Geração de Scripts Frida Guiada por Contexto Dinâmico para Análise de Segurança Android

Francisco Sales[1], **Eduardo Souto**[1], **Horácio Fernandes**[1]

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
69080-900 – Manaus – AM – Brasil

{francisco.sales,esouto,horacio}@icomp.ufam.edu.br

Abstract. *Dynamic instrumentation is fundamental for Android application security analysis, but requires specialized knowledge of Frida scripting, Android development techniques, and the internal structure of the target application. This paper investigates whether runtime-collected context improves Frida script generation by language models for security tasks. The investigated approach incorporates into the generation process information extracted from the running application, such as loaded classes, methods, third-party libraries, native modules, and local storage mechanisms, according to the specificity level of the query. This approach was implemented in a reference system used in the experimental evaluation. The evaluation was conducted in two complementary scenarios: i) 10 intentionally vulnerable Android applications, encompassing 83 security tasks, 92 natural language queries, and 460 executions; and ii) 54 applications from the Ghera repository, each targeting the isolation of a specific vulnerability. The results indicate success rates of 95.7% and 93.0%, respectively. Furthermore, in an ablation experiment conducted on a subset of 12 tasks, the configuration without dynamic context achieved 60%, compared to 98.3% with context.*

Resumo. *A instrumentação dinâmica é fundamental para a análise de segurança de aplicações Android, mas exige conhecimento especializado em scripts Frida, em técnicas de desenvolvimento Android e sobre a estrutura interna da aplicação-alvo. Este artigo investiga se o contexto coletado em tempo de execução melhora a geração de scripts Frida por modelos de linguagem para tarefas de segurança. A abordagem investigada incorpora ao processo de geração informações extraídas da aplicação em execução, como classes carregadas, métodos, bibliotecas de terceiros, módulos nativos e mecanismos de armazenamento local, de acordo com o nível de especificidade da consulta. Essa abordagem foi implementada em um sistema de referência utilizado na avaliação experimental. A avaliação foi conduzida em dois cenários complementares: i) 10 aplicações Android intencionalmente vulneráveis, abrangendo 83 tarefas de segurança, 92 consultas em linguagem natural e 460 execuções; e ii) 54 aplicações do repositório Ghera, cada uma voltada ao isolamento de uma vulnerabilidade específica. Os resultados indicam taxas de sucesso de 95,7% e 93,0%, respectivamente. Além disso, no experimento de ablação conduzido em um subconjunto de 12 tarefas, a configuração sem contexto dinâmico alcançou 60%, em comparação com 98,3% na configuração com contexto.*

1. Introdução

A instrumentação dinâmica é uma técnica fundamental na análise de segurança de aplicações Android, pois permite observar e modificar o comportamento de programas em tempo de execução para identificar vulnerabilidades, interceptar dados sensíveis e contornar mecanismos de proteção [Sutter et al. 2024, Enck et al. 2014]. Nesse contexto, o Frida [Frida 2026] consolidou-se como um dos principais frameworks para instrumentação dinâmica, permitindo a injeção de scripts JavaScript em processos em execução sem necessidade de recompilação [D'Elia et al. 2019]. Apesar de seu poder e flexibilidade, o uso efetivo do Frida ainda exige conhecimento especializado sobre sua API, sobre o funcionamento interno da aplicação-alvo e sobre técnicas de instrumentação adaptadas a diferentes cenários, o que cria uma barreira de entrada significativa para analistas menos experientes.

Para reduzir essa dificuldade, a comunidade de segurança desenvolveu ferramentas e repositórios que simplificam tarefas recorrentes. Soluções como Objection [Objection 2026] e MEDUSA [MEDUSA 2026] oferecem comandos e módulos prédefinidos para operações comuns, como *bypass* de SSL pinning e detecção de root. Já o Frida CodeShare [Frida CodeShare 2026] disponibiliza scripts compartilhados pela comunidade para reutilização em diferentes cenários. Embora úteis, essas abordagens apresentam limitações importantes: comandos embutidos restringem o analista a um conjunto fixo de funcionalidades, enquanto scripts comunitários normalmente exigem seleção manual e adaptação ao contexto específico de cada aplicação.

Nesse cenário, modelos de linguagem de grande escala (LLMs) surgem como uma alternativa promissora, pois têm demonstrado forte capacidade de geração de código a partir de descrições em linguagem natural [Chen et al. 2021, Rozière et al. 2023, Fan et al. 2023], inclusive em tarefas relacionadas à segurança [Xu et al. 2025, Pearce et al. 2023]. Entretanto, sua aplicação direta à geração de scripts de instrumentação dinâmica ainda enfrenta uma limitação central: sem informações concretas sobre a aplicação-alvo, o modelo tende a produzir scripts que referenciam classes, métodos ou bibliotecas inexistentes, comprometendo sua executabilidade. Assim, o desafio não consiste apenas em utilizar um LLM para gerar código, mas em estruturar adequadamente a entrada do modelo de modo a representar adequadamente o contexto da aplicação a ser instrumentada.

Este trabalho parte da hipótese de que a incorporação de contexto coletado em tempo de execução pode melhorar a adequação, a executabilidade e a consistência de scripts Frida gerados por modelos de linguagem para análise de segurança Android. Para investigar essa hipótese, propomos uma abordagem de geração de scripts guiada por contexto dinâmico, na qual informações observadas durante a execução da aplicação, como classes carregadas, métodos disponíveis, bibliotecas de terceiros, módulos nativos e mecanismos de armazenamento local, são incorporadas ao processo de geração. Além disso, a quantidade e o tipo de contexto fornecido ao modelo variam conforme o nível de especificidade da consulta formulada pelo analista de segurança, buscando equilibrar relevância semântica sem introduzir detalhes irrelevantes que possam prejudicar a geração do script.

Como prova de conceito dessa abordagem, implementamos o sistema *FridaForge*, que integra coleta de contexto, construção de prompts, geração de scripts e execução da instrumentação. Avaliamos a abordagem proposta em dois cenários complementares. No

primeiro, utilizamos um conjunto de 10 aplicações Android intencionalmente vulneráveis, no qual definimos 83 tarefas de segurança e 92 consultas em linguagem natural, executadas cinco vezes cada, totalizando 460 execuções. No segundo, realizamos uma avaliação complementar com 54 aplicações do repositório Ghera [Mitra and Ranganath 2017], cada uma voltada ao isolamento de uma vulnerabilidade específica, o que nos permite observar o comportamento da abordagem em cenários mais controlados. Os resultados indicam taxas de sucesso de 95,7% nas 10 aplicações do conjunto OWASP e 93,0% nas aplicações do repositório Ghera.

As principais contribuições deste trabalho são: i) uma abordagem para geração de scripts Frida guiada por contexto dinâmico coletado em tempo de execução; ii) uma estratégia progressiva de injeção de contexto condicionada ao nível de especificidade da consulta submetida pelo analista; iii) uma implementação de prova de conceito, denominada *FridaForge*, que operacionaliza essa abordagem; e iv) uma avaliação empírica em aplicações vulneráveis, analisando taxas de sucesso, modos de falha e o papel do contexto dinâmico na adequação dos scripts gerados.

O restante deste artigo está organizado da seguinte forma: a Seção 2 discute os trabalhos relacionados; a Seção 3 apresenta a abordagem proposta e sua implementação de referência; a Seção 4 descreve o desenho experimental, os resultados obtidos e as ameaças à validade; e a Seção 5 conclui o trabalho.

2. Trabalhos Relacionados

Os trabalhos mais próximos deste estudo situam-se na interseção entre instrumentação dinâmica de aplicações Android e geração de código assistida por modelos de linguagem. No ecossistema de instrumentação baseado em Frida, soluções como Objection [Objection 2026] e MEDUSA [MEDUSA 2026] oferecem comandos e módulos prédefinidos para tarefas recorrentes, como *bypass* de SSL pinning e detecção de root. Essas ferramentas reduzem parte do esforço operacional do analista, mas permanecem limitadas a um conjunto fixo de operações,dificultando sua adaptação a aplicações com estruturas, bibliotecas ou pontos de instrumentação específicos. O Frida CodeShare [Frida CodeShare 2026], por sua vez, amplia a disponibilidade de scripts por meio de repositórios comunitários, porém mantém elevada a dependência de busca manual, seleção e adaptação ao contexto da aplicação-alvo.

Além dessas soluções, o Dexcalibur [Michel 2020] combina análise estática parcial com instrumentação dinâmica, gerando interceptações a partir de artefatos obtidos do código da aplicação. Essa abordagem representa um avanço em relação ao uso exclusivo de scripts fixos, mas ainda depende de informações derivadas de código decompilado e não de contexto efetivamente observado durante a execução. De modo geral, essas ferramentas partem de uma visão prévia da aplicação ou de catálogos de scripts reutilizáveis, em vez de tratar a geração da instrumentação como um processo condicionado pela intenção do analista e pelo estado real do aplicativo em tempo de execução.

Em paralelo, modelos de linguagem de grande escala vêm demonstrando forte capacidade de geração de código a partir de descrições em linguagem natural, como evidenciado por trabalhos com Codex [Chen et al. 2021] e Code Llama [Rozière et al. 2023], e sistematizado em levantamentos recentes sobre LLMs aplicados à engenharia de software [Fan et al. 2023]. No domínio de segurança, esses modelos também têm sido em-

pregados em tarefas como detecção de vulnerabilidades, reparo automático de código inseguro [Pearce et al. 2023] e análise de malware [Xu et al. 2025]. Entretanto, a aplicação direta de LLMs à geração de scripts de instrumentação dinâmica ainda enfrenta uma limitação importante: sem informações concretas sobre a aplicação-alvo, o modelo tende a produzir referências inexistentes ou a selecionar pontos de instrumentação inadequados, comprometendo a executabilidade do script gerado.

O trabalho mais próximo da presente proposta é o CovAgent [Minn et al. 2026], que emprega agentes baseados em LLM e análise estática de código Smali para gerar scripts Frida com foco em cobertura de testes GUI em Android. Embora compartilhe a ideia de usar modelos de linguagem na geração de scripts Frida, o CovAgent difere deste trabalho em três aspectos centrais: primeiro, seu objetivo é cobertura de testes, e não análise de segurança; segundo, sua principal fonte de informação é o código decompilado, e não o comportamento observado em tempo de execução; e por último, sua geração não é estruturada em torno da especificidade da consulta formulada pelo analista.

Diante desse cenário, a lacuna que motivou esta pesquisa não é apenas a ausência de soluções que gere scripts automaticamente, mas a falta de uma abordagem que combine: i) a intenção expressa pelo analista em linguagem natural; ii) informações concretas coletadas da aplicação em execução; e iii) uma estratégia de fornecimento progressivo de contexto ao modelo gerador. Este trabalho investiga exatamente essa lacuna por meio de uma abordagem de geração de scripts Frida guiada por contexto dinâmico, cuja implementação de prova de conceito é realizada no sistema *FridaForge*.

3. Geração de Scripts Frida Guiada por Contexto Dinâmico

Esta seção apresenta a abordagem proposta para geração de scripts Frida guiada por contexto dinâmico observado em tempo de execução. Em vez de tratar a geração de scripts como uma simples tarefa de síntese de código a partir de texto livre, a abordagem modela esse processo como a combinação entre: i) a intenção expressa pelo analista em linguagem natural; ii) informações concretas coletadas da aplicação em execução; e iii) uma estratégia de fornecimento progressivo de contexto ao modelo gerador. O sistema *FridaForge* constitui a implementação de referência dessa abordagem.

3.1. Visão Geral da Abordagem

A abordagem proposta organiza a geração de scripts em quatro etapas principais: i) interpretação da consulta formulada pelo analista; ii) coleta de contexto da aplicação em execução; iii) construção da entrada de geração a partir da consulta e do contexto observado; e iv) geração, validação e execução do script Frida. A Figura 1 apresenta esse fluxo de operação. A ideia central é que a adequação do script depende não apenas da capacidade do modelo de gerar código, mas também da qualidade da representação do alvo a ser instrumentado.

Nesse processo, o analista não precisa conhecer previamente a estrutura interna da aplicação nem a API do Frida em detalhe. Em vez disso, ele formula uma consulta em linguagem natural descrevendo o objetivo da instrumentação, como interceptar credenciais, observar operações criptográficas ou realizar contorno de mecanismos de proteção. A abordagem utiliza essa intenção como ponto de partida e a complementa com informações obtidas diretamente da aplicação em execução.

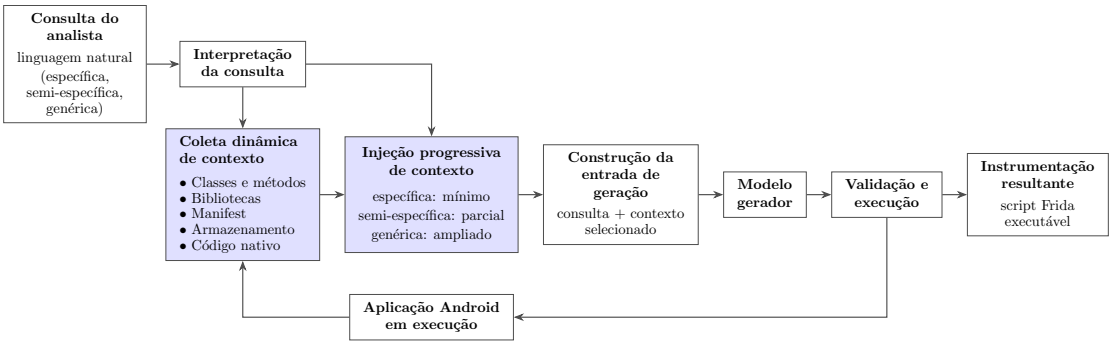

Figura 1. Visão geral da abordagem proposta para geração de scripts Frida guiada por contexto dinâmico.

3.2. Modelagem da Consulta e Especificidade

As consultas submetidas pelo analista são classificadas automaticamente, a partir de padrões sintáticos identificados no texto, em três níveis de especificidade. No nível específico, a consulta explicita diretamente classes e métodos da aplicação por meio de uma referência totalmente qualificada, como: `pacote.Classe.metodo()`, reduzindo a ambiguidade da tarefa. No nível semi-específico, a consulta menciona nomes de classes, métodos ou componentes relevantes de forma parcial, sem informar o caminho de pacote completo. Na ausência de qualquer um desses padrões, a consulta é classificada como genérica, e a entrada é tratada como uma descrição da intenção de análise em alto nível.

Essa distinção é importante porque a quantidade de informação necessária para geração varia conforme o grau de detalhe presente na consulta. Consultas mais específicas já delimitam explicitamente o alvo da instrumentação, enquanto consultas mais genéricas exigem maior apoio de contexto para que o modelo identifique quais pontos da aplicação são mais relevantes para a tarefa solicitada.

3.3. Coleta Dinâmica de Contexto

O principal elemento diferenciador da abordagem é a coleta de contexto em tempo de execução antes da geração do script. Após estabelecer conexão com a aplicação por meio do Frida, o sistema enumera classes carregadas pertencentes ao escopo do aplicativo, seus métodos e assinaturas, bibliotecas de terceiros detectadas, módulos nativos carregados e mecanismos locais de armazenamento, como bancos de dados e chaves persistidas em *SharedPreferences*.

Esse contexto busca reduzir a ocorrência de referências inexistentes nos scripts gerados. Em vez de depender exclusivamente de conhecimento prévio do modelo sobre padrões de desenvolvimento Android, a abordagem fornece evidências observadas diretamente no alvo instrumentado. Com isso, a geração passa a considerar métodos efetivamente disponíveis, bibliotecas realmente carregadas e artefatos concretos da aplicação em execução.

A Figura 2 apresenta um exemplo de contexto dinâmico coletado da aplicação InsecureBankv2. Nesse caso, o sistema identifica o pacote da aplicação, classes relevantes, permissões declaradas, mecanismos de armazenamento local e módulos nativos carregados, fornecendo uma representação útil para orientar a geração de scripts voltados à interceptação de autenticação e acesso a dados armazenados de forma insegura.

```
Package: com.android.insecurebankv2
Classes: 27 (CryptoClass, DoTransfer, TrackUserContentProvider, ...)
Permissions: INTERNET, SEND_SMS, READ_CONTACTS, ... (19)
Database: mydb (tables: names)
SharedPreferences: superSecurePassword, EncryptedUsername)
Native modules: 1 app / 366 total (no protections)
```

Figura 2. Exemplo de contexto dinâmico coletado pelo FridaForge para a aplicação InsecureBankv2.

3.4. Injeção Progressiva de Contexto

A abordagem não fornece a mesma quantidade de contexto para todas as consultas. Em vez disso, adota uma estratégia progressiva de injeção de contexto, condicionada ao nível de especificidade da entrada.

Para consultas específicas, o sistema fornece apenas a instrução do analista e regras de geração, pois o alvo da instrumentação já está explicitamente indicado. Para consultas semi-específicas, a entrada inclui as classes e métodos da aplicação identificados por seus caminhos completos de pacote, além da identificação das bibliotecas de terceiros detectadas, sem aplicação de filtragem semântica adicional sobre esse conjunto, permitindo ao modelo resolver referências parciais presentes na consulta. Para consultas genéricas, a entrada inclui contexto mais amplo, abrangendo, além de classes e bibliotecas, informações sobre o arquivo de manifesto, mecanismos de armazenamento local e módulos nativos.

Essa estratégia busca fornecer ao modelo informações úteis para a tarefa sem introduzir detalhes irrelevantes que possam prejudicar a geração do script. Em outras palavras, o objetivo é ajustar a quantidade de contexto à ambiguidade da consulta: consultas mais específicas exigem menos apoio contextual, enquanto consultas mais genéricas dependem de uma visão mais ampla da aplicação para que o modelo identifique pontos adequados de instrumentação.

3.5. Construção da Entrada de Geração

Após a coleta e seleção do contexto, a abordagem constrói uma entrada de geração composta por três elementos principais: instruções de sistema sobre a tarefa de instrumentação, descrição estruturada do contexto selecionado e consulta formulada pelo analista. O objetivo é transformar a intenção do usuário e o estado observado da aplicação em uma representação que seja útil para o modelo gerador.

Esse processo permite que consultas em alto nível sejam convertidas em scripts potencialmente executáveis. Por exemplo, ao receber a consulta “bypass de SSL pinning” em uma aplicação que carrega a biblioteca OkHttp, a entrada de geração pode incluir informações sobre a classe *CertificatePinner* e seus métodos relevantes, favorecendo a produção de scripts compatíveis com esse cenário. A Figura 3 apresenta um exemplo simplificado da estrutura de entrada utilizada no processo de geração.

```
[System]
You are an expert in Frida instrumentation for Android security testing.

[Context]
Package: owasp.sat.agoat
Dependency: OkHttp 3.8.0 (CertificatePinner)
Classes: EmulatorDetectionActivity,
  InsecureStorageSharedPrefs1Activity, ...
Relevant methods:
  CertificatePinner.check(String, List): void
  CertificatePinner.check$okhttp(...): void

[Query]
bypass SSL Pinning

[Expected Output]
Executable Frida JavaScript code only.
```

Figura 3. Exemplo simplificado da entrada de geração, combinando a consulta do analista e o contexto dinâmico coletado da aplicação.

3.6. Geração, Validação e Execução

A geração do script é realizada por um modelo de linguagem a partir da entrada construída nas etapas anteriores. O script retornado é então extraído da resposta textual e submetido a uma validação sintática antes da execução, que verifica o balanceamento de delimitadores e a presença de estruturas características de scripts Frida; em seguida, o script é injetado no processo da aplicação em execução. Essa etapa é importante para evitar que explicações textuais, trechos incompletos ou código incompatível sejam encaminhados diretamente ao ambiente de instrumentação.

Uma vez validado, o script é executado sobre a aplicação-alvo, permitindo realizar tarefas como interceptação de dados sensíveis, observação de operações criptográficas e contorno de mecanismos de proteção. Assim, o resultado final da abordagem não é apenas a síntese textual de código, mas a produção efetiva de uma instrumentação funcional sobre a aplicação em execução.

4. Experimentos e Resultados

4.1. Desenho Experimental

Os experimentos foram conduzidos em ambiente macOS, utilizando um emulador Android com API Level 36 (Android 16) e Frida versão 17.4.1 para instrumentação dinâmica. A etapa de geração foi realizada com o modelo Claude Sonnet 4, escolhido com base na experiência prévia da equipe de desenvolvimento em tarefas de geração de código. Todas as aplicações foram instaladas em um emulador configurado com permissões de root por meio do comando `adb root`, permitindo a injeção de código via Frida. A implementação de referência da abordagem, o *FridaForge*, foi executada em modo interativo, com registro das consultas submetidas, do contexto coletado e dos scripts gerados.

A avaliação experimental foi estruturada em três frentes. A primeira consiste na análise da abordagem em aplicações Android intencionalmente vulneráveis de dois repositórios, sempre com o uso do contexto dinâmico coletado em tempo de execução.

A segunda consiste em um experimento sem contexto dinâmico, realizado para quantificar a contribuição desse componente na geração dos scripts. A terceira consiste em uma comparação com soluções existentes amplamente utilizadas pela comunidade de segurança, com o objetivo de situar a cobertura prática alcançada pela abordagem proposta.

No âmbito da primeira frente, a avaliação com contexto dinâmico foi organizada em dois cenários complementares. O primeiro considera um conjunto de 10 aplicações Android intencionalmente vulneráveis, selecionadas a partir do catálogo de aplicações de referência do OWASP Mobile Application Security Testing Guide (MASTG) [OWASP MASTG 2026]. A seleção priorizou aplicações que contemplassem diversidade de vulnerabilidades, mecanismos de proteção instrumentáveis e diferentes tecnologias de implementação, incluindo aplicações desenvolvidas em Java, Kotlin e Jetpack Compose. A Tabela 1 apresenta o conjunto de aplicações utilizado.

Tabela 1. Aplicações utilizadas na avaliação experimental.

ID	Aplicação	Ref.	Tarefas	Consultas[1]
A1	AndroGoat	MASTG-APP-0001	15	20
A2	DIVA	MASTG-APP-0007	13	17
A3	DodoVulnerableBank	MASTG-APP-0008	8	8
A4	InsecureBankv2	MASTG-APP-0010	8	8
A5	OVAA	MASTG-APP-0013	6	6
A6	Finstergram	MASTG-APP-0016	6	6
A7	MASTestApp-NETWORK	MASTG-APP-0018	6	6
A8	BugBazaar	MASTG-APP-0029	6	6
A9	VulnForum	MASTG-APP-0031	7	7
A10	Damn Vulnerable Bank	EXT-001	8	8
	Total		**83**	**92**

As 83 tarefas correspondem a objetivos de análise e instrumentação definidos para as aplicações avaliadas, a partir das vulnerabilidades, fluxos sensíveis e mecanismos de proteção presentes em cada app. Essas tarefas cobrem categorias de análise de segurança alinhadas ao OWASP MASVS [OWASP MASVS 2026], incluindo contorno de mecanismos de proteção, interceptação de dados sensíveis, extração de informações protegidas, monitoramento de componentes de plataforma e inspeção de práticas criptográficas e de rede inseguras. Para cada tarefa, foram formuladas consultas em até três níveis de especificidade, conforme descrito na Seção 3. No total, foram utilizadas 92 consultas em linguagem natural. Cada consulta foi executada cinco vezes, totalizando 460 execuções, de modo a capturar a variabilidade inerente à geração por modelos de linguagem e permitir observar a consistência dos resultados entre execuções independentes. Esse valor foi adotado como compromisso entre estabilidade da medida e custo experimental. Entre execuções consecutivas, o estado da aplicação foi reiniciado com os comandos `adb shell am force-stop` e `adb shell pm clear`, a fim de reduzir interferências entre rodadas experimentais.

O segundo cenário consiste em uma avaliação complementar com aplicações do repositório Ghera [Mitra and Ranganath 2017], utilizada para observar o comportamento

[1]Algumas tarefas foram avaliadas com mais de uma consulta, variando o nível de especificidade (específica, semi-específica ou genérica), o que explica a diferença entre o total de tarefas e o de consultas.

da abordagem em aplicações que isolam vulnerabilidades específicas. Os detalhes desse cenário complementar são apresentados na Subseção 4.2.2.

O critério de sucesso adotado foi binário. Uma execução foi considerada bem-sucedida quando o script gerado executou corretamente e atingiu o objetivo da tarefa proposta, como contornar um mecanismo de proteção, interceptar o dado de interesse ou expor a operação alvo. Caso o script não executasse, falhasse sintaticamente, ou produzisse um resultado desalinhado da tarefa solicitada, a execução era contabilizada como falha.

4.2. Resultados

4.2.1. Análise das Aplicações Vulneráveis

A Tabela 2 apresenta os resultados consolidados por aplicação no conjunto de 10 aplicações Android intencionalmente vulneráveis avaliadas neste trabalho.

Tabela 2. Resultados consolidados por aplicação.

Aplicação	Tarefas	Execuções	Sucessos	Taxa
A1 – AndroGoat	15	100	99	99,0%
A2 – DIVA	13	85	85	100%
A3 – DodoVulnerableBank	8	40	29	72,5%
A4 – InsecureBankv2	8	40	32	80,0%
A5 – OVAA	6	30	30	100%
A6 – Finstergram	6	30	30	100%
A7 – MASTestApp-NETWORK	6	30	30	100%
A8 – BugBazaar	6	30	30	100%
A9 – VulnForum	7	35	35	100%
A10 – Damn Vulnerable Bank	8	40	40	100%
Total	**83**	**460**	**440**	**95,7%**

No conjunto avaliado, a abordagem alcançou taxa geral de sucesso de 95,7% (440/460 execuções). Das 10 aplicações analisadas, 7 alcançaram 100% de sucesso e 2 concentraram a maior parte das falhas observadas: DodoVulnerableBank (72,5%) e InsecureBankv2 (80,0%). As falhas observadas concentram-se em três categorias principais: incompatibilidades estruturais com bibliotecas legadas, como o Apache HttpClient, principal causa das falhas em DodoVulnerableBank; interceptações em métodos genéricos ou APIs de alto volume, o que pode sobrecarregar o ambiente de execução ou não atingir o alvo correto; e limitações na visibilidade do contexto coletado, como classes internas de `AsyncTask` não enumeradas ou seleção incorreta de overloads em APIs de I/O, sendo essas duas últimas categorias responsáveis pela maior parte das falhas em InsecureBankv2. Em sua maioria, esses modos de falha são identificáveis a partir da saída da instrumentação, o que sugere que refinamentos adicionais na coleta de contexto ou na etapa de geração podem reduzir esse impacto. Esses resultados indicam que a abordagem mantém desempenho elevado em uma diversidade de aplicações vulneráveis, embora existam cenários específicos em que a geração de scripts ainda apresenta dificuldades.

A Tabela 3 apresenta os resultados agregados por tipo de consulta. Consultas específicas e semi-específicas obtiveram 100% de sucesso, enquanto consultas genéricas alcançaram 94,8%. Todas as 20 falhas observadas ocorreram em consultas genéricas.

Tabela 3. Resultados por tipo de consulta.

Tipo de Consulta	Execuções	Sucessos	Taxa
Específica	25	25	100%
Semi-Específica	40	40	100%
Genérica	395	375	94,8%
Total	**460**	**440**	**95,7%**

Esse resultado mostra que consultas com maior grau de especificidade tendem a reduzir a ambiguidade da tarefa de instrumentação. Por outro lado, como consultas genéricas constituem a maior parte do conjunto avaliado, seu desempenho continua sendo particularmente relevante para a caracterização da abordagem em cenários de uso mais flexíveis.

A Tabela 4 apresenta o contexto coletado por aplicação. Observa-se que a quantidade e o tipo de informações variam conforme a estrutura de cada aplicativo, refletindo diferenças em bibliotecas, mecanismos de armazenamento e componentes nativos presentes em runtime.

Tabela 4. Contexto coletado por aplicação.

Aplicação	Classes	Bibliotecas	Nativos	Armazenamento
A1 – AndroGoat	22	1	0	SQLite, SharedPrefs
A2 – DIVA	16	0	1	SQLite
A3 – DodoVulnerableBank	18	0	0	SQLite, SharedPrefs
A4 – InsecureBankv2	27	0	0	SQLite, SharedPrefs
A5 – OVAA	21	3	0	SharedPrefs
A6 – Finstergram	34	0	0	SQLite, SharedPrefs
A7 – MASTestApp-NET.	12	1	0	–
A8 – BugBazaar	45	5	0	SQLite, SharedPrefs
A9 – VulnForum	198	3	0	SharedPrefs, Arquivo
A10 – DVB	24	0	2	SharedPrefs

Como complemento à avaliação nas aplicações vulneráveis mais completas, analisamos também o desempenho da abordagem nas aplicações do repositório Ghera [Mitra and Ranganath 2017]. Diferentemente do conjunto anterior, no qual cada aplicação reúne múltiplas funcionalidades e múltiplas tarefas de segurança, as aplicações do repositório Ghera são projetadas para isolar uma vulnerabilidade específica.

Das 60 aplicações disponíveis no repositório, 54 delas foram consideradas adequadas para instrumentação dinâmica. As demais foram descartadas por envolverem vulnerabilidades puramente declarativas[2] ou por exigirem infraestrutura backend indisponível. Para cada aplicação considerada, foi formulada uma consulta genérica descrevendo a intenção de análise, executada cinco vezes (N=5), totalizando 270 execuções.

[2]Decorrentes de características estáticas do código ou configuração, sem manifestação observável em tempo de execução.

A Tabela 5 apresenta os resultados consolidados por categoria.

Tabela 5. Resultados da avaliação complementar no repositório Ghera.

Categoria	Aplicações	Execuções	Sucessos	Taxa
Crypto	5	25	17	68,0%
ICC	13	65	64	98,5%
Networking	8	40	31	77,5%
NonAPI	1	5	5	100%
Permission	2	10	10	100%
Storage	7	35	35	100%
System	7	35	35	100%
Web	11	55	54	98,2%
Total	**54**	**270**	**251**	**93,0%**

Nesses experimentos, a abordagem alcançou taxa geral de sucesso de 93,0% (251/270 execuções). As menores taxas ocorreram em Networking (77,5%) e Crypto (68,0%), refletindo maior dificuldade em APIs menos frequentes ou estruturas mais específicas. No Crypto, falhas concentram-se em `KeyStore.setEntry` (0%), enquanto no Networking o modelo confunde interfaces com implementações concretas e omite overloads corretos.

Em conjunto, esses resultados mostram que a abordagem permanece eficaz mesmo em aplicações de pequeno porte e com contexto limitado, ainda que evidenciando maior variabilidade nessas categorias ao mesmo tempo em que evidenciam categorias de vulnerabilidade em que a geração de scripts ainda apresenta maior variabilidade.

4.2.2. Impacto do Contexto Dinâmico

Para avaliar a contribuição do contexto dinâmico na geração de scripts, conduzimos um experimento no qual o sistema operou sem o bloco de contexto coletado em tempo de execução. A fim de isolar o efeito causal do contexto, utilizamos uma amostra estratificada de tarefas, deliberadamente contrastante em relação ao grau de dependência da estrutura interna da aplicação, em vez do conjunto completo de tarefas avaliadas. O protocolo manteve a entrada padrão do FridaForge e o identificador do aplicativo alvo, suprimindo exclusivamente o bloco de contexto dinâmico; as demais condições foram mantidas: mesmo modelo, mesma consulta genérica por tarefa e $N = 5$ execuções por tarefa.

Para esse experimento, selecionamos 12 tarefas do conjunto de aplicações vulneráveis, organizadas em dois grupos segundo o grau de dependência em relação ao contexto da aplicação. O primeiro grupo reúne tarefas cujo objetivo pode ser atingido por meio de APIs genéricas do framework Android, como *bypass* de SSL pinning, interceptação de SharedPreferences, operações criptográficas e extração de credenciais em SQLite. O segundo grupo reúne tarefas cuja instrumentação depende da estrutura interna da aplicação-alvo, como fluxos de autenticação, operações bancárias e acesso a classes específicas não documentadas publicamente. A Tabela 6 apresenta os resultados das 60 execuções realizadas sem contexto dinâmico e os compara com aqueles obtidos no experimento principal.

Tabela 6. Comparação entre execuções sem contexto dinâmico e o experimento principal (taxa de sucesso em %).

Grupo	Tarefa	Sem Contexto	Com Contexto
APIs genéricas do framework	Bypass de SSL pinning	80	100
	Leitura de SharedPreferences	100	100
	Interceptação criptográfica	80	80
	Exploração de path traversal	40	100
	Extração de credenciais SQLite	100	100
Estrutura interna da aplicação	Bypass de autenticação	0	100
	Captura de credenciais de login	60	100
	Interceptação de transferências	0	100
	Interceptação de beneficiários	20	100
	Extração de histórico bancário	100	100
	Extração de dados de perfil	80	100
	Captura de alteração de senha	60	100
Geral		**60**	**98,3**

Os resultados mostram que o impacto do contexto dinâmico varia de acordo com a natureza da tarefa. No primeiro grupo, predominam tarefas que podem ser resolvidas com conhecimento amplamente disponível sobre APIs do ecossistema Android. Nesses casos, a configuração sem contexto dinâmico obteve desempenho próximo ao experimento principal, alcançando 100% em SharedPreferences e SQLite, e mantendo desempenho equivalente na interceptação criptográfica. A principal exceção foi a tarefa de *path traversal*, na qual a geração sem contexto distribuiu as interceptações por APIs genéricas do framework, enquanto a geração com contexto instrumentou diretamente classes específicas da aplicação, como `TheftOverwriteProvider.openFile` e `FileUtils.copyToCache`, elevando a taxa de sucesso de 40% para 100%.

O comportamento muda substancialmente no segundo grupo, formado por tarefas que dependem da estrutura interna da aplicação-alvo. Nessas situações, a ausência de contexto leva o modelo a inferir fluxos e classes plausíveis, porém inexistentes na implementação real. Isso ocorre, por exemplo, no *bypass* de autenticação do InsecureBankv2, em que a configuração sem contexto dinâmico obteve 0% ao interceptar métodos inexistentes, como `processLogin` e `validateCredentials`, sem identificar que o fluxo real passa por `DoLogin.doInBackground` via `AsyncTask`. O mesmo padrão aparece em tarefas bancárias, nas quais a ausência de contexto levou à geração de scripts baseados em uma arquitetura genérica, com referências a classes inexistentes como `TransferActivity`, `BankingService` e `AccountManager`.

Mesmo nesse segundo grupo, observa-se que tarefas parcialmente dependentes de estruturas internas ainda podem apresentar sucesso intermediário quando há apoio de APIs genéricas do framework. Isso ocorre na captura de credenciais de login e na alteração de senha, em que a configuração sem contexto dinâmico atingiu 60% ao explorar mecanismos como SharedPreferences e `JSONObject`. Contudo, sem o contexto coletado em execução, o modelo não interceptou classes internas responsáveis pelas chamadas de rede via `AsyncTask`, o que limitou a cobertura completa das tarefas.

No agregado, a diferença entre a configuração sem contexto dinâmico (60,0%) e o experimento com contexto (98,3%) indica que o contexto coletado em tempo de execução tem papel particularmente relevante em tarefas cuja instrumentação depende

de fluxos, classes e pontos de interceptação específicos da aplicação-alvo. Esses resultados reforçam que a principal contribuição do contexto dinâmico não está em substituir o conhecimento prévio do modelo sobre APIs amplamente conhecidas, mas em reduzir ambiguidades quando a tarefa exige acesso a estruturas internas não explicitadas na consulta.

4.2.3. Comparação com Soluções Existentes

A Tabela 7 situa o *FridaForge* em relação a duas soluções amplamente adotadas pela comunidade de segurança móvel: Objection [Objection 2026], toolkit de exploração em tempo de execução construído sobre o Frida, e o Frida CodeShare [Frida CodeShare 2026], principal repositório comunitário de scripts reutilizáveis. O objetivo não é avaliar as ferramentas em condições equivalentes, mas contrastar dois paradigmas: comandos embutidos e fixos, que caracterizam as soluções existentes, e a geração sob demanda guiada por contexto dinâmico, proposta neste trabalho.

Tabela 7. Comparação entre FridaForge, Objection e Frida CodeShare.

Critério	FridaForge	Objection	CodeShare
Cobertura de tarefas	83/83 (100%)	3/83 (3,6%)	0/83 (0%)
Automação	Ling. natural	Built-in	Manual
Contexto do app	Sim	Não	Não
Requer expertise Frida	Não	Parcial	Sim

Nessa perspectiva, o Objection oferece suporte nativo a um conjunto bem definido de funcionalidades recorrentes, como *bypass* de SSL pinning e detecção de root, cobrindo 3 das 83 tarefas avaliadas. Essa limitação não decorre de deficiência na implementação, mas da própria natureza de sua arquitetura, baseada em operações pré-definidas que não se adaptam à estrutura interna de cada aplicação. O Frida CodeShare, por sua vez, amplia a disponibilidade de scripts por meio de contribuições comunitárias, porém exige busca manual, seleção e adaptação ao contexto do alvo, razão pela qual não foi contabilizado como solução com cobertura automática direta. A principal diferença evidenciada pela comparação é que a abordagem proposta combina automação via linguagem natural com contexto dinâmico da aplicação, permitindo gerar scripts específicos para o alvo instrumentado sem depender de um catálogo fixo de operações nem de customização manual.

4.3. Ameaças à Validade

Em relação à validade interna, a etapa de geração utiliza um modelo de linguagem não determinístico, de modo que a mesma consulta pode produzir scripts estruturalmente distintos entre execuções. Para reduzir esse efeito, cada consulta foi executada cinco vezes. Além disso, o experimento sem contexto dinâmico foi conduzido em um conjunto selecionado de tarefas, adequado para evidenciar a contribuição do contexto, embora não esgote todas as combinações possíveis de tarefas e aplicações avaliadas neste estudo. Adicionalmente, a avaliação binária de sucesso foi definida e aplicada pela mesma equipe responsável pelo sistema, o que introduz risco de viés de confirmação, parcialmente mitigado pela objetividade do critério (execução correta do script e captura do alvo).

Em relação à validade externa, a avaliação foi conduzida em aplicações intencionalmente vulneráveis amplamente utilizadas na comunidade de segurança móvel e em

aplicações do repositório Ghera, projetadas para isolar vulnerabilidades específicas. Esse desenho permite observar o comportamento da abordagem em cenários diversificados e controlados, mas não cobre toda a complexidade arquitetural, o grau de ofuscação e os mecanismos de proteção presentes em aplicações de produção. A generalização da abordagem para aplicações com proteções mais extensas, como frameworks RASP e ofuscação de código via ferramentas comerciais, permanece como questão em aberto. Adicionalmente, os experimentos foram realizados com um único modelo de linguagem, o Claude Sonnet 4, de modo que investigações futuras com outros modelos podem complementar a análise apresentada.

5. Conclusão

Este artigo investigou a geração de scripts Frida guiada por contexto dinâmico para análise de segurança Android. Partindo da hipótese de que informações coletadas em tempo de execução podem melhorar a adequação, a executabilidade e a consistência dos scripts gerados por modelos de linguagem, propusemos uma abordagem que combina a intenção expressa pelo analista em linguagem natural com contexto observado diretamente na aplicação em execução. O sistema *FridaForge* foi utilizado como implementação de referência dessa abordagem.

A avaliação empírica em 10 aplicações Android intencionalmente vulneráveis, abrangendo 83 tarefas de segurança, 92 consultas e 460 execuções, indicou taxa de sucesso de 95,7%. Esse resultado foi complementado por uma avaliação em 54 aplicações do repositório Ghera, na qual a abordagem alcançou 93,0% de sucesso em cenários de vulnerabilidades isoladas. Além disso, o experimento sem contexto dinâmico mostrou uma diferença expressiva em relação ao experimento principal, especialmente em tarefas dependentes da estrutura interna da aplicação, reforçando a relevância do contexto coletado em tempo de execução para a geração de scripts mais adequados ao alvo instrumentado. Como trabalhos futuros, destacam-se a integração com mecanismos de análise estática de binários nativos, a incorporação de refinamento iterativo a partir da saída da instrumentação e a ampliação da avaliação para aplicações reais e outros modelos de linguagem.

Agradecimentos

O presente trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (AUXPE-CAPES-PROEX) - Código de Financiamento 001. Adicionalmente, este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado do Amazonas - FAPEAM - por meio dos projetos PDPG-CAPES e POSGRAD 2025-2026 e POSGRAD 2026-2027.

Declaração de Uso de IA Generativa

Em conformidade com o Código de Conduta para Autores em Publicações da SBC, declaramos os seguintes usos de Inteligência Artificial Generativa neste trabalho: i) o modelo Claude (Anthropic) é utilizado como componente central do sistema FridaForge, utilizado como implementação de referência da abordagem. conforme descrito na Seção 3, constituindo parte do objeto de estudo do artigo; e ii) ferramentas de IA generativa foram utilizadas como apoio à revisão linguística do manuscrito. Os autores assumem total responsabilidade pelo conteúdo final do trabalho.

Referências

Chen, M., Tworek, J., Jun, H., et al. (2021). Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

D'Elia, D. C., Coppa, E., Nicchi, S., Palmaro, F., and Cavallaro, L. (2019). Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, ACM Asia CCS '19, pages 15–27, New York, NY, USA.

Enck, W., Gilbert, P., Han, S., et al. (2014). TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2).

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53.

Frida (2026). Frida: A world-class dynamic instrumentation toolkit. `https://frida.re`. Acesso em: abril de 2026.

Frida CodeShare (2026). Frida CodeShare: Community Frida scripts repository. `https://codeshare.frida.re/`. Acesso em: abril de 2026.

MEDUSA (2026). MEDUSA: Mobile edge-dynamic unified security analysis. `https://github.com/Ch0pin/medusa`. Acesso em: abril de 2026.

Michel, G.-B. (2020). Dexcalibur: Android reverse engineering platform. `https://github.com/FrenchYeti/dexcalibur`. Acessado em: abril de 2026.

Minn, W., Demissie, B. F., Tun, Y. N., Liu, J., Ceccato, M., Shar, L. K., and Lo, D. (2026). Covagent: Overcoming the 30% curse of mobile application coverage with agentic ai and dynamic instrumentation. *arXiv preprint arXiv:2601.21253*.

Mitra, J. and Ranganath, V.-P. (2017). Ghera: A repository of android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, pages 43–52, Toronto, Canada. ACM.

Objection (2026). Objection: Runtime mobile exploration. `https://github.com/sensepost/objection`. Acesso em: abril de 2026.

OWASP MASTG (2026). OWASP MASTG – apps. `https://mas.owasp.org/MASTG/apps/`. Acessado em: abril de 2026.

OWASP MASVS (2026). OWASP mobile application security verification standard (MASVS). `https://mas.owasp.org/MASVS/`. Acesso em: maio de 2026.

Pearce, H., Tan, B., Ahmad, B., Karri, R., and Dolan-Gavitt, B. (2023). Examining zero-shot vulnerability repair with large language models. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, pages 2339–2356.

Rozière, B., Gehring, J., Gloeckle, F., et al. (2023). Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950.

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

Xu, H., Wang, S., Li, N., Wang, K., Zhao, Y., Chen, K., Yu, T., Liu, Y., and Wang, H. (2025). Large language models for cyber security: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*