Do Bruto ao Brilho: Compactação Semântica com Preservação de Proveniência para Traços Dinâmicos de Malware Android

Cláudio Torres Júnior[1], **João Pincovscy**[2,3], **André Grégio**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
SecRET (`secret.inf.ufpr.br`) – Curitiba – PR – Brasil

[2]Centro de Pesquisa e Desenvolvimento para a Segurança das Comunicações (CEPESC)
[3]UnDF – Universidade do Distrito Federal – Brasília – DF – Brasil

{claudio.torres, gregio}@ufpr.br, pincovscy@cepesc.gov.br

Resumo. *A interpretação sequencial de logs de análise dinâmica de malware Android é custosa, pois a comparação entre execuções e verificação manual envolve correlação de dados massivos. Neste trabalho, apresenta-se SELENE, uma camada de organização semântica pós-coleta para gerar uma hierarquia auditável composta por eventos comportamentais, sinais de ciclo de vida target-aware, grafos de execução e pacotes de contexto com proveniência até o dado bruto. Foram avaliadas mais de 70 mil execuções (em Android 10 e 14), mostrando que a versão do Android altera a superfície comportamental observável e que pode-se alcançar compactação de mais de duas ordens de magnitude no tamanho dos traços e F1 médio de 0,983 frente a um oráculo independente.*

Abstract. *Interpreting sequentially Android malware dynamic analysis logs has high cost, since comparing runs or verifying them manually requires correlating massive data. In this work, we introduce SELENE, a post-collection semantic organization layer to generate an auditable hierarchy of behavioral events, lifecycle signals, execution graphs and context packs with provenance until the raw data. We evaluate more than 70K Android 10 and 14 malware traces, showing that the Android version impacts the observable behavioral surface. We have also accomplished over two orders of magnitude for compacting traces, and an average F1 of 0.983 when comparing our results with an independent oracle.*

1. Introdução

A análise dinâmica é uma das principais formas de observar comportamentos de malware Android que dependem da execução, como comunicação de rede, uso de serviços do sistema, acesso a dados privados e carregamento dinâmico de código [Tam et al. 2015, Sutter et al. 2024]. A evolução de plataformas de execução em larga escala ampliou essa observabilidade ao combinar ambientes controlados, múltiplas versões do Android e *tracers* configuráveis. Esse avanço desloca o gargalo da coleta para a interpretação: quanto mais artefatos são produzidos, maior é o custo de leitura, comparação e verificação.

Tal problema é evidente em traços de `strace` (chamadas de sistema) e registros de `logcat` (mensagens do Android). Uma única execução pode produzir centenas de milhares de linhas, misturando ações do aplicativo alvo, inicialização do sistema, serviços

Android, eventos do emulador e erros globais não relacionados ao APK analisado. O traço bruto continua necessário para auditoria, mas é inadequado como primeira unidade de análise em larga escala. A literatura trata partes desse problema. Trabalhos baseados em chamadas de sistema reconstroem comportamento ou treinam classificadores [Tam et al. 2015, Canfora et al. 2015, Bhat et al. 2023, Vyšniūnas et al. 2024, Malik et al. 2026]. Trabalhos de abstração e compressão de *logs* reduzem volume por meio de *templates*, variáveis e padrões recorrentes [He et al. 2017, Veuskens et al. 2023, Liu et al. 2019, Wei et al. 2021, Yu et al. 2026]. Abordagens de proveniência correlacionam eventos e preservam relações causais em sistemas multi-entidade [Hassan et al. 2020, Li et al. 2024]. Ainda assim, permanece pouco explorado o espaço entre essas linhas: compactar traços dinâmicos Android com semântica comportamental, separação entre alvo e ruído e ponte verificável até o dado bruto.

Este trabalho apresenta o **SELENE** (*Semantic Evidence Layer for Execution Navigation and Explainability*). Organizamos `strace`, `logcat` e metadados em uma hierarquia composta por L0 estruturado, agregações L0.5, eventos L1, sinais de ciclo de vida, representações em grafo, índice de evidências e pacotes de contexto. Cada *claim* derivado mantém referência para a evidência que o sustenta, preservando o *raw* como raiz auditável. O SELENE foi avaliado em 44.485 execuções de malware Android no Android 10 e em 30.746 APKs também executados no Android 14. A avaliação cobre custo dos traços brutos, redução por camada, fidelidade semântica frente a um oráculo independente, navegabilidade dos *claims* e diferenças pareadas entre versões do Android.

As principais contribuições deste trabalho são: (i) a introdução de uma camada de compactação semântica auditável que reduz o volume de linhas de `strace` em mais de duas ordens de grandeza sem descartar a rastreabilidade até o dado bruto (hierarquia de evidências com proveniência fim-a-fim para traços dinâmicos Android); (ii) um processo de navegação hierárquica verificável entre pacotes de contexto (compact, standard, evidence-rich, hybrid), eventos L1, agregações L0.5 e as linhas originais de *strace/logcat*, permitindo que o analista parta de um sumário de poucos milhares de tokens e chegue à evidência bruta em poucos passos; e (iii) viabilização de comparações horizontais entre versões do Android ao fornecer representações comportamentais padronizadas e pareáveis por APK (a aplicação sobre 30.746 amostras pareadas em Android 10 e 14 revelou que a versão do sistema altera significativamente a superfície observável, evidenciando a necessidade de análises multiambiente).

2. Trabalhos Relacionados

A literatura que fundamenta o **SELENE** distribui-se em quatro linhas: reconstrução semântica a partir de *syscalls*, abstração e compressão de *logs*, proveniência em eventos correlacionados e análise dinâmica Android em escala. Em conjunto, esses trabalhos indicam que compactar traços dinâmicos envolve preservar sinal comportamental, separar atividade do alvo de ruído sistêmico e manter rastreabilidade até o dado bruto. A Tabela 1 resume o posicionamento do **SELENE** frente aos trabalhos mais próximos.

Agrupamento de *syscalls* e reconstrução semântica.

O uso de chamadas de sistema para inferir comportamento Android é consolidado. Crowdroid mostrou que frequências de *syscalls* produzem assinaturas úteis para detecção [Burguera et al. 2011], e DroidScope reconstruiu visões do sistema operacio-

nal e da máquina virtual Dalvik a partir da camada de virtualização [Yan and Yin 2012]. O CopperDroid aproximou-se diretamente do problema deste artigo ao combinar *syscalls*, o mecanismo IPC Binder, RPC e dependências entre chamadas para reconstruir comportamentos de alto nível [Tam et al. 2015].

Trabalhos posteriores reforçam que grupos, sequências e contexto de execução são mais informativos do que chamadas isoladas [Canfora et al. 2015, Bhat et al. 2023, Vyšniūnas et al. 2024, Malik et al. 2026]. Esses resultados sustentam a taxonomia L1 do **SELENE**: sua organização não parte de uma classificação arbitrária, mas da intuição documentada de que eventos derivados de contexto, recurso, dependência e janela temporal são mais estáveis que o traço cru. O Quark ilustra princípio semelhante em uma camada estática, ao construir regras semânticas sobre *bytecode*, permissões e APIs [Quark Engine Project 2026]; neste artigo, o foco está nos observáveis dinâmicos e na preservação de ponte para `strace` e `logcat`.

Abstração e compressão de *logs*.

A literatura de *logs* mostra que a redução de volume depende da separação entre estrutura recorrente e variáveis dinâmicas. Drain extrai *templates* de forma *online* [He et al. 2017], VALB preserva categorias de variáveis para tarefas posteriores [Veuskens et al. 2023], e LogZip, LogReducer e DeLog exploram *templates*, variáveis, assinaturas e padrões recorrentes para compressão [Liu et al. 2019, Wei et al. 2021, Yu et al. 2026]. O **SELENE** incorpora essas ideias, mas não busca apenas reduzir *bytes*: seu objetivo é reduzir custo de leitura em traços Android mantendo significado comportamental e proveniência.

Proveniência e correlação multi-fonte.

OmegaLog e T-Trace mostram que a interpretação melhora quando *logs* são correlacionados a grafos de proveniência e relações causais [Hassan et al. 2020, Li et al. 2024]. O **SELENE** adota a mesma exigência de não desconectar resumo e evidência, mas em outro nível de granularidade: execuções individuais de APKs em ambiente Android, nas quais cada *claim* deve permanecer verificável no artefato bruto correspondente.

Análise dinâmica Android e evasão.

A análise dinâmica evoluiu em cobertura e instrumentação, mas a organização dos artefatos coletados permanece um problema aberto em execuções de grande volume [Sutter et al. 2024]. Estudos sobre furtividade e evasão mostram que o ambiente de análise pode alterar o comportamento observado [Kumar et al. 2023, Suo et al. 2025], o que reforça a necessidade de comparar execuções entre versões. Neste trabalho, a coleta foi realizada por uma infraestrutura modular de análise dinâmica Android, com suporte a *tracers* plugáveis e múltiplas versões do sistema. O **SELENE** atua após essa coleta, organizando os artefatos produzidos em evidências compactas, navegáveis e auditáveis.

3. SELENE

O **SELENE** atua após a coleta de artefatos de análise dinâmica Android, incluindo `strace`, `logcat` e metadados da execução. Seu objetivo é converter artefatos extensos e heterogêneos em uma hierarquia de evidências para triagem, comparação e investigação. A premissa é preservar o dado bruto como raiz auditável e, ao mesmo tempo, oferecer

Tabela 1. Posicionamento qualitativo do SELENE frente a trabalhos próximos.

Trabalho	Sinal / representação	Distinção frente ao SELENE
Crowdroid	Syscalls; frequências	Detecção por assinatura; sem navegação até o *raw*.
DroidScope	SO, Dalvik, APIs e instruções	Observabilidade de execução; sem compactação pós-coleta.
CopperDroid	Syscalls, Binder e comportamentos	Reconstrução semântica; não foca custo de leitura.
Quark	Bytecode, APIs e *behavior maps*	Análise estática; complementar à execução dinâmica.
VALB	Logs, *templates* e variáveis	Abstração genérica; sem semântica Android.
LogZip / LogReducer / DeLog	Logs, padrões e assinaturas	Compressão em *bytes*; sem triagem comportamental.
OmegaLog	Logs e auditoria com proveniência	Causalidade multi-serviço; não execução de APK.
T-Trace	*Syslog*, *firewall* e comunidades	Campanhas APT; não traço dinâmico Android.
SELENE	`strace`, `logcat`, L0–L1 e contexto	Compactação semântica auditável e *target-aware*.

camadas intermediárias que reduzam ruído, organizem sinais comportamentais e permitam retornar à evidência original.

A Figura 1 resume o fluxo de processamento. Em **Artefatos brutos**, temos os arquivos de entrada preservados pelo sistema: `strace`, `logcat` e metadados da execução. A partir desse ponto, o processamento se divide em dois ramos: o ramo superior trata o `strace`; o ramo inferior trata o `logcat`. Ambos convergem em representações intermediárias, que alimentam o índice de evidências e os pacotes de contexto.

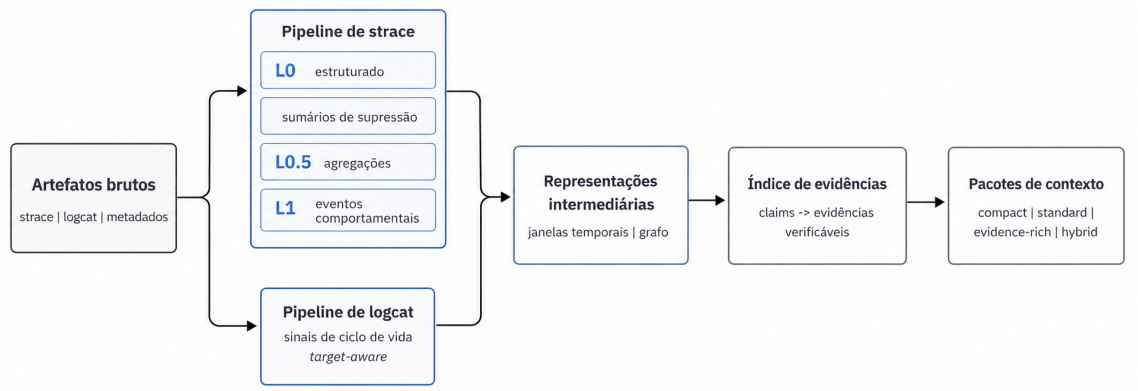

Figura 1. Fluxo do SELENE: `strace` gera L0, sumários de supressão, L0.5 e L1; `logcat` gera sinais de ciclo de vida *target-aware*; ambos alimentam representações intermediárias, índice de evidências e pacotes de contexto.

No **Pipeline de strace**, a camada **L0** normaliza cada *syscall* com tempo, PID, nome da chamada, argumentos, retorno, erro e recurso associado. Em seguida, os **sumários de supressão** agregam padrões recorrentes de infraestrutura que não precisam permanecer como linhas independentes. A camada **L0.5** consolida eventos próximos por recurso, tipo e janela temporal. Por fim, a camada **L1** converte eventos materializados em categorias semânticas, como IPC/Binder, rede, memória executável, acesso a `/proc`, atividade em diretórios privados, checagem de ambiente e sondagem por erro.

No **Pipeline de logcat**, ocorre a extração de sinais de ciclo de vida da execução. Esse processamento é *target-aware*: mensagens globais do sistema são separadas de eventos associados ao aplicativo analisado. Os sinais incluem execução completa, término precoce, *crash* Java, *crash* nativo, ANR, encerramento forçado, LMKD, erro de

inicialização e negações SELinux.

Os eventos L1 e os sinais de ciclo de vida são reunidos em **Representações intermediárias**, compostas por janelas temporais e grafo de execução. Essas representações preservam a relação entre comportamento, recurso observado e evidência de origem. O **Índice de evidências** liga cada *claim* a uma evidência verificável, como evento L1, marcador de ciclo de vida, janela temporal, nó ou aresta do grafo, ou linha do artefato bruto. Por fim, os **Pacotes de contexto** exportam a análise em quatro níveis de detalhe: *compact*, *standard*, *evidence-rich* e *hybrid*. Esses pacotes são pontos de entrada para leitura e comparação, não substitutos do dado bruto.

A Figura 2 resume a navegação entre os artefatos produzidos. A análise pode começar em um pacote de contexto, seguir para um *claim* e consultar o índice de evidências. A partir dele, o analista alcança eventos L1, agregações L0.5 e linhas correspondentes do `strace` bruto. Para sinais de ciclo de vida, a navegação segue do índice para o marcador *target-aware* e para as linhas de `logcat` que sustentam a interpretação.

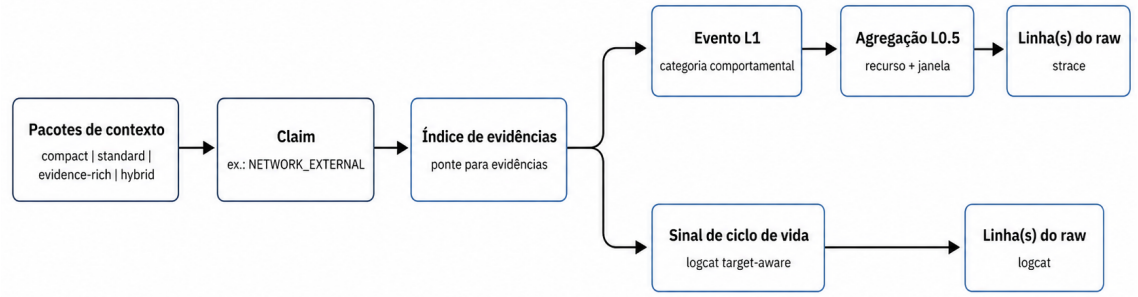

Figura 2. Navegação entre artefatos produzidos. O analista parte de um pacote de contexto, seleciona um *claim*, consulta o índice de evidências e alcança eventos intermediários ou linhas do artefato bruto correspondente.

4. Metodologia

A metodologia foi organizada para separar três decisões experimentais: a definição do corpus, a materialização das camadas do SELENE e a validação da fidelidade semântica e da navegabilidade estrutural. Os artefatos de pesquisa estão disponíveis publicamente.[1]

4.1. Corpus e escopo experimental

Os APKs utilizados neste estudo foram obtidos do AndroZoo, repositório público amplamente usado em pesquisas de segurança Android [Allix et al. 2016]. Como rótulos de antivírus são heterogêneos, utilizamos o Euphony para normalizar detecções e agrupar amostras em famílias de malware [Hurier et al. 2017]. O corpus inicialmente inspecionado contém 111.496 SHA-256 únicos coletados entre 2022 e 2024, além de 12.454 amostras de 2025.

As amostras de 2025 foram usadas apenas como diagnóstico de viabilidade do corpus recente e não integraram o núcleo experimental. Esse recorte apresentou muitos

[1]Código, configurações e instruções
`https://github.com/secretrdlab/SELENE`.
Artefatos de análise gerados disponíveis em:
`https://huggingface.co/datasets/serrooT/artemis-android-dynamic-traces`.

pacotes incompletos ou estruturalmente problemáticos: 7.622 amostras dependiam de *split APKs* ausentes, 123 falharam por incompatibilidade de ABI e apenas 2.874 executaram corretamente no Android 10. Como essas falhas podem refletir incompletude do pacote, e não comportamento do malware, o núcleo experimental foi concentrado nas amostras de 2022 a 2024.

A primeira etapa foi executada no Android 10, adotado como linha de base operacional de triagem. Essa escolha permitiu identificar, com menor custo computacional, amostras executáveis e artefatos válidos para análise comportamental. Após filtros de integridade, remoção de execuções sem traços válidos e congelamento da seleção experimental, 44.485 execuções compõem o conjunto Android 10 analisado neste artigo.

Para comparação horizontal, as amostras executáveis também foram reexecutadas no Android 14. O conjunto Android 14 processado contém 30.751 execuções válidas, das quais 30.746 pareiam por SHA-256 com o Android 10. As comparações entre versões são realizadas apenas sobre essa interseção.

4.2. Coleta dos artefatos

A coleta acopla o `strace` ao PID (identificador de processo) da aplicação após sua inicialização, acompanhando processos e *threads* derivados (`-f`), descritores de arquivo (`-y -yy`), marcas temporais (`-tt`) e argumentos longos (`-s 2048 -v`). Os artefatos brutos são preservados em texto ou comprimidos com `zstd`, mantendo a auditoria posterior. A coleta foi realizada com o ARTEMIS [Júnior et al. 2025].

4.3. Taxonomia comportamental de syscalls

A seleção de chamadas e recursos monitorados segue uma taxonomia comportamental baseada na literatura da Seção 2. A materialização dos eventos **L1** considera chamada, argumentos, descritor, caminho, recurso, retorno, erro e contexto da execução, conforme resumido na Tabela 2.

Tabela 2. Taxonomia comportamental usada para materialização de eventos L1.

Classe	Evidências de baixo nível	Interpretação no L1
IPC/Binder	`ioctl` em `/dev/binder`; `BINDER_WRITE_READ`	Comunicação com serviços Android
Rede externa	TCP/UDP com endpoint externo; `connect`, `sendto`, `recvfrom`	Comunicação remota observável
Rede local	UNIX sockets, loopback, pipes, `eventfd`	Comunicação local ou entre processos
Arquivos privados	`/data/data/<pkg>`, `/data/user/0/<pkg>`	Persistência ou dados locais do app
Procfs	`/proc`, `/proc/self`, `maps`, `cmdline`, `status`	Introspecção de processo e ambiente
Ambiente	`prctl`, propriedades, fingerprint, caminhos de sistema/emulador	Checagem de contexto; possível anti-análise
Memória exec.	`mmap`/`mprotect` com `PROT_EXEC`	JIT ou carga dinâmica
Probing de erro	`ENOENT`, `EACCES`, `EPERM`, falhas repetidas	Sondagem de ambiente ou dependências

TLS, atividade genérica de processo e eventos SELinux foram mantidos como categorias diagnósticas, e não como sinais centrais da métrica principal. Essa separação evita misturar fenômenos de granularidade distinta, pois tais eventos podem refletir bibliotecas, serviços do sistema, infraestrutura do ambiente ou formas diferentes de observação, e não necessariamente comportamento específico do APK alvo.

4.4. Calibração e congelamento do pipeline

O pipeline foi construído de forma incremental para reduzir ajuste oportunista. Pilotos em subconjuntos menores validaram o *parsing* de `strace`, a identificação de recursos, a materialização de eventos e a supressão de ruídos recorrentes. Em seguida, uma seleção reprodutível de 500 execuções validou a integração entre eventos L1, ciclo de vida, grafos e pacotes de contexto.

Antes da escala completa, as correções foram restritas a consistência e classificação: propagação de `package_name`, separação entre sinais globais e sinais do aplicativo-alvo, e refinamento de padrões para reduzir falsos positivos em *crashes* e erros de lançamento. A configuração foi então congelada e aplicada ao conjunto Android 10 completo. Ao final, 11 das 44.485 execuções falharam no *parsing* do `strace` e foram registradas explicitamente, sem descarte silencioso.

4.5. Métricas e validação

A avaliação mede quatro propriedades: redução de volume, interpretação da execução, fidelidade semântica e navegabilidade estrutural. Para redução de volume, usamos duas métricas complementares. A **redução estrutural** (RR_L) mede a fração do traço bruto que deixa de existir como linha independente após a materialização de uma camada L (L0, L0.5 ou L1). A **compressão em bytes** (CR_{bytes}) mede o ganho de armazenamento de um artefato derivado X em relação ao *raw*. As duas métricas capturam efeitos distintos: uma camada pode reduzir muitos registros e ainda ocupar bastante espaço por registro.

$$RR_L = 1 - \frac{N_L}{N_{raw}}, \quad CR_{bytes}(X) = \frac{|raw|}{|X|},$$

em que N_{raw} é o número de linhas do `strace` descomprimido, N_L é o número de registros materializados na camada L, e $|raw|$ e $|X|$ são os tamanhos em bytes do traço bruto e do artefato derivado. Medimos CR_{bytes} nas formas descomprimida e comprimida com `zstd`. O **SELENE** é avaliado primariamente por RR_L, que reflete sua proposta de compactação semântica; CR_{bytes} é reportado como ponto de contato com a literatura de compressão de logs [Liu et al. 2019, Wei et al. 2021, Yu et al. 2026].

A fidelidade da compactação é avaliada por presença de sinais, correção de evidências e resolubilidade de referências, e não por igualdade linha a linha com o traço bruto. Essa distinção é necessária porque os pacotes compactos são intencionalmente sumarizados: eles devem preservar sinais comportamentais relevantes e ao menos uma evidência representativa, não todas as ocorrências repetitivas.

Para evitar validação circular, definimos um oráculo independente sobre o dado bruto, composto por 30 regras conservadoras aplicadas diretamente a `strace` e `logcat`, sem reaproveitar a lógica de materialização do pipeline. A validação semântica foi realizada em 1.000 execuções estratificadas por família de malware, cobrindo as 47 famílias do conjunto. Nessa amostra, as regras produziram 49.345 evidências brutas associadas aos sinais avaliados.

Para cada execução r, sinal s e nível de contexto t, definimos $O_{r,s} = 1$ quando o oráculo encontra ao menos uma evidência bruta do sinal no *run*, e $C_{r,s,t} = 1$ quando o pacote de contexto do nível t contém ocorrência do mesmo sinal, seja como flag binária ou

contagem positiva em `event_type_counts`. A partir desses valores, derivamos $TP_{s,t}$, $FP_{s,t}$ e $FN_{s,t}$ e calculamos:

$$\mathrm{Recall}_{s,t} = \frac{TP_{s,t}}{TP_{s,t} + FN_{s,t}}, \qquad \mathrm{Precision}_{s,t} = \frac{TP_{s,t}}{TP_{s,t} + FP_{s,t}},$$

$$\mathrm{F1}_{s,t} = \frac{2\,\mathrm{Recall}_{s,t}\,\mathrm{Precision}_{s,t}}{\mathrm{Recall}_{s,t} + \mathrm{Precision}_{s,t}}.$$

A métrica principal considera as oito classes comportamentais da Tabela 2 e é agregada como **média aritmética por sinal**, sem ponderação por suporte. Esse agregado evita que sinais comuns ocultem perdas em sinais raros. TLS, *crash* nativo, *kill*, LMKD, atividade genérica de processo e SELinux são reportados separadamente como diagnósticos, por exigirem atribuição ou calibração mais fina.

Também avaliamos se a hierarquia permite navegação verificável. Um *claim* é uma afirmação comportamental apresentada ao analista, como presença de comunicação externa, acesso a `/proc`, mapeamento de memória executável ou ANR associado ao alvo. Para cada *claim*, verificamos se há referência de evidência, número de linha, tipo de evento, correspondência com evento L1 ou evidência de ciclo de vida, referência sintaticamente válida ao artefato bruto e consistência de pacote.

A taxa de resolução estrutural é definida como:

$$StructuralResolution_t = \frac{|\{c \in \mathcal{C}_t : \mathrm{resolved}(c) = 1\}|}{|\mathcal{C}_t|},$$

em que $\mathcal{C}_t$ é o conjunto de *claims* emitidos pelo contexto t e $\mathrm{resolved}(c)$ indica que o *claim* possui ponte estrutural para evidência do mesmo tipo. Essa validação foi aplicada a todos os pacotes de contexto gerados no conjunto Android 10. Execuções sem qualquer *claim* foram contabilizadas separadamente e não tratadas como falha de navegação.

Por fim, a comparação Android 10/14 foi conduzida apenas sobre a interseção pareada de 30.746 APKs com execuções processadas nos dois ambientes. Para sinais binários pareados, usamos o teste de McNemar com correção de continuidade sobre os pares discordantes, em que b é o número de APKs com o sinal presente apenas no Android 10 e c apenas no Android 14, com estatística $\chi^2 = (|b-c|-1)^2/(b+c)$ e valor-p associado. Dado o grande tamanho amostral, a interpretação enfatiza direção e magnitude da mudança, não apenas a significância.

5. Resultados e Discussão

Esta seção apresenta os resultados do **SELENE** no conjunto Android 10 congelado, com 44.485 execuções, e na interseção pareada Android 10/14, com 30.746 APKs. A avaliação combina resultados de escala, validação semântica contra o dado bruto, validação estrutural de navegabilidade, estudo de caso e análise das ameaças à validade.

5.1. Custo dos traços crus e redução por camada

O conjunto Android 10 totalizou 9.584.296.522 linhas de `strace`, com mediana de 101.144 linhas por execução e P95 de 759.971 linhas. Em armazenamento, os artefatos

brutos medidos ocuparam aproximadamente 499,77 GiB comprimidos e 3.032,64 GiB descomprimidos, equivalentes a uma razão $\mathrm{CR}_{\mathrm{bytes}} \approx 6,07\times$ no agregado. Como mostra a Tabela 3, o `strace` foi o artefato dominante, com 427,63 GiB comprimidos por zstd e 2.283,98 GiB descomprimidos ($CR_{bytes} \approx 5{,}34\times$), enquanto os logs de dispositivo apresentaram a maior taxa por arquivo ($\approx 11{,}1\times$).

Tabela 3. Custo de armazenamento dos principais artefatos crus no Android 10. CR_{bytes} é a razão descomprimido/comprimido por artefato.

Artefato	Comprimido	Descomprimido	CR_{bytes}
`strace`	427,63 GiB	2.283,98 GiB	5,34×
Logs de dispositivo	66,69 GiB	743,21 GiB	11,14×
`analysis.log`	2,58 GiB	2,58 GiB	1,00×
`metadata.json`	1,22 GiB	1,22 GiB	1,00×
Memória, proc. e prop.	1,65 GiB	1,65 GiB	1,00×
Total medido	499,77 GiB	3.032,64 GiB	6,07×

Após a estruturação e compactação semântica, a camada L0 materializada preservou 2.121.279.961 registros (22,1% do raw), os sumários de supressão agregaram 3.984.391.098 (41,6%) e as 3.478.625.463 linhas restantes (36,3%) correspondem a syscalls fora do conjunto de interesse, anotações sintáticas do `strace` (linhas `<unfinished/resumed>` não pareadas, cabeçalhos, marcadores) e linhas com falha de parsing pontual, descartadas pelo filtro de materialização sem agregação. A camada L1 produziu 73.673.283 eventos comportamentais. A Tabela 4 mostra a contabilidade completa por camada, com a coluna "Proporção do raw" correspondendo a N_L/N_{raw} (de forma que $RR_L = 1 - N_L/N_{raw}$). Os eventos L1 representam 0,77% do volume bruto, equivalente a $RR_{L1} \approx 99{,}23\%$, sem remover a raiz auditável dos artefatos originais.

Tabela 4. Total de linhas do `strace` no Android 10: as três primeiras categorias particionam o raw; as demais representam camadas semânticas derivadas.

Camada	Total	Mediana/run	Proporção do raw
Raw `strace`	9.584.296.522	101.144	1,000
L0 materializado	2.121.279.961	24.484,5	0,221
L0 suprimido (agregado)	3.984.391.098	34.664,5	0,416
Descartado (fora do escopo)	3.478.625.463	–	0,363
L0.5 eventos	73.635.326	1.073	0,00768
L1 eventos	73.673.283	1.044	0,00769

A redução é explicada, em parte, pela concentração de ruídos recorrentes. Dentro da linha "L0 suprimido", dois padrões dominam: operações associadas a `/dev/goldfish_pipe` (3.652.080.999 linhas, ligadas à infraestrutura do emulador) e leituras repetitivas de DEX/APK (332.310.099 linhas); juntos correspondem à totalidade do agregado de supressão. Esses eventos não são descartados silenciosamente: são agregados em sumários para preservar o custo e a recorrência do fenômeno, sem dominar a primeira camada de interpretação.

A compactação se traduz também em ganho de armazenamento. A Tabela 5 reporta, para cada camada e para os pacotes de contexto, o tamanho em disco do artefato derivado

e o CR_{bytes} medido contra o `strace` bruto descomprimido (2.283,98 GiB). Os *parquets* de L0, L0.5 e L1 reduzem o volume em $40\times$, $378\times$ e $447\times$, respectivamente, e os pacotes *compact* chegam a $8.817\times$ a menos que o `strace` de entrada, ainda preservando ponteiros para o dado original. Os grafos completos por execução não entram nesse contraste por incluírem relações temporais explícitas em volume maior; na avaliação de leitura, eles são consumidos por meio dos pacotes *graph* e *hybrid*.

Tabela 5. Tamanho em disco e CR_{bytes} dos artefatos derivados (Android 10), medidos contra o `strace` bruto descomprimido (2.283,98 GiB).

Artefato derivado	Tamanho	CR_{bytes}
L0 (*parquet*, todas as partições)	57,39 GiB	39,8×
L0.5 (*parquet*)	6,05 GiB	377,7×
L1 (*parquet*)	5,11 GiB	446,8×
Pacote *compact* (JSONL)	0,26 GiB	8.817,4×
Pacote *graph* (JSONL)	0,47 GiB	4.833,0×
Pacote *hybrid* (JSONL)	0,77 GiB	2.974,3×
Pacote *standard* (JSONL)	0,95 GiB	2.400,5×
Pacote *evidence-rich* (JSONL)	3,13 GiB	730,1×

A redução obtida deve ser lida como compactação semântica e não como substituição do traço bruto: a camada L1 não preserva contagem exata nem ordem completa de todas as chamadas, mas mantém sinais comportamentais, evidências representativas e ponte para o dado original. L0 preserva estrutura próxima ao *raw*, L0.5 reduz redundância local e L1 converte eventos em categorias semânticas navegáveis.

No mesmo conjunto, a baseline `zstd` reduz o `strace` em $5{,}34\times$ sem fornecer navegação; o *compact* ocupa $8.817\times$ menos e referencia o *raw*. O custo total do SELENE inclui o *raw* comprimido e os derivados. LogZip, LogReducer e DeLog não foram executados porque seus formatos e conjuntos não permitem comparação direta com `strace`, limitação desta avaliação.

5.2. Ciclo de vida das execuções

O `logcat` complementa o `strace` ao explicar estados não inferíveis apenas por chamadas de sistema, como ANR (aplicação sem resposta) e LMKD (encerramento por baixa memória). No conjunto de 44.485 execuções, 20.758 terminaram antes do esperado, 44.163 observaram o PID alvo e 33.618 mantiveram o processo vivo ao fim. A Tabela 6 resume os marcadores *target-aware*.

Esses resultados mostram que execuções curtas ou pouco informativas no `strace` não devem ser interpretadas isoladamente. Elas podem refletir falhas de inicialização, ANRs, encerramentos, incompatibilidades, ausência de estímulo suficiente, possível evasão ou comportamento que exige novos experimentos. Por esse motivo, sinais de ciclo de vida são mantidos separados entre globais e associados ao alvo: tratá-los como evidências operacionais, e não como prova automática de evasão, é a leitura coerente com a separação *target-aware* adotada.

5.3. Custo de leitura dos contextos

Os pacotes de contexto reduzem o custo de leitura sem substituir o traço bruto. Como mostra a Tabela 7, o nível *compact* apresentou mediana de 1.587 *tokens*, enquanto o *hybrid*

Tabela 6. Resumo de ciclo de vida no conjunto Android 10 (44.485 execuções; marcadores *target-aware*).

Métrica	Valor	%	Métrica	Valor	%
Término precoce	20.758	46,66	*Crash* nativo	2.077	4,67
Término m. precoce	380	0,85	ANR	6.164	13,86
PID alvo observado	44.163	99,28	LMKD	1.936	4,35
Processo vivo ao fim	33.618	75,57	*Force-stop*/kill	16.585	37,28
Não observado ao fim	10.773	24,22	Erro de inicialização	3.927	8,83
Crash Java	10.290	23,13	Precoce desconhecido	10.411	23,40

apresentou mediana de 4.824 *tokens*, abaixo do *standard* em 97,05% das execuções. O *evidence-rich* preserva mais evidências, mas com custo mediano maior. Na tabela, *graph* e *hybrid* usam seus próprios pacotes; os demais usam os resumos textuais.

Tabela 7. Custo de leitura dos pacotes de contexto no Android 10.

Contexto	Mediana tokens	P95 tokens	Retenção binária
compact	1.587	1.673	1,000
graph	2.930	3.105	0,887
hybrid	4.824	4.983	1,000
standard	5.936	6.149	1,000
evidence-rich	19.988	20.969	1,000

A principal implicação é que o analista pode iniciar pela representação compacta e abrir o bruto apenas quando necessário. O *graph* adiciona relações temporais e de recursos; o *hybrid* as combina ao *compact* com custo menor que o *standard* na maioria dos casos. O fluxo pode apoiar triagem, perícia e detectores, mas ganhos humanos ou de acurácia não foram medidos.

5.4. Fidelidade semântica e navegabilidade

A fidelidade semântica foi avaliada em uma amostra estratificada de 1.000 execuções usando um oráculo independente sobre `strace` e `logcat` (Seção 4). A métrica principal considera oito sinais centrais derivados de chamadas de sistema, agregados pela média aritmética por sinal. No nível *compact*, esses sinais apresentaram *recall* médio de 0,990, precisão média de 0,978 e F1 médio de 0,983. A Tabela 8 detalha os valores por sinal.

Tabela 8. Fidelidade semântica dos sinais centrais no nível *compact*.

Sinal	Recall	Precision	F1
Binder IPC	1,000	1,000	1,000
Rede externa	1,000	0,988	0,994
Rede local	0,934	1,000	0,966
Memória executável	1,000	1,000	1,000
Acesso a `/proc`	0,991	1,000	0,995
Arquivos privados do app	0,997	0,969	0,982
Checagem de ambiente	0,998	1,000	0,999
Sondagem por erro	1,000	0,864	0,927

O menor valor de precisão ocorreu em sondagem por erro (0,864), efeito de uma regra conservadora que prefere reter padrões de *probing* a descartá-los. Como esse sinal é usado para triagem e não como evidência conclusiva isolada, o resultado é aceitável para o objetivo da camada *compact*. Os valores iguais a 1,000 em alguns sinais refletem sua natureza sintática e diretamente observável no `strace`, como IPC/Binder, memória executável e acesso a recursos bem definidos. Para reduzir validação circular, o oráculo e a lógica L0→L1 foram implementados de forma independente, sem reuso de código.

Categorias como TLS, *crash* nativo, *kill*, LMKD, atividade genérica de processo e SELinux foram reportadas separadamente como diagnósticos. Essa decisão evita misturar sinais centrais, derivados de evidências estáveis, com categorias cuja atribuição ao alvo pode depender de contexto adicional ou de mensagens globais do `logcat`.

A navegabilidade estrutural foi avaliada nos 44.485 pacotes de contexto por nível. Para os oito sinais centrais, todos os 279.179 *claims* por nível apresentaram resolução estrutural, correspondência de tipo e consistência de pacote iguais a 1,000 nos quatro níveis de contexto (*compact*, *standard*, *evidence-rich* e *hybrid*). Para todos os *claims* primários, incluindo diagnósticos e ciclo de vida, a resolução estrutural variou entre 0,948 e 0,951, com correspondência de tipo e consistência de pacote iguais a 1,000.

Os casos não resolvidos concentram-se em *claims* diagnósticos, como TLS e SELinux, que não exigem evidência do mesmo tipo. Assim, a validação semântica mede se os sinais centrais preservam comportamento relevante frente ao oráculo independente, enquanto a validação estrutural mede se esses sinais podem ser navegados em escala até evidências verificáveis.

5.5. Estudo de caso: navegação até o raw

O estudo de caso instancia o fluxo da Figura 2 em uma execução real do conjunto Android 10. A amostra pertence à família `gustuff` (SHA-256 `4d4f8599...e8292656`) e contém um *claim* `NETWORK_EXTERNAL`. Como mostra a Tabela 9, a navegação parte do pacote *compact*, segue pelo índice de evidências até o evento L1 e sua agregação L0.5, e termina nas linhas do `strace` bruto que sustentam a afirmação.

Tabela 9. Navegação para um *claim* `NETWORK_EXTERNAL` da amostra `gustuff`.

Camada	Evidência navegada
compact	`NETWORK_EXTERNAL`; recurso `TCP:54.156.36.158:443`; linhas `14539`, `14542`, `14547`, `17169` e `17173`.
L1	Evento comportamental de rede externa associado ao mesmo recurso e ao identificador L0.5 `..._00000219`.
L0.5	Agregação `NETWORK_IO` no recurso `TCP:54.156.36.158:443`, com 13 chamadas entre $t = 76944{,}02$ s e $t = 76944{,}74$ s.
L0 / *raw*	Linhas do `strace` confirmam socket TCP externo na porta 443, com chamadas `getsockopt`, `recvfrom` e `sendto`; o conteúdo enviado inclui prefixos compatíveis com tráfego TLS.

Nesse exemplo, a confirmação do canal externo exige a leitura de cinco linhas selecionadas, em vez das 244.669 linhas do `strace` bruto do *run*. O caso ilustra a função do SELENE: reduzir o caminho de auditoria sem remover o artefato original da verificação.

5.6. Comparação pareada Android 10 e Android 14

O conjunto Android 14 contém 30.751 execuções; 30.746 pareiam por SHA-256 com o Android 10, e as 5 amostras exclusivas do Android 14 foram excluídas da comparação

pareada. O conjunto completo Android 10, com 44.485 execuções, é usado para resultados de escala; a comparação entre versões usa apenas os pares por SHA-256, evitando viés por diferença de cobertura.

Na interseção pareada, o custo total comprimido dos principais artefatos crus permaneceu praticamente estável: 192,88 GiB no Android 10 e 192,23 GiB no Android 14. Entretanto, a composição do custo mudou, como mostra a Tabela 10. O `strace` diminuiu no Android 14, passando de 164,82 GiB para 146,60 GiB comprimidos e de 1.172,87 GiB para 902,96 GiB descomprimidos. Em contrapartida, os logs de dispositivo cresceram de 26,21 GiB para 42,73 GiB comprimidos e de 279,70 GiB para 462,07 GiB descomprimidos. Assim, a versão do sistema não altera apenas o comportamento observado, mas também o perfil operacional dos artefatos produzidos.

Tabela 10. Razão Android 14/Android 10 dos artefatos crus pareados.

Artefato	Razão comp.	Razão descomp.
`strace`	0,89×	0,77×
Logs de dispositivo	1,63×	1,65×
`analysis.log`	1,00×	1,00×
`metadata.json`	0,85×	0,85×
Memória, proc. e prop.	1,96×	1,96×
Total medido	1,00×	0,94×

Os rótulos de ciclo de vida também mudaram entre versões. Apenas 13.260 dos 30.746 pares, ou 43,13%, mantiveram o mesmo `lifecycle_summary_label`. O Android 14 apresentou aumento de término precoce em relação ao Android 10, de 51,10% para 67,14%, embora o término muito precoce tenha diminuído de 0,97% para 0,64%. Isso indica que a diferença entre versões não se reduz a falhas imediatas de execução, mas altera o perfil temporal observado. Na Tabela 11, Δ é a diferença em pontos percentuais, e "Discord." indica a fração de pares em que o sinal mudou entre versões.

Tabela 11. Mudanças pareadas de sinais comportamentais entre Android 10 e 14.

Sinal	A10	A14	Δ p.p.	Discord.
`BINDER_IPC`	97,03%	99,92%	+2,89	3,03%
`NETWORK_EXTERNAL`	36,66%	41,29%	+4,63	22,44%
`NETWORK_LOCAL`	89,96%	93,64%	+3,68	8,44%
`MEMORY_EXEC`	31,71%	50,04%	+18,33	32,67%
`PROC_ACCESS`	87,47%	78,69%	-8,78	22,98%
`APP_PRIVATE_FILE`	94,42%	96,40%	+1,98	7,12%
`ENV_CHECK`	96,64%	98,52%	+1,88	4,64%
`ERROR_PROBING`	96,87%	99,08%	+2,21	3,95%

Aplicando o teste de McNemar descrito na Seção 4, as três maiores mudanças foram estatisticamente significativas ($p < 0{,}001$). Em `MEMORY_EXEC`, 7.841 APKs passaram a exibir o sinal apenas no Android 14 contra 2.205 apenas no Android 10 ($\chi^2 \approx 3.161$). Em `PROC_ACCESS`, a direção se inverte: 4.882 APKs perderam o sinal no Android 14 contra 2.183 que passaram a exibi-lo ($\chi^2 \approx 1.030$). Em `NETWORK_EXTERNAL`, 4.162 APKs exibiram o sinal apenas no Android 14 contra 2.738 apenas no Android 10 ($\chi^2 \approx 294$).

Esses resultados reforçam que a versão do Android altera a superfície comportamental observada para o mesmo conjunto de APKs. A comparação pareada ajuda a

distinguir sinais estáveis, variações de *runtime* e limitações da instrumentação, justificando a análise horizontal multiambiente.

5.7. Ameaças à validade

Validade interna. O acoplamento do `strace` ao PID pode perder eventos iniciais e introduzir sobrecarga; mensagens `logcat` sem identificador permanecem ambíguas. Mitigamos esses riscos com calibração em 500 execuções, congelamento do pipeline e registro das 11 falhas de *parsing*.

Validade de construto. A compactação *lossy* preserva sinais e referências, não contagem ou ordem completas. Um oráculo independente sobre 1.000 execuções reduz validação circular; categorias ambíguas (TLS, *crash*, *kill*, LMKD, processo e SELinux) ficam fora da métrica central.

Validade externa. Resultados variam com versão, emulador, SELinux, ABI e entrada; ausência de sinal significa apenas ausência de observação. A comparação Android 10/14 reduz essa limitação, mas versões posteriores, com novos caminhos e restrições, exigem recalibração.

Custo computacional. Medimos armazenamento e leitura, mas não tempo, CPU e pico de memória comparáveis por estágio; esses custos devem ser medidos antes de concluir sobre vazão operacional.

Validade estatística e reprodutibilidade. Estudos de malware estão sujeitos a viés temporal, vazamento entre conjuntos, deriva de famílias e dependência de rótulos de antivírus [Pendlebury et al. 2018, Botacin et al. 2021, Miranda et al. 2022]. Para mitigar esses riscos, registramos o período, os critérios de seleção, as falhas de execução e a composição dos conjuntos; além disso, congelamos a seleção experimental e reportamos as métricas por escopo.

6. Conclusão

Este artigo apresentou o **SELENE**, uma camada pós-coleta que transforma artefatos extensos de análise dinâmica Android em uma hierarquia compacta e auditável de evidências com proveniência. A proposta preserva o traço bruto como raiz de verificação, mas desloca a leitura, a comparação e a triagem para representações mais manejáveis.

Em 44.485 execuções no Android 10, os eventos L1 corresponderam a cerca de 0,77% das linhas de `strace`. Em uma amostra estratificada de 1.000 execuções, os oito sinais centrais atingiram F1 médio de 0,983 frente a um oráculo independente, mantendo ligações verificáveis entre *claims* e evidências. A comparação de 30.746 APKs no Android 10 e 14 mostrou ainda que a versão do sistema altera significativamente a superfície comportamental observável e o perfil de armazenamento.

O SELENE não substitui o traço bruto, não é uma nova *sandbox* e não classifica malware. Sua contribuição é uma camada intermediária para compactar, organizar e verificar evidências em tarefas de triagem, análise forense, resposta a incidentes e detecção. Código, configurações e dados derivados estão disponíveis nos repositórios indicados na Seção 4. Trabalhos futuros ampliarão as versões e linhas de base avaliadas, medirão os custos computacionais e investigarão o impacto sobre analistas e detectores.

Referências

Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA. ACM.

Bhat, P., Behal, S., and Dutta, K. (2023). A system call-based android malware detection approach with homogeneous & heterogeneous ensemble machine learning. *Computers & Security*, 130:103277.

Botacin, M., Ceschin, F., Sun, R., Oliveira, D., and Gregio, A. (2021). Challenges and pitfalls in malware research. *Computers & Security*, 106:102287.

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (2011). Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 15–26. ACM.

Canfora, G., Mercaldo, F., Visaggio, C. A., Martinelli, F., and Santone, D. (2015). Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20. ACM.

Hassan, W. U., Noureddine, M. A., Datta, P., and Bates, A. (2020). Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

He, P., Zhu, J., Zheng, Z., and Lyu, M. R. (2017). Drain: An online log parsing approach with fixed depth tree. In *Proceedings of the IEEE International Conference on Web Services*, pages 33–40.

Hurier, M., Suarez-Tangil, G., Dash, S. K., Bissyandé, T. F., Traon, Y. L., Klein, J., and Cavallaro, L. (2017). Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 425–435. IEEE Press.

Júnior, C. T., Filho, D. F., Pincovscy, J., and Grégio, A. (2025). Artemis: Uma plataforma modular para execução, monitoração e investigação de aplicativos android suspeitos. In *Anais do XXV Simpósio Brasileiro de Cibersegurança*, pages 147–162, Porto Alegre, RS, Brasil. SBC.

Kumar, S., Mishra, D., Panda, B., and Shukla, S. K. (2023). Inviseal: A stealthy dynamic analysis framework for android systems. *ACM Transactions on Privacy and Security*, 26(3).

Li, T., Liu, X., Qiao, W., Zhu, X., Shen, Y., and Ma, J. (2024). T-trace: Efficiently constructing attack provenance graphs from syslogs and firewall logs against advanced persistent threats. *IEEE Transactions on Dependable and Secure Computing*, 21(3):1179–1195.

Liu, J., Zhu, J., He, S., He, P., Zheng, Z., and Lyu, M. R. (2019). Logzip: Extracting hidden structures via iterative clustering for log compression. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM.

Malik, S., Singh, N., and Tripathy, S. (2026). Semantic characterization of android malware through runtime system call analysis. *Journal of Information Security and Applications*, 98:104406.

Miranda, T. C., Gimenez, P.-F., Lalande, J.-F., Tong, V. V. T., and Wilke, P. (2022). Debiasing android malware datasets: How can i trust your results if your dataset is biased? *IEEE Transactions on Information Forensics and Security*, 17:2182–2197.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., and Cavallaro, L. (2018). Tesseract: Eliminating experimental bias in malware classification across space and time. *arXiv preprint arXiv:1807.07838*.

Quark Engine Project (2026). Quark-engine: Obfuscation-neglect android malware scoring system. `https://quark-engine.readthedocs.io/`. Documentação oficial; ferramenta de análise estática e mapeamento de comportamentos em nível de API/Dalvik.

Suo, D., Xue, L., Huang, W., Tan, R., and Sun, G. (2025). Assessing the capability of android dynamic analysis tools to combat anti-runtime analysis techniques. *arXiv preprint arXiv:2512.12551*.

Sutter, T., Kehrer, T., Rennhard, M., Tellenbach, B., and Klein, J. (2024). Dynamic security analysis on android: A systematic literature review. *IEEE Access*, 12:57261–57287.

Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Veuskens, F., Cassee, N., and Demeyer, S. (2023). Valb: Variable-aware log benchmarking to evaluate log parsers for generated and real logs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

Vyšniūnas, T., Čeponis, D., Goranin, N., and Čenys, A. (2024). Risk-based system-call sequence grouping method for malware intrusion detection. *Electronics*, 13(1):206.

Wei, J., Zhang, G., Wang, Y., Liu, Z., Zhu, Z., Chen, J., Sun, T., and Zhou, Q. (2021). On the feasibility of parser-based log compression in large-scale cloud systems. In *19th USENIX Conference on File and Storage Technologies*, pages 249–264. USENIX Association.

Yan, L.-K. and Yin, H. (2012). Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, pages 569–584.

Yu, S., Wu, Y., Xu, J., Fu, Y., Wang, N., Liu, M., Jiang, P., Zhang, X., Jia, T., He, P., and Li, Y. (2026). Delog: An efficient log compression framework with pattern signature synthesis. *arXiv preprint arXiv:2601.15084*.