Assessing the Security Coverage of the Google Play Integrity API on Android

Francis Vargas[1], **Angelo Gaspar**[1], **Diego Kreutz**[1], **Rodrigo Mansilha**[1]

[1]AI Horizon Labs – PPGES – Universidade Federal do Pampa (UNIPAMPA)

{francisvargas,angelonogueira}.aluno@unipampa.edu.br
{diegokreutz,rodrigomansilha}@unipampa.edu.br

Abstract. *This work evaluates the Google Play Integrity API using experiments across four attack scenarios: compromised environments, APK tampering, dynamic instrumentation, and request replay. The results show that the API correctly flags device and binary modifications but remains ineffective against runtime manipulation and replay attacks. We conclude that Play Integrity is useful as an integrity signal source but must be combined with server-side verification, nonces, and application hardening to provide meaningful protection in real Android applications.*

1. Introduction

With Android established as the dominant mobile platform ($\approx$ 72% in 2025)[1], attack surfaces such as *root*, unofficial ROMs, repackaging, and traffic interception continue to expand, as documented in prior work [Kim et al. 2021]. Despite layered security architectures and Google Play Protect, recent security bulletins [2] continue to report critical vulnerabilities across the ecosystem, highlighting persistent security challenges.

In this security context, the Play Integrity API has been introduced as the official successor to SafetyNet, providing a unified attestation mechanism that evaluates device integrity, application authenticity, and account validity[3] [Niemi et al. 2023] . Despite these advancements, important questions remain regarding its practical resilience against advanced threat vectors, including dynamic instrumentation, APK tampering, and adversarial network manipulation. Several of these limitations were extensively documented in its predecessor, which reinforces the need for a systematic empirical assessment of the current implementation.

This paper presents a controlled empirical evaluation to systematically map the Play Integrity API's (Application Programming Interface) security coverage. We investigate four critical scenarios: device integrity compromise, APK modification, runtime instrumentation, and request handling under adverse conditions, while comparing Classic and Standard operational modes. Our contribution provides a granular assessment of native protections versus those requiring external controls, offering evidence-based guidance for integrating the API into comprehensive Android security strategies.

[1]https://gs.statcounter.com/os-market-share/mobile/worldwide
[2]https://source.android.com/docs/security/bulletin
[3]https://developer.android.com/google/play/integrity

2. Google Play Integrity API

The Play Integrity API verifies whether an app interaction originates from a legitimate binary running on an authentic Android device, providing signals that help the backend determine whether to permit or block sensitive actions. Verdicts are delivered as JSON objects and consolidate indicators related to the application, device, account, and runtime environment. Figure 1 illustrates the attack vectors addressed by these verdicts in the Play Integrity API.

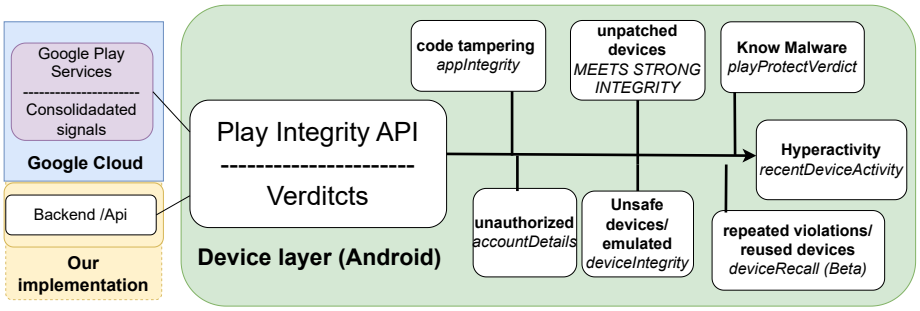

Figure 1. Attack vectors covered by Play Integrity API verdicts.

The API provides two operational modes, Standard and Classic, summarized in Table 1 as well as their capabiliteis and limitation in Table 2. In this work, we adopted both modes, in which the application obtains an integrity token and sends it to the backend, which forwards it to Google Play Integrity for decryption and validation before enforcing the corresponding decision based on the returned signals. In Classic mode, the backend must additionally generate and verify a per-request *nonce*.

Table 1. Play Integrity: Operational Modes

Aspect	Standard (*StandardIntegrity*)	Classic (*IntegrityManager*)
Typical purpose	Recurring checks with low latency (*rate-limited*).	One-off/high-value actions with tighter control over the flow.
Mitigations	Automatic (Google Play): device-side cache, `requestHash` (action binding), in-path replay mitigation.	Backend-driven: unique/unpredictable nonce, non-repudiation, channel/request binding, time window/expiration.
Latency (approx.)	Hundreds of ms (with warm-up/cache).	Seconds (fresh evaluation on each invocation).
Developer responsibility	Low: integrate the token and validate server-side.	High: generate/validate nonce, enforce non-repudiation, revocation and binding policies.

3. Related Work

Table 3 summarizes the principal studies on attestation and integrity verification for Android devices, listing for each work the API used, its scope, and methodology. Two groups of contributions are evident: threat analyses and authentication or security mechanisms, and experimental studies that employ Play Integrity or SafetyNet to construct security controls. Analyses include [Ibrahim et al. 2021], which examines SafetyNet and its vulnerabilities; [Niemi et al. 2023], which surveys attestation platforms but offers only limited discussion of concrete security mechanisms; and [Ruggia et al. 2024], which analyzes Android malware and how Google Play's current authentication tools address it. Proposals include [Steinböck et al. 2025], which develops an anti-tampering tool and evaluates mobile applications, [Samper and Ferreira 2024],

Table 2. Play Integrity: Capabilities and Common Limitations Across Modes

Aspect	Common capabilities and limitations
Robustness (HW/TEE)	Exposes `MEETS_BASIC/DEVICE/STRONG_INTEGRITY`; `STRONG` requires device HW/TEE support (varies by OEM/model/ROM).
Anti-tampering (APK)	`appRecognitionVerdict` (PR/UV) validates version/signature; `requestHash` binds the verdict to the protected action (in Standard), while in Classic the *nonce* fulfills this role via server-side validation.
Compromised environment	Covers root/bootloader/emulation via multi-layer signals (static/dynamic/behavioral). **Does not** provide anti-instrumentation (Frida/Xposed) on its own—requires client-side RASP/obfuscation and server-side policies.
Consolidated signals	Unified payload: Application (PR/UV), Device (basic/device/strong), Licensing (L/U), `requestDetails` (timestamp, hash/nonce).
Scenario coverage	**C1** (device) and **C2** (APK) are covered; **C3** (instrumentation) is out of scope; **C4** (replay/non-repudiation) depends on the backend.

Notes: (i) `MEETS_STRONG_INTEGRITY` depends on HW/TEE and OEM policies; (ii) replay defense/non-repudiation is a backend responsibility (especially in Classic mode); (iii) anti-instrumentation requires client hardening (RASP/obfuscation) and server-side signal correlation.

Table 3. Synthesis of related work.

Identification	API(s)	Objective / Scope	Methodology
SafetyNOT (Ibrahim et al., 2021)	SafetyNet	Large-scale misuse of SafetyNet	Empirical measurement (large-scale)
XFVSes (Zhang et al., 2023)	SafetyNet; Play Integrity; Camera APIs	Security of cross-side facial verification	Measurement + semi-automated detection; tool
Platform Attestation Survey (Niemi et al., 2023)	DHA; Knox; Play Integrity; etc.	Comparative survey of platform attestation	SLR/Survey; comparative analysis
Leveraging Remote Attestation (Samper & Ferreira, 2024)	Play Integrity; DeviceCheck/App Attest	Secure middleware for messaging with RA	Prototype; lab-controlled experiment
SoK: Hardening Techniques (Steinböck et al., 2025)	SafetyNet; Play Integrity; App Attest	Adoption/effectiveness of RASP	Hybrid analysis + tool (HALY)
Unmasking the Veiled (Ruggia et al., 2024)	SafetyNet; Play Integrity	Evasion (DETs/IETs) in malware/goodware	Dynamic analysis; sandbox; probes
This work	**Play Integrity**	Coverage/limitations by scenario (C1–C4)	Lab-controlled; app + backend

which analyzes device-to-device communication scenarios, and [Zhang et al. 2023], which investigates security weaknesses in real applications.

Our work differs from prior studies by empirically comparing Play Integrity's Standard and Classic modes in a real application and backend deployment across four scenarios (C1–C4), by disentangling what the API verifies natively (app, device, and licensing verdicts) from what requires server-side logic (non-repudiation, replay and relay protection, action binding) and quantifying accuracy per layer, and by demonstrating that dynamic instrumentation tools such as Frida can circumvent client-side checks, highlighting the need for minimum client hardening (RASP or obfuscation) and server-side signal correlation. The literature indicates that platform attestations provide strong device and package guarantees but are insufficient to prevent in-app manipulation or replay and relay attacks without backend validation, with persistent gaps in runtime state assessment, cryptographic binding, and robust client-side enforcement.

4. Methodology

We conducted an empirical validation in a controlled environment, treating the Play Integrity API as an external signal provider for backend decision-making, and all artifacts required to reproduce our experiments[4] are publicly accessible. Instead of analyzing

[4]`https://github.com/francis-vargas/assessing-play-integrity-security`

internal mechanisms, we focused on production-relevant evidence such as app binary and signature verification, device integrity state, licensing and account checks, and environment or abuse indicators. These verdicts were processed by the backend to enforce blocking or mitigation policies across the evaluated scenarios.

To examine the effectiveness and limitations of the Play Integrity API under realistic adversarial conditions, we built a testbed composed of a Python 3.x backend for token validation, multiple Android devices in distinct states (clean, rooted, and emulated), and an application implementing both Standard and Classic API modes. Using tools such as Frida, apktool, and apksigner, we simulated APK tampering, dynamic instrumentation, replay and concurrency attacks, and communication interference. Consolidated backend logs and Integrity verdicts supported a direct comparison between the two API modes and enabled a systematic analysis of their limitations and corresponding countermeasures.

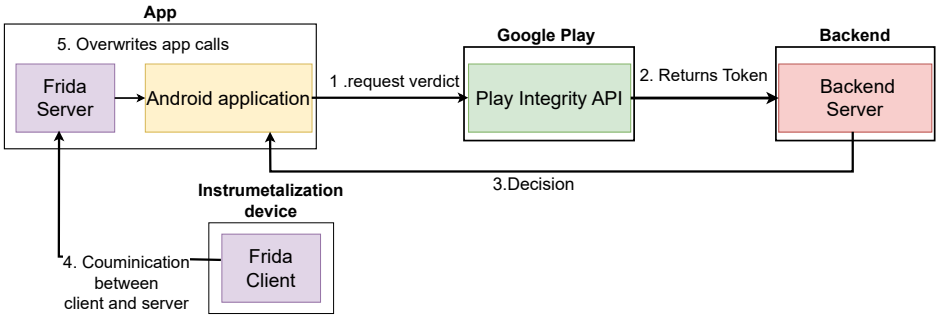

Figure 2. Test flowchart of Play Integrity API.

Figure 2 shows the nominal execution flow (solid line) and the Frida bypass path. To evaluate client-side bypass, we executed Frida 15.1.17 on a rooted device (with *frida-server*) to overwrite the app's validation methods (requestClassicToken(), requestStandardToken(), requestCombinedToken()) at runtime. The injected JavaScript hook intercepted these calls and suppressed token issuance while logging activity as "blocked," effectively neutralizing the client-side integration without influencing the API itself. This provides direct evidence that local enforcement is bypassable. The implications are clear: backend-only enforcement should be mandatory through the rejection of actions not accompanied by a fresh, action-bound token, supported by nonce strategies with anti-replay and channel binding, in addition to minimum client hardening measures such as obfuscation, anti-hooking, RASP, and telemetry collection.

4.1. Evaluation parameters and values

The parameters used in the evaluation and their assigned values are summarized in Table 4. We varied device type, Android version, enabled tools, nonce configuration, and API invocation mode. The application sequentially requested tokens from multiple devices to ensure comparable inputs. The resulting measurements were analyzed descriptively, examining proportions and variability across equivalent runs, with emphasis on consistency, predictability, and detection boundaries. Latency was excluded, as it is not a controlled or meaningful variable for security assessment.

4.2. Evaluation scenarios

The evaluation was structured into four scenarios that encompass the main attack vectors, including device state, APK tampering, dynamic instrumentation, replay attempts, and

Table 4. Parameters and values.

Parameters	Values
Device type	Emulator*; Real (non-rooted)*; Real (rooted)*
Android version (environment)	10 (real); 10 (rooted)*; 11 (real)*; 13 (real)*; 14 (emulated)*; 15 (real)*; 16 (emulated)*
App version	5 (1.0.4); 5 (1.0.4) tampered; 11 (1.1.1)
Enabled tool	None*; Frida; Magisk*
Nonce strategy	Reused; Unique*
API mode	Classic*; Standard*

Note: Items marked with * repeat across all test scenarios.

communication interference. Table 5 summarizes the API coverage and the rationale for each scenario, outlining how each test case targets specific integrity guaranties and exposes the limits of Play Integrity in isolation.

Table 5. Experimental scenarios evaluated with the Play Integrity API.

Scenario	Covered by API spec	Motivation	Method	Specif. Param.
S1: **Device integrity**	Yes	Differentiate clean devices from compromised ones (root/bootloader), validating protection against tampered environments.	Real vs. rooted comparison	App=5 (1.0.4)
S2: **APK modification**	Yes	Detect tampering (recompilation/ re-signing), simulating reverse engineering and redistribution of modified apps.	APK modification/ rebuild	App=5 (1.0.4) tampered
S3: **Adversarial runtime**	No	Check whether runtime manipulation (Frida/Xposed) typical of dynamic instrumentation is detected.	Simple Frida hook	5 App=(1.0.4) tampered, Enabled tool=Frida
S4: **Repeated and concurrent requests**	Partial (depends on backend validation)	Assess resilience to replay/stress and the need for complementary server-side controls.	Controlled nonce reuse	Nonce strategy= Reused nonce, App version=11 (1.1.1)

Note: Unless otherwise specified, iterate over all default parameter values listed in Table 4.

5. Results and Discussion

Table 6 summarizes the outcomes for scenarios S1–S4, covering app status (PR/UV/∅), device signals (`basic`, `device`, `strong`), licensing (L/U), and Android or app versions; both Classic and Standard modes exhibited identical behavior. In S1 (device integrity), non-rooted devices produce PR with `basic`+`device` and `strong` when TEE or hardware-backed attestation is available, whereas rooted devices suppress integrity signals but may still report PR and L if the app was legitimately installed. In S2 (APK modification), repackaging consistently results in UV, altered `certificateSha256Digest`, and licensing U. In S3 (adversarial runtime), Frida-based instrumentation does not alter verdicts (PR and `basic`/`device`), demonstrating that Play Integrity does not function as an anti-hooking mechanism. In S4 (repeated and concurrent requests), the first request is valid, but replaying the same nonce produces no anti-replay response and must be rejected exclusively by server-side logic. Overall, Play Integrity reliably reports the environment and package state (S1–S2) but offers no guaranties against manipulation after attestation or out-of-channel replay (S3–S4), which reinforces that its effectiveness depends on disciplined backend enforcement rather than the intrinsic precision of the signals.

Notation: PR=`PLAY_RECOGNIZED`, UV=`UNRECOGNIZED_VERSION`, ∅=no payload; device `basic`/`device`/`strong`=`MEETS_BASIC`/`DEVICE`/`STRONG_INTEGRITY`; L/U=`LICENSED`/`UNEVALUATED`.

Table 6. Accuracy split by coverage

Scenario	Android	App version	Device	Request	App	basic	device	strong	Lic.	n	Prec. (API)	Prec. (Back.)
T1: Is the device intact (no root / unlocked bootloader / emulation)?												
S1.1	13	5 (1.0.4)	Real (non-rooted)	Classic/Standard	PR	basic	device		L	3	100%	–
S1.2	15	5 (1.0.4)	Real (non-rooted)	Classic/Standard	PR	basic	device		L	3	100%	–
S1.3	10	5 (1.0.4)	Real (rooted)	Classic/Standard	PR				U	3	100%	–
S1.4	16	5 (1.0.4)	Emulated	Classic/Standard	∅				U	3	100%	–
S1.5	11	5 (1.0.4)	Real (non-rooted)	Classic/Standard	PR	basic	device	strong	L	3	100%	–
S1.6	14	5 (1.0.4)	Emulated	Classic/Standard	∅				U	3	100%	–
T2: Is the app intact and recognized (not repackaged / re-signed)?												
S2.1	13	5 (1.0.4) (mod)	Real (non-rooted)	Classic/Standard	UV				U	3	100%	–
S2.2	10	5 (1.0.4) (mod)	Emulated	Classic/Standard	∅				U	3	100%	–
S2.3	15	5 (1.0.4) (mod)	Real (non-rooted)	Classic/Standard	UV				U	3	100%	–
S2.4	16	5 (1.0.4) (mod)	Emulated	Classic/Standard	∅				U	3	100%	–
S2.5	11	5 (1.0.4) (mod)	Real (non-rooted)	Classic/Standard	UV				U	3	100%	–
S2.6	14	5 (1.0.4) (mod)	Emulated	Classic/Standard	∅				U	3	100%	–
T3: Is client-side dynamic instrumentation (Frida) detected?												
S3.1	13	5 (1.0.4) (mod)	Real (non-rooted)	Classic/Standard	UV	basic	device		U	3	–	–
S3.2	10	5 (1.0.4) (mod)	Real (rooted)	Classic/Standard	∅				U	3	–	–
S3.3	15	5 (1.0.4) (mod)	Real (non-rooted)	Classic/Standard	UV	basic	device		U	3	–	–
S3.4	16	5 (1.0.4) (mod)	Emulated	Classic/Standard	∅				U	3	–	–
S3.5	11	5 (1.0.4) (mod)	Real (non-rooted)	Classic/Standard	UV	basic	device	strong	U	3	–	–
S3.6	14	5 (1.0.4) (mod)	Emulated	Classic/Standard	∅				U	3	–	–
T4: Are repeated requests with the same *nonce* blocked by the integration?												
S4.1	13	11 (1.1.1)	Real (non-rooted)	Classic/Standard	PR	basic	device		L	6	–	100%
S4.2	10	11 (1.1.1)	Real (rooted)	Classic/Standard	UV				U	6	–	100%
S4.3	15	11 (1.1.1)	Real (non-rooted)	Classic/Standard	PR	basic	device		L	6	–	100%
S4.4	16	11 (1.1.1)	Emulated	Classic/Standard	∅				U	6	–	100%
S4.5	11	11 (1.1.1)	Real (non-rooted)	Classic/Standard	PR	basic	device	strong	L	6	–	100%
S4.6	14	11 (1.1.1)	Emulated	Classic/Standard	∅				U	6	–	100%

Note: **Prec. (API)** measures accuracy for natively covered verdicts (app/device/strong/license). **Prec. (Backend)** measures accuracy for additional controls (e.g., non-repudiation and *nonce* replay in T4). "–" indicates not applicable / not measured. n indicates number of requisitions.

Table 7. Summary of results of the scenarios

Scenario	Objective	Layer/target	Findings
C1	Device integrity (root)	Dev	Intact: MDI; rooted: E{} (signal suppression); detects root
C2	APK tampering (repackaging)	APK	UV + divergent certDigest; MDI maintained; prevents modified APKs
C3	Dynamic instrumentation (Frida)	Dev	MDI/PR maintained; bypassable locally; additional hardening needed
C4	Replay/nonce reuse	Srv, Net	API accepts token; backend blocks (HTTP 400); server-side non-repudiation

Abbreviations: PR = *PLAY_RECOGNIZED*; **UV** = *UNRECOGNIZED_VERSION*; **MDI** = *MEETS_DEVICE_INTEGRITY*; **E{}** = empty object (API omits signals in compromised environment).

Our results indicate that Play Integrity should function as a security signal oracle rather than a final control mechanism. Effective implementation requires server-side validation with unique per-action nonces, short-lived tokens, and request-action binding, complemented by client-side hardening through obfuscation, anti-hooking defenses, and root detection. Furthermore, integrating integrity signals with complementary telemetry sources, such as usage patterns, reputation data, and rate limiting information, significantly improves detection accuracy while reducing false negative rates. A summarization of the findings is presented at table 7.

This study is limited by its controlled lab scope, narrow device/Android coverage, use of basic instrumentation on unhardened apps, and an anti-replay analysis that omitted

stronger cryptographic bindings. External validity may be affected by Play Services updates and OEM/TEE differences, and we did not measure latency, operational cost, or scalability for high-demand settings. Developers should treat Play Integrity as one signal in a broader hardening stack: use Standard for recurring checks and Classic for high-value actions, while enforcing backend controls such as per-action nonces, rate limiting, and anomaly detection. When policies depend on DEVICE/STRONG verdicts, account for device/ROM fragmentation with fallback or step-up flows to avoid locking out legitimate users. Researchers can extend this work by testing a wider mix of devices, Android versions, and tooling, and by refining FP/FN metrics under realistic attack workloads to strengthen external validity.

6. Conclusion

Our evaluation demonstrates that the Play Integrity API effectively addresses conventional attack vectors such as device compromise and application tampering; however, its security guaranties require integration within a comprehensive defense-in-depth strategy. Essential implementation practices include server-side exclusive validation, per-action unique nonces, secure channel binding, code obfuscation, and anti-hooking protections augmented by security telemetry.

Future work will expand testing to more devices, Android versions, and hardware-backed protections, including the strong Integrity tier and TEE. Despite its advances, the Play Integrity API remains only as effective as the rigor of its implementation and the additional controls used to compensate for its inherent limitations.

Agradecimentos. Esta pesquisa recebeu apoio parcial da CAPES[5], sob o Código de Financiamento 001.

References

Ibrahim, M., Imran, A., and Bianchi, A. (2021). Safetynot: on the usage of the safetynet attestation api in android. In *Proceedings of the 19th ACM MobiSys*.

Kim, S., Jee, K., Park, J., and Shin, J. (2021). SafetyNOT: On the usage pitfalls of android SafetyNet API. *IEEE TDSC*, 20(1).

Niemi, A., Nayani, V., Moustafa, M., and Ekberg, J.-E. (2023). Platform attestation in consumer devices. *ResearchGate*.

Ruggia, A., Nisi, D., Dambra, S., Merlo, A., Balzarotti, D., and Aonzo, S. (2024). Unmasking the veiled: A comprehensive analysis of android evasive malware. In *Proceedings of the 19th ACM*.

Samper, J. and Ferreira, B. (2024). Leveraging remote attestation apis for secure image sharing in messaging apps. *IACR Cryptology ePrint Archive*.

Steinböck, M., Troost, J., van Beijnum, W., Seredynski, J., Bos, H., Lindorfer, M., and Continella, A. (2025). Sok: Hardening techniques in the mobile ecosystem — are we there yet? In *Proceedings of the IEEE EuroS&P*.

Zhang, Z., Zhang, Y., and Lin, Z. (2023). On the (in)security of manufacturer-provided remote attestation frameworks in android. In *LNCS*, volume 14274. Springer.

[5]`https://www.gov.br/capes/pt-br`