UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE E INOVAÇÃO

RICARDO JOSÉ MARTINCOSKI

Automação de testes para *drivers out-of-tree* do *kernel* Linux

Monografia de Conclusão de Curso apresentada como requisito parcial para a obtenção do grau de Especialista em Engenharia de Software e Inovação

Orientadora: Profª. Drª. Érika Fernandes Cota

Porto Alegre
2021

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Martincoski, Ricardo José

Automação de testes para *drivers out-of-tree* do *kernel* Linux / Ricardo José Martincoski. – Porto Alegre: PPGC da UFRGS, 2021.

70 f.: il.

Monografia (especialização) – Universidade Federal do Rio Grande do Sul. Curso de Especialização em Engenharia de Software e Inovação, Porto Alegre, BR–RS, 2021. Orientadora: Érika Fernandes Cota.

1. Teste de software embarcado. 2. Ferramentas de automação de teste. 3. Ferramentas *open source*. 4. Testes de unidade. I. Cota, Érika Fernandes. II. Título.

RESUMO

O uso de testes automatizados de software já é o padrão da indústria de software de uma forma geral. O teste de software embarcado possui especificidades, principalmente na camada que é altamente dependente do hardware. A maioria dos *frameworks* de teste *open source* mais usados não se adequam a este contexto, principalmente no caso de *drivers out-of-tree* do *kernel* Linux. Um *driver* é chamado de *out-of-tree* quando é mantido em repositório de código-fonte separado da árvore (repositório) do *kernel*. Algumas restrições se aplicam a esse tipo de *driver*: é comum ele precisar suportar várias versões de *kernel*, geralmente recorrendo a código que é condicionalmente compilado ou não, dependendo de contra qual versão de *kernel* o *driver* está sendo compilado; no sistema em que o *driver* é executado, ele fica guardado em um arquivo binário separado do *kernel*, e é carregado sob demanda; o *driver* acaba não podendo ser testado com a maioria das ferramentas que são distribuídas com o *kernel*, pois estas foram criadas apenas para testar código que já está integrado no repositório do *kernel*. Este trabalho propõe o uso de uma combinação de ferramentas *open source* para gerar um ambiente próximo ao cenário real, o hardware final ou *target*, para então executar testes de unidade em *drivers out-of-tree*. Os requisitos da automação de teste e das ferramentas dessa automação são listados e a adequação das ferramentas é analisada. Uma pequena infraestrutura é implementada, usando o *framework* de teste embutido no Buildroot, que por sua vez usa qemu para simular o *target* com um *kernel* vivo, e o *framework* KUnit, recentemente integrado no *kernel*, para realizar testes de unidade. Alguns casos de teste são criados para o estudo de caso, um *driver open source*, e um exemplo de ambiente de integração contínua é criado usando serviços *online* gratuitos para ilustrar a solução proposta. Todos os arquivos necessários para reproduzir os resultados desse trabalho são publicados como *open source*. A combinação de ferramentas *open source* selecionada é comparada com os requisitos previamente levantados e ela atende à necessidade de observabilidade e controlabilidade, enquanto executa testes de unidade em um ambiente o mais próximo possível do cenário real.

Palavras-chave: Teste de software embarcado. ferramentas de automação de teste. ferramentas *open source*. testes de unidade.

Test automation for kernel Linux out-of-tree drivers

ABSTRACT

Automated software testing is the *de facto* standard in the software industry. Testing embedded software brings some specific challenges, specially for the software part highly coupled to the hardware. Most of the usual open-source test frameworks are not suitable to this context, specially when speaking about out-of-tree device drivers for the Linux kernel. A driver is called out-of-tree when it is maintained in a source-code repository separate from the kernel tree (repository). This kind of driver has some restrictions: it often needs to support multiple kernel versions, usually resorting to some code being conditionally compiled or not, depending on the kernel version the driver is compiled against; in the system in which the driver runs, it lives in a binary file separate from the kernel itself, and it is loaded on demand; it can't be tested using most of tools distributed together to the kernel since these tools usually only support code already integrated to the kernel repository. This document proposes to use a composite of open-source tools to provide an environment close to the real scenario, the target hardware, to execute unit tests for out-of-tree drivers. The requirements for both the automated test and its tools are listed and the tools suitability is analyzed. A small infrastructure is developed, using both the test framework embedded in Buildroot, which in its turn uses qemu to simulate the target with a live kernel, and the recent KUnit framework from the kernel itself to run units tests. Some test cases are created for the case study, an open-source driver, and an example of a continuous integration pipeline is created using free online services in order to illustrate the proposed solution. All the files needed to reproduce the results from this document are released as open source. The composite of open-source tools is compared to previously listed requirements and it does accomplish the required observability and controllability, while running unit tests in an environment as close as possible to the real scenario.

Keywords: embedded software testing, test automation tools, open-source tools, unit testing.

LISTA DE ABREVIATURAS E SIGLAS

GB	*Gigabyte*
LKM	*loadable kernel module*
Mbps	*Megabits per second*
PCIe	*Peripheral Component Interconnect Express*
SVN	*Subversion*
TB	caso de teste criado usando o *framework* do Buildroot
TK	caso de teste criado usando o *framework* KUnit
URL	*Uniform Resource Locator*
USB	*Universal Serial Bus*
Wi-Fi	*Wireless Fidelity*

LISTA DE FIGURAS

LISTA DE TABELAS

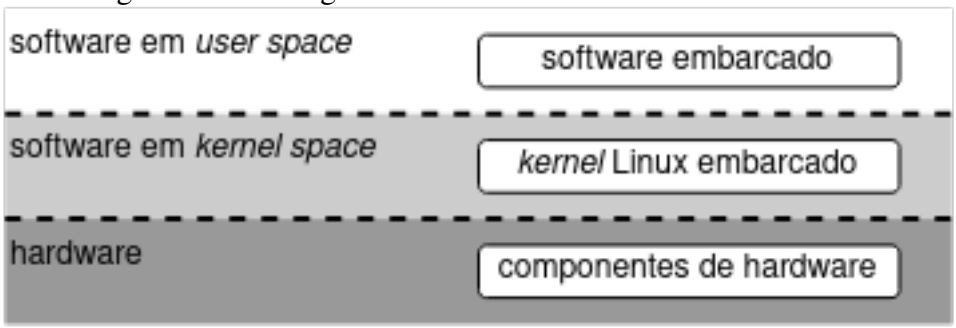
SUMÁRIO

1 INTRODUÇÃO

O uso de sistemas embarcados já faz parte do dia a dia atual, tanto no ambiente residencial (LIU; CHEN; HUANG, 2012), por exemplo telefone celular e *smart TV*, quanto em empresas (SUNG; CHOI, 2002), por exemplo no controle de processos fabris. No desenvolvimento de produtos embarcados, em busca de maior reúso de software, para diminuir o *time to market*, é cada vez mais comum o uso de processadores que são capazes de serem controlados por um sistema operacional (KANG; KWON; LEE, 2005). Uma das possibilidades de sistema operacional para embarcados é o Linux (SKRZYPIEC; MARSZAŁEK, 2020). Uma das principais motivações para interesse em usar Linux em sistemas embarcados é o baixo custo (UBM TECHNOLOGY, 2013 apud ÅHMAN, 2017).

É comum, ao desenvolver um sistema embarcado, ser necessário adaptar o software para itens específicos de hardware, o que torna necessária, no caso do Linux, a criação de *device drivers* (KANG; KWON; LEE, 2005; SKRZYPIEC; MARSZAŁEK, 2020). A Figura 1.1 mostra as camadas comuns em um sistema com Linux embarcado, em que o *kernel* Linux abstrai os detalhes do hardware para o software da aplicação.

Figura 1.1: Visão geral de um sistema com Linux embarcado



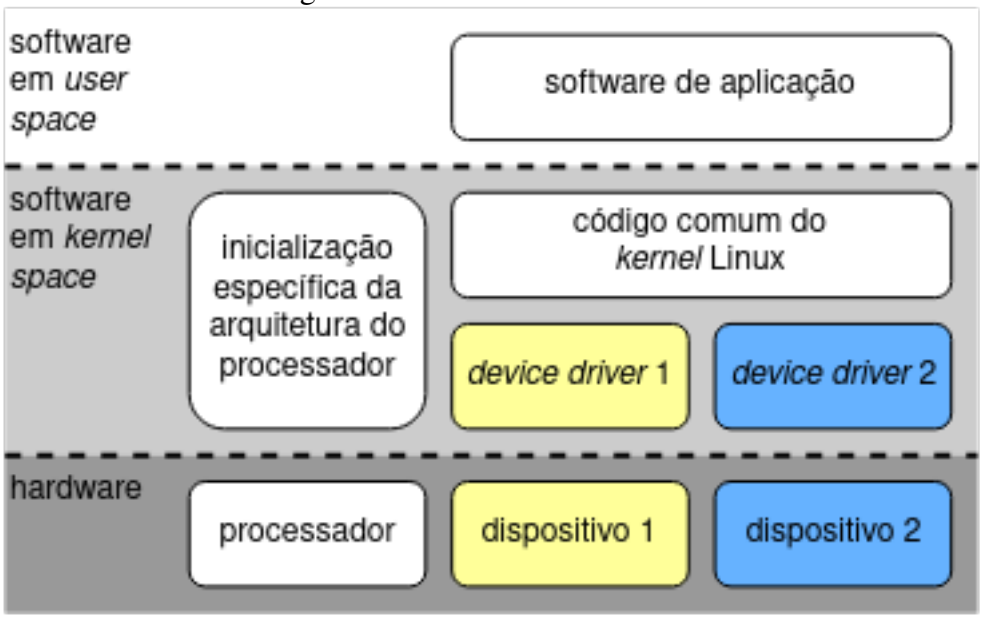
Fonte: O Autor

O uso de testes automatizados de software já é o padrão da indústria de software de uma forma geral (GAROUSI; AMANNEJAD; CAN, 2015). O crescimento do interesse no uso de testes automatizados na área de software embarcado é demonstrado pelo aumento do número de publicações sobre o assunto (GAROUSI et al., 2018). O teste de software embarcado possui especificidades, principalmente na camada que é altamente dependente do hardware (BAJER; SZLAGOR; WRZESNIAK, 2015).

O *kernel* Linux pode ser observado como sendo divido em três áreas, como ilustrado na Figura 1.2: o código específico por arquitetura (que configura o mínimo do hardware de cada arquitetura para poder operar), o código comum (escalonador de processos, gerenciador de memória, etc.), e os *drivers* de dispositivos. O repositório de código do

kernel Linux possui *drivers* para vários dispositivos (teclado, *mouse*, etc.), mas é também possível manter um *driver* de dispositivo em um repositório separado (*out-of-tree*) (PETAZZONI et al., 2021), sendo exemplos os *drivers* proprietários e também *drivers open source* que estão em desenvolvimento e não foram aceitos ainda na base de código do *kernel*.

Figura 1.2: Partes do *kernel* Linux



Fonte: O Autor

Muitas vezes esse código de *drivers out-of-tree* acaba sendo testado apenas em nível de sistema, pois depende diretamente do hardware.

O objetivo deste trabalho é analisar a adequação de ferramentas de automação de testes ao cenário de *drivers out-of-tree*, estendendo o que já foi feito para sistema embarcado sem sistema operacional em trabalhos semelhantes (GOMES, 2010; MONONEN, 2020), e também complementando a análise feita por Åhman (2017) para *drivers* do *kernel* Linux.

Os objetivos secundários são:

- Estudar as ferramentas *open source* que podem ser aplicadas ao contexto do trabalho;
- Aplicar as ferramentas a um estudo de caso;
- Avaliar a adequação das ferramentas.

Os capítulos a seguir estão organizados da seguinte forma:

- No Capítulo 2 são apresentados os principais conceitos relacionados ao contexto de *drivers out-of-tree* do *kernel* Linux e as especificidades de automação de teste para

software embarcado. Também são analisados trabalhos anteriores;

- No Capítulo 3 são apresentadas a proposta deste trabalho e a metodologia utilizada;
- No Capítulo 4 são analisadas as ferramentas e escolhidas as ferramentas a serem utilizadas;
- No Capítulo 5 é definido e implementado o estudo de caso;
- No Capítulo 6 são analisados os resultados;
- No Capítulo 7 são apresentadas as considerações finais e as sugestões de trabalhos futuros.

No restante do documento os nomes de sistemas de software, linguagens de programação e de arquivos que fazem parte da implementação são listados com `fonte monoespaçada`. Sempre que possível, ao longo das figuras foi mantido o mesmo padrão de cores (por exemplo para o *driver* que está sendo testado) a fim de facilitar a correlação entre as figuras durante a leitura do documento.

Buscando evitar ambiguidades, o documento conta com um glossário com os jargões da área de estudo, e, além disso, foram usados três termos distintos para se referir a software, dependendo de seu uso:

- software: é o software que é objeto de desenvolvimento ou teste, um exemplo é o *driver* que está sendo testado;
- ferramenta: é o software que pode ser instalado no computador de desenvolvimento para assistir o desenvolvedor no desenvolvimento ou teste de software, um exemplo são os *frameworks* de teste;
- comando comum em distribuições Linux: é o software que normalmente já vem pré-instalado nas distribuições Linux usadas no computador de desenvolvimento, um exemplo é o comando `ls`.

2 CONCEITOS BÁSICOS

Nesse capítulo são apresentados os principais conceitos sobre o funcionamento do *kernel* Linux e seus *drivers* de dispositivos que influenciam na escolha de ferramentas para desenvolver teste automatizado.

Apesar de existirem mais de quinhentas mil linhas de documentação no repositório do *kernel*[1], esta documentação é distribuída em partes que abordam ou tarefas específicas ou subsistemas do *kernel* específicos, e não apresenta uma visão geral do projeto. Este capítulo foi baseado na experiência do Autor, que trabalha há dez anos com Linux embarcado.

2.1 Tipos de *drivers* de dispositivos no *kernel* Linux

Os termos abaixo foram extraídos da documentação no repositório do *kernel*[1], mas nessa documentação não há uma definição explícita desses termos. A definição foi baseada na experiência do Autor.

Com relação ao lugar onde o código-fonte é guardado, os *drivers* de dispositivos podem ser classificados em:

- *in-tree* – quando o código-fonte está na mesma árvore (repositório) do *kernel* Linux, e portanto ou foi aceito no repositório principal do *kernel* ou faz parte de algum repositório específico de algum fabricante de processadores (*fork*);
- *out-of-tree* – quando o código-fonte está em repositório de código separado do *kernel* Linux.

Com relação ao comportamento em tempo de execução (*runtime*) determinado por configurações em tempo de compilação, é possível ter *drivers*:

- *built-in* – são aqueles que fazem parte do binário principal do *kernel* que é carregado no sistema embarcado no momento em que o sistema é ligado;
- módulo inserível em tempo de execução (LKM – da língua inglesa, *loadable kernel module*) – são aqueles que foram compilados para funcionar com o *kernel* mas ficam em binários separados no sistema de arquivos, e são carregados sob demanda.

O *kernel* permite as combinações listadas na Tabela 2.1.

[1] *The Linux Kernel documentation* disponível em <https://www.kernel.org/doc/html/latest/index.html>, acesso em 28 de agosto de 2021

Tabela 2.1: Opções de comportamento em tempo de execução disponíveis para serem selecionadas em tempo de compilação de acordo com onde está o código-fonte do *driver*

	built-in	módulo inserível
in-tree	X	X
out-of-tree		X

Fonte: O Autor

Os *drivers out-of-tree* ficam em repositórios de código-fonte separados do *kernel* e devem ser compilados contra uma árvore do *kernel* já compilada. Muitas vezes esses *drivers* possuem código que é condicionalmente compilado ou não, dependendo de qual a versão de *kernel* com que ele será usado, pois à medida que novas versões de *kernel* são criadas, novas funcionalidades são adicionadas.

Também é comum esse código possuir partes que são condicionalmente compiladas ou não, dependendo da configuração de funcionalidades escolhidas durante a compilação do *kernel*. Em sistemas embarcados, é prática comum habilitar apenas as funcionalidades do *kernel* necessárias para a operação do hardware, dado que o hardware muitas vezes possui menos recursos do que um computador de uso geral. Além disso, a variabilidade de combinações de dispositivos que realmente são usados no sistema é muito pequena quando comparada com um computador de uso geral.

Os *drivers out-of-tree* são sempre módulos inseríveis, nunca *built-in*. Uma vez que um *driver* que foi compilado como módulo inserível foi de fato inserido no *kernel* que está em execução, o comportamento é o mesmo que de um *driver built-in*.

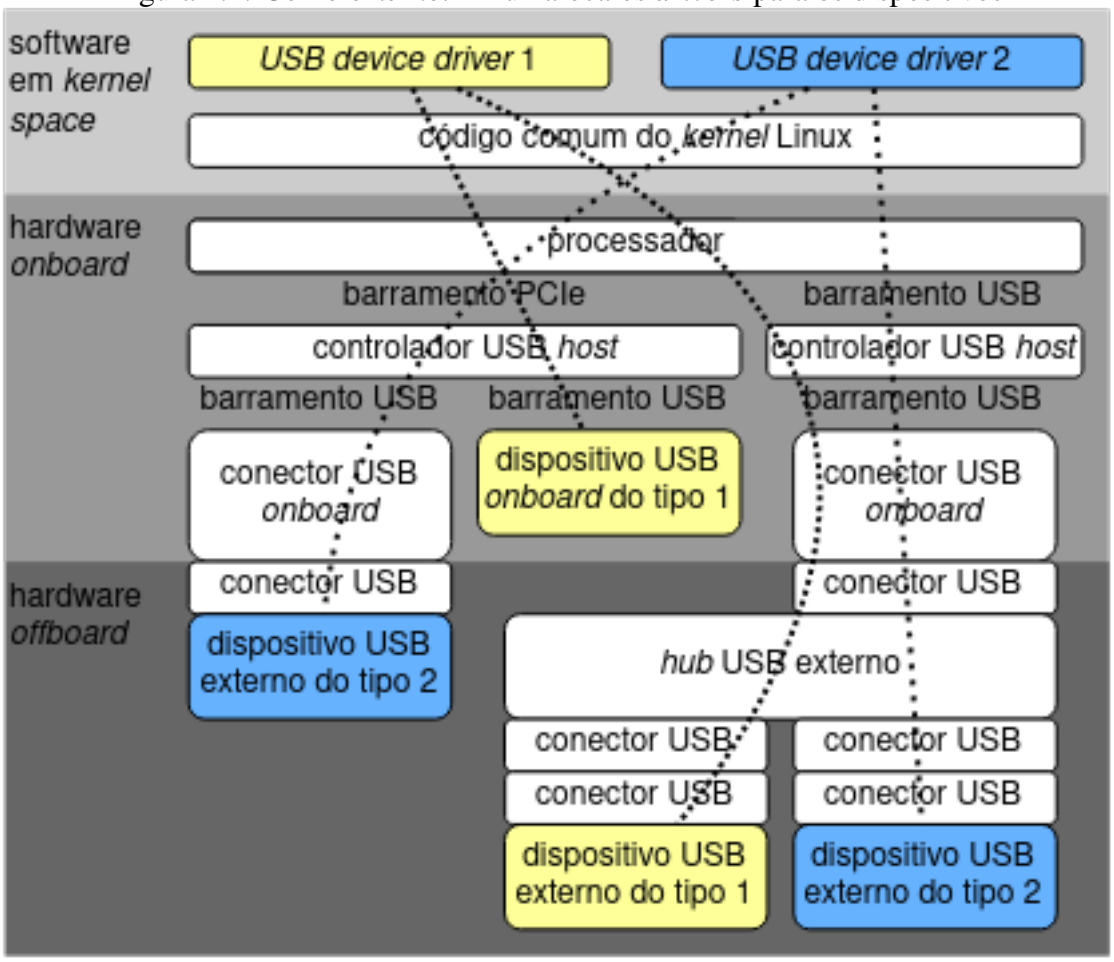
2.2 Suporte a novos dispositivos usando o *kernel* Linux

É comum, em sistemas embarcados, a necessidade de dar suporte em software a um componente de hardware novo e/ou proprietário. A adoção do *kernel* Linux no projeto de um sistema embarcado acarreta grande reúso de código, tanto na área de protocolos quanto no suporte a dispositivos e barramentos de conexão. Por exemplo, o *kernel* Linux já contém embutido o suporte a diversos protocolos, como *Ethernet* e *Bluetooth*, fazendo com que não seja necessário reimplementá-los a cada novo dispositivo suportado. Ele suporta também diversos barramentos de dados, por exemplo PCIe e USB.

Como mostra a Figura 2.1, para usar, no projeto de um sistema embarcado, um novo dispositivo USB, seja ele soldado na placa de circuito impresso (*onboard*) ou com conector externo (*offboard*), não é necessário reescrever todo o código de interface com o

hardware, pois o *kernel* Linux já provê *drivers in-tree* para muitos dispositivos, incluindo, por exemplo, controladores USB *host* e barramentos. O mesmo *driver* é usado para vários dispositivos do mesmo tipo que estejam ligados ao sistema, independente de como no hardware cada um é acessível ao processador. Se existirem dois ou mais dispositivos do mesmo tipo no sistema, uma instância do *driver* é usada para cada um deles.

Figura 2.1: Como o *kernel* Linux aloca os *drivers* para os dispositivos

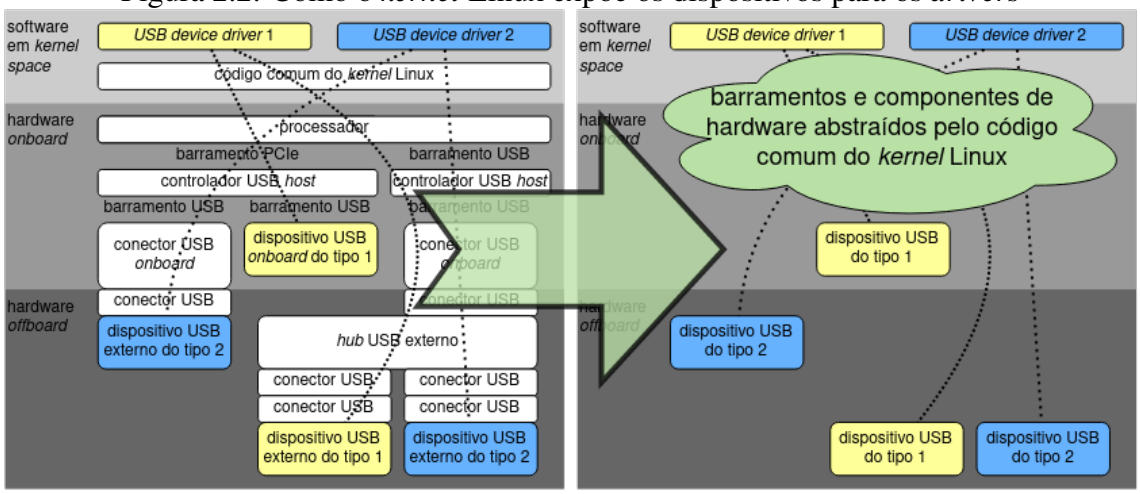


Fonte: O Autor

O código comum do *kernel*, incluindo outros *drivers in-tree*, acaba abstraindo para o *driver* se o dispositivo que ele representa está *onboard* ou *offboard*. Da mesma forma, o *kernel* abstrai para o *driver* os barramentos e dispositivos intermediários entre o processador e o dispositivo em questão, como mostra a Figura 2.2. Apenas o fato de que o dispositivo é um dispositivo USB fica visível para o *driver*, que pode então acessar os registradores do hardware através das funções comuns do kernel para dispositivos USB.

O *kernel* já possui diversos tipos de interfaces que podem ser acessadas pela aplicação em *user space*, mostradas na Figura 2.3: a interface *net* modela interfaces de rede, enquanto que outros dispositivos podem ser expostos como dispositivos de bloco (*block*),

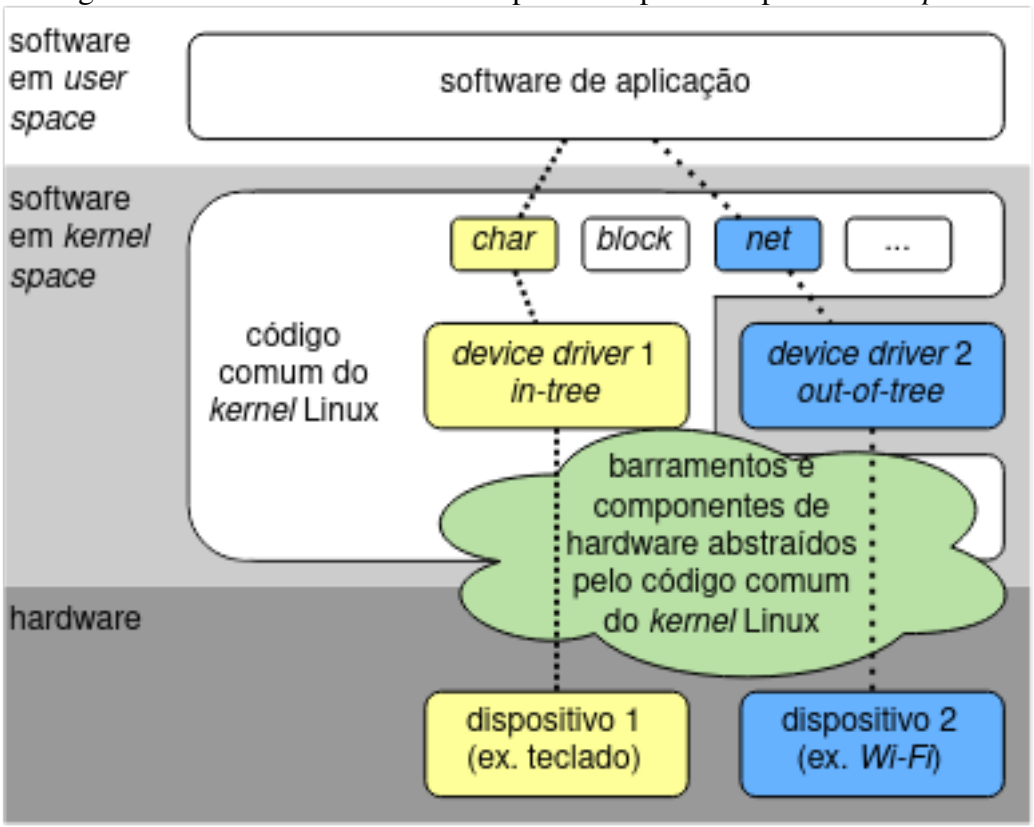
Figura 2.2: Como o *kernel* Linux expõe os dispositivos para os *drivers*



Fonte: O Autor

por exemplo um *pendrive*, ou dispositivos de caracteres (*char*), como um teclado. Todos os tipos de interface estão disponíveis para todos os tipos de *drivers*.

Figura 2.3: Como o *kernel* Linux expõe os dispositivos para o *user space*



Fonte: O Autor

Portanto no exemplo de um novo dispositivo USB, apenas o *driver* que mapeia as suas especificidades precisa ser desenvolvido.

2.3 Automação de teste para software embarcado

Em uma pesquisa de opinião sobre em que área do teste de software a indústria mais encontra desafios que poderiam ter ajuda do meio acadêmico (GAROUSI et al., 2017) um dos principais itens é a automação das atividades de teste (que inclui, entre outros, a criação de testes) e as ferramentas relacionadas a essas atividades.

Como argumenta Galin (2018), o uso de testes incrementais ao invés de apenas testar o software completo traz várias vantagens, facilitando encontrar e resolver problemas. O uso de testes de unidade também facilita o teste de situações-limite, como as reações do software a uma falha de hardware (MAIER; KLEEBERGER, 2016). Portanto quando se busca criar um conjunto de testes automatizados para evitar regressões, é desejável que esse conjunto inclua testes de unidade para que consiga atingir uma grande cobertura de código, aumentando assim a chance de detectar problemas no software.

No entanto, o teste de software embarcado, principalmente de suas camadas mais baixas, que são mais dependentes do hardware, acaba trazendo dificuldades específicas:

- Testes em paralelo são dificultados pelo tempo de implantação do software no hardware (BAJER; SZLAGOR; WRZESNIAK, 2015);
- As situações-limite são mais difíceis de testar no hardware final, e normalmente requerem testes que injetem falhas (MAIER; KLEEBERGER, 2016);
- O uso de técnicas de desenvolvimento como *Test Driven Development* para software embarcado ainda é uma área em desenvolvimento e tema de pesquisa (BAJER; SZLAGOR; WRZESNIAK, 2015; MONONEN, 2020);
- O código das camadas mais baixas possui alto acoplamento com o hardware (GOMES, 2010);
- A linguagem `C` ainda é a mais usada em sistemas embarcados segundo dados de 2013 (UBM TECHNOLOGY, 2013 apud ÅHMAN, 2017), e existem muito mais ferramentas para teste de unidade em `C++` do que em `C`.

Em geral, para lidar com essas limitações, recorre-se a algum de tipo de simulação, seja pelo uso de simuladores do hardware como um todo, simuladores de processador ou, ainda, pelo uso de dublês de teste (*stubs*, *mocks* ou modelos) compilados para o computador usado para desenvolvimento (*host*). O uso de simulação traz as seguintes vantagens:

- Remove a necessidade de ter um protótipo funcional do hardware para iniciar o desenvolvimento (GOMES, 2010);

- Remove a limitação de tempo de implantação do software no hardware final, pois muitas vezes o hardware final (*target*) tem menos recursos que o *host* (BAJER; SZLAGOR; WRZESNIAK, 2015).

Testar *drivers* do *kernel* Linux também traz algumas dificuldades específicas:

- O *kernel* Linux é usado em diversas arquiteturas de processadores, portanto deve funcionar em diferentes tipos de *endianness* (*little endian*, *big endian*) e com processadores com diferentes tamanhos de palavra (de 32 bits, de 64 bits);
- O *kernel* Linux é escrito na linguagem `C`, e existem muito mais ferramentas para teste de unidade em `C++` do que em `C`.

Usar *mocks* e *stubs* em geral só é suportado pelas ferramentas em *user space*. Apesar de algumas ferramentas de teste em `C++` suportarem testar código em `C`, esse uso acaba forçando a criação de *mocks* de várias funções do *kernel* para que o código possa ser testado em *user space*.

2.4 Trabalhos relacionados

Foram analisados diversos trabalhos relacionados e abaixo são apresentados os que mais se assemelham à proposta. Para classificar as ferramentas usadas foram consideradas *open source* apenas as ferramentas que disponibilizam o código-fonte com a licença explícita, com o arquivo ou repositório listado em ferramentas de busca usuais da *Internet* e que a URL de *download* esteja funcional e contenha de fato o código-fonte e não um binário executável.

Bastien et al. (2004) usam simulação do hardware como um todo, com uma ferramenta comercial, para testar software de aplicação crítica sem sistema operacional, injetando falhas de hardware e testando a resposta do software. Segundo Gomes (2010) essa ferramenta comercial, chamada `Simics`, usa internamente modelos de dispositivos de hardware.

Maier and Kleeberger (2016) abordam a automação do teste de uma forma mais teórica, portanto acabam não listando as ferramentas usadas, mas sugerem o uso de testes de unidade e de simulação do hardware como um todo para testar software sem sistema operacional, injetando falhas em hardware simulado e testando a resposta do software.

Gomes (2010) usa modelos para os dispositivos de hardware executando no computador de desenvolvimento (*host*) para realizar testes de unidade para software sem sis-

tema operacional. O código-fonte usado para executar os testes automatizados não está disponível como *open source* pois o trabalho apresenta uma metodologia para que esse código seja criado ao mesmo tempo que o software que será testado, mas uma possível reprodução do experimento deve ser facilitada devido à existência de diversos exemplos de código no trabalho. Apesar de o software que foi testado no exemplo do trabalho ser escrito em `C++`, o código de teste exemplificado não usa classes e portanto poderia ser aplicado também para testar software escrito em `C`.

Mononen (2020) usa um *framework open source* para criar *mocks* para usar o computador de desenvolvimento (*host*) para realizar testes de unidade em software sem sistema operacional. Apesar de a ferramenta de teste usada, chamada `GoogleTest`, ser escrita em `C++` ela suporta testar código em `C`, mas no contexto deste trabalho o uso forçaria a criação de *mocks* para várias funções do *kernel* para que o código possa ser testado em *user space*.

Åhman (2017) usa ferramentas *open source* e cria um *framework* que não é *open source* para realizar testes de unidade, usando o computador de desenvolvimento (*host*), em *device drivers* inseríveis no *kernel* Linux. Os testes são realizados criando *mocks* para as funções do *kernel* em *user space*. O uso de `qemu` foi considerado mas foi abandonado pois a proposta do *framework* é de não modificar o código-fonte do *driver*. O trabalho mostra apenas a estrutura geral do *framework* e poucos exemplos, o que dificulta a reprodução do experimento. Segundo a descrição da ferramenta criada, ela tenta compilar o *driver* a ser testado em *user space*, detecta que a compilação falhou devido à inexistência da declaração de uma função do *kernel* e cria um *mock* inicialmente vazio para essa função, a ser preenchido futuramente pelo desenvolvedor. Esse processo é repetido iterativamente de forma automática até que o *driver* compile, gerando um *framework* de testes para aquele *driver*. A conclusão do trabalho é que o uso do *framework* criado traz mais vantagens quando o código do *driver* é complicado, é reusado frequentemente entre projetos e faz poucas chamadas a funções do *kernel*.

Witkowski et al. (2007) apresentam um *framework* para gerar modelos para *device drivers* para ser usado em ferramentas que não são *open source*. Neste caso, diferentemente dos outros trabalhos relacionados listados, o modelo é um modelo abstrato do *driver* e não um modelo do dispositivo de hardware, a ser checado contra predicados previamente estabelecidos de o quê um *driver* deve fazer e não fazer. Infelizmente o código-fonte da ferramenta `DDverify`[2] não é mantido, seriam necessárias adaptações

[2] `DDverify` disponível em <http://www.cprover.org/ddverify>, acesso em 15 de maio de 2021

para funcionar com as dependências, uma vez que o trabalho não cita a versão das dependências usadas. Para a dependência `SatAbs`[3] ao tentar fazer o *download* do código-fonte só existem os binários pré-compilados em 2006 e o repositório que usa o controle de versão `SVN` não está funcional. Além disso a licença de ambos `DDverify` e `SatAbs` é restritiva e exige que todas as instalações sejam comunicadas por *e-mail* para o autor.

A Tabela 2.2 lista as características das ferramentas, do software testado e do tipo de teste usado nos trabalhos relacionados. Como pode-se observar na tabela, todos os trabalhos listados apresentam soluções que poderiam ser usadas para testar software escrito em `C`. São maioria (quatro entre seis) os trabalhos que levam em conta *endianness* ou tamanho de palavra do processador. Mas um destes usa ferramenta comercial, outro não especifica as ferramentas, um terceiro define uma metodologia para criar os testes e a própria ferramenta junto com o a criação do código-fonte, e o quarto usa ferramentas que não são *open source*. Também são maioria (quatro entre seis) os trabalhos que testam software sem sistema operacional. Também são maioria (quatro entre seis) os trabalhos que usam testes de unidade. Metade dos trabalhos não depende de licenças de ferramentas comerciais. Apenas um dos trabalhos descreve como reproduzir o experimento de forma determinística, e este testou software sem sistema operacional.

Os trabalhos que produzem um ambiente mais próximos do cenário real para a execução do teste, o que melhora a qualidade do teste, usam ferramentas comerciais e testam apenas software sem sistema operacional, além de não descreverem como reproduzir o experimento de forma determinística. Nenhum dos trabalhos publica como *open source* todos os arquivos necessários para reproduzir o experimento. Dos trabalhos que não dependem de licenças de ferramentas comerciais, todos usam *stubs* e *mocks* no *host*, metade testa software sem sistema operacional, e a outra metade usa ferramentas *open source* para criar uma nova ferramenta mas não publica essa nova ferramenta como *open source*.

Nenhum dos trabalhos usa simulação do processador para melhorar a qualidade do teste, apesar de existirem ferramentas *open source* que implementem essa simulação. Um dos trabalhos, desenvolvido antes da criação do *framework* `KUnit` (analisado no Capítulo 4), considerou mas descartou esse uso de simulação do processador, adotando uma abordagem de gerar *mocks* e *stubs* para todas as funções do *kernel* usadas pelo *driver* testado. O referido trabalho cita como "grande desapontamento" o fato de que o *framework* criado funcionar para o *driver* que serviu de exemplo, mas para outros *drivers* não fun-

[3] `SatAbs` disponível em <http://www.cprover.org/satabs>, acesso em 15 de maio de 2021

Tabela 2.2: Comparação de características dos trabalhos relacionados

Trabalho	Bastien et al. (2004)	Maier and Kleeberger (2016)	Gomes (2010)	Mononen (2020)	Åhman (2017)	Witkowski et al. (2007)
Usa ferramentas comerciais	X					
Usa ferramentas que não são *open source*						X
Usa ferramentas *open source*				X	X	X
Define uma metodologia de criação de testes		X	X	X		
Cria ferramenta que não é *open source*					X	X
Cria ferramenta *open source*						
Executa testes no hardware real				X		
Usa simulação do hardware como um todo	X	X				
Usa simulação do processador						
Usa *stubs*, *mocks* ou modelos no *host*			X	X	X	X
Leva em conta *endianness*	X	X				X
Leva em conta tamanho de palavra do processador	X	X	X			X
Testa software sem sistema operacional	X	X	X	X		
Testa software com sistema operacional					X	X
Não menciona linguagem do software testado	X	X				
Software testado escrito em `C++`			X	X		
Software testado escrito em `C`					X	X
Solução poderia testar software escrito em `C`	X	X	X	X	X	X
Teste de injeção de falhas	X	X				
Teste de unidade		X	X	X	X	
Teste baseado em modelo do dispositivo	X		X			
Teste baseado em modelo abstrato do *driver*						X
Reprodução de experimento não depende de licenças de ferramentas comerciais			X		X	X
Trabalho descreve como reproduzir o experimento de forma determinística			X			

Fonte: O Autor

cionar de imediato pois cada *driver* usa um subconjunto diferente de funções do *kernel* e o *parser* que cria os *mocks* precisaria se adaptado para suportar novas funções. E a conclusão do mesmo trabalho é que o *framework* não traz vantagem a todos os *drivers*, sendo mais vantajoso quando o código do *driver* é complicado, é reusado frequentemente entre projetos e faz poucas chamadas a funções do *kernel*.

3 PROPOSTA E METODOLOGIA

A proposta deste trabalho é avaliar a adequação de ferramentas *open source* para criar testes contra regressões durante o desenvolvimento e manutenção de *drivers out-of-tree*.

Assim, a partir da definição dos requisitos de teste e de automação, diferentes ferramentas são analisadas e a ferramenta mais adequada é avaliada em profundidade através da implementação de um estudo de caso. Neste capítulo são listados os requisitos da automação do teste e das ferramentas de automação. No Capítulo 4 é realizada a pesquisa de ferramentas *open source*, e são feitos o estudo dessas ferramentas e a seleção da ferramenta mais adequada. No Capítulo 5 é realizada a análise experimental e no Capítulo 6 são apresentados os resultados, comparando com os requisitos aqui levantados.

Durante a elaboração do trabalho buscou-se gerar um ambiente para o teste próximo ao cenário real, facilitar a reprodução do experimento e não depender de licenças de ferramentas comerciais.

3.1 Listagem de requisitos da automação do teste

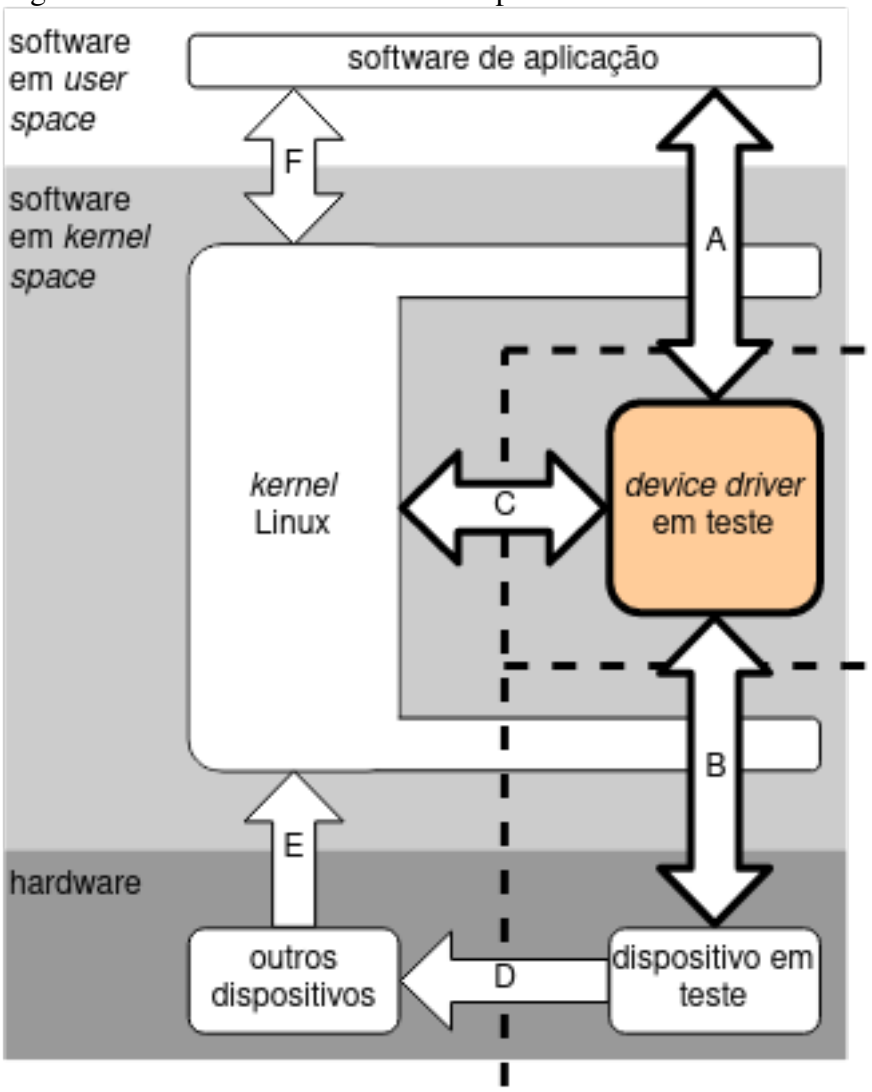
Para automatizar o teste de um *driver* em um sistema embarcado, devem-se considerar os aspectos de controlabilidade e observabilidade do *driver* em teste.

A Figura 3.1 apresenta as interfaces de um *driver* sob teste:

- A = o software da aplicação em *user space* se comunica com o *driver* através de interfaces padronizadas do *kernel* para fazer configurações, obter o estado atual e trocar dados;
- B = o *driver* se comunica com o dispositivo que ele controla através da abstração que o *kernel* faz dos outros componentes de hardware, como foi mostrado na Figura 2.2;
- C = são as interfaces padrão de todos os *drivers*;
- D,E = a inserção e remoção física do dispositivo são reconhecidas por outro componente de hardware e notificadas ao *kernel*, que por sua vez ativa alguma das interfaces C;
- F = representa as interações do software da aplicação com o *kernel* que não estão diretamente ligadas ao *driver* sob teste, em geral usam apenas chamadas de sistema

(*syscalls*) para trabalhar com arquivos, temporização, etc.

Figura 3.1: Isolando um *device driver* para realizar testes de unidade



Fonte: O Autor

Para que se possa realizar o teste de unidade do *driver* é necessário que se possa observar e controlar as interfaces A, B e C.

3.2 Listagem de requisitos das ferramentas de automação do teste

Para que se possa executar um ou mais casos de teste automaticamente é necessário que a ferramenta consiga fazer a orquestração de execução de testes. Para que cada caso de teste tenha um resultado positivo ou negativo é necessário que a ferramenta consiga testar estados observáveis, comparando com valores esperados e gerar o resultado. Isso é feito com o uso de *assertions*. Para que o ambiente do *driver* sob teste seja contro-

lável é essencial que se possa substituir, no momento do teste, algumas funções das quais ele depende, usando *mock* de funções.

Buscando uma maior qualidade do teste, o teste deve ser realizado em um ambiente o mais próximo possível do hardware real. Excluindo o uso de hardware real para evitar as limitações listadas na Seção 2.3, abaixo são listadas as alternativas (as mesmas listadas na Tabela 2.2), das mais próximas para as menos próximas:

- simulação do hardware como um todo, com modelos de dispositivos inclusive do dispositivo cujo *driver* está sendo testado;
- simulação do processador com modelos de dispositivos mas não do dispositivo cujo *driver* está sendo testado;
- criação de dublês de teste (*stubs*, *mocks* ou modelos) no computador de desenvolvimento *host*.

Uma consequência da terceira opção no contexto deste trabalho é a criação de dublês de teste para várias funções do *kernel* Linux pois o teste precisaria ser realizado em *user space*. Nessa opção, por um lado o teste fica um pouco mais distante do cenário real, mas por outro lado são facilmente obtidas a observabilidade e a controlabilidade necessárias.

Uma consequência das duas primeiras opções é a execução de um *kernel* vivo, ou seja, ter um *kernel* em execução com todas as suas partes essenciais (escalonador de processos, gerenciador de memória, etc.) e que consegue interagir com o *user-space*. Para que se possa fazer isso no cenário a que se aplica o trabalho, é necessário compilar tanto o *kernel* quanto o *driver out-of-tree*.

É importante ressaltar também que se um modelo do dispositivo cujo *driver* está sendo testado for usado, ele deve permitir o controle de suas interações com o *driver* para que se possa obter uma alta cobertura do código.

Dependendo das ferramentas utilizadas, se essas permitirem compilação cruzada e simulação de diferentes arquiteturas de processador, já são então levados em conta o *endianness* e o tamanho de palavra do processador.

A maioria das ferramentas foi criada ou para realizar testes em *user space* de uma aplicação, ou então para testar o *kernel* que está em execução de dentro do sistema que está sendo testado, usando também apenas acesso de *user space* e limitando-se ao teste das interfaces expostas, e isso limitaria a observabilidade. Mas como deseja-se testar um *driver* do *kernel* e conseguir observar o estado interno, o ideal é usar uma ferramenta

que execute em *kernel space*. Isso evita também a criação de dublês de teste para muitas funções do *kernel*.

Algumas dificuldades no uso de ferramentas são esperadas, relacionadas a especificidades de teste de software embarcado e/ou com Linux:

- A maioria das ferramentas de testes foi criada para executar testes no computador de desenvolvimento (*host*) e não suporta compilação cruzada;
- A maioria das ferramentas é criada assumindo que haverá bibliotecas (`libc` por exemplo) e comandos (`bash` por exemplo) no sistema que será testado;
- A maioria das ferramentas que acompanham o *kernel* Linux foram criadas para suportar apenas código *in-tree*;
- A maioria das ferramentas que acompanham o *kernel* Linux foram criadas para executar testes de sistema, em *user space*;
- Existem muito mais ferramentas de teste de unidade para `C++` do que para `C`, principalmente devido a dificuldades que uma linguagem que não é orientada a objetos traz para a criação do *framework* de testes. O *kernel* Linux e seus *drivers* são escritos em `C`.

3.3 Estudo de caso

O estudo de caso é realizado com um *driver open source*. Os principais requisitos para considerar um *driver* para ser usado como estudo de caso são:

- Ter código *open source*, visando facilitar a reprodutibilidade do experimento deste trabalho;
- Código ser disponibilizado como *out-of-tree*.

Além disso, também é dada preferência para algumas características que podem facilitar trabalhos futuros que necessitem de acesso físico ao dispositivo:

- O dispositivo estar à venda no comércio eletrônico para o público em geral;
- O autor ter acesso ao dispositivo.

3.4 Implementação

É escolhido um grupo de ferramentas que mais se adequam ao contexto. As ferramentas escolhidas são aplicadas ao *driver* escolhido e alguns testes automatizados são criados para o *driver* como prova de conceito. É avaliado o atendimento dos requisitos de automação previamente levantados.

4 ANÁLISE DE FERRAMENTAS DE AUTOMAÇÃO DE TESTES PARA SOFTWARE EMBARCADO

4.1 Estudo de ferramentas

A Tabela 4.1 lista algumas ferramentas *open source* comuns relacionadas à automação de teste e/ou sistemas embarcados que poderiam ser usadas. A tabela apresenta ainda as licenças de acordo com a listagem SPDX[1] e a URL da documentação[2]. Na coluna "Usado por" são listados tanto trabalhos relacionados que usaram a ferramenta quanto outras ferramentas da mesma tabela que têm a ferramenta em questão em suas dependências.

A Tabela 4.2 apresenta as características principais dessas ferramentas. Essa tabela foi criada a partir da análise da documentação de cada ferramenta e, em casos em que a documentação não foi suficiente, também a partir da leitura de alguns arquivos do código-fonte da ferramenta. O propósito da tabela é dar uma visão geral de cada ferramenta, além de relacionar cada uma com trabalhos listados na bibliografia e com suas principais dependências, sejam outras ferramentas da mesma tabela, sejam bibliotecas ou comandos comuns em distribuições Linux.

4.2 Análise de adequação das ferramentas

A Tabela 4.3 apresenta uma análise de como cada ferramenta listada na Tabela 4.1 se comporta em relação aos requisitos de automação levantados na Seção 3.2.

Como mostrado na Tabela 4.3, nenhuma das ferramentas analisadas permite o uso de modelos de dispositivos como *plug-in*. Isso dificulta o uso de modelo para o dispositivo cujo *driver* está sendo testado pois ele ficaria pouco controlável, uma vez que seria necessário recompilar o simulador de hardware (`qemu` ou `gem5`) completamente para cada modificação no comportamento do modelo. Devido a isso, que tecnicamente não impede a implementação mas aumenta muito a complexidade, optou-se por usar simulação do processador.

Na mesma tabela pode-se observar que a maioria das ferramentas atende a poucos critérios. E a ferramenta que atende a mais critérios é o `Buildroot`, faltando apenas

[1] *SPDX License List* disponível em <https://spdx.org/licenses>

[2] Todas as URLs listadas nesta seção foram acessadas em 15 de maio de 2021.

Tabela 4.1: Ferramentas que podem ser usadas para automatizar testes de software embarcado com sistema operacional

Ferramenta	**Licenças**	**Documentação**	**Usado por**
`Buildroot`	GPL-2.0-or-later	<https://buildroot.uclibc.org>	`LKMC`
`fff` - *Fake Function Framework*	MIT	<https://github.com/meekrosoft/fff>	Åhman (2017)
`gem5`	principais: BSD-3-Clause, MIT, Apache-2.0	<https://www.gem5.org>	`LKMC`
`GoogleTest`	BSD-3-Clause	<https://github.com/google/googletest>	Mononen (2020)
`Kselftest` - *Linux Kernel Selftests*	GPL-2.0-only	<https://www.kernel.org/doc/html/latest/dev-tools/kselftest.html>	
`KUnit` - *Unit Testing for the Linux Kernel*	GPL-2.0-only	<https://www.kernel.org/doc/html/latest/dev-tools/kunit/index.html>	
`LKMC` - *Linux Kernel Module Cheat*	GPL-3.0-only	<https://github.com/cirosantilli/linux-kernel-module-cheat>	
`LTP` - *Linux Test Project*	GPL-2.0-or-later	<https://github.com/linux-test-project/ltp>	
`out-of-tree`	AGPL-3.0-only	<https://github.com/jollheef/out-of-tree>	
`qemu`	principal: GPL-2.0-only	<https://www.qemu.org>	`Buildroot`, `LKMC`, `out-of-tree`
`Unity` - *Unity Test*	MIT	<https://github.com/ThrowTheSwitch/Unity>	Åhman (2017)

Fonte: O Autor

dois deles. Para complementar pode-se usá-lo combinado ao `KUnit` que é a única ferramenta listada na tabela que permite testes em *kernel space*.

4.3 Ferramentas selecionadas

A ferramenta `KUnit` é um *framework* criado para executar testes de unidade em *kernel space*. Ele é composto por duas partes:

- o mecanismo de execução dos testes: este pode ser usado tanto por *drivers built-in* quanto *drivers* inseríveis. Para os *drivers built-in* os testes são executados na subida

Tabela 4.2: Características principais das ferramentas listadas na Tabela 4.1

Ferramenta	**Características principais**
`Buildroot`	provê infraestrutura para compilar *kernel* Linux, *drivers out-of-tree* e milhares de comandos em *user space* para várias arquiteturas de processador, possui também um *framework* de testes de sistema usando `qemu`
`fff`	usado por Åhman (2017) para facilitar o *mock* de funções em `C`, é implementado em um único arquivo *header* (.h) usando macros
`gem5`	simulador de arquitetura de processador, poderia simular hardware como um todo a partir de uma lista preexistente de modelos de dispositivos, não suporta *plug-in* de dispositivos, sendo necessário modificar o código-fonte e recompilar para adicionar dispositivos. Pode operar como um *plug-in* para a ferramenta comercial `Simics`
`GoogleTest`	foi usado por Mononen (2020), suporta testar aplicações em `C`, mas foi descartado por Åhman (2017) pela dificuldade adicional de compilar o código testado em `C` junto com o *framework* em `C++`
`Kselftest`	testes necessitam `bash` e outros comandos embarcados e são executados em *user space*
`KUnit`	ferramenta mais recente, integrada no *kernel* em 2019, suporta execução no *host*, segundo Higgins (2020), um dos autores da ferramenta, pode ser adaptada para execução em `qemu` ou hardware real
`LKMC`	usa `qemu` ou `gem5`, com `Buildroot`, para prover um ambiente de depuração e aprendizagem, não foi criado para automatizar testes
`LTP`	testes escritos em `C` ou *shell scripts* para executar em *user space* e testar os comandos comuns de uma distribuição Linux e sua interação com o *kernel*
`out-of-tree`	criado para testar a compatibilidade de um *driver out-of-tree* contra diversas versões de *kernel*, usa `qemu` mas não suporta diferentes arquiteturas de processador
`qemu`	simulador de hardware como um todo com uma lista preexistente de modelos de dispositivos, não suporta *plug-in* de dispositivos, sendo necessário modificar o código-fonte e recompilar para adicionar dispositivos
`Unity`	permite a criação de *assertions* usando macros, Åhman (2017) usou com `fff` em vez do `GoogleTest`, pois estas ferramentas são para testar código `C`

Fonte: O Autor

do sistema, ou seja, quando o *kernel* é carregado no sistema e começa a operar. Para os *drivers* inseríveis os testes são executados quando o *driver* é de fato inserido no *kernel* que já está em execução;

- o orquestrador de testes, também chamado `kunit_tool`: este suporta apenas

Tabela 4.3: Análise da relação das características das ferramentas listadas na Tabela 4.1 com os requisitos de automação levantados na Seção 3.2

Característica	Buildroot	fff	gem5	GoogleTest	Kselftest	KUnit	LKMC	LTP	out-of-tree	qemu	Unity
Permite a orquestração de execução de testes	X			X	X	X		X	X		X
Permite *assertions*	X			X		X		X			X
Permite *mock* de funções		X		X		X					
Permite testes de aplicações em *user space*	X	X		X	X			X	X		X
Permite testes da interface que o *kernel* expõe ao *user space*	X			X	X			X	X		X
Permite testes em *kernel space*						X					
Leva em conta *endianness*	X		X		X	X	X			X	
Leva em conta tamanho de palavra do processador	X		X		X	X	X			X	
Compila *kernel*	X					X	X				
Compila *driver out-of-tree*	X						X		X		
Realiza compilação cruzada	X						X				
Permite executar um *kernel* vivo	X		X				X		X	X	
Permite simulação em nível de instrução de várias arquiteturas de processador	X		X				X			X	
Usa modelos de dispositivos	X		X				X			X	
Permite modelo de um novo dispositivo como *plug-in*											

Fonte: O Autor

compilar e executar no *host* o *kernel* com *drivers built-in*. Ele possui também um *parser* que converte as mensagens do formato bruto para um formato legível, mostrando o resultado "*PASSED*" ou "*FAILED*" para cada caso de teste e um sumário de quantos testes passaram ou falharam. Esse *parser* pode ser usado com o resultado bruto que o mecanismo de execução de teste gera para qualquer tipo de *driver*.

Segundo a documentação da ferramenta, o código de testes de unidade e o próprio *framework* não devem ser compilados na versão de produção do software, pois o `KUnit` não foi criado com este fim e é possível que os testes reduzam a estabilidade ou a segurança do sistema. A ferramenta não possui funções ou macros para substituir funções já definidas por *mocks*, no entanto ela executa em *kernel space* e isso permite que o código

do *driver* seja escrito de forma que funções possam ser substituídas por *mocks* através de indireções ou de definições condicionadas à compilação dos testes de unidade, como recomendado na documentação da ferramenta. A ferramenta provê mecanismos para criar *mocks* de objetos que, no caso do *kernel* que é escrito em `C`, são estruturas com valores e ponteiros de função. A ferramenta provê macros para realizar tanto *assertions* (parar imediatamente o caso de teste e falhar) quanto *expectations* (marcar o teste como falha mas continuar o caso de teste). Detalhes sobre o uso do `KUnit` são apresentados no Capítulo 5.

A ferramenta `Buildroot` foi criada para automatizar a geração de imagens para sistemas embarcados que executem *kernel* Linux, *drivers out-of-tree* e comandos em *user space*. Ela não é um *framework* de testes, mas possui um embutido, que foi criado para testar a própria ferramenta. Ele é escrito em `Python` usando os módulos `nose2` e `unittest` e permite executar em `qemu` uma imagem gerada, chamando comandos em *user space* e fazendo *assertions* da saída e do resultado de cada comando. Informações mais detalhadas do `Buildroot` e seu *framework* de testes são apresentadas no Capítulo 5 e no APÊNDICE B.

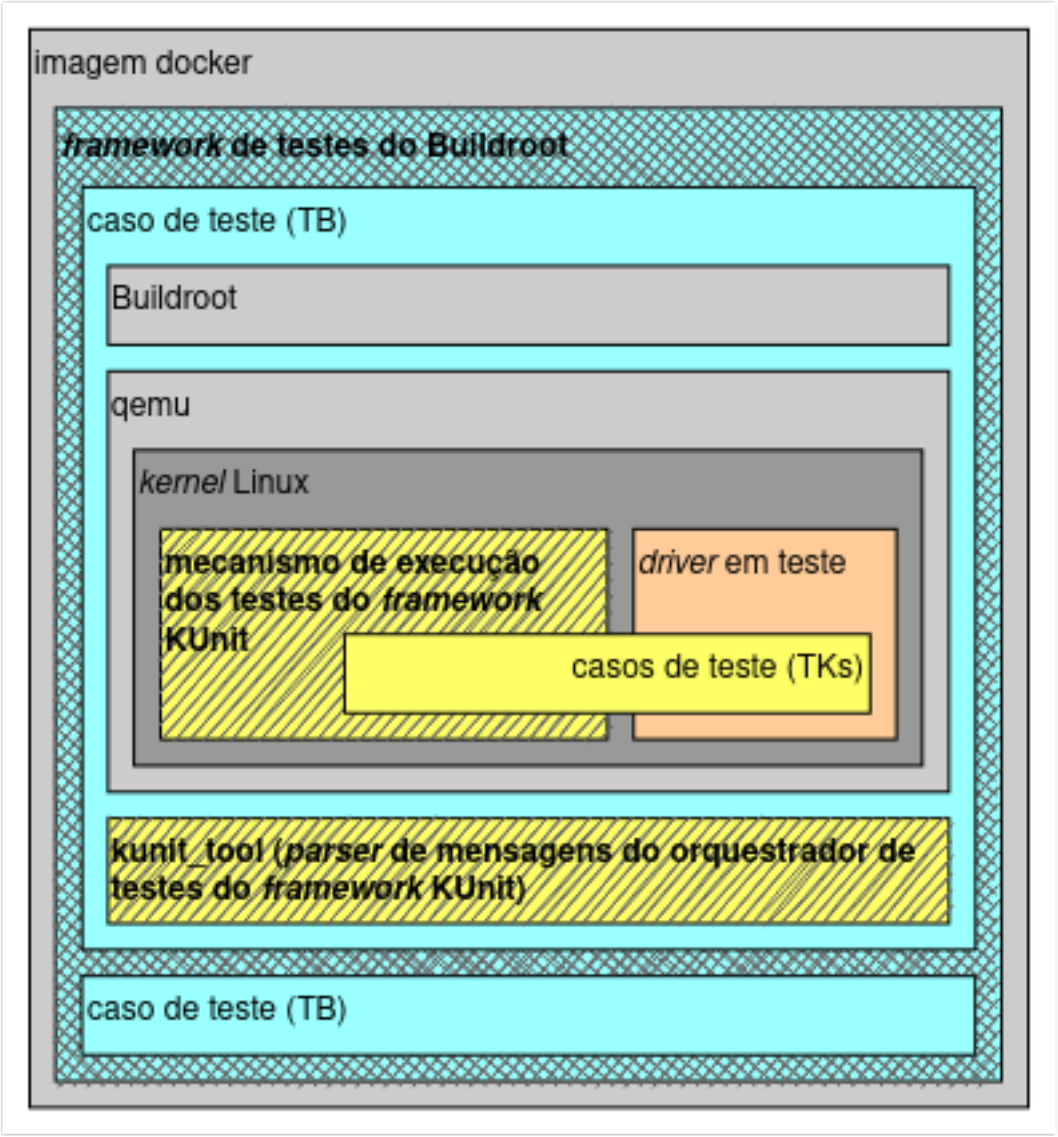
4.4 Combinação das ferramentas selecionadas

Para os objetivos deste trabalho, a proposta é usar um *framework* de teste executando dentro do outro, como mostrado na Figura 4.1, e fazer o *framework* do `Buildroot` expor os resultados do *framework* `KUnit`. Dessa forma, para cada caso de teste no *framework* do `Buildroot`:

1. Usando o `Buildroot`, é gerada uma imagem contendo o *kernel* Linux e o *driver* em teste;
2. A imagem gerada é iniciada em `qemu` e os casos de teste de unidade que estão no *driver* são executados pelo `KUnit` quando o *driver* é inserido no *kernel*;
3. Os resultados dos testes de unidade são coletados do `qemu`, processados pelo `kunit_tool` e exibidos ao desenvolvedor pelo caso de teste externo.

No restante do documento, buscando concisão e a fim de evitar ambiguidades, são usadas as abreviaturas “TB” (com plural TBs), para caso(s) de teste criado(s) usando o *framework* do `Buildroot`, e “TK” (com plural TKs), para caso(s) de teste criado(s) usando o *framework* `KUnit`.

Figura 4.1: *Framework* KUnit (padrão achurado simples) executando dentro do *framework* de testes do Buildroot (padrão achurado duplo)



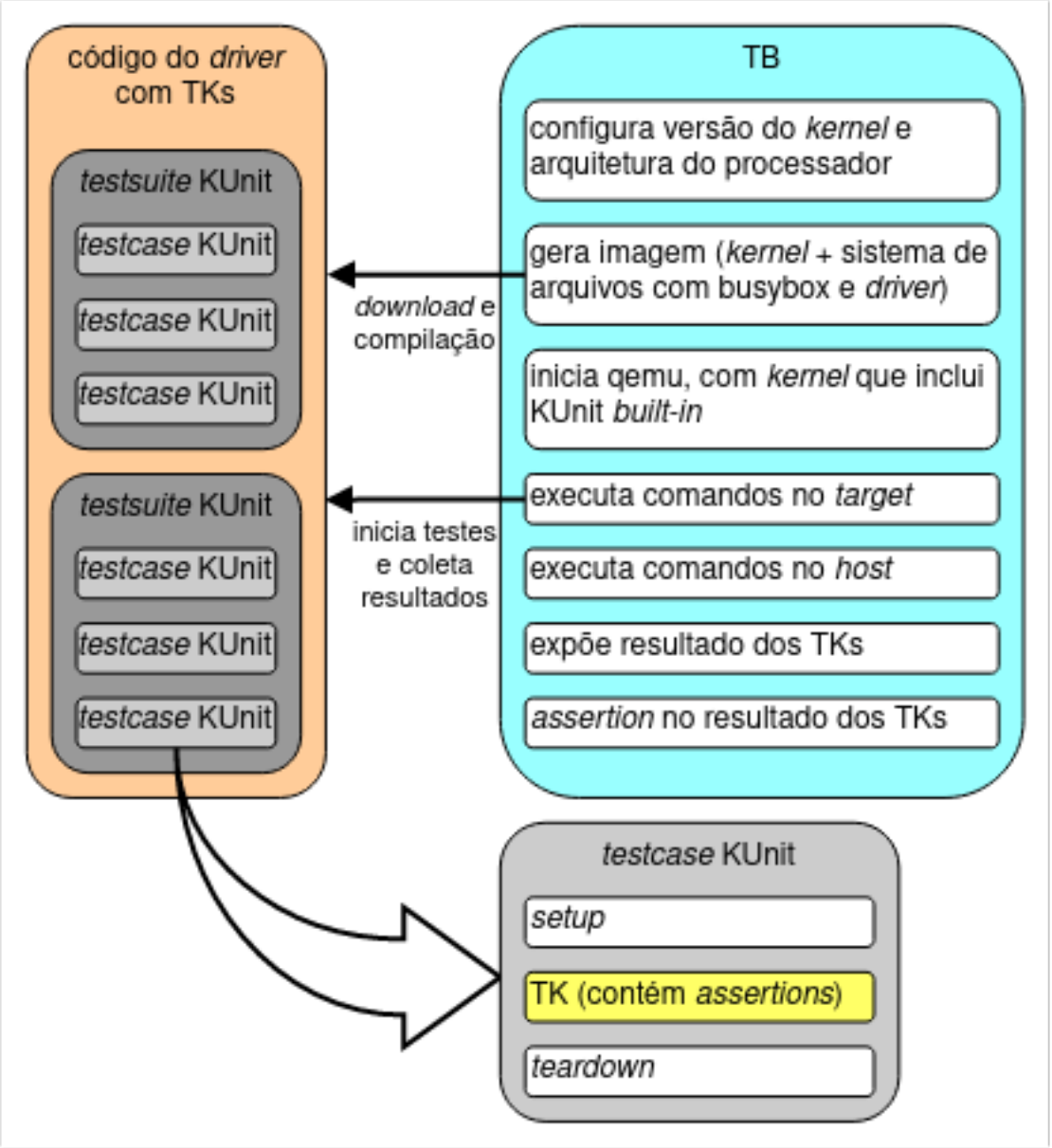
Fonte: O Autor

A Figura 4.2 mostra a relação entre um TB e vários TKs. Os testes de unidade são realizados pelos TKs, enquanto que o TB gera o ambiente desejado para os TKs executarem com uma dada versão de *kernel* em uma arquitetura de processador. Cada TB representa um único caso de teste com um único resultado positivo ou negativo que depende do resultado de todos os TKs que ele executa: se todos os TK tiveram resultado positivo o TB tem resultado positivo, se ao menos um TK gerou resultado negativo ou se houve qualquer outra falha (de compilação por exemplo) o TB gera resultado negativo.

O *framework* de teste do `Buildroot` suporta executar vários TBs. Seria possível, por exemplo:

- Criar vários TBs que contenham os TKs para o mesmo *driver*, variando a arquite-

Figura 4.2: Relação entre TB e TKs



Fonte: O Autor

tura do processador que é simulada pelo `qemu` ou a versão do *kernel* ou a configuração de funcionalidades do *kernel* a serem habilitadas na compilação;

- Criar um TB que testa um conjunto de *drivers*, executando os TKs de cada um deles, ou seja, executar, em um mesmo ambiente, testes de unidade em cada um deles;
- Criar um TB que testa um conjunto de *drivers*, executando TKs especiais que exercitem todos os *drivers* juntos, ou seja, executar testes de integração no conjunto de *drivers*.

Por questão de simplificação, uma vez que o propósito do trabalho é validar a

adequação das ferramentas e não necessariamente exercitar todos os casos suportados, é usado um único TB com vários TKs.

A execução de um TB é composta de duas etapas: primeiro é gerada a imagem a ser usada no teste e então ela é colocada em execução no `qemu` para que os TKs possam executar.

A Figura 4.3 mostra a primeira etapa, em que o TB primeiro configura o `Buildroot` para usar o que essa ferramenta chama de "br2-external", depois configura o que será compilado (com uma sintaxe semelhante à mostrada no segundo comentário da figura), e a compilação inicia. Um "br2-external" contém, entre outras coisas, uma ou mais receitas de compilação (ou seja, a lista de comandos que é usada para compilar um software). No exemplo deste documento o "br2-external" contém apenas a receita para compilar o *driver* a ser testado.

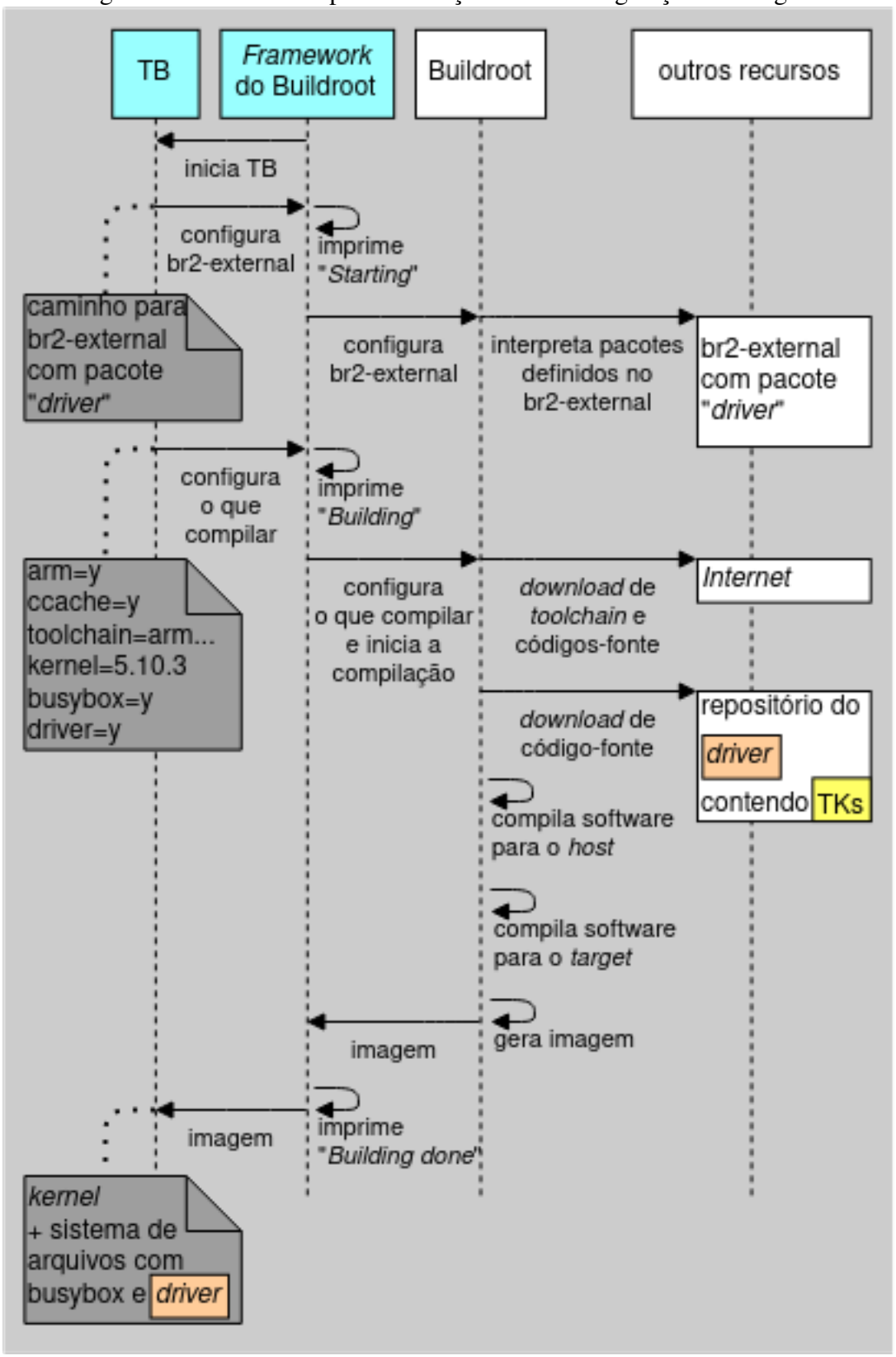
Ainda na primeira etapa, o `Buildroot` busca o *toolchain* e os códigos-fonte da *Internet*, busca o código-fonte do *driver* e compila o software. Finalmente a imagem do *kernel* e a imagem do sistema de arquivos contendo o *driver* e o `busybox` são geradas.

Note que na figura há dois passos de compilação do software: primeiro são compiladas algumas ferramentas para a máquina usada no desenvolvimento (*host*) que serão necessárias para a compilação do software para o *target* ou a geração da imagem. Alguns exemplos disso são o `ccache`, usado na Seção 6.3, `bison` e `flex`, usados na compilação do *kernel*, e o `squashfs`, usado na geração da imagem do sistema de arquivos.

Note também na configuração passada ao `Buildroot`, que aparece no segundo comentário da figura, que foi habilitado não apenas o *kernel* e o *driver*, mas também o `busybox`. Este foi habilitado na compilação para prover para o TB alguns comandos em *user space* comuns em distribuições Linux, por exemplo `modprobe` e `rmmod`, necessários durante a próxima etapa.

A Figura 4.4 mostra a segunda etapa, em que a imagem é iniciada no `qemu` e recebe o comando `modprobe` do TB para inserir o *driver* com TKs no *kernel*. O `KUnit` que já está em execução desde que o *kernel* inicializou, executa cada um dos TKs e guarda o resultado de cada *testsuite* em um arquivo (por exemplo `/sys/kernel/debug/kunit/r8152-getversion-test/results`), em formato semelhante ao mostrado no primeiro comentário da figura. O TB coleta esses resultados e os processa usando o *parser* do `kunit_tool`, gerando um formato mais legível, semelhante ao mostrado no segundo comentário da figura. Finalmente o TB expõe esses resultados dos TKs no formato mais legível e marca seu próprio resultado dependendo do

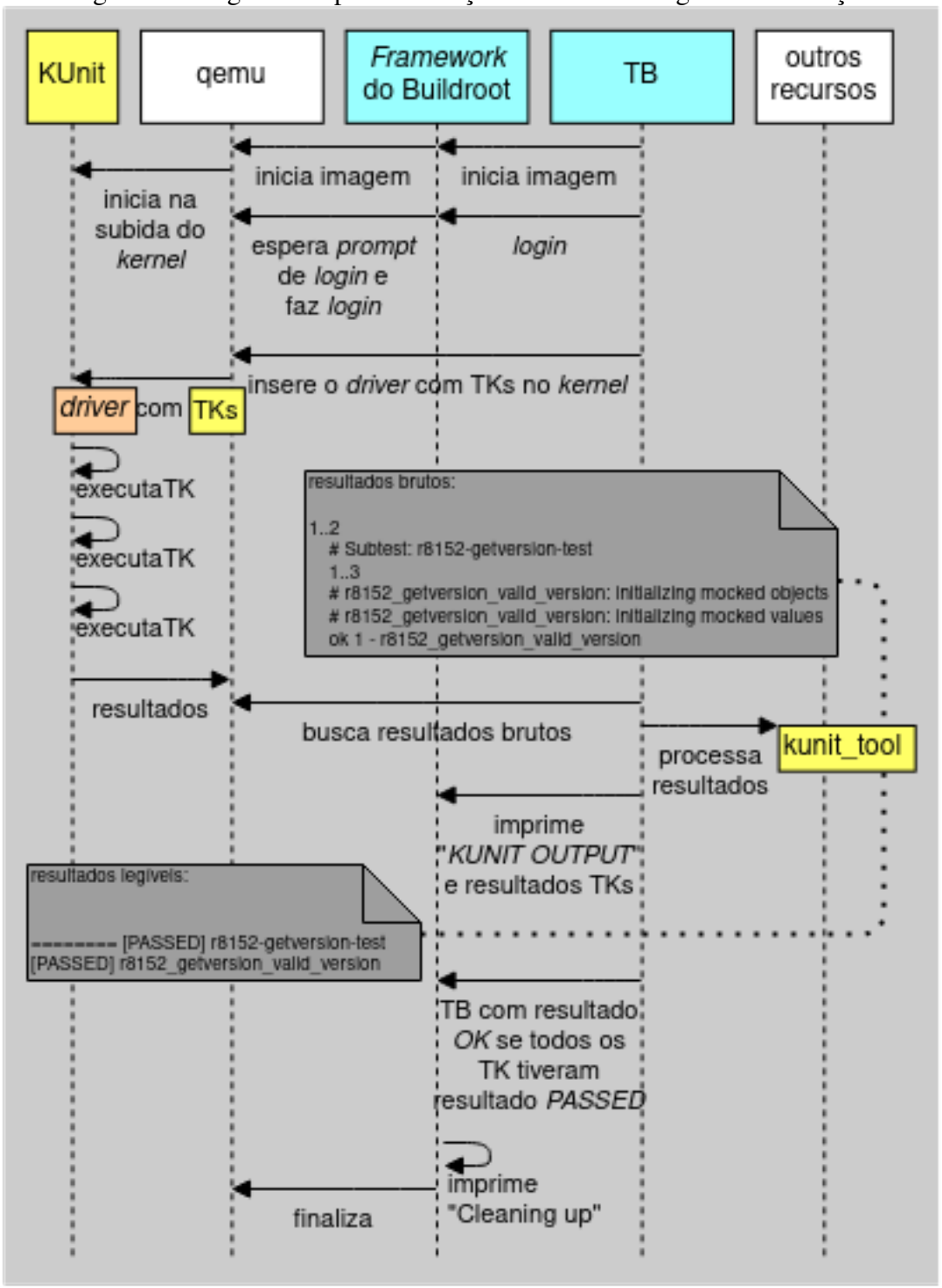
Figura 4.3: Primeira etapa de execução de um TB: geração da imagem



Fonte: O Autor

resultado dos TKs. Todos os TK devem passar para o TB resultar em “*OK*”.

Figura 4.4: Segunda etapa da execução de um TB: imagem em execução



Fonte: O Autor

5 ESTUDO DE CASO

O experimento é realizado partindo da escolha de um *driver*, passando pela montagem de uma infraestrutura de testes e terminando com a criação de um conjunto de testes de unidade para esse *driver*.

O experimento visa exercitar alguns casos de uso de desenvolvimento e manutenção de *drivers out-of-tree* usando testes de unidade para prevenir regressões, listados abaixo:

1. Implementação de novas funcionalidades no *driver*;
2. Implementação de novos casos de teste de unidade;
3. Verificação automática em ambiente de integração contínua para prevenir regressões.

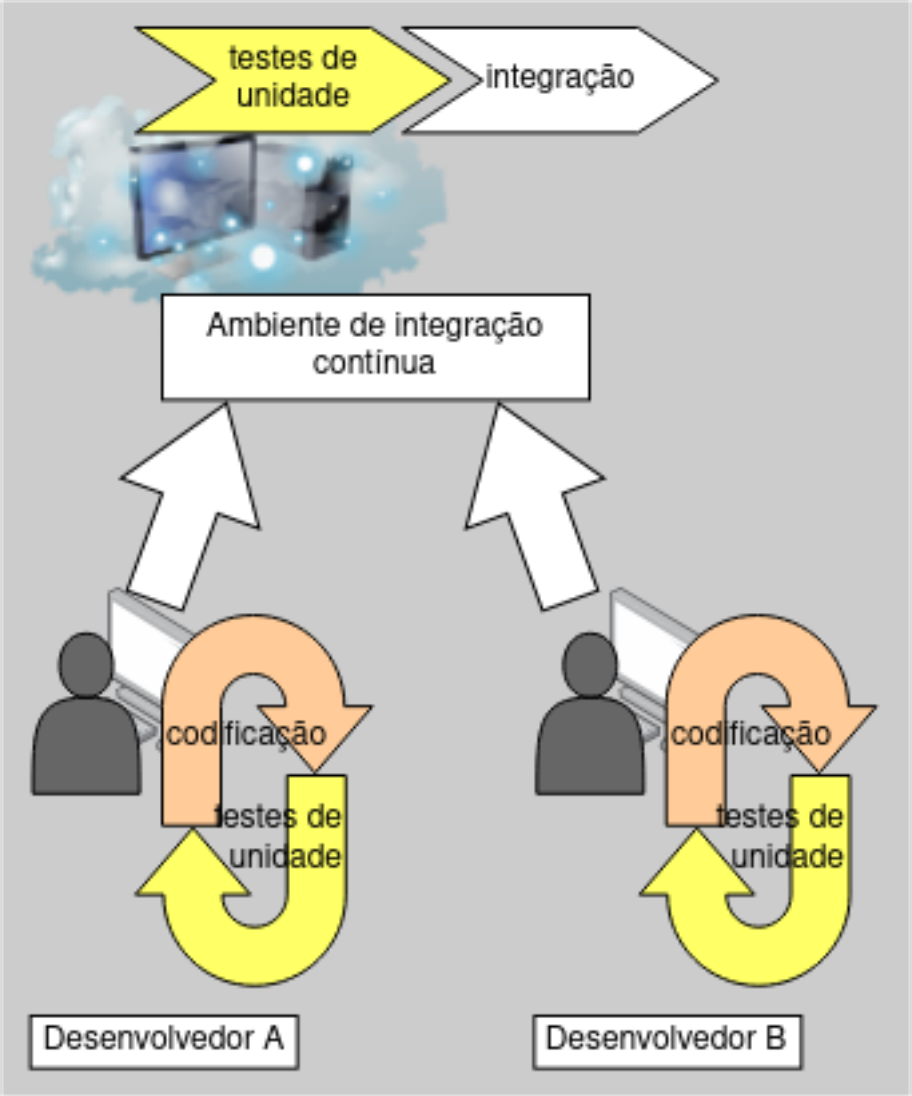
Os dois primeiros casos fazem parte do fluxo normal de desenvolvimento no computador do desenvolvedor. Um alto desempenho da infraestrutura que leve a um reteste rápido é desejável nesses casos, diminuindo assim o tempo gasto pelo desenvolvedor esperando pelo reteste e maximizando o aproveitamento do seu tempo para análise e alteração do código.

O último caso constitui o fluxo comum de integração contínua. Um alto desempenho da infraestrutura é mais importante neste caso principalmente se existirem muitos desenvolvedores que alterem o mesmo repositório, para assim evitar longas esperas para que uma alteração, que passe em todos os testes de unidade, seja integrada automaticamente.

A Figura 5.1 ilustra esses dois cenários principais, que servem como premissa de uso da implementação: o uso no computador do desenvolvedor e o uso no ambiente de integração contínua.

Este capítulo define qual é o *driver* usado como estudo de caso, as razões da escolha e cita um possível uso em sistema embarcado para o dispositivo cujo *driver* é testado. São listadas as etapas da implementação do estudo de caso, realizada de forma incremental. A seguir é descrita a dinâmica interna da implementação, quando em execução, correlacionando com as mensagens que são exibidas ao desenvolvedor, para um exemplo de TB que passa, e, portanto, todos os TKs que ele executou passaram. E finalmente é ilustrada a relação entre uma mudança de código que introduz um erro de comportamento no *driver* e os códigos do *driver* e do TK envolvidos.

Figura 5.1: Ilustração dos dois cenários principais que a implementação visa cobrir



Fonte: O Autor

5.1 Definição do estudo de caso

Na Tabela 5.1 são listados alguns *drivers* que atendem os requisitos listados na Seção 3.3, com a URL[1]e a descrição do uso.

Tabela 5.1: *Drivers open source* que podem servir de estudo de caso

Dispositivo	**Código-fonte do *driver***	**Descrição do uso**
Ralink RT3290	<https://github.com/loimu/rtbth-dkms>	*Driver* do *chipset Bluetooth* usado no *notebook* HP Pavilion
Ralink RTL8153	<https://github.com/wget/realtek-r8152-linux>	*Driver* do adaptador de rede USB 3.0 *Ethernet Gigabit* UE300/UE303 Tp Link

Fonte: O Autor

[1]As URLs foram acessadas em 15 de maio de 2021.

Como mencionado na Seção 2.2, no *kernel* Linux em geral os *drivers* de dispositivos ficam separados das implementações de protocolos. Portanto, o *driver* de um dispositivo *Bluetooth* não reimplementa todo o protocolo de comunicação *Bluetooth*, assim como um *driver* de dispositivo *Ethernet* não reimplementa esse protocolo, mas é responsável pela inserção e remoção do dispositivo, pela programação e desprogramação quando for o caso, e pelas configurações do dispositivo.

Entre os dispositivos listados foi escolhido o **Ralink RTL8153** pois apresenta um custo de aquisição menor, o que pode facilitar trabalhos futuros baseados neste trabalho.

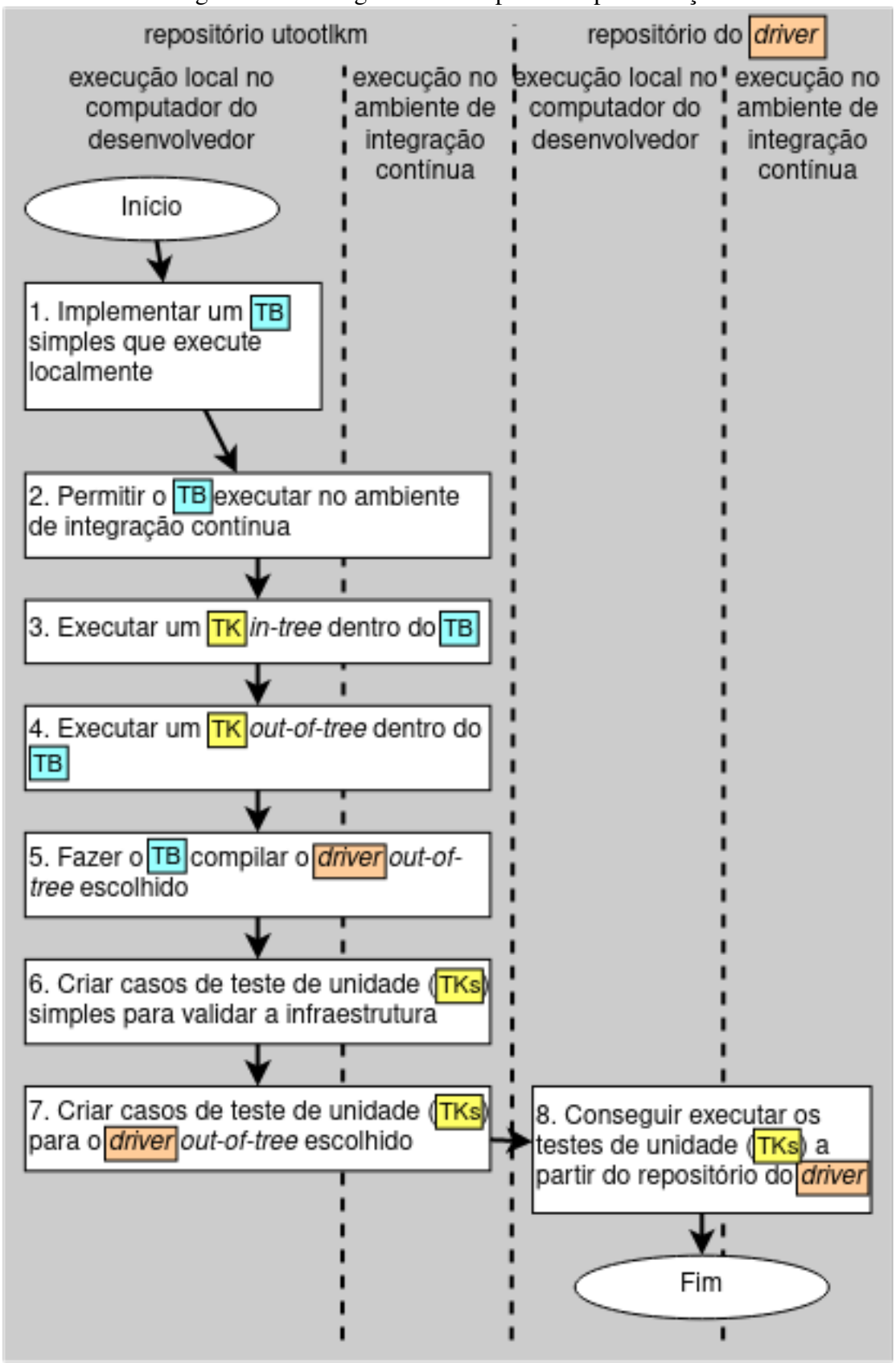
Atualmente já está disponível o acesso a *Internet* em residências com velocidades acima de 100 Mbps e o uso de interfaces *Ethernet* que não são *Gigabit* acaba limitando o acesso de um determinado equipamento a um máximo de 100 Mbps. Um exemplo de aplicação desse dispositivo em um sistema embarcado seria usá-lo montado diretamente no circuito que use um processador que não possua uma interface *Ethernet* ou possua mas que não opere diretamente em modo *Gigabit*, provendo uma interface *Ethernet* que seja capaz de usar toda a banda de *Internet* quando necessário.

5.2 Passos da implementação

A implementação foi realizada de forma incremental, buscando criar gradativamente a ligação entre as duas ferramentas, partindo da camada de fora para a de dentro, mostradas na Figura 4.1. O fluxograma da Figura 5.2 mostra as etapas utilizadas, que são detalhadas no APÊNDICE A. O fluxograma mostra também, para cada etapa, qual repositório contém o código dos testes de unidade (TKs) e a quais casos de uso a etapa buscou cobrir: execução local no computador do desenvolvedor ou execução no ambiente de integração contínua. Essa ordem usada na implementação, de fora para dentro, facilitou o teste de cada passo, pois a cada passo era adicionado pouco código.

A implementação buscou permitir que ambos os casos de uso descritos acima pudessem ser iniciados a partir do repositório do *driver*, mantendo assim o código dos testes de unidade (TKs) junto ao código do *driver* e permitindo que, se mais de um *driver* usasse a mesma infraestrutura, cada um pudesse ter o seu próprio ambiente de integração contínua. Para atingir esse objetivo, primeiramente foi implementado apenas um TB que executasse localmente e depois foi criado o ambiente de integração contínua para ele. Depois o TB foi modificado para que executasse TKs, primeiro alguns que já existem *in-tree* no repositório do *kernel* e então esses mesmos TKs, mas transformados em um

Figura 5.2: Fluxograma das etapas da implementação



Fonte: O Autor

driver out-of-tree. O *driver* escolhido para o estudo de caso foi compilado pelo TB, e então testes de unidade (TKs) foram criados para esse *driver*. Finalmente, os TKs foram movidos para o repositório do *driver*.

5.3 Exemplo da execução de um TB que passa

A Figura 5.3 apresenta as mensagens que são exibidas ao desenvolvedor durante a execução de um TB e abaixo é descrito o que está acontecendo quando as mensagens são geradas.

Quando o desenvolvedor chama `make test` em uma cópia local (procedimento descrito no APÊNDICE C) é iniciada a imagem `docker` (Figura B.8) e dentro dela é iniciada a execução do TB (linhas 35 e 36). Na execução no ambiente de integração contínua a única diferença é que o próprio ambiente primeiro inicia a imagem `docker` (linha 34) para então chamar o mesmo comando que a execução de `make test` em uma cópia local chamaria dentro da imagem (linha 35).

O *framework* de teste do `Buildroot` compila a imagem especificada no TB e a inicia no `qemu` (linhas 37 a 40). O TB insere o *driver out-of-tree* com os TKs no *kernel* que está executando no `qemu`, busca os resultados brutos, os processa com o *parser* do `kunit_tool` e exibe a saída no resultado do TB (linhas 41 a 54). O TB finaliza a execução com o resultado "*OK*" (linhas 55 a 58).

5.4 Exemplo da execução de um TK que falha

As Figuras 5.4 a 5.7 demonstram a relação entre uma mudança de código que introduz um erro no *driver*, as mensagens que são mostradas ao desenvolvedor quando um TK detecta este erro, o código do TK que detecta o erro e o código do *driver* que teve seu comportamento alterado pelo erro introduzido, e que é exercitado pelo TK que detectou o erro.

A Figura 5.4 apresenta um exemplo de mudança no código do *driver* que introduz um erro de comportamento. A Figura 5.5 mostra as mensagens geradas pelo `KUnit` durante a execução do TK que detecta o erro introduzido. A Figura 5.6 apresenta o código do TK que detecta o erro introduzido pela mudança no *driver*. A Figura 5.7 mostra o código do *driver* que teve o comportamento alterado pela introdução do erro, e que é

Figura 5.3: Execução dos testes de unidade criados para o *driver*

```
33 Executing "step_script" stage of the job script
34 Using docker image sha256:0ed25eaa3112bb837580ec7fce53d391d578bc257fcb89
   m:20210525.2139 with digest ricardomartincoski/utootlkm@sha256:3a72462957
   0f3ff ...
35 $ make V=1 test-inside-docker
36 /builds/RicardoMartincoski/utootlkm/buildroot/support/testing/run-tests
   lkm/test-output --download /builds/RicardoMartincoski/utootlkm/download -
37 .13:14:09 TestLinuxKunitQemuR8152                 Starting
38 13:14:11 TestLinuxKunitQemuR8152                 Building
39 13:20:11 TestLinuxKunitQemuR8152                 Building done
40 13:20:11 TestLinuxKunitQemuR8152                 Rebuilding driver unde
41 13:21:31 TestLinuxKunitQemuR8152                 EXPECTING FROM KUNIT:
   d. KUNIT OUTPUT:
42 [13:21:31] ============================================================
43 [13:21:31] ======== [PASSED] r8152-getversion-test ========
44 [13:21:31] [PASSED] r8152_getversion_valid_version
45 [13:21:31] [PASSED] r8152_getversion_invalid_version
46 [13:21:31] [PASSED] r8152_getversion_invalid_device
47 [13:21:31] ============================================================
48 [13:21:31] ======== [PASSED] r8152-probe-test ========
49 [13:21:31] [PASSED] r8152_probe_invalid_version
50 [13:21:31] [PASSED] r8152_probe_fills_device_info
51 [13:21:31] [PASSED] r8152_probe_fills_driver_ops
52 [13:21:31] [PASSED] r8152_probe_fills_driver_ops_depending_on_version
53 [13:21:31] ============================================================
54 [13:21:31] Testing complete. 7 tests run. 0 failed. 0 crashed.
55 13:21:31 TestLinuxKunitQemuR8152                 Cleaning up
56 ----------------------------------------------------------------------
57 Ran 1 test in 442.210s
58 OK
60 Saving cache for successful job
67 Uploading artifacts for successful job
74 Cleaning up file based variables
76 Job succeeded
```

Fonte: O Autor

exercitado pelo TK que detecta o erro.

Esta sequência de figuras demonstra que, ao deliberadamente introduzir um erro no comportamento do *driver*, um TK, que exercita uma parte do código que teve o comportamento alterado pelo erro, falha como esperado. A mensagem gerada pelo `KUnit` indica que a falha foi detectada pela linha 76 do código do TK pois o erro introduzido faz

o código do *driver* erroneamente associar a versão 1 (linha 18302 da Figura 5.7) quando deveria associar a versão 3 (linha 18308 da mesma figura).

Figura 5.4: Exemplo de mudança no *driver* que faz um TK falhar

Fonte: O Autor

Figura 5.5: Mensagens geradas pelo KUnit para uma falha de TK

```
============================================================
======== [FAILED] r8152-getversion-test ========
[FAILED] r8152_getversion_valid_version
    # r8152_getversion_valid_version: initializing mocked objects

    # r8152_getversion_valid_version: initializing mocked values

    # r8152_getversion_valid_version: EXPECTATION FAILED at ../realtek-r8152-linux/r8152-test.c:76
    Expected rtl_get_version(intf) == (u8) RTL_VER_03, but
        rtl_get_version(intf) == 1
        (u8) RTL_VER_03 == 3
    not ok 1 - r8152_getversion_valid_version

[PASSED] r8152_getversion_invalid_version
```

Fonte: O Autor

Figura 5.6: Código do TK que detectou o erro introduzido

```
65  static void r8152_getversion_valid_version(struct kunit *test)
66  {
67          struct usb_interface *intf = test->priv;
68
69          mock.usb_control_msg.ocp_data = 0x4c00;
70          KUNIT_EXPECT_EQ(test, rtl_get_version(intf), (u8) RTL_VER_01);
71
72          mock.usb_control_msg.ocp_data = 0x4c10;
73          KUNIT_EXPECT_EQ(test, rtl_get_version(intf), (u8) RTL_VER_02);
74
75          mock.usb_control_msg.ocp_data = 0x5c00;
76          KUNIT_EXPECT_EQ(test, rtl_get_version(intf), (u8) RTL_VER_03);
77  }
```

Fonte: O Autor

Figura 5.7: Código do *driver* que teve o comportamento alterado pela introdução do erro

```
18295          if (ret > 0)
18296                  ocp_data = (__le32_to_cpu(*tmp) >> 16) & VERSION_MASK;
18297
18298          kfree(tmp);
18299
18300          switch (ocp_data) {
18301          case 0x4c00:
18302                  version = RTL_VER_01;
18303                  break;
18304          case 0x4c10:
18305                  version = RTL_VER_02;
18306                  break;
18307          case 0x5c00:
18308                  version = RTL_VER_03;
18309                  break;
```

Fonte: O Autor

6 RESULTADOS

6.1 Dificuldades enfrentadas e simplificações adotadas

O *framework* de testes do `Buildroot` ainda não suporta TB de fora de seu repositório. Para contornar isso, foi usada uma cópia dinâmica do TB que está no `utootlkm` para dentro de seu submódulo. Quando o desenvolvedor chama `make test` o TB é copiado para dentro do diretório em que o *framework* o espera, o TB é executado e depois a cópia é apagada.

Na criação do ambiente de integração contínua do TB, buscou-se reusar a imagem `docker` que a comunidade do `Buildroot` já usa para executar seus testes. Não foi possível continuar a usar essa imagem ao passar a chamar o *parser* do `kunit_tool` de dentro do TB, pois este necessita de `Python` em versão maior ou igual a `3.6`. Para solucionar isso foi criada uma nova imagem `docker` (Figura B.8) com base na imagem usada pelo `Buildroot` e publicada usando o serviço gratuito do `DockerHub`. Para aumentar a reprodutibilidade foi mantido no repositório `utootlkm` o arquivo `Dockerfile` que gera essa imagem a partir de uma versão fixa de uma imagem de uma distribuição (`debian:buster-20210511`).

O `Makefile` do *driver* escolhido possui dois caminhos principais de execução. O caminho padrão não é adequado para compilação cruzada pois assume que o *driver* está sendo compilado para o *host* e executa comandos no *host* para já remover (`rmmod`) e inserir (`modprobe`) o *driver* no computador em que está sendo realizada a compilação. Para usar o outro caminho, que apenas compila o *driver*, é necessário forçar o valor da variável `KERNELRELEASE`. Para os comandos criados para executar o teste (`make test`) é suficiente apenas usar um valor que faça esse outro caminho executar (`make KERNELRELEASE=0 test`). Isso foi feito buscando limitar as alterações no `Makefile` original do *driver* escolhido.

Os "*Shared Runners*" são máquinas virtuais que o `GitLab` provê gratuitamente para os projetos que ele hospeda executarem compilações, testes e integração contínua, eles iniciam a imagem `docker` configurada por repositório com uma linha de comando que não é configurável. Ao mover o TB e os TKs para o repositório do *driver* (último passo descrito na Seção 5.2), o `docker` acaba sendo iniciado com apenas o submódulo `utootlkm` compartilhado, o que faz com que a receita do *driver* não consiga acessar o código-fonte do *driver* com TKs diretamente do diretório acima deste, o que, por sua

vez, dificultaria o uso da solução para o desenvolvimento do *driver* e de TKs pois seria necessário primeiro fazer a publicação de uma modificação no repositório do *driver* para depois esta ser testada. Para contornar isso, foi usada uma cópia dinâmica do código-fonte do *driver* com TKs. Quando o desenvolvedor chama `make KERNELRELEASE=0 test` o código-fonte é copiado para dentro do diretório que já é compartilhado com o `docker` e a receita do *driver* foi adaptada para usar essa cópia para compilar o *driver*.

Os "*Shared Runners*" do `GitLab` são gratuitos para projetos *open source* mas possuem um limite de horas mensal. Foi usado apenas um TB por simplicidade pois atendia o propósito do trabalho, mas também para limitar o número de horas usadas. Foram feitas várias melhorias de performance na infraestrutura, sumarizadas na Seção 6.3, para facilitar o uso durante o desenvolvimento e manutenção do *driver* com TKs, mas também para limitar o número de horas usadas.

O `KUnit` possui vários recursos que não foram explorados durante a implementação. Para limitar o escopo do trabalho, na criação de TKs foram usadas, por simplicidade, comparações (*expectations* / *assertions*) simples, apesar de, por exemplo, existirem variantes que permitem gerar uma mensagem customizada para cada possível falha.

A documentação do `KUnit` recomenda para fazer *mock* de funções usar indireção ou definições condicionadas à compilação dos testes de unidade. O *driver* escolhido não foi desenvolvido pensando em ser testado com testes de unidade. Para limitar as alterações necessários no *driver* para testá-lo, optou-se pelo uso de definições condicionadas à compilação (diretiva de compilador `ifdef`) ao invés de usar indireção.

A documentação do `KUnit` recomenda, para poder testar funções estáticas sem precisar expor os símbolos dessas funções (ou seja, transformá-las em não estáticas), o uso de inclusão do arquivo que contém os TKs no arquivo do *driver* de forma condicionada à compilação dos testes de unidade (diretiva de compilador `include` condicionada por diretiva de compilador `ifdef`). Para limitar as alterações necessários no *driver* para testá-lo, optou-se por fazer o inverso, incluindo o arquivo do *driver* na compilação do arquivo que contém os TKs. O resultado disso é que um arquivo do tipo `.c` é incluído em outro arquivo do tipo `.c`.

Para os TKs criados foi usado um *mock* do dispositivo que responde apenas o suficiente para que as duas funções escolhidas possam ser testadas: a leitura de um único registrador que indica a versão do hardware. Para testar outras funções provavelmente seria necessário estender o mecanismo criado para que suporte respostas variadas dependendo de qual registrador está sendo acessado.

Seria tecnicamente possível simular todo o comportamento do dispositivo criando um modelo dele no `qemu`, mas seria difícil de controlar a resposta durante o teste de unidade, e portanto o modelo deveria ser razoavelmente completo para ser útil. Para evitar essa complexidade extra de implementação optou-se por usar o `qemu` apenas para simular a arquitetura do processador com os modelos de dispositivos já existentes e prover um *kernel* vivo para o `KUnit` executar.

6.2 A adequação das ferramentas

A combinação escolhida de ferramentas atende a quase todos os critérios listados na Seção 3.2 e na Tabela 4.3.

O `KUnit`:

- permite testes em *kernel space*;
- permite a orquestração de execução dos TKs;
- permite *assertions* nos TKs;
- permite *mock* de funções pois executa em *kernel space*.

O *framework* de testes do `Buildroot` através de seu código que usa `Python` e os módulos `nose2` e `unittest`:

- permite a orquestração de execução dos TBs.
- permite *assertions* nos TBs.

O TB através do uso do próprio `Buildroot`:

- compila *kernel*;
- compila *driver out-of-tree*;
- realiza compilação cruzada.

O TB por usar `qemu` para simular o *target*:

- permite testes de aplicações em *user space*, funcionalidade usada pelo TB para inserir o *driver* com TKs no *kernel* e coletar informações para depuração;
- permite testes da interface que o *kernel* expõe ao *user space*, funcionalidade usada pelo TB para coletar os resultados dos TKs;
- leva em conta *endianness*;
- leva em conta tamanho de palavra do processador;

- permite executar um *kernel* vivo;
- permite simulação em nível de instrução de várias arquiteturas de processador;
- usa modelos de dispositivos.

Os TKs por executarem dentro de um *kernel* vivo no `qemu`:

- levam em conta *endianness*;
- levam em conta tamanho de palavra do processador.

O uso dessa combinação de ferramentas permite controlar as interfaces do código testado e observar os estados internos, atendendo aos requisitos listados na Seção 3.1.

Uma consequência de como essas ferramentas funcionam é que, se a ferramenta de simulação `qemu` suportar a arquitetura semelhante o suficiente a do hardware final, é possível usar no teste o mesmo *toolchain* que é usado para o hardware final (*target*), mantendo assim a versão do compilador e as opções passadas para este, por exemplo as opções de otimização de código, iguais entre o binário testado por testes de unidade e o binário usado no hardware final.

Apesar de não ter sido encontrada ferramenta *open source* que possibilite a simulação de um modelo para o dispositivo cujo *driver* está sendo testado que seja facilmente controlado pelo código do teste de unidade, é possível escrever *mocks* para acessos específicos ao dispositivo. Esses *mocks* não deixam de ser um modelo simplificado de parte do comportamento do dispositivo.

6.3 Desempenho da infraestrutura

Como mostra a Figura 6.1, com menos de mil linhas de código foi possível combinar duas ferramentas *open source* para executar testes de unidade em um *driver out-of-tree* em um cenário próximo ao cenário real.

Como ilustrado no APÊNDICE B, é possível usar essa pequena infraestrutura criada e publicada com licença *open source* para criar um ambiente de integração contínua para um *driver out-of-tree*.

O tempo médio para execução completa dos testes (que inclui a compilação da imagem para o `qemu`) é otimizado da seguinte forma:

- Para uso local, o `Buildroot` já cria uma *cache* de *download* do código-fonte do *kernel*, de comandos em *user space* e de software usado durante a compilação. Essa

Figura 6.1: Comandos mostrando o tamanho da implementação descrita na Seção 5.2

```
utootlkm$ git checkout gitlab/driver_repo_ci
HEAD is now at 8e44bb6 .gitignore: the files copied from parent repo
utootlkm$ git ls-files | xargs wc -l 2>/dev/null
    8 .gitignore
    5 .gitmodules
    1 Config.in
  280 LICENSE
   51 Makefile
   33 README.md
    8 docker-run
    2 external.desc
    1 external.mk
   15 package/realtek-r8152-linux/Config.in
   34 package/realtek-r8152-linux/realtek-r8152-linux.mk
   77 support/docker/Dockerfile
    0 support/testing/tests/__init__.py
  110 support/testing/tests/test_linux_kunit.py
   71 support/testing/tests/test_linux_kunit_r8152.py
  696 total
utootlkm$
```

```
realtek-r8152-linux$ git checkout gitlab/driver_repo_ci
HEAD is now at 74e6dec Avoid running undesirable commands on host
realtek-r8152-linux$ git diff --stat=40 gitlab/master gitlab/driver_repo_ci
 .gitlab-ci.yml |  21 ++
 .gitmodules    |   4 +
 .utootlkm      |   1 +
 Makefile       |  20 ++
 ReadMe.txt     |  10 +
 r8152-test.c   | 200 ++++++++++++++++++
 r8152.c        |  24 +-
 7 files changed, 272 insertions(+), 8 deletions(-)
realtek-r8152-linux$
```

Fonte: O Autor

cache foi configurada na infraestrutura criada para ser armazenada dentro do diretório principal da infraestrutura. O ambiente de integração contínua foi configurado para guardar esse diretório de *cache* de *downloads* de uma execução para outra.

- O `Buildroot`, antes de compilar uma imagem, testa a versão de alguns comandos e de algumas ferramentas do *host* e os usa se tiverem em versões adequadas. Para o `Buildroot` que está sendo usado na infraestrutura criada, o *host* é a imagem `docker`. Na imagem `docker` publicada foi instalada a versão de `cmake` maior que `3.15` para evitar a compilação dessa ferramenta para o *host* a cada execução do teste.
- O TB foi configurado para habilitar o uso de `ccache`, que acelera muito a recompilação de arquivos quando houve poucas mudanças, e guardar essa *cache* de compilação dentro do diretório principal da infraestrutura. O ambiente de integração contínua foi configurado para guardar esse diretório de *cache* de compilação de

uma execução para outra. Comparado com o código do *kernel*, que no TB é fixo em uma versão, e contém milhões de linhas, o código do *driver*, que contém apenas milhares de linhas, é pequeno; portanto a taxa de acerto dessa *cache* é alta.

- O `Buildroot` é capaz de gerar um *toolchain* contendo `gcc`, `libc`, etc. para então usá-lo para fazer a compilação cruzada do software para o *target*, mas isso é um processo demorado (em geral mais de 30 minutos). O TB foi configurado para usar um *toolchain* pré-compilado que é disponibilizado pela comunidade do `Buildroot`, evitando assim a compilação do *toolchain* a cada execução do teste.
- Durante o primeiro uso, em um dado computador, da infraestrutura criada o `docker` faz o *download* da imagem publicada e armazena em uma *cache* própria, nos usos subsequentes essa imagem é reusada. O ambiente de integração contínua criado como exemplo utiliza o que o `GitLab` chama de "*Shared Runners*", que são máquinas virtuais que podem ser usadas por projetos *open source* gratuitamente por até um determinado número de horas por mês, e eles já implementam uma *cache* própria de imagens `Docker`.
- O uso do `DockerHub` também possui um limite mensal de consumo de banda de *download* quando uma imagem é armazenada gratuitamente. Devido ao item anterior esse limite não deve ser excedido, e ainda o tempo de execução de novos testes não é penalizado pois o *download* dessa imagem é feito apenas uma vez.

Na Tabela 6.1 é feita uma comparação entre os tempos, mostrando que o tempo para reexecutar o TB pode ter ganho de até 70% com as *caches* mencionadas.

Tabela 6.1: Comparação de tempo de execução do TB com relação ao uso das *caches*

Taxa de acerto das *caches* usadas	**Tempo total**	**URL da amostra**
0%	28 min 39 s	<https://gitlab.com/RicardoMartincoski/utootlkm/-/jobs/1333512588>
100%	8 min 35 s	<https://gitlab.com/RicardoMartincoski/utootlkm/-/jobs/1335400717>
0%	28 min 33 s	<https://gitlab.com/RicardoMartincoski/realtek-r8152-linux/-/jobs/1336824826>
100%	8 min 33 s	<https://gitlab.com/RicardoMartincoski/realtek-r8152-linux/-/jobs/1336769889>

Fonte: O Autor

A imagem `docker` também contém, herdados da cópia do arquivo `Dockerfile` do `Buildroot`, vários comandos como `gcc`, `qemu`, `Python` e `nose2`. Isso torna o ambiente da infraestrutura mais reprodutível, uma vez que não possui muitas dependên-

cias com o computador do desenvolvedor, apenas uma versão recente dos comandos `git`, `make` e `docker`, e dentro da imagem do `docker` as versões de todos os comandos e ferramentas são fixas, independente de essa imagem ser regerada em outro computador pois no `Dockerfile` foi usada uma versão com data fixa como base.

A escolha das ferramentas, todas *open source*, e dos serviços disponíveis *online* na *Internet*, todos com planos gratuitos para um uso razoável por projetos *open source*, faz o custo inicial de uma solução semelhante ser zero para projetos *open source*.

Para adaptar a solução para testar outro *driver out-of-tree* seria necessário criar poucos arquivos. Seria necessário alterar dois repositórios. No repositório `utootlkm`: criar um novo "pacote" `Buildroot` com a receita para compilar o novo *driver* e copiar ou alterar o TB ajustando o nome do *driver* e o nome do "pacote". No repositório do novo *driver*: adicionar o `utootlkm` como submódulo, alterar o `Makefile` e escrever pelo menos um TK.

6.4 Limitações da infraestrutura e do escopo do experimento

As ferramentas *open source* de simulação de processador analisadas não simulam a temporização do hardware com precisão. Portanto os testes realizados com essas ferramentas são úteis para verificar a característica funcional do comportamento, mas não a característica temporal.

Por usar o `KUnit` a solução pode ser apenas usada com versões recentes de *kernel*, a partir da `5.5`. Não foi avaliada a viabilidade e/ou esforço para fazer *backport* da implementação para versões anteriores.

As versões de *kernel* também estão limitadas à lista que o `Buildroot` suporta compilar. Em teoria todas as versões de *kernel* até o lançamento de uma dada versão de `Buildroot` devem ser suportadas por essa versão, mas para versões futuras de *kernel* é possível que seja necessário atualizar a versão de `Buildroot` para uma versão futura para dar suporte.

As arquiteturas de processador que são suportadas nos testes estão limitadas à lista que ambos o `qemu` e o *framework* de testes do `Buildroot` suportem.

O *driver* usado como estudo de caso é relativamente simples, consistindo de apenas um arquivo do tipo `.c` de 19 mil linhas. Não foi avaliado se há esforço extra para testar *drivers* que tenham o código organizado em diversos arquivos e diretórios.

Para dar suporte a testar um novo *driver* com a infraestrutura atual é necessário

modificar o `utootlkm`, não bastando adicioná-lo como submódulo de um repositório de *driver*. Deve ser tecnicamente possível mover tanto a receita de compilação do *driver* quanto o TB para o repositório do *driver*, mesmo que para isso seja usada cópia dinâmica de arquivos, mas não foi avaliado o esforço dessa implementação pois não era necessária para a validação da adequação das ferramentas.

Não é possível usar a infraestrutura para testar *drivers* proprietários pois os símbolos do `KUnit` só estão disponíveis para *drivers* que se declaram *open source*. Por exemplo, se um *driver* contém a chamada da macro `MODULE_LICENSE("Proprietary")`, a chamada da macro `kunit_test_suites`, usada para registrar TKs para execução, acaba tentando usar o símbolo exportado pelo *kernel* usando a macro `EXPORT_SYMBOL_GPL(__kunit_test_suites_init)` e o *kernel* recusa esse uso.

Como citado anteriormente seria possível criar vários TBs para testar os mesmos TKs em variadas arquiteturas de processador e versões de *kernel*. Mas dependendo de quantas combinações forem usadas isso pode trazer um desempenho insatisfatório para o desenvolvimento dos *drivers*. Seria necessário nesse caso usar, por exemplo, alguma técnica de particionamento do espaço de entradas do teste, considerando as arquiteturas e as versões de *kernel* como entradas, e/ou dividir os testes em conjuntos, um que seja usado para validar cada entrega, evitando assim a entrada de regressões, e outro que seja executado periodicamente mas que detecta regressões apenas após a integração delas.

Não foi avaliada a adequação das ferramentas para testar funções que executem em contexto de interrupção de hardware.

6.5 Comparação com trabalhos relacionados

A Tabela 6.2 repete as características dos trabalhos relacionados, listadas na Tabela 2.2, adicionando uma coluna para realizar a comparação com este trabalho.

Como analisado na Seção 6.2, este trabalho atendeu a quase todos os requisitos listados para as ferramentas de automação de teste (Seção 3.2) e a todos os requisitos listados para a automação do teste (Seção 3.1). Apenas a qualidade teórica do teste (proximidade com o cenário real) acabou ficando subótima, mas mesmo assim já ficou acima da dos trabalhos relacionados que usaram ferramentas *open source*. E este trabalho também publicou a solução como *open source* para que o experimento possa ser reproduzido de forma determinística e sem depender de licenças de ferramentas comerciais.

Tabela 6.2: Comparação de características entre este trabalho e os trabalhos relacionados

Trabalho	Bastien et al. (2004)	Maier and Kleeberger (2016)	Gomes (2010)	Mononen (2020)	Åhman (2017)	Witkowski et al. (2007)	Este trabalho
Usa ferramentas comerciais	X						
Usa ferramentas que não são *open source*						X	
Usa ferramentas *open source*				X	X	X	X
Define uma metodologia de criação de testes		X	X	X			
Cria ferramenta que não é *open source*					X	X	
Cria ferramenta *open source*							X
Executa testes no hardware real				X			
Usa simulação do hardware como um todo	X	X					
Usa simulação do processador							X
Usa *stubs*, *mocks* ou modelos no *host*			X	X	X	X	
Leva em conta *endianness*	X	X				X	X
Leva em conta tamanho de palavra do processador	X	X	X			X	X
Testa software sem sistema operacional	X	X	X	X			
Testa software com sistema operacional					X	X	X
Não menciona linguagem do software testado	X	X					
Software testado escrito em `C++`			X	X			
Software testado escrito em `C`					X	X	X
Solução poderia testar software escrito em `C`	X	X	X	X	X	X	X
Teste de injeção de falhas	X	X					
Teste de unidade		X	X	X	X		X
Teste baseado em modelo do dispositivo	X		X				
Teste baseado em modelo abstrato do *driver*						X	
Reprodução de experimento não depende de licenças de ferramentas comerciais			X		X	X	X
Trabalho descreve como reproduzir o experimento de forma determinística			X				X

Fonte: O Autor

7 CONCLUSÃO

Embora o uso de testes automatizados de software já seja o padrão da indústria de software de uma forma geral, alguns contextos como o teste de software embarcado, principalmente de suas camadas mais dependentes do hardware, possuem especificidades às quais a maioria das ferramentas de automação de teste não se adequam. Muitas vezes o código de *drivers* de dispositivos do *kernel* Linux é testado apenas em nível de sistema. Os *drivers out-of-tree* possuem ainda mais limitações para serem testados pois várias ferramentas que são distribuídas com o *kernel* se adequam apenas ao contexto de *drivers in-tree*.

Buscando avaliar a adequação de ferramentas *open source* para criar testes contra regressões durante o desenvolvimento e manutenção de *drivers out-of-tree*, neste trabalho foram levantados os requisitos da automação do teste e das ferramentas para esta automação. Ferramentas *open source* foram pesquisadas e analisadas contra esses requisitos, buscando usar um cenário o mais próximo do real para executar testes de unidade. Algumas ferramentas foram selecionadas e aplicadas a um *driver open source* que serviu como estudo de caso.

Foi realizada a implementação de uma pequena infraestrutura que usa o *framework* de testes do `Buildroot` para gerar um ambiente em que os testes de unidade criados para o *driver* sejam executados pelo `KUnit`, e avaliada a adequação dessas ferramentas aos requisitos levantados previamente. Com essa implementação e o uso de serviços gratuitos foi gerado um ambiente de integração contínua de exemplo, publicado com licença *open source* para facilitar a reprodução dos resultados desse trabalho.

Foi possível demonstrar que, combinando duas ferramentas *open source* (uma que não é um *framework* de testes mas contém um, e a outra bem recente, publicada em 2018 e aceita no *kernel* em 2019), é possível executar testes de unidade em um *driver out-of-tree* em um ambiente próximo ao do cenário real, apesar de essa proximidade com o cenário real ser subótima. Portanto os objetivos desse trabalho, definidos no início do documento, foram alcançados.

A elaboração deste trabalho, além da aplicação de conhecimentos técnicos, trouxe e reforçou outros aprendizados: é possível atingir uma alta taxa de reúso com um pouco mais de planejamento e estudo de soluções, evitando assim "reinventar a roda" ao tentar criar uma solução do zero por não encontrar uma que se adeque imediatamente ao contexto; o surgimento de novas ferramentas é constante e é importante sempre reavaliar a

lista disponível, pois um trabalho com os mesmos objetivos que este poderia ter resultado diferente se tivesse sido feito há alguns anos ou se for feito daqui alguns anos; existem vários serviços *online* com planos gratuitos de uso, às vezes limitados a projetos com algum tipo de licença ou com algum limite de quantidade de uso, que podem facilitar muito a criação de soluções e validação de ideias; e ao reusar soluções é importante ter cuidado com as licenças, não apenas para não infringi-las ao criar a solução derivada, mas também para que quem reusar a solução derivada não acabe infringindo-as.

Como trabalhos futuros, seria possível: usar um modelo do dispositivo para testar o *driver*, talvez fazendo o `Buildroot` recompilar o `qemu` com o modelo e usar essa versão ao invés da versão pré-instalada no `docker` para executar os testes, talvez criando esse modelo como um outro *driver* usado apenas durante o TK e controlado por este; passar a medir a cobertura de código dos testes de unidade; reestruturar o código do *driver* para que seja mais fácil criar testes de unidade com grande cobertura de código; ou avaliar a viabilidade de usar a combinação de ferramentas com versões mais antigas de *kernel*.

REFERÊNCIAS

BAJER, M.; SZLAGOR, M.; WRZESNIAK, M. Embedded software testing in research environment. a practical guide for non-experts. In: **2015 4th Mediterranean Conference on Embedded Computing (MECO)**. Piscataway: IEEE, 2015. p. 100–105. ISBN 9781479919765.

BASTIEN, B. et al. **A Technique for Performing Fault Injection in System Level Simulations for Dependability Assessment**. Dissertation (Master) — University of Virginia, 2004. Disponível em: <https://www.researchgate.net/publication/249896898_A_Technique_for_Performing_Fault_Injection_in_System_Level_Simulations_for_Dependability_Assessment>. Acesso em: 04 fev. 2021.

GALIN, D. Software quality: Concepts and practice. In: ____. John Wiley & Sons, Inc., 2018. chp. Software Testing, p. 255–317. ISBN 9781119134527.

GAROUSI, V.; AMANNEJAD, Y.; CAN, A. B. Software test-code engineering: A systematic mapping. **Information and Software Technology**, Elsevier B.V., v. 58, p. 123–147, fev. 2015. ISSN 09505849.

GAROUSI, V. et al. What we know about testing embedded software. **IEEE Software**, IEEE Computer Society, v. 35, p. 62–69, jul. 2018. ISSN 07407459.

GAROUSI, V. et al. What industry wants from academia in software testing? hearing practitioners' opinions. In: **Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering**. New York: Association for Computing Machinery, 2017. p. 65–69. ISBN 9781450348041.

GOMES, H. V. **Metodologia de Projeto de Software Embarcado Voltada ao Teste**. 104 p. Dissertation (Master) — Universidade Federal do Rio Grande do Sul, 2010. Disponível em: <http://www.lume.ufrgs.br/handle/10183/27671>. Acesso em: 04 fev. 2021.

HIGGINS, B. **KUnit - Unit Testing for the Linux Kernel**. linux.conf.au, 2020. Disponível em: <https://www.youtube.com/watch?v=78gioY7VYxc>. Acesso em: 15 fev. 2021.

KANG, B.; KWON, Y.-J.; LEE, R. A design and test technique for embedded software. In: **Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA'05)**. Los Alamitos: IEEE Computer Society, 2005. p. 160–165. ISBN 0769522971.

LIU, C. H.; CHEN, S. L.; HUANG, T. C. A model-based testing tool for embedded software. In: **2012 Sixth International Conference on Genetic and Evolutionary Computing**. IEEE Computer Society, 2012. p. 180–183. ISBN 9780769547633.

MAIER, P. R.; KLEEBERGER, V. B. Embedded software reliability testing by unit-level fault injection. In: **2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)**. IEEE, 2016. p. 410–416. ISBN 9781467395694.

MONONEN, S. **Evaluation of Test-Driven Approaches for Embedded Software Development**. Dissertation (Master) — Aalto University, jul. 2020. Disponível em: <http://urn.fi/URN:NBN:fi:aalto-202008235020>. Acesso em: 04 fev. 2021.

PETAZZONI, T. et al. **Linux Kernel and Driver Development Training**. 2021. Disponível em: <https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf>. Acesso em: 15 fev. 2021.

SKRZYPIEC, P.; MARSZAŁEK, Z. Linux kernel driver for external analog-to-digital and digital-to-analog converters. In: **2020 27th International Conference on Mixed Design of Integrated Circuits and System (MIXDES)**. IEEE, 2020. p. 255–259. ISBN 9781728197814.

SUNG, A.; CHOI, B. An interaction testing technique between hardware and software in embedded systems. In: **Ninth Asia-Pacific Software Engineering Conference, 2002.** IEEE, 2002. p. 457–464. ISBN 9780769518503.

UBM TECHNOLOGY. **Embedded market study**. 2013. Disponível em: <http://www.iuma.ulpgc.es/~nunez/UBM2013EmbeddedMarketStudyb.pdf>. Acesso em: 01 maio 2017.

WITKOWSKI, T. et al. Model checking concurrent linux device drivers. In: **Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering**. New York: Association for Computing Machinery, 2007. p. 501–504. ISBN 9781595938824.

ÅHMAN, N. **Testing of Linux Kernel Modules**. Dissertation (Master) — Tampere University of Technology, jun. 2017. Disponível em: <https://trepo.tuni.fi/handle/123456789/24806>. Acesso em: 04 fev. 2021.

GLOSSÁRIO

Árvore – um repositório de código-fonte. Por exemplo, a expressão "árvore do *kernel* Linux" se refere ao repositório que guarda seu código-fonte.

Assertion – é a comparação de estados observáveis, durante a execução de um caso de teste, que se tiver resultado negativo interrompe-o imediatamente, marcando-o como falha.

Backport – atividade de desenvolvimento de software que busca levar funcionalidades que inicialmente foram desenvolvidas apenas para versões novas de um sistema para alguma versão mais antiga. É comum em sistemas embarcados fazer o *backport* apenas de correções de segurança em aplicações e também no *kernel* Linux.

Barramento – conexão física entre componentes de hardware que permite a troca de dados entre esses componentes.

`Bash`[1] – provê vários comandos comuns em distribuições Linux, é comumente usado em distribuições para computadores de uso geral.

`Bison`[2] – é um gerador genérico de *parsers* escritos em `C` ou `C++`, usado durante as primeiras fases da compilação de um *kernel* Linux.

Br2-external – é um *plug-in* para o `Buildroot` que pode conter, entre outras coisas, uma ou mais receitas de compilação (chamadas "pacotes") para software que a ferramenta ainda não conhece.

`Busybox`[3] – provê vários comandos comuns em distribuições Linux, é comumente usado em sistemas Linux embarcados, substituindo o `bash`, pois usa muito menos recursos de hardware que aquele, uma vez que implementa apenas as funcionalidades mais usadas.

`Ccache`[4] – implementa uma *cache* para a compilação de software escrito nas linguagens `C` e `C++`, agilizando a recompilação quando o código do software mudou pouco.

Compilação cruzada – compilação de um software em um computador de uso geral (*host*) gerando binários que executam apenas em outro computador ou equipamento (*target*), geralmente que usa outra arquitetura de processador. É comum na geração de binários para sistemas embarcados.

Container – forma de virtualização de partes de um sistema de software, isolando a execução desse software do computador em que ele está executando.

---

[1] `bash` disponível em <https://www.gnu.org/software/bash>, acesso em 28 de agosto de 2021

[2] `bison` disponível em <https://www.gnu.org/software/bison>, acesso em 28 de agosto de 2021

[3] `busybox` disponível em <https://www.busybox.net>, acesso em 28 de agosto de 2021

[4] `ccache` disponível em <https://ccache.dev>, acesso em 28 de agosto de 2021

`Docker`[5] – ferramenta que permite a criação e execução de *containers*.

`DockerHub`[6] – serviço *online* que hospeda *containers* `docker` e possui planos gratuitos de uso.

Expectation – é a comparação de estados observáveis, durante a execução de um caso de teste, que se tiver resultado negativo marca-o como falha sem interrompê-lo.

`Flex`[7] – é um gerador de ferramentas que reconhecem padrões lexicais em texto, usado durante as primeiras fases da compilação de um *kernel* Linux.

Fork – cópia de um repositório de código-fonte que diverge do repositório original. Com relação ao *kernel* Linux é comum fabricantes de processador manterem *forks* até que o suporte a esses processadores seja enviado e aceito no repositório principal do *kernel*. Um exemplo de *fork* do *kernel* Linux é o *kernel* usado em sistemas `Android`[8], comum em telefones celulares e *smart TV*.

`Gcc`[9] – compilador de várias linguagens, incluindo `C` e `C++`, comumente usado para compilar software para sistemas embarcados que usam Linux.

`Git`[10] – ferramenta de controle versão de código-fonte. O controle é distribuído, ou seja, cada cópia de um repositório pode operar de forma independente, como um *fork*. Suporta um repositório de código apontar para uma versão de outro repositório através da funcionalidade de submódulo.

`GitLab`[11] – serviço *online* que hospeda código-fonte que usa `git`, semelhante ao `GitHub`[12], mas disponibiliza também um ambiente de integração contínua com máquinas virtuais, chamadas "*Shared Runners*", para serem usadas gratuitamente pelos projetos que ele hospeda executarem compilações, testes e integração contínua.

Host – computador usado para desenvolvimento de software. Em contraste com *target*.

Kernel space – ambiente em que um software executa sem muitas restrições de permissão, podendo acessar diversos endereços de memória ou ainda executar em contexto de tratamento de interrupções de hardware. Em contraste com *user space*.

Kernel vivo – *kernel* em execução com todas as suas partes essenciais (escalonador de processos, gerenciador de memória, etc.) e que consegue interagir com o *user-space*.

---

[5] `docker` disponível em <https://www.docker.com>, acesso em 28 de agosto de 2021

[6] `DockerHub` disponível em <https://hub.docker.com>, acesso em 28 de agosto de 2021

[7] `flex` disponível em <https://github.com/westes/flex>, acesso em 28 de agosto de 2021

[8] *Common Android Kernel Tree* disponível em <https://android.googlesource.com/kernel/common>, acesso em 28 de agosto de 2021

[9] `gcc` disponível em <https://gcc.gnu.org>, acesso em 28 de agosto de 2021

[10] `git` disponível em <https://git-scm.com>, acesso em 28 de agosto de 2021

[11] `GitLab` disponível em <https://gitlab.com>, acesso em 28 de agosto de 2021

[12] `GitHub` disponível em <https://github.com>, acesso em 28 de agosto de 2021

`Libc` – termo genérico usado para referenciar a principal biblioteca em linguagem `C` de um sistema que usa Linux, que provê implementação de funções básicas como `open`, `malloc` e `printf`. Comumente se refere à `glibc`[13] em computadores de uso geral que usam Linux, e à `uClibc-ng`[14] em sistemas embarcados que usam Linux.

`Make`[15] – ferramenta que permite a automatização de processos comuns durante o desenvolvimento de um software, inclusive a compilação. É possível criar com ela receitas para várias atividades, por exemplo, se no arquivo `Makefile` de um software estiverem definidos os comandos a serem executados para a atividade `test` de um software, basta o desenvolvedor chamar `make test` para que essa sequência de comandos seja executada.

Mock – dublê de testes com funcionalidade limitada controlada pelo caso de teste. O *mock* de uma função poderia, por exemplo, contar quantas vezes a função foi chamada pelo código testado.

`Nose2`[16] – módulo `Python` que orquestra a execução de testes automáticos. Depende do `unittest` e adiciona relatórios de execução e paralelização dos testes.

Offboard – são os componentes de hardware que não estão na placa de circuito de um equipamento mas podem ser conectados a esse equipamento através de algum conector. Um exemplo de componente *offboard* para um computador de uso pessoal são os componentes de hardware que estão dentro de um teclado que está conectado ao computador através de um conector USB. O processador e o sistema operacional desse computador precisa se comunicar e tratar sinais desses componentes, mesmo eles não fazendo parte da placa de circuito impresso do computador, comumente conhecida pelo termo "placa mãe".

Onboard – são os componentes de hardware que são soldados à placa de circuito impresso no momento da fabricação de um equipamento.

Open source – de forma geral esse termo é usado para indicar que um software foi publicado com uma licença que permite que ele seja redistribuído. Existem infinitas licenças e formas de distribuição possíveis. No contexto deste trabalho o termo foi usado para designar apenas as ferramentas que disponibilizam o código-fonte com a licença explícita, com o arquivo ou repositório listado em ferramentas de busca usuais da *Internet* e que a URL de *download* esteja funcional e contenha de fato o código-fonte e não um binário

[13] `glibc` disponível em <https://www.gnu.org/software/libc>, acesso em 28 de agosto de 2021
[14] `uClibc-ng` disponível em <https://uclibc-ng.org>, acesso em 28 de agosto de 2021
[15] `make` disponível em <https://www.gnu.org/software/make>, acesso em 28 de agosto de 2021
[16] `nose2` disponível em <https://github.com/nose-devs/nose2>, acesso em 28 de agosto de 2021

executável, como definido na Seção 2.4.

Open-source – o mesmo que "*open source*" mas usado como adjetivo que precede um nome na língua inglesa.

Plug-in – extensão de um software que pode ser usada com o software original sem a necessidade de recompilar o software original.

Receita de compilação – lista de todos os comandos que precisam ser executados em uma ordem predeterminada para compilar um software específico.

`squashfs` – tipo de sistema de arquivos comumente usado em sistemas embarcados usando Linux.

Stub – dublê de testes sem funcionalidade real, que apenas responde minimamente para que o teste seja realizado.

Submódulo – funcionalidade da ferramente `git` que permite um repositório incluir outro por indireção, indicando a versão do código desse segundo repositório associada a cada versão de código do primeiro.

`SVN`[17] – ferramenta de controle versão de código-fonte. O controle é centralizado, várias operações realizadas pelo desenvolvedor precisam ser aprovadas pelo computador que contém o repositório.

Target – hardware no qual o software desenvolvido será embarcado. Em contraste com *host*.

Toolchain – conjunto de ferramentas necessário para a compilação cruzada de software.

`Unittest`[18] – módulo `Python` que orquestra a execução de testes automáticos.

User space – ambiente em que um software executa com muitas restrições de permissão, de modo geral cada processo tem acesso apenas à sua própria região de memória e as interações com dispositivos de hardware são feitas através de chamadas de sistema para o *kernel* do sistema operacional. Em contraste com *kernel space*.

[17] `SVN` disponível em <https://subversion.apache.org>, acesso em 28 de agosto de 2021

[18] `unittest` disponível em <https://docs.python.org/3/library/unittest.html>, acesso em 28 de agosto de 2021

APÊNDICE A — DETALHAMENTO DAS ETAPAS DA IMPLEMENTAÇÃO DO EXPERIMENTO

Cada etapa do fluxograma da Figura 5.2 é detalhada abaixo:

1. Implementar um TB simples que execute localmente – Foi criado um TB que compila um *kernel*, o inicia em `qemu` e executa comandos em *user space* nessa imagem executando em `qemu`. Para possibilitar a execução automática desse TB foi criado o que o `Buildroot` chama de "br2-external", criado um `Makefile` que possibilitasse ao desenvolvedor iniciar os testes com apenas o comando `make test`, adicionado o `Buildroot` como um submódulo `git`, e adicionado um TB mínimo.
2. Permitir o TB executar no ambiente de integração contínua – Foi criado o repositório <https://gitlab.com/RicardoMartincoski/utootlkm> (iniciais de "*Unit Testing Out-Of-Tree Linux Kernel Modules*"), incluindo um arquivo `.gitlab-ci.yml` que configura o repositório para que, a cada modificação enviada, seja executado o equivalente a `make test`, reusando a mesma imagem `docker` que o <https://gitlab.com/buildroot.org/buildroot> já usa (é necessário especificar uma imagem para habilitar a integração contínua no `GitLab`).
3. Executar um TK *in-tree* dentro do TB – Foi utilizado um TK que já está integrado no *kernel*, com módulo *built-in*. Para isso foi modificado o TB criado anteriormente para que habilitasse na compilação do *kernel* os *drivers in-tree* necessários e para que durante a execução em `qemu` coletasse os resultados brutos e então chamasse o *parser* do `kunit_tool`. Foi necessário criar uma imagem `docker` (Figura B.8) pois o `kunit_tool` necessita de `Python` em versão maior ou igual a `3.6`. Também foi alterado o comportamento original do TB para que, além do seu próprio resultado, também mostrasse o resultado dos TKs que ele executou.
4. Executar um TK *out-of-tree* dentro do TB – Foi usado o exemplo do `KUnit` que já está integrado no *kernel*, mas transformando-o em *driver out-of-tree*. Para isso foi copiado o arquivo `kunit-example-test.c` do repositório do *kernel*, criado um `Makefile` e uma receita de compilação (chamada no `Buildroot` de "pacote"), e ajustado o TB criado anteriormente para que passasse a compilar essa versão *out-of-tree* do *driver* ao invés da *in-tree*, e para que durante a execução em `qemu` inserisse o *driver* no *kernel*.

5. Fazer o TB compilar o *driver out-of-tree* escolhido – Foi criada a receita de compilação do *driver* e adaptado o TB para que o compilasse.
6. Criar casos de teste de unidade (TKs) simples para validar a infraestrutura – Foram criadas duas *testsuites* do `KUnit` com dois TKs cada uma. Cada TK apenas compara um valor constante do *driver* contra um valor fixo, dois deles foram feitos com um valor fixo propositalmente errado para exercitar ambos os resultados "*PASSED*" e "*FAILED*".
7. Criar casos de teste de unidade (TKs) para o *driver out-of-tree* escolhido – Foram escolhidas duas funções do *driver* original para serem testadas: a `rtl_get_version` que faz o mapeamento entre um valor lido do dispositivo para o valor de uma variável interna que ajusta o comportamento do *driver* ao modelo do produto, e a `rtl8152_probe` que detecta qual o modelo do produto (chamando a `rtl_get_version`), configura vários parâmetros do *driver* e preenche a estrutura que serve como um objeto a ser passado para o *kernel*. Para isso foram necessárias pequenas modificações no *driver* original para permitir a criação de um *mock* (versão com funcionalidade limitada controlada pelo TK) de uma função de acesso ao hardware chamada `usb_control_msg` (interface B na Figura 3.1) e a criação de dois *stubs* (versão sem funcionalidade real) das funções `register_netdev` e `sysfs_create_group` que expõem interfaces para o *user space* (interface A na Figura 3.1). Para testar as funções escolhidas não foi necessário criar *mock* ou *stub* de funções da interface C na Figura 3.1, pois foi possível obter um comportamento determinístico dos outros sistemas do *kernel* apenas criando objetos *stub* na inicialização do TK. Essas modificações no *driver* original foram guardadas em um arquivo do tipo `.patch` para que o `Buildroot` aplicasse antes de compilar o *driver*.
8. Conseguir executar os testes de unidade (TKs) a partir do repositório do *driver* – Foi habilitado um novo ambiente de integração contínua, agora no repositório que contém o código do *driver*. Para isso foi modificado o repositório criado anteriormente para que pudesse operar como um submódulo `git` de outro, foi criado o repositório <https://gitlab.com/RicardoMartincoski/realtek-r8152-linux> como uma cópia do repositório original do *driver*, adicionando o primeiro repositório criado como submódulo deste, copiado o `.gitlab-ci.yml` criado anteriormente, alterado o `Makefile` que o *driver* original já continha para possibilitar executar os testes com apenas o comando `make KERNELRELEASE=0 test`, aplicada a mo-

dificação criada anteriormente para permitir o *mock*/*stub* de três funções do *driver* original e copiados os TKs criados anteriormente.

Intercaladas aos passos acima, foram feitas também algumas melhorias de performance na infraestrutura criada, sumarizadas na Seção 6.3.

Após as etapas acima, também foram criados alguns exemplos de pequenas modificações no *driver* para demonstrar como seria o funcionamento de um ambiente de integração contínua que utilizasse a infraestrutura criada para executar testes de unidade em *driver out-of-tree*. Isso é detalhado no APÊNDICE B.

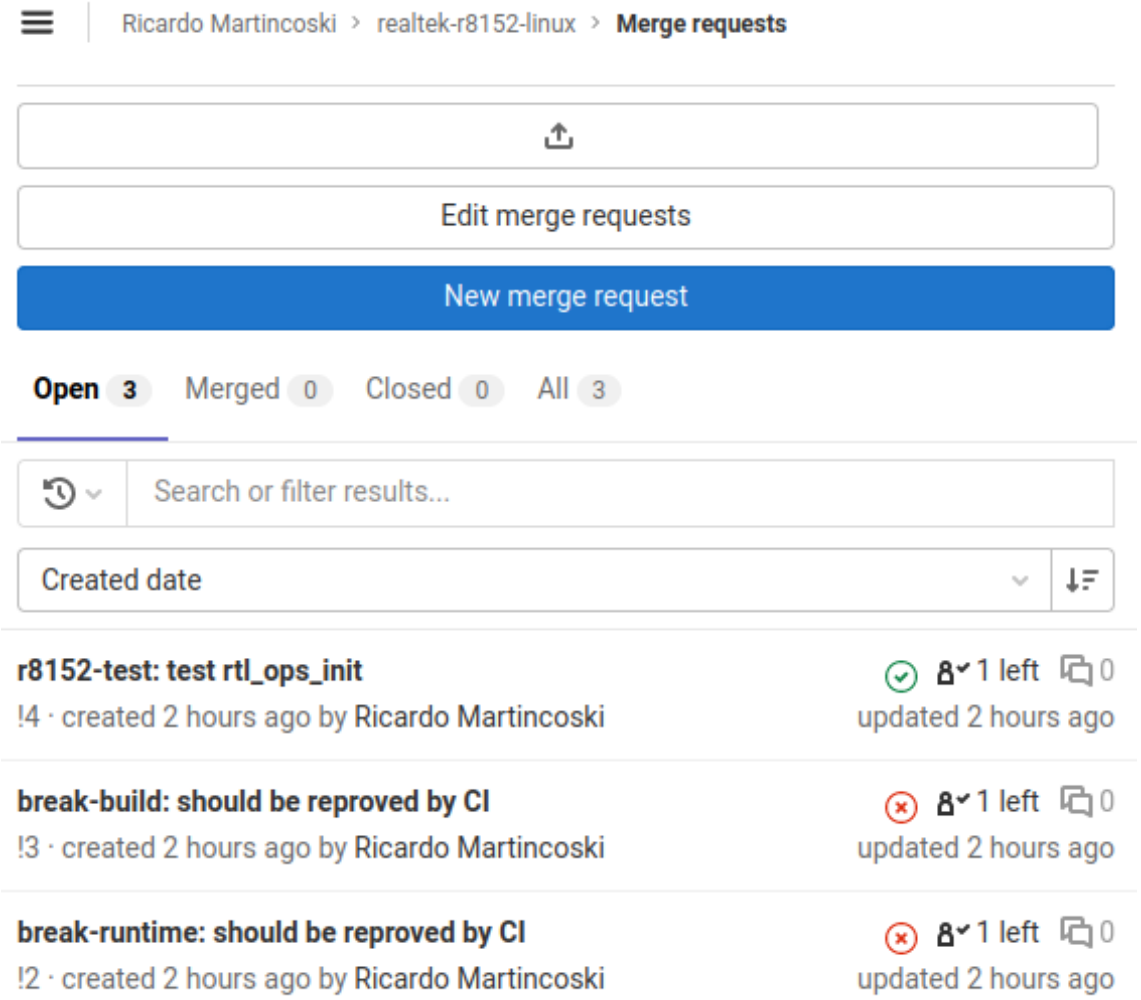
A dinâmica de execução de um TB mostrada na Figura 4.3 e na Figura 4.4 é uma versão simplificada. Além do que é mostrado nas figuras, também foram adicionados alguns passos à implementação para facilitar a depuração em caso de falha:

- Não excluir o que foi compilado pelo TB (excluir é o comportamento padrão do *framework* do `Buildroot`) para agilizar a recompilação e consequentemente o reteste;
- Forçar a recompilação do *driver* com TKs para garantir que sempre que o desenvolvedor chamar `make test`, independente de o teste já ter executado antes, será testado o código do *driver* e dos TKs que está naquele momento no repositório do *driver*;
- Chamar o comando `dmesg`, que lista todas as mensagens do *kernel* e seus *drivers*, na imagem em execução no `qemu` e guardar o conteúdo em um arquivo;
- Guardar o conteúdo do resultado bruto dos TKs em um arquivo.

APÊNDICE B — MUDANÇAS DE CÓDIGO USADAS PARA EXERCITAR O AMBIENTE DE INTEGRAÇÃO CONTÍNUA

Para ilustrar o uso de um ambiente de integração contínua foram criadas três modificações pontuais no código, no repositório <https://gitlab.com/RicardoMartincoski/realtek-r8152-linux>, cada uma a partir da versão inicial dos testes previamente adicionados, ilustradas na Figura B.1.

Figura B.1: Lista de mudanças de código criadas para exercitar o ambiente de integração contínua



Fonte: O Autor

A primeira mudança foi criada para que o *driver* em teste se comportasse de maneira errônea no cenário de algum dos TKs que foram criados. A relação dessa mudança de código no *driver* com o código do TK e as mensagens do TK expostas pelo TB são analisadas na Seção 5.4. Essa mudança está ilustrada na Figura 5.4 e o resultado do TB falhando devido à falha no TK na Figura B.2 e na Figura B.3.

Figura B.2: TB falhando devido à falha do TK, parte 1

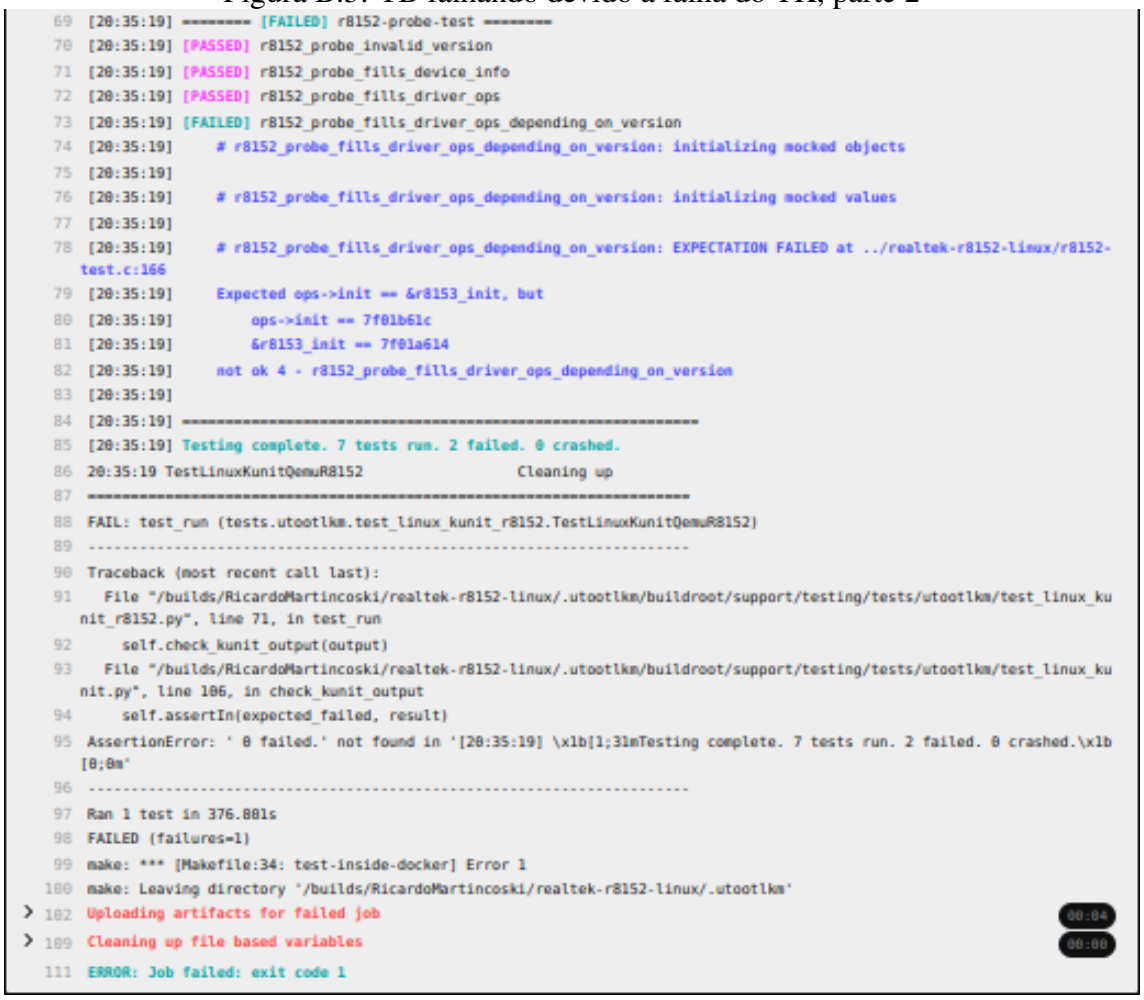
Fonte: O Autor

A segunda mudança foi criada para que o *driver* em teste não compilasse. Essa mudança está ilustrada na Figura B.4 e o resultado do TB falhando na Figura B.5.

A terceira mudança foi criada para adicionar TKs. Essa mudança está ilustrada parcialmente na Figura B.6 e o resultado do teste passando na Figura B.7.

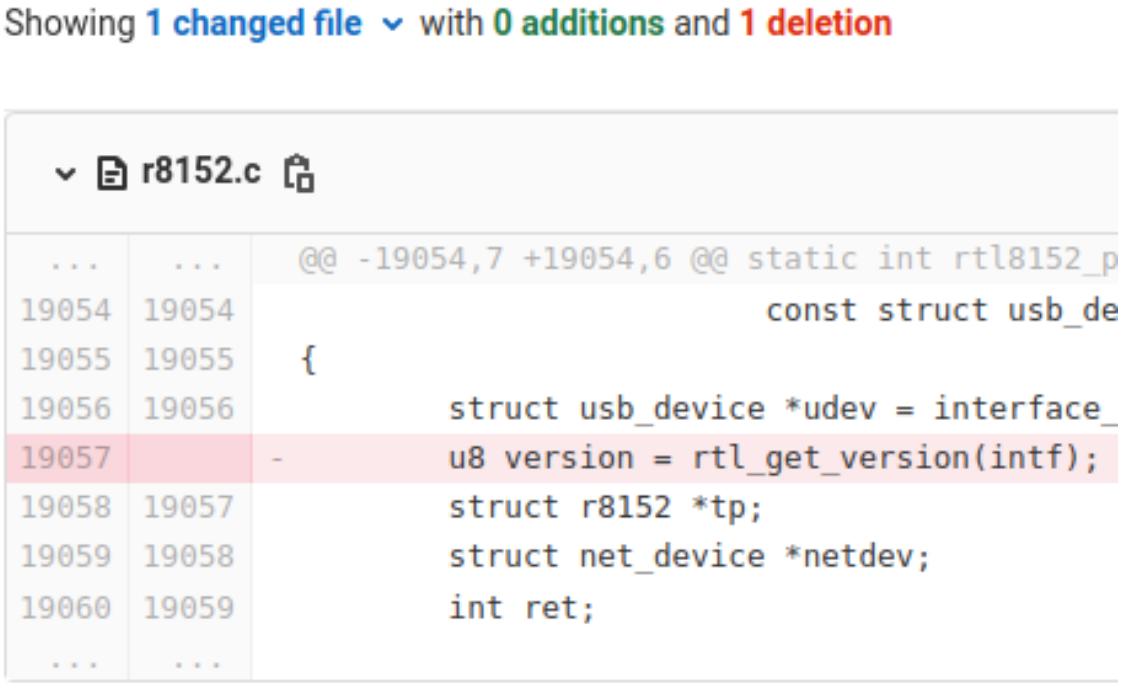
Todas as modificações usaram a imagem `docker` disponível em <https://hub.docker.com/r/ricardomartincoski/utootlkm/tags> mostrada na Figura B.8.

Figura B.3: TB falhando devido à falha do TK, parte 2



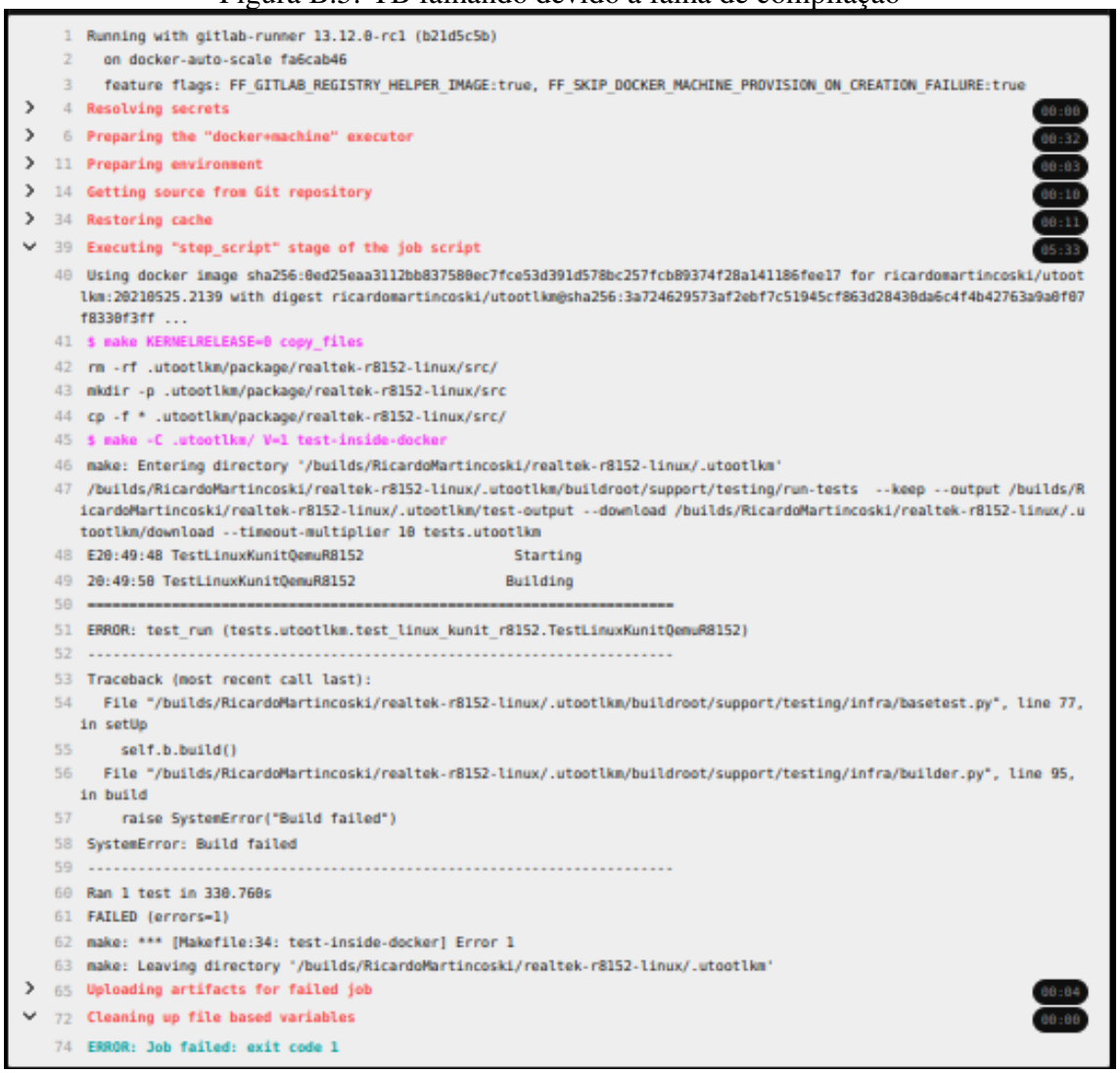
Fonte: O Autor

Figura B.4: Exemplo de mudança no *driver* que faz a compilação falhar



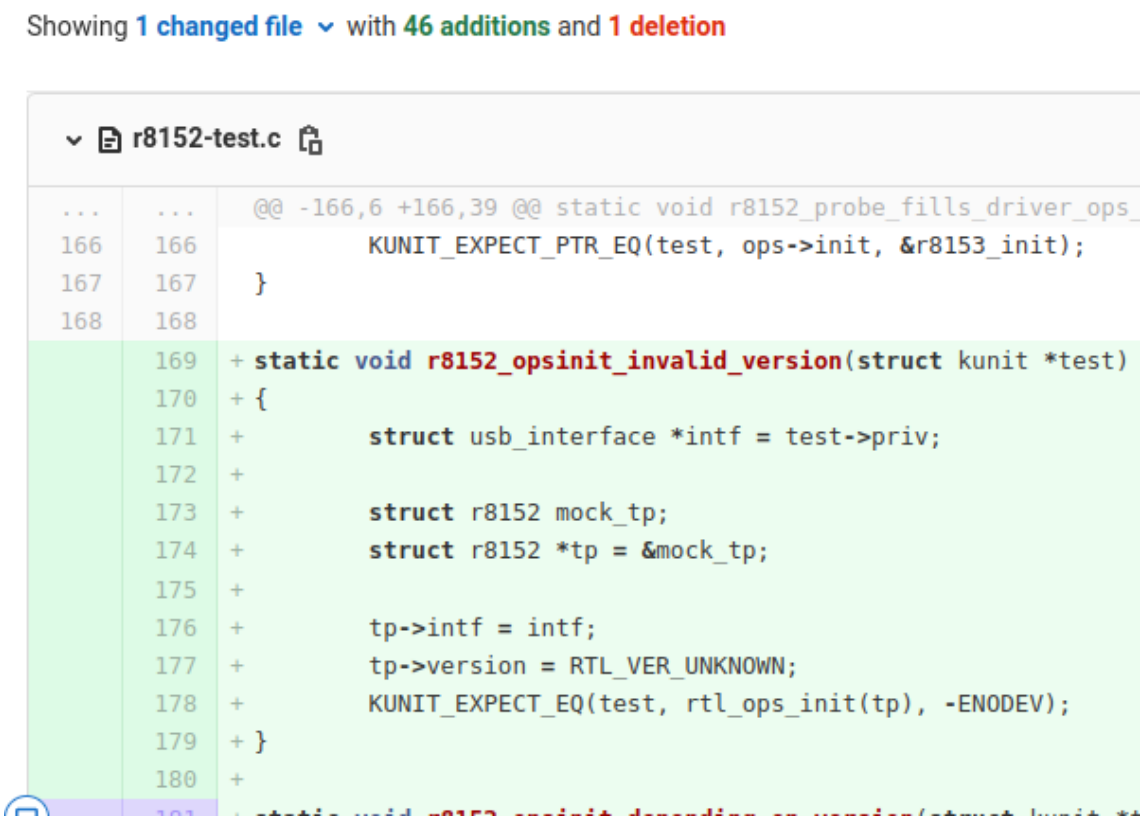
Fonte: O Autor

Figura B.5: TB falhando devido à falha de compilação



Fonte: O Autor

Figura B.6: Exemplo mostrando parte da adição de TKs que passam

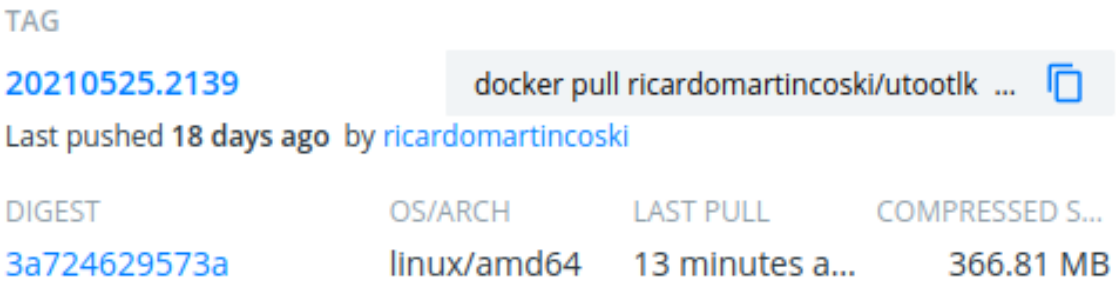


Fonte: O Autor

Figura B.7: TB passando após serem adicionados TKs

Fonte: O Autor

Figura B.8: Imagem docker utilizada pela infraestrutura



Fonte: O Autor

APÊNDICE C — COMO REPRODUZIR OS RESULTADOS DO EXPERIMENTO

Todos os arquivos necessários para reproduzir o experimento desde trabalho foram publicados com licença *open source*.

Em um computador com o sistema operacional `Ubuntu 20.04.2.0` e acesso à *Internet* os resultados dos testes podem ser reproduzidos com os passos descritos abaixo. O procedimento completo deve demorar entre uma a duas horas, dependendo do computador e do acesso à *Internet*, e usa aproximadamente 6 GB de disco.

1) Instale o `docker`, adicione o usuário ao grupo e verifique que a permissão foi concedida:

```
$ sudo apt install docker.io
$ sudo groupadd docker
$ sudo usermod -aG docker $USER # NOTE: logout/login after this
$ docker run hello-world
```

2) Instale `git` e `make`:

```
$ sudo apt install git make
```

3) Faça cópia dos repositórios e execute os testes:

```
$ git clone --depth 1 --recurse-submodules -b v1.0 \
  https://gitlab.com/RicardoMartincoski/utootlkm.git
$ git clone --depth 1 --recurse-submodules -b driver_repo_ci \
  https://gitlab.com/RicardoMartincoski/realtek-r8152-linux.git
$ make -C utootlkm/ test
$ make -C realtek-r8152-linux/ KERNELRELEASE=0 test
```