DOI: 10.21088/jcfs.3107.6874.4126.2

Journal of Clinical Forensic Sciences
Volume 4 Number 1, January - June 2026

ORIGINAL ARTICLE

Dynamic Code Updates: A Hidden Threat in App Store Security

Vinay Chauhan [1], Neeraj Kumar[2], Shivam Aggarwal[3], Ayush Gupta[4], Seepe Sharma[5], Dev Brat Mishra[6]

HOW TO CITE THIS ARTICLE:

Vinay Chauhan, Neeraj Kumar, Shivam Aggarwal et. al, Dynamic Code Updates: A Hidden Threat in App Store Security. Jr of Clinical Forensic Sci. 2026; 4(1): 13–18

ABSTRACT

The exponential growth of mobile applications has dramatically reshaped the global software ecosystem, with Android and iOS app stores acting as the primary gateways for billions of users worldwide. These platforms rely on extensive vetting mechanisms, such as Google's *Bouncer* and Apple's *App Review Process*, to detect and block malicious content before publication. However, a critical and increasingly exploited weakness has emerged the abuse of dynamic code updates and in-app updating mechanisms to deliver malicious payloads *after* an app's approval. In this approach, attackers publish benign applications that pass all automated and manual inspections, but later inject harmful code through dynamic class loading (DCL), reflection, or hidden update APIs. Once activated, these components can steal sensitive data, escalate privileges, or enable remote control, all while appearing legitimate to users. Empirical research, including *StaDynA*[2], *StaDART*[1], and large-scale studies like *Poeplau et al.*[4] and *DyDroid*[5], has revealed the alarming prevalence of such mechanisms. Findings indicate that more than 80% of analyzed malware samples use reflection, while nearly 20% employ dynamic class loading to bypass static detection. These techniques exploit the limitations of traditional app store security, which assumes code immutability after publication. Furthermore, manufacturer-level Android customizations exacerbate the risk: preinstalled, over privileged vendor apps and delayed security patch rollouts

AUTHOR'S AFFILIATION:

[1] Security Researcher, M.C.A, Chandigarh University, Ludhiana, Punjab, India.

[2] Senior Scientific Assistant (Crime Scene Management Division) Forensic Science Laboratory, NCT of Delhi, India.

[3] Student, National Forensic Sciences University (An Institution of National Importance, Ministry of Home Affairs, Government of India).

[4] Security Researcher, MCA, Institute of Professional Excellence and Management (IPEM), Ghaziabad, Uttar Pradesh, India.

[5] Security Researcher, Deputy Vice president Infosec, Noida, India.

[6] Assistant Professor, Fish Biology Research Lab, Department of Zoology, Tilak Dhari Collage, Jaunpur, Uttar Pradesh, India.

CORRESPONDING AUTHOR:

Neeraj Kumar, Senior Scientific Assistant (Crime Scene Management Division) Forensic Science Laboratory, NCT of Delhi, India.

E-mail: nk00267@gmail.com

➢ **Received:** 23-12-2025 ➢ **Accepted:** 29-01-2026

provide fertile ground for post-installation attacks.[3] The combination of dynamic updates, runtime reflection, and vendor weaknesses thus creates a multi-layered threat that undermines user trust and platform integrity. This paper consolidates findings from leading research and demonstrates how current defense mechanisms fall short against dynamic update threats. It advocates for a hybrid static–dynamic analysis framework, mandatory update transparency policies, and continuous post-installation monitoring as essential countermeasures. The objective is to inform developers, regulators, and end-users about the real-world dangers of post-publication malware evolution and emphasize the urgency of establishing stricter runtime and policy-level controls to safeguard the mobile ecosystem.

KEYWORDS

• Android Security • Dynamic Code Loading • Reflection • In-App Updates • Malware Evasion • App Store Vetting

INTRODUCTION

Mobile applications have become indispensable to modern life, serving as the foundation for communication, banking, entertainment, and enterprise operations. The Android and iOS ecosystems dominate the global mobile landscape, collectively distributing millions of apps through centralized stores such as Google Play and Apple App Store. According to recent industry statistics, over 120 billion app downloads occur annually, making mobile applications one of the most widely consumed digital commodities. While this ecosystem enables innovation and accessibility, it also presents an attractive and highly scalable target for cybercriminals seeking to exploit user trust. The prevailing assumption behind app store security models is that once an application passes review and is published, its behavior remains consistent throughout its lifecycle. However, this assumption is increasingly invalid. Modern app development frameworks especially Android's support dynamic code updates, reflection, and in-app delivery mechanisms that allow executable code to be fetched or modified after installation. Although originally designed for legitimate purposes such as modular feature delivery, faster bug fixes, and A/B testing, these mechanisms have been weaponized by adversaries to distribute post-publication malware. Attackers exploit these features to bypass static code analysis by embedding only benign logic in the initial submission, while malicious payloads are delivered later through dynamic updates. *Zhauniarovich et al.* in *StaDynA*[2] and Ahmad et al. in *StaDART*[1] empirically demonstrated that static analysis detects fewer than 30% of dynamic update-based threats, while hybrid analysis that monitors runtime execution improves accuracy to above 80%. Similarly, *Poeplau et al.*[4] showed that a significant portion of both benign and malicious apps incorporate dynamic loading capabilities, with 19.9% of malware samples exploiting DCL for evasion. Reflection, detected in over 81% of analyzed malware, is frequently used to obscure method calls and execution paths, making the code nearly impossible to trace during static analysis. Compounding the threat, vendor-specific Android customizations introduce additional vulnerabilities. *Wu et al.*[3] reported that 85.78% of preloaded vendor apps are over-privileged, while up to 85% of vulnerabilities originate from OEM modifications. Such design flaws offer attackers new attack surfaces when combined with dynamically injected payloads.

This paper seeks to raise awareness of the dynamic update threat model, explain its technical underpinnings, and consolidate the fragmented research landscape. It highlights how dynamic updates undermine the static trust model of app stores, provides a comparative analysis of detection frameworks, and proposes actionable recommendations for mitigating post-deployment code manipulation in mobile ecosystems.

BACKGROUND AND RELATED WORK

The Android ecosystem's openness and flexibility have enabled a vibrant developer community but also introduced complex security challenges. Android applications are primarily developed in Java or Kotlin and execute atop the Dalvik or ART virtual

machines, which support Dynamic Class Loading (DCL) and reflection two mechanisms that allow applications to modify behavior at runtime. While these features were initially designed to enhance modularity and reduce update overhead, they have become key enablers of sophisticated malware.

1. Dynamic Class Loading (DCL)

Dynamic Class Loading allows an application to load and instantiate classes that were not present in the original application package (APK). Using APIs such as DexClassLoader, PathClassLoader, or loadClass(), developers can fetch external .dex or .jar files from remote servers or local storage and execute them dynamically. This technique is widely used in legitimate applications to enable plugin architectures, third-party library updates, and feature modularization. However, cybercriminals exploit DCL to deliver malicious payloads post-deployment, allowing their apps to remain undetected during the app store's static vetting process.

Studies such as *StaDynA*[2] and *Poeplau et al.*[4] have shown that over 19% of malware samples use DCL to evade detection. The dynamic loading process typically involves obfuscating the URL or encryption key that fetches the code, decrypting it at runtime, and invoking malicious methods using reflection. Because these payloads are not part of the initial APK, static analyzers cannot evaluate their contents.

2. Reflection and Code Obfuscation

Reflection is a Java feature that enables programs to inspect and modify their own structure at runtime. While essential for certain frameworks and serialization libraries, it is one of the most exploited techniques for obfuscation in Android malware. By dynamically constructing method names or class identifiers at runtime, attackers obscure control flow and make static call graph construction nearly impossible. Reflection also enables invoking private or hidden APIs, expanding an attacker's capability to bypass Android's security boundaries.

Research such as DroidRA, Ripple, and EspyDroid+[7] has attempted to resolve reflection targets using static string analysis, partial evaluation, or dynamic tracing. However, these approaches face limitations when dealing with encrypted or network-generated strings. Attackers often use polymorphic code generation and remote command-and-control servers to build reflective calls dynamically, making detection and analysis highly challenging.

3. Dynamic Updates and In-App Delivery

The growing use of in-app updates and dynamic delivery systems, such as Google's "Play Core Library",[10] further complicates security enforcement. These mechanisms allow developers to push modular updates (e.g., new features or bug fixes) without re-submitting the entire application. Although convenient, they open a door for adversaries to deliver unverified or manipulated modules under the guise of legitimate updates. Attackers can exploit this framework to inject malicious .dex files directly into production apps, bypassing store verification.

4. Previous Research and Measurement Studies

The *StaDART* framework[1] introduced a hybrid analysis system that combined static and runtime instrumentation to detect dynamically loaded malicious code. Their evaluation showed that while static tools detected less than 30% of dynamically updating threats, *StaDART* achieved over 80% accuracy. Similarly, *StaDynA*[2] analyzed 29,000 Android applications and found that 81% of malware samples used reflection to hide malicious logic.

Poeplau et al.[4] provided empirical evidence that over 1 in 5 Android apps employ some form of dynamic code loading, with many abusing the feature to evade Google Bouncer. Follow-up studies such as *DyDroid*[5] and *Grab'n Run*[6] validated these findings and proposed mitigations like signed module enforcement and secure loading APIs.

Finally, research by *Wu et al.*[3] on vendor customizations highlighted how manufacturer modifications often exacerbate the risks posed by dynamic updates. Their work revealed that 85.78% of vendor pre-installed apps were over-privileged, and nearly 85% of firmware vulnerabilities stemmed from OEM alterations, increasing the probability of privilege escalation when combined with dynamic payload injection.

5. Summary of Observations

- From the literature, several key trends emerge:

- Dynamic loading and reflection are increasingly common in both benign and malicious apps.
- Existing detection mechanisms whether static or dynamic are inadequate for runtime behavior analysis.
- Hybrid methods (StaDART-like) show promise but face scalability challenges.
- Vendor customizations and delayed security patching amplify exposure to dynamically injected code. These findings collectively underscore the urgent need for more transparent, runtime-aware, and vendor-aligned defense mechanisms.

THREAT MODEL AND ATTACK VECTORS

The threat model examined in this study considers adversaries who exploit Android's flexibility in code execution and the permissiveness of app store policies to deliver malicious functionality after publication. The adversary's core advantage lies in the separation between app vetting time and runtime execution. Since app store verification is largely static, code that is downloaded or generated after approval remains unchecked.

1. Adversary Capabilities and Objectives

- An attacker is assumed to have the ability to:
- Develop and publish applications through legitimate developer accounts on official app stores.
- Embed benign code and conceal the malicious component's delivery mechanism using encryption, obfuscation, or hidden update APIs.
- Remotely host or control servers that distribute malicious payloads post-installation.
- The objectives typically include:
- Data Exfiltration: Harvesting sensitive user data such as credentials, SMS, GPS, and contact lists.
- Privilege Escalation: Leveraging vendor or system-level APIs to gain elevated privileges.
- Command and Control (C2): Enabling remote instructions to alter the app's behavior dynamically.
- Monetization and Fraud: Triggering unauthorized in-app purchases, ad fraud, or clickjacking campaigns.

2. Attack Lifecycle

- **Stage 1 – Publication of a Benign App:** The attacker develops an apparently harmless app (e.g., utility or game) that passes static and automated security checks.
- **Stage 2 – Dormant Phase:** The application operates normally until it receives specific triggers such as a remote command, device condition, or time delay.
- **Stage 3 – Dynamic Update or Code Fetch:** Using DCL APIs (DexClassLoader, PathClassLoader) or in-app update mechanisms, the app downloads malicious code from a remote server. Payloads are often encrypted or obfuscated to evade network and antivirus scans.
- **Stage 4 – Execution of Malicious Logic:** The loaded module is executed using reflection, enabling invocation of hidden methods or exploitation of private APIs.
- **Stage 5 – Persistence and Concealment:** Once deployed, the app may self-delete downloaded payloads or modify logs to erase traces. This cycle can repeat indefinitely, effectively creating a post-publication polymorphic malware.

3. Reflection-Based Evasion

Reflection enables dynamic invocation of methods whose names are constructed at runtime. Malware often encrypts class and method identifiers, decrypting them only after successful execution triggers. This prevents static analysis from identifying dangerous API calls. Additionally, attackers may check for emulator or sandbox environments and execute benign behavior to evade automated detection.

4. Vendor Customization Exploitation

Vendor-modified Android distributions frequently expose privileged services. Dynamically loaded code can abuse these to perform restricted operations, such as changing system settings, bypassing permission prompts, or accessing hardware-level resources. In extreme cases, these modifications provide backdoors for privilege escalation that persist across OS versions.[3]

5. Example Scenarios

- **Malicious Update Injection:** A weather app releases a "data enhancement" in-app update that downloads and executes a hidden cryptocurrency miner.
- **Conditional Activation**: A utility app fetches a secondary .dex file only when a certain geographic location or network condition is met.
- **API Misuse:** A customized vendor API originally designed for system diagnostics is invoked by reflective calls from dynamically loaded code to obtain kernel-level data.

6. Threat Scope

The scale of this threat is substantial. Analysis from *DyDroid*[5] and *StaDynA*[2] suggests that approximately 20% of sampled malware employ DCL-based dynamic updates. In controlled tests by *StaDART,*[1] over 33% of malware samples introduced new dangerous permissions *after* installation permissions absent during initial store submission. This demonstrates that dynamic updates enable not just behavioral changes but also security policy circumvention.

7. Summary

Dynamic updates and reflection empower adversaries to create shape-shifting applications that evolve over time. The combination of remote code loading, runtime obfuscation, and vendor weaknesses makes this an enduring and scalable attack model. The following sections provide empirical comparison tables and propose multi-layered detection strategies to address this increasingly sophisticated threat vector.

METHODOLOGY

This research synthesizes findings from multiple empirical studies *StaDynA,*[2] *StaDART*[1], *Poeplau et al.*[4], *DyDroid,*[5] and *Wu et al.*[3] covering more than 29,000 Android apps and multiple vendor firmware images. The methodology focuses on:

1. Measuring DCL and reflection usage in benign vs. malicious apps.
2. Comparing static, dynamic, and hybrid analysis results.
3. Assessing vendor firmware vulnerabilities and overprivilege patterns.

COMPARATIVE ANALYSIS AND RESULTS

1. Dynamic Feature Usage

Table 1: Usage of Dynamic Code Loading and Reflection in Android Apps

Category of Apps	Dynamic Class Loading Usage	Reflection Usage	Notes
Benign Apps	~18.5%	~88%	Reflection used for legitimate modularity.
Malware Samples	~19.9%	~81%	Used to hide or delay execution of payloads.
Both Used (DCL + Reflection)	~12%	–	Most evasive category for static analysis tools.

(*Sources: Poeplau et al.*[4]; *DyDroid*[5])

2. Vendor Customization Impact

Table 2: Vendor Customization Security Impact

Security Metric	Average Impact	Description
Over-privileged Preloaded Apps	85.78%	Apps request excessive permissions.[3]
Vulnerabilities from Vendor Modifications	Up to 85%	OEM code introduces exploitable flaws.[3]
Delayed Patch Rollout	60–90%	Security patches often delayed.[3]
Private API Exposure	Common	Enables privilege escalation by loaded modules.[3]

3. Detection Effectiveness: Static vs Hybrid

Table 3: Detection Effectiveness: Static vs Hybrid Analysis.

Analysis Technique	Detection Rate on Dynamic Threats	Limitation
Static Analysis	<30%	Fails to detect runtime-fetched code.[4]
Dynamic Sandbox	45–55%	Payloads may not trigger during tests.[5]
Hybrid (StaDART-like)	>80%	High accuracy but computationally costly.[1]

DISCUSSION

Dynamic updates challenge the assumption of immutable software between release and execution. The combination of reflection and runtime code loading effectively divides malicious behavior into stages: a benign carrier app and a hidden payload.[4,5] Sandboxing systems can miss payloads due to environment checks, while string and reflection analyses remain limited when faced with dynamically constructed identifiers.[7] Vendor-level flaws amplify these risks: preinstalled apps with broad privileges or exposed APIs can be exploited by dynamically injected modules[3]

The persistence of such vulnerabilities underscores the need for ecosystem-wide cooperation and hybrid detection mechanisms.

RECOMMENDATIONS

1. Hybrid Static–Dynamic Scanning: App stores should adopt hybrid frameworks such as *StaDART*[1] to analyze runtime behaviors.
2. Update Transparency: Require explicit disclosure of dynamic code-loading capabilities.
3. Runtime Permission Re-validation: Force reauthorization when dangerous permissions are added post-installation.
4. Signed Module Enforcement: Mandate cryptographic signing of dynamically loaded modules.[6]
5. OEM Hardening: Standardize vendor security practices to minimize overprivilege.[3]
6. Continuous Monitoring: Maintain registries of dynamically updating apps and run post-release analyses.

LIMITATIONS AND FUTURE WORK

This study reanalyzes existing data rather than introducing new empirical datasets. Future work should measure the prevalence of post- update behavior changes in live ecosystems and develop scalable hybrid analysis frameworks suitable for app store deployment.[1,5,7]

CONCLUSION

Dynamic code updates and reflection create a potent attack surface enabling apps to change behavior post-approval. Studies such as *StaDynA*,[2] *StaDART*,[1] and *Poeplau et al.*[4] confirm that traditional analysis fails against runtime-loaded code. Vendor customizations[3] further amplify risks. The next generation of defenses must integrate hybrid detection, transparent update policies, and runtime verification to protect users from post-publication code injection.

REFERENCES

1. M. Ahmad, V. Costamagna, B. Crispo, F. Bergadano, Y. Zhauniarovich, "StaDART: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications," Journal of Systems and Software, vol. 159, 2020.
2. Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, F. Massacci, "StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications," Proc. ACM CODASPY, 2015.
3. L. Wu, M. Grace, Y. Zhou, C. Wu, X. Jiang, "The Impact of Vendor Customizations on Android Security," Proc. ACM CCS, 2013.
4. S. Poeplau, S. Fratantonio, A. Bianchi, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," Proc. NDSS, 2014.
5. Z. Qu et al., "DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications," ACM Digital Library, 2018.
6. L. Falsina et al., "Grab'n Run: Secure and Practical Dynamic Code Loading for Android," Proc. ACSAC, 2015.
7. R. Chatterjee et al., "DroidRA, Ripple, and EspyDroid+: Reflection Analysis and Mitigation in Android," IEEE Trans. Software Eng., 2021.
8. A. Aysan et al., "Analysis of Dynamic Code Updating in Android from a Security Perspective," IET Information Security, 2019.
9. W. Elsersy et al., "The Rise of Obfuscated Android Malware and Impacts on Detection," PeerJ Comput. Sci., 2022.
10. Google Android Developers, "Dynamic Delivery and In-App Updates," 2024.