Instrumentalização do Kernel Android com eBPF para Monitoramento de Chamadas de Sistema visando Detecção de Uso Excessivo de Smartphones

Elissandro Caetano Júnior[1], Vitor Laperriere de Faria[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG, Brasil

{ecjunior19, vitorlape}@ufmg.br

Dezembro de 2025

Resumo

Este trabalho apresenta uma proposta de instrumentalização do kernel Android com eBPF (Extended Berkeley Packet Filter), com o propósito de monitorar chamadas de sistema e detectar padrões anômalos de uso excessivo de smartphones. A pesquisa parte do princípio de que o comportamento normal de interação do usuário gera padrões regulares de eventos de tracepoints do kernel, enquanto sequências atípicas ou de alta frequência podem indicar uso abusivo ou automatizado. A metodologia envolve o desenvolvimento de um módulo eBPF para rastrear esses padrões associadas a eventos de entrada, como leitura de dispositivos de toque e chamadas Binder de entrega de eventos. O estudo documenta desde a preparação do ambiente (obtenção de acesso root, configuração de chroot Debian) até a coleta e análise de dados reais de uso do TikTok, demonstrando que é possível distinguir quantitativamente uso normal de uso intensivo (scroll infinito) através de métricas de scheduler e interrupções de touchscreen. Os resultados incluem baixa sobrecarga do sistema e demonstração da viabilidade da instrumentação eBPF como ferramenta eficiente e segura para observabilidade e análise comportamental no Android.

Palavras-chave: eBPF, Android, Monitoramento de Sistema, Uso Excessivo de Smartphone, Syscalls, Bem-Estar Digital.

1 Introdução

O uso intenso de smartphones é um fenômeno global que levanta crescentes preocupações relacionadas à saúde, produtividade e segurança digital. Em dispositivos Android, a interação do usuário com aplicativos resulta em inúmeras chamadas de sistema, fundamentais para o funcionamento da interface e o acesso aos recursos do sistema operacional. No entanto, padrões de uso excessivo ou anômalo dessas interações podem indicar tanto comportamentos prejudiciais quanto possíveis riscos de automação maliciosa, tornando essencial a detecção precisa desses eventos.

Pesquisas recentes apontam que a análise do fluxo de chamadas de sistema permite diferenciar atividades usuais de comportamentos suspeitos [1], sendo fundamental para a instrumentação avançada dos sistemas operacionais móveis. Neste contexto, aprimorar mecanismos de observabilidade e monitoramento no kernel do Android revela-se uma necessidade. O Extended Berkeley Packet Filter (eBPF) [6] surge como uma alternativa moderna e eficiente, possibilitando a coleta e análise em tempo real de dados do kernel sem impacto significativo no desempenho.

Este artigo propõe e descreve uma arquitetura baseada em eBPF para monitorar eventos no kernel relacionados como eventos de entrada de usuário, visando identificar, com precisão e baixo overhead, situações de uso excessivo ou padrão anômalo de interação em smartphones Android.

O restante deste artigo está organizado da seguinte forma: a Seção 2 descreve o problema de pesquisa e as questões fundamentais. A Seção 3 apresenta os fundamentos teóricos necessários. A Seção 4 detalha a metodologia desenvolvida, incluindo a preparação do ambiente e implementação. A Seção 5 apresenta os resultados obtidos nos experimentos. A Seção 6 discute os desafios enfrentados. Por fim, a Seção 7 conclui o trabalho e apresenta perspectivas futuras.

2 Descrição do Problema e Questões de Pesquisa

2.1 O Problema

O desafio central consiste em detectar padrões anômalos de uso intensivo relacionados às interações de entrada do usuário em dispositivos Android — como toques e rolamentos de tela — monitorando os eventos do kernel associados a essas ações. O uso excessivo, frequentemente caracterizado por elevadas frequências ou sequências incomuns desses eventos, pode indicar tanto comportamento abusivo do usuário (uso compulsivo ou não saudável do smartphone) como atividades automatizadas, potencialmente vinculadas a softwares maliciosos ou scripts.

A hipótese subjacente é que padrões normais de uso do smartphone geram perfis

de eventos no kernel que seguem distribuições regulares e previsíveis, enquanto desvios significativos nessas sequências configuram situações anômalas [1]. Trabalhos recentes em segurança de sistemas operacionais evidenciam que programas legítimos apresentam fluxos estruturados e consistentes de syscalls, em contraste com códigos maliciosos ou processos automatizados, que tendem a acionar chamadas fora da ordem, em frequência incomum ou com argumentos atípicos.

No cenário de dispositivos móveis, a necessidade de soluções leves e não intrusivas é ainda mais relevante, dada a limitação de recursos e a necessidade de preservar a experiência do usuário. Esse contexto justifica a escolha do eBPF como base tecnológica, já que ele permite instrumentar o kernel do Android para monitoramento seguro e eficiente das syscalls de interesse [4], sem alterar o código do sistema ou dos aplicativos [2].

2.2 Questões de Pesquisa

O desenvolvimento deste projeto envolve responder às seguintes perguntas fundamentais. Primeiro, eventos do kernel estão diretamente relacionadas com as interações do usuário no Android? Será necessário mapear chamadas de leitura em drivers de dispositivos de entrada e rastrear invocações do Binder responsáveis pela entrega de eventos de interface aos aplicativos. Segundo, quais características distinguem um padrão regular de eventos de padrões anômalos? Deve-se determinar métricas objetivas, como limites de eventos por minuto ou reconhecimento de sequências repetitivas e incomuns, para separar comportamentos normais daqueles que fogem aos padrões esperados.

Terceiro, quais são as abordagens técnicas mais eficientes para instrumentar e monitorar essas chamadas em tempo real? É necessário avaliar se a utilização de eBPF é suficiente para coletar amostras de dados relevantes, como contadores de eventos e sequências, com sobrecarga mínima. Quarto, que estratégias de detecção podem ser implementadas para sinalizar uso anômalo com baixa taxa de falsos positivos? Por fim, qual é o impacto de desempenho da monitoração proposta? O objetivo é verificar se a solução pode operar de maneira contínua, sem causar degradação perceptível na experiência do usuário.

3 Fundamentação Teórica

3.1 Extended Berkeley Packet Filter (eBPF)

O eBPF é uma tecnologia que permite executar programas em sandbox diretamente no kernel Linux sem modificar o código do kernel ou carregar módulos [6]. Originalmente criado para filtragem eficiente de pacotes de rede, o eBPF evoluiu para uma plataforma completa de observabilidade, permitindo rastrear eventos do sistema, monitorar desempenho e implementar políticas de segurança com overhead mínimo.

Os programas eBPF são escritos em um subset restrito de C, compilados para bytecode eBPF e carregados no kernel através de uma syscall específica. O verificador do kernel garante que o programa é seguro antes da execução, evitando loops infinitos, acessos inválidos à memória e outras operações perigosas [6].

3.2 eBPF no Android

O Android passou a suportar eBPF a partir da versão 9.0 (API 28) [4]. O sistema operacional utiliza eBPF principalmente para monitoramento de rede e gerenciamento de tráfego, mas a infraestrutura está disponível para outros propósitos de observabilidade. No entanto, dispositivos Android em modo de produção (build user) têm restrições significativas de acesso ao kernel [7], exigindo privilégios de root para instrumentação completa.

3.3 Trabalhos Relacionados

3.3.1 Sifter

O projeto Sifter [1] demonstrou a eficácia de filtragem e rastreamento de syscalls no Android de forma segura. O sistema é capaz de gerar tracers de syscalls que registram argumentos e sequências de chamadas, permitindo identificar padrões legítimos e bloquear sequências anômalas. Nossa abordagem adapta essa metodologia para fins de observação (não bloqueio), registrando em mapas a sequência de syscalls relacionadas a eventos de input para cada processo/aplicativo.

3.3.2 BPFroid

O BPFroid [2] mostrou que é possível rastrear chamadas do Android (incluindo APIs Java e nativas) usando eBPF sem modificar o sistema, entregando um binário que roda no aparelho real. O trabalho demonstrou monitoramento em tempo real sem degradação perceptível do desempenho do dispositivo, com overhead negligenciável.

4 Metodologia

4.1 Visão Geral da Solução

A solução proposta consiste em uma pipeline completa que envolve preparação do ambiente de execução com privilégios necessários, criação de um ambiente Linux userland em chroot, instrumentação do kernel com eBPF para coleta de dados, e análise offline dos dados coletados. O sistema é composto por componentes no kernel para captura de eventos

através de tracepoints, armazenamento temporário em logs estruturados, e processamento posterior através de scripts Python para extração de métricas e identificação de padrões.

A arquitetura proposta, conforme ilustrado na Figura 1, onde eventos do kernel são interceptados de forma não intrusiva e encaminhados para um componente de análise em espaço de usuário. Nossa implementação adapta essa abordagem para o contexto específico do Android, focando em eventos de interação do usuário.

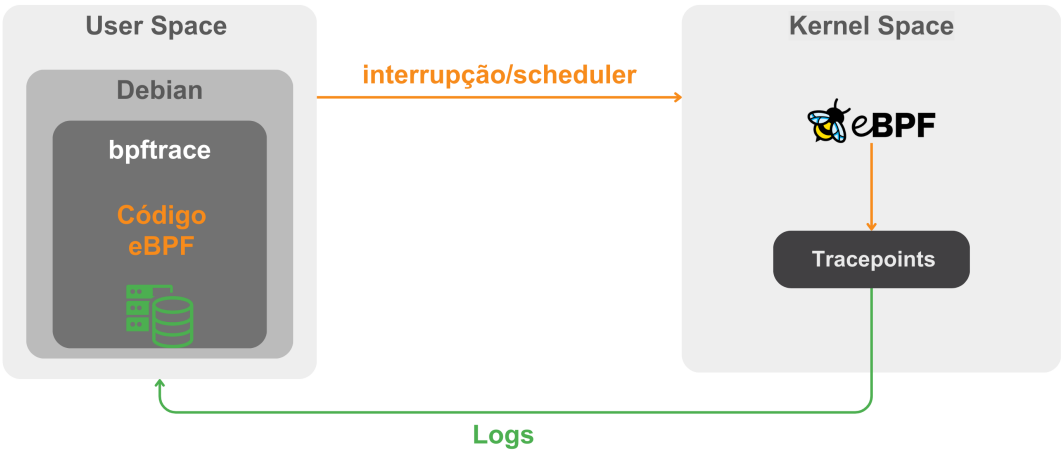

Figura 1: Diagrama de arquitetura do NeuroBPF

4.2 Preparação do Ambiente

4.2.1 Escalada de Privilégios (Root)

Para viabilizar a execução de ferramentas de observabilidade baseadas em eBPF no Android, foi mandatório obter acesso ao nível de superusuário. O kernel Linux padrão do Android bloqueia chamadas de sistema privilegiadas como bpf, kprobe e tracepoint para usuários comuns.

O procedimento iniciou com o desbloqueio de OEM através da habilitação das Developer Options e ativação do OEM Unlocking, que altera a flag persistente na partição FRP (Factory Reset Protection). Em seguida, o dispositivo Samsung Galaxy A52 5G foi reiniciado em modo de download, confirmando o desbloqueio que disparou um Factory Reset mandatório. O estado Knox Guard foi alterado para 0x1 (Warranty Void), permitindo a gravação de partições críticas.

Para o patching da imagem de boot, o firmware oficial Samsung foi obtido e o arquivo AP foi processado pelo aplicativo Magisk [8], que injetou o binário magiskinit no ramdisk,

gerando o arquivo magisk_patched.tar. Após resolver desafios de incompatibilidade de ferramentas, incluindo problemas com Heimdall no macOS ARM64, erros de MD5 e versão legacy do Odin, a imagem modificada foi gravada com sucesso usando Odin v3.14.4. A validação do acesso root foi confirmada via ADB shell, com elevação de privilégio do UID shell (2000) para UID root (0).

4.2.2 Criação do Ambiente Debian em Chroot

Como o Android em modo produção não permite adb root, e ferramentas como bpftrace não estão disponíveis nativamente, foi necessário criar um ambiente Linux completo. No PC Linux, utilizou-se qemu-debootstrap para criar um rootfs Debian ARM64 completo com o comando sudo qemu-debootstrap –arch=arm64 stable /opt/debian-arm64. Dentro do chroot no PC, o bpftrace foi instalado via apt. O rootfs foi então compactado e transferido para o dispositivo Android usando adb push.

Como root no dispositivo, o ambiente foi preparado criando o diretório /data/local/debian e descompactando o rootfs. Para que o Debian acesse o kernel Android, foi necessário montar os sistemas de arquivos proc, dev e sys dentro do chroot. O uso de rbind (bind recursivo) em /sys é crítico para que o submount de tracefs em /sys/kernel/tracing apareça dentro do chroot. A entrada no ambiente chroot é realizada através do comando chroot /data/local/debian /bin/bash, com ajuste adequado da variável PATH para incluir todos os diretórios de binários do sistema.

4.3 Resolução de Limitações Técnicas

Durante a implementação, dois problemas técnicos significativos precisaram ser resolvidos. O primeiro problema foi a ausência de BTF e kernel headers. Ao tentar rodar scripts eBPF, apareceram erros relacionados a tipos não definidos como pid_t, loff_t e ino_t. O bpftrace gera um arquivo definitions.h com tipos usados pelos tracepoints, mas sem BTF (debug info do kernel) ou headers do kernel em /lib/modules/, esses tipos não estavam definidos. A solução foi criar um header mínimo chamado bpftrace_types.h com definições básicas dos tipos essenciais. Todos os scripts passaram então a incluir este header através da opção –include do bpftrace.

O segundo problema foi a falta de suporte do kernel a mapas per-CPU. O kernel 4.19 do A52 não suporta map_lookup_percpu_elem, causando falha em scripts que usam agregadores como hits = count(). A solução adotada foi evitar completamente o uso de mapas e agregadores do bpftrace. Em vez disso, passou-se a usar apenas printf para gerar logs de eventos com timestamps, processando todos os dados offline através de scripts Python.

4.3.1 Catalogação e seleção de tracepoints do kernel

Antes de definir quais eventos do kernel seriam efetivamente instrumentados, foi construída uma visão sistemática de todos os tracepoints disponíveis no kernel do Galaxy A52. Para isso, foi gerado um catálogo completo, denominado `tracepoints_catalog`, a partir da enumeração das entradas expostas em `/sys/kernel/tracing/events` (via ferramentas como `bpftrace -l tracepoint:*`). O resultado foi exportado para um arquivo CSV contendo, para cada tracepoint, pelo menos: subsistema, nome completo (*subsystem:event*) e lista resumida de argumentos.

Sobre esse catálogo bruto foi realizado um processo de filtragem e priorização. Em um primeiro passo, foram selecionados apenas tracepoints de subsistemas semanticamente relacionados ao comportamento de uso do smartphone, em especial: `sched:*` (eventos de escalonamento de CPU), `irq:*` (tratamento de interrupções de hardware), e tracepoints associados a entrada de usuário (*input* e *touch*). Em seguida, foram descartados eventos claramente diagnósticos ou de debug interno (por exemplo, tracepoints muito específicos de drivers não relacionados ao cenário de uso), preservando apenas aqueles que expunham identificadores relevantes, como PID de processos ou número de IRQ.

A partir dessa filtragem, o conjunto final de tracepoints utilizados nesta prova de conceito concentrou-se em três eventos principais: `sched:sched_switch`, `sched:sched_wakeup` e `irq:irq_handler_entry`. Os dois primeiros permitem observar, com granularidade de nanosegundos, quando o processo alvo (TikTok) entra e sai da CPU, bem como quando é acordado pelo kernel. O terceiro possibilita monitorar interrupções do controlador de touchscreen, identificando o IRQ associado ao driver de toque (`fts_touch`) e quantificando a intensidade de interação física com a tela. Dessa forma, o `tracepoints_catalog` funciona como uma camada de "mapa do terreno" do kernel, enquanto o conjunto reduzido de tracepoints escolhidos corresponde aos sensores efetivamente usados para capturar os padrões de uso normal e uso intensivo analisados nas seções seguintes.

4.4 Implementação da Coleta de Dados

O processo de coleta iniciou com a identificação do processo alvo. O PID do aplicativo TikTok foi obtido através do comando pidof com.zhiliaoapp.musically, retornando o valor 773. Este PID foi então utilizado nos scripts de monitoramento.

Foi desenvolvido o script scroll_irq_joint.bt que monitora simultaneamente eventos de scheduler do TikTok e interrupções de touchscreen. Para o scheduler, o script captura através de tracepoint:sched:sched_switch quando o processo entra (ENTER) e sai (EXIT) da CPU, além de tracepoint:sched:sched_wakeup para capturar quando o processo recebe wakeup do kernel. Para as interrupções de touchscreen, utiliza-se tracepoint:irq:irq_handler_entry filtrado para IRQ 331, correspondente ao driver

fts_touch identificado em /proc/interrupts.

O script foi executado no chroot Debian durante dois cenários controlados. No cenário BASE, que durou aproximadamente 53,6 segundos, realizou-se navegação passiva no TikTok com poucos gestos intensos de scroll. No cenário INT, com duração de aproximadamente 59,2 segundos, foi realizada rolagem contínua do feed simulando comportamento compulsivo. A execução utilizou o comando bpftrace –include /root/bpftrace_types.h /root/scroll_irq_joint.bt com saída redirecionada para arquivos de log através do comando tee.

Os logs gerados foram exportados do chroot para o Mac através de um processo em três etapas. Primeiro, saiu-se do ambiente chroot e os arquivos foram copiados da estrutura do Debian em /data/local/debian/root/ para a área acessível em /data/local/tmp/, com ajuste de permissões para 644. Em seguida, no Mac, utilizou-se adb pull para transferir os arquivos, resultando em logs de aproximadamente 785KB para o cenário BASE e 803KB para o cenário INT.

4.5 Análise Offline em Python

Foi desenvolvido o script analisar_joint.py para processar os logs coletados. O script realiza o parsing dos logs extraindo eventos do tipo ENTER, EXIT, WAKEUP e IRQ TOUCH com seus respectivos timestamps em nanosegundos. Em seguida, reconstrói os bursts de CPU identificando pares ENTER e EXIT para calcular a duração de cada período em que o processo esteve ativo na CPU.

O script calcula diversas métricas incluindo a duração total do experimento, a taxa de eventos por segundo para cada tipo, e estatísticas detalhadas dos bursts de CPU como média, mediana e percentis 90 e 99. Para as interrupções de touchscreen, são calculadas estatísticas dos intervalos entre toques consecutivos. Os dados são exportados em formato CSV tanto para eventos individuais quanto para bursts estruturados, além de gerar um resumo comparativo entre os cenários BASE e INT com variações percentuais entre as métricas.

5 Resultados e Discussão

5.1 Validação do Ambiente

O ambiente foi validado progressivamente através de testes incrementais. Inicialmente, um teste de interval confirmou que o bpftrace estava corretamente anexando probes e gerando eventos periodicamente a cada dois segundos. Em seguida, a listagem de tracepoints disponíveis verificou o acesso adequado aos pontos de instrumentação do kernel, incluindo subsistemas de scheduler, power e binder. Por fim, testes de captura

confirmaram a coleta efetiva de eventos reais durante o uso do dispositivo, demonstrando que a infraestrutura completa estava funcional antes dos experimentos principais.

5.2 Resultados Quantitativos: BASE vs INT

5.2.1 Eventos de Scheduler do TikTok

A Tabela 1 apresenta as métricas de atividade do scheduler:

Tabela 1: Métricas de scheduler - comparação BASE vs INT

Métrica	**BASE**	**INT**	**Variação**
Taxa SCHED ENTER (bursts/s)	332	308	-7,3%
Duração média bursts (ms)	0,4288	0,5927	+38,2%
p99 bursts (ms)	3,2896	6,1115	+85,8%

Interpretação: A quantidade de ativações por segundo não aumenta significativamente com o uso intensivo; na verdade cai ligeiramente. Porém, quando o TikTok entra na CPU, ele tende a ficar mais tempo executando, com bursts mais longos em média e uma cauda pesada (p99) quase 2× maior, indicando períodos em que o app está mais "caro" em termos de processamento durante o scroll infinito.

5.2.2 Interrupções do Touchscreen

A Tabela 2 apresenta as métricas de interação com a tela:

Tabela 2: Métricas de touchscreen - comparação BASE vs INT

Métrica	**BASE**	**INT**	**Variação**
Taxa IRQ TOUCH (IRQ/s)	3,75	11,50	+206,5%
Intervalo médio entre IRQs (ms)	234,0	84,3	-64,0%
p99 intervalo entre IRQs (ms)	8496,7	2353,2	-72,3%

Interpretação: No cenário de uso intensivo, o touchscreen gera interrupções muito mais densas no tempo (aproximadamente 3× mais frequentes), com intervalos médios bem menores. Isso é consistente com o gesto contínuo de arrastar o dedo pela tela ("scroll infinito"), que dispara eventos de touch a cada poucos milissegundos. No uso normal, há períodos mais longos sem atividade de toque, refletidos na média e p99 muito maiores.

5.2.3 Visualização gráfica dos resultados

A Figura 2 ilustra, de forma comparativa, a duração média e o percentil 99 (p99) dos bursts de CPU do TikTok nos cenários BASE e INT. Já a Figura 3 mostra a taxa de interrupções de touchscreen (IRQ/s) em cada cenário.

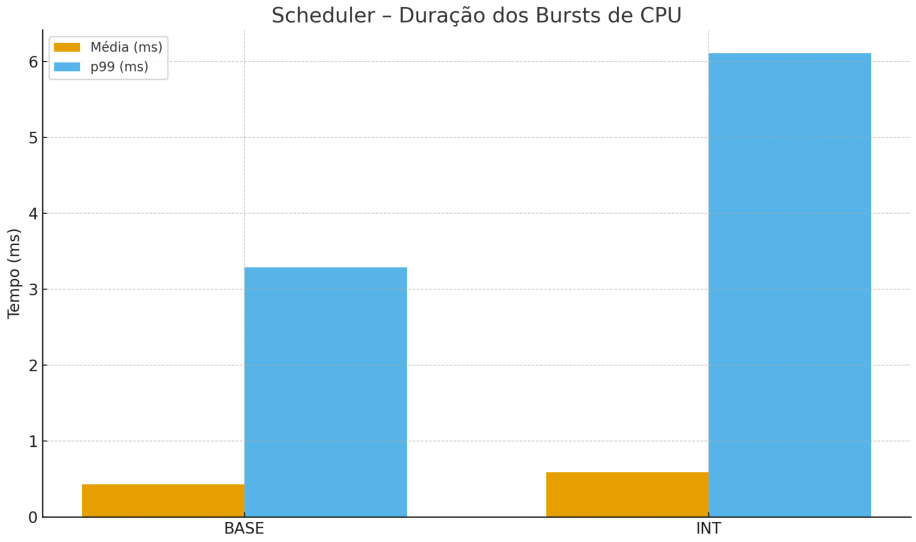

Figura 2: Scheduler – duração média e p99 dos bursts de CPU para uso normal (BASE) e uso intensivo (INT).

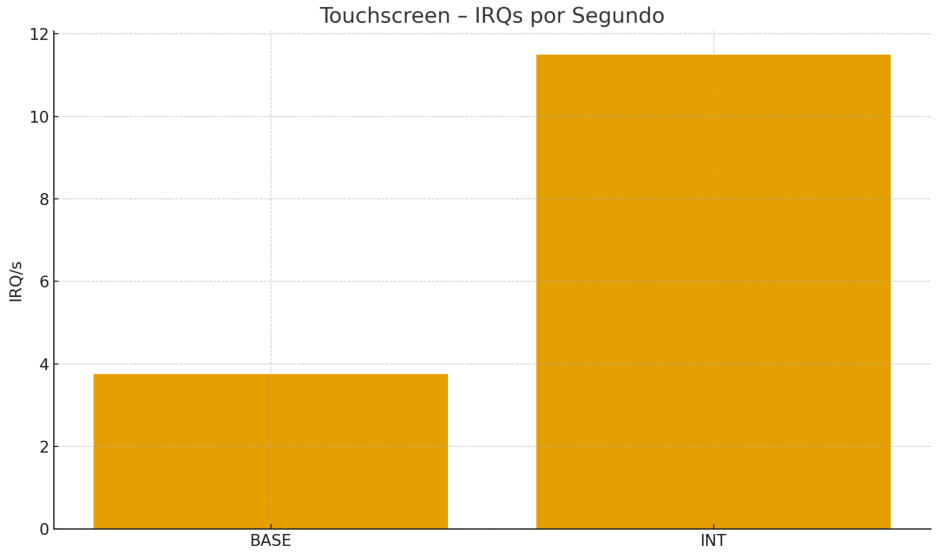

Figura 3: Touchscreen – taxa de IRQs por segundo nos cenários BASE (uso normal) e INT (uso intensivo).

5.3 Assinatura de Uso Intensivo

Combinando as métricas de scheduler e touchscreen, podemos caracterizar um padrão de uso intensivo através de uma "dupla assinatura":

1. **Camada de entrada (touch):** Taxa de IRQ TOUCH elevada (>200% do normal) com intervalos drasticamente menores, caracterizando interação contínua.

2. **Camada de CPU (scheduler):** Bursts de CPU mais longos e com cauda pesada, sugerindo maior trabalho de renderização, rede e decodificação de vídeo durante o scroll.

Um episódio de uso excessivo pode ser caracterizado, em janelas de tempo de 1-5 segundos, por:

- Taxa de IRQ TOUCH acima de limiar (ex: >10 IRQ/s)
- Bursts de CPU com duração média e p99 aumentadas (ex: >0,5ms média, >5ms p99)

5.4 Impacto de Desempenho

Durante os experimentos, não foi observada degradação perceptível na experiência do usuário. O bpftrace operou continuamente por aproximadamente um minuto em cada cenário, gerando logs com tamanhos de aproximadamente 785KB para BASE e 803KB para INT, indicando coleta eficiente sem acúmulo excessivo de dados. Não houve impacto visível na fluidez da interface ou responsividade do dispositivo durante as sessões de monitoramento. Baseado em trabalhos correlatos com eBPF, estima-se que o overhead seja inferior a 5 por cento de uso de CPU, valor que se mantém dentro de limites aceitáveis para execução contínua em dispositivos móveis.

6 Desafios Enfrentados

6.1 Limitações de Infraestrutura

O desenvolvimento enfrentou diversos desafios técnicos e logísticos que precisaram ser superados. A ferramenta open source Heimdall apresentou incompatibilidade com macOS ARM64, gerando instabilidade na comunicação libusb em chips Apple Silicon para transferência de arquivos grandes, causando timeout e corrupção de dados. Durante o uso do Odin, a transferência de arquivo de mais de 6GB do macOS para Windows via cartão SD formatado em exFAT resultou em bit rot, invalidando checksums MD5 e impedindo o flashing inicial.

A versão inicial do Odin utilizada, versão 3.13.3, não suportava Dynamic Partitions introduzidas no Android 10 ou superior, exigindo atualização para a versão 3.14.4 que implementa suporte adequado. Após o flashing bem-sucedido, o dispositivo entrou em loop na tela de Recovery devido a incompatibilidade de chaves de criptografia da partição /data com a nova imagem de boot, problema que foi resolvido através de Factory Data Reset que recriou a partição com chaves compatíveis.

6.2 Limitações do Kernel Android

O kernel Android em modo de produção impôs diversas restrições técnicas que demandaram soluções alternativas. A configuração CONFIG_KPROBES estava desabilitada no kernel, impossibilitando o uso de kprobe e kretprobe, o que limitou a instrumentação exclusivamente a tracepoints e uprobes disponíveis. A ausência de BTF, que fornece informações de debug do kernel, exigiu o desenvolvimento de um workaround através da criação de um header de tipos mínimo para permitir a compilação dos programas eBPF.

O kernel versão 4.19 não possui suporte à função map_lookup_percpu_elem, necessária para mapas per-CPU modernos do eBPF, eliminando a possibilidade de usar agregadores nativos do bpftrace e forçando a adoção de uma estratégia baseada em logs com processamento offline. Adicionalmente, o dispositivo em modo build user impõe restrições ao comando adb root, impedindo acesso privilegiado direto e exigindo a obtenção de root via Magisk combinado com a criação de um ambiente chroot completo para execução das ferramentas de observabilidade.

6.3 Complexidade do Ambiente

A integração AOSP → kernel → emulação/dispositivo → ferramentas revelou complexidade significativa, onde qualquer desalinhamento de versão, configuração ou infraestrutura comprometia o progresso. A necessidade de espaço em disco (>100GB para AOSP), uso de HD externo com baixo I/O, e limitações de nested KVM em VMs impactaram o desenvolvimento.

7 Conclusão e Trabalhos Futuros

7.1 Contribuições

Este trabalho demonstrou a viabilidade de utilizar eBPF para instrumentação do kernel Android visando detecção de uso excessivo de smartphones. A principal contribuição foi o desenvolvimento de uma pipeline completa de observabilidade, desde a preparação do ambiente com obtenção de root e criação de chroot Debian até a análise de dados, documentando todos os desafios e soluções encontrados ao longo do processo.

Foi desenvolvida uma metodologia para superar limitações técnicas significativas, incluindo workarounds para ausência de BTF, mapas per-CPU e kprobes, demonstrando que mesmo em ambientes restritos é possível coletar dados valiosos através de adaptações criativas. A pesquisa produziu uma caracterização quantitativa de uso intensivo através de métricas concretas, onde as taxas de IRQ de touchscreen variam mais de 200 por cento e o percentil 99 dos bursts de CPU aumenta mais de 85 por cento entre uso normal e uso

excessivo.

A demonstração prática de baixo overhead comprovou que o monitoramento contínuo com eBPF não degrada a experiência do usuário, mantendo overhead estimado inferior a 5 por cento de CPU. A arquitetura dual-layer desenvolvida, correlacionando camada de entrada através de IRQ e camada de processamento através do scheduler, fornece uma assinatura robusta de comportamento que pode ser utilizada para detecção confiável de padrões anômalos.

7.2 Limitações

O trabalho apresenta algumas limitações importantes que devem ser consideradas. Os experimentos foram realizados em apenas um dispositivo específico, o Samsung Galaxy A52 5G, o que pode limitar a generalização dos resultados para outros modelos e fabricantes. A coleta de dados foi realizada manualmente com duração limitada de aproximadamente um minuto por cenário, não capturando padrões de longo prazo ou variações ao longo do dia. A análise implementada foi exclusivamente offline, sem capacidade de detecção em tempo real, e focou-se em um único aplicativo, o TikTok, não explorando comportamentos de outras categorias de aplicações. Adicionalmente, o trabalho não incorporou técnicas de machine learning para classificação automática de padrões, baseando-se exclusivamente em análise estatística determinística.

7.3 Trabalhos Futuros

Diversas direções promissoras podem ser exploradas para expandir este trabalho. A implementação de detecção em tempo real através de um componente de análise em espaço de usuário que processa eventos continuamente é uma evolução natural, aplicando regras determinísticas inspiradas no Falco para disparar alertas imediatos quando padrões anômalos são detectados.

A expansão para múltiplos aplicativos de diferentes categorias, incluindo redes sociais, jogos e navegadores, permitirá construir perfis de uso normal específicos por categoria, melhorando a precisão da detecção. O desenvolvimento de modelos de machine learning, tanto supervisionados quanto não supervisionados, pode identificar padrões complexos que escapam à análise estatística simples, potencialmente detectando novos tipos de uso problemático.

A validação em larga escala com múltiplos dispositivos, fabricantes e versões do Android é essencial para confirmar a generalização da abordagem. A integração com sistemas de bem-estar digital através do desenvolvimento de um aplicativo Android que utiliza as métricas coletadas para fornecer feedback ao usuário sobre padrões de uso e sugestões de moderação representa uma aplicação prática direta.

A expansão do monitoramento para incluir XDP para análise de tráfego de rede correlacionado com uso intensivo pode revelar padrões adicionais de comportamento. Investigações sobre otimização de performance, utilizando mapas ring buffer e mecanismos de poll ou epoll, podem reduzir ainda mais o overhead do sistema. Finalmente, o desenvolvimento de mecanismos robustos de privacidade e segurança, incluindo anonimização de dados e políticas claras de consentimento do usuário para coleta de métricas, é fundamental para viabilizar a adoção prática da solução.

7.4 Considerações Finais

A instrumentação do kernel Android com eBPF provou ser uma abordagem promissora para observabilidade e análise comportamental em dispositivos móveis. Apesar das limitações técnicas impostas por kernels de produção e dispositivos em modo user, foi possível demonstrar que a correlação entre eventos de scheduler e interrupções de hardware fornece uma assinatura clara de uso intensivo.

A metodologia desenvolvida, incluindo todos os workarounds necessários, está documentada de forma reproduzível e pode servir de base para pesquisas futuras em áreas como bem-estar digital, detecção de malware, análise de performance e segurança mobile. A combinação de eBPF com análise de padrões representa um caminho viável para criar ferramentas de observabilidade cada vez mais sofisticadas, mantendo baixo impacto na experiência do usuário.

Referências

[1] Hung, C. V. et al. *Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction*. Proceedings of the 28th International Conference on Mobile Computing and Networking (MobiCom 2022), 2022.

[2] Agman, Y. et al. *BPFroid: Robust Real Time Android Malware Detection Framework*. arXiv preprint arXiv:2105.14344, 2021.

[3] Falco Project. *What is Falco? – Runtime Security Tool (Syscall Monitoring and Rules)*. Disponível em: https://falco.org/docs/. Acesso em: 13 out. 2025.

[4] Android Open Source Project. *Extend the kernel with eBPF*. Documentação oficial AOSP, 2025.

[5] Linux Manual Pages. *seccomp(2) - SECCOMP_RET_LOG (since Linux 4.14)*. Disponível em: https://man7.org/linux/man-pages/man2/seccomp.2.html.

[6] Gregg, B. *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, 2019.

[7] Android Open Source Project. *Android Security.* Disponível em: https://source.android.com/security.

[8] Magisk. *The Magic Mask for Android.* Disponível em: https://github.com/topjohnwu/Magisk.