UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Guilherme Soares Correa

Registro de Eventos de Falhas Silenciosas no Linux: Contribuições para o Subsistema LRAC

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Guilherme Soares Correa

Registro de Eventos de Falhas Silenciosas no Linux: Contribuições para o Subsistema LRAC

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Rivalino Matias Júnior

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2025

Guilherme Soares Correa

Registro de Eventos de Falhas Silenciosas no Linux: Contribuições para o Subsistema LRAC

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Prof. Dr. Rivalino Matias Júnior
Orientador

Prof. Dr. Celso Maciel da Costa

Prof. Dr. Marcelo Barros de Almeida

Uberlândia, Brasil
2025

Lista de ilustrações

Lista de tabelas

Lista de Códigos

Lista de abreviaturas e siglas

LRAC	*Linux Reliability Analysis Component*
RAC	*Reliability Analysis Component*
RM	*Reliability Monitor*
PC	Pontos de Coleta
MCDF	Módulo de Coleta de Dados de Falha
lracd	Serviço de Armazenamento de Registros de Falha
SO	Sistema Operacional
MTTR	*Mean Time To Repair* (tempo médio para reparo)
MTTD	*Mean Time To Detect* (tempo médio para detecção)
RFC 3164	RFC 3164 – Protocolo *Syslog* da Berkeley Software Distribution (BSD)
PRI	*Priority value field* de uma mensagem *Syslog*
HEADER	*Header field* de uma mensagem *Syslog*
MSG	*Message content field* de uma mensagem *Syslog*
CPU	*Central Processing Unit*
KB	*Kilobyte* (10^3 bytes, prefixo SI)
NTP	*Network Time Protocol*
IA	Inteligência Artificial (*Artificial Intelligence*)
ASCII	*American Standard Code for Information Interchange*
E/S	Entrada/Saída
IPC	*Inter-Process Communication*
PID	*Process Identifier*
FTP	*File Transfer Protocol*

Sumário

1 Introdução

Atualmente, a confiabilidade de sistemas computacionais é uma preocupação cada vez maior. Esses sistemas controlam a maioria das atividades cotidianas, já que estão profundamente integrados à sociedade, sendo utilizados em diversos contextos, inclusive em setores sensíveis como saúde, transporte e finanças. Em especial, eles exercem papel central em atividades que afetam diretamente o bem-estar humano, mas enfrentam desafios crescentes, como falhas de software, erros do usuário e defeitos de hardware. Assim, a capacidade de lidar com falhas é um requisito crucial para os sistemas computacionais modernos. Diante disso, garantir seu funcionamento correto é fundamental e, por isso, diversas pesquisas têm sido realizadas pela importância da confiabilidade aplicada a sistemas computacionais (DAVID; CARLYLE; CAMPBELL, 2007).

1.1 Visão geral

Este trabalho tem como objetivo contribuir para o aprimoramento do LRAC (*Linux Reliability Analysis Component*), um subsistema do *kernel* Linux desenvolvido para registrar (*logging*) dados de falhas de software em sistemas Linux (MACIEL; MATIAS, 2015). A coleta de dados de eventos de falha de software é a base para estudos de confiabilidade de software. Sem ela, se tornam impraticáveis esses estudos. Apesar dessa importância, muitos sistemas operacionais não apresentam uma infraestrutura sofisticada para a coleta de dados de eventos de falha. Um exemplo desses sistemas é o Linux, que, apesar de realizar o *log* de eventos, não captura diversas informações essenciais para estudos detalhados de confiabilidade. Para suprir essa necessidade, foi criado o LRAC. Este trabalho incorpora ao arcabouço do LRAC a capacidade para o registro de falhas silenciosas (MACIEL; MATIAS, 2015).

No contexto deste trabalho, usamos o termo **falhas silenciosas** como equivalente a falhas *non-crash-based*, isto é, falhas em que a própria aplicação encerra sua execução de forma controlada após um erro interno. Este trabalho concentra-se na captura e registro desses eventos, fornecendo subsídios para que outras ferramentas possam, posteriormente, realizar a identificação e análise de padrões associados a falhas silenciosas. A proposta deste estudo é fornecer subsídios para a implementação de um mecanismo no LRAC capaz de registrar tais eventos, ampliando a capacidade de detecção e análise de falhas que comprometem o funcionamento do sistema.

Para isso, foram realizados experimentos com diversas aplicações, nos quais se injetaram faltas com o objetivo de observar o comportamento das aplicações diante de falhas silenciosas. A partir dessas observações, buscou-se compreender como essas falhas

se manifestam e como podem ser detectadas, de modo a subsidiar o desenvolvimento de um mecanismo no LRAC voltado ao registro automático desses eventos.

1.2 Objetivo geral

Analisar falhas silenciosas em aplicações no ambiente Linux, com o objetivo de caracterizar seus comportamentos e, assim, fornecer subsídios para a implementação de um mecanismo no LRAC capaz de identificá-las e registrá-las no momento em que ocorrem.

1.3 Objetivos específicos

- Observar o comportamento de aplicações no ambiente Linux diante de falhas silenciosas induzidas, identificando eventos de encerramento controlado causados por essas falhas.
- Localizar e analisar os pontos do *kernel* do sistema operacional Linux onde é possível coletar e registrar dados relevantes sobre essas falhas.
- Avaliar a latência entre a ocorrência da falha e o encerramento da aplicação, bem como a ocorrência de falsos positivos, quando o encerramento é feito manualmente pelo usuário.

1.4 Justificativa

A confiabilidade dos sistemas computacionais é cada vez mais crítica, especialmente em ambientes onde a disponibilidade ininterrupta é exigida. Embora o Linux possua mecanismos para capturar *crashes* (casos em que o SO força o encerramento do processo), há uma lacuna significativa no registro de falhas silenciosas, situações em que a própria aplicação detecta um erro interno e encerra-se de modo controlado, sem gerar *core dumps* nem alarmes evidentes. Investigar e registrar esses eventos preenche um espaço pouco explorado na literatura de confiabilidade de software, contribuindo para reduzir o *Mean Time To Repair* (MTTR) e aprimorar a robustez de serviços que dependem de alta disponibilidade (MACIEL; MATIAS, 2015; PHAM, 2000).

Do ponto de vista prático, a ausência de dados sobre falhas silenciosas dificulta diagnósticos deste tipo de falha, inclusive, deixando muitas vezes de contabilizá-las nos cálculos de confiabilidade dos sistemas de software. Ao estender o LRAC para capturar esses eventos, este trabalho beneficia diretamente a indústria ao fornecer subsídios para detecção precoce através dos dados registrados. Além disso, o trabalho estimula novos

estudos em análise de falhas e, por conseguinte, contribui para fortalecer o ecossistema Linux.

2 Revisão da Literatura

2.1 Fundamentação Teórica

A confiabilidade de software é um tema de grande relevância e objeto de debates constantes no campo da computação. Neste capítulo, são apresentados os principais conceitos relacionados à confiabilidade, incluindo definições e classificações de falhas. Também são descritos trabalhos relacionados que tratam de técnicas, ferramentas e estudos voltados à análise de falhas em sistemas computacionais, com o objetivo de embasar teoricamente o desenvolvimento da pesquisa. Por fim, são apresentadas ferramentas utilizadas ao longo do trabalho e conceitos essenciais para o entendimento da proposta.

2.1.1 Confiabilidade de Software

A confiabilidade do software é um atributo crucial que representa a capacidade do sistema de executar suas funções conforme especificado, por um período determinado e sob condições definidas, sem apresentar falhas. Isso posto, o conceito é formalmente definido como a probabilidade de um programa de computador desempenhar suas funções por um determinado período e sob condições operacionais específicas, sem que ocorram falhas na prestação dos seus serviços (IEEE, 1990).

2.1.1.1 Falta (Defeito/*Bug*)

O conceito de *falta* (*fault*) é fundamental para a compreensão das ameaças à confiabilidade em sistemas computacionais. Segundo Avizienis et al. (AVIZIENIS et al., 2004), uma falta é a causa adjudicada ou hipotetizada de um erro, sendo uma condição ou evento que, se encontrado ou ativado, pode levar o sistema a falhar no cumprimento de suas funções requisitadas.

No domínio da engenharia de software, uma falta é frequentemente associada a um defeito presente em um artefato de software, como código-fonte, especificação de requisitos ou projeto de software. Rohr et al. (HAMILL; GOSEVA-POPSTOJANOVA, 2015) refinam a definição ao caracterizar uma falta como o conjunto mínimo de desvios do código correto, tal que a execução do código desviante pode desencadear um erro. Essa ênfase na minimalidade implica que todas as partes do desvio são necessárias para constituir a causa potencial do erro.

Na prática do desenvolvimento de software, os termos "*bug*" e "defeito" (*defect*) são comumente usados como sinônimos de falta, especialmente ao se referirem a incorreções no código-fonte (AVIZIENIS et al., 2004). Esses defeitos podem ser introduzidos em qualquer

fase do ciclo de vida do software, abrangendo desde requisitos mal especificados até erros de implementação.

2.1.1.2 Erro

O conceito de erro é central para a compreensão da confiabilidade em sistemas computacionais. Segundo Avizienis et al. (AVIZIENIS et al., 2004), um erro é definido como o estado interno de um sistema que resulta do acionamento de uma falta (*fault*). Esse estado representa um desvio do comportamento esperado, que, se não for corrigido, poderá levar à manifestação de uma falha (*failure*).

Em sistemas de software, o erro pode se materializar de diversas formas, como variáveis com valores incorretos, estados inconsistentes de dados ou operações que não respeitam as especificações previstas. Rohr et al. (HAMILL; GOSEVA-POPSTOJANOVA, 2015) apontam que o erro é uma consequência direta da execução de código afetado por faltas previamente inseridas no sistema.

2.1.1.3 Falha

O conceito de *falha* (*failure*) é fundamental na análise da confiabilidade de sistemas computacionais. De acordo com Avizienis et al. (AVIZIENIS et al., 2004), uma falha ocorre quando o serviço fornecido por um sistema se desvia daquele especificado, representando uma manifestação externa de um erro interno. Em outras palavras, é o momento em que o sistema não cumpre sua função conforme o esperado pelo usuário ou conforme definido nas especificações.

Falhas podem ser classificadas de diversas maneiras. Avizienis et al. (AVIZIENIS et al., 2004) propõem uma taxonomia baseada em atributos como:

- **Detectabilidade**: falhas podem ser detectadas ou não detectadas.
- **Consequência**: falhas podem ser benignas ou catastróficas.
- **Persistência**: falhas podem ser permanentes, transitórias ou intermitentes.

No contexto da engenharia de software, Chillarege (CHILLAREGE, 1996) destaca que falhas são frequentemente resultado da propagação de erros não tratados, muitas vezes originados de faltas latentes no sistema. A identificação e correção dessas falhas são essenciais para melhorar a confiabilidade do software. Portanto, a análise e mitigação de falhas são componentes essenciais na engenharia de sistemas confiáveis e seguros.

2.1.2 Classificação de Falhas

Falhas em sistemas computacionais podem ser classificadas quanto ao seu comportamento após a manifestação. De acordo com Avizienis et al. (AVIZIENIS et al., 2004), essa categorização é essencial para entender os impactos causados no sistema e para definir estratégias apropriadas de detecção, recuperação e tolerância a falhas.

Dois grandes grupos de comportamento de falhas são amplamente reconhecidos na literatura: **falhas por encerramento abrupto** (*crash-based*) e **falhas silenciosas** (*non-crash-based*). Conforme discutido por Schroeder e Gibson (SCHROEDER; GIBSON, 2010), falhas *crash-based* são aquelas em que o sistema ou componente afetado tem sua execução interrompida, de forma abrupta, tornando o erro evidente para o usuário. As falhas silenciosas podem ser divididas em duas categorias distintas; a primeira refere-se a situações em que a aplicação encerra sua execução de forma espontânea, sem sinais evidentes de falha, ou seja, o encerramento não é abrupto e não é identificado pelo SO como consequência de uma falha. A segunda categoria corresponde a casos em que a aplicação continua em execução após a falha, sem alertas ou interrupções perceptíveis, e passa a produzir resultados incorretos. Este trabalho aborda o primeiro tipo de falha silenciosa, cujo encerramento não é detectado como falha pelo sistema.

Santos et al. (SANTOS; MATIAS; TRIVEDI, 2019) enfatizam que falhas silenciosas representam desafios significativos, pois podem comprometer a integridade dos dados ou a disponibilidade dos serviços antes de serem detectadas. Essa dificuldade em identificar falhas silenciosas em tempo hábil é uma das razões pelas quais a pesquisa na área de confiabilidade de sistemas tem dedicado atenção crescente a esse tipo de falha.

A correta classificação das falhas quanto ao comportamento, portanto, é um passo fundamental para o desenvolvimento de técnicas de detecção automática, mitigação e recuperação de falhas em sistemas modernos, especialmente em ambientes críticos como serviços em nuvem e data centers (AVIZIENIS et al., 2004; SCHROEDER; GIBSON, 2010; SANTOS; MATIAS; TRIVEDI, 2019).

2.1.2.1 Encerramento Abrupto (*Crash-based*)

Falhas por encerramento abrupto (*crash-based failures*) ocorrem quando a execução de um software é interrompida pelo *kernel* do SO imediatamente após a ocorrência de um erro. Nessa situação, o processo não se encerra voluntariamente; ele é forçado a parar. Por ser um evento brusco e imposto pelo SO, a falha costuma ser detectável e acompanhada de evidências explícitas, como mensagens de erro, registros em *log* ou alertas enviados aos usuários.

Segundo Schroeder e Gibson (SCHROEDER; GIBSON, 2010), falhas *crash-based* são comuns, embora impactem a disponibilidade, tendem a ser mais facilmente detectadas

e diagnosticadas em comparação com falhas silenciosas. A natureza abrupta dessas falhas facilita a ativação de mecanismos automáticos de recuperação, como reinicialização de serviços ou *failover* para sistemas redundantes.

Em sistemas de grande escala, como data centers e ambientes de computação em nuvem, o tratamento de falhas *crash-based* é frequentemente realizado por estratégias baseadas em reinicialização rápida (*crash-only software*), conforme discutido por Candea e Fox (CANDEA; FOX, 2003). Essa abordagem visa simplificar a recuperação, assumindo que falhas são inevitáveis e que o melhor caminho é detectar rapidamente a falha e restabelecer o serviço de forma automatizada.

Recentemente, avanços em aprendizado de máquina têm sido aplicados para prever e mitigar falhas *crash-based*. Por exemplo, o modelo CrashEventLLM utiliza grandes modelos de linguagem para prever eventos de falha com base em *logs* do sistema, permitindo ações proativas para evitar interrupções (MUDGAL; ARBAB; KUMAR, 2024).

Embora sejam relativamente mais fáceis de lidar do que outros tipos de falha, falhas *crash-based* ainda representam riscos relevantes para a continuidade dos serviços, especialmente em sistemas críticos que exigem alta disponibilidade (AVIZIENIS et al., 2004; MACIEL, 2024) .

2.1.2.2 Silenciosas (*Non-crash-based*)

Retomando a definição previamente apresentada, falhas silenciosas (*non-crash-based failures*) podem ser divididas em dois subtipos: **(i)** aquelas em que a aplicação é encerrada de forma espontânea e não abrupta, sem que o sistema operacional registre o evento como uma falha; e **(ii)** aquelas em que a aplicação continua em execução, porém gerando resultados incorretos de forma não evidente. Este trabalho concentra-se exclusivamente no primeiro subtipo, encerramento não detectado como falha.

Os principais desafios desse tipo de falha envolvem:

- **observabilidade limitada** – sinais de falha podem ficar restritos a métricas de baixo nível ou a *logs* verbosos;
- **propagação de estado incorreto** – o defeito difunde-se por módulos independentes (ex.: microsserviços) antes da detecção;
- **alta latência de descoberta** – quanto maior o atraso, maior o esforço de recuperação e a perda de confiança (SANTOS; MATIAS; TRIVEDI, 2019).

2.1.2.3 Comparativo: Falhas *Crash-based* vs. *Non-Crash-Based*

A compreensão das diferenças entre falhas por *crash-based* e *non-crash-based* é essencial para o desenvolvimento de sistemas resilientes e confiáveis. Cada tipo de fa-

lha possui características distintas que impactam diretamente as estratégias de detecção, mitigação e recuperação.

Tabela 1 – Comparativo entre falhas *crash-based* e sub-tipos de falhas *non-crash-based*

Aspecto	**Falhas *crash-based***	**Falhas *non-crash-based***	
		Encerramento não detectado	**Continuidade com erro**
Detecção	Imediata; processo para ou gera *logs* explícitos (SCHROEDER; GIBSON, 2010).	Tardia; aplicação termina "silenciosamente", sem registro de falha pelo SO.	Muito tardia; aplicação segue ativa produzindo saídas incorretas sem alertas (SANTOS; MATIAS; TRIVEDI, 2019).
Impacto	Interrupção de serviço; pode ser mitigada com reinicialização ou *failover* (CANDEA; FOX, 2003).	Serviço indisponível até reinicialização manual; perda de contexto de execução.	Compromete integridade de dados e confiança; corrupção pode propagar-se (AVIZIENIS et al., 2004).
Exemplos	Queda de servidor por falha de hardware detectada.	Aplicativo fecha sozinho após exceção não capturada; SO registra saída normal.	Cálculo científico continua, mas gera resultados imprecisos devido a *overflow* silencioso.
Recuperação	Reinicializações automáticas; redundância; *checkpoint/rollback*.	Reinicialização do serviço e restauração de estado persistente.	Diagnóstico manual, verificação de integridade e possível reprocessamento de dados.

2.1.3 Registro de falhas

O registro de falhas consiste no processo de captura, armazenamento e disponibilização de informações sobre eventos anômalos ou falhas ocorridas durante a operação de um SO. No Windows, destaca-se o Reliability Analysis Component (RAC); já no Linux, cabe ressaltar o *Syslog*, responsável pelo registro geral de eventos, e o *Auditd*, voltado à auditoria de chamadas de sistema e acessos sensíveis.

2.1.3.1 RAC (Windows)

O RAC é um componente integrado ao SO Windows cuja função primordial é registrar eventos de falhas (MICROSOFT, 2020). O RAC é responsável por coletar uma ampla gama de dados, incluindo notificações de instalação e atualização, componentes de

software internos do sistema, falhas de aplicativos, falhas de *kernel* e até mesmo notificações relacionadas a componentes de hardware (MACIEL, 2015).

Através desse mecanismo de coleta de dados, o RAC fornece uma visão abrangente e detalhada da confiabilidade do sistema. Ele captura informações sobre eventos críticos, como instalações e atualizações de software, que podem afetar o desempenho e a estabilidade do SO. Além disso, o RAC monitora ativamente as falhas de aplicativos e do *kernel*, identificando possíveis problemas e contribuindo para uma melhor compreensão das áreas que requerem atenção e otimização (MACIEL, 2015).

Ao compilar esses dados e alimentá-los no *Reliability Monitor* (MICROSOFT, 2008), o sistema é capaz de calcular o índice de estabilidade, uma métrica essencial para avaliar a confiabilidade global do SO . Com base nesse índice, os usuários podem tomar decisões informadas sobre a manutenção e o aprimoramento contínuo do sistema, visando melhorar sua estabilidade e evitar falhas indesejadas (MACIEL, 2015).

Em suma, o RAC desempenha um papel vital na coleta de dados de confiabilidade e fornece informações cruciais para a avaliação e monitoramento do SO Windows. Esses dados, originados de várias fontes, são essenciais para garantir a estabilidade e a confiabilidade do sistema, permitindo que os usuários tomem medidas proativas para melhorar a experiência de uso e evitar interrupções indesejadas.

2.1.3.2 *Syslog* & *Auditd* (Linux)

O *Syslog* é um protocolo padronizado que especifica tanto o formato das mensagens quanto o mecanismo de transporte para que aplicações e dispositivos encaminhem eventos de *log* a coletores centralizados, de forma independente de plataforma ou fabricante (GERHARDS, 2009). Sua popularidade advém da simplicidade e flexibilidade com que permite registrar diferentes tipos de eventos, como mensagens informativas, alertas e erros (ORLOVSKYI, 2024). No contexto deste trabalho, o interesse recai especificamente sobre as mensagens de *log* relacionadas a eventos de falha, pois elas constituem uma importante fonte de dados para análise de confiabilidade em sistemas Linux.

No Linux, o *Syslog* é implementado através de um processo *daemon* (ex. *syslogd*). O *daemon* atua como um agente de processamento de *logs*: ele captura mensagens do sistema em tempo real e, de acordo com sua configuração, pode gravá-las em arquivos de *log* locais, encaminhá-las a um servidor central de *logs* (por exemplo, via protocolo *syslog*) ou apresentá-las na saída padrão para visualização imediata.

Uma das implementações originais mais antigas de um *daemon syslog* para Linux era simplesmente referida como *syslog* ou sysklogd. Mais tarde, surgiram implementações mais modernas e comumente usadas, como *rsyslog* ou *syslog-ng*. Estas também foram feitas especificamente para Linux. *Rsyslog* é um *daemon syslog* leve e de alto desempenho, com

uma ampla gama de recursos. Ele normalmente vem pré-instalado em muitas distribuições Linux (tanto baseadas em Debian quanto em RedHat) (ORLOVSKYI, 2024).

Rsyslog, como muitos outros *daemons syslog*, escuta um soquete Unix `/dev/log` por padrão. Ele encaminha os dados recebidos para um arquivo `/var/log/syslog` em distribuições baseadas em Debian ou para `/var/log/messages` em sistemas baseados em RedHat (ORLOVSKYI, 2024).

Algumas mensagens de *log* específicas também são armazenadas em outros arquivos em /var/log, mas a linha de fundo é que tudo isso pode, é claro, ser configurado para atender às suas necessidades (ORLOVSKYI, 2024).

Cada mensagem *Syslog* segue o formato do Protocolo *Syslog* da Berkeley Software Distribution (RFC 3164) e consiste em três partes: PRI, HEADER e MSG. O PRI é um código numérico que representa a *facility* e a severidade. O HEADER contém informações de data e hora, bem como o nome da máquina que enviou a mensagem. A MSG é a mensagem real (ORLOVSKYI, 2024). Exemplo:

```
<34>Oct 11 22:14:15 mymachine su: 'su root' failed for lonvick on /dev/pts/8
```

Existem oito níveis de severidade no *Syslog*, variando de 0 (emergência) a 7 (*debug*). Cada grau indica a importância da mensagem, com o nível 0 sendo o mais crítico e o nível 7 o menos crítico. A tabela a seguir apresenta esses níveis de log:

Tabela 2 – Níveis de *Log Syslog* (ORLOVSKYI, 2024)

Nível	**Descrição**
0	Emergência: o sistema é inutilizável
1	Alerta: ação deve ser tomada imediatamente
2	Crítico: condições críticas
3	Erro: condições de erro
4	Aviso: condições de aviso
5	Nota: condições normais, mas significativas
6	Informação: mensagens informativas
7	Debug: mensagens de nível de depuração

Syslog facilities são códigos que representam a origem da mensagem. Existem 24 *facilities* disponíveis, variando de 0 (kern) a 23 (local7). Todas as *facilities* disponíveis podem ser vistas na Tabela 3.

Tabela 3 – *Facilities* do *Syslog* (ORLOVSKYI, 2024)

Código	***Facility***	**Descrição**
0	*kern*	Mensagens do *kernel*
1	*user*	Mensagens de nível de usuário
2	*mail*	Sistema de correio
3	*daemon*	Processos do sistema
4	*auth*	Segurança/autorização
5	*syslog*	Mensagens geradas internamente pelo *syslogd*
6	lpr	Linha de impressora
7	*news*	Rede de notícias
8	*uucp*	Sistema UUCP
9	*clock*	Sistema de relógio
10	*authpriv*	Segurança/autorização (privado)
11	ftp	Servidor *File Transfer Protocol* (FTP)
12	ntp	Protocolo de tempo de rede
13	*logaudit*	*log* de auditoria
14	*logalert*	*log* de alerta
15	cron	Agendador *cron*
16–23	local0–local7	*Facilities* locais reservadas para uso personalizado por processos que não se encaixam nas categorias definidas acima.

Cada *facility* tem um propósito específico, permitindo que as mensagens sejam categorizadas com base em sua origem. Note que esses códigos numéricos podem não ser muito intuitivos ou fáceis de lembrar. Portanto, os *daemons* do *Syslog* muitas vezes usam "palavras-chave" em vez dos códigos numéricos para tornar as mensagens mais legíveis.

Um exemplo disso é a mensagem "facility: daemon", cujo *daemon* do *syslog* pode mostrar no lugar "facility: 3".

Em resumo, o *Syslog* é um protocolo padrão empregado para transmitir *logs* de eventos e mensagens de diagnóstico em uma rede. No entanto, é importante salientar que o *Syslog* não gera *logs* automaticamente. Em vez disso, cabe aos desenvolvedores de aplicativos a responsabilidade de integrar chamadas de *log* em seus programas. Essas chamadas, por sua vez, enviam mensagens ao *daemon Syslog*.

O audit *daemon* (*auditd*), com base em regras e propriedades pré-configuradas, gera entradas de *log* para registrar informações sobre os eventos que estão acontecendo no sistema. Os administradores usam essas informações para analisar o que deu errado com as políticas de segurança e melhorá-las ainda mais, tomando medidas adicionais (MADABHUSHANA, 2021).

O *auditd* é uma parte fundamental do Sistema de Auditoria Linux, projetado para monitorar e registrar atividades do ambiente com base em regras definidas pelo administrador. Ele captura informações detalhadas sobre eventos relevantes para a segurança,

registrando-os para análise posterior. Isso pode variar desde a inicialização e desligamento do SO até acesso a arquivos, eventos de rede e até mesmo tentativas de violação de segurança (GUR, 2024).

O `audit.rules` é um arquivo contendo regras de auditoria que serão carregadas pelo *script* de inicialização do daemon sempre que ele for iniciado. O programa `auditctl` é usado pelos *initscripts* para realizar essa operação. A sintaxe para as regras é essencialmente a mesma que ao digitar um comando `auditctl` em um *prompt* de *shell*, exceto por não ser necessário digitar o nome do comando `auditctl`, pois isso é implícito (GRUBB, s.d.).

As diretrizes do *auditd* podem ser amplamente categorizadas em controle, sistema de arquivos e chamada de sistema. Os critérios de controle gerenciam como o sistema de auditoria opera, definindo parâmetros como o número máximo de políticas ativas ou quanto tempo os *logs* são mantidos. Os parâmetros do sistema de arquivos especificam o monitoramento do acesso a certos arquivos ou diretórios. As normas de chamada de sistema são usadas para rastrear o uso de chamadas de sistema por usuários ou processos. Essas diretrizes podem ser adicionadas ao diretório /etc/audit/rules.d/audit.rules ou diretamente através do comando auditctl. (GUR, 2024).

Para escrever regras para chamadas de sistema:

```
-a [action], [filter] -S [syscall] -F [field=value] -k [keyname]
```

Pode-se utilizar a *flag* `-a` seguida por `action, filter` para escolher quando um evento é registrado, onde `action` pode ser *always* (sempre criar um evento) ou *never* (nunca criar um evento) (GROENEVELDT, 2024).

Tabela 4 – Filtros e suas descrições (GROENEVELDT, 2024)

Filtro	**Descrição**
task	registra eventos de criação de tarefas
entry	registra pontos de entrada de chamadas de sistema
exit	registra saídas/resultados de chamadas de sistema
user	registra eventos do espaço do usuário
exclude	exclui eventos do registro

O restante dos parâmetros utilizados na criação de regras pode ser observado na Tabela 5, a seguir:

Tabela 5 – Opções de comando e suas descrições (GROENEVELDT, 2024)

Opção	**Descrição**
-S	Especifica a chamada de sistema a ser monitorada (nome ou número).
-F	Define um ou mais filtros para selecionar critérios de correspondência.
-k	Identificador de chave que permite agrupar e facilitar buscas nos registros.

Para escrever regras de controle, os seguintes comandos podem ser utilizados:

```
-D
-b 8192
-f 1
--backlog_wait_time 60000
```

A Tabela 6 descreve a função de cada um desses comandos:

Tabela 6 – Comandos de controle e suas descrições (GROENEVELDT, 2024)

Comando	**Descrição**
-D	Remove todas as regras existentes ao iniciar. Isso garante que o *auditd* leia as configurações de forma limpa, da primeira à última linha.
-b 8192	Define o número máximo de *buffers* de auditoria no *kernel*.
-f 1	Define o modo de falha do *auditd* como "logar falhas".
–backlog_wait_time 60000	Estabelece o tempo de espera (em milissegundos) para o sistema aguardar antes de descartar registros ao atingir o limite de *backlog*.

Para monitorar arquivos e diretórios, a seguinte regra pode ser aplicada:

```
-w [path-to-file] -p [permissions] -k [keyname]
```

A Tabela 7 apresenta a descrição de cada parâmetro:

Tabela 7 – Parâmetros de monitoramento de arquivos (GROENEVELDT, 2024)

Comando	**Descrição**
-w	Caminho do arquivo ou diretório a ser monitorado.
-p	Permissões a serem monitoradas: leitura (r), escrita (w), execução (x) e alteração de atributos (a).
-k	Nome do identificador de chave, usado para facilitar a filtragem dos registros.

Exemplo 1: Monitoramento do arquivo `/etc/passwd`:

```
-w /etc/passwd -p warx -k passwd_changes
```

Essa regra monitora o arquivo `/etc/passwd` para operações de escrita (w), alteração de atributos (a), leitura (r) e execução (x). Os eventos capturados são marcados com a chave `passwd_changes`, facilitando sua identificação nos *logs* (GUR, 2024).

Exemplo 2: Monitoramento de chamadas relacionadas ao *login/logout*:

```
-a always,exit -F arch=b64 -S sethostname -S setdomainname -k system-locale
```

Essa regra configura a auditoria para sempre registrar (`-a always`) chamadas do sistema ao serem concluídas (`exit`) em sistemas com arquitetura de 64 bits (`arch=b64`). São monitoradas especificamente as chamadas `sethostname` e `setdomainname`, e os eventos gerados são identificados com a chave `system-locale` (GUR, 2024).

Em termos gerais, o *auditd* é uma ferramenta extremamente poderosa para controle e monitoramento detalhado de diversas operações no SO. Assim como ocorre com o *syslog*, sua eficácia depende de uma configuração adequada, exigindo conhecimento por parte de administradores ou desenvolvedores para garantir seu uso correto.

2.1.4 Instrumental usado

Durante a pesquisa foi necessário utilizar ferramentas para auxiliar na emulação da falha e análise das consequências no SO. Abaixo detalho o funcionamento de cada uma dessas ferramentas.

2.1.4.1 Injeção de falta no *kernel* Linux

A injeção de falta é uma técnica essencial utilizada para avaliar a robustez e a resiliência de sistemas computacionais, permitindo a emulação controlada de condições adversas e a observação do comportamento do sistema sob tais circunstâncias.

No contexto deste trabalho, falta é definida como uma modificação deliberada inserida em um ponto específico do fluxo de execução, por exemplo, a alteração do valor de retorno ou a inserção de um comportamento anômalo em uma chamada de sistema suportada pelo mecanismo de injeção. Essa modificação, quando executada, provoca um erro no estado interno do sistema ou da aplicação. Importante destacar que o erro resultante pode ou não evoluir para uma falha: em alguns casos observados experimentalmente, uma única ocorrência do erro não foi suficiente para interromper a aplicação, sendo necessárias múltiplas ocorrências para que a falha (como a terminação inesperada do processo) se manifestasse.

No *kernel* Linux, existe uma infraestrutura dedicada à injeção de falta que facilita a implementação dessa técnica e permite explorar diferentes cenários de erro para identificar e corrigir vulnerabilidades antes que estas afetem ambientes de produção (COMMUNITY, 2024).

Para utilizar essa infraestrutura, é necessário recompilar o *kernel* com as opções de configuração correspondentes habilitadas (por exemplo, `CONFIG_FAULT_INJECTION` e parâmetros específicos da falha a ser injetada) e alterar o arquivo de configuração do *kernel* (`.config`) para ativar os módulos de interesse. Somente após essa preparação

é possível utilizar a interface exposta pelo sistema de arquivos `debugfs`, que permite ajustar parâmetros como: probabilidade de ocorrência do erro, intervalo entre injeções, número máximo de ocorrências e nível de verbosidade das mensagens de *log*. Por exemplo, o arquivo `/sys/kernel/debug/fail_slab/probability` permite definir a probabilidade percentual de um erro ser injetado em chamadas subsequentes (COMMUNITY, 2024).

A seguir, apresenta-se a lista de tipos de injeção disponíveis:

Tabela 8 – Opções de injeção de faltas no *Kernel* Linux (COMMUNITY, 2024)

Nome	Descrição
failslab	Injeta falta em alocações de memória do tipo slab, afetando funções como `kmalloc()` e `kmem_cache_alloc()`.
fail_page_alloc	Injeta falta em alocações de páginas de memória, impactando chamadas como `alloc_pages()` e `get_free_pages()`.
fail_futex	Injeta falta em operações de futex, permitindo a emulação de *deadlocks* e erros relacionados ao endereço do usuário.
fail_make_request	Injeta falta de entrada/saída em dispositivos de bloco específicos, afetando a função `generic_make_request()`.
fail_mmc_request	Injeta falta em operações MMC em dispositivos específicos.
fail_function	Injeta falta em funções específicas do *kernel*, que são marcadas com a macro `ALLOW_ERROR_INJECTION()`.
NVMe fault injection	Injeta falta de status NVMe e sinalizadores de nova tentativa em dispositivos NVMe específicos.

Para que a injeção de falta funcione corretamente, é preciso configurar diversos parâmetros através da interface `debugfs`. Os principais incluem:

- **Tipo da falta:** Define qual mecanismo de injeção será ativado (ex: `fail_function`, `failslab`, conforme Tabela 8).
- **Função Alvo:** Especifica a função do *kernel* onde a falta deve ser potencialmente injetada (relevante para `fail_function`).
- **Probabilidade (*probability*):** Indica a porcentagem (0 a 100) de chance de a falta ocorrer a cada chamada elegível.
- **Filtro de Tarefa (*task_filter*):** Determina se a injeção afetará apenas o *Process Identifier* (PID) especificado (requer configuração adicional por processo) ou todas as tarefas do sistema.
- **Intervalo (`interval`):** Define quantas vezes a função alvo deve ser chamada sem falta entre duas injeções consecutivas. Um valor 0 geralmente significa que a falha pode ocorrer em todas as chamadas elegíveis (respeitando a probabilidade).

- **Número de Vezes (`times`):** Limita o número total de vezes que a falta será injetada.
- **Contador de espaço (`space`):** Funciona como um contador regressivo. A injeção só começa a ocorrer após este valor chegar a zero. Ele é decrementado pelo parâmetro `size` a cada chamada da função `should_fail(size)`.
- **Verbosidade (`verbose`):** Controla a quantidade de mensagens de diagnóstico que a infraestrutura de injeção de falta envia para o *log* do *kernel* (*dmesg*).

2.1.4.2 *Ftrace*

O *Ftrace* é um rastreador interno projetado para ajudar desenvolvedores e projetistas de sistemas a encontrar o que está acontecendo dentro do *kernel*. Ele pode ser usado para depurar ou analisar latências e problemas de desempenho que ocorrem fora do espaço do usuário (ROSTEDT, 2008).

Embora o *ftrace* seja normalmente considerado um rastreador de funções, ele é, na verdade, uma estrutura com diversos utilitários de rastreamento. Há um rastreamento de latência para examinar o que ocorre entre interrupções desabilitadas e habilitadas, bem como para preempção e desde o momento em que uma tarefa é ativada até o momento em que ela é efetivamente agendada (ROSTEDT, 2008).

Um dos usos mais comuns do *ftrace* é o rastreamento de eventos. Em todo o *kernel*, existem centenas de pontos de eventos estáticos que podem ser habilitados por meio do sistema de arquivos *tracefs* para ver o que está acontecendo em determinadas partes do *kernel* (ROSTEDT, 2008).

O funcionamento do *ftrace* é baseado no sistema de arquivos *tracefs*, geralmente montado em `/sys/kernel/tracing`. O *ftrace* utiliza o *tracefs* para armazenar arquivos de controle e exibir a saída do *ftrace*. Quando o *tracefs* é configurado no *kernel* (o que ocorre automaticamente ao selecionar qualquer opção do *ftrace*), o diretório `/sys/kernel/tracing` é criado. Para montar esse diretório, pode-se adicionar a seguinte linha ao arquivo `/etc/fstab`:

```
tracefs        /sys/kernel/tracing        tracefs defaults        0       0
```

Alternativamente, a montagem pode ser realizada em tempo de execução com o comando:

```
mount -t tracefs nodev /sys/kernel/tracing
```

Para facilitar o acesso, a documentação sugere criar um link simbólico com:

```
ln -s /sys/kernel/tracing /tracing
```

Essa configuração garante que os arquivos de controle e saída do *ftrace* estejam acessíveis para configuração e análise (ROSTEDT, 2008).

A Tabela 9 apresenta alguns arquivos-chave para o funcionamento do *ftrace*:

Tabela 9 – Arquivos do *tracefs* e suas funções, conforme (ROSTEDT, 2008)

Nome	Descrição
current_tracer	Define/exibe o *ftrace* atual, limpa o *buffer* ao mudar.
tracing_on	Habilita/desabilita escrita no *buffer* do *ftrace* (0/1).
trace	Saída legível para humanos, estática, limpa com escrita O_TRUNC.
buffer_size_kb	Define/exibe tamanho do *buffer* por CPU em Kilobyte (KB).
set_ftrace_pid	Traça *threads* com PIDs listados, suporta *"function-fork"*.

O *ftrace* é amplamente utilizado em cenários como depuração de *drivers*, análise de latências em chamadas de sistema (ROSTEDT, 2008).

2.1.4.3 Medição de tempo dentro do *kernel*

A função `ktime_get_ns` é uma ferramenta essencial do *kernel* Linux para obter medições de tempo de alta precisão dentro do ambiente do *kernel*. Ela retorna o tempo atual em nanosegundos, representado como um inteiro de 64 bits (`u64`), e é baseada no relógio `CLOCK_MONOTONIC`. Esse relógio é monotônico, ou seja, avança continuamente a partir de um ponto de partida não especificado (geralmente o momento do *boot* do sistema) e não é afetado por ajustes no tempo do sistema, como mudanças de fuso horário, sincronização de tempo via *Network Time Protocol* (NTP) ou operações como `settimeofday`. Essa característica torna `ktime_get_ns` ideal para medir intervalos de tempo, garantindo que as medições sejam consistentes e não sejam influenciadas por alterações externas no tempo do sistema (The Linux Kernel Organization, 2023).

Conforme descrito na documentação oficial do *kernel* Linux (The Linux Kernel Organization, 2023), `ktime_get_ns` é uma variante das funções `ktime_get`, mas retorna diretamente o tempo em nanosegundos, o que é mais conveniente para certos casos de uso. Essa abordagem permite cálculos precisos de duração de operações ou eventos dentro do *kernel*, sendo essencial para análise de desempenho e depuração. Por exemplo, para medir o tempo de execução de uma operação específica, pode-se usar o Código 1.

```
1 begin
2     u64 start_time = ktime_get_ns();
3     u64 end_time = ktime_get_ns();
4     u64 duration = end_time - start_time;
5 end
```

Código 1 – Exemplo de uso do `ktime_get_ns`.

No Código 1, a variável `duration` contém o tempo decorrido em nanosegundos, permitindo uma análise precisa do desempenho da operação.

A escolha de `ktime_get_ns` para projetos que exigem precisão temporal é justificada por sua capacidade de fornecer tempos com granularidade de nanosegundos, necessária para medir com exatidão a execução de funções ou o tempo entre eventos críticos. Além disso, sendo baseada em `CLOCK_MONOTONIC`, ela garante que as medições não sejam afetadas por alterações no tempo do sistema, assegurando consistência nas análises temporais.

2.2 Trabalhos Correlatos

A maioria dos estudos que se dedica à análise de dados de falhas de SOs concentra-se predominantemente na plataforma Windows. No contexto deste capítulo, serão abordados alguns estudos significativos que empregaram técnicas de instrumentação para coleta de dados, com o objetivo de obter informações relevantes sobre confiabilidade. Esses estudos representam contribuições importantes para a compreensão e o aprimoramento da confiabilidade dos SOs.

O estudo de Matias et al. (2014) analisou mais de 30.000 falhas reais de SO coletadas em diferentes ambientes de trabalho do Windows 7. Os resultados mostraram que a principal causa de falhas do SO está relacionada aos serviços de SO e que as distribuições Gama e Weibull apresentaram o melhor ajuste aos dados de falhas de SO. As falhas do *kernel* foram mais predominantes em ambientes corporativos do que em acadêmicos .

Murphy (MURPHY, 2004), bem como Yuan e Zhang (YUAN; ZHANG, 2011) mostram, em seus estudos, que falhas de software tendem a crescer de acordo com o tamanho dos programas. Isso significa que, quanto mais complexo e extenso for um programa, maior será a probabilidade de falhas ocorrerem. Ademais, a natureza integrada dos subsistemas de um SO consiste em outro desafio, uma vez que torna a propagação de erros no *kernel* difícil de prevenir e isolar (TANENBAUM; HERDER; BOS, 2006). Isso pode levar a falhas em cascata e problemas de segurança. Portanto, é importante desenvolver estratégias para aumentar a confiabilidade e a segurança dos SOs.

Sob a perspectiva de (SWIFT; BERSHAD; LEVY, 2003), as extensões do *kernel* do SO (por exemplo, *drivers* de dispositivo) correspondem a mais de 70% do código do *kernel* do Linux, enquanto o Windows XP apresenta mais de 35.000 *drivers* diferentes. Esse estudo indica que as extensões de terceiros consistem em uma das principais causas de falhas do Windows XP, onde os *drivers* foram responsáveis por 85% das falhas relatadas. Achados semelhantes foram encontrados no Linux (CHOU et al., 2001), onde os erros de *drivers* foram predominantes em comparação com outras partes do *kernel*.

No estudo conduzido por (GANAPATHI; PATTERSON, 2005), utilizou-se o software Microsoft's Corporate Error Reporting a fim de coletar dados relacionados a falhas em SOs. Especificamente, foram considerados apenas eventos ocorridos no nível do *kernel* do SO Windows. Um servidor foi ajustado para receber notificações de falhas de um cliente, representado por um sistema computacional típico, com o objetivo de criar uma base de dados para análises posteriores. A seleção de ambientes para coleta de dados foi realizada nos laboratórios de pesquisa do *Department of Electrical Engineering and Computer Sciences at University of California Berkeley*. A cada ocorrência de um *crash* em um dos computadores do laboratório, era gerado um "*minidump*" contendo informações resumidas sobre o estado do sistema no momento em que o problema ocorreu. Como resultado dessa pesquisa, foi possível constatar que o SO Windows não era o principal responsável pela maioria dos *crashes* observados nos computadores do laboratório, mas sim aplicações de usuário, como os navegadores de internet.

Posteriormente, Ganapathi (GANAPATHI; GANAPATHI; PATTERSON, 2006) realizou uma análise de 2.528 falhas ocorridas no *kernel* do SO Windows XP. Os resultados obtidos reforçaram as descobertas do estudo anterior, assim como os resultados encontrados por (SWIFT; BERSHAD; LEVY, 2003), destacando que os *drivers* de dispositivo são a principal causa das falhas no SO.

Todavia, em (GANAPATHI; PATTERSON, 2005) e (GANAPATHI; GANAPATHI; PATTERSON, 2006), o enfoque específico na coleta de dados de falhas que ocorreram consistiu apenas no nível do *kernel* do SO Windows. No entanto, é importante ressaltar que os dados de falhas coletados por meio da ferramenta desenvolvida neste trabalho vão além das falhas ocorridas exclusivamente no *kernel* do SO.

No estudo realizado por (KALYANAKRISHNAM; KALBARCZYK; IYER, 1999), foram examinados os dados de falhas de 70 servidores Windows NT, que foram coletados durante um período de seis meses. No estudo mencionado, foi empregado um sistema de *log* que tinha como objetivo coletar informações sobre falhas de subsistemas e eventos específicos do SO. O instrumento desenvolvido neste trabalho proporciona a criação de pontos de coleta em qualquer parte do código-fonte do SO Linux. Além disso, no estudo de (MATIAS et al., 2014) e (MATIAS; OLIVEIRA; ARAUJO, 2013), é destacada a relevância de investigar falhas que possam surgir em aplicações, serviços e no núcleo do SO.

Zhang et al. (ZHANG et al., 2020) realizaram um estudo empírico sobre falhas em trabalhos de aprendizado profundo em ambientes compartilhados. O estudo analisou 4.960 falhas reais ocorridas em uma plataforma de *deep learning* da Microsoft, classificando-as em 20 categorias distintas. Entre as descobertas, destaca-se que quase metade dos problemas está relacionada à interação dos programas com o ambiente de execução, e não com a lógica do código em si. Esses achados reforçam a importância de mecanismos

de coleta e análise de falhas diretamente no nível da infraestrutura, objetivo igualmente perseguido pela proposta do LRAC.

Recentemente, Mudgal et al. (MUDGAL; ARBAB; KUMAR, 2024) introduziram o CrashEventLLM, um modelo baseado em aprendizado profundo para interpretar automaticamente grandes volumes de *logs* de falhas, otimizando a análise e a categorização de eventos em *data centers*.

Zhao et al. (ZHAO et al., 2023) apresentaram o AnoFusion, uma plataforma que integra múltiplas fontes de dados (métricas de performance, eventos de sistema, *logs*) para detectar anomalias de forma mais precisa e confiável, identificando situações em que a aplicação falha e gerando dados sobre essas situações.

Esses trabalhos recentes evidenciam uma tendência crescente em direção ao uso de métodos de detecção mais automatizados, baseados em inteligência artificial, além de abordagens mais abrangentes para a caracterização de falhas em SOs. Muitos desses estudos exploram dados gerados nativamente pelo próprio SO, uma estratégia que o LRAC visa aprimorar, fornecendo mecanismos mais estruturados e completos para a coleta e interpretação desses eventos.

3 A Plataforma LRAC

Este capítulo descreve em detalhe o LRAC, uma infraestrutura de captura, classificação e armazenamento de eventos de falha para o SO Linux. A LRAC foi originalmente proposta como prova de conceito por (MACIEL; MATIAS, 2015).

3.1 Contextualização e Motivação

A experiência bem-sucedida do RAC no ecossistema Windows demonstrou que registros de falha ricos e normalizados possibilitam pesquisas avançadas em engenharia de confiabilidade de software. No Linux, os mecanismos tradicionais *dmesg* e *syslog* registram apenas informações genéricas, dificultando a distinção entre falhas semanticamente diferentes. Para suprir essa lacuna, MACIEL; MATIAS propuseram o LRAC, cujo objetivo central é coletar, em tempo real, dados estruturados sobre falhas que ocorrem tanto no espaço de *kernel* quanto no espaço de usuário do Linux (MACIEL; MATIAS, 2015).

3.2 Visão Geral da Arquitetura

A Figura 1 ilustra o fluxo de tratamento de um evento de falha na plataforma LRAC (adaptado de (MACIEL; MATIAS, 2015)). O processo inicia-se com a ocorrência de uma exceção, como uma `Segmentation Fault`.

1. Um **Ponto de Coleta (PC)** intercepta a falha dentro do *kernel*;(MACIEL; MATIAS, 2015)
2. A informação de contexto (identificador da falha, fonte do erro) é capturada pelo **Módulo de Coleta de Dados de Falha (MCDF)**;(MACIEL; MATIAS, 2015)
3. O MCDF armazena a notificação em um *buffer* interno e expõe os dados via arquivo virtual `/proc/lrac_data`;(MACIEL; MATIAS, 2015)
4. O *daemon* de serviço de armazenamento de registros de falha (**`lracd`**) lê o arquivo `/proc/lrac_data`, formata os registros no padrão definido e persiste em um arquivo *American Standard Code for Information Interchange* (ASCII);(MACIEL; MATIAS, 2015)
5. Ferramentas externas podem utilizar os dados armazenados para análise de confiabilidade.(MACIEL; MATIAS, 2015)

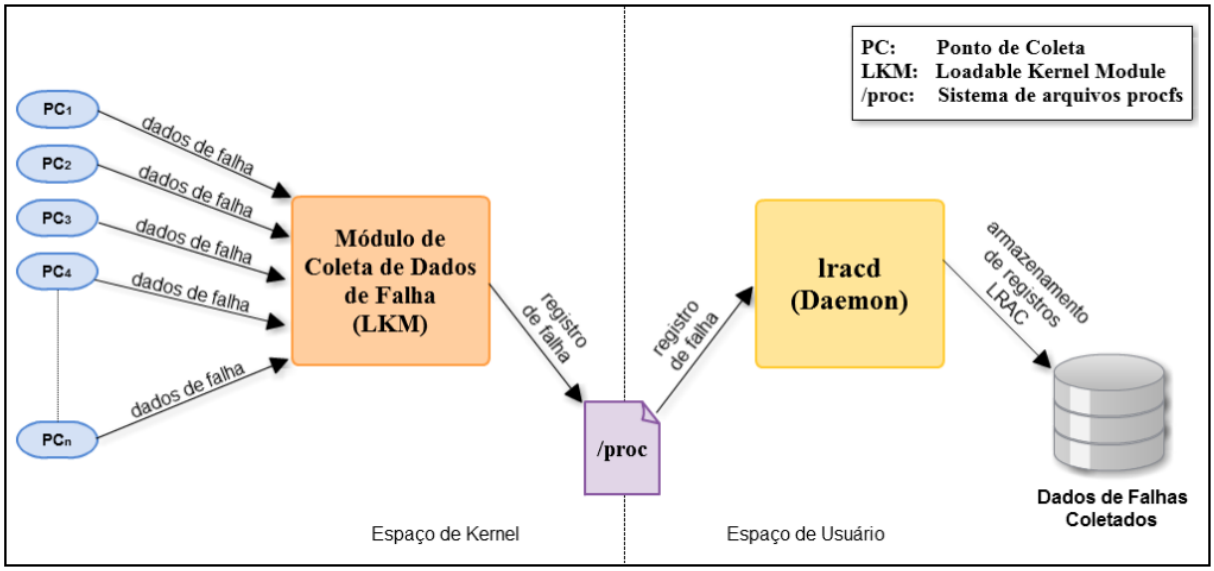

Figura 1 – Fluxo de coleta de um evento de falha na plataforma LRAC (MACIEL; MATIAS, 2015).

3.3 Componentes Principais

Esta seção tem como objetivo fornecer detalhes sobre os principais componentes do LRAC.

3.3.1 Pontos de Coleta (PCs)

Um PC é uma localização no código do *kernel* Linux onde é possível detectar a ocorrência de falhas de software e acionar o LRAC (MACIEL; MATIAS, 2015). Cada PC chama a função `lrac_collecting_point()`, passando informações sobre o evento, como o identificador da falha e o endereço relacionado (não apenas o ponteiro de registradores). A seleção dos PCs é baseada na análise do código-fonte do *kernel* e em práticas semelhantes às utilizadas por instrumentos como o *ftrace* (MACIEL; MATIAS, 2015).

3.3.2 Módulo de Coleta de Dados de Falha (MCDF)

O MCDF é implementado como um módulo de *kernel* carregável (*Loadable Kernel Module* - LKM) (MACIEL; MATIAS, 2015). Seu papel é capturar as notificações de falhas recebidas dos PCs e armazená-las temporariamente em um *buffer* interno. Os dados são disponibilizados para o espaço de usuário por meio de um arquivo no sistema `/proc`, utilizando mecanismos de espera em `wait_queue` para acordar o processo de coleta (MACIEL; MATIAS, 2015).

3.3.3 Serviço `Lracd`

O *daemon* `lracd` é iniciado no espaço de usuário e atua em conjunto com o MCDF (MACIEL; MATIAS, 2015). Seu funcionamento é:

1. Abrir o arquivo `/proc/lrac_data`;
2. Em *loop*, ler registros de falha, formatá-los de acordo com a estrutura `lrac_record` e armazená-los em arquivo ASCII no disco;
3. Registrar a hora do evento no banco de dados de falhas.

```
struct lrac_record {
    unsigned int    failure_id;
    pid_t           source_id;
    char            source_name[SNAME_LEN];
    char            date_ymd[11];
    char            time_hms[9];
};
```

Figura 2 – Estrutura de dados de falha do LRAC (MACIEL; MATIAS, 2015).

4 Proposta de Detecção de Falhas silenciosas no LRAC

Este capítulo descreve os experimentos realizados para avaliar a viabilidade de integrar ao LRAC um mecanismo capaz de registrar falhas silenciosas. O objetivo central é verificar, em ambiente controlado, se a análise temporal entre a ocorrência de uma falha e o término do processo pode ser utilizada para distingui-la de um encerramento normal. Para tanto, foram consideradas duas hipóteses:

1. Os mecanismos padrão de *logging* (*dmesg*, *Syslog*) não são capazes de registrar falhas silenciosas, o que reforça a necessidade de utilizar métodos alternativos de detecção, como o proposto neste estudo
2. O intervalo de tempo entre a falha e o término do processo apresenta um padrão característico, capaz de diferenciar falhas silenciosas de encerramentos normais.

A fim de verificar tais hipóteses, adotou-se a chamada de sistema `read()` como estudo de caso, buscando identificar padrões temporais e avaliar a viabilidade dessa medição como método de distinção. Ressalta-se que a *system call* `read()` integra a infraestrutura de injeção de faltas do *kernel*, o que otimiza a prova de conceito; entretanto, a metodologia descrita é aplicável a quaisquer outras chamadas de sistema ou classes de falha. As seções seguintes detalham o ambiente experimental, os procedimentos adotados e os resultados obtidos.

4.1 Ambiente Experimental

Todos os experimentos foram conduzidos em um ambiente de hardware composto por um processador Intel Core i5-10300H, 16 GB de memória RAM e 50 GB de espaço em disco disponível. O SO utilizado foi o Fedora 39. A versão base do *kernel* Linux utilizada foi a 6.1, sendo posteriormente atualizada para a versão 6.1.11. Abaixo estão listadas as aplicações utilizadas na pesquisa, selecionadas com base em critérios de variabilidade e popularidade. Dessa forma, a escolha contemplou diferentes categorias de uso, como comunicação, edição de imagens e vídeos, navegação web, reprodução multimídia e ferramentas de desenvolvimento. Além disso, os aplicativos incluídos estavam entre aqueles que eram amplamente utilizados pela comunidade Linux, a fim de refletir cenários reais de uso.

Tabela 10 – Aplicações utilizadas nos testes de injeção de falta e análise.

Aplicação	**Aplicação**	**Aplicação**
nano	Gedit	VLC
GIMP	Thunderbird	Inkscape
Audacity	VS Code	Blender
Discord	GNU Octave	Wireshark
TeamViewer	Zoom	Kate
KolourPaint	Krita	Pinta
digiKam	Skype	Google Chrome
Firefox	Elinks	Opera
Midori	Telegram	Kodi (XBMC)
SMPlayer	Totem	Konqueror
Midnight Commander	GNOME Files (Nautilus)	Thunar
Nemo	PCManFM	R

4.2 Verificação de *logs* do Sistema

A fase inicial objetivou verificar a capacidade dos mecanismos de *log* padrão do SO em registrar as falhas silenciosas de uma aplicação, isto é, o encerramento espontâneo decorrente de uma falha ocorrida durante a chamada de sistema `read()`.

4.2.1 Procedimento de injeção de falta

Para emular faltas no nível de *kernel*, utilizou-se a infraestrutura nativa de injeção de faltas do Linux, previamente habilitada durante o processo de compilação por meio da alteração do arquivo `.config`. Essa configuração ativa o subsistema responsável por induzir, de forma controlada, erros em chamadas de sistema, os quais podem ou não resultar em falhas observáveis. Assim, com o *kernel* instrumentado, foi desenvolvido um *script* (*readInjectionFail.sh*), cujo pseudocódigo é apresentado no Código 2, com a finalidade de automatizar a configuração e a execução da injeção.

No início da execução, o *script* define o tipo de falha (*fail_function*) e a função alvo (*___x64_sys_read*), correspondente à chamada de sistema `read()`. Em seguida, são configurados parâmetros essenciais para o experimento, como o valor de retorno forçado (*retval*), o filtro de *task* para selecionar a execução a partir de um processo específico, a probabilidade de ocorrência do erro, fixada em 100%, para garantir sua manifestação e o intervalo entre ativações. O parâmetro */times* foi configurado com um valor elevado (999999999) para ampliar o espaço de execução e facilitar a análise, evitando a necessidade de reiniciar o *script* em execuções prolongadas. Ajustes complementares, como *verbose* e *space*, também são aplicados para fins de monitoramento e depuração.

```
procedure falha_kernel_sys_read
begin
    TIPO ← "fail_function"
    FUNCAO ← "__x64_sys_read"

    escrever("/sys/debug/" + TIPO + "/inject", FUNCAO)
    escrever("/sys/debug/" + TIPO + "/" + FUNCAO + "/retval", 100)
    escrever("/sys/debug/" + TIPO + "/task-filter", "Y")
    escrever("/sys/debug/" + TIPO + "/probability", 100)
    escrever("/sys/debug/" + TIPO + "/interval", 0)
    escrever("/sys/debug/" + TIPO + "/times", 999999999)
    escrever("/sys/debug/" + TIPO + "/space", 0)
    escrever("/sys/debug/" + TIPO + "/verbose", 0)

    msg("Pressione uma tecla para finalizar.")
    aguardar_entrada()

    limpar_injecao("/sys/debug/" + TIPO + "/inject")
end
```

Código 2 – Pseudo código para ativar a injeção de falta no Linux.

De forma complementar, o procedimento experimental consistiu em executar a aplicação alvo e, em seguida, identificar o seu PID. Com essa informação, a injeção de falta foi ativada no *shell* por meio do comando `echo 1 | sudo tee /proc/[PID]/make-it-fail`, direcionando a modificação para o processo em questão. Essa falta, quando atingia o ponto de injeção durante a execução, provocava um erro no fluxo normal da aplicação, que poderia ou não evoluir para uma falha perceptível.

Em aplicações compostas por múltiplas *threads*, a injeção de falta direcionada apenas à *thread* principal não garante que o comportamento resultante afete todo o processo, pois as demais *threads* não injetadas podem continuar executando normalmente, produzindo resultados parciais ou mascarando o efeito do erro.

Para contornar essa limitação, foi desenvolvido o *script* apresentado no Código 3. Ele identifica o PID da aplicação alvo, constrói o caminho para o arquivo de controle `/make-it-fail` e executa o comando de injeção de forma a abranger todas as *threads* associadas ao processo. Dessa forma, a injeção é propagada a toda a aplicação, evitando que partes não afetadas permaneçam em execução e garantindo que o experimento reflita fielmente o impacto do erro induzido em um ambiente *multithread*.

```
procedure AtivaInjectionAppComMultiProcesso(appName)
begin
    command ← "pgrep " + appName
    fp ← PopenRead(command)                    // abre pipe para ler a
        saída do pgrep

    if fp = NULL then
        Imprimir "Erro ao obter os PIDs da aplicação."
        return 1
    endif

    while LerInteiro(fp, pid) = SUCESSO do
        fail_command ← "echo 1 | sudo tee /proc/" + pid +
            "/make-it-fail"
        SystemCall(fail_command)
    end while

    Pclose(fp)                                 // fecha o pipe
end
```

Código 3 – Pseudo código de ativação de injeção de falta em aplicações multi-processo.

Logo após a execução da aplicação e a injeção da falta, observou-se que, em tempo de execução, o processo era encerrado devido ao erro resultante da operação `read()` modificada. Em seguida, os *logs* do sistema foram inspecionados por meio dos comandos `journalctl -b | grep [nome_da_aplicacao]` e `dmesg`, com o objetivo de localizar mensagens que indicassem a causa do encerramento.

Neste trabalho, considera-se como *log* útil aquele que estabelece uma relação explícita entre o motivo do término da aplicação e a falha decorrente da falta injetada, por exemplo, mensagens que indiquem diretamente o encerramento provocado pela modificação na chamada `read()`.

4.3 Pontos de referência para medição temporal no *kernel*

Para que o tempo da ocorrência do erro e o término da aplicação possa ser calculado de forma consistente, foi necessário identificar, no *kernel*, dois pontos de referência fundamentais: o instante inicial, correspondente ao momento exato em que o erro ocorre durante a execução da chamada de sistema `read()`, e o ponto final comum de término, isto é, a etapa do fluxo de encerramento pela qual todos os processos inevitavelmente passam antes de serem finalizados. A identificação precisa desses locais permite padronizar a coleta de tempos e assegurar que a medição do intervalo entre a ativação do erro e o encerramento do processo seja comparável entre diferentes aplicações e cenários de execução.

4.3.1 Identificação do Padrão Comum de Término

Para identificar um ponto de término comum, ou seja, uma função no *kernel* invocada no encerramento de qualquer aplicação, independentemente da forma como ela é finalizada, foi necessário observar e comparar diferentes fluxos de finalização. Essa análise permitiria selecionar um local único e confiável para marcar o instante em que o processo inicia efetivamente seu encerramento. Para essa investigação, foram elaboradas três variações simples de código em formato de pseudocódigo, cada uma simulando um cenário distinto de término de aplicação, a fim de explorar a maior quantidade de variações possíveis:

- **Encerramento com *return***: o programa é finalizado naturalmente com um retorno explícito da função `main`, equivalente a um `exit(0)` implícito, comportamento representado no Código 4.
- **Encerramento com chamada de sistema (*_exit*)**: o código utiliza diretamente a *system call* `_exit(0)` para encerrar o processo, sem executar as rotinas de finalização da biblioteca padrão, como liberação de *buffers*, como está ilustrado no Código 5.
- **Encerramento natural (sem retorno explícito)**: o programa apenas atinge o fim da função *main*, sem retorno declarado. O compilador interpreta esse comportamento como um *return 0* implícito, cenário representado no Código 6.

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
5     sleep(10);
6     return 0;
7 }
```

Código 4 – Código em C utilizada para emular o encerramento da aplicação via instrução *return*

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    sleep(5);
    _exit(0);
}
```

Código 5 – Código em C utilizada para emular o encerramento da aplicação via *system call*

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    sleep(5);
}
```

Código 6 – Código em C utilizada para emular o encerramento natural da aplicação

Em cada execução das variações de código apresentadas anteriormente, foi utilizada a ferramenta de *tracing* nativa do Linux (*ftrace*) para registrar, em nível de *kernel*, a sequência de funções chamadas durante o encerramento de cada processo. No contexto deste trabalho, um *trace* corresponde ao registro cronológico dessas chamadas, armazenado em formato texto, o que permite inspecionar detalhadamente o fluxo de execução até o término do programa.

Inicialmente, analisaram-se os *traces* obtidos nas variações simuladas, comparando-os linha a linha para identificar funções comuns presentes na fase final do encerramento. Essa análise inicial apontou a função `do_exit`, localizada no arquivo `kernel/exit.c`, como candidata recorrente. Para confirmar essa hipótese, foram realizados testes com 16 aplicações reais, escolhidas por representarem diferentes categorias de uso e, principalmente, por já terem demonstrado em testes prévios, com base na análise dos registros do *syslog* e do *dmesg*, que encerravam sua execução de forma controlada quando submetidas à injeção de falta.

Essas aplicações foram submetidas a dois tipos de experimento:

- **Encerramento voluntário**: aplicações executadas e finalizadas manualmente pelo usuário.
- **Encerramento induzido**: aplicações submetidas à injeção de falta, aguardando que o encerramento ocorresse automaticamente.

Em cada execução, o *ftrace* registrou o fluxo completo de chamadas no *kernel*. A inspeção comparativa revelou que, independentemente da forma de encerramento, todas as aplicações analisadas passavam pela função `do_exit` nos instantes finais de execução, confirmando-a como ponto comum de término. Um exemplo de *trace* obtido no fechamento automático do aplicativo Discord, após a injeção de falta, é apresentado a seguir. Todos os demais casos analisados seguiram o mesmo padrão, evidenciando a chamada a `do_exit` como evento terminal:

```
1 ...
2 Discord-10399    [004] .N.1.  3979.256734: do_exit <-do_group_exit
3 Discord-10399    [004] .N.1.  3979.256734: _raw_spin_lock_irq <-do_exit
4 Discord-10399    [004] dN.2.  3979.256734: _raw_spin_unlock_irq <-do_exit
5 Discord-10399    [004] .N.1.  3979.256735: _raw_spin_lock_irq <-do_exit
6 Discord-10399    [004] dN.2.  3979.256735: _raw_spin_unlock_irq <-do_exit
7 Discord-10399    [004] .N.1.  3979.256735: exit_signals <-do_exit
8 ...
```

Código 7 – Trecho do *trace* da aplicação Discord que mostra o método `do_exit` sendo evocado

4.3.2 Identificação do ponto de coleta na chamada *read*

Para identificar o ponto exato no *kernel* onde o erro ocorre na chamada de sistema `read()`, foram adotados dois métodos complementares: a análise das chamadas presentes nos arquivos de *trace* obtidos na etapa anterior e a leitura direta do código-fonte do *kernel*. A partir dessa investigação, localizou-se a definição da chamada de sistema por meio da macro `SYSCALL_DEFINE3`, especificamente na seguinte forma:

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
```

Essa definição encontra-se no arquivo `fs/read_write.c`, responsável por implementar as operações de leitura e escrita no sistema de arquivos. Com base nesse ponto, foi elaborado o pseudocódigo apresentado no Código 8, que registra um carimbo de tempo no momento em que o erro é detectado, armazenando-o no contexto do processo para posterior cálculo do intervalo de interesse.

Entretanto, durante os testes, verificou-se que essa abordagem não era eficaz para o cenário estudado. Isso porque, quando a injeção de falta está ativada, a interceptação ocorre antes da execução da função `read()` analisada, provocando imediatamente um erro e retornando-o à aplicação. Dessa forma, o método definido em `fs/read_write.c` não chega a ser invocado, impossibilitando a captura do instante nesse ponto específico do código.

Diante da limitação observada, foi necessário buscar um novo ponto no *kernel* capaz de registrar o instante em que o erro ocorre efetivamente nesse cenário. Para isso, analisaram-se os métodos responsáveis pela própria mecânica de injeção de falta no Linux.

Essa investigação levou ao arquivo `lib/fault-inject.c`, que contém a função `should_fail_ex`. Essa função é responsável por avaliar, em tempo de execução, se a falta deve ou não ser injetada, retornando o resultado dessa decisão. No fluxo interno dessa função, identificou-se um trecho específico que é executado sempre que a injeção de falta está ativada e o erro é efetivamente produzido. Nesse ponto, foi inserida a lógica de captura de tempo, registrando um *timestamp* de alta precisão no momento exato da ocorrência do erro e armazenando-o no contexto do processo para posterior cálculo do intervalo de interesse.

O pseudocódigo no Código 9 ilustra a implementação dessa modificação, que permitiu a continuidade da pesquisa mesmo com a interceptação ocorrendo antes da chamada original a `read()`.

```
1  procedure SYSCALL_DEFINE3(fd, buf, count)
2  begin
3      ret ← ksys_read(fd, buf, count)  // Chama a função de leitura no
           kernel
4
5      if ret < 0 then
6          timestamp_ns ← ObterTimestampAtualEmNs()
7          TempoQueOcorreuErro ← timestamp_ns  // Armazena timestamp no
               processo atual
8      endif
9
10     retornar ret
11 end
```

Código 8 – Local proposto para implementar a busca do momento que ocorre erro na *syscall read*

```
1  procedure should_fail_ex(attr, size, flags) → booleano
2  begin
3      ...
4
5      falha:
6          timestamp_ns ← ObterTimestampAtualEmNs()
7          TempoQueOcorreuErro ← timestamp_ns
8
9      ...
10 end
```

Código 9 – Local para buscar tempo quando ocorre erro por injeção de falta

4.4 Nomenclatura proposta

Para quantificar o intervalo de tempo transcorrido entre a injeção de falta e o término do processo afetado (métrica denotada por T_3), foram definidos, dentro do *kernel*, dois instantes de referência e o intervalo resultante:

- T_1 — **Momento do erro:** carimbo de tempo gravado imediatamente após o ponto de injeção (por exemplo, logo após a chamada `read()` que introduz o erro).
- T_2 — **Momento de término:** carimbo de tempo capturado no primeiro ponto comum de finalização do processo, isto é, na entrada de `do_exit()` antes da liberação final de recursos.
- T_3 — **Intervalo:** espaço de tempo da entre o momento da falha até o momento do término (T_2 - T_1).

Como eventos e dados manipulados no espaço do *kernel* só podem ser acessados pelo próprio *kernel*, tornou-se necessário garantir que os marcadores de tempo fossem armazenados em uma área pertencente a ele. Para isso, optou-se por utilizar a estrutura `task_struct`, definida no arquivo *include/linux/sched.h*, que é usada pelo *kernel* para representar cada processo e manter dados essenciais sobre seu estado e execução. Ao adicionar os campos T_1, T_2 e T_3 à `task_struct`, tornou-se possível manter os marcadores de tempo acessíveis em qualquer ponto do núcleo, independentemente do contexto de execução.

Como essas variáveis são criadas no momento em que o processo nasce, foi preciso garantir que iniciassem com valores limpos, evitando que resíduos de memória (*lixo de memória*) afetassem os resultados. Para isso, no arquivo *kernel/fork.c*, dentro do método `copy_process`, responsável por inicializar a nova estrutura de processo, cada campo foi explicitamente configurado com zero.

Para facilitar a manipulação dos marcadores de tempo e do código de erro associado, foi criado o arquivo *lrac.h* no diretório *kernel*, contendo *macros* que centralizam essas operações. O Código 10 apresenta essas *macros*:

- **`STORE_T1(err)`** — Registra o instante inicial (T_1) no momento da ocorrência do erro e armazena o código de erro associado.
- **`STORE_T2()`** — Registra o instante final (T_2) no momento em que a aplicação é encerrada.
- **`GET_T3()`** — Calcula o intervalo de tempo (T_3) como a diferença entre T_2 e T_1, retornando o resultado.

- **`GET_LRAC_ERR_NUM()`** — Retorna o código de erro registrado, permitindo identificar o tipo de falha detectada.

```
1  #include <asm/errno.h>
2  #include <linux/ktime.h>
3  #include <linux/sched.h>
4
5  #define STORE_T1(err) do { \
6      current->t1 = ktime_get_ns(); \
7      current->lrac_erro_enum = err; \
8  } while (0)
9
10 #define STORE_T2() do { \
11     current->t2 = ktime_get_ns(); \
12 } while (0)
13
14 #define GET_T3() ({ \
15     current->t3 = current->t2 - current->t1; \
16     current->t3; \
17 })
18
19 #define GET_LRAC_ERR_NUM() (current->lrac_erro_enum)
```

Código 10 – Macro *lrac.h*

4.5 Medição do Tempo de Fechamento da Aplicação

O objetivo desta etapa foi mensurar o tempo de encerramento da aplicação em dois cenários distintos. O primeiro corresponde aos casos em que o término ocorre automaticamente como consequência direta da injeção de falta. O segundo refere-se a um cenário de falso positivo, em que a falta é injetada, mas a aplicação continua em execução e é encerrada apenas posteriormente por ação do usuário.

Para viabilizar essa variação, foi necessário ajustar o parâmetro que controla a quantidade de faltas injetadas por vez. Quando esse valor é configurado para um número elevado, a aplicação tende a encerrar-se automaticamente devido à sequência contínua de erros, causando a falha. Por outro lado, ao definir valores baixos (por exemplo, entre 1 e 10), a aplicação geralmente permanece em execução mesmo após a ocorrência do erro, permitindo que seja finalizada manualmente, caracterizando o falso positivo. Em alguns casos, contudo, não foi possível realizar esse segundo teste, pois, mesmo com apenas uma falta injetada, a aplicação foi encerrada.

Os procedimentos adotados em cada cenário foram os seguintes:

- **Término Induzido pela Falha:** Injeção de falta na `read()` e coleta do T_3 logado atravez da ferramenta *dmesg*.

- **Teste de Falso Positivo (Término Manual):** Injeção de falta, interação por alguns segundos, e fechamento manual pelo usuário. O T_3 resultante coletado atravez da ferramenta *dmesg*.

4.5.1 Término Induzido pela Falha

Este experimento teve como objetivo mensurar o intervalo de tempo entre a ocorrência de uma falha silenciosa e o encerramento da aplicação. Inicialmente, a falta foi injetada; em seguida, o comportamento da aplicação foi monitorado, aguardando-se o seu término. Em determinados casos, foi necessário interagir com áreas específicas, como o menu de configurações, para que o encerramento ocorresse. Após a finalização, procedeu-se à análise dos *logs* do sistema, por meio do comando *dmesg*, a fim de identificar o T_3 vinculado ao PID. Com base nessa informação, determinou-se o tempo decorrido desde a falha até o fechamento da aplicação.

4.5.2 Teste de Falso Positivo

O segundo experimento teve como objetivo mensurar o tempo de fechamento da aplicação em situações de falso positivo, ou seja, quando um erro ocorre, mas o sistema permanece operante e é encerrado manualmente pelo usuário em um momento posterior. Para isso, foi feito a injeção de falta semelhante à do experimento anterior, porém configurada com um número reduzido de ocorrências, de modo a garantir que a aplicação continuasse em execução por um intervalo antes do término. Após a injeção, realizou-se uma breve interação com o sistema, emulando o uso contínuo mesmo após a ocorrência do erro. O encerramento foi feito manualmente, seja pelo botão de fechamento da janela ("X") ou por comandos específicos, no caso de aplicações sem interface gráfica. Em seguida, o comando *dmesg* foi empregado para identificar o T_3 vinculado ao PID, isto é, o tempo decorrido entre a falha e o fechamento manual do programa.

4.5.3 Observações sobre Falsos Positivos

No processo de análise de falsos positivos, foram identificadas situações específicas que impediram a mensuração precisa do tempo de fechamento da aplicação.

Em determinados casos, a aplicação encerrou-se de forma controlada após a ocorrência de um único erro, o que inviabilizou a caracterização desse cenário como um falso positivo. Nesses casos, a falha provocou o fechamento imediato, tornando impossível medir o tempo de encerramento manual pelo usuário.

Além disso, em aplicações com mais de um processo, houve situações em que foi possível registrar o tempo de encerramento apenas para um dos processos, pois, ao

aumentar a intensidade da injeção de falta (configurada no *script*), a aplicação encerrava-se imediatamente.

5 Resultados

Este capítulo apresenta os resultados obtidos ao longo das fases experimentais deste trabalho. Os testes realizados tiveram como foco a análise da ocorrência de falhas silenciosas no sistema operacional Linux.

5.1 Observabilidade do Encerramento via *logs* Padrão

A primeira análise teve como objetivo verificar se o encerramento automático das aplicações, após a injeção de falta na chamada de sistema `read()`, era devidamente registrado pelos mecanismos padrão de *logging* do Linux. Para isso, foram inspecionados os registros produzidos pelo *dmesg* e pelo *journalctl* (*syslog*) imediatamente após cada execução.

A Tabela 11 apresenta um resumo dos resultados obtidos para cada aplicação testada, indicando, nas colunas correspondentes, se houve registro relevante em cada um dos dois mecanismos e, quando aplicável, a figura associada ao conteúdo do *log*. A última coluna indica o comportamento observado no encerramento, incluindo casos em que o aplicativo não apresentou fechamento forçado ou permaneceu em execução.

Tabela 11 – Observabilidade do encerramento de aplicações via *logs* do sistema (*dmesg*, *syslog*) após falha injetada na `read()`.

Aplicação	***Log dmesg***	***Log syslog***	**SO**
Audacity	Vazio	Vazio	Não fechou
Blender	Vazio	Figura 8	Fechou
digiKam	Vazio	Vazio	Figura 3
Discord	Vazio	Figura 9	Fechou
Elinks	Vazio	Vazio	Fechou.
Firefox	Não	Não	Aplicacao parou de responder e não apresentou nada para fechamento forçado.
Gedit	Vazio	Figura 4	Fechou
GIMP	Vazio	Vazio	Não fechou
GNOME Files (Nautilus)	Figura 22	Vazio	A interface continuou funcionando mas sem a possibilidade de abrir nenhum arquivo. Não apresentou opção para fechamento forçado.
GNU Octave	Vazio	Vazio	Figura 3
Google Chrome	Figura 14	Figura 15	Figura 3
Inkscape	Figura 6	Figura 7	Fechou
Kate	Vazio	Vazio	Figura 3
Kodi (XBMC)	Figura 18	Figura 19	Fechou
KolourPaint	Vazio	Vazio	Figura 3
Konqueror	Vazio	Vazio	Figura 3
Krita	Figura 10	Figura 11	Fechou
Midori	Vazio	Figura 17	Fechou
Midnight Commander	Vazio	Vazio	Fechou
Nano	Vazio	Vazio	Fechou
Nemo	Vazio	Figura 24	Fechou
Opera	Vazio	Figura 16	Figura 3
PCManFM	Vazio	Figura 25	Fechou
Pinta	Vazio	Figura 12	Fechou
R	Vazio	Vazio	Fechou
Skype	Vazio	Figura 13	Mensagem "não respondendo".
SMPlayer	Vazio	Figura 20	Figura 3
TeamViewer	Vazio	Vazio	Figura 3
Telegram	Vazio	Vazio	Não fechou
Thunar	Vazio	Figura 23	Fechou
Thunderbird	Vazio	Figura 5	Fechou
Totem	Vazio	Figura 21	Fechou
VLC	Vazio	Vazio	Não fechou
VS Code	Vazio	Vazio	Não fechou
Wireshark	Vazio	Vazio	Figura 3
Zoom	Vazio	Vazio	Figura 3

A partir dessa análise, constatou-se que, das 36 aplicações avaliadas, 17 encerraram-se em decorrência da injeção de falta. Dentre essas, 4 não apresentaram qualquer entrada de *log*, enquanto as demais (13 aplicações) registraram apenas mensagens genéricas, sem estabelecer uma relação explícita entre a falha ocorrida na chamada `read()` e o término do processo. Esse resultado reforça a hipótese levantada no Capítulo 4, de que os mecanismos padrão de *logging* não são suficientes para detectar, de forma confiável, o momento de ocorrência de falhas silenciosas.

A Figura 3 apresenta um caso típico em que não houve geração de *log* textual, mas o sistema exibiu uma mensagem gráfica informando que a aplicação GNU Octave "não está respondendo". Essa interface gráfica oferece ao usuário a opção de aguardar ou forçar o encerramento, caracterizando um fechamento não automático, mas induzido pela percepção de travamento. Esse comportamento foi observado em diversas aplicações

listadas na Tabela 11, como GNU Octave, Wireshark, TeamViewer, Zoom, Kate e outras, que não geraram registros relevantes no *dmesg* ou *syslog*. Esse tipo de ocorrência não é considerado um *log* útil, pois não estabelece uma relação explícita com a falha injetada, mas apenas indica um estado de travamento detectado pelo ambiente gráfico.

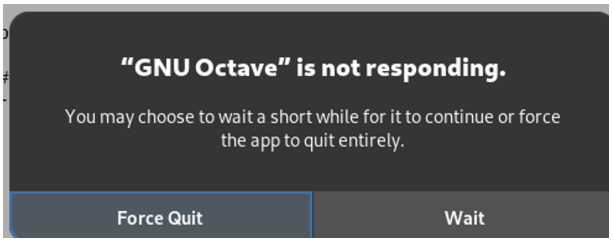

Figura 3 – Mensagem "não respondendo"com opção de fechamento forçado

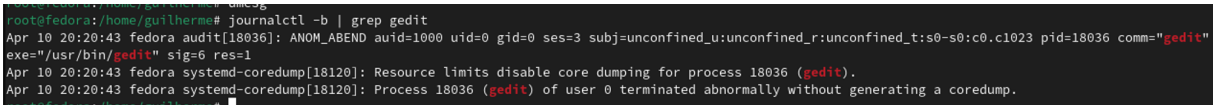

Figura 4 – *Log syslog* da aplicação Gedit

As demais imagens de erro encontram-se reunidas no Apêndice A. A análise cruzada dos *logs dmesg* e *syslog* evidencia que, do total de 37 aplicações avaliadas, apenas 17 se encerraram em decorrência da injeção de falta. Dentre essas, 13 geraram algum tipo de registro, porém as mensagens coletadas mostraram-se insuficientes para estabelecer, de forma inequívoca, a relação causal com a falha silenciosa. Em todos os casos observados, os registros se limitaram a indicar o PID e o status de término, sem oferecer detalhes sobre o motivo do encerramento. Esses resultados reforçam a necessidade de mecanismos complementares de observabilidade capazes de registrar com precisão a ocorrência de falhas silenciosas.

5.2 Medição de Tempo entre Falha e Término (T_3)

Para os testes apresentados a seguir, foram consideradas apenas as 17 aplicações que se encerraram após a injeção de falta. As Tabelas 12 e 13 apresentam os resultados da medição do tempo T_3. Foram realizados 5 testes para cada aplicação e calculado o tempo mínimo, tempo máximo, tempo médio e desvio padrão.

5.2.1 Tempo para Autoencerramento Pós-Falha

A Tabela 12 sumariza os tempos T_3 medidos para os casos em que a aplicação encerrou sua execução espontaneamente após a falha injetada na `read()`.

Tabela 12 – Tempo medido em segundos para aplicações com auto-encerramento após falha na `read()`.

Aplicação	**Tempo Mínimo (s)**	**Tempo Máximo (s)**	**Tempo Médio (s)**	**Desvio Padrão (s)**
Krita	1,330.433.029	1,476.757.726	1,390.551.269	0,058.535.834
Nano	0,000.386.974	0,000.449.602	0,000.417.094	0,000.205.823
Inkscape	0,340.218.793	0,346.844.407	0,343.564.899	0,002.750.237
PCManFM	0,200.667.521	0,211.114.076	0,202.975.324	0,004.084.110
Blender	0,047.200.175	0,115.275.780	0,071.395.544	0,024.863.442
Discord (proc. 1)	1,195.860.337	1,506.393.506	1,315.148.290	0,102.715.616
Discord (proc. 2)	0,848.139.832	1,192.848.409	1,041.219.736	0,124.571.506
Discord (proc. 3)	0,469.822.429	0,668.167.386	0,575.034.388	0,063.402.135
Elinks	0,001.671.277	0,007.127.779	0,005.370.104	0,002.084.871
Gedit	0,368.308.084	0,393.345.982	0,379.718.368	0,009.793.900
Kodi (proc. 1)	0,805.881.895	0,909.116.837	0,871.019.925	0,036.688.224
Kodi (proc. 2)	0,000.117.777	0,000.231.702	0,000.161.997	0,000.043.084
Midnight Commander	0,000.654.569	0,001.547.506	0,001.201.305	0,000.324.055
Midori	1,653.056.308	1,842.023.288	1,730.621.941	0,066.634.618
Nemo	0,279.912.373	0,299.520.695	0,289.181.903	0,006.819.491
Pinta (proc. 1)	1,305.454.391	1,363.323.574	1,334.792.763	0,020.880.369
Pinta (proc. 2)	0,000.140.751	0,000.155.099	0,000.146.371	0,000.049.704
Pinta (proc. 3)	0,000.034.864	0,000.063.358	0,000.043.574	0,000.010.439
R (proc. 1)	0,002.108.665	0,012.806.651	0,006.968.939	0,003.842.907
R (proc. 2)	0,000.034.216	0,000.192.717	0,000.121.760	0,000.046.958
Thunar	0,215.958.957	0,262.175.768	0,233.922.452	0,018.138.473
Thunderbird	3,544.407.652	3,822.598.081	3,669.624.262	0,102.693.338
Totem	0,314.729.524	0,343.780.029	0,323.865.821	0,010.488.390
Média geral	0,563.902.276	0,610.029.818	0,581.616.727	0,025.948.728

A Tabela 12 apresenta os tempos medidos entre a ocorrência da falha na chamada `read()` e o encerramento automático das aplicações afetadas. Os valores estão expressos em segundos e incluem o tempo mínimo, máximo, médio e o desvio padrão para cada aplicação observada. Avaliou-se tanto aplicações monoprocessadas quanto multiprocessadas (identificadas por diferentes processos da mesma aplicação).

Constatou-se uma ampla variação nos tempos médios, com valores que vão de poucos microssegundos, como no caso do terceiro processo do Pinta (aproximadamente 43µs), até vários segundos, como no caso do Thunderbird, que apresentou tempo médio superior a 3,6 s. Aplicações com interface gráfica rica ou comportamento interno mais complexo, como Krita, Midori, Discord e Pinta, apresentaram tempos de resposta maiores. Já programas de linha de comando, como Nano, Elinks e R, exibiram encerramento praticamente imediato após a falha.

De forma complementar, nota-se a consistência nos tempos medidos para cada aplicação, pois os desvios padrão foram majoritariamente baixos. A linha final da tabela apresenta a média geral dos tempos mínimos, máximos, médios e dos desvios padrão, servindo como referência agregada para comparação entre os comportamentos.

5.2.2 Tempo para Encerramento Manual Pós-Falha (Falso Positivo)

A Tabela 13 apresenta os tempos T_3 medidos no cenário em que a aplicação foi encerrada manualmente após a falha. Executaram-se 5 testes para cada aplicação e, posteriormente, foram calculados o tempo mínimo, tempo máximo, tempo médio e desvio padrão.

Tabela 13 – Tempo do encerramento manual da aplicação após a falha `read()`.

Aplicação	**Tempo Mínimo (s)**	**Tempo Máximo (s)**	**Tempo Médio (s)**	**Desvio Padrão (s)**
Krita	4,650.255.297	11,882.439.891	7,576.008.412	2,685.201.676
Nano	4,297.201.617	4,893.324.558	4,689.268.290	0,210.402.648
Inkscape	5,457.842.103	7,954.580.091	6,611.148.557	0,983.199.071
PCManFM	5,243.165.122	9,278.186.810	7,382.102.497	1,331.042.359
Blender	4,546.927.330	9,579.117.203	6,723.174.582	2,018.009.311
Discord (proc. 1)	*			
Discord (proc. 2)	*			
Discord (proc. 3)	*			
Elinks	*			
Gedit	7,907.364.252	11,548.381.937	9,432.226.105	1,374.030.172
Kodi (proc. 1)	9,613.464.438	18,521.801.267	13,699.255.127	3,288.235.490
Kodi (proc. 2)	*			
Midnight Commander	*			
Midori	7,051.413.892	9,000.840.471	7,988.183.264	0,770.436.711
Nemo	5,964.366.317	7,434.616.589	6,919.249.306	0,537.391.097
Pinta (proc. 1)	8,835.832.089	13,749.625.046	10,855.839.545	1,771.059.006
Pinta (proc. 2)	*			
Pinta (proc. 3)	*			
R (proc. 1)	*			
R (proc. 2)	*			
Thunar	5,299.264.589	9,336.446.123	7,393.182.014	1,453.203.859
Thunderbird	6,913.290.861	12,363.570.180	9,606.130.228	1,632.042.390
Totem	5,498.746.132	8,303.384.902	6,966.150.734	0,988.319.473
Média geral	6,251.983.183	10,295.622.518	8,141.506.157	1,476.872.344

* Não foi possível testar pois mesmo com apenas 1 falta injetada a aplicação se fechou.

A Tabela 13 apresenta os tempos de encerramento das aplicações após a injeção da falta na chamada `read()`, considerando que o encerramento foi realizado manualmente pelo usuário. Os tempos estão expressos em segundos, e as medidas incluem valores mínimos, máximos, médios e o desvio padrão.

A média geral observada foi de aproximadamente 8.14 s, com desvio padrão médio de 1.47 s, o que indica uma grande variabilidade entre os comportamentos observados nas diferentes aplicações. Algumas aplicações, como Discord, R, Elinks, entre outras, não puderam ser analisadas por não responderem adequadamente ao processo de encerramento após a falha.

Observa-se que, mesmo nos casos mais rápidos, como Nano (4.29 s) e Blender (4.54 s), o tempo de encerramento manual permaneceu consideravelmente maior do que os tempos registrados anteriormente nos testes de encerramento automático. A maioria das aplicações apresenta tempos médios acima de 4 s, chegando a ultrapassar 13 s em casos como o Kodi (proc. 1).

Esses resultados indicaram que, mesmo nos melhores casos, o autoencerramento segue mais rápido, além de ter uma variância maior, pois depende de um *input* externo (usuário fechar manualmente).

6 Conclusão

Este trabalho teve como objetivo analisar falhas em aplicações no ambiente Linux, com foco em identificar padrões de encerramento automático após falhas silenciosas na chamada `read()`, contribuindo com o subsistema LRAC. A proposta foi avaliada por meio de experimentos que envolveram injeção de falta, instrumentação do *kernel* e análise dos *logs* gerados pelas aplicações testadas.

Os resultados mostraram que, apesar do uso de mecanismos de *log* padrão como *syslog* e *dmesg*, muitas aplicações não registram eventos úteis de encerramento. Isso evidencia a limitação dos sistemas atuais em detectar falhas do tipo *non-crash-based*, reforçando a relevância da proposta deste trabalho. A estratégia adotada — utilizando pontos comuns de término (`do_exit`) e medição temporal (`ktime_get_ns`) — permitiu obter métricas precisas sobre o comportamento das aplicações diante de falhas, mesmo em contextos com múltiplos processos ou encerramento manual.

Entretanto, o desenvolvimento enfrentou desafios importantes, entre eles a necessidade de implementar pontos de coleta específicos para cada tipo de erro, o que demanda um mapeamento criterioso do código do *kernel*. Além disso, destaca-se o Thunderbird, cuja finalização demandou mais de 3,6 s. Esse comportamento pode estar associado ao fato de a aplicação realizar acesso a um grande número de arquivos antes de encerrar. Embora esse tempo elevado não inviabilize os demais experimentos, representa um sinal de alerta que merece investigação mais aprofundada.

Como trabalhos futuros, propõe-se:

- Investigação das causas que levam aplicações com maior consumo de memória a apresentarem tempos de finalização mais elevados.
- A extensão do LRAC para incorporar heurísticas que identifiquem encerramentos causados por erros internos das aplicações com o uso da *system call* do tipo `read()`.
- A aplicação desta abordagem em outros tipos de *system calls*, além de `read()`.
- A criação de um padrão para a inserção desse tipo de erro no LRAC.

Em síntese, este trabalho apresentou uma prova de conceito que demonstrou ser possível detectar e caracterizar falhas silenciosas de forma integrada ao *kernel* Linux. Os resultados obtidos reforçam a relevância da proposta e indicam que há espaço para avançar nesse caminho, seja pela aplicação a outros tipos de chamadas de sistema, seja pela definição de padrões mais abrangentes de coleta no LRAC.

Referências

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. **IEEE Transactions on Dependable and Secure Computing**, IEEE, v. 1, n. 1, p. 11–33, 2004. Disponível em: <https://ieeexplore.ieee.org/document/1335465>. Citado 5 vezes nas páginas 12, 13, 14, 15 e 16.

CANDEA, G.; FOX, A. Crash-only software. In: **Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9**. USA: USENIX Association, 2003. p. 12. Citado 2 vezes nas páginas 15 e 16.

CHILLAREGE, R. What is software failure? **IEEE Transactions on Reliability**, IEEE, v. 45, n. 3, p. 354–355, 1996. Disponível em: <https://ieeexplore.ieee.org/document/536980?arnumber=536980>. Citado na página 13.

CHOU, A. et al. An empirical study of operating systems errors. In: ACM. **Proceedings of the 18th ACM symposium on Operating systems principles**. Stanford, CA, USA, 2001. v. 35, n. 1, p. 73–88. Disponível em: <https://dl.acm.org/doi/10.1145/502034.502042>. Citado na página 26.

COMMUNITY, T. L. K. D. **Fault Injection Capabilities Infrastructure**. 2024. Acessado em abril de 2025. Disponível em: <https://docs.kernel.org/fault-injection/fault-injection.html>. Citado 3 vezes nas páginas 4, 22 e 23.

DAVID, F.; CARLYLE, J.; CAMPBELL, R. Exploring recovery from operating system lockups. In: . Santa Clara, CA, USA: [s.n.], 2007. p. 351–356. Disponível em: <https://www.researchgate.net/publication/220881043_Exploring_Recovery_from_Operating_System_Lockups>. Citado na página 9.

GANAPATHI, A.; GANAPATHI, V.; PATTERSON, D. Windows xp kernel crash analysis. In: **Proceedings of the 20th Conference on Large Installation System Administration**. USA: USENIX Association, 2006. (LISA '06), p. 12–12. Citado na página 27.

GANAPATHI, A.; PATTERSON, D. A. Crash data collection: a windows case study. In: IEEE. **2005 International Conference on Dependable Systems and Networks (DSN'05)**. Yokohama, Japan, 2005. p. 280–285. Disponível em: <https://ieeexplore.ieee.org/document/1467802>. Citado na página 27.

GERHARDS, R. **The Syslog Protocol**. [S.l.], 2009. Internet Standard. Disponível em: <https://www.rfc-editor.org/info/rfc5424>. Citado na página 17.

GROENEVELDT, R. **Linux detection engineering with Auditd**. Elastic Security Labs, 2024. Disponível em: <https://www.elastic.co/security-labs/linux-detection-engineering-with-auditd>. Citado 3 vezes nas páginas 4, 20 e 21.

GRUBB, S. **audit.rules(7) – Linux man page**. die.net, s.d. Acesso em: 9 ago. 2025. Disponível em: <https://linux.die.net/man/7/audit.rules>. Citado na página 20.

GUR, I. **Mastering auditd in RHEL: Ensuring Security Through Auditing**. Insentra, 2024. Disponível em: <https://www.insentragroup.com/us/insights/geek-speak/modern-workplace/mastering-auditd-in-rhel-ensuring-security-through-auditing/>. Citado 3 vezes nas páginas 20, 21 e 22.

HAMILL, M.; GOSEVA-POPSTOJANOVA, K. Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system. **Software Quality Journal**, Kluwer Academic Publishers, USA, v. 23, n. 2, p. 229–265, 2015. ISSN 0963-9314. Disponível em: <https://doi.org/10.1007/s11219-014-9235-5>. Citado 2 vezes nas páginas 12 e 13.

IEEE. IEEE Standard, **IEEE Standard Glossary of Software Engineering Terminology**. 1990. Disponível em: <https://ieeexplore.ieee.org/document/159342>. Citado na página 12.

KALYANAKRISHNAM, M.; KALBARCZYK, Z. T.; IYER, R. K. Failure data analysis of a lan of windows nt based computers. In: **Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems**. Lausanne, Switzerland: IEEE, 1999. p. 178–187. Disponível em: <https://ieeexplore.ieee.org/document/805094>. Citado na página 27.

MACIEL, V. F. **Desenvolvimento de um Instrumental para Coleta de Dados de Confiabilidade no SistemaOperacional Linux**. Uberlândia, Brasil: Trabalho de conclusão de curso não publicado, 2015. Citado na página 17.

MACIEL, V. F. **Infraestrutura de kernel para coleta de dados de eventos de falha no Linux**. Dissertação (Dissertação (Mestrado em Ciência da Computação)) — Universidade Federal de Uberlândia, 2024. Disponível em: <https://repositorio.ufu.br/handle/123456789/44430>. Citado na página 15.

MACIEL, V. F.; MATIAS, R. Instrumental para coleta de dados de confiabilidade no sistema operacional linux. Uberlândia, Brasil, 2015. Disponível em: <https://www.researchgate.net/publication/284264799_Instrumentation_for_Collecting_Reliability_Data_on_Linux_Operating_Systems>. Citado 6 vezes nas páginas 3, 9, 10, 29, 30 e 31.

MADABHUSHANA, A. B. **Configure Linux system auditing with auditd**. Red Hat, 2021. Disponível em: <https://www.redhat.com/sysadmin/configure-linux-auditing-auditd>. Citado na página 19.

MATIAS, J.; OLIVEIRA, G. D.; ARAUJO, L. B. d. Operating system reliability from the quality of experience viewpoint: An exploratory study. In: ACM. **Proceedings of the 28th Annual ACM Symposium on Applied Computing**. New York, NY, United States, 2013. p. 1644–1649. Disponível em: <https://dl.acm.org/doi/10.1145/2480362.2480669>. Citado na página 27.

MATIAS, R. et al. An empirical exploratory study on operating system reliability. In: ACM. **Proceedings of the 29th Annual ACM Symposium on Applied Computing**. Gyeongju Republic of Korea, 2014. p. 1523–1528. Disponível em: <https://dl.acm.org/doi/10.1145/2554850.2555021>. Citado 2 vezes nas páginas 26 e 27.

MICROSOFT. **Using Reliability Monitor for Troubleshooting**. 2008. Blog *Ask The Performance Team*, Microsoft Tech Community. Atualizado em 2019. Acessado em 4 ago 2025. Disponível em: <https://techcommunity.microsoft.com/blog/askperf/using-reliability-monitor-for-troubleshooting/372962>. Citado na página 17.

_____. **Reliability Analysis Component (RAC)**. 2020. <https://learn.microsoft.com/en-us/windows/win32/win7devguide/compatibility-and-reliability>. Acessado em 04 ago 2025. Citado na página 16.

MUDGAL, P.; ARBAB, B.; KUMAR, S. S. Crasheventllm: Predicting system crashes with large language models. **arXiv preprint arXiv:2407.15716**, 2024. Disponível em: <https://arxiv.org/abs/2407.15716>. Citado 2 vezes nas páginas 15 e 28.

MURPHY, B. Automating software failure reporting. **Queue - System Failures**, v. 2, n. 8, p. 42–48, 2004. Disponível em: <https://dl.acm.org/doi/10.1145/1036474.1036498>. Citado na página 26.

ORLOVSKYI, S. **The Syslog Handbook – How to Collect and Redirect Logs to a Remote Server**. freeCodeCamp Press, 2024. Disponível em: <https://www.freecodecamp.org/news/what-is-syslog-handbook/>. Citado 4 vezes nas páginas 4, 17, 18 e 19.

PHAM, H. **Software Reliability**. Singapore: Springer, 2000. ISBN 9813083840. Citado na página 10.

ROSTEDT, S. **ftrace - Function Tracer**. kernel.org, 2008. Acessado em abril de 2025. Disponível em: <https://www.kernel.org/doc/html/latest/trace/ftrace.html>. Citado 3 vezes nas páginas 4, 24 e 25.

SANTOS, C. A. R.; MATIAS, R.; TRIVEDI, K. S. An empirical exploratory analysis of failure sequences in a commodity operating system. In: IEEE. **2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC)**. 2019. p. 1–8. Disponível em: <https://ieeexplore.ieee.org/document/9046072>. Citado 3 vezes nas páginas 14, 15 e 16.

SCHROEDER, B.; GIBSON, G. A. A large-scale study of failures in high-performance computing systems. **IEEE Transactions on Dependable and Secure Computing**, IEEE, v. 7, n. 4, p. 337–350, 2010. Disponível em: <https://ieeexplore.ieee.org/document/4775906>. Citado 2 vezes nas páginas 14 e 16.

SWIFT, M. M.; BERSHAD, B. N.; LEVY, H. M. Improving the reliability of commodity operating systems. In: ACM. **Proceedings of the 19th ACM Symposium on Operating Systems Principles**. Bolton Landing, NY, USA, 2003. p. 207–222. Disponível em: <https://dl.acm.org/doi/10.1145/945445.945466>. Citado 2 vezes nas páginas 26 e 27.

TANENBAUM, A.; HERDER, J. N.; BOS, H. Can we make operating systems reliable and secure? **Computer**, p. 44–51, 2006. Disponível em: <https://ieeexplore.ieee.org/document/1631939>. Citado na página 26.

The Linux Kernel Organization. **ktime accessors**. kernel.org, 2023. Acessado em abril de 2025. Disponível em: <https://www.kernel.org/doc/html/latest/core-api/timekeeping.html>. Citado na página 25.

YUAN, D.; ZHANG, C. Evaluation strategy for software reliability. **IEEE Transactions on Dependable and Secure Computing**, IEEE, Ningbo, China, 2011. Disponível em: <https://ieeexplore.ieee.org/document/6066612>. Citado na página 26.

ZHANG, R.; XIAO, W.; ZHANG, H.; LIU, Y.; LIN, H.; YANG, M. An empirical study on program failures of deep learning jobs. In: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (ICSE '20), p. 1159–1170. Disponível em: <https://doi.org/10.1145/3377811.3380362>. Citado na página 27.

ZHAO, C.; MA, M.; ZHONG, Z.; ZHANG, S.; TAN, Z.; XIONG, X.; YU, L.; FENG, J.; SUN, Y.; ZHANG, Y. et al. Robust multimodal failure detection for microservice systems. **arXiv preprint arXiv:2305.18985**, 2023. Disponível em: <https://arxiv.org/abs/2305.18985>. Citado na página 28.

Apêndices

APÊNDICE A – Imagens

Figura 5 – *Log syslog* da aplicação Thunderbird

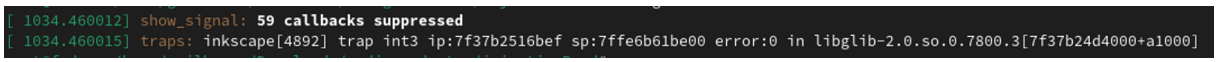

Figura 6 – *Log dmesg* da aplicação Inkscape

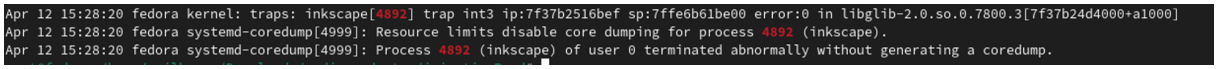

Figura 7 – *Log syslog* da aplicação Inkscape

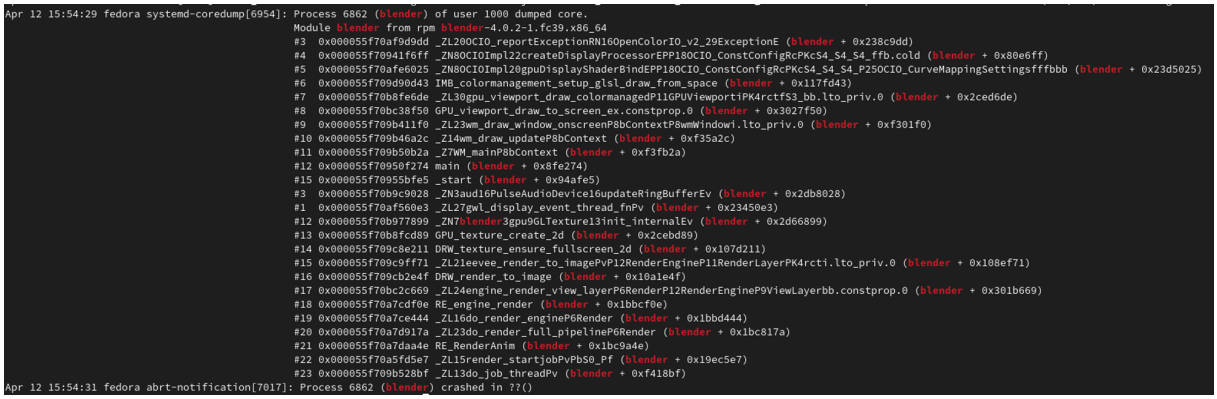

Figura 8 – *Log syslog* da aplicação Blender

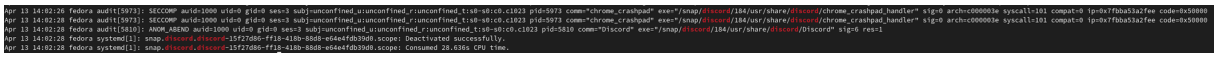

Figura 9 – *Log syslog* da aplicação Discord

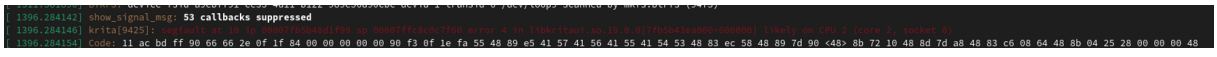

Figura 10 – *Log dmesg* da aplicação Krita

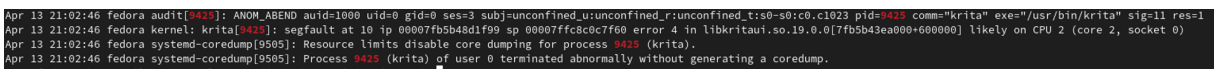

Figura 11 – *Log syslog* da aplicação Krita

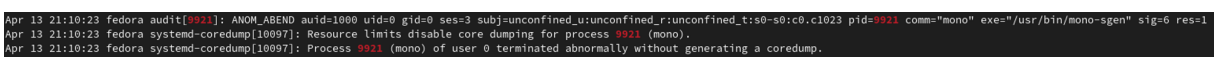

Figura 12 – *Log syslog* da aplicação Pinta

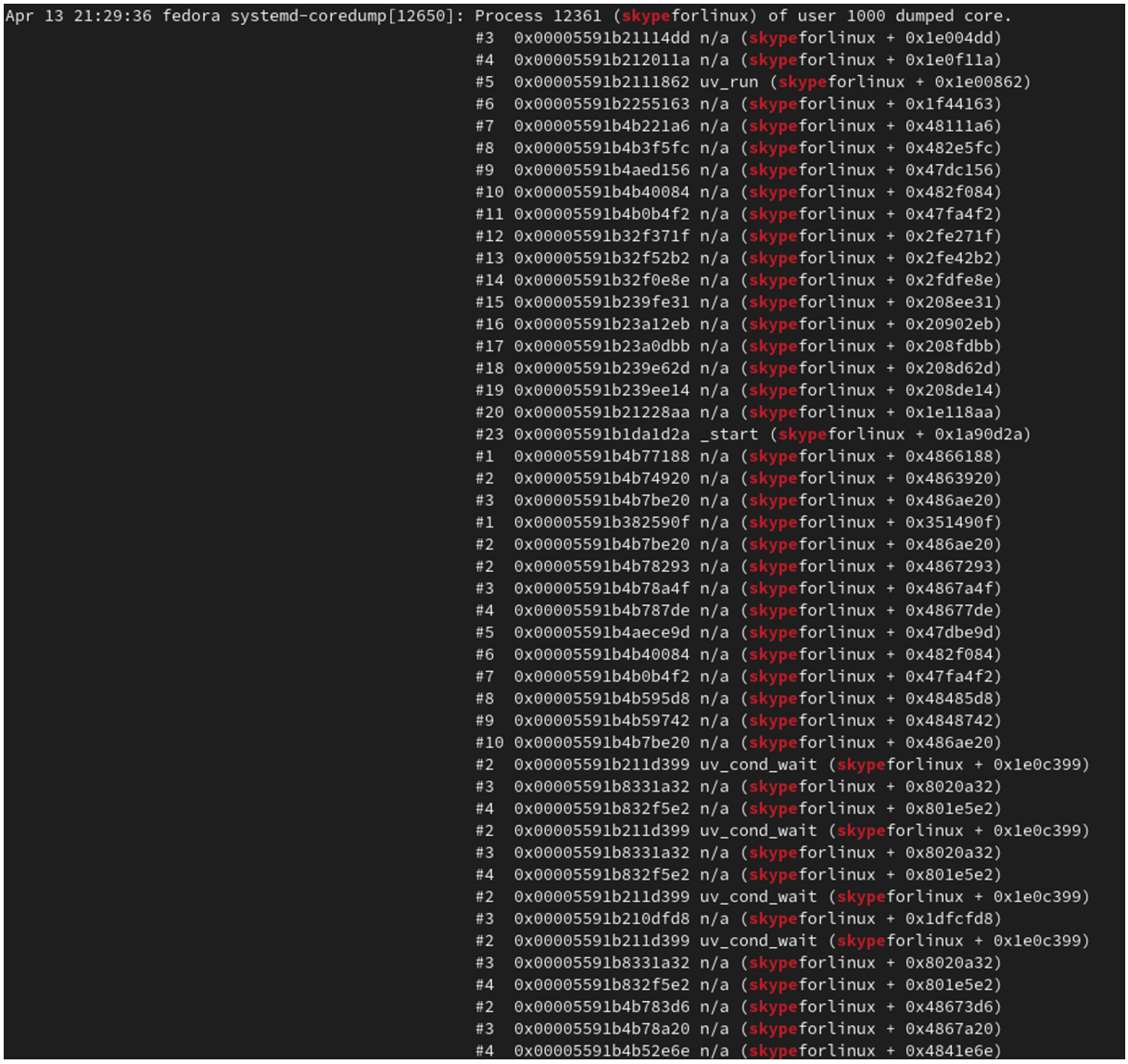

Figura 13 – *Log syslog* da aplicação Skype

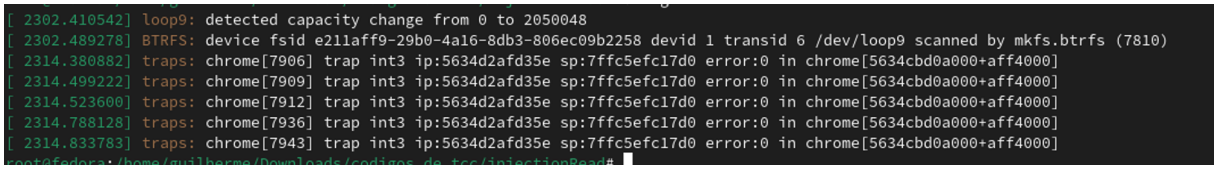

Figura 14 – *Log dmesg* da aplicação Google Chrome

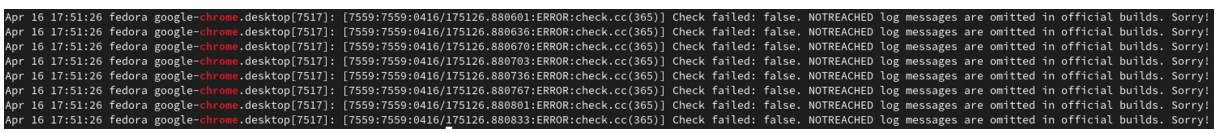

Figura 15 – *Log syslog* da aplicação Google Chrome

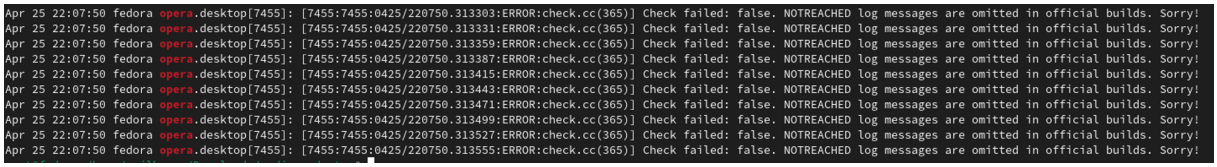

Figura 16 – *Log syslog* da aplicação Opera

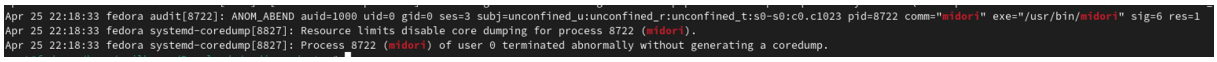

Figura 17 – *Log syslog* da aplicação Midori

Figura 18 – *Log dmesg* da aplicação Kodi

Figura 19 – *Log syslog* da aplicação Kodi

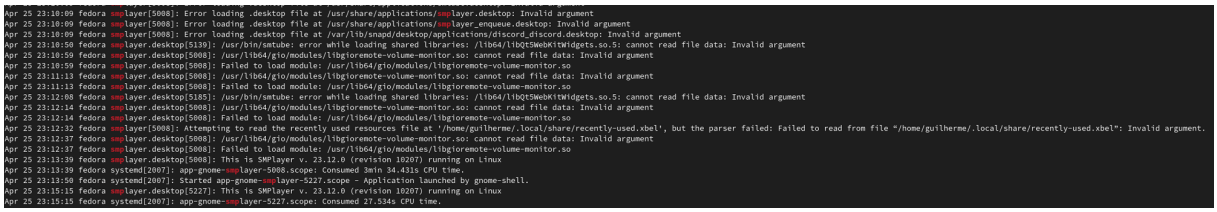

Figura 20 – *Log syslog* da aplicação SMPlayer

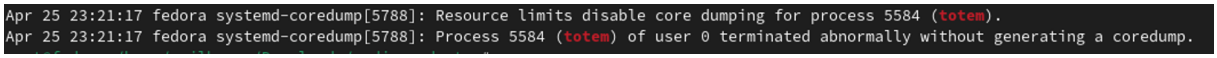

Figura 21 – *Log syslog* da aplicação Totem

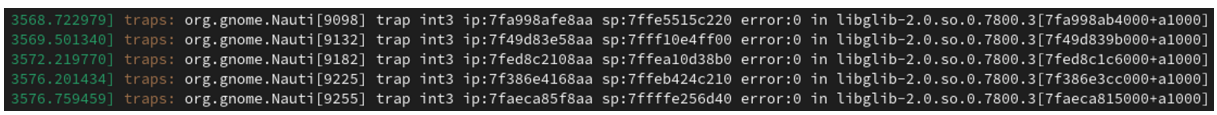

Figura 22 – *Log dmesg* da aplicação GNOME Files (Nautilus)

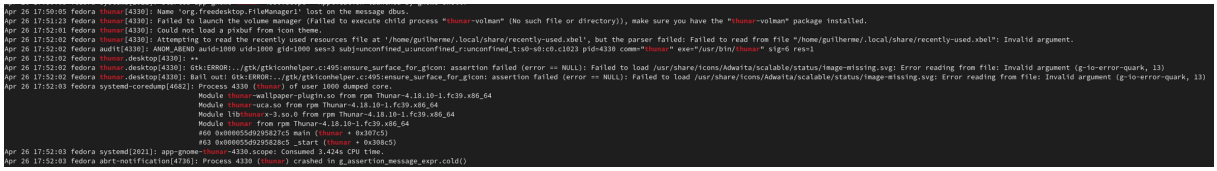

Figura 23 – *Log syslog* da aplicação Thunar

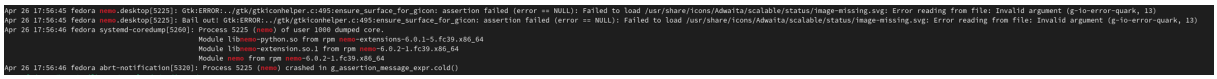

Figura 24 – *Log syslog* da aplicação Nemo

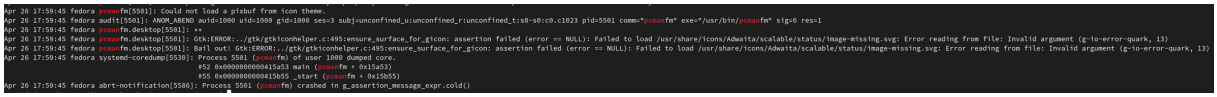

Figura 25 – *Log syslog* da aplicação PCManFM