# Adapting Linux Kernel Test Files to the Kernel Test Framework

Arild Lillegård


Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2019

# Adapting Linux Kernel Test Files to the Kernel Test Framework

Arild Lillegård

Adapting Linux Kernel Test Files to the Kernel Test Framework

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

# Abstract

Frequent testing of the Linux kernel is important to keep quality high. There are several tools and frameworks available for testing in user space. However, the support for unit testing in kernel space is lacking. Test files found in the kernel source tree invent incompatible features for error reporting, assertions and test output instead of using a common framework. The Kernel Test Framework (KTF) attempts to tackle this problem.

In this thesis the framework was explored and evaluated. Four test files from the kernel source tree were examined and converted to the framework. The conversion process was approached in two ways: two test files were converted by a Python script developed for this purpose, and two other test files were manually edited. Comparisons with other frameworks were also made and discussed.

The results showed that the framework can be used for all the four test files. Three test files worked well with the framework, but the fourth received less benefit from the framework due to its structure. Results also showed that a fully scripted conversion is possible using the Python script, but not for every test file. Weaknesses in the framework were identified and reported.

# Contents

# List of Tables

# Acknowledgements

I want to thank my supervisor Knut Omang for all the support and help through this project.

I also want to thank the students I have worked with during my time at the *Institute of Informatics (IFI)* at the university. The students at the program room *Bliss* have long been a great source of motivation and joy through the years.

# Chapter 1

# Introduction

## 1.1 Motivation

Most software of a non-trivial size contains bugs. Developers spend considerable amounts of time on finding and fixing these bugs; time that could otherwise be spent on development.

Testing is a commonly used approach to find and fix bugs at an earlier stage of development, thereby reducing the total time spent on fixing them [12]. By using large collections of well written test cases, we can in some cases uncover defects in a matter of minutes or seconds after they have been introduced, way before they get released into a production system. Even when bugs do slip into a production system, having a well made test suite can reduce the time needed to narrow down the cause of the bug.

However, there are limits to how much we can test. One of the fundamental principles within software testing is that exhaustive testing of a system is impossible [12]. That is, unless the system is trivial in size and complexity, we cannot test every possible state the system can be in. A consequence of this principle is that we have to put on a limit on the amount of resources to use on testing, as resources are limited. At the same time, we also want to get the most out of the time invested in testing.

To reap the benefits of testing with less of the drawbacks, tools and frameworks created for this purpose should be easy and straightforward to use. It is also important that the test suites we write are quick to run, to encourage frequent use. Meeting these condition increases the likelihood of tests being written and used on a regular basis, thus improving the chance of bugs being found and fixed while the code is still fresh in the mind of the developer [14].

The need for testing in the Linux kernel is no less than for other software. With its millions of lines of code and rapid rate of change [10], testing is important to keep quality high. To help its developers and users with this task, several frameworks and test suites have been created to make testing more accessible. The Linux Test Project and kselftest are two of the test suites in current use. These suites are frequently used by test projects such as Linaro's LKFT and Fuego, that test new versions of the kernel as they are released.

However, unit testing inside the kernel is an area that lacks support. This lack can be seen in the many test files in the kernel source tree that does not rely on a shared testing framework; instead, they reinvent the wheel by creating constructs that are similar to each other, but incompatible across the test files.

Therefore, new frameworks have evolved in an attempt to fill this void. One of these frameworks is the Kernel Testing Framework (KTF), made to support unit testing in kernel space, in addition to user space support [1]. This framework provides a set of macros and utilities that are similar to what can be found in traditional unit testing frameworks. But instead of running the tests from user space, KTF compiles tests into kernel modules that can be loaded into the kernel at runtime to test it from within.

## 1.2 Goals

In this thesis we will explore this framework by converting some kernel test files to the framework to see how KTF functionality fits the need of existing test files. A script will also be developed to determine if it can make the conversion process more effective. Linux kernel test files will be converted both manually and by using the script.

## 1.3 Limitations

The scope of this project is mainly limited to the Kernel Test Framework and its use. Although there are other alternatives out there that will be mentioned, they are not the focus of this thesis and will receive far less attention.

There are also features in the Kernel Test Framework that will be left unexplored. This is because these features have not been needed for the test files in this project.

It's also assumed that the reader has some knowledge about the C and Python programming languages.

## 1.4 Thesis outline

**Chapter 1** introduces core concepts related to software testing, and the main technologies used in this project. The technologies explained here are virtual machines, regular expressions and version control systems. Concepts and technologies explained here are aimed at readers who are unfamiliar with these, and can be skipped if they are already known.

The background continues with **Chapter 2**, where information about the Linux kernel is introduced. We will see some of the differences between the Linux kernel and other software, and take a brief look at some aspects of its development. The chapter continues with configuration and building, before mentioning some of the test suites and frameworks that are used by the kernel community.

**Chapter 3** explains the methodology and how to setup the environment used in this project, including framework and dependencies. The chapter ends with instructions on how to manually compile the kernel.

**Chapter 4** introduces the Kernel Test Framework and how to use it. Core features are also explained, along with examples of how they can be used.

In **Chapter 5** we examine the `test_xarray.c` test file and discuss the steps needed to convert it to KTF. We will also introduce the Python script that is used to convert this test file to KTF.

In **Chapter 6** we examine `test_rhashtable.c` and how this file can be converted manually. The chapter will also provide a brief explanation of how two smaller test files can be converted.

We end the thesis in **Chapter 7**, where we summarize and discuss the results from this project. We will also discuss some of the similarities and differences between KTF and other test frameworks. Finally, we will conclude the thesis and provide suggestions for further work.

## 1.5 Contributions

The test files and script used in this project can all be found on github [7].

# Chapter 2

# Background

Before we can begin exploring the research question, we will take a look at some of the technical concepts and terminologies used in this thesis. It is assumed that the reader has some familiarity with both the C and Python programming languages.

## 2.1 Test types

There are multiple approaches to testing a system, depending on what aspects we want to test. We can test if a feature works as intended, with or without knowledge of its implementation, or we can test how well it works. We often also want to retest a component after a new feature or bug fix have been introduced. All these different approaches to testing are called *test types*, and the descriptions below are based on the test types explained in the Foundations of Software Testing book [13].

### 2.1.1 Functional testing

The first and most obvious way to test a system or component is to test it does the right things. That is, to test if it behaves as expected by using its interfaces, and checking the results through its return values and interface methods. This way of testing, without concern for the implementation, is called *black-box testing*, and allows the implementation to change freely as long as the interface stays the same. To be able to test this way, the test writers need to know what results or behavior to expect from the component or system if it works as inteded.

### 2.1.2 Non-functional testing

*Non-functional testing* is used to determine how well a system or component works, and if the minimal requirements are met. This includes testing how fast the system can respond to a request or how many concurrent users it can handle before reaching a certain delay. Like functional testing, a black-box approach is often used as implementation specific features are not relevant for this test type.

### 2.1.3 Structural testing

A third way of testing is to test the internal structure of a test object rather than its interface. With *structural testing* we are more concerned with measuring how well the test object is tested given some metric, for example the percentage of decision branches or statements that are executed in the other tests. If these percentages are too low, this means that we need to write more tests. Because we are testing code that might be difficult to test from the outside using black-box testing, this form of testing can be combined with the other types of testing to increase test coverage even further. This kind of testing is called *white-box* testing, as we are looking "inside" the box.

### 2.1.4 Confirmation and regression testing

The last types of testing that will be explained are *confirmation testing* and *regression testing*. Both types of testing are performed after the code has been changed.

Confirmation testing is performed first to confirm that the bug fix or new feature works as intended. For a bug fix, this is done by rerunning the tests that previously failed, and taking the steps necessary to make sure that the tests passes. Having the tests pass does not guarantee that the code works exactly as intended, but at least it works for the cases we test, increasing confidence in the code.

Regression testing is the next step after the confirmation tests passes. The goal of these tests is to make sure that the newly made changes havn't introduced unintended side-effects, called *regressions*, elsewhere in the system. This type of testing is especially important to do, as letting these new defects slip through will make them harder to fix at a later stage when we no longer know when they were introduced. Furthermore, regression testing should also be performed whenever the environment changes, such as when a library is updated or the database system is replaced.

## 2.2 Test levels

When a system is tested for defects, the testing process is often split up into multiple *test levels*. Each test level focuses on a specific layer of abstraction, ranging from individual functions to treating the whole system as one unit. Although the number of test levels to use depends on the project, we will focus on the four levels specified in the Foundations of Software Testing book [13], due to their widespread adoption.

### 2.2.1 Unit testing

The first and lowest level of testing is *unit testing*, where the focus is to thoroughly test individual units of code isolated from the rest of the system. These units can either be functions, methods, interfaces or whole classes. Unit testing is sometimes also called *component testing*, although this term

can also be used for testing larger units than unit testing. As there is much more to be said about unit testing, this topic will get a more in-depth explanation in section 2.3.

### 2.2.2 Integration testing

While unit testing is about testing individual units of code in isolation, *Integration testing* is used to test how two or more components act when connected. Integration testing is thus more focused on testing the interfaces between components and/or systems, and whether this communication works as intended or not. This process can be done in two main ways; either by integrating components one-by-one and testing between each new integration, or by integrating the whole system at once and then performing the testing. The former approach gives more control when tests fail, as there is less code to search through to find the cause, while the latter approach is quicker if most interfaces are already well tested.

### 2.2.3 System testing

Once the whole system has been fully integrated and all interfaces have been properly tested, *system testing* is used to test the whole system at once. This is done to make sure that the system fulfills all functional and non-functional requirements. Consequently, the environment of the system should also be as close as possible to the live environment it will be used in, although this stage of testing is still done by the developers or testers themselves. System testing also include testing parts of the system that isn't code, such as configuration files and documentation.

### 2.2.4 Acceptance testing

The last level of testing is *acceptance testing*, used to decide if the system is ready for release. This test level is done by the future users of the system, rather than its developers, as the actual users will approach the system in a different way than its developers. Compared to system testing, acceptance testing is a more informal way of testing, with a focus on usability over technical requirements.

## 2.3 Unit testing

In section 2.2.1 unit testing was introduced as the first test level, with a focus on testing individual units of code in isolation. In this section we will take a closer look at this concept, as there is much more to be said about it.

Unit testing a piece of code means that the unit should have tests covering its most common use cases, preferably more if possible. Each test should also be run on a regular basis, to increase the chance of detecting unwanted side effects and regressions at an early stage. In general, the quicker a defect is found, the easier it is to fix.

One of the fundamental building blocks in unit testing is the *test case*. A test case consists of a set of input values, expected output values, execution preconditions and postconditions. They are represented in code as *assertions*, statements that are used to check if assumptions hold true or not. This often takes the form of a comparison between the return value of a function call or a variable, and an expected value. If the comparison evaluates to true, the test case succeeds, otherwise it fails. Unit testing frameworks keep track of both the failed and passed tests, and this is used to determine which assumptions that do not hold.

To keep a good structure, we often organize test cases into multiple *test functions*. Each test function focuses on one aspect of the unit under test, such as testing one specific function or testing a certain class of input for a function. A more concrete example is that to test a function that adds two numbers together and returns the results, we could use one test function for integers, another test function for floating point numbers, and a third function for invalid types.

Finally, we can organize multiple test functions into *test suites*. This allows us to run only certain categories of tests if we don't have the time to run every single one. One common use case is to have separate test suites for regression and confirmation tests that are run after every new bug fix or new feature. Another use case is to have a separate test suite for every subsystem, in addition to a test suite with all the tests.

How many of the tests to run depends on what we want to achieve and the time available. Ideally, we would like to run all the tests every time a change has been made. However, this can take too long with a large collection of tests. There are cases where running every single test for a system can take hours or even days, and then the use of targeted test suites can be useful to save time. When preparing a new release of a live system, a complete test suite can be used to still get the benefits of a full test.

### 2.3.1 Examples

Cmocka is one of the many libraries available for unit testing [19][18]. This library is written in C, and only requires its own files and a few standard library headers to use. Therefore, we will use this library to show how to unit testing can be done in code.

Listing 2.1: Example of a minimal cmocka file

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

static void empty_test_function(void **state) {
  (void)state;  /* Suppress warning about unused argument */
}

int main(void) {
  const struct CMUnitTest tests[] = {
```

```
    cmocka_unit_test(empty_test_function),
  };

  return cmocka_run_group_tests(tests, NULL, NULL);
}
```

We begin by including the necessary header files. Cmocka depends on all the header files shown in listing 2.1, so we include these before the cmocka header. Cmocka must also be installed locally before use, but here we assume that's already done. We then create an empty test function that does nothing. If we wanted to add test cases to this function, we would use one of the assert functions provided by the library [17]. These work much like the assert macro in the standard library, by checking the provided expression and performing some action on failure. But instead of stopping the program, the cmocka variants rather force the current test function to return and report it as a failure in the output. To show how this would affect the final output, we'll add a few test cases to the existing test function:

Listing 2.2: Example of a test function with failing and passing test cases.

```
static void empty_test_function(void **state) {
  (void)state;

  int two = 2;
  assert_int_equal(two, 2);    /* This passes */
  assert_int_equal(two + 3, 10); /* This fails! */
  assert_int_equal(5, 5);      /* Never executed! */
}
```

Given that the name of the file is `tests.c`, we can compile and run the test file the following way:

Listing 2.3: The final output from the examples above.

```
$ gcc -o tests tests.c -L/usr/lib/ -lcmocka
$ ./tests
[==========] Running 1 test(s).
[ RUN      ] empty_test_function
[  ERROR   ] --- 0x5 != 0xa
[   LINE   ] --- test_cmocka.c:11: error: Failure!
[  FAILED  ] empty_test_function
[==========] 1 test(s) run.
[  PASSED  ] 0 test(s).
[  FAILED  ] 1 test(s), listed below:
[  FAILED  ] empty_test_function

1 FAILED TEST(S)
```

We can clearly see from the output in listing 2.3 which assertion that failed. Adding additional test functions would also give us two more lines of output per test function, each with its own FAILED or OK status.

### 2.3.2 The role of unit testing

One use case of unit tests that we have not covered so far is that they can help document how a piece of code is supposed to work. By listing up expected values or side-effects of a function through assertions, the reader can get an idea of what behavior to expect, and potential edge cases to be wary about. Unlike comments, that have a tendency to become outdated over time, well written unit tests provide documentation that hold true as long as the test passes.

Furthermore, unit tests help show how one or more functions can be used together. For example, if one specific function has to be run before another one, like setup or configuration functions, creating a test function that shows this can serve as an example of use. This can useful for reminding the original author of how the code can be used, months or even years after it was first written.

A third benefit of unit tests is that they help refactoring. This is because the unit tests help improve our confidence in that new changes doesn't break the unit, as long as the tests passes. If they fail, we still get some information about where the new defects may be located, given that the *test coverage* isn't too low. Test coverage is used to determine to what degree a specific unit has been tested, and it should be sufficiently high for the tests to be able to detect new defects and failed assumptions effectively.

## 2.4 Test-driven development

*Test-Driven Development (TDD)* [11] is an alternative way of developing software, where the unit tests are written before the code that is tested. This is the opposite of the traditional development approach of writing the code before the test cases. When developing software this way, we repeat the following three-step process:

1. Write a unit test, run it and see it fail. To make this possible, we may have to create an empty function (or method) that returns a value that makes the test fail.

2. Implement the target function in a way that makes it pass, run it and see it pass. This first implementation doesn't have to perfect, it just needs to make the test pass.

3. Refactor the function. If the code within the function is duplicated elsewhere, we may want to refactor those functions as well. Once this is done, run the relevant tests again to make sure that the refactoring didn't break anything.

When following this process, we get a very short feedback loop. Depending on how much code is written between each iteration, an iteration can be as short as a few minutes. This leads to almost instant feedback on the code from the tests, and if written properly, this should help detect bugs and their location quicker. How much code to write

between each iteration depends on both the complexity of the problem and how familiar the programmer is with TDD, but each iteration should be quick regardless.

Whenever we experience duplication of code, we might want to spend some extra time refactoring and generalizing the code. This can occur in both the code and the tests themselves, and in both cases refactoring might be needed. This adds extra time to the feedback loop, but this is time we'll hopefully save in the long run. Overall, the extra refactoring step has the benefit of making refactoring a natural part of the development process, thus increasing code quality.

Another advantage of TDD is that it tends to make us write code with less *coupling*. Coupling refers to dependencies across files, function or classes, and it leads to increased complexity of the code. High degrees of coupling make bug fixing and maintenance more difficult and time-consuming, while also making the code less modular. Also, a change one place in the code base can affect code located anywhere else in the system, and this is thus something we want to avoid if possible. This also makes it more difficult to test a piece of code in isolation. Consequently, writing the tests first can lead to less coupling in both the tests and the code base, as less coupled code is easier to test.

## 2.5 Mocking

*Mocking* is a technique used in testing where one or more components are replaced by simpler *mock objects* to gain better control over the test environment. A mock object has the same interface as the component it replaces, but its implementation is often much simpler; the body of a complicated method may even be replaced with a single return statement that always returns the same value.

A reason for using mock objects is to better isolate a component when unit testing. With ordinary unit testing, a component often still relies on other components, through references or side effects, in order to function. If these dependencies have faults themselves, this can implicitly affect the component under test and make it appear faulty when it's not. Thus, mock objects can be used to guarantee a certain behavior from the dependencies of a component.

For example, let's say we want to create a Network class for communicating over a network. This class could then perform the communication by calling methods on a Socket object. If we want to test how the Network class handles rare error caused by the Socket object, we can create a mock version of the Socket class with the same methods as the original one, but where the behavior is hardcoded for this specific error testing in mind. Then, we can force the mock object to fail in any way we want, without having to reproduce software or hardware errors that might be difficult to reproduce otherwise. This then allows us to better test the error handling of the Network class.

Another reason for using mock objects is performance. When re-

peatedly invoking code with a high startup time, mocking can be used to speed things up when the implementation is of little importance. For example, if we create a class that communicates with a database, the database creation itself can be a huge detriment to the performance of the test suite. Mocking the database can consequently save time and make testing more frequent, if we only care about the communication with the database. Slow test suites can in itself be a threat to thorough and frequent testing, by tempting the developers to either skip tests or run them less frequently.

Nevertheless, a potential drawback of mocking is that the behavior of the mock object can sometimes differ more than intended. If we oversimplify a mock object too much, or if we simulate another component wrongly, mocking can potentially give us a false sense of security. There can also be error situations that may occur in a real situation that we simply don't think about when creating mock objects. As such, mocking can introduce new problems if we are not careful, and it should therefore not be used as a full substitute to testing with real components.

## 2.6 Virtual machines

A *virtual machine (VM)* is a software emulation of a physical machine running an operating system (OS). Virtual machines allow several operating systems to run simultaneously on the same physical machine. For example, a user who wants to use both the Windows and Linux operating systems at the same time can install one of them as the *host operating system*, which is the main operating system, while the other OS runs inside it as a *guest operating system*. Although the guest OS runs as a program inside the host OS, the guest OS still runs as if it was the only operating system running on the machine.

One common use case of virtual machines is to use this technology to let multiple users share the same physical machine. The physical machine itself can located anywhere as long as it is connected to the internet, and each user gets access to their own isolated VM on this machine. For users that don't need powerful hardware, sharing the hardware costs this way can potentially save money by only paying for the resources needed. This is often done by renting one or more VMs from a *cloud provider*, a company specialized in hosting VMs.

Using a cloud provider can also increase scalability of a business, by paying more to get the resources needed. A sudden increase in users can be handled this way, as the cloud provider is responsible for the hardware. Whether this strategy alone is enough also depends on how scalable the software product is, so paying more isn't always enough. However, outsourcing hardware management this way can potentially save both time and money, in addition to provide faster scaling.

Another feature of VMs is the ease of taking *snapshots*. A snapshot is a backup of the current state of a virtual machine, somewhat similar to a traditional backup of an operating system. The main difference lies in the speed of the backup process, as taking a snapshot can often be done in a

matter of seconds. This makes it quite easy to take snapshots before major changes, so that a rollback to a previously working state is quite simple to do if something goes wrong. Consequently, the use of snapshots and rollbacks can be used to speed up recovery.

An added benefit of using a VM is a potential increase in security. Note the use of the word "potential", as there are also risks in sharing the hardware with someone else; we'll come back to that in the next paragraph. Security can be increased by the extra layer between the guest OS and the hardware. If the guest OS is compromised, control of the entire system isn't necessarily lost, as a compromised VM doesn't have complete control over the underlying hardware. Also, a compromised VM can be replaced with an uncompromised one if the problem is discovered. Furthermore, a compromised VM won't affect the other VMs running on the same hardware, unless the malware manages to escape into the host OS.

Malware escaping from a VM is one of the risks referred to in the previous paragraph. If the piece of malware is sophisticated enough, it may be able to discover that it is running inside a virtual machine. If this is the case and there are security vulnerabilities inside the virtualization software, full control over the host operating system can be obtained, putting the other guest OSes at risk. Sharing hardware in the cloud thus poses some additional risks, although it's difficult to escape a VM.

One of the disadvantages of VMs is that the performance of a guest OS may be cut to a fraction of what it would otherwise be. Running an entire OS inside another one increases the total workload, and as such, the performance is reduced. Also, the instructions to be executed are emulated, further increasing the overhead. The performance footprint of the last point can be addressed by hardware support for virtualization, often present in modern hardware, however, this still doesn't remove the fact that reduced performance is one of the main disadvantages of VMs.

## 2.7 Regular expressions (Regex)

*Regular expressions* [54] [52], or *regex* in short form, is a powerful tool used for text processing. It's often used for locating or replacing patterns in a string of text, using a compact syntax to specify a pattern to look for. This pattern can consist of a combination of ordinary characters with literal meaning and metacharacters with special meaning. Together they form what may look like a random combination of characters, although the patterns do make sense upon close inspection.

There are several different syntaxes for writing regular expressions, depending on the tool or programming language used. Here we will focus on the syntax used by the `re` module in Python, although the differences in syntax between tools and languages are small.

First we have the literal characters, such as 'a', '9' or ' '. These represent themselves, so the regular expression `"foo = 2"` will match the string `"foo = 2"` if it's present in the text. This is also the case for control characters like `\n` for newline or `\t` for tab.

Secondly we have the special metacharacters. These characters do not represent literal characters, but rather have other purposes like wildcard matching, making certain groups of characters optional or creating groups of characters that should not occur in the pattern. When using the `re` module mentioned earlier, the following characters are treated as metacharacters: `. ^$ + ? { } [ ] \| ( )`. It's still possible to match these characters literally, but this requires a preceding \ to escape them.

As a certain understanding of regular expressions can be useful for the later chapters, we will quickly go through each of the metacharacters to explain what they do. The `.` character is the closest one can get to a wildcard character. It matches all characters except newline by default, and can consequently be used for capturing arbitrary content found between two characters. For example, to capture the contents between a pair of `{}`, the regex `"\{.\}"` can be used. Note that the curly braces have been escaped with a preceding \, as otherwise the expression would have a different result.

Next, a pair of `[]` can be used to specify a set of characters that can occur next. The string `"[abc0-9]"` means that the next character can either be 'a', 'b', 'c' or a digit from zero and up to nine. This set can also be negated by placing the `^` as the first character in the set, like `"[^abc0-9]"`. In that case, the pattern has the opposite effect by capturing any character that is not found in the set.

We also have the `()` used as a *capture group*. A capture group can be used for multiple purposes, such as to split up a regular expression into smaller logical units, specify a subgroup that can be fetched individually or to be used together with a *quantifier* to specify how many times a subgroup or character should be repeated. This can either be done with `*` for zero or more repetitions, `+` for one or more repetitions, `?` for zero or one repetition, or a lower and upper bound inside a pair of `\\`. Also, the string `"[abc]"` can potentially be rewritten as `"(a|b|c)"`.

To wrap up, here's an example using most of the features shown so far; writing the regex `"[+-]?[0-9]+(.|,)?[0-9]{1,5}"` would match a string "beginning with an optional `+` or `-`, followed by one or more digits from zero to nine, then an optional period or comma, and ending with one to five digits".

## 2.8 Version control systems

A *version control system (VCS)* [22] [23] is a system that keeps track of all the files in a project, including the changes to these files through the entire lifespan of the project. A VCS often rely on a central *repository* where the files are stored. This central repository is used by all the participants to keep their own, local repositories synchronized with other ones.

When a developer updates one of these files locally, the changes can be marked for distribution to the other participants through a *commit*. These changes can then be synchronized into the central repository through a subsequent *push*. The other developers can download these changes and

update their local repositories through a *pull*. By explicitly needing to push and pull updates, two or more developers can modify the same file at the same time without sudden interference.

As long as the users update different parts of the same document, changes are merged automatically by the system. However, if two developers update the same parts of the same document without an intermediate pull, this can lead to a *merge conflict*. In that case, the conflict must be resolved manually by one of the developers.

Another important feature of a VCS are the *branches*, parallell versions of the same project. Branches can be used to develop multiple features on the same project in isolation, in order to avoid having changes on one branch affect the other branches until the feature is complete. For example, a developer working on a new feature or a bug fix can create a new branch for this purpose and not have his files be modified by developers working on other branches. This developer can still push and pull changes from other developers on the same branch, but changes from other branches are ignored. Once the feature or bug fix is complete, this branch can be *merged* with the other branches; at that point any potential merge conflicts are handled. The benefit of working this way is to be able to postpone merge conflicts until the work on a feature or bug fix is complete.

An extra benefit of using a VCS is that it also functions as a backup service, but with the added possibility of reverting the project back to any previous version. This feature can be useful if a breaking change is introduced and a developer wants to revert it completely. There is also the option of downloading the full project at once if a local repository is made inacessible. Consequently, a VCS also serves as a backup tool in addition to code sharing.

For a more information about version control systems in general, see [22].

# Chapter 3

# Linux Fundamentals

The previous chapter introduced several core concepts related to testing, virtual machines and regular expressions. Having some knowledge about this terminology is crucial to be able to follow the thesis.

In this chapter we will begin the discussion around the research question, by introducing some aspects of the Linux kernel and its development. This will be done to create a context for the following chapters. The information presented here will be kept at an overall level, to avoid going too much into detail.

We will first take a brief look at what an operating kernel is and how the Linux kernel is different from other software. We will then examine some aspects of its development, configuration and building. Finally, we will look at some of the ways the kernel is tested today.

## 3.1 What is an operating system kernel?

An operating system *kernel* makes up the core parts of an operating system, providing the most important features that an operating system needs in order to function. A kernel usually does not include preinstalled programs or the graphical user interface that a user can see, but rather features that the user cannot see and takes for granted, such as scheduling processor cycles between multiple processes, the file system, handling keyboard input, hardware drivers and so on. The number of core features that are handled by the kernel itself, rather than by other parts of the operating system, depends on the type of the kernel used. The Linux kernel is a *unikernel*, meaning that most of the operating system features are handled by the kernel. A *microkernel* on the other hand, keeps the kernel small by outsourcing many tasks to separate programs instead.

The Linux kernel makes up the core parts of several operating systems, including the Linux distributions and the Android mobile OS. The Linux distributions, which refers to a family of operating system variants built around the Linux kernel, are widely used by software developers, large companies, and most of the websites and large supercomputers today [5] as an alternative to the more well known Windows and Mac operating systems.

## 3.2 What differentiates Linux from other software?

### 3.2.1 Kernel mode

One of the major differences between the Linux kernel and other large software systems is that the kernel runs in *kernel mode*. Kernel mode allows every processor instruction to be used and the whole memory to be accessed. This is in contrast to *user mode*, used by normal programs, where only some of the memory and a subset of the processor instructions are available. Bugs in code that runs in kernel mode can potentially hang the computer or open up for security breaches, increasing the importance of testing and early bug discovery.

Another consequence of kernel mode is that there are certain precautions to take when testing kernel code. When ordinary user mode programs have finished execution, the operating system will make sure that the resources used are properly released afterwards. For example, all memory used by a program will be released when the program terminates, even dynamic memory that should be freed explicitly. Also, files that have not been properly closed will be handled by the operating system, in case the programmer has forgotten to do so. This sort of automatic cleanup is not performed for code running in the kernel, and this includes unit tests running in kernel mode. These resources, like heap memory and kernel data structures, must be freed manually; forgetting to do so can potentially crash the kernel in the worst case.

### 3.2.2 Open source

Another difference between Linux and some of the other software, is the *open source* nature of the project. Open source means that the source code of the software is publicly available, making it readable by anyone. A common argument for this way of developing software is that there are potentially more eyes on the code that may discover bugs and security holes. The open source nature of Linux also means that more people have the chance to contribute to the code.

### 3.2.3 Highly configurable

The large number of configuration options available makes the kernel highly configurable. This flexibility grants freedom to the user, but it also comes with a cost; it is practically impossible to test that every combination works as intended, leading to bugs that are not unconvered by tests. Consequently, it's often the default and most used configurations that get tested for each supported architecture.

To give a short summary of the challenges with testing, an article published in 2006 [14] stated that:

> The open source development model and Linux in particular introduces some particular challenges. Open-source projects

> generally suffer from the lack of a mandate to test submissions and the fact that there is no easy funding model for regular testing. Linux is particularly hard hit as it has a constantly high rate of change, compounded with the staggering diversity of the hardware on which it runs. It is completely infeasible to do this kind of testing without extensive automation. [14]

## 3.3 Development

The Linux kernel consists of millions of lines of code, and it continues to increase in size as new features are added. The development of the kernel is a collective project driven by over 3000 developers [42] from all around the world; this stands in constrast to large software systems developed by a single large company. Thus, both the philosophy behind the project, and the way it is developed, differs from other large projects.

### 3.3.1 The Linux kernel mailing list

The *Linux kernel mailing list (lkml)* is one of the main means of communication between kernel developers, featuring discussions, announcements and sharing of new patches [30]. There are multiple mailing lists used by the kernel community, and the many subsystems within the kernel often have their own mailing lists in addition to the main one. A mailing list is comparable to a internet forum, but instead of using a webpage to host the discussions, e-mails are used instead. These e-mails are sent to the subscribers of that particular mailing list, although online archives also exist.

### 3.3.2 Development trees

New features and bug fixes will propagate through multiple maintainer trees before it's considered ready for release. While some developers are working on a kernel version that is nearly ready for release, other developers are simultaneously implementing new features for the subsequent release candidate. There are also developers that work on older kernel versions that still needs bug fixes and security updates, even though those kernels were originally released several years ago. To enable this parallell development, the kernel community uses multiple *development trees* [31] for the different kernel versions. Each development tree consists of one or more repositories controlled by a version control system, most often `git`.

In addition to having development trees for the different kernel versions, the many subsystems within the kernel also have their own internal development trees. Each of these subsystem trees have a maintainer that is responsible for merging incoming changes for that particular tree. When a developer wants to apply patches to that subsystem tree, he or she will post the patch on a mailing list for review; if the patch is accepted, the maintainer applies the patch to the repository of that tree.

### 3.3.3 The single versus group maintainership models

As mentioned in the previous subsection, many of the subsystems within the kernel have traditionally used a single maintainer model. However, in the last few years there have been discussions about the scalability of this model [27][49] [46]. As seen in these articles, some argue that the single maintainer model can become a bottleneck in a busy subsystem, or that the model can lead to delayed updates if the maintainer is either overloaded or busy elsewhere. Others claim that group maintainership would not work for their subsystem, due to either the amount of patches that are rejected, coordination problems that can follow, or a lack of submaintainers that can be trusted with full commit rights [49]. The full discussion is outside the scope of this thesis, but the articles cited earlier in this paragraph can be read for more information.

### 3.3.4 Configuration and building

#### Make

The first build system we'll look at is *make* [25][16]. The *make* system allows the user to build a project with the `make` command, given that the current directory contains a *Makefile*. A Makefile contains the necessary steps to build a project from its source files, and can either be written manually or be generated by another program. Makefiles also specify the relationship between files, allowing the *make* system to determine which source files that needs recompilation. Furthermore, *make targets* can be used to create custom commands for that specific project; for instance, a `clean` target is often created for deleting temporary files, and this command can be used by writing `make clean`.

#### Kconfig and Kbuild

Before the kernel can be compiled, it must first be configured. The kernel features a large number of configuration options that can be used to fine-tune the kernel for the user's specific needs; however, the default configuration is often good enough. The kernel is configured with the *Kconfig* [20][59] tool found in the `scripts/kconfig` directory of the kernel source code. This tool provides several interfaces to choose between for performing the configuration; it features both text-based, menu-based and graphical alternatives. To use Kconfig, run the command `make <interface-name>`, where `<interface-name>` must be replaced with one of the names listed below (for example `make menuconfig`):

- config: Command-oriented
- nconfig: ncurses menu-based
- menuconfig: menu-based
- xconfig: Qt-based frontend

After the kernel has been configured, the *Kernel Build System (Kbuild)* [21] is used to build the kernel into a runnable executable. Without going into much detail, Kbuild uses the `.config` file from Kconfig, together with a hierarchy of Makefiles, to recursively build the relevant kernel components; which components to build depends on the chosen configuration. Once all the components have been built, the result is a Linux kernel image file (kilde (linuxjournal artikkel? <www.linuxjournal.com/content/kbuild-linux-kernel-build-system>)).

#### Cmake

*Cmake* is a cross-platform build for generating build files for the platform used [50]. While Kconfig and Kbuild are used for building the kernel, *cmake* is often used for user space software. The build files it generates is specific for the platform it's run on, so on Linux it produces Makefiles. Consequently, *Cmake* doesn't compile the project itself, but leaves that to a platform specific build system like *make*.

As *cmake* generates a large amount of temporary files, this tool should be run from a *build directory*, separate from where the source files are located. For instance, if the source files of a project is located in `~/src/foo`, *cmake* should be run from `~/build/foo`, with the source directory path as an argument. Both the *make* and *cmake* systems are used in section 4.3.

## 3.4 Kernel modules

The Linux kernel supports *loadable kernel modules (LKMs)* to extend the kernel with new functionality dynamically [33]. This means that the modules can be loaded into the kernel while it's running, to add or replace functionality without needing to restart or recompile the whole operating system. The modules are instead compiled separately and loaded or unloaded manually by the user.

## 3.5 Testing

Early and extensive testing is key to keep quality high. The earlier a bug is discovered, the quicker it generally is to fix. One reason for this is that the code is still fresh in the mind of the developer [14]; another reason is that there are less fresh code to search through to find the cause of the problem.

Several test suites and tools have been made in an attempt to achieve this goal. Many of them have been developed independently of each other and began their life as simple scripts, later evolving into test suites. These test suites contain numerous types of tests, some of which are (section 4.2 in [14] mentions several of these categories):

- Build tests: tests used to check for problems when building the kernel on multiple architectures and configurations.

- Static verification tests: used to discover problems through static analysis of the code.

- Functional and unit tests: used for functional and unit testing, as explained in section 2.1.1 and 2.2.1.

- Performance tests: used to measure the performance or stability of a component. Two examples include testing the disk or network performance.

- Stress tests: used to check how the system performs when pushed close to its limits.

- Profiling and debugging: used to gather information about exactly what the system or component is doing.

Ideally, all new code should be tested on every supported architecture and configuration before being added. However, this is practically impossible due to the countless combinations of hardware and kernel configurations possible. Accessible test tools and frameworks can improve the situation, by lowering the barrier of entry for developers to test their own code.

### 3.5.1 Problems of fully testing the kernel

One of the seven principles in software testing states that exhaustive testing is impossible [12], at least for software of a non-trivial size. This is especially true for a project of the size of the Linux kernel for a variety of reasons.

One reason for the difficulty of testing the kernel is that '... there is no good way to test it except running it', as stated by Greg Kroah Hartman in a Google Tech Talk in 2008 (14:27-14:36 in [26]). He further stated that due to the number of hardware and configuration combinations possible, in addition to all the possible ways to interact with it, they rely on the developers and users of Linux to test it on their machines. He also mentioned that one cannot test everything with unit tests. (14:36-15:42 in [26]).

Another reason is the lack of an overarching test strategy. (kilde trengs!) The developers are expected to test their code before releasing it, but there are no common test framework used by all developers. Instead, there are a multitude of different frameworks and test suites out there, in addition to all the bash scripts and individual test files made to fulfill the needs of that specific developer. Furthermore, there is also the issue of tests being created by companies that keep them for themselves, for various reasons. The result is a number of scripts, tools and frameworks that do not cooperate well, tests that aren't shared and tests that are difficult to use by other developers.

### 3.5.2 Testing tools and frameworks

We will now take a look at some of the tools and frameworks available for testing kernel code. These tools and frameworks will only get a brief introduction, as this section is meant to give an impression of what is available. Another reason for this is also the lack of documentation related to their degree of adoption.

#### kselftest

*Kselftest* [41][34] [57] is a test suite used for regression testing of the kernel; however, it's focus is rather to be used as a quick sanity check by developers than to extensibly test the kernel for regressions [34]. Kselftest is located under `tools/testing/selftests` in the kernel source tree, but can also be installed and used on a running kernel with a few commands.

#### The Linux Test Project (LTP)

The *Linux Test Project (LTP)* [45] [14] is a test suite used for functional and regression testing of the kernel (kilde). It contains over 3000 test cases and is capable of testing a number of aspects of the kernel, including the file system, memory, system calls, network etc. Due to the number of test cases available, LTP is also used by some of the frameworks mentioned below. LTP provides a black-box approach to testing the kernel.

#### Avocado

*Avocado* [51][53] is an automated testing framework written in Python and serves as a successor of the Autotest [6][9] [8] framework which it replaces. The Avocado framework is capable of running tests written in Python, as well as any executable as long as it returns 0 on success or a non-zero value otherwise. One of the goals of the framework is to provide a powerful, but simple framework, as developers otherwise could choose to create their own testing scripts instead of using the framework. Both [53] and [51] can be seen for a more thorough introduction.

#### IBM Autobench

*IBM Autobench* [61][14] is a proprietary tool used for testing and detecting performance regressions between releases. It regularly checks for new kernel releases and patches that are automatically downloaded, built and benchmarked. While benchmarking a kernel, statistics are gathered and later compared to earlier bechmarks to detect regressions over time. A job file is used to tell Autobench what to do, and is written in its own custom language and a combination of bash and perl scripts.

### Linux Kernel Functional Testing (LKFT)

Linaro's *Linux Kernel Functional Testing (LKFT)* [4][39] is a framework for functional testing of new Linux kernel releases. Like IBM Autobench, LKFT automatically downloads, builds and tests new versions of the kernel for regressions whenever they are released. The framework targets several development branches, including 4.4, 4.9, 4.14 and 4.19 [44], although these branches are not the only ones. The tests are run on multiple environments and four different architectures, namely arm32, arm64, x86_64 and i386 at the time this was written [58]. The framework uses test cases from other testing tools, such as **kselftest**, **Linux Test Project (LTP)** and **Libhugetlbfs**; in total over 20.000 tests are run per kernel [39].

### Fuego

*Fuego* [40] is another test framework for testing new kernel releases, but is geared more towards embedded devices. Fuego comes with a number of tests and wrappers for building, deploying and running them, and uses a *container* to reduce the number of installation issues. A container is a lightweight alternative to a virtual machines, with less overhead. Like the LKFT framework, Fuego uses test cases from other frameworks in addition to its own, including LTP and kselftests. Test cases written for these frameworks are automatically obtained by Fuego. See the presentation at [40] for more information.

### Kernel Test Framework

*The Kernel Test Framework (KTF)* [35] [24][47] [48] is a new unit testing framework aimed at white-box testing of the kernel. It does this by compiling test files into kernel modules that are inserted into the kernel during runtime. This approach enables tests to execute in the live kernel environment, without need for mocking. As this framework will be the main focus of this thesis, it will get a more in-depth introduction in chapter 5.

### KUnit

*KUnit* [37][38] is another new unit testing framework, targetting some of the same areas as KTF, although with a different approach. It provides several of the same features, but instead of running the tests inside a kernel running on real or emulated hardware (VM), KUnit runs its tests inside *User Mode Linux (UML)*. UML is a Linux architecture that compiles into a program to be run as a user level program. This approach to testing comes with both advantages and disadvantages, which we will we discuss later in subsection 8.3.3.

# Chapter 4

# Planning and preparing the project

In the previous chapter we examined some aspects of the Linux kernel and its development. We also took a brief look at a few of the testing tools that are currently used.

This chapter will be used to describe the chosen approach and how to prepare an environment for kernel testing.

## 4.1 Approach

The focus of this project will be the use of the Kernel Test Framework, the process of converting kernel test files to the framework, and the difficulties that arise. The experiments will take place on a virtual machine running Linux, and relies on the Kernel Test Framework, Googletest and Python being installed. We will approach the conversion process in two ways: by using a script and by manual editing. The former approach will be done by using a Python script to modify the files `lib/test_xarray.c` and `lib/test_sort.c`, while the latter approach will be done by manually converting `lib/test_rhashtable.c` and `lib/test_string.c`. Both approaches will require some analysis of the test files, but to varying degrees.

### 4.1.1 Virtual machine setup and management

For reasons that will be explained in subsection 4.1.2, the experiments will take place on a virtual machine hosted in the `UH-IaaS` cloud platform, based on the OpenStack technology [15]. The virtual machine itself will run Ubuntu Linux version 18.04 LTS, but the exact version of the kernel itself may vary through the project as the system is kept up-to-date. The SSH technology will be used for remote-controlling the virtual machine, and administration will be done through the `UH-IaaS` web interface. Although the exact specifications of the virtual machine is unknown, the following resources should be available:

- 1 Virtual CPU

- 4 GB of RAM
- 40 GB of storage

### 4.1.2 Why use a virtual machine?

There are several reasons for using a virtual machine for testing kernel code; the first reason being that virtual machines provide an easy way to create backups through the snapshot feature. A snapshot is quite literary a snapshot of the current state of the virtual machine. Using this feature before any major changes are done, enables us to quickly go back to a previously working state if we accidentally mess up the virtual machine. This includes accidental deletion of important files or changes that are difficult to revert in other ways. Also, unlike a full backup of the file system on a regular machine, creating a snapshot of a virtual machine is usually a fast action to perform. Consequently, the snapshot feature of virtual machines can save time when things go wrong.

Another advantage of virtual machines is the speed of recovery. If we manage to crash the entire kernel, restarting a virtual machine can often be faster than restarting a physical machine, although this depends on machine in question. Kernel panics and lack of response are problems that may be encountered in this project, and due to the time this can save in the long run, quicker restarts is another reason to use a virtual machine.

However, a potential disadvantage of using a virtual machine is the extra resources a virtual machine requires. This is due to the additional overhead caused by emulating a physical machine inside an operating system, slowing down both the host and guest operating systems. To partly circumvent this problem, the virtual machine used here will run in the cloud, to get most of the benefits with less of the performance penalty on the local machine. Still, the extra overhead will likely have an effect on the performance of the test files.

### 4.1.3 Data collection and analysis

As mentioned in the beginning of this chapter, the main focus of this thesis is the use of KTF, as well as an evaluation of both the conversion process and the framework. As such, there is a limited focus data collection, although some will be still collected.

The limited data collected during this project will be related to the test files being modified. This data includes the original length of the test files, the conversion approach and the resulting number of `TEST` functions in the modified files.

### 4.1.4 Limitations of the chosen approach

The limited resources of the virtual machine may have an effect on the experiments, such as the time taken to run the test suites or problems occuring due to lack of memory or processing time. Also, because kernel configurations may be dependent on the underlying hardware, the kernel

configuration on one machine will most likely differ from that used on another, potentially affecting the results of test files in general.

## 4.2 Preparing the virtual machine

### 4.2.1 VM security

Before installing any required tools, a few steps should be taken to improve the security of the virtual machine. When running a machine that is accessible from the Internet, be it physical or virtual, one should expect that it will be attacked as soon as it goes online. The more services that are accessible from the outside, the more attack vectors can be exploited using vulnerabilities in said software. Therefore, we will take some measures to reduce the likelihood of the virtual machine being compromised:

#### Closing unused ports

The first measure is partly done already by the default security policy used by OpenStack. By default, all ports are inaccessible from the outside unless manually opened. However, as we are going to use SSH for remote controlling, at least one port needs to be opened. We could either choose to use the default port or make SSH use another one. The latter could deter the attacks from some of the simpler attack performed by the simplest attack scripts, although it is doubtful that it will help much against most attackers, who would probably use port scanning to find the SSH server port anyways.

#### Harden SSH server

A second security measure is to install a few tools to automatically block repeated requests from the same IP adresses. This is yet another small measure that will increase the security against less sophisticated attacks, but not necessarily stop the more sophisticated ones. As we can see in listing 4.2.1, it did not take long before connections from other machines was attempted. In the beginning, the packets apparently came from one of two IP adresses, however, soon more IP adressses began to appear in the log file.

```
Aug 28 13:12:55 kernel-testing sshd[1656]: Invalid user pi from
    189.45.79.185
Aug 28 13:12:55 kernel-testing sshd[1656]:
    input_userauth_request: invalid user pi [preauth]
Aug 28 14:06:15 kernel-testing sshd[13802]: Invalid user
    default from 5.188.10.7
6
Aug 28 14:06:15 kernel-testing sshd[13802]:
    input_userauth_request: invalid user
 default [preauth]
Aug 28 14:06:16 kernel-testing sshd[13802]: Connection closed
    by 5.188.10.76 port 45404 [preauth]
```

```
Aug 28 14:06:17 kernel-testing sshd[13804]: Invalid user ftp
    from 5.188.10.76
Aug 28 14:06:17 kernel-testing sshd[13804]:
    input_userauth_request: invalid user ftp [preauth]
Aug 28 14:06:17 kernel-testing sshd[13804]: Connection closed
    by 5.188.10.76 port 53213 [preauth]
Aug 28 14:06:18 kernel-testing sshd[13807]: Invalid user guest
    from 5.188.10.76
Aug 28 14:06:18 kernel-testing sshd[13807]:
    input_userauth_request: invalid user guest [preauth]
Aug 28 14:06:18 kernel-testing sshd[13807]: Connection closed
    by 5.188.10.76 port 38360 [preauth]
```

To try to counter this, the `fail2ban` and `denyhosts` programs were installed to automatically block such requests. Then some changes was done to the `/etc/ssh/ssh_config` file to further strengthen security, such as disabling root login and requiring SSH keys instead of passwords for authentication.

## 4.3 Installing dependencies and googletest

The Kernel Testing Framework, which will be introduced in chapter 5, is a separate framework that can be compiled and installed on a precompiled kernel. Before downloading and installing the framework, make sure the following dependencies are installed:

- git
- cmake
- g++
- googletest

On debian based systems, both `git`, `cmake` and `g++` can be installed with the `sudo apt install <package>` command, where `<package>` is replaced with the name of the package.

Next, clone and build googletest by following the steps in listing 4.1:

Listing 4.1: The necessary steps to clone and build googletest from scratch

```
# Create a source directory for the repository.
mkdir ~/src                         # Create source directory
    if necessary
cd ~/src
git clone https://github.com/knuto/googletest.git

mkdir ~/build/‘uname -r‘            # (Optional) Create a
    separate build
                                    # directory for compilation
                                        and building
cd ~/build/‘uname -r‘
```

```
mkdir googletest
cd googletest
cmake ~/src/googletest              # Generate a Makefile
make                                # Compile and build
sudo make install                   # Final installation step
```

For those new to this way of building and installing programs, we are essentially downloading the source code from a repository, configuring it automatically through cmake to generate a `Makefile`, and then compile the program locally using said `Makefile`. Lastly, `make install` is used to finish the installation.

The reason for using ‘`uname -r`‘ in the path is that this ensures that both KTF and the test suites must be rebuilt if the Linux version changes. This is not necessarily the case for `googletest`, although we still do it for consistency. This also enables us to have one source directory, and multiple build directories for each project. For example, both `googletest`, KTF and the test suites all have one source directory each under `~/src`, in addition to one build directory per Linux version previously used.

## 4.4 Installing KTF

After the dependencies have been handled, clone the repository found at `https://github.com/knuto/ktf` into a desired directory, for example `~/src/ktf`. The official installation instructions can be found at [2], but they will also be shown here along with potential problems and fixes. The steps taken are quite similar to the ones used for googletest, and are shown in listing 4.2.

Listing 4.2: How to clone and build KTF from scratch

```
cd ~/src
git clone https://github.com/knuto/ktf # Clone the directory
cd ktf                              # Enter the new directory
    ...
autoreconf                          # ... and configure it.

cd ~/build/‘uname -r‘               # (Optional) If separate
    build directory
mkdir ktf                           # (Optional) Same as above
~/src/ktf/configure KVER=‘uname -r‘ # Generate a Makefile
make                                # Compile and build
sudo make install
```

## 4.5 Potential problems

This section will contain a list of problems that may be encountered when installing KTF and its dependencies, along with solutions.

### 4.5.1 Missing C++ compiler

The following error message indicates that a C++ compiler is missing: `"No CMAKE_CXX_COMPILER could be found"`. This also means that no Makefile has been generated. Install g++ as shown in section 4.3.

### 4.5.2 Autoconf missing

If `autoreconf` doesn't work, make sure to install `autoconf` with `sudo apt install autoconf` if you use a debian based system.

### 4.5.3 Autoreconf locale warnings

If everything goes well when running autoreconf, it shouldn't give any output. One of the issues encountered when running this command locally for the first time, was several repetitions of this output:

```
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
    LANGUAGE = (unset),
    LC_ALL = (unset),
    LC_PAPER = "nb_NO.UTF-8",
    LC_ADDRESS = "nb_NO.UTF-8",
    LC_MONETARY = "nb_NO.UTF-8",
    LC_NUMERIC = "nb_NO.UTF-8",
    LC_TELEPHONE = "nb_NO.UTF-8",
    LC_IDENTIFICATION = "nb_NO.UTF-8",
    LC_MEASUREMENT = "nb_NO.UTF-8",
    LC_TIME = "nb_NO.UTF-8",
    LC_NAME = "nb_NO.UTF-8",
    LANG = "en_US.UTF-8"
    are supported and installed on your system.
perl: warning: Falling back to a fallback locale ("en_US.UTF-8").
```

This output was repeated up to 6 times when fisrt running the command, but a fix was found here: [29]. The reason was apparently that the ssh connection was sending environment variables to the virtual machine, and this was fixed by commenting out the line containing the string `SendEnv LANG LC_*` in the file `/etc/ssh/ssh_config` file.

### 4.5.4 Missing packages

Other errors can also appear if some packages are missing. For example, if running make without the the package `libnl-genl-3-dev` installed, one can get the following output:

```
/usr/bin/ld: cannot find -lnl-genl-3
collect2: error: ld returned 1 exit status
```

Lacking the package `pkg-config` can for some reason also cause this problem. Make sure to check that as well.

### 4.5.5 Configure failed

Like CMake, the configure command checks that all requirements for the project are met. If one or more requirements are missing, no Makefile is generated. If this happens, make sure that the `libnl-3-dev` og `pkg-config` are installed.

## 4.6 Compiling a kernel from source

Precompiled kernels suffice for everyday use; however, there are cases where manual compilation is needed. For example, when working with the `xarray` data structure in chapter 6, a newer version of the kernel was manually compiled to get access to the header files of the data structure. These header files were not present on the current system, as the data structure was too new. Compiling a new kernel solved that problem. Therefore, this section will be used to explain the kernel compilation process, based on the instructions from [36]. [28] and [42] can also be checked for alternative instructions.

### 4.6.1 Installing required packages and downloading the kernel source tree

Make sure the following packages are installed before proceeding [36].:

- libncurses5-dev
- gcc
- make
- git
- exuberant-ctags
- bc
- libssl-dev

Next, enter a suitable source directory (like `~/src/linux-stable`) and clone the kernel source code with
`git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`.
The repository cloned with this command contains a *stable* [65][60] version of the kernel, meaning that the kernel is considered ready for release. The repository address can be replaced with another one if a different kernel version is desired.

### 4.6.2 Configuration, building and installation

Before we proceed to the building phase, the kernel must be configured. This can either be done with the command shown in subsection 3.3.4, or by reusing the configuration file of the currently used kernel; we will focus on the latter approach for simplicity. This also means that all new configuration options will be set to their default values.

Listing 4.3 shows the whole process. After the kernel source code has been cloned, the configuration file of the current kernel is copied to this directory. Then, `sudo yes "" | sudo make oldconfig` makes sure that the current configuration file is reused, with all new configuration options are enabled. Finally, `sudo make` compiles the kernel files, while `sudo make modules_install install` finishes the whole process.

Listing 4.3: How to clone and configure the kernel using the old configuration file.

```
cd ~/src
git clone
    git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git

# Copy existing configuration file.
cd linux-stable
sudo cp /boot/config-‘uname -r‘ .config

# Set new configuration options to "yes"
sudo yes "" | sudo make oldconfig

# Compile and build.
sudo make
sudo make modules_install install
```

### 4.6.3 Running the new kernel

If the whole process succeeds, a list of all the installed kernels should appear when restarting the machine. Otherwise, the newest kernel will likely be used by default.

# Chapter 5

# Kernel Test Framework

In the previous chapter we looked at how to install the Kernel Test Framework (KTF) and its dependencies. This chapter will introduce the framework and show to use it. We will then examine some of its core features, including contexts and fixtures.

## 5.1 Introducing the Kernel Test Framework

As briefly introduced in section 3.5.2, the *Kernel Test Framework (KTF)* [35] [24][47] [48] is a new framework that attempts to make unit testing in the kernel easier and more accessible. By testing the kernel from kernel space, with access to exported and non-exported APIs [1], code paths that are difficult to trigger from user space can be tested. The framework is not meant to replace the existing test frameworks, but rather provide a different approach to kernel testing. Tests are compiled into kernel modules that can be loaded into the kernel during runtime. This approach allows kernel specific headers to be accessed.

One of the goals of KTF is to make test-driven development in the kernel more accessible [1]. By providing a framework created with this in mind, it should lower the bar of entry to this way of development in the kernel. Test-driven development encourages testing and refactoring to be an integral part of the development process. This could potentially contribute to more kernel bugs to be found at earlier stage, as well as provide documentation to kernel code in the form of tests.

Another goal of KTF is to allow developers to run tests on precompiled kernels, where recompilation of the kernel is either not feasible or possible. By allowing tests to inserted into the kernel at runtime, instead of being compiled into the kernel itself, new kernel space tests can be written and used on precompiled kernels. This approach also enables kernel data structures to be tested without the hassle of bringing them to user space [1]. KTF is currently built out-of-tree, but this could change in the future.

### 5.1.1 Creating, building and running a new test suite

Before we can begin testing, we need to create a test suite. KTF lets us do this by using the `ktfnew` command found under `~/src/ktf/scripts/`. The script takes the desired name of the test suite as a parameter and creates a directory under `~/src/` with this name. This new directory will contain all the necessary files to begin testing.

When entering the directory just created, we see a number of new files, most of which can be completely ignored. The only file we care about is the one with the same name as the test suite, located in the `kernel/` subdirectory. Listing 5.1 shows the layout of a new test suite.

Listing 5.1: The structure of the test suite directory after running `ktfnew`

```
$ ~/src/ktf/scripts/ktfnew mysuite
    Creating a new project under /home/ubuntu/src/mysuite
$ ls ~/src/
    googletest ktf mysuite
$ ls ~/src/mysuite/
    ac         autom4te.cache configure.ac m4         Makefile.in
    aclocal.m4 configure      kernel       Makefile.am
$ ls ~/src/mysuite/kernel/
    Makefile.in mysuite.c
```

We will take a look at the default test file in 5.1.2, but for now we just want to run it as it is.

#### Compiling and building

The next step is to compile and build the tests into a kernel module. Although we can use a shared directory for both source and build files, a cleaner solution is to use a separate directory for the build files. By doing this we can reuse the same source code each time we need to rebuild the test suite. As kernel modules must be built for the current version of the kernel, we will likely need to rebuild the test suite multiple times, even if we never modify the tests themselves.

Listing 5.2 shows the steps required to build a test suite into a separate build directory. In this example we assume that both the `build` directory and its subdirectory already exist. Note that we use the command ‘`uname -r`‘ as part of the path; this command outputs the current version of the kernel we are using, and updating the system may change this value. By creating and using a subdirectory named by this command, we are forced to rebuild our test suites whenever the kernel version changes. We will see why in subsection 5.2.2. Once we have created and entered the right directory, we call `configure` once to generate a Makefile and some other files that are used to build the test suite. We then call `make` every time we want to rebuild the test suite.

Listing 5.2: Compiling and building the test suite

```
$ cd ~/build/‘uname -r‘
```

```
$ mkdir mysuite                  # If not already done
$ cd mysuite/
$ ~/src/mysuite/configure KVER=‘uname -r‘ # Only used once
$ make                                    # Used for every rebuild
```

### Loading and running

The next step is to load the newly built kernel module into the kernel. Make sure that the `ktf.ko` module is inserted first, as otherwise we will get the following error message: `insmod: ERROR: could not insert module kernel/mysuite.ko: Unknown symbol in module`. The KTF module only needs to be inserted once after every reboot of the system; once it's inserted, it will stay in the kernel until it's either unloaded or the system shuts down. Next, we insert the test module into the system using the `insmod` command, as shown in listing 5.3. Unlike the KTF module, however, the test module should be unloaded, rebuilt and then inserted again if we change the test code. Also note that two versions of the same test module cannot be loaded at the same time. Finally, we use the `ktfrun` command to run all the loaded test suites.

Listing 5.3: Loading and running the `mysuite` test suite

```
$ cd ~/build/‘uname -r‘/mysuite
$ make                              # After every change.
$ sudo insmod ../ktf/kernel/ktf.ko  # After every reboot.
$ sudo rmmod kernel/mysuite.ko      # If already inserted.
$ sudo insmod kernel/mysuite.ko     # Insert module.
$ ../ktf/user/ktfrun                # Run all tests.
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from simple
[ RUN      ] simple./t1
[       OK ] simple./t1 (0 ms)
[----------] 1 test from simple (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (1 ms total)
[  PASSED  ] 1 test.
```

## 5.1.2 A look at the default test file

Now that we have created and tried to run the new test suite, we will take a look at the default test file. The test file can always be found under the `kernel` subdirectory, and it has the same name as the test suite. If we open `mysuite.c`, we will see the following code:

```
#include <linux/module.h>
#include "ktf.h"

MODULE_LICENSE("GPL");
```

```
KTF_INIT();

TEST(simple, t1)
{
    EXPECT_TRUE(true);
}

static void add_tests(void)
{
    ADD_TEST(t1);
}

static int __init mysuite_init(void)
{
    add_tests();
    return 0;
}

static void __exit mysuite_exit(void)
{
    KTF_CLEANUP();
}

module_init(mysuite_init);
module_exit(mysuite_exit);
```

The file begins by including the KTF and module system headers, as both headers are needed for KTF to function. `KTF_INIT()` is then called outside the functions to initialize the framework and add KTF specific code. Likewise, `KTF_CLEANUP()` is called in the exit function to safely release the resources used once the test completes execution. Proper cleanup after execution is especially important inside the kernel, as the resources used must be manually freed. After the initialization code, the `TEST` macro is used once to define a KTF test function. This macro takes two parameters: the name of the test suite, and the name of the new function; both names will appear in the KTF output. This function is then added to KTF by using the `ADD_TEST` macro.

To see how we can edit the test file, we will add two new test functions; one function will succeed and the other function will fail:

```
...
TEST(simple, t1)
{
    EXPECT_TRUE(true);
}

// first new test function
TEST(simple, t2)
{
    EXPECT_FALSE(true);
}
```

```
// second new test function
TEST(foo, bar)
{
    EXPECT_TRUE(1 == 1);
}

static void add_tests(void)
{
    ADD_TEST(t1);
    // register both functions so KTF can use them.
    ADD_TEST(t2);
    ADD_TEST(bar);
}
...
```

In order to rebuild the test suite, we run `make` from the build directory. Then we unload the old version of the module with `rmmod`, before we reload it with `insmod`. If we try to reload the new module straight away, we get the following error:

```
$ sudo insmod kernel/mysuite.ko
insmod: ERROR: could not insert module kernel/mysuite.ko: File
    exists
```

The whole rebuild process is shown below:

```
$ cd ~/build/‘uname -r‘/mysuite
$ make
$ sudo rmmod kernel/mysuite.ko
$ sudo insmod kernel/mysuite.ko
$ ../ktf/user/ktfrun
[==========] Running 3 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 1 test from foo
[ RUN      ] foo./bar
[       OK ] foo./bar (0 ms)
[----------] 1 test from foo (0 ms total)

[----------] 2 tests from simple
[ RUN      ] simple./t1
[       OK ] simple./t1 (0 ms)
[ RUN      ] simple./t2
/home/ubuntu/build/4.4.0-134-generic/mysuite/kernel/mysuite.c:17:
    Failure
Failure '!(true)' occurred
[  FAILED  ] simple./t2, where GetParam() = "t2" (0 ms)
[----------] 2 tests from simple (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 2 test cases ran. (1 ms total)
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] simple./t2, where GetParam() = "t2"
```

KTF comes with several example test files. These can be found under `/build/‘uname -r‘/ktf/examples` and is worth a look. For more information about KTF and its features, see the official API in [3]

### 5.1.3 ASSERT_* vs. EXPECT_* assertions

KTF divides its assertion macros into two categories: `ASSERT_*` and `EXPECT_*` assertions. Both assertion categories evaluates an expression to decide if a test case succeeds or fails; the differences lies in how failures are handled. The first category will stop execution of the current function and either return or jump to a label upon failure. This is useful if the function cannot continue if a test case fails. The second category of assertions will not have any effect on execution if a test case fails.

## 5.2 Using KTF features on a kernel data structure

To give a short and consise introduction to some of the tools KTF offers, we will take a look at how we can write a very simple test suite for a kernel data structure: `struct rhashtable`. This data structure will get a more thorough introduction in chapter 7, but for now we can think of it as a generic hash table with concurrency support and automatic resizing. See [62] for an alternative introduction.

### 5.2.1 Setting up a rhashtable

Before we take a look at how we can create a rhashtable for our own use, we need to determine what we want to store in it. We can store whatever data we want in it, as long as the following requirements are fulfilled:

- The elements stored should be a struct, as certain fields must be present.
- The struct type stored must have a field that can be used as a key. This field can have any type.
- The struct type stored must also have a field with the `struct rhash_head` as its type. This field is used internally by the rhashtable functions.

The struct can of course include more fields than this, such as a reference counter, other fields for lifetime management, or multiple data fields. However, such fields will be left out in the following examples for simplicity. Instead, we will define and use the type `struct object` to represent a data type with the minimal amount of fields needed to be stored in the rhashtable.

Listing 5.4: The data storage struct, `struct my_data`, and the element type, `struct object`

```
struct my_data {
    int data;
};

struct object {
    int key;
    struct rhash_head head;
    struct my_data data;
};
```

As the rhashtable is a generic data structure, we also need to fill out a parameter struct to tell the rhashtable functions how to treat our data. The definition of the parameter struct can be seen in listing 5.5, however most of the fields are optional. The fields that must be filled are listed below:

- `u16 head_offset`
- `u16 key_offset`
- `u16 key_len`

The `head_offset` and `key_offset` fields are used to tell the rhashtable functions where the key and the `struct rhash_head` fields are located. `key_len` is also required, as we can use any type as a key. By leaving out the rest of the fields in the `struct rhashtable_params`, their default values are used.

Listing 5.5: The original definition of `struct rhashtable_params`

```
struct rhashtable_params {
    u16         nelem_hint;
    u16         key_len;
    u16         key_offset;
    u16         head_offset;
    unsigned int        max_size;
    u16         min_size;
    bool            automatic_shrinking;
    u8          locks_mul;
    u32         nulls_base;
    rht_hashfn_t        hashfn;
    rht_obj_hashfn_t    obj_hashfn;
    rht_obj_cmpfn_t     obj_cmpfn;
};
```

Now that we got that covered, we can finally create a rhashtable with the minimal options required, as shown in listing 5.6. In this example, all of the test code is placed inside a single function. The rhashtable is initialized with `rhashtable_init` and later cleaned up with `rhashtable_destroy`. `rhashtable_init` returns -EINVAL on failure, so we will use a check on the return value as an assertion. Although this is a simple example, it shows how a test function in KTF can look.

Listing 5.6: Creating a minimal test suite for the `struct rhashtable` type

```
#include <linux/module.h>
#include <linux/rhashtable.h>
#include "ktf.ko"

MODULE_LICENSE("GPL");

KTF_INIT();

struct my_data {
    int data;
};

struct object {
    int key;
    struct rhash_head head;
    struct my_data data;
};

TEST(rh_init, t1) {
    struct rhashtable my_table;
    struct rhashtable_params rht_params = {
        .head_offset = offsetof(struct object, head),
        .key_offset = offsetof(struct object, key),
        .key_len = sizeof(int),
    };
    int success = rhashtable_init(&my_table, &rht_params);
    EXPECT_TRUE(success != -EINVAL);

    rhashtable_destroy(&my_table);
}

static void add_tests(void) {
    ADD_TEST(t1);
}
static void __init rhashsuite_init(void) {
    add_tests();
    return 0;
}
static void __exit rhashsuite_exit(void) {
    KTF_CLEANUP();
}

module_init(rhashsuite_init);
module_exit(rhashsuite_exit);
```

### 5.2.2 Recompilation after updating the system

It's important to note that test suites must be rebuilt after the system is updated. When kernel modules are built, they are built for the exact kernel they will be loaded into. This restriction was intentionally added to the kernel build system, to prevent users from inserting modules that are built

for another kernel. If such modules were inserted, they could break other parts of the kernel and generate problems that can be avoided by having this restriction.

If a user is unaware of this detail, and attempts to insert a kernel module built for an older kernel, the following error message will appear:

```
$ sudo insmod ../ktf/kernel/ktf.ko
insmod: ERROR: could not insert module ../ktf/kernel/ktf.ko:
    Invalid module format
```

This retriction applies to all kernel modules, and not just KTF. Whenever the installed kernel changes through an update, KTF itself must also be rebuilt in addition to the test suites. As mentioned earlier, we want to rebuild KTF and the test suites under a new `‘uname -r‘` directory to detect such changes easier. Listing 5.7 shows how to rebuild KTF in this way, while 5.8 shows to rebuild the test suite.

Listing 5.7: How to rebuild KTF after an update.

```
$ cd ~/build
$ mkdir -p ‘uname -r‘
$ cd ‘uname -r‘
$ mkdir ktf
$ cd ktf/
$ ~/src/ktf/configure KVER=‘uname -r‘
$ make
$ sudo make install
```

Listing 5.8: How to rebuild the test suite after an update.

```
$ cd ~/build/‘uname -r‘
$ mkdir rhashsuite
$ cd rhashsuite
$ ~/src/rhashsuite/configure KVER=‘uname -r‘
```

### 5.2.3 Setup and teardown in KTF

Before we proceed with the rhashtable examples, we will introduce a new feature that might prove useful for setup and teardown code in the test environment. In the rhashtable examples earlier, we performed everything from setup to teardown code inside the same `TEST` function. For a simple test case such as the one above, that is fine. However, if we have multiple test functions that rely on the same code for setup and teardown, there is a better way. KTF includes support for this through *contexts* and *fixtures*.

Contexts and fixtures are both used to provide an environment outside the test functions themselves. Instead of using global variables to create this environment, we can use these features instead for a cleaner approach. The main difference between the two features in KTF, is that a context is created once and shared between several test functions, either because the setup and teardown code is expensive to run, or because we want to continue

altering the same environment across test functions; fixtures on the other hand are created and destroyed between every test function that uses it. Thus, fixtures can be used to emulate the 'setup' and 'teardown' functions that unit test frameworks often provide.

### 5.2.4 Using fixtures

As mentioned above, fixtures allow multiple functions to share the same code for setup and teardown. For instance, one might want all test functions to begin execution with the exact same "global" state. This state can then be initialized inside a fixture setup function and cleaned up in a fixture teardown function, that are executed before and after every test function is run.

KTF provides the following macros for handling fixtures:

- DECLARE_F(fixture_name)
- SETUP_F(fixture_name, setup_function_name)
- TEARDOWN_F(fixture_name, teardown_function_name)
- INIT_F(fixture_name, setup_function_name, teardown_function_name)

The example below shows how these macros can be used:

```
#include <linux/module.h>
#include <linux/rhashtable.h>
#include "ktf.h"

MODULE_LICENSE("GPL");

KTF_INIT();

struct my_data {
    int data;
};

struct object {
    int key;
    struct my_data data;
    struct rhash_head head;
};

static struct rhashtable_params rht_params = {
    .head_offset = offsetof(struct object, head),
    .key_offset = offsetof(struct object, key),
    .key_len = sizeof(int),
};


// Fixture setup
DECLARE_F(fixture_test)
    struct rhashtable my_table;
```

```
};

SETUP_F(fixture_test, fsetup)
{
    int success = rhashtable_init(&fixture_test->my_table,
        &rht_params);
    fixture_test->ok = true;
}

TEARDOWN_F(fixture_test, fteardown)
{
    rhashtable_destroy(&fixture_test->my_table);
}

INIT_F(fixture_test, fsetup, fteardown);

... // the rest of the file is shown in the next example.
```

The `DECLARE_F` macro is used to declare the fixture. All variables defined here we will be accessible inside functions defined with `TEST_F`. Note the lack of an introducing `{` after `DECLARE_F(fixture_test);` this is not a typo, but intended.

`SETUP_F` is then used to initialize the fixture. The code written here is run before every `TEST_F` function is called. Inside `SETUP_F` and `TEARDOWN_F`, the fixture members can be accessed through a struct with the same name as the fixture itself - in this case the struct can be accessed through the symbol `fixture_test`, as seen in the listing. Inside `TEST_F` functions, the symbol `ctx` is used instead. In addition to taking the fixture name as the first parameter, `SETUP_F` and `TEARDOWN_F` both take a function name as the second parameter. Finally, `INIT_F` is called to tie the fixture together.

`TEST_F` functions can now be declared in the same way as the `TEST` functions, apart from the extra argument and the usage of `ctx`:

```
...

TEST(simple, t1)
{
    EXPECT_TRUE(true);
}

TEST_F(fixture_test, ts, f1)
{
    struct object obj = {
        .key = 1,
        .data = {123},
    };
    EXPECT_TRUE(atomic_read(&ctx->my_table.nelems) == 0);

    rhashtable_insert_fast(&ctx->my_table, &obj.head,
        rht_params);
    EXPECT_TRUE(atomic_read(&ctx->my_table.nelems) == 1);
}
```

```
TEST_F(fixture_test, ts, f2)
{
    EXPECT_TRUE(atomic_read(&ctx->my_table.nelems) == 0);
}

static void add_tests(void)
{
    ADD_TEST(t1);
    ADD_TEST(f1);
    ADD_TEST(f2);
}

static int __init fixture_test_init(void)
{
    add_tests();
    return 0;
}
static void __exit fixture_test_exit(void)
{
    KTF_CLEANUP();
}

module_init(fixture_test_init);
module_exit(fixture_test_exit);
```

As seen in the example, the same name is used as the first argument to `TEST_F` as the other fixture related code. This is important in case there are more than one fixture declared in the same file. The other two arguments are the same as for `TEST` functions: the name of the test suite, and the function name. Once again, the fixture data can be accessed through the `ctx` name.

### 5.2.5 Using context

As opposed to fixtures, a `context` allows the same state to be shared among several test functions without storing all the variables in the global namespace. Instead a context struct can be created and used to store the global state for the functions sharing the same context. This can also be done without utilizing the context related features, although they offer some degree of abstraction.

To create a context, define a struct that contains at least a `struct ktf_context` field. Then declare an instance of this struct which will be used by `KTF_CONTEXT_ADD` to keep track of the state:

```
struct my_ctx {
    struct ktf_context k;
    int counter;
};

static struct my_ctx some_ctx = { .counter = 1 };
```

```
static int __init context_test_init(void)
{
    KTF_CONTEXT_ADD(&some_ctx.k, "data");
    add_tests();
    return 0;
}
```

The `char *` argument `"data"` is used as an identifier for the context, and it's used by `KTF_CONTEXT_GET` and `KTF_CONTEXT_FIND` to find a pointer to the right context object. `KTF_CONTEXT_FIND` returns a pointer to the whole struct, in this case a `struct my_ctx`, while `KTF_CONTEXT_FIND` returns a pointer to the `struct ktf_context` object within. The latter macro is used for cleanup together with `KTF_CONTEXT_REMOVE`, which should be called in the exit function of the test suite. If this is not done, the module will be kept in use after the test suite has finished execution, giving the following error message when trying to `rmmod` it: `rmmod: ERROR: Module module_name is in use`.

We will finish of this chapter with a full example of how contexts can be used:

```
#include <linux/module.h>
#include "ktf.h"

MODULE_LICENSE("GPL");

KTF_INIT();

struct my_ctx {
    struct ktf_context k;
    int counter;
};

static struct my_ctx some_ctx = { .counter = 1 };


TEST(simple, t1)
{
    struct my_ctx *data_ctx = KTF_CONTEXT_GET("data", struct
        my_ctx);
    struct my_ctx *no_ctx = KTF_CONTEXT_GET("invalid", struct
        my_ctx);

    EXPECT_TRUE(data_ctx != NULL);
    EXPECT_TRUE(data_ctx->counter == 1);
    data_ctx->counter++;

    EXPECT_TRUE(no_ctx == NULL);
}

TEST(simple, t2)
{
    struct my_ctx *data_ctx = KTF_CONTEXT_GET("data", struct
```

```
        my_ctx);
    EXPECT_TRUE(data_ctx->counter == 2);
    data_ctx->counter += 3;
}

TEST(simple, t3)
{
    struct my_ctx *data_ctx = KTF_CONTEXT_GET("data", struct
        my_ctx);
    EXPECT_TRUE(data_ctx->counter == 5);
}

static void add_tests(void)
{
    KTF_CONTEXT_ADD(&some_ctx.k, "data");

    ADD_TEST(t1);
    ADD_TEST(t2);
    ADD_TEST(t3);
}

static int __init context_test_init(void)
{
    add_tests();
    return 0;
}
static void __exit context_test_exit(void)
{
    struct ktf_context *pctx = KTF_CONTEXT_FIND("data");
    KTF_CONTEXT_REMOVE(pctx);

    KTF_CLEANUP();
}

module_init(context_test_init);
module_exit(context_test_exit);
```

# Chapter 6

# Converting test_xarray by script

In the following chapter we will explore the `test_xarray.c` file and see how it's structured. We will then discuss how we can modify the file to make it work within the Kernel Test Framework. Once that is done, we will take a look at how we can automate the conversion process by using a Python script.

## 6.1 Introduction

XArray [43] [63][67] [66] [64] is a kernel data structure that attempts to provide the convenience of a resizeable array, but with more features and better performance on multiple areas. This includes being cache-friendly, provide efficient dynamic resizing and enable lookups without locking. The underlying data structure is based on the radix tree data structure, but with a new interface [63]; it essentially works like an array of pointers.

The XArray type offers two APIs: the normal API and the advanced API. The former is simpler and easier to use, whereas the latter offers more flexibility and better performance. The advanced API also requires the user to handle locking manually, which is handled automatically by the normal API.

## 6.2 Core functions and macros

There are two ways to create and initialize a new XArray, either by static or dynamic allocation; use `DEFINE_XARRAY` for static allocation or `xa_init` for dynamic allocation. To use the advanced API, the `XA_STATE` or `XA_STATE_ORDER` macro should also be used to create and initialize a `struct xa_state`. This helper data structure will contain both the XArray itself and several other fields used to improve the efficiency of use.

For storing data, the `xa_store` function stores the new entry and returns the old element if it exists. To only insert an entry if the index is unused, use `xa_insert` instead as it returns -EEXIST if the index is non-empty. `xa_alloc`

can be used instead to store an entry at an unused index. Deletion can be done by a call to `xa_erase` or to `xa_store` with a NULL pointer as data. Retrieving an entry can be done with `xa_load`.

Finally, iteration is done with either `xa_for_each`, `xa_find` or `xa_find_after`. `xa_extract` can be used to convert the XArray into an ordinary array, and `xa_destroy` removes all entries. Memory allocated in the entries should be manually freed before calling `xa_destroy` to avoid memory leaks.

## 6.3 A look at the test_xarray.c file

Before we discuss exactly how to modify the file, we will take a brief look at how it's written. One of the first things of notice is the clean structure, as if the file was written for a test framework. This is due to the following reasons: 1) each function specializes on one or a few specific features to test, with little setup or teardown code required; 2) assertion logic is handled by an assertion macro instead of using if-tests together with counters and print statements; 3) the function parameters are exactly the same for nearly every function, with a few exceptions; 4) the return value of the test functions themselves are irrelevant; and 5) the test functions are independent and called sequentially in the main function. All of these properties should make it easier to adapt the file to KTF.

Execution begins in the `xarray_checks()` function, whose main job is to call the test functions. The main function is shown in listing 6.1, although most function calls are removed for clarity. The `&array` parameter is a pointer to a statically allocated struct defined just above this function, and the struct is used by most test function as a shared test object.

Listing 6.1: The main function of test_xarray.c

```
static int xarray_checks(void)
{
    check_xa_err(&array);
    check_xas_retry(&array);
    ...
    check_store_range(&array);
    check_store_iter(&array);

    check_workingset(&array, 0);
    check_workingset(&array, 64);
    check_workingset(&array, 4096);

    printk("XArray: %u of %u tests passed\n", tests_passed,
        tests_run);
    return (tests_run == tests_passed) ? 0 : -EINVAL;
}
```

Unlike several other test files found in the kernel, `test_xarray.c` utilizes a single macro, `XA_BUG_ON`, for nearly every assertion statement in the file; there are a few exceptions where kernel macros are used instead,

but that is rare. The `XA_BUG_ON` macro is defined in the top of the file, and is shown in listing 6.2. This macro takes a `struct xarray *xa` and an assertion expression as parameters, and this information is used to perform an assertion check, print debug information if the check fails, and increment a few counters when needed. In that sense, the macro performs a few of the core tasks of a testing framework, but in a short and consise way.

Listing 6.2: Original definition of XA_BUG_ON

```
#undef XA_BUG_ON
#define XA_BUG_ON(xa, x) do {                   \
    tests_run++;                                \
    if (x) {                                    \
        printk("BUG at %s:%d\n", __func__, __LINE__); \
        xa_dump(xa);                            \
        dump_stack();                           \
    } else {                                    \
        tests_passed++;                         \
    }                                           \
} while (0)
```

To show an example of how the test functions can be structured, we will look at the function `check_xa_err`. As we can see in listing 6.3, the function consists entirely of assertions with state modifications done inside them. This is how assertions and state modifications are done in the other functions as well, although some of them also perform state modifications outside the `XA_BUG_ON` calls as well.

Also worth mentioning is that the function signature is kept the same for every test function except `check_workingset` and `check_xa_alloc`. They are all defined as `static noinline void`, something we can take advantage of later. They also take the same `struct xarray *xa` argument as well. Consequently, there are several similarities between the functions that can be used in regular expressions.

Listing 6.3: Example of a small test function.

```
static noinline void check_xa_err(struct xarray *xa)
{
    XA_BUG_ON(xa, xa_err(xa_store_index(xa, 0, GFP_NOWAIT)) !=
        0);
    XA_BUG_ON(xa, xa_err(xa_erase(xa, 0)) != 0);
#ifndef __KERNEL__
    /* The kernel does not fail GFP_NOWAIT allocations */
    XA_BUG_ON(xa, xa_err(xa_store_index(xa, 1, GFP_NOWAIT)) !=
        -ENOMEM);
    XA_BUG_ON(xa, xa_err(xa_store_index(xa, 1, GFP_NOWAIT)) !=
        -ENOMEM);
#endif
    XA_BUG_ON(xa, xa_err(xa_store_index(xa, 1, GFP_KERNEL)) !=
        0);
    XA_BUG_ON(xa, xa_err(xa_store(xa, 1, xa_mk_value(0),
        GFP_KERNEL)) != 0);
```

```
    XA_BUG_ON(xa, xa_err(xa_erase(xa, 1)) != 0);
}
```

## 6.4 How to convert the file

Now that we have an idea of how the test file is structured, we will discuss the changes necessary to make the file work with KTF. As we will focus on the main points of interest, there may be certain changes that won't be addressed here; these will be handled later.

### 6.4.1 Handling assertions

As mentioned in the previous section, the `XA_BUG_ON` macro is used for nearly every assertion in the file. Clearly, this should ease the convertion of assertions, as there are mainly one assertion pattern to worry about.

There are two ways of handling the `XA_BUG_ON` macro; we can either keep the macro and replace its body, or we can replace all its occurences instead. If we keep the macro, the whole body can be replaced with a single KTF assertion, as the debug code within it is redundant when using KTF. When the debug code is gone, the macro is no longer needed. Consequently, we will go for the latter approach of replacing the macro entirely. As for what to replace the macro calls with, `EXPECT_FALSE` should be favored over `ASSERT_FALSE` since execution should continue if an assertion fails.

### 6.4.2 Choosing functions to redefine with `TEST`

The next step is to decide which functions that should be redefined with the `TEST` macro. The ideal situation would be to redefine every test function using the macro, but this isn't possible in the macros current state. The main reason being that the macro doesn't accept any user specified parameters. This limitation is likely made because a well written test function shouldn't need any arguments; any additional data should be provided by a fixture or context instead.

Luckily, most test functions in this file only take a `struct xarray *xa` as the only argument. This pointer can easily be provided by a context, which solves this problem for most functions.

There are functions with unique parameter lists that aren't good candidates for this solution, mainly helper functions and the `check_workingset` function. Although the extra parameters could be provided by a context, this is neither a scalable solution nor how a context should be used. Consequently, there are some functions that aren't suited for the `TEST` macro. Subsection 6.4.5 will provide a more detailed discussion of how these exceptions can be handled.

Another requirement for using the `TEST` macro is that the return type of the function must be `void`. Again, this requirement is met for most of the test functions, except the helper functions. Test functions that meet both

requirements can be redefined with `TEST`, while the rest of the functions should not. This includes every test function called in listing 6.1, with `check_workingset` as the only exception.

### 6.4.3 Similarities between test functions

There are some additional similarities between the test functions that should be mentioned. For example, all functions defined in this file are `static`, many of them are also `noinline`, and every function that is not a helper returns `void`. The functions also share the same `check_` prefix, and all but two of them take `struct xarray *xa` as the only argument; the exceptions are `check_xa_alloc` that takes no arguments, and `check_workingset` that takes two.

### 6.4.4 Letting helper functions use EXPECT_FALSE

As we concluded with in subsection 6.4.1, the assertion statements will be replaced with calls to `EXPECT_FALSE`. However, this macro only works if it has access to a `struct ktf_test *self`. This pointer is provided by the `TEST` macro, and will consequently not be accessible in the helper functions by default. To solve this problem, every non-`TEST` function should receive this as an extra parameter.

### 6.4.5 Handling unique parameter lists

As we discussed in subsection 6.4.2, there are a few test functions that differ from the majority. `check_workingset` is the most notable one, and we will thus discuss how it should be handled.

The main problem with this function is its additional parameter, as we still want to redefine this function like the other ones. In addition to the extra argument, there are also three calls to this function, each with a different integer as argument. As `TEST` can't take extra arguments, we must find a workaround.

One potential solution is to create a dummy `TEST` function containing all three function calls. As long as we add the extra context code inside it, this solution could work. The extra `struct ktf_test *self` parameter must still be added to the old function, but that shouldn't be a problem. However, a disadvantage with this solution is that if one of the function calls fail when running the test suite, we won't which of them it was.

Another solution is to create one dummy `TEST` function for each of the three function calls. Each dummy could share the name of the original function, but with a numeric suffix at the end. This solution would give us the benefits of the previous solution, in addition to better output from KTF. The main drawback is the lack of scalability and the increased size of the code. With only three extra test cases, however, this solution will suffice. Listing 6.4 shows how this could be done for one of these calls.

Listing 6.4: Potential solution for handling extra arguments to test functions

```
TEST(test_xarray_rewrite, check_workingset_3_) {
    struct array_context *actx = KTF_CONTEXT_GET("array",
        struct array_context);
    struct xarray *xa = actx->xa;

    check_workingset(self, xa, 4096);
}

static noinline void check_xa_alloc(void) {
    ...
    ADD_TEST(check_workingset_3_);
    ...
}
```

## 6.5 Introducing the conversion script

Now that we know what to change, we will see how a Python script can be used to perform the changes for us. The goal of this approach is to examine if this approach can speed up the conversion process. Ideally, the script should be capable of fully converting the file, given the right parameters. If not, it should at least be able to execute some of the steps, to reduce the amount of manual editing required.

### 6.5.1 The three iterations of the script

The script underwent three iterations before reaching its final form. The first iteration consisted entirely of regular expressions written for `sed` [56], a tool used for transforming and filtering text. Initially, the idea was to use this tool to speed up the conversion process by using regular expressions for most of the modifications; the few remaining modifications could then be done manually. This appeared like a suitable approach, mainly due to the many repeatable patterns in the file. Several single-line modifications also showed promising results for this approach, until multiline patterns were encountered. Although `sed` does have some support for multi-line editing, its cumbersome way of handling this task led to a new attempt in Python instead.

The second iteration was similar to the first: the plan was once again to use regular expressions for most of the conversion, and use manual editing for the last part. To improve the structure of the script the regular expressions were placed into functions, and a small amount of additional logic were added as well. Initially no additional state was stored, so certain subregexes had to be run multiple times to get the desired result; one example is the regular expression used for finding all static functions in the file. As additional state and logic was introduced to better accomodate this, the limitations of a pure regex approach grew clearer. As a result of

this, a third iteration was initiated.

To better handle the limitations of the previous solution, the new `Converter` class was introduced. Initially, the class only consisted of the previously used regex functions and some extra state related to function names in the test file. As already mentioned, certain regular expressions were used multiple times in other regular expressions, to find information relevant for multiple tasks; this could now be done once in the constructor of the class and be stored for later. However, this solution also turned out to be insufficent, due to difficulties in distinguishing test functions from helper functions. Therefore, the class was changed to provide this and other information as parameters instead.

As the class gradually received more parameters, the script became better suited to handle other test files as well. This increased generality also changed the goal of the script; the script should be able to handle most test files with a certain structure, and not just `test_xarray.c`. And if the script can't convert the whole file, it should be able to perform at least some of the process.

### 6.5.2 Class `Converter`

The class revolves around three main components: a set of *data* describing the desired transformations, a set of regular expressions, and a set of format strings. The data are provided by the user in the form of a *dictionary* - a *Hash Map* in other programming languages - as one of the parameters to the contructor. The contents of this parameter is then stored in the object for future use. The other two components are built into the class and are used by the its methods. These methods must be called manually by the user, and it's required for the class to modify the test file.

Most of the methods follow a three-step process for modifying the test file. First, one or more format strings are retrieved, depending on the complexity of the method. These strings are stored in the class and they are used for code generation. Second, the format strings are filled out with data that the user provided earlier; the result is one or more pieces of code that will inserted into the test file next. Third, a regular expressions is retrieved from a dictionary of regular expressions in the class. This regular expression is then used to replace a pattern in the test file with the new code.

Each method that is called modifies the same internal representation of the test file. This string is set in the beginning of the constructor and it will contain the converted test file once all the desired methods have been called. The `result` method must be called last to write this string to the specified output file. We will see an example of how to use the class after the `data` parameter has been explained.

There are currently 13 fields in the `data` dictionary that are supported; these fields are all shown and explained in the list below. Most of the fields are optional, and how many fields to fill depends on how much work the user wants the script to do. Some of the fields also have default values associated to them. For example, the `"init_code"` field that is

used to specify initialization code for KTF have a default value. Unless the user wants to provide additional setup code for contexts or fixtures, this field should be ignored; the default value contains the setup code required. Several fields are used directly in regular expressions and require the substring `\g<1>` to be present. These cases are explicitly mentioned.

- **"test_functions":** The string provided here should contain the names of all test functions in the file. This does not include helper functions that should be kept as they are. The functions named in this string can later be redefined to KTF `TEST` functions. Each name is separated by whitespace.
- **"init_code"**: The default value of this field a call to `KTF_INIT()` before the main function. If any additional setup code is required, such as creating a new context, this field can be used. Note that `KTF_INIT()` must still be called outside the main function as part of the multiline string. The substring `\g<1>` must also be present, as it is used to represent the signature of the main function.
- **"exit_code"**: Like the previous field, the default value is good enough unless additional cleanup code is required. If the previous field was specified, this field should most likely be used as well. The minimum code required here is a call to `KTF_CLEANUP()`. This string should begin with `\g<1>`, which represents the signature of the exit function.
- **"include_code"**: This field is used for the header inclusion and should be left empty in most cases. The default value is the inclusion of the KTF header file. This string should begin with `\g<1>`, which represents the include statement matched by the regex.
- **"new_types"**: The use case for this field is to specify new types used by KTF. New types are required when working with contexts. Unless new types are required, this field should be ignored. The string stored here should begin with the substring `\g<1>`, which represents the code matched with the regex.
- **"boilerplate_code"**: The code specified here is added to the beginning of every function named in the **"test_functions"** field. One use case is when adding context code to every test function. This string should begin with `\g<1>`, which represents the signature of the matched function.
- **"test_suite_name"**: The name entered here is used by the `TEST` macro as the name of the test suite. If this field is left empty, the name of the initialization function is used instead.
- **"context_args"**: The string entered here should contain the parameters of functions that should be redefined with `TEST`. This field is used if either 1) a set of common arguments should be supplied by a context instead, or 2) if the parameters should be removed. For example, to target functions with `struct xarray *xa` or `void` as their

only parameter, use the string `"struct xarray [*]xa|void"`. If this is not specified, functions with these parameters will be ignored when functions are redefined with `TEST`.

- **"common_call_args"**: Serves the same purpose as the previous field, except that this is used to specify the argument sent to the function, rather than the parameter signature. For the example under **"context_args"**, the value entered here could be `"&array"`.
- **"extra_dummy_args_call"**: This field is used in cases where dummy functions are created to wrap function calls. The value specified here is used in the wrapped function call.
- **"blacklist"**: The list of strings entered here is used to specify functions that should not be redefined with the `TEST` macro. This field is mainly aimed at helper functions or functions with unique parameter lists.
- **"replacements"**: This field is used to replace code patterns in the test file with other code patterns. The main use case is to replace assertion patterns with KTF assertions. The value of this field is a list of tuples, where each tuple contains two regex strings.
- **"should_add_new_main"**: Set this field to the boolean value `True` if a new main function should be defined. One use case for this field is when the test file only contains a single function, or if the main function contains test code.

Most of the fields mentioned above have a corresponding method that must be called manually. The reason for this choice is to let the user choose exactly which parts of the conversion process that should be automated. Because several fields have a default value that the user may want to use, the user shouldn't need to fill these fields in order to have their methods called.

## 6.6 Using the conversion script

The parameters and methods calls used for converting `test_xarray.c` can be found in the Python file `convert_wrapper_xarray.py` in the Github repository [7].

## 6.7 Summary

In this chapter we have focused on conversion of `test_xarray.c`. The data structure tested in the file was briefly introduced, along with some of its related functions and macros. We continued by discussing the structure of the test file, where we identified several features that can make a scripted conversion viable. These features included common parameter lists between the functions, a lack of return values, and the use of a single assertion mechanism. This information was used in the following

discussion about how to make the file compatible with KTF. The chapter was ended by introducing the conversion script and how it was used to convert the test file.

The following chapter will focus on the conversion of `test_rhashtable.c`. This test file has an entirely different structure that will be converted in a manual way. The results of the conversion of `test_xarray.c` will be presented and discussed in chapter 8.

# Chapter 7

# Converting test_rhashtable manually

## 7.1 Introduction

The rhashtable [62] [55] is a kernel data structure that we briefly introduced in section 5.2. As mentioned earlier, the `struct rhashtable` type serves as a generic and relativistic hash table, with automatic resizing and concurrency support. Therefore, the data structure bears some resemblance to the hash tables found in other programming languages, although the implementation is tailored for use in the kernel.

One of the differences between rhashtables and hash tables in other contexts, is the use of the *Read-Copy Update (RCU)* [32] mechanism for concurrency. RCU is an alternative to the more traditional read-write locks, and it allows a rhashtable to be read even while it's being resized [55]. A rhashtable consists of an array of *hash buckets*, where each hash bucket is a linked lists of data elements.

## 7.2 Core functions and macros

In section 5.2.1 we saw how to setup a rhashtable with default values for most of the options. In this section we will take a quick look at some of the most important functions for this data structure.

To create and initialize a new rhashtable, first define an empty `struct rhashtable` and a filled `struct rhashtable_params` as seen in listing 7.1. Most of the fields in the parameter struct are optional, but the following three are required: `head_offset`, `key_offset` and `key_len`. Next, call the function `rhashtable_init` with a pointer to the rhashtable and another pointer to the parameter struct. The table can later be destroyed with `rhashtable_destroy`, or with `rhashtable_free_and_destroy` if memory needs to be manually released.

The rhashtable is capable of storing any type of struct, given that the type contains at least a `struct rhash_head` and a key field. This also means that the user can provide custom hashing or object comparison functions, by using the parameter struct.

Listing 7.1: A minimal way to create and destroy a rhashtable

```
struct my_data {
    int data;
};

struct object {
    int key;
    struct rhash_head head;
    struct my_data data;
};

int main(void)
{
    struct rhashtable my_table;
    struct rhashtable_params rht_params = {
        .head_offset = offsetof(struct object, head),
        .key_offset = offsetof(struct object, key),
        .key_len = sizeof(int),
    };
    int success = rhashtable_init(&my_table, &rht_params);
    ..
}
```

There are several functions available for inserting data into the table, such as `rhashtable_lookup_get_insert_fast`, `rhashtable_lookup_insert_fast` and `rhashtable_insert_fast`. Lookup can be done with `rhashtable_lookup_fast`, and removal with `rhashtable_remove_fast`.

Iteration over all the objects in the table requires an additional data structure, `struct rhashtable_iter`, as well as the usage of up to six different functions: `rhashtable_walk_enter`, `rhashtable_walk_start_check`, `rhashtable_walk_start`, `rhashtable_walk_next`, `rhashtable_walk_stop` and `rhashtable_walk_exit`.

There's also a `struct rhltable` type with many of the same features; the main difference is the added support for storing multiple elements with the same key.

## 7.3 A look at the test_rhashtable.c file

The main function of this module is quite different from the one we saw in `test_xarray.c`. Instead of a sequence of calls to independent test functions, this main function contains a mix of initialization code, cleanup, calls to test functions, error counting and print statements. Several variables are created, memory is allocated and parameters are set before the first assertion construct is encountered: an if-test that returns a negative value upon failure. This is how most assertions are done in this file; an if-test is used for the assertion check, while its body can contain anything from a single return statement to a combination of memory deallocation, print statements, error counting and control flow statements.

Following this initial if-test is the first of several for-loops. The number

of iterations for these loops are controlled by static variables set in the beginning of the file. These variables are tied to the Linux module system through the use of `module_param` and `MODULE_PARM_DESC`, allowing the user to set their values when inserting the modules into the kernel. In constrast, `test_xarray.c` did not use this feature of the module system.

The body of the first for-loop contains several elements that could have been separated into several different functions. At the beginning of every iteration, the same `struct rhashtable` object is reinitialized with `rhashtable_init`, before it's sent as parameter the local test function `test_rhashtable`. Finally, `rhashtable_destroy` is used for cleanup before the next iteration of the loop. A better way to handle this logic using KTF features would have been to create `test_rhashtable` as an independent test function, move the setup and cleanup code inside a fixture, and provide the last argument in the function call through a context instead. This solution would provide a smaller main function, and make `test_rhashtable` more independent from the rest of the file. The contents and structure of `test_rhashtable` and similar functions will be discussed later.

Another characteristic of the error handling is that failures are reported by calling a print function; in this case `pr_warn` is used. Then a negative value is returned, preventing the test file from executing any more tests. A KTF solution here would be to use either `ASSERT_TRUE` or `ASSERT_FALSE` for this purpose, or the equivalent `EXPECT_*` variant if execution should continue upon failure. These solutions would provide the same benefits with less code, in addition to error counting and more detailed output.

Following the loop, more memory is managed, with calls to two test functions inbetween. This leads to the a pair of loops that are responsible to starting and stopping thread-related test code. The last few lines are used for cleanup tasks and a call to the final test function. This approach to writing test code is fundamentally different to the code found in `test_xarray.c`.

There are in total 11 other functions in addition to the main one. Most of these are not called from main, `test_rht_init`, but rather from the ones mentioned earlier in addition to a few others not mentioned. Also, some of these perform both initialization and cleanup code, while other do not. Additonally, the number of lines in a function vary widely, ranging from 20 lines at the shortest to over 150 lines at the longest. Just like the main function, these use a combination of if tests, debug printing, goto statements and return values in order to communication tests passing or failing.

Although this approach is quite different from the way testing is done with a unit testing framework, it is understandable that this is the way the module is written. At the time this test module was written, a unit testing framework in kernel space was lacking. Consequently, the authors of the test file had to create their own way of testing their code. With the presence of KTF and other similar frameworks, this could potentially change.

## 7.4 Converting to KTF

Unlike `test_xarray.c` that we looked at in the previous chapter, `test_rhashtable.c` has a structure that makes it less suited for a scripted conversion. There are parts of the conversion that can be scripted, such as the addition of boilerplate KTF code; however, there are several problems with the file that need to be dealt with manually. For example, the sequential nature of the test code is an issue for the KTF `TEST` macro. When KTF calls its test functions, the execution order of `TEST` functions are currently not the same as the order functions are added with the `ADD_TEST`. Another problem is that the functions in this file have unique parameter lists that can be difficult to deal with using the current KTF features. A third challenge is the many different ways assertions are handled. Hence, a manual conversion is preferable.

In this section we will discuss how this test file will be converted. As there are more patterns to handle here than in the previous test file, we will be looking at more special cases. This section will also be more technical than the discussions in the previous chapter.

### 7.4.1 Converting assertions

One of the main differences from the `xarray` test file is the lack of explicit assertion mechanisms. As mentioned in the previous section, assertions in this test file are mainly handled through if-tests, although the bodies of these tests vary from case to case; failures are handled through either a `return`, `continue`, `break` or `goto` statement. These statements are often combined with either print-statements, error-counting or memory deallocation. In some of the cases we can replace the if-tests and their bodies with single calls to KTF assertions; in other cases we just replace parts of the if-test body. In effect these cases must be dealt with on a case-to-case basis, although we will categorize them and discuss them below.

#### If-tests with return

One of the most commonly found assertion patterns in this file, is an if-test that returns if the expression evaluates to true. This pattern can be found in two variants: one with a return value and another variant without it. Which assertion macro to replace the if-test with depends on the type of data that is tested, as well as how a failure should be handled. For a compound expression such as the one seen in listing 7.2, `ASSERT_FALSE` would have been the best fit if no value was returned. In the example, however, a value is returned, a case that was not yet covered by the existing KTF assertions when the conversion was done. For this and several similar cases, an `ASSERT_FALSE_RETVAL` variant was created and used, and it functions the same way as the original `ASSERT_FALSE` statement, except for the additional return value argument. This case can thus be replaced by `ASSERT_FALSE_RETVAL(expected && !obj, -ENOENT);`.

Listing 7.2: One of the most common assertion pattern in the test file.

```
if (expected && !obj) {
    pr_warn("Test failed: Could not find key %u\n", key.id);
    return -ENOENT;
}
```

#### Print statements within and outside if-tests

Print statements should generally be removed when using KTF. However, some of the print statements have calls to test functions as one of their arguments. In these cases, the function calls should be kept while the print statements themselves are removed. The function calls that are kept should then be wrapped inside one of the `EXPECT_*` macros, to improve the output from KTF. The arguments to the print statements that have side-effects may be kept or removed, depending on their purpose.

#### Debug statements and if-tests

This category refers to statements or if-tests whose only purpose is to help debugging and error reporting. One example is an if-test that increments an error counter if the test passes; the error counter is then printed at a later stage to report the overall status of the executed tests. As KTF already have built-in features for this, these cases can usually be removed straight away.

#### If-tests with a `break`, `continue` or `goto` statement

As it was mentioned in subsection 7.4.1, assertions that call `break` or `continue` upon failure was added to the framework to better handle these assertion patterns; the `goto` variant was already present and was therefore not added. In all three cases, the if-statement can be replaced with either a `ASSERT_TRUE_*` or `ASSERT_FALSE_*`, where the `*` is substituted with either `CONT`, `BREAK` or `GOTO`. The expression in the if-test is used as the first argument to one of these macros, and a label is also supplied for the `goto` cases. These three cases plus the `RETVAL` one mentioned earlier cover the majority of assertions statements that was added to this file. There were also some cases where the plain `ASSERT_TRUE` and `ASSERT_FALSE` were used, but those were few.

#### If-tests with additional logic

There are some if-tests that act as assertions, that cannot be replaced with KTF assertions in a clean way. These tests have additional logic that must be kept for the test file to function correctly, often related to memory management. There are at least two ways we can handle this situation: we can either keep the if-test as it is and add an `EXPECT_*` assertion within it, or we can replace the if-test with a `GOTO` assertion and move logic further down. The latter approach would require a goto-label with an extra `continue` or `break` above it, to avoid the code from being run in normal

circumstances. As this is messy and error-prone, the other solution appear to be the better one. Listing 7.4.1 shows an example of the latter solution, where EXPECT_INT_GE was added.

```
if (err < 0) {
    EXPECT_INT_GE(err, 0);
    vfree(tdata);
    vfree(objs);
    return;
}
```

### 7.4.2 BUG_ON

There was only a single use of this macro in the entire file, BUG_ON(!obj);. As it is used as a standalone statement, it was initially replaced with EXPECT_TRUE(obj != NULL);; however, this was later changed to ASSERT_TRUE(obj != NULL); after looking at the definition of the BUG_ON macro. According to the comment above its definition, its use appear to be strongly discouraged unless there is no other way to handle a potential failure. The exact behavior of BUG_ON depends on the kernel configuration used, but it will often generate a stack dump in the logs and require a reboot of the system. To avoid this behavior, this statement should be replaced with ASSERT_TRUE.

### 7.4.3 WARN

How calls to WARN should be handled depends on the context of its use. If it is used as a standalone statement, EXPECT_FALSE should be the best choice. When it is used as the expression of an if-test, one of the ASSERT_* variants should be a better fit; exactly which assertion macro to choose depends on what control flow statements that are used in the body of the if-test. Like the if-test assertions, the use of this macro comes in several variants, and the context should be considered to find the best fit.

### 7.4.4 Converting function definitions

Ideally we would like as many functions as possible to be defined using the TEST macro, to get an individual assertion count and status for every single function. In the case of this module, however, it would be difficult to to do this with more than one single function, without larger rewriting of the flow and structure. One of the reasons for this is the highly sequential nature of the module. As earlier discussed, the main function does not only call the other test functions, it also performs quite a bit of initialization and cleanup as well. Furthermore, it appears like the order of the function calls does matter because of this structure, while at the same time, functions registered with the ADD_TEST macro are not necessarily run in the same order as they are added. This can cause problems if some of the functions are run in a different order than intended, which is thus an argument

against redefining functions with the TEST macro. On the other hand, KTF assertions require access to a ktf_test *self in order to work, and this pointer is made accessible by the TEST macro; thus, it is necessary to have at least one TEST function defined for KTF assertions to work. One way to solve this issue is redefine the main function with the macro, while the rest of the functions receive the ktf_test *self pointer through an additional parameter. This solution also requires that a dummy function is created and used as the argument to the module_init call; this dummy function is then responsible for calling ADD_TEST on the previous main function, and for creating the necessary context.

Nevertheless, a drawback of this solution is that we don't get the same detailed feedback as we did with test_xarray.c, although it will have to do.

### 7.4.5 Adding self pointer to function calls

As mentioned multiple times earlier, all non-TEST functions will need access to a ktf_test *self parameter to use KTF assertions. This parameter must thus be added to all functions that use KTF features; the self-pointer must also be added to the calls to said functions.

### 7.4.6 Adding self pointer to thread function

There is one function where the technique explained above will not work: threadfunc. This is a function that will be executed in several, parallell kernel threads at once, and these threads are spawned through calls to the kthread_run function. A quick and easy way to provide the ktf_test *self parameter to these threads is by using a context. As we are not going to change the memory address stored in the pointer, this approach should work. The context can be created in the dummy main function before the call to ADD_TEST, and later be teared down in the exit function of the module.

### 7.4.7 Adding initialization and cleanup code

Finally, the setup and teardown code for KTF must be added. As we have discussed earlier, the minimal code required is the same across test suites. In this file we will use a context as well, so we will also need to define a new struct type, create a static instance of it, and add it with KTF_CONTEXT_ADD in the main function. For cleanup we also add calls to KTF_CONTEXT_FIND and KTF_CONTEXT_REMOVE in the exit function, in addition to the mandatory KTF_CLEANUP call.

## 7.5 Extending KTF with new assertions

During the conversion of this file, a few new macros was proposed to be added to the framework. These macros were added to cover assertion

patterns in the `test_rhashtable.c` file that weren't already covered by KTF.

An example of one such assertion pattern is an if-test with a `continue` statement in its body. In order to replace this construct with one of the existing KTF macros, the macro would have to perform a `continue` if the assertion fails. Although there was an assertion macro available that could do that, the macro only worked for integer comparisons. Consequently, it couldn't handle general expressions, so `ASSERT_TRUE_CONT` and `ASSERT_FALSE_CONT` were created. This is also the reason for creating the other four macros as well; variants of these macros also existed, but only for specific use cases like integer comparisons.

- ASSERT_TRUE_RETVAL
- ASSERT_FALSE_RETVAL
- ASSERT_TRUE_CONT
- ASSERT_FALSE_CONT
- ASSERT_TRUE_BREAK
- ASSERT_FALSE_BREAK

## 7.6 Converting two smaller test files

We will now take a look at two smaller test files to see how this category of test files work with the framework. As these files are considerably shorter than the two previous files, each file will be given a subsection each. The first file is converted manually, while the second file is converted both manually and by the script.

### 7.6.1 lib/test_string.c

`test_string.c` is another test file found in the
`lib` directory of the kernel source tree. Its purpose is to test four versions of `memset` family of functions: `memset`, `memset16`, `memset32` and `memset64`. The file contains three test functions and a main function, with a total length of 142 lines. Neither of its functions take any parameters, and the three test functions return an integer to report the test result. The results of the function calls are checked in the main function, and the two print functions `pr_info` and `pr_crit` are used for output to the user.

The body of each test function begins with a few variable declarations and a call to `kmalloc`, followed by an if-test to check if the allocation fails. The memory allocation is kept as it is, while the if-test and the subsequent return statement is replaced with `ASSERT_OK_ADDR(p)`. Next, a double for-loop running for 256 * 256 iterations performs a call to `memset` and then `memset16/32/64`, followed by another for-loop with three if-tests and a goto-statement within each. All of these three if-tests are replaced

with a ASSERT_FALSE_GOTO(<expression>, fail); statement each, where <expression> is the original assertion expression and fail represents the label that is jumped to upon failure. Following this label, located at the end of the function, is a call to kfree and an if-test to determine what value to return. We remove this last if-test and keep the kfree call.

We then finish the file by adding all the boilerplate KTF code that is needed. Neither fixtures nor contexts are needed, so there are no need to add code for that. The main function consists of calls to the test functions, if-tests to check the results, and print statements to report the test results to the user; All this code is replaced with three calls to ADD_TEST. The signature of the three test functions are also replaced with calls to the KTF TEST macro. Finally, as the file lacks an exit function, this is added at the end of the file in order to call KTF_CLEANUP. When running the fully converted test file, all three test functions passes in roughly half a second in total.

### 7.6.2 lib/test_sort.c

test_sort.c is another short test file located under the lib directory in the kernel. The goal of the file is to test the sort function made available in the linux/sort.h header. The test file itself is quite small, containing only 50 lines of code, thus making it the shortest test file encountered so far. The file was initially converted manually, but later a fully scripted approach was also taken. Here the scripted approach will be presented.

The test file consists of three functions: a main function containing all the test code, a comparison function used by sort, and an empty exit function. The main function first calls kmalloc_array to allocate memory for an array; the function returns at this point if the allocation fails. Next, the array is filled with test data before it's sorted by the sort function. Finally, a loop checks if every element in the array is smaller than the following element. A jump to the end of the function is used if this condition is false for any element.

Table 7.6.2 shows the Python code used to convert the script. The parameter dictionary specifies that the main function test_sort_init should be redefined to a KTF TEST function; the should_add_new_main is consequently set to True to tell the script to add a new one. This is necessary as KTF requires at least one TEST function to be defined, and this function must then be registered with ADD_TEST from a new main function. The name of the test suite is set to test_sort_rewrite, to match the name of the test suite. The replacements field is used to convert the two return statements to KTF assertions.

An object of the Converter class is then created with the dictionary as one of its arguments. 6 methods are then called to include the KTF header file, add the mandatory calls to KTF_INIT() and KTF_CLEANUP(), redefine the main function, add a new main function, and finally replace return statements with KTF assertions.

The resulting test file runs and succeeds without requiring any manual editing.

Table 7.1: The python code used to convert the file.

```
test_sort_rules_2 = {
    "test_functions":
    ["test_sort_init"],

    "test_suite_name": "test_sort_rewrite",

    "blacklist": ["cmpint"],

    "replacements": [
        ("return err;", "ASSERT_INT_EQ(err, 0);")
    ],

    "should_add_new_main": True
}

state = Converter(full_source_path, full_target_path,
    test_sort_rules_2, True)
state.add_include_code() \
    .add_init_code_to_main() \
    .add_exit_code() \
    .convert_to_test_common_args() \
    .use_replacements() \
    .result()
```

## 7.7 Summary

In this chapter we have examined the `test_rhashtable.c` test file and discussed how it can be converted to KTF manually. The chapter began with a short description of the data structure and an introduction to how it can be used. We continued with a description of the main function in the test file. A few suggestions were proposed as to how the main function could have been written differently with KTF. We also presented information about how the file is structured, compared to `test_xarray.c`. This section was followed by a technical discussion about how the test file can be changed to work with KTF. The majority of the discussion was related to how assertions should be handled. Unlike `test_xarray.c`, this test file does not use any macros or functions to handle assertions; instead, if-tests are often used together with other control-flow statements. Six new macros to KTF were proposed, to cover assertion patterns that were not covered by the existing KTF macros. We ended the chapter by discussing the conversion of two smaller test file, `test_string.c` and `test_sort.c`. The former was converted manually and the latter through both a scripted and manual approach.

In the next chapter, we will present the results of the conversion of all the four test files. We will also discuss the conversion process and the current state of KTF, before we finally conclude the thesis.

# Chapter 8

# Results, discussion and conclusion

We will begin this chapter by summarizing the process so far. Then the results of the project will be presented, followed by a discussion and conclusion.

## 8.1 Summary

In chapter 3, we looked at some of the tools and frameworks used for testing the Linux kernel today. This included frameworks that automatically build and test new versions of the kernel as they are released, like Linaro's LKFT and Fuego. Avocado was another framework mentioned that enables developers to write tests in Python or any other language, as opposed to the more commonly used C and Bash languages for writing kernel tests. Also mentioned were some of the test suites used by developers and the two first frameworks mentioned above, as well as the two unit testing frameworks KUnit and KTF, where the latter has been the main focus of this thesis. This is by no means an extensive list of the testing tools used today, but is meant to show some of the tools currently available.

Chapter 4 explained the chosen approach and how to setup the environment used in this project.

Chapter 5 introduced the Kernel Test Framework along with some of its core features. We looked at some of the core features of the framework, like contexts and fixtures, and several examples of their use.

In chapter 6 we looked at our first conversion of a kernel test file, `test_xarray.c`. The chapter began with a brief introduction to the `struct xarray` data structure and some its most important functions. The XArray data structure can presented as a combination of a hashtable and a resizable array of pointers.

Next, we saw how the test file had a clear structure, with the main function consisting almost exclusively of calls to the test functions. These functions in turn all took a single pointer argument each, making them almost completely independent from each other. Each function also

| File name | Number of lines (before) | Approach | Num. tests |
|---|---|---|---|
| test_xarray.c | 1330 (1238) | Scripted | 21 |
| test_rhashtable.c | 772 (823) | Manual | 1 |
| test_string.c | 114 (142) | Manual | 3 |
| test_sort.c | 61 (51) | Manual and Scripted | 1 |

Table 8.1: Overview of the modified test files

focused on one specific aspect of the data structure it wanted to test. This structure made the file straightforward to both read and understand.

Another feature of the test file was the use of a single assertion macro for performing every test assertion with a few expections. This macro was defined in the top of the file, and contained a fail test, an error counter and printing of debug information upon failure of the test.

The combination of a clear structure and the use of a single assertion macro meant that a scripted conversion was deemed to be possible. A series of regular expressions was written in an attempt to convert the file to KTF in a more efficient manner than manual conversion. This didn't work well, so a Python script was written instead. The goal of this script was to ease the use of KTF for new users, given that the script could be made general enough. Given the right parameters, the script was also capable of adapting the test file to KTF.

In chapter 7, we moved our focus over to the next test file, `test_rhashtable.c`. Like the previous chapter, it began with short introduction to the `struct rhashtable` data structure and its main functions. Following this was a discussion about the structure of the file and the choices made during the process of conversion.

The script used for `struct xarray` was initially meant to be used for this test file as well. However, the script turned out to be a unsuitable tool for the task, due to the lack of structure of the test file. Therefore, a manual conversion was done instead.

At the end of the chapter, we looked at the conversion of some smaller test files. These test files were both shorter and quicker to convert than the first two. As we have already discussed the process in detail, these test files were described in a section each.

## 8.2 Results

In this project we have seen how existing kernel test files can be modified to use the Kernel Test Framework. This conversion process was done by both manual editing and by using a Python script. The test suites created for each of the files all ran and completed, although the `test_rhashtable` file didn't pass all its tests. Table 8.2 shows the files modified. The numbers inside parenthesis indicate the original size of each file.

The last column shows the number of times the KTF `TEST` macro was used. The low number shown for `test_rhashtable` is not ideal, but is caused by the way the file is structured. The KTF `TEST` macro currently

| Macro names |
|---|
| ASSERT_TRUE_RETVAL |
| ASSERT_FALSE_RETVAL |
| ASSERT_TRUE_CONT |
| ASSERT_FALSE_CONT |
| ASSERT_TRUE_BREAK |
| ASSERT_FALSE_BREAK |

Table 8.2: New KTF macros proposed

does not support function arguments, which means that most functions that heavily rely on arguments are difficult to convert to the macro.

In addition to the modified test files, the earlier mentioned Python script was also created. The script in its current form has limited uses for test files without a specific structure, but could still have some use for simpler tasks. For test files with certain structure, the script can do more.

During this project, six new macros were proposed to be added to KTF, presented in table 8.2.

Feedback to the authors of KTF was also given. This helped uncover problems when installing or using the framework that the authors themselves did not encounter. Feedback was also given about limitations of the framework, and constructs that was difficult to handle with its current features.

Although not the main focus in the project, chapter 3 also introduced some of the testing tools currently in use. These were only briefly described, partly due to a lack of documentation surrounding their adoption.

## 8.3 Discussion

The following section will be used to discuss and evalute the findings so far. We will begin with the conversion script before we proceed to discuss KTF in general.

### 8.3.1 Conversion script

In chapter 6 we saw how the scripted conversion was executed. The goal was to see if this approach could work and potentially ease the transition to using the framework.

As the script was developed while analyzing the `test_xarray.c` file, it was expected that the script should be able to fully convert this test file. This expectation was also met when it turned out that the converted test file was usable by KTF without requiring any manual editing at all. Furthermore, nearly every function except the main, exit and helper functions could be redefined with the `TEST` macro - although there were one exception that required a workaround; every assertion had a KTF equivalent; and additional arguments to the test functions could be

supplied by using a context. Thus, the scripted conversion of this test file can be considered a success.

Perhaps the main reason for why this approach works is because of how `test_xarray.c` is structured. The `TEST` macro used for defining test functions in KTF does not accept any parameters from the user. Any parameters needed must instead be supplied using either a context, fixture or global variable, requiring all test functions to share the exact same parameter list. This is the case for nearly all test functions in `test_xarray.c`. The pointer parameter that these functions take can be accessed through a shared context, thus solving this problem for most of the functions. Performing said changes through a script was straightforward for this test file.

Another reason for why this approach works is that the test file uses a single assertion macro for all but a few cases. Once again we have a consistent pattern that is well suited for regex substitution and automation.

On the other hand, the script in its current form performs poorly when used on the `test_rhashtable.c` file. This test file is in multiple aspects quite different from `test_xarray.c`: The main function contains a mix of setup code, calls to test functions and cleanup code, while the main function of the previous test file only contained functin calls; most test functions have unique lists that are difficult to replace with existing KTF features; there are many different assertion patterns to handle, and few of them are as simple as a single macro or function call. The mentioned differences between `test_xarray.c` and `test_rhashtable.c` are just some of reasons for why the latter file was converted manually.

It's still worth mentioning that the script is capable of performing some parts of the conversion of `test_rhashtable.c`; however, its usefulness for this test file is far less than for `test_xarray.c`. For example, the script can redefine the main function with the `TEST` macro, and add a new main function in its place. The script can also add boilerplate KTF code, but this is the limit of what it can do for this specific file.

The smallest of the four test files, `test_sort.c`, was converted both manually and scripted to compare the two approaches. Although it's small size likely have an impact on the result, both approaches were roughly equally fast. The few parameters required meant that little time was needed to use the `Conversion` class to convert it. Performing the code changes manually required roughly the same amount of writing, so the results were similar. Still, the experiences with this test file is likely not representative for larger test files and should be taken with a grain of salt.

The overall experience with the four test files shows that the scripted approach works for at least some of the conversion process. The script should be capable of adding at least the boilerplate KTF code to most test files, given the assumption that it uses the Linux module system. Whether a fully scripted conversion is possible depends on how the test file is written; it worked for `test_xarray.c`, but not for `test_rhashtable.c`. There is currently not enough data to determine whether a fully scripted conversion is worth the time required compared to a semi-automatic approach. As explained earlier, the script was developed while analysing

`test_xarray.c`, and thus it is unclear how much time this approach will require for other large test files. The smaller `test_sort.c` was converted through both a manual and fully scripted approach, but it is too small in size to give us a representative impression of the time required. Still, by using the script to add boilerplate KTF code and using mostly default values, some of the conversion can be automated without spending much time on finding parameters or understanding the script. This combination of scripted and manual conversion might be more useful for new users of the framework than a fully scripted conversion.

### 8.3.2 Using KTF

The framework worked as expected for `test_xarray.c`. The structure that made a fully scripted conversion possible, also meant that little workaround was required to make the test file work with KTF. The one challenge that had to be dealt with was the test function that took an additional parameter compared to the other test functions. This function was called three times from main, with a different second argument for each call. The workaround consisted of creating three dummy functions that would hold one function call each, solving the problem. Apart from this detail, the test file is a good candidate for KTF.

The framework also worked well for the smaller test files described in section 7.6. As none of the test functions required any arguments, they could be redefined to KTF test functions straight away. This allowed KTF to output the status for each test function individually, improving the output quality for `test_string.c`. The conversion also reduced the number of lines from 142 to 114 for this file, thus providing some simplication as well. The conversion had less of an effect for `test_sort.c` due its single test function. This also required a new main function to be defined as a workaround, which explains why the length increased by 6 lines after the conversion. The original assertions in both files mainly consisted of if-tests, and they were thus simple to replace with KTF assertions.

Nevertheless, the framework provided fewer benefits for the less structured `test_rhashtable.c` file from chapter 7. One of the reasons are the unique argument lists of the functions. We saw in section 6.4.5 that there are ways to circumvent this problem, by either using contexts, fixtures or dummy functions. However, none of these solutions are ideal for this file, as the test functions called from main all share the same initialization code. In order to move these function calls into dummy `TEST` functions, most of the shared variables and initialization code would need to be copied along with the function calls. How much of the shared code to duplicate would vary from function to function, but it would definitely increase the size of the file and likely introduce new bugs to the already large test file.

This alternative solution would still have the benefits of splitting up the main function into several smaller functions, and increase the number of `TEST` functions for better KTF output. However, it was not done in this project as the solution in chapter 7 lead to a more direct translation, with a reduction in code size as an added benefit. The main disadvantage of the

chosen solution is thus less KTF output due to the single `TEST` function.

Still, the conversion to KTF did provide several benefits to the `test_rhashtable.c` file, despite the reduced amount of test output. First, the framework provides test output that is consistent across test files. Second, the KTF assertion statements provide more information about the location and cause of failures than the previous solution did. Failed assertions now show both the line number and the comparison that failed. Third, by removing print statements and replacing multi-line constructs with single-line assertions, the test file decreased in size by roughly 50 lines.

As it has been mentioned several times through this thesis, KTF in its current form struggles with test functions that take parameters. This is a design choice, as it can be argued that test functions shouldn't take any parameters; they should instead be independent and focus on one specific task each. Although this point may be true for unit tests, the test files found in the kernel source tree are not necessarily unit tests in the traditional sense. Some of these test files may require a rewrite, or changes to the framework, to get the most benefit from the framework. Still, as long as there are at least one `TEST` function present, the framework will work and give some output.

Finally, it should be mentioned that the tests are currently not executed in the same order as they are registered with `ADD_TEST`. They are instead ordered alphabetically and executed in that order. This did not appear to cause any problems for test files converted so far, but it might affect the result for files where the execution order of test functions does matter. This is one area of improvement that was discovered and reported during the project, and may change in the future.

### 8.3.3 KTF vs. other frameworks

There is currently an ongoing discussion on the Linux kernel mailing lists about the KTF and KUnit frameworks. Both frameworks have been introduced as the new framework for unit testing in the kernel, and it's currently unknown whether only one of them will survive or if both will find their use.

As described in section 3.5 there are a number of test files in the kernel that perform unit testing, but without the use of a common framework. Had all these files been converted to either KTF or KUnit, these files could possibly see more use and inspire other developers to write more tests, as the developers wouldn't need to create their own testing utilities from scratch. Also, the use of a shared framework would make the test files easier to read for other developers as well. The conversion process described in chapter 7 is an example of how a common test framework can reduce the number of lines and lessen the need for custom debug code.

Both KTF and KUnit provide similar features for writing tests. For example, the macros used for creating test functions work in familiar ways, and both frameworks have support for setup and teardown functions as expected, although with some differences in the way they are registered. It's not unlikely that both frameworks present the tools necessary to write

the same tests.

However, there is a distinct difference in how the tests are run by the two frameworks. KTF runs its tests inside the kernel as kernel modules, enabling the user to test a module against other parts of the kernel or the hardware itself. As a faulty test can crash the entire kernel, this way of testing is often done inside a virtual machine.

KUnit on the other hand runs its tests inside User Mode Linux as mentioned in section 3.5.2. This approach lets the the user test in user space, and without the need of a separate virtual machine. Although testing in user space has it benefits, it also limits the access to hardware, while also requiring more use of mocking than the previous approach. This can be an advantage in cases where mocking and isolation are what we want, but it's also less suited for testing drivers or for testing code against specific architectures.

There is also the third alternative of writing tests for one of the older frameworks, like kselftest, instead of learning one of the former two. As mentioned earlier, the tests written for kselftest are used by LKFT and Fuego for their automated test runs. Also, kselftest should be more familiar to kernel developers than KTF and KUnit, due to its age.

However, the goal of both KTF and KUnit is to provide better facilities for performing white-box testing of the kernel. Although kselftest does allow the user to write kernel mode tests, the focus of this framework has still been testing from userspace. The kernel test feature was also added near the end of this thesis, so this option wasn't available until recently. Therefore, KTF and KUnit appear to provide better facilities for this kind of testing, due to their specialization on the area.

## 8.4 Conclusion

The goal of this thesis has been to convert Linux kernel test files to the Kernel Test Framework, and evalute both the framework and the process. The test files were converted in two ways: `test_xarray.c` and `test_sort.c` were converted by a Python script developed for this purpose, while `test_rhashtable.c` and `test_string.c` were edited manually.

The conversion process began with `test_xarray.c`. This test file was fully converted by the script, with no manual editing required. As the conversion script was developed while analyzing this test file, the positive results was expected. Every test function received its own status lines in the KTF output, and all the tests passed upon execution. The clean structure of the file also made it well suited for KTF, as this is how the framework is meant to be used. There was, however, one test function that required a workaround solution in order to be converted, but overall the conversion of `test_xarray.c` was considered a success.

The conversion of `test_sort.c` and `test_string.c` went equally well. Every test function in both files were converted to KTF `TEST` functions, and every assertion pattern had a corresponding KTF assertion. Consequently, both test files worked well with KTF.

The `test_rhashtable.c` file received less direct benefits from the framework, as the structure of the file is different from what KTF expects. Only a single function was redefined with the `TEST` macro, thus reducing the amount of output that KTF provides. Larger changes to the file could have improved compatibility with KTF, although this would have required a major rewrite of the entire file. Still, the conversion to KTF did provide some benefits to the file, such as consistent output across the four test files, more information when assertions fail, and a reduction in the length of the file.

We have seen that the conversion script is capable of either fully or partly converting test files to KTF. How much of the conversion process it can automate depends on the structure of the test files it is given. As we discovered with `test_xarray.c` and `test_sort.c`, a fully scripted conversion is possible if the input test file has a structure that works well with KTF. However, finding all the right parameters for a fully scripted approach takes time that could otherwise be spent on manual editing; there is currently not enough data to determine if this approach is worth the time required, compared to a semi-scripted conversion. Also, the lack of structure in `test_rhashtable.c` made this test file unsuitable for a fully scripted conversion. Overall, the script is likely best used as part of a semi-scripted conversion, where the script adds boilerplate KTF code to the test file, while the rest of the conversion is done manually.

KTF in its current form worked for every test file that was converted, but the amount of output from the framework depended on the structure of the test files. There are certain structural patterns that the framework struggles with, as we have seen several examples of. Thus, KTF enforces a certain way of writing test files in order to get the most benefit from using the framework. However, this is also one of the reasons for using a test framework; it helps us structure our tests in a good way. As such, it is not surprising that `test_rhashtable.c` received less benefit of the framework. From the results gained so far, the framework has met the needs of most of the converted test files.

In this thesis we have also taken a brief look at some of the other test frameworks that currently exist. We have briefly discussed some of the similarities and differences between KTF and another emerging framework, KUnit. Although they both provide similar features for unit testing kernel code, they execute their tests in two different environments; KTF compiles tests into kernel modules that are executed in kernel space, while KUnit runs its tests inside User Mode Linux (UML). Thus, the former framework has a larger emphasis on how the test code iteracts with a live kernel, while the latter framework has more emphasis on mocking and testing code in isolation from its dependencies. It is currently too early to conclude whether one of the frameworks will win or if both find their own niche.

During this project several contributions have been made to KTF. By installing the framework in a different environment than it was developed, problems previously unknown to its authors have been encountered, reported and fixed. Furthermore, the use of the framework has generated

feedback about weaknesses and areas of improvement. If the conversion script is further developed, it can hopefully be used to speed up conversion of other test files. New macros were also developed to cover assertion patterns that were previouly lacking support. Finally, the converted test files can serve as examples of how the framework can be used.

## 8.5 Limitations and further work

The script used in chapter 6 should be seen as a prototype of how a conversion script could be made and should not be considered a finished product. Although it works for the `test_xarray.c` file, it was also made with this specific file in mind. Consequently, the script is best suited for test files with the same kind of structure. Hopefully it's general enough to also be used on other test files as well, given a certain amount of manual analysis on the target test files beforehand. Regardless, the script could certainly be extended to support more test files.

The test files converted so far shows how KTF can be used on kernel test files, and maybe it inspires others to continue the conversion process. There are plenty of other test files around the kernel that may be good candidates for this framework, even if they were not mentioned in this thesis.

As a framework for this area is now available, more test files should be written to detect more kernel bugs. The support for Test-Driven Development should also be better with this framework, and perhaps this approach will aid in detecting more bugs at an earlier stage. The framework can also be used to write tests after a bug has been fixed, to ensure that the bug does not reoccur later.

There are several features of the framework that have neither been mentioned nor used in this thesis, as the test files that were converted did not need them. Using these features on other test files may provide additional benefits to kernel test files.

The framework is currently installed out-of-tree, and bringing it into the kernel source tree may increase its use due to better availability. The extra steps needed in order to setup the framework may discourage developers from using the framework.

Furthermore, several other testing frameworks have been mentioned, but without being used. As such, this thesis is likely favoured in direction of KTF, since this is the only test framework that was put to the test. An idea for future research could be the use and evaluation of other kernel test frameworks, with comparisons to KTF.

# Bibliography


[1] *1. Background and motivation*. URL: http://heim.ifi.uio.no/~knuto/ktf/introduction.html (visited on 01/06/2019).

[2] *4. Building and installing KTF*. URL: http://heim.ifi.uio.no/~knuto/ktf/installation.html (visited on 31/08/2018).

[3] *6. KTF programming reference*. URL: http://heim.ifi.uio.no/~knuto/ktf/progref.html#assertions (visited on 12/09/2018).

[4] *About Linaro's Linux Kernel Functional Testing*. en. URL: https://lkft.linaro.org/about/ (visited on 11/04/2019).

[5] *About The Linux Foundation*. en-US. URL: https://www.linuxfoundation.org/about/ (visited on 11/10/2018).

[6] John Admanski and Steve Howard. 'Autotest — Testing the Untestable'. en. In: (), p. 12.

[7] Arild. *Backup of files used in the master project. Contribute to Arildlil/master-vm-files development by creating an account on GitHub*. original-date: 2018-10-15T09:44:25Z. July 2019. URL: https://github.com/Arildlil/master-vm-files (visited on 19/07/2019).

[8] *Autotest - Fully automated tests on Linux*. original-date: 2011-09-19T15:25:16Z. June 2018. URL: https://github.com/autotest/autotest (visited on 06/06/2018).

[9] *Autotest White Paper — autotest 0.16.3-44-g0d527f documentation*. URL: http://autotest.readthedocs.io/en/latest/main/general/WhitePaper.html (visited on 06/06/2018).

[10] Kamalesh Babulal and Balbir Singh. 'Keeping the Linux Kernel Honest'. en. In: (), p. 14. URL: http://edgyu.excess.org/ols/2008/Kamalesh%20Babulal%20-%20Keeping%20The%20Linux%20Kernel%20Honest%20(Testing%20Kernel.org%20kernels).pdf.

[11] Kent Beck. *Test Driven Development: By Example*. English. 1 edition. Boston: Addison-Wesley Professional, Nov. 2002. ISBN: 978-0-321-14653-3.

[12] Rex Black, Veenendaal, Erik Van and Dorothy Graham. 'Foundations of Software Testing'. English. In: *Foundations of Software Testing*. Third. Cengage Learning EMEA; 3 edition (January 6, 2012), pp. 40–44. ISBN: ISBN-13: 978-1-4080-4405-6.

[13] Rex Black, Veenendaal, Erik Van and Dorothy Graham. *Foundations of Software Testing*. English. Third. Cengage Learning EMEA; 3 edition (January 6, 2012). ISBN: ISBN-13: 978-1-4080-4405-6.

[14] Martin Bligh and Andy P Whitcroft. 'Fully Automated Testing of the Linux Kernel'. en. In: (), p. 16. URL: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-113-126.pdf.

[15] *Build the future of Open Infrastructure*. en. URL: https://www.openstack.org/ (visited on 29/03/2019).

[16] *Building and Installing Software Packages for Linux: Using Make*. URL: http://www.tldp.org/HOWTO/Software-Building-HOWTO-3.html (visited on 29/05/2019).

[17] *cmocka: Assert Macros*. URL: https://api.cmocka.org/group__cmocka__asserts.html (visited on 11/06/2018).

[18] *cmocka: Main Page*. URL: https://api.cmocka.org/#main-test (visited on 11/06/2018).

[19] *cmocka - unit testing framework for C*. URL: https://cmocka.org/ (visited on 11/06/2018).

[20] 11 Oct 2018 Cao Jin Feed 19up. *Exploring the Linux kernel: The secrets of Kconfig/kbuild*. en. URL: https://opensource.com/article/18/10/kbuild-and-kconfig (visited on 09/11/2018).

[21] 11 Oct 2018 Cao Jin Feed 52up. *Exploring the Linux kernel: The secrets of Kconfig/kbuild*. en. URL: https://opensource.com/article/18/10/kbuild-and-kconfig#build (visited on 29/05/2019).

[22] *Git - About Version Control*. URL: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control (visited on 30/01/2019).

[23] *Git - Reference*. URL: https://git-scm.com/docs (visited on 30/01/2019).

[24] *GitHub - knuto/ktf: Kernel Test Framework - a unit test framework for the Linux kernel*. URL: https://github.com/knuto/ktf (visited on 31/08/2018).

[25] *GNU Make*. en. URL: https://www.gnu.org/software/make/ (visited on 29/05/2019).

[26] GoogleTechTalks. *Greg Kroah Hartman on the Linux Kernel*. URL: https://www.youtube.com/watch?feature=player_detailpage&v=L2SED6sewRw#t=867s (visited on 25/04/2019).

[27] *Group maintainership models [LWN.net]*. URL: https://lwn.net/Articles/705228/ (visited on 24/10/2018).

[28] *How to Compile the Linux Kernel*. en. Sept. 2010. URL: https://www.linux.com/learn/how-compile-linux-kernel (visited on 11/01/2019).

[29] *How to fix a locale setting warning from Perl?* URL: https://stackoverflow.com/questions/2499794/how-to-fix-a-locale-setting-warning-from-perl (visited on 01/03/2019).

[30] *HOWTO do Linux kernel development — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/process/howto.html#mailing-lists (visited on 28/05/2019).

[31] *HOWTO do Linux kernel development — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/process/howto.html#the-development-process (visited on 28/05/2019).

[32] *Kernel Korner - Using RCU in the Linux 2.5 Kernel | Linux Journal*. URL: https://www.linuxjournal.com/article/6993 (visited on 23/05/2019).

[33] *Kernel module - ArchWiki*. URL: https://wiki.archlinux.org/index.php/Kernel_module (visited on 29/04/2019).

[34] *Kernel self tests [LWN.net]*. URL: https://lwn.net/Articles/608959/ (visited on 11/10/2018).

[35] *Kernel Test Framework documentation*. URL: http://heim.ifi.uio.no/~knuto/ktf/index.html (visited on 07/06/2018).

[36] *KernelBuild - Linux Kernel Newbies*. URL: https://kernelnewbies.org/KernelBuild (visited on 05/02/2018).

[37] *kunit: introduce KUnit, the Linux kernel unit testing framework [LWN.net]*. URL: https://lwn.net/Articles/769358/ (visited on 21/11/2018).

[38] *KUnit - Unit Testing for the Linux Kernel — The Linux Kernel 4.19.0-00033-gc58019fb4fe15 documentation*. URL: https://google.github.io/kunit-docs/third_party/kernel/docs/ (visited on 22/11/2018).

[39] LinaroOrg. *YVR18-313:What is LKFT and how does it improve the Linux kernel overall quality*. URL: https://www.youtube.com/watch?v=LMs7vCGv8as (visited on 11/04/2019).

[40] Linux Foundation Events. *Introduction to the Fuego Test System by Tim Bird*. URL: https://www.youtube.com/watch?v=YbL8oauJv1c (visited on 30/04/2019).

[41] *Linux Kernel Selftests*. URL: https://www.kernel.org/doc/Documentation/kselftest.txt (visited on 04/06/2018).

[42] *Linux Kernel Testing and Debugging | Linux Journal*. URL: https://www.linuxjournal.com/content/linux-kernel-testing-and-debugging (visited on 31/05/2019).

[43] Linux Plumbers Conference. *LPC2018 - XArray*. URL: https://www.youtube.com/watch?v=5EFh4v8vwoc (visited on 04/02/2019).

[44] *LKFT tests the following branches*. en. URL: https://lkft.linaro.org/branches/ (visited on 11/04/2019).

[45] *LTP - Linux Test Project*. URL: http://linux-test-project.github.io/ (visited on 08/02/2018).

[46] *Maintainers Don't Scale*. URL: https://blog.ffwll.ch/2017/01/maintainers-dont-scale.html (visited on 24/10/2018).

[47] Greg Marsden. *Oracle's new Kernel Test Framework for Linux*. URL: https://blogs.oracle.com/linux/oracles-new-kernel-test-framework-for-linux-v2 (visited on 21/11/2018).

[48] Greg Marsden. *Writing kernel tests with the new Kernel Test Framework (KTF)*. URL: https://blogs.oracle.com/linux/writing-kernel-tests-with-the-new-kernel-test-framework-ktf (visited on 21/03/2019).

[49] *On Linux kernel maintainer scalability [LWN.net]*. URL: https://lwn.net/Articles/703005/ (visited on 25/10/2018).

[50] *Overview | CMake*. en-US. URL: https://cmake.org/overview/ (visited on 29/05/2019).

[51] PyCon CZ. *Amador Pahim: Automated Testing Framework*. URL: https://www.youtube.com/watch?v=eTR-LvW80pM (visited on 30/04/2019).

[52] *re — Regular expression operations — Python 3.7.2 documentation*. URL: https://docs.python.org/3/library/re.html (visited on 07/02/2019).

[53] RedHatCzech. *Lucas Meneghel Rodrigues - Avocado - Next Generation Test Framework*. URL: https://www.youtube.com/watch?v=xMXS7NB4WSs (visited on 30/04/2019).

[54] *Regular Expression HOWTO — Python 3.7.2 documentation*. URL: https://docs.python.org/3/howto/regex.html (visited on 06/02/2019).

[55] *Relativistic hash tables, part 1: Algorithms [LWN.net]*. URL: https://lwn.net/Articles/612021/ (visited on 14/03/2019).

[56] *sed (1) - stream editor for filtering and transforming text*. en. URL: /man-page/Linux/1/sed/ (visited on 27/05/2019).

[57] *start [Kernel self-test]*. URL: https://kselftest.wiki.kernel.org/ (visited on 31/05/2019).

[58] *The following boards are used to test Linux kernels in LKFT*. en. URL: https://lkft.linaro.org/boards/ (visited on 11/04/2019).

[59] *The Kernel Configuration and Build Process | Linux Journal*. URL: https://www.linuxjournal.com/article/6568 (visited on 08/11/2018).

[60] *The Linux Kernel Archives - Releases*. URL: https://www.kernel.org/category/releases.html (visited on 31/05/2019).

[61] *The Need for Speed and Automation | Linux Journal*. URL: https://www.linuxjournal.com/article/7160 (visited on 27/03/2019).

[62] *The rhashtable documentation I wanted to read [LWN.net]*. URL: https://lwn.net/Articles/751374/ (visited on 24/09/2018).

[63] *The XArray data structure [LWN.net]*. URL: https://lwn.net/Articles/745073/ (visited on 22/11/2018).

[64] Josh Triplett, Paul E McKenney and Jonathan Walpole. 'Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming'. en. In: (), p. 14.

[65] *Which Linux Kernel Version Is 'Stable'?* en. Feb. 2018. URL: https://www.linux.com/blog/learn/2018/2/which-linux-kernel-version-stable (visited on 31/05/2019).

[66] Matthew Wilcox. *Documentation/core-api/xarray.rst*. (Visited on 29/01/2019).

[67] *XArray — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/latest/core-api/xarray.html (visited on 26/11/2018).