# SKTRC: Rastreamento Direcionado e Persistente de Eventos do Kernel Linux para Análise de Vulnerabilidades

**Sidney P. J. S. Pinto[1], Ivanilson F. V. Junior[1], Galileu B. Sousa[1], Felipe S. Dantas Silva[1]**

[1]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte (IFRN)
LaTARC Research Lab. Natal, RN – Brasil.
sidney.pedro@academico.ifrn.edu.br
{ivanilson.junior,galileu.batista,felipe.dantas}@ifrn.edu.br

***Abstract.*** *Vulnerabilities in Linux Kernel components may involve internal states that are difficult to observe and transient events that are relevant for security analysis. Although there are well-established tracing mechanisms, these approaches do not always combine targeted collection, contextualization, persistence, and low instrumentation effort. This work introduces the Simple Kernel Trace (SKTRC), a Kernel module aimed at targeted and persistent tracing of internal system events. SKTRC was evaluated in a Proof of Concept (PoC) executed on a native bare metal Linux installation and compared with the native event tracing mechanism. The evaluation used components associated with the Copy Fail vulnerability, which was recently disclosed after remaining present for years in the Linux Kernel. The results indicate that SKTRC makes it possible to collect contextualized logs during the controlled execution of a vulnerability PoC, reduces the instrumentation effort, and facilitates subsequent analysis of security-relevant events.*

## 1. Introdução

Falhas de segurança em componentes do *Kernel* Linux representam um risco crítico, pois ocorrem em uma camada privilegiada do sistema operacional e podem comprometer a estabilidade, a integridade e a confiabilidade de todo o ambiente computacional. Diferentemente de falhas observadas em espaço de usuário (*user space*), vulnerabilidades em espaço de kernel (*kernel space*) podem envolver estados internos de difícil inspeção, caminhos de execução sensíveis e eventos transitórios que nem sempre permanecem disponíveis para análise posterior [Chen et al. 2025].

Esse problema torna-se ainda mais relevante durante a investigação de vulnerabilidades exploráveis (*exploits*). A execução de uma prova de conceito (*Proof of Concept* – PoC) de exploração de vulnerabilidade pode acionar caminhos específicos do *Kernel*, modificar estados internos e produzir eventos cuja observação depende de instrumentação adequada [Alam et al. 2017]. No entanto, a coleta desses eventos é desafiadora, pois os registros podem ser sobrescritos, perdidos ou apresentados sem contexto suficiente para apoiar a análise técnica do comportamento observado [Mahadevan et al. 2026].

O *Kernel* Linux dispõe de mecanismos consolidados de rastreamento, como o *event tracing*, *ftrace* [Faseeha et al. 2025], *Kprobes*, *pstore* e *extended Berkeley Packet Filter* (eBPF). Embora essas soluções sejam úteis em diferentes cenários de depuração, desempenho e observabilidade, nem sempre oferecem, de forma integrada, persistência nativa, identificação contextual, controle fino sobre os eventos registrados e simplicidade

de instrumentação em cenários voltados à análise de vulnerabilidades. Além disso, sua utilização frequentemente exige configuração complexa ou conhecimento aprofundado sobre seus mecanismos internos.

Esses recursos priorizam a coleta genérica de eventos ou análise de desempenho, oferecendo controle limitado sobre a granularidade da instrumentação e sobre a persistência estruturada das informações coletadas, bem como restrições à obtenção de dados armazenados em regiões protegidas de memória. Diante dessas limitações, abordagens dessa natureza não são capazes de atender a cenários de análise que requeiram contabilização precisa de eventos e controle sobre a geração e a persistência de registros, como no processo de *boot* do sistema, em que múltiplos subsistemas podem necessitar de rastreamento simultâneo [Rosset et al. 2026].

Consequentemente, os mecanismos atualmente disponíveis não fornecem, de forma integrada, uma solução simples e autocontida para coleta direcionada, contextualizada e persistentemente estruturada de informações internas do *Kernel*. Em particular, permanece a ausência de um mecanismo leve que permita ao desenvolvedor definir explicitamente quais eventos e estados internos do *Kernel* devem ser observados e registrados de forma persistente.

Essa limitação é particularmente relevante em cenários críticos, como quando informações de diferentes origens são registradas de forma intercalada ou quando registros anteriores são sobrescritos, que demandam análise detalhada do comportamento do sistema durante as fases iniciais de execução, como em investigações de *boot* e de *crashes* [Mazza et al. 2025].

Diante dessa lacuna, este trabalho propõe o *Simple Kernel Trace* (SKTRC), um módulo de *Kernel* projetado para apoiar a análise de vulnerabilidades por meio do rastreamento direcionado e persistente de eventos internos do sistema, com escopo de rastreamento definido pelo desenvolvedor. O SKTRC permite instrumentar pontos específicos do *Kernel*, associar registros a contextos definidos pelo desenvolvedor, armazenar temporariamente os eventos em memória e persistir os dados quando o sistema de arquivos se torna disponível, facilitando a análise posterior do seu comportamento.

O SKTRC foi desenvolvido em forma de módulo de *Kernel* (*Kmod*), o que lhe permite receber informações dos demais subsistemas. Essas informações começam a ser obtidas a partir do momento de sua instanciação, a qual ocorre após o carregamento do *Kernel* na memória, ainda durante o *boot*. Os dados coletados são inicialmente armazenados em um *buffer* interno e, posteriormente, persistidos em um arquivo definido pelo desenvolvedor, permitindo sua análise posterior em ambiente externo ao contexto sob investigação.

A avaliação do SKTRC foi realizada por meio de uma PoC baseada em componente do *Kernel* Linux associado à vulnerabilidade *Copy Fail*[1], rastreada como CVE-2026-31431. Nesse cenário, o SKTRC é utilizado para rastrear eventos acionados durante a execução controlada da PoC de vulnerabilidade, permitindo observar o caminho de execução instrumentado e registrar informações relevantes para análise posterior. A avaliação compara o uso do SKTRC com o sistema nativo de *event tracing*, considerando a expressividade dos registros, a identificação contextual, a persistência das informações

[1] `https://nvd.nist.gov/vuln/detail/CVE-2026-31431`

e o esforço de instrumentação.

As principais contribuições deste trabalho, materializadas no SKTRC, são:

1. Propor o SKTRC como mecanismo leve de rastreamento direcionado e persistente para apoio à análise de vulnerabilidades no *Kernel* Linux.
2. Permitir a associação de registros a contextos específicos de execução, favorecendo a interpretação de eventos acionados durante a execução controlada de PoCs de vulnerabilidade.
3. Viabilizar a persistência de registros para análise posterior, mesmo quando a coleta ocorre em momentos nos quais a escrita imediata em disco ainda não está disponível.
4. Demonstrar, por meio de uma PoC em ambiente nativo sobre *hardware* físico, a viabilidade do SKTRC como ferramenta auxiliar para investigação de eventos internos relevantes à análise de vulnerabilidades no *Kernel*.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta um comparativo entre o SKTRC e soluções similares, internas e externas ao *Kernel*. A Seção 3 descreve a arquitetura e o funcionamento do módulo proposto. Na Seção 4, uma prova de conceito demonstra os efeitos práticos do uso do SKTRC e os resultados obtidos, além de discutir restrições operacionais e desafios de pesquisa na área. Ao final, a Seção 5 sintetiza as contribuições do trabalho e apresenta perspectivas para pesquisas futuras.

## 2. Trabalhos Relacionados

Esta seção discute mecanismos de instrumentação, rastreamento e observabilidade aplicáveis à análise de falhas de segurança no *Kernel* Linux. O objetivo é comparar abordagens capazes de apoiar a coleta de eventos internos do sistema, considerando sua utilidade para investigação de vulnerabilidades, preservação de registros e análise posterior de comportamentos críticos.

A literatura relacionada inclui tanto mecanismos nativos do *Kernel* quanto ferramentas externas de instrumentação. Em [Zhang et al. 2019], os autores apresentam um mecanismo de coleta de eventos durante o processo de *boot*, cenário relevante para a observação de falhas que ocorrem em fases iniciais da execução. Já em [Khan and Ezzati-Jivan 2023], os autores propõem uma metodologia para otimização de *traces* conforme necessidades específicas de análise. Por sua vez, [Esteves et al. 2023] apresenta o DIO, uma ferramenta baseada em eBPF voltada ao rastreamento de operações de entrada e saída.

O próprio *Kernel* Linux oferece mecanismos consolidados de instrumentação, como o *tracefs* [Aranya 2004], *event tracing* [Ts'o et al. 2009], *ftrace* [Rostedt 2008], *Kprobes* [Billimoria 2022], *pstore* [Luck 2011] e eBPF [Song and Li 2024]. Esses mecanismos são amplamente utilizados para depuração, análise de desempenho e observabilidade. No entanto, quando considerados sob a perspectiva de análise de segurança, ainda apresentam limitações quanto à persistência dos registros, à identificação contextual dos eventos e ao controle direcionado sobre os pontos instrumentados.

Para comparar essas abordagens, foram definidos critérios associados à utilidade dos mecanismos em cenários de investigação de falhas e vulnerabilidades no *Kernel*.

- **C1**: Persistência dos registros para análise posterior, visto que a preservação dos registros é necessária para viabilizar análises posteriores fora do contexto imediato de execução.
- **C2**: Identificação contextual do evento registrado, uma vez que a contextualização dos eventos é fundamental para localizar com precisão sua origem em funções, subsistemas ou módulos.
- **C3**: Suporte à coleta de eventos durante o processo de *boot*, já que falhas críticas e eventos de difícil reprodução podem ocorrer ainda nas fases iniciais do sistema.
- **C4**: Possibilidade de customização das mensagens de rastreamento, visto que a flexibilidade na definição dos registros amplia a adequação do mecanismo a diferentes necessidades de investigação e análise técnica.
- **C5**: Disponibilidade do código-fonte e documentação da solução, por serem elementos indispensáveis ao uso correto, à reprodutibilidade e à avaliação técnica da abordagem.

A partir desses critérios, a Tabela 1 compara o SKTRC com os mecanismos analisados quanto à sua adequação a cenários de investigação de segurança no *Kernel* Linux.

**Tabela 1. Comparativo entre mecanismos de rastreamento quanto à adequação para análise de vulnerabilidades no *Kernel* Linux.**

| Mecanismos | C1 | C2 | C3 | C4 | C5 | Versões do *Kernel* Linux testadas |
|---|---|---|---|---|---|---|
| *tracefs* [Aranya 2004] | ✗ | ✗ | ✓ | ✓ | ✓ | 2.4.20, 6.12 |
| *ftrace* [Rostedt 2008] | ✗ | ✓ | ○ | ✓ | ✓ | 2.6.28-rc2, 3.10, 4.13 |
| *event tracing* [Ts'o et al. 2009] | ✗ | ✓ | ✓ | ✓ | ✓ | 2.6.34-rc3 |
| *pstore* [Luck 2011] | ✗ | ✓ | ✓ | ✗ | ✓ | ○ |
| *SystemTap* [O'Brien 2017] | ✗ | ✓ | ○ | ✓ | ✓ | 2.6.16.60 |
| [Zhang et al. 2019] | ✓ | ✗ | ✓ | ✗ | ✗ | 4.9.67, 4.9.59 |
| *Kprobes* [Billimoria 2022] | ✗ | ✓ | ○ | ✓ | ✓ | 2.6.9-rc2 |
| [Khan and Ezzati-Jivan 2023] | ✗ | ✓ | ✗ | ○ | ✓ | ○ |
| DIO [Esteves et al. 2023] | ✓ | ✓ | ✗ | ✗ | ✓ | 5.4.0 |
| eBPF [Song and Li 2024] | ✗ | ✓ | ✗ | ✓ | ✓ | 3.15, 3.16, 3.18 |
| **SKTRC** | ✓ | ✓ | ✓ | ✓ | ✓ | 3.18.71, 6.12.73 |

**Legenda:** ✓: atende; ✗: não atende; ○: não aplicável ou não informado.

Observa-se que as abordagens existentes atendem apenas parcialmente aos critérios definidos e não combinam, em um mesmo mecanismo, rastreamento direcionado, contextualização dos registros, persistência e flexibilidade de customização. Essa limitação é particularmente relevante na análise de vulnerabilidades, em que a execução controlada de *exploits* pode acionar eventos internos transitórios, de difícil reprodução e sujeitos à perda [Yang et al. 2025]. Nesse contexto, a próxima seção apresenta o SKTRC, um módulo de *Kernel* projetado para apoiar a investigação desses eventos por meio de coleta direcionada, contextualizada e persistente.

## 3. SKTRC

Esta seção apresenta a proposta do SKTRC em dois níveis complementares. Inicialmente, são descritas sua arquitetura e os elementos que sustentam a coleta contextualizada e a persistência dos registros. Em seguida, é detalhado seu funcionamento, desde a geração dos eventos até o armazenamento temporário e a persistência das informações coletadas, com foco em seu uso como mecanismo auxiliar para análise de vulnerabilidades no *Kernel* Linux.

### 3.1. Arquitetura do SKTRC

O SKTRC foi concebido para viabilizar o rastreamento direcionado e persistente de eventos em módulos e subsistemas do *Kernel* Linux. Sua arquitetura combina instrumentação seletiva, organização explícita de contextos, armazenamento temporário e persistência dos registros para análise posterior.

A Figura 1 apresenta uma visão geral da arquitetura de *software* do SKTRC, posicionando o módulo em relação aos subsistemas instrumentados, às interfaces internas do *Kernel*, ao sistema de arquivos e à camada de persistência. Os elementos associados ao SKTRC são destacados em relação aos componentes já existentes no sistema operacional, evidenciando sua contribuição no fluxo de coleta e tratamento dos eventos registrados. Nessa arquitetura, o SKTRC recebe eventos gerados por funções previamente selecionadas e os encaminha para um *buffer* interno ou para o arquivo de saída, conforme a disponibilidade do sistema de arquivos.

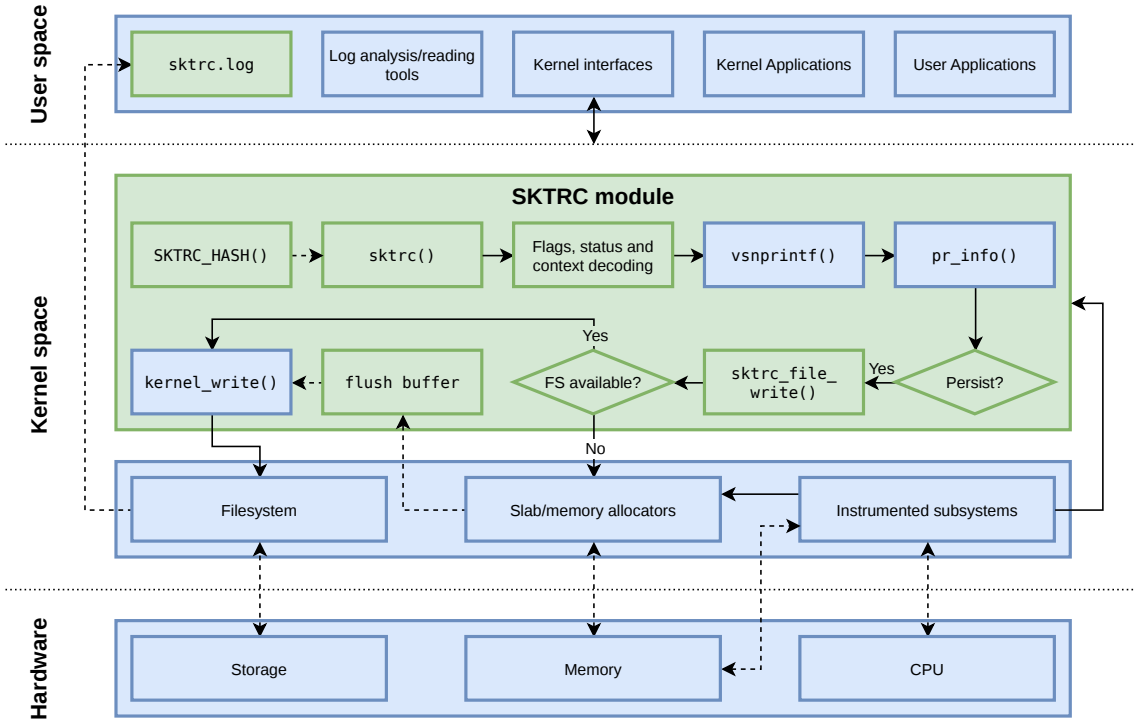



**Figura 1. Arquitetura do SKTRC e seu posicionamento em relação a subsistemas, interfaces do Kernel Linux e mecanismos de persistência.**

Conforme pode ser observado na Figura 1, os elementos centrais dessa arquitetura são a função `sktrc()`, o `flush buffer` para armazenamento temporário de registros, a função `sktrc_file_write()` e o arquivo de persistência `sktrc.log`. Em conjunto, esses elementos implementam o fluxo de coleta, armazenamento temporário e persistência dos *traces* gerados pelos módulos instrumentados.

A integração com outros módulos ocorre por meio da inclusão de cabeçalho correspondente nos trechos instrumentados do código-fonte do *Kernel* e pela adição da dependência nos arquivos `Kconfig`[2]. A partir disso, os módulos podem registrar eventos

[2] `https://docs.kernel.org/kbuild/kconfig-language.html`

por meio da função `sktrc()`, informando o contexto do evento e, opcionalmente, uma mensagem formatada. Após essa integração, o *Kernel* pode ser recompilado integral ou parcialmente, conforme o escopo do subsistema instrumentado.

Os contextos utilizados pelo SKTRC são compostos pelo identificador do subsistema e pela funcionalidade associada, sendo definidos diretamente no arquivo `sktrc.c`. Detalhes adicionais de implementação são apresentados na Figura 5, localizada no Apêndice A, que reúne a estrutura de arquivos e diretórios, a organização dos contextos de rastreamento, as macros utilizadas na instrumentação e o caminho de persistência dos registros. O código-fonte completo está disponível no repositório oficial do SKTRC[3].

### 3.2. Fluxo de funcionamento do SKTRC

A Figura 2 sintetiza o ciclo de vida de um registro no SKTRC, desde sua geração em um ponto instrumentado do *Kernel* até sua persistência em arquivo. O fluxo evidencia a separação entre captura do evento, armazenamento temporário em memória e gravação definitiva quando o sistema de arquivos está disponível.

Conforme ilustrado na Figura 2, o funcionamento do SKTRC tem início nos pontos do código previamente instrumentados, onde a função `sktrc()` é chamada com um identificador de contexto em formato *hash* e, opcionalmente, uma *string* de formatação compatível com convenções de saída textual formatada amplamente empregadas em interfaces como `printk()` e `printf()`.

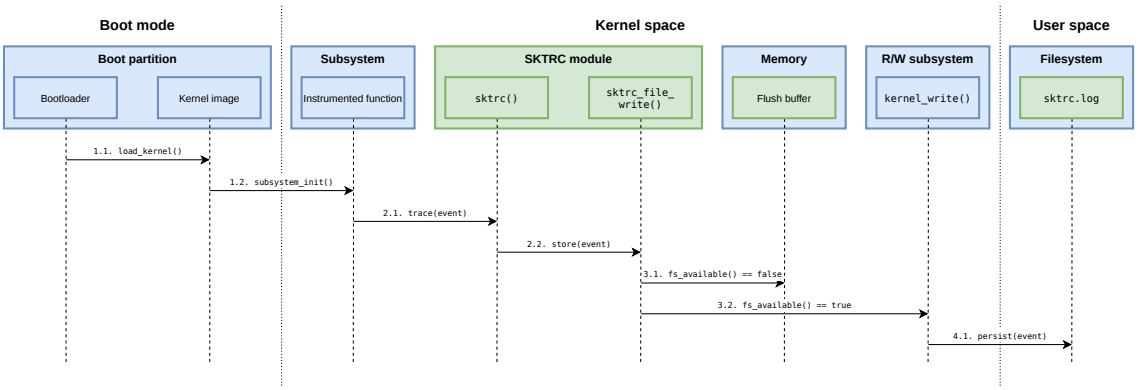



**Figura 2. Fluxo de funcionamento do SKTRC, incluindo geração do trace, armazenamento temporário em *buffer* e persistência em disco.**

O processo começa com o carregamento do *Kernel* na etapa 1.1. Em seguida, o módulo instrumentado é inicializado e passa a emitir registros por meio da função `sktrc()` na etapa 1.2. A partir desse ponto, um evento de *trace* é gerado em um trecho previamente instrumentado do código, conforme indicado na etapa 2.1. O identificador relacionado ao evento pode ser obtido por meio da macro `SKTRC_HASH`, apresentada na Figura 5(c), que codifica em quatro *bytes* informações sobre persistência, estado e contexto do evento. Com base nesses parâmetros, a função `sktrc()` interpreta o *trace* recebido e define como o registro será tratado. Na etapa 2.2, as informações do evento são preparadas para armazenamento temporário ou persistência imediata.

[3] `https://github.com/latarc/sktrc`

Em seguida, o SKTRC verifica se o sistema de arquivos já está disponível. Se essa condição ainda não for satisfeita, como representado na etapa 3.1, o registro é armazenado em memória para persistência posterior. Caso o sistema de arquivos esteja disponível, como mostrado na etapa 3.2, o conteúdo armazenado em memória, quando existente, é encaminhado para persistência, seguido das informações do evento recém-gerado.

Se não houver novos eventos após a montagem do sistema de arquivos, o módulo auxiliar `sktrc_flush` deve ser carregado manualmente para forçar a persistência dos dados ainda mantidos em memória. Por fim, os registros são gravados no arquivo definido pela constante `SKTRC_LOG`, conforme indicado na etapa 4.1 e detalhado na Figura 5(d).

Esse encadeamento demonstra que o SKTRC separa a captura do *trace* de sua gravação definitiva em disco, permitindo preservar eventos mesmo quando o sistema de arquivos ainda não está disponível. Essa separação é particularmente relevante em cenários de *boot* e em contextos nos quais a escrita imediata em disco ainda não pode ser realizada.

O SKTRC pode ser utilizado em diferentes subsistemas do *Kernel*, exceto em trechos de código dos quais depende diretamente, como funções e componentes utilizados pelo próprio mecanismo de persistência. Essa restrição preserva a consistência do módulo sem comprometer sua aplicabilidade em outros pontos do sistema. Como sua interface de uso se concentra na função `sktrc()`, o desenvolvedor pode incorporar o mecanismo de forma seletiva a diferentes pontos do *Kernel*, com granularidade por ele definida.

Além da flexibilidade de instrumentação, o SKTRC também permite o ajuste seletivo do rastreamento. Esse controle pode ocorrer ao nível de subsistema, simplesmente por meio da substituição do cabeçalho de inclusão, ou ao nível de evento, bastando a substituição da função `sktrc()` por `sktrcn()`, sua variante não operacional. Em termos práticos, a inclusão de `trace/sktrc.h` pode ser substituída por `trace/sktrcn.h` para desabilitar o rastreamento de um subsistema inteiro, enquanto chamadas pontuais a `sktrc()` podem ser substituídas por `sktrcn()` para suprimir eventos específicos.

## 4. Prova de Conceito

Esta seção apresenta a prova de conceito (PoC) elaborada para avaliar, em cenário controlado de análise de vulnerabilidade, o uso do SKTRC em comparação com o sistema de *event tracing*. O objetivo não é esgotar as possibilidades de uso do módulo, mas evidenciar, por meio de um experimento reprodutível, diferenças de implantação, expressividade dos registros e esforço de instrumentação.

### 4.1. Cenário experimental

Os experimentos foram realizados em um computador *desktop* com CPU AMD Ryzen 7 5700, 32 GB de memória RAM, executando o sistema operacional Debian GNU/Linux 13.3 com o *Kernel* versão 6.12.73, sob o sistema de arquivos `ext4` e com o pacote `build-essential`, em sua versão 12.12, instalado.

Dentre as soluções de rastreamento discutidas na Seção 2, o *event tracing* foi selecionado para esta PoC por representar o mecanismo nativo cujas capacidades mais se aproximam daquelas oferecidas pelo SKTRC. Para registrar um evento com *event tracing*, cada rastreio deve ser explicitamente definido por meio da macro `TRACE_EVENT`

em arquivos de cabeçalho do *Kernel*, podendo inclusive demandar alterações no arquivo `Makefile` do subsistema. Já no SKTRC, basta definir os contextos no arquivo `sktrc.c` e inserir as chamadas à função `sktrc()` nos pontos de interesse.

Com base nessa diferença de integração, foi definido um cenário experimental com escopo delimitado. Para favorecer a compreensão da lógica executada e permitir a análise das diferenças entre as abordagens, a PoC foi construída a partir da execução controlada de um cenário associado à vulnerabilidade rastreada como CVE-2026-31431, também identificada como *Copy Fail*.

A comparação considera a forma de instrumentação, a estrutura dos registros gerados e o esforço necessário para implantação. Essa escolha também reforça o caráter didático e reprodutível do experimento, uma vez que o código-fonte instrumentado está disponível e pode ser diretamente reutilizado. Durante a execução controlada da PoC associada à vulnerabilidade[4], regiões de *page cache* da memória são modificadas com fragmentos de uma carga controlada, enquanto os eventos associados ao caminho de execução são rastreados.

Nesse contexto, o subsistema vulnerável foi instrumentado no *Kernel* com ambas as abordagens avaliadas. As funções instrumentadas na PoC, associadas ao caminho de execução analisado, foram `crypto_authenc_esn_decrypt()`, `crypto_authenc_esn_module_init()` e `aead_recvmsg()`. A implementação da PoC está disponível em repositório específico do SKTRC[5]. Os principais trechos da instrumentação são apresentados na Figura 6, localizada no Apêndice B. No caso do *event tracing*, o arquivo `poc_copy_fail.h` contém a definição dos eventos em arquivo de cabeçalho, conforme mostra a Figura 6(c). No caso do SKTRC, o arquivo `sktrc.c` apresenta a definição dos contextos de rastreamento no módulo, conforme mostra a Figura 6(d).

Essa comparação já evidencia que o esforço de instrumentação cresce de forma distinta entre as abordagens. No caso do *event tracing*, a inclusão de novos rastreamentos exige a definição explícita de eventos por meio de múltiplas macros e estruturas auxiliares. No SKTRC, por outro lado, a expansão do rastreamento depende essencialmente da declaração dos contextos e da inserção das chamadas à função `sktrc()`.

### 4.2. Resultados

A partir do cenário proposto, a PoC permitiu comparar a estrutura, a contextualização e a persistência dos registros produzidos pelos dois mecanismos de rastreamento. As saídas obtidas estão nos arquivos `/sys/kernel/tracing/trace` e `/var/log/sktrc.log`, e seus resultados são apresentados na Figura 3.

A Figura 3(a) mostra a saída produzida pelo *event tracing*, na qual a identificação contextual depende principalmente do nome do evento registrado na coluna `FUNCTION`, acompanhado da mensagem formatada correspondente. Já a Figura 3(b) apresenta a saída gerada pelo SKTRC, organizada em termos de estado, contexto e mensagem customizada, o que demonstra maior controle de customização sem exigir a definição de novos eventos.

Essa comparação evidencia que o SKTRC fornece registros mais diretamente con-

[4] `https://copy.fail/#exploit`

[5] `https://github.com/latarc/sktrc_poc_kernel`

```
# tracer: nop
# ...
#           TASK-PID     CPU#  |||||  TIMESTAMP
↪  FUNCTION
#              | |         |   |||||     |         |
       swapper/0-1       [000] .....     0.138974:
       ↪  loaded: Instrumented module loaded for
       ↪  controlled analysis.
         python3-1207    [014] .....    19.597922:
         ↪  receiving: Receiving payload fragment...
         python3-1207    [014] .....    19.597961:
         ↪  overwriting: Overwriting data at address
         ↪  ffff8b1e055a7920 (0x464c457f41414141)
         python3-1207    [014] .....    19.598392:
         ↪  receiving: Receiving payload fragment...
         python3-1207    [014] .....    19.598406:
         ↪  overwriting: Overwriting data at address
         ↪  ffff8b1e055a79a0 (0x0001010241414141)
# ...
         python3-1207    [014] .....    19.601790:
         ↪  receiving: Receiving payload fragment...
         python3-1207    [014] .....    19.601804:
         ↪  overwriting: Overwriting data at address
         ↪  ffff8b1e055a8c20 (0x732f6e6941414141)
         python3-1207    [014] .....    19.601884:
         ↪  receiving: Receiving payload fragment...
         python3-1207    [014] .....    19.601899:
         ↪  overwriting: Overwriting data at address
         ↪  ffff8b1e055a8ca0 (0x0000006841414141)
```

**(a) Saída produzida pelo *event tracing*.**

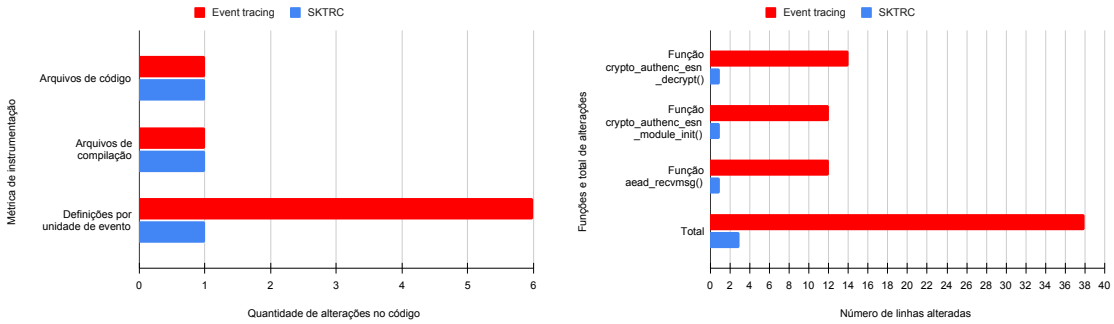
```
[+] [authencesn:crypto_authenc_esn_module_init]:
↪  Instrumented module loaded for controlled analysis.
[+] [sktrc:sktrc_init]: SKTRC loaded!
[*] [algif_aead:aead_recvmsg]: Receiving payload
↪  fragment...
[*] [authencesn:crypto_authenc_esn_decrypt]:
↪  Overwriting data at address ffff8b1e055a7920
↪  (0x464c457f41414141)
[*] [algif_aead:aead_recvmsg]: Receiving payload
↪  fragment...
[*] [authencesn:crypto_authenc_esn_decrypt]:
↪  Overwriting data at address ffff8b1e055a79a0
↪  (0x0001010241414141)
# ...
[*] [algif_aead:aead_recvmsg]: Receiving payload
↪  fragment...
[*] [authencesn:crypto_authenc_esn_decrypt]:
↪  Overwriting data at address ffff8b1e055a8c20
↪  (0x732f6e6941414141)
[*] [algif_aead:aead_recvmsg]: Receiving payload
↪  fragment...
[*] [authencesn:crypto_authenc_esn_decrypt]:
↪  Overwriting data at address ffff8b1e055a8ca0
↪  (0x0000006841414141)
[+] [sktrc:sktrc_flush]: SKTRC flushed!
```

**(b) Saída produzida pelo SKTRC.**

**Figura 3. Saídas de rastreamento obtidas na PoC com o *event tracing* e o SKTRC.**

textualizados, uma vez que a identificação do evento incorpora explicitamente o subsistema e a funcionalidade rastreados. Além disso, o módulo oferece maior flexibilidade na formulação das mensagens, sem exigir a criação de novas estruturas de evento para cada novo rastreamento.

A Figura 4 sintetiza quantitativamente o esforço necessário para instrumentar a PoC nas duas abordagens. A Figura 4(a) compara o esforço estrutural da instrumentação, considerando os arquivos envolvidos e a quantidade de definições ou chamadas requeridas para cada evento instrumentado. A Figura 4(b) mostra o volume de alterações no código-fonte necessário para instrumentar as funções avaliadas. Em ambos os casos, há redução expressiva do esforço de implantação com o uso do SKTRC em relação ao *event tracing*.




**(a) Esforço estrutural.**

**(b) Alterações no código-fonte.**

**Figura 4. Esforço de instrumentação na PoC.**

A Figura 4(a) mostra que o *event tracing* requer 6 definições para cada evento instrumentado, enquanto o SKTRC requer 1 chamada de rastreamento, o que corresponde a 16,67% do total exigido pela primeira abordagem. Esse resultado é coerente com a

Figura 4(b), na qual o SKTRC apresenta média de 1 linha alterada, ao passo que o *event tracing* apresenta média de 12,67 linhas alteradas, indicando que o SKTRC exige apenas 7,89% do total de linhas necessárias para instrumentação com *event tracing*.

Além disso, em cenários nos quais os rastreamentos ocorrem antes da disponibilidade plena do sistema de arquivos, ambos os mecanismos podem exigir procedimentos adicionais para preservar os registros coletados. No caso do SKTRC, a persistência pode depender do carregamento posterior do módulo auxiliar `sktrc_flush`. Na PoC, esse mecanismo foi utilizado para garantir a preservação dos registros coletados durante o fluxo analisado. De modo semelhante, o *event tracing* exigiu a adição da opção `trace_event=poc_copy_fail` às configurações de inicialização[6] para evitar que seus rastreamentos fossem sobrescritos durante esse processo, pois o *ring buffer* reservado para os mecanismos de *trace* é compartilhado com outros eventos produzidos pelos módulos do *Kernel*.

### 4.3. Limitações e desafios de pesquisa

Em cenários com grande volume de eventos, especialmente durante fases iniciais do *boot* ou durante a execução controlada de PoCs de vulnerabilidades, os mecanismos avaliados estão sujeitos a limitações relacionadas ao armazenamento temporário e à preservação dos registros. No *event tracing*, essa condição pode levar à rápida ocupação do *ring buffer* e à sobrescrita de eventos anteriores. No SKTRC, a limitação ocorre quando há geração intensiva de eventos antes da disponibilidade da persistência em disco. Esses aspectos reforçam a necessidade de pesquisas futuras sobre retenção, priorização e persistência antecipada de registros, especialmente em cenários de segurança com alta intensidade de eventos.

## 5. Conclusão

Este trabalho apresentou o SKTRC, um módulo voltado ao rastreamento direcionado e persistente de eventos internos relevantes à análise de vulnerabilidades do *Kernel* Linux. A proposta combina instrumentação seletiva, contextualização dos registros e persistência das informações coletadas, permitindo apoiar a investigação posterior de eventos acionados em cenários controlados de segurança.

A solução foi avaliada por meio de uma prova de conceito baseada em componentes associados à vulnerabilidade *Copy Fail* (CVE-2026-31431), em comparação com o sistema nativo de *event tracing*. Os resultados indicaram que o SKTRC produz registros mais diretamente contextualizados e exige menor esforço de instrumentação, tanto em termos estruturais quanto em volume de alterações no código-fonte. Esses resultados reforçam sua viabilidade como alternativa leve e autocontida para apoiar análises de segurança em nível de *Kernel*.

Como trabalhos futuros, pretende-se ampliar a avaliação experimental do SKTRC para diferentes classes de vulnerabilidades e subsistemas do *Kernel*, uma vez que a PoC com o *Copy Fail* representa um cenário inicial de validação da proposta. Também serão aprimorados mecanismos de retenção, priorização e proteção dos registros coletados, com foco em cenários de análise de segurança com alta intensidade de eventos.

[6] `https://www.kernel.org/doc/html/latest/trace/events.html#boot-option`

## Referências

Alam, D., Zaman, M., Farah, T., Rahman, R., and Hosain, M. S. (2017). Study of the Dirty Copy on Write, a Linux Kernel memory allocation vulnerability. In *2017 International Conference on Consumer Electronics and Devices (ICCED)*, pages 40–45.

Aranya, A. (2004). Tracefs: A File System to Trace Them All. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA. USENIX Association.

Billimoria, K. N. (2022). *Linux Kernel Debugging: Leverage proven tools and advanced techniques to effectively debug Linux kernels and kernel modules*. Packt Publishing Ltd.

Chen, Z., Li, Z., Song, Z., Shi, Z., and Sun, L. (2025). Exp-Arch: A Novel LLM-Powered Approach for Facilitating Exploit Primitive Assessment in the Linux Kernel. In *2025 IEEE 31st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 01–10.

Esteves, T., Macedo, R., Oliveira, R., and Paulo, J. (2023). Diagnosing applications' I/O behavior through system call observability. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 1–8.

Faseeha, U., Jamil Syed, H., Samad, F., Zehra, S., and Ahmed, H. (2025). Observability in Microservices: An In-Depth Exploration of Frameworks, Challenges, and Deployment Paradigms. *IEEE Access*, 13:72011–72039.

Khan, M. A. and Ezzati-Jivan, N. (2023). Multi-level Adaptive Execution Tracing for Efficient Performance Analysis. In *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 104–109.

Luck, T. (2011). *Pstore: Generic interface to platform dependent persistent storage – The Linux Kernel Documentation*. The Linux Kernel Organization. Disponível em: `https://www.kernel.org/doc/Documentation/ABI/testing/pstore`. Acesso em: 21 Mar. 2026.

Mahadevan, S. V., Takano, Y., and Miyaji, A. (2026). Enhancing Information Flow Control in eBPF Programs via Taint Tracking Mechanisms. Los Alamitos, CA, USA. IEEE Computer Society.

Mazza, S., Calabrò, F., Valla, F., Amer, A., and Facchinetti, T. (2025). Evaluation of the boot time in a Linux automotive environment with security constraints. In *2025 IEEE 30th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4.

O'Brien, D. (2017). Teaching Operating Systems Concepts with SystemTap. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, page 335–340, New York, NY, USA. Association for Computing Machinery.

Rosset, T., Wisniewski, L., and Scanzio, S. (2026). A Survey on Platform-Level Data Quality: From Visibility to Controllability. *IEEE Access*, 14:48262–48276.

Rostedt, S. (2008). *ftrace - Function Tracer – The Linux Kernel Documentation*. Disponível em: `https://docs.kernel.org/trace/ftrace.html`. Acesso em: 2 Mar. 2026.

Song, L. and Li, J. (2024). eBPF: Pioneering Kernel Programmability and System Observability - Past, Present, and Future Insights. In *2024 3rd International Conference on Artificial Intelligence and Computer Information Technology (AICIT)*, pages 1–10.

Ts'o, T., Zefan, L., and Zanussi, T. (2009). *Event Tracing – The Linux Kernel Documentation*. The Linux Kernel Organization. Disponível em: `https://docs.kernel.org/trace/events.html`. Acesso em: 21 Mar. 2026.

Yang, Z., Solanki, S., Rixner, S., and Dautenhahn, N. (2025). Hi-Res: Precise Exploit Detection Using Object-Granular Memory Monitoring. In *2025 IEEE Security and Privacy Workshops (SPW)*, pages 79–90.

Zhang, B., Yang, K., Wang, L., Tan, Y.-a., and Hu, S. (2019). Tracing Android Kernel Codes at Early Stage without Extra Hardware Components. In *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*, pages 210–216.

# Apêndice A. Elementos de implementação do SKTRC

```
sktrc
├── COPYING
├── README.org
└── src
    ├── Makefile
    ├── sktrc.c
    ├── sktrc_debug.c
    ├── sktrc_flush.c
    ├── sktrc.h
    └── sktrcn.h
```

**(a) Estrutura do diretório-fonte.**

```
static const char SKTRC_CTX[0x10][0x100][0x2a] = {
  {               // 0x0000  XXXX
    // ...
  },
  {               // 0x0001  XXXX
    "qseecom",           // 0000
    "qseecom_init",      // 0001
    "qseecom_exit",      // 0002
    // ...
    "send_raw_scm"       // 000f
  }
};
```

**(b) Contextos de rastreamento.**

```
// ...
#define SKTRC_HASH(flags, sign, mod, func) \
  (((flags & 0x3) << 30) + ((sign & 0x3) << 28) + \
   ((mod & 0xfff) << 16) + (func & 0xffff))

extern u_int8_t sktrc(u_int32_t hash, const char *fmt,
↪  ...);
#define sktrcn(hash, fmt, ...) if (0);
// ...
```

**(c) Macros de implementação.**

```
static const char *const SKTRC_LOG =
  "/data/local/tmp/sktrc.log";
```

**(d) Caminho da persistência.**

**Figura 5. Elementos da implementação do SKTRC: estrutura do diretório-fonte, contextos de rastreamento, macros de instrumentação e persistência.**

# Apêndice B. Detalhes da instrumentação adotada na PoC

```
// ...
static int crypto_authenc_esn_decrypt(struct
↪  aead_request *req)
// ...
  /* Move high-order bits of sequence number to the
  ↪  end. */
  scatterwalk_map_and_copy(tmp, dst, 0, 8, 0);
  trace_overwriting(&dst[assoclen+cryptlen],
  ↪  *((u64*)tmp));
  sktrc(SKTRC_HASH(0, 0, 2, 1), "Overwriting data at
  ↪  address %016llx (0x%016llx)",
  ↪  &dst[assoclen+cryptlen], *((u64*)tmp));
   scatterwalk_map_and_copy(tmp, dst, 4, 4, 1);
   scatterwalk_map_and_copy(tmp + 1, dst, assoclen +
   ↪  cryptlen, 4, 1);
// ...
static int __init crypto_authenc_esn_module_init(void)
{
  trace_loaded(0);
  sktrc(SKTRC_HASH(0, 1, 2, 2), "Instrumented module
  ↪  loaded for controlled analysis.");
  return
  ↪  crypto_register_template(&crypto_authenc_esn_tmpl);
}
// ...
```

**(a) Arquivo `authencesn.c`.**

```
// ...
static int aead_recvmsg(struct socket *sock, struct
↪  msghdr *msg,
// ...
  int ret = 0;

  lock_sock(sk);
  trace_receiving(0);
  sktrc(SKTRC_HASH(0, 0, 3, 1), "Receiving payload
  ↪  fragment...");
  while (msg_data_left(msg)) {
    int err = _aead_recvmsg(sock, msg, ignored, flags);
// ...
```

**(b) Arquivo `algif_aead.c`.**

```
//...
TRACE_EVENT(overwriting,
  TP_PROTO(struct scatterlist *addr, u64 data),
  TP_ARGS(addr, data),
  TP_STRUCT__entry(
    __field(struct scatterlist*, addr)
    __field(u64, data)
  ),
  TP_fast_assign(
    __entry->addr = addr;
    __entry->data = data;
  ),
  TP_printk("Overwriting data at address %016llx
  ↪  (0x%016llx)",
    __entry->addr, __entry->data)
  );

TRACE_EVENT(loaded,
  TP_PROTO(int required),
  TP_ARGS(required),
  TP_STRUCT__entry(
    __field(int, required)
  ),
  TP_fast_assign(
    __entry->required = required;
  ),
  TP_printk("Instrumented module loaded for controlled
  ↪  analysis.%s",
    __entry->required ? "" : "")
  );

TRACE_EVENT(receiving,
  TP_PROTO(int required),
  TP_ARGS(required),
  TP_STRUCT__entry(
    __field(int, required)
  ),
  TP_fast_assign(
    __entry->required = required;
  ),
  TP_printk("Receiving payload fragment...%s",
    __entry->required ? "" : "")
  );
//...
```

**(c) Arquivo `poc_copy_fail.h`.**

```
// ...
  {                                  // 0x0002  XXXX
    "authencesn",                           // 0000
    "crypto_authenc_esn_decrypt",           // 0001
    "crypto_authenc_esn_module_init"        // 0002
  },
  {                                  // 0x0003  XXXX
    "algif_aead",                           // 0000
    "aead_recvmsg"                          // 0001
  }
// ...
```

**(d) Arquivo `sktrc.c`.**

**Figura 6. Implementação da instrumentação adotada no cenário experimental da PoC.**