UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Marcos Douglas Rebelo de Moraes

Técnicas de Ocultação de Root: Uma Avaliação de Efetividade

Recife

Agosto de 2025

Marcos Douglas Rebelo de Moraes

Técnicas de Ocultação de Root: Uma Avaliação de Efetividade

Artigo apresentado ao Curso de Bacharelado em Sistemas de Informação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Universidade Federal Rural de Pernambuco – UFRPE
Departamento de Estatística e Informática
Curso de Bacharelado em Sistemas de Informação

Orientador: Lidiano Augusto Nobrega de Oliveira

Recife
Agosto de 2025

Técnicas de Ocultação de Root: Uma Avaliação de Efetividade

[Marcos Douglas Rebelo de Moraes] [1], **[Lidiano Augusto Nobrega de Oliveira]** [1]

[1]Departamento de Estatística e Informática – Universidade Federal Rural de Pernambuco
Rua Dom Manuel de Medeiros, s/n, - CEP: 52171-900 – Recife – PE – Brasil

marcos.moraes@ufrpe.br, lidiano.oliveira@ufrpe.br

Resumo. *Este estudo apresenta uma análise prática sobre a efetividade de técnicas de ocultação de Root frente a mecanismos de detecção empregados por aplicativos Android. Foram avaliadas diferentes abordagens de ocultação do acesso Root, utilizando ferramentas como Magisk, KernelSU e APatch, em conjunto com módulos como Zygisk Next e Zygisk Assistant. Os testes foram realizados em dispositivos reais, com as versões 11 e 15 do sistema Android, e tiveram como objetivo verificar a capacidade dos aplicativos em identificar a presença de Root. Os resultados demonstraram que, embora alguns aplicativos empreguem mecanismos avançados de proteção — como soluções RASP —, todas as técnicas de detecção foram subvertidas com sucesso por pelo menos uma das abordagens avaliadas. Tais achados reforçam que a detecção de Root, embora útil, é insuficiente como mecanismo isolado de segurança. Assim, ressalta-se a importância da adoção de práticas como desenvolvimento seguro e análises recorrentes de segurança ao longo de todo o ciclo de vida da aplicação.*

Palavras-chave*: Android, Root, Detecção de Root, Ocultação de Root, Magisk, Apatch, KernelSU.*

Abstract. *This study presents a practical analysis of the effectiveness of Root hiding techniques against detection mechanisms employed by Android applications. Different approaches to hiding Root access were evaluated using tools such as Magisk, KernelSU, and APatch, in conjunction with modules like Zygisk Next and Zygisk Assistant. Tests were conducted on real devices running Android versions 11 and 15, aiming to assess the applications' ability to detect the presence of Root. Results showed that, although some applications employ advanced protection mechanisms — such as RASP solutions — all detection techniques were successfully bypassed by at least one of the evaluated hiding methods. These findings reinforce that Root detection, while useful, is insufficient as a standalone security mechanism. Therefore, the adoption of practices such as secure development and ongoing security assessments throughout the application lifecycle is strongly recommended.*

Keywords*: Android, Root, Root Detection, Root Hiding, Magisk, Apatch, KernelSU.*

1. Introdução

Estudos indicam que uma parcela significativa dos aplicativos Android apresenta vulnerabilidades conhecidas. Um relatório da [Synopsys 2021] revelou que 63% dos aplicativos populares analisados em 2021 continham componentes com falhas de segurança

conhecidas, destacando o estado crítico da segurança em dispositivos móveis e a constante corrida entre técnicas de proteção e subversão.

Muitos usuários optam por habilitar o acesso *Root* em seus dispositivos Android por diversas razões [NordVPN 2023]. Uma das principais motivações é a possibilidade de remover aplicativos pré-instalados pelo fabricante (*bloatware*), que normalmente não podem ser desinstalados e consomem recursos do sistema. O *Root* também permite personalizações profundas, como a modificação de temas, animações e ícones, além de ajustes finos no comportamento do sistema operacional. Outra razão comum é o acesso a funcionalidades avançadas por meio de aplicativos que requerem permissões de superusuário, como *firewalls*, bloqueadores de anúncios em nível de sistema e ferramentas de automação.

Usuários também recorrem ao *Root* para otimizar o desempenho e a autonomia da bateria, controlando processos em segundo plano e ajustando o uso energético. De mesmo modo, aqueles que desejam manter seus dispositivos atualizados, mesmo após o fim do suporte oficial, também utilizam *Root* para corrigir verificações de integridade resultantes da instalação de ROMs personalizadas. Por fim, o acesso *Root* possibilita a realização de *backups* completos do sistema e dos dados do usuário, algo inviável com as permissões padrão do Android.

Com o forte avanço da transformação digital, como destacado por [Transformação Digital 2019], também houve um aumento significativo no número de aplicativos voltados para as mais diversas finalidades. Ao mesmo tempo, a crescente preocupação com a segurança no ambiente digital — evidenciada por iniciativas como a Lei Geral de Proteção de Dados (LGPD) — impulsionou o desenvolvimento de mecanismos de proteção contra dispositivos modificados [Talsec 2024].

Entre esses mecanismos de proteção, destaca-se a detecção de *Root* no Android, amplamente utilizada como medida preventiva para mitigar riscos relacionados à integridade e segurança dos dados. Dispositivos com acesso *Root* oferecem permissões elevadas que permitem a modificação arbitrária do comportamento de aplicativos, interceptação de dados sensíveis, injeção de código malicioso e subversão de mecanismos de segurança, como criptografia ou autenticação. Assim, ao detectar a presença de *Root*, os aplicativos podem restringir funcionalidades, recusar a execução ou acionar protocolos de segurança, protegendo-se contra fraudes, engenharia reversa, roubo de dados e outros tipos de abuso.

Todavia, entre os usuários que optam por utilizar dispositivos com *Root* habilitado, surgiu a demanda por formas de contornar essas restrições sem abrir mão das permissões elevadas [ValueMentor 2024]. Para atender a essa demanda, diversas técnicas e ferramentas foram e vêm sendo desenvolvidas com o objetivo de preservar a funcionalidade do *Root* ao mesmo tempo em que evitam a detecção por parte dos aplicativos.

Tradicionalmente, tais técnicas envolvem o uso de engenharia reversa com análise estática e instrumentação dinâmica para modificar o comportamento dos aplicativos, desativando mecanismos de detecção ou alterando bibliotecas internas. No entanto, trata-se de um processo manual que frequentemente demanda um tempo considerável e exige alto grau de conhecimento técnico. Além disso, em determinadas situações, essas alterações podem comprometer o funcionamento do aplicativo e nem sempre garantem compatibilidade com futuras atualizações.

Juntamente às técnicas de engenharia reversa, também surgiu uma abordagem mais flexível e genérica: modificar o próprio sistema operacional Android para ocultar a presença do *Root*, mantendo os aplicativos intactos. Essa estratégia permite que o ambiente de execução aparente estar em um dispositivo íntegro e sem *Root*, ainda que com permissões de superusuário habilitadas. Com foco em ferramentas como Magisk [Wu 2025], KernelSU [Weishu 2025], APatch [bmax121 2025], bem como o uso de módulos como o Zygisk Next [Dr-TSNG 2025], que operam diretamente no sistema para interceptar ou desviar chamadas sensíveis utilizadas na detecção de *Root*.

O êxito dessas ferramentas pode variar ao longo do tempo, à medida que o sistema operacional Android evolui e novas técnicas de detecção são implementadas por desenvolvedores de aplicativos. Assim, ainda são poucos os trabalhos que analisam de forma prática e comparativa técnicas de ocultação de *Root* baseadas na modificação do sistema operacional Android. Por isso, este artigo tem como objetivo principal avaliar a efetividade dessas abordagens diante de aplicativos Android selecionados, com ênfase em aplicações relevantes, bancárias ou governamentais.

De modo a guiar o desenvolvimento do estudo, foram definidos os seguintes objetivos específicos:

- Descrever as principais técnicas utilizadas na detecção de *Root* por aplicativos, incluindo verificação de pacotes instalados, permissões de escrita em diretórios protegidos, presença de arquivos binários e análise de propriedades do sistema;
- Apresentar as técnicas de ocultação de *Root* Magisk, KernelSU, APatch e os módulos utilizados;
- Destacar a importância de práticas de desenvolvimento seguro e análises recorrentes de segurança, ressaltando a ineficácia da dependência exclusiva de mecanismos de detecção de *Root* como estratégia de proteção contra fraudes e violações de integridade.

2. Trabalhos Relacionados

Diversos estudos abordam os riscos associados ao uso de dispositivos Android com *Root*, assim como os mecanismos empregados por aplicativos para detectar e mitigar esse tipo de modificação. Esta seção apresenta os principais trabalhos relacionados ao tema, incluindo técnicas utilizadas para a obtenção de *Root*, métodos para sua ocultação e análises da efetividade das soluções de detecção de *Root* empregadas em aplicativos.

[Moraes and Vilela 2021] investigaram técnicas práticas de detecção e evasão de *Root* em dispositivos Android, avaliando a eficácia de aplicativos populares e propondo melhorias em métodos de detecção para fortalecer a segurança dos dispositivos. No entanto, a análise se restringiu ao uso do Magisk.

[Kamal et al. 2024] estudaram as mudanças sutis e significativas no comportamento e dados de dispositivos Android após a obtenção de *Root*, enfatizando vulnerabilidades que permitem burlar verificações de *Root* em aplicativos bancários e outras proteções, com foco no Android 13 e utilizando apenas o Magisk como mecanismo de ocultação de root.

[Sun et al. 2015] analisaram métodos de *rooting* em Android e realizaram uma avaliação prática da eficácia das técnicas de detecção de *Root* em 182 aplicativos, con-

cluindo que os métodos de detecção atuais são pouco eficazes e recomendando abordagens baseadas em kernels protegidos ou ambientes de execução confiáveis.

[Geist et al. 2016] investigaram técnicas de detecção e evasão de *root/jailbreak* em Android e iOS, concluindo que os métodos existentes são amplamente suscetíveis à subversão, inclusive manipulando diretamente instruções em linguagem assembly. Entretanto a subversão das detecções focou na implementação de técnicas manuais.

[Nguyen-Vu et al. 2017] pesquisaram a evolução da disputa entre técnicas de detecção e evasão de *Root* no Android, demonstrando que a maioria dos aplicativos de verificação são vulneráveis a técnicas como *hooking* de APIs e renomeação de arquivos. O estudo abordou o uso de técnicas antigas, como o RootCloak, uma técnica utilizada quando o *Root* ainda era obtido modificando a partição `System`.

[Kellner et al. 2019] examinaram a eficácia dos mecanismos de detecção de *jailbreak* em aplicativos bancários. O estudo revelou que 44% dos aplicativos analisados não implementavam qualquer forma de detecção. Isso os torna vulneráveis mesmo sem a necessidade de engenharia reversa ou manipulação direta, evidenciando uma confiança excessiva na segurança provida pelo sistema operacional.

[Soewito and Suwandaru 2022] demonstraram que a detecção de *Root* baseada em chamadas Java pode ser facilmente burlada por meio do *hooking* de funções, ressaltando a necessidade de técnicas de detecção mais robustas para proteger dados sensíveis. Foram utilizadas apenas técnicas manuais, visando modificar funcionalidades de detecção implementadas pelo aplicativo.

[Casati and Visconti 2018] analisaram diversas aplicações Android e mostraram que 51,6% delas apresentam falhas que podem levar ao vazamento de dados sensíveis em dispositivos com *Root*, expondo usuários a ataques como MITM (*man-in-the-middle*). Contudo, os

Com base nos trabalhos relacionados citados, é possível concluir que, embora todas as abordagens possuam um contexto em comum: a detecção de root, nenhuma delas se propôs a realizar uma análise de detecção de *Root* com foco em técnicas que não modifiquem diretamente o aplicativo e utilizando modificações diretamente no *kernel*.

3. Referencial Teórico

3.1. Android

O Android é um sistema operacional baseado no *kernel* Linux, desenvolvido inicialmente pela Android Inc. e adquirido pelo Google em 2005. Sua arquitetura é composta por diversas camadas, entre as quais se destacam: o *kernel* Linux, as bibliotecas nativas, o Android Runtime (ART), o framework de aplicação e as aplicações do usuário.

Apesar de ter herdado a base Unix de permissões e processos, o Android implementa mecanismos de segurança adicionais, com destaque para a *sandbox* de aplicativos. Na qual, cada processo é executado isoladamente, com um identificador de usuário (UID) exclusivo atribuído no momento da instalação. Esse isolamento evita que aplicativos acessem diretamente os dados uns dos outros, a menos que permissões explícitas sejam concedidas, promovendo uma separação robusta entre os componentes do sistema.

O modelo de permissões do Android também segue a abordagem declarativa,

exigindo que os aplicativos solicitem previamente o acesso a recursos sensíveis, como câmera, microfone, armazenamento ou localização. Essas permissões devem ser autorizadas pelo usuário, o que oferece uma camada adicional de controle e transparência.

Outro componente fundamental é o *Verified Boot*, mecanismo que verifica criptograficamente a integridade de cada etapa da inicialização do sistema, desde o *bootloader* até a partição do sistema. Caso seja detectada alguma modificação não autorizada — como a instalação de uma ROM personalizada ou a adulteração do *kernel* —, o sistema pode impedir a inicialização ou alertar o usuário.

O Android também incorpora o SELinux (*Security-Enhanced Linux*), uma extensão de segurança desenvolvida pela NSA (*National Security Agency*) para o kernel Linux e baseada em controle de acesso obrigatório (*Mandatory Access Control* — MAC). O SELinux restringe as ações que um processo pode executar, com base em um conjunto de políticas predefinidas. Mesmo processos com privilégios de *Root* são limitados por essas políticas, o que representa uma importante mitigação contra escalonamento indevido de privilégios e execução de código malicioso. No Android, o SELinux opera geralmente no modo *enforcing*, bloqueando qualquer ação que viole suas regras, em vez de apenas registrar o incidente. Essa abordagem garante uma aplicação rígida das políticas de segurança e dificulta significativamente a exploração de falhas no sistema.

Contudo, caso o dispositivo possua acesso *Root*, é possível modificar o modo de operação do SELinux, alterando-o de *enforcing* para *permissive*. Essa modificação reduz a rigidez das restrições impostas e permite que ações normalmente bloqueadas sejam executadas, o que, embora útil para fins de testes ou desenvolvimento, também representa um risco adicional à segurança do sistema.

Essas camadas de segurança, quando combinadas, criam um ambiente controlado e resistente a ameaças, dificultando a obtenção e o uso de privilégios elevados no sistema. Entretanto, em função da natureza aberta do sistema Android, o processo de obtenção de privilégios elevados (root) torna-se um tema recorrente e relevante, tanto do ponto de vista técnico quanto de segurança.

3.2. Root

O termo *Root* refere-se ao superusuário em sistemas operacionais baseados em Unix, como o Linux, sobre o qual o Android se fundamenta. Este usuário possui acesso irrestrito a todos os comandos e arquivos do sistema, sendo capaz de realizar qualquer modificação, inclusive aquelas que afetem a integridade e a segurança do ambiente operacional. O acesso *Root* é comumente associado à administração do sistema e, por esse motivo, é restrito por padrão, especialmente em sistemas voltados ao usuário final, como o Android.

Em termos técnicos, o funcionamento do *Root* está diretamente relacionado ao modelo de permissões implementado no *kernel* dos sistemas baseados em Unix. O *kernel*, ou núcleo do sistema operacional, é o componente central responsável por intermediar a comunicação entre o hardware e o software, além de gerenciar recursos como memória, processos e dispositivos de entrada e saída.

No Linux, o controle de acesso a arquivos, processos e dispositivos é baseado em identificadores de usuário (UIDs), identificadores de grupo (GIDs) e permissões

atribuídas a cada recurso. Cada arquivo ou diretório possui permissões específicas que determinam quem pode lê-lo (r), escrevê-lo (w) ou executá-lo (x). Essas permissões são organizadas em três níveis: dono (usuário), grupo e outros.

O superusuário, ou *Root*, é identificado pelo UID 0. Ao contrário dos demais usuários, o *Root* não está sujeito às mesmas restrições impostas pelo sistema de permissões, podendo acessar ou modificar qualquer recurso do sistema. Esse privilégio absoluto, embora necessário para tarefas administrativas, representa um risco potencial à segurança caso seja obtido de forma indevida. Processos executados com privilégios de *Root* têm autoridade total sobre o sistema, o que pode permitir a execução de códigos maliciosos com impacto elevado.

3.3. Obtenção de *Root* no Android

Por padrão, os dispositivos Android não oferecem acesso *Root* ao usuário final. Essa decisão foi tomada por razões de segurança, com o objetivo de proteger o sistema e os dados do usuário contra ações não autorizadas. Obter acesso *Root* significa escalar privilégios para atuar como superusuário (UID 0), adquirindo permissões administrativas que normalmente não estão disponíveis ao usuário. Esse processo, conhecido como "fazer *Root* no dispositivo", permite modificar arquivos protegidos do sistema, controlar recursos restritos, capturar dados de outros aplicativos, entre outras ações. Embora frequentemente utilizado por desenvolvedores, pesquisadores e entusiastas, o *Root* também representa um ponto sensível sob a ótica da segurança.

Existem diferentes métodos para obtenção de *Root* no Android, que variam conforme a versão do sistema, o fabricante do dispositivo e o nível de bloqueio do *bootloader*. O *bootloader* é o primeiro componente executado ao ligar o dispositivo, responsável por iniciar o sistema operacional. Em dispositivos modernos, ele é bloqueado por padrão para garantir que apenas versões oficiais e assinadas do sistema possam ser carregadas.

O método e os passos necessários para desbloquear o *bootloader* variam significativamente entre os fabricantes. Em geral, esse processo envolve a ativação das opções de desenvolvedor, o uso da interface *ADB* (Android Debug Bridge) e comandos específicos, como *fastboot oem unlock*. Contudo, cada fabricante pode impor suas próprias restrições: algumas marcas exigem o cadastro do usuário em plataformas como fóruns ou comunidades da empresa, envio de solicitações formais para obtenção de códigos de desbloqueio ou até mesmo se recusam a oferecer um método de desbloqueio, impossibilitando o *bootloader* de ser desbloqueado.

Após o desbloqueio do *bootloader*, o método tradicional para obter acesso *Root*, conhecido como *systemfull root*, modifica diretamente a partição `system`, alterando arquivos essenciais para conceder privilégios de superusuário. Embora funcional, essa abordagem traz consequências negativas como aumento da probabilidade de detecção por mecanismos de segurança, impedimento de atualizações oficiais e possíveis instabilidades no sistema.

Para superar essas limitações, surgiu a técnica conhecida como *systemless root*. Diferentemente do *systemfull root*, que altera permanentemente arquivos na partição do sistema, o *systemless root* evita qualquer modificação direta nessa partição crítica. Esse método concede acesso *Root* ao alterar o ambiente de inicialização do dispositivo, principalmente através da modificação da partição `boot`, que contém a imagem do *kernel*

e o ramdisk (um sistema de arquivos temporário utilizado durante a inicialização). Ao modificar o ramdisk ou introduzir camadas intermediárias — conhecidas como *mount namespaces* ou técnicas de sobreposição de sistema —, o *systemless root* injeta os binários e permissões necessárias para o acesso *Root*, sem tocar nos arquivos originais do sistema.

Essa abordagem não apenas preserva a integridade da partição do sistema, facilitando a instalação de atualizações oficiais e reduzindo a chance de detecções de integridade, como também torna o processo de *unroot* (remoção do *Root*) mais simples e menos invasivo, já que basta restaurar a imagem de *boot* original para remover o acesso *Root*.

3.4. Técnicas de Detecção de Root

Aplicativos que implementam detecção de *Root* frequentemente empregam múltiplas técnicas para identificar se o dispositivo Android está com acesso *Root* habilitado. Quando o resultado de uma detecção é positivo, o aplicativo pode considerar o dispositivo como comprometido e, dependendo da política de segurança adotada, limitar funcionalidades ou encerrar sua execução. A Figura 1 ilustra dois exemplos de mensagens de erro exibidas quando o acesso *Root* é detectado.

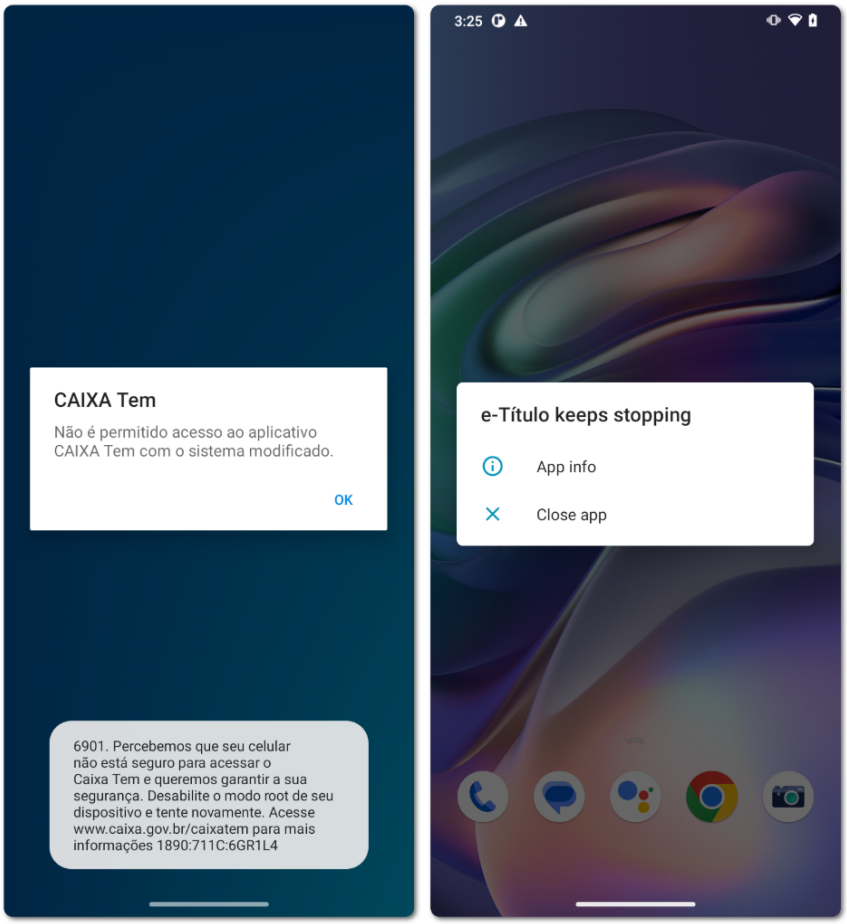

Figura 1. Exemplos de mensagens de erro na detecção de *Root*.

As abordagens mais tradicionais empregadas por esses mecanismos de detecção serão listadas a seguir.

3.4.1. Checagem de Aplicativos Instalados

Uma das técnicas mais diretas para detectar dispositivos com acesso *Root* é a verificação da presença de aplicativos comumente associados ao processo de *rooting*, realizada por meio da inspeção dos pacotes instalados no sistema. Essa checagem se baseia no nome do pacote (*package name*) de ferramentas populares como Magisk (`com.topjohnwu.magisk`), SuperSU (`eu.chainfire.supersu`), KernelSU (`me.weishu.kernelsu`) e frameworks de modificação como Xposed (`de.robv.android.xposed.installer`) e EdXposed (`org.meowcat.edxposed.manager`).

A detecção é feita sem a necessidade de privilégios de superusuário, utilizando chamadas comuns do sistema Android disponíveis para qualquer aplicativo. Entre os métodos mais usados estão as APIs do *Package Manager* (PM), como getInstalledPackages(), getInstalledApplications() e getPackageInfo(), além de consultas via *intents* e comandos de terminal. Essas abordagens permitem ao aplicativo acessar a lista de pacotes instalados e identificar, com base em nomes conhecidos, ferramentas associadas ao *Root*.

3.4.2. Checagem de Arquivos

Outra técnica comum de detecção de *Root* consiste na verificação da presença de arquivos e binários associados ao superusuário. O binário `su`, por exemplo, costuma ser instalado em locais específicos do sistema, como `/system/bin/su`, `/system/xbin/su` ou `/sbin/su`. A simples existência desse arquivo, mesmo que inativo, pode indicar que o dispositivo possui *Root*. Ferramentas auxiliares como `BusyBox`, frequentemente incluídas em ambientes com *Root*, também são alvos dessa verificação.

A checagem é realizada por meio de tentativas de leitura desses caminhos, utilizando comandos acessíveis a qualquer aplicativo, como `File.exists()`, `canExecute()` ou buscas via `Runtime.exec()` com comandos como `which` e `ls`. Essas abordagens não requerem privilégios de superusuário e permitem identificar modificações no sistema com base na presença de arquivos típicos de ambientes modificados.

3.4.3. Checagem de Processos, Serviços e Tarefas

A inspeção de processos e serviços em execução é uma estratégia útil para identificar a presença de ferramentas de *Root* ativas no sistema. Aplicativos como o Magisk frequentemente executam serviços em segundo plano que deixam rastros no ambiente de execução. A detecção pode ser feita por meio de comandos como `ps`, `top` ou pela leitura de diretórios no sistema de arquivos `/proc`, que fornece informações detalhadas sobre os processos ativos.

Essa técnica permite identificar nomes de processos, argumentos de execução e até caminhos de execução relacionados a ferramentas de *Root*, mesmo que os arquivos

correspondentes estejam ocultos.

3.4.4. Checagem de Execução de Comandos

Uma abordagem comum na detecção de *Root* é a tentativa de executar comandos que normalmente exigem privilégios de superusuário. Exemplos incluem `su`, `id`, `which su` ou a execução de comandos em diretórios protegidos do sistema. A simples invocação do comando `su`, por exemplo, pode indicar se o binário está presente e acessível.

Caso esses comandos sejam executados com sucesso ou retornem informações atípicas, é um forte indicativo de que o dispositivo possui *Root* habilitado. Essa verificação é particularmente eficaz quando combinada com outras técnicas, pois fornece evidências em tempo de execução do comprometimento do sistema.

3.4.5. Checagem de Propriedades do Sistema

O Android mantém um conjunto de propriedades do sistema acessíveis via o comando `getprop`, que podem revelar modificações no sistema. Valores como `ro.debuggable=1` e `ro.secure=0` sugerem que o sistema está em modo de desenvolvimento, condição frequentemente associada à presença de *Root*. Propriedades específicas de ROMs personalizadas, como `ro.lineage.device`, também podem ser utilizadas na detecção, apontando que o dispositivo não executa uma versão oficial do sistema.

3.4.6. Checagem de Permissões dos Diretórios

Diretórios protegidos do sistema, como `/system`, `/vendor`, `/sbin` e `/data`, normalmente são montados como somente leitura em dispositivos sem *Root*. Técnicas de detecção podem tentar criar ou modificar arquivos de teste nesses locais para verificar se permissões elevadas estão ativas. A capacidade de escrita nesses diretórios indica, com alta probabilidade, que o dispositivo possui acesso *Root* habilitado.

3.4.7. Métodos Avançados de Detecção

Cada uma dessas técnicas pode ser usada isoladamente ou em conjunto para aumentar a confiabilidade da detecção. No entanto, com o avanço de ferramentas e técnicas mais avançadas de ocultação de *Root*, muitas dessas verificações podem ser contornadas ou ocultadas, exigindo estratégias de detecção mais sofisticadas, como o uso de RASP (*Runtime Application Self-Protection*) e análise de integridade estática e em tempo real.

O RASP é uma tecnologia de proteção que atua em tempo de execução, embutida diretamente no código do aplicativo. Ao contrário das abordagens tradicionais baseadas em verificações externas ao aplicativo, o RASP atua de forma integrada ao código, monitorando em tempo real o ambiente de execução e reagindo automaticamente a comportamentos suspeitos. Ele é capaz de reagir automaticamente a atividades suspeitas,

podendo, por exemplo, bloquear a execução ao identificar tentativas de depuração (*debugging*), injeção de código, manipulação de memória ou a presença de ambientes com acesso a *Root*.

Acerca da análise de integridade estática, trata-se de checar se o arquivo *APK (Android Application Pack)* original do aplicativo foi modificado ou adulterado antes mesmo de sua execução. Essa verificação geralmente envolve a validação das assinaturas digitais, soma de verificação (hash) dos arquivos e a comparação com valores esperados, de modo a garantir que o aplicativo não sofreu alterações maliciosas, como a inserção de código malicioso ou remoção de proteções.

Análise de integridade em tempo real consiste na verificação contínua de arquivos, permissões, processos e configurações do sistema durante a execução do aplicativo, com o objetivo de detectar alterações não autorizadas. Essa abordagem vai além da checagem estática inicial, pois considera que o estado do dispositivo pode mudar dinamicamente, especialmente em dispositivos comprometidos. Ao identificar inconsistências, o aplicativo pode tomar medidas como negar acesso a determinadas funcionalidades, exibir alertas ao usuário ou encerrar sua execução de forma segura.

Essas estratégias visam elevar o nível de resiliência das aplicações frente a ataques, sendo amplamente adotadas em aplicativos financeiros, governamentais ou que lidam com informações sensíveis.

3.5. Métodos de Ocultação

A subversão de mecanismos de detecção de *Root* é, via de regra, um processo manual e trabalhoso. Envolve a compreensão do comportamento interno do aplicativo, geralmente por meio de análise estática e dinâmica, isto é, engenharia reversa e instrumentação em tempo de execução, respectivamente. Nesta seção, são apresentadas as principais abordagens empregadas para ocultar o acesso *Root* aos mecanismos de detecção implementados por aplicativos Android.

A subversão de mecanismos de detecção de *Root* é, via de regra, um processo manual e trabalhoso. Envolve a compreensão do comportamento interno do aplicativo, geralmente por meio de engenharia reversa — termo que engloba tanto a análise estática quanto a análise dinâmica.

3.5.1. Análise Estática

A análise estática permite investigar a estrutura interna de um aplicativo Android a partir do seu arquivo de instalação no formato *APK*, possui como objetivo compreender onde e como a detecção de *Root* é realizada. Essa análise é feita sem executar o aplicativo, focando na extração e interpretação de seu código e recursos. As principais ferramentas utilizadas são:

- JADX: ferramenta de decompilação que converte bytecode Dalvik em código Java legível, facilitando a compreensão da lógica da aplicação. No entanto, sua capacidade de reconstrução do código-fonte pode ser prejudicada em casos de ofuscação pesada.

- Apktool: ferramenta amplamente utilizada para desmontar e remontar arquivos *APK*, fornecendo acesso aos recursos XML e ao código Smali — a forma textual do bytecode Dalvik. É útil para localizar e alterar lógicas de verificação, classes relevantes e arquivos de configuração.
- Ghidra: ferramenta de engenharia reversa desenvolvida pela NSA, capaz de analisar e modificar binários compilados, como bibliotecas nativas presentes nos *APKs*. É especialmente eficaz na análise de código C/C++ compilado para arquitetura ARM, comumente empregado em verificações de *Root* de baixo nível.

3.5.2. Análise Dinâmica

Na análise dinâmica, o comportamento de um aplicativo é modificado durante sua execução, sem a necessidade de alterar seus arquivos estaticamente. Essa técnica é poderosa para contornar verificações de *Root* diretamente na memória, permitindo interceptações, modificações de retorno de funções e injeção de códigos personalizados. As principais ferramentas utilizadas são:

- Frida: framework de instrumentação dinâmica amplamente adotado na análise de segurança de aplicativos móveis. Permite a interceptação de chamadas de métodos Java e nativos, leitura e modificação de variáveis em tempo real, além da injeção de código personalizado. Pode ser executado em dispositivos com *Root* ou integrado ao *APK* via *Frida Gadget*.
- Objection: ferramenta baseada em Frida com interface simplificada, voltada para análises rápidas de aplicativos Android e iOS. Automatiza tarefas como enumeração de classes, análise de permissões, subversão de detecção de *Root* e extração de dados sensíveis.

3.5.3. Outras Abordagens

Apesar da viabilidade dessas abordagens, muitos aplicativos implementam contramedidas para dificultar ou impedir sua execução. No caso da análise estática, é comum o uso de técnicas de ofuscação — como a renomeação de classes, métodos e variáveis — que comprometem significativamente a legibilidade do código, enquanto trechos críticos são frequentemente migrados para bibliotecas nativas escritas em C ou assembly. Esse tipo de código, além de mais complexo, costuma ser proprietário e exige tempo de pesquisa e ferramentas específicas para análise em baixo nível.

Na instrumentação dinâmica, mecanismos como *anti-Frida* (verificação de artefatos ou classes conhecidas), *anti-debugging*, validação de integridade da memória e estratégias de RASP são amplamente utilizados. Esses mecanismos buscam identificar alterações no ambiente de execução ou a presença de ferramentas de análise, podendo forçar o encerramento do aplicativo ou provocar comportamentos inesperados. Além disso, muitos aplicativos adotam bibliotecas prontas para detecção de *Root*, como a *RootBeer* [Alexander-Bown 2017], que implementa um conjunto de heurísticas conhecidas para detectar dispositivos comprometidos de forma rápida e integrada ao código da aplicação.

Embora essas proteções sejam, em tese, sempre passíveis de subversão, o sucesso nessas situações geralmente costuma demandar tempo e esforço, especialmente diante de implementações de código nativo proprietário desenvolvido especificamente para dificultar a análise. Assim, a aplicação de técnicas de ocultação de *Root* permanece possível, mas enfrenta desafios cada vez maiores frente à evolução das estratégias defensivas empregadas por aplicativos sensíveis.

Diante da dificuldade exigida pelas abordagens baseadas em engenharia reversa e instrumentação dinâmica, surgem estratégias voltadas à ocultação do *Root* diretamente a partir do sistema operacional. Diferentemente das técnicas anteriores, que atuam sobre o aplicativo, essas soluções operam em camadas mais baixas do Android, com o objetivo de mascarar traços do acesso *Root* antes mesmo que a aplicação entre em execução.

Embora não sejam necessariamente novas, essas ferramentas vêm se tornando cada vez mais sofisticadas e populares, principalmente por reduzirem a necessidade de conhecimentos avançados em engenharia reversa e por oferecerem maior usabilidade. Seu funcionamento baseia-se na interceptação de acessos a arquivos, comandos e propriedades do sistema que possam denunciar a presença de *Root*, assim como na manipulação do ambiente de execução. Dessa forma, a partir de um dispositivo com *Root*, alterações manuais, complexas e direcionadas para cada aplicativo deixam de ser requeridas ao executar aplicativos que implementem mecanismos de detecção de *Root*.

3.6. Ferramentas de Root

Nesta seção, serão apresentadas as principais ferramentas utilizadas para obtenção e gerenciamento do acesso *Root* em dispositivos Android. Embora todas essas soluções tenham como base a modificação do sistema operacional, cada uma implementa abordagens únicas para garantir permissões elevadas, ao mesmo tempo em que busca minimizar a exposição e a detecção pelos mecanismos de segurança dos aplicativos.

3.6.1. Magisk

O Magisk é uma das ferramentas mais populares para obtenção e gerenciamento de acesso *Root* no Android. Sua principal característica é a adoção da abordagem *systemless*, que evita modificações diretas na partição */system*. Por meio de técnicas como *bind mounts*, o Magisk permite sobrescrever arquivos do sistema em tempo de execução, sem alterar os arquivos fisicamente gravados no disco. Essa estratégia reduz o risco de detecção por verificações baseadas em integridade de sistema e facilita a instalação e reversão de atualizações.

O Magisk oferece suporte à execução de módulos personalizados que viabilizam a criação de várias funcionalidades. Ele também inclui o *MagiskHide*, um recurso nativo que altera o nome do pacote do aplicativo Magisk para um identificador aleatório e oculta processos e arquivos associados ao *Root*, a fim de contornar verificações simples de presença de aplicativos suspeitos.

Uma evolução significativa foi a introdução do *Zygisk*, um recurso que permite injetar código diretamente no processo *Zygote* — responsável pela criação de todos os processos de aplicativos e componentes do sistema Android. Com o Zygisk, módulos do

Magisk podem atuar desde o início da execução de cada processo, possibilitando o redirecionamento de chamadas de sistema, a manipulação de variáveis sensíveis e a ocultação de sinais de *Root*. Como essas intervenções ocorrem em tempo de execução e não modificam arquivos no armazenamento persistente, tornam-se mais difíceis de detectar por verificações estáticas.

Por ser um projeto de código aberto, o Magisk — especialmente seus recursos como o Zygisk — passou a ser alvo de soluções de segurança especializadas, incluindo mecanismos RASP. Nesse contexto, a comunidade desenvolveu diversos módulos com o objetivo de estender as capacidades de ocultação do Magisk, buscando superar as limitações impostas por novas estratégias de detecção. Entre os diversos módulos existentes com essa finalidade, optou-se por destacar os seguintes, devido à sua ampla adoção e eficácia nos testes realizados:

- Zygisk Next: É uma implementação independente do Zygisk. Originalmente de código aberto, seu desenvolvimento tornou-se fechado para dificultar a análise por parte de soluções *anti-Root*, o que dificulta a criação de contra-medidas baseadas em assinatura ou comportamento. Permite injetar código nos processos derivados do Zygote de forma personalizada. Por atuar diretamente no ciclo de vida de inicialização dos aplicativos, é eficaz para mascarar indicadores de *Root* em fases iniciais da execução.
- Zygisk Assistant: módulo desenvolvido para o framework Zygisk, com foco em ocultar a presença do *Root* e do próprio Zygisk, dificultando sua detecção por aplicativos de segurança. Atua por meio da manipulação de propriedades do sistema e injeção de código em processos críticos.
- Reset LOS Props: módulo simples que modifica propriedades do sistema Android para remover referências à ROM personalizada LineageOS. Especificamente, ele altera todas as propriedades que contenham o nome ”lineage”, substituindo-as por valores padrão, com o objetivo de evitar que aplicativos detectem a utilização da ROM personalizada e, assim, reduzir a exposição a mecanismos de bloqueio ou restrição.

Apesar desses módulos adicionais, ferramentas de análise e proteção em tempo real começaram a rastrear artefatos e comportamentos característicos do Magisk, o que resultou no desenvolvimento de abordagens mais sofisticadas de detecção, diminuindo a efetividade do Magisk em cenários de segurança mais restritos.

3.6.2. KernelSU

O KernelSU é uma solução de *Root* que, diferentemente do Magisk, opera diretamente no *kernel* do sistema Android. Essa abordagem oferece maior profundidade e controle sobre o ambiente de execução, dificultando a detecção por parte de mecanismos convencionais baseados em verificações de arquivos ou processos em espaço de usuário. Por atuar no espaço de *kernel*, o KernelSU consegue fornecer permissões de superusuário com mínima exposição de artefatos detectáveis por aplicações.

Assim como o Magisk, o KernelSU também suporta a execução de módulos personalizados, embora com diferenças de implementação — como a localização de arquivos

e uso de variáveis de ambiente distintas. Muitos módulos originalmente desenvolvidos para o Magisk podem ser adaptados para funcionar com o KernelSU, desde que respeitadas essas particularidades. No entanto, o KernelSU não possui suporte nativo ao Zygisk; para isso, é necessário o uso de módulos auxiliares, como o ZygiskNext, que integram capacidades de injeção em tempo de execução semelhantes às do Magisk.

A instalação do KernelSU varia conforme o tipo de *kernel* utilizado no dispositivo. Em dispositivos com suporte ao GKI (Generic *kernel* Image), o processo de instalação pode ser feito com o download de uma imagem de boot já modificada contendo o KernelSU embutido — disponibilizada pelo próprio projeto no GitHub. Em dispositivos com kernels legados (não-GKI), é necessário compilar o *kernel* manualmente com o código do KernelSU integrado, exigindo familiaridade com o processo de compilação do *kernel* Android. Alternativamente, também é possível encontrar ROMs personalizadas que já incluem o KernelSU integrado ao sistema.

O suporte oficial abrange as arquiteturas `arm64-v8a` e `x86_64`, o que garante compatibilidade com a maioria dos dispositivos modernos. Sua atuação em camadas mais profundas do sistema e a ausência de modificações no espaço de usuário o tornam especialmente eficaz para contornar técnicas de detecção baseadas em *user space*, embora também estejam surgindo soluções específicas de segurança voltadas à detecção dessa abordagem.

3.6.3. APatch

O APatch é uma solução de *Root* que modifica diretamente a imagem de boot do sistema, aplicando alterações tanto no *kernel* quanto no sistema Android durante o processo de inicialização. Diferentemente do KernelSU, que exige a compilação manual do *kernel*, o APatch atua sobre a imagem de boot já compilada, tornando o processo mais simples e genérico.

Para utilizá-lo, é necessário extrair a imagem de boot do dispositivo e aplicar as modificações por meio de seu próprio aplicativo *APK*. Caso o *kernel* seja compatível, o processo resulta na geração de uma nova imagem de boot com o APatch integrado. A imagem modificada pode então ser instalada via *fastboot*, de forma similar à instalação do Magisk.

Assim como outras soluções modernas, o APatch não oferece por si só mecanismos avançados de ocultação de *Root*. Por isso, seu uso é frequentemente combinado com módulos auxiliares como o *Zygisk Next* e o *Zygisk Assistant*, que atuam em tempo de execução para esconder a presença do *Root* e impedir a detecção por parte de aplicativos sensíveis. Essa integração é essencial para evitar a exposição de artefatos durante a inicialização ou durante a execução de chamadas sensíveis.

Embora seja uma abordagem mais acessível e que dispense recompilação do *kernel*, sua compatibilidade é limitada a determinados dispositivos, devido a incompatibilidades de versões do *kernel* e do Android.

Por atuar em camadas profundas do sistema e evitar alterações no espaço de usuário, o APatch apresenta um bom potencial de subversão frente a mecanismos convencionais de detecção de *Root*, sendo uma alternativa promissora para dispositivos com-

patíveis.

4. Metodologia

Esta seção descreve os procedimentos, ferramentas e dispositivos utilizados para a avaliação das técnicas de ocultação de *Root* em aplicativos Android.

Os testes foram realizados em dois dispositivos distintos, um Motorola Moto G10 e um Xiaomi Redmi Note 6 Pro. Na Figura 2 estão dispostas duas capturas de tela contendo informações como modelo e versão do Android sobre cada um dos respectivos aparelhos.

Para as técnicas usando o Magisk ou APatch, utilizou-se um Moto G10 com sistema original da fabricante, com o sistema operacional Android na versão 11. Os testes com o KernelSU foram conduzidos em um Redmi Note 6 Pro com a ROM LineageOS baseada no Android 15. Para retratar um cenário mais amplo e realista, optou-se por utilizar uma variedade de dispositivos e sistemas, contemplando tanto ambientes com o sistema original quanto com ROMs personalizadas.

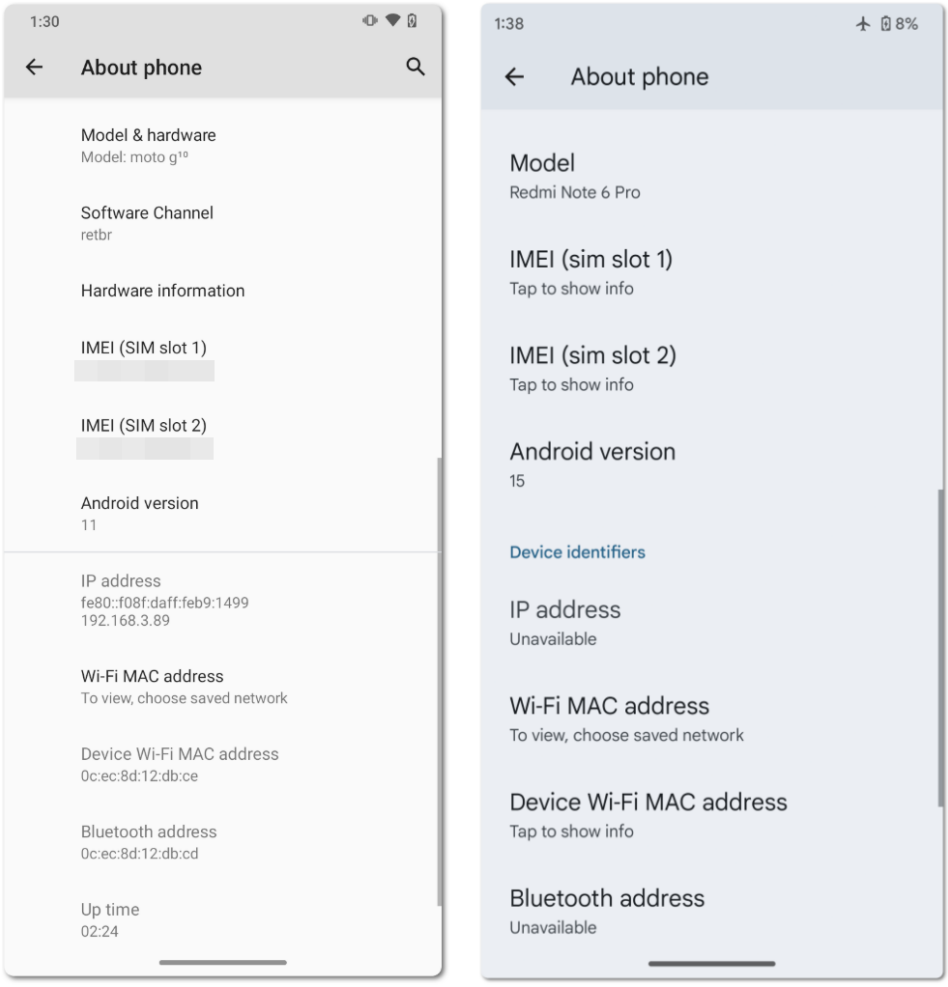

Figura 2. Especificações dos dispositivos utilizados (Moto G10 e Redmi Note 6 Pro).

Foram definidos quatro ambientes de teste para a avaliação da eficácia das técnicas de ocultação de *Root*, que serão apresentadas a seguir.

4.1. Cenário 1: Magisk (configuração padrão)

O acesso *Root* foi obtido utilizando o Magisk, com as técnicas nativas de ocultação habilitadas: alteração do nome do pacote para um identificador aleatório, ativação do Zygisk e inclusão do aplicativo sob teste na lista de negação do Magisk, responsável por definir em quais aplicativos a ocultação de *Root* será realizada.

A Figura 3 apresenta a aplicação das configurações descritas para o cenário 1.

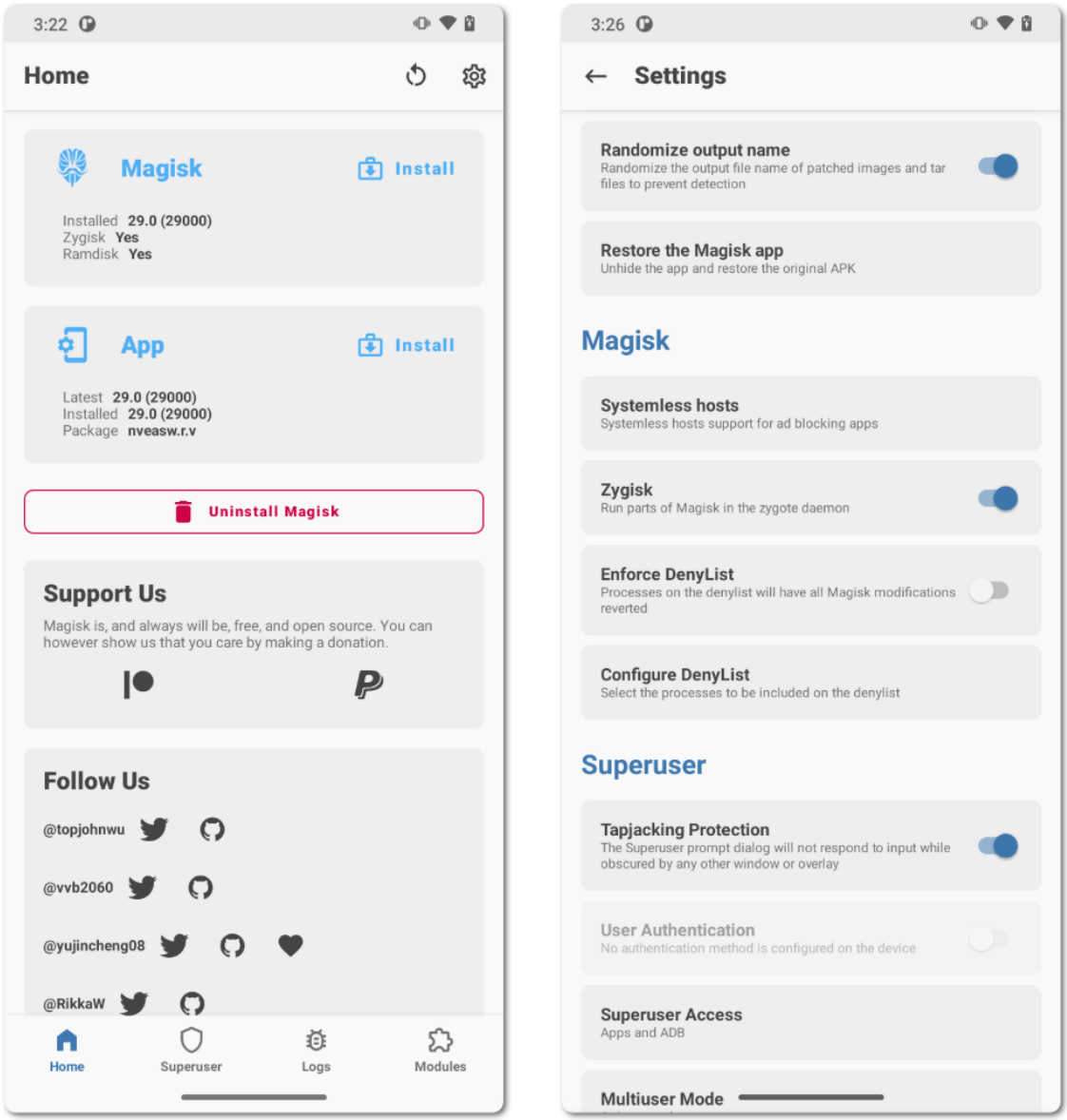

Figura 3. Configuração padrão do Magisk, em execução no dispositivo Moto G10.

4.2. Cenário 2: Magisk + Módulos

Utilizou-se novamente o Magisk, porém com os módulos adicionais *Zygisk Next* e *Zygisk Assistant* instalados. O Zygisk nativo do Magisk foi desativado, e as funções de ocultação passaram a ser responsabilidade dos módulos externos.

A Figura 4 ilustra a aplicação das configurações descritas para esse cenário.

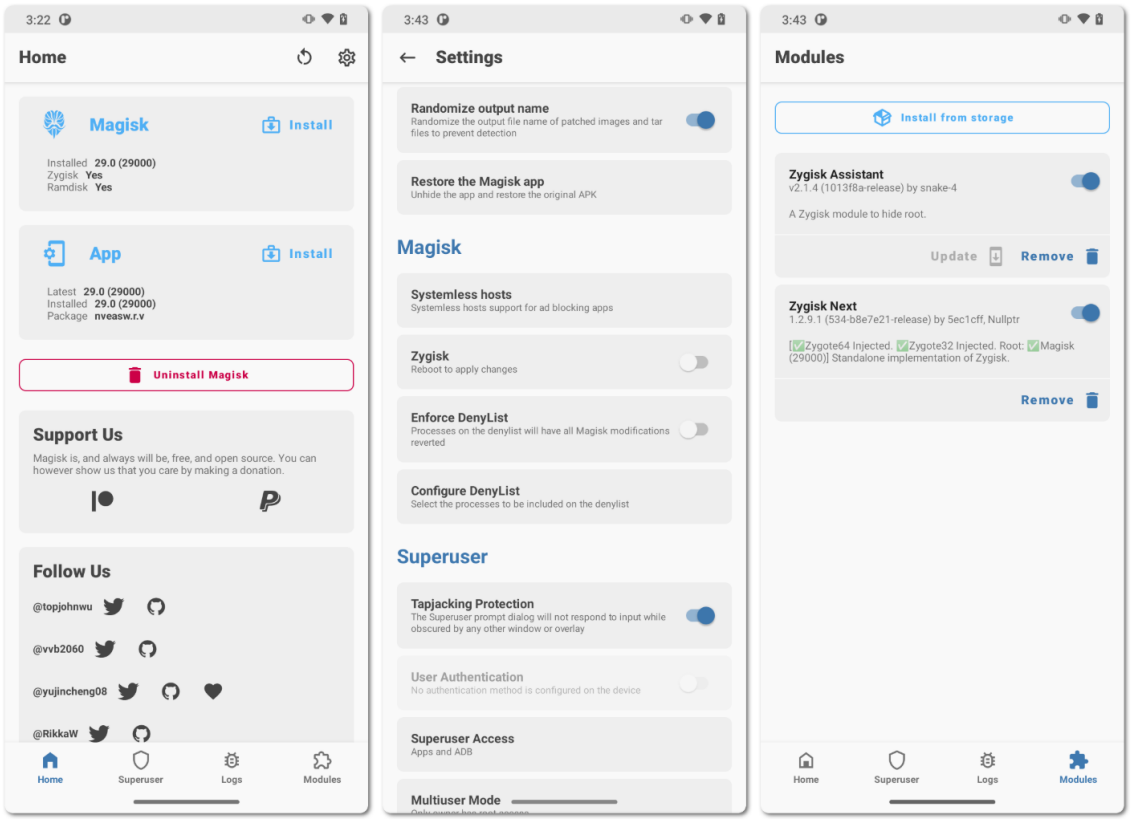

Figura 4. Configuração do Magisk acrescida dos módulos *Zygisk Next* e *Zygisk Assistant*, em execução no dispositivo Moto G10.

4.3. Cenário 3: KernelSU + Módulos

Neste cenário, o acesso *Root* foi fornecido pelo KernelSU, em conjunto com os módulos *Zygisk Next*, *Zygisk Assistant* e *Reset LOS Props*. Adicionalmente, o pacote do aplicativo KernelSU foi removido para reduzir a superfície de detecção.

As configurações adotadas no cenário 3 estão ilustradas na Figura 5.

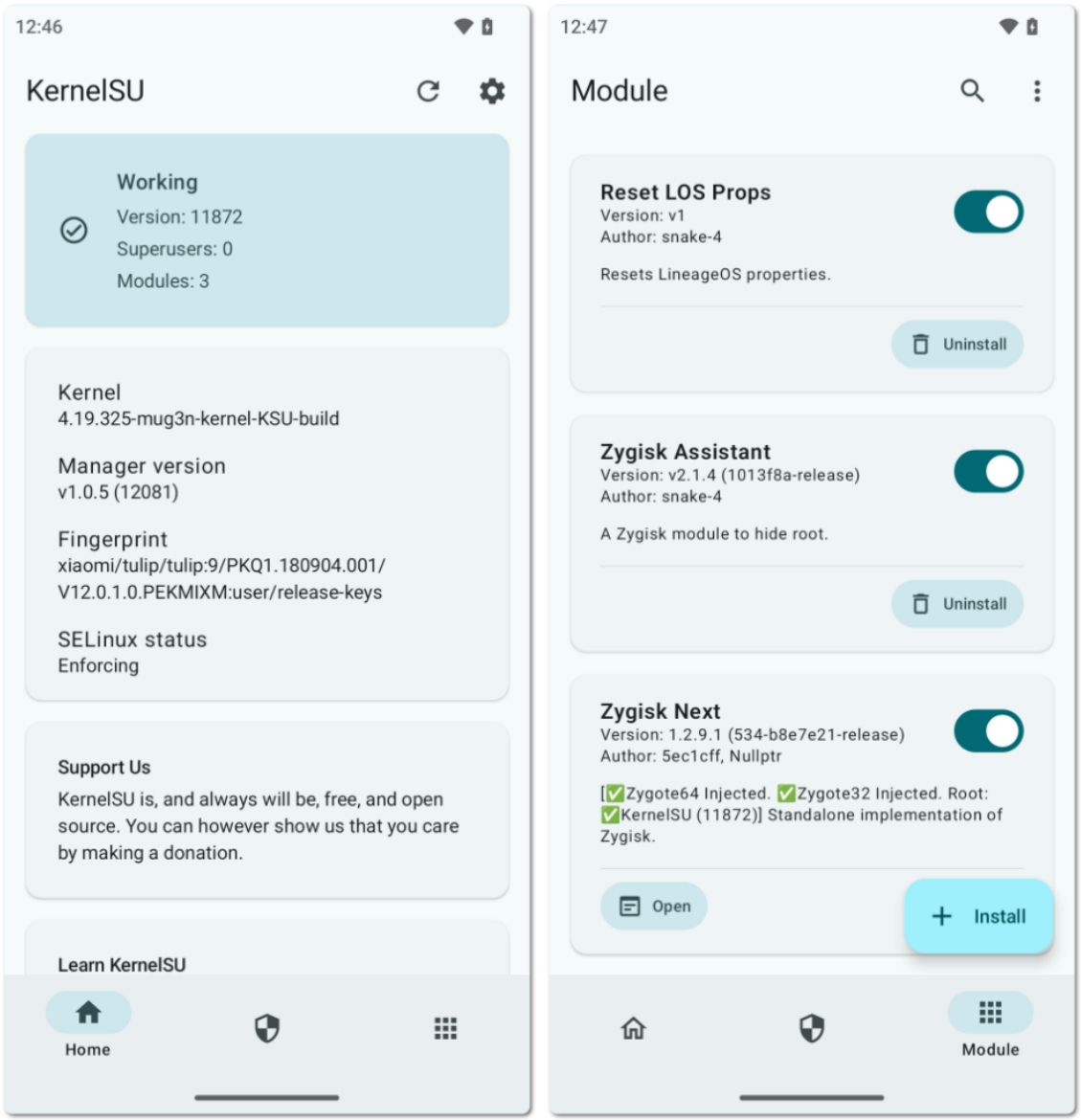

Figura 5. Configuração do KernelSU acrescida dos módulos *Zygisk Next*, *Zygisk Assistant* e *Reset LOS Props* em execução no dispositivo Redmi Note 6 Pro.

4.4. Cenário 4: APatch + Módulos

A quarta abordagem empregou o APatch como técnica de *Root*. Foram utilizados os módulos *Zygisk Next* e *Zygisk Assistant*, e, assim como no cenário anterior, o pacote do aplicativo foi excluído do sistema após a instalação.

A Figura 6 evidencia a aplicação das configurações descritas para o cenário 4.

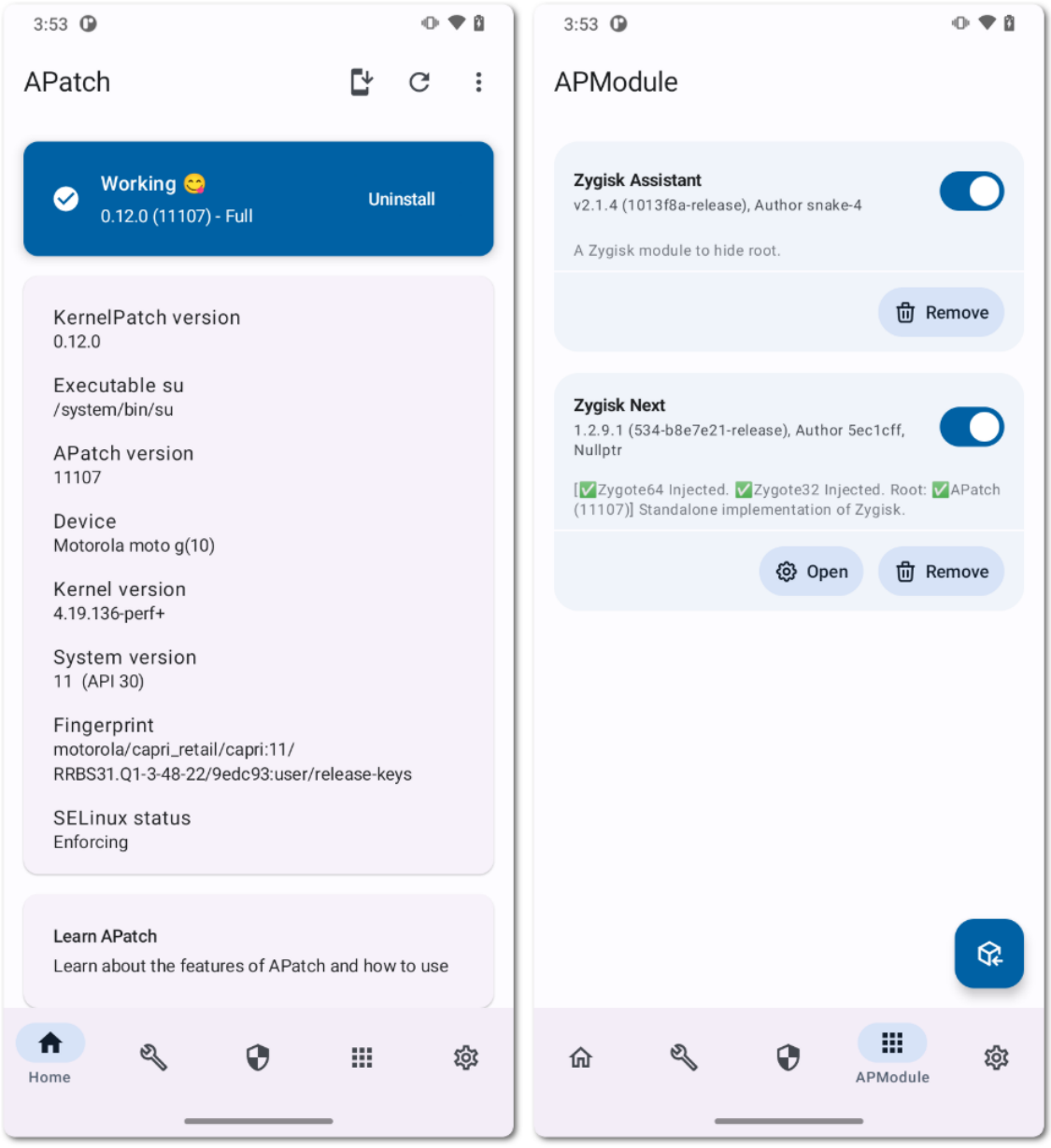

Figura 6. Configuração do APatch acrescida dos módulos *Zygisk Next* e *Zygisk Assistant*, em execução no dispositivo Moto G10.

Cada configuração foi testada frente a um conjunto de aplicativos com mecanismos próprios de detecção de *Root*, com o objetivo de verificar se a técnica de ocultação empregada foi capaz de subverter com sucesso a verificação.

5. Resultados

Esta seção apresenta os resultados obtidos a partir da metodologia descrita anteriormente.

Com o objetivo de avaliar a eficácia das diferentes técnicas de ocultação de *Root*, foram selecionados 15 aplicativos. A escolha destes aplicativos foi definida utilizando os seguintes critérios:

- Aplicativos financeiros ou governamentais.
- Aplicativos populares, foi selecionado os 6 aplicativos mais baixados na categoria Finanças da Google Play Store no Brasil.
- Aplicativos que implementem proteções reconhecidamente robustas, optou-se por escolher aplicativos que utilizem mecanismos de proteção variados e preferencialmente com RASP.

Dentre eles, 12 pertencem ao setor financeiro e 3 são de natureza governamental. Na Tabela 1 é apresentado o nome, pacote, versão e proteção utilizada para cada um dos aplicativos utilizados nos testes.

Tabela 1. Informações técnicas sobre os aplicativos utilizados.

Nome	Pacote	Versão	Proteção Utilizada
Axis Mobile	com.axis.mobile	10.82	Protectt
Banco do Brasil	br.com.bb.android	9.72.2.1	DexGuard 9.x + Arxan
Bradesco	com.bradesco	4.68.0	DexProtector + Arxan
BTG Pactual	com.btg.pactual.banking	2.32.2	DexGuard 9.x
CAIXA	br.com.gabba.Caixa	5.17.0	Appdome
CAIXA Tem	br.gov.caixa.tem	1.91.16	Appdome
e-Titulo	br.jus.tse.eleitoral.etitulo	2025.07.11	DexProtector
Gov.br	br.gov.meugovbr	3.7.40	RootBeer
Inter	br.com.intermedium	14.0.2	DexGuard 9.x
Mercado Pago	com.mercadopago.wallet	2.392.1	DexGuard 9.x
Nubank	com.nu.production	9.31.87-minApi28	DexGuard 9.x + Arxan + RootBeer
Picpay	com.picpay	11.85.4	DexGuard 9.x + Arxan
Santander	com.santander.app	25.5.5.0	DexGuard 9.x
Santander UK	uk.co.santander.santanderUK	5.22.0	DexGuard + RootBeer
Singpass	sg.ndi.sp	25.2.1	DexGuard 9.x

Dos aplicativos testados, 14 utilizavam RASP e a única exceção foi o aplicativo Gov.br. Devido a essas proteções serem produtos de *software* proprietários e de código fechado, é impraticável analisar a implementação dos mecanismos de detecção empregados em cada uma dessas proteções. Entretanto, considerando a finalidade do mecanismo de detecção de *Root*, é razoável inferir que, uma vez implementado e a presença de *Root* detectada, o funcionamento do aplicativo será comprometido e resultará normalmente em sua interrupção. Dessa forma, utilizou-se como métrica de efetividade da ocultação a verificação de eventuais mudanças no comportamento do aplicativo ou interrupções em suas funcionalidades.

A Tabela 2 apresenta um panorama consolidado dos resultados obtidos, listando o nome do aplicativo e o resultado da detecção de *Root* obtido para cada cenário previamente definido. O cenário 1 corresponde ao Magisk com suas configurações padrão, enquanto o cenário 2 utiliza o Magisk com os módulos Zygisk Next e Zygisk Assistant. O

cenário 3 representa o uso do KernelSU, com os mesmos módulos anteriores acrescidos do módulo Reset LOS Props. Por fim, o cenário 4 refere-se ao APatch, combinado com os módulos Zygisk Next e Zygisk Assistant.

Tabela 2. Efetividade da detecção de *Root* para cada técnica utilizada.

Aplicativo	Cenário 1	Cenário 2	Cenário 3	Cenário 4
Axis Mobile	Detectado	Detectado	Não Detectado	Detectado
Banco do Brasil	Não Detectado	Não Detectado	Não Detectado	Não Detectado
Bradesco	Detectado	Detectado	Não Detectado	Não Detectado
BTG Pactual	Não Detectado	Não Detectado	Não Detectado	Não Detectado
CAIXA	Detectado	Detectado	Detectado	Não Detectado
Caixa Tem	Detectado	Detectado	Detectado	Não Detectado
e-Titulo	Detectado	Detectado	Não Detectado	Não Detectado
Gov.br	Não Detectado	Não Detectado	Não Detectado	Não Detectado
Inter	Não Detectado	Não Detectado	Não Detectado	Não Detectado
Mercado Pago	Não Detectado	Não Detectado	Não Detectado	Não Detectado
Nubank	Não Detectado	Não Detectado	Não Detectado	Não Detectado
Picpay	Detectado	Detectado	Não Detectado	Não Detectado
Santander	Detectado	Detectado	Não Detectado	Não Detectado
Santander UK	Detectado	Detectado	Não Detectado	Não Detectado
Singpass	Detectado	Detectado	Não Detectado	Não Detectado

Dos 15 aplicativos testados, verificou-se que 10 adotaram algum tipo de verificação para restringir o uso em dispositivos modificados. Entre eles, destaca-se o aplicativo Gov.br, que possui integrada em seu código-fonte a biblioteca RootBeer — desenvolvida especificamente para detecção de *Root* —, mas que não identificou a presença de acesso privilegiado em nenhum dos cenários propostos, o que sugere uma implementação inadequada ou desativada.

Em relação aos outros 5 aplicativos, constatou-se que não apresentam restrições com base na detecção de *Root*, uma vez que funcionaram normalmente mesmo sem o uso de mecanismos avançados de ocultação. Vale ressaltar que a ausência dessa detecção não implica necessariamente em uma vulnerabilidade de segurança. Essa escolha pode refletir diferentes estratégias de proteção, como o foco no endurecimento da segurança no *back-end* da aplicação — isto é, a camada responsável pelo processamento, lógica de negócio e gerenciamento de dados no servidor. Além disso, a decisão de restringir o funcionamento do aplicativo em dispositivos com acesso privilegiado apresenta vantagens e desvantagens: embora possa aumentar a segurança ao reduzir a superfície de ataque no cliente, também pode comprometer a usabilidade ao limitar o número de usuários aptos a utilizar o serviço.

Entre os aplicativos que implementaram algum tipo de detecção, constatou-se que nenhum foi capaz de detectar o acesso *Root* em todos os cenários testados. Os resultados indicam que a eficácia das técnicas de ocultação depende não apenas da ferramenta empregada, mas também do ambiente em que ela é aplicada — sistemas com ROMs personalizadas tendem a deixar rastros que podem ser identificados por verificações mais robustas. Como exemplo, destaca-se o comportamento do aplicativo e-Título, que, além

da detecção de *Root*, também identifica a presença da ROM personalizada LineageOS, utilizada no dispositivo do cenário 4. No entanto, para esse caso, o uso do módulo *Reset LOS Props* foi suficiente para mascarar as propriedades do sistema associadas à ROM personalizada, viabilizando a execução normal do aplicativo mesmo com acesso *Root* habilitado.

Também foi identificado que os aplicativos Caixa, Caixa Tem e Gov.br implementam verificações adicionais, como a checagem do valor da propriedade do sistema `ro.debuggable`, que indica se o modo desenvolvedor está habilitado, restringindo seu funcionamento enquanto esse recurso estiver ativo.

É possível concluir que mesmo diante de mecanismos avançados de verificação — como os empregados pelas soluções RASP —, as técnicas de ocultação mais recentes (KernelSU ou APatch, com módulos auxiliares) demonstraram-se competentes em subverter todas as detecções de *Root*. No entanto, salienta-se que, individualmente, nenhum cenário proposto obteve 100% de eficácia na ocultação. Já no caso do Magisk, mesmo com o uso de módulos adicionais, a ferramenta mostrou-se majoritariamente ineficaz frente aos aplicativos que implementaram mecanismos de detecção.

Durante o período de desenvolvimento deste trabalho, alguns aplicativos receberam atualizações que passaram a detectar a presença do pacote do aplicativo APatch. Embora isso não altere os resultados apresentados, evidencia a constante evolução dos mecanismos de proteção contra *Root*.

6. Discussões e Trabalhos Futuros

Os resultados obtidos demonstram que as técnicas de ocultação de *Root* baseadas em modificações de baixo nível do sistema, como KernelSU e APatch, associadas a módulos auxiliares, apresentaram alta efetividade na subversão dos mecanismos de detecção implementados por aplicativos. Esse cenário indica que abordagens que atuam diretamente no *kernel* ou no fluxo de execução em tempo real tendem a ser mais difíceis de serem detectadas pelas soluções tradicionais baseadas em verificações de espaço de usuário.

Por outro lado, a persistente detecção de *Root* por meio do pacote do aplicativo e da ROM personalizada evidencia a constante evolução dos mecanismos de proteção contra *Root*. Isso indica que o embate entre técnicas de ocultação e mecanismos de detecção permanece dinâmico e em constante evolução, reforçando a importância de mais pesquisas para acompanhar as inovações tanto no lado ofensivo quanto no defensivo.

Para trabalhos futuros, destacam-se as seguintes possibilidades:

- Desenvolvimento de *Model Context Protocols* (MCPs) para integrar ferramentas de engenharia reversa, como o Frida, facilitando a análise estática e instrumentação dinâmica, e reduzindo o tempo e esforço necessários para subverter detecções complexas.
- Análise aprofundada dos módulos auxiliares disponíveis, visando identificar possíveis vetores de melhoria ou pontos fracos que possam ser explorados por mecanismos de detecção.
- Estudo dos métodos de ocultação funcionais buscando por vestígios que possibilitem detectá-los.

Com a crescente adoção de dispositivos móveis em ambientes críticos, a segurança dos aplicativos e a proteção contra adulterações do sistema tornam-se essenciais. Assim, a disputa pelo aprimoramento das técnicas de ocultação de *Root* é relevante tanto para usuários avançados quanto para desenvolvedores, que são motivados a aumentar a maturidade de segurança de suas aplicações.

Referências

Alexander-Bown, S. (2017). RootBeer: A Root Detection Library for Android. Disponível em: https://github.com/scottyab/rootbeer. Acesso em: 30 jul. 2025.

bmax121 (2025). APatch - A Secure and Stealthy Root Solution for Android. Disponível em: https://github.com/bmax121/APatch. Acesso em: 27 mai. 2025.

Casati, L. and Visconti, A. (2018). The Dangers of Rooting: Data Leakage Detection in Android Applications. *Mobile Information Systems*, 2018. Disponível em: https://doi.org/10.1155/2018/6020461. Acesso em: 27 mai. 2025.

Dr-TSNG (2025). ZygiskNext - Extended Zygisk Module for Magisk. Disponível em: https://github.com/Dr-TSNG/ZygiskNext. Acesso em: 27 mai. 2025.

Geist, D., Nigmatullin, M., and Bierens, R. (2016). Jailbreak/Root Detection Evasion Study on iOS and Android. Disponível em: https://rp.os3.nl/2015-2016/p51/report.pdf. Acesso em: 27 mai. 2025.

Kamal, B. A., Shahid, A. B., Sajjad, S. M., Kifayat, K., and Ul Hassan, K. M. (2024). A Novel Technique of Android Stealth Rooting. In *2024 17th International Conference on Development in eSystem Engineering (DeSE)*, pages 275–280.

Kellner, A., Horlboge, M., Rieck, K., and Wressnegger, C. (2019). False Sense of Security: A Study on the Effectivity of Jailbreak Detection in Banking Apps. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–14.

Moraes, V. and Vilela, J. (2021). Uma Avaliação do Cenário de Detecção e Evasão do Acesso Root no Android. In *Anais do XXI Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 57–70, Porto Alegre, RS, Brasil. SBC.

Nguyen-Vu, L., Chau, N.-T., Kang, S., Jung, S., and Zhang, Z. (2017). Android rooting: An arms race between evasion and detection. *Sec. and Commun. Netw.*, 2017. Disponível em: https://doi.org/10.1155/2017/4121765. Acesso em: 27 mai. 2025.

NordVPN (2023). Why Root Android Phones? The Pros and Cons of Rooting an Android. Disponível em: https://nordvpn.com/blog/why-you-shouldnt-root-android/. Acesso em: 6 jul. 2025.

Soewito, B. and Suwandaru, A. (2022). Android Sensitive Data Leakage Prevention with Rooting Detection Using Java Function Hooking. *Journal of King Saud University-Computer and Information Sciences*, 34(5):1950–1957.

Sun, S.-T., Cuadros, A., and Beznosov, K. (2015). Android Rooting: Methods, Detection, and Evasion. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, page 3–14, New York, NY, USA. Association for Computing Machinery.

Synopsys (2021). Synopsys Research Reveals Significant Security Concerns in Popular Mobile Apps Amid Pandemic. Disponível em: https://news.synopsys.com/2021-03-25-Synopsys-Research-Reveals-Significant-Security-Concerns-in-Popular-Mobile-Apps-Amid-Pandemic. Acesso em: 9 mai. 2025.

Talsec (2024). Why Root Detection is Critical for Security. Disponível em: https://docs.talsec.app/glossary/root-detection/why-root-detection-is-critical-for-security. Acesso em: 27 mai. 2025.

Transformação Digital (2019). O que é Transformação Digital? Disponível em: https://transformacaodigital.com/o-que-e-transformacao-digital/. Acesso em: 27 mai. 2025.

ValueMentor (2024). My Fav 7 Methods for Bypassing Android Root Detection. Disponível em: https://valuementor.com/blogs/my-fav-7-methods-for-bypassing-android-root-detection. Acesso em: 27 mai. 2025.

Weishu, T. (2025). KernelSU - Root Implemented in Kernel Space. Disponível em: https://github.com/tiann/KernelSU. Acesso em: 27 mai. 2025.

Wu, J. (2025). Magisk - The Magic Mask for Android. Disponível em: https://github.com/topjohnwu/Magisk. Acesso em: 27 mai. 2025.