An Empirical Comparison of Security and Privacy Characteristics of Android Messaging Apps

Ioannis Karyotakis*
AUEB & NTUA
Athens, Greece
karyotakisg@aueb.gr

Foivos Timotheos Proestakis*
AUEB & NTUA
Athens, Greece
proestakis@aueb.gr

Evangelos Talos*
AUEB & NTUA
Athens, Greece
vtalos@aueb.gr

Diomidis Spinellis
AUEB & TU Delft
Greece & Netherlands
dds@aueb.gr

Nikolaos Alexopoulos
AUEB
Athens, Greece
alexopoulos@aueb.gr

Abstract

Mobile messaging apps are central to user privacy, yet their practical implementations remain understudied. Using a hybrid static–dynamic methodology, we compare the Android clients of Meta Messenger, Signal, and Telegram. Our results show clear differences: Signal has the smallest attack surface, Messenger exhibits extensive background activity, and Telegram requests the most high-risk permissions. All three apps comply with the Android permission model.

CCS Concepts

• **Security and privacy** → *Social network security and privacy.*

Keywords

Android, Mobile Privacy, Static Analysis, Network Analysis

ACM Reference Format:
Ioannis Karyotakis, Foivos Timotheos Proestakis, Evangelos Talos, Diomidis Spinellis, and Nikolaos Alexopoulos. 2026. An Empirical Comparison of Security and Privacy Characteristics of Android Messaging Apps. In *The 41st ACM/SIGAPP Symposium on Applied Computing (SAC '26), March 23–27, 2026, Thessaloniki, Greece.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3748522.3779858

1 Introduction

Mobile messaging applications are core communication infrastructure for billions. Although their cryptographic protocols (e.g., the Signal Protocol) are formally verified [2], a holistic comparison of their real-world implementations—including architecture, permissions, and attack surface—is still lacking.

We address this gap with a hybrid methodology covering (i) *static characteristics* (permissions, exported components, code complexity) and (ii) *dynamic behavior* (network activity and permission-dependent resource access). We evaluate Signal, Telegram, and Meta Messenger, three widely deployed apps representing distinct design and development models.

To bypass strict certificate pinning, we employ kernel-level tracing,[1] enabling us to attribute encrypted traffic and resource usage directly to processes—exposing security and privacy (S&P) behaviors invisible to conventional proxy-based tools.
Availability. Code and data are publicly available.[2]

2 Methods

Our methodology follows differential testing [4], applied consistently across all applications. For static analysis, we extracted production-signed APKs from physical devices using ADB and inspected them with standard Android tooling. Using `aapt` and `apkanalyzer`, we obtained manifest metadata (permissions, exported components, package structure). Androguard[3] was used to compute Dalvik bytecode metrics (classes, methods, fields). MobSF[4] provided automated security assessments, including insecure configurations, risky API usage, and other S&P-relevant issues.

Dynamic analysis focused on network activity and access to restricted resources. Network traffic was captured using `tcpdump`[5], while SliceDroid [1] attributed kernel-level events (via ftrace [5]) to specific apps, including operations routed through Android Framework daemons. To enhance attribution of encrypted traffic, we instrumented SliceDroid with kprobes on socket functions (`tcp_sendmsg`, `udp_sendmsg`, and receive counterparts) and matched kernel events to packet captures using protocol, port, event order, and packet length. Geolocation of remote peers was enriched using a public service.[6]

Each app was evaluated under four reproducible scenarios: foreground/background with full permissions and foreground/background with all dangerous permissions revoked. Foreground tests involved composing and sending a single message (~10 seconds); background tests kept the device idle for 5 minutes. All scenarios were repeated ten times per app. Statistical comparisons used the Kruskal–Wallis H test, followed by Mann–Whitney U tests with Bonferroni correction where applicable.

*These authors contributed equally to this work.

[1]https://github.com/SliceDroidTeam/SliceDroid
[2]https://zenodo.org/records/17306016
[3]https://github.com/androguard/androguard
[4]https://github.com/MobSF/Mobile-Security-Framework-MobSF
[5]https://www.tcpdump.org/
[6]https://ip-api.com/

3 Experiments

We evaluate the three applications using the static and dynamic methods introduced in Section 2. All experiments were conducted on the latest stable releases from the Google Play Store: Messenger (523.0.0.53.109), Signal (7.54.1), and Telegram (12.0.1).

3.1 Static analysis

3.1.1 Application Packaging and Complexity. All clients target Android API 34+, though minimum SDKs vary (Signal: 21, Telegram: 23, Messenger: 28). Signal produces the largest base APK by embedding translations, whereas Telegram and Messenger optimize size via feature-specific splits. However, Messenger exhibits the highest code complexity (11 DEX files, ≈500k methods)—double Telegram's volume. Native architecture also differs: Telegram uses a single monolithic library, while Signal modularizes native functionality.

3.1.2 Attack surface metrics. Regarding attack surface [3], Messenger exposes the highest number of entry points (25 services, 15 receivers, 7 providers). Signal is the most restrictive (7 services, 0 providers), while Telegram falls in between. Exported UI Activities are comparable across all apps.

3.1.3 Permissions. Messenger requests the largest permission set (87), while Signal remains leaner (72 total, 19 dangerous). Telegram requests the fewest permissions overall but the most *dangerous* ones (25). Unlike Signal, both Telegram and Messenger seek broad system-level privileges (e.g., phone-call control, overlay windows) that expand their attack surface beyond core messaging needs.

3.1.4 Security warnings and risk. While MobSF rates all apps as "Medium" risk, Messenger triggers substantially more warnings (118) than Telegram (59) or Signal (55). Messenger's volume stems from third-party SDKs and configuration issues, though its TLS alerts are false positives superseded by native certificate pinning (`Fizz`). Notably, Telegram is the only client permitting cleartext traffic.

Table 1: Combined static-analysis metrics across clients.

Metric	Signal	Telegram	Messenger
Exported Components			
Activities	19	14	13
Services	7	15	25
Receivers	4	3	15
Permissions			
Dangerous	19	25	24
Total	72	71	87
Static Analysis Warnings			
Total findings	55	59	118

3.2 Dynamic analysis

We conduct dynamic analysis on a Nothing Phone 2a running Android 14, following the methodology of Section 2.

3.2.1 Overall Network Activity. We report median values across ten runs for byte traffic. We observed the following notable discrepancies:

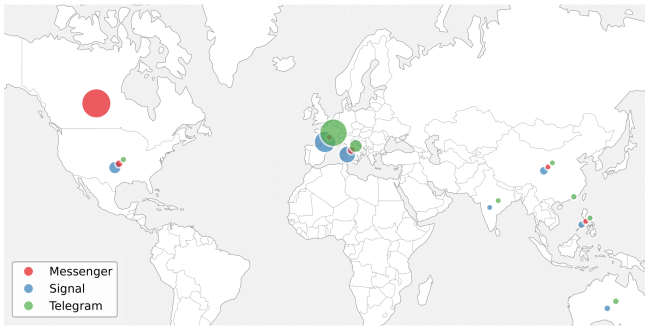

Figure 1: Remote peer distribution by traffic volume.

Overall activity. Messenger generated the highest foreground traffic (≈64 kB sent, 18 kB received), dwarfing Signal, which was the most frugal (≈3.5 kB sent, 1 kB received). Telegram fell in between. Notably, in background scenarios, only Messenger transmitted data (≈1.5 kB); Signal and Telegram remained silent.

Transport-layer protocols. Messenger relied significantly on UDP, especially in foreground scenarios with more than 80% of its traffic being UDP. Signal and Telegram did not generate any UDP traffic in most runs (median=0). High UDP traffic generated by Messenger can be attributed to the adoption of the QUIC protocol.

Effect of permissions. Paradoxically, Telegram generated substantially higher traffic *without* permissions (≈223 kB total) than with full permissions (≈15 kB). This likely stems from repetitive error-handling loops triggered by permission denials.

3.2.2 Remote Peers. Reflecting our European vantage point, Telegram and Signal concentrated connections within Europe, though Signal maintained a broader distribution with nodes in the US and Asia. In contrast, Messenger routed the vast majority of traffic to North America (specifically Canada), exhibiting the least geographically distributed infrastructure of the three. The map of Figure 1 visualizes these geographic patterns, highlighting Messenger's North America–centric footprint and the comparatively broader distribution of Signal.

3.2.3 Permission Compliance. Using SliceDroid, we verified that all applications correctly adhered to imposed restrictions (camera, microphone, contacts) in *no permissions* scenarios, with no unexpected resource access detected.

Acknowledgments: This project has received funding from the EU under the MSCA grant agreement No. 101108713.

References

[1] Nikolaos Alexopoulos, Simon Althaus, and Diomidis Spinellis. 2025. SliceDroid: Towards Reconstructing Android Application I/O Behaviors from Kernel Traces. Version 0.1. (2025). doi:10.5281/zenodo.15784277.

[2] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. 2017. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 451–466. doi:10.1109/EUROSP.2017.27.

[3] Pratyusa K. Manadhata and Jeannette M. Wing. 2011. An attack surface metric. *IEEE Trans. Software Eng.*, 37, 3, 371–386. doi:10.1109/TSE.2010.60.

[4] William M. McKeeman. 1998. Differential testing for software. *Digit. Tech. J.*, 10, 1, 100–107. https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf.

[5] Steven Rostedt. 2009. Finding origins of latencies using ftrace. *Proc. RT Linux WS*.