Linux kernel device driver testing

How are device drivers being tested?

Marcelo Schmitt

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Master of Science

Program:	Computer Science
Advisor:	Prof. Dr. Paulo Roberto Miranda Meirelles
Coadvisor:	Prof. Dr. Fabio Kon

This study was financed in part by the Coordenação de Aperfeiçoamento
de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001

São Paulo
October, 2022

Linux kernel device driver testing

How are device drivers being tested?

Marcelo Schmitt

Acknowledgments

For supporting me throughout this work, I am grateful to my mother, Marlene, and my brother Victor, who helped me a lot over the time it took to complete this dissertation. I thank my advisors, Paulo Meirelles and Fabio Kon, for supporting and guiding me during this study. A great thanks to CEPID (Centro de Pesquisa, Inovação e Difusão em Neuromatemática) for hosting our Device Driver Testing Survey and to Nelson Lago, Rodrigo Siqueira, Melissa Wen, Lucas Stankus, André Almeida, Nícolas Prado, and The Linux Foundation Technical Advisory Board staff for providing expert insights on the design of our survey. I want to acknowledge Antonio Terceiro and Auri Vincenzi for reviewing this work and providing many improvement suggestions. Last but not least, a special thanks to all Linux kernel maintainers who participated in the aforementioned survey. Your feedback supported our research with valuable information about the state of practice of Linux device driver testing. This study would not comprise such a broad set of insights without your help. Thank you all.

Resumo

Marcelo Schmitt. **Teste de Drivers de Dispositivo do kernel Linux: *Como drivers de dispositivo estão sendo testados?***. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Drivers de dispositivo são uma parte essencial do kernel do Linux. Bugs nesses componentes podem comprometer a estabilidade de qualquer sistema operacional GNU/Linux. Para mitigar isso, os drivers de dispositivo devem ser testados em vários cenários de caso de uso. No entanto, isso nem sempre é facilmente alcançável porque os drivers de dispositivo dependem de componentes de hardware que podem operar de forma não determinística, falhar inesperadamente ou estar indisponíveis para os desenvolvedores. Esta pesquisa caracteriza como os drivers de dispositivo do kernel Linux são testados. Para isso, realizamos um mapeamento sistemático de literatura formal, uma revisão da literatura cinzenta e uma pesquisa com mantenedores de drivers de dispositivos do Linux. Por meio desses métodos de pesquisa, podemos oferecer uma visão abrangente do estado da prática dos testes de drivers de dispositivo do kernel Linux. Resumimos as informações reunidas em um catálogo de ferramentas de teste usadas para testar o kernel do Linux e seus drivers de dispositivo. Além disso, avaliamos as ferramentas que se mostraram mais promissoras para uso diário por desenvolvedores Linux. Por fim, oferecemos um ampla caracterização das ferramentas de teste do kernel Linux.

Palavras-chave: Linux. Teste de Software. Driver de dispositivo.

Abstract

Marcelo Schmitt. **Linux kernel device driver testing: *How are device drivers being tested?***. Thesis (Master's). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Device drivers are an essential part of the Linux kernel. Bugs in these components may compromise the stability of any GNU/Linux operating system. To mitigate that, device drivers should be tested against many use case scenarios. However, that is not always easily achievable because device drivers rely on hardware components that might operate nondeterministically, fail unexpectedly, or be unavailable to developers. This research characterizes how Linux kernel device drivers are tested. To accomplish that, we carried out a mapping study, a grey literature review, and a survey with Linux device driver maintainers. Through these research methods, we are able to offer a comprehensive overview of the state of the practice about tests on Linux kernel device drivers. We have summarized the information gathered in a catalog of test tools used to test the Linux kernel and its device drivers. Further, we have evaluated those tools that showed the most promising for daily use by Linux developers. Finally, we offer an extensive characterization of Linux kernel testing tools.

Keywords: Linux. Software Test. Device driver.

List of Figures

List of Tables

List of Programs

Contents

Chapter 1

Introduction

Device drivers are an important part of the Linux kernel and represent about 66%[1] of the project code lines. Although the code portion that drivers represent in a conventional operating system (OS) can vary, some of these components are indispensable to the system's operation. In addition, the Linux kernel is widely used in a series of applications as cloud service providers, embedded systems, smartphones, and supercomputers (Corbet and Kroah-Hartman, 2017). Thus, testing is fundamental to increase the Linux kernel's confidence level and the operation of GNU/Linux systems submitted to different workloads. In particular, device drivers require differentiated approaches because they are relatively difficult to test. A fact that instigates to question: how are device drivers being tested? This work tries to find out how developers are testing Linux kernel device drivers.

1.1 Problem outline

From the experience of our research group in Linux kernel development and a systematic mapping study conducted during this work, we have found evidence that (1) the testing tools proposed by academic works are not being used by the Linux development community and that (2) the testing tools that kernel developers are using are not well covered in the academic literature.

There is a disconnection between academia and industry regarding practices for testing Linux kernel device drivers. On the one hand, Kernel developers do not adopt the testing tools proposed by academia. On the other hand, test tools extensively promoted by the Linux community are not the object of academic study. This scenario leads us to consider that academia does not fully understand the testing needs of Linux kernel developers, whereas the project may not leverage innovative ideas published by academia.

Thus, we identified the following research problems:

- There are no references in the academic literature reporting the testing practices and tools used by the Linux kernel development community.

[1] Estimate calculated with data from cloc tool: https://github.com/AlDanial/cloc

- Researchers spend time developing several software testing tools, but the Linux kernel development community does not adopt them.
- No publication approaches Linux kernel device driver developer's difficulties regarding driver testing.

1.2 Research Objectives

By including the Linux kernel as our research target, we cannot ignore the role of the community in maintaining such a massive codebase. In recent years, more than 4000 people have contributed to the advancement of the Linux kernel each year (Stewart *et al.*, 2020). Thus, we outline our goals to contribute to both academia and the development community.

We pursued the following research objectives:

RO1. Provide a list of tools used by the Linux kernel community to test the Linux kernel.

RO2. Identify testing strategies and tools used by Linux kernel device driver maintainers.

RO3. Improve coverage of formal literature regarding the state-of-the-practice of Linux kernel device driver testing.

By achieving *RQ1*, we produced a list of test tools that provide a reference for further academic works related to software testing and the Linux kernel. In accomplishing *RO2*, we boost researchers and practitioners by pointing out promising paths for those considering studying or contributing to existing tools. Upon reaching *RO3*, we hope researchers can make better-guided decisions when proposing new testing tools for the Linux kernel or even consider contributing to existing ones. We have evidence that some of the test tools introduced by academics have been discontinued and fallen into disuse. Falling into disuse, however, is not a phenomenon restricted to Linux kernel testing tools. About 40% of the static analysis tools published in ASE (International Conference on Automated Software Engineering) and SCAM (International Working Conference on Source Code Analysis & Manipulation) between 1991 and 2015 are in a closedown stage (Costa *et al.*, 2018).

1.3 Research Questions and Plan

The problems described in Section 1.1 suggest an update of the academic literature. Furthermore, our objectives require the observation of subjective aspects linked to the community. After all, the perception of a contribution as an improvement depends on the community's opinion. Linux kernel maintainers accept patches only when submitters convince them the proposed changes are beneficial. Patches are pulled to the kernel, not pushed. While it is possible to ask many questions about the testing practices adopted by the Linux community, we limited the scope of this work to the procedures focused on testing device drivers. Assessing test methods from all the Linux subsystems would require investigating a much larger material. In this work, we seek to provide reasonable answers to the following questions:

RQ1. How are Linux device drivers being tested?

RQ2. What testing tools are being used by the Linux community to test the kernel? What are the main features of these tools?

RQ3. What features does the community desire for a testing tool? About the tools that are already in use, what could be improved?

This research took place in 3 primary phases. In the first phase, we conducted a mapping study; in the second phase, we conducted a grey literature review (GLR); and in the third phase, we conducted a survey with Linux kernel device driver maintainers. We then triangulate our findings from academic papers, informal literature, and community developers by synthesizing different points of view on device driver testing.

1.4 Thesis Structure

The remaining of this work is structured in four more chapters. Chapter 2 provides some background on subjects related to the Linux kernel device driver and software testing. We describe the applied research methods in Chapter 3. Chapter 4 presents and examines the results of our research. In Chapter 5, we conclude this work by discussing the limitations of our work and possible future investigations.

Chapter 2

Background

To discuss the state-of-the-practice of Linux kernel testing, it is worth briefly reviewing some related concepts.

2.1 Linux kernel device drivers

The Linux kernel is accounted for many essential operating system tasks such as memory management, process scheduling, data storage, network communication, and many others (*About Linux Kernel* 2021). The kernel must operate several devices with highly distinct characteristics and complexity to provide fundamental system functionality. Moreover, it is reasonable to avoid mixing the control logic of different devices with one another and with core system logic. Thus, it is usual to encapsulate code for managing a device (or a family of related devices) into a device driver. “A driver is a piece of software whose aim is to control and manage a particular hardware device, hence the name device driver” (Madieu, 2017). To be more specific, we consider that a device driver is characterized by a well-delimited piece of code (usually a file or a few files) whose purpose is to control the operation of a hardware design (or a set of related hardware designs). Usually, hardware designs are described in documents called datasheets or blueprints, which in turn describe the components and operation of a hardware device.

Entire subsystems and kernel portions not restricted to the operation of a single device (or set of devices), e.g., file systems, network stack, process scheduler, memory manager, etc., are not device drivers. Although these components may contain device drivers, they are not device drivers. Next, we must attend to the concepts of software testing.

2.2 Linux Kernel Development Process

At each release, Linux incorporates changes from hundreds of developers worldwide. These developers may contribute to the project in many ways, such as by improving the documentation, fixing bugs, introducing new features, and providing support for new device drivers. The contributions to Linux (often called **patches**) are sent through email to appropriate mailing lists. There are several mailing lists at which Linux developers and maintainers review and discuss changes to the kernel.

When a maintainer accepts a patch, they include it in their development repository. Many (if not all) Linux maintainers have their development repositories (or **trees**) hosted at Kernel.org[1]. Since development trees are publicly accessible, test rings may perform tests on early phases of Linux kernel development. Moreover, Linux kernel developers and maintainers may request test ring administrators to add their repositories to the test infrastructure. After a repository is added to a test ring, it gets periodically pulled for testing (KHAN, 2021b; KROAH-HARTMAN, 2022). Thus, after a patch gets into a development tree, it may be subjected to many tests from Linux kernel test systems.

The Linux kernel source code is logically divided into several subsystems. “A subsystem is a representation for a high-level portion of the kernel as a whole (A. R. JONATHAN CORBET and KROAH-HARTMAN, 2005).” A subsystem may also be understood as an abstraction to refer to some part of the kernel responsible for some system functionality, such as process scheduling, memory management, networking, etc. Most subsystems have one or more developers who take the overall responsibility for the code on that subsystem. These developers are known as subsystem maintainers (*How the development process works* 2021).

At the beginning of each development cycle, Linus Torvalds declares he will accept new features for the Linux kernel throughout a period known as the **merge window**. Then, during a typically two-week span, subsystem maintainers ask Linus to add (pull) changes from their trees into his repository (known as the **mainline** kernel). After pulling patches during those couple of weeks, Linus declares the merge window closed and stops merging new features for the next Linux release. The kernel produced by Linus at the end of a merge window is an artifact that urges testing since it is the bedrock for the upcoming Linux release.

The weeks that follow the merge window are known as a stabilization period during which Linus and many other kernel developers try to fix as many bugs and regressions as possible. That is as also a time of intense testing by robots, automated test systems, and test rings. The testing and bug-hunting season usually lasts six to eight weeks until Linus declares the release candidate to be the new Linux kernel release (*How the development process works* 2021). After that, Linus opens a new merge window and the process repeats for a newer Linux release.

2.2.1 Next trees

Reviewers, testers, maintainers, and developers alike may want to view the changes queued for the next kernel release in an integrated form so they can avoid merging conflicts. However, pulling and merging patches from several subsystem trees is a cumbersome and error-prone task.

To overcome this problem, the community provides **-next trees**, which bring together patches from several subsystems. The main tree for merging patches queued for the next

[1] https://git.kernel.org/

Linux release is linux-next. By design, linux-next is a snapshot of how the mainline should be after the next merge window closes (*How the development process works* 2021). Since linux-next comprises changes intended for the forthcoming Linux kernel, the tree may enable contributions to be tested weeks ahead they reach the mainline. Refer to *How the development process works* (2021) for a detailed reference on the Linux kernel development process.

2.3 Software Testing

Software testing is "an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" ("IEEE Standard Glossary of Software Engineering Terminology" 1990). We, however, consider a slightly broadened concept of software testing by also comprising compilation time code inspection such as checks made by static analysis tools. We believe this notion better aligns with complementary definitions of software testing. For instance, Claudi and Dragoni (2011) stated that "in software engineering, testing is the process of validation, verification and reliability measurement that ensures the software to work as expected and to meet requirements". Likewise, Mathur (2013) suggests that software testing "is the process of determining if a program behaves as expected." We understand that analyzing the structures and properties of the source code may provide insight into whether a program will function as desired, thus possibly making part of a software testing activity.

Nevertheless, regardless of the program under test, it is intrinsic to the software testing activity to compare measured behavior with the desired behavior described by formal or informal requirements (Mathur, 2013). Moreover, many hardware manufacturers provide datasheets describing the components and the functioning of the devices they supply. Therefore, we may say that device driver testing is the act of evaluating, validating, or verifying whether a device driver (or parts of it) operates the hardware it supports as described by its design. However, the reality is more complicated than the theory, meaning not all hardware devices work precisely as their datasheets describe or even come with accessible blueprints. In such cases, device drivers must also deal with nonconforming behavior or resort to reverse engineering.

Mathur (2013) proposed a comprehensive categorization of software testing techniques based on four classifiers. The classifiers ponder the resources for generating the tests, the development phase in which the tests are carried out, the objective of the test activity, and the characteristics of the artifact under test. However, those classifiers are not well suited for distinguishing between Linux kernel test practices for a few reasons.

First, Linux is a free software project, so the source code is accessible to everyone. Therefore, the kernel source could be used as an inspiration for any testing practice. Thus, testing techniques usually classified as black-box could be considered black-box and white-box. For instance, Andrey Konovalov advises reading the code to understand what types of input the system expects and to identify which parts of it can be targeted in the context of fuzzing the Linux kernel (Konovalov, 2021b).

In addition, a kernel developer has access to all phases of the project lifecycle. They can run unit tests on the initial versions of the developed code. There is even a framework called KUnit for writing unit tests built into the kernel. After making the desired changes, developers may compile and install the kernel for integration testing. At this testing phase, subsystem code can be exercised manually or with the help of additional testing tools such as Kselftest. Also, kernel developers often work on improvements to existing functionality. In such cases, the tests performed can be considered regression tests. Since GNU/Linux distributions use or adapt the mainline or stable kernels to make the operating system, testing against any of these trees is also a form of beta-testing (*Fedora Linux Kernel Overview* 2021; *Kernel - Fedora Project Wiki* 2021; *About Debian* 2021).

Also, developers are interested in testing the Linux kernel to identify regressions. When responding to a bugfix rollback, Torvalds (2007) says why regressions are particularly unwanted in the kernel:

> Because it is much more important to make slow, but steady progress and have people know things improve (or at least not "deprove"). We do not want any kind of "brownian motion development".

Other reasons to test the kernel may include checking the behavior under invalid inputs or high workloads, verifying compatibility with external components, investigating security aspects, and more. Thus, robustness testing, stress testing, interface testing, and security testing are examples of tests to which the Linux kernel can be submitted. There are no GUI tests as Linux does not contain any GUI components.

Finally, considering the artifact under test classifier, testing techniques applied over Linux can be classified as operating system testing or merely code testing. That said, much of the testing over the Linux kernel would be classified as black-and-white-box regression OS testing according to Mathur's classifiers. However, to provide a more informative categorization of kernel testing tools, we renounce Mathur's classifiers to appraise the means used to perform kernel tests. Thus, we decided to consider as a test technique the answers to the question: what was done to test X? Where X is some device driver or parts of one. For instance, someone could decide to test a device driver by feeding random values to its interface (fuzzing), creating a model of it and using properties of that model to perform tests (model-based testing), instrumenting the code to measure runtime activity (performance testing, stress testing), etc.

Furthermore, different test techniques imply different stages of the generation and execution of test cases and subsequent analysis of results. For example, tools based on models (model-based testing) start from a program model for test generation. Fault injection tools need to instrument code or intercept system calls to test target software. In both cases, an essential preparation step is done either before specifying which properties to test or before defining fault injection sites. We decided to look upon such preliminary activities as part of the test case generation. We have regarded the tasks related to the execution of software components and the control of the conditions under which code execution takes place solely as "test execution". In addition, we took the term "test assessment" to refer to the activities related to assessing aspects of the software under test. With this, it was possible to display the characteristics of the tools concisely in Table 3.2 (presented in Chapter 4).

Chapter 3

Research Methods

To answer the proposed research questions, we resorted to a systematic mapping study, a grey literature review, and a survey with Linux kernel device driver maintainers. First, we conducted a literature mapping study to acquire information about device driver test tools from consolidated peer-reviewed articles published in media that follow the scientific methodology. After that, we extended our body of knowledge about Linux kernel test tools with data from a grey literature review. Lastly, we surveyed device driver maintainers to validate preliminary conclusions and further investigate topics related to driver testing.

3.1 Literature Systematic Mapping Study

To get an overview of the testing techniques and tools used on Linux, we conducted a mapping study of the formal literature. For this task, we consulted the following digital libraries:

- ACM Digital Library (https://dl.acm.org)
- IEEE Xplore (https://ieeexplore.ieee.org)
- Scopus (https://www.scopus.com)

We chose these publication search engines because they provide valuable publications on the computer science field of study. ACM Digital Library contains more than 600,000 full-text articles from leading computing researchers (Computing Machinery, 2021). IEEE Xplore provides access to more than five million documents from highly-cited publications in electrical engineering, computer science, and electronics (IEEE, 2021). Lastly, Scopus indexes comprehensive content from over 25,000 active titles and 7,000 publishers. It covers 240 disciplines and claims to greatly reduce the odds of missing key publications (Elsevier, 2021b; Elsevier, 2021a).

After defining the search libraries for the mapping study, we created a comprehensive search string that would allow us to get several articles related to Linux kernel testing. We then derived the search string from the objects of interest for this search. This study is limited by the broader topic of GNU/Linux operating systems. Within that topic, we are interested in the Linux kernel. Finally, from the many subjects related to Linux, we want to analyze device driver tests. So the broader context for us is *Linux*, from which we observe its *kernel*[1]. In addition, we look for *test* practices related to *device drivers*. Our search string could be something like "linux AND kernel AND test AND driver", but we decided not to use the last level of specificity to minimize the risk of missing relevant publications. A test tool may not be designed to test device drivers, but Linux developers might find a way to use it for that. Finally, we added synonyms and terms related to the objects of interest. Table 3.1 shows the words used in the search string. The columns are connected with AND, and items within columns are connected with OR.

Research area	Subarea 1	Subarea 2
linux	kernel	test
	kernel space	testing
	operating system	validation
	subsystem	verification

Table 3.1: *Search string terms for the mapping study.*

The complete search string is as follows:

linux AND (kernel OR "kernel space" OR "operating system" OR subsystem) AND (test OR testing OR validation OR verification)

We queried each library on 2021-01-21 and saved the metadata from the articles retrieved in the files *scopus.csv*, *ieee.csv*, and *acm.bib* available at https://gitlab.com/Marcelosc/ime-usp-masters-dissertation/-/tree/dissertation-final/literature-review/academic.

3.1.1 Publication selection

The library search returned 5,018 articles, which underwent a selection process. In the first stage, a title analysis was carried out, especially keeping those containing the word *test* or some variant of it. When in doubt about some title, we included it on the list. We also removed duplicated results and non-computer science work. From this first selection stage, we selected 399 articles, but, as it was still a large number of publications, we had to be more judicious in the second selection stage. So, we kept only articles whose titles contained the terms *linux*, *kernel*, *test*, some variant of the word *test*, or that somehow referred to software tests.

In the third stage, we read the abstract from the 62 articles selected in the previous step, classifying them into four categories according to the characteristics of each work.

- Articles that presented the Linux kernel as a means of testing conventional Linux applications (developed in user space) were rated as of little relevance.

[1] Linux is the name of the kernel of Linux-based operating systems. So, technically, the pair *Linux kernel* is a redundancy. However, misuses of the term Linux abound in the literature such that not adding the term kernel to the search string would bring lots of unrelated work. Also, we use the pair *Linux kernel* throughout this work to make clear that we are talking about an operating system kernel.

- Articles that presented changes to the Linux kernel intending to implement software testing for conventional Linux applications were rated as relevant.
- Articles that presented some means of testing the Linux kernel as a whole or parts of it were rated as very relevant.
- Articles that could have been disregarded based on previous criteria but retained for the benefit of the doubt were classified as irrelevant.

Next, in the fourth selection step, we did a speed reading to select articles that specifically focused on testing the Linux kernel or its components. Works studying GNU/Linux systems as a single component were disregarded, even when indirectly testing parts of the kernel. We also dismissed articles that did not present test results or tools, even when presenting test practices or discussing components to be tested. With these criteria, 19 articles were selected for full reading, thus completing the fourth selection stage.

3.1.2 Publication assessment

We then fully read each of the 19 articles selected from the speed reading step to understand which techniques and testing tools are used to test Linux kernel drivers. We found articles describing fault injection testing, fuzzing, static and dynamic code analysis, symbolic execution, hardware virtualization, and simulation. Table 3.2 lists the papers appraised in the systematic mapping study, whether they reported any testing tool, which test techniques were explored, and if tests were automated.

From 19 academic papers, we identified 17 test tools and one debug tool designed for the Linux kernel. From those, we selected 6 test tools for usage assessment because they presented means of automated testing device drivers that kernel developers could use in the early development stages. Finally, Table 3.3 summarizes the mapping study phases and the number of publications handled.

3.2 Grey Literature Review

Not satisfied with the evidence gathered from the academic literature, we decided to further investigate device driver tests by conducting a systematic review of non-academic publications. For analyzing non-science-oriented documents, we followed the grey literature review (GLR) guidelines proposed by Wen *et al.* (2020). Our main objective in this investigative phase was to seek answers to research questions *RQ2* and *RQ3*. Also, careful exploration of non-academic publications may strengthen our research's validity by exploiting an additional source of evidence.

3.2.1 Grey Literature Review Planning

According to Wen *et al.* (2020), a GLR plan covers four essential steps to ensure the quality of the documents selected for analysis:

1. Outline the problem and define the research question.
2. Define the inclusion and exclusion criteria.

Identifier and Reference	*Tool Name*	*Testing Techniques*	*Automated*
L1 B. Chen *et al.* (2020)	COD	Concrete and symbolic code execution	Test case generation & replay
L2 Cong *et al.* (2015)	ADFI	Fault injection	Test scenario generation & execution
L3 Zaidenberg and Khen (2015)	LgDb	Does not apply	Does not apply
L4 Garn and Simos (2014)	Eris	Model based & combinatorial testing	Test suite generation, execution & assessment
L5 Mohan *et al.* (2018)	CRASHMONKEY and ACE	Fuzzing	Test case generation, execution & assessment
L6 Buchacker and Sieh (2001)	FAU Machine	Fault injection	Not
L7 Zhai *et al.* (2008)	Not named	Unclear	Unclear
L8 Shahpasand *et al.* (2016)	TIMEOUT	Fault injection	Test case generation
L9 Y. Chen *et al.* (2013)	KIS	Static & dynamic analysis	Yes, remote service
L10 Drebes and Nanya (2008)	DMA Fault Injector	Fault injection	Not
L11 Kim *et al.* (2009)	MOKERT	Model based testing & model checking	Model generation & fail replay
L12 Renzelmann *et al.* (2012b)	SymDrive	Symbolic execution	Stub generation
L13 Cai *et al.* (2007)	UKTI	Component emulation	Unclear
L14 Bai *et al.* (2016)	EH-Test	Fault injection	Test case generation & execution
L15 D. Chen *et al.* (2020)	Dogfood	Model based testing	Test case generation & execution
L16 Claudi and Dragoni (2011)	Lachesis	Model based & fuzz testing	Unclear
L17 Yuqing *et al.* (2012)	ScheduleBench	Performance (instrumentation) testing	Not
L18 Rothberg *et al.* (2016)	Troll	Configuration testing	Does not apply
L19 K.P *et al.* (2015)	-	Component emulation	No

Table 3.2: *Articles selected by systematic mapping study study.*

Selection phase	Number of articles (in -> out)
1) Title analysis	5018 -> 399
2) Title selection	399 -> 62
3) Abstract analysis	62 -> 27
4) Speed reading	27 -> 19
5) Full reading	19 -> 6

Table 3.3: *Mapping study phases*

3. Develop a relaxed search string.
4. Define the resource-types to consider.

We now provide specifications for these components, each tuned to conduct a GLR to gather information about the tools used for Linux kernel testing.

Problem outline & research questions

At the end of our mapping study, we evaluated the use of 6 Linux test tools proposed by selected academic publications. The conclusions of this investigative activity are presented in Subsection 4.1.2. We could not run five out of six tools that underwent usage assessment. That result indicates that some tools proposed by academic publications might not be maintained anymore. Because of that, we suspected that the Linux kernel development community was not using some tools promoted by scientific articles. Also, the academic publications selected by our mapping study do not examine the development community's engagement in using the reported test tools nor any discussion of their expectations concerning Linux test tools. In addition, from our previous experience with Linux kernel development, we have evidence that academic publications do not cover some of the test tools used in daily Linux development. To reduce the apparent gap between the formal literature and the state-of-the-practice of Linux kernel device driver tests, we defined the organization of a list of Linux kernel test tools as one of our research objectives (*RO1*). Reaching *RO1* should be a natural accomplishment for answering *RQ2* and *RQ3*. Therefore, the research questions *RQ2* and *RQ3* guided our investigation through the grey literature content.

Inclusion and exclusion criteria

Collecting information from non-traditional sources may harm the validity of the work since non-academic publications do not necessarily follow the scientific methodology or undergo peer review. To mitigate the risks posed by including grey literature as a source of information, we limited our searches to a select group of sites that we have come to refer to as data sources.

In addition, we defined selection criteria to take documents from the data sources. We adjusted Wen *et al.* (2020) inclusion and exclusion criteria to the specificities of our study. Instead of exclusion criteria, we applied pre-inclusion criteria throughout each speed reading step to include only documents relevant to our research objectives. Additionally, the inclusion criteria were used during each complete read step to select publications with pertinent information to our study. Applying these criteria, we should reduce the risks of including information from non-academic sources in our research.

Pre-inclusion criteria

- The document is publicly accessible.
- Available in English.
- Published between 2011 and 2021.
- Most current version of the document.

- The content is not centered on social issues or flame wars.
- Published online by institutions, industry-oriented magazines, and practitioners of the FLOSS area.
- The document is published by: (1) a reputable organization or magazine; (2) an individual author associated with such organizations and magazines; or (3) a practitioner with more than five years of FLOSS experience.
- The search terms are used in a context somewhat related to kernel testing.

Inclusion Criteria

It refers to software testing (automated or not) in the Linux kernel by:

1. (1) reporting practices;
2. (2) presenting statistics;
3. (3) expressing an opinion;
4. (4) or studying the project development or its community.

Develop a relaxed search string

The search string used in the mapping study has been modified to return results related only to the Linux kernel and augmented to include more words related to testing.

linux AND kernel AND (test OR testing OR validate OR validation OR verify OR verification)

Table 3.4 shows the terms used in the search. Columns are connected by AND and words within the same column have been connected by OR.

Research area	Subarea 1	Subarea 2
linux	kernel	test testing validate validation verify verification

Table 3.4: *Search String Terms for GLR.*

Wen *et al.* (2020) reported that different data sources offer diverse capabilities. According to them, some data sources may provide filters for a specific type of category or tag or even allow regex in search terms. However, not all data sources have tools with advanced search options or a clear description of how searches are done. To surpass these differences and standardize our search method, we assumed that each data source search mechanism is capable of searching for terms linked by *and*. Such capability granted, we decomposed the search string into smaller strings, s1 to s6, such that the merged results of searching from s1 through s6 would be equivalent to the results obtained using the original search string.

Search string variations:

1. s1 = **linux AND kernel AND test**
2. s2 = **linux AND kernel AND testing**
3. s3 = **linux AND kernel AND validate**
4. s4 = **linux AND kernel AND validation**
5. s5 = **linux AND kernel AND verify**
6. s6 = **linux AND kernel AND verification**

Define the resource-types to consider

We were able to shorten the data source selection phase by taking advantage of the selection of Linux kernel data sources provided by Wen *et al.* (2020). With their data sources and a couple of additional ones, we defined a set of websites with embedded search engines to conduct our document search. Each data source in Table 3.5 is maintained by prominent organizations and developers that have long been committed to Linux development.

Data Source	*URL*
The Linux Foundation	linuxfoundation.org
Linux.com	linux.com
Kernelnewbies	kernelnewbies.org
Linux Weekly News (LWN)	lwn.net
Linux Journal	linuxjournal.com
The Linux Kernel documentation	kernel.org/doc/html/latest/index.html

Table 3.5: *Data sources with search engine.*

The first data source in our list is linuxfoundation.org, maintained by The Linux Foundation. The foundation hosts Linux and supports its creator Linus Torvalds and lead maintainer Greg Kroah-Hartman (*A Beginner's Guide to Linux Kernel Development (LFD103)* 2022). The Linux Foundation also manages The Linux Kernel Organization by providing technical, financial, and staffing support for running and maintaining Linux related infrastructure (*The Linux Kernel Archives - About* 2022).

The second reference in our list is linux.com, a news, information, and tutorials website which aims to inform and prepare open source professionals who are building the next generation of open technologies. Since 2009, Linux.com is also hosted by The Linux Foundation (*About Linux.com* 2022; *Linux Foundation to Build New Linux.com Community* 2022).

Next on our list is kernelnewbies.org, a wiki maintained by a community of aspiring Linux kernel developers who work to improve their Kernels and more experienced developers willing to share their knowledge (*Linux_Kernel_Newbies* 2022). Kernelnewbies' wiki has a known group of editors, including distinguished Linux kernel developers such as Greg Kroah-Hartman, Julia Lawall, Jonathan Corbet, Daniel Vetter, and many others (*EditorsGroup* 2022).

Linux Weekly News (LWN) and Linux Journal are well-known magazines that publish news and articles related to Linux and associated open source projects. LWN began at the end of 1997 as a consulting company's side project through which its editors shared the results of their efforts to keep up with developments from all over the Linux community. Over the years, LWN has grown with Linux and become one of the definitive Linux news sites, intending to be the premier news and information source for the free software community (*The LWN.net FAQ* 2022). Linux Journal had its kick-off in 1994 and thus comprised more than 25 years of publications as of April 2021. Dedicated to delivering publications that cultivate the Open Source philosophy principles, the magazine explores trending, timeless and practical topics about Linux and related technologies (Searls, 2019; *About Linux Journal* 2022).

Lastly, the Linux Kernel documentation is the official project's documentation maintained within the project repository itself (*index.rst « Documentation* 2022).

Even though we have not classified the collected documents into *shades of GL* as suggested by Adams *et al.* (2017), many of the selected publications would fit what they have called a second degree of grey literature. The documents hosted by our data sources mainly consist of news articles, documentation pages, wiki pages, and industry publications. We believe that the characteristics of the selected data sources and our pre-inclusion and inclusion criteria should guarantee a moderate degree of expertise and authority of the sources.

We decided not to evaluate audiovisual content due to the effort required to transcribe and incorporate this type of material into the research. However, we do support slides linked to lectures in video format. To deal with the heterogeneous and weak search mechanisms in the selected data sources, we used our multipart search string to uniform the document search and collection procedure. Our systematic grey literature review process is described by the algorithms below.

Organization of the Grey Literature Review Process

We conducted the planned Grey Literature Review in three main phases: data collection, preliminary analysis, and complete analysis. The data collection phase aims to gather URLs for grey literature documents according to a well-defined procedure for applying each search string variant to each data source in Table 3.5. We began by taking a data source and its search engine to search for the first search string variant. We then collected the URLs for the first five search hits, annotated from what reference we stopped, and searched for the next search string variant. We repeated the previous step until we had dug all search string variants. When done, we took another data source and repeated it all over again. The data collection phase is over when all data sources have been searched.

We designed the preliminary analysis phase to verify which documents comply with each pre-inclusion criterion. Algorithm 3.1 describes this phase with a few organizational tasks needed to conduct the review. During the complete analysis phase, we assessed the content of documents and their relevance to the Linux kernel testing subject. Document appraisal at this phase ensured that each publication meets at least one inclusion criteria, cutting exerts reporting the state of the Linux kernel testing practice and collecting reference links for snowballing when appropriate. Algorithm 3.2 describes the complete analysis phase.

Program 3.1 Steps for preliminary assessment of GL documents.

```
  ▷ Main objective of evaluating whether the document meets the pre–inclusion factors
FUNCTION speed_read()
{
    Check if the publication is from 2011 or later.
    Read document title and abstract if it has. Write down the title. Give a title if it does
        not have one.
    Discard based on title if it is the same as another document already analyzed.
    Check the authorship of the publication.
    Check the publication source's reliability/reputation.
    Search (Ctrl + f) for each of the words: test; validat; verif.
    Identify the context in which the searched word appears.
  ▷ Identifying the context may consist of reading an entire paragraph.
    If it meets all the pre–incluson criteria, mark the document for the full read phase.
    if (document was marked) {
  ▷     This should avoid losing content with links that may come to break.
        Save the page in pdf format (File menu -> print)
        Save the page with the title name.
    } else {
        Write down the reasons for exclusion.
    }
}
```

The plan delineated for our GLR process also foresees a snowballing step that parallels the Backward Snowballing procedure described by Wohlin (2014). The guidelines for the snowballing procedure advise that it is important to decide on either inclusion or exclusion of publications before using them for snowballing. Our GLR process complies with that recommendation by assuring that every document listed for snowballing had been cited by an article we had selected for inclusion during the evaluation of publications from data source searches. The complete analysis phase (Algorithm 3.2) ensures that each document included by snowballing has been cited by at least one publication selected by an earlier document assessment process.

Although our data sources are different in formality level, we chose to treat each one equally, i.e., we made no distinction between them when searching for documents. Indeed, our data collection phase resembles a breadth-first search. Algorithm 3.3 provides a structured view of our GLR process.

Program 3.2 Steps for complete evaluation of GL documents.

```
 ▷ Main objective of evaluating whether the document meets the inclusion criteria
FUNCTION full_read()
{
    Full read the document and examine all information available.
    if (document meets at least one inclusion criterion) {
        Mark the document as selected.
        Give the document an ID and write down the reasons for its inclusion.
        Save the snippets that contain information pertinent to Linux kernel testing.
        Collect names, links, repositories, and any other data related to Linux test tools.
        if (document was obtained from a data source)  ▷ Single level snowballing.
            Based on context, add snowballing candidates to the non−evaluated list.

    } else {
        Write down the reasons for exclusion.
        if (document is a landing page just pointing to another document)
            Add the document being pointed to the non−evaluated list.
    }
    Mark the document as evaluated (remove it from the non−evaluated list).
}
```

Program 3.3 GLR process overview.

```
 ▷ Systematic GLR process
for (2 times) {
    for (each of the data sources with search engine) {  ▷ Document collection
        Take note of the days GL documents were collected.
        for (each search string variant) {
            Apply the search string over the data source.
            Append the next five results to the list of unassessed URLs.
            Take note of which result to continue from in the next iteration.
        }
    }
document_assessment:
    Remove documents with duplicate URLs from the spreadsheet.
    for (each document not evaluated) {  ▷ Preliminary document evaluation
        speed_read()
    }
    for (each marked document) {  ▷ Full document review
        full_read()
    }
    if (list of non−evaluated documents is not empty) {
        goto document_assessment;  ▷ Snowballing
    }
}
```

3.3 Community Survey

The third research method we employed was a survey. But, before we talk about the survey itself, let us first explain why a survey is valuable to help answer our research questions.

Linux kernel features are declared within files named *Kconfig*. The entries kept by *Kconfig* files may specify attributes such as symbol name, list of dependencies, and a help message. Kconfig entries are important because they allow Linux features to be conditionally compiled, which enables sensible decreases in the resulting kernel image size and building time.

Each device driver must declare a Kconfig entry. Moreover, it is common practice to provide the help attribute with information such as "Say yes here to build support for ... ", "This option provides functionality to ... ", or "This driver adds support for ... ". Thus, we can estimate the number of device drivers by counting how many of these pieces of advice appear in Kconfig files. By the 5.17 Linux release, a naive shell command can tally at least 2243 device drivers in the Linux kernel.

```
find -name Kconfig | xargs cat | grep -i -c "to compile this driver"
```

We consider this an appropriate lower bound estimate of the number of Linux kernel device drivers. The searched string would be misleading if it appeared in features other than those provided by device drivers. At the same time, it seems unlikely that such a piece of advice would appear more than once within the help message of a driver configuration. We also note that not all device drivers have such a string in their Kconfig entry help message, which means that there are actually more than 2243 device drivers in the mainline kernel.

We do not presume these drivers were made by just a few developers. Instead, one would guess Linux kernel device drivers have been written by hundreds of developers worldwide. Indeed, as of April 2022, our *get_driver_maintainers.awk* program tallies 1211 distinct device driver maintainers listed within the kernel MAINTAINERS file[2].

If we consider that some drivers support more than one hardware design, then one would doubtlessly need thousands of devices to test all Linux kernel device drivers on actual hardware. It seems far-fetched to assume Continuos Integration (CI) systems have all those devices. Not to say that the semiconductor industry releases new hardware designs frequently. Without hardware, a virtual device model, an emulation system, or alike, we cannot make even basic soundness testing such as probing and binding a driver to the devices it tries to support.

On the other hand, it seems fair to assume device driver authors have access to the parts for which they're developing drivers. After all, why would anyone (or any company) create a driver for a device they do not own or have access to? So, it turns out that if device drivers are ever runtime tested by anyone, these people are most likely their authors. Since we are interested in assessing how device drivers are being tested (*RQ1*), a word from those who presumably runtime tested them should be of great significance.

[2] Maintainer count by the Linux 5.17.8 release

Talking to device driver developers would also help us identify what testing tools are used by the Linux community to test the kernel (*RQ2*) and understand what features the community desires for a testing tool (*RQ3*). Hence, we decided to run a survey with Linux kernel device driver maintainers. The Linux kernel device driver maintainers are listed under the project's base directory in the MAINTAINERS file[3]. Often, developers create device drivers and submit them to the Linux mailing lists. If the driver gets accepted by the community, it is added to the Linux kernel, and a new entry is added to the MAINTAINERS file, setting at least one person as the maintainer for that piece of code. Usually, these individuals are the authors of the submitted drivers. So, by contacting driver maintainers, we believe in having a fair chance of talking to the early testers of Linux kernel device drivers. Also, even though the current maintainers of a device driver might not be their original authors, we may expect them to look after that peace of code by possibly performing some tests when needed.

3.3.1 Survey Design

We designed a survey drawing inspiration from Shuah Khan and Kate Stewart's Linux community research disclosed in August 2021 (see Appendix B). Our Device Driver Testing Survey contained a welcome screen plus four parts, one page each. The welcoming page introduced participants to the survey by clarifying it was part of scientific research, telling them that the participation was voluntary and that responses would be anonymous. A single yes or no mandatory question asked attendees for consent to participate in the survey.

The next survey sections were:

1. PART I - Community Role
2. PART II - Testing habits
3. PART III - Driver testing
4. FINAL PART - Feedback

Part I contains questions about respondents' role in the Linux kernel development. Part II includes questions related to general Linux testing practices. Part III asks about specific driver test tools. Lastly, the final part allows participants to leave a contact email address if they would like to receive a notification with results from our research. The survey questions and results are detailed in Chapter 4. A copy of the survey is included in Appendix C.

We started designing the survey by sketching up what would be a perfect response and verifying that it would help us answer our research questions. After that, we conceived the survey questions to invite those desired replies from the participants. To capture the information we were seeking, we made use of Yes or No questions, multiple-choice checklists, list radios, arrays, and short and long free text fields. When appropriate, we enabled option order randomization to avoid introducing answer bias, as participants would focus their attention on the very first options and not those in the middle.

[3] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/MAINTAINERS

We submitted our survey design through two review phases. Three Linux kernel developers analyzed the Device Driver Testing Survey in the first review phase. We asked all the reviewers to read the survey invitation message and complete the questionnaire. The reviewers suggested a few improvements which were accepted and incorporated into the survey design. In the second review phase, two free software developers checked the Device Driver Testing Survey. Both of them had previous experience in contributing to the Linux kernel. Nevertheless, only one suggestion to the survey design was raised during the second review phase. After that, we considered our survey to be reasonably refined.

Concerning ethical issues, we observe that our survey complies with all three basic ethical principles provided by *The Belmont Report* (1979), a reference for bio-medical and behavioral research involving human subjects. The report contains a distinction between research and practice, a discussion about three basic ethical principles, and remarks about the application of those principles. The ethical principles provided by The Belmont Report are:

1. **Respect for persons**, meaning that researchers should give weight to individuals' opinions and choices. In most cases, this translates into providing subjects with adequate information about the possible benefits and harms of participating in research so they can judge whether to participate.
2. **Beneficence**, which boils down to two general rules: (1) do not harm and (2) maximize possible benefits and minimize possible harms. However, these principles may conflict with each other, leading to reflection on when its justifiable to seek certain benefits despite the risks involved.
3. **Justice**, in the sense that research burdens and benefits should be distributed in an acceptable way. For instance, researchers should not select subjects simply because of their easy availability or disadvantaged position, but because of reasons related to the research context. Conversely, research benefits should be shared with subjects whenever possible.

The first page of our survey clearly states the potential harms of participating in the Device Driver Testing Survey. It indicates the estimated amount of time it may draw from respondents and advise that survey records would not contain personal information from participants unless they intentionally provide so. On the other hand, the disclosure letter mentions the main benefits expected from this research. Moreover, we observed Belmont's application remarks for conducting research involving human subjects. For instance, in the survey invitation email, we informed our research purposes (characterize testing practices) and procedures (questionnaire). Moreover, the welcome page declared it was a voluntary survey and displayed our contact email so developers could ask us any questions about it. So, we consider that developers had adequate information to comprehend how they would participate in our research and assess whether they would like to. Hence, we claim that our research complies with the *respect for persons* ethical principle.

With respect to the *beneficence* ethical principle, we argue that the Device Driver Testing Survey does not harm the Linux kernel community because:

1. It is presented in a concise way so subjects can quickly understand what it is all about.

2. It is a voluntary survey, so encumbered developers may choose not to participate or opt out anytime.
3. It does not require much time to complete.
4. It was delivered to individual driver developers and did not go to any kernel development mailing list, so we did not spam other kernel developers and readers.

Also, by making the survey anonymous, we trimmed the risks of violating the developers' privacy while minimizing any worries they might have about exposing erratic practices or habits.

Finally, our approach is equitable, not burdening experienced developers or project leaders more than new driver maintainers. The results of this research are going to be accessible to the whole Linux community, allowing everyone to benefit from our findings, especially device driver developers and testers. Hence, our survey also complies with the principle of *justice*.

Not only did we observe The Belmont Report's ethical principles, but we also consulted the Linux Research Guidelines page[4] for advice on how the Linux community expects researchers to interact with kernel developers. The community guidelines state that research with developers must be opt-in and done with the explicit agreement of individual developers. In addition, the documentation page says that research should be done with full disclosure to participants and that everyone reaching the community should be working in good faith to make Linux better. We claim that our research fulfills all these requirements. First, the Device Driver Testing Survey is voluntary. Second, participants must explicitly declare consent to participate in the survey by choosing the yes option in the first question. Third, the invitation email and questionnaire welcome screen expressly stated that the survey was part of scientific research. Lastly, we do not want to cause any harm to the community and hope the results of our study may help guide the development of testing tools for device drivers.

We acknowledge FEITELSON (2021) survey about the ethics of experiments on open-source projects and his argument that ethics guidelines based on bio-medical research need adjustments for application in software engineering research. For what applies to this research, we claim that our approach followed FEITELSON (2021) ethical guidelines. Subsection 3.3.2 details how we looked after the practical ethical issues, such as only targeting potentially interested developers.

As a final validation step, we contacted the Linux Foundation Technical Advisory Board to ask their opinion about our approach to the community. We were happy to receive positive feedback from the board and a few suggestions which helped us improve the questionnaire.

The survey was implemented on a LimeSurvey[5] instance hosted by CEPID (Centro de Pesquisa, Inovação e Difusão em Neuromatemática). None of the questions were mandatory except the first one, which asked for participant consent.

[4] https://www.kernel.org/doc/html/latest/process/researcher-guidelines.html

[5] https://github.com/LimeSurvey/LimeSurvey

3.3.2 Survey Recruitment

We sought to hear the experiences of Linux kernel developers in testing device drivers. However, sending the survey to mailing lists could have stolen attention (and time) from developers not involved in driver testing. Instead, we developed an AWK program to gather the email addresses of developers who maintain artifacts under the *drivers* directory. The *get_driver_maintainers.awk* program collected 1253 distinct addresses from everyone listed in Linux's MAINTAINERS file as a maintainer for any object under the *drivers* directory. Then, we sent the survey through email only to those developers. Our approach was similar to that of Shuah Khan and Kate Stewart on their community survey in August 2021, with the difference that we only targeted invitations to a subset of developers listed in Linux's MAINTAINERS file.

Individually contacting device driver maintainers helps restrain invitations to potentially interested developers and reduces the number of unsolicited invitations, as suggested by Feitelson (2021). His research also evidences that most open source developers think there is no ethical concern in approaching developers to ask them about their code or the considerations guiding its writing, provided that they are advised about partaking in research.

To avoid disclosing maintainers' email addresses (even though they're publicly available in the MAINTAINERS file), we set them on the BCC (Blind Carbon Copy) field and set ourselves as recipients for each email. Also, to prevent email service providers from classifying our messages as spam, we prepared several email files, each one with a maximum of twenty addresses on its BCC field. Then, we were careful to send only one email per hour during (Brazilian) daytime on weekdays until we had sent them all. As a result, it took from 2022-05-16 to 2022-05-24 to send all survey invitations. Before sending each message, we also reviewed each address on the BCC field and removed those that belonged to early survey reviewers and those we deemed to be mailing lists. We ditched a total of eight addresses in this manner. At last, we closed the survey on 2022-06-27, roughly a month after we sent the last invitations.

To measure our audience's interest, we monitored our survey's response rate throughout the days we were sending invitations. After each day, we calculated the survey's response rate as the number of complete responses divided by the number of invitations sent minus the number of emails that bounced (did not make it to their recipients). The response rate after each day was 4.2%, 4.6%, 5.1%, 5.3%, 6.1%, 6.2%, and 6.4%. These results were not far below the typical 5% response rate observed by Forrest Shull (2008). Indeed, we consider that the increase in the response rate throughout the days indicates that Linux maintainers were interested in the software test subject.

In total, we sent 1245 messages inviting Linux kernel device driver maintainers to participate in our survey. Seventy-two of those messages bounced (they did not reach their recipients). We had 295 (23.69%) replies, from which 210 were partial responses and 85 were complete submissions. Hence, the survey's final response rate was 7.25%.

Chapter 4

Linux Kernel Test Tools

In this chapter, we speak about some of the test tools for Linux kernel device drivers. We divided the sections according to the search method used to find each tool. Thus, we distinguish test tools developed as part of academic works from tools developed in other contexts.

4.1 Academia Tools

As a result of our mapping study (Table 3.3), we selected six tools to assess their use in practice. We consider these tools because they showed promising means of automated testing Linux device drivers that kernel developers could apply in their routine. Now we characterize these tools and their main features.

4.1.1 Tool Characterization

Studies on Linux kernel driver testing proposed different approaches to finding and resolving system bugs. Table 3.2 summarizes the test tools and techniques reported by the papers assessed. From this list, we took six tools to undergo a usage evaluation. We now depict the operation of these tools according to what was described in their respective articles.

Renzelmann *et al.* (2012b) focused on testing Linux kernel device drivers using symbolic execution. This technique consists of replacing a program's input with symbolic values. Rather than using the actual data for a given function, symbolic execution comes up with input values throughout the range of possible values to each parameter. **SymDrive** intercepts all calls into and out of a driver with stubs that call a test framework and checkers. Stubs may invoke checkers passing the set of parameters for the function under analysis, the function's return, and a flag indicating whether the checker is running before or after the function under test. Users can access the driver state by calling a supporting library. Thus, checkers can evaluate the behavior of a function under test from the execution conditions and the obtained results.

To automate driver testing with SymDrive, developers may script the creation of symbolic devices by passing additional parameters to the `insmod` command when loading the test framework.

Buchacker and Sieh (2001) developed a framework for testing fault tolerance of GNU/Linux systems by injecting faults in an entirely simulated running system. **FAU machine** runs a User Mode Linux (UML) port of the Linux kernel, which maps every UML process onto a single process in the host system. Thus, a complete virtualized machine runs on top of a real-world Linux machine as a single process. For injecting faults into the virtualized system, the framework launches a second process in the host system. Every time a UML process makes a system call, return from a system call, or receives a signal, it is stopped by the auxiliary host process. The host process then decides whether the halted process will continue with or without the signal received, if errors should be returned from system calls instead of the actual values, and so on. This technique of virtualization combined with the interception of processes has the benefits of maintaining binary compatibility of programs, allowing fault injection in core kernel functionalities, peripheral faults, external faults, real-time clock faults, and interrupt/exception faults.

To run tests with an FAU machine, one must:

1. Prepare the configuration files for virtual machine setup.
2. Run the test system, which will inject faults in the VM at runtime.
3. Evaluate the results collected from the VM kernel log, log files, and user-mode application logs.

Step (1) may be automated by scripting VM setup and generating fault descriptions from a model configuration file for a series of experiments.

Cong *et al.* (2015) introduced a tool that generates fault scenarios for testing device drivers based on previously collected runtime traces. **ADFI** (Automatic Driver Fault Injection) hooks internal kernel API so that function calls and return values are intercepted and recorded in trace files. A fault scenario generator takes trace files as input and iteratively produces fault scenarios where an intercepted return to a driver is replaced by a fault. Each fault scenario is then run, and the resulting stack traces are collected to feed further iterations of the fault scenario generator. ADFI employs this test method to assess driver error handling code paths that, otherwise, would rarely be followed.

According to Cong *et al.* (2015), the efforts to run ADFI include:

1. Preparing a configuration file for driver testing.
2. Crash analysis.
3. (Optionally) compilation flag modification to support test coverage.

ADFI automatically runs each generated fault scenario, one after another, so test execution is automated.

Bai *et al.* (2016) focused on device driver testing through a similar approach. They developed a kernel module to monitor and record driver runtime information. Further, a pattern-based fault extractor takes runtime data plus driver source code and kernel interface functions as input and extracts target functions from them. **EH-Test** considers target functions taking into account driver-specific knowledge such as function return types and whether values returned by functions are checked inside some "if" statement. Since the C programming language has no built-in error handling mechanism (such as "try-catch"), developers often use an "if" statement to decide whether error handling code should be triggered in a device driver. Then, a fault injector module generates test cases in which target function returns are replaced by faulty values. Finally, a probe inserter generates a separate loadable driver for each test case. These loadable driver modules have target function calls replaced by an error function in their code.

Most EH-Test workflow is automated, from target function extraction to fault injection and test-case execution. The manual work consists of writing pair checkers for resource-acquiring and resource-release functions and rebooting the system when crash bugs are detected.

B. Chen *et al.* (2020) presented a test approach based on hybrid symbolic-concrete (concolic) execution. Their work focus on testing LKM (Linux Kernel Modules) using two main techniques: (1) automated test case generation from LKM interfaces with concolic execution; (2) automated test case replay that repeatedly reproduces detected bugs.

During test case generation, the **COD** Agent component sequentially executes commands from an initial test case to trigger functionalities of target LKMs through the base kernel. Two custom kernel modules intercept interactions between base Linux kernel and LKMs under test and add new tainted values to a taint analysis engine. When all commands in the test harness are finished, COD captures the runtime execution trace into a file and sends it to a symbolic engine. A trace replayer performs symbolic analysis over the captured trace file, then sends a set of generated test cases back to the execution environment. These steps then repeat to produce more test cases until some criteria (such as elapsed time) are met.

In test case replay mode, COD Test Case Replayer picks a test case and executes the commands in the test harness to trigger functionalities of target LKMs. Three custom kernel modules intercept the interactions between kernel and LKMs under test, modify these interactions when needed, and capture kernel API usage information. After all commands in the test harness are finished, COD retrieves the kernel API usage information from the custom kernel modules and checks for potential bugs. This process repeats for each test case given as input.

Although COD provides a highly automated workflow that automatically generates and reproduces test cases, manual user effort is still needed. Kernel API changes in new Linux versions may require adjusting the Kprobes defined in COD. Also, users need to double-check reported bugs because COD can issue false positives.

Rothberg *et al.* (2016) developed a tool to generate representative kernel compilation configurations for testing. **Troll** parses files locally for configuration options (#ifdef) and creates a partial kernel compilation configuration. This initial step is called sampling. Each partial configuration is then abstracted by a node in a configuration compatibility graph (CCG). In this graph, mutually compatible configurations are linked by an edge. In the next step (merging), Troll looks up the CCG for the largest click (set of nodes that are all linked together) and merges all those partial configurations that belong to the click.

The compilation configuration obtained with the largest click covers most of the #ifdef and generates several warnings when Sparse analyzes the code generated by such an arrangement. Given a valid kernel configuration file providing good coverage of different configurations (#ifdef), then automated tests are more likely to find bugs.

These were the six test tools selected for usage assessment: SymDrive, FAU machine, ADFI, EH-Test, COD, and Troll. Let's now report how we evaluated them.

4.1.2 Tool Usage Assessment

At the beginning of 2021, we carried out an evaluation process to assess the usage of each selected test tool. This review process consisted of looking for each project's repositories, reading their documentation, installing each project's dependencies, compiling their source code, and contacting their respective authors by email when facing setbacks. We used a QEMU virtual machine with Ubuntu 18.04LTS as the evaluation environment. Our goal was to reproduce the tests described in the articles and expand our knowledge about each tool. With practical testing experience with these testing tools, we would better ponder recommending them to fellow kernel developers or not.

SymDrive stood out among related works as a testing tool for drivers in the kernel through symbolic code execution. To set up SymDrive, we followed the installation steps listed on their developer's page (Renzelmann *et al.*, 2012a). One of the first steps of the setup consists of compiling and installing S2E, a software platform that provides functionalities for symbolic execution on virtual machines. The S2E documentation mentions the use of Ubuntu as a prerequisite for setting up the platform (Cyberhaven, 2020b; Herrera, 2020; Cyberhaven, 2020a), even though we have found indications of compatibility with other operating systems after inspecting the compilation and installation scripts. Moreover, installation script error messages notifying us that S2E is compatible only with a restricted set of processors. However, even though we had configured the VM with a compatible processor, the installation scripts kept failing due to insufficient system memory (despite our 10GB system RAM).

We discovered that the S2E mailing list was semi-open, meaning that only subscribed addresses may send emails to it. To subscribe to the S2E list, one must send a subscription request to be assessed by a moderator. However, it took a month for an S2E moderator to accept our subscription request to their mailing list. By the time they granted access to us, we were assessing other testing tools and did not want to come back to this one. Finally, our email to the authors of the SymDriver paper was unanswered. So, after a series of setbacks related to installation and lack of access to support, we gave up on installing S2E and evaluating the use of SymDrive.

FAU machine offers a broad test platform, allowing injection of many types of faults at diverse points of a GNU/Linux system as a whole. To test with an FAU machine, we wrote to the respective paper authors, who then pointed out where to find its source code. Next, we downloaded the associated repositories and installed the packages needed for build and installation. The project documentation is outdated and is not maintained by the developers. For instance, two packages indicated in the documentation as necessary for the build are deprecated and no longer needed. In reply to one of our messages, the project maintainer said that questions could be answered by email: *"Just forget *any* documentation you find regarding FAUmachine. None is correct any more. Sorry for that. We just don't have time to update these documents. I think you must ask your questions using e-mail"*.

After compiling and installing an FAU machine, we tried to run some tests by setting up an example from FAU source files. The experiment consisted of starting a virtual machine and installing a Debian image on its disk. However, the experiment run script failed during image installation. Our following email to the maintainer asking for help with the experiment went unanswered. Still, within the menus and options in the virtual machine management window, it was possible to see items referring to system fault injections. The evaluation of these tests, however, could not be completed.

ADFI and EH-Test proposed fault injection tests focused on device drivers. However, Cong *et al.* (2015) article has no link or web page address for the ADFI project repository. Moreover, ADFI authors did not respond to our email asking how to get ADFI. Thus, it was not possible to evaluate ADFI as we could not even get the tool. As for EH-Test, we downloaded the tool's source code and, with some adjustments, we managed to build some of the test modules. However, some EH-Test components do not build with current GCC and LLVM versions. We mailed Bai *et al.* (2016) asking for some installation and usage guidance, but we had no feedback.

For reasons analogous to ADFI, COD could not be tested either. There is no repository link or instruction on getting COD in B. Chen *et al.*, 2020. We sent an email to the paper authors, but that was unanswered.

Lastly, since Troll was designed to generate Linux kernel build configurations, it does not fit into the kernel tests category. Despite that, we decided to give Troll a try. Nevertheless, on our first shot, we found that some new Kconfig features were not supported by Undertaker, a software whose output was needed to feed Troll. Also, the Undertaker mailing list was semi-open. Since our adjustments to the kernel symbols were insufficient to make Undertaker generate partial kernel configurations, our last resort was to reach Troll's developers. Surprisingly, the authors were very responsive and helped us to set up the latest Undertaker version. After that, we ran an example from Troll documentation that generates Linux kernel compilation settings. The uses of Troll presented in the reference article are analogous to the documentation example we ran, except that they require more than 10GB of system memory to complete. Even though we have not been able to recreate the configuration files for the scenarios explored in Rothberg *et al.* (2016) article, we believe Troll would have generated them if we had provided it with enough computational resources.

From what we have observed, academic publications introduce test tools as promising solutions to leverage the practice of automated tests in the Linux kernel. However, our experience evaluating them suggests that most of those tools are outdated, incompatible with newer software versions, or have limited support from their original developers. This outcome (summarized in Table 4.1) led us to seek Linux kernel test tools in additional sources.

Identifier and Reference	*Tool Name*	*Able to reproduce tests?*	*Comments*
L1 B. Chen *et al.* (2020)	COD	No	No repository link. No response from maintainers.
L2 Cong *et al.* (2015)	ADFI	No	No repository link. No response from maintainers.
L6 Buchacker and Sieh (2001)	FAU Machine	No	Unable to run test cases.
L12 Renzelmann *et al.* (2012b)	SymDrive	No	Unable to install. No response from maintainers.
L14 Bai *et al.* (2016)	EH-Test	No	Incompatible with newer compiler versions. No response from maintainers.
L18 Rothberg *et al.* (2016)	Troll	Partially	Required considerable computational resources.

Table 4.1: *Summary of tool usage assessment conclusions.*

4.2 Community Tools

To complement the findings from the formal literature, we have conducted a grey literature review that has revealed an additional set of Linux kernel test tools. As described by Algorithm 3.3, we carried on the GLR in two iterations. Each iteration brought new evidence of testing tools and practices for assessing the Linux kernel functionality. We now present statistics from our GLR, followed by a characterization of the most cited tools and our considerations about them.

4.2.1 Summary of Grey Literature Review Outcomes

The first grey literature document collection stage took place on April 27, 2021, and brought 107 unique publications. We then screened those publications in a preliminary analysis phase, which resulted in the selection of 47 documents for the next stage. In a complete document analysis step, we found 23 publications with information about Linux kernel tests. These publications reported practices, statistics, opinions, or studies about Linux kernel testing and its community. Throughout the selection process, we collected 21 different citation links for snowballing, which went through the same selection phases and criteria, resulting in 8 additional documents identified with relevant information about Linux testing. Thus, the first iteration of the grey literature review provided a total of 31 (23 + 8) documents containing practices, statistics, opinions, or studies about the tools used to test the Linux kernel.

The second grey literature document collection phase took place on October 11, 2021, bringing 94 new publications to the preliminary analysis step. The preliminary document analysis phase then filtered those publications out to only 27 documents. Next, the complete analysis step selected 15 out of those 27 productions and brought 12 documents for snowballing. We then selected nine additional publications from the snowballing references through our review process. At that point, we relaxed our structured review process to enable another snowballing phase. Throughout the review process of the twelve publications selected for snowballing, we collected 16 additional publications for a second level of snowballing. Then, after that second level of snowballing phase, we accepted another ten publications to embody our publication list. Finally, the second iteration of the grey literature review brought a total of 34 (15 + 9 + 10) documents containing practices, statistics, opinions, or studies reporting tools used to test the Linux kernel.

To justify holding a second level of snowballing, we convey that, while selecting publications from the first snowballing list, we noticed that some documents provided citations to pages that, by the context, would contain relevant content about Linux testing. We estimated to miss important information if we followed the planned review protocol to the letter. Thus, we agreed to adapt the review process to evaluate some (16) additional documents. We believe our decision was proper since we selected 10 of those 16 publications.

There was a slight decrease in the number of unique publications found in each GLR iteration. While the first collection phase accumulated 107 documents, the second collection phase brought in only 94 additional publications. We point out that the difference between the number of publications collected in the first and second iteration of GLR is due to the depletion of some data sources. For example, Kernelnewbies returned only nine results for the "linux kernel test" search (we collected five in the first data collection phase and another four in the second). Linux Journal returned only three results for the "linux kernel verify" search (we collected three in the first phase of data collection and none in the second). Also, as in the first data collection phase, some data sources returned identical pages from searches with different strings. Some of these pages were collected but then removed during a duplicate URL removal step. The outcomes from the first and second GLR iterations are summarized in Table 4.2 and Table 4.3, respectively.

Selection phase	*# input documents*	*# output documents*
1) Collection	-	107
2) Preliminary analysis	107	47
3) Complete analysis	47	**23**
4) Snowballing Preliminary analysis	21	11
5) Snowballing Complete analysis	11	**8**
Selected documents		**31**

Table 4.2: *Summary of the first GLR iteration.*

Selection phase	*# input documents*	*# output documents*
1) Collection	-	94
2) Preliminary analysis	94	27
3) Complete analysis	27	**15**
4) Snowballing Preliminary analysis	12	10
5) Snowballing Complete analysis	10	**9**
6) 2nd Level Snowballing Preliminary analysis	16	10
7) 2nd Level Snowballing Complete analysis	10	**10**
Selected documents		**34**

Table 4.3: *Summary of the second GLR iteration.*

The complete list of documents assessed throughout our GLR is available at https://gitlab.com/Marcelosc/ime-usp-masters-dissertation/-/blob/dissertation-final/literature-review/LSMS_and_GLR.ods. In that list, we brought together the data source, identifier (ID), URL, title, year of publication, the reason for exclusion (when appropriate), the reason for inclusion (when appropriate), information pertinent to Linux tests, and additional notes for each publication appraised by our research. The list of selected publications is shown in Table 4.4. The *Data Source* column indicates the search engine from which we retrieved the documents or the publication that cited them when we were snowballing.

ID	*Data Source*	*Title*	*Year*
G1	Kernelnewbies	Linux_Kernel_Tester's_Guide_Appendix_A	2021
G2	Kernelnewbies	Linux_Kernel_Tester's_Guide_Chapter1	2021
G3	Kernelnewbies	Linux_Kernel_Tester's_Guide_Chapter2	2021
G4	Kernelnewbies	Linux_Kernel_Tester's_Guide_Chapter3	2021
G5	Linux Documentation	ABI testing symbols	2021
G6	Linux Documentation	Linux Kernel Selftests	2021
G7	Linux Documentation	How the development process works	2021
G8	Linux Documentation	A Tour Through RCU's Requirements	2021
G9	Linux Documentation	xpad - Linux USB driver for Xbox compatible controllers	2021
G10	Linux Documentation	Linux Input Subsystem userspace API » 1. Introduction	2021
G11	Linux Journal	Linux Kernel Testing and Debugging	2014
G12	Linux Journal	Unit Testing in the Linux Kernel	2019
G13	Linux.com	Kernel Developers Summarize Linux Storage Filesystem and Memory Management Summit	2015
G14	Linux.com	Status of Embedded Linux: Tim Bird Warns of Slow Progress on Linux Shrinkage	2016
G15	LWN	Free user space for non-graphics drivers	2020
G16	LWN	Maintaining stable stability	2020
G17	LWN	A realtime developer's checklist	2020
G18	LWN	Linux 5.12's very bad, double ungood day	2021
G19	LWN	Patching until the COWs come home (part 2)	2021
G20	LWN	Some 5.12 development statistics	2021
G21	LWN	An update on the UMN affair	2021
G22	Linux Foundation	Fuzzing Linux Kernel	2021
G23	Linux Foundation	Kernel Validation With Kselftest	2021
G24	G14	Fuego	2018
G25	G11	LTP HowTo	2012
G26	G11	Smatch The Source Matcher	2021
G27	Linux.com	So, you are a Linux kernel programmer and you want to do some automated testing...	2021
G28	G11	Ktest	2017
G29	G16	syzbot	2021

continue ⟶

Table 4.4: *Documents selected from the grey literature review.*

ID	*Data Source*	*Title*	*Year*
G30	G6	Kernel self-test	2019
G31	LWN	Distributed Linux Testing Platform KernelCI Secures Funding and Long-Term Sustainability as New Linux Foundation Project	2019
G32	Kernelnewbies	Linux_Kernel_Tester's_Guide_Introduction	2017
G33	Linux Foundation	Linux Kernel Developer: Arnd Bergmann	2017
G34	Linux Foundation	Linux Kernel Developer: Laura Abbott	2017
G35	Linux Foundation	Linux Kernel Developer: Shuah Khan	2017
G36	Linux Foundation	Oracle Q&A: A Refresher on Unbreakable Enterprise Kernel	2018
G37	Linux Foundation	Real-Time Linux Continues Its Way to Mainline Development and Beyond	2018
G38	LWN	The RCU API, 2019 edition	2019
G39	LWN	Scheduler behavioral testing	2019
G40	LWN	Calibrating your fear of big bad optimizing compilers	2019
G41	LWN	Portable and reproducible kernel builds with TuxMake	2021
G42	Linux Documentation	OMAP4 ISS Driver	2012
G43	Linux Documentation	Linux Joystick support - Introduction	2021
G44	Linux.com	How Continuous Integration Can Help You Keep Pace With the Linux Kernel	2016
G45	Linux.com	Fixing the Linux Graphics Kernel for True DisplayPort Compliance, Or: How to Upstream a Patch	2017
G46	Linux.com	How Facebook Uses Linux and Btrfs: An Interview with Chris Mason	2016
G47	Linux Foundation	2020 Linux Kernel History Report	2020
G48	G35	2017 Linux Kernel Development Report	2017
G49	G39	ARM-Software/lisa - README.rst	2021
G50	G39	A survey of scheduler benchmarks	2017
G51	G40	A formal kernel memory-ordering model (part 2)	2017
G52	G41	Linaro/tuxmake - README.md	2021
G53	G44	Welcome to KernelCI	2021
G54	G44	Rapid Operating System Build and Test	2021
G55	Linux.com	Testing Btrfs On The Linux 3.16 Kernel	2014
G56	G47	coccicheck [Wiki]	2018
G57	G47	linux-kernel-bot-tests - start [Wiki]	2016
G58	G47	Linux Kernel Performance	2021
G59	G47	kernelslacker/trinity: Linux system call fuzzer - README	2017
			continue ⟶

Table 4.4: *Documents selected from the grey literature review.*

ID	*Data Source*	*Title*	*Year*
G60	G47	LCA: The Trinity fuzz tester	2013
G61	G47	Statistics from the 5.4 development cycle	2019
G62	G47	Linaro's Linux Kernel Functional Test framework | LKFT	2021
G63	G47	Tests in LKFT	2021
G64	G53	Continuous Kernel Integration (CKI) Project	2021
G65	G44	drm / igt-gpu-tools - README.md	2021

Table 4.4: *Documents selected from the grey literature review.*

After each GLR iteration, we read the pertinent snippets from selected documents once more to digest the information about the test tools used to test the Linux kernel. The main objectives of this content analysis and synthesis stage were to answer the research questions *RQ1* and *RQ3* and develop a list of Linux kernel test tools (*RO1*).

Table A.1 contains the preliminary list of Linux kernel test tools, comprising seventy-two test suites either cited by formal articles or community publications. The most mentioned of those tools was cited by eight documents, whereas many other test tools were mentioned by just one publication each. We also provide a spreadsheet[1] with the name, estimated activity status, testing techniques, repository, and supporting publications for each Linux test tool we identified throughout our literature reviews.

Even though we have estimated the activity status of over those seventy tools, providing a detailed characterization of them all would be a laborious task. Instead, we argue that a promising tool would gather more users over time and, consequently, be more cited by literature. Thus, we decided to investigate the most mentioned tools only. Excluding some tools from a further analysis may limit our conclusions because it may be that some of them are indeed useful for testing Linux in some contexts. We note, however, that our approach was careful to avoid disregarding tools employed in kernel testing. Therefore, some test tools may target kernel areas other than device drivers. Moreover, using some tools for driver testing may be farfetched. Hence, even though we did not analyze every Linux kernel test tool found, the results from this study should still provide a good picture of the tools used to test Linux device drivers.

4.2.2 Tool Characterization

After finishing the GLR, we defined the criteria to select some testing tools for evaluation. First, we established that at least three GL publications should have cited each test tool. The test suites selected by this criterion are kselftest (with seven citations); 0-day test robot (with six citations); KernelCI (five citations); LKFT, Trinity, Syzkaller, LTP (each one with four citations); ktest, Smatch, coccicheck, jstest, TuxMake (each with three citations). Also, we want to be able to test device drivers with the aid of these test tools. So, we defined another criterion to dismiss tools that would not allow us to test drivers. Nevertheless, all

[1] https://gitlab.com/Marcelosc/ime-usp-masters-dissertation/-/blob/dissertation-final/literature-review/LSMS_and_GLR.ods

tools mentioned above might be helpful for testing device drivers, so we kept nine test tools for usage evaluation. Before we present our experience with these tools, let us briefly describe them.

Kselftest

Kernel selftests (kselftest) is a unit and regression test suite distributed with the Linux kernel tree under the *tools/testing/selftests/* directory (*Linux Kernel Selftests* 2021; Khan, 2021a; *Kernel self-test* 2019). Kselftest contains tests for various kernel features and subsystems such as breakpoints, cpu-hotplug, efivarfs, ipc, kcmp, memory-hotplug, mqueue, net, powerpc, ptrace, rcutorture, timers, and vm sub-systems (Khan, 2014). These tests are intended to exercise individual code paths and terminate in less than 20 minutes (*Linux Kernel Selftests* 2021; *Kernel self-test* 2019). Kselftest consists of shell scripts and user-space programs that test kernel API and features. Test cases may span kernel and use-space programs working in conjunction with a kernel module to test (Khan, 2021a). Even though kselftest's main purpose is to provide kernel developers and end-users a quick method of running tests against the Linux kernel, the test suite is run every day on several Linux kernel integration test rings such as the 0-Day robot and Linaro Test Farm (*Kernel self-test* 2019). It is stated that someday Kselftest will be a comprehensive test suite for the Linux kernel (*Linux Kernel Developer: Shuah Khan* 2017; G. K.-H. Jonathan Corbet, 2017).

0-day test robot

The 0-day test robot is a test framework and infrastructure that runs several tests over the Linux kernel, covering core components such as virtual memory management, I/O subsystem, process scheduler, file system, network, device drivers, and more (*Linux Kernel Performance* 2021). Static analysis tools such as Sparse, Smatch, and Coccicheck are run by 0-day as well (*2020 Linux Kernel History Report* 2020). These tests are provided by Intel as a service that picks up patches from the mailing lists and tests them, often before they are accepted for inclusion (G. K.-H. Jonathan Corbet, 2017). 0-day also tests key developers' trees before patches move forward in the development process. The robot is accounted for finding 223 bugs during a development period of about 14 months from Linux release 4.8 to Linux 4.13 (which came out September 3, 2017). With that, the 0-day robot achieved the rank of top bug reporter for that period (G. K.-H. Jonathan Corbet, 2017). Despite that, analyzing Linux 5.4 development cycle, Corbet, 2019 reported that there had been worries that Intel's 0-day test service was not proving as useful as it once was.

KernelCI

KernelCI is an effort to test upstream Linux kernels in a continuous integration (CI) fashion. The project's main goal is to improve the quality, stability, and long-term maintenance of the Linux kernel. It is a community-led test system that follows an open philosophy to enable the same collaboration to happen with testing as open source does to the code itself (*Distributed Linux Testing Platform KernelCI Secures Funding and Long-Term Sustainability as New Linux Foundation Project* 2019; *Welcome to KernelCI* 2021). KernelCI generates various configurations for different kernel trees, submits boot jobs to several labs worldwide, collects, and stores test results into a database. The test database kept by

KernelCI includes tests run natively by KernelCI, but also Red Hat's CKI, Google's syzbot, and many others (Vizoso, 2016; *Welcome to KernelCI* 2021).

LKFT

Linaro's LKFT (Linux Kernel Functional Testing) is an automated test infrastructure that builds and tests Linux release candidates on the arm and arm64 hardware architectures (*2020 Linux Kernel History Report* 2020). The mission of LKFT is to improve the quality of Linux by performing functional testing on real and emulated hardware targets. Weekly, LKFT runs tests over 350 release-architecture-target combinations on every git-branch push made to the latest 6 Linux long-term-stable releases, linux-next, and the mainline tree. In addition, Linaro claims that their test system can consistently report results from nearly 40 of these test setup combinations in under 48 hours (*Rapid Operating System Build and Test* 2021; *Linaro's Linux Kernel Functional Test framework* 2021). LKFT incorporates and runs tests from several test suites such as LTP, kselftest, libhugetlbfs, perf, v4l2-compliance tests, KVM-unit-tests, SI/O Benchmark Suite, and KUnit (*Tests in LKFT* 2021).

Trinity

Trinity is a random tester (fuzzer) specialized in testing the system call interfaces that the Linux kernel presents to user space (Kerrisk, 2013). Trinity employs some techniques to pass semi-intelligent arguments to the syscalls being called. For instance, it accepts a directory argument from which it will open files and pass the corresponding file descriptors to system calls under test. This feature can be helpful for discovering failures in filesystems. Thus, Trinity can find bugs in parts of the kernel other than the system call interface. Some areas where people used Trinity to find bugs include the networking stack, virtual memory code, and drivers (Jones, 2017; Kerrisk, 2013).

Syzkaller

Syzkaller is said to be a state-of-the-art Linux kernel fuzzer (Konovalov, 2021a). The syzbot system is a robot developed as part of the syzkaller project that continuously fuzzes main Linux kernel branches and automatically reports found bugs to kernel mailing lists. Syzbot can test patches against bug reproducers. A feature that may be useful for testing bug fix patches, debugging, or checking if the bug still happens. While syzbot can test patches that fix bugs, it does not support applying custom patches during fuzzing. It always tests vanilla unmodified git trees. Nonetheless, one can always run syzkaller locally on any kernel to better test a particular subsystem or patch (Vyukov *et al.*, 2021). Syzbot is receiving increasing attention from kernel developers. For instance, Sasha Levin said he hoped that failure reproducers from syzbot fuzz testing could be added as part of testing for the stable tree at some point (Edge, 2020).

LTP

The Linux Test Project (LTP) is a test suite that contains a collection of automated and semi-automated tests to validate the reliability, robustness, and stability of Linux and related features (Khan, 2014; Iyer, 2012). By default, the LTP run script includes tests for filesystems, disk I/O, memory management, inter process communication (IPC), the process scheduler, and the system call interface. Moreover, the test suite can be customized by adding new tests, and the LTP project welcomes contributions (Khan, 2014).

Some Linux testing projects are built on top of LTP or incorporate it somewhat. For example, LTP was chosen as a starting point for Lachesis, whereas the LAVA framework provides commands to run LTP tests from within it (Khan, 2014). Another test suite that runs LTP is LKFT (*2020 Linux Kernel History Report* 2020; *Tests in LKFT* 2021).

ktest

ktest provides an automated test suite that can build, install, and boot test Linux on a target machine. It can also run post-boot scripts on the target system to perform further testing (Khan, 2014; Jordan, 2021). ktest has been included in the Linux kernel repository under the directory *tools/testing/ktest*. The tool consists of a perl script (ktest.pl) and a set of configuration files containing test setup properties. In addition to the build and boot tests, ktest also supports git bisect, config bisect, randconfig, and patch check as additional types of tests. If a cross-compiler is installed, ktest can also run cross-compile tests (*Ktest* 2017; Khan, 2014).

Smatch

Smatch (the source matcher) is a static analyzer developed to detect programming logic errors. For instance, Smatch can detect errors such as attempts to unlock an already unlocked spinlock. It is written in C and uses Sparse as its C parser. Also, Smatch is run on Linux kernel trees by autotest bots such as 0-day and Hulk robot (Khan, 2014; *Smatch The Source Matcher* 2021; *2020 Linux Kernel History Report* 2020).

Coccinelle / coccicheck

Coccinelle is a static analyzer engine that provides a language for specifying matches and transformations in C code. Coccinelle is used to aid the collateral evolution of source code and to help catch specific bugs that have been expressed semantically. Collateral evolution is needed when client code has to be updated due to development in API code. Renaming a function, adding function parameters, and reorganizing data structures are examples of changes that may lead to collateral evolution. For instance, coccicheck, a collection of semantic patches that uses the Coccinelle engine, aids developers in chasing and fixing bugs. coccicheck is available in the Linux kernel under a make target with the same name (*Coccinelle: A Program Matching and Transformation Tool for Systems Code* 2022; N. P. Luis R. Rodriguez, 2016; V. R. Luis R. Rodriguez T. B., 2016). Moreover, coccicheck is run on Linux kernel trees by automated test robots such as 0-day and Hulk robots (*2020 Linux Kernel History Report* 2020; N. P. Luis R. Rodriguez, 2016).

jstest

jstest is a user space utility program that displays joystick information such as device status and incoming events. One can use it to test the Linux joystick API's features and a joystick driver's functionality (*xpad - Linux USB driver for Xbox compatible controllers* 2021; *Linux Joystick support - Introduction* 2021; Kitt, 2009).

TuxMake

TuxMake, by Linaro, is a command line tool and Python library designed to make building the Linux kernel easier. It seeks to simplify Linux kernel building by providing a consistent command line interface to build the kernel across various architectures, toolchains, kernel configurations, and make targets. By removing the friction of dealing with different build requirements, TuxMake assists developers, especially newcomers, to build the kernel for uncommon toolchain/architecture combinations. Moreover, TuxMake comes with a set of curated portable build environments distributed as container images. These versioned and hermetic filesystem images make it easier to describe and reproduce builds and build problems. Although it does not support every Linux make target, the TuxMake team plans to add support for additional targets such as kselftest, cpupower, perf, and documentation. TuxMake is part of TuxSuite, which in turn makes part of Linaro's main Linux testing effort (Rue, 2021; Dan Rue, 2021; *Rapid Operating System Build and Test* 2021).

Thus, we have presented the main characteristics of the most cited Linux kernel test tools by community publications. To avoid mixing evidence from each data collection method, we used only references from community publications to support the initial description of the test tools brought by our GLR. Now that we are familiar with the tooling discussed in the grey literature, let us examine how it would be to test device drivers with them.

4.2.3 Tool Usage Assessment

Just as we assessed the use of test tools found during our mapping study, we evaluated the use of tools found during our gray literature review. Our evaluation process was similar to the one followed previously. We searched the project repositories, looked for instructions for installation and use in the available documentation, installed the tools, and, finally, made basic use of each.

We visited the commit history of each project and their corresponding mailing lists between 2022-01-12 and 2022-01-24. Moreover, we estimated each project's activity status as active for tools that received contributions between the beginning of 2021 to March 17, 2022 (roughly a year); inactive for the tools that did not have updates after 2020; and unknown for those we could not find a source code repository. For tools incorporated into larger projects (such as tools maintained within the Linux kernel tree), we consider only the changes made to the subdirectories implementing the test apparatus. Lastly, we labeled tools that provided tests as a service as active, even though we could not inspect their source code repository.

Moreover, as we intend to explore software that developers could integrate into their development workflows, we did not evaluate tools that we could not run locally. Since we did not find the source code for tests run in the 0-day test robot, KernelCI, and LKFT, we did not assess them. Unlike our previous evaluation environment, we ran most tools (kselftest, ktest, smatch, coccinelle, jstest, TuxMake) on a Lenovo ThinkPad T480 with Debian 12/testing. We assessed Trinity, Syzkaller, and LTP usage in QEMU virtual machines with Debian 11/stable each. We now report our experience with each evaluated tool.

Kselftest

We had a smooth experience while using kselftest. kselfttest is available within the Linux kernel repository under the *tools/testing/selftests/* directory. There are recent patches and ongoing discussions on the project mailing list[2] and recent commits in the subsystem's tree. The documentation presents all the instructions necessary to compile and run the tests. There are also sections exemplifying how to run only subsets of the tests. Some kselftest tests require additional libraries, listed with the *kselftest_deps.sh* script. Unfortunately, by the date we evaluated kselftest, its documentation did not mention the build dependencies script. Nevertheless, the documentation had enough information for us to run some tests without any problem.

Trinity

Trinity is accessible through a repository on GitHub[3]. The latest commit to that repository was from about one month and a half behind the date we inspected it. From the recent commit history, we estimate that the project change rate is roughly one commit per month and that three core developers have maintained the tool. Trinity documentation is scarce and has not been updated for four years. Although some usage examples exist, the documentation does not contain a tool installation guide.

In our experience with Trinity, we let the fuzzer run for a few minutes. It looks like Trinity is still working the same way Konovalov (2021a) described: *"Trinity is a kernel fuzzer that keeps making system calls in an infinite loop."* There is no precise number of tests to run and no time limit for their completion. After being interrupted, the program shows the number of executed system calls, how many ended successfully, and how many terminated with failures.

Syzkaller

Syzkaller source code is hosted on a GitHub repository[4], which has received several contributions in the weeks that preceded our evaluation window. The project mailing list[5] was busy with several messages during those days. Also, from what we observed in the commit history, five core developers have committed most of the changes to Syzkaller.

The Syzkaller documentation is relatively complete. It contains detailed instructions on installing and using Syzkaller and several troubleshooting sections with tips against possible setup problems. The documentation also includes pages describing how the fuzzer works, how to report bugs found in the Linux kernel, and how to contribute to the tool.

When run, Syzkaller prints execution environment information to the terminal and activates an HTTP server. The server pages display detailed test information such as code

[2] https://lore.kernel.org/linux-kselftest/

[3] https://github.com/kernelslacker/trinity

[4] https://github.com/google/syzkaller

[5] https://groups.google.com/forum/#!forum/syzkaller

coverage, execution logs, the number of syscall sequences executed, the number of crashes, etc.

LTP

The Linux Test Project (LTP) is available at a GitHub repository[6] which many developers have committed to in the weeks preceding our evaluation window for the tool. There were also a few discussions in progress on the project's mailing list[7]. In addition, the LTP documentation contains a tool installation and usage guide, as well as other information we found helpful.

We ran a few LTP syscall tests separately and had a pleasing first impression of the test suite. The completion time of each test was short, and their results (pass or fail) were very clear. It took about 30 minutes to run the entire collection of system call tests. LTP also has a set of device driver tests, but many are outdated and do not work anymore.

ktest

ktest is available from within the Linux kernel repository under the *tools/testing/ktest/* directory. Even though ktest is included in a fast-paced changing project such as Linux, its latest contribution dates from five months before our inspection date. ktest documentation is sparse and has only a description of the configuration options and a brief description of the existing example configuration files. There is no installation guide nor any list of test dependencies. To set up ktest, we followed the guidelines provided by Jordan (2021) and adapted several runtime configurations. We only ran an elementary build and boot test over a couple of patches. Despite our hard time setting up ktest, we think it may help automate many test activities mentioned in the literature, such as patch checking, bisecting, and config bisecting.

Smatch

We got Smatch by cloning its repository at https://repo.or.cz/w/smatch.git. The project's commit history had contributions dated to days close to the time we evaluated the tool, although a single developer had authored most of those changes. The mailing list archives[8] we found have registered no messages for years. Smatch also has a mailing list at vger.kernel.org[9], but we did not find mail archives for those. The Smatch documentation is brief, yet it contains instructions on installing and using the source matcher. Within a few minutes, we set up Smatch and ran some static tests against Linux drivers.

Coccinelle / coccicheck

Coccinelle can be obtained through the package manager of many GNU/Linux distributions, as a compressed tar.gz file from the project's web page, or through a GitHub

[6] https://github.com/linux-test-project/ltp

[7] https://lore.kernel.org/ltp/

[8] https://sourceforge.net/p/smatch/mailman/

[9] http://vger.kernel.org/vger-lists.html#smatch

repository. Coccinelle's repository had commits recently to the day we evaluated the static analyzer, most of which were by a single developer. Also, the project's mailing list[10] had ongoing conversations and patches under review.

The Linux kernel documentation has a page with installation and usage instructions for Coccinelle and coccicheck. Moreover, the kernel has a Makefile target named "coccicheck" for running coccicheck semantic patches. In a few minutes, we installed Coccinelle and ran some checks on Linux drivers.

jstest

jstest is part of the Linux Console Project and can be obtained from Source Forge[11] or through the package manager of some GNU/Linux distributions. However, by the date we evaluated jstest, the project's repository was about a year without updates, and the associated mailing lists[12] were without discussions or patches for even longer. Despite that, the jstest documentation was helpful as it listed dependency packages and the installation steps for the tool. Also, the manual page is brief yet informative and tells what is needed to use the tool. To use jstest, one must start the application with the path to a joystick or gamepad device. jstest then displays the inputs obtained from joysticks and gamepads and thus might be helpful to test the functioning of drivers for these devices in a black-box fashion.

TuxMake

TuxMake is available from its GitLab repository[13] and can also be downloaded as a package for many GNU/Linux distros. The contributions to the project's repository are recent to the date we evaluated the tool. In addition, the TuxMake documentation contains installation instructions and examples of how to use the tool. We found no difficulty in getting and using TuxMake. However, we note that TuxMake focuses on building the kernel and thus only builds the artifacts bound to make targets, not triggering the execution of test cases even when those targets would do so by default.

4.3 Summarization

To some extent, several tools can facilitate device driver testing. It is nearly impossible to analyze them all. Yet, this research covered the twenty Linux kernel testing tools selected by our study for being either focused on driver testing or most cited by online publications. These tools make up a heterogeneous group of test solutions comprising various features and test techniques. From unit testing to end-to-end testing, dynamic or static analysis, many ways of putting Linux to the test have been conceived.

Table 4.5 matches test types and tools according to what we found expressly reported

[10] https://lore.kernel.org/cocci/

[11] https://sourceforge.net/projects/linuxconsole/

[12] https://sourceforge.net/p/linuxconsole/mailman/

[13] https://gitlab.com/Linaro/tuxmake

in the literature. Note that some test types intersect with each other. For instance, unit testing may be considered a sort of regression testing, fuzzing is also an end-to-end test, and fault injection tools often instrument the source code to trigger error paths. Thus, it is more than possible that some test tools are not matched with all types of tests they can provide.

Types of Tests	*Tools*
Unit testing	Kselftest, KUnit
Regression testing	Kselftest, LKFT, 0-Day
Stress testing	Kselftest, LTP
Functional testing	LTP, Kselftest, LKFT, 0-day
Fuzz testing	Trinity, Syzkaller/Syzbot
Reliability testing	LTP
Robustness testing	LTP
Stability testing	LTP
Build testing	ktest, 0-day, TuxMake, KernelCI, LKFT
Static analysis	Smatch, Coccinelle/coccicheck, Sparse
Performance testing	0-day, kselftest
Symbolic execution	SymDrive
Fault injection	FAUMachine, ADFI, EH-Test
Code Instrumentation	ADFI, COD
Concolic Execution	COD
Local analysis and grouping	Troll

Table 4.5: *Test types and test tools.*

To compare the effort required to use each of the evaluated test tools, we established a set of tasks that comprise most of the work needed to set up and run a test tool. Then, we gave tools an effort point for each activity we had to carry on to test with them. The tasks we considered for set up effort are:

- A) install system packages, if it was needed to install packages other than the ones required to build the Linux kernel.
- B) download and build a source code repository, if it was needed to download the source code and build it locally.
- C) create a VM, if the tests were potentially destructive and could cause problems to the running system.
- D) configure a VM, if it was needed to do additional VM configuration such as installing packages, enabling ssh, messing up with grub, etc.
- E) write/edit configuration files, if it was needed to create or modify configuration files for the tests to run.

Moreover, we gave five effort points to test tools we could not set up after trying all applicable tasks above. The setup effort ratings were set to None for tools with no effort points, Low for tools with one or two points, Moderate for tools with three to four points, and High for tools with five points. Now, for comparing with respect to usage effort, we assigned effort points for each of the following requirements:

- R) the tests can run as an usual application program.
- S) the tests have to run inside a VM due to risk of compromising the running system.
- T) the tests require a high amount of CPU or memory to run.

We gave three effort points to test tools from which we couldn't get test results either because we could not run them ourselves or we didn't find how to get their test results. Next, we set the usage effort ratings to None, Low, Moderate, and High for tools with zero, one, two, and three effort points, respectively. Finally, we made a setup versus usage effort chart (Figure 4.1) to provide an overview of how easy (or not) it is to use each test tool.

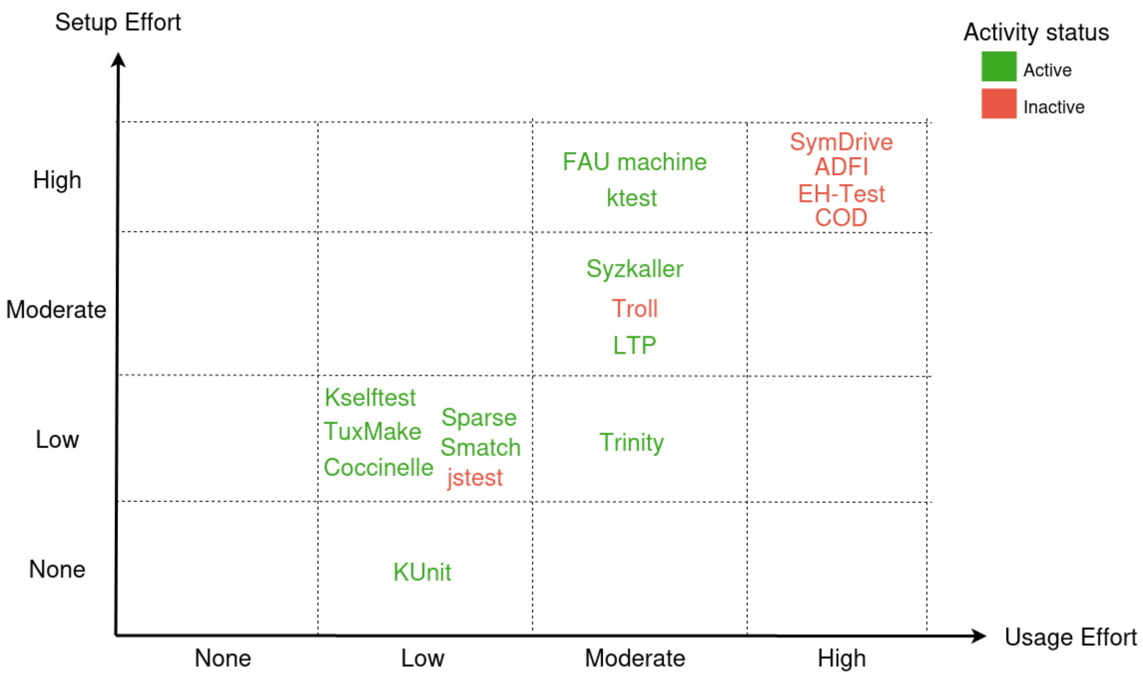


Figure 4.1: *Set up and usage effort of Linux kernel test tools.*

Table 4.6 lists the traits attributed to each test tool. The cases where we were unable to set up or run a tool are indicated with a U. Also, there are two slight adjustments we'd like to note. First, we gave an additional D to ktest because it took us almost three days to set up everything needed to run it. Second, we added two extra points to FAU machine since its documentation was completely outdated and unusable.

Tool	*Setup Effort Points*	*Usage Effort Points*
Kselftest	A	R
Trinity	B, C	R, S
Syzkaller	A, B, E	R, S
LTP	B, C, D	R, S
ktest	A, C, D, E, extra D	R, S
Smatch	A, B	R
Coccinelle	A	R
jstest	A	R
TuxMake	A	R
Sparse	A	R
KUnit	-	R
SymDrive	U	U
FAU machine	A, B, E, +2	R, S
ADFI	U	U
EH-Test	U	U
COD	U	U
Troll	A, B, E	R, T

Table 4.6: *Setup and usage effort points of Linux kernel test tools.*

Chapter 5

Survey Result Analysis

Our Device Driver Testing Survey was active from 2022-05-16 to 2022-06-26 and had 295 replies, from which 210 were partial responses and 85 were complete submissions. On average, these 85 developers took 506 seconds each (nearly eight minutes and a half) to answer the survey questions. Whereas the fastest respondent took 102 seconds, the most attentive participant took 1876 seconds (roughly half an hour) to complete the survey. The median survey completion time was 408 seconds (6.8 minutes). Figures 5.1 and 5.2 show the distribution of survey response times. We consider these response times plausible and thus did not exclude any particular reply from our analysis. Finally, all 85 participants who completed the survey answered yes to question 1 ("Please, confirm that you agree to take part of this survey."), and thus have consented to participate in the survey. We now analyze the results from these 85 device driver maintainers.

5.1 PART I - Community Role

Part I of our driver testing survey was about social aspects related to participation within the Linux community. The results from question 2 ("In which of these roles do you identify yourself within the Linux kernel development community?") are shown in Figure 5.3.

As expected, most (82%) respondents consider themselves maintainers of Linux. This high identification with the maintainer role might be biased because our invitation message pointed out that the recipients were listed in Linux's MAINTAINERS file. Nevertheless, it is interesting to note that some of them (18%) did not identify themselves as maintainers. This result suggests that, to some developers, the Linux kernel maintainership role would be associated with something more than being listed in the MAINTAINERS file. One possible interpretation is that those who did not check the maintainer role might think that maintainers are characterized by additional responsibilities or status, such as keeping development trees or pertaining to Linus Torvalds' trust network.

Also, most respondents consider themselves active contributors (66%) or users (62%) of Linux. Almost half of the respondents said to be reviewers (48%) for the Linux kernel, and a reasonable share of them declared themselves expert contributors (31%). Only a small

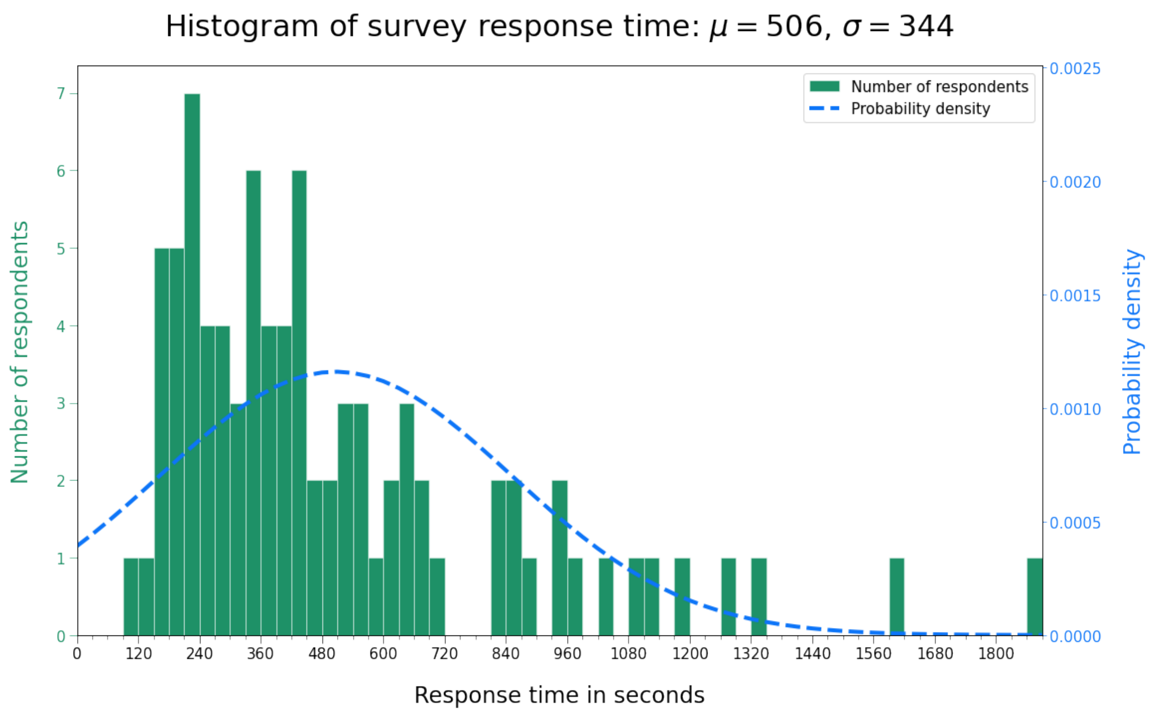


Figure 5.1: *Histogram of survey completion time.*

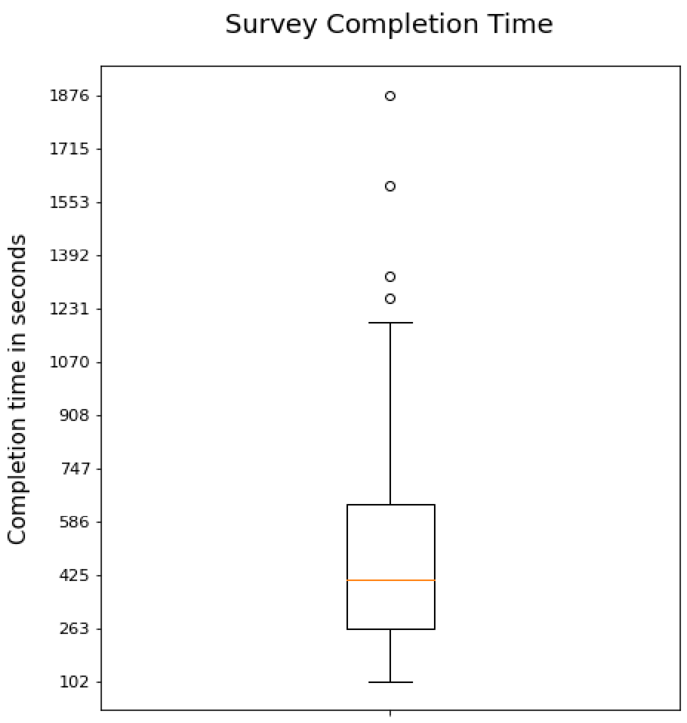


Figure 5.2: *Survey response time box-and-whiskers plot.*

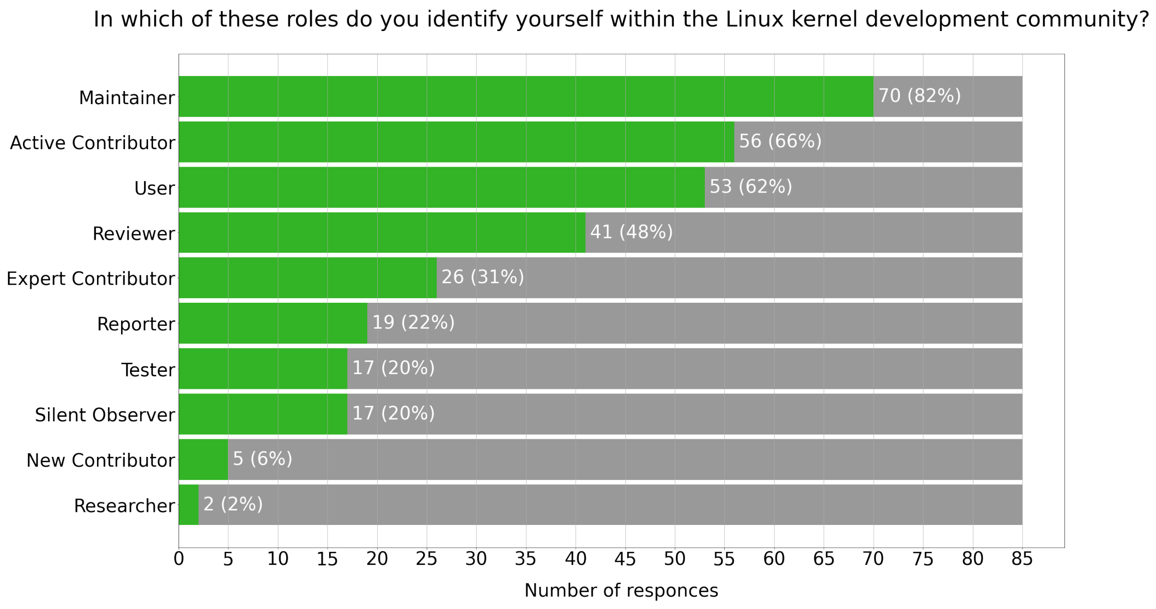


Figure 5.3: *Roles survey attendants play within the Linux kernel community.*

portion of kernel developers have considered themselves software testers (20%). Yet, we argue that this outcome does not compromise the survey results concerning device driver testing. As noted by Mathur (2013), "*it is hard to imagine an individual who assumes the role of a developer but never that of a tester, and vice versa*". In the same way, we consider that it would be hard to develop a device driver without being able to test it during its development process. Moreover, the development community might refuse to include undertested or unsound driver implementations. Thus, we recognize that Linux kernel developers are responsible for some portion of device driver tests. The results shown in Figure 5.3 indicate that the responses to our survey comprise an appropriate source of information to help answer *RQ1* ("How are Linux device drivers being tested?").

With respect to the experience of the survey respondents (question 3: "How many years of Linux kernel development experience do you have?"), most participants had 6 to 10 years (28%) of kernel development experience, and many others had 1 to 5 years (26%) of involvement (see Figure 5.4).

Very few developers have said to have less than one year (2%) experience. This outcome strengthens the validity of the results to be presented since a reasonable share (44%) of the answers came from highly experienced Linux kernel developers with eleven or more years of experience working on the project.

The last question related to social aspects aimed to characterize whether driver developers had financial support to contribute. The results from question 4 ("Do you contribute to the Linux kernel as part of your job (paid work)?") are shown in Figure 5.5.

The majority (63%) of respondents said to contribute to the Linux kernel as part of their jobs. Despite high, this result diverges significantly from the 2017 Linux Kernel Development Report (Corbet and Kroah-Hartman, 2017) that evidenced that over 85% of all kernel development was done by developers who were being paid for their work. However, in contrast to our survey, the 2017 report estimate was based on the use of

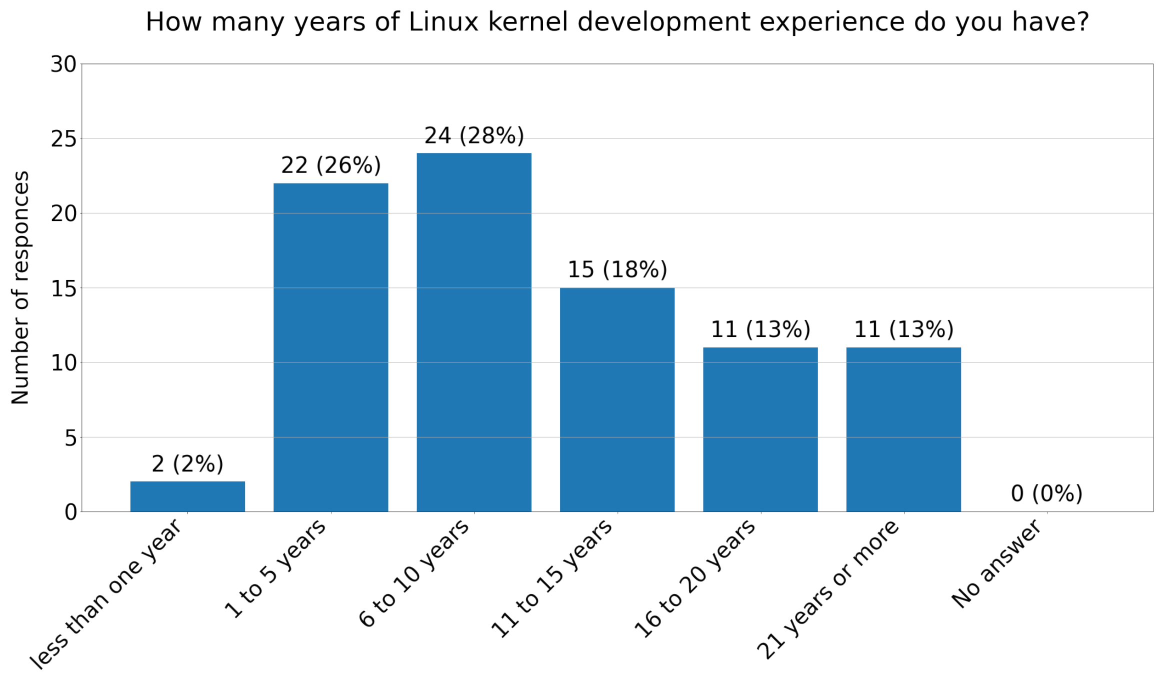


Figure 5.4: *Survey attendants' years of Linux development experience.*

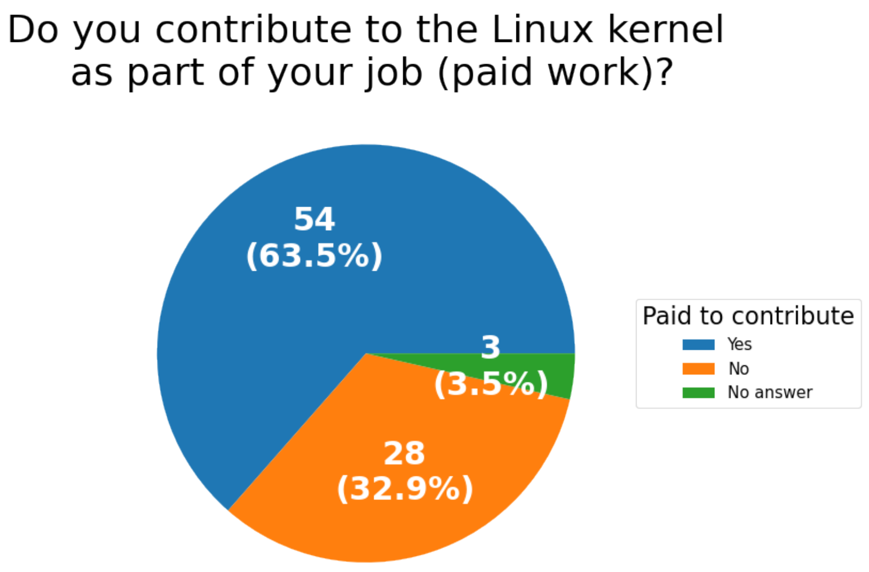


Figure 5.5: *Paid Linux kernel device driver maintainers.*

company email addresses, sponsorship information included in the code submitted to Linux, or information provided by developers themselves. Hence, their results are not directly comparable to ours. Nevertheless, we consider a couple of possible reasons for observing a lower rate of paid developers in our survey. First, some device drivers were developed before 2017, when, according to that year's development report, larger shares of Linux development would come from unpaid developers. Thus, unless those developers have got hired to do kernel development, we would observe higher rates of volunteer contributors among those early Linux developers. Second, developers may get hired to develop and maintain device drivers at an initial moment but then change jobs and continue maintaining drivers as volunteers.

5.2 PART II - Testing habits

The second part of the survey asked participants about general kernel testing habits. We began by checking how often developers test their code before submitting it to mailing lists and what kinds of test tools they use for that testing. Figure 5.6 concisely aggregates the results for question 5 ("How much do you consider yourself involved with kernel testing?").

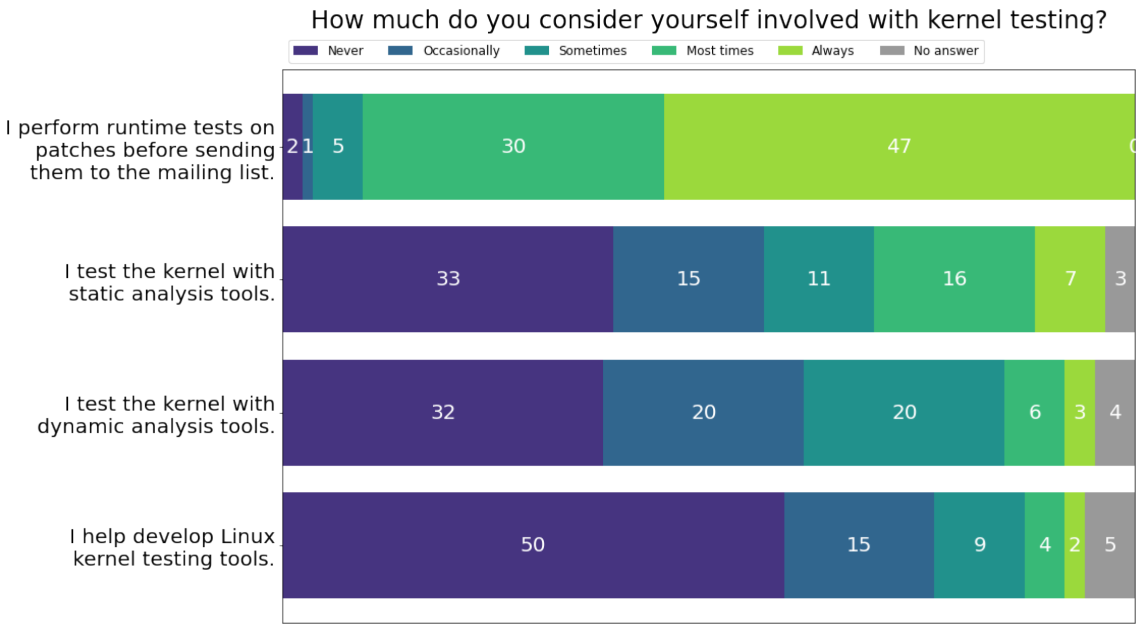


Figure 5.6: *Gerenal kernel testing habits of survey participants.*

If we aggregate the results from the first subquestion, we see that 90% of device driver maintainers say to test their contributions very often (always or, at least, most times). This result indicates that driver maintainers are committed to testing the changes proposed for the Linux kernel.

The kinds of tests performed by these maintainers may vary, though. 30% of driver maintainers who said to always runtime test their patches don't use any static or dynamic analysis tool. Among maintainers who runtime test patches most of the time, 17% don't use any static or dynamic analysis tool. These results indicate that a reasonable number of maintainers test their drivers with other types of tools, with custom scripts, or manually. Nevertheless, 19% of driver maintainers who always runtime test their patches said to use static analysis tools very often (always or, at least, most times), and 15% of them said to use dynamic analysis tools very often (always or, at least, most times).

We see that device driver maintainers run static analysis tools slightly more often than dynamic analysis testing tools. While 27% of driver maintainers said to use static analysis tools frequently (always or most of the time), only 11% of maintainers said to run dynamic analysis tools as often. We believe the contrasting results for usage regularity are due to the different usage requirements of those tools. Usually, static analysis tools are easier to set up and require less time to run than dynamic analysis tools.

As an indication that our results are not biased by possibly having too many responses from sorftware testers, we note that only 18% of maintainers have reported being frequently (always, most times, or sometimes) involved in developing Linux kernel testing tools.

The results from question 6 ("Have you registered any development tree you maintain or use with any kernel testing service?") provide an insight into the number of Linux kernel development trees served by automated testing (see Figure 5.7).

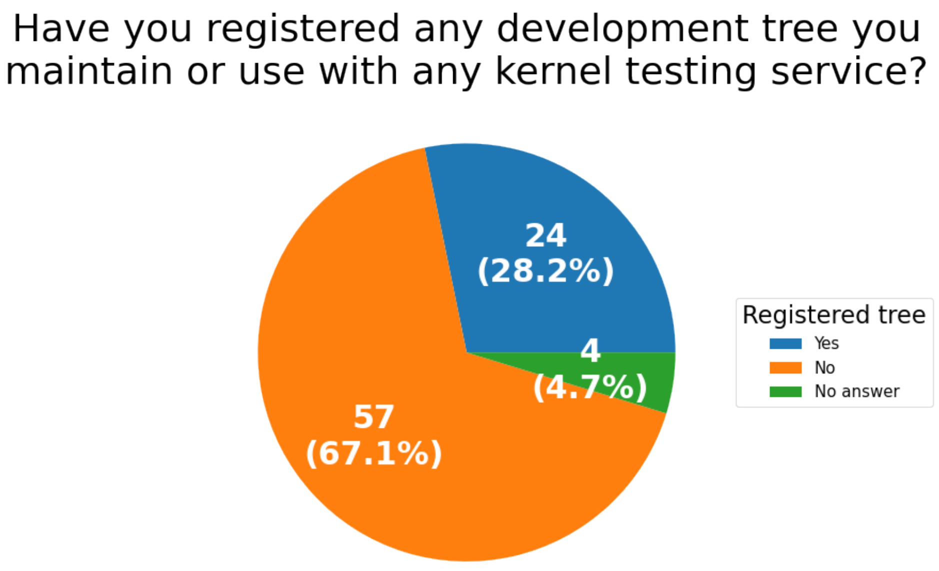


Figure 5.7: *Maintainers with development trees registered for automated testing services.*

We see that 28% of device driver maintainers reported having registered their development trees with Linux kernel testing services. Because not every device driver maintainer maintains a development tree, we consider this result as a lower bound estimative of the portion of trees covered by testing services. If we filter the responses by developers with six or more years of Linux development experience, we see that 34% of maintainers have registered their development trees with testing services. These results indicate that Linux kernel maintainers desire assistance from automated testing services since many had already registered their development trees for testing. Moreover, the desire for automated tests tends to grow among experienced maintainers.

Question 7 ("Does your organization provide any infrastructure to test the patches you submit to the kernel?") results are summarized in Figure 5.8.

On the one hand, 43% of paid device driver maintainers said their employers keep a CI system to test the Linux kernel. This evidence that a reasonable share of employed kernel developers is supported by their organizations for testing contributions to Linux. On the other hand, another 43% of paid maintainers don't receive the same assistance from their

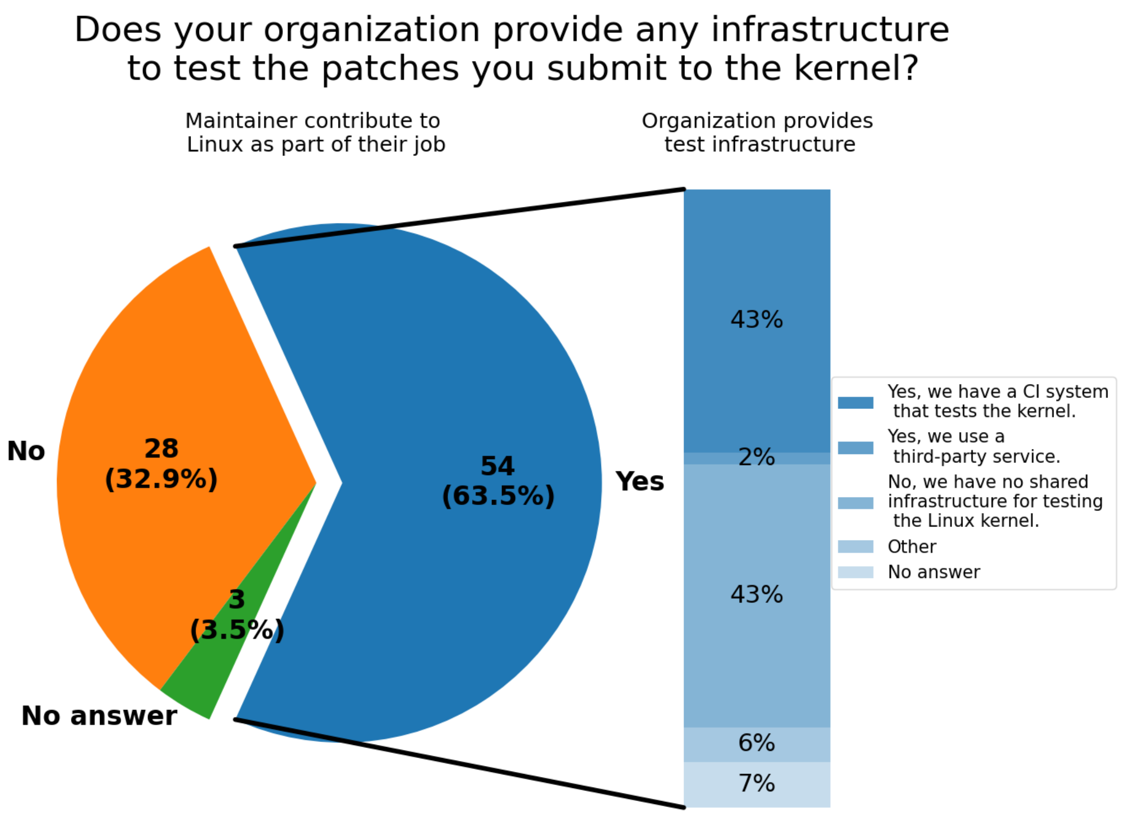


Figure 5.8: *Testing infrastructure provided by employers.*

employers. It turns out that this outcome might indicate a good opportunity for automated testing service sellers to expand their businesses. However, if we aggregate maintainers whose organizations do not provide support for testing the Linux kernel with those who are not paid to contribute, then we see that CI test systems may not serve roughly 60% of driver maintainers.

The last question related to general kernel testing asked participants if they used any particular tool to test the kernel (question 8: “Do you use any tool to test the Linux kernel yourself? If so, which?”). We included this as an open-ended question in Part II because we intended to avoid bias from a suggestive question listing several testing tools in Part III.

From the 39 answers to question 8, we identified a total of 38 different tools (Figure 5.9). The most spontaneously cited tool was sparse (18%), followed by kselftest (13%), smatch (6%), and KASAN (6%). Many of the most cited tools in these answers were also listed in question 11 (G3Q3). This result shows that through our GLR review, we successfully identified many of the test tools used by the Linux kernel development community (*RQ2*).

Despite a large number of tools pointed out by survey participants, some individuals said not to run any test tool or to use custom test scripts: “*No. To determine whether a driver works only runtime testing is used.*”, “*custom scripts to upload system images to devices I use*”, “*shell scripts and simple user api tests*”, “*printk()*”. Also, some developers reported the activity of automated testing tools: “*No, but I’ve seen syzbot & others have tested my patches*”, “*I rely on the Linux test bot and community review for the rest*”.

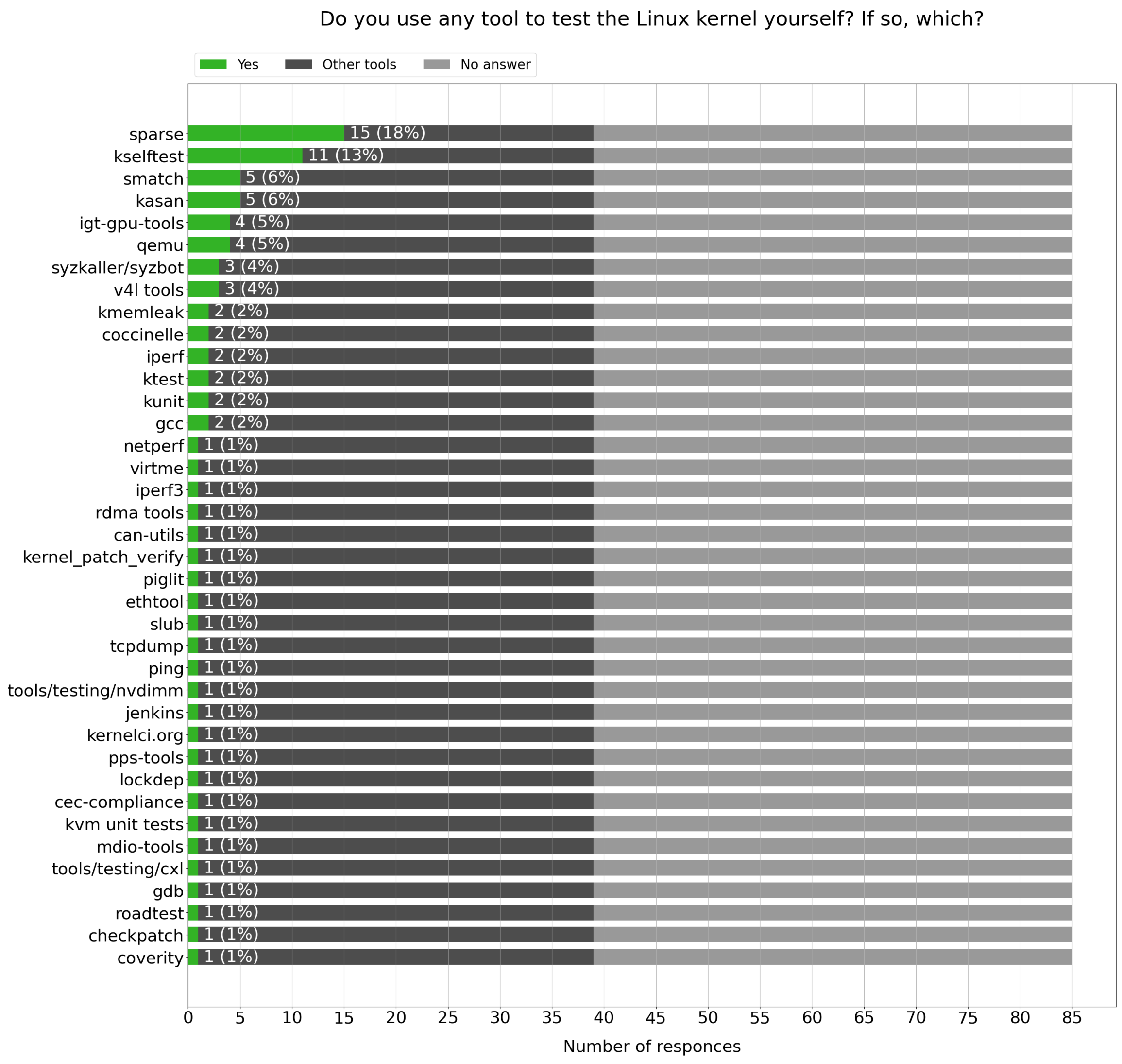


Figure 5.9: *Tools that driver maintainers use to test the Linux kernel.*

5.3 PART III - Driver testing

The third part of the Device Driver Testing Survey contained questions specific to Linux kernel device driver testing. The first question of Part III asked what types of drivers participants have tested (question 9: "What types of device drivers have you ever tested?"). From the 83 responses to this question, we identified 96 distinct types of device drivers. Table 3 shows the top ten driver types reported and the number of maintainers who cited them. Only two (2%) individuals that completed the survey did not answer question 9.

Remarkably, many Linux driver maintainers have tested platform drivers (34%). While this result could have been biased by the question hint, which listed `platform` as an example of driver type, we have little concern about this issue. The Linux kernel declares a specific abstraction (`struct platform_driver`) for describing platform drivers. Thus, maintainers who have worked with platform drivers would know so because they would

What types of device drivers have you ever tested?	
Driver Type	*Count (Percentage)*
platform	29 (34%)
PCI/PCIe	28 (33%)
network	25 (29%)
USB	24 (28%)
I2C	16 (19%)
input	14 (16%)
clock	11 (13%)
GPU	11 (13%)
SPI	11 (13%)
wifi	9 (11%)

Table 5.1: *Top ten driver types reported in response to question 9.*

have used Linux's abstraction for describing such drivers. A similar argument applies to other driver categories in Table 5.1.

In short, a platform device is an autonomous hardware entity on a system. Examples of platform devices are sensors and controllers integrated into system-on-chip platforms. Typically, these devices are not connected through any standardized bus (such as PCI, USB, or SPI) but are directly addressable by the CPU (*Platform Devices and Drivers* 2022). Thus, about a third of driver maintainers have been involved in testing drivers designed for custom hardware platforms, indicating that hardware manufacturers' choices substantially impact Linux developers' work.

The tenth survey question encouraged participants to report activities they carry on during device driver testing (question 10: "Do you carry out any of these activities when testing the drivers you maintain?"). However, listing all tasks one might take to test a device driver is not conceivable because such procedures could be very hardware-specific. In response to us, the Linux Foundation Technical Advisory Board staff pointed out that the issues involved in writing a driver for a PCI device are much different than those of writing a USB driver, and even those are different from writing a driver for a PHY device, and so on. We suppose that the procedures to test such drivers may also differ significantly. Therefore, we designed question 10 to assess how developers even test a driver. Nonetheless, to provide some guidance to the answers, we listed a few generic activities we knew could make part of a testing procedure. Figure 5.10 shows the results for these listed options.

We can see the great majority of device driver maintainers probe and bind the drivers they maintain to a device when performing tests. We consider that good news since it indicates that most drivers pass at least basic soundness tests. Notwithstanding, 16% of maintainers could probably benefit from some assistance. Many hurdles may prevent a maintainer from testing a driver, for instance, not having the required hardware, an unsuitable test infrastructure, or not having the time to do tests.

These issues are, of course, not easy to deal with. A possible solution to the first of these problems could be to develop virtual devices to mimic the behavior of an actual hardware

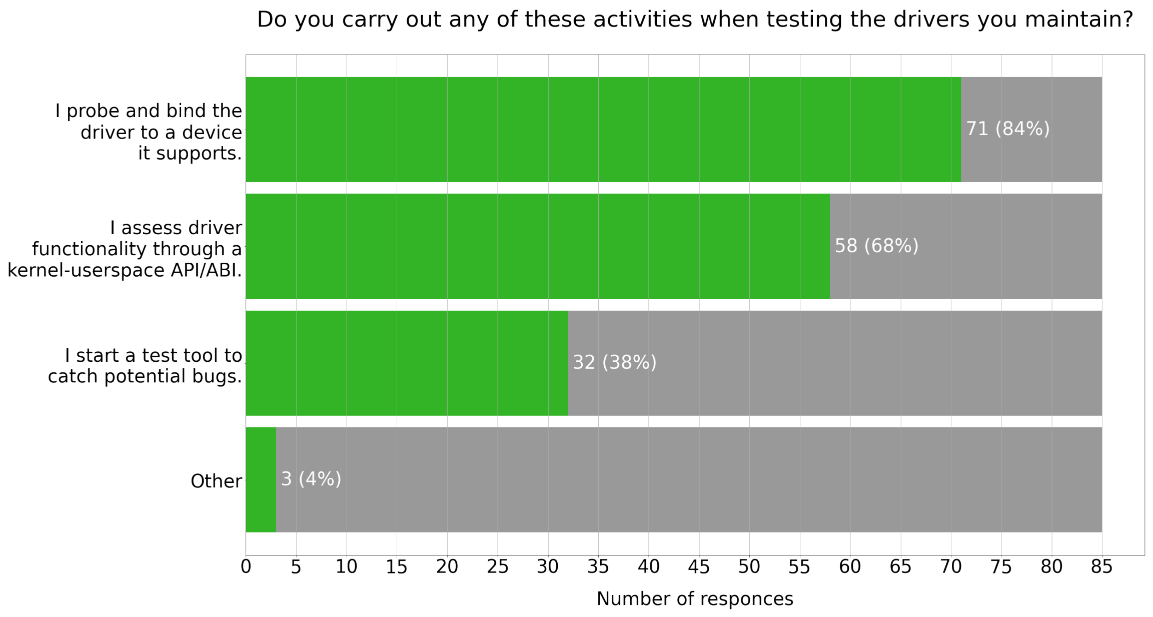


Figure 5.10: *How Linux maintainers test the drivers they maintain.*

counterpart. Despite its limitations, virtual devices have been used in some cases[1] [2] and explored by our research group[3]. The available test tools provide several features to test device drivers, and as they improve over time, testing device drivers should get easier. Lastly, balancing the workload among driver maintainers is an open challenge because there is no standard way to estimate the amount of work it can take to maintain a driver over some time. Moreover, the rapid growth of the Linux kernel source code has evidenced a maintainership scalability problem that does not have any clear solution.

Since developing diverse types of driver poses different issues to developers, we seek to assess whether distinct types of drivers also require particular testing procedures. To examine that, we plotted the answers to question 10 grouped by the driver types reported in answers to question 9 (Figure 5.11). Due to the many driver types reported, we only analyzed grouped responses for the three most cited driver types.

Compared to PCI and network driver maintainers, platform driver maintainers appear to rely more on kernel-userspace ABI/APIs to test their drivers. Also, network driver maintainers are proportionally more into testing tools than developers from the other two groups. These distinctions hold if we examine subquestion comments grouped by driver type. More maintainers who have tested platform drivers have acknowledged the use of kernel-userspace ABIs ("*sysfs attributes must work as intended*", "*I used to do this, but now mostly use in-kernel tests.*"). Likewise, more maintainers that tested network drivers reported running test tools to catch potential bugs ("*Smatch Sparse*", "*make coccicheck or equivalent*").

After analyzing all comments to question 10, we highlight a few additional insights.

[1] https://lore.kernel.org/linux-iio/20210207154623.433442-1-jic23@kernel.org/

[2] https://lore.kernel.org/linux-iio/20210614113507.897732-1-jic23@kernel.org/

[3] https://bcc.ime.usp.br/tccs/2021/lpstankus/monografia.pdf

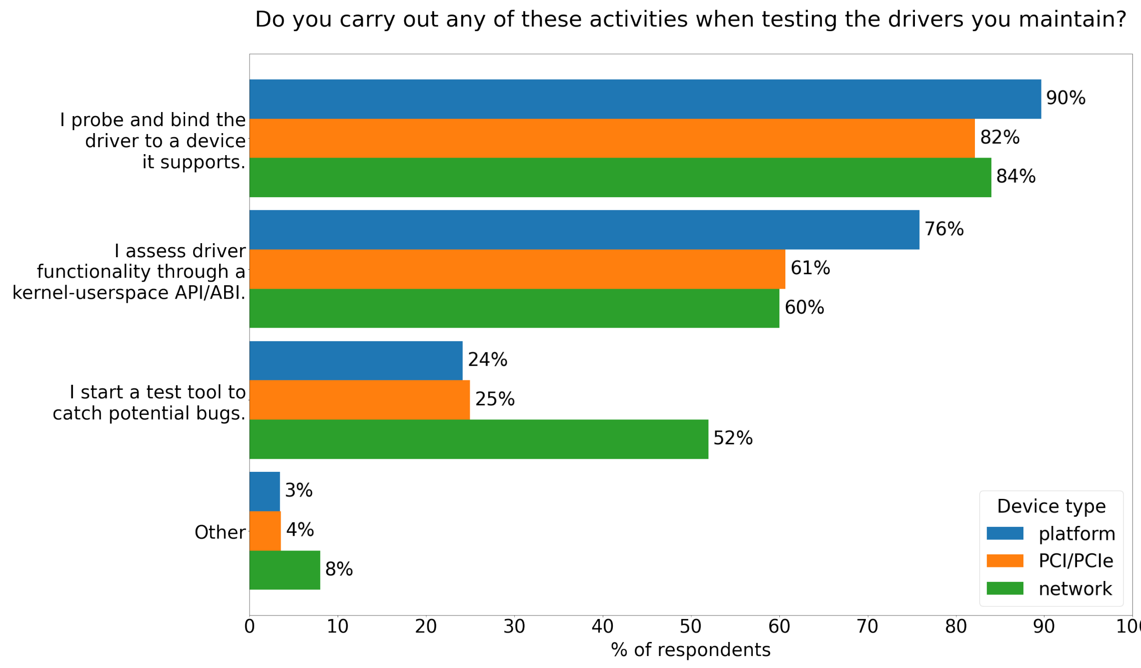


Figure 5.11: *How Linux maintainers have tested platform, PCI, and network drivers.*

First, some maintainers have conditioned the conduct of tests to the possession of hardware. Among the comments to the *I probe and bind the driver to a device it supports* subquestion are: "*runtime testing on real hardware*", "*Only if I have that device*", and "*...or an emulation of the supported device*". Among the comments to the *Other* option is "*Reproduce bugs and verify patches fix them. Always if I have the hw to reproduce*". Thus, even though the Linux kernel is free software, it might be hard to test some parts of it without specific non-free hardware devices. This constraint is especially true for device drivers. Not having access to the hardware supported by a driver may greatly hinder any effort toward testing the driver because one might not even be able to run its initialization procedures. We have observed a few efforts toward minimizing the limitations that missing hardware poses to software testing. Some of these initiatives are SymDrive, the KUnit mocking framework, and the use of QEMU virtual devices. In particular, we believe that using QEMU emulated devices for driver testing might become a tendency among developers since QEMU was the fifth most spontaneously cited tool in question 8.

Second, similar to the reports to question 8, we had many testing tools being cited in comments to the *I start a test tool to catch potential bugs* subquestion: "*kernel debug options like lockdep and kasan check for locking and memory errors*", "*KUnit*", "*v4l2-compliance/cec-compliance*", "*Smatch Sparse*", "*those found in tools/ in the kernel tree*", "*make coccicheck or equivalent*". Despite that, there is surely a portion of developers running their own scripts since one of the respondents said to run "*fio, custom scripts*".

With question 11 ("How familiar are you with these tools/testing infrastructure?"), we can estimate what test tools the Linux community uses to test the kernel (*RQ2*) and how frequently they run tests. Because several test tools are available, we limited our investigation of usage frequency to the ones most cited by the literature. Question 11 presented a list of twenty testing tools or testing infrastructures to the respondents, eighteen of them selected for being most cited in academic studies or community publications and reviewed by us in Sections 4.1 and 4.2. Two additional testing tools, KUnit and Sparse, were included on the list by the authors of this research. We added KUnit to the tool list because we had assessed it during the early phases of this research and because it is developed as part of the Linux kernel. We also included Sparse because it was cited by tools appraised in our formal and grey literature review. Moreover, Sparse is the default check tool enabled by passing C=1 or C=2 flags to Linux's top Makefile when compiling the kernel. Figure 5.12 shows the results to question 11.

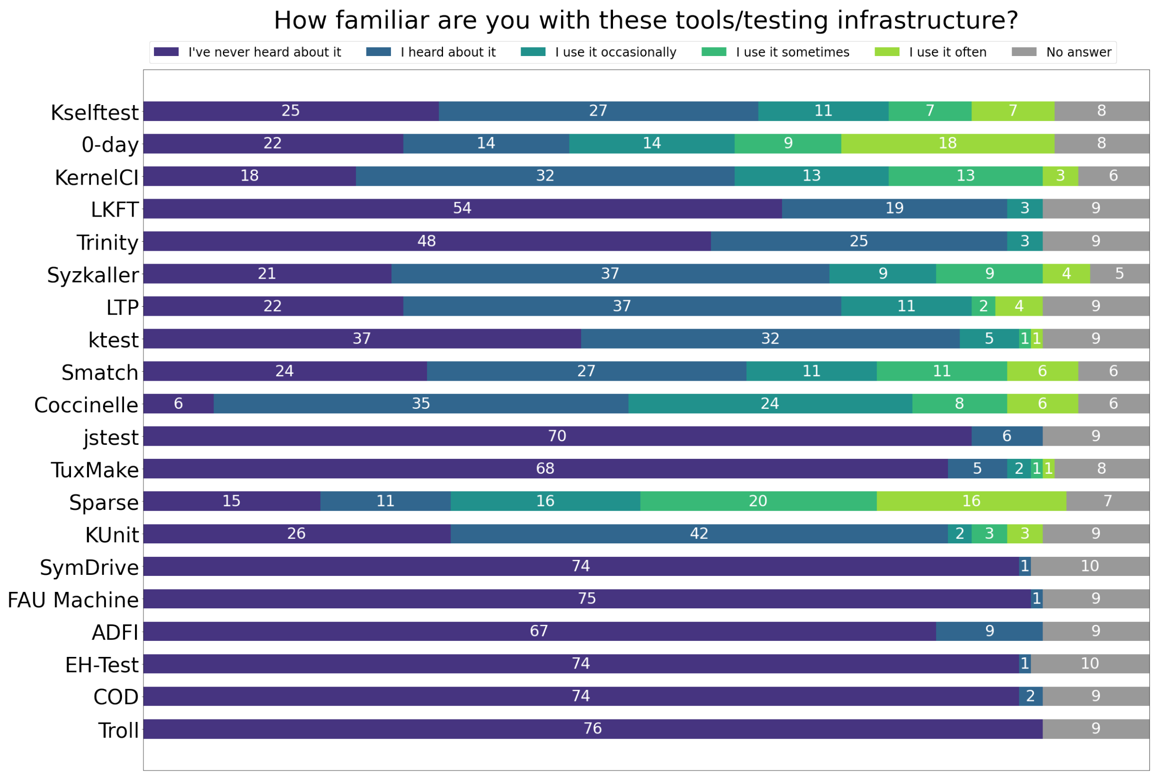


Figure 5.12: *How familiar are Linux driver maintainers with driver testing tools.*

Among the Linux driver maintainers who completed the Device Driver Testing Survey, the 0-day test robot and Sparse are the most frequently used test toolings. Also, Coccinelle is the most known test tool since it got the lowest number of marks for the *I've never heard about it* option. Similar to what was observed from the answers to question 5, question 11 results also reveal a preference for static analysis tools over dynamic analysis. An expressive number of respondents reported using Sparse, Smatch, or Coccinelle frequently (sometimes or always). In contrast, fewer individuals said to run Trinity, Syzkaller, LTP, Ktest, jstest, KUnit, SymDrive, FAU machine, ADFI, EH-test, or COD as often. The exception was Kselftest, which maintainers said to run roughly as frequently as Smatch or Coccinelle.

We believe the reason for preferring static over dynamic analysis test tools is because, in contrast to dynamic testing tools, static analysis tools allow one to inspect the whole source tree or just specific files within it. With that, static testing tools ease the detection and fixing of problems during Linux development. However, developers should be aware that those tools suffer from false positives (*Kernel Testing Guide* 2022).

Another remarkable result is the recognition given to the 0-day test robot. As an automated test service, developers don't use the bot by initiating a test tool as they would with any locally installed software. Instead, they benefit from the service through reports sent by 0-day when it finds potential bugs in code sent to the mailing lists or within monitored development trees. Thus, despite having no direct control over the start of tests performed by 0-day, many developers perceive they are using the tool very often.

Lastly, no driver maintainer declared to use any of the six testing tools introduced by academic works assessed in this research. In fact, very few maintainers do even know about those tools. This result indicates that, so far, driver testing tools introduced by academic works have failed to impact the Linux development community.

At the end of Part III, we presented respondents with three open-ended questions to allow them to report any other piece of information they would judge relevant to driver testing. The first of them (question 12) asked if we should consider any other tool for device driver testing. Other than the tools reported in question 8, maintainers also mentioned kcsan, and klocwork in response to question 12. Some respondents also went further by commenting on details of test tools or giving their opinions about them: "*I don't know any of the test tools; but in 'make menuconfig' I noticed that the I2C stack (and probably others, too) support fault injection.*", "*The intel-gfx / i915 devs have extensive CI for the drm/kms subsystem. The sparse/build tests are universal (for drm/kms code) the actual functionality tests are all run on Intel graphics only*", "*gitlab-ci or equivalent should be used by maintainers to ensure tests are run and pass before merging.*", "*roadtest for i2c drivers. Not upstream yet but very useful.*" Last, we highlight a couple of comments mentioning QEMU as a supporting tool for device driver testing: "*QEMU Emulation, Mocked Emulation*", "*It's not strictly a testing tool, but qemu can be helpful for testing drivers on emulated hardware*".

With the next open-ended question (question 13: "What do you think are the main challenges for device driver testing?"), we intended to hear directly from driver maintainers about the troubles and challenges related to testing device drivers. To our surprise, 57 out of 85 maintainers who have completed the survey have answered question 13.

The most mentioned issue was having access to the hardware required to do tests, as reported by 25 maintainers. This outcome should be of moderate concern since it indicates that 29% of device driver maintainers may find it hard to test changes against real hardware devices. Developers may have difficulty testing device drivers either because they do not have access to a specific device or because they do not have access to all different device variants supported by a driver. Some respondents reported as challenges: "*HW access - I do not have working HW for the stuff I maintain these days*", "*from subsystem maintainer PoV: missing hardware availability*", "*Hardware availability for drivers that support many devices with small (or not so small) differences in the programming model.*"

The limitation imposed by the lack of hardware to test a device driver might impact the extension of tested code that goes upstream, as reported by a driver maintainer: "*Hardware availability. I cannot test on hardware that I do not own. To perform tests for hardware that I do not own, I have to modify the driver itself, which means I am not submitting the exact driver that I have tested.*"

A possible solution to the hardware availability problem would be to develop virtual devices to enable drivers to probe and bind to a device, allowing at least a few code paths to be tested. Further, a custom virtual device would also respond to driver requests, mocking operations from a hardware counterpart. Some survey participants were categorial about that in their responses, classifying device emulation as a challenge for driver testing: "*Lack of hardware meaning emulation development is often necessary. Sometimes stubbing functionality is sufficient*", "*Emulating the device side*", "*Emulating hardware*".

However, virtual devices are not a silver bullet. First, an emulated device would only be able to provide fake data since it would hardly interface with other devices on the system or ever capture real-world data. Second, designing a proper device model for some hardware pieces might be tricky because they may have a large state space or non-deterministic behavior. In the words of survey participants, testing a device driver may be challenging due to the "*lack of good documentation and behavior models that could be used to test drivers without the hardware.*" or because of "*hardware behavior unpredictability and range of potential states*". Another maintainer argued that "*You need real hardware. Because anything else (eg stubs, emulations, or models) have to be bug-for-bug compatible for the testing to be relevant. Real hardware also exhibits timing variations/interactions that are nearly impossible to replicate any other way, and most of the time you'll find "undocumented features"*". Last, even though virtual devices may help test device drivers under certain limitations, most hardware devices don't have a software counterpart. Due to that, this test strategy may imply an additional burden to device driver developers, who would first have to develop a virtual device with which they would test their patches.

Aside from difficulties having access to hardware, device driver maintainers reported a few more obstacles related to driver testing. One of those issues with driver testing stems from the fact that the Linux kernel is highly configurable. The kernel has thousands of configuration options. Each of them may include a different source file for compilation or even specific parts of source files. If, for instance, one may enable a driver by setting a single configuration option, then the driver is expected to work in every kernel built from any other combination of values assigned to the remaining configuration options. However, even though some bugs may occur only with specific kernel configurations, it's impractical to test a driver with every possible valid kernel configuration because there are billions of them. Some developers have expressed this difficulty: "*One problem is the number of possible combined configuration options. You may need to test more than one kernel.*", "*issues may only show up in unusual combinations of kernel configurations; also: combinatorial explosion of options*". While the combinatorial explosion of configuration options is not a problem exclusive to device drivers, it indeed constitutes a challenge when it comes to testing the Linux kernel.

Testing Linux kernel device drivers is especially challenging because it "*involves determining if a piece (or many different pieces) of hardware behaves as expected, as well as whether a program behaves as expected.*" In this context, "*the hardware is critical, and most hardware designs don't foresee testing as you might wish*". For example, it may not be easy to test device drivers under particular operating conditions. For instance, a hardware piece may require specific handling procedures when working over a predefined temperature. Setting up exact environmental conditions or "*being able to reproduce various external conditions to trigger device behavior*" may be hard to accomplish sometimes.

On top of that, hardware may behave unpredictably or in unexpected ways that complicate device driver development and testing. "*[...] Hardware can behave in unexpected ways, and it can be difficult to prepare a driver for all situations that can occur. [...]*", "*hardware behavior unpredictability and range of potential states*", "*The fickleness of hardware. In some cases, the lack of decent documentation of the hardware*".

Regarding test tools, even though there is no consensus about whether to test device drivers against emulated or against actual hardware, a few maintainers expressed concerns about how tests are carried on. "*The testing community is very much oriented around software testing other software. Testing hardware states is different. If you want to control the test rig as much as you do with software testing software, you have to first implement an emulation of the hardware to be able to e.g. provoke different fault states (fault injection).*" Another maintainers testified that: "*as for writing tests, for example unit tests, it can be really hard to accurately model the hardware component. This means that for proper testing, a system with the target hardware platform must be available for anyone wanting to run such tests. [...] Bots, like the aforementioned syzbot, have no real way to test the driver's functionality (in fact, the only reports I got back from them were compile errors for some more obscure platforms).*" Yet another survey participant told that "*at least for media drivers, there are too many possible permutations to test. Unit tests typically are not really suitable for these types of drivers, the tests have to happen at the system level, hence the development [...] of the compliance utilities maintained in https://git.linuxtv.org/v4l-utils.git/*". A fourth device driver maintainer mentioned that "*the learning curve for testing tools/infrastructures can be very high. With there being so many of them it is impossible to study them all and pick the best ones for your area.*"

We've also obtained replies conveying challenges related to automated device driver testing. "*Hardware that may fail being part of testing loop and it is hard to distinguish in an automated system whether the failure is due to driver bug or a hw problem.*", "*Board farm management for testing a wide diversity of hardware.*", "*Automating testing - especially automating the hardware feedback loop: By capturing video, emulating input devices, designing homebrew storage devices with fault injection, etc.*", Nevertheless, despite the adversities of automated testing, no one objected to it. In fact, some participants have expressed that automated testing is the way to go. "*Centralizing CI services involves booting boards, which is hard, but is necessary for linux to start achieving usable quality at release time*", "*Automating the testing can also be sometimes very difficult. Manually testing will usually just take too long.*"

Lastly, some developers complained about hardware manufacturers' support for device driver testing. "*The funding of hardware companies accorded to software testing is mostly still very limited, if not unwanted.*", "*Hardware availability of niche devices - even when you work for the device manufacturer, you may only have very limited access to flaky hardware, which makes it very hard to automate testing*", "*At least from Nouveau's perspective, the bigger issue with testing is finding someone to handle the bringup of infrastructure, along with reliable hosting infrastructure that we could hook up to real hw. We're spending enough time just trying to keep up!*"

To summarize, the challenges most reported by device driver maintainers were having access to the hardware (mentioned by 29% of respondents), followed by automating tests (referred by 7% of respondents), then enhancing test tools (6% of respondents).

The last question of Part III was question 14: "What would you recommend the Linux kernel community do to improve device driver testing?". Thirty-one device driver maintainers answered this question. The top three most reported suggestions were to advance in automated test and CI systems (mentioned by 9% of developers), to invest more in hardware emulation, virtual models, or mock frameworks (8% of respondents), and to ask for more support from hardware manufacturers (5% of participants). Maintainers also suggested we (the Linux kernel community) should enhance the documentation with a summary of available test tools and best practice guidelines and help educate ourselves about better testing methods.

Once more, the answers reveal that there is no consensus about whether or not the community should go for testing drivers against virtual devices. While the majority of maintainers see benefits in using virtual devices for driver testing, a few others argue against it and consider "*there is no substitute for actual hardware when it comes to device drivers*".

An intereseting suggestion in the direction of having more tests on hardware devices considered the possibility of putting efforts in "*making it easy for testing, and for reporting test results, for those with access to the real device*".

Moreover, we identified several improvement suggestions to current test tools and infrastructure. These ideas and opinions reflect the desires of device driver maintainers with respect to the features offered by available test tools (*RQ3*).

From those who wanted more automated tests, we note a positive opinion about current testing services and an appeal for hooking more test systems to development trees. "*Hire more people to work on testing: writing test scenarios, writing tools for automated testing on target HW, writing tools for automated testing via mocks without target HW, proving public servers for running automated tests (e.g. for every sent patch), increase test coverage, [...]*", "*Offline testing, such as KernelCI and 0-day, seem extremely useful to me as follow-up for pre-posting (in-house) testing. I'd recommend exploring how such approaches can be extended both in terms of coverage and when testing occurs - before/after patches are posted/merged.*", "*Improve the discoverability of current processes for hooking into things like KernelCI. Another place I think we could improve on is companies that have their own testing farms, but that don't have them publicly accessible.*" "*Implement SW-controlled test benches for every bus/protocol using HW that supports endpoint and controller roles. Hook that to the specific kernel maintainer branches so they are run on every patch inclusion.*"

Some other maintainers expressed a desire for device mocking or emulation systems, one of them reporting a positive experience with those. "*Invest more in compose-able emulation and mocked models.*", "*Maybe a driver/hardware mocking framework. If that exists I am unaware of it.*", "*Write device emulations to test device drivers in, e.g. QEMU. If available the tests can even be automated.*", "*For the media subsystem it was very helpful to create virtual drivers (i.e. drivers emulating media hardware), as that is very useful to catch regressions in core media frameworks, and it allows testing media APIs for hardware types that are otherwise very difficult to obtain (if at all).*"

We observed a relatively high number of responses asking for device mocking, virtualization, or emulation. We believe the reason for that is because these strategies are seen as solutions to two major issues reported by maintainers. First, virtualization provides a way of testing device drivers without hardware. Second, software artifacts may be easier to manage than the hardware devices they emulate, simplifying the creation and maintenance of automated test environments. Thus, device virtualization was mentioned by many developers.

Chapter 6

Triangulation and Discussion

Now that we have collected information from three different research methods (formal literature, grey literature, and community survey), let's compare our findings and merge them through synthesis by integration. According to Rousseau *et al.* (2008), synthesis by integration is characterized by comparing evidence involving two or more data collection methods. Likewise, we triangulated the data from our research to corroborate a comprehensive view of how Linux kernel device drivers are being tested (*RQ1*).

6.1 Triangulation

Overall, we did not identify any conflicting information between formal and grey literature publications. However, we note that grey literature publications did not mention any test tool introduced by the academic papers we assessed in this study. Therefore, we focus on comparing information about tools reported in community publications.

Starting with LTP, Khan (2014) and Iyer (2012) say that LTP contains a collection of tools to test the reliability, robustness, and stability of the Linux kernel and related features. Further, Zaidenberg and Khen (2015) adds that the test suite methodology is also based on regression testing.

Still talking about the Linux Test Project, Claudi and Dragoni (2011) states that LTP can test essential Linux kernel features such as filesystems, device drivers, memory management, scheduler, disk I/O, networking, syscalls, and IPC. Similarly, Khan (2014) says that, by default, LTP run script includes tests for filesystems, disk I/O, memory management, inter process communication (IPC), the process scheduler, and the system call interface. Also, both Claudi and Dragoni (2011) and Khan (2014) assert that some Linux testing projects are built on top of LTP or incorporate it somewhat. Regarding device driver testing, Renzelmann *et al.* (2012b) reported that LTP can invoke drivers and verify their behavior, but it requires the device to be present. They also mention that LTP cannot verify properties of individual driver entry points because it runs tests at the system-call level.

Rothberg *et al.* (2016) were consistent with Coccinelle's home page (*Coccinelle: A Program Matching and Transformation Tool for Systems Code* 2022) when they cited Coccinelle as a Linux kernel static analysis tool that also helps code collateral evolution. They've also acknowledged the activity of continuous integration (CI) testing services, rating the 0-day test robot as the most prominent tool among automated testing services. One year later, G. K.-H. Jonathan Corbet (2017) ranked 0-day as the top bug reporter during the development period between 4.8 and 4.13 Linux kernel releases.

Y. Chen *et al.* (2013) introduced a test tool called KIS (Kernel Instant bug testing Service) which runs static analysis tests with GCC, Sparse, Smatch, and Coccinelle. They also mentioned that KIS runs dynamic analysis tests with Trinity, xfstests, and mmtests. In turn, B. Chen *et al.* (2020) commented that dynamic analysis tools have been receiving more attention in recent years and mentioned Syzkaller among fuzz testing tools.

With respect to Kselftest, both *Linux Kernel Selftests* (2021), *Kernel self-test* (2019), and Rothberg *et al.* (2016) say the test suite has unit tests that should execute quickly and exercise individual code paths.

Thus, formal and grey literature complement each other in describing Linux kernel test tools.

6.2 Discussion

We highlight that neither the community publications reviewed by us nor the responses to survey questions 8 and 12 have mentioned any testing tool introduced by academic papers assessed during our mapping study. In addition, the results from survey question 11 indicate that no device driver maintainer uses any of the testing tools proposed by academic papers we've assessed. Therefore, we consider that the Linux community is not using those test tools presented by academic studies to test the kernel.

Instead, we back on three pieces of evidence to provide a list of the test tools used by the Linux community to test the kernel (*RQ2*). The first piece of evidence we consider is the collection of responses to survey question 8 ("Do you use any tool to test the Linux kernel yourself? If so, which?") and to question 12 ("Should we consider any other tool for device driver testing? If so, which?"). These were open-ended (non-suggestive) questions in which device driver maintainers could report any tool they felt they used for testing the kernel. We will denote evidence from answers to those questions by the *spontaneous* keyword.

The second piece of evidence we consider comes from the collection of test tools we organized from information gathered during our mapping study and grey literature review (Table A.1). Every tool there was referenced by at least one publication, even though many were mentioned by three articles or more. Moreover, we have inspected the repositories of each tool in that table and estimated their activity status based on whether they have received contributions from January 2021 to March 2022. Thus, we consider (and denote) that any tool in Table A.1 regarded as being actively developed has *activity* evidence in favor of it.

The third and last piece of evidence we considered is the set of responses to survey question 11 ("How familiar are you with these tools/testing infrastructure?"). This question presented a list of twenty tools we assessed throughout our research and offered five options to let respondents report if they knew about those tools and how often they used each of them. If a particular tool got at least one reply asserting a driver maintainer uses it, then we understand such response as *usage frequency* evidence indicating that the specific tool is being used.

With that, we estimate with high confidence that Coccinelle, KernelCI, Kselftest, ktest, KUnit, Smatch, Sparse, and Syzkaller/Syzbot are being used by the community to test Linux. These tools have *spontaneous*, *activity*, and *usage frequency* pieces of evidence indicating their use.

In addition to those, we also have high confidence that kernel developers use 0-day, LKFT, LTP, Trinity, and TuxMake because these are supported by *activity* and *usage frequency* pieces of evidence.

Another few tools we highly believe are being used by Linux developers are perf, igt-gpu-tools, and kvm unit tests. *Spontaneous* and *activity* pieces of evidence support these.

Some other tools reported in responses to question 8 or question 12 were not present in Table A.1 either because we did not identify them during our literature review or because we disregarded them as test tools. Consequently, we did not assess their development activity status or usage nor presented them in question 11. Anyway, we list those tools here since they have *spontaneous* evidence of use and thus are probably being used by device driver maintainers, even though we are not sure all of them fit the software test category. These tools are kernel_patch_verify, rdma tools, tcpdump, lockdep, piglit, slub, pps-tools, roadtest, tools/testing/cxl, virtme, coverity, cec-compliance, checkpatch, can-utils, mdio-tools, ethtool, ping, kmemleak, qemu, kasan, kcsan, klocwork.

Lastly, Table A.1 has tools that were not mentioned in responses to questions 8, 11, or 12. Nevertheless, we believe that Linux developers may use those tools which only have *activity* pieces of evidence in their favor. However, we cannot estimate a usage probability for those tools because they differ in the number of publications mentioning each one. Also, we perceive different data sources with slightly different regards. Thus, the test tools with only *activity* evidence supporting them may have a low to high chance of being used by the Linux community. To simplify the visualization, we added the number of referencing publications after each activity evidence marker in Table 6.1. Table 6.1 shows the complete list of testing tools being used by the Linux community to test the kernel (*RQ2*, *RO1*) and the types of evidence supporting them.

Tool	*Supporting evidence*
Coccinelle	*spontaneous, activity, usage frequency*
KernelCI	*spontaneous, activity, usage frequency*
Kselftest	*spontaneous, activity, usage frequency*
ktest	*spontaneous, activity, usage frequency*
KUnit	*spontaneous, activity, usage frequency*
Smatch	*spontaneous, activity, usage frequency*
Sparse	*spontaneous, activity, usage frequency*

continue ⟶

Table 6.1: *Test tools used by the Linux community to test the Linux kernel.*

Tool	*Supporting evidence*
Syzkaller/Syzbot	*spontaneous, activity, usage frequency*
0-day	*activity, usage frequency*
LKFT	*activity, usage frequency*
LTP	*activity, usage frequency*
Trinity	*activity, usage frequency*
TuxMake	*activity, usage frequency*
perf	*spontaneous, activity*
IGT GPU Tools	*spontaneous, activity*
KVM Unit Tests	*spontaneous, activity*
kernel_patch_verify	*spontaneous*
RDMA Tools	*spontaneous*
tcpdump	*spontaneous*
lockdep	*spontaneous*
Piglit	*spontaneous*
SLUB	*spontaneous*
pps-tools	*spontaneous*
roadtest	*spontaneous*
CXL Tests	*spontaneous*
virtme	*spontaneous*
Coverity	*spontaneous*
cec-compliance	*spontaneous*
checkpatch	*spontaneous*
can-utils	*spontaneous*
mdio-tools	*spontaneous*
ethtool	*spontaneous*
ping	*spontaneous*
kmemleak	*spontaneous*
QEMU	*spontaneous*
KASAN	*spontaneous*
KCSAN	*spontaneous*
klocwork	*spontaneous*
Dr. Checker	*activity* (1)
DIFUZE	*activity* (1)
mmtest	*activity* (1)
KCOV	*activity* (1)
cyclictest	*activity* (1)
hackbench	*activity* (3)
rcutorture	*activity* (2)
Hulk Robot	*activity* (2)
Buildbot	*activity* (1)
Continuous Kernel Integration (CKI)	*activity* (2)
AutoTest	*activity* (2)
	continue ⟶

Table 6.1: *Test tools used by the Linux community to test the Linux kernel.*

Tool	*Supporting evidence*
xfstests	*activity* (3)
LAVA - Linaro Automated Validation Architecture	*activity* (2)
Fuego	*activity* (2)
KMSAN	*activity* (1)
LISA	*activity* (2)
KTSAN	*activity* (1)
LKMM / litmus-test	*activity* (1)
herd	*activity* (2)
TuxSuite	*activity* (1)
Schbench	*activity* (1)
Rt-app	*activity* (1)
Phoronix Test Suite	*activity* (1)
Marvin	*activity* (1)
Undertaker	*activity* (1)
S Suite	*activity* (1)

Table 6.1: *Test tools used by the Linux community to test the Linux kernel.*

As for the features of these tools, some provide unit tests, and others do integration tests, stress tests, fuzz testing, and many other types of tests listed in the *Testing Techniques* column of Table A.1. Most of the tools in Table 6.1 may be run by individual developers, and some of them are run by automated test services that fetch and test patches from mailing lists or from Linux kernel trees. Detailed information about the tools that we assessed during this research and their capabilities were presented in Subsection 4.2.2.

Coming back to the broader research question *RQ1*, we now use the evidence collected throughout this study to outline how Linux device drivers are tested. We will divide the answer to this question into two parts: tests performed by individual community members who develop, maintain, or test Linux device drivers; and tests performed by automated test systems and CI rings.

First, according to our survey results, 90% of Linux kernel device driver maintainers do test patches frequently (always or, at least, most times) before sending them to any mailing list. When testing drivers they maintain, 84% of Linux developers do at least basic soundness tests such as probing and binding drivers to devices they supports. Moreover, 68% of driver maintainers also assess driver functionality through kernel space to user space ABI, and 38% of them run test tools to catch potential bugs. Respectively, 19% and 15% of driver maintainers who always runtime test their patches said to run either static or dynamic analysis tools very often (always or, at least, most times). With some regularity, 42% of device driver maintainers run Sparse, 20% run Smatch, 16% run Kselftest, 16% run Coccinelle, 15% run Syzkaller, 7% run KUnit, 7% run LTP, 4% run Trinity, 2% run TuxMake, and 2% of them run ktest.

Throughout our GLR, we have found a few examples of developers and companies said to have been testing the Linux kernel. The Linux Foundation carried on a series of interviews with Linux kernel developers. To them, Arnd Bergmann said to have started doing a lot of build-testing to improve the quality of merged contributions (*Linux Kernel Developer: Arnd Bergmann* 2017). Laura Abbott said to have spent a lot of time testing and reviewing patches for kernel hardening (*Linux Kernel Developer: Laura Abbott* 2017). Shuah Khan said to have boot tested stable kernel release candidates (*Linux Kernel Developer: Shuah Khan* 2017). A kernel development report revealed that Linus Torvalds routinely boot tests the kernel that results after accepting a pull request (G. K.-H. Jonathan Corbet, 2017). Another report from the Linux Foundation pointed out that the Real-Time Linux development team will have to test and adjust new incoming features to maintain the kernel's real-time capability (*Real-Time Linux Continues Its Way to Mainline Development and Beyond* 2017). Companies seem to have more specific interests. Collabora intends to have the DRM subsystem under continuous integration, validating new drivers with the IGT test suite (Vizoso, 2016). ARM and Linaro use a real-time workload simulator called rt-app to trigger specific scheduler code paths to test small scheduling and load-balancing changes (Fleming, 2017). Oracle tests Linux kernels with workloads related to their products, such as Oracle Engineered Systems, Oracle Cloud Infrastructure, and enterprise deployments for Oracle customers (*Oracle Q&A: A Refresher on Unbreakable Enterprise Kernel* 2018).

Besides the tools mentioned above, evidence indicates that another myriad of software is used to test Linux device drivers. Table 6.1 lists the tools for which we have acquired evidence of usage in the context of Linux kernel testing. Even though not all of those are strictly test tools, most of them are available to anyone. Thus, many other individual developers, who are not necessarily driver maintainers, may be running them to test Linux somehow.

Speaking about things that are not precisely test tools, many survey participants have reported test practices not related to any test tool in particular. The most mentioned of those practices was using hardware emulation, device models, or mocks, to carry on the tests over device drivers. Particularly, QEMU was mentioned by some maintainers as a tool that allowed such tests. In addition, a few other maintainers said running custom scripts or using in-kernel functionality (such as fault injection capabilities) to test device drivers. We estimate that the number of developers performing tests through custom scripts, QEMU models, or manual inspection is not negligible. 30% of driver maintainers who said to always runtime test their patches don't use any static or dynamic analysis tool. They represent 16% of all driver maintainers and constitute a lower bound estimative of the portion of developers who might benefit from integrating test tools into their workflows.

Most of the tests performed by individual developers probably occur during the patch development and review cycle since the Linux kernel community adopts an RTC (Review Then Commit) policy for incorporating changes.

As the second part of the answer to *RQ1*, we now talk about tests performed by machines. There is a number of test robots and CI systems testing the Linux kernel. Throughout this study, we identified a few of them: 0-day, LKFT, KernelCI, Syzbot, Hulk, CKI, and Buildbot. These rings often provide tests as a service and incorporate several test tools such as Smatch, Coccinelle, LTP, Kselftest, libhugetlbfs, v4l2-compliance tests, and many others. Nevertheless, the set of monitored trees and the frequency each one undergoes testing may vary from robot to robot. For instance, the 0-day test robot fetches patches from mailing lists and key developers' trees and has a response time of one hour around the clock (hence the 0-day name). KernelCI focuses on testing upstream kernels by generating several kernel build configuration sets and submitting them to boot tests in various labs worldwide. Syzbot fuzzes main Linux kernel trees and reports bugs to kernel mailing lists. LKFT focus on testing Linux release candidates, linux-next, and long-term-stable releases on the arm and arm64 hardware architectures. LKFT is said to report test results in up to 48 hours.

We recall that 28% of device driver maintainers reported having registered their development trees with Linux kernel testing services. This rate rises to 34% among maintainers with six or more years of Linux development experience. However, we consider these to be lower-bound estimates of the portion of trees covered by testing services since most (if not all) development trees are publicly available and, therefore, can be obtained and tested without consent from any developer or maintainer. Moreover, because Linux kernel development trees are accessible, they might be monitored by many other test robots beyond those identified by our study. 43% of maintainers who contribute as part of their jobs said their organizations had a CI system to test the Linux kernel. Whether the reports from those systems are publicly accessible or not, sharing test results may contribute to a higher collaboration toward Linux kernel testing. Indeed, one of the suggestions for improving device driver testing was for companies to make their test systems publicly accessible.

As a gross estimate of how many developers benefit from test services, we point out that, respectively, 32%, 19%, and 4% of maintainers said to use the 0-day, KernelCI, and LKFT services with some regularity (often or, at least, sometimes).

Aside from outlining current Linux kernel testing tools, we indicate opportunities to improve existing test tools based on suggestions from community developers (*RQ3*). Starting with Trinity, an ongoing task is to add support for new system calls and system call flags introduced by more recent kernel releases. Enhancing support for network protocols and adding tests for common syscall patterns such as open, read, close was also on the wishlist of the fuzzer maintainer Dave Jones (Kerrisk, 2013). Although Dave suggested these improvements in 2013, some of those proposals, and many others, remain in the project's TODO file.

In August 2014, kernel developers desired Kselftest features such as the ability to execute tests in a few minutes or seconds, run groups of tests at once, and that test source code was kept in the kernel source tree. A few additional features were suggested then, but those requests seem to have been fulfilled throughout the years.

Back in 2016, Vizoso (2016) reported that an area where KernelCI could improve was in its test coverage. He commented that even though build and boot regressions were annoying for developers because they impacted everybody working on the affected configurations and hardware, regressions on peripheral support or other subsystems not triggered during the boot process could still make rebases costly.

As more general suggestions regarding Linux kernel testing, we note that in an interview published in the 2017 Linux Kernel Development Report (G. K.-H. Jonathan Corbet, 2017), Dan Williams reported that the community should work on accelerating the growth of a unit test culture for the Linux kernel. Also, Khan (2014) mentioned that there is no requirement that testers should be developers.

From our Device Driver Testing Survey results, we point out that 28% of device driver maintainers reported having registered their development trees with automated test services, thus indicating an appreciation of test bots and CI rings. Nevertheless, from the responses to question 14, we observed various suggestions for improving and expanding current test services. The directions are for writing automated tests with mocks, increasing test coverage, increasing the number of test iterations, and making closed test farms open and publicly accessible.

Moreover, the lack of hardware to test patches was the top challenge reported by driver maintainers in responses to question 13. Conversely, solutions that would allow testing without hardware were also mentioned by many developers in several questions. For instance, four developers have said using QEMU to test the kernel in their responses to question 8. Later, QEMU was also cited twice more in answers to both questions 12 and 14 as a tool that may help driver testing due to its ability to emulate hardware.

Thus, the evidence we have gathered suggests that the Linux kernel community has two top desires concerning driver testing. One, to expand existing CI and automated test systems, and two, to have more emulation and mocking implementations for testing drivers when the required hardware is unavailable.

6.3 Considerations on Linux Kernel Device Driver Tests

Linux kernel device drivers may be hard to test due to many issues. For example, one often needs specific hardware to exercise most, if not all, code paths of a driver. Also, laying devices under particular operating conditions or providing them with certain stimuli may be difficult. In addition, kernel crashes complicate the process of getting test output compared to application crashes. Further, hardware may eventually fail and behave unpredictably.

Alternatively, one could exercise device driver code using emulation, virtualization, mocking, or software alike. However, it's not clear whether software can replace hardware for all test purposes. Correctly mimicking hardware behavior and creating test cases that properly simulate real-world usage are some challenges to these test approaches.

While CI rings and test robots may compile and boot test the Linux kernel, they might lack test cases to assess device driver functionality. According to Linux maintainers who participated in our survey, extending CI systems may be challenging apart from the hardware access issue because there is "*no culture of supplying unit tests and functional tests with drivers.*" We recall that Vizoso (2016) has commented about improving the test coverage of KernelCI and his concerns about regressions in code for driving peripherals. Moreover, a survey participant reported they only got compile time error reports from test robots. From such evidence, one might wonder if the current automated test services provide adequate test coverage for device driver code.

Whatever the answer to this question, we should not disregard automated test services, for providing high device driver test coverage is a tough affair. One reason for that is that, even though not all hardware devices need or have an in-kernel driver for them, the overall number of devices that do need (and have) in-kernel drivers tends to grow as hardware manufacturers keep releasing new designs. Moreover, hardware designs may differ a lot from each other, possibly having particular interfaces, register maps, operating modes, capabilities, execution contexts, and other distinguishing characteristics that can turn the process of creating test cases into a device-specific task. Thus, providing extensive test coverage for Linux kernel device drivers might require a continuous effort to expand test infrastructure as more drivers are developed.

Thinking in a scenario where hardware devices keep diversifying, we wonder whether hardware manufacturers shouldn't behave more proactively in leveraging device driver tests. For example, if manufacturers could provide virtual devices able to mock their hardware designs, then automated test systems would be able to use them to exercise additional driver code paths. Thinking further, we wonder if one could automate the process of creating device mocks from the hardware design. For instance, if one could generate a state diagram from a hardware description (such as a VHDL[1] program), then that model could probably be used to create a QEMU virtual device to mock the real one. Having accessible device mocks would then leverage Linux kernel device driver tests.

[1] VHDL stands for VHSIC Hardware Description Language. VHSIC is an acronym for Very High Speed Integrated Circuits.

Chapter 7

Conclusion

The Linux kernel is tested by many community developers and automated test services in several different ways. However, to the extent we could explore, the academic literature does not properly cover test practices of the Linux kernel development community. In particular, no publication has addressed the state-of-practice of Linux device driver tests. This research investigated how Linux kernel device drivers are being tested. With data from a grey literature review, we characterized the twelve most mentioned test tools in community publications. With additional data from a community survey, we achieved our second research goal of identifying test tools and strategies adopted by Linux device driver maintainers. By synthesizing data from our mapping study, grey literature review, and community survey, we achieved our first research objective of providing a list of tools used by Linux kernel developers to test the kernel. Finally, we achieved our third research objective by providing a dissertation addressing the state-of-practice of Linux kernel device driver testing.

7.1 Contributions

This research has provided a broad view of how the Linux kernel is being tested. We have provided a list of tools used to test the Linux kernel based on evidence from three scientific research methods. In addition, we have characterized eighteen test tools for assisting Linux kernel tests. Moreover, we have conducted a survey with device driver maintainers to enlighten the state-of-practice of Linux kernel device driver testing, including its challenges and possible advances. We provided reasonable answers to three original research questions, which contribute to enlarging the scientific body of knowledge related to the Linux kernel. The largest open-source software project up today.

7.2 Threats to Validity

Concerned about missing important data related to device driver testing, we relaxed our tool selection criteria when planning our grey literature review. When reading documents from the formal literature, we selected tools for assessment only if they were advocated as automated driver test tools. In contrast, when reviewing community publications, we selected a diverse pool of driver test tools for evaluation, whether they were designed for automated tests or not. Because we did not select non-automated Linux kernel test tools from academic works for assessment, the considerations we made don't truly generalize for Linux device driver testing tools. Moreover, it's hard to argue that our work properly covers Linux kernel automated driver test tools because we have little information about some of them.

Throughout this study, we have assessed 399 articles in the field of computer science and 250 publications from the grey literature. From them, we have focused our attention on 19 papers and 65 community documents only. However, we cannot guarantee to have acquired enough information to describe the actual state of practice of Linux kernel device driver testing. There may be many more publications with relevant information about device driver tests that were not covered by our study. Thus, we believe that further investigation, especially of grey literature documents, would lead to a more accurate understanding of driver test practices. Also, the document selection for the mapping study and the grey literature review was made by a single person; therefore, the list of selected publications and tools is not free of reviewer bias issues.

After conducting our Device Driver Testing Survey, we realized that device drivers are scattered throughout the Linux kernel source in directories other than the *drivers* directory. For instance, we have identified drivers under the *block*, *net*, and *sound* directories. Thus, we did not reach all device driver maintainers with our survey. Nonetheless, we believe that we have indeed contacted a representative portion of Linux kernel device driver maintainers. With a few modifications to our *get_driver_maintainers.awk* program, we counted the number of driver entries in Linux's MAINTAINERS file and the number of those entries that did not declare files under the *drivers* directory. For the 5.17 stable Linux release, there are 1496 driver entries declared in the MAINTAINERS file, and only 49 of those artifacts do not contain file patterns that match the *drivers* directory. Thus, as far as we can estimate, we missed a maximum of 49 (4%) of driver maintainers in our survey invitations.

Finally, even though we have addressed some of the test tools that run automated test services and CI rings, the information we obtained is shallow. We didn't find, for example, precise information about how regularly automated test tools run their jobs or what events if any, trigger test execution. Thus, our considerations about automated test tools cannot refute any information given directly by automated test service maintainers.

7.3 Future Work

Evidence indicates that test robots lack testing coverage, especially for device driver functionality. Nevertheless, most device driver maintainers agree that expanding automated testing is the way to go. So, despite their current limitations, automated tests hold a positive opinion from Linux kernel developers. Exploring the role of Linux CI rings and test robots may help better understand their impact on the project and comprehend why developers keep a positive opinion about them.

Appendix A

Linux Kernel Test Tools from the Literature

After finishing our systematic mapping study and our grey literature review, we comprised the Linux kernel test tools we identified into a table with information such as tool name, estimated activity status, testing techniques, repository, and supporting publications. The information in Table A.1 later served as usage evidence for the tools identified throughout the literature review processes.

Tool Name	*Activity Status*	*Testing Techniques*	*Citations*
COD	Inactive	Concolic Execution, instrumentation	L1
ADFI (Automatic Driver Fault Injection)	Inactive	Instrumentation, fault injection	L2
LgDb	Inactive	Debug	L3
FAUMachine	Active	Fault injection	L6
TIMEOUT	Unknown	Instrumentation, fault / delay injection	L8
SymDrive	Inactive	Symbolic execution	L1, L12
EH-Test	Inactive	Fault injection	L14
Troll	Inactive	Local analysis and grouping	L18
Kprobe	Unknown		L1
LDV	Unknown		L1
WHOOP	Inactive		L1
Dr. Checker	Active		L1
DSAC	Unknown		L1
DEADLINE	Unknown		L1
DCNS	Unknown		L1

continue ⟶

Table A.1: *Linux kernel test tools identified in the literature.*

Tool Name	*Activity Status*	*Testing Techniques*	*Citations*
DIFUZE	Active		L1
TriforceAFL	Inactive		L1
kAFL	Inactive		L1
Razzer	Inactive		L1
DDT	Unknown		L1
CAB-Fuzz	Unknown		L1
Linux Fault Injection Capabilities Infrastructure (LFII)	Inactive	Fault injection	L2, G11
mmtest	Active		L9
PF-Miner	Unknown		L14
ktest	Active	Build testing	G11, G27, G28
Trinity	Active	Fuzz testing	L9, G22, G47, G59, G60
KCOV	Active		G22
Syzkaller / Syzbot	Active	Fuzz testing	G16, G22, G29, L1, G47
cyclictest	Active	Load testing/ stress testing	G17
hackbench	Active	Load testing/ stress testing	L16, G17, G50
Linux Test Project (LTP)	Active	Functional testing, reliability testing, robustness testing, stability testing, stress testing	L3, L12, L16, G11, G25, G47, G63
Sparse	Active	Static analysis	G11, G47
Smatch	Active	Static analysis	G11, G26, G47
RCU-torture	Active	Stress test	G8, G38
KUnit	Active	Unit testing	G12
Kselftest	Active	Unit testing, regression test, stress testing, functional testing, performance testing	L18, G6, G11, G23, G30, G35, G48, G57
Coccinelle / coccicheck	Active	Static analysis, semantic patches	L18, G47, G56, G57
0-day test robot	Active	Regression testing, functional testing, build testing, performance testing	L18, G35, G47, G48, G57, G58, G61
Hulk Robot	Acitve		G47, G61
Buildbot	Active	Build testing	G47

continue ⟶

Table A.1: *Linux kernel test tools identified in the literature.*

Tool Name	*Activity Status*	*Testing Techniques*	*Citations*
LKFT (Linux Kernel Functional Testing)	Active	Regression testing, functional testing, build testing	G47, G54, G62, G63
Continuous Kernel Integration (CKI)	Active		G53, G64
KLive	Unknown		G1
AutoTest	Active		G3, G11
xfstests	Active		L9, G5, G13
FAFT (Fully Automated Firmware Testing)	Inactive		G5
jstest	Inactive		G9, G10, G43
LAVA - Linaro Automated Validation Architecture	Active		G11, G54
Fuego	Active		G14, G24
perf	Active		G17, G63
KMSAN	Active		G29
KernelCI	Active	Build testing	G31, G44, G47, G53, G57
LISA	Active		G39, G49
KTSAN	Active		G40
LKMM / litmus-test	Active	Matemathical / formal methods	G40
herd	Active	Matemathical / formal methods	G40, G51
TuxMake	Active	Build testing	G41, G52, G54
TuxSuite	Active		G54
IGT GPU Tools	Active		S1, G44, G65
libnvdimm	Active		G12, G48
Schbench	Active		G50
Adrestia	Inactive		G50
Rt-app	Active	End-to-end testing	G50
Phoronix Test Suite	Active		G55
Marvin	Active		G57
Xen tests	Unknown		G57
Undertaker	Active		G57
Libhugetlbfs	Inactive		G63
Video4Linux (v4l2)	Active		G63
KVM Unit Tests	Active		G63
S Suite	Active		G63

Table A.1: *Linux kernel test tools identified in the literature.*

Appendix B

get_driver_maintainers.awk

In August 2021, Shuah Khan and Kate Stewart launched a survey to Linux kernel developers. They wanted to assess the effectiveness of Linux Foundation efforts to encourage the participation of underrepresented groups of developers in the Linux kernel community. Another goal of their research was to understand the reasons why developers stop or take a break from active participation in the community. Thus, they targeted their survey to all developers listed in the Linux kernel MAINTAINERS file as stated in a snippet of the e-mail they sent to them.

```
‘‘Your input is important to us, hence we are reaching out to you by
sending email to all your contact addresses listed in the MAINTAINERS
file. We apologize if there is duplication.’’
```

Unlike Shuah and Stewart’s research, our topic does not relate to all Linux kernel developers. Instead, we seek especially the opinion and experience of device driver developers. Thus, we made an AWK program to retrieve only the set of developers that maintain artifacts under the drivers directory. Designed to receive a MAINTAINERS file as an argument and print out the names and email addresses of Linux kernel device driver maintainers, we named it *get_driver_maintainers.awk*. We then contacted device driver developers by sending our survey to all email addresses obtained from *get_driver_maintainers.awk*. The program source code can be seen below.

We extracted maintainers’ email addresses from the Linux kernel 5.17 MAINTAINERS file because it was the latest stable Linux version by the time we started inviting developers to answer our survey. Program B.2 lists the set of commands we ran.

Program B.1 get_driver_maintainers.awk.

```
#!/usr/bin/awk -f

BEGIN {
   FPAT = "([^\t<>]+)"
   get_devs = 0 # flag to start recording driver developer’s emails
   new = 0 # flag to tell apart each entry
   lost = 0 # sanity check we don’t miss any driver maintainer
   print_names = 1
   print_emails = 1
   print_statistics = 1
   named_artifacts = 0 # number of named entries/artifacts under drivers dir
}
{
if (/Maintainers List/)
   get_devs = 1

if (get_devs == 1) {
   if (/^$/) {
      new = 1
      delete emails_buf # clear email buffer
   } else {
      if (/M:\t/)
         emails_buf[$3] = $2 # use email address as key and name as value

      # We expect no M:\t line after a F:\t line within a single entry.
      if (/M:\t/ && new == 0)
         lost++ # Must not happen

      if (/F:\tdrivers\// && new == 1) {
         if (length(emails_buf) > 0) {
            for (email in emails_buf) {
               dev_addresses[email] = emails_buf[email] # hash table
               dev_names[emails_buf[email]] = 0 # to count distinct names
            }
         }
         named_artifacts++;
         new = 0
      }
   }
}
}
END {
   for (email in dev_addresses) {
      if (print_names && print_emails)
         printf "%s <%s>\n", dev_addresses[email], email
      else if (print_names)
         printf "%s\n", dev_addresses[email]
      else if (print_emails)
         printf "%s\n", email
   }
   if (print_statistics) {
      printf "%d named artifacts under drivers dir\n", named_artifacts
      if (lost > 0)
         printf "[Warn] %d uncounted device driver developer(s)!\n", lost
      printf "%d device driver maintainers\n", length(dev_names)
      printf "%d addresses of driver maintainers\n", length(dev_addresses)
   }
}
```

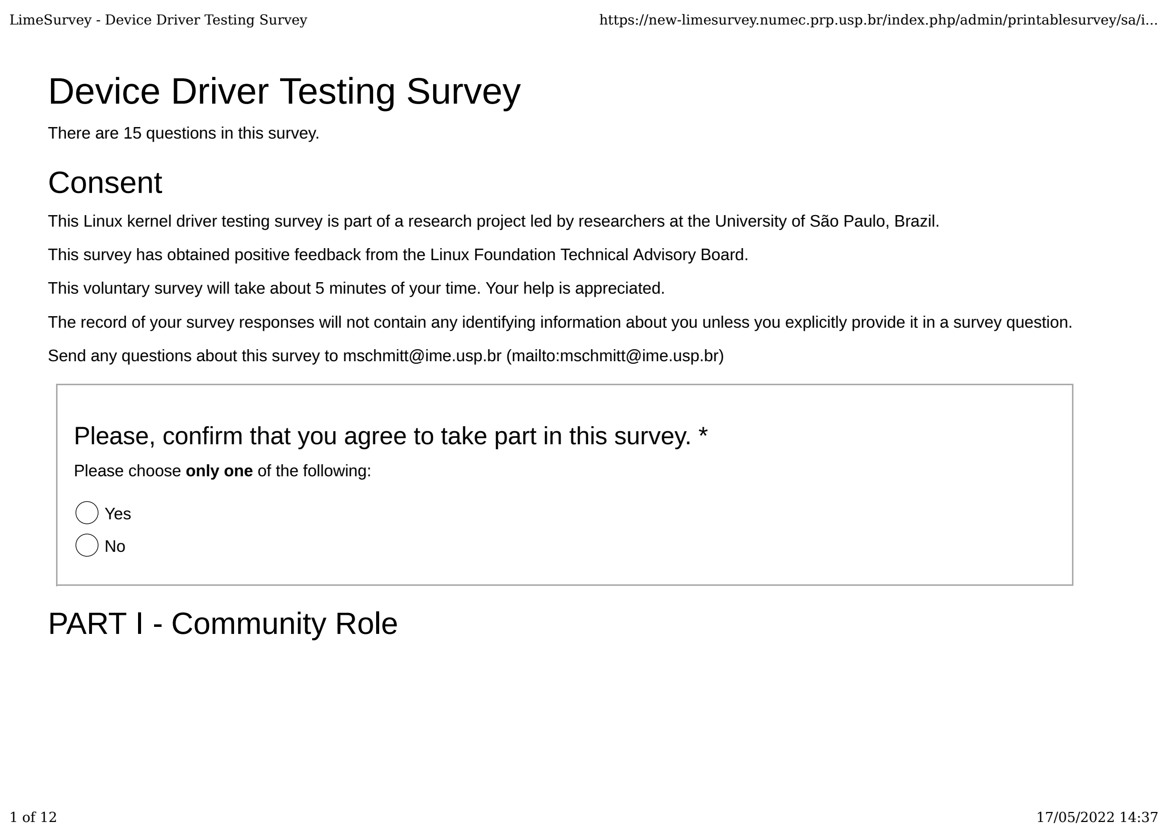
Program B.2 Set of commands to extract maintainer's email addresses.

```
wget https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/plain/
    MAINTAINERS?h=v5.17.8 -O MAINTAINERS_5.17 && \
./get_driver_maintainers.awk MAINTAINERS_5.17 | \
sort | tee email_list
```

Appendix C

Device Driver Testing Survey

This appendix contains a printable version of the device driver testing survey we invited Linux kernel developers to answer.



Device Driver Testing Survey

There are 15 questions in this survey.

Consent

This Linux kernel driver testing survey is part of a research project led by researchers at the University of São Paulo, Brazil.

This survey has obtained positive feedback from the Linux Foundation Technical Advisory Board.

This voluntary survey will take about 5 minutes of your time. Your help is appreciated.

The record of your survey responses will not contain any identifying information about you unless you explicitly provide it in a survey question.

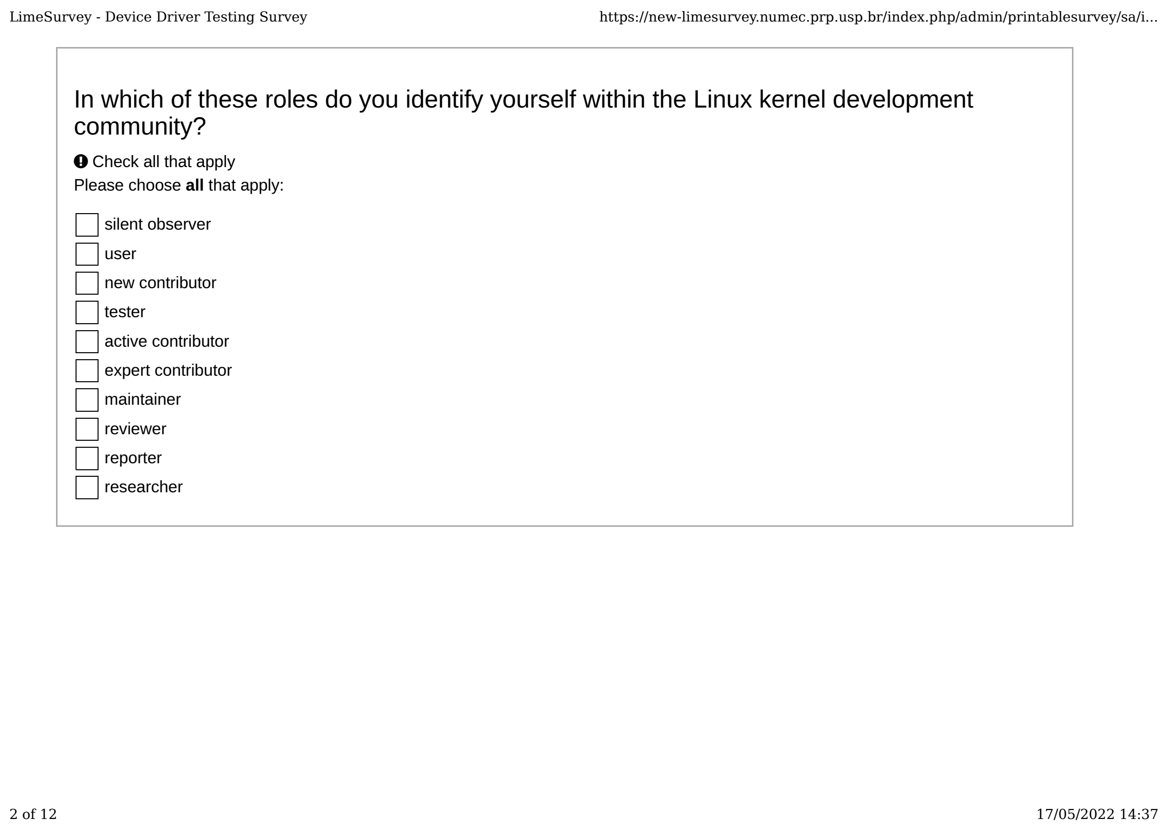
Send any questions about this survey to mschmitt@ime.usp.br (mailto:mschmitt@ime.usp.br)

Please, confirm that you agree to take part in this survey. *

Please choose **only one** of the following:

○ Yes
○ No

PART I - Community Role

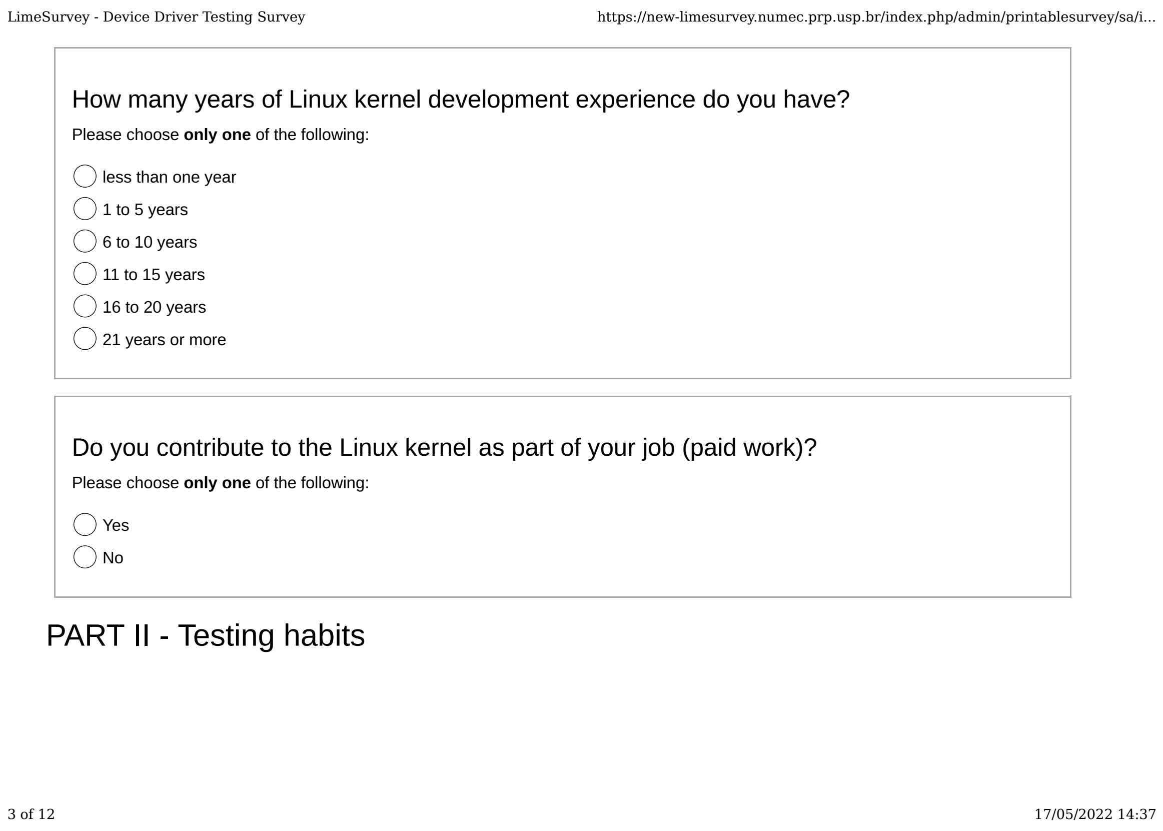


In which of these roles do you identify yourself within the Linux kernel development community?

❗ Check all that apply

Please choose **all** that apply:

- ☐ silent observer
- ☐ user
- ☐ new contributor
- ☐ tester
- ☐ active contributor
- ☐ expert contributor
- ☐ maintainer
- ☐ reviewer
- ☐ reporter
- ☐ researcher



How many years of Linux kernel development experience do you have?

Please choose **only one** of the following:

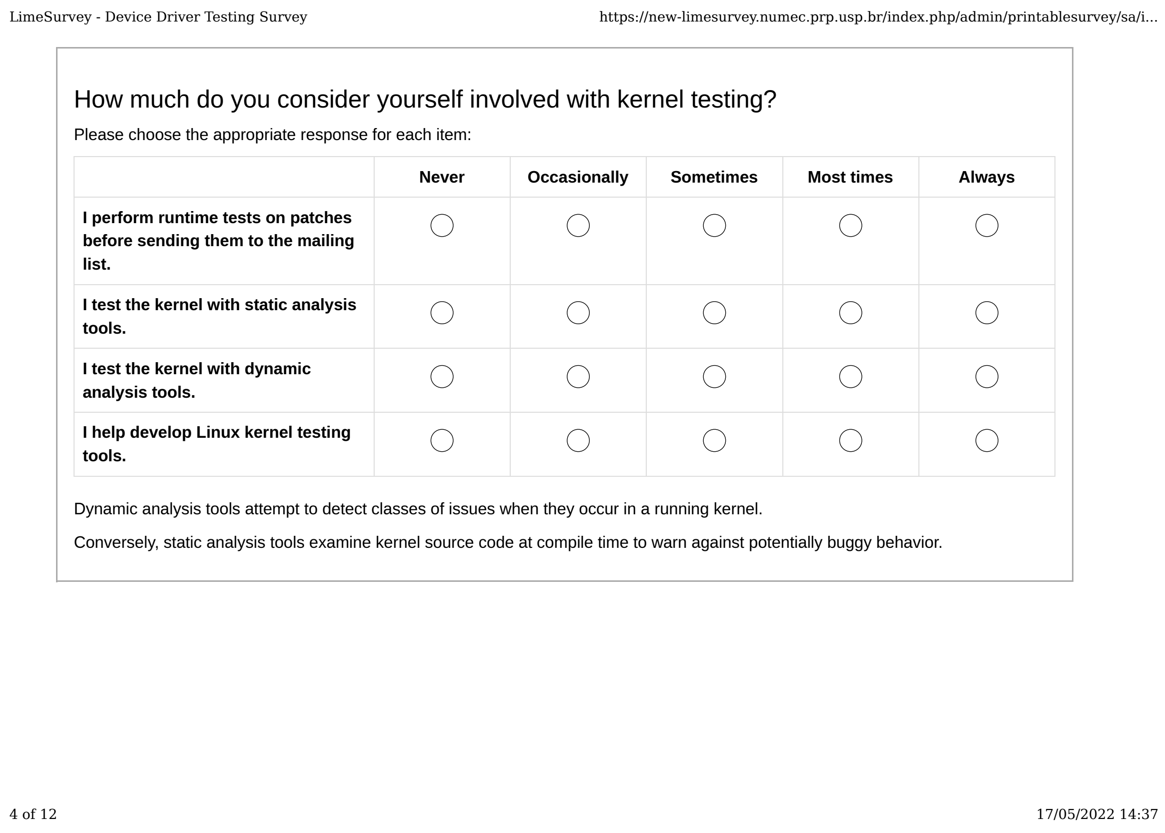
- ◯ less than one year
- ◯ 1 to 5 years
- ◯ 6 to 10 years
- ◯ 11 to 15 years
- ◯ 16 to 20 years
- ◯ 21 years or more

Do you contribute to the Linux kernel as part of your job (paid work)?

Please choose **only one** of the following:

- ◯ Yes
- ◯ No

PART II - Testing habits



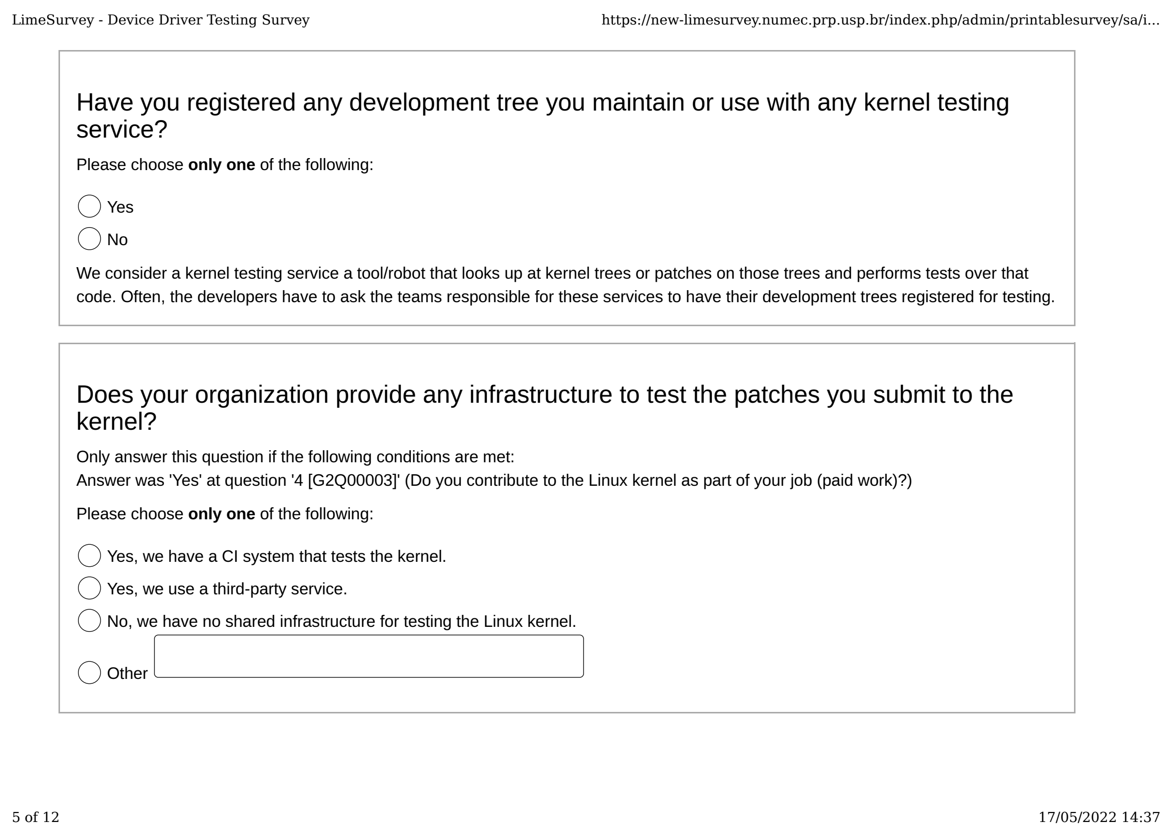
How much do you consider yourself involved with kernel testing?

Please choose the appropriate response for each item:

	Never	Occasionally	Sometimes	Most times	Always
I perform runtime tests on patches before sending them to the mailing list.	○	○	○	○	○
I test the kernel with static analysis tools.	○	○	○	○	○
I test the kernel with dynamic analysis tools.	○	○	○	○	○
I help develop Linux kernel testing tools.	○	○	○	○	○

Dynamic analysis tools attempt to detect classes of issues when they occur in a running kernel.

Conversely, static analysis tools examine kernel source code at compile time to warn against potentially buggy behavior.



Have you registered any development tree you maintain or use with any kernel testing service?

Please choose **only one** of the following:

◯ Yes

◯ No

We consider a kernel testing service a tool/robot that looks up at kernel trees or patches on those trees and performs tests over that code. Often, the developers have to ask the teams responsible for these services to have their development trees registered for testing.

Does your organization provide any infrastructure to test the patches you submit to the kernel?

Only answer this question if the following conditions are met:
Answer was 'Yes' at question '4 [G2Q00003]' (Do you contribute to the Linux kernel as part of your job (paid work)?)

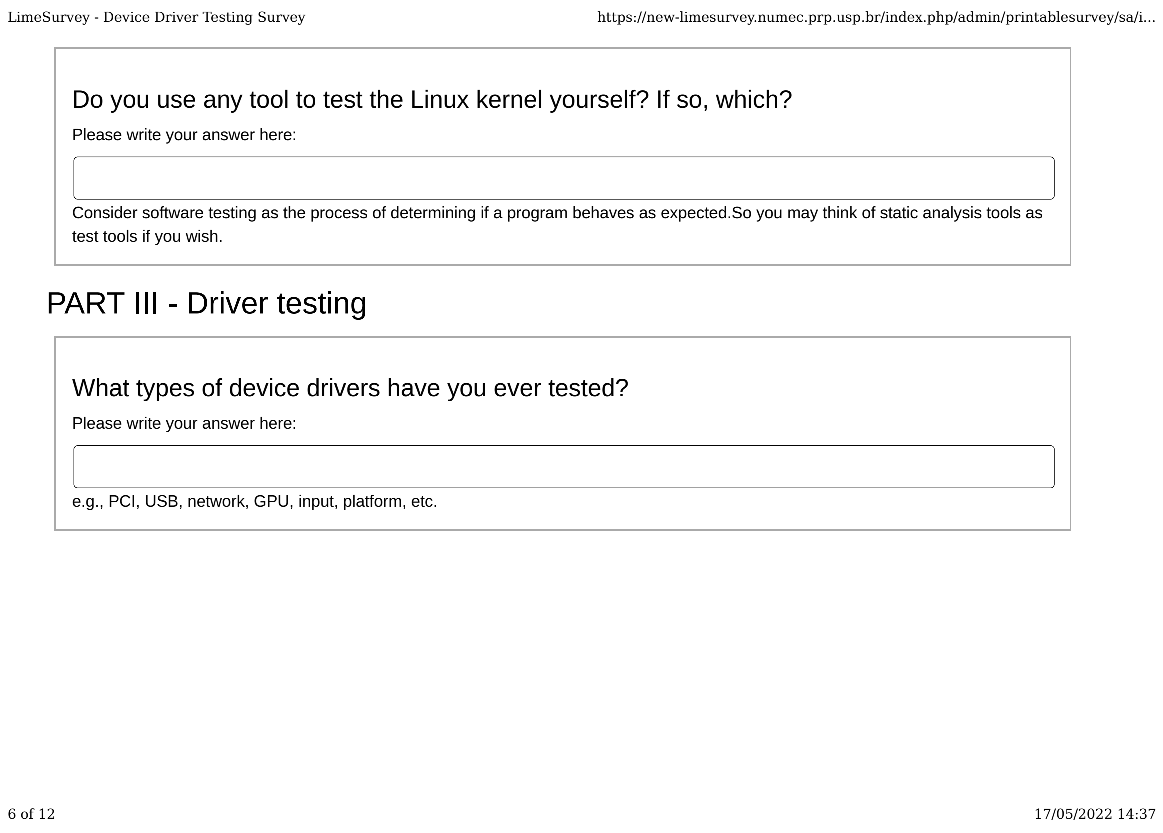
Please choose **only one** of the following:

◯ Yes, we have a CI system that tests the kernel.

◯ Yes, we use a third-party service.

◯ No, we have no shared infrastructure for testing the Linux kernel.

◯ Other



Do you use any tool to test the Linux kernel yourself? If so, which?

Please write your answer here:

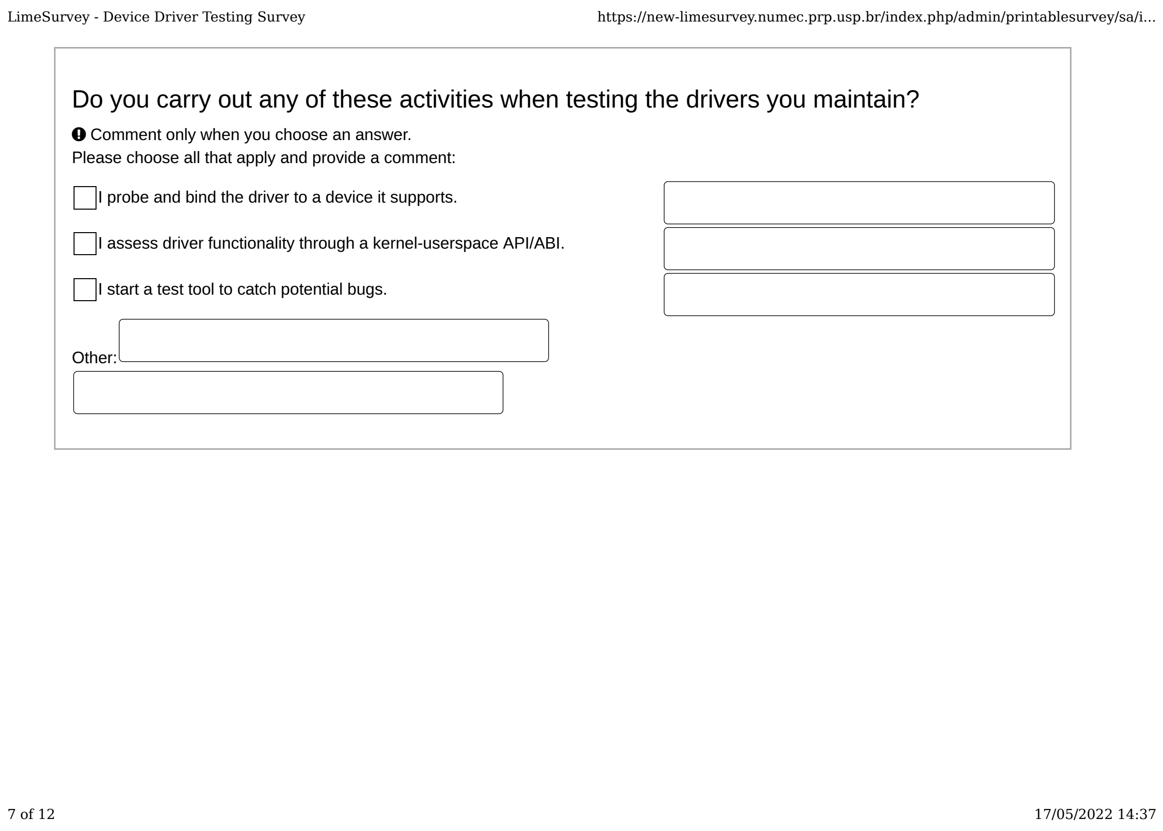
Consider software testing as the process of determining if a program behaves as expected.So you may think of static analysis tools as test tools if you wish.

PART III - Driver testing

What types of device drivers have you ever tested?

Please write your answer here:

e.g., PCI, USB, network, GPU, input, platform, etc.



LimeSurvey - Device Driver Testing Survey https://new-limesurvey.numec.prp.usp.br/index.php/admin/printablesurvey/sa/i...

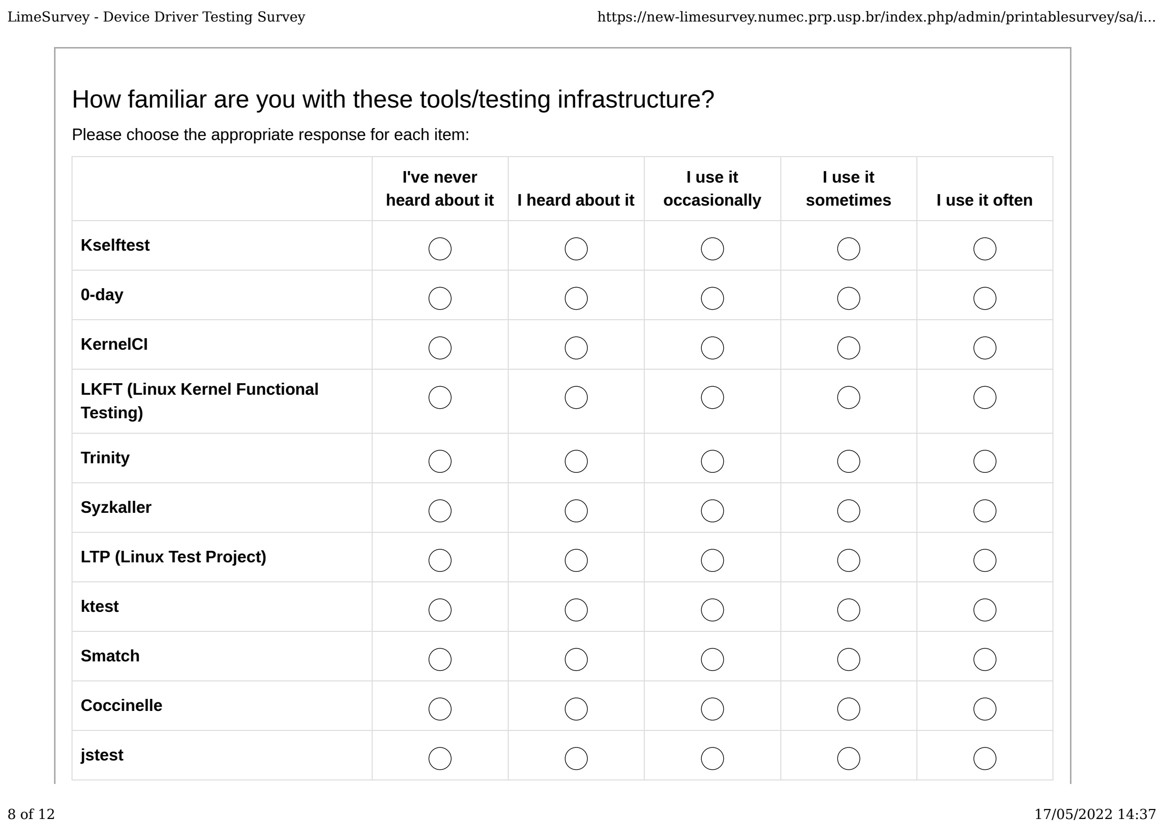
Do you carry out any of these activities when testing the drivers you maintain?

❗ Comment only when you choose an answer.
Please choose all that apply and provide a comment:

☐ I probe and bind the driver to a device it supports.	
☐ I assess driver functionality through a kernel-userspace API/ABI.	
☐ I start a test tool to catch potential bugs.	

Other:

7 of 12 17/05/2022 14:37



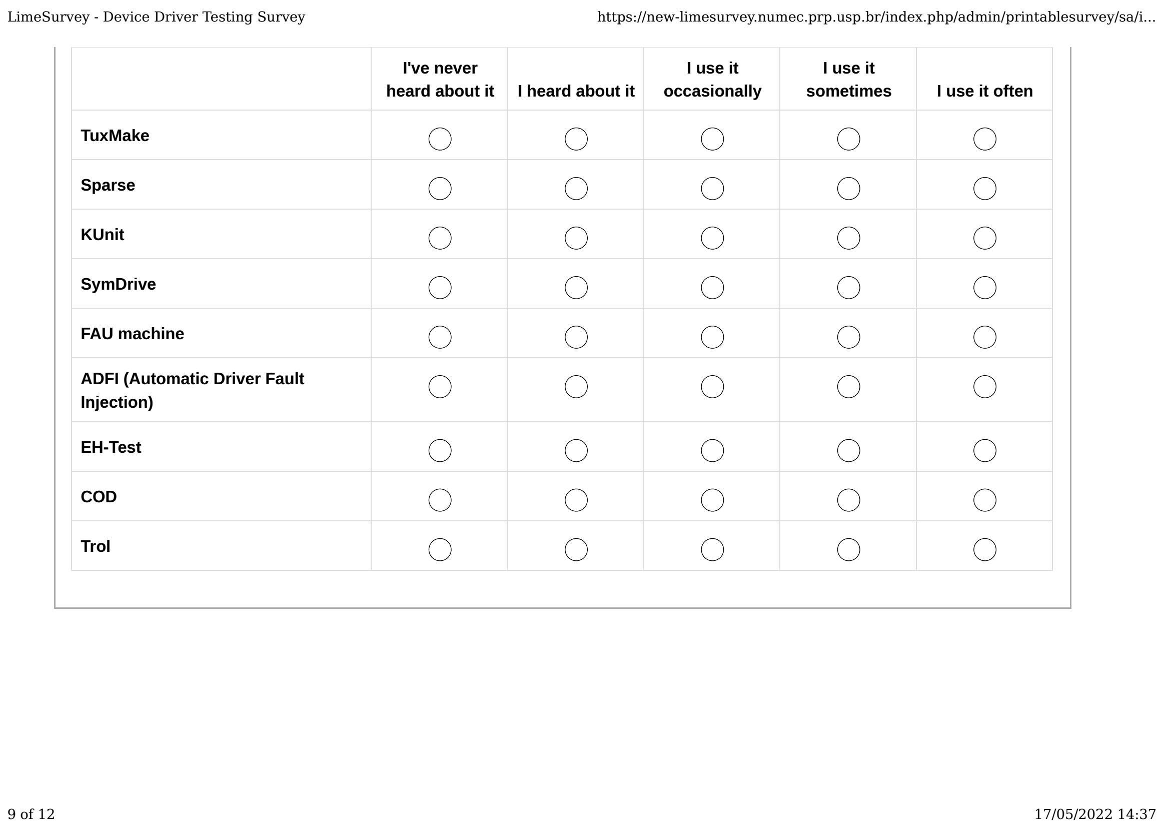
LimeSurvey - Device Driver Testing Survey https://new-limesurvey.numec.prp.usp.br/index.php/admin/printablesurvey/sa/i...

How familiar are you with these tools/testing infrastructure?

Please choose the appropriate response for each item:

	I've never heard about it	I heard about it	I use it occasionally	I use it sometimes	I use it often
Kselftest	○	○	○	○	○
0-day	○	○	○	○	○
KernelCI	○	○	○	○	○
LKFT (Linux Kernel Functional Testing)	○	○	○	○	○
Trinity	○	○	○	○	○
Syzkaller	○	○	○	○	○
LTP (Linux Test Project)	○	○	○	○	○
ktest	○	○	○	○	○
Smatch	○	○	○	○	○
Coccinelle	○	○	○	○	○
jstest	○	○	○	○	○

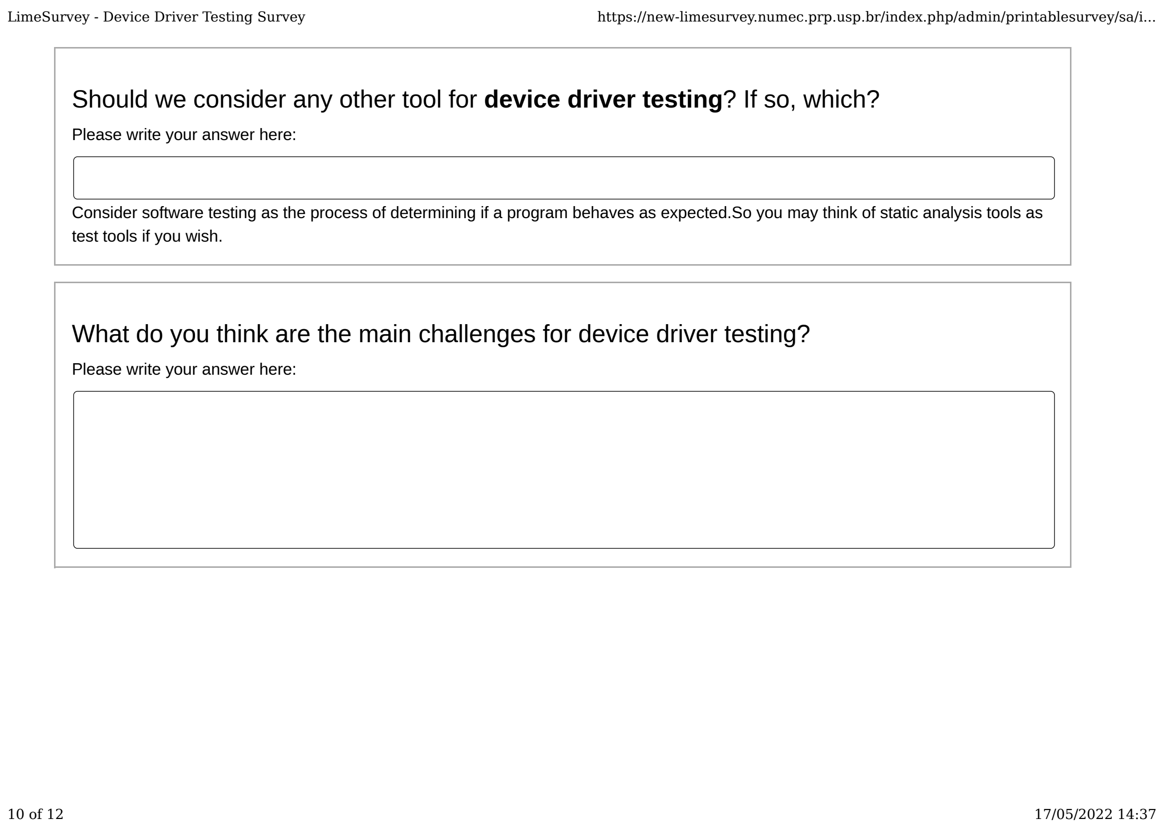
8 of 12 17/05/2022 14:37



LimeSurvey - Device Driver Testing Survey https://new-limesurvey.numec.prp.usp.br/index.php/admin/printablesurvey/sa/i...

	I've never heard about it	I heard about it	I use it occasionally	I use it sometimes	I use it often
TuxMake	○	○	○	○	○
Sparse	○	○	○	○	○
KUnit	○	○	○	○	○
SymDrive	○	○	○	○	○
FAU machine	○	○	○	○	○
ADFI (Automatic Driver Fault Injection)	○	○	○	○	○
EH-Test	○	○	○	○	○
COD	○	○	○	○	○
Trol	○	○	○	○	○

9 of 12 17/05/2022 14:37



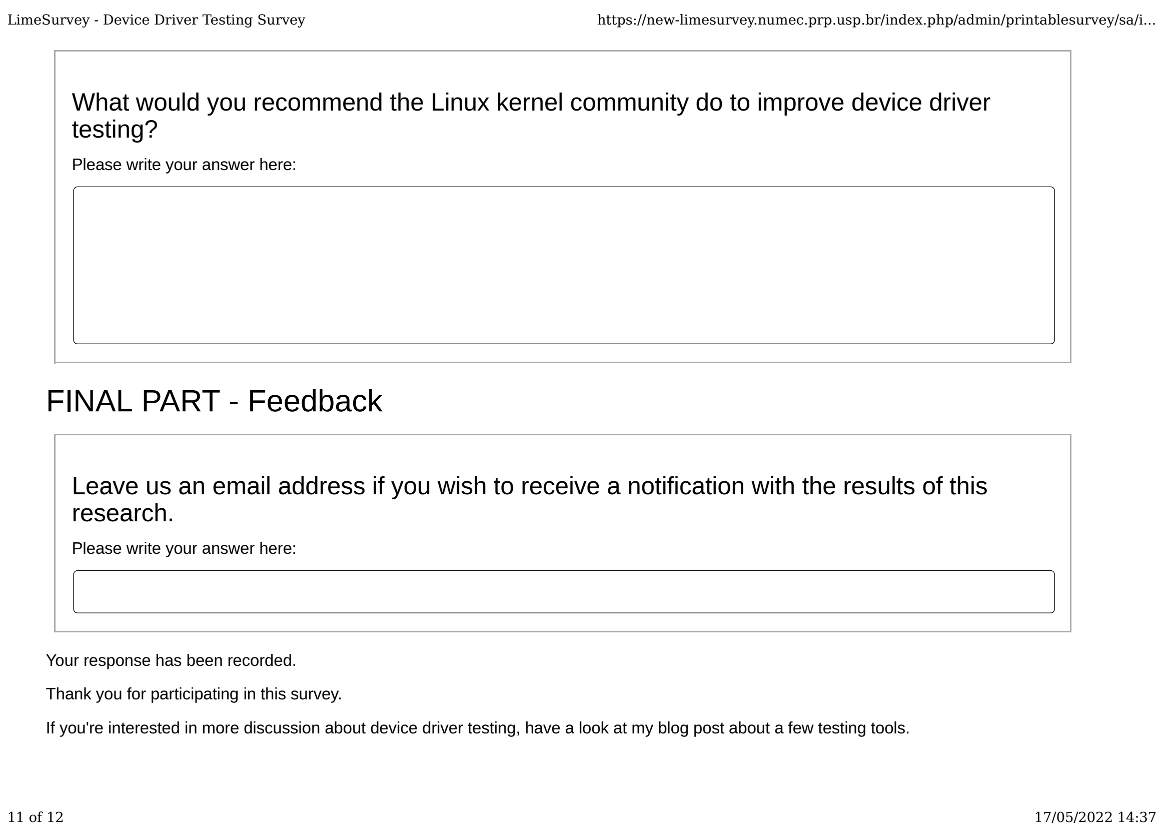
Should we consider any other tool for **device driver testing**? If so, which?

Please write your answer here:

Consider software testing as the process of determining if a program behaves as expected.So you may think of static analysis tools as test tools if you wish.

What do you think are the main challenges for device driver testing?

Please write your answer here:



What would you recommend the Linux kernel community do to improve device driver testing?

Please write your answer here:

FINAL PART - Feedback

Leave us an email address if you wish to receive a notification with the results of this research.

Please write your answer here:

Your response has been recorded.

Thank you for participating in this survey.

If you're interested in more discussion about device driver testing, have a look at my blog post about a few testing tools.

Submit your survey.
Thank you for completing this survey.

Appendix D

Device Driver Testing Survey Invitation Letter

Dear Linux kernel Maintainer,

I am Marcelo Schmitt, a graduate student at the University of São Paulo (Brazil). In addition to maintaining the AD7292 device driver, I work on a research project about Linux device drivers under the supervision of professors Paulo Meirelles and Fabio Kon.

Our goal is to characterize the current practices concerning device driver testing and grasp what one could do to improve the effectiveness of those tests.

Our work has already provided a contribution to the Kernel Testing Guide documentation page.
Link: https://lore.kernel.org/linux-doc/cover.1648674305.git.marcelo.schmitt1@gmail.com

I would highly appreciate it if you could take about 5 minutes of your time to answer the survey below, which will help us identify device driver test habits and practices. We are contacting you because you are listed in the MAINTAINERS file as the maintainer of some artifact under the drivers directory.

Your response is very important to us. Not only will reporting your unique experience greatly assist our research efforts, but your participation might help guide the development of driver testing tools. We apologize if you are not a device driver developer.

The survey can be found at:
https://new-limesurvey.numec.prp.usp.br/index.php/313985?lang=en

Thanks,
Marcelo

Appendix E

Device Driver Testing Survey Results

This appendix contains statistical data from complete responses to objective questions of the Device Driver Testing Survey we carried on with Linux kernel device driver maintainers.

The questionnaire had a total of fifteen questions structured in five groups:

1. Group 1 - Welcoming page
2. Group 2 - PART I - Community Role
3. Group 3 - PART II - Testing habits
4. Group 4 - PART III - Driver testing
5. Group 5 - FINAL PART - Feedback

To avoid misunderstandings, we identify each question by its group, question number, and, when applicable, subquestion number. For instance, we denote the tenth survey question as group 3, question 2.

In which of these roles do you identify yourself within the Linux kernel development community?		
Answer	*Count*	*Gross percentage*
Researcher	2	2.35%
New Contributor	5	5.88%
Silent Observer	17	20.00%
Tester	17	20.00%
Reporter	19	22.35%
Expert Contributor	26	30.58%
Reviewer	41	48.23%
User	53	62.35%
Active Contributor	56	65.88%
Maintainer	70	82.35%
Total (gross)	306	360.00%

Table E.1: *Complete responses to group 2, question 1.*

How many years of Linux kernel development experience do you have?		
Answer	*Count*	*Gross percentage*
less than one year	2	2.35%
1 to 5 years	22	25.88%
6 to 10 years	24	28.23%
11 to 15 years	15	17.64%
16 to 20 years	11	12.94%
21 years or more	11	12.94%
No answer	0	0.00%
Total (gross)	85	100.00%

Table E.2: *Complete responses to group 2, question 2.*

Do you contribute to the Linux kernel as part of your job (paid work)?		
Answer	*Count*	*Gross percentage*
Yes	54	63.52%
No	28	32.94%
No answer	3	3.52%
Total (gross)	85	100.00%

Table E.3: *Complete responses to group 2, question 3.*

How much do you consider yourself involved with kernel testing? I perform runtime tests on patches before sending them to the mailing list.		
Answer	*Count*	*Gross percentage*
Never	2	2.35%
Occasionally	1	1.17%
Sometimes	5	5.88%
Most times	30	35.29%
Always	47	55.29%
No answer	0	0.00%
Total (gross)	85	100.00%

Table E.4: *Complete responses to group 3, question 1, subquestion 1.*

How much do you consider yourself involved with kernel testing? I test the kernel with static analysis tools.		
Answer	*Count*	*Gross percentage*
Never	33	38.82%
Occasionally	15	17.64%
Sometimes	11	12.94%
Most times	16	18.82%
Always	7	8.23%
No answer	3	3.52%
Total (gross)	85	100.00%

Table E.5: *Complete responses to group 3, question 1, subquestion 2.*

How much do you consider yourself involved with kernel testing? I test the kernel with dynamic analysis tools.		
Answer	*Count*	*Gross percentage*
Never	32	37.64%
Occasionally	20	23.52%
Sometimes	20	23.52%
Most times	6	7.05%
Always	3	3.52%
No answer	4	4.70%
Total (gross)	85	100.00%

Table E.6: *Complete responses to group 3, question 1, subquestion 3.*

How much do you consider yourself involved with kernel testing? I help develop Linux kernel testing tools.		
Answer	*Count*	*Gross percentage*
Never	50	58.82%
Occasionally	15	17.64%
Sometimes	9	10.58%
Most times	4	4.70%
Always	2	2.35%
No answer	5	5.88%
Total (gross)	85	100.00%

Table E.7: *Complete responses to group 3, question 1, subquestion 4.*

Have you registered any development tree you maintain or use with any kernel testing service?		
Answer	*Count*	*Gross percentage*
Yes	24	28.23%
No	57	67.05%
No answer	4	4.70%
Total (gross)	85	100.00%

Table E.8: *Complete responses to group 3, question 2.*

Does your organization provide any infrastructure to test the patches you submit to the kernel?		
Answer	*Count*	*Gross percentage*
Yes, we have a CI system that tests the kernel.	23	42.59%
Yes, we use a third-party service.	1	1.85%
No, we have no shared infrastructure for testing the Linux kernel.	23	42.59%
Other	3	5.55%
No answer	4	7.40%
Total (gross)	54	100.00%

Table E.9: *Complete responses to group 3, question 3.*

Do you carry out any of these activities when testing the drivers you maintain?		
Answer	*Count*	*Gross percentage*
I probe and bind the driver to a device it supports.	71	83.52%
I assess driver functionality through a kernel-userspace API/ABI.	58	68.23%
I start a test tool to catch potential bugs.	32	37.64%
Other	3	3.52%
Total (gross)	164	192.94%

Table E.10: *Complete responses to group 4, question 2.*

How familiar are you with these tools/testing infrastructure? kselftest		
Answer	*Count*	*Gross percentage*
I've never heard about it	25	29.41%
I heard about it	27	31.76%
I use it occasionally	11	12.94%
I use it sometimes	7	8.23%
I use it often	7	8.23%
No answer	8	9.41%
Total (gross)	85	100.00%

Table E.11: *Complete responses to group 4, question 3, subquestion 1.*

How familiar are you with these tools/testing infrastructure? 0-day		
Answer	*Count*	*Gross percentage*
I've never heard about it	22	25.88%
I heard about it	14	16.47%
I use it occasionally	14	16.47%
I use it sometimes	9	10.58%
I use it often	18	21.17%
No answer	8	9.41%
Total (gross)	85	100.00%

Table E.12: *Complete responses to group 4, question 3, subquestion 2.*

How familiar are you with these tools/testing infrastructure? KernelCI		
Answer	*Count*	*Gross percentage*
I've never heard about it	18	21.17%
I heard about it	32	37.64%
I use it occasionally	13	15.29%
I use it sometimes	13	15.29%
I use it often	3	3.52%
No answer	6	7.05%
Total (gross)	85	100.00%

Table E.13: *Complete responses to group 4, question 3, subquestion 3.*

How familiar are you with these tools/testing infrastructure? LKFT		
Answer	*Count*	*Gross percentage*
I've never heard about it	54	63.52%
I heard about it	19	22.35%
I use it occasionally	3	3.52%
I use it sometimes	0	0.00%
I use it often	0	0.00%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.14: *Complete responses to group 4, question 3, subquestion 4.*

How familiar are you with these tools/testing infrastructure? Trinity		
Answer	*Count*	*Gross percentage*
I've never heard about it	48	56.47%
I heard about it	25	29.41%
I use it occasionally	3	3.52%
I use it sometimes	0	0.00%
I use it often	0	0.00%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.15: *Complete responses to group 4, question 3, subquestion 5.*

How familiar are you with these tools/testing infrastructure? Syzkaller		
Answer	*Count*	*Gross percentage*
I've never heard about it	21	24.70%
I heard about it	37	43.52%
I use it occasionally	9	10.58%
I use it sometimes	9	10.58%
I use it often	4	4.70%
No answer	5	5.88%
Total (gross)	85	100.00%

Table E.16: *Complete responses to group 4, question 3, subquestion 6.*

How familiar are you with these tools/testing infrastructure? LTP		
Answer	*Count*	*Gross percentage*
I've never heard about it	22	25.88%
I heard about it	37	43.52%
I use it occasionally	11	12.94%
I use it sometimes	2	2.35%
I use it often	4	4.70%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.17: *Complete responses to group 4, question 3, subquestion 7.*

How familiar are you with these tools/testing infrastructure? ktest		
Answer	*Count*	*Gross percentage*
I've never heard about it	37	43.52%
I heard about it	32	37.64%
I use it occasionally	5	5.88%
I use it sometimes	1	1.17%
I use it often	1	1.17%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.18: *Complete responses to group 4, question 3, subquestion 8.*

How familiar are you with these tools/testing infrastructure? Smatch		
Answer	*Count*	*Gross percentage*
I've never heard about it	24	28.23%
I heard about it	27	31.76%
I use it occasionally	11	12.94%
I use it sometimes	11	12.94%
I use it often	6	7.05%
No answer	6	7.05%
Total (gross)	85	100.00%

Table E.19: *Complete responses to group 4, question 3, subquestion 9.*

How familiar are you with these tools/testing infrastructure? Coccinelle		
Answer	*Count*	*Gross percentage*
I’ve never heard about it	6	7.05%
I heard about it	35	41.17%
I use it occasionally	24	28.23%
I use it sometimes	8	9.41%
I use it often	6	7.05%
No answer	6	7.05%
Total (gross)	85	100.00%

Table E.20: *Complete responses to group 4, question 3, subquestion 10.*

How familiar are you with these tools/testing infrastructure? jstest		
Answer	*Count*	*Gross percentage*
I’ve never heard about it	70	82.35%
I heard about it	6	7.05%
I use it occasionally	0	0.00%
I use it sometimes	0	0.00%
I use it often	0	0.00%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.21: *Complete responses to group 4, question 3, subquestion 11.*

How familiar are you with these tools/testing infrastructure? TuxMake		
Answer	*Count*	*Gross percentage*
I’ve never heard about it	68	80.00%
I heard about it	5	5.88%
I use it occasionally	2	2.35%
I use it sometimes	1	1.17%
I use it often	1	1.17%
No answer	8	9.41%
Total (gross)	85	100.00%

Table E.22: *Complete responses to group 4, question 3, subquestion 12.*

How familiar are you with these tools/testing infrastructure? Sparse		
Answer	*Count*	*Gross percentage*
I've never heard about it	15	17.64%
I heard about it	11	12.94%
I use it occasionally	16	18.82%
I use it sometimes	20	23.52%
I use it often	16	18.82%
No answer	7	8.23%
Total (gross)	85	100.00%

Table E.23: *Complete responses to group 4, question 3, subquestion 13.*

How familiar are you with these tools/testing infrastructure? KUnit		
Answer	*Count*	*Gross percentage*
I've never heard about it	26	30.58%
I heard about it	42	49.41%
I use it occasionally	2	2.35%
I use it sometimes	3	3.52%
I use it often	3	3.52%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.24: *Complete responses to group 4, question 3, subquestion 14.*

How familiar are you with these tools/testing infrastructure? SymDrive		
Answer	*Count*	*Gross percentage*
I've never heard about it	74	87.05%
I heard about it	1	1.17%
I use it occasionally	0	0.00%
I use it sometimes	0	0.00%
I use it often	0	0.00%
No answer	10	11.76%
Total (gross)	85	100.00%

Table E.25: *Complete responses to group 4, question 3, subquestion 15.*

How familiar are you with these tools/testing infrastructure? FAU Machine		
Answer	*Count*	*Gross percentage*
I've never heard about it	75	88.23%
I heard about it	1	1.17%
I use it occasionally	0	0.00%
I use it sometimes	0	0.00%
I use it often	0	0.00%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.26: *Complete responses to group 4, question 3, subquestion 16.*

How familiar are you with these tools/testing infrastructure? ADFI		
Answer	*Count*	*Gross percentage*
I've never heard about it	67	78.82%
I heard about it	9	10.58%
I use it occasionally	0	0.00%
I use it sometimes	0	0.00%
I use it often	0	0.00%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.27: *Complete responses to group 4, question 3, subquestion 17.*

How familiar are you with these tools/testing infrastructure? EH-Test		
Answer	*Count*	*Gross percentage*
I've never heard about it	74	87.05%
I heard about it	1	1.17%
I use it occasionally	0	0.00%
I use it sometimes	0	0.00%
I use it often	0	0.00%
No answer	10	11.76%
Total (gross)	85	100.00%

Table E.28: *Complete responses to group 4, question 3, subquestion 18.*

How familiar are you with these tools/testing infrastructure? COD		
Answer	*Count*	*Gross percentage*
I've never heard about it	74	87.05%
I heard about it	2	2.35%
I use it occasionally	0	0.00%
I use it sometimes	0	0.00%
I use it often	0	0.00%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.29: *Complete responses to group 4, question 3, subquestion 19.*

How familiar are you with these tools/testing infrastructure? Troll		
Answer	*Count*	*Gross percentage*
I've never heard about it	76	89.41%
I heard about it	0	0.00%
I use it occasionally	0	0.00%
I use it sometimes	0	0.00%
I use it often	0	0.00%
No answer	9	10.58%
Total (gross)	85	100.00%

Table E.30: *Complete responses to group 4, question 3, subquestion 20.*

References

[*2020 Linux Kernel History Report* 2020] *2020 Linux Kernel History Report*. The Linux Foundation. 2020. URL: https://linuxfoundation.org/wp-content/uploads/2020_kernel_history_report_082720.pdf (visited on 11/11/2021) (cit. on pp. 35–37).

[*A Beginner's Guide to Linux Kernel Development (LFD103)* 2022] *A Beginner's Guide to Linux Kernel Development (LFD103)*. The Linux Foundation. 2022. URL: https://trainingportal.linuxfoundation.org/learn/course/a-beginners-guide-to-linux-kernel-development-lfd103/course-introduction/the-linux-foundation?page=1 (visited on 10/26/2022) (cit. on p. 15).

[*About Debian* 2021] *About Debian*. 2021. URL: https://www.debian.org/intro/about.en.html (visited on 08/20/2021) (cit. on p. 8).

[*About Linux Journal* 2022] *About Linux Journal*. Linux Journal. 2022. URL: https://www.linuxjournal.com/aboutus (visited on 10/26/2022) (cit. on p. 16).

[*About Linux Kernel* 2021] *About Linux Kernel*. The Linux Kernel Organization. 2021. URL: https://www.kernel.org/linux.html (visited on 08/25/2021) (cit. on p. 5).

[*About Linux.com* 2022] *About Linux.com*. The Linux Foundation. 2022. URL: https://www.linux.com/about/ (visited on 10/26/2022) (cit. on p. 15).

[Adams *et al.* 2017] R. J. Adams, P. Smart, and A. S. Huff. "Shades of grey: guidelines for working with the grey literature in systematic reviews for management and organizational studies". In: *International Journal of Management Reviews* 19.4 (2017) (cit. on p. 16).

[Bai *et al.* 2016] Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu. "Testing error handling code in device drivers using characteristic fault injection". In: *Proceedings of the 2016 USENIX Annual Technical Conference, USENIX ATC 2016*. June 2016, pp. 635–647. ISBN: 978-193197130-0 (cit. on pp. 12, 26, 29, 30).

[Buchacker and Sieh 2001] K. Buchacker and V. Sieh. "Framework for testing the fault-tolerance of systems including os and network aspects". In: *Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*. 2001, pp. 95–105. DOI: 10.1109/HASE.2001.966811 (cit. on pp. 12, 26, 30).

[Cai *et al.* 2007] Lin-Zan Cai, Rong-Shiung Wu, Wen-Ting Huang, and Farn Wang. "Test automation for kernel code and disk arrays with virtual devices". In: *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 505–508. isbn: 9781595938824. doi: 10.1145/1321631.1321720. url: https://doi.org/10.1145/1321631.1321720 (cit. on p. 12).

[B. Chen *et al.* 2020] Bo Chen, Zhenkun Yang, Li Lei, Kai Cong, and Fei Xie. "Automated bug detection and replay for cots linux kernel modules with concolic execution". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020, pp. 172–183. doi: 10.1109/SANER48275.2020.9054797 (cit. on pp. 12, 27, 29, 30, 64).

[D. Chen *et al.* 2020] Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. "Testing file system implementations on layered models". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1483–1495. isbn: 9781450371216. doi: 10.1145/3377811.3380350. url: https://doi.org/10.1145/3377811.3380350 (cit. on p. 12).

[Y. Chen *et al.* 2013] Yu Chen *et al.* "Instant bug testing service for linux kernel". In: *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. 2013, pp. 1860–1865. doi: 10.1109/HPCC.and.EUC.2013.347 (cit. on pp. 12, 64).

[Claudi and Dragoni 2011] Andrea Claudi and Aldo Franco Dragoni. "Testing linux-based real-time systems: lachesis". In: *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. 2011, pp. 1–8. doi: 10.1109/SOCA.2011.6166244 (cit. on pp. 7, 12, 63).

[*Coccinelle: A Program Matching and Transformation Tool for Systems Code* 2022] *Coccinelle: A Program Matching and Transformation Tool for Systems Code*. 2022. url: https://coccinelle.gitlabpages.inria.fr/website/ (visited on 02/14/2022) (cit. on pp. 37, 63).

[Computing Machinery 2021] Association for Computing Machinery. *About the ACM Organization*. 2021. url: https://www.acm.org/about-acm/about-the-acm-organization (visited on 08/06/2021) (cit. on p. 9).

[Cong *et al.* 2015] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. "Automatic fault injection for driver robustness testing". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, 2015, pp. 361–372. isbn: 9781450336208. doi: 10.1145/2771783.2771811. url: https://doi.org/10.1145/2771783.2771811 (cit. on pp. 12, 26, 29, 30).

[Corbet 2019] Jonathan Corbet. *Statistics from the 5.4 development cycle.* Nov. 2019. URL: https://lwn.net/Articles/804119/ (visited on 11/24/2021) (cit. on p. 35).

[Corbet and Kroah-Hartman 2017] Jonathan Corbet and Greg Kroah-Hartman. *2017 Linux Kernel Development Report.* 2017. URL: https://www.linuxfoundation.org/wp-content/uploads/linux-kernel-report-2017.pdf (visited on 08/02/2021) (cit. on pp. 1, 47).

[Costa *et al.* 2018] Joenio Costa, Christina Chavez, and Paulo Meirelles. "On the sustainability of academic software". In: *BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING* 4 (2018). DOI: 10.1145/3266237.3266243 (cit. on p. 2).

[Cyberhaven 2020a] Cyberhaven. *Building the S2E platform manually - S2E 2.0 documentation.* 2020. URL: http://s2e.systems/docs/BuildingS2E.html (visited on 08/16/2021) (cit. on p. 28).

[Cyberhaven 2020b] Cyberhaven. *Creating analysis projects with s2e-env - S2E 2.0 documentation.* 2020. URL: http://s2e.systems/docs/s2e-env.html (visited on 08/16/2021) (cit. on p. 28).

[Dan Rue 2021] Antonio Terceiro Dan Rue. *Linaro/tuxmake - README.md.* Linaro. 2021. URL: https://gitlab.com/Linaro/tuxmake (visited on 11/17/2021) (cit. on p. 38).

[*Distributed Linux Testing Platform KernelCI Secures Funding and Long-Term Sustainability as New Linux Fou*
Distributed Linux Testing Platform KernelCI Secures Funding and Long-Term Sustainability as New Linux Foundation Project. reTHINKit Media. 2019. URL: https://www.prnewswire.com/news-releases/distributed-linux-testing-platform-kernelci-secures-funding-and-long-term-sustainability-as-new-linux-foundation-project-300945978.html (visited on 06/30/2021) (cit. on p. 35).

[Drebes and Nanya 2008] Roberto Jung Drebes and Takashi Nanya. "Limitations of the linux fault injection framework to test direct memory access address errors". In: *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing.* 2008, pp. 146–152. DOI: 10.1109/PRDC.2008.44 (cit. on p. 12).

[Edge 2020] Jake Edge. *Maintaining stable stability.* July 2020. URL: https://lwn.net/Articles/825536/ (visited on 04/27/2021) (cit. on p. 36).

[*EditorsGroup* 2022] *EditorsGroup.* Linux Kernel Newbies. 2022. URL: https://kernelnewbies.org/EditorsGroup (visited on 10/26/2022) (cit. on p. 16).

[Elsevier 2021a] Elsevier. *Content - How Scopus Works.* 2021. URL: https://www.elsevier.com/solutions/scopus/how-scopus-works/content (visited on 08/06/2021) (cit. on p. 9).

[Elsevier 2021b] Elsevier. *How Scopus works: Information about Scopus product features*. 2021. url: https://www.elsevier.com/solutions/scopus/how-scopus-works (visited on 08/06/2021) (cit. on p. 9).

[*Fedora Linux Kernel Overview* 2021] *Fedora Linux Kernel Overview*. 2021. url: https://docs.fedoraproject.org/en-US/quick-docs/kernel/overview/ (visited on 08/20/2021) (cit. on p. 8).

[Feitelson 2021] Dror G. Feitelson. *"We do not appreciate being experimented on": Developer and Researcher Views on the Ethics of Experiments on Open-Source Projects*. Dec. 2021. url: https://arxiv.org/pdf/2112.13217.pdf (visited on 05/25/2022) (cit. on pp. 22, 23).

[Fleming 2017] Matt Fleming. *A survey of scheduler benchmarks*. June 2017. url: https://lwn.net/Articles/725238/ (visited on 11/17/2021) (cit. on p. 68).

[Forrest Shull 2008] Dag I. K. Sjøberg Forrest Shull Janice Singer, ed. *Guide to Advanced Empirical Software Engineering*. 1st ed. Springer London, 2008. doi: 10.1007/978-1-84800-044-5 (cit. on p. 23).

[Garn and Simos 2014] Bernhard Garn and Dimitris E. Simos. "Eris: a tool for combinatorial testing of the linux system call interface". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. 2014, pp. 58–67. doi: 10.1109/ICSTW.2014.7 (cit. on p. 12).

[Herrera 2020] Adrian Herrera. *s2e-env/README.md at master - S2E/s2e-env*. 2020. url: https://github.com/S2E/s2e-env/blob/master/README.md (visited on 08/16/2021) (cit. on p. 28).

[*How the development process works* 2021] *How the development process works*. The kernel development community. 2021. url: https://www.kernel.org/doc/html/latest/process/2.Process.html (visited on 04/27/2021) (cit. on pp. 6, 7).

[IEEE 2021] IEEE. *About IEEE Xplore*. 2021. url: https://ieeexplore.ieee.org/Xplorehelp/overview-of-ieee-xplore/about-ieee-xplore (visited on 08/06/2021) (cit. on p. 9).

["IEEE Standard Glossary of Software Engineering Terminology" 1990] "Ieee standard glossary of software engineering terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84. doi: 10.1109/IEEESTD.1990.101064 (cit. on p. 7).

[*index.rst « Documentation* 2022] *index.rst « Documentation*. The kernel development community. 2022. url: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/index.rst?h=linux-6.0.y (visited on 10/27/2022) (cit. on p. 16).

[Iyer 2012] Manoj Iyer. *LTP HowTo*. 2012. url: http://ltp.sourceforge.net/documentation/how-to/ltp.php (visited on 04/27/2021) (cit. on pp. 36, 63).

[A. R. Jonathan Corbet and Kroah-Hartman 2005] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition.* O'Reilly, Feb. 2005, p. 370 (cit. on p. 6).

[G. K.-H. Jonathan Corbet 2017] Greg Kroah-Hartman Jonathan Corbet. *2017 Linux Kernel Development Report.* LWN.net, The Linux Foundation. 2017. URL: https://www.linuxfoundation.org/wp-content/uploads/linux-kernel-report-2017.pdf (visited on 11/15/2021) (cit. on pp. 35, 64, 68, 69).

[Jones 2017] Dave Jones. *Linux system call fuzzer - README.* 2017. URL: https://github.com/kernelslacker/trinity (visited on 11/23/2021) (cit. on p. 36).

[Jordan 2021] Daniel Jordan. *So, you are a Linux kernel programmer and you want to do some automated testing...* Oracle. Feb. 2021. URL: https://blogs.oracle.com/linux/ktest (visited on 07/06/2021) (cit. on pp. 37, 40).

[K.P *et al.* 2015] Dileep K.P, Devesh G, A. Raghavendra Rao, Suman M, and S.V. Srikanth. "Verification of linux device drivers using device virtualization". In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom).* 2015, pp. 694–698 (cit. on p. 12).

[*Kernel - Fedora Project Wiki* 2021] *Kernel - Fedora Project Wiki.* The Fedora Project. 2021. URL: https://fedoraproject.org/wiki/Kernel (visited on 08/20/2021) (cit. on p. 8).

[*Kernel self-test* 2019] *Kernel self-test. start.* The kernel development community. 2019. URL: https://kselftest.wiki.kernel.org/ (visited on 04/27/2021) (cit. on pp. 35, 64).

[*Kernel Testing Guide* 2022] *Kernel Testing Guide.* The kernel development community. 2022. URL: https://www.kernel.org/doc/html/latest/dev-tools/testing-overview.html (visited on 07/25/2022) (cit. on p. 56).

[Kerrisk 2013] Michael Kerrisk. *LCA: The Trinity fuzz tester.* Feb. 2013. URL: https://lwn.net/Articles/536173/ (visited on 11/24/2021) (cit. on pp. 36, 69).

[Khan 2014] Shuah Khan. *Linux Kernel Testing and Debugging.* July 2014. URL: https://www.linuxjournal.com/content/linux-kernel-testing-and-debugging (visited on 04/27/2021) (cit. on pp. 35–37, 63, 70).

[Khan 2021a] Shuah Khan. *Kernel Validation With Kselftest.* 2021. URL: https://linuxfoundation.org/webinars/kernel-validation-with-kselftest/ (visited on 04/27/2021) (cit. on p. 35).

[Khan 2021b] Shuah Khan. *Mentorship Session: Kernel Validation With Kselftest.* The Linux Foundation. 2021. URL: https://www.youtube.com/watch?v=mpO_iDEMqWQ (visited on 11/03/2022) (cit. on p. 6).

[Kim *et al.* 2009] Moonzoo Kim, Shin Hong, Changki Hong, and Taeho Kim. "Model-based kernel testing for concurrency bugs through counter example replay". In: *Electronic Notes in Theoretical Computer Science* 253.2 (2009). Proceedings of Fifth Workshop on Model Based Testing (MBT 2009), pp. 21–36. issn: 1571-0661. doi: https://doi.org/10.1016/j.entcs.2009.09.049. url: https://www.sciencedirect.com/science/article/pii/S1571066109004034 (cit. on p. 12).

[Kitt 2009] Stephen Kitt. *jstest - joystick test program*. Apr. 2009. url: https://sourceforge.net/p/linuxconsole/code/ci/master/tree/docs/jstest.1 (visited on 02/14/2022) (cit. on p. 37).

[Konovalov 2021a] Andrey Konovalov. *Fuzzing Linux Kernel*. 2021. url: https://linuxfoundation.org/webinars/fuzzing-linux-kernel/ (visited on 04/27/2021) (cit. on pp. 36, 39).

[Konovalov 2021b] Andrey Konovalov. *Mentorship Session: Fuzzing the Linux Kernel*. 2021. url: https://www.youtube.com/watch?v=4IBWj21tg-c (visited on 08/11/2021) (cit. on p. 7).

[Kroah-Hartman 2022] Greg Kroah-Hartman. *Mentorship Session: Trust and the Linux Kernel Development Model*. The Linux Foundation. 2022. url: https://www.youtube.com/watch?v=YhDVC7-QgkI (visited on 11/03/2022) (cit. on p. 6).

[*Ktest* 2017] *Ktest*. Embedded Linux Wiki. Nov. 2017. url: https://elinux.org/Ktest (visited on 07/06/2021) (cit. on p. 37).

[*Linaro's Linux Kernel Functional Test framework* 2021] *Linaro's Linux Kernel Functional Test framework*. Linaro. 2021. url: https://lkft.linaro.org/ (visited on 11/24/2021) (cit. on p. 36).

[*Linux Foundation to Build New Linux.com Community* 2022] *Linux Foundation to Build New Linux.com Community*. The Linux Foundation. 2022. url: https://www.linuxfoundation.org/press/press-release/linux-foundation-to-build-new-linux-com-community (visited on 10/26/2022) (cit. on p. 15).

[*Linux Joystick support - Introduction* 2021] *Linux Joystick support - Introduction*. The kernel development community. Nov. 2021. url: https://www.kernel.org/doc/html/latest/input/joydev/joystick.html (visited on 11/08/2021) (cit. on p. 37).

[*Linux Kernel Developer: Arnd Bergmann* 2017] *Linux Kernel Developer: Arnd Bergmann*. The Linux Foundation. 2017. url: https://linuxfoundation.org/blog/linux-kernel-developer-arnd-bergmann/ (visited on 10/19/2021) (cit. on p. 68).

[*Linux Kernel Developer: Laura Abbott* 2017] *Linux Kernel Developer: Laura Abbott*. The Linux Foundation. 2017. url: https://linuxfoundation.org/blog/linux-kernel-developer-laura-abbott/ (visited on 10/19/2021) (cit. on p. 68).

[*Linux Kernel Developer: Shuah Khan* 2017] *Linux Kernel Developer: Shuah Khan*. The Linux Foundation. 2017. URL: https://linuxfoundation.org/blog/linux-kernel-developer-shuah-khan/ (visited on 10/19/2021) (cit. on pp. 35, 68).

[*Linux Kernel Performance* 2021] *Linux Kernel Performance*. 2021. URL: https://01.org/lkp (visited on 11/23/2021) (cit. on p. 35).

[*Linux Kernel Selftests* 2021] *Linux Kernel Selftests*. The kernel development community. 2021. URL: https://www.kernel.org/doc/html/latest/dev-tools/kselftest.html (visited on 04/27/2021) (cit. on pp. 35, 64).

[*Linux_Kernel_Newbies* 2022] *Linux_Kernel_Newbies*. Linux Kernel Newbies. 2022. URL: https://kernelnewbies.org/ (visited on 10/26/2022) (cit. on p. 15).

[N. P. Luis R. Rodriguez 2016] Nicolas Palix Luis R. Rodriguez. *coccicheck [Wiki]*. 2016. URL: https://bottest.wiki.kernel.org/coccicheck (visited on 11/18/2021) (cit. on p. 37).

[V. R. Luis R. Rodriguez T. B. 2016] Valentin Rothberg Luis R. Rodriguez Tyler Baker. *linux-kernel-bot-tests - start [Wiki]*. 2016. URL: https://bottest.wiki.kernel.org/ (visited on 11/23/2021) (cit. on p. 37).

[Madieu 2017] John Madieu. *Linux Device Drivers Development*. Packt Publishing, 2017, p. 16 (cit. on p. 5).

[Mathur 2013] Aditya P. Mathur. *Foundations of Software Testing*. 2nd ed. Pearson India, May 2013. ISBN: 9789332517660 (cit. on pp. 7, 47).

[Mohan *et al.* 2018] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, and Pandian Raju. “Finding crash-consistency bugs with bounded black-box crash testing”. In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’18)*. Oct. 2018, pp. 33–50. ISBN: 978-1-939133-08-3 (cit. on p. 12).

[*Oracle Q&A: A Refresher on Unbreakable Enterprise Kernel* 2018] *Oracle Q&A: A Refresher on Unbreakable Enterprise Kernel*. The Linux Foundation. 2018. URL: https://linuxfoundation.org/blog/oracle-qa-a-refresher-on-unbreakable-enterprise-kernel/ (visited on 10/19/2021) (cit. on p. 68).

[*Platform Devices and Drivers* 2022] *Platform Devices and Drivers*. The kernel development community. July 2022. URL: https://www.kernel.org/doc/html/latest/driver-api/driver-model/platform.html (visited on 07/12/2022) (cit. on p. 53).

[*Rapid Operating System Build and Test* 2021] *Rapid Operating System Build and Test*. Linaro. 2021. URL: https://www.linaro.org/os-build-and-test/ (visited on 11/17/2021) (cit. on pp. 36, 38).

[*Real-Time Linux Continues Its Way to Mainline Development and Beyond* 2017] *Real-Time Linux Continues Its Way to Mainline Development and Beyond*. The Linux Foundation. 2017. URL: https://linuxfoundation.org/blog/real-time-linux-continues-its-way-to-mainline-development-and-beyond/ (visited on 10/19/2021) (cit. on p. 68).

[Renzelmann *et al.* 2012a] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. *SymDrive Download and Setup*. 2012. URL: https://research.cs.wisc.edu/sonar/projects/symdrive/downloads.shtml (visited on 08/16/2021) (cit. on p. 28).

[Renzelmann *et al.* 2012b] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. "Symdrive: testing drivers without devices". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. Oct. 2012, pp. 279–292 (cit. on pp. 12, 25, 30, 63).

[Rothberg *et al.* 2016] Valentin Rothberg, Christian Dietrich, Andreas Ziegler, and Daniel Lohmann. "Towards scalable configuration testing in variable software". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 156–167. ISBN: 9781450344463. DOI: 10.1145/2993236.2993252. URL: https://doi.org/10.1145/2993236.2993252 (cit. on pp. 12, 27, 29, 30, 63, 64).

[Rousseau *et al.* 2008] Denise M. Rousseau, Joshua Manning, and David Denyer. "Evidence in management and organizational science: assembling the field's full weight of scientific knowledge through syntheses". In: AIM Working Paper Series 2008. Advanced Institute of Management Research, 2008. DOI: 10.2139/ssrn.1309606 (cit. on p. 63).

[Rue 2021] Dan Rue. *Portable and reproducible kernel builds with TuxMake*. Jan. 2021. URL: https://lwn.net/Articles/841624/ (visited on 11/02/2021) (cit. on p. 38).

[Searls 2019] Doc Searls. *Linux Journal at 25*. Apr. 2019. URL: https://www.linuxjournal.com/content/linux-journal-25 (visited on 10/26/2022) (cit. on p. 16).

[Shahpasand *et al.* 2016] Raheleh Shahpasand, Yasser Sedaghat, and Samad Paydar. "Improving the stateful robustness testing of embedded real-time operating systems". In: *2016 6th International Conference on Computer and Knowledge Engineering (ICCKE)*. 2016, pp. 159–164. DOI: 10.1109/ICCKE.2016.7802133 (cit. on p. 12).

[*Smatch The Source Matcher* 2021] *Smatch The Source Matcher*. 2021. URL: http://smatch.sourceforge.net/ (visited on 07/06/2021) (cit. on p. 37).

[Stewart *et al.* 2020] Kate Stewart *et al. 2020 Linux Kernel History Report*. 2020. URL: https://www.linuxfoundation.org/wp-content/uploads/2020_kernel_history_report_082720.pdf (visited on 08/02/2021) (cit. on p. 2).

[*Tests in LKFT* 2021] *Tests in LKFT*. Linaro. 2021. URL: https://lkft.linaro.org/tests/ (visited on 11/24/2021) (cit. on pp. 36, 37).

[*The Belmont Report* 1979] *The Belmont Report*. The National Commission for the Protection of Human Subjects of Biomedical and Behavioral Research. Apr. 1979. URL: https://www.hhs.gov/ohrp/regulations-and-policy/belmont-report/read-the-belmont-report/index.html (visited on 05/20/2022) (cit. on p. 21).

[*The Linux Kernel Archives - About* 2022] *The Linux Kernel Archives - About*. Linux Kernel Organization, Inc. 2022. URL: https://www.kernel.org/category/about.html (visited on 10/26/2022) (cit. on p. 15).

[*The LWN.net FAQ* 2022] *The LWN.net FAQ*. Eklektix, Inc. 2022. URL: https://lwn.net/op/FAQ.lwn (visited on 10/26/2022) (cit. on p. 16).

[Torvalds 2007] Linus Torvalds. *Re: [patch] revert: [NET]: Fix races in net_rx_action vs netpoll*. 2007. URL: https://lwn.net/Articles/243460/ (visited on 08/11/2021) (cit. on p. 8).

[Vizoso 2016] Tomeu Vizoso. *How Continuous Integration Can Help You Keep Pace With the Linux Kernel*. Nov. 2016. URL: https://www.linux.com/audience/enterprise/how-continuous-integration-can-help-you-keep-pace-linux-kernel/ (visited on 11/08/2021) (cit. on pp. 36, 68–70).

[Vyukov *et al.* 2021] Dmitry Vyukov, Andrey Konovalov, and Marco Elver. *syzbot*. 2021. URL: https://github.com/google/syzkaller/blob/master/docs/syzbot.md (visited on 04/27/2021) (cit. on p. 36).

[*Welcome to KernelCI* 2021] *Welcome to KernelCI*. KernelCI. 2021. URL: https://kernelci.org/ (visited on 11/17/2021) (cit. on pp. 35, 36).

[Wen *et al.* 2020] Melissa Wen, Leonardo Leite, Fabio Kon, and Paulo Meirelles. "Understanding FLOSS through community publications: strategies for grey literature review". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '20. Association for Computing Machinery, 2020, pp. 89–92 (cit. on pp. 11, 13–15).

[Wohlin 2014] Claes Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering". In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. London, England, United Kingdom: Association for Computing Machinery, 2014. ISBN: 9781450324762. DOI: 10.1145/2601248.2601268. URL: https://doi.org/10.1145/2601248.2601268 (cit. on p. 17).

[*xpad - Linux USB driver for Xbox compatible controllers* 2021] *xpad - Linux USB driver for Xbox compatible controllers*. The kernel development community. 2021. URL: https://www.kernel.org/doc/html/latest/input/devices/xpad.html (visited on 07/01/2021) (cit. on p. 37).

[Yuqing *et al.* 2012] Lan Yuqing, Xu Hao, and Liu Xiaohui. "The research of performance test method for linux process scheduling". In: *2012 Fourth International Symposium on Information Science and Engineering*. 2012, pp. 216–219. doi: 10.1109/ISISE.2012.54 (cit. on p. 12).

[Zaidenberg and Khen 2015] Nezer J. Zaidenberg and Eviatar Khen. "Detecting kernel vulnerabilities during the development phase". In: *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*. 2015, pp. 224–230. doi: 10.1109/CSCloud.2015.91 (cit. on pp. 12, 63).

[Zhai *et al.* 2008] Gaoshou Zhai, Jie Zeng, Miaoxia Ma, and Liang Zhang. "Implementation and automatic testing for security enhancement of linux based on least privilege". In: *2008 International Conference on Information Security and Assurance (isa 2008)*. 2008, pp. 181–186. doi: 10.1109/ISA.2008.61 (cit. on p. 12).