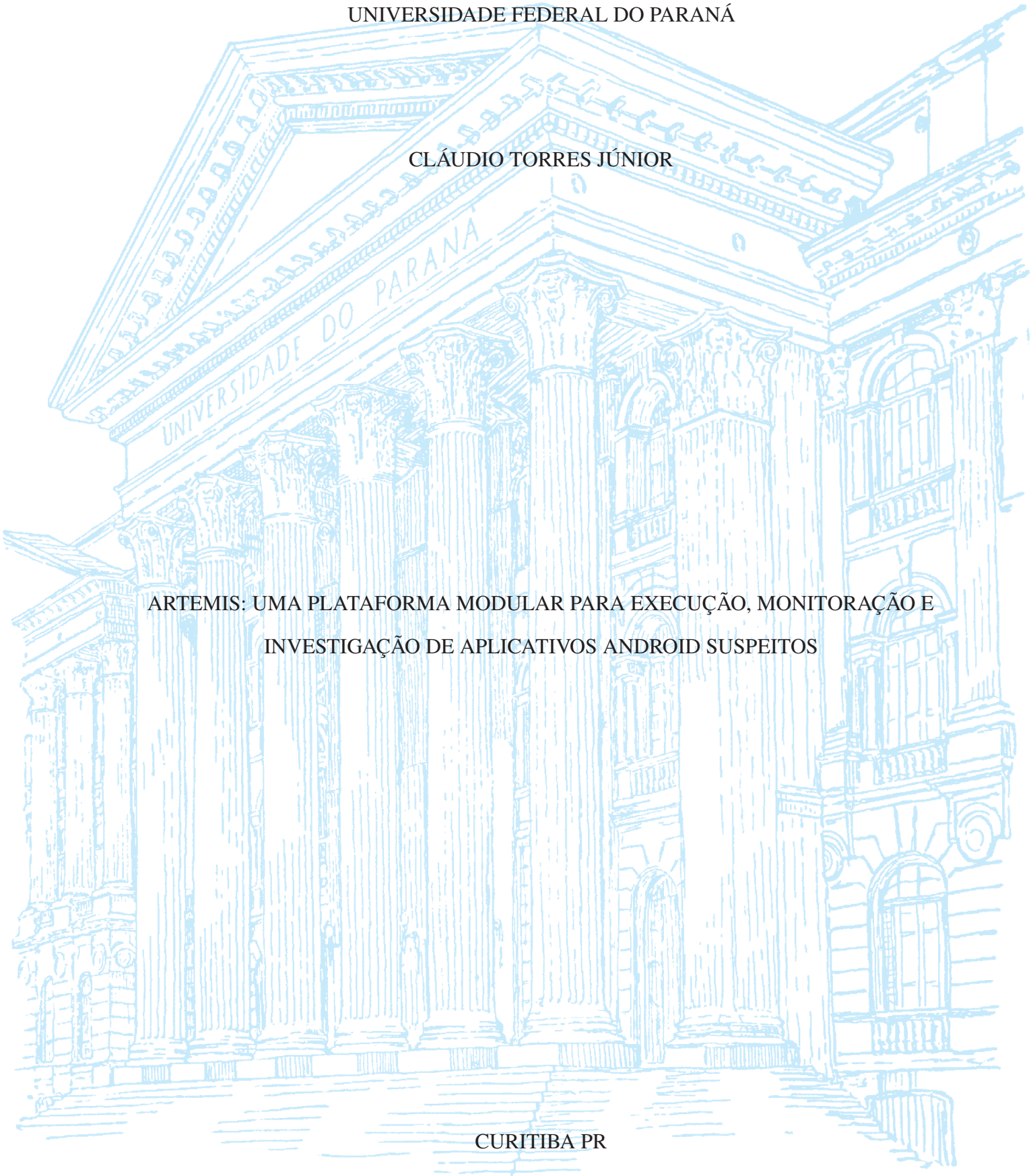
UNIVERSIDADE FEDERAL DO PARANÁ

CLÁUDIO TORRES JÚNIOR



ARTEMIS: UMA PLATAFORMA MODULAR PARA EXECUÇÃO, MONITORAÇÃO E INVESTIGAÇÃO DE APLICATIVOS ANDROID SUSPEITOS

CURITIBA PR

2025

CLÁUDIO TORRES JÚNIOR

ARTEMIS: UMA PLATAFORMA MODULAR PARA EXECUÇÃO, MONITORAÇÃO E INVESTIGAÇÃO DE APLICATIVOS ANDROID SUSPEITOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientador: André Grégio.

CURITIBA PR

2025

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Torres Júnior, Cláudio
Artemis: uma plataforma modular para execução, monitoração e investigação de aplicativos Android suspeitos / Cláudio Torres Júnior. – Curitiba, 2025.
1 recurso on-line : PDF.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: André Ricardo Abed Grégio

1. Android (Recurso eletrônico). 2. Informática - Segurança de dados. 3. Telefone móvel - Segurança. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Grégio, André Ricardo Abed. IV. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **CLAUDIO TORRES JUNIOR**, intitulada: **ARTEMIS: Uma Plataforma Modular para Execução, Monitoração e Investigação de Aplicativos Android Suspeitos**, sob orientação do Prof. Dr. ANDRÉ RICARDO ABED GRÉGIO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 17 de Junho de 2025.

Assinatura Eletrônica
27/06/2025 06:10:18.0
ANDRÉ RICARDO ABED GRÉGIO
Presidente da Banca Examinadora

Assinatura Eletrônica
26/06/2025 15:16:17.0
VINÍCIUS FÜLBER GARCIA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
30/06/2025 15:55:21.0
MARCUS FELIPE BOTACIN
Avaliador Externo (TEXAS A&M UNIVERSITY)

"O sucesso nasce do querer, da determinação e da persistência."
— José de Alencar

AGRADECIMENTOS

Agradeço, em primeiro lugar, ao meu orientador Prof. Dr. André Grégio, que tem acompanhado minha trajetória desde a graduação. Sua orientação, paciência e dedicação foram fundamentais para meu desenvolvimento acadêmico e pessoal, sempre incentivando a busca por novos desafios e pela excelência.

Estendo meus agradecimentos aos colegas e amigos que, de diferentes formas, contribuíram com apoio, troca de ideias e companheirismo ao longo dessa jornada.

Por fim, agradeço à minha família, pelo incentivo constante, compreensão e apoio incondicional em todos os momentos.

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 maliciosos, 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: mobile security, dynamic analysis, Android malware, microservices, multi-version instrumentation

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

ARTEMIS	Android Runtime Tracing, Execution and Malware Investigation System
APK	Android Package
API	Application Programming Interface
ABI	Application Binary Interface
UI	User Interface
YAML	YAML Ain't Markup Language
Redis	Remote Dictionary Server
MinIO	Minimal Input/Output (armazenamento de objetos)
JSON	JavaScript Object Notation
PCAP	Packet Capture
VMI	Virtual Machine Introspection
LKM	Loadable Kernel Module
ADB	Android Debug Bridge
CPU	Central Processing Unit
RAM	Random Access Memory
QEMU	Quick Emulator
SQL	Structured Query Language
JWT	JSON Web Token
S3	Simple Storage Service
REST	Representational State Transfer
CORS	Cross-Origin Resource Sharing
TCP	Transmission Control Protocol
HTTP	HyperText Transfer Protocol
SSL	Secure Sockets Layer

SUMÁRIO

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. Em 2024, apenas cerca de 21% dos dispositivos utilizavam Android 13 (lançado em 2022), enquanto versões mais antigas como Android 11, 10 e 9 ainda correspondiam juntas a uma parcela majoritária da base instalada (SL, 2024). Essa diversidade de níveis de API, arquiteturas de CPU (e.g., ARM32, ARM64, x86_64) e *forks* de fabricantes compromete a representatividade e a cobertura das plataformas tradicionais para análise dinâmica de *malware* Android. Muitos *frameworks* de análise legados, como o DroidBox (limitado ao Android 4.x) (Lantz, 2012), não acompanham as versões mais recentes do sistema, impossibilitando a avaliação de ameaças direcionadas a releases modernos.

Além disso, malwares sofisticados empregam técnicas de anti-análise para detectar se estão sendo executados em um emulador ou ambiente instrumentado, alterando seu comportamento ou abortando a execução para evitar detecção. Verificações como *build fingerprint* do dispositivo, presença de binários de `su`, diferença de arquitetura de CPU ou ausência de sensores reais (e.g., acelerômetro, giroscópio) são indícios comumente usados para identificar um *sandbox* (Research, 2021). Ao mesmo tempo, mecanismos como *SSL pinning* (que impede a interceptação de tráfego criptografado) e checagens de depurador ativo (*anti-debug*) são empregados para frustrar analisadores. Estas barreiras, somadas à fragmentação, agravam o desafio de obter observabilidade completa do comportamento malicioso em diferentes versões do Android (Neuner et al., 2014).

Para superar tais desafios, este artigo apresenta **ARTEMIS** (*Android Runtime Tracing, Execution and Malware Investigation System*): não uma simples sandbox, mas uma plataforma integrada de orquestração de *sandboxes*, emuladores e ferramentas de análise em múltiplos níveis. Projetada sobre uma arquitetura de microsserviços, ARTEMIS incorpora um agendamento assíncrono de tarefas, configuração declarativa em YAML e suporte multi-versão (Android 10 a 14, além de dispositivos físicos). Seu *pipeline* modular de instrumentação inclui, entre outros, Frida, `strace`, `ftrace`, `tcpdump` e *Monkey*, permitindo executar centenas de instâncias de análise em paralelo, escalando conforme a infraestrutura disponível e garantindo ampla cobertura comportamental e robustez contra técnicas de evasão.

Com o objetivo principal de servir como bloco básico na construção de sistemas de análise de aplicações móveis, as contribuições da plataforma ARTEMIS são as seguintes:

1. **Pipeline declarativo e suporte multi-versão/ABI**: definição totalmente declarativa em YAML de *tracers*, estímulos de interface (e.g., *Monkey*) e parâmetros de emulador sem necessidade de recompilar nada no sistema. A arquitetura de microsserviços orquestra análises simultâneas em múltiplas versões do Android (10–14) e em dispositivos físicos, com mecanismos adaptativos de *fallback* que alternam automaticamente entre emulador `x86_64` e dispositivo ARM64 ao detectar inconsistências de ABI.

2. **Instrumentação poderosa e resistência à anti-instrumentação**: uso de técnicas modernas (e.g., Frida) para monitorar e modificar o comportamento em tempo real, inclusive burlando verificações de ambiente (e.g., *build fingerprint*, presença de `su`, simulação de sensores). Esse suporte inclui, também, *hooks* in-guest via Frida e `strace` com injeção retardada para mitigar detecções baseadas em depuração (*anti-debug*).

3. **Coleta abrangente de artefatos de execução**: captura completa de todos os artefatos gerados durante a execução do app, incluindo logs de aplicação, chamadas de sistema (*syscalls*), tráfego de rede (PCAP), capturas de tela e metadados, com persistência num histórico cumulativo que viabiliza comparações longitudinais, triagem reversa de erros e auditorias forenses de comportamento.

4. **Simulação avançada de entradas e interações**: motor de estímulos capaz de exercitar funcionalidades dos aplicativos analisados mesmo sem intervenção humana, combinando padrões de toque, gestos e eventos de sistema para maximizar a cobertura de código e detectar comportamentos ocultos.

5. **Arquitetura escalável e extensível com interface web**: design modular baseado em microsserviços, apoiado por filas de tarefas assíncronas, que permite processar múltiplas análises em paralelo conforme a infraestrutura disponível. Inclui persistência de dados e uma interface web para visualização dos resultados, histórico de execuções e colaboração entre analistas.

2 LITERATURA E TECNOLOGIAS

Nesta seção, revisamos as principais iniciativas de análise dinâmica de *malware* para Android de forma cronológica, destacando sua evolução técnica e os pontos fortes e fracos de cada abordagem. Em seguida, detalhamos as cinco categorias de tecnologias que fundamentam a plataforma **ARTEMIS**: 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). Os primeiros sistemas de análise dinâmica de *malware* Android surgiram por volta de 2010, focados em obter visibilidade básica sobre comportamentos suspeitos. O *AASandbox* ((Bläsing et al., 2010)) criou a base para execução controlada de APKs em um ambiente isolado, coletando chamadas de sistema e atividades de rede mínimas. No mesmo ano, o *TaintDroid* ((Enck et al., 2010a)) apresentou um avanço significativo ao introduzir rastreamento de fluxo de informação em tempo real (*dynamic taint tracking*) integrado diretamente à máquina virtual Dalvik. Essa abordagem modificava a VM para marcar dados sensíveis (como localização ou contatos) com *tags* de rastreamento e monitorar sua propagação, permitindo detectar vazamentos de informações com um *overhead* moderado, tipicamente inferior a 15%. Sua principal limitação era o escopo restrito ao *runtime* gerenciado (bytecode Java), sendo incapaz de observar diretamente comportamentos implementados em código nativo ou vazamentos por canais encobertos, uma fragilidade que malwares poderiam explorar para evadir a detecção.

Em 2012, o *DroidScope* ((Yan e Yin, 2012a)) inovou ao unificar Virtual Machine Introspection (VMI) em QEMU para código nativo com o rastreamento em nível de Dalvik, permitindo reconstruir uma "visão semântica"completa da execução do malware ao correlacionar eventos de alto nível com as instruções de baixo nível subjacentes. Ainda em 2011, o *DroidBox* ((Lantz, 2011)), derivado do Android Sandbox Project do Honeynet, tornou-se amplamente utilizado por sua capacidade de instrumentação automatizada. Ele combinava modificações no *framework* Java com o módulo de rastreamento do TaintDroid para registrar operações como envio de SMS, acesso a arquivos e tráfego de rede em uma imagem customizada do Android 2.3/4.1. Contudo, por operar majoritariamente no espaço de usuário, atividades maliciosas puramente nativas que não invocassem APIs Android não eram capturadas. Todas essas ferramentas pioneiras ficaram restritas a versões antigas do Android (2.3–4.x) e exigiam sistemas customizados, limitando sua aplicabilidade conforme o Android evoluía e, notadamente, com a transição do Dalvik para o *runtime* ART, que quebrou a compatibilidade de muitas delas.
Avanços intermediários (2014–2018). A partir de 2014, a necessidade de plataformas mais escaláveis e com maior cobertura impulsionou novas arquiteturas. O *Andrubis* ((van der Veen et al., 2014)), uma adaptação do famoso *sandbox* Anubis de Windows, foi projetado para analisar aplicativos em larga escala, combinando análise estática e dinâmica em um *cluster* de servidores. Dinamicamente, ele utilizava um emulador QEMU customizado que monitorava tanto o nível Dalvik quanto o sistema operacional, capturando *syscalls* via introspecção. Para contornar a falta de interação humana, Andrubis aplicava técnicas de estímulo como eventos de UI aleatórios (via Monkey) e simulação de SMS, aumentando a cobertura de código. Em 2015, o *CopperDroid* ((Tam et al., 2015a)) aprimorou a abordagem de VMI para reconstruir automaticamente comportamentos de alto nível a partir da observação de chamadas de sistema

(*syscalls*). Sua instrumentação no *hypervisor* QEMU permitia monitorar todas as *syscalls* sem modificar o *framework* Android, conferindo cobertura para ações iniciadas tanto por código Java quanto nativo. Nesse mesmo período, o *TaintART* ((Sun et al., 2016)) enfrentou o desafio da nova *runtime* do Android, adaptando o TaintDroid para o ART. Ele inseria lógica de marcação tanto no *bytecode* quanto em componentes nativos do ART, mas mantinha a limitação de focar no código gerenciado e introduzia um *overhead* de desempenho e energia significativo em dispositivos reais.

Soluções Open Source e Industriais (2015–2022). Entre 2015 e 2022, ferramentas de código aberto ganharam popularidade por sua praticidade, adotando instrumentação via *hooking* em vez de emulação completa. O *CuckooDroid* ((Cuc, 2021)), uma extensão do Cuckoo Sandbox, e o *MobSF* ((Mob, 2020)) utilizaram *frameworks* de *hook* dinâmico como Xposed e Frida para monitorar a execução de aplicativos em dispositivos ou emuladores padrão. Essa abordagem em nível de *framework* é mais fácil de manter e adaptar a novas versões do Android, porém oferece um escopo de monitoramento limitado, pois malwares podem contornar a camada de APIs públicas invocando diretamente chamadas nativas. Além disso, a presença do próprio mecanismo de *hook* (seja Xposed ou Frida) é um artefato detectável que malwares sofisticados exploram para evadir a análise.

Plataformas Recentes (2018–2025). Nos últimos anos, a análise dinâmica evoluiu para focar em furtividade (*stealthiness*) e em abordagens híbridas para combater técnicas de evasão cada vez mais robustas. O *DroidDungeon* ((Baggio et al., 2023)), por exemplo, é um exemplo recente que combina instrumentação em nível de kernel com monitoramento em espaço de usuário para enganar o malware. Essa abordagem híbrida utiliza um módulo de kernel para interceptar chamadas e retornar respostas "falsas", porém coerentes com um dispositivo real, a fim de esconder os artefatos de emulação e burlar verificações de integridade. Embora essas novas arquiteturas aumentem a transparência, elas também adicionam complexidade de implementação.

Limitações Persistentes. Apesar dos avanços, um desafio transversal a todas as abordagens é a **cobertura comportamental**. Muitos malwares empregam gatilhos condicionais, como ativação após interações específicas do usuário ou eventos externos, para dificultar a observação de suas rotinas maliciosas. Ferramentas que não incluem mecanismos de automação de UI ou simulação de eventos externos podem falhar em ativar partes significativas do código. Estudos demonstram que a introdução de estímulos aumenta a detecção; por exemplo, o CopperDroid acionou comportamentos adicionais em mais de 60% dos malwares testados usando um simples gerador de eventos (Tam et al., 2015a). Outra limitação crítica é a **transparência do ambiente**. Emuladores customizados frequentemente ficam presos a versões antigas do Android e geram discrepâncias (como IMEI padronizado, ausência de sensores reais) que malwares podem usar para detectar o *sandbox* e alterar seu comportamento. (Baggio et al., 2023) relatam que aproximadamente 14% dos malwares em testes se recusaram a executar sua carga maliciosa em *sandboxes* que não implementavam contramedidas antievasão robustas.

A Tabela 2.1 sintetiza uma comparação teórica entre as soluções mais relevantes citadas acima e o **ARTEMIS**, destacando cada quesito técnico.

Legenda das colunas. Cada critério da Tabela 2.1 foi padronizado para reduzir ambiguidades:

- *Escalab.* — capacidade de análises paralelas e facilidade de crescimento.
- *Config.* — grau de automação e reprodutibilidade do ambiente.
- *Histórico* — persistência e consulta, de logs simples até bancos de dados completos.
- *Versões* — faixas Android suportadas nativamente.
- *Inputs* — geração de estímulos: *Monkey*, automação de UI, eventos externos.

- *Sobrec.* — custo relativo de CPU/memória/tempo de análise.
- *Extens.* — possibilidade de extensão: APIs de plugins, hooking, VMI, integrações.
- *Erros* — estratégias de recuperação: retry, fallback, auto-adaptação.
- *Anti-ev.* — contramedidas contra evasão (realismo do ambiente).

Escalas objetivas (critérios de qualificação).

- *Escalab.*: **Baixa** = 1–2 paralelos; **Média** = 3–10; **Alta** = 10–100; **Altíssima** = >100 ou SaaS elástico.
- *Config.*: **Baixa** = setup manual; **Média** = scripts/perfis básicos; **Alta** = configuração declarativa (p.ex., YAML); **Fechada** = serviço proprietário sem customização.
- *Histórico*: **Baixa** = logs brutos; **Média** = JSON/relatórios simples; **Alta** = banco de dados consultável; **Robusta** = DB + relatórios ricos/linha do tempo.
- *Inputs*: **Baixa** = Monkey apenas; **Média** = Monkey + SMS/broadcast; **Alta** = automação de UI + eventos do SO; **Altíssima** = sensores, GPS, chamadas, push, cenários declarativos.
- *Sobrec.*: **Baixa** < 30%; **Moderada** 30–80%; **Alta** > 80% (instrumentação pesada/VMI).
- *Extens.*: **Baixa** = pipeline rígido; **Média** = alguns hooks; **Alta** = módulos e API estáveis; **Altíssima** = SDKs + VMI/hooking versátil e scripting interativo.
- *Erros*: **Fraca** = aborta; **Média** = retry limitado; **Alta** = fallback de perfil/VM; **Altíssima** = auto-adaptação (troca de estratégia ou migração para hardware real).
- *Anti-ev.*: **Fraca** = emulação evidente; **Moderada** = randomização básica; **Alta** = perfis realistas e hooks stealth; **Altíssima** = execução em hardware real com perfis ricos.

Tabela 2.1: 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	Monkey (500 eventos)	Alta	Baixa	Aborta em erro	Fraca
TaintDroid	Baixa (1)	ROM Dalvik	Logcat	≤4.3	Estímulo externo	Alta	Baixa	Aborta em erro	Fraca
DroidScope	Baixa (1)	Plugins VMI	Logs brutos	2.3	Script básico	Alta	Média	Aborta em erro	Média
DroidBox	Baixa (1)	Script de emulador	JSON simples	4.1–4.x	Monkey + SMS	Alta	Baixa	Sem retry (1 execução)	Fraca
Andrubis	Alta (cluster)	Automação padronizada	Banco de dados	4.2	Monkey + eventos do sist.	Alta	Média	Retry limitado (próxima amostra)	Média
ANANAS	Baixa (local)	Módulos selecionáveis	Logs modulares	2.3–4.x	Monkey modular	Variável	Alta	Aborta em erro	Média
CopperDroid	Média (manual)	QEMU modificado	Relatórios de comportamento	4.1	Script	Moderada	Baixa	Saída limitada	Média
CuckooDroid	Alta (10–15)	Perfis de dispositivo	Banco de dados	4.x	Monkey + automação	Moderada	Alta	Retry na inicialização	Moderada
MobSF (dinâmico)	Média (Docker)	GUI + Frida	Banco de dados	≥7	Manual / script	Moderada	Alta	Reexecução manual	Dependente do analista
BareDroid	Alta (físicos)	Flash / reset	Logs centralizados	4.4	Mínimo	Baixa	Baixa	Ignora (continua)	Altíssima
VirusTotal Zenbox	Altíssima (cloud)	Fechada	Banco de dados	4.4, 8.0, 13	Automação comum	Moderada	Fechada	Sem *fallback*	Alta
Joe Sandbox	Alta (cloud)	Scripts interativos	Banco de dados robusto	4–11+	Automação / script	Alta	Alta	*Fallback* automático	Muito alta
ANY.RUN Android	Alta (SaaS)	Padrão único	Sessões	7–8	Manual	Baixa	Baixa	Tratamento manual	Alta
Hatching Triage	Alta (API/SOC)	Perfis fixos	Banco de dados	7–10	Automação / opcional	Moderada	Alta	Intervenção ao vivo	Alta
DroidHook	Variável	Xposed	JSON	6–11+	Monkey / manual	Moderada	Alta	Reexecução manual	Alta
DroidDungeon	Alta (cluster)	Interno (YAML-like)	Histórico cumulativo	8–12+	Automação / manual	Alta	Altíssima	De *auto* → físico	Muito alta
ARTEMIS	**1–*N***	**YAML declarativo**	**Hist. cumulativo (DB)**	**10–14+**	**Monkey / GUI / YAML**	**Moderada**	**Altíssima**	**Retry + auto-adaptação**	**Adaptativa**

2.2 PRINCIPAIS FERRAMENTAS DE ANÁLISE DINÂMICA DE MALWARE ANDROID

Nesta seção, apresenta-se cada ferramenta listada na Tabela 2.1, destacando, para cada uma, aspectos correspondentes às colunas: escalabilidade, configuração, histórico, versões suportadas, modos de *input*, sobrecarga de recursos, extensibilidade, tratamento de erros e resistência a técnicas de evasão.

2.2.1 AASandbox

O *AASandbox* (2010) foi projetado para executar um único APK em um emulador Android 2.2/2.3 com um módulo de kernel customizado que interceptava *syscalls* do aplicativo (Bläsing et al., 2010). Quanto à **escalabilidade**, rodava apenas uma instância por vez (baixa); a **configuração** exigia *patch* no kernel e emulador fixo, sem flexibilidade de versões. O **histórico** de execuções era mínimo, limitando-se a *logs* em arquivo local, sem banco de dados persistente. Suportava apenas **Android 2.2–2.3** (versões antigas), e os **inputs** vinham exclusivamente do *Monkey* básico ou *scripts* manuais. A **sobrecarga** era alta, pois a instrumentação no kernel aumentava uso de CPU/RAM; a **extensibilidade** era baixa, pois alterar o módulo de kernel demandava recompilar e adaptar o código para cada versão. Em caso de **erros**, o sistema abortava a execução sem *retry* ou *fallback*. Não havia mecanismos de **anti-evasão**, de modo que malwares podiam detectar a presença do módulo via listagem de módulos carregados.

2.2.2 TaintDroid

O *TaintDroid* (2010) modifica a Dalvik VM para rastrear *taints* em tempo real (Enck et al., 2010b). A **escalabilidade** é baixa (uma instância por vez em emulador), e a **configuração** requer uma ROM customizada baseada em Dalvik, sem suporte a ART. O **histórico** consiste em *logs* de *taint* e chamadas de API, armazenados localmente, sem base cumulativa. Suporta **Android 2.3–4.x** (Dalvik); não funciona em ART (Android 5+). Os **inputs** precisam ser gerados externamente (por exemplo, via ADB ou *Monkey*), sem automação integrada. A **sobrecarga** é moderada a alta (30 % de overhead), e a **extensibilidade** é baixa, pois alterações à VM demandam recompilação completa. Em situações de **erros** (por exemplo, conflito com bibliotecas nativas), o sistema simplesmente falha sem mecanismo de *retry*. A **anti-evasão** é fraca: malwares podem carregar bibliotecas nativas que contornem o rastreamento de *taint* ou detectar que a Dalvik foi modificada.

2.2.3 DroidScope

O *DroidScope* (2012) utiliza introspecção em QEMU para capturar instruções Dalvik e nativas simultaneamente, reconstruindo a execução do aplicativo (Yan e Yin, 2012b). Sua **escalabilidade** é baixa (apenas emulações isoladas), e a **configuração** exige QEMU modificado e desativação do JIT da Dalvik. O **histórico** é composto por traços brutos de instruções e chamadas de sistema, sem armazenamento cumulativo estruturado. Suporta apenas **Android 2.3**; não há suporte a versões posteriores. Os **inputs** dependem de *scripts* externos ou do *Monkey* básico, sem geração guiada. A **sobrecarga** é alta, pois desativar o JIT e rastrear instruções nativas penaliza significativamente o desempenho. A **extensibilidade** é média; adicionar novos módulos de rastreamento em QEMU requer conhecimento profundo do *hipervisor*. Em caso de **erros**, a emulação era encerrada sem *fallback*. A **anti-evasão** é média: por operar "fora" do sistema convidado, evita parte das verificações em nível de aplicativo, mas malwares ainda podem detectar sinais de emulação no QEMU.

2.2.4 DroidBox

O *DroidBox* (2011) combina TaintDroid com ganchos no *framework* Java para monitorar vazamentos de dados, SMS, rede e acessos a arquivos em Android 4.x (Lantz, 2012). Apresenta **escalabilidade** baixa (instância única), **configuração** fixa em emulador Android 4.1, e o **histórico** consiste em JSON simples gerado localmente. Suporta apenas **Android 4.1–4.3**, não funciona em ART. Os **inputs** são gerados via *Monkey* e eventos de SMS simulados, sem automação refinada. A **sobrecarga** é alta devido à instrumentação na VM Dalvik e ao monitoramento adicional; a **extensibilidade** é baixa, pois requer modificação do *framework* Java a cada atualização do Android. Em **erros**, a execução aborta sem *retry*. A **anti-evasão** é fraca: malwares detectam ausência de elementos de sistema real (por exemplo, serviços do Google) e identificam a instrumentação via verificações de assinaturas de método.

2.2.5 Andrubis

O *Andrubis* (2014) integra sandboxing, TaintDroid e `tcpdump` num emulador Android 4.2, coletando pacotes de rede e rastros de *taint* via módulo VMI (Fratantonio et al., 2014). A **escalabilidade** era alta em *cluster*, analisando milhares de apps em paralelo; a **configuração** exigia QEMU modificado, Android 4.2 *custom* e ambientes de rede virtualizados. O **histórico** inclui banco de dados de execuções (logs, PCAP, metadados) com persistência. Suporta **Android 4.2**; sem suporte a versões modernas. Os **inputs** combinam *Monkey* e simulação de eventos de sistema (SMS, GPS). A **sobrecarga** é alta, pois há múltiplos níveis de instrumentação (VMI, Dalvik, `tcpdump`); a **extensibilidade** é média, pois novos módulos podem ser adicionados ao *pipeline*, porém exigem automação avançada. Em caso de **erros** (por exemplo, falha de ABI), há *fallback* para desmontar o emulador e tentar novamente, mas sem categorização automática de falhas. A **anti-evasão** é média: desvia algumas verificações básicas, mas malwares que testam serviços ausentes do Google ou detectam QEMU conseguem evadir a análise.

2.2.6 ANANAS

O *ANANAS* (2013) é arquitetado em módulos independentes que monitoram chamadas de sistema via LKM e instrumentam chamadas de *framework* Java (Neuner et al., 2014). Sua **escalabilidade** é baixa (execução local), e a **configuração** requer kernel e *runtime* Dalvik customizados para Android 2.3–4.x. O **histórico** armazena *logs* modulares de cada execução, mas sem interface web para visualização. Suporta apenas **Android 2.3–4.2**, sem versões posteriores. Os **inputs** são gerados por módulos *Monkey* customizados; não há exploração guiada embutida. A **sobrecarga** varia de acordo com os módulos ativos, mas tende a ser média-alta devido à instrumentação no kernel e na VM. A **extensibilidade** é alta, pois novos módulos simplesmente se conectam à arquitetura plugável, embora exijam desenvolvimento em C para o kernel. Em **erros**, as execuções abortam se um módulo falha, sem *retry* automático. A **anti-evasão** é média, pois módulos de kernel podem ser detectados, mas nem todos os rastreamentos em *guest* são expostos ao app.

2.2.7 CopperDroid

O *CopperDroid* (2015) utiliza VMI em QEMU para interceptar *syscalls* e reconstruir ações de alto nível em Android 2.3 (Tam et al., 2015b). A **escalabilidade** é média (execução manual em protótipo), e a **configuração** demanda QEMU modificado e imagem do Android 2.3. O **histórico** gera relatórios comportamentais em alto nível, mas não preserva *logs* de baixo nível. Suporta apenas **Android 2.3**. Os **inputs** são restritos a *scripts* rudimentares ou *Monkey*. A

sobrecarga é moderada, pois não altera o sistema convidado, mas o *hipervisor* impõe custo; a **extensibilidade** é baixa, pois requer alterar o *code base* do *hipervisor* para ampliar rastros. Em **erros**, falha silenciosa se os módulos VMI encontrarem inconsistências, sem *fallback*. A **anti-evasão** é média, pois não deixa artefatos no SO convidado, mas malwares podem identificar sinais de QEMU via instruções atípicas.

2.2.8 CuckooDroid

O *CuckooDroid* (2015) adapta o Cuckoo Sandbox para APKs usando Xposed em Android 4.x (Revivo et al., 2015). A **escalabilidade** é alta (herda a arquitetura do Cuckoo), e a **configuração** envolve contêineres Docker com Android 4.1.2 e Xposed Framework. O **histórico** registra evidências em banco de dados SQL, incluindo *snapshots* e *logs*. Suporta **Android 4.1–4.4**. Os **inputs** combinam *Monkey* e *scripts* Xposed que simulam eventos, mas sem exploração guiada. A **sobrecarga** é moderada a alta, devido à sobreposição de Xposed e contêiner; a **extensibilidade** é alta, pois novos módulos Python podem ser adicionados ao Cuckoo. Em caso de **erros** (por exemplo, falha do Xposed), o sistema faz *retry* em outro *worker*, mas sem categorização detalhada. A **anti-evasão** é moderada, pois o uso do Xposed pode ser detectado pelos malwares, embora contêiners forneçam algum grau de isolamento.

2.2.9 MobSF (dinâmico)

O *MobSF* (2016–) combine análise estática, instrumentação via Frida e *mitmproxy*, suportando **Android 4.1–12+** em emuladores e dispositivos reais (Zorz, 2016; Hacking Articles, 2021; Appknox, 2024). A **escalabilidade** é média (instâncias separadas em Docker), e a **configuração** requer imagem Docker com dependências (Frida, mitmproxy) e emulador. O **histórico** é armazenado em banco SQLite, com relatórios detalhados; oferece interface web. Os **inputs** podem ser manuais ou via *scripts* Python usando ADB/UIAutomator. A **sobrecarga** é moderada: Frida impõe *overhead*, mas escalabilidade em Docker mitiga impacto; a **extensibilidade** é alta, pois novos módulos (MJpeg, BurpSuite) podem ser integrados. Em **erros**, o usuário precisa reiniciar manualmente, sem *retry* automático. A **anti-evasão** depende do analista: Frida oferece alguns *bypasses*, mas malwares ainda detectam *hooks* se não forem mascarados adequadamente.

2.2.10 BareDroid

O *BareDroid* (2015) executa apps em *hardware* físico (Android 4.4) para minizar artefatos de emulação (Mutti et al., 2015). A **escalabilidade** é alta se houver escala de dispositivos, mas implica alto custo; a **configuração** exige *farm* de dispositivos reais e *scripts* de *flash/reset*. O **histórico** centraliza *logs* em servidor remoto, mas não mantém versões cumulativas extensas. Suporta **Android 4.4** (e ocasionalmente 5.0), sem compatibilidade ampla. Os **inputs** são mínimos (dispositivos podem interagir manualmente ou via *scripts* simples). A **sobrecarga** é baixa (não há emulação), porém a **extensibilidade** é limitada ao *hardware* disponível. Em **erros**, a falha em um dispositivo é simplesmente isolada, sem *fallback* automatizado; o analista precisa intervir. A **anti-evasão** é altíssima, pois não há emulação, dificultando a detecção por malwares.

2.2.11 VirusTotal Zenbox

O *VirusTotal Zenbox* (2023) executa instrumentação Frida em **Android 13** para coletar comportamento em *runtime* (, VirusTotal). A **escalabilidade** é altíssima (rodando em nuvem), e a **configuração** é fechada, gerenciada pela VirusTotal. O **histórico** alimenta o banco de dados

do VirusTotal, com registros estruturados de cada execução. Suporta **Android 4.4, 8.0, 13**. Os **inputs** são gerados automaticamente via heurísticas internas (sem exposição aos usuários). A **sobrecarga** é moderada, pois a instrumentação Frida roda em infraestrutura dedicada; a **extensibilidade** é interna, sem possibilidade de módulos externos. Em **erros**, a análise falha silenciosamente e passa para outra instância (sem *feedback* detalhado ao usuário). A **anti-evasão** é boa: Zenbox aplica técnicas de *bypass* padrão de Frida, mas malwares muito sofisticados ainda podem escapar.

2.2.12 Joe Sandbox Mobile

O *Joe Sandbox Mobile* (2024) é plataforma comercial ativa que instrumenta APKs em dispositivos virtuais **4–11+** com *fallback* automático em caso de detecção (LLC, 2024). A **escalabilidade** é alta, rodando em nuvem com balanceamento; a **configuração** é fechada e padronizada pelo fornecedor. O **histórico** é robusto, com banco de dados estruturado contendo *logs*, capturas de tela e PCAP. Os **inputs** combinam *Monkey* customizado e *scripts* proprietários, sem intervenção do usuário. A **sobrecarga** é alta, dado o nível de instrumentação detalhada, mas mitigada por infraestrutura dedicada; a **extensibilidade** é boa, pois Joe Sandbox adiciona novas técnicas a cada versão, mas sem disponibilizar código para clientes. Em **erros**, há *retry* automático em outro *worker* ou reexecução sem instrumentação para contornar *anti-sandbox*. A **anti-evasão** é muito alta, com simulação fidedigna de dados do dispositivo, detecção de *hooks* e estratégias adaptativas.

2.2.13 ANY.RUN Android

O *ANY.RUN Android* (2025) oferece sandbox ARM interativo via SaaS (ANY.RUN, 2025). A **escalabilidade** é alta (infraestrutura em nuvem), mas limitada a instâncias alocadas; a **configuração** é manual via interface web, sem personalização profunda. O **histórico** armazena sessões isoladas, mas não mantém base cumulativa unificada. Suporta **Android 7–8**. Os **inputs** são gerados manualmente pelo analista via interface gráfica, sem automação embutida. A **sobrecarga** é baixa, pois se baseia em VM ARM otimizada; a **extensibilidade** é baixa, pois não há suporte a módulos externos. Em **erros**, a sessão cai e deve ser reiniciada manualmente. A **anti-evasão** é alta para essa categoria de serviço, mas malwares podem detectar a infraestrutura de nuvem.

2.2.14 Hatching Triage

O *Hatching Triage* (2024) provê sandbox Android **7–10** via API/SOC, gerando relatórios detalhados (Hatching, 2024). A **escalabilidade** é alta em nuvem privada, e a **configuração** é feita via perfis fixos pré-definidos. O **histórico** alimenta banco de dados central com metadados de cada execução. Os **inputs** são gerados automaticamente com heurísticas internas ou manual via API. A **sobrecarga** é moderada, dado o balanceamento de recursos; a **extensibilidade** é boa, pois novos perfis podem ser adicionados, mas sem personalização por módulo de terceiros. Em **erros**, o sistema reinicia automaticamente a instância. A **anti-evasão** é alta, com módulos para camuflar sinais de VM, mas não totalmente infalível contra malwares avançados.

2.2.15 DroidHook

O *DroidHook* (2023) é *framework* acadêmico para instrumentação in-guest em Android 6–11+ via Xposed (ResearchGate, 2023). A **escalabilidade** é variável (dependendo de instâncias

de emulador), e a **configuração** exige instalar Xposed em cada emulador físico ou em VM. O **histórico** consiste em *logs* JSON gerados localmente, sem base central. Os **inputs** combinam *Monkey* e *scripts* de interação básica. A **sobrecarga** é moderada-alta, pois o Xposed impõe *overhead* significativo; a **extensibilidade** é alta, permitindo adicionar novos *hooks* declarando módulos. Em **erros**, falha silenciosa se o Xposed não carregar ou o app detectar a *framework*, sem retry integrado. A **anti-evasão** é alta para instrumentação Java, mas falha contra verificações nativas de `ptrace` ou *fingerprinting*.

2.2.16 DroidDungeon

O *DroidDungeon* (2024) introduziu motor declarativo em YAML para orquestrar *tracers* e módulos de evasão em Android 8–12+ (Research, 2024b). A **escalabilidade** é alta em *cluster* acadêmico, e a **configuração** via YAML simplifica suporte a múltiplas versões. O **histórico** é armazenado em banco local, com logs cumulativos. Suporta **Android 8–12**, facilitando extensão a novas versões via YAML. Os **inputs** combinam *Monkey* custom e *scripts* Python, ajustáveis em YAML. A **sobrecarga** é alta, pois executa múltiplos *tracers* simultâneos; a **extensibilidade** é altíssima, permitindo adicionar novos módulos declarando-os em YAML. Em **erros**, há *fallback* automático para execução em dispositivo físico se o emulador falhar. A **anti-evasão** é muito alta, pois inclui módulos para *spoofing* de *build.prop*, sensores e resposta a verificações nativas, mas ainda pode ser contornada por técnicas novas.

2.3 TECNOLOGIAS DA PLATAFORMA ARTEMIS

A seguir, detalhamos as cinco categorias de tecnologias centrais para a construção da **ARTEMIS**, explicitando como cada uma delas é implementada ou suportada pelo sistema, e citando referências que comprovem seu uso ou fundamentem suas vantagens e limitações.
1. Técnicas de Instrumentação. Para observar o comportamento em tempo de execução de aplicações maliciosas sem depender exclusivamente de `ptrace`, ARTEMIS suporta três abordagens complementares:

- **Virtual Machine Introspection (VMI).** *VMI* permite monitoramento externo ao sistema convidado, diretamente no *hipervisor*, coletando chamadas de sistema sem modificar o kernel ou usar `ptrace`. Essa abordagem, pioneira em ferramentas como *DroidScope* e *CopperDroid*, garante alta furtividade e resistência a técnicas comuns de anti-`ptrace` ((Networks, 2024)). Embora não esteja ativa na versão atual do protótipo, a arquitetura modular de ARTEMIS facilita a integração futura de módulos VMI em emuladores QEMU modificados, como demonstrado em trabalhos recentes de comparação de *sandboxes* ((Neuner et al., 2014)) e arquiteturas híbridas.

- **`ftrace` no kernel Android.** O `ftrace` ((Developers, 2014)) é uma ferramenta nativa de rastreamento de funções e eventos do kernel Android, com *overhead* relativamente baixo. ARTEMIS inclui módulos compilados como *loadable kernel modules* (LKM) e um *script bash* que configuram o `ftrace` diretamente, sendo capazes de registrar entradas e saídas de funções críticas (e.g., `execve()`, `open()`, `socket()`, etc.), viabilizando análise granular de execuções e detecção de chamadas suspeitas de forma não intrusiva.

- **Hooking in-guest (Frida / Xposed / `strace`).** Por meio de Frida, Xposed ou `strace` injetado em tempo de execução, ARTEMIS injeta ganchos diretamente dentro do sistema

convidado para interceptar APIs Java e chamadas de sistema. A injeção retardada ("*delayed injection*") reduz a janela de detecção por *anti-debuggers* ou *scripts* que verificam a ausência de `ptrace` ((Research, 2021)). Frida, em particular, permite modificar o comportamento de métodos nativos e Java em tempo real, uma técnica popularizada por *frameworks* como MobSF e CuckooDroid , mas cuja presença é um conhecido vetor de evasão.

2. Coleta de Artefatos Comportamentais. O ARTEMIS utiliza diversos *tracers* e ferramentas para reunir artefatos durante a execução das aplicações:

- **Traces de chamadas de sistema (syscalls).** Obtidos via VMI ou ganchos *in-guest* (`strace`/`ftrace`), esses traces revelam operações de E/S de arquivos, criação de processos e uso de *sockets*, fundamentais para reconstruir semanticamente o comportamento do malware, como demonstrado pelo *CopperDroid*.
- **Tráfego de rede (PCAP).** Capturado com `tcpdump` ou *mitmproxy*, gera arquivos PCAP que expõem protocolos, domínios de C2 e fluxo de dados criptografados, essenciais para análise de comunicação em tempo real, uma prática comum em *sandboxes* como *Andrubis*.
- **Chamadas de API sensíveis.** Ganchos via Frida ou Xposed interceptam APIs que acessam SMS, contatos, criptografia ou recursos do sistema, registrando parâmetros e valores de retorno para análise posterior. Essa técnica é utilizada em MobSF ((Zorz, 2016)) e DroidHook ((ResearchGate, 2023)) para mapear *calls* críticas.
- **Evidências de UI e contexto.** *Logs* de `logcat`, capturas de tela periódicas e eventos de entrada (toques, *swipes*, inserção de texto) são reunidos para documentar a interação do *malware* com a interface gráfica, ajudando a reconstruir cenários de uso, como explorado pelo *Andrubis* para aumentar a cobertura de código.
- **Outros artefatos customizáveis.** O analista pode declarar módulos adicionais via YAML para coletar *dumps* de memória, *snapshots* de processos ou quaisquer *logs* específicos (p.e., dados de sensores simulados), ampliando o leque de inspeção a depender do escopo da análise, seguindo a abordagem flexível proposta por *DroidDungeon*.

3. Técnicas de Simulação de Entrada. Para exercitar funcionalidades dos aplicativos sem depender exclusivamente de interação manual, ARTEMIS permite múltiplos modos de estímulo via configuração YAML:

- **Geração aleatória (Monkey).** O clássico *Monkey* gera eventos pseudo-aleatórios de toque, clique e rolagem. Sua eficácia em descobrir funcionalidades adicionais foi demonstrada em sistemas como *CopperDroid* e *Andrubis*. Em ARTEMIS, o *Monkey* é ajustado dinamicamente conforme heurísticas definidas em YAML, reduzindo execuções redundantes.
- **Exploração guiada por heurísticas.** Perfis de configuração permitem que análises estáticas ou modelos de aprendizado (p.e., *RLDroid* (Series, 2025)) guiem a exploração da UI, priorizando componentes identificados como críticos (login, funções de compra, envio de SMS, etc.). Essa abordagem melhora a cobertura de código em cenários em que a aleatoriedade se mostra insuficiente.

- **Interação programática (ADB/UIAutomator).** Scripts Python utilizam ADB ou UIAutomator para executar toques, *swipes* e inserção de texto em campos-chave, configurados por meio de parâmetros YAML. Isso garante repetibilidade e flexibilidade na execução de fluxos de uso específicos.

- **Modos customizáveis.** Bibliotecas externas ou simuladores especializados podem ser integrados ao pipeline: basta implementar a interface de entrada definida pelo ARTEMIS e declarar o módulo no arquivo YAML, habilitando o uso de *frameworks* de *fuzzing* de entrada ou módulos de ML personalizados ((ANY.RUN, 2025; Hatching, 2024)).

4. Estratégias Anti-Evasão. Malwares modernos empregam verificações para detectar se estão em ambiente controlado ou emulador e escapar da análise. ARTEMIS inclui mecanismos reativos e proativos para mitigar esses artifícios:

- **Spoofing de propriedades de hardware.** Falsificação de IMEI/IMSI, valores de sensores (acelerômetro, giroscópio) e campos de `build.prop` para mascarar o ambiente de emulação. Malwares frequentemente verificam esses valores em busca de padrões anômalos (ex: IMEI preenchido com zeros) para detectar *sandboxes*. Implementações de *spoofing* podem ser ativadas via YAML antes da inicialização do APK.

- **Neutralização de checagens de *anti-debug* e *anti-emulator*.** Aplicação de patches em tempo de boot ou injeção de respostas simuladas a APIs de detecção (e.g., checagens de `getprop` ou presença de `/system/xbin/su`). Malwares também podem detectar artefatos de emulação como QEMU via checagens de propriedades do *build* ou instruções CPUID anômalas. Conjuntamente, o uso de VMI reduz a probabilidade de ser detectado por processos que monitoram `ptrace`.

- **Injeção adaptativa de tracers.** Dependendo do perfil de evasão identificado (e.g., se o APK falha ao detectar ausência de sensores ou aborta em emulator), ARTEMIS alterna dinamicamente entre VMI, `ftrace` ou *hooks in-guest* (Frida/Xposed/`strace`), garantindo maior robustez, uma abordagem inspirada por arquiteturas híbridas como a do *DroidDungeon*.

- **Módulos customizáveis de evasão.** Novas técnicas de *bypass* (p.e., *patching* de bibliotecas nativas, manipulação de chamadas de função via eBPF) podem ser adicionadas declarando-se *scripts* ou binários que rodem antes e durante a análise, facilitando atualização ágil conforme novas técnicas de evasão surgem, como proposto por *frameworks* flexíveis como o *DroidHook*.

5. Análise Dinâmica Multi-Versão. A fragmentação do ecossistema Android impõe a necessidade de suportar múltiplos níveis de API e arquiteturas de CPU (ARM32, ARM64, x86_64). Muitas das primeiras ferramentas ficaram amarradas a versões específicas, como *DroidScope* ao Android 2.3 ou *CopperDroid* ao 4.1 , reduzindo sua eficácia contra malwares recentes. ARTEMIS foi validado com imagens Docker/QEMU de Android 10–14, além de dispositivos físicos rodando versões legadas (Android 8-10). A orquestração por microsserviços e Redis (ou similar) garante distribuição automática e balanceada das instâncias de análise entre diferentes "*workers*", cada um configurado para executar em um ABI específico ((Developers, 2024; SL, 2024; Statista, 2024)).

Sempre que surgir uma nova versão do Android (e.g., Android 15 no futuro), basta adicionar a imagem correspondente ao repositório de *dumps* QEMU e incluir seu identificador no arquivo YAML, sem necessidade de alterar a lógica de orquestração ou pipeline. Isso contrasta com abordagens tradicionais que, frequentemente, exigem recompilar o emulador ou portar todas as instrumentações a cada nova *release* ((Tam et al., 2015b; Revivo et al., 2015)).

3 ARQUITETURA E FLUXO DE OPERAÇÃO DO ARTEMIS

ARTEMIS (“Android Runtime Tracing, Execution and Malware Investigation System”) foi concebido como plataforma modular, extensível e orientada a serviços para análise dinâmica de *malware* Android. Sua estrutura combina microserviços com um *pipeline* declarativo, permitindo orquestrar desde poucas até centenas de instâncias de análise em paralelo, conforme a infraestrutura disponível. Nesta seção descrevemos os módulos que compõem a plataforma e o fluxo de operação de ponta a ponta.

3.1 ARQUITETURA GERAL

A topologia do ARTEMIS divide-se em dois domínios lógicos principais: o *Motor de Análise* e o *Backend*, que cooperam de forma assíncrona para oferecer escalabilidade horizontal, observabilidade e fácil reconfiguração. Há ainda o *Frontend* (interface web) que se comunica exclusivamente com o Backend via APIs REST.

3.1.1 Pipeline Declarativo com YAML

A espinha dorsal da flexibilidade e extensibilidade do ARTEMIS reside no seu **pipeline declarativo**, controlado por um arquivo de configuração no formato YAML. Este arquivo é o principal ponto de controle para qualquer execução de análise, permitindo que pesquisadores e analistas definam com precisão o ambiente, as ferramentas e os comportamentos a serem observados, sem a necessidade de modificar ou recompilar o código-fonte da plataforma.

É por meio do arquivo YAML que se alcança uma **alta cobertura de análise**, pois ele permite a fácil inserção e combinação de diferentes *sandboxes*, emuladores (AOSP, Genymotion), *tracers* (como `strace`, `tcpdump`, `ftrace` e scripts Frida) e simuladores de inputs (desde o simples `Monkey` até estratégias mais inteligentes). Essa abordagem permite que uma mesma amostra de *malware* seja submetida a múltiplos cenários de análise, variando desde a versão do Android até as ferramentas de monitoramento ativadas.

Ao final de cada execução (*run*), a plataforma gera um **relatório completo e autocontido**, que inclui não apenas os artefatos coletados (logs, capturas de rede, traces de sistema), mas também a configuração YAML exata que foi utilizada. Este acoplamento entre dados e metadados é fundamental, pois possibilita a realização de **triagens reversas** (entendendo por que um comportamento específico ocorreu ao revisar a configuração) e **estudos horizontais**, onde os resultados de uma mesma amostra em diferentes abordagens e configurações podem ser comparados diretamente para identificar técnicas de evasão ou comportamentos condicionais do *malware*.

3.1.2 Motor de Análise

O *Motor de Análise* é o domínio responsável por materializar as diretivas do arquivo YAML. Ele executa cada APK em um ambiente controlado, orquestra os *tracers* e simuladores de input, e consolida os artefatos comportamentais.

A Figura 3.1 apresenta uma visão geral da interação entre os componentes. O fluxo se inicia com o **Coordenador da Análise** recebendo a tarefa de analisar um APK malicioso. Ele é responsável pela **Inicialização/configuração do dispositivo** e por acionar o **Workflow da análise**. Ao final do processo, ocorre a **Geração de relatórios** com todos os artefatos coletados.

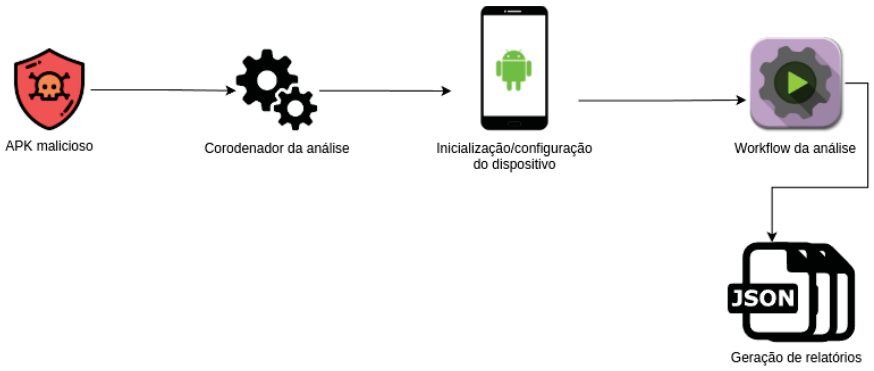


Figura 3.1: Visão geral do pipeline de análise de malware.

Este processo de alto nível é detalhado na Figura 3.2, que ilustra o pipeline interno do Motor de Análise. Ele é dividido em quatro fases sequenciais, garantindo que cada etapa, desde a preparação do ambiente até a compilação final do relatório, seja executada de forma ordenada e robusta.

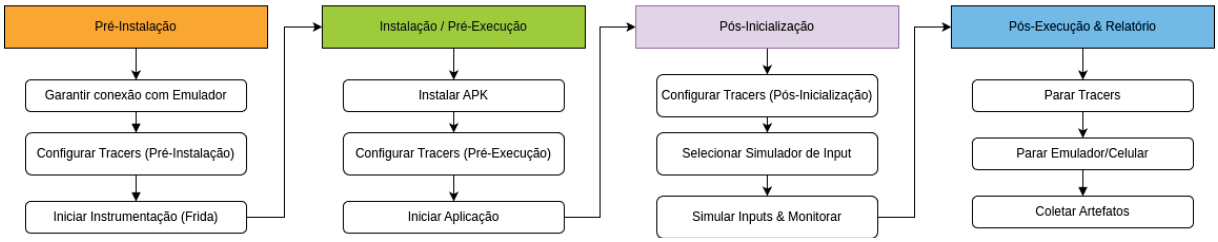


Figura 3.2: Detalhamento do *Workflow de Análise* em quatro fases: Pré-instalação, Instalação/Pré-Execução, Pós-Inicialização e Pós-Execução & Relatório.

Fase 1: Pré-Instalação. Nesta fase inicial, o motor prepara o ambiente. A ação `Garantir conexão com Emulador` é executada, onde o sistema inicia ou reconecta ao dispositivo, restaurando um *snapshot* limpo se disponível, com base nas diretivas da seção `Emulator` do YAML, como `avd_name` e `snapshot_name`. Em seguida, a etapa `Configurar Tracers (Pré-Instalação)` ativa os *tracers* definidos na lista `pre_installation_tracers` do YAML. Como exemplificado na Listagem 1, a inicialização de ferramentas como o `tcpdump` nesta fase é estrategicamente importante para capturar todo o tráfego de rede desde o primeiro momento, incluindo possíveis comunicações estabelecidas pelo próprio processo instalador do Android. Por fim, se a chave `use_frida: true` estiver habilitada na seção `Instrumentation`, a ação `Iniciar Instrumentação (Frida)` garante que o `frida-server` apropriado esteja em execução, preparando o terreno para as instrumentações que ocorrerão nas fases subsequentes.

Fase 2: Instalação/Pré-Execução. Com o ambiente pronto, a ação `Instalar APK` é realizada, utilizando o método especificado em `install_method` (e.g., `adb`). O sucesso da instalação é verificado, podendo detectar falhas como `INSTALL_FAILED_NO_MATCHING_ABIS` e acionar o *Error Handler*. Depois, `Configurar Tracers (Pré-Execução)` inicia *tracers* que monitoram eventos entre a instalação e a execução, conforme a lista `pre_launch_tracers`. A Listagem 2 ilustra a ativação do `logcat`, uma ferramenta indispensável para capturar mensagens do sistema e do aplicativo que podem indicar problemas de inicialização ou comportamentos anômalos precoces. A fase culmina com `Iniciar`

Listagem 1 Exemplo de configuração YAML para um tracer de pré-instalação. O `tcpdump` é iniciado para monitorar a rede antes mesmo da instalação do APK. A variável `$OUTPUT_FILE` é resolvida em tempo de execução para um caminho de saída padronizado.

```
# analysis.yaml: Seção de tracers pré-instalação
Analysis:
  pre_installation_tracers:
    - name: tcpdump
      is_system_tracer: true
      binary_path: /system/bin/tcpdump
      command_args: "-i any -w ${OUTPUT_FILE}"
      output_extension: "pcap"
      required: false
```

`Aplicação`, que lança o app, seja diretamente via `uiautomator` ou através do Frida (`launch_method: frida`) para permitir a instrumentação desde a primeira instrução, conforme definido no YAML.

Listagem 2 Exemplo de configuração YAML para iniciar o `logcat` após a instalação, mas antes da primeira execução do aplicativo. Isso garante a captura de logs relacionados à inicialização da aplicação.

```
# analysis.yaml: Seção de tracers pré-execução
Analysis:
  pre_launch_tracers:
    - name: logcat
      is_system_tracer: true
      binary_path: /system/bin/logcat
      command_args: "-v threadtime -f ${OUTPUT_FILE}"
      output_extension: "txt"
      required: false
```

Fase 3: Pós-Inicialização. Com o aplicativo ativo, a análise dinâmica começa. `Configurar Tracers (Pós-Inicialização)` atrela *tracers* ao processo do app, especificados em `post_launch_tracers`. Em seguida, `Selecionar Simulador de Input` carrega o método de interação com a UI especificado pela chave `input_method` no YAML. A Listagem 3 demonstra uma configuração poderosa: o método de input é o `monkey`, cujos parâmetros são detalhados na seção `InputMonkey` (e.g., `throttle`, `pct_touch`), enquanto o tracer `strace` é configurado para se anexar ao processo do app através da variável `$APP_PID`. Esta variável é um placeholder que o ARTEMIS resolve dinamicamente, demonstrando a capacidade do sistema de orquestrar ferramentas de forma contextual. A ação `Simular Inputs & Monitorar` executa a interação com a UI enquanto os *tracers* coletam dados e scripts Frida podem detectar técnicas de evasão. Um *Heartbeat Monitoring* verifica a vitalidade do app.

Fase 4: Pós-Execução & Relatório. A fase final consolida os dados. `Parar Tracers` encerra todos os processos de monitoramento de forma segura. `Parar Emulador/Celular`

Listagem 3 Exemplo de YAML configurando um método de input (`monkey`) e um tracer pós-inicialização (`strace`). O `strace` é dinamicamente anexado ao PID do aplicativo analisado, e o comportamento do `monkey` é finamente ajustado por seus próprios parâmetros.

```
# analysis.yaml: Configurando input e tracers pós-inicialização
Analysis:
  input_method: monkey
  post_launch_tracers:
    - name: strace
      is_system_tracer: true
      binary_path: /system/bin/strace
      command_args: "-o ${OUTPUT_FILE} -f -tt -v -p ${APP_PID}"
      output_extension: "txt"
      required: true

InputMonkey:
  timeout_sec: 180
  throttle: 200
  pct_touch: 80
  pct_motion: 10
  seed: 3
  ignore_crashes: true
```

desliga ou desconecta o dispositivo. A ação `Coletar Artefatos` realiza o download de todos os arquivos gerados (logs, pcaps, traces) para uma pasta local, cujo caminho foi definido pela variável `output_folder` na seção `General` do YAML. Por fim, um relatório consolidado é compilado e enviado ao Backend, juntamente com a cópia exata do arquivo de configuração YAML utilizado, garantindo a rastreabilidade e a reprodutibilidade da análise.

3.1.3 Backend

O *Backend* da plataforma, implementado com Flask e PostgreSQL e apoiado por Redis e MinIO, gerencia o ciclo de vida das análises, armazena os resultados e expõe uma API RESTful. Seus componentes são ilustrados na Figura 3.3.

Os principais componentes incluem um **Servidor Flask** (a API), um banco de dados **PostgreSQL** (BD Postgres), um **MinIO Object Storage**, **Redis** (Fila Redis) e o **Gerenciador de Workers**.

Fluxo Operacional no Backend. Conforme a Figura 3.3, quando um **Usuário** faz o **(1) Upload de APK**, a API primeiro **(2) Armazena o APK** no MinIO e **(3) Cria um registro** no BD Postgres. Em seguida, a API vai **(4) Enfileirar a tarefa** na Fila Redis. O Redis **(5) Atribui a tarefa** a um Gerenciador de Workers disponível. O worker inicia o trabalho, primeiro fazendo o **(6) Obter APK** do MinIO para então **(7) Iniciar a análise** no pipeline. Ao final, os resultados são processados: **(8a) Armazena relatórios** no MinIO, **(8b) Atualiza métricas** e **(8c) Atualiza status** no BD Postgres. Posteriormente, o usuário pode fazer uma **(9) Request dos resultados**, e a API irá **(10a) Obter dados da análise** do BD Postgres e **(10b) Obter relatórios** do MinIO para apresentar ao usuário.

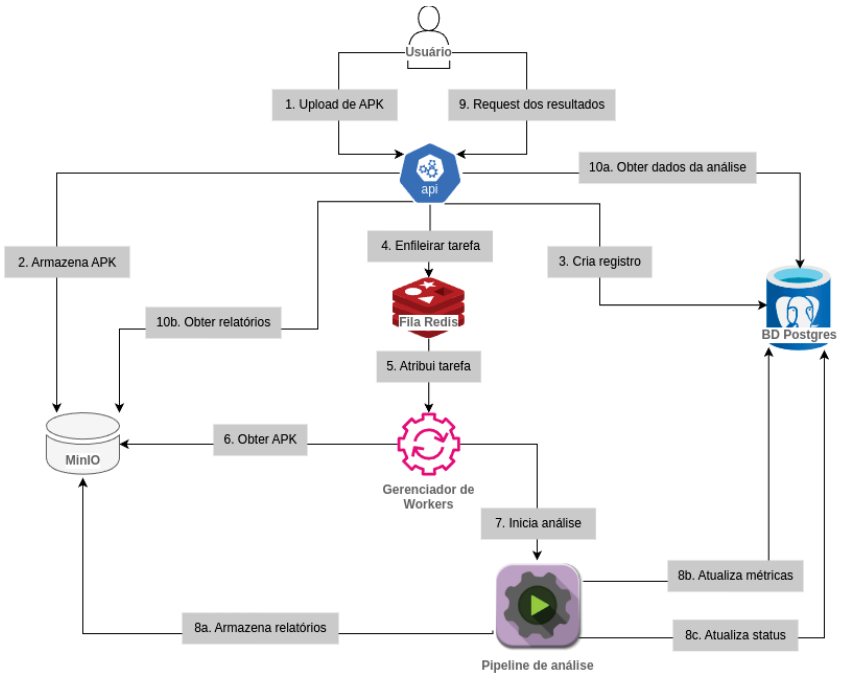


Figura 3.3: Fluxo de dados do Backend: (1) usuário envia APK; (2–3) Backend armazena binário no MinIO e cria registros no PostgreSQL; (4–5) tarefa é enfileirada no Redis e atribuída a um worker; (6–7) worker obtém APK e executa análise; (8a–c) resultados são armazenados no MinIO e métricas atualizadas no PostgreSQL; (9–10) usuário consulta resultados e artefatos via API.

4 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.

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.

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 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, **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. Esses números demonstram a complexidade inerente ao processo de análise em larga escala de aplicativos Android, evidenciando que quase 40% das amostras encontram algum obstáculo que impede a conclusão do diagnóstico. Tal taxa de falhas reforça a necessidade de um framework de análise que cubra amplamente casos de erro diversos, de modo a maximizar a cobertura e minimizar os falsos-negativos.

5.1 ANÁLISE DE METADADOS ESTÁTICOS

Para extrair estaticamente metadados críticos às fases de análise, utilizamos um pipeline em camadas (Tabela 5.1), 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. Ao combinar ferramentas de diferentes sensibilidades e abordagens, foi possível mitigar a fragilidade de cada método isolado e recuperar os campos críticos em cenários adversos, demonstrando que a cooperação entre ferramentas especializadas eleva significativamente o recall global dos atributos.

Tabela 5.1: 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

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). Esse salto na taxa de recuperação ilustra o valor de "fatiar" o problema e empregar múltiplas camadas de análise. Ferramentas rápidas como `aapt` capturam a maioria das informações triviais, mas as mais robustas — embora mais lentas — resgatam metadados críticos quando há ofuscação ou empacotamentos não convencionais. Sem esse *fallback* em camadas, muitos APKs teriam campos não extraídos, comprometendo fases seguintes de análise e reduzindo a cobertura global do framework.

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

5.1.0 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`).

Esses índices revelam que a maioria das falhas ocorre ainda na fase de instalação, destacando a relevância de mecanismos de *fallback* para compatibilidade de ABI e assinatura, pois, sem isso, mais de 59% das amostras seriam descartadas precocemente. Além disso, a ocorrência de 34,33% de `launch_error` mostra que nem toda APK instalada necessariamente é executável em ambiente controlado: fatores como bibliotecas ausentes e proteções internas podem interromper a inicialização, justificando a inclusão de verificações estáticas e dinâmicas adicionais antes de prosseguir com a instrumentação. As falhas de *runtime_tracer* e desconhecidas, ainda que menos frequentes, apontam lacunas na robustez do pipeline, reforçando a necessidade de estratégias de triagem reversa para identificar padrões e reduzir retrabalho computacional.

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. Essa abordagem acumulativa não apenas acelera reaplicações de testes, mas também permite coletar estatísticas de falha detalhadas para refinar heurísticas de detecção precoce de problemas.

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 5.2 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.

É relevante notar que, dentre as 2.909 falhas de instalação, 2.840 (97,63%) se devem a incompatibilidade de ABI, o que demonstra, por si só, a magnitude do desafio imposto pela fragmentação de arquiteturas no ecossistema Android. Apenas 1,75% dos casos se referem a splits ausentes e menos de 1% a falhas desconhecidas ou de `sharedUserId`, indicando que, após resolver a questão de ABI, a plataforma atinge quase 99% de cobertura nessa fase. Essa estatística reforça que um framework que não trate adequadamente múltiplas ABIs estaria perdendo quase toda a amostra testável logo na etapa inicial, comprometendo toda a avaliação subsequente.

Tabela 5.2: 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%

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.

A Tabela 5.3 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); **both_x86_and_arm** inclui apenas bibliotecas compiladas para a arquitetura x86; **arm64_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 5.3: 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

Observando os dados, percebe-se que mais da metade dos APKs (51,97%) não possui bibliotecas nativas (categoria `no_abis`), o que indica que, em tese, esses aplicativos não dependeriam de emulação ou *fallback* específico para código nativo. No entanto, mesmo entre esses, a taxa de sucesso de 72,90% revela que aproximadamente 27% falharam por outros motivos (por exemplo, proteções anti-emulador, lógica interna de instalação ou bibliotecas incorretas). Já os APKs `both_x86_and_arm`, que poderiam rodar nativamente em emuladores x86, apresentaram 91,81% de sucesso, mas com tempo médio de execução mais elevado (603,49 s), em função da sobrecarga de incluir múltiplas arquiteturas.

As categorias que incluem apenas código ARM (`arm64_only`, `arm_only` e `arm64_and_arm`) falharam quase completamente em ambiente x86, comprovando a limitação do emulador sem um *fallback* dedicado a hardware físico. A única exceção é o conjunto `x86_only`, com apenas 16 APKs, mas que obteve 100% de sucesso, pois não há empecilhos arquiteturais para a execução. Esses resultados reforçam que, sem um mecanismo eficiente de detecção e redirecionamento para dispositivos físicos, o framework deixaria de analisar mais de 18% dos APKs (soma dos grupos ARM) já na etapa de instalação/emulação inicial.

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, seja pela ausência de bibliotecas específicas ou pela presença de proteções de segurança.

A Tabela 5.4 mostra as principais bibliotecas necessárias para a execução desses APKs.

Tabela 5.4: 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

A predominância de bibliotecas como `libbreakpad-core.so` (134 ocorrências) e `libhermes.so` (120 ocorrências) sugere que muitos aplicativos dependem de ambientes de depuração ou de runtimes específicos (no caso, o Hermes para React Native). A ausência desses binários ocasiona falhas críticas na carga do processo de UI, implicando que somente a extração estática não seria suficiente para identificar a necessidade desses artefatos. Desta forma, é imprescindível que o framework efetue verificações dinâmicas adicionais ou possua estratégias de *fallback* que incluam bibliotecas essenciais em tempo de execução, garantindo maior cobertura e reduzindo o número de `launch_errors`.

Ao se fazer a análise dos tempos de execução para as 12.466 APKs, observou-se **Tempo Médio** de 521,45 segundos (mediana de 513,67 s, mínimo de 88,56 s e máximo de 1.649,26 s), considerando que as **análises bem-sucedidas (7.590)** duraram 600,60 s em média enquanto que as **análises falhas (4.876)** duraram 397,9 s em média. Já os abortos precoces, isto é, execuções que duraram $\approx$<400 s, reduziram o tempo médio em aproximadamente 33,8%, ao passo que indicaram onde ajustes na estratégia de *fallback* são cruciais. Esses valores mostram que as falhas ocorrem majoritariamente em estágios iniciais da análise — seja na instalação, lançamento ou configuração do *tracer* — o que explica sua duração média significativamente menor (397,9 s). Por outro lado, as análises bem-sucedidas atingem quase o limite de tempo máximo configurado (600,60 s), pois percorrem todas as fases até a coleta completa de artefatos e relatórios. Em particular, verificou-se que o tipo de falha mais custoso (507,23 s) é o `runtime_tracer_error`, indicando que erros durante a instrumentação ativa consomem recursos substanciais antes de abortar, o que reforça a necessidade de identificar esses problemas o mais cedo possível, reduzindo ciclos desperdiçados. Registrar relatórios em todas as fases permite triagem reversa automatizada e reexecuções muito mais eficientes, realocando esforços de computação para APKs ainda não processados ou para novas configurações de teste.

Na Tabela 5.5 temos o tempo médio por tipo de erro.

Ao correlacionar os tempos médios com as quantidades, observa-se que, embora o `apk_installation_error` seja o mais frequente (2.909 ocorrências), seu tempo médio (339,62 s) é inferior ao dos demais, refletindo o fato de muitas instalações falharem cedo, antes de avançar para instrumentações pesadas. Já o `launch_error` (1.674 ocorrências) consome em média 481,08 s, pois envolve a inicialização completa da app até o momento em que falha, o que demanda mais tempo de CPU e uso de recursos. O `runtime_tracer_error`, embora

Tabela 5.5: 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

menos frequente (280 ocorrências), atinge o maior tempo médio (507,23 s), apontando que problemas durante a coleta ativa de traces são particularmente custosos. Esse cenário justifica a adoção de mecanismos de detecção precoce dentro do tracer e a capacidade de trocar para abordagens menos intrusivas assim que um padrão de falha for identificado.

Como visto na Tabela 5.3, foram calculados os tempos médios de execução por ABI, levando à conclusão que `arm_only` é a configuração mais rápida (290,79 s) e `both_x86_and_arm` é a mais lenta (603,49 s), 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. Esse acréscimo médio por nível de API indica que, para manter a escalabilidade do framework, é necessário prever margens de tempo adicionais ao planejar a infraestrutura de análise, pois versões mais novas do Android naturalmente exigem mais ciclos de CPU para verificação de políticas de segurança e execução de novos componentes de sistema. Sem esse dimensionamento, haveria um risco real de sobrecarregar ambientes de teste e introduzir *bottlenecks* de fila.

5.1.0 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*.

O alto percentual de uso de técnicas de evasão revela que, se o framework dependesse exclusivamente de `strace`, quase 9 em cada 10 aplicativos não forneceriam dados de execução, provocando um viés nas análises estáticas e dinâmicas restantes. A capacidade de detectar o uso de estratégias de anti-emulator e ajustar a instrumentação dinamicamente é, portanto, fundamental para preservar a representatividade da amostra e evitar que análises privilegiem apenas aplicativos menos sofisticados em termos de proteções.
Impacto da Instrumentação Intrusiva. Dos APKs com proteção Anti-Debug, 1.674 falharam em gerar relatórios, indicando que interromperam sua execução ao detectar tentativas 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.

Esses resultados quantificam o ganho de cobertura que a instrumentação menos intrusiva proporciona: 80,20% das amostras que, inicialmente, se recusavam a rodar sob `strace` foram resgatadas. Considerando que tais aplicativos costumam implementar checagens sofisticadas de depuração, a adoção de `ftrace` ou outras técnicas de VMI (Virtual Machine Introspection) se mostra essencial para reduzir a taxa de falsos-negativos resultantes de defesas ativas.

5.1.0 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.

Essa recuperação quase total de amostras por meio de rotas alternativas demonstra que uma infraestrutura de análise que não considere *fallback* em hardware real tende a subestimar o universo de APKs testáveis em mais de 2.800 amostras, comprometendo a validade estatística de quaisquer métricas calculadas e limitando o conjunto de dados a apenas as poucas dezenas que realmente rodam no emulador x86_64. A adoção de dispositivos ARM serve, portanto, como camada obrigatória para assegurar a completude das análises.

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.

Essa observação corrobora que, mesmo após contornar incompatibilidades arquiteturais, ainda há falhas relacionadas à estrutura lógica dos APKs (como módulos faltantes ou certificação inválida) que demandam novas camadas de cheque estático/dinâmico. Um framework com cobertura otimizada deve, portanto, prover não apenas emulação e transferência de execução para hardware real, mas também checagens adicionais de integridade e composição de pacotes.

5.1.0 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. Por exemplo, detectou-se que certos triggers anti-emulador só são ativados em versões superiores a 12, enquanto outros fluxos de inicialização divergem entre 10 e 11. Essas nuances somente são perceptíveis quando se avalia a mesma amostra em múltiplas versões simultaneamente, comprovando que a escalabilidade e

historicidade de execuções paralelas são requisitadas para capturar a totalidade dos cenários de comportamento.

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). Essa estatística ressalta que, mesmo com uma ampla cobertura de extração estática e instrumentação em emulador x86, mais de metade das falhas só podem ser corrigidas ao recorrer a hardware ARM real, confirmando a indispensabilidade dessa etapa para qualquer framework que vise alta completude.

(F2) **Instrumentação furtiva.** O *pipeline* centrado em `ftrace` recuperou 80,20% das amostras anti-debug que falhavam com *ptrace*. Isso demonstra que, sem técnicas de instrumentação que operem abaixo do nível de detecção do aplicativo, cerca de 1 em cada 5 amostras equipadas com proteções seriam permanentemente ignoradas, gerando um viés na análise de segurança.

(F1–F3) **Triagem reversa.** Artefatos de falha, desde a Fase 1, orientam reexecuções direcionadas, economizando recursos. Ao armazenar logs, traces e relatórios parciais, o pipeline consegue inferir rapidamente a causa raiz e reexecutar apenas as amostras que precisem de outro perfil ou estratégia, evitando repetições desnecessárias de etapas já concluídas com sucesso.

(F0–F4) **Histórico cumulativo.** Armazenar todos os relatórios viabiliza análises longitudinais e auditoria forense. O compilado histórico de execuções, abarcando configurações de ABI, versões de API e perfis de instrumentação, permite reconstruir cenários de ataque, validar hipóteses de evasão e até mesmo reprojetar políticas de segurança para firmwares futuros.

(F1–F4) **Escalabilidade elástica.** Microsserviços e filas permitem crescimento sem refatorações profundas. Graças à arquitetura modular e ao uso de filas para balanceamento de carga, novas máquinas ou instâncias podem ser adicionadas conforme exigido pela demanda, sem interromper o pipeline ou provocar gargalos críticos.

5.2.0 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. Essas considerações ilustram que, apesar da ampla cobertura alcançada, nenhum framework é isento de limitações. A escolha de usar `Monkey` para estímulo de UI, por exemplo, impede o acionamento de rotas condicionais que exigem interação humana específica, o que pode resultar em falsos-negativos ao não acionar funcionalidades críticas. Além disso, a implantação em cluster e concorrência intensiva pode introduzir variações de tempo que apps exploram para detectar anomalias (por exemplo, medindo latência de I/O). Quanto ao viés de amostragem, a predominância de APKs disponíveis no AndroZoo pode não refletir plenamente o perfil de aplicativos de mercados específicos (por exemplo, lojas regionais). Por fim, a evolução contínua do Android demanda revisões periódicas do framework, de modo a incorporar novas APIs,

mecanismos de segurança e formatos de pacote introduzidos a cada nova versão, sob pena de a plataforma se tornar rapidamente obsoleta.

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 maliciosos 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.

6.0 TRABALHOS FUTUROS

Diversas direções de pesquisa e evolução da plataforma foram identificadas a partir das limitações e observações obtidas ao longo deste trabalho. A seguir, listam-se possíveis abordagens e extensões para ARTEMIS, sem detalhamento exaustivo, mas apontando caminhos promissores:

- **Aumento da evasão de detecção de sandboxes:** Malwares podem reconhecer padrões característicos de sandboxes e métodos de análise comumente empregados. Assim, tornar a infraestrutura mais heterogênea — combinando diferentes técnicas de instrumentação, artefatos de sistema e perfis de execução — ajuda a dificultar a identificação do ambiente de teste pelos códigos maliciosos.

- **Integração de fluxo de análise manual para analistas:** Atualmente, a maior parte das etapas é automatizada, gerando logs e relatórios finais. Uma interface que permita ao analista conectar-se em tempo real (por exemplo, através de streaming de vídeo do Android) e interagir manualmente com o aplicativo (abrir telas, inserir credenciais, navegar nos menus) pode complementar a análise automática, permitindo observar comportamentos não acionados por estímulos genéricos.

- **Customização de cenários de execução:** Permitir configurações declarativas para simular elementos externos ao dispositivo diminui o viés de execução padrão. Exemplos incluem: simular geolocalização específica, emular um número de telefone SIM ativo, inserir contatos na agenda ou arquivos pré-existentes (imagens, documentos) antes de rodar o malware. Essas variações podem revelar comportamentos condicionados a dados presentes no dispositivo.

- **Suporte a emuladores ARM:** Além do fallback para dispositivos físicos ARM64, testar e adicionar suporte a emuladores baseados em ARM na nuvem fornece uma camada intermediária de análise mais próxima do ambiente real, sem depender exclusivamente de hardware físico. Essa ampliação pode ser feita por meio de instâncias ARM em provedores de nuvem ou por contêineres especializados.

- **Registro de contexto de execução:** Adicionar um campo nos relatórios que indique se a análise foi realizada em emulador ou em dispositivo físico, e ainda qual arquitetura (ARM/x86) foi utilizada, permite reunir estatísticas mais precisas sobre desempenho, taxa de falhas e evasões específicas a cada plataforma. Esse metadado facilita o gerenciamento de um pool de dispositivos físicos (com IDs pré-definidos) e emuladores (com portas configuradas).

- **Permitir scripts de eventos personalizados:** Oferecer ao usuário a possibilidade de fornecer um script (por exemplo, em Python) que defina sequências de eventos, toques e entradas específicas — substituindo ou complementando estímulos genéricos como Monkey — pode acionar fluxos internos do aplicativo que exigem sequência de interações mais elaborada. Isso amplia a cobertura de código e aumenta a chance de revelar comportamento malicioso condicionado.

- **Fallback baseado em versão do Android:** Inspirado em sistemas como Joe Sandbox, implementar lógica que, ao não detectar comportamento suspeito em uma versão (por exemplo, Android 10), automaticamente reexecute o mesmo aplicativo em uma API anterior (por exemplo, Android 7) — ou vice-versa — permite capturar gatilhos condicionados ao SDK. Essa abordagem adaptativa maximiza a probabilidade de acionar funcionalidades escondidas.

- **Interface gráfica para analista com controle remoto:** Em futuras evoluções, prover ao analista uma janela com streaming de vídeo do dispositivo (emulador ou físico) e permitir controle de mouse/teclado (por exemplo, via emulador Cuttlefish ou soluções similares) possibilita ações manuais mais complexas, como entrada de credenciais, navegação em fluxos de login ou uso de teclados personalizados, trazendo maior fidelidade à experiência de um usuário real.

- **Uso de técnicas de visão computacional para direcionamento de inputs:** A aplicação de métodos como DeepSeek para analisar as telas do aplicativo em tempo real e sugerir entradas ou cliques específicos pode orientar de forma inteligente a exploração de fluxos de UI complexos, reduzindo o tempo de configuração manual e aumentando a eficácia no acionamento de comportamentos condicionados a determinados componentes visuais.

REFERÊNCIAS

(2020). *Mobile Security Framework (MobSF) Documentation - Dynamic Analysis*. MobSF Team. Accessed 12-Jan-2022.

(2021). *CuckooDroid: Android Malware Analysis Module (Documentation)*. Checkpoint Software Technologies. Accessed 18-Jan-2022.

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

Appknox (2024). Why Open Source MobSF Isn't Enough for Mobile App Security Testing?

Baggio, A., Maiorca, D., Giacinto, G., Almadori, L., Stivaletti, S., Lain, D. e Torre, D. D. (2023). Unmasking the veiled: A comprehensive analysis of android evasive malware. Em *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security (AsiaCCS)*.

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

Developers, G. A. (2014). Android runtime (art) and dalvik. `https://source.android.com/devices/tech/dalvik/`.

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. e Sheth, A. N. (2010a). TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. Em *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, páginas 393–407.

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

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

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

Hacking Articles (2021). Android Pentest: Automated Analysis using MobSF.

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

Lab, K. (2024). Mobile malware evolution 2024. Relatório técnico, Kaspersky Lab.

Lantz, P. (2011). An android application sandbox for dynamic analysis. Dissertação de Mestrado, Lund University. M.S. Thesis.

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. e Vigna, G. (2015). Baredroid: Large-scale analysis of android apps on real devices. Em *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, páginas 71–80.

Networks, U. P. A. (2024). Dynamic android malware analysis: Hooking framework design. https://unit42.paloaltonetworks.com/.

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

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

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

Research, T. M. (2021). Evasive malware techniques targeting android. Relatório técnico, Trend Micro.

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

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

Series, R. C. (2025). Rldroid: Reinforcement learning for android ui exploration. https://conf.researchr.org/.

SL, U. T. (2024). Android report 2024: Most used brands, browsers and os versions. Accessed: 2025-06-04.

Statista (2024). Distribution of android versions worldwide. https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/.

Sun, M., Wei, T. e Lui, J. C. S. (2016). TaintART: A practical multi-level information-flow tracking system for android runtime. Em *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, páginas 331–342.

Tam, K., Khan, S., Fattori, A. e Cavallaro, L. (2015a). CopperDroid: Automatic reconstruction of android malware behaviors via VMI. Em *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

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

van der Veen, V., Fratantonio, Y., Baggili, I. e et al. (2014). ANDRUBIS - 1,000,000 apps later: A view on current android malware behaviors. Em *Proc. of the 3rd Intl. Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*.

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

Yan, L. e Yin, H. (2012a). DroidScope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. Em *Proceedings of the 21st USENIX Security Symposium*.

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

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