Automated tests on Linux kernel device drivers

How are device drivers being tested?

Marcelo Schmitt

Report presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
for the Master of Science
qualifying examination

Program: Ciência da Computação
Advisor: Prof. Dr. Paulo Roberto Miranda Meirelles
Coadvisor: Prof. Dr. Fabio Kon

During this work, the author was supported by the Coordination for the Improvement of Higher Education Personnel - Brazil (CAPES)

São Paulo
July 20, 2021

Automated tests on Linux kernel device drivers

How are device drivers being tested?

Marcelo Schmitt

This is the original version of the qualifying text prepared by candidate Marcelo Schmitt, as submitted to the Examining Committee.

I authorize the reproduction and disclosure of this work, total or partial, by any conventional or electronic means, for study and research purposes, provided the mention of the source.

Resumo

Marcelo Schmitt. **Testes Automatizados sobre drivers do kernel Linux: *Como drivers de dispositivo estão sendo testados?***. Exame de Qualificação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Drivers de dispositivo constituem parte importante do kernel Linux, e portanto, devem ser amplamente testados para que assegurar a robustês de sistemas operacionais GNU/Linux utilizados em diversos contextos, de supercomputadores a Internet das Coisas. Este trabalho procura responder como desenvolvedores do Linux testam os drivers de dispositivo do kernel. Para responder a essa pergunta de forma precisa, recorremos a um mapeamento de literatura formal, estamos conduzindo uma revisão de literatura cinzenta, e planejamos realizar uma pesquisa com desenvolvedores do Linux. Os métodos de pesquisa propostos nos permitirão oferecer uma visão abrangente do estado-da-arte e estado-da-prática dos testes sobre drivers de dispositivo do kernel Linux. Sintetizamos as informações encontradas até o momento em um catálogo de ferramentas de teste que são utilizadas para testar o kernel Linux e seus drivers de dispositivo. Ao longo das próximas etapas de pesquisa, aprimoraremos o catálogo de ferramentas de teste e faremos uma avaliação criteriosa daquelas que se mostrarem mais promissoras ao uso cotidiano por desenvolvedores do Linux. Por fim, ofereceremos um conjunto de recomendações para desenvolvedores do Linux interessados em aplicar testes em sua rotina de desenvolvimento.

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

Abstract

Marcelo Schmitt. **Automated tests on Linux kernel device drivers: *How are device drivers being tested?***. Qualifying Exam (Master's). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

Device drivers are an important part of the Linux kernel. They must be extensively tested to ensure the robustness of GNU/Linux operating systems in many contexts, from supercomputers to the Internet of Things. This research seeks to identify how Linux developers are testing kernel device drivers. To answer this question accurately, we carried out a mapping study, we are conducting a grey literature review, and we plan to collect information directly from Linux developers. The proposed research methods will allow us to offer a comprehensive overview of the state-of-the-art and state-of-the-practice about tests on Linux kernel device drivers. We have summarized the information found so far in a catalog of testing tools used to test the Linux kernel and its device drivers. In the following research steps, we will refine the test tools catalog and carefully evaluate those that show the most promising daily use by Linux developers. Finally, we will offer a set of guidelines for Linux developers interested in applying tests in their development routine.

Keywords: Linux. Test. Device driver.

List of Figures

5.1 Research planning. 31

List of Tables

3.1 Search string terms for the mapping study. 10
3.2 Articles selected by systematic mapping study study. 11
3.3 Mapping study phases 12
3.4 Search String Terms for GLR. 14
3.5 Data sources with search engine. 16
3.6 Senior kernel developer blogs 18

4.1 Documents selected from the grey literature review. 24
4.2 Main columns of the Linux kernel test tools catalog. 25

List of Programs

3.1 Steps for preliminary assessment of GL documents. 16
3.2 Steps for complete evaluation of GL documents. 17

3.3 GLR process overview. 17

Contents

1 Introduction **1**

1.1 Problem outline 1
1.2 Research Objectives 2
1.3 Research Questions 2
1.4 Research Design 3

2 Background **5**

2.1 Linux and device drivers 5
2.2 Software Testing 5

3 Research Methods **9**

3.1 Literature Systematic Mapping Study 9
3.1.1 Publication selection 10
3.1.2 Publication assessment 11
3.2 Grey Literature Review 12
3.2.1 GLR Planning 12

4 Linux Kernel Testing Tools **19**

4.1 Academia Tools 19
4.1.1 Main Tools Features 19
4.1.2 Tool Usability Evaluation 21
4.2 Community Tools 23
4.2.1 Results of the first iteration of grey literature review 23
4.2.2 Main Tools Features 27
4.3 Preliminary Conclusions 28

5 Work Plan **31**

References 33

Index 39

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 and companies 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 automated testing of Linux kernel device drivers. On the one hand, Kernel developers do not adopt the testing tools proposed by academia. On the other hand, testing tools extensively promoted by the kernel 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:

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

- There are no references in the academic literature reporting the testing practices and tools used throughout the Linux kernel development process. There is no publication indicating the community's expectations regarding testing tools either.
- Researchers spend time developing test tools that, while showing promising results, do not win the kernel developers' liking and fall into disuse.

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 will pursue the following research objectives:

RO1. Catalog the automated testing tools that are currently in use in the kernel, their characteristics, how they work, and in what contexts they are used.

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

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

RO4. Identify opportunities to improve testing tools in use by the kernel community.

By achieving *RO1*, we will produce a tool catalog describing the state-of-practice of testing in the Linux kernel, which may provide a reference for further academic work. Such a catalog will also contribute to producing recommendations to Linux developers interested in improving their workflow with automated testing tools. Upon reaching *RO3*, we hope researchers can make better-guided decisions when proposing new testing tools for the Linux kernel or even considering contributing to existing ones. We have evidence that some of the testing 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). Also, with *RO2* and *RO4*, we aim to benefit researchers and practitioners by pointing out promising paths for those considering studying or contributing to existing tools.

1.3 Research Questions

The problems described in Section 1.3 suggest an update of the academic literature concerning the current practice of testing the Linux kernel. 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. Also, the acceptance of patches only happens after the maintainer(s) is convinced of the benefits advocated by the sender. 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. To assess testing methods from all the Linux subsystems would require investigating a much larger amount of material. In this work, we seek to provide sound 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?

RQ4. How to improve automated tests on drivers in the kernel?

RQ5. What can one do to contribute to Linux kernel automated driver testing?

1.4 Research Design

This research should take place in 3 main phases. In the first phase, we conducted a mapping study to understand which testing techniques and tools were used to test Linux kernel device drivers. Among the test tools mentioned, six underwent usability assessment in which we sought to determine the reproducibility of the tests reported in the respective articles. In the second phase, we are conducting a grey literature review through which we aim to identify the practices of the Linux community when testing device drivers. We are building a testing tool catalog with the tools reported by both formal and grey literature. The Linux kernel testing tools catalog summarizes the main features of each tool found in the literature, including the source repository, the testing techniques applied, and use case examples. We describe the research methods and present the preliminary results from these phases in Chapters 3 and 4.

In the third phase of this research, we will seek information directly from the Linux kernel development community members by applying surveys or interviews. With material from academia, grey literature, and community participants, we will triangulate our findings by synthesizing different points of view on the same problem. After the end of the third phase, we will describe the state-of-the-art and state-of-practice testing of Linux kernel device drivers. With a broad view of the testing tools and the context in which each is used, we will compile the knowledge gained into a set of recommendations for Linux kernel developers.

Chapter 2

Background

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

2.1 Linux and device drivers

The Linux kernel is accounted for many essential operating system tasks such as memory management, process scheduling, data storage, network communication, and more (Organization, 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, a device driver is defined by a well-delimited piece of code (usually a file or a few files) to control the operation of a hardware device (or a family of devices).

To better distinguish between device drivers and other Linux kernel components, we consider examples of not device drivers to be entire subsystems and kernel portions not restricted to the operation of a single device (or set of devices), e.g., file system, network stack, process scheduler, memory manager, etc. Although these parts of the system may contain device drivers, they are not device drivers themselves. Next, we must attend to the concepts of software testing.

2.2 Software Testing

From the IEEE standard glossary of software engineering terminology, 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). Claudi and Dragoni (2011) complements by adding 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".

Thus, 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 data sheets describing the components and the functioning of the devices they supply. Therefore, we may say that testing device drivers are the act of evaluating, validating, or verifying whether a device driver (or parts of it) operates the hardware it supports as described by their blueprint. In addition to identifying device driver testing practices, we want to classify the techniques used for this purpose to achieve our research objectives *RO1* and *RO2*.

Mathur (2013) proposes a comprehensive categorization of software testing techniques based on four classifiers. These 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, regarding the Linux kernel, these classifiers apply under a few conditions.

First, as Linux is a free software project, 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 done over any of these trees is also a form of beta-testing (*Fedora Linux Kernel Overview* 2021; *Kernel - Fedora Project Wiki* 2021; *About Debian* 2021).

As for the purposes of testing the Linux kernel, special attention is paid to identifying 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 could 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 forgo 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 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

We resorted to a systematic mapping study (SMS) and a grey literature review (GLR) to answer the proposed research questions. We conducted a literature mapping to establish a body of knowledge consolidated by peer-reviewed articles published in media that require methodological rigor. At the same time, a grey literature review is imperative to capture the state-of-the-practice in an ever-changing project like Linux.

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 more than 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).

The next step of this mapping study was to create a comprehensive search string that would allow us to get several articles related to kernel testing. The search string was then derived from the objects of interest for this search. Within GNU/Linux system development, we matter the kernel. From that, we pay special attention to testing device drivers. So the broader context for us is *Linux*, from which we observe its *kernel*[1]. In addition, we

[1] This is technically a redundancy because the kernel of Linux-based OSes is itself called Linux. However,

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

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

look for *testing* practices applied 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 kernel testing tool could be configurable to test device drivers, even when its conventional usage does not foresee it. Finally, we added synonyms and terms related to search objects. Table 3.1 shows the terms used in the search string. Columns are connected with AND, and items within columns are connected with OR.inally, we added synonyms and terms related to search objects. Table 3.1 shows the terms used in the search string. Columns are connected with AND, and items within columns are connected with OR.

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)

3.1.1 Publication selection

The library search returned 5,018 articles, which then went through 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, it was included on the list. Duplicated results and non-computer science work were also removed. From this first selection stage, 399 articles were selected but, as it was still a large number of publications, we had to be more rigorous in the second selection stage, and so we kept only articles whose titles contained “linux”, “kernel”, “ test”, some variant of “test”, or that somehow referred to tests even if with other words.

In the third stage, we read the abstract from the 62 articles selected in the previous stage, 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.
- 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.

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.

- Articles that could have been disregarded based on previous criteria but which had been retained for the benefit of the doubt were classified as not relevant.

Next, in the fourth selection step, a speed reading was done to select articles that specifically focus on testing the Linux kernel or parts of it. Works that, although indirectly testing parts of the kernel, focus on the GNU/Linux system as a whole or its general functionality have been put aside. We also filtered out articles focused on testing practices or which components should be better tested but did not present test results or tools. With these criteria, 19 articles were selected for full reading, thus completing the fourth selection stage.

3.1.2 Publication assessment

Each of the 19 articles selected from the speed reading step was then fully read 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.

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.*

Among the 17 testing (and 1 debug-focused) tools proposed by the academic articles as a test solution for the Linux kernel, 5 of them were selected for usability 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 at each one.

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 -> 5

Table 3.3: *Mapping study phases*

3.2 Grey Literature Review

Conducting a systematic review of grey literature will be essential in answering research questions and achieving the proposed objectives. In this sense, we are following the literature review methodology proposed by Wen *et al.* (2020), making some minor adjustments necessary to fit our work. With enough information from a grey literature review, we may be able to provide a reasonable answer to "*RQ2.* What testing tools are being used by the Linux community to test the kernel? What are the main features of these tools?" and to "*RQ3.* What features does the community desire for a test tool? About the tools that are already in use, what could be improved?". Moreover, a GLR may provide much of the information needed to achieve "*RO1.* Catalog the automated testing tools that are currently in use in the kernel, their characteristics, how they work, and in what contexts they are used".

3.2.1 GLR Planning

We followed the methodology for GLR proposed by Wen *et al.* (2020) to define our GLR planning. According to Wen *et al.* (2020), a GLR planning covers four essential steps to ensure the quality of the selected documents:

1. Outline the problem and define the research question.

2. Define the inclusion and exclusion criteria.

3. Develop a relaxed search string.

4. Define the resource-types to consider.

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

Problem outline & research questions

From our mapping study and usability evaluation of proposed testing tools, we have evidence that device driver testing tools suggested by academic publications are deprecated and may not have been used by the community. There are no references in the academic literature that present the automated testing tools in use in Linux kernel development, nor any indication of the community's expectations with testing tools. Moreover, from our development experience, we have evidence that some of the testing tools used in day-to-day development are not covered in the formal literature. Thus, one of the objectives of this research is to provide a catalog of automated testing tools that are currently in use in the kernel, their characteristics, how they work, and in what contexts they are used.

In this regard, the research question "*RQ2.* What testing tools are being used by the Linux community to test the kernel? What are the main features of these tools?" and "*RQ3.* What features does the community consider ideal/desirable for a test tool? About the tools/techniques that are already in use, what could be improved?" guided our investigation through the grey literature content.

Inclusion and exclusion criteria

Collecting information from non-traditional sources may impair 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, our searches were limited to a select set of sites that we have come to refer to as data sources. Our data sources were defined based on Wen's selection, plus the official Linux kernel documentation. Each data source is recognized as a valuable source of information by institutions that have long been committed to Linux development or by prominent developers in the community.

In addition, selection criteria were defined to pick out the documents found from the data sources. The inclusion and exclusion criteria were adapted and refined based on Wen's criteria and are intended to reduce the risks to the validity of the research. Throughout the literature review, the exclusion criteria were applied during each speed reading step to discard documents inappropriate for the research objectives, while the inclusion criteria were applied during each complete read step to select documents with information relevant to the object of research.

Exclusion 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.*

Based on Wen's experience with GL data sources, different data sources offer different capabilities. Some may offer 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. Assuming that the data source search tools have at least the ability to search for terms linked by "and", the search string has been decomposed into smaller strings s1, s2, ..., s6, so that merging the search results s1, ..., s6 is equivalent to the results that would be 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. With the data sources provided by her and some additional ones, we defined a set of websites with embedded search engines to conduct our document search. Although we have not classified the collected documents into "shades of GL" as suggested by Adams *et al.* (2017), many of the selected documents are news articles, wiki pages, documentation, company publications, among other formats that fit what would be the second degree of grey literature. This is due to the characteristics of the selected data sources and also to the exclusion and inclusion criteria used, which 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 work. 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 procedure of document search and collection. Our systematic GL review process is described by the algorithms below.

Organization of the Grey Literature Review Process

We conducted the planned Grey Literature Review process considering three main review phases, namely 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 begin by taking a data source and using its search engine to search for the first search string variant. We collect the URL for the first five search hits, annotate from what reference we stopped, and then search for the following search string variant. Repeat the previous step until there is no next search string variant to apply. When done, take another data source and repeat all over again. The data collection phase is over when all data sources have been searched.

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 GL documents. Indeed, our data collection phase resembles a breadth-first search. Program 3.3 provides a structured view of our GLR process.

The preliminary analysis phase was designed to systematically verify which documents are compliant with each of the exclusion criteria. Program 3.1 describes this phase with a few organizational tasks needed to conduct the review. During the complete analysis phase, documents and their content were assessed for relevance about the Linux kernel testing subject. Document appraisal at this phase ensured that each document 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. Program 3.2 describes this phase under algorithmic perspective.

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

Table 3.5: *Data sources with search engine.*

Program 3.1 Steps for preliminary assessment of GL documents.

```
1    ▷ Main objective of evaluating whether the document meets the exclusion factors
2   FUNCTION speed_read()
3   {
4       Check if the publication is from 2011 or later.
5       Read document title and abstract if it has. Write down the title. Give a title if it does
            not have one.
6       Discard based on title if it is the same as another document already analyzed.
7       Check the authorship of the publication.
8       Check the publication source's reliability/reputation.
9       Search (Ctrl + f) for each of the words: test; validat; verif.
10      Identify the context in which the searched word appears.
11   ▷ Identifying the context may consist of reading an entire paragraph.
12      If it meets all the exclusion criteria, mark the document for the full read phase.
13      if (document was marked) {
14   ▷        This should avoid losing content with links that may come to break.
15          Save the page in pdf format (File menu -> print)
16          Save the page with the title name.
17      } else {
18          Write down the reasons for exclusion.
19      }
20  }
```

The Linux development community has experienced developers working on the project for years. Some of these developers maintain blogs where they post content related to the Linux kernel along with their opinions on issues about the project's development. Table 3.6 lists some experienced kernel developers along with their respective blogs. However, most developer blogs do not have search engines or, when they do, the results that such engines bring are unsatisfactory. Because of this, developer blogs have not been included in the list of data sources. We plan to inspect developer blogs in later stages of the GLR manually.

In addition to the document collection phase, our grey literature review process also allows for collecting documents by snowballing, appointing experts, and including artifacts

Program 3.2 Steps for complete evaluation of GL documents.

```
1    ▷ Main objective of evaluating whether the document meets the inclusion criteria
2   FUNCTION full_read()
3   {
4       a. Cut snippets to support a model of the current state of practice in testing the
            Linux kernel. // May quoted on argumentation.
5       b. Collect names, links, repositories, etc., from kernel testing tools. Take note of the
            citation excerpts (with a link) for each reference found.
6       if (document was obtained from a data source) {   ▷ Single level snowballing.
7           c. Collect citations (links) for snowballing and add them to the list of non–
                evaluated documents.
8       }
9       Remove the document from the list of non–evaluated documents (mark as evaluated
            ).
10  }
```

Program 3.3 GLR process overview.

```
1    ▷ Systematic GLR process
2   for (3 times) {
3       for (each of the data sources with search engine) {   ▷ Document collection
4           Take note of the days GL documents were collected.
5           for (each search string variant) {
6               Apply the search string over the data source.
7               Append the next five results to the list of unassessed URLs.
8               Take note of which result to continue from in the next iteration.
9           }
10      }
11  document_assessment:
12      Remove documents with duplicate URLs from the spreadsheet.
13      for (each document not evaluated) {   ▷ Preliminary document evaluation
14          speed_read()
15      }
16      for (each marked document) {   ▷ Full document review
17          full_read()
18      }
19      if (list of non–evaluated documents is not empty) {
20          goto document_assessment;   ▷ Snowballing
21      }
22  }
```

Kernel developer	*Blog URL*
Greg KH	kroah.com
Daniel Vetter	blog.ffwll.ch
Sage Sharp	sage.thesharps.us
Laura Abbott	www.labbott.name/
Shuah	www.gonehiking.org/ShuahLinuxBlogs/

Table 3.6: *Senior kernel developer blogs*

found during kernel development activity or by interacting with the Linux community. Documents collected by these additional sources are added to the list of non-evaluated documents and are subjected to preliminary and complete analysis.

Chapter 4

Linux Kernel Testing Tools

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

4.1 Academia Tools

From the academic studies that reported testing tools (Table 3.2), we selected five of them to evaluate the tools presented by them. We start by talking about the main features of these tools.

4.1.1 Main Tools Features

Studies related to Linux kernel driver testing propose different approaches and testing techniques to find and resolve system bugs. Table 3.2 summarized the testing tools and techniques reported by the papers assessed. From this list, we took five tools to undergo a usage evaluation. We now depict the operation of these tools according to what was described in the 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. Driver state can be accessed 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 value, 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 a 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, and (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 by generating fault descriptions from a master 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 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 aiming 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; and (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. 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. The C programming language has no built-in error handling mechanism (such as “try-catch”), so developers often use an “if” statement to decide whether error handling code should be triggered in device drivers. Next, 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 of the 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 reproduced 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 the 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 (few) files locally for configuration options (#ifdef) and creates a partial kernel compilation configuration. This initial step is called sampling. Each partial configuration is 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 biggest click covers most of the #ifdef and generates several warnings when Sparse analyzes the code generated by such arrangement. With a valid configuration providing good coverage of different configurations (#ifdef), further automated testing is more likely to find bugs.

4.1.2 Tool Usability Evaluation

To evaluate each selected testing tool, we carried out several activities that included looking for project repositories, reading documentation files, installing project dependen-

cies, compiling source code files, and contacting the respective paper authors by email when facing setbacks. A `qemu` virtual machine running Ubuntu 18.04LTS, kernel 5.4, was used 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 recommend them to fellow kernel developers.

Symdrive stood out between 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. S2E documentation mentions the use of Ubuntu as a prerequisite for setting up the platform (Cyberhaven, 2020b; Herrera, 2020; Cyberhaven, 2020a). Nevertheless, we found indications of compatibility with other OSes after inspecting the compilation and installation scripts. In addition, installation script error messages have informed us that S2E is compatible only with a restricted set of processors. Further, configured the VM with a compatible processor, 10GB of system memory was not enough to prevent installation scripts from failing due to lack of RAM.

When looking for help with installing S2E, we found that the S2E mailing list is semi-open so that only subscribed addresses can post emails, and moderators must approve new subscriptions. Indeed, our request to participate on the S2E mailing list was accepted, though it took a month for it to happen. By the time it was granted us access, we were assessing other testing tools. 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 the SymDrive tool.

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 the FAU machine, we wrote to the respective paper authors who pointed out the tool's repository. 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 the FAU machine, we tried to run some tests by setting up an example from FAU source files. The experiment consists 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 experimenting went unanswered. Still, within the menus and options in the virtual machine management window, it was possible to see items referencing system fault injection. The evaluation of these tests, however, cannot be completed.

ADFI and EH-Test proposed fault injection tests focused on device drivers. There is no link or address to any web page for the ADFI project repository in the article we found about it. Moreover, the authors did not respond to our email asking for an archive to get

ADFI. Thus, it was not possible to test 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. Though, some EH-Test components did not compile with newer compilers versions. We mailed the principal author asking for some installation and usage guidance but had no feedback.

For reasons analogous to ADFI, COD could not be tested either. There is no repository link in the article or indication of how to get the tool. The email sent to the authors asking how to get COD was unanswered.

Lastly, Troll was designed to generate Linux kernel build configurations. Thus, it does not fit into the kernel tests category but comes in as an auxiliary tool. Despite this, we decided to give Troll a try. The first drawback we found was 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 paper authors. Surprisingly, the authors were very responsive and helped us set up an unlisted Undertaker version. After that, we ran an example from Troll documentation that generates Linux kernel compilation settings. The uses presented in the reference article are analogous to the documentation example but require more system memory and, because of that, we could not reproduce them.

From what we have seen so far, academic publications introduce testing tools as promising solutions to leverage the practice of automated testing in the Linux kernel. However, usability evaluation of those tools showed that most of them are outdated, incompatible with newer software versions, or have limited support by their original developers. This finding leads us to seek Linux kernel testing tools through additional sources.

4.2 Community Tools

To complement the findings from the formal literature, we are conducting a grey literature review that has already revealed an additional set of testing tools for the Linux kernel.

4.2.1 Results of the first iteration of grey literature review

- The first stage of GL document collection brought 107 different publications.
- The subsequent speed reading step selected 47 out of 107 publications.
- In the full read stage, we found only 23 documents contained excerpts reporting test practices in the kernel, statistics, opinion, or studies about Linux and its community related to software testing.
- During the full read stage, 23 links were collected for snowballing (including two repeated entries).
- Of the 21 distinct snowballing links, 11 documents were selected after speed reading.

- After fully reading the 11 documents selected by the previous snowballing step, only eight were found relevant information about tests in the kernel.
- Thus, the first iteration of the grey literature review found a total of 31 (23 + 8) documents containing practices, statistics, opinions, or studies about the tools used to test the Linux kernel.

The 31 documents selected by our grey literature review are shown in Table 4.1. At the end of the first iteration of the GLR, the relevant snippets from each selected document were re-read and evaluated for information about 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.* How are Linux device drivers being tested?", "*RQ3.* What features does the community desire for a test tool? What techniques that are already in use, what could be improved?", and corroborate the goal "*RO1.* Catalog the automated testing tools that are currently in use in the kernel, their characteristics, how they work, and in what contexts they are used".

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
			continue ⟶

Table 4.1: *Documents selected from the grey literature review.*

ID	Data Source	Title	Year
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
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

Table 4.1: *Documents selected from the grey literature review.*

The main features of the test tools found in the formal literature and grey literature have been summarized in the Linux kernel test tools catalog. We integrate the relevant content of each document by adding trinkets in which testing tools are cited. The catalog holds fields for name, repository, estimated activity status, testing techniques, execution environment, use context, use cases, and most relevant citations for each reported test tool. However, not all fields could be filled due to the lack of data in the documents evaluated so far. Table 4.2 shows the name, estimated status, testing techniques, and citations for the testing tools found so far.

Tool Name	Estimate Status	Testing Techniques	Citations
ktest	Active	End-to-end testing, fuzz testing (randconfig)	G11, G27, G28
fault-injection	Probably active	Fault injection	G11
Trinity	Probably active	Fuzz testing	L9, G22
KCOV	Active	Fuzz testing	G22
syzkaller / Syzbot	Active	Fuzz testing	G16, G22, G29, L1
cyclictest	Probably active	Load testing/ stress testing	G17
hackbench	Probably active	Load testing/ stress testing	L16, G17
Linux Test Project (LTP)	Probably active	Reliability testing, robustness testing, stability testing, stress testing	L3, L12, L16, G11, G25
Sparse	Active	Static analysis	G11
Smatch	Apparently inactive	Static analysis	G11, G26
rcutorture	Probably active	Stress test	G8
KUnit	Active	Unit tests	G12
			continue ⟶

Table 4.2: *Main columns of the Linux kernel test tools catalog.*

Tool Name	*Estimate Status*	*Testing Techniques*	*Citations*
Kselftest	Active	Unit tests, regression tests, stress test	L18 ,G6, G11, G23, G30
Kprobe			L1
LDV			L1
WHOOP			L1
Dr. Checker			L1
DSAC			L1
DEADLINE			L1
DCNS			L1
DIFUZE			L1
TriforceAFL			L1
kAFL			L1
Razzer			L1
DDT			L1
COD	Apparently inactive	Concrete and symbolic code execution, code instrumentation	L1
ADFI	Apparently inactive	Fault injection, code instrumentation,	L2
Eris		Model based & combinatorial testing	L4
LgDb		Debug	L3
CRASHMONKEY and ACE		Fuzzing	L5
FAUMachine	Active	Fault injection	L6
TIMEOUT		Code instrumentation, fault injection	L8
KIS		Static & dynamic analysis	L9
DMA Fault Injector		Fault injection	L10
MOKERT		Model based testing	L11
SymDrive	Apparently inactive	Symbolic execution	L1, L12
UKTI		Component emulation	L13
EH-Test	Apparently inactive	Fault injection	L14
Dogfood		Model based testing	L15
Lachesis		Model based & fuzz testing	L16
ScheduleBench		Performance (instrumentation) testing	L17
Troll	Apparently inactive	Local sampling, local analysis, graph clique search	L18
			continue ⟶

Table 4.2: *Main columns of the Linux kernel test tools catalog.*

Tool Name	*Estimate Status*	*Testing Techniques*	*Citations*
CAB-Fuzz			L1
mmtest			L9
PF-Miner			L14
Coccinelle			L18
0-day test robot			L18
KLive	Apparently inactive		G1
AutoTest	Apparently active		G3, G11
xfstests	Apparently active		L9, G5, G13
FAFT	Maybe active		G5
jstest	Probably active		G9, G10
LFII			L2
LAVA	Probably active		G11
Fuego	Apparently active		G14, G24
perf	Active		G17
KMSAN	Probably active		G29
KernelCI	Active		G31

Table 4.2: *Main columns of the Linux kernel test tools catalog.*

4.2.2 Main Tools Features

Here we present the main features of the three most cited tools by documents reviewed so far. Regarding the tools below, there is information obtained from both formal and grey literature.

Linux Test Project (LTP)

The Linux Test Project (LTP) is an open-source project which goal is to deliver test suites to the Linux open-source community (Iyer, 2012). LTP contains a collection of tools to test the reliability, robustness, and stability of the Linux kernel and related features (Khan, 2014; Iyer, 2012). Further, Zaidenberg and Khen (2015) adds that the test suite methodology is also based on regression testing.

LTP can test essential Linux kernel features such as filesystems, device drivers, memory management, scheduler, disk I/O, networking, syscalls, and IPC (Claudi and Dragoni, 2011). About device driver testing, Renzelmann *et al.* (2012b) reports that LTP can invoke drivers and verify their behavior but require 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.

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 (Claudi and Dragoni, 2011; Khan, 2014). According to Claudi and Dragoni (2011), the LTP has been increasingly used by kernel developers and testers in recent years and has almost become a standard to test the Linux kernel.

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, 2021; *Kernel self-test* 2019). Kselftest contains tests for various kernel features and subsystems such as tests for 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 be small developer-focused tests that target individual code paths and short-running units supposed to terminate in a timely fashion of 20 minutes (*Linux Kernel Selftests* 2021; ROTHBERG *et al.*, 2016). Kselftest consists of shell scripts and user-space programs that test kernel API and more features. Test cases may span kernel and use-space with user programs working in conjunction with a kernel module to test (KHAN, 2021). 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).

Syzkaller / Syzbot

Syzkaller is 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 bugs reproducers. This can 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 for better testing a particular subsystem or patch (VYUKOV *et al.*, 2021). Syzbot is receiving increasing attention from kernel developers. For instance, Sasha Levin said that 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).

4.3 Preliminary Conclusions

We reached 399 articles in the field of computer science in our literature map. We took notice of 17 Linux kernel test tools and evaluated the usability of 5 of them. To our surprise, none of the evaluated tools could be used due to several problems, from obtaining the source code to configuring, installing, and running the test apparatus. Furthermore, we did not find any report of the use of any tool from Table 3.2 in 107 grey literature documents.

Something seems wrong with the way academia has approached the topic of software testing when the system under test is the Linux kernel. Academic papers seem eager to show that their testing solution is great for some testing situations, but none of the proposed tools seem good enough for Linux developers. We found articles citing tools actively developed by kernel developers, but otherwise, we did not find papers discussing the use or improvement of community-supported tools.

Therefore, we see with some concern the proposal to develop new Linux testing tools from academia. Our hunch is that the adoption of testing tools in a dynamic and rapidly changing environment like Linux depends on the support of community members who act as evangelists in favor of using such tools. Without this constraint, the test suite tends to become outdated, unusable, and, ultimately, discontinued.

Chapter 5

Work Plan

As stated in the Introduction chapter, this research has three investigation phases. In phase 1, we collected information from the formal literature to start a list of kernel testing tools that could help developers test device drivers. Next, we assessed some of the tools concerning usability from a kernel developer point of view. We discussed our findings in Chapter 4 (Linux Kernel Testing Tools). In phase 2, we are conducting a GLR to find what the community tells about kernel testing tools. So far, we have found evidence that the testing tools used by kernel developers vary from the ones we identified in academic literature. We plan to appraise more community publications throughout the next few months to consolidate our understanding of the testing practices reported by non-academic sources. After that, we will start phase 3, in which we survey kernel developers for their testing habits. Figure 5.1 shows a detailed Gantt chart with the planned activities.

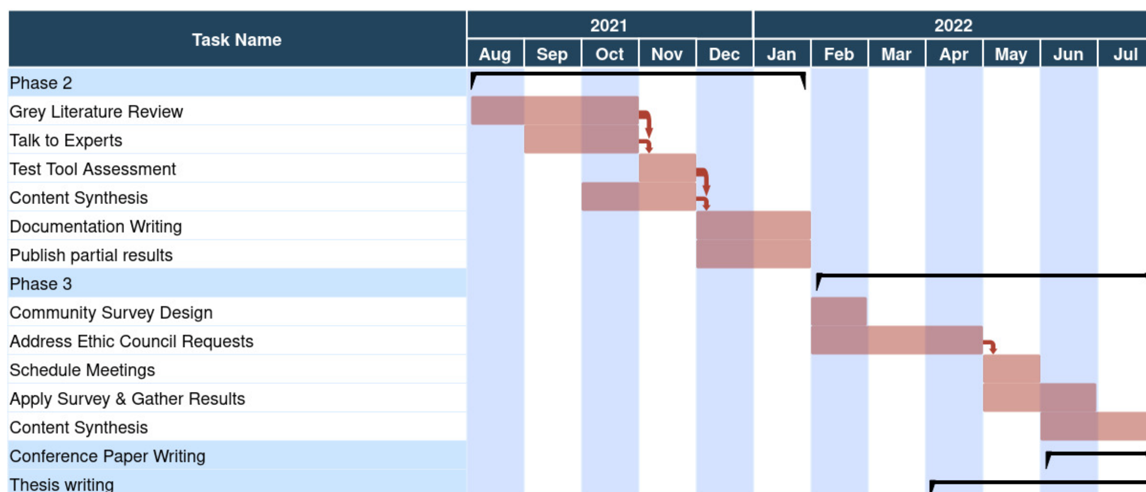

Figure 5.1: *Research planning.*

The second phase of investigative research (**Grey Literature Review**) aims to collect information from online documents crafted by Linux community members. As soon as we get a consistent body of knowledge, we will submit our findings to a specialized Linux journal or magazine and propose an update to the Linux kernel documentation. This step

is important because it will allow us to validate our findings with the Linux community. The list of steps we plan to take is further described below.

Grey Literature Review: complete two more iterations of GLR.

Talk to Experts: ask kernel experts for additional grey publications about Linux testing.

Test Tool Assessment: evaluate the usability of selected testing tools.

Content Synthesis: combine the content from the GLR into the testing tool catalog.

Documentation Writing: propose a documentation update to the Linux community.

Publish partial results: submit an article to a specialized Linux magazine.

The third investigative phase of research (**Community Survey**) will bring us information directly from the people involved in developing the Linux kernel. As we will be interacting with individuals from diverse backgrounds and ethnic origins, we must first reach an ethics council to suit our survey to the ethical and legal constraints that such intervention may imply. An additional description of the planned activities for this phase is given below.

Community Survey Design: sketch a survey to be applied to kernel developers.

Address Ethics Council Requests: submit the survey to the ethics council and apply the requested changes.

Schedule Meetings: if we decide to conduct interviews, talk to volunteers to schedule the meetings.

Apply Survey & Gather Results: hand over the survey for participants to answer. If we decide to conduct interviews, carry on the meetings.

Content Synthesis: combine the information from the surveys into the testing tool catalog.

Finally, we will provide our findings to academia by submitting a conference or journal paper which will contain the main conclusions of the herein proposed thesis.

References

[*About Debian* 2021] *About Debian*. 2021. URL: https://www.debian.org/intro/about.en.html (visited on 08/20/2021) (cit. on p. 6).

[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. 15).

[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. 11, 20).

[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. 11, 20).

[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. 11).

[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. 11, 21).

[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. 11).

[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 p. 11).

[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. 5, 11, 27).

[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. 11, 20).

[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 p. 1).

[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) (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. 22).

[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. 22).

[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. 11).

[Edge 2020] Jake Edge. *Maintaining stable stability*. July 2020. url: https://lwn.net/Articles/825536/ (visited on 04/27/2021) (cit. on p. 28).

[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. 6).

[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. 11).

[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. 22).

[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. 5).

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

[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. 11).

[*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. 6).

[*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 p. 28).

[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. 27, 28).

[KHAN 2021] Shuah KHAN. *Kernel Validation With Kselftest*. 2021. URL: https://linuxfoundation.org/webinars/kernel-validation-with-kselftest/ (visited on 04/27/2021) (cit. on p. 28).

[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. 11).

[KONOVALOV 2021a] Andrey KONOVALOV. *Fuzzing Linux Kernel*. 2021. URL: https://linuxfoundation.org/webinars/fuzzing-linux-kernel/ (visited on 04/27/2021) (cit. on p. 28).

[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. 6).

[*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 p. 28).

[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 p. 6).

[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. 11).

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

[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. 22).

[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. 11, 19, 27).

[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. 11, 21, 28).

[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. 11).

[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).

[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. 6).

[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. 28).

[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 p. 12).

[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. 11).

[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. 11, 27).

[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. 11).

Index

C

Captions, *see* Legendas
Código-fonte, *see* Floats

D

Define the resource-types to consider, 15
Develop a relaxed search string, 14

E

Equações, *see* Modo Matemático

F

Figuras, *see* Floats
Floats
    Algoritmo, *see* Floats, Ordem
Fórmulas, *see* Modo Matemático

G

GLR Planning, 12
Grey Literature Review, 12

I

Inclusion and exclusion criteria, 13
Inglês, *see* Língua estrangeira

M

Main Tools Features, 19, 27

O

Organization of the Grey Literature Review Process, 15

P

Palavras estrangeiras, *see* Língua estrangeira
Problem outline & research questions, 13

R

Research Methods, 9
Rodapé, notas, *see* Notas de rodapé

S

Subcaptions, *see* Subfiguras
Sublegendas, *see* Subfiguras

T

Tabelas, *see* Floats

V

Versão corrigida, *see* Tese/Dissertação, versões
Versão original, *see* Tese/Dissertação, versões