Private Cellular Network Backup Calling, PLMN Whitelist, and 5G SA Enablement 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, add private Home PLMN whitelists, and 5G Standalone (SA) baseband allocations into the system process.

Prerequisites for Testing

- **Target Device**: Google Pixel 6 or newer physical hardware reference device featuring Tensor Architecture and an Exynos baseband.
- **Operating System**: Android 17 mobile architecture baseline.
- **Privilege Level**: Pre-rooted via Magisk or KernelSU with an unlocked superuser (su) binary resident in the system path.
- **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

Open Device Alliance

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:

```
None
cd ~/Downloads

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

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

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 &"
```

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

```
None
frida-ps -U
```

Phase 3: Create Your Custom Hook File (The Golden Path)

To bypass unexpected text formatting anomalies, generate your structured injection script directly from your terminal session. This configuration incorporates an **overload-agnostic reflection asset writer** to bypass Android 17 runtime return validation checks, whitelists non-commercial private PLMN arrays, and unblocks 5G Standalone configurations.

This script features state-tracked logging arrays and exception gates (trackedSubIds, hasLoggedException) that limit carrier-specific confirmations and dynamic loop crash logs to exactly once per simulation session, isolating your Mac from terminal console floods on high-frequency hot paths.

Copy and paste this entire block into your Mac Terminal and hit **Enter**:

None

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

Java.perform(function () {

    var macroSubId = 1;             // Baseline default
initializations

    var privateSubId = -1;          // Flagged to -1 to actively
catch unassigned slots

    var privateMatched = false;     // Tracking latch for the
elimination pass

    var trackedSubIds = [];         // Prevents hot-path terminal
logging floods

    var hasLoggedException = false;    // Exception flood gate


    console.log("[Frida] Hook successfully attached. Initializing
Android 17 Auto-Switching Engine...");
```

```
    // Pure reflection field setter helper (Circumvents Android 17
JNI type validation limits)
    function setFieldPure(obj, fieldName, value) {
        try {
            var cls = obj.getClass();
            var field = null;
            while (cls !== null) {
                try {
                    field = cls.getDeclaredField(fieldName);
                    break;
                } catch (e) {
                    cls = cls.getSuperclass();
                }
            }
            if (field !== null) {
                field.setAccessible(true);

                // Dynamically route primitives through explicit
reflection methods to satisfy JNI type rules
                if (typeof value === 'boolean') {
                    field.setBoolean(obj, value);
                } else {
```

```
                    field.set(obj, value);
                }
                return true;
            }
        } catch (err) {
            // Muted internally to completely isolate
high-frequency hot paths from loop crashes
        }
        return false;
    }

    // Dynamic list interceptor and memory property modifier
    function modifySubscriptionList(rawList) {
        if (rawList === null) return;
        try {
            var List = Java.use('java.util.List');
            var SubscriptionInfo =
Java.use('android.telephony.SubscriptionInfo');
            var listContainer = Java.cast(rawList, List);

            // Pass 1: Explicit Keyword Evaluation
            for (var k = 0; k < listContainer.size(); k++) {
```

```
                var item = listContainer.get(k);

                if (item === null) continue;


                var subInfoItem = Java.cast(item,
SubscriptionInfo);

                var sId = subInfoItem.getSubscriptionId();

                var rawName = subInfoItem.getDisplayName();

                var carrierName = rawName ? rawName + "" :
"unknown";

                var nameLower = carrierName.toLowerCase();


                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;

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

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

                           nameLower.includes("gxc") ||
nameLower.includes("cbrs")) {
```

```
                    privateSubId = sId;
                    privateMatched = true;
                }
            }

            // Pass 2: Older Guide Process of Elimination Safety
Net (Merged Integration)
            if (listContainer.size() === 2 && !privateMatched) {
                for (var j = 0; j < listContainer.size(); j++) {
                    var itemCheck = listContainer.get(j);
                    if (itemCheck === null) continue;
                    var subInfoCheck = Java.cast(itemCheck,
SubscriptionInfo);
                    var checkId = subInfoCheck.getSubscriptionId();
                    if (checkId !== macroSubId) {
                        privateSubId = checkId;
                        privateMatched = true;
                    }
                }
            }

            // Pass 3: Memory Modifications Injection Window
```

```
            for (var m = 0; m < listContainer.size(); m++) {
                var itemMod = listContainer.get(m);
                if (itemMod === null) continue;

                var subInfoMod = Java.cast(itemMod,
SubscriptionInfo);
                var curId = subInfoMod.getSubscriptionId();
                var curName = subInfoMod.getDisplayName() ?
subInfoMod.getDisplayName() + "" : "unknown";

                if (curId === macroSubId || curId === privateSubId)
{
                    var currentClassLoader =
subInfoMod.getClass().getClassLoader();
                    var factory =
Java.ClassFactory.get(currentClassLoader);

                    var ParcelUuidContext =
factory.use('android.os.ParcelUuid');
                    var mockGroupUuid =
ParcelUuidContext.fromString("abcd1234-5678-abcd-1234-abcdef123456"
);

                    // Overwrite private field block references
directly in memory
```

```
                    var okGroup = setFieldPure(subInfoMod,
"mGroupUuid", mockGroupUuid);



                    // Force framework repository fields to
recognize the private SIM as an opportunistic profile

                    var okOpp = true;

                    if (curId === privateSubId) {

                        okOpp = setFieldPure(subInfoMod,
"mIsOpportunistic", true);

                    }



                    // Print status confirmation tracking once per
unique execution session

                    if (trackedSubIds.indexOf(curId) === -1) {

                        if (okGroup && okOpp) {

                            console.log("[Frida] -> Dynamic
grouping and profile modifications applied to subId=" + curId + "
(" + curName + ")");

                            trackedSubIds.push(curId);

                        } else {

                            console.log("[Frida ERROR] ->
Structural memory modification failed for subId=" + curId + " (" +
curName + ").");

                            trackedSubIds.push(curId);

                        }
```

```
                    }

                }

            }

        } catch (ex) {

            if (!hasLoggedException) {

                console.log("[Frida CRITICAL ERROR]
modifySubscriptionList threw an unhandled exception: " + ex + "
(Hot-path muted)");

                hasLoggedException = true;

            }

        }

    }


    // Overload-agnostic hooker for getActiveSubscriptionInfoList

    try {

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

        var activeSubListMethods =
SubscriptionManager.getActiveSubscriptionInfoList.overloads;


        for (var i = 0; i < activeSubListMethods.length; i++) {

            activeSubListMethods[i].implementation = function () {
```

```
                var res =
this.getActiveSubscriptionInfoList.apply(this, arguments);

                modifySubscriptionList(res);

                return res;

            };

        }

        console.log("[Frida] -> Successfully hooked all overloads
of getActiveSubscriptionInfoList.");

    } catch (err) {

        console.log("[Frida ERROR] Failed to hook
getActiveSubscriptionInfoList overloads: " + err);

    }



    // Overload-agnostic hooker for
getCompleteActiveSubscriptionInfoList

    try {

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

        var completeSubListMethods =
SubscriptionManager.getCompleteActiveSubscriptionInfoList.overloads
;



        for (var j = 0; j < completeSubListMethods.length; j++) {

            completeSubListMethods[j].implementation = function ()
{
```

```
                var res =
this.getCompleteActiveSubscriptionInfoList.apply(this, arguments);
                modifySubscriptionList(res);
                return res;
            };
        }
        console.log("[Frida] -> Successfully hooked all overloads
of getCompleteActiveSubscriptionInfoList.");
    } catch (err) {}

    // Safe primitive type hook for opportunistic target matching
    try {
        var SubscriptionInfo =
Java.use('android.telephony.SubscriptionInfo');
        SubscriptionInfo.isOpportunistic.implementation = function
() {
            var subId = this.getSubscriptionId();
            if (subId === privateSubId) {
                return true;
            }
            return this.isOpportunistic();
        };
```

```
        console.log("[Frida] -> Successfully armed SubscriptionInfo
Opportunistic Forcer.");

    } catch (err) {}



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

    // ASYNCHRONOUS CONTEXT DISCOVERY ENGINE

    // Self-retrying structural back-off loop to completely prevent
early-injection faults

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

    function initializeCarrierConfigs() {

        try {

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

            var app = ActivityThread.currentApplication();



            if (!app) {

                setTimeout(initializeCarrierConfigs, 200);

                return;

            }
```

```
            var appContext = app.getApplicationContext();
            if (!appContext) {
                setTimeout(initializeCarrierConfigs, 200);
                return;
            }

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

            var carrierConfigService =
appContext.getSystemService('carrier_config');
            var subscriptionManager =
SubscriptionManager.from(appContext);

            if (!carrierConfigService || !subscriptionManager) {
                setTimeout(initializeCarrierConfigs, 200);
                return;
            }
```

```
            console.log("[Frida] Application context and managers
acquired stably. Executing initialization bootstrapper...");



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



            // Execute the initial pass parsing definitions to
update macroSubId and privateSubId values

            var bootstrapList =
subscriptionManager.getActiveSubscriptionInfoList();

            modifySubscriptionList(bootstrapList);



            var nrThresholds = Java.array('int', [-123, -118, -108,
-98]);

            var lteThresholds = Java.array('int', [-120, -115,
-110, -100]);

            var privatePlmns = Java.array('java.lang.String',
['00101', '99999', '315010']);

            var nrAvailabilities = Java.array('int', [1, 2]);



            // Private SIM Configurations (Restored to TRUE per
your laboratory topology specification)

            var privateBundle = PersistableBundle.$new();

            privateBundle.putBoolean('is_private_network_bool',
true);
```

```
            privateBundle.putBoolean('force_home_network_bool',
true);


privateBundle.putStringArray('gsm_nonroaming_networks_string_array'
, privatePlmns);


privateBundle.putIntArray('carrier_nr_availabilities_int_array',
nrAvailabilities);


privateBundle.putIntArray('5g_nr_ssrsrp_thresholds_int_array',
nrThresholds);


privateBundle.putIntArray('lte_rsrp_thresholds_int_array',
lteThresholds);

            carrierConfigManager.overrideConfig(privateSubId,
privateBundle);



            // Commercial Primary SIM Configurations

            var vzwBundle = PersistableBundle.$new();



            try {

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

                var switchPolicy =
TelephonyManager.OPP_AUTO_DATA_SWITCH_POLICY_FOR_PERFORMANCE.value;

                console.log("[Frida] Dynamic Resolution:
OPP_AUTO_DATA_SWITCH_POLICY_FOR_PERFORMANCE = " + switchPolicy);
```

```
                vzwBundle.putInt('opp_auto_data_switch_policy_int',
switchPolicy);
            } catch (constantErr) {
                vzwBundle.putInt('opp_auto_data_switch_policy_int',
3); // Fallback stability fallback policy
            }



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);
            vzwBundle.putInt('carrier_default_wfc_ims_mode_int',
2);



            // Older Guide Anchor Key: Unblocks the voice plane
tunnel during total macro network registration rejections


vzwBundle.putBoolean('no_cellular_on_wlan_opportunity_bool', true);



vzwBundle.putLong('opportunistic_network_data_switch_hysteresis_tim
e_long', 3000);
```

```
vzwBundle.putLong('opportunistic_network_data_switch_exit_hysteresis_time_long', 1000);


            // Automated Link Budget Trigger boundaries

vzwBundle.putInt('opportunistic_network_entry_threshold_rsrp_int', -102);

vzwBundle.putInt('opportunistic_network_exit_threshold_rsrp_int', -118);

vzwBundle.putInt('opportunistic_network_entry_threshold_rssnr_int', 3);

vzwBundle.putInt('opportunistic_network_exit_threshold_rssnr_int', -3);



vzwBundle.putIntArray('5g_nr_ssrsrp_thresholds_int_array', nrThresholds);

            vzwBundle.putIntArray('lte_rsrp_thresholds_int_array', lteThresholds);

            carrierConfigManager.overrideConfig(macroSubId, vzwBundle);


            console.log("[Frida] -> Carrier configurations applied successfully via background allocation.");
```

```
        } catch (configErr) {
            console.log("[Frida ERROR] Background carrier configuration application fault: " + configErr);
        }
    }

    // Launch the discovery tracker
    initializeCarrierConfigs();
});
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 12,000 to 13,500 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:

Open Device

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, write our global tracking parameters to the system databases and completely flush the telephony process memory space **before** running Frida. This forces the newly initialized baseband container to parse the tables cleanly.

In your Mac terminal window, run the following six configuration commands:

Bash

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 "settings put global wfc_ims_enabled 1"

/Users/networkfx/Downloads/platform-tools/adb shell su -c "settings put global wfc_ims_mode 2"

/Users/networkfx/Downloads/platform-tools/adb shell su -c "settings put global wfc_ims_roaming_mode 2"

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

(Allow 10 seconds for the baseband and Shannon telephony processes to re-initialize before proceeding).

Phase 5: Start the Hooking Process

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

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

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):

```
None
   . . . .   Connected to Pixel 7 (id=33020DLH200034)
Attaching...
[Frida] Hook successfully attached. Initializing Android 17
Auto-Switching Engine...
[Frida] -> Successfully hooked all overloads of
getActiveSubscriptionInfoList.
[Frida] -> Successfully hooked all overloads of
getCompleteActiveSubscriptionInfoList.
[Frida] -> Successfully armed SubscriptionInfo Opportunistic
Forcer.
[Frida] Application context and managers acquired stably. Executing
initialization bootstrapper...
[Frida] -> Dynamic grouping and profile modifications applied to
subId=1 (Verizon )
[Frida] -> Dynamic grouping and profile modifications applied to
subId=4 (GXC-PRV8)
```

```
[Frida] -> Carrier configurations applied successfully via
background allocation.
[Pixel 7::PID::9211 ]->
```

Phase 6: Verify Active Telemetry Cache

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

1. Confirm Active Carrier Configuration Overrides

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
"gsm_nonroaming_networks_string_array|carrier_nr_availabilities_int
_array|is_private_network_bool"
```

2. Live Telephony Database Verification (Android 17 Mapping)

Verify that the defaultDataSubId and activeDataSubId tokens are correctly assigned, while isolating your local memory configurations:

```
None
/Users/networkfx/Downloads/platform-tools/adb shell su -c "dumpsys
isub" | grep -iE "subid|opportunistic|group"
```

3. Live Telemetry Target Logging Stream

To stream active baseband tracking changes and signal evaluations directly to your host window, run this logging filter:

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

4. Verify Opportunistic Status & ONS Core Allocation

Confirm that the OpportunisticNetworkService core daemon actively maps the parameters within the execution tree:

```
None
/Users/networkfx/Downloads/platform-tools/adb shell su -c "dumpsys
ions"
```

Phase 7: Check Signal Metrics Against Threshold Parameters

To evaluate whether your active RF environments fit within the handoff boundaries configured in your script, run this query to pull the real-time rsrp and rssnr levels from the baseband power plane:

```
None
/Users/networkfx/Downloads/platform-tools/adb shell su -c "dumpsys
telephony.registry" | grep -i "mSignalStrength="
```

Phase 8: Simulate the Signal Handoff

Crucial Hardware Transceiver Workaround:

On Google Pixel Tensor architectures, toggling **Mobile Radio Power** to **OFF** inside the phone's engineering menu (*#*#4636#*#*) pulls 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 method below:

1. Turn off **Wi-Fi** on your Pixel device to isolate the active transceiver paths.
2. On your physical Pixel handset, navigate to **Settings** -> **Network & internet** -> **SIMs** -> Tap your **Commercial SIM (MNO)**.
3. Scroll down and toggle the **Automatically select network** switch to **OFF**.
4. Let the scanning loop complete, then select an incompatible carrier or a forbidden network from the list.
5. The MNO line will instantly drop to an OUT_OF_SERVICE state due to PLMN rejection. Because of the mIsOpportunistic=true memory override injected in Phase 3, **mIsIwlanPreferred will instantly flip to true**, routing live voice channels into your private data network backhaul tunnel (accessNetworkTechnology=18(IWLAN)).

Open Device Alliance

Phase 9: 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 Verizon'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, Verizon's secure tunnel will fail to attach silently.

1. DNS Resolution Inside the Private Core

When Verizon loses its macro signal, it immediately asks the active data connection (your Private SIM, Slot 1) 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 or 1.1.1.1.** The phone will try to resolve Verizon'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 cannot resolve public internet domains, the backup calling tunnel dies before it even starts (UnknownHostException).

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. Verizon 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 security 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 to keep latent paths from closing.

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. Verizon'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 + Verizon's IPsec tunnel will exceed the network limits. Packets will fragment, get dropped by your firewall, and the call or tunnel 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

Open Device Alliance

- **SIM Slot Dependency:** SIM slot numbers inside your logs can shift depending on whether you are utilizing a physical SIM array or an eSIM profile; modify your target privateSubId variables inside the code to reflect your layout if your slot maps alter.
- **Volatile Memory Lifespan Rules:** Remember that all memory overrides injected via Frida operate strictly in-memory. Soft reboots, phone app crashes, or flushing the phone container with a killall command will drop the session, requiring you to re-run your dynamic Phase 5 instrumentation line.
- **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.

Open Device Alliance