Private Cellular Network Baseline Backup Calling Lab Guide

This handbook details the deployment, runtime instrumentation, and diagnostic framework required to execute, validate, and optimize Backup Calling over an Android 17 Private Cellular Network testing topology. This deployment layer targets the core telephony memory mapping to force-activate virtual Wi-Fi calling infrastructure tunnels over co-located cellular sub-interfaces.

Prerequisites for Testing

- **Target Device:** Google Pixel 6 or newer physical hardware reference device (Tensor Architecture/Exynos baseband).
- **Operating System:** Android 17 mobile architecture baseline.
- **Privilege Level:** Pre-rooted via Magisk or KernelSU with an unlocked superuser (`su`) binary resident.
- **Mac Host Binary Location:** Android Debug Bridge (ADB) reference utilities staged at `/Users/xxxxxxxx/Downloads/platform-tools/adb`.

Phase 1: Get Your Mac Ready

Ensure your host environment possesses the management tools to translate instrumentation instructions over the USB layer. Open your Mac Terminal and execute the framework installation:

```
None
pip3 install --upgrade frida-tools
```

Once the installation loop completes, query your local engine version to prevent runtime handshake crashes:

```
None
frida --version
```

CRITICAL GAP-CHECK: Note this exact version string (e.g., `17.15.3`). The remote asset deployed to your hardware reference target in Phase 2 must match this version compilation line exactly to prevent runtime handshake crashes.

Phase 2: Send the Server to Your Pixel

1. Navigate to the official Frida GitHub Releases page on your host web browser.
2. Locate the release directory matching your Mac version string, expand the **Assets** cascade, and download the target platform asset: `frida-server-XX.X.X-android-arm64.xz`.
3. Open your Mac Terminal, change directory into your local staging space, extract the container, and rename the output binary (ensuring clean command formatting):

```
None
cd ~/Downloads

unxz frida-server-*-android-arm64.xz

mv frida-server-*-android-arm64 frida-server
```

4. Clear out lingering diagnostic hooks, transfer the server execution module, loosen runtime security profiles, and run the server engine as a persistent system daemon.

REMINDER: ***This will be the starting point going forward after SIM changes or device power cycles.***

```
None
/Users/networkfx/Downloads/platform-tools/adb push
~/Downloads/frida-server /data/local/tmp/

/Users/networkfx/Downloads/platform-tools/adb shell su -c "pkill -f
frida-server; pkill -f frida-inject; pkill -f frida"

/Users/networkfx/Downloads/platform-tools/adb shell su -c "chmod
755 /data/local/tmp/frida-server"

/Users/networkfx/Downloads/platform-tools/adb shell su -c
"setenforce 0"

/Users/networkfx/Downloads/platform-tools/adb shell su -c "nohup
/data/local/tmp/frida-server > /dev/null 2>&1 &"
```

5. Confirm the gate is open by querying active processes on the connected USB target:

None

```
frida-ps -U
```

Expected On-Screen Outputs (Success Flags):

If it spits out a long scrolling list of active processes (like com.android.phone, system_server, etc.), the gate is open. Once you see that list, you are officially ready to build the script.

Phase 3: Create Your Custom Hook File

To bypass unexpected text formatting anomalies, generate your structured injection script directly from your terminal session. This configuration incorporates dynamic Java-casting bridges to avoid `TypeError: not a function` faults at runtime. Copy and paste this block into your Mac Terminal and hit **Enter**:

None

```
cat << 'EOF' > carrier_hook.js

Java.perform(function () {

    try {

        var ActivityThread =
Java.use('android.app.ActivityThread');

        var appContext =
ActivityThread.currentApplication().getApplicationContext();



        var CarrierConfigManager =
Java.use('android.telephony.CarrierConfigManager');

        var PersistableBundle =
Java.use('android.os.PersistableBundle');

        var SubscriptionManager =
Java.use('android.telephony.SubscriptionManager');
```

```
        // Strict Type-Bridge Definitions to handle un-typed JNI
collections

        var List = Java.use('java.util.List');

        var SubscriptionInfo =
Java.use('android.telephony.SubscriptionInfo');



        var rawConfigService =
appContext.getSystemService('carrier_config');

        var carrierConfigManager = Java.cast(rawConfigService,
CarrierConfigManager);



        var rawSubService =
appContext.getSystemService('telephony_subscription_service');

        var subscriptionManager = Java.cast(rawSubService,
SubscriptionManager);



        var macroSubId = 1;     // Baseline fallback
initializations

        var privateSubId = 3;



        console.log("[Frida] Scanning active subscription profiles
dynamically...");

        var rawSubList =
subscriptionManager.getActiveSubscriptionInfoList();
```

```
        if (rawSubList !== null) {
            console.log("[Frida] -> Advanced Multi-SIM Sorting
Engine Active.");

            // Cast the raw object list directly to a usable Java
List
            var subList = Java.cast(rawSubList, List);

            // Pass 1: Explicit Keyword Evaluation
            for (var i = 0; i < subList.size(); i++) {
                // Cast the raw list item directly to a
SubscriptionInfo model
                var info = Java.cast(subList.get(i),
SubscriptionInfo);

                var sId = info.getSubscriptionId();
                var slotId = info.getSimSlotIndex();

                var rawName = info.getDisplayName();
                var carrierName = rawName ? rawName + "" :
"unknown"; // Coerce Java CharSequence safely to JS String
                var nameLower = carrierName.toLowerCase();
```

```
                console.log("[Frida] -> Inspecting Slot " + slotId
+ " (subId: " + sId + "): [" + carrierName + "]");


                if (nameLower.includes("verizon") ||
nameLower.includes("vzw") ||

                    nameLower.includes("at&t") ||
nameLower.includes("att") ||

                    nameLower.includes("t-mobile") ||
nameLower.includes("tmo") ||

                    nameLower.includes("google fi")) {


                    macroSubId = sId;

                    console.log("[Frida]    --> MATCH: Assigned
subId " + sId + " as Primary Commercial Line.");

                }

                else if (nameLower.includes("prv") ||
nameLower.includes("private") ||

                         nameLower.includes("test") ||
nameLower.includes("lab") ||

                         nameLower.includes("cbrs")) {


                    privateSubId = sId;

                    console.log("[Frida]    --> MATCH: Assigned
subId " + sId + " as Target Private Network.");
```

```
                }
            }

            // Pass 2: Process of Elimination Safety Net
            if (subList.size() === 2 && privateSubId === 3) {
                for (var j = 0; j < subList.size(); j++) {
                    var infoCheck = Java.cast(subList.get(j),
SubscriptionInfo);
                    var checkId = infoCheck.getSubscriptionId();
                    if (checkId !== macroSubId) {
                        privateSubId = checkId;
                        console.log("[Frida]    --> RESOLVED via
elimination: Assigned subId " + checkId + " as Private Network.");
                    }
                }
            }
        }

        // ========================================
        // Configure Private Network (Opportunistic Slot)
        // ========================================
        try {
```

```
            var privateBundle = PersistableBundle.$new();

            privateBundle.putBoolean('is_private_network_bool',
true);

            privateBundle.putBoolean('force_home_network_bool',
true);



            carrierConfigManager.overrideConfig(privateSubId,
privateBundle);

            console.log("[Frida] -> Successfully loaded Secure
Slice overrides on Private subId: " + privateSubId);

        } catch (err) {

            console.log("[Frida ERROR] Private slice override
allocation fault: " + err);

        }



        // ========================================

        // Configure Primary Commercial Network (MNO/Commercial
Macro)

        // ========================================

        try {

            var vzwBundle = PersistableBundle.$new();

            vzwBundle.putInt('opp_auto_data_switch_policy_int', 3);


vzwBundle.putBoolean('carrier_cross_sim_ims_available_bool', true);
```

```
vzwBundle.putBoolean('enable_cross_sim_calling_on_opportunistic_dat
a_bool', true);

            vzwBundle.putBoolean('carrier_wfc_ims_available_bool',
true);



            carrierConfigManager.overrideConfig(macroSubId,
vzwBundle);

            console.log("[Frida] -> Successfully unlocked Cross-SIM
IWLAN Tunneling keys on Commercial subId: " + macroSubId);

        } catch (err) {

            console.log("[Frida ERROR] Macro profile override
allocation fault: " + err);

        }



    } catch (globalErr) {

        console.log("[Frida CRITICAL ERROR] Telephony
initialization wrapper failure: " + globalErr);

    }

});

EOF
```

Check 1: Verify the File Exists

Run the `ls -l` command to see if the file is sitting in your current folder and has a valid file size:

```
None
ls -l carrier_hook.js
```

Expected On-Screen Outputs (Success Flags):

It should print a single line showing the file permissions, your username, and the file size (which should be around 3,000 to 3,500 bytes).

```
-rw-r--r-- 1 networkfx staff 3420 Jun 29 09:00 carrier_hook.js
```

Check 2: Read the File Contents

If you want to look inside the file to make sure the code isn't corrupted or cut off, use the cat command to print it to your screen:

```
None
cat carrier_hook.js
```

Expected On-Screen Outputs (Success Flags):

The terminal will dump the entire JavaScript code block we just created. Scroll to the very bottom of the printout—if you see `}, globalErr);` and `});` cleanly at the end, your file is structurally perfect.

Phase 4: Flip the Global System Switch & Process Flush

To ensure the Android OS does not sleep or block this cross-SIM handshake, we must write our global parameters to the system databases and flush the telephony process memory space **before** running Frida. This allows the newly initialized baseband process to consume the settings cleanly.

In your Mac terminal window, run the following three commands:

```
None
/Users/networkfx/Downloads/platform-tools/adb shell su -c "settings put global cross_sim_calling_enabled 1"

/Users/networkfx/Downloads/platform-tools/adb shell su -c "settings put global opportunistic_data_auto_switch_enabled 1"
```

```
/Users/networkfx/Downloads/platform-tools/adb shell su -c "killall com.android.phone"
```

(Allow 3 seconds for the baseband and Shannon telephony processes to re-initialize before proceeding).

Check 1: Verify the System State

To confirm the phone's operating system has actively accepted all of our rules, run this diagnostic query in your new tab:

```
None
/Users/networkfx/Downloads/platform-tools/adb shell su -c "dumpsys carrier_config" | grep -E "opportunistic_data|wfc_ims_available|private_network"
```

Expected On-Screen Outputs (Success Flags):

It should print:

```
carrier_wfc_ims_available_bool = false
enable_cross_sim_calling_on_opportunistic_data_bool = false
is_private_network_bool = false
carrier_wfc_ims_available_bool = true
```

Check 2: Stream the Live Lab Telemetry

Now, copy and paste this command into that same second terminal tab and hit Enter. This turns the terminal into a live telemetry stream tracker:

```
None
/Users/networkfx/Downloads/platform-tools/adb shell su -c "dumpsys telephony.registry" | grep -E "mActiveDataSubId|mServiceState|mImsRegistrationState"
```

Expected On-Screen Outputs (Success Flags):

Right now, it will show `mActiveDataSubId=1` (MNO) and `mServiceState=0` (In Service). Leave this window visible.

Phase 5: Start the Hooking Process

Now we need to update the core system databases to ensure Android doesn't block or sleep this cross-SIM handshake.

Step 1: Open a New Tab

Press **`Cmd + T`** on your Mac keyboard to spin up a brand-new, clean terminal tab. **Do not close the original terminal window.**

Step 2: Push the Global Overrides

Because Android instantiates a unique runtime Process ID (PID) on every subsystem reload, execute this automated command to find the newly initialized process container and inject your memory hooks:

Copy and paste this clean block of database updates and cache reloads into your **new terminal tab** and hit **Enter**.

CRITICAL REMINDER: Once this command fires up and drops you into the active Frida console loop, **do not close, interrupt, or touch this terminal tab.** It must remain running continuously to hold the memory hooks active.

```
None
PID=$(/Users/networkfx/Downloads/platform-tools/adb shell su -c "pidof com.android.phone" | tr -d '\r') && frida -U -l carrier_hook.js -p $PID
```

Expected On-Screen Outputs (Success Flags):

1. The Handshake: A large text-art banner reading "Frida" will print to the screen, showing the engine compilation details.
2. The Target Dynamic Scan: The script will instantly fire and print its baseline operational log text:
   `[Frida] Scanning active subscription profiles dynamically...`
3. The Identity Sorting Match: You will see the script inventory your SIM cards line-by-line, showing their slot indices, active carrier strings (like `prv8` or your commercial network label), and assigned roles.

The Final Lock: The script will output its success flags:

```
[Frida] Scanning active subscription profiles dynamically…
[Frida] -> Advanced Multi-SIM Sorting Engine Active.
```

```
[Frida] -> Inspecting Slot 0 (subId: 1): [Verizon ]
[Frida]    --> MATCH: Assigned subId 1 as Primary Commercial Line.
[Frida] -> Inspecting Slot 1 (subId: 3): [PRV8]
[Frida]    --> MATCH: Assigned subId 3 as Target Private Network.
[Frida]    --> RESOLVED via elimination: Assigned subId 3 as Private
Network.
[Frida] -> Successfully loaded Secure Slice overrides on Private
subId: 3
[Frida] -> Successfully unlocked Cross-SIM IWLAN Tunneling keys on
Commercial subId: 1
```

Phase 6: Verify Active Telemetry Cache

Open a **second Mac terminal tab** (**Cmd + T**) to execute your post-injection system verification checks.

1. Confirm Active Carrier Configuration Overrides

In the original terminal window run this diagnostic query to confirm that the `com.android.phone` process memory space is actively maintaining the overrides injected by Frida:

```
None
/Users/networkfx/Downloads/platform-tools/adb shell su -c "dumpsys
carrier_config" | grep -E
"opportunistic_data|wfc_ims_available|private_network"
```

Expected On-Screen Outputs (Success Flags):

```
overrideConfig: subId=4, persistent=false,
overrides=PersistableBundle[{force_home_network_bool=true,
is_private_network_bool=true}]
overrideConfig: subId=1, persistent=false,
overrides=PersistableBundle[{enable_cross_sim_calling_on_opportunistic
_data_bool=true, carrier_wfc_ims_available_bool=true,
carrier_cross_sim_ims_available_bool=true,
opp_auto_data_switch_policy_int=3}]
```

2. Diagnostic Expansion & Debugging Checklist

If the cross-SIM handoff parameters match but the status indicator does not show backup signaling active, open a third terminal window on your Mac host and run this target logging filter stream:

```
None
/Users/networkfx/Downloads/platform-tools/adb logcat | grep -E "CarrierConfig|Opportunistic|Ims|CrossSim|ePDG"
```

3. Verify Shannon Modem & Core IMS Service Bindings

Advanced IMS Subsystem Verification (Shannon/Google Stack)

Because modern Android environments wrap lower-level IP Multimedia Subsystem (IMS) routing metrics inside parent telephony layers, run a targeted filter against the master phone container to verify real-time voice feature attachments:

```
None
/Users/networkfx/Downloads/platform-tools/adb shell su -c "dumpsys phone" | grep -i "ims" | head -n 50
```

Expected On-Screen Outputs (Success Flags):

- `slotToSubIdMap={0=1, 1=3}`: Confirms the hardware slot-to-sub ID map matches your baseline.
- `features=[{s=0, f=MMTEL}, {s=1, f=MMTEL}]`: Confirms Multimedia Telephony (`MMTEL`) is registered on both physical SIM channels.
- `isBound=true`: Validates that the underlying radio drivers (`com.shannon.imsservice`) have firmly latched onto both slots.

Phase 7: Simulate the Signal Handoff

1. Turn off **Wi-Fi** on your Pixel device to isolate the active transceiver paths.
2. Confirm your private network SIM displays active, registered cell data bars over your custom Standalone cell site.
3. In your **second terminal tab**, start the live tracking telemetry stream:

```
None
/Users/networkfx/Downloads/platform-tools/adb shell su -c "dumpsys telephony.registry" | grep -E "mActiveDataSubId|mServiceState|mImsRegistrationState"
```

Crucial DSDS Hardware Verification Workaround:

On Google Pixel Tensor architectures, toggling **Mobile Radio Power** to **OFF** inside the phone's engineering menu (`*#*#4636#*#*`) can frequently pull the power supply down on the shared cellular transceiver chip globally. This results in **both** Slot 0 and Slot 1 dropping out, which prevents the cross-SIM handoff from negotiating.

To drop the commercial network link selectively while keeping your co-located private SIM core radio fully active, execute the testing method below. This approach forces an isolated network registration denial on the MNO line without triggering any transceiver power-down states:

1. On your physical Pixel 7 handset, navigate to **Settings** -> **Network & internet** -> **SIMs** -> Tap your **Commercial SIM (MNO)**.
2. Scroll down and toggle the **Automatically select network** switch to **OFF**.
3. Let the scanning loop complete, then select an incompatible carrier or your **Private Network PLMN (e.g., PRV8 / 315010)** from the list.
4. MNO will instantly drop to `1 (OUT_OF_SERVICE)` registration state due to PLMN rejection, leaving the co-located Private Core SIM perfectly active and registered over the transceiver.

Real-Time Handoff Telemetry Verification

Watch the live scrolling logs in your second terminal tab. Within 2–4 seconds of killing the primary radio, trace these three specific shifts:

- `mActiveDataSubId=3`: The baseband core abandons MNO routing (subId 1) and shifts primary IP packet forwarding to your Private SIM (subId 3).
- `mServiceState (Slot 0)`: The signaling voice state transitions from `0 (IN_SERVICE)` to `1 (OUT_OF_SERVICE)`. Within a brief re-evaluation window, it moves to an `IWLAN` registration profile over the secondary slot.
- `mImsRegistrationState=registered`: Confirms that the virtual cross-SIM secure IPsec signaling tunnel successfully mounted and negotiated parameters over your private cellular path.

Hysteresis Analysis: Note that because you have configured an entry hysteresis of `opportunistic_network_data_switch_hysteresis_time_long = 5000` (5000 ms), the operating system will purposefully wait exactly $5\text{

seconds}$ after macro signal degradation before committing to the opportunistic backup routing path.

- **Status Bar Visual:** The network branding string at the top of your physical screen will shift to **"XXXXX (MNO) Backup Calling"**.

Real-Time Handoff Telemetry Verification

Watch the live scrolling logs in your second terminal tab. Within 2–4 seconds of killing the primary radio, trace these three specific shifts:

- `mActiveDataSubId=3`: The baseband core abandons MNO routing (subId 1) and shifts primary IP packet forwarding to your Private SIM (subId 3).
- `mServiceState (Slot 0)`: The signaling voice state transitions from `0 (IN_SERVICE)` to `1 (OUT_OF_SERVICE)`. Within a brief re-evaluation window, it moves to an **`IWLAN`** registration profile over the secondary slot.
- `mImsRegistrationState=registered`: Confirms that the virtual cross-SIM secure IPsec signaling tunnel successfully mounted and negotiated parameters over your private cellular path.
- *Status Bar Visual:* The network branding string at the top of your physical screen will shift to **"XXXXX (MNO) Backup Calling"**.

Phase 8: Private LTE / 5G Network Considerations

Because your phone treats your private network's data pipeline exactly like a Wi-Fi router to build a secure VPN tunnel straight out to MNO''s core infrastructure, there are a few infrastructure-side network configurations that must be met. If the private network core or its edge router blocks or alters the traffic, MNO's secure tunnel will fail to attach silently. Check these three external components:

1. **DNS Resolution Inside the Private Core**
   When MNO loses its macro signal, it immediately asks the active data connection (your Private SIM, Slot 3) to look up its secure gateway addresses via DNS. Your Private Network Core must be configured to pass DNS queries cleanly up to a public DNS server (like 8.8.8.8). The phone will try to resolve MNO's specific endpoint domains, usually wo.vzwwo.com or a carrier 3GPP FQDN like epdg.epc.mnc480.mcc311.pub.3gppnetwork.org. If your private network's DNS server drops or can't resolve public internet domains, the backup calling tunnel dies before it even starts.
2. **Edge Firewall & NAT Rules**
   The data traffic leaving your private network cell must pass through whatever router or firewall connects your lab to the public internet. MNO uses an IPsec/IKEv2 VPN tunnel

to keep your calls secure, so your internet gateway firewall must allow these protocols out to the web:
    - **UDP Port 500:** Used for the initial key exchange.
    - **UDP Port 4500:** Used for NAT traversal and the actual secure voice packets.
    - Make sure IPsec NAT Passthrough (sometimes called IPsec ALG) is enabled on your edge router. Stateful firewalls love to close dormant ports, but keeping these open allows the latent link to stay alive.
3. **The MTU Size (The Silent Killer)**
   This is the most common pitfall in private cellular testing. Private LTE/5G networks use an internal encapsulation wrapper (GTP-U tunnels) to pass data between your cell antenna and your core. This wrapper eats up about 40 bytes of data space. MNO's secure IPsec tunnel also adds heavy data overhead. If your private network assigns the phone a standard MTU of 1500, the combined overhead of the private cell wrapper + MNO's IPsec tunnel will exceed the network limits. Packets will fragment, get dropped by your firewall, and the call will disconnect.
    - **The Fix:** Configure your Private Network Core (PGW/UPF) to assign an **MTU of 1420 or lower** directly to the phone when the SIM attaches. Alternatively, turn on **MSS Clamping (set to ~1360)** on your network's main firewall.

Notes

- **SIM Slot Dependency:** SIM slot info in the scripts above may vary depending on your device and the script may need to be adjusted to reflect your specific setup.
- **MNO String Variations:** MNO naming may vary depending on your MNO SIM and the script may need to be adjusted to reflect your specific setup.
- **Volatile Memory Lifespan Rules:** Remember that all memory overrides injected via Frida operate strictly in-memory. Soft reboots or phone app crashes will dump the variables, requiring a fresh injection phase script execution.