ARTEMIS: Uma Plataforma Modular para Execução, Monitoração e Investigação de Aplicativos Android Suspeitos

Cláudio Torres Júnior[1], Dario Fernandes Filho[1], João Pincovscy[2], 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)
Brasília – DF – Brasil

{ctjunior, dsffilho, gregio}@inf.ufpr.br, pincovscy@cepesc.gov.br

Resumo. *A fragmentação de versões do Android e as limitações de ferramentas atuais para monitoração efetiva da execução de APKs dificultam a análise de malware. Neste artigo apresenta-se ARTEMIS, uma plataforma baseada em arquitetura de microsserviços capaz de orquestrar análises paralelas em instâncias heterogêneas, testada com 100 emuladores (Android 10–14) e dispositivos físicos. Em estudo de caso com 12.466 APKs maliciosas, ARTEMIS alcançou taxa de instalação de 98,7% (arquitetura adaptativa) e recuperação de 80,2% dos APKs com falha por detecção de depuração (pipeline modular). ARTEMIS oferece análises em larga escala, histórico de execuções e estratégias anti-evasão, essenciais para combater ameaças móveis modernas.*
Palavras-chave*: segurança móvel, análise dinâmica, malware Android, microsserviços, instrumentação multi-versão.*

Abstract. *The fragmentation of Android versions and the limitations of current tools for effectively monitoring APK execution make malware analysis difficult. This paper presents ARTEMIS, a microservices-based platform capable of orchestrating parallel analysis across heterogeneous instances, tested with 100 emulators (Android 10–14) and physical devices. In a case study with 12,466 malicious APKs, ARTEMIS achieved a 98.7% installation rate (adaptive architecture) and 80.2% recovery of failed APKs through debug detection (modular pipeline). ARTEMIS provides large-scale analysis, execution history, and anti-evasion strategies, essential for combating modern mobile threats.*
Keywords*: .*

1. Introdução

O sistema operacional Android detém, há vários anos, a liderança no mercado global de *smartphones*, com cerca de 70% de participação mundial [GlobalStats 2024]. Essa popularidade faz da plataforma um alvo preferencial de cibercriminosos, que exploram vulnerabilidades conhecidas e técnicas de evasão cada vez mais sofisticadas para distribuir *malware* móvel [Lab 2024, Research 2024a]. Um dos principais obstáculos à análise de *malware* Android é a intensa *fragmentação* do ecossistema: apesar dos ciclos regulares de atualização promovidos pelo *Google*, grande parte dos dispositivos permanece operando em versões sem suporte oficial (e.g., Android 10–12), expondo-se assim à falhas antigas

por longos períodos [Developers 2024, Statista 2024]. Essa diversidade de níveis de API, arquiteturas de CPU e *forks* de fabricantes compromete a representatividade e a cobertura das plataformas tradicionais para análise dinâmica de *malware* Android.

Abordagens de análise dinâmica como *CopperDroid* [Tam et al. 2015], *CuckooDroid* [Revivo et al. 2015] e *MobSF* [Zorz 2016] apresentam limitações críticas, tais como cobertura restrita a poucas versões do Android, arquiteturas monolíticas que suportam baixo paralelismo e exigem recompilação para adicionar novos *tracers* ou instrumentadores, proteção insuficiente contra mecanismos anti-análise que resultam em aborto precoce de execuções e ausência de histórico cumulativo de execuções, dificultando comparações longitudinais e triagem reversa de erros. Para superar tais desafios, este artigo apresenta ***ARTEMIS (Android Runtime Tracing, Execution and Malware Investigation System)***: um sistema que *não é uma sandbox*, mas uma plataforma de orquestração inteligente de *sandboxes*, emuladores e ferramentas de análise em diversos níveis. Para tanto, ARTEMIS foi idealizada por meio de uma arquitetura baseada em microsserviços, com agendamento assíncrono de tarefas, configuração declarativa facilitada, suporte multiversão (Android 10 ao 14 e dispositivos físicos), e um *pipeline* modular de instrumentação (e.g., Frida, `strace`, `ftrace`, `tcpdump`, *Monkey*, etc.) Com isso, **ARTEMIS** permite coordenar centenas de instâncias de análise em paralelo, escalando conforme a infraestrutura disponível e garantindo alta cobertura comportamental e resistência a evasão.

Com o objetivo principal de ser um bloco básico na construção de sistemas de análise de aplicações móveis, as contribuições da plataforma ARTEMIS são as seguintes: (i) provisão de um *pipeline* totalmente declarativo em YAML para definir *tracers*, estímulos de UI e parâmetros de emulador sem recompilar o sistema, com base em arquitetura de microsserviços capaz de orquestrar análises simultâneas em múltiplas versões do Android (10–14) e em dispositivos reais, escalando conforme a infraestrutura disponível e mecanismos adaptativos de *fallback* — por exemplo, alternância automática entre emulador `x86_64` e dispositivos ARM64 ao detectar erro de ABI (*Application Binary Interface*); (ii) armazenamento e visualização de um histórico cumulativo das execuções de artefatos na plataforma, com registro completo destes (*logs*, PCAP, capturas de tela, metadados) para comparações longitudinais e triagem reversa de erros; (iii) suporte a múltiplas técnicas de monitoramento, tais como ***Virtual Machine Introspection (VMI)***, para monitoramento externo ao sistema convidado diretamente no *hipervisor* para coleta de chamadas de sistemas (*syscalls*), **rastreamento interno do *kernel* Android**, usando pontos de instrumentação nativos para capturar funções e eventos do *kernel*, ***Hooks in-guest*** via Frida e `strace` com injeção retardada para mitigar detecções baseadas em depuração (*anti-debug*).

2. Literatura e Tecnologias

Nesta seção, as principais iniciativas de análise dinâmica de *malware* para Android são revistas cronologicamente sob a ótica de sua evolução técnica. Além disso, as tecnologias por trás dos conceitos que fundamentam o projeto da plataforma **ARTEMIS** são apresentadas, agrupados em cinco pilares: técnicas de instrumentação, coleta de artefatos comportamentais, simulação de entrada, estratégias anti-evasão e suporte multi-versão.

2.1. Trabalhos Relacionados

Primeiros esforços (2010–2013). Em 2010, o *AASandbox* inaugurou a análise comportamental de APKs [Bläsing et al. 2010], enquanto o *TaintDroid* introduziu rastreamento de fluxo de informação em tempo real na VM Dalvik [Enck et al. 2010]. Em 2012, o *DroidScope* combinou introspecção de máquina virtual (VMI) para traçar código nativo e *Dalvik* simultaneamente [Yan and Yin 2012], e o *DroidBox* uniu o *TaintDroid* a ganchos no *framework* Java para registrar vazamento de dados, SMS e rede [Lantz 2012].

Avanços (2014–2018). *Andrubis* (2014) integrou *sandboxing*, *TaintDroid* e `tcpdump` em um emulador Android 4.2 [Fratantonio et al. 2014]. Já *ANANAS* propôs uma arquitetura modular com monitor de *syscalls* no *kernel* [Neuner et al. 2014]. Por fim, *CopperDroid* (2015) aperfeiçoou a VMI para reconstruir operações de arquivo, IPC e rede em Android 2.3 [Tam et al. 2015].

Soluções *Open Source* e Industriais (2015–2022). O *CuckooDroid* estendeu o Cuckoo Sandbox para APKs, usando *Xposed* em Android 4.1.2 [Revivo et al. 2015]. O *MobSF* (2016–) adicionou Frida e *mitmproxy*, suportando Android 4.1–12 em emuladores e dispositivos reais [Zorz 2016]. O *BareDroid* avaliou amostras em *hardware* físico, reduzindo artefatos de emulação [Mutti et al. 2015]. Embora pioneiras, essas ferramentas exigem automação manual de *inputs* ou infraestrutura física onerosa; o **ARTEMIS** combina cobertura multi-versão, orquestração declarativa em YAML e *fallback* inteligente, habilitando escalabilidade horizontal sem grandes fricções.

Plataformas Recentes (2018–2025). Serviços comerciais como o *VirusTotal Zenbox* (instrumentação Frida no Android 13) e o *Joe Sandbox Mobile* fornecem relatórios detalhados e mapeamento MITRE ATT&CK [(VirusTotal) 2023, LLC 2024]. Provedores como *ANY.RUN Android* e *Hatching Triage* oferecem ambientes ARM interativos para análise em tempo real [ANY.RUN 2025, Hatching 2024]. Novas iniciativas, como o *DroidHook* e o *DroidDungeon*, focam em anti-evasão e suporte multi-versão [ResearchGate 2023, Research 2024b]. Entretanto, a maior parte é fechada ou limita a customização, enquanto **ARTEMIS** provê um ecossistema extensível e *open-source*.

Apesar dos avanços ao longo do tempo, as soluções existentes para análise de *malware* Android geralmente apresentam limitações persistentes: (i) **cobertura de versões restrita:** suporte apenas a *releases* antigas ou recentes, deixando lacunas em sistemas legados [Tam et al. 2015, Revivo et al. 2015]; (ii) **pipelines pouco flexíveis:** alterações de instrumentação requerem modificação no código-fonte [Zhou 2020]; (iii) **sobrecarga e detecção:** instrumentação pesada (Xposed, múltiplos tracers) aumenta CPU/RAM e expõe a *sandbox* [Fratantonio et al. 2014, Intelligence 2023]; (iv) **automação manual de *inputs*:** necessidade de *scripts* específicos ou intervenção humana para estímulo de UI [Zorz 2016]; (v) **tratamento de erros limitado:** falta de políticas robustas de *retry* e categorização de falhas [Fratantonio et al. 2014, Research 2021]; (vi) **ausência de histórico longitudinal:** resultados sobrescritos impedem comparações entre execuções; (vii) **customização restrita:** difícil integração de novos *tracers* ou técnicas de evasão. A Tabela 1 compara a proposta com outras soluções, partindo do ARTEMIS em sua forma mais "crua" (apenas o *Android Emulator* do repositório, atuando como *sandbox* padrão). Embora usem a mesma base técnica, o ARTEMIS permite reconfigurar escalabilidade, *tracing* e instrumentação, ao contrário de *sandboxes* com *pipelines* fixos. A comparação é entre "*sandbox* + orquestração flexível" e "*sandbox* isolada". Orquestradores genéricos

como *CyberChef* ou *AssemblyLine* foram excluídos por não suportarem Android nativamente, o que deslocaria o foco para esforço de integração, e não para funcionalidade pronta para uso.

Tabela 1. Comparação teórica de ferramentas de análise dinâmica de malware Android

Ferramenta	Escalab.	Config.	Histórico	Versões	Inputs	Sobrec.	Extens.	Erros	Anti-ev.
AASandbox	Baixa (1)	*Kernel* fixo	Logs básicos	≤2.3	Monkey500	Alta	Baixa	Abortam	Fraca
TaintDroid	Baixa (1)	ROM Dalvik	Logcat	≤4.3	Externo	Alta	Baixa	Abortam	Fraca
DroidScope	Baixa (1)	Plugins VMI	Brutos	2.3	Script básico	Alta	Média	Abortam	Médio
DroidBox	Baixa (1)	Script emulador	JSON simples	4.1–4.x	Monkey+SMS	Alta	Baixa	1 execução	Fraca
Andrubis	Alta (cluster)	Auto padronizado	DB	4.2	Monkey+sist.	Alta	Média	Retry→próximo	Médio
ANANAS	Baixa (local)	Módulos selecionáveis	Logs modulares	2.3–4.x	Monkey modular	Var.	Alta	Abortam	Médio
CopperDroid	Média (manual)	QEMU modificado	Relat. comport.	4.1	Script	Mod.	Baixa	Pouco output	Médio
CuckooDroid	Alta (10–15)	Perfis/disp.	DB	4.x	Monkey+auto	Mod.	Alta	Retry init	Moderado
MobSF (dinâmico)	Média (Docker)	GUI + Frida	DB	≥7	Manual/script	Mod.	Alta	Manual rerun	Depende analista
BareDroid	Alta (físicos)	Flash/reset	Logs centralizados	4.4	Mínimo	Baixa	Baixa	Ignora	Altíssima
VirusTotal Zenbox	Altíssima (cloud)	Fechada	DB	4.4, 8.0, 13	Auto comum	Mod.	Fechada	Sem fallback	Boa
Joe Sandbox	Alta (cloud)	Scripts interativos	DB robusto	4–11+	Auto/script	Alta	Boa	Auto fallback	Muito alta
ANY.RUN Android	Alta (SaaS)	Padrão único	Sessões	7–8	Manual	Baixa	Baixa	Manual	Alta
Hatching Triage	Alta (API/SOC)	Perfis fixos	DB	7–10	Auto/opção	Mod.	Boa	Live intervene	Alta
DroidHook	Variável	Xposed	JSON	6–11+	Monkey/manual	Mod.	Alta	Manual retry	Alta
DroidDungeon	Alta (cluster)	Interno (YAML-like)	Cumulativo	8–12+	Auto/manual	Alta	Altíssima	Auto→físico	Muito alta
ARTEMIS	**1–*N***	**YAML declarativo**	**DB Cumulativo**	**10–14+**	**Monkey/gui/YAML**	**Mod.**	**Altíssima**	**Retry + auto-adapt.**	**Adaptativa**

Legenda das colunas:
Escalab.: Escalabilidade; *Config.*: Configuração de ambiente; *Histórico*: Histórico de execuções; *Versões*: Versões Android suportadas; *Inputs*: Modo de simulação de entrada; *Sobrec.*: Sobrecarga de recursos; *Extens.*: Extensibilidade; *Erros*: Tratamento de falhas; *Anti-ev.*: Técnicas anti-evasão.
Fonte: Adaptado a partir de documentação oficial e testes próprios com as ferramentas listadas.

2.2. Tecnologias da Plataforma ARTEMIS

Técnicas de Instrumentação. Para observar o comportamento em tempo de execução de aplicações maliciosas, **ARTEMIS** integra três abordagens complementares de instrumentação, sendo que a personalização via YAML permite acoplar novos módulos de instrumentação conforme a necessidade de cada análise:

- ***Virtual Machine Introspection* (VMI) (conceitual)** – monitoramento externo ao sistema convidado, realizado no *hipervisor*, que captura chamadas de sistema sem depender de `ptrace`, garantindo maior furtividade. Embora não tenha sido testada na implementação atual, a arquitetura do ARTEMIS permite a integração de módulos VMI em emuladores modificados;
- **`ftrace`** – ferramenta de rastreamento interna do *kernel* Android, utilizada por meio de módulos de *kernel* externos, que registra funções e eventos do *kernel* com baixo *overhead*; esta abordagem foi amplamente testada em nosso estudo; e
- **Hooking in-guest** – inserção de ganchos via Frida, Xposed ou `strace` dentro do próprio sistema convidado, possibilitando interceptação de APIs Java/nativas; a injeção retardada desses *tracers* reduz a janela de detecção por mecanismos de depuração.

Enquanto a VMI oferece alta transparência ao custo de maior complexidade de infraestrutura; o `ftrace` equilibra furtividade e desempenho; o *hooking in-guest* é flexível, porém mais exposto a técnicas de evasão baseadas em `ptrace`.

Coleta de Artefatos Comportamentais. Os artefatos abaixo são exemplos do que pode ser coletado pelo **ARTEMIS** de acordo com os *tracers* e instrumentadores definidos no arquivo de configuração YAML:

- **Traces de syscalls** – obtidos via VMI ou `strace`/`ftrace`, revelam operações de arquivo, processos e *sockets*;
- **Tráfego de rede** – capturado com `tcpdump` ou *mitmproxy*, gerando arquivos PCAP que expõem protocolos, domínios e canais de comando e controle (C2);
- **Chamadas de API** – ganchos via Frida ou Xposed registram parâmetros e valores de retorno de APIs sensíveis (SMS, telefone, criptografia);
- **Evidências de UI** – logs de `logcat`, capturas de tela e eventos de entrada documentam a interação com a interface; e
- **Outros artefatos customizáveis** – o analista pode incluir extensões, como *dumps* de memória, *snapshots* de processos ou qualquer outro tipo de log suportado pelos módulos declarados.

Técnicas de Simulação de Entrada. Os métodos abaixo foram validados no estudo de caso, mas o pipeline do **ARTEMIS** é totalmente maleável e permite a adição de novos modos de estímulo via YAML:

- **Geração aleatória (*Monkey*)** – eventos pseudo-aleatórios para cobertura rápida da UI; utilizado em todas as análises automáticas;
- **Exploração guiada** – heurísticas ou análises estáticas direcionam a execução a componentes-alvo, conforme definido em perfis de configuração;
- **Interação programática** – argumentos são definidos em YAML e usados por *scripts* Python (ADB/UIAutomator) que executam toques, *swipes* e inserção de texto; e
- **Modos customizáveis** – é possível adicionar simuladores externos, desde que sigam a interface base de entrada. Após integrá-los ao pipeline, basta configurá-los via YAML.

Estratégias Anti-Evasão. As abordagens a seguir foram testadas no protótipo, mas o **ARTEMIS** suporta dinamicamente novos mecanismos de anti-evasão através de sua configuração YAML:

- ***Spoofing* de propriedades** – falsificação de IMEI/IMSI, sensores e campos `build.prop` para mascarar ambientes de emulação;
- **Neutralização de checagens** – aplicação de *patches* ou respostas simuladas para APIs de *anti-debug* e *anti-emulator*, injetados no *boot* ou em tempo de execução;
- **Injeção adaptativa de tracers** – seleção dinâmica de VMI, `ftrace` ou *hooking in-guest* (Frida/Xposed/`strace`), conforme o perfil de evasão detectado; e
- **Módulos customizáveis** – novos métodos de detecção e *bypass* podem ser adicionados declarando-se *scripts* ou binários para execução antes e durante a análise.

Análise Dinâmica Multi-Versão. A fragmentação do Android requer suporte a múltiplos *API levels*. O **ARTEMIS** foi validado com versões do Android 10–14 e legadas, usando imagens Docker/QEMU ou dispositivos físicos. A orquestração por microsserviços e Redis garante distribuição automática e escalável das análises. A arquitetura é flexível, permitindo a inclusão rápida de novas versões via YAML. No entanto, funcionalidades específicas, como novos modelos de permissão ou mudanças futuras na ART, podem demandar ajustes adicionais.

3. Arquitetura e Fluxo de Operação ARTEMIS

ARTEMIS ("Android Runtime Tracing, Execution and Malware Investigation System") foi concebido como uma plataforma modular, extensível e orientada a serviços para análise dinâmica de *malware* Android. Sua estrutura combina uma arquitetura de microsserviços com um *pipeline* declarativo, que juntos permitem coordenar desde poucas até centenas (ou potencialmente milhares) de instâncias de análise em paralelo, conforme a infraestrutura disponível. Nesta seção são explicados os módulos que compõem a plataforma e o fluxo de sua operação.

3.1. Arquitetura

A topologia do ARTEMIS divide-se em dois domínios lógicos—*motor de análise* e *backend*—que cooperam de forma assíncrona para oferecer escalabilidade horizontal, observabilidade completa e fácil reconfiguração.

3.1.1. Motor de Análise

O motor de análise é responsável por executar cada APK em ambiente controlado, orquestrar os *tracers* configurados em YAML e consolidar os artefatos comportamentais. Em resumo, o APK é recebido pelo Coordenador de Análise, o dispositivo é inicializado/configurado e o *Workflow* de Análise é disparado, resultando em relatórios JSON, pcaps, texto, etc. A visão geral deste fluxo de operação pode ser vista na Figura 1 e, em seguida, descreve-se cada um de seus componentes principais:

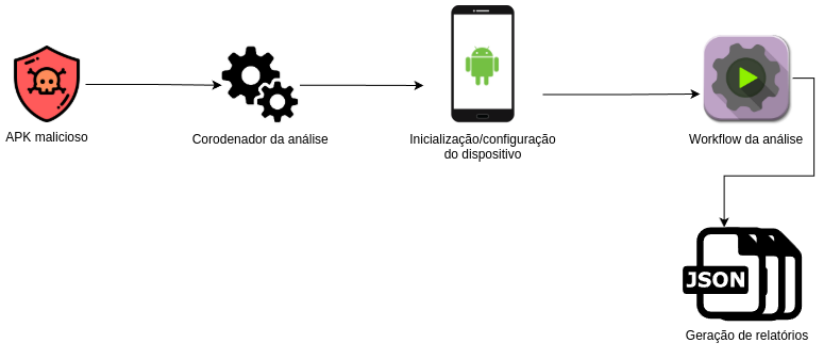

Figura 1. Visão geral do pipeline de análise de malware.
Fonte: Os Autores.

- **Coordenador de Análise:** inicializa o emulador ou dispositivo, aplica a configuração YAML, sincroniza tracers e consolida os resultados;
- **Motor de *Workflow*:** organiza o processo em quatro fases (Pré-instalação, Instalação, Pós-instalação e Análise), conforme ilustrado na Figura 2;
- **Interface de Dispositivo:** abstrai comandos ADB/fastboot, suportando emuladores QEMU, Genymotion e dispositivos físicos ARM;
- **Gerenciador de Tracers:** carrega e encerra qualquer *tracer* — nativo (`strace`, `tcpdump`), *kernel* (`ftrace`) ou customizado (scripts Frida/Xposed) — sem recompilar o motor; e
- **Gerenciador de Erros e Metadados:** produz *logs* estruturados, classifica exceções e aplica políticas de *retry* automáticas, garantindo robustez e rastreabilidade.

3.1.2. Fluxo de Operação das Análises

O *Workflow* de Análise, ilustrado na Figura 2, implementa um *pipeline* dinâmico em quatro fases sequenciais que guiam todo o processo de análise do APK:

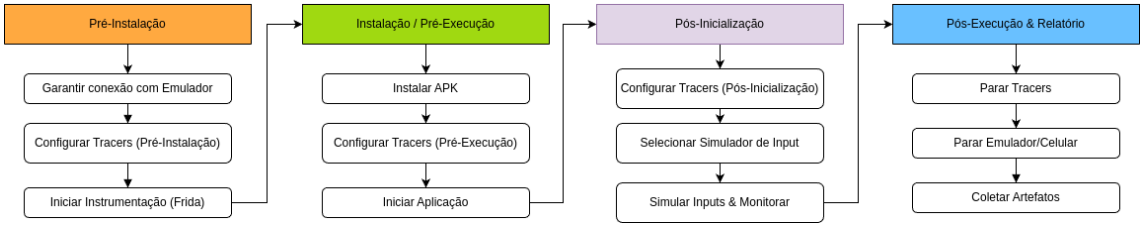

Figura 2. Detalhamento do *Workflow de Análise* em quatro fases: Pré-instalação, Instalação/Pré-Execução, Pós-Inicialização e Pós-Execução & Relatório.
Fonte: Os Autores.

Fase 1: Pré-Instalação. Esta fase prepara o ambiente antes da instalação do aplicativo a partir do seguintes passos: (i) **Garantir conexão com Emulador:** Após ter o identificador do dispositivo configurado, testamos se a conexão é persistente; (ii) **Configurar Tracers (Pré-Instalação):** Inicia os *tracers* definidos para esta fase (ex: captura de rede); (iii) **Iniciar Instrumentação Frida:** Inicializa o servidor Frida e carrega scripts para *bypasses* de segurança.

Fase 2: Instalação/Pré-Execução. Nesta fase, o APK é instalado e preparado para execução seguindo os passos: (i) **Instalar APK:** Implanta o aplicativo no dispositivo, normalmente via ADB; (ii) **Configurar Tracers (Pré-Execução):** Configura *tracers* adicionais após a instalação; (iii) **Iniciar Aplicação:** Lança o aplicativo usando o método definido na configuração.

Fase 3: Pós-Inicialização. Esta fase estabelece o monitoramento em tempo real, por meio de: (i) **Configurar Tracers (Pós-Inicialização):** Inicia *tracers* que requerem o aplicativo em execução; (ii) **Selecionar Simulador de *Input*:** Prepara o método de simulação conforme configurado; (iii) **Simular *Inputs* & Monitorar:** Executa o método de simulação selecionado para exercitar o aplicativo enquanto os *tracers* monitoram seu comportamento.

Fase 4: Pós-Execução & Relatório. Finaliza a análise e coleta os resultados: (i) **Parar Tracers:** Encerra todos os *tracers* ativos; (ii) **Parar Emulador/Celular:** Desliga o emulador ou desconecta do dispositivo; (iii) **Coletar Artefatos:** Consolida todos os arquivos gerados durante a análise e produz um relatório estruturado. Os artefatos gerados incluem traces de *syscalls*, PCAPs, *logs* de API, `logcat`, capturas de tela e um JSON de metadados (configuração, *hashes*, tempos, anomalias). Cada execução gera um identificador único, permitindo comparações longitudinais entre diferentes análises da mesma amostra.

3.1.3. Backend

O *backend* da plataforma ARTEMIS segue uma arquitetura de microserviços projetada para orquestrar análises de *malware* Android de forma escalável. Como ilustrado na Figura 3, o sistema implementa um fluxo completo desde o *upload* do APK até a entrega dos resultados. Esta arquitetura modular permite que o sistema escale horizontalmente—nos testes foram usados até 100 emuladores em paralelo, com o limite determinado apenas pelos recursos disponíveis na infraestrutura. A separação entre API, processamento assíncrono e armazenamento garante alta disponibilidade mesmo durante análises intensivas.

3.2. Frontend

O *front-end*, desenvolvido em *React*, já consome as APIs de autenticação, submissão e consulta de análises, gestão de APKs e rotas administrativas. Optamos por não detalhá-lo, pois o foco está na orquestração dos microsserviços, e não na interface.

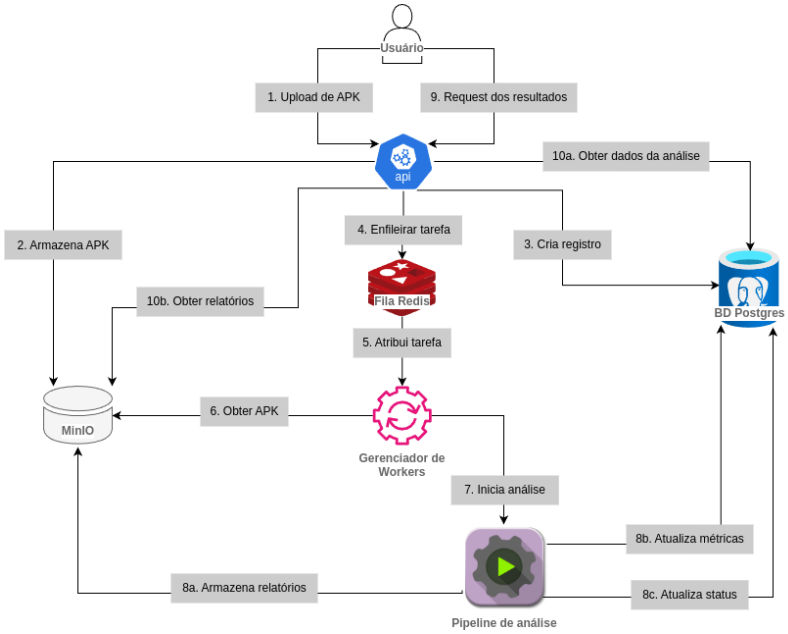

Figura 3. Fluxo de dados do backend: (1) o usuário submete APK; (2-3) o sistema armazena o binário no MinIO e cria registros no PostgreSQL; (4-5) uma tarefa é enfileirada no Redis e atribuída a um worker; (6-7) o worker obtém o APK e inicia a análise; (8a-c) os resultados são armazenados como relatórios no MinIO, enquanto métricas e status são atualizados no PostgreSQL; (9-10) o usuário pode então consultar os resultados e obter dados detalhados da análise via API.

Fonte: Os Autores.

4. Metodologia e Experimentos

Para avaliar o ARTEMIS, definiu-se um experimento em larga escala que combinou automação em massa com métricas de sucesso, desempenho e cobertura de evasão. O ambiente de testes utilizado possui os seguintes componentes: **Infraestrutura** - servidor com 2 TB de RAM e 2× Intel Xeon Gold 6338 (128 threads); **Pools de Análise** - 100 emuladores Android 10 (x86_64, 1 GiB RAM, 1 vCPU) e 4 dispositivos físicos ARM64 (Pixel 3/4); **Tracers Avaliados** - `strace`, `ftrace`, Frida, `tcpdump`, `logcat`, conforme

configurado em YAML; **Ferramentas de UI** - *Monkey*, exploração guiada e *scripts* ADB definidos em YAML.

Foi empregado um conjunto de **12.466** APKs maliciosos do repositório AndroZoo (coleta 2023). Esse montante corresponde aos arquivos válidos de um lote inicial de 12.536 *hashes*: **70** downloads retornaram vazios e foram descartados.

Justificativa. No contexto Android, o *Software Development Kit (SDK)* define dois parâmetros fundamentais: `minSdkVersion`, que estabelece o nível mínimo de API em que um app pode ser instalado, e `targetSdkVersion`, que indica o nível de API para o qual o app foi compilado e testado.

APKs de 2023 (n=12.466). A `minSdkVersion` varia de API 3 a 29 (média 20,74; mediana 21) — incluindo apps legados abaixo e acima do nível 21, exigido pelo Jetpack/AndroidX desde abril de 2024 [Stack Overflow users 2024] — e a `targetSdkVersion` abrange API 29 a 35 (média 31,30; mediana 32), cobrindo desde Android 10 (API 29, lançado em 03/09/2019 [Wikipedia contributors 2025]) até migrações para Android 14 (API 34 [Android Authority 2023]).

APKs de 2024 (n=9.114). A `minSdkVersion` permanece em API 3–29 (média 21,69; mediana 21), enquanto a `targetSdkVersion` varia de API 30 a 53 (média 32,66; mediana 33), com mais de 85 % dos apps em API 33–34, em função da política do Google Play que exige `targetSdkVersion` mínimo 34 para novos envios e atualizações desde 31/08/2024 [Google 2025].

Essa diversidade maior de SDKs em 2023 é essencial para uma análise comparativa robusta em emuladores Android 10 e 14.

4.1. Experimentos

Todos os experimentos descritos a seguir avaliam a plataforma **ARTEMIS** sobre os **12.466 APKs válidos**, mostrando empiricamente como um *framework* modular, declarativo e com histórico completo de análises pode superar as limitações de *sandboxes* públicas em profundidade, evasão e escalabilidade, garantindo cobertura abrangente mesmo diante da diversidade do ecossistema Android.

Para explorar aspectos complementares da análise dinâmica de *malware*, definimos três perfis de execução, cada um projetado para revelar um tipo distinto de limitação ou ponto forte:

i) **Emulador `x86_64` + `strace`**: Android 10 (API 29) com 1 GiB RAM e 1 vCPU. Coleta de chamadas de sistema (`strace`), logs de aplicação (`logcat`), pacotes de rede (`tcpdump`) e injeção de 900 eventos `Monkey` (≈3 min), simulando um sandbox intrusivo padrão;
ii) **Emulador `x86_64` stealth**: Mesma configuração de *hardware*, porém sem qualquer *tracer* baseado em `ptrace` no *boot*, para medir o impacto de instrumentação intrusiva na evasão de amostras; e
iii) **Dispositivos físicos `ARM64`**: Google Pixel 3 e 4 rodando Android 10, usados como *fallback* para casos de incompatibilidade de ABI detectados nos emuladores.

Dessa forma, foi possível avaliar simultaneamente a profundidade de coleta, a resiliência contra técnicas de detecção de instrumentação e a cobertura de ABI do mundo

real. Para tanto, o experimento foi organizado em cinco fases sequenciais, com artefatos e metadados acumulados ao longo do processo:

Fase 0 – Preparação: validação das amostras, registro de *hashes* e tamanhos iniciais;
Fase 1 – Emulação `strace`: primeira execução massiva no perfil (i), gerando artefatos completos de sistema, rede e *logs* de aplicação;
Fase 2 – Emulação stealth: reexecução dos APKs que falharam na fase 1 sem *tracer*, quantificando técnicas de evasão baseadas em `ptrace`;
Fase 3 – Fallback ARM64: instalação dos mesmos APKs falhos em dispositivos físicos, avaliando recuperação de casos de incompatibilidade de ABI detectadas nos emuladores; e
Fase 4 – Paralelas multi-versão: seleção de 100 APKs (20 por versão: 10-14) para análises simultâneas, alternando `strace` e `ftrace`.

A próxima seção detalha cada uma dessas fases, explicando em profundidade os métodos e os critérios utilizados nas execuções e análises realizadas.

Em cada fase foram extraídos 12 campos de metadados (pacote, atividades, permissões, e etc.) para enriquecer o perfil dos APKs e facilitar diagnósticos posteriores. O *cluster* contou com **100 emuladores Android 10 (x86_64)**, cada um com 1 GiB RAM e 1 vCPU, hospedados em um servidor com 2 TB RAM e dois Intel Xeon Gold 6338 (128 threads). Sob carga máxima, o uso total de RAM ficou em torno de 7%, sem saturação dos *NUMA nodes*. Esses resultados comprovam que a arquitetura baseada em microsserviços com filas Redis possibilita escalabilidade praticamente linear, permitindo adicionar novos *workers* sem alterações no código.

5. Testes e Resultados

Com base nos experimentos descritos na Seção 4.1, **12.466** análises foram realizadas (distribuídas sequencialmente pelas Fases 1 a 4)[1], sendo que **7.590 (60,89%)** delas foram bem-sucedidas e **4.876 (39,11%)** apresentaram falhas.

5.1. Análise de Metadados Estáticos

Para extrair estaticamente metadados críticos às fases de análise, utilizamos um pipeline em camadas (Tabela 2), ordenado por velocidade de extração dos 12 campos: `package_name`, `app_label`, `main_activity`, `sdk_min`, `sdk_target`, `permissions`, `version_code`, `version_name`, `activities`, `certificates`, `ABIs` e `services`. Cada ferramenta supera limitações da anterior: `aapt` é rápida, mas falha com manifestos ofuscados; `apkutils` faz parsing estruturado, porém não trata bem compressões atípicas; `androguard` recupera *bytecode* e certificados obfuscados, mas trava em empacotamentos dinâmicos; `direct_extraction` acessa diretamente o ZIP e XML cifrados; e `binary_manifest` decodifica o manifesto binário sem descompilar o APK. Essa estratégia em camadas garante cobertura completa dos metadados, mesmo sob ofuscação avançada, assegurando análises precisas.

Com a combinação dessas ferramentas, aumentamos o *recall* dos atributos principais de 63,98% para 75,98%, um ganho absoluto de 12 p.p. (18,75% relativo).

[1] A Fase 0 é meramente preparatória e não gera métricas de execução.

Tabela 2. Uso e eficácia das ferramentas de extração estática.

Ferramenta	Uso (%)	Sucesso (%)	Campo(s)-chave
aapt	77,74	63,98	package_name, permissions
apkutils	11,41	60,91	permissions
androguard	0,55	100 / 42,60	app_label / certificates
apktool	<0,10	—	parsing de Manifest
direct_extraction	10,21	61,21	main_activity
binary_manifest	<0,05	—	campos residuais

Fase 1 — Emulação `strace`: Diagnóstico Inicial

Uma vez que um dos objetivos do ARTEMIS é identificar as falhas para que se possa melhorar tanto a plataforma quanto obter o máximo de informações possíveis de uma APK executada, investigou-se os motivos por trás dessas falhas, explicados a seguir:

- **59,66%** (2.909) de erros de instalação (`apk_installation_error`);
- **34,33%** (1.674) de erros na inicialização do app (`launch_error`);
- **5,74%** (280) de erros durante a execução ativa de um *tracer* (`runtime_tracer_error`);
- **0,25%** (12) de erros desconhecidos pelo pipeline (`unknown_error`);
- **0,02%** (1) de erro na configuração inicial do *tracer* (`pre_install_tracer_error`).

Como cada falha produz artefatos completos (*logs*, capturas de pacotes, *traces*) registrados no histórico, é possível realizar uma *triagem reversa* automática, isto é, lançar perfis alternativos que disparam apenas quando pertinentes, economizando inúmeras horas de CPU.

A fim de se verificar as falhas observadas no ARTEMIS, iniciou-se a investigação pelo maior conjunto, que compreende os principais erros de instalação (2.909 APKs). A Tabela 3 apresenta os códigos de erro identificados durante a instalação das APKs sob análise, onde **NO_MATCHING_ABIS** indica que o dispositivo não suporta nenhuma das *Application Binary Interfaces* (ABI) presentes no pacote (por exemplo, `armeabi-v7a`, `arm64-v8a`, `x86`), impossibilitando o carregamento de bibliotecas nativas; **MISSING_SPLIT** sinaliza a falha ao instalar um *split-APK* em razão da ausência de um ou mais módulos necessários (como arquivos de arquitetura, idioma ou densidade de tela) no conjunto de APKs; **UNKNOWN_FAILURE** corresponde a uma falha genérica não categorizada pelo instalador do Android; e **SHARED_USER_INCOMP** refere-se à incompatibilidade no uso de `sharedUserId` entre APKs dependentes, geralmente resultante de divergências na assinatura digital ou nas declarações de permissão, o que impede a instalação conjunta para preservar a integridade do sistema.

Uma vez que a fragmentação de ABIs em APKs é um fator crítico de falha na execução em emuladores x86, considerando incompatibilidades no conjunto de instruções da CPU, extensões específicas de *hardware*, orientação de bytes (*endianness*) e convenções de chamada entre código nativo e a runtime do Android, realizou-se uma análise sobre o conjunto de 12.466 APKs com base na configuração de suas bibliotecas nativas.

Tabela 3. Tipos de erros encontrados, quantidade de APKs que os apresentaram e porcentagem da falha de instalação.

Código de Erro	APKs	% de Instalações
NO_MATCHING_ABIS	2.840	97,63%
MISSING_SPLIT	51	1,75%
UNKNOWN_FAILURE	17	0,58%
SHARED_USER_INCOMP.	1	0,03%

A Tabela 4 apresenta, de forma unificada, a distribuição das APKs por tipo de ABI, a taxa de sucesso de execução e o tempo médio de execução em ambiente x86. As categorias são definidas da seguinte forma: **no_abis** representa APKs sem bibliotecas nativas (por exemplo, escritos apenas em Java ou Kotlin); **x86_only** inclui apenas bibliotecas compiladas para a arquitetura x86; **both_x86_and_arm** corresponde a APKs multiplataforma, contendo bibliotecas para x86 e ARM; **arm64_only** representa APKs com suporte exclusivo a ARM64 (AArch64); **arm64_and_arm** refere-se a APKs com bibliotecas tanto para ARM64 quanto para ARM 32-bit; e **arm_only** representa APKs com suporte apenas para ARM 32-bit.

Tabela 4. Estatísticas por configuração de ABI: distribuição no conjunto de APKs, taxa de sucesso e tempo médio de execução.

ABI	# APKs	% Total	Sucesso	Média (s)
no_abis	6.479	51,97%	72,90%	561,65
both_x86_and_arm	3.102	24,88%	91,81%	603,49
arm64_and_arm	1.642	13,17%	0,00%	357,15
arm64_only	638	5,12%	0,47%	348,57
arm_only	589	4,72%	0,00%	290,79
x86_only	16	0,13%	100,00%	596,89

Considerando os 1.674 erros de inicialização das *apps* (`launch_error`), o código `app_not_running`, que representa apps que não foram inicializados corretamente, apareceu em 100% dos casos. Destes, 87,93% eram APKs do tipo `no_abis` e 10,81% do tipo `both_x86_and_arm`, totalizando 98,74% dos casos. Os 1,26% restantes distribuem-se entre outras configurações de ABI. Isto indica que, mesmo sendo instalados corretamente, tais APKs possuem algum outro bloqueio para inicialização sem falhas. A Tabela 5 mostra as principais bibliotecas necessárias para a execução desses APKs.

Ao se fazer a análise dos tempos de execução para as 12.466 APKs, observou-se **Tempo Médio** de 521,45 segundos (mediana de 513,67s, mínimo de 88,56s e máximo de 1.649,26s), considerando que as **análises bem-sucedidas (7.590)** duraram 600,60s em média enquanto que as **análises falhas (4.876)** duraram 397,9s em média. Já os abortos precoces, isto é, execuções que duraram $\approx$<400s) reduziram o tempo médio em aproximadamente 33,8%, ao passo que indicaram onde ajustes na estratégia de *fallback* são cruciais. Com isso, identificou-se que as falhas tendem a ocorrer em estágios iniciais da análise, o que explica sua duração média significativamente menor, e que o tipo de

Tabela 5. Principais bibliotecas ausentes em APKs com `launch_errors`.

Biblioteca	Ocorrências
libbreakpad-core.so	134
libhermes.so	120
libmmkv.so	41
libflutter.so	17
libjscexecutor.so	17

falha mais custoso (507,23s) ocorre quando o *tracer* dá problema durante a execução. Registrar relatórios em todas as fases permite triagem reversa automatizada e reexecuções muito mais eficientes. Na Tabela 6 temos o tempo médio por tipo de erro.

Tabela 6. Tempo médio por tipo de erro, seguido da quantidade de APKs que apresentou cada um dos erros (4.876 APKs).

Tipo de Erro	Tempo Médio (s)	Quantidade
runtime_tracer_error	507,23	280
launch_error	481,08	1.674
apk_installation_error	339,62	2.909
pre_install_tracer_error	227,58	1
unknown_error	389,42	12

Como visto na Tabela 4, foram calculados os tempos médios de execução por ABI, levando à conclusão que `arm_only` é a configuração mais rápida (290,79s) e `both_x86_and_arm` é a mais lenta (603,49s), devido ao custo das operações/traduções de instruções em diferentes arquiteturas. A cada incremento de versão de API, foi observado um acréscimo de aproximadamente 11,7 s no tempo de processamento, sugerindo maior complexidade no *bytecode*, multiplicidade das verificações de segurança e permissões nas versões mais recentes do Android.

Fase 2 — Emulação `stealth`: Reexecução Sem *ptrace*

Reexecutamos todos os APKs que foram instalados com sucesso mas não geraram relatórios na Fase 1. Essa etapa quantificou o impacto das proteções *anti-debug*/*anti-emulator*.

Proteções Anti-Debug e Anti-Emulador. Foram identificadas **10.876** APKs (87,25%) fazendo uso de ao menos uma técnica de evasão, com média de 2,38 métodos por aplicativo. As três técnicas mais frequentes encontradas foram: *emulator detection* (84,85%, 10.577), *root detection* (71,80%, 8.950) e *anti-debug* (51,15%, 6.376). Observou-se também que os *tracers* baseados em *ptrace* (`strace` e frida) são facilmente identificados por proteções *Anti-Debug*. Ao injetá-los tardiamente ou alternar para `ftrace`/hooking *in-guest*, a cobertura de amostras analisáveis foi ampliada em 6,8% em relação a análises restritas ao *ptrace*.

Impacto da Instrumentação Intrusiva. Dos APKs com proteção Anti-Debug, 1.674 falharam em gerar relatórios, indicando que interromperam sua execução ao detectar ten-

tativas de depuração. Dentre esses, 1.491 (89,07%) não estavam corrompidos e foram reprocessados com uma abordagem alternativa. Ao remover o uso do `strace`, 101 APKs (6,77% do grupo) passaram a produzir relatórios completos, dos quais 81 (80,20%) concluíram com êxito utilizando apenas `ftrace`, uma técnica que não depende de *ptrace*. Isso demonstra que a capacidade de alternar declarativamente entre `strace`, `ftrace` e, futuramente, VMI — sem recompilar o *kernel* — permitiu recuperar a maioria das amostras afetadas por defesas Anti-Debug. Como o `ftrace` opera no nível de *kernel*, ele evita interações diretas com processos via chamadas de depuração, tornando-se menos detectável por mecanismos de proteção no espaço do usuário.

Fase 3 — *Fallback* ARM64 em Dispositivos Físicos

Realizar o roteamento adaptativo a dispositivos físicos (*fallback* ARM64) recuperou **98,70%** das falhas de instalação, eliminando virtualmente os falsos-negativos por incompatibilidade de ABI. Entre os **2.909** APKs que não puderam ser instalados no emulador x86_64, **2.872** (**98,70%**) instalaram-se sem erros em smartphones Pixel 3/4 ARM64, restando apenas **37** (**1,30%**) com problemas persistentes.

Esses casos residuais limitam-se aos códigos `MISSING_SPLIT` (ausência de módulos obrigatórios de arquitetura, idioma ou densidade de tela) e `NO_CERTIFICATES`, decorrente da falta de assinatura digital válida. Assim, a estratégia de *fallback* em *hardware* real elevou a cobertura de instalação para praticamente 99%. O resultado sugere ainda que um segundo nível de *fallback*, baseado no carregamento dinâmico de bibliotecas antes da inicialização, pode mitigar grande parte dos `launch_errors` remanescentes e ampliar a cobertura global da plataforma.

Fase 4 — Execuções Paralelas Multi-Versão (APIs 10–14)

Na fase 4, 100 APKs (20 para cada API 10–14) foram executados em paralelo usando perfis `strace`/`ftrace`. Essas versões foram escolhidas pois representam quase a totalidade de dispositivos atualmente em operação. Todos os *jobs* concluíram sem sobrescrever artefatos anteriores, mantendo um histórico completo de cada execução. A orquestração paralela e cumulativa possibilitou identificar comportamentos condicionais de cada SDK, fundamentais para mapear padrões de evasão ligados às versões do Android.

5.2. Lições Aprendidas por Fase (F#)

- **(F3) Compatibilidade de ABI.** O *fallback* em dispositivo físico recuperou 58,89% das falhas totais (2.872/4.876);
- **(F2) Instrumentação furtiva.** O *pipeline* centrado em `ftrace` recuperou 80,20% das amostras anti-debug que falhavam com *ptrace*;
- **(F1–F3) Triagem reversa.** Artefatos de falha, desde a Fase 1, orientam reexecuções direcionadas, economizando recursos;
- **(F0–F4) Histórico cumulativo.** Armazenar todos os relatórios viabiliza análises longitudinais e auditoria forense; e
- **(F1–F4) Escalabilidade elástica.** Microsserviços e filas permitem crescimento sem refatorações profundas.

Limitações e Ameaças à Validade. O estímulo via `Monkey` pode não acionar fluxos complexos de UI; **concorrência intensiva.** Execuções paralelas podem introduzir *timing side-channels*; **viés de amostragem.** A base AndroZoo, embora variada, pode refletir preferências regionais; **evasões avançadas.** Técnicas baseadas em virtualização de *hardware* ou gatilhos externos permanecem fora do escopo; **suporte de versão.** Focamos no Android 10–14, exigindo adaptações para o Android 15 e posteriores.

6. Conclusão

Este trabalho apresenta a plataforma ARTEMIS, uma solução dinâmica para análise de malware Android que supera desafios como fragmentação do ecossistema, técnicas de evasão e escalabilidade, utilizando uma arquitetura de microsserviços, agendamento assíncrono via Redis e configuração em YAML para orquestrar emuladores e dispositivos físicos em paralelo, suportando múltiplas versões do Android (10–14). Um estudo com 12.466 APKs maliciosas demonstrou alta eficácia, como taxa de instalação de 98,7%, recuperação de 80,2% dos APKs que falhavam por detecção de depuração e armazenamento de artefatos para análises longitudinais. Limitações incluem o uso do *Monkey* para estímulo de UI, padrões de temporização suscetíveis a detecção e evasões avançadas. Os resultados e o código-fonte do **ARTEMIS**, bem como os conjuntos de dados utilizados, estarão disponíveis como recurso aberto para a comunidade, fomentando novos desenvolvimentos de análise de malware Android.

Agradecimentos

Este trabalho foi apoiado pelo Centro de Computação Científica e Software Livre - C3SL da UFPR, em parceria com o Ministério da Saúde, bem como pelo CNPq via bolsa de produtividade em pesquisa.

Referências

Android Authority (2023). Android 14 release schedule.

ANY.RUN (2025). Interactive android sandbox for malware analysis. `https://any.run/`.

Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., and Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 55–62. IEEE.

Developers, G. A. (2024). Android platform versions dashboard. `https://developer.android.com/about/dashboards`.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2010). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 393–407.

Fratantonio, Y., van der Veen, V., and Platzer, C. (2014). Andrubis: Android malware under the magnifying glass. Number TR-ISECLAB-0414-001.

GlobalStats, S. (2024). Mobile operating system market share worldwide. `https://gs.statcounter.com/os-market-share/mobile/worldwide`.

Google (2025). Meet google play's target api level requirement. Accessed: May 2025.

Hatching (2024). Triage: Advanced android sandbox for malware analysis. https://tria.ge/.

Intelligence, I. S. (2023). Reducing resource overhead in malware sandboxing. https://securityintelligence.com/.

Lab, K. (2024). Mobile malware evolution 2024. Technical report, Kaspersky Lab.

Lantz, P. (2012). Droidbox: Android application sandbox. https://github.com/pjlantz/droidbox.

LLC, J. S. (2024). Joe sandbox mobile - android dynamic analysis. https://www.joesecurity.org/.

Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., and Vigna, G. (2015). Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, pages 71–80.

Neuner, S., van der Veen, V., Lindorfer, M., Huber, M., Merzdovnik, G., Mulazzani, M., and Weippl, E. (2014). Enter sandbox: Android sandbox comparison. In *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST)*.

Research, C. P. (2024a). Mobile security report 2024. Technical report, Check Point.

Research, E. (2024b). Droiddungeon: Bypassing android malware evasion techniques. https://s3.eurecom.fr/.

Research, T. M. (2021). Evasive malware techniques targeting android. Technical report, Trend Micro.

ResearchGate (2023). Droidhook: A flexible android dynamic analysis framework. https://www.researchgate.net/.

Revivo, I., Caspi, O., and Shalyt, M. (2015). Cuckoodroid – fighting the tide of android malware. Check Point Blog.

Stack Overflow users (2024). What is the minsdkversion for targetsdkversion 34?

Statista (2024). Distribution of android versions worldwide.

Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Internet Society.

(VirusTotal), G. T. I. (2023). Virustotal zenbox: Dynamic malware analysis. https://docs.virustotal.com/.

Wikipedia contributors (2025). Android 10 — android version history.

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

Zhou, P. (2020). Limitations and extensions of cuckoo sandbox for android analysis. https://medium.com/.

Zorz, M. (2016). Mobsf: Security analysis of android and ios apps. Help Net Security.