Documentation — Enhanced Dynamic Analysis Frida Script

Dynamic Analysis

Dynamic Analysis is the process of examining how an application behaves **while it is running** in a real or controlled environment.

Instead of only inspecting code or files statically, dynamic analysis focuses on **observing the actual execution flow, interactions, and data usage** of an application during runtime.

It allows analysts to understand what an app *really does* when it is launched, used, and connected to the system, the network, and various device resources.

Key Characteristics of Dynamic Analysis

1. Real-Time Observation

The analysis is performed on a running application. This allows you to monitor behaviors that cannot be seen through static code inspection alone (e.g., dynamically loaded classes, runtime permissions, encrypted strings, or conditional logic).

2. Execution-Based Behavior Tracking

Dynamic analysis reveals the real actions taken by the app, such as:

- Network requests
- File operations
- Data storage patterns
- Access to sensors and hardware
- Background processes and threads
- Cryptographic operations
- Runtime permission usage

These behaviors can differ significantly from what static analysis predicts.

3. Detection of Hidden or Trigger-Based Behavior

Some apps execute certain actions only under specific conditions, such as:

- When a button is pressed
- When logged in
- When a certain server responds
- When a device is rooted or connected to the internet

Dynamic analysis triggers and observes these conditional behaviors.

4. Environment Interaction Monitoring

Apps interact with multiple system components and APIs. Dynamic analysis tracks such interaction in real time:

- Operating system APIs
- File system
- Telephony services
- Location services
- Camera/microphone activation
- Inter-process communication (IPC)
- Network stack

This helps create a complete behavior profile.

5. Useful for Security and Privacy Assessment

Dynamic analysis identifies:

- Data leakage channels
- Usage of sensitive permissions
- Suspicious network calls
- Unauthorized access to sensors
- Hidden background activity
- Malicious payload execution

Why Dynamic Analysis is Important

A. Captures Real Behavior

Applications often change behavior based on the environment. Dynamic analysis reveals the *actual* operations performed, not just the theoretically possible ones.

B. Complements Static Analysis

Static analysis shows **what the code could do**, while dynamic analysis shows **what the code really does**.

C. Detects Runtime Obfuscation Bypass

Many apps use obfuscation, encryption, or anti-analysis techniques that hide logic from static analysis. But during runtime, these hidden parts get executed and can be observed.

D. Ideal for Malware, Privacy, or Behavior Analysis

Dynamic monitoring exposes:

- C2 server communication
- Stealthy SMS sending
- Hidden camera/mic access
- Data exfiltration
- Behavior triggered only after certain delays or interactions

1 — High-level purpose and overview

Purpose:

This Frida script instruments an Android app at runtime to monitor and report (via send) a set of high-value actions that are commonly interesting when performing dynamic security, privacy, or behavior analysis of APKs. The script hooks Java APIs to observe:

- Network requests
- File access / file creation
- SharedPreferences writes
- Cryptographic API usage
- Location access
- Contacts queries
- SMS sending
- Camera and microphone usage

Why this is useful:

When analyzing an APK dynamically, you want to know what the app **actually does at runtime** (not only what static analysis shows). This script records events that indicate potentially sensitive or risky behavior (exfiltration, credential handling, data-leak vectors, covert channels, permission usage) and timestamps them. Those events form an evidence trail that can be correlated with UI actions, user inputs, or other behaviors.

2 — Execution context and mechanism (how Frida and `Java.perform` fit)

- **Frida runtime:** The script runs inside the target app process while the app is executing. Frida allows you to override (intercept) Java methods and constructors in the Dalvik/ART environment.
- **`Java.perform(function() { ... });`:** Ensures instrumentation runs when the Java runtime is ready. All hooks are installed inside this callback.
- **`try { ... } catch(e) { console.log("... hook failed"); }`:** Each hook is protected by try/catch so a failure to hook one API does not prevent the rest of the script from installing other hooks. This makes the script resilient across different apps and Android versions.

3 — What `send(...)` does (telemetry interface)

- Each hook uses `send(...)` to deliver a JSON-like object to the Frida host (the tool controlling Frida, e.g., a Python or Node.js client).
- The host receives these messages and can log, persist, or visualize them in real time.

- The send payload includes `type` (category), event-specific fields (e.g., `url`, `path`, `key`), and a `timestamp` (ISO string). Timestamps let you correlate events across multiple hooks and with user actions or network traffic captures.

4 — Detailed walkthrough of each hooked area

For each item below: I summarize the API being hooked, what the script records, and why that observation matters.

A. Network monitoring — `java.net.HttpURLConnection.getInputStream()`

- **What is hooked:** When the app opens an HTTP(S) connection and reads the response input stream via `getInputStream()`.
- **What is recorded:** Request method (GET/POST/etc.), the URL (from `getURL().toString()`), and a timestamp.
- **Why it matters:** Network requests are prime indicators of data exfiltration or communication with command-and-control servers. Capturing the URL and method helps identify endpoints, APIs, and potentially suspicious domains or IPs contacted by the app. Even if TLS encrypts the payload, the hostnames and endpoints reveal intent.

B. File access — `java.io.File(String path)` constructor

- **What is hooked:** Creation/initialization of `File` objects given a string path.
- **What is recorded:** The file path string and a timestamp.
- **Why it matters:** File paths show where the app reads from or writes to on the filesystem (internal storage, external storage, cache, `/data/` directories). This helps detect when apps create configuration files, store sensitive tokens, or save exfiltrated data.

C. SharedPreferences writes — `SharedPreferences.Editor.putString(key, value)`

- **What is hooked:** Writes to SharedPreferences via `putString`.
- **What is recorded:** The key and a (full or truncated) value and a timestamp.
- **Why it matters:** SharedPreferences is a common place apps store credentials, tokens, flags, or configuration. Monitoring writes can reveal where secrets or persistent identifiers are kept.

D. Crypto operations — `javax.crypto.Cipher.getInstance(String transformation)`

- **What is hooked:** Calls to obtain `Cipher` instances (e.g., `AES/CBC/PKCS5Padding`).
- **What is recorded:** The transformation string (algorithm/mode/padding) and timestamp.
- **Why it matters:** Observing which cryptographic primitives are used helps identify whether the app uses standard, weak or custom crypto. It can indicate encryption of stored or transmitted data, or attempts to obfuscate payloads. If suspicious transformations or custom cipher names appear, they warrant deeper inspection.

E. Location access — `android.location.LocationManager.getLastKnownLocation(provider)`

- **What is hooked:** Calls to obtain last known location using a provider (`gps`, `network`, etc.).
- **What is recorded:** The provider name and timestamp.
- **Why it matters:** Location access is privacy-sensitive. Detecting when the app requests location helps verify permission usage and intent (tracking, geofencing, targeted behavior) and supports privacy compliance analysis.

F. Contacts access — `ContentResolver.query(...)` for URIs that include `"contacts"`

- **What is hooked:** Content resolver queries; the script filters URIs that include the word `"contacts"`.
- **What is recorded:** The URI string and timestamp when contacts are queried.
- **Why it matters:** Access to contacts can enable spam, contact harvesting, or social graph theft. Detecting contacts queries shows whether the app reads user address books.

G. SMS sending — `android.telephony.SmsManager.sendTextMessage(...)`

- **What is hooked:** Outgoing SMS sending API.
- **What is recorded:** Destination number, truncated preview of message text (first 50 chars), and timestamp.

- **Why it matters:** SMS sending is high-risk: premium number abuse, exfiltration, or verification bypass. Logging destination and message preview indicates suspicious messaging activity or misuse of telephony.

H. Camera access — `android.hardware.Camera.open(id)`

- **What is hooked:** Attempts to open a hardware camera (by id).
- **What is recorded:** Camera id and timestamp.
- **Why it matters:** Camera access may reveal attempts to capture images or video. Detecting access helps identify surveillance or covert recording behavior.

I. Microphone access — `android.media.MediaRecorder.start()`

- **What is hooked:** Start of audio recording via `MediaRecorder`.
- **What is recorded:** Event type `microphone_access` and timestamp.
- **Why it matters:** Microphone usage can indicate audio recording. Logging this event helps catch potentially covert audio capture or permission misuse.

5 — Overall event flow during analysis

1. **Attach Frida to the running app** and load this script. (This step is performed by the analyst via their Frida client — not part of the script.)
2. **Script installs hooks** in `Java.perform`. Each successful hook begins intercepting calls to the targeted Java methods/constructors.
3. **When the app executes one of the hooked APIs**, control passes to the script's replacement implementation.
4. **Replacement implementation calls `send(...)`** with a structured event payload describing what happened and when.
5. **Replacement then delegates to the original method** (the original behavior is invoked to avoid breaking app logic).
6. **The Frida host receives events** in real time and can log, visualize, or store them. The analyst correlates these events with network traces, UI interactions, or other monitoring outputs.

This flow means the script provides **non-disruptive observation**: it reports data but attempts to preserve the original behavior so the app runs normally (minimizing behavioral changes introduced by instrumentation).

6 — How this supports dynamic analysis objectives

- **Behavioral discovery:** Reveals runtime behaviors that static analysis may miss (dynamic URLs, generated file paths, runtime crypto choices).
- **Evidence collection:** Generates time-stamped events for reports and reproducible analysis steps.
- **Prioritization:** Points analysts to suspicious endpoints, files, or APIs to inspect further (e.g., capture network traffic to a logged URL, or dump a recorded file path).
- **Validation:** Confirms whether suspected behavior (from static analysis) actually occurs at runtime.
- **Debugging and reverse engineering aid:** Shows which APIs and code paths the app takes in response to inputs or UI flows.

7 — Typical analysis workflow using the script (conceptual steps)

1. **Set up the analysis environment** (emulator or instrumented device) and prepare network capture (pcap) and logs.
2. **Launch target app** and attach Frida, load the script.
3. **Interact with the app** (automated UI flows or manual exploration) to trigger behavior.
4. **Monitor real-time send output** to observe events captured by the script.
5. **Correlate events** with network captures, system logs, and UI actions to reconstruct cause/effect.
6. **Follow up**: Use outputs to decide next steps — deeper tracing, memory dumps, hooking additional APIs, or forensic extraction of files noted by the script.

8 — Safety, ethics, and legal considerations

- **Only analyze apps you own or are authorized to analyze.** Reverse engineering and runtime inspection of third-party apps may be restricted by law or license.
- **Protect user privacy.** The script can capture personal data — treat all captures as sensitive and handle them under appropriate privacy and data protection rules.
- **Responsible disclosure.** If you discover vulnerabilities, follow responsible disclosure policies before public release.