a panic occurring indicates that things have gone terminally wrong and executing code as normal is no longer possible. Thus writing to a file at this point might not be possible or, even worse, cause a corruption of the file system.

An alternative is to implement an input/output control (ioctl) kernel module that writes the gathered data to disk on demand using the existing internal file systems. This could then be used together with a symbolic link between the guest and host machine to extract it from the emulator. Our implementation of such kernel modules can be found in appendix B.

Even so, we still have one issue to address; what happens if the kernel enters a panic before we have the opportunity to export our data through our module? Since we are in fact running the kernel through an emulator, we have access to its toolbox of commands via the QEMU monitor, regardless of whether the emulated kernel panics or not. In particular, the

QEMU monitor offers us the command *memsave*, which takes a virtual memory address, a size and a file path as arguments. Given a known address and size, this command allows us to dump the raw memory from this address up to a certain size directly into a file on the host.

Chapter 4

Evaluation

In this chapter we will go over our solutions to problems presented above, some concrete results and provide a basis for answering our research questions with a discussion.

4.1 Experimental setup

The setup used consists of virtualisation-enabled machines. The most interesting is a computer with an Intel(R) Xeon(R) CPU E5-2680 v4 2.40 GHz with 56 logical cores and 128GB memory. The other two are laptops with Intel(R) Core(TM) i5-1145G7 2.60GHz with 8 logical cores and 32GB memory, and Intel(R) Core(TM) i5-1245U with 12 logical cores 1.60 GHz and 16GB memory respectively.

For QEMU, we used version 8.1.4 and 6.2.0 and ran mainly the *qemu-system-x86_64* emulator with flags allowing it to use all available cores, 1GB of memory and *kvm* enabled. The kernel version used for testing and probe development is 6.8.0-rc1 and 6.8.0-rc6 with default configurations for x86(_64) targets.

The setup for DMCE and its dependencies is *git version 2.34.1*, *GNU bash, version 5.1.16(1)-release*, *Python 3.10.12*, and *clang-check (llvm) version 17*.

The described hardware and software will be used to run and test instrumented kernel images. The probe implementations of the *trace* and *heatmap* use cases will be evaluated incrementally to arrive at a final implementation, as will the extraction of the gathered data. Furthermore, they will be used to gather data about how the kernel behaves when certain parts are instrumented to illustrate the feasibility of instrumenting the kernel using DMCE.

4.2 Results

To help answer RQ 1 we began trying to compile and run the kernel with a minimally invasive probe, a stub, that simply returns when called. This proved successful and proved that the kernel can indeed run when instrumented in this manner. However, it did not work without some tweaking. The kernel contains a lot of intricate macros and constructs that, when DMCE and clang-check do not find a declaration, are treated as call expressions. When said macros are then expanded during compilation, they become incompatible with the comma notation used for probe insertion. One such example would be if the macro in question encapsulates its code within a *do-while* loop as seen below.

```
#define SOME_MACRO(arg) \
do { \
    // Some interesting macro code \
} while(0);
```

If *SOME_MACRO* is interpreted as a call-expression during static analysis due to e.g. not being found by clang-check, its expansion causes compilation issues as it is not a C-expression, and thus should be excluded from probing. Ultimately, we ended up with a rather long list of constructs (see figure 4.4) that cause these compilation issues, which we in turn tell DMCE to ignore when inserting probes.

To tackle RQ 2 we started implementing probes using available kernel mechanisms instead of the C standard library. In particular, we put most of our effort into a *trace* and *heatmap* probe respectively since a heatmap is more or less an extension of coverage and trace enables more in-depth analysis. In sections 3.3 & 3.4 we brought up different problems encountered while working on our implementations. The first of the problems is how we allocate buffers for storing information gathered during runtime. For this, we opted for the approach exploiting the C standard and compiler attributes for weak symbols. This gave us a zero-initialised buffer shared among all probes which we could easily find the address and size of by either printing it with *printk* or having a gander in the appropriate ELF-file. The address and size were then used together with the QEMU monitor's command *memsave* to dump the buffer's memory to a file on the host machine's disk, which happens to work well with DMCE as raw binary is the preferred format.

Before discovering the amenities of the QEMU monitor, however, we explored the kernel module route for extracting the probe data. That is, declaring the buffer as a pointer to an external symbol, that in turn was declared in the module. This led us to implement two versions of a kernel module (implementations can be found in appendix B) capable of outputting the data to disk; both using a symbolic link between host and guest machines, but using the *dev-* and *proc*-filesystems respectively. Both approaches yielded the same result, i.e. data on the host machine's disk, but the *proc* version needed to take the kernel version into account when setting up the structures needed for the file operations. However, The kernel module approach suffers from one fatal flaw; how do we extract the probe data if the kernel panics? This prompt is the seed that resulted in the aforementioned solution using the QEMU monitor. Nevertheless, using a pointer for our buffer entails explicitly allocating memory for it. To ensure only allocating to it once, we once again exploited the C standard and GCC's attributes to create a shared and atomic flag to indicate whether the buffer was being or had been allocated.

This brings us to the next problem faced when implementing the probes; recursion. As described in section 3.3.2, when probing subsystems the probes themselves have a dependency to, we risk seemingly endless recursion. We examined several options, i.e. per-file, per-cpu and per-kthread approaches to recursion prevention. Ultimately, we found that the per-kthread approach was most reliable as it prevents the same process or task from getting stuck in recursion without potentially interfering with other processes or tasks. For an implementation of this, we refer to listing C.1 in appendix C. In the implementation, we make use of the macro *current* which expands to some inline assembly which ultimately returns a pointer to a *task_struct*, in which we can find the current kthread's process id. Additionally, similar to how the main buffer is shared among all files by exploiting the C standard and compiler attributes, an array keeping track of each process' probing status with an optimistic upper bound is used to ensure that we do not generate bogus data as a side effect of running our probe from different files.

Probed system	Timestamp	Cores	Average boot time[s]
None	N/A	1	8.3389664
None	N/A	4	11.8409898
None	N/A	8	12.7012772
None	N/A	12	13.4008224
Scheduler	frequent	1	9.5897324
Scheduler	frequent	4	15.2275568
Scheduler	frequent	8	20.8250404
Scheduler	frequent	12	24.6956386
Scheduler	coarse	1	9.4316688
Scheduler	coarse	4	13.3019116
Scheduler	coarse	8	14.1654928
Scheduler	coarse	12	13.9971258
Scheduler	rdtsc	1	9.695497
Scheduler	rdtsc	4	13.185071
Scheduler	rdtsc	8	14.959568
Scheduler	rdtsc	12	15.355636
Locking	frequent	1	8.8024434
Locking	frequent	4	12.2985034
Locking	frequent	8	13.003216
Locking	frequent	12	13.873114
Locking	coarse	1	8.7193754
Locking	coarse	4	11.6472528
Locking	coarse	8	12.7189084
Locking	coarse	12	13.6294404
Locking	rdtsc	1	8.884228
Locking	rdtsc	4	11.902195
Locking	rdtsc	8	13.306985
Locking	rdtsc	12	13.228649
Entry	frequent	1	8.403948
Entry	frequent	4	11.884388
Entry	frequent	8	13.328205
Entry	frequent	12	13.418529
Entry	coarse	1	8.659519
Entry	coarse	4	11.764442
Entry	coarse	8	12.951333
Entry	coarse	12	14.111808
Entry	rdtsc	1	8.417612
Entry	rdtsc	4	11.295550
Entry	rdtsc	8	12.519886
Entry	rdtsc	12	13.759620

Table 4.1: Average boot times for probe statuses and number of cores. The boot is considered done when the *init* process is started. The stated boot times are the averages of five boots per configuration.

	Probe hits	Probe location (file:line)
trace	**46914801**	
	36364448	kernel/trace/trace_events.c:2757
	5264112	kernel/trace/trace_events.c:2766
	3326824	kernel/trace/trace_events.c:2758
sched	**3176112**	
	98618	kernel/sched/core.c:7394
	60151	kernel/sched/core.c:559
	60105	kernel/sched/core.c:603
locking	**1840213**	
	97586	kernel/locking/rwsem.c:1346
	97586	kernel/locking/rwsem.c:1347
	97586	kernel/locking/rwsem.c:1622
rcu	**847171**	
	79537	kernel/rcu/rcu_segcblist.h:82
	63073	kernel/rcu/tree.c:227
	54151	kernel/rcu/rcu_segcblist.h:95
entry	**196268**	
	88513	kernel/entry/common.c:177
	88513	kernel/entry/common.c:186
	4333	kernel/entry/common.c:104
printk	**112404**	
	6817	kernel/printk/printk_ringbuffer.c:357
	5692	kernel/printk/printk_ringbuffer.c:366
	3537	kernel/printk/printk_ringbuffer.c:436
events	**76138**	
	23860	kernel/events/core.c:4337
	23857	kernel/events/core.c:4341
	23856	kernel/events/core.c:4144
irq	**11284**	
	1106	kernel/irq/settings.h:117
	1095	kernel/irq/chip.c:504
	1095	kernel/irq/handle.c:206
cgroup	**6286**	
	260	kernel/cgroup/pids.c:66
	258	kernel/cgroup/cgroup.c:770
	166	kernel/cgroup/cgroup.c:6401
futex	**1813**	
	268	kernel/futex/core.c:786
	268	kernel/futex/core.c:787
	141	kernel/futex/core.c:1091
dma	**424**	
	335	kernel/dma/swiotlb.c:816
	6	kernel/dma/direct.c:577
	6	kernel/dma/direct.c:578

Table 4.2: Table showing the three probes executing the most in each kernel sub-directory during a typical boot with 4 cores. Sub-directories which did not produce data are omitted for brevity.

To provide answers for RQ 3, we tweaked the implementation of the trace probed and observed the run-time performance of the probed kernel. In the case of noticeable performance issues, we analysed what could be the underlying cause, by both utilising our heatmap probe and investigating implementation details. We also studied how useful the produced traces were by looking at how much of the source code had been probed.

Table 4.1 shows the time it took to boot when different subsystems have been probed and when emulating different numbers of cores. The probe used is the trace probe, with three different methods used to measure timestamps, the high resolution *ktime_get_mono_fast_ns*, the coarser but faster *ktime_get_coarse_ns*, as well as the x86 assembly instruction *rdtsc*. The subsystems chosen to be measured are the scheduler, the locking subsystem, and the entry subsystem since probes in these subsystems get called a lot more frequently than other subsystems, apart from *trace*, as can be seen in table 4.2.

To understand the performance impact on a higher level we used our heatmap probe to pinpoint what probes run the most. We used the command *dmce-summary-bin* with the flag *"–filter"*, which takes the output of a heatmap probe and outputs a list of source code lines which account for a given percentage of all probe hits. By adding this list to the DMCE filter, this removes the probes accounting for that percentage of all probe hits. Figure 4.1 shows the number of executed probes as a function of the percentage of probe hits removed this way. In this case, the graph shows that very few probes (in the *trace* sub-directory, as seen in table 4.2, as well as one other file, *kallsyms.c*) caused the majority of the probe hits. While 4921 probes executed during boot, 47% of the hits were caused by 3 probes, and 79% were caused by 20 probes, meaning that the remaining 4901 probes accounted for only 21% of total probe hits. In order to verify that removing the most frequently executed probes this way translates directly to removing overhead, we measured the boot time when different when different percentages of probe hits have been filtered away, shown in figure 4.3. While there is some variation, the figure shows the trend that the boot time decreases linearly with the amount of probe hits filtered, showing that this is an effective way of removing overhead generated by the probes. Further, we ran the same test but without including the *trace* sub-directory and *kallsyms.c*, as they make up a vast majority of the total probe hits, and as seen in figure 4.2, which from the negative slope beginning almost immediately, shows that the overhead is more evenly distributed among the executed probes.

Due to the trouble DMCE has while constructing the AST using clang-check, as described in 3.2.1, some parts of the code might not be probed, leading to a lower probe coverage. This may impact the usefulness of the traces since larger chunks of code may have been executed between two traces, due to expressions there not being recognized in the source code. However, it stands to argue that the trace might still be usable in the sense that it is still possible to tell that the code between to trace entries have executed.

If the trace probe is used to determine the underlying cause of a crash, another factor in the usefulness of the trace is how quickly trace entries stop being generated after the error occurs. This is desirable since we are mainly interested in knowing what lead to the error, not what happened afterwards. When the kernel detects an error it starts a panic and continues executing panic code. To limit the number of traces collected after the error is triggered, we want to stop tracing early during the panic. We found through testing that this can be achieved either by registering a callback function to the *panic_notifier_list*, as described in 3.4, that disables tracing, or by manually creating a breakpoint at the very start of the panic function that tells DMCE to disable tracing.

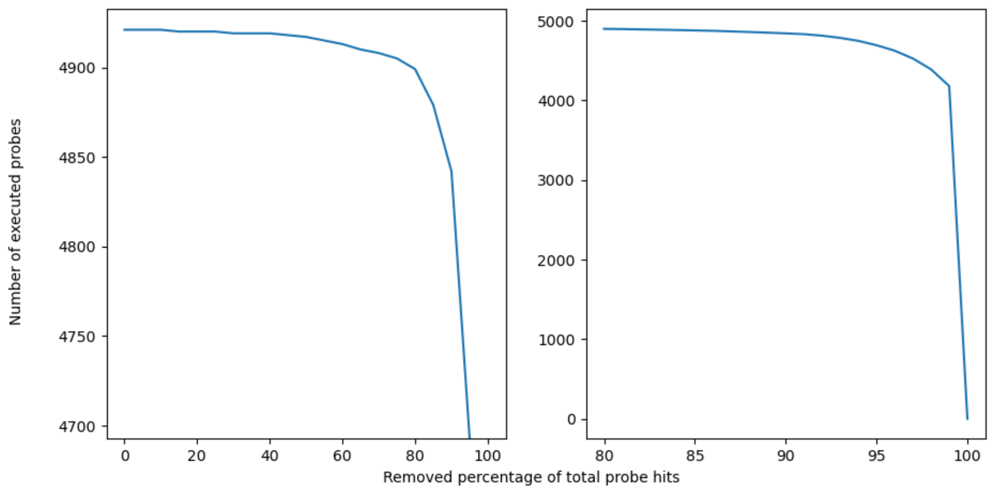


Figure 4.1: Graphs illustrating the number of executed probes in the *kernel* directory as a function of the percentage of total probe hits removed. The number of hits for each probe is measured from a typical boot with four cores. The y-axis for the left-most graph has a smaller interval to more clearly show where the drop in probe amounts occurs. The right-most graph uses the same data set as the left-most, but on the x-axis interval 80-100 to make up for what the left-most does not show.

4.3 Discussion

4.3.1 RQ 1

To answer the question *"To what extent is it possible to statically instrument the kernel code with probes using DMCE?"* and consequently *"How can problems encountered with the probe analyser/writer be handled?"*, we will discuss our findings while experimenting and how we approached problems arising along the way.

With the framework DMCE provides it is possible to instrument the vast majority of the kernel and its subsystems. The subsystem that proved difficult to probe, however, was the timekeeping. Inserting stub probes that do nothing seemed to work without any issue (likely due to the compiler optimising away the, in reality, dead code), but as soon as we tried using e.g. the heatmap probe, the kernel would panic early on during boot. A potential reason for this issue while probing might be the usage of the *sequence counter* consistency mechanism in this subsystem. Slightly simplified, this mechanism ensures the consistency of a resource by having a sequence number associated with the resource that is incremented whenever the resource is updated. A reader is then able to ensure the consistency of this resource by reading the sequence number, then reading the resource, and finally the sequence number again. If the read sequence numbers differ, it means that the resource has been updated and the process is retried [20]. The situation where this might cause an issue is when multiple probes are called while reading from the resource, and the probes take a significant amount of time to run.

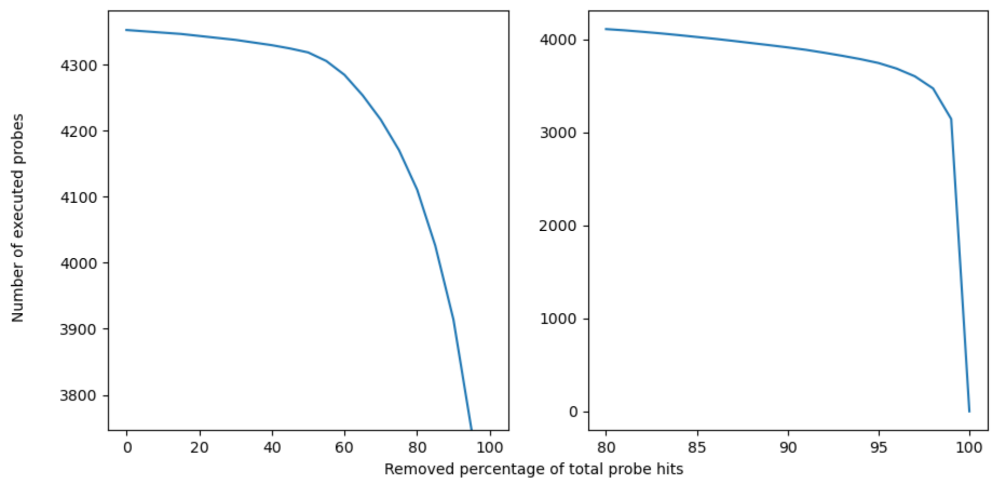


Figure 4.2: Graphs illustrating the number of executed probes in the *kernel* directory (excluding *trace* and *kallsyms.c*) as a function of the percentage of total probe hits removed. The number of hits for each probe is measured from a typical boot with four cores. The y-axis for the left-most graph has a smaller interval to more clearly show where the drop in probe amounts occurs. The right-most graph uses the same data set as the left-most, but on the x-axis interval 80-100 to make up for what the left-most does not show.

If this happens, it is more likely that a writer has updated the sequence number, in which case the reader must retry its critical section, and as a result run the probes again. Since the timekeeping subsystem handles resources that are periodically updated, if the time spent by the probes in a critical section exceeds the period of an update, it could essentially cause an infinite loop of the reader. Whether this potential issue actually occurs is not something we managed to determine, so further analysis of this and possible solutions are left for future work.

It is not certain that exactly every expression gets instrumented in a given subsystem. The kernel contains a myriad of convoluted macros and constructs that DMCE, by default, will try to ignore. The reason for the reluctance to instrument macros is simply their unpredictability in what they will expand to. Additionally, as described in section 3.2.1, the way some parts of the kernel are built, DMCE will sometimes miss certain macros, mistaking them for a call expression and thus insert a probe anyway. Depending on the resulting macro expansion, this may render the comma notation invalid and cause the compilation to fail. A collection of macros that we found to be causing such issues can be seen in figure 4.4. The list might seem quite large, but we found that a significant portion of these macros are only used once or twice (most likely for readability). Some of them however are synchronisation-related, e.g. *READ_ONCE* & *lock_acquire*, which may be used to a broader extent, meaning we lose accuracy relative to the amount of synchronisation. Instead of using comma notation, the GCC extension *statement expressions* allow a larger portion of these macros to be probed.

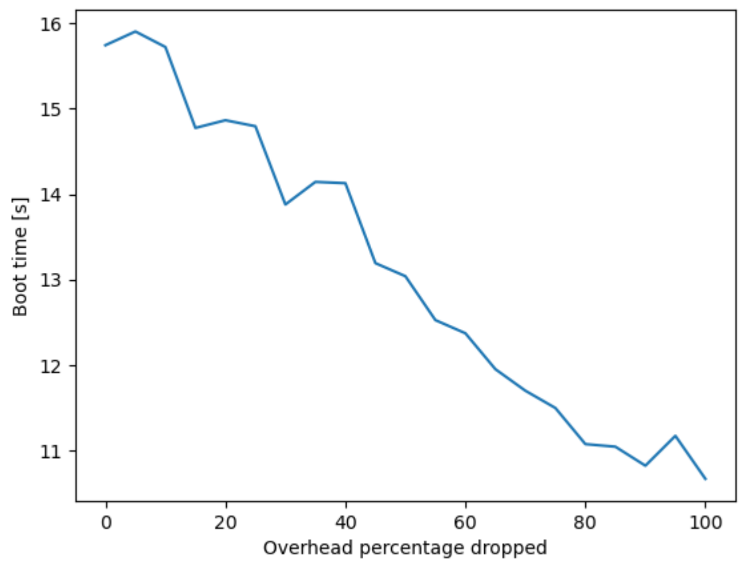


Figure 4.3: Graph illustrating the boot times when the kernel has been probed with our trace probe and increasing percentages of total probe hits have been filtered out. Each data point is the average of five measurements while booting with four cores. The boot is considered done when the *init* process is started.

4.3.2 RQ 2

We will answer this question (*How can we design and write probes for common analysis use cases (e.g. heatmap & trace) in kernel space?*) by revisiting the problems described in section (3.3) and discuss them in combination with the solutions we found and implemented.

Several kernel-related issues must be taken into account when writing probes for the kernel. More specifically, only use functionality already implemented in the kernel while also avoiding recursion if we were to probe said functionality, and store and allocate our data buffer appropriately in a global context to ease the process of then extracting it.

Writing code in kernel space differs considerably from writing programs intended to run in user space. First and foremost, we no longer have access to the amenities of the C standard library, or any other library for that matter. In other words, we must rely on our own proficiency in C and the functionality provided by the kernel itself. An example of this presents itself when we need to allocate a bunch of memory. In userland, we simply call e.g. *malloc* or *calloc* and tell them how much memory we want. Within the kernel, however, there are additional things to consider. In most cases (and generally preferred) we can use *kmalloc* similarly to *malloc* but with an extra flag telling the system how we want to allocate it (most commonly *GFP_KERNEL*). However, this does not work in every case. Kmalloc only guarantees to allocate up to 128KB [9], possibly causing issues if the amount we allocate exceeds this, depending on the build configuration and target architecture. The number of probes and memory needed may vary considerably depending on the scope of a given probing. In other words, since we know we might exceed this limit, it is better to use e.g. *vmalloc* which instead guarantees at least the amount we allocate in contiguous virtual memory. Another

```
RCU_LOCKDEP_WARN,kfree_rcu,kvfree_rcu_mightsleep,
kexec_flush_icache_page, spin_acquire,lock_acquire_exclusive,
spin_release,lock_acquire,lock_release,SCHED_WARN_ON,
kthread_init_work,LIST_HEAD,update_rt_rq_load_avg,rcu_sleep_check,
update_idle_cfs_rq_clock_pelt,update_rt_rq_load_avg,u64_u32_store,
WRITE_ONCE,rcu_sleep_check,schedstat_inc,u64_u32_load_copy,
__schedstat_set,kthread_init_work,mutex_init,
lockdep_set_class_and_name,set_current_state,spin_lock_init,
SCHED_WARN_ON,rcu_assign_pointer,WRITE_ONCE,lock_acquire,
mutex_acquire_nest,lock_contended,set_current_state,lock_release,
mutex_release,lockdep_init_map,__set_current_state,wait_var_event,
__bpf_prog_array_free_sleepable_cb,lockdep_set_class_and_name,
kprobe_flush_task,sched,likely,for_each,foreach,BUG,unreachable,cpu,
barrier,BLANK,rmb,irq,rdmsr,raw_spin_lock_init,arch_end_context_switch,
smp,pte_unmap,preempt,lock_cmos,free_vm86,wmb,dev_level_ratelimited,
dev_warn_ratelimited,dev_level_once,dev_warn_once,spin_lock_init,
mutex_init,deactivate_mm,init_rwsem,might_sleep,setup_thread_stack,
mt_set_external_lock,rcu_dereference,READ_ONCE
```

Figure 4.4: A comma-separated list of macros that were not successfully detected by DMCE to be macros, and which caused compilation failure when used with the comma operator due to not expanding to an expression.

thing to consider when explicitly allocating the data buffer is of course to not do it more than once. This is not in any way unique to probing in kernel space but does require its own way of achieving it. Unlike the provided user space solution which makes use of the lock mkdir provides, we can make use of an atomic flag declared with the *weak* attribute described in section 3.3.2, thus granting the same atomicity as the mkdir lock. However, atomically checking a condition every time a probe executes, may prove slow if our probes are hit increasingly often. Compare this with table 4.2, where we can see that different parts of the kernel hit significantly more probes during boot than others. To optimise this, at least a bit, we opted to use a local static variable on top of the shared atomic one, to exploit the speculative branching of some architectures using the *unlikely* macro.

Since we are indeed confined to kernel space and using kernel mechanisms, an important question to ask oneself when writing a probe is *"Will I ever use this probe to probe the mechanisms I am using within the probe?"*. If the answer is "no", then we do not have to worry, but if the answer is "yes", "maybe" or any other positively affirming expression, we need to consider probe recursion and bogus probe hits. As described in section 3.3.2, there are different ways to approach this, but as we found in section 4.2, only the latter is suitable. This prevents both endless recursion as well as producing data as a side effect of probing.

4.3.3 RQ 3

To answer the questions *Can DMCE's probe-based approach be used to trace the Linux kernel with reasonable accuracy?* and *How can we handle run-time performance issues caused by the overhead introduced by the instrumentation?*, we analyse the trace probe, the traces it produces, and its effects on kernel performance as well as identify the bottlenecks of the introduced overhead.

In short, we found that DMCE's probe based approach is capable of generating traces with relatively high accuracy, e.g. the loss of probes due to manual filtering overhead did not affect the trace sufficiently to render the trace unusable. Furthermore, the performance loss caused by introduced overhead was found to depend on what parts of the kernel was probed as well as implementation details of the probe. Additionally, filtering out the probes responsible for the most overhead improves performance without affecting the trace much.

It was found that the trace probe is capable of causing a significant performance penalty, as can be seen in table 4.1. When running on a single-core system, the overhead of using the trace probe was reasonably low, the difference between the unprobed boot time and the slowest configuration was only 15%. However, the performance quickly degrades as the number of cores increases, when running on 12 cores, the overhead of the slowest configuration compared to the unprobed kernel was 84%. Interestingly, the large difference in performance was only seen when probing the scheduler, not when probing other subsystems, such as locking, which is also shown in table 4.1. The scheduler is the subsystem where the probes are called the most frequently during boot, aside from *trace*, as shown in table 4.2, so it is not surprising that it causes the largest performance degradation. However, if the performance was directly related to the number of times the probes were called, it would not explain how the locking subsystem, which calls its probes 58% as often as the scheduler does, only has an overhead of 3.5% while running on 12 cores, as compared to the 84% overhead when probing the scheduler.

The largest source of performance issues was found to be the collection of timestamps using the high-resolution function *ktime_get_mono_fast_ns*. When using the faster, less precise function *ktime_get_coarse_ns* or the assembly instruction *rdtsc* instead, the performance overhead significantly decreases. When probing the scheduler and using 12 cores, the overhead while using *ktime_get_coarse_ns* is only 4.4%, and the overhead while using *rdtsc* is 14.6%. While *ktime_get_coarse_ns* leads to the highest performance, it also causes a deterioration in the quality of the traces.

DMCE sorts all trace entries based on their timestamps, so if the granularity of the timestamps is not high enough, the final order of the entries might not be the same order as they occurred while the kernel ran as many trace entries can get the same timestamp. As mentioned in section 3.3.3, the resolution of the timestamps using the function *ktime_get_coarse_ns* is at best one millisecond. One of the traces we gathered using this method of collecting timestamps showed that over 1000 trace entries might have received the same timestamp. In this case, it was impossible to follow the control flow of the execution that was traced.

The x86 instruction *rdtsc* provides as high resolution as *ktime_get_mono_fast_ns*, but with a significantly lower performance overhead, so it seems to be the best choice when building the kernel for the x86 architecture. This shows that architecture-specific code in the probes can be used to improve the run-time performance of the probes, but a non-architecture-specific function should be used as a fallback when there is no such suitable mechanism for the architecture.

Implementation details aside, another efficient way to identify performance bottlenecks is to utilise a heatmap probe in combination with *dmce-summary-bin*'s *"–filter"* flag, as described in section 4.2. We see from the linear trend of figure 4.3 that removing the probes causing a certain percentage of all probe hits directly reduce the overhead caused by probing. Combining this with figure 4.1, we see that a very small number of probes account for a large part of the overhead generated by the probes. In this case, these few probes can be excluded to significantly improve performance while barely affecting the probe coverage and the usefulness of the trace.

In section 4.2, two different ways of pausing tracing during kernel panic are discussed. Inserting a breakpoint at the start of the panic function has the advantage of happening earlier than registering a callback to the *panic_notifier_list*, as this happens after some setup of the panic environment has been executed. As such, using breakpoints leads to fewer unnecessary traces, which simplifies the process of analysing the traces and discovering what caused the panic. In cases where there exists some pre-existing knowledge regarding where the error happens, a breakpoint may be inserted even earlier than when the panic function is called, as the kernel only starts to panic when the error is detected, not when it is caused.The disadvantage of using breakpoints is that they need to be manually inserted into the source code. As such, this solution does not fit if there is a need for the trace probe to work by default, however, when modifying the source code is acceptable, using breakpoints does work well. Our trace probe, shown in appendix C, uses the *panic_notifier_list*.

4.4 Limitations

4.4.1 Emulated environment

One relevant point to consider is the very fact that we are running our probed kernel images in an emulated environment. While the purpose of an emulator is to provide an environment that reflects the targeted hardware, it still does not run on said hardware when performing tests. In other words, there is no guarantee that a probed kernel would run on an actual machine with a specific architecture just because it can in an emulator.

This also impacts performance measurements. When measuring the boot times, see table 4.1, this is done in the emulator, which runs on the host machine and is subject to being scheduled by the host operating system. Although QEMU is fully capable of emulating multiple cores, it is not a guarantee that QEMU will get access to all cores at the same time, and guaranteed that it will not be interrupted while running. In contrast, booting the kernel on a dedicated machine gives it complete control over all cores.

4.4.2 Static analysis

At the time of writing, there is no way to use DMCE without having a dependency to *clang-check*. In other words, the only available AST to look for expression in is the one generated by *clang-check*. A successful, or rather complete, probing of any given source code is therefore assuming *clang-check* can handle it all. DMCE does in fact handle *GCC*'s torture tests [8], but sometimes need the user to point out *include*-directories to look for declarations in when generating the AST. This becomes increasingly problematic as the build systems grow more complex. An example of this (briefly mentioned in section 3.2.1) can be found within architecture-specific and build configuration code, where certain *.c*-files are directly included (without include-directives in themselves) in another source file. It turns out this is a compilation-time optimisation when building using different configurations. In general, including *.c*-files directly is considered bad practice and thus should not be all too common. However, in special cases, such as the kernel, it is most certainly something to consider when evaluating the performance of static analysis.

This, in turn, begs another question; to what degree must a given project be able to be probed to produce sufficient value? For the use cases we have implemented, i.e. heatmap and trace, "good" instrumentation would be at least one inserted probe per edge of the *control-flow graph* [2] within the scope of the probing. This would mean that every possible execution path within our probing scope could be traced. But what if some of the paths cannot be probed for some previously discussed reason; how useful is the instrumentation then? There will be some loss in accuracy, but not necessarily enough to make a trace unusable. What constitutes an unusable trace would be trace entries with a sufficiently large gap in between them such that it is difficult or impossible to tell what code has been run in the meantime. Looking at the graphs in figures 4.1 & 4.2, for example, we can see that the most significant drop in probe amounts happens around the upper half of the 80-100% interval of removed overhead. That said, even if we intentionally remove instrumentation causing the most overhead, the risk of a trace becoming unusable is low.

4.4.3 Changes to the internal Linux kernel API

The internals of the kernel are free to change whenever there is a technical reason for it, as long as the external API is maintained. Since our probes use internal kernel functions, the probes may no longer be valid in a future release of the kernel. Similarly, the probes are not guaranteed to function for older kernel versions either, since the functionality they may rely on might not have been implemented yet. This is due to the level of abstraction that DMCE operates on, i.e. in the source code itself. This would not be a problem with something like eBPF that does not modify and run in the kernel, but due to its higher abstraction level, it may leave out some details that DMCE would not.

An example of this can be seen in the *proc*-version of our kernel modules (see listing B.1). The module contains conditional pre-processor directives to determine whether the kernel version is sufficiently high to use *proc* operations, which were added in kernel version 5.6.0. In other words, if we did not take this into account and opted to only use the *proc* operations API, the module would not compile for versions lower than 5.6.0 and thus be useless unless changed.

Chapter 5
Conclusions

In this section, we summarise our findings and discuss what a continuation of this work might entail.

5.1 Summary of results

Through experimentation and implementation of kernel probes based on the available user space probes in DMCE, we show that it is possible to instrument the Linux kernel's source code with this approach (except e.g. time-keeping), and with relative ease collect the generated data in an emulated environment using the provided toolbox of the emulator and custom kernel modules.

We provide a framework for how to approach writing a probe when instrumenting the kernel, by discussing the problems we have stumbled upon, and propose possible solutions to said problems. Furthermore, we present what considerations to make regarding the extraction of the collected data and what scenarios one or the other is more suitable.

Additionally, we show that it is possible to generate a source code level trace from the kernel. The overhead and performance implications vary with the rate at which the probes are hit, and what parts of the kernel have been probed.

5.2 Future work

Our work is intended to serve as a basis for how to make use of a source code level probe approach when performing analyses on the Linux kernel. That said, some possible improvements and ideas can be expanded upon. We have implemented probes for a couple of use cases, but what use cases there are, is limited only by imagination (and possibly hardware). One idea might be to combine the traces generated by DMCE with other sophisticated analysis tools such as *bpftrace*[1].

One of the issues we encountered is that the way the kernel is built makes clang-check struggle with constructing a complete AST. Finding a way to mitigate this issue would both improve the probe coverage, leading to more useful analyses and reduce the number of invalid probes inserted.

One limitation of our work is that it only tackles extracting the probe data when using an emulator to run the probed kernel. It would be very valuable to investigate a solution for achieving this on actual hardware, i.e. bootstrapping the instrumented image. A possibility for this is by loading a crash kernel on panic to extract the binary probe data.

Lastly, it was found probing some files caused the kernel to stop working, either by generating a panic or simply freezing during boot, such as when probing the timekeeping subsystem. While we have hypotheses as to potential issues, further investigation to determine the cause would be beneficial as this might allow more of the kernel to be probed.

[1]`https://bpftrace.org/`

References

[1] Linux operating systems: Distributions. https://web.archive.org/web/20181003201630/http://swift.siphos.be/linux_sea/whatislinux.html#idm3571768989216 [Online; accessed 2024-06-16].

[2] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, jul 1970.

[3] Yucong Chen, Xianzhi Tang, Shuaixin Xu, Fangfang Zhu, Qingguo Zhou, and Tien-Hsiung Weng. Analyzing execution path non-determinism of the linux kernel in different scenarios. *Connection Science*, 35(1):2192442, 2023.

[4] Gnu Compiler Collection. Extensions to the c language family. https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html [Online; accessed: 2024-05-17].

[5] Gnu Compiler Collection. Function attributes. https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Function-Attributes.html#:~:text=The%20weak%20attribute%20causes%20the,used%20with%20non%2Dfunction%20declarations. [Online; accessed: 2024-05-17].

[6] Gnu Compiler Collection. Specifying attributes of variables. https://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html [Online; accessed: 2024-05-17].

[7] Gnu Compiler Collection. Statement exprs. https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html.

[8] Gnu Compiler Collection. Torture tests. https://gcc.gnu.org/onlinedocs/gccint/Torture-Tests.html.

[9] Jonathan Corbet, Alessandro Rubini, , and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly Media, Inc., 2005.

[10] Al Danial. Count lines of code. https://github.com/AlDanial/cloc.

[11] J. Edge, 2019. https://lwn.net/Articles/803347/ [Online; accessed 2024-06-16].

[12] M. Flemming, 2017. https://lwn.net/Articles/740157/ [Online; accessed 2024-06-16].

[13] Chris Hankin Flemming Nielson, Hanne Riis Nielson. *Principles of Program Analysis*. Springer Berlin, Heidelberg, 1999.

[14] International Organization for Standardization. C iso/iec 9899. https://www.iso-9899.info/wiki/The_Standard [Online; accessed: 2024-05-15].

[15] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall, 2012. Prentice Hall Professional.

[16] Hsin-Wei Hung. *Securing Operating Systems with Dynamic Program Analysis*. PhD thesis, University of California, 2023.

[17] The kernel development community. Cpu architectures. https://www.kernel.org/doc/html/v6.3/arch.html [Online; accessed: 2024-05-17].

[18] The kernel development community. ktime accessors. https://docs.kernel.org/core-api/timekeeping.html [Online; accessed: 2024-05-17].

[19] The kernel development community. Rust. https://docs.kernel.org/rust/ [Online; accessed: 2024-05-17].

[20] The kernel development community. Sequence counters and sequential locks. https://docs.kernel.org/locking/seqlock.html [Online; accessed: 2024-05-17].

[21] Greg Kroah-Hartman. Driving me nuts - things you never should do in the kernel. *Linux Journal*, 2005. https://www.linuxjournal.com/article/8110 [Online; accessed: 2024-05-17].

[22] Youlin Li, Weina Niu, Yukun Zhu, Jiacheng Gong, Beibei Li, and Xiaosong Zhang. Fuzzing logical bugs in ebpf verifier with bound-violation indicator. In *ICC 2023 - IEEE International Conference on Communications*, pages 753–758, 2023.

[23] The LTTng Project. Lttng. https://lttng.org/[Online; accessed: 2024-05-16].

[24] QEMU. Quick emulator (qemu). https://www.qemu.org/ [Online; accessed: 2024-05-15].

[25] E. Rocca, 2021. https://lwn.net/Articles/852112/ [Online; accessed 2024-06-16].

[26] Gabriel Ryan, Abhishek Shah, Dongdong She, and Suman Jana. Precise detection of kernel data races with probabilistic lockset analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2086–2103, 2023.

[27] Jonas Skeppstedt and Christian Söderberg. *Writing Efficient C Code*. Independently published, 2011.

[28] Systemtap. https://sourceware.org/git/systemtap.git[Online; accessed: 2024-05-16].

[29] H. Waly and B. Ktari. A complete framework for kernel trace analysis. In *2011 24th Canadian Conference on Electrical and Computer Engineering(CCECE)*, pages 001426–001430, 2011.

[30] Patrik Åberg. Did my code execute? `https://github.com/PatrikAAberg/dmce` [Online; accessed: 2024-05-15].

Appendices

Appendix A
Author contribution

This appendix contains a table specifying individual contributions to the report.

Section	Andreas	Jesper
Abstract	X	
Introduction		
- Contribution	X	
- Related work	X	
Background		
- C/C++		X
- The linux kernel		X
- QEMU	X	X
- Program analysis		X
- DMCE	X	X
Method		
- Probing the kernel		X
- Writing kernel friendly probes	X	X
- - The allocation consideration	X	
- - The recursion excursion	X	
- - Timestamps		X
- Extracting the probe data	X	
Evaluation		
- Experimental setup	X	
- Results	X	X
- Discussion	X	X
- - RQ 1	X	X
- - RQ 2	X	
- - RQ 3		X
- Threats to validity	X	X
- - Emulated environment	X	X
- - Static analysis limitations	X	
- - Changes to the internal Linux kernel API	X	X
Conclusion		
- Summary of results	X	
- Future work		X

Table A.1: Table underlining who had the most responsibility for a given part of the report (two X's = roughly equal responsibility).

Appendix B
Kernel modules

In this appendix are our respective kernel module implementations. One utilising the *proc* system and the other the *dev* system.

Listing B.1: Kernel module using the proc system to output data gathered by DMCE

```
/* BEGIN DMCE MODULE */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/uaccess.h>
#include <linux/version.h>
#include <linux/minmax.h>
#include <linux/init.h>
#include <linux/atomic.h>
#include <linux/printk.h>
#include <linux/vmalloc.h>
#include <asm/page.h>

#define DMCE_RACE_TRACK_K 1

#define PROCFS_NAME "dmce"
#define PROCFS_MAX_SIZE (1024 * 1024 * 8)

#if LINUX_VERSION_CODE >= KERNEL_VERSION(5, 6, 0)
#define HAVE_PROC_OPS
#endif

static struct proc_dir_entry *proc_file;
static size_t procfs_buffer_size = 0;

#if DMCE_RACE_TRACK_K
int nbr_probes;
atomic_t dmce_buffer[1024];
```

```
#else
extern int nbr_probes;
extern atomic_t dmce_buffer[];
#endif

static ssize_t proc_read(struct file *fp, char __user *buf, size_t
    buf_len, loff_t *offset) {
    if (*offset) {
        pr_debug("procfs_read:␣END\n");
        *offset = 0;
        return 0;
    }

    int *dmce_tmp_buffer = vmalloc(sizeof(atomic_t) * nbr_probes);

    for (size_t i = 0; i < nbr_probes; i++) {
        dmce_tmp_buffer[i] = atomic_fetch_add(0, &dmce_buffer[i]);
    }

    procfs_buffer_size = min(sizeof(atomic_t) * nbr_probes, buf_len
        );

    if (copy_to_user(buf + *offset, dmce_tmp_buffer + *offset,
        procfs_buffer_size)) {
        vfree(dmce_tmp_buffer);
        return -EFAULT;
    }
    vfree(dmce_tmp_buffer);

    *offset += procfs_buffer_size;

    pr_info("procfs_read:␣read␣%lu␣bytes\n", procfs_buffer_size);
    return procfs_buffer_size;
}

static ssize_t proc_write(struct file *fp, const char __user *buf,
    size_t buf_len, loff_t *offset) {
    procfs_buffer_size = min(PROCFS_MAX_SIZE, buf_len);

    if (copy_from_user(dmce_buffer, buf, procfs_buffer_size))
        return -EFAULT;

    *offset += procfs_buffer_size;

    pr_info("procfs_write:␣write␣%lu␣bytes\n", procfs_buffer_size);
    return procfs_buffer_size;
}

#ifdef HAVE_PROC_OPS
static const struct proc_ops proc_file_fops = {
    .proc_read = proc_read,
    .proc_write = proc_write,
};
#else
static const struct file_operations proc_file_fops = {
    .read = proc_read,
    .write = proc_write,
};
#endif

int __init init_module(void) {
    proc_file = proc_create(PROCFS_NAME, 0644, NULL, &
        proc_file_fops);
    if (NULL == proc_file) {
        proc_remove(proc_file);
        pr_alert("error:␣could␣not␣create␣/proc/%s\n", PROCFS_NAME)
            ;
        return -ENOMEM;
    }

    pr_info("/proc/%s␣created\n", PROCFS_NAME);

    return 0;
}

void __exit cleanup_module(void) {
    proc_remove(proc_file);
    pr_info("/proc/%s␣removed\n", PROCFS_NAME);
}

module_init(init_module);
module_exit(cleanup_module);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("In-tree␣dmce␣proc␣module");
/* END DMCE MODULE */
```

Listing B.2: Kernel module using the dev system to output data gathered by DMCE

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/cdev.h>
#include <linux/atomic.h>
#include <linux/minmax.h>

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Driver␣for␣extracting␣data␣from␣DMCE␣trace");

#define MAX_TRACES_TO_RETURN 10000

typedef struct {
    uint64_t timestamp;
    uint64_t probenbr;
    uint64_t cpu;
} dmce_probe_entry_t;

extern dmce_probe_entry_t dmce_buffer[];
extern atomic_t dmce_buffer_size;
extern atomic_t no_probe;
```

```

static dev_t dev;
static struct cdev my_device;

static struct class *my_class;

//Prevent traces while open, otherwise reading traces can create
    new traces, creating a loop
static int dmce_trace_open(struct inode *, struct file *){
    atomic_set(&no_probe, 1);
    return 0;
}

static int dmce_trace_release(struct inode *, struct file *){
    atomic_set(&no_probe, 0);
    return 0;
}

static ssize_t dmce_trace_read(struct file *file, char __user *
    user_buffer,
                    size_t size, loff_t *offset)
{
    if(*offset < 0){
        return 0;
    }

    int buffer_size = atomic_read(&dmce_buffer_size);
    printk(KERN_ALERT "dmce_trace:␣%d␣trace␣events\n", buffer_size)
        ;
    if(buffer_size >= MAX_TRACES_TO_RETURN){
        buffer_size = MAX_TRACES_TO_RETURN - 1;
    }

    const size_t elems_left = buffer_size - *offset;
    if(elems_left <= 0){
        return 0;
    }

    const size_t size_of_elem = sizeof(dmce_probe_entry_t);
    const size_t num_elems_fit = min(size / size_of_elem,
        elems_left);
    const size_t bytes = num_elems_fit*size_of_elem;

    if (copy_to_user(user_buffer, dmce_buffer + *offset, bytes)){
        printk(KERN_ALERT "dmce_trace:␣copy_to_user␣failed\n");
        return -EFAULT;
    }
    *offset += num_elems_fit;

    return bytes;
}

//Can be used for giving commands to the DMCE trace (start, stop,
    etc.)
static ssize_t dmce_trace_write(struct file *file, const char
    __user *user_buffer,
```

```c
                    size_t size, loff_t *offset)
{
    char *str = kmalloc(size + 1, GFP_KERNEL);
    unsigned long bytes_left = copy_from_user(str, user_buffer, size);
    if(bytes_left > 0){
        // handle incoming command
    }
    str[size] = '\0';
    printk(KERN_CRIT "dmce_trace:␣received␣(%s)\n", user_buffer);
    kfree(str);
    return size;
}

static const struct file_operations fops = {
        .owner = THIS_MODULE,
    .open = dmce_trace_open,
    .release = dmce_trace_release,
    .read = dmce_trace_read,
    .write = dmce_trace_write,
};

static int __init dmce_trace_init(void)
{
    int error;

    if ((error = alloc_chrdev_region(&dev, 0, 1, "dmce_trace")) < 0) {
        printk(KERN_ERR
               "dmce_trace:␣Couldn't␣alloc_chrdev_region,␣error=%d\n",
               error);
        return 1;
    }

    my_class = class_create("mydriverclass");
    device_create(my_class, NULL, dev, NULL, "dmce_trace");
    cdev_init(&my_device, &fops);

        error = cdev_add(&my_device, dev, 1);
        if (error) {
                printk(KERN_ERR
                       "dmce_trace:␣Couldn't␣cdev_add,␣error=%d\n", error);
                return 1;
        }

    return 0;
}

static void __exit dmce_trace_exit(void) {
    class_destroy(my_class);
}

module_init(dmce_trace_init);
module_exit(dmce_trace_exit);
```

Appendix C
Probe implementations

This appendix contains implementations of the *trace* and *heatmap* probes.

Listing C.1: Kernel adapted heatmap probe implementation

```c
#ifndef __DMCE_PROBE_FUNCTION_BODY__
#define __DMCE_PROBE_FUNCTION_BODY__

#include <linux/slab.h>
#include <linux/atomic.h>
#include <linux/printk.h>
#include <linux/string.h>
#include <linux/syscalls.h>

#define DMCE_BUF_SIZE (sizeof(atomic64_t) * DMCE_NBR_OF_PROBES)
#define DMCE_MAX_NUM_KTHREADS_HOPEFULLY (1024 * 16)
#define DMCE_NO_RECURSE 1

atomic64_t __weak dmce_buffer[DMCE_NBR_OF_PROBES];
atomic_t __weak dmce_buffer_allocated;
int __weak dmce_anti_recurse_check[DMCE_MAX_NUM_KTHREADS_HOPEFULLY
    ];
int __weak nbr_probes;

static int done_init = 0;

static void dmce_probe_body(unsigned int probenbr) {
#if DMCE_NO_RECURSE
    pid_t kthread_id = current->pid;
    if (unlikely(dmce_anti_recurse_check[kthread_id]))
        return;
#endif

    if (unlikely(!done_init)) {
        if (!atomic_fetch_inc(&dmce_buffer_allocated)) {
            nbr_probes = DMCE_NBR_OF_PROBES; // Propagate to dmce kernel module
            printk("dmce_probe:␣dmce_buffer␣allocated␣at␣address␣with␣size:␣0x%px␣%lu", &dmce_buffer, (long unsigned) DMCE_BUF_SIZE);
        }
        done_init = 1;
    }

#if DMCE_NO_RECURSE
    dmce_anti_recurse_check[kthread_id] = 1;
#endif
    atomic64_fetch_inc(&dmce_buffer[probenbr]);
#if DMCE_NO_RECURSE
    dmce_anti_recurse_check[kthread_id] = 0;
#endif
}
#endif
```

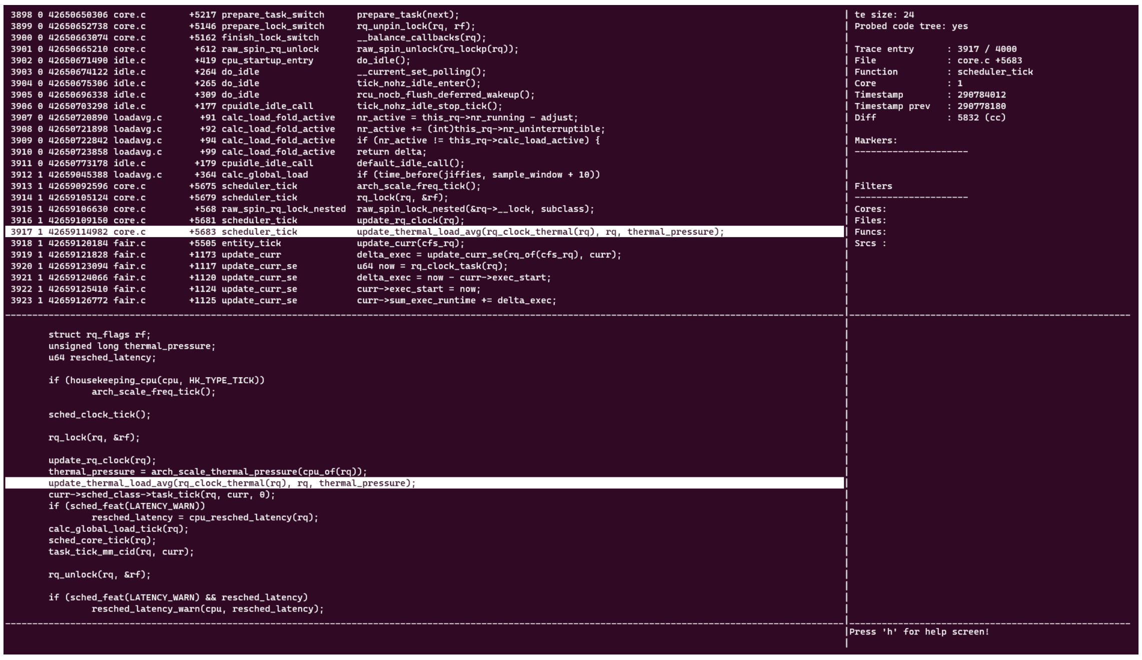
]

Listing C.2: Kernel adapted trace probe implementation

```
#ifndef __DMCE_PROBE_FUNCTION_BODY__
#define __DMCE_PROBE_FUNCTION_BODY__

#include <linux/atomic.h>
#include <linux/slab.h>
#include <linux/smp.h>
#include <linux/ktime.h>
#include <linux/sched.h>
#include <linux/notifier.h>
#include <linux/reboot.h>
#include <linux/panic_notifier.h>
#include <linux/fs.h>
#include <linux/string.h>
#include <linux/compiler_attributes.h>

#define DMCE_MAX_HITS 1000
#define DMCE_NUM_CORES 12

#define DMCE_MAX_NUM_KTHREADS_HOPEFULLY (1024*16)

typedef struct {
    uint64_t timestamp;
    uint64_t probenbr;
    uint64_t cpu;
} dmce_probe_entry_t;

dmce_probe_entry_t __weak dmce_buffer[DMCE_MAX_HITS*DMCE_NUM_CORES];
unsigned int __weak dmce_buffer_indices[DMCE_NUM_CORES];

int __weak no_probe;
atomic_t __weak init_done;
int __weak init_done_local = 0;
int __weak dmce_anti_recurse_check[DMCE_MAX_NUM_KTHREADS_HOPEFULLY
```

```
    ];

int __weak dmce_probe_callback(struct notifier_block *nb, unsigned
    long event, void *_args);
int __weak dmce_probe_callback(struct notifier_block *nb, unsigned
    long event, void *_args){
    WRITE_ONCE(no_probe, 1);
    return NOTIFY_DONE;
}

struct notifier_block __weak dmce_callback_block = {
    .notifier_call = dmce_probe_callback,
    .priority = 10,     //High priority to stop probing quickly
};

void __weak dmce_probe_body(unsigned int probenbr)
{
    pid_t kthread_id = current->pid;
    if(unlikely(dmce_anti_recurse_check[kthread_id])){
        return;
    }

    if(unlikely(READ_ONCE(no_probe))){
        return;
    }

    dmce_anti_recurse_check[kthread_id] = 1;

    if(unlikely(!init_done_local)){
        //init
        if(!atomic_fetch_or(1, &init_done)){
            int ret;
            ret = atomic_notifier_chain_register(&
                panic_notifier_list, &dmce_callback_block);
            if(ret){
                pr_crit("dmce_trace:␣couldn't␣register␣panic␣
                    notifier");
            }
        }
        init_done_local = 1;
    }

    const unsigned int core_id = smp_processor_id();
    const unsigned int index = dmce_buffer_indices[core_id]++ %
        DMCE_MAX_HITS;
    const unsigned int actual_index = core_id*DMCE_MAX_HITS + index
        ;

    dmce_buffer[actual_index].timestamp = ktime_get_mono_fast_ns();
    dmce_buffer[actual_index].probenbr = probenbr;
    dmce_buffer[actual_index].cpu = core_id;

    dmce_anti_recurse_check[kthread_id] = 0;
}
#endif
```

Appendix D
Trace example



```
3898 0 42650650306 core.c        +5217 prepare_task_switch      prepare_task(next);
3899 0 42650652738 core.c        +5146 prepare_lock_switch      rq_unpin_lock(rq, rf);
3900 0 42650663074 core.c        +5162 finish_lock_switch       __balance_callbacks(rq);
3901 0 42650665210 core.c         +612 raw_spin_rq_unlock       raw_spin_unlock(rq_lockp(rq));
3902 0 42650671490 idle.c         +419 cpu_startup_entry        do_idle();
3903 0 42650674122 idle.c         +264 do_idle                  __current_set_polling();
3904 0 42650675306 idle.c         +265 do_idle                  tick_nohz_idle_enter();
3905 0 42650696338 idle.c         +309 do_idle                  rcu_nocb_flush_deferred_wakeup();
3906 0 42650703298 idle.c         +177 cpuidle_idle_call        tick_nohz_idle_stop_tick();
3907 0 42650720890 loadavg.c       +91 calc_load_fold_active    nr_active = this_rq->nr_running - adjust;
3908 0 42650721898 loadavg.c       +92 calc_load_fold_active    nr_active += (int)this_rq->nr_uninterruptible;
3909 0 42650722842 loadavg.c       +94 calc_load_fold_active    if (nr_active != this_rq->calc_load_active) {
3910 0 42650723858 loadavg.c       +99 calc_load_fold_active    return delta;
3911 0 42650773178 idle.c         +179 cpuidle_idle_call        default_idle_call();
3912 1 42659045388 loadavg.c      +364 calc_global_load         if (time_before(jiffies, sample_window + 10))
3913 1 42659092596 core.c        +5675 scheduler_tick           arch_scale_freq_tick();
3914 1 42659105124 core.c        +5679 scheduler_tick           rq_lock(rq, &rf);
3915 1 42659106630 core.c         +568 raw_spin_rq_lock_nested  raw_spin_lock_nested(&rq->__lock, subclass);
3916 1 42659109150 core.c        +5681 scheduler_tick           update_rq_clock(rq);
3917 1 42659114982 core.c        +5683 scheduler_tick           update_thermal_load_avg(rq_clock_thermal(rq), rq, thermal_pressure);
3918 1 42659120184 fair.c        +5505 entity_tick              update_curr(cfs_rq);
3919 1 42659121828 fair.c        +1173 update_curr              delta_exec = update_curr_se(rq_of(cfs_rq), curr);
3920 1 42659123094 fair.c        +1117 update_curr_se           u64 now = rq_clock_task(rq);
3921 1 42659124066 fair.c        +1120 update_curr_se           delta_exec = now - curr->exec_start;
3922 1 42659125410 fair.c        +1124 update_curr_se           curr->exec_start = now;
3923 1 42659126772 fair.c        +1125 update_curr_se           curr->sum_exec_runtime += delta_exec;
```

```
te size: 24
Probed code tree: yes

Trace entry      : 3917 / 4000
File             : core.c +5683
Function         : scheduler_tick
Core             : 1
Timestamp        : 290784012
Timestamp prev   : 290778180
Diff             : 5832 (cc)

Markers:
---------------------


Filters
---------------------
Cores:
Files:
Funcs:
Srcs :
```

```
        struct rq_flags rf;
        unsigned long thermal_pressure;
        u64 resched_latency;

        if (housekeeping_cpu(cpu, HK_TYPE_TICK))
                arch_scale_freq_tick();

        sched_clock_tick();

        rq_lock(rq, &rf);

        update_rq_clock(rq);
        thermal_pressure = arch_scale_thermal_pressure(cpu_of(rq));
        update_thermal_load_avg(rq_clock_thermal(rq), rq, thermal_pressure);
        curr->sched_class->task_tick(rq, curr, 0);
        if (sched_feat(LATENCY_WARN))
                resched_latency = cpu_resched_latency(rq);
        calc_global_load_tick(rq);
        sched_core_tick(rq);
        task_tick_mm_cid(rq, curr);

        rq_unlock(rq, &rf);

        if (sched_feat(LATENCY_WARN) && resched_latency)
                resched_latency_warn(cpu, resched_latency);
```

```
Press 'h' for help screen!
```

Figure D.1: Figure illustrating how a trace can be viewed. The top left segment displays a list of all collected trace events, while the bottom left shows the surrounding code of the probed line.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-06-13

EXAMENSARBETE Did my kernel code execute?
STUDENTER Andreas Bartilson, Jesper Kristiansson
HANDLEDARE Jonas Skeppstedt (LTH), Patrik Åberg (Ericsson)
EXAMINATOR Flavius Gruian (LTH)

Probing the Inner Machinery of the Linux Kernel

POPULÄRVETENSKAPLIG SAMMANFATTNING **Andreas Bartilson, Jesper Kristiansson**

The computer is more prevalent than ever and they all have an operating system. With security becoming increasingly important, so does the ability to perform comprehensive analyses of the systems enabling our digital infrastructure. We present a new way to instrument the Linux kernel by adapting the Ericsson tool *Did My Code Execute?*.

If you have ever used a computer you have undoubtedly used an operating system to run programs to do work or everyday things. To do these things with relative ease, the software must be reliable and safe, which has become increasingly important in the modern digital society. As is true for most things, to improve reliability and safety, we require the ability to analyse the structure and behaviour of systems to identify flaws and risks that need to be fixed.

To analyse programs (specifically those written in the programming languages C and C++), an open-source tool such as Ericsson's *Did My Code Execute?* (DMCE) can be used. By analysing the source code it inserts probes on lines of code that execute simultaneously as the code and can gather (almost) any information we would like. Our thesis applies this approach and adapts DMCE to run in the operating system's code, specifically the Linux kernel.

We successfully adapt DMCE to probe the kernel and show that it is possible to generate a trace of the code execution. Additionally, this can be done without much performance overhead and in cases where the overhead proves substantial, we found that this can be mitigated without a significant loss in accuracy or coverage.

On the way, we found that several problems need to be considered when using this approach on an operating system kernel rather than a normal program. Normally when writing software, there usually exists a standard library of functions that the programmer can use to simplify development. The operating system does not have this standard library, so we explore how the probes, and extraction of data generated by the probes, can be adapted to only use functionality already defined in the operating system's code base. This in turn becomes another problem; what happens if we probe the functionality we are using within the probes? We get stuck in an endless loop as we enter probe after probe. This however we found can be worked around by, given an ID of the thread executing, keeping track of whether we are already inside a probe or not.

So how do we get the information out of the running instrumented kernel? In the case it does not crash, we can use custom kernel modules to utilise the built-in file systems. If it does crash, however, we found that it is possible to dump all memory to a file, and with a known location, we can extract the data we collected.