Operational Analysis of Communication Channels in Mobile RCS

By Ireneusz Tarnowski

Published: 2025-12-23 · Archived: 2026-04-05 21:08:10 UTC

12 min read

Dec 22, 2025

Press enter or click to view image in full size

The analysis examines a sample of mobile malware used in highly targeted operations where the priority is not infection scale, but persistent, long-term control over a single victim device. The code combines characteristics of operator-grade spyware — extended telemetry collection, strong environmental awareness, and cloud-based C2 communications — with a real-time access component implemented via a Fast Reverse Proxy tunnel. The report demonstrates how a seemingly benign application becomes a carrier for a durable command channel leveraging Google infrastructure and reverse tunnels, while preserving full niche deployment and OPSEC parameters.

Aplication

The initial infection vector makes the operation resemble the numerous ongoing mass-scale campaigns. Through social engineering, the adversary persuades a selected victim to download and instal a mobile application that

impersonates another trusted application. The downloaded APK functions as a classic dropper. A dropper is a form of malicious application whose purpose is to deliver and execute an embedded payload — typically by unpacking or decoding it once on the device. In this case, its goal is to obtain all required user permissions (by presenting screens borrowed from the spoofed trusted application), after which it instals and launches a second application (stage two) retrieved from \assets\apk\target.apk.

The installed application is, in reality, yet another dropper. Static analysis of this stage does not immediately reveal hostile capabilities, and nothing obviously malicious can be observed at first glance. At this point, the malware retrieves from its internal resources a file with a .json extension (its entropy is 7.99, indicating strong packing or encryption; in practice, it is a container holding an encrypted DEX library later injected into the active application). The file is encrypted with the symmetric stream cipher RC4 (KSA/PRGA). The decryption key is trivial to locate in the disassembled code. After decryption, functions previously hidden within that library are executed, including:

- real-time screen capture and streaming,
- remote device management (VNC/RDP-style interaction through Accessibility; real-time operator control).

Initially, the sample appeared to be a RAT. However, a full inspection of its functionality and operator workflow indicates that it is an advanced spyware platform — an RCS (Remote Control System).

During analysis, a multi-layer command-and-control architecture was identified, designed to ensure persistent, covert, and operator-driven access to the compromised device. The communication model separates signalling and task orchestration from interactive remote access, reducing observable network traffic, and bypassing traditional beacon-based detection mechanisms.

The remainder of the report focuses on this non-obvious communication strategy. It describes the behaviour of the C2 component, the mechanism used to initiate connections to the controlling server, and observable artefacts indicating preparation for long-term persistence on the device.

Press enter or click to view image in full size

Figure 1. Application configuration parameters.

Command and Control

The central component of the infrastructure is an HTTPS-accessible Command-and-Control (C2) server that functions as a device registry, configuration repository, and task-coordination point. The C2 is operated directly by a human operator who manually initiates actions and manages access sessions for individual devices.

Communication between the operator and the malicious application is carried out using standard web interfaces or APIs, making malicious traffic difficult to distinguish from ordinary HTTPS activity at the network level.

On the infected device, the **BackendApi** class manages all communications with the operator's C2 server. This module performs, among other things:

- device registration within the adversary's panel,
- heartbeat signaling,
- command retrieval via polling,
- transmission of single or batched events,
- uploading the installed-applications list,
- retrieval of system information,
- transmission of identifiers and complete telemetry.

The implementation clearly includes a stable device identifier (**generateStableDeviceId**) based on **android_id**. This method generates a persistent tracking ID that is used throughout the life of the application. At initialization, the malware collects environmental data, including:

- battery level and charging state,
- type of network connection,
- currently foreground applications,
- full list of installed applications,
- SIM card information (card presence, operator, phone number, where possible),
- device memory information,
- screen state,
- external IP address.

This represents device profiling prior to further exploitation and deployment of remote management (RMM). This precise victim profile is characteristic of narrow and highly selective attacks in which the adversary tailors subsequent infection stages based on the victim's identity. This approach avoids accidental infections and increases the likelihood that it remains undetected.

If the attack proceeds, the adversary supplies a Fast Reverse Proxy (FRP) tunnel configuration from the C2, allowing remote access to the device. The FRP parameters (server address, port, token) are delivered dynamically at runtime and are not hard-coded in the application. The polling and heartbeat mechanisms allow the operator to regularly issue commands and collect telemetry — including the application list, battery status, network type, memory information, SIM details and the currently active application.

The FRP tunnel, meanwhile, is only started after the malicious backend delivers its configuration. Once established, it enables interactive operator access to the device, while the FRP server itself resides within the attacker-controlled infrastructure, separate from both the device and the HTTPS communication channel.

FCM — Firebase Cloud Messaging

The signalling channel is implemented using **Firebase Cloud Messaging (FCM)**. FCM is a Google service designed for the asynchronous delivery of push messages to mobile applications and web clients. In legitimate scenarios, it is used to send notifications, synchronize data, or trigger application actions without maintaining a continuous network connection. From a technical perspective, FCM operates as a message broker: after installation, an application registers itself within Google's infrastructure and receives a unique token or subscribes to a predefined topic. The application server sends messages to Google and sends them to the intended devices — even when the application is inactive or the handset is in sleep mode. A critical property of FCM is that network traffic flows exclusively between the device and Google infrastructure, meaning that from the perspective of networks and security controls, the communication appears to be legitimate, system-level Android traffic.

For the FCM functionality, the configuration class (Fig. 1) contains the key parameter:

private static final String FCM_TOPIC = "device_commands";

This indicates that the application subscribes to a global FCM topic rather than an individual token. Every device on which the application is installed joins the same logical communication channel. In practice, this allows the C2 operator to send a single instruction that Google will distribute to any number of infected devices, without connecting to them directly.

In this architecture, FCM is not used to transmit operational data, but to provide asynchronous control signalling. The C2 operator sends a push to the device through the Google infrastructure, which is received in onMessageReceived(). The JSON payload is converted into an internal command identifier, followed by JSON → executeXYZ() mapping — directly translating the signalling message into calls targeting specific functional modules of the malware. Each invocation not only launches an action but also activates or reconfigures persistence and supervision subsystems: FRPC initialization, client re-registration, overlay configuration, WebSocket channel activation, tunnel authentication, screen-stream setup, heartbeat, and logging. Consequently, FCM functions as a "wake+task" signalling channel, where the push does not carry the operational command itself, but instructs the device to retrieve it from the Backend API. This approach reduces power consumption, minimizes network traces, eliminates cyclic DNS/HTTP polling, and significantly increases stealth.

With this model, the C2 can issue categories of FCM-driven commands that together provide full functional coverage of the device. Communication-layer commands (e.g., executeEnableWebSocket() or executeDisableWebSocket()) enable and disable the WebSocket channel and FRPC tunnel, thus controlling remote-access capability. Connectivity continuity commands (heartbeat, polling) maintain session availability without periodic telemetry. Instrumentation commands request permissions, restart Accessibility or MediaProjection services, or bring the application to the front. The screen-capture segment enables or disables screen recording and manipulates overlays, allowing adversary-driven interaction. The event-logger layer provides operator-grade support — logging configuration, counter retrieval, and event flushing. Finally, status-reporting

commands (executePing(), updateCommandStatus()) maintain a reporting loop, giving the operator a quasi-RTT perspective.

One of the key elements is the launch of an FRP client (frpc — Fast Reverse Proxy Client), which establishes an outbound reverse-tunnel connection to an intermediary FRP server (frps — Fast Reverse Proxy Server) controlled by the operator. This tunnel enables remote, interactive access to device functions without exposing any ports on the victim side. All communication is initiated as outbound traffic from the device, allowing the malware to bypass firewalls and NAT restrictions.

This is precisely the model observed in advanced mobile malware and commercial-grade spyware platforms, where push-based signalling replaces persistent TCP channels and minimizes artefacts, logs, and network-level detection opportunities.

FRP — Fast Reverse Proxy

After receiving a remote command via FCM, the application first contacts the operator's Backend API, which serves as the control layer and distributes the network configuration for remote-access channels. As part of this exchange, the malware retrieves the current FRP configuration profile, including the FRPS server address, ports, tunnel type, authorization keys, and any optional parameters related to the WebSocket protocol used for screen streaming or control traffic. The FRPC client itself is not compiled into the application code. Instead, it is extracted from the application resources as an encoded binary file, then decoded, and written to disk as an executable. This allows it to be launched outside the lifecycle of the malware application (Fig. 2).

Once the proxy client binary is prepared, the application launches FRPC as a separate system process, ensuring that the tunnel remains active even if Android terminates the parent malware process. This process establishes an initial outbound connection to the FRPS server using the supplied parameters and immediately reports its authentication status to the operator. As a result, the client is registered on the FRPS side, allowing the operator to observe the infection as an available device identified by a unique identifier. FRPC then transitions into an idle / keep-alive state, maintaining the control channel but refraining from establishing active reverse-proxy traffic until the operator explicitly requests a session.

Press enter or click to view image in full size

```
public final File extractFrpcFromAssets() throws IOException {
    try {
        File file = new File(this.context.getDataDir(), "frpc");
        if (file.exists() && file.length() > 1000000) {
            Logger logger = Logger.INSTANCE;
            logger.d(TAG, "FRPC already extracted: " + file.getAbsolutePath());
            if (makeExecutable(file)) {
                return file;
            }
            Logger.e$default(logger, TAG, "❌ Failed to make existing FRPC executable, re-extracting...", null, 4, null);
            file.delete();
        }
        Logger.INSTANCE.i(TAG, "📦 Extracting FRPC binary from assets...");
        if (file.exists()) {
            file.delete();
        }
        InputStream inputStreamOpen = this.context.getAssets().open("frpc/frpc");
        try {
            FileOutputStream fileOutputStream = new FileOutputStream(file);
            try {
                byte[] bArr = new byte[8192];
                for (int i = inputStreamOpen.read(bArr); i >= 0; i = inputStreamOpen.read(bArr)) {
                    fileOutputStream.write(bArr, 0, i);
                }
                fa.l(fileOutputStream, null);
                fa.l(inputStreamOpen, null);
                Logger logger2 = Logger.INSTANCE;
                logger2.i(TAG, "✅ FRPC extracted: " + file.getAbsolutePath() + " (" + file.length() + " bytes)");
                if (makeExecutable(file)) {
                    return file;
                }
                Logger.e$default(logger2, TAG, "❌ Failed to make FRPC executable", null, 4, null);
                return null;
            } finally {
            }
        } finally {
        }
    } catch (Exception e) {
        yd0.p(e, new StringBuilder("Failed to extract FRPC: "), Logger.INSTANCE, TAG, e);
        return null;
    }
}
```

Figure 2. Class responsible for extracting the FRPC client from the application resources.

At this stage, the device waits for a remote reverse-connection establishment, i.e., for a scenario in which the FRPS server opens an inbound port and forwards traffic down the tunnel directly to the handset. When the operator issues such an initiation, FRPC establishes a specific tunel — TCP, WebSocket, RDP-like, ADB-over-TCP, or screen-streaming — and maps it to a local port on the device. Because the tunnel effectively acts as a port router, the application does not need to expose explicit RAT-style logic. The operator can attach arbitrary tools that operate locally on the system, such as Accessibility-based WebDriver controls, MediaProjection for screen capture, key overlays, or a local shell/terminal.

Subsequently, the tunnel is maintained by FRP heartbeat mechanisms, which in practice means continuous link persistence without the malware performing periodic beaconing. If the channel is disrupted, the binary autonomously reconnects; if the system terminates the process, the application restarts it via a resident service. This ultimately results in a state where the device remains fully accessible to the operator as a network-reachable asset, ready for manual, interactive control.

The established FRP tunnel constitutes the primary operational channel, used for remote sessions that include system interface manipulation, screen capture, application interaction, and other post-compromise activities. The operator accesses the device through the FRP infrastructure rather than directly, which further complicates the attribution and attack-topology analysis. All identified network connections are initiated exclusively from the infected device. No listening services or direct inbound connections were observed. Communication with the Google infrastructure (FCM), the C2 backend, and the FRP server is conducted using widely adopted protocols and cloud services, allowing effective blending of malicious traffic into legitimate network background noise.

Press enter or click to view image in full size

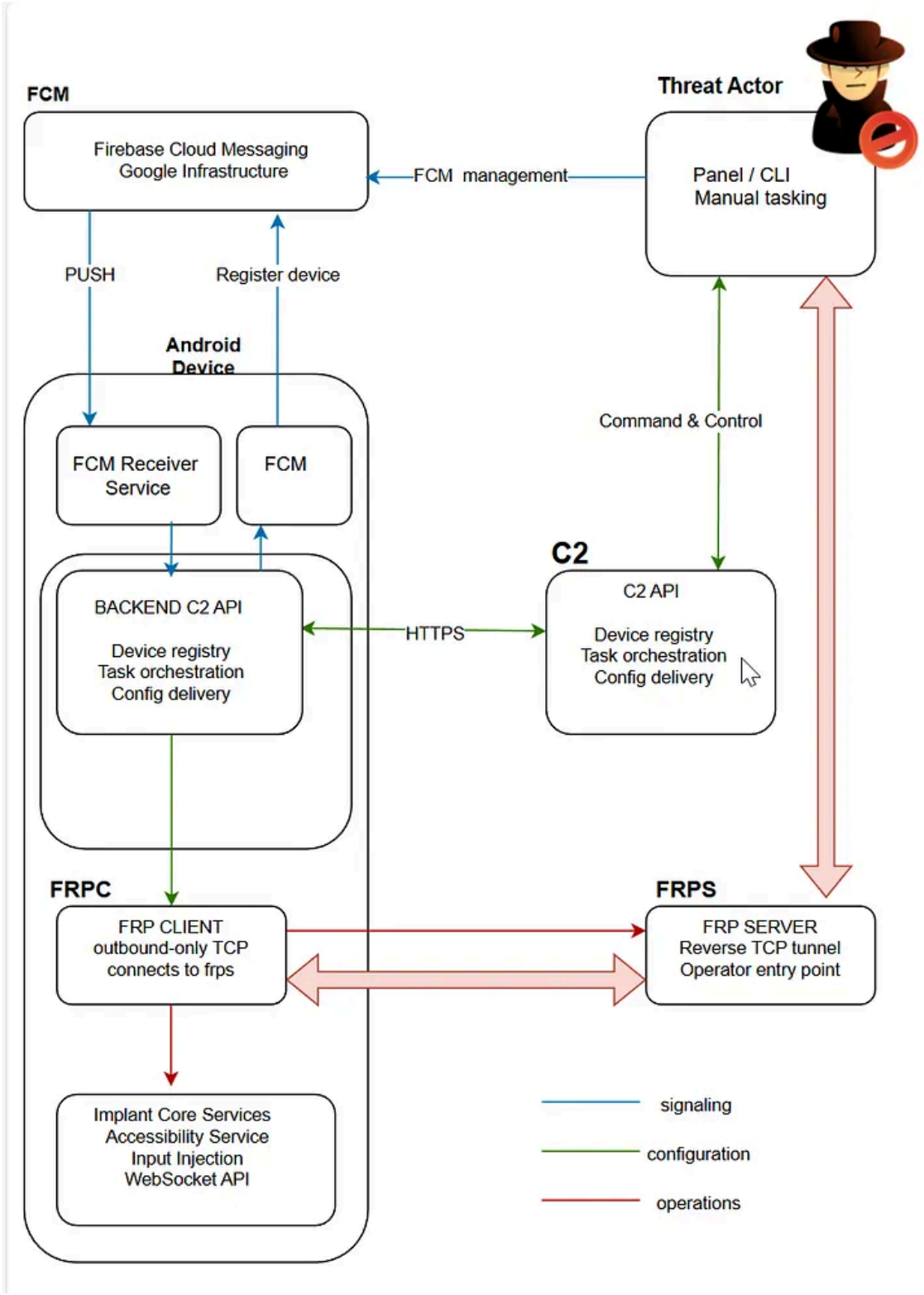

Figure 3. Communication flow between FCM, FRPC, and FRPS modules.

Although malware maintains a conventional HTTPS channel to the Backend API — used primarily for configuration retrieval, command acknowledgements, session parameter updates, and status reporting — this channel does not serve as the transactional layer for remote device control. The Backend API fulfills a signalling and administrative role (control logic, telemetry, device registration), while the persistent and operational

functionality is delegated to a dedicated FRPC process, launched as a separate system process and maintaining a continuous reverse-proxy channel.

As a result, communication is explicitly divided into three layers: FCM acting as an asynchronous wake-up mechanism, Backend API functioning as the malware's administrative control panel, and FRP serving as the actual operational channel that materializes remote access to the device.

Summary

The applied architecture — featuring separation of communication channels, the use of external cloud infrastructure for signalling, and manually operated interactive access — clearly indicates an advanced operational model. Such solutions are characteristic of targeted, operator-driven activities, rather than mass-scale, fully automated malware campaigns.

No attribution effort was undertaken for this malware. The identified campaign exhibits traits of a highly targeted and deliberately planned operation, which justifies its classification as APT-grade malware. In this case, the operator does not rely on a broad distribution but instead conducts activities requiring precise, sustained control over a single device. The use of a reverse communication model, dynamic retrieval of operational parameters, and FRP tunnelling capabilities point to an operational requirement for long-term, low-visibility access, consistent with the tradecraft of APT groups or advanced cybercrime-as-a-service providers.

At the infrastructure level, the observed domain was active for approximately ten days, during which time it was registered, provisioned with an SSL certificate, and subsequently decommissioned after the operational phase was completed. This sequence — short exposure window, enforced encrypted communications and rapid abandonment of infrastructure — aligns with a "burn-after-use" infrastructure model, commonly employed in campaigns that assume the risk of operator deanonymization. This behaviour is not typical of low-tier cybercrime operations, which often rely on inexpensive, long-lived domains, reused panels, or cost-constrained rotation. In this case, the operational cost and the intent to minimize forensic traces are clearly visible.

The communication vector further demonstrates that the software is not a conventional trojan but rather an access-and-surveillance mechanism. The application maintains operator control through FCM push signalling, performs telemetry-based heartbeat operations, retrieves C2 configuration, and activates a Fast Reverse Proxy client using dynamically distributed parameters. The result is a bidirectional access channel in the user's ecosystem, enabling exposure of local services without requiring a public IP address on the victim side.

Samples are generated and modified on a per-instal basis; FRP configuration is retrieved ephemerally for each execu The FRP, and domains and certificates exhibit a deliberately short lifespan. These characteristics prevent the creation of traditional detection signatures and further suggest that an actor employs strong operational security practices, deliberate infrastructure rotation, and minimization of retrospectively detectable artefacts.

Consequently, the analyzed code should not be considered commodity malware, but rather a tool capable of precise device control, covert communications, and sustained operator oversight, placing it squarely within the spectrum of TTPs used by APT groups or entities offering comparable commercial capabilities.

From an attribution perspective, it should be noted that the sample exhibits two distinctly different approaches to code obfuscation, both deviating from standard practices commonly observed in mobile malware. Once the primary application component is decrypted, the codebase remains coherent and readable: class, method, and variable names are preserved; there is no evidence of automated obfuscation; and some modules even implement local file-based event logging — an uncommon feature in malware designed purely for concealment. This profile suggests a high level of software engineering competence and development practices more typical of professional software production than of malware authored solely for illicit distribution.

At the same time, a different layer of application — covering the operational “engine” and structural dependencies — is clearly obfuscated. Classes lack semantic naming, and structures rely on automatically generated field identifiers and index-based mappings (e.g., *SparseIntArray → index → value*), complicating the reconstruction of the logical flow. This divergence suggests a development process in which portions of the code may have been derived from an existing, legitimate codebase (or maintained by experienced developers), while the operational components –responsible for class bindings, runtime parameters, and execution logic — were subjected to automated obfuscation at a later stage. It cannot be ruled out that the development was conducted collaboratively and that some contributors may not have been fully aware of the operational purpose of the application.

END OF ANALYSIS — 21 December 2025 (IR3k)

The above analysis does not cover the topic exhaustively and focuses solely on selected, noteworthy functionalities of this malware. The malware itself contains numerous additional features that could be described; however, this analysis concentrates exclusively on communication-related components. The purpose of this report is educational, illustrating alternative and advanced techniques implemented in professional-grade malware. Any inaccuracies or errors that may appear are unintentional.

This analysis has been published on the CERT Orange Polska portal at:
https://cert.orange.pl/aktualnosci/operacyjna-analiza-kanalow-komunikacyjnych-mobilnego-rcsa/
