

raelize

Breaking SoC Security by Glitching OTP Data Transfers

Niek Timmers
niek@raelize.com
[@tieknimmers](https://twitter.com/tieknimmers)

Cristofaro Mune
cristofaro@raelize.com
[@pulsoid](https://twitter.com/pulsoid)

Introduction.

raelize

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,...)

SoC security

- Security features are becoming increasingly relevant
 - Also present on low-cost SoCs
- OEM use cases (examples):
 - Sensitive/critical devices
 - Device-based services
 - IP protection

ESP32-S2 SoC

32-bit MCU & 2.4 GHz Wi-Fi

Common Specifications

- High-performance 240 MHz single-core CPU
- Ultra-low-power performance: fine-grained clock
- Security features: eFuse, flash encryption, secure boot
- Peripherals include 43 GPIOs, 1 full-speed USB interface, ADC, DAC, touch sensor, temperature sensor
- Availability of common cloud connectivity agent



ESPRESSIF



MICROCHIP

SAM L10/L11 Family

Ultra Low-Power, 32-bit Cortex-M23 MCUs with TrustZone, Crypto, and Enhanced PTC

Features

13. SAM L11 Security Features.....	5
13.1. Features.....	5
13.2. ARM TrustZone Technology for ARMv8-M.....	5
13.3. Crypto Acceleration.....	6
13.4. True Random Number Generator (TRNG).....	7
13.5. Secure Boot.....	7
13.6. Secure Pin Multiplexing on SERCOM.....	7

Secure boot on embedded Sitara™ processors

AN12312

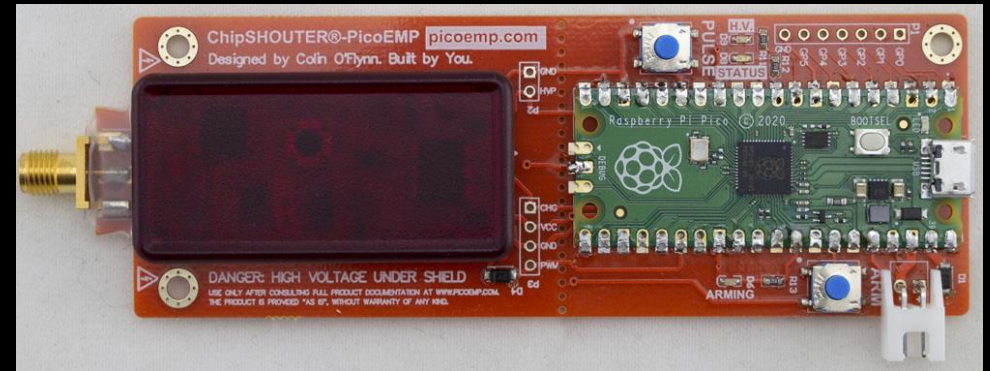
Secure Boot on i.MX 8 and i.MX 8X Families using AHB

Rev. 0 — May 2019

Application Note

Fault Injection (FI)

- Concept now familiar to security researchers
- More accessible **tooling**:
 - Hi [NewAE!](#)
- **Advanced** techniques presented at public security conferences:
 - Hi [hardwear.io!](#)
- Still missing (but improving!):
 - Field systematization
 - Integration of security, academia, industry approach/results



[ChipSHOUTER-PicoEMP](#)

Goals

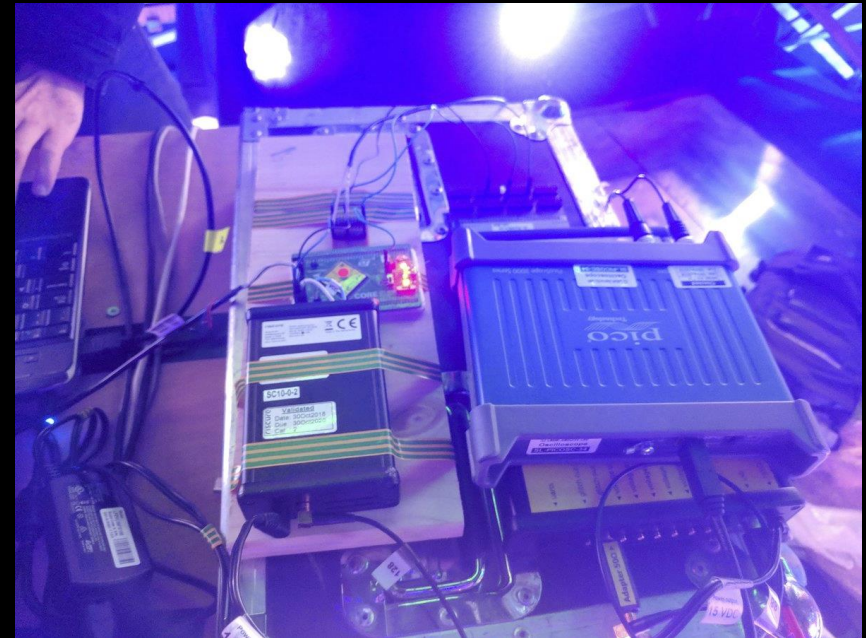
- Show how SoC **security** depends on OTP data
- Present advanced FI attacks targeting **OTP** data **transfers**
- Share our results (**EMFI** on ESP32):
 - ...
 - Be patient! 😊
- Discuss possible **countermeasures**

Contribute to the field

Context.

In 2019, we...

- Presented Secure Boot talk at BlueHat IL (Feb 2019)
- Bypassed **encrypted** secure boot
 - We love live demos!
- **Described** an FI attack targeting OTP transfers:
 - Generic. Not specific to a SoC.



ESP32

- Popular IoT system-on-chip (SoC)
 - 32-bit Xtensa LX6 CPU
 - **Internal** and **external** memories
 - Many peripherals and other interfaces
- **Hardware**-backed security features:
 - Secure Boot
 - Flash Encryption
 - JTAG disable



Lots of interest!

First FI public attacks

- **Limited Results** performed multiple FI attacks towards ESP32
 - **VCC** glitching
- Targeted AES Crypto core (driver)
- Secure Boot bypass
- Dump of OTP protected keys:
 - Secure Boot and Flash Encryption keys
 - Targeted OTP data **transfer**

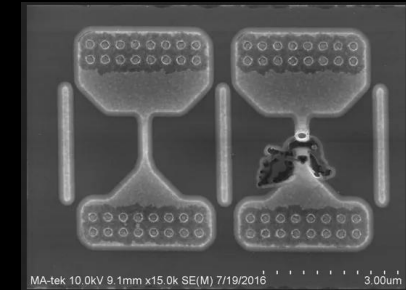
Our FI attacks (2020)

- Performed with **EM FI**
- Secure Boot bypass:
 - Achieved PC control with FI (via instruction modification)
- Flash Encryption bypass
- Secure Boot + Flash Encryption bypass:
 - Leveraged SRAM data persistence + PC control with FI (via instruction modification)
 - Single glitch attack
 - No knowledge of encryption key required

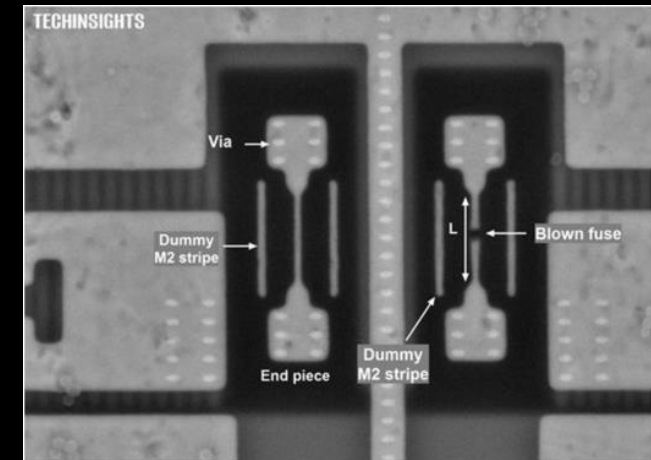
One Time Programmable (OTP) Memory.

OTP: Memory Cell

- Bits can be set only once and cannot revert
 - Physical modification of the cell
- No observable difference after programming with secure variants
 - E.g. antifuse
- Typically:
 - A programmed cell represents a '1'
 - OTP contains data (e.g. configuration) or secrets (e.g. keys)



Programmed poly Efuse ([Source](#))



Programmed Efuse ([Source](#))

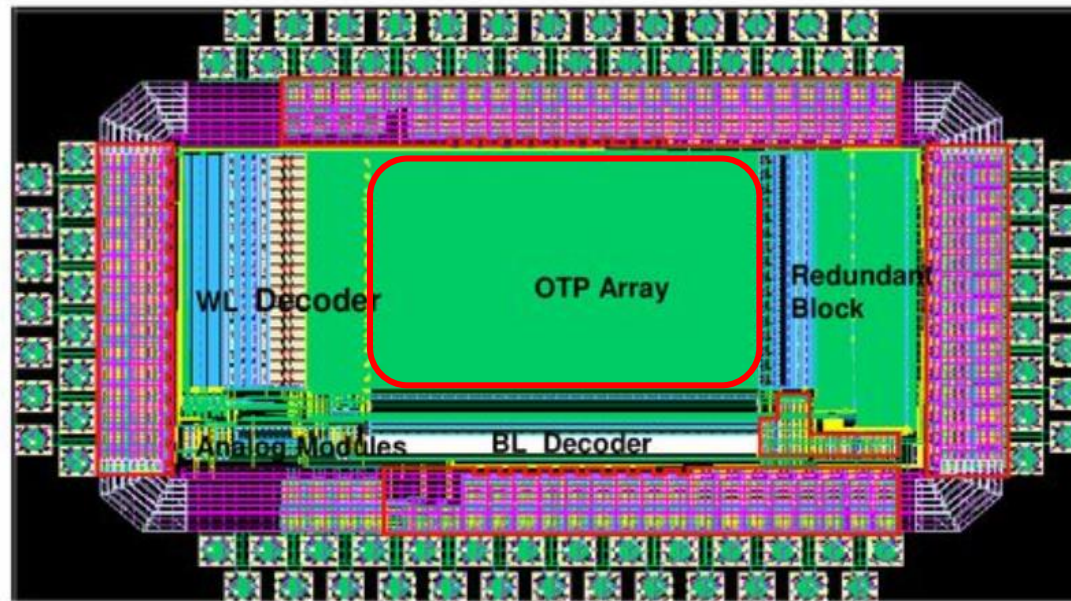


Programmed poly antifuse ([Source](#))

OTP: Cell Array



512Kb OTP Specifications



Process: Silterra110nm 1.5V/3.3V Ultra Low Leakage Logic Process

Density: 16Kx32bits

Bit Program Operation

32bits Read Operation

Operation Temperature: -40°C ~ 125°C

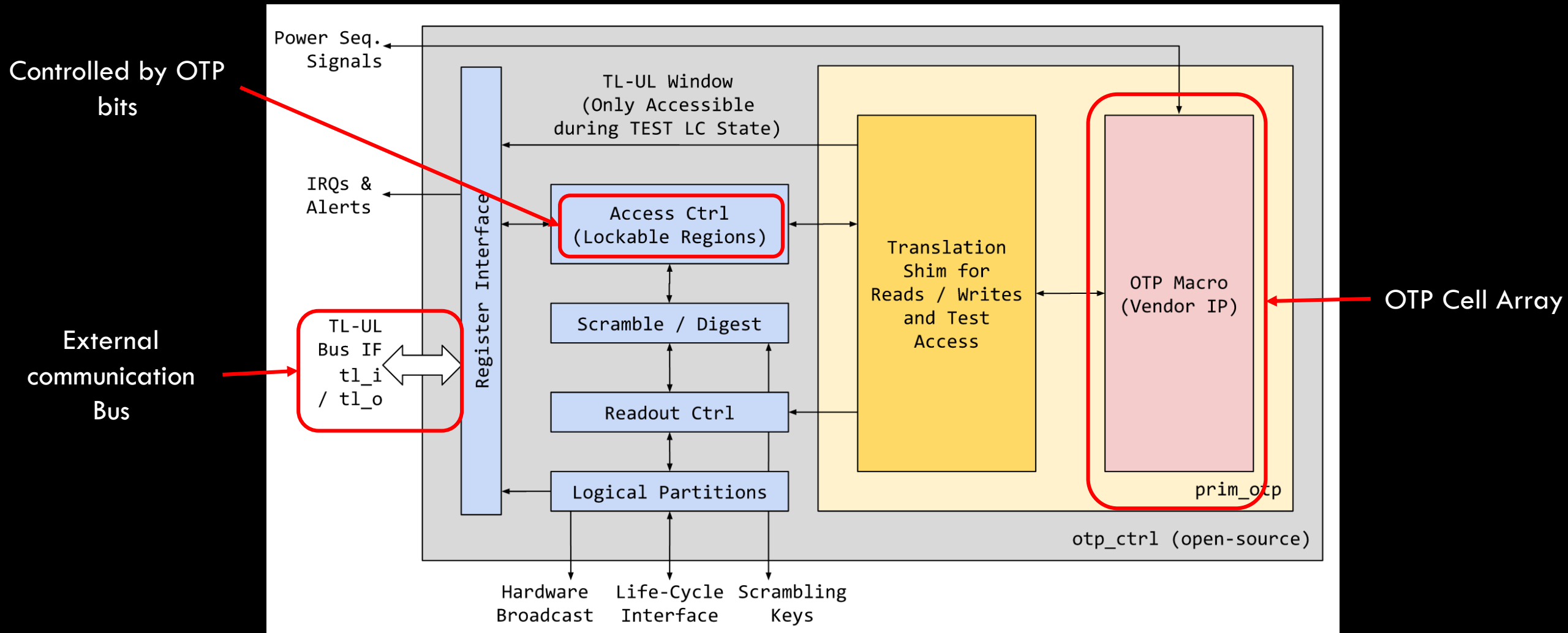
IP Size: 1340um * 670um

Example of OTP Cell Array organization ([Source](#))

OTP: Memory organization

- Typically divided in **regions**
- Single Bit access/control:
 - For security features configuration (e.g. JTAG disable)
- Multiple bits access/control:
 - For bulk data (e.g. keys)
- Regions (or subsets) can be **read/write protected**:
 - Dedicated OTP bits
- Write protection needed for preserving '0' values:
 - i.e. Region is "locked"

OTP controller



Example: ESP32

#	field_name	efuse_block	bit_start	bit_count
1	WR_DIS_FLASH_CRYPT_CNT	EFUSE_BLK0	2	1
2	WR_DIS_BLK1	EFUSE_BLK0	7	1
3	WR_DIS_BLK2	EFUSE_BLK0	8	1
4	WR_DIS_BLK3	EFUSE_BLK0	9	1
5	RD_DIS_BLK1	EFUSE_BLK0	16	1
6	RD_DIS_BLK2	EFUSE_BLK0	17	1
7	RD_DIS_BLK3	EFUSE_BLK0	18	1
8	FLASH_CRYPT_CNT	EFUSE_BLK0	20	7
26	ENCRYPT_CONFIG	EFUSE_BLK0	188	4
27	CONSOLE_DEBUG_DISABLE	EFUSE_BLK0	194	1
28	ABS_DONE_0	EFUSE_BLK0	196	1
29	DISABLE_JTAG	EFUSE_BLK0	198	1
30	DISABLE_DL_ENCRYPT	EFUSE_BLK0	199	1
31	DISABLE_DL_DECRYPT	EFUSE_BLK0	200	1
32	DISABLE_DL_CACHE	EFUSE_BLK0	201	1
33	ENCRYPT_FLASH_KEY	EFUSE_BLK1	0	256
34	SECURE_BOOT_KEY	EFUSE_BLK2	0	256

R/W protection

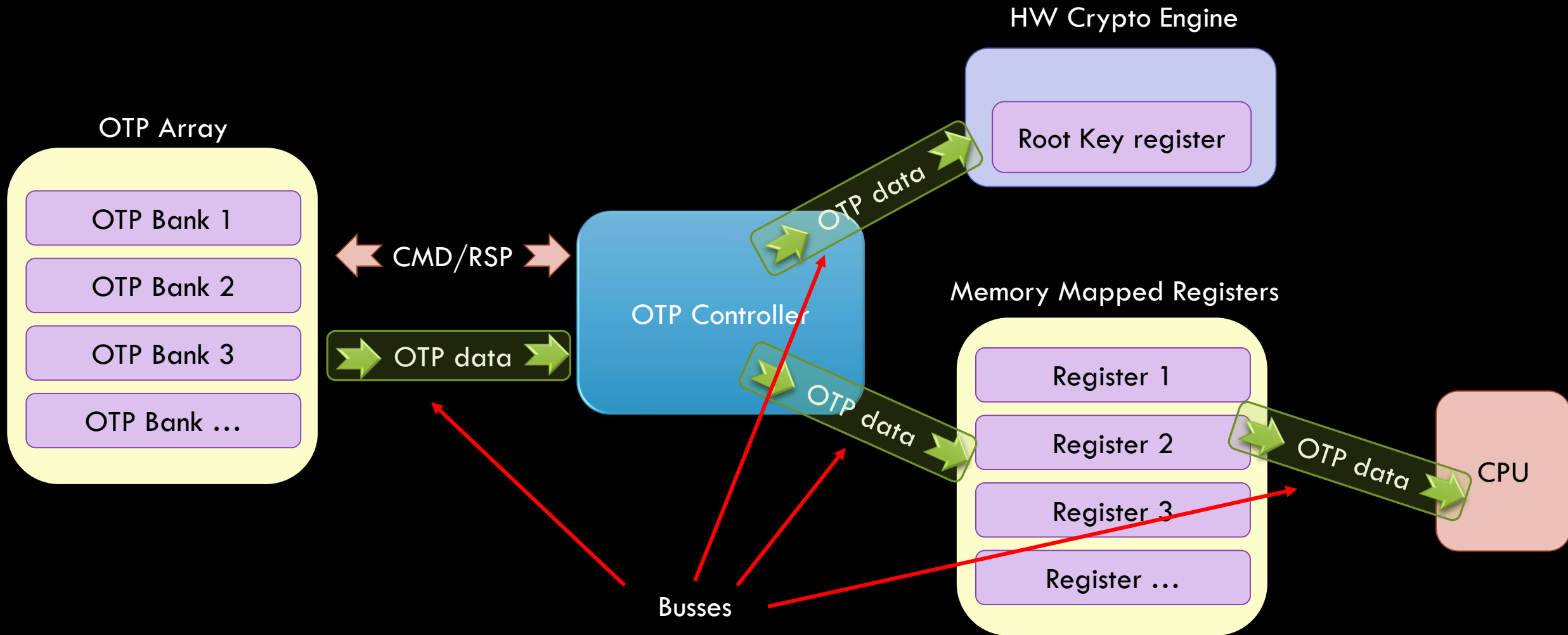
Configuration

Keys

OTP data transfers

- Reading from OTP may be slow
- Data is usually transferred to “shadow registers”
 - To configure HW logic
 - Faster access from SW
- Some shadow registers may be memory mapped
 - For consumption by SW
 - E.g. Secure Boot enable bits (used by ROM code)
- Others may be only accessible by HW logic:
 - E.g. Unique device keys for KDF/Key Ladder operations

SoC HW Initialization



FI opportunities

- OTP data transferred at boot time
 - At multiple **times**
 - Across multiple **busses** (with different speed)
- A successful **glitch** on ANY of those busses may:
 - Modify transferred value(s)
 - Leave OTP copy unaffected
- And **bypass**
 - SW-based security features (CPU reads incorrect value)
 - HW-based security features (HW logic incorrectly configured)

Does it really work!?

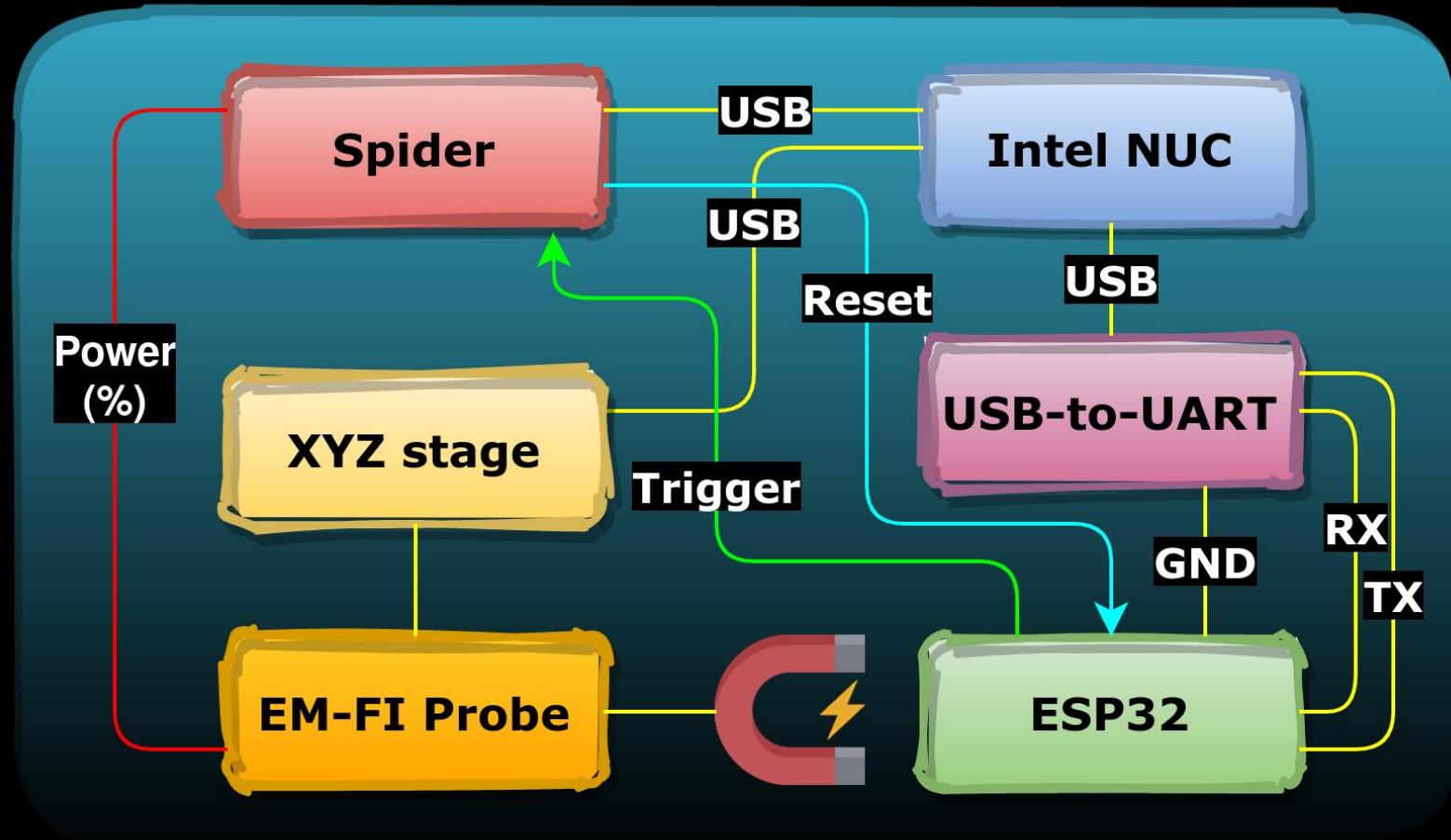
FI Characterization.

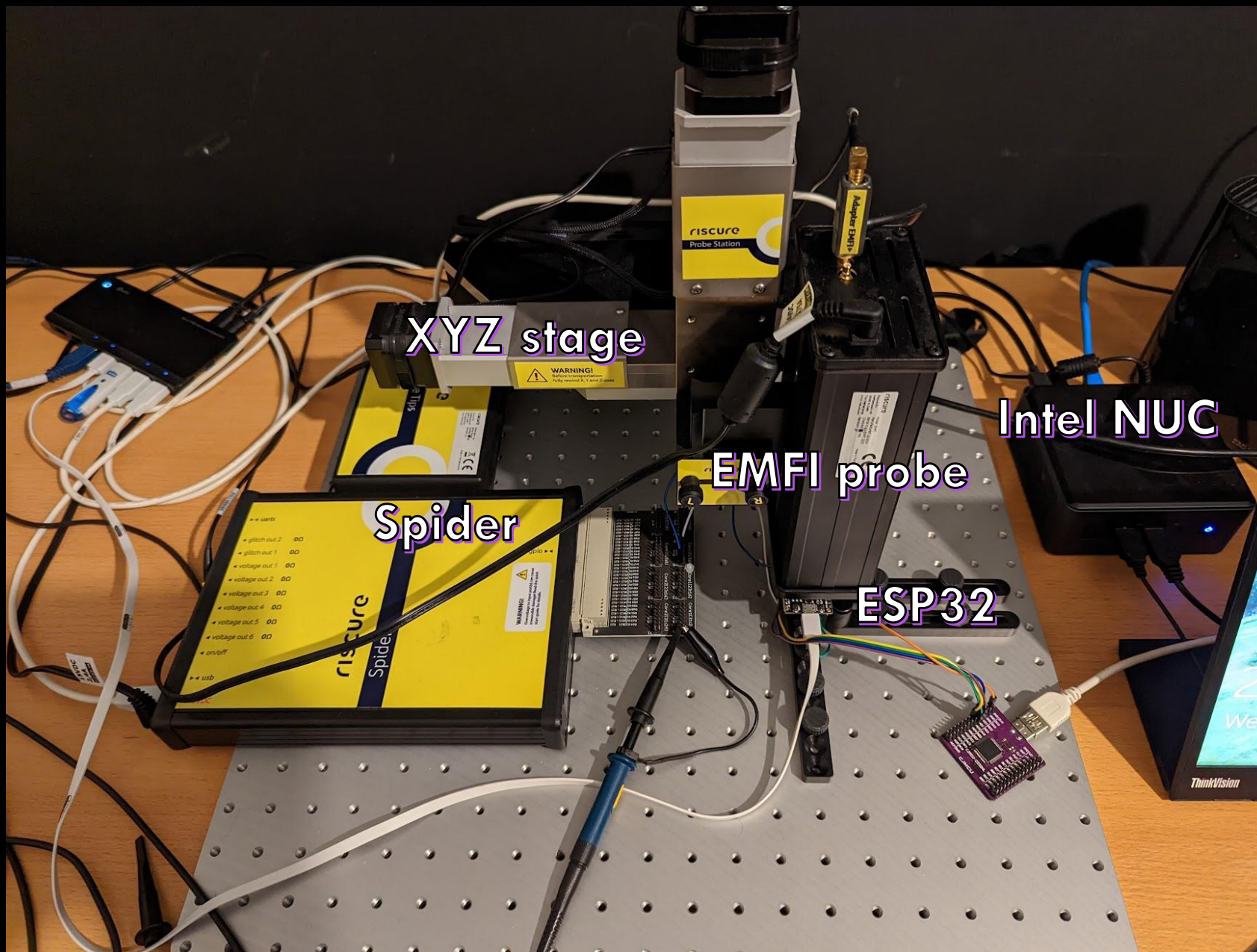
Our target

- Module: ESP32-WROOM-32
- SoC: ESP32 (**first** series)
 - D0WDQ6
 - No specific FI countermeasures
- We have **not** tested newer ESP32 SoC:
 - E.g. ESP32-S2, ESP32-V2,..
 - FI countermeasures are present



EMFI Setup overview





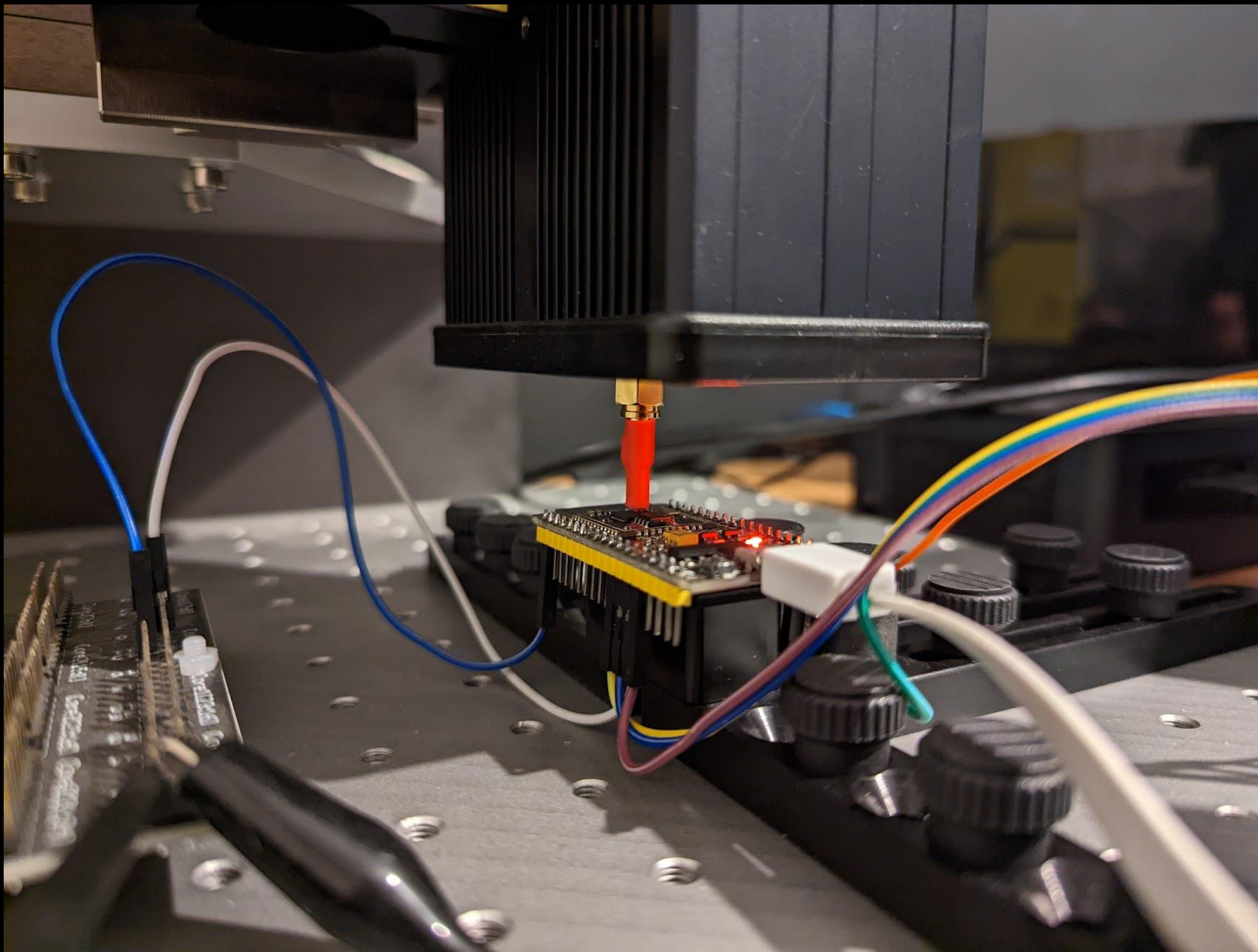
XYZ stage

Intel NUC

EMFI probe

Spider

ESP32



Experiment Setup (1)

- We set the following fuses:
 - ABS_DONE0 for enabling Secure Boot
 - SECURE_BOOT_KEY (BLK2) for setting the Secure Boot key
 - RD_DIS_BLK2 for enabling read protection of Secure Boot key
- We program a **valid** bootloader that prints the fuses

Experiment Setup (2)

- We:
 - scan the entire chip surface
 - Glitch at boot time
 - before first ROM printout
- Goal:
 - Find a vulnerable location
 - **Change** any of the fuse bits in Block 0 (Configuration) or Block 2 (Secure Boot Key)

Results classification:

```
block0 = b'Block 0: 00020100a3d2c1e00001c8c90000a00000000132000000000000014'
block0_glitched = b'Block 0: [a-z0-9]{56}'

block2 = b'Block 2: 0000000000000000000000000000000000000000000000000000000000000000'
block2_glitched = b'Block 2: [a-z0-9]{64}'

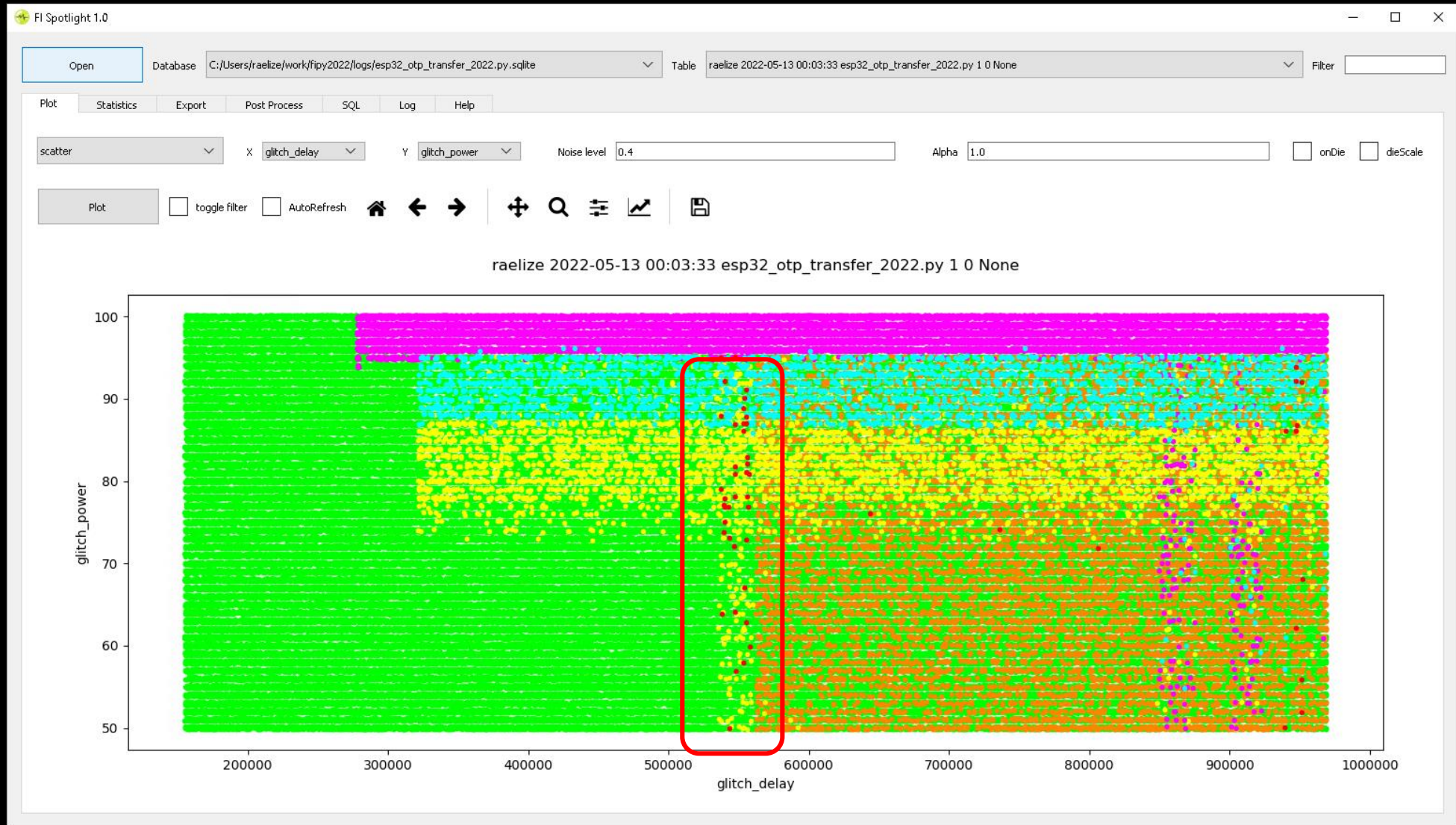
if spider_timeout:
    color = ResultColor.PINK
elif block0 in response and block2 in response:
    if b'ets Jun  8 2016 00:22:57\r\n\r\nrst:0x1 (POWERON_RESET)' in response:
        if b'cmd len' in response:
            color = ResultColor.CYAN
        else:
            color = ResultColor.GREEN
    elif b'ets Jun  8 2016 00:22:57\r\n\r\nrst:0x6 (SDIO_RESET)' in response:
        color = ResultColor.MAGENTA
    elif b'ets Jun  8 2016 00:22:57\r\n\r\nrst:0x10 (RTCWDT_RTC_RESET)' in response:
        color = ResultColor.ORANGE
    elif b'ets Jun  8 2016 00:22:57\r\n\r\nrst:0x4 (OWDT_RESET)' in response:
        color = ResultColor.CYAN
    elif b'ets Jun  8 2016 00:22:57\r\n\r\nrst:0x13 (TG0WDT_SYS_RESET)' in response:
        color = ResultColor.GREY
    elif b'ets Jun  8 2016 00:22:57\r\n\r\nrst:0x13 (TG1WDT_SYS_RESET)' in response:
        color = ResultColor.GREY
elif b'DOWNLOAD_BOOT' in response:
    color = ResultColor.WHITE
elif response.startswith(b'ets Jun  8 2016 00:22:57\r\n\r\nrst:0x1 (POWERON_RESET)') and re.search(block0_glitched, response) and re.search(block2_glitched, response):
    color = ResultColor.RED
else:
    color = ResultColor.YELLOW
```

Performed **during** experiment

Results: XY scan

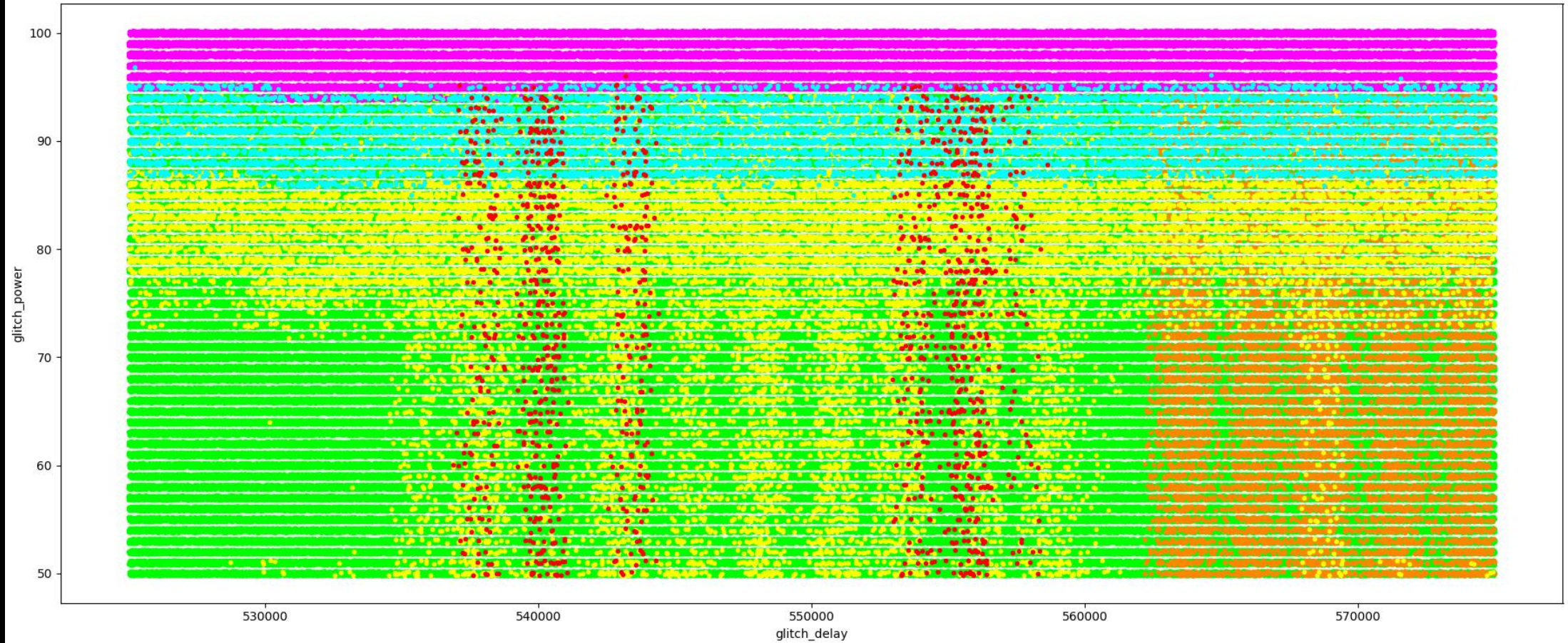


We have successful glitches!



Let's zoom in...

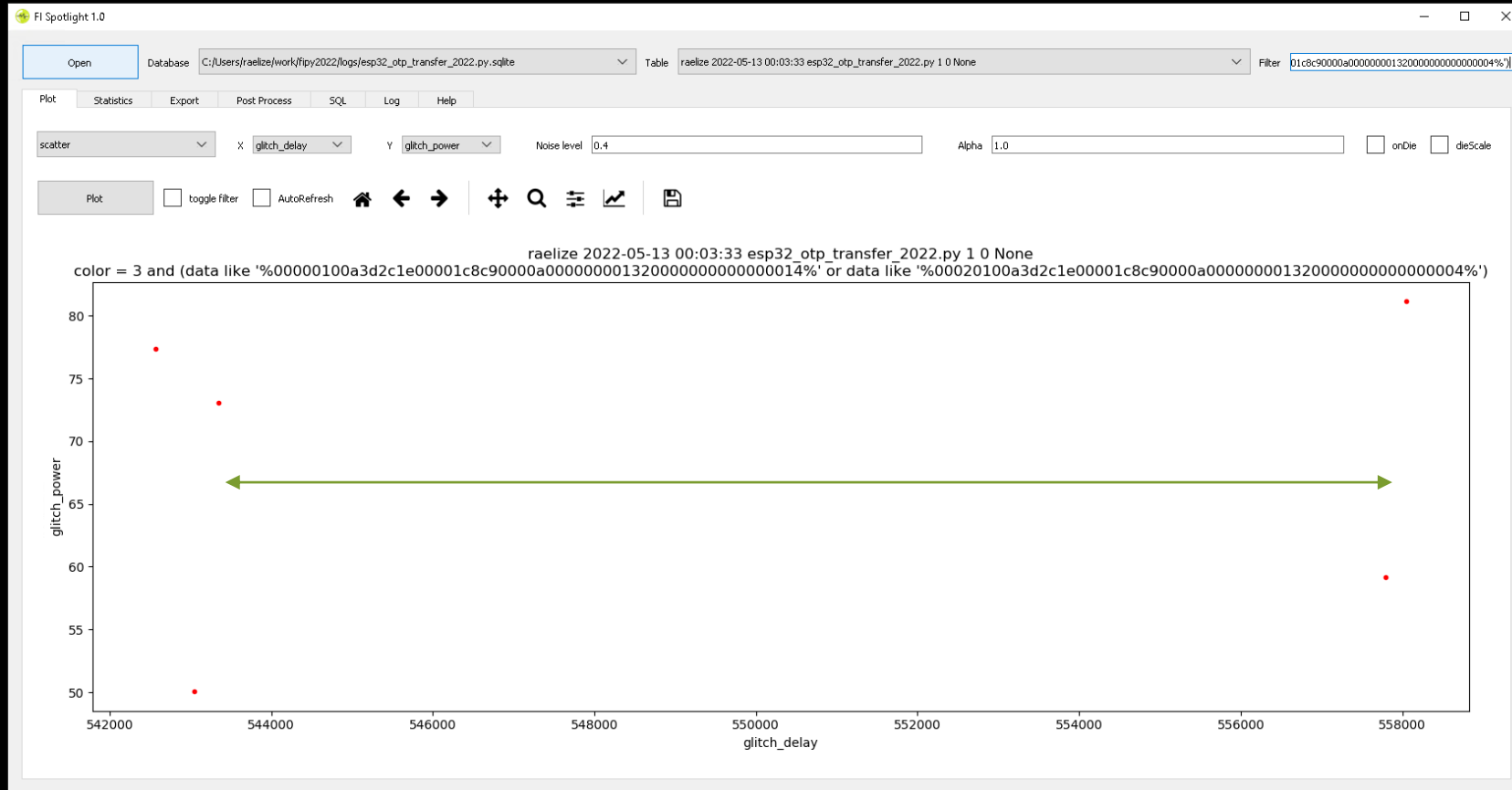
raelize 2022-05-13 09:35:19 esp32_otp_transfer_2022.py 1 0 None



Results analysis

```
Raelize.....Block 0: 00020100a3d2c1e00001c8c90000a00000000132000000000000014..
Raelize.....Block 0: 00020100a3d2c1e00001c8890000a00000000132000000000000014..
Raelize.....Block 0: 0002010083d2c1e00001c8c90000a00000000132000000000000014..
Raelize.....Block 0: 00020100a3d2c1e00001c8c90000a00000000032000000000000014..
Raelize.....Block 0: 00020100a3d2c1e00001c8c90000800000000132000000000000014..
Raelize.....Block 0: 00020100a3d2c1e00001c8490000a00000000132000000000000014..
Raelize.....Block 0: 00000100a3d2c1e00001c8c90000a00000000132000000000000014..
Raelize.....Block 0: 00020100a3d2c1e00001c8c90000a00000000132000000000000004..
Raelize.....Block 0: 00020100a3d20e00001c8c90000a00000000132000000000000014..
```

Timing dependency



Block 0: 00020100a3d2c1e00001c8c90000a00000000132000000000000014..

Block 0: 00000100a3d2c1e00001c8c90000a00000000132000000000000014..

Block 0: 00020100a3d2c1e00001c8c90000a00000000132000000000000004..

Observations

- Successful glitches in a narrow time region:
 - Between 535us and 560us from reset
 - Group at specific timing

- Single bit glitches of Block 0 (!):
 - Mostly “1” turned to “0”

We can **disable** security features!

- Strong timing dependency of affected bits

We can **select** which one

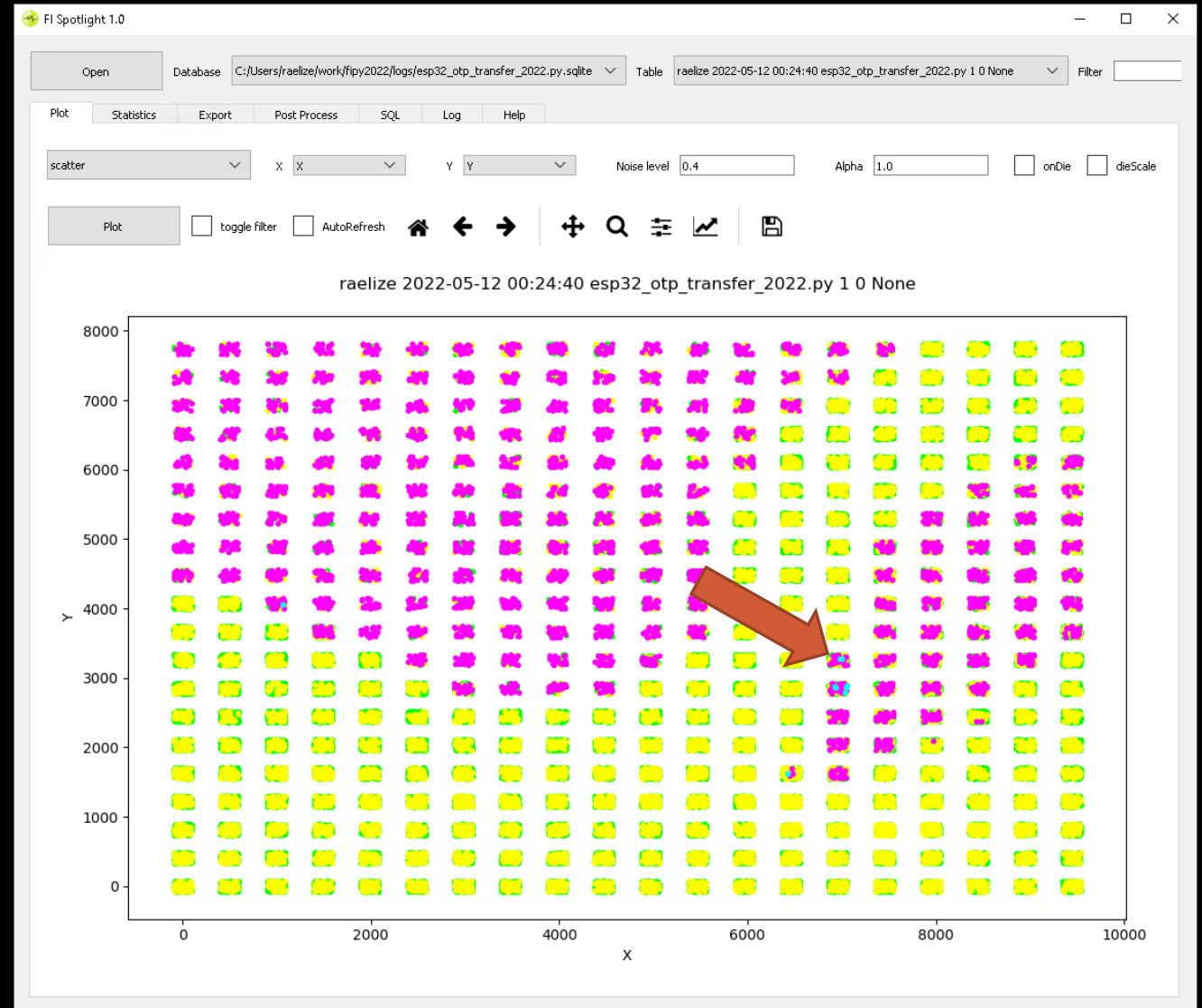
OTP bits “shooting”



([Source](#))

Reducing parameters space

- We fix probe at a specific location:
 - Removes location from parameters
- Selection criteria:
 - % of successful glitches
- Applies for all the next experiments



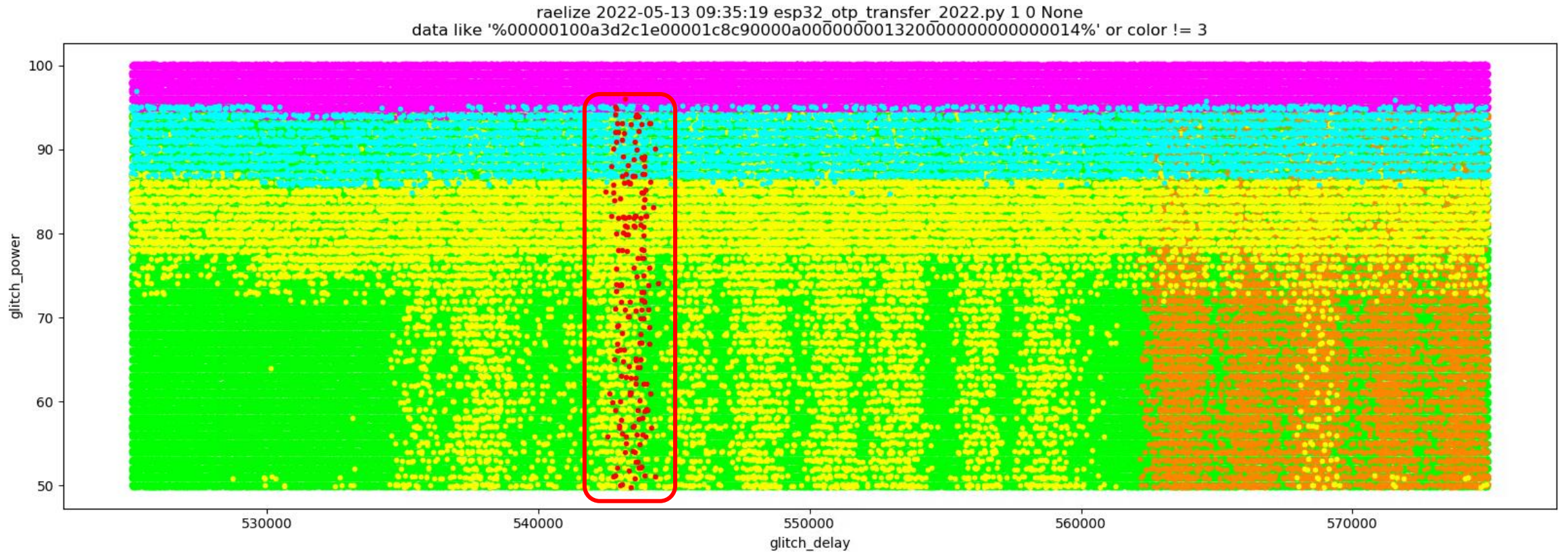
Let's bypass security features!

1. Reading Secure Boot Key.

Setup

- We set the following fuses:
 - ABS_DONE0 for enabling Secure Boot (SB)
 - SECURE_BOOT_KEY (BLK2) for setting the SB key
 - RD_DIS_BLK2 for enabling read protection of SB key
- We program a **valid** bootloader that prints the fuses
 - SB key is unreadable due to read protection
- Goal:
 - Extract SB key
 - i.e. we need to change 1 fuse bit (RD_DIS_BLK2) during the transfer

Successful glitches!



Results analysis

```
Block 0: 00020100a3d2c1e00001c8c90000a000000001320000000000000014..
```

[illegible][illegible]

RD_DIS_BLK2 changed!

Block 0: 00000100a3d2c1e00001c8c90000a00000001320000000000000014..

Success rate: Focused

- We also fix timing and power:
 - We choose one successful experiment randomly
- All parameters are now fixed
- Success rate: $\sim 1\%$:
 - Full Secure Boot Key obtained
 - No **bruteforce** needed

ColorNumber	Colors	counts	percentage
3	Red	190	1,03848
4	Green	14522	79,3725
5	Yellow	1311	7,1655
7	Magenta	1915	10,4668
8	Cyan	358	1,95671

We read SB key within **minutes**...


Let's double down!

2. SB bypass + Reading SB Key.

The idea

- We can glitch specific Block 0 bits by attack timing
- Secure Boot and its key protected by “distant” OTP bits

#	field_name	efuse_block	bit_start	bit_count
1	WR_DIS_FLASH_CRYPT_CNT	EFUSE_BLK0	2	1
2	WR_DIS_BLK1	EFUSE_BLK0	7	1
3	WR_DIS_BLK2	EFUSE_BLK0	8	1
4	WR_DIS_BLK3	EFUSE_BLK0	9	1
5	RD_DIS_BLK1	EFUSE_BLK0	16	1
6	RD_DIS_BLK2	EFUSE_BLK0	17	1
28	ABS_DONE_0	EFUSE_BLK0	196	1
29	DISABLE_JTAG	EFUSE_BLK0	198	1



Can we glitch both?

Double glitch: Setup (1)

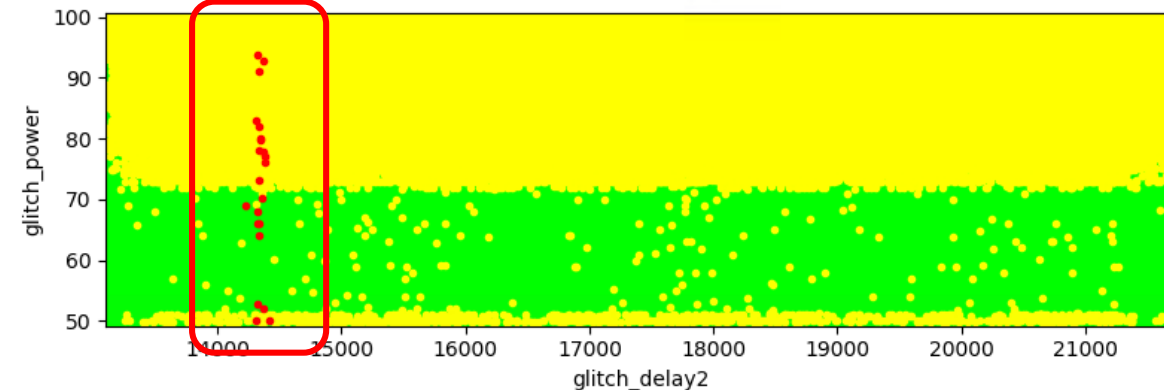
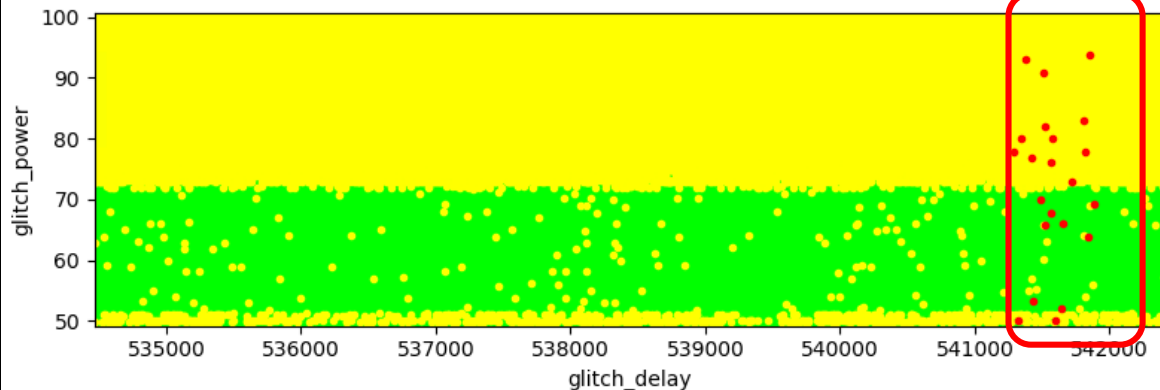
- We set the following fuses:
 - ABS_DONE0 for enabling Secure Boot (SB)
 - SECURE_BOOT_KEY (BLK2) for setting the SB key
 - RD_DIS_BLK2 for enabling read protection of SB key
- We program an **invalid** bootloader that prints the fuses:
 - Not executed unless Secure Boot is bypassed
- Goal:
 - Bypass SB and print SB key
 - i.e. we need to change 2 fuse bits (RD_DIS_BLK2, ABS_DONE0) during the transfer

Double glitch: Setup (2)

- Glitch 1 aimed at RD_DIS_BLK2
 - Timing (glitch_delay) still randomized
 - We re-use approximate timing from previous experiment
- Glitch 2 aimed at ABS_DONE0
 - Timing (glitch_delay2) randomized
 - Counted from glitch 1 timing
 - We know the boundaries where Block 0 is affected

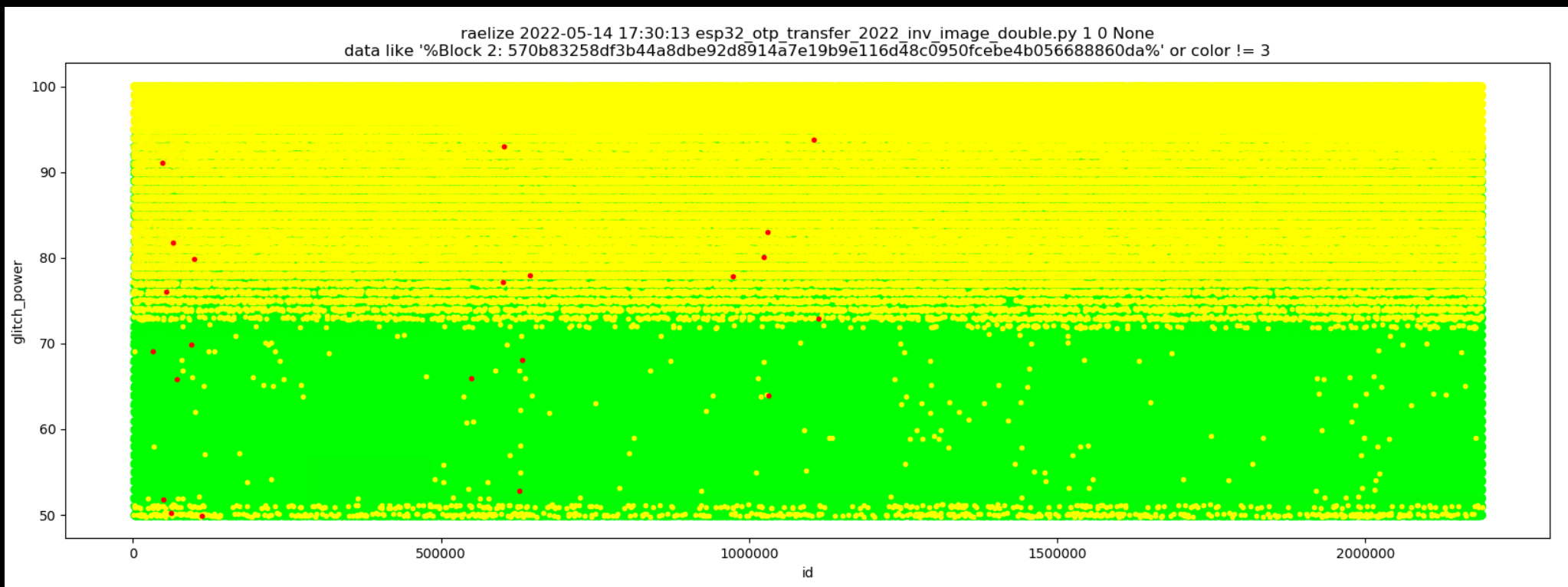
Successful glitches!

raelize 2022-05-14 17:30:13 esp32_otp_transfer_2022_inv_image_double.py 1 0 None
data like '%Block 2: 570b83258df3b44a8dbe92d8914a7e19b9e116d48c0950fceb4b056688860da%' or color != 3



- Glitch 1: similar timing as previous experiment
- Glitch 2: timing is basically constant:
 - Counting from glitch 1 drastically reduces jitter

Something interesting...



- Successful glitches occur in groups:
 - Experiment ID is sequential
 - Reason unknown. Guesses: Room temperature effects? SoC status?

Success rate: Focused

- We fix timing and power:
 - We choose one successful experiment randomly
- Success rate: $\sim .05\%$:
 - Full Secure Boot Key obtained
 - No **bruteforce** needed
 - Secure Boot bypassed

ColorNumber	Colors	counts	percentage
3	Red	22	0,050555
4	Green	35659	81,9427
5	Yellow	7836	18,0068

We bypass SB and read SB key within **1 hour**...

Bypassing Secure Boot and Flash Encryption?



We tried. No success...

...but we got an **idea!**

3. JTAG glitching.

Experiment Setup (1)

- We set the following fuses:
 - ABS_DONE0 for enabling Secure Boot
 - SECURE_BOOT_KEY (BLK2) for setting the Secure Boot key
 - RD_DIS_BLK2 for enabling read protection of Secure Boot key
 - FLASH_CRYPT_CNT for enabling Flash Encryption
 - FLASH_ENCRYPTION_KEY (BLK2) for setting the Flash Encryption key
 - RD_DIS_BLK1 for enabling read protection of Flash Encryption key
 - DISABLE_JTAG for disabling the JTAG interface
- We program a **valid** bootloader that prints the fuses

Experiment Setup (2)

- Goal:
 - Open JTAG interface
 - i.e. we need to change 1 fuse bit (DISABLE_JTAG) during the transfer
 - Use JTAG to extract **plaintext** firmware
 - ...and bypass Flash Encryption
- Once opened, JTAG can be also used to execute arbitrary code:
 - i.e. **bypass** Secure Boot

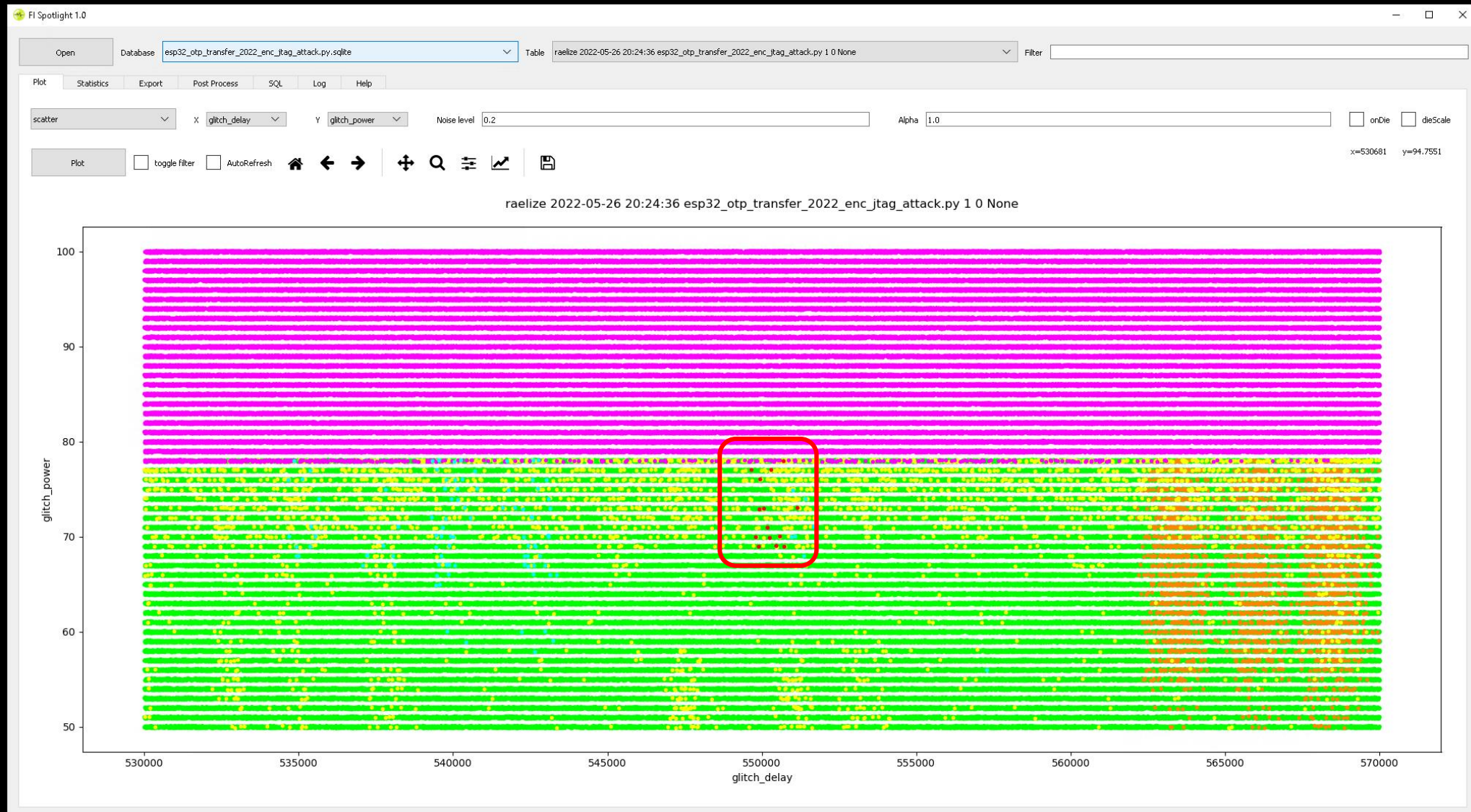
Results classification:

```
block0 = b'Block 0: 00130180a3d2c1e00001c8c90000a00000000132f000000000000054\r\n'
block0_glitched = b'Block 0: [a-z0-9]{56}'

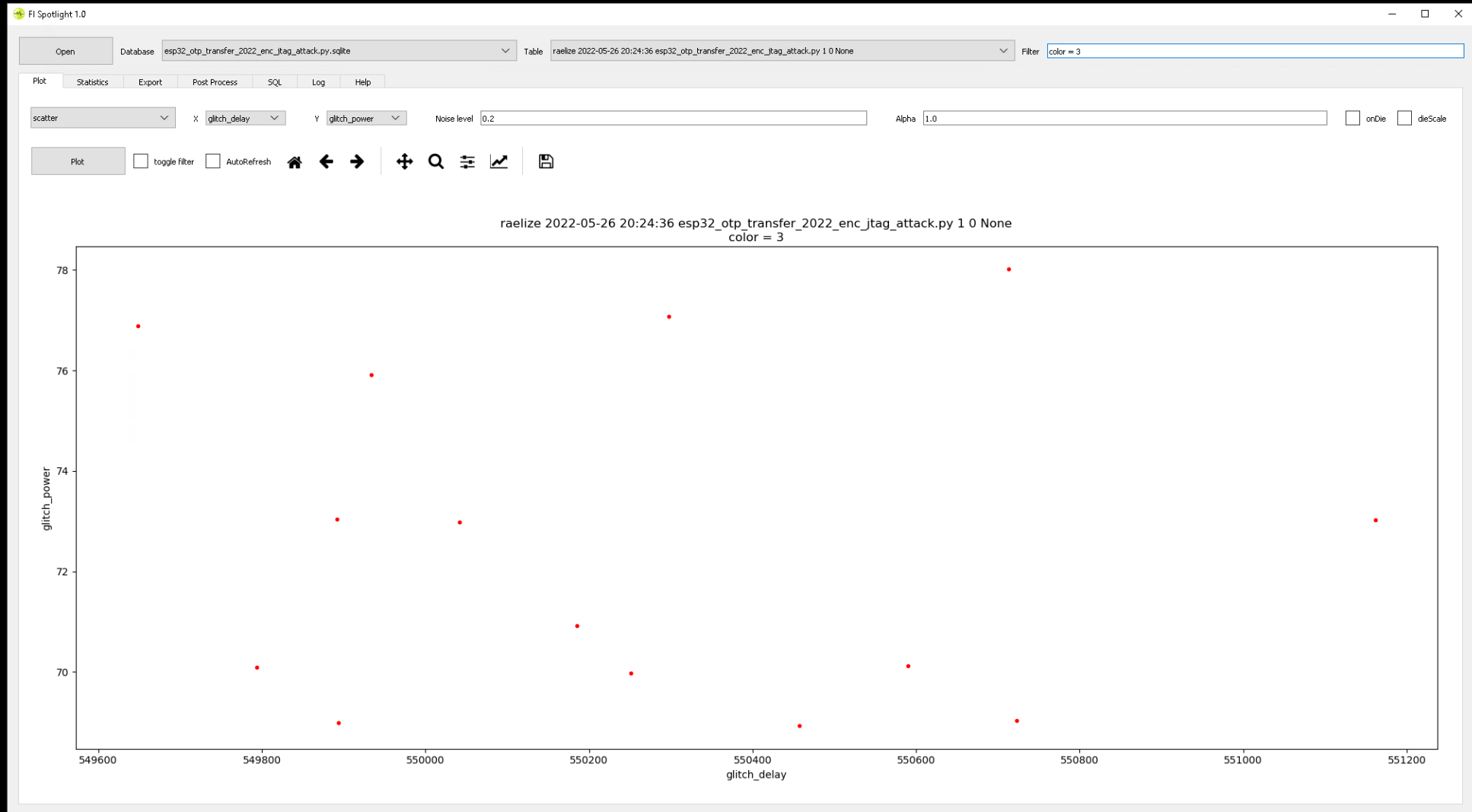
if spider_timeout:
    color = ResultColor.PINK
elif re.search(b'^ets Jun  8 2016 00:22:57\r\n\r\nrst:0x1 \(\s*POWERON_RESET\s*\)', response) and re.search(block0, response):
    color = ResultColor.GREEN
elif re.search(b'^ets Jun  8 2016 00:22:57\r\n\r\nrst:0x6 \(\s*SDIO_RESET\s*\)', response):
    color = ResultColor.MAGENTA
elif re.search(b'^ets Jun  8 2016 00:22:57\r\n\r\nrst:0x10 \(\s*RTCWDT_RTC_RESET\s*\)', response):
    color = ResultColor.ORANGE
elif response.startswith(b'ets Jun  8 2016 00:22:57\r\n\r\nrst:0x1 (POWERON_RESET)') and re.search(block0_glitched, response):
    if b'00000014' in response:
        color = ResultColor.RED
    else:
        color = ResultColor.CYAN
else:
    color = ResultColor.YELLOW
```

Check if JTAG_DISABLE is unset

Successful glitches!



Timing (Successful glitches)



Success rate: Focused

- We fix timing and power:
 - We choose one successful experiment randomly
- Success rate: $\sim 0.05\%$:
 - JTAG open \rightarrow we read Flash in plaintext
 - Flash Encryption bypassed
 - Secure Boot bypassed

ColorNumber	Colors	counts	percentage
3	Red	12	0,118977
4	Green	5535	54,878
5	Yellow	172	1,70533
6	Orange	4	0,0396589
7	Magenta	4360	43,2282
8	Cyan	3	0,0297442

We open JTAG within **a few minutes...**

Wanna see it?

Demo.

Flash Encryption bypassed

```
. Let's try to read some data from flash!
.
. Please note, flash offset 0 is mapped at 0x3f400000
.
tieknimmers:$ telnet 127.0.0.1 4444
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Open On-Chip Debugger
> halt
Target halted. PRO_CPU: PC=0x40080959 (active)      APP_CPU: PC=0x00000000
> esp32 mdw 0x3f401000 40
0x3f401000 3f401000 40080700 000000ee 00000000 ..@?...@.....
0x3f401010 00000000 01000000 3fff0020 00000b30 ..... ..?0...
0x3f401020 3f401020 00000001 3ff5f000 ff045000 .@?...?..P..
0x3f401030 000000ac 00000000 00000004 00000005 .....
0x3f401040 3f401040 00000007 6c656152 2e657a69 @.@?...Raelize.
0x3f401050 000a2e2e 636f6c42 3a30206b 30250020 ....Block 0: .%0
0x3f401060 3f401060 6b636f6c 203a3120 6f6c4200 `.@?lock 1: .Blo
0x3f401070 32206b63 4200203a 6b636f6c 203a3320 ck 2: .Block 3:
0x3f401080 3f401080 50504100 6f6f6200 73652e74 ..@?.APP.boot.es
0x3f401090 00323370 3b305b1b 576d3333 75252820 p32..[0;33mW (%u
```

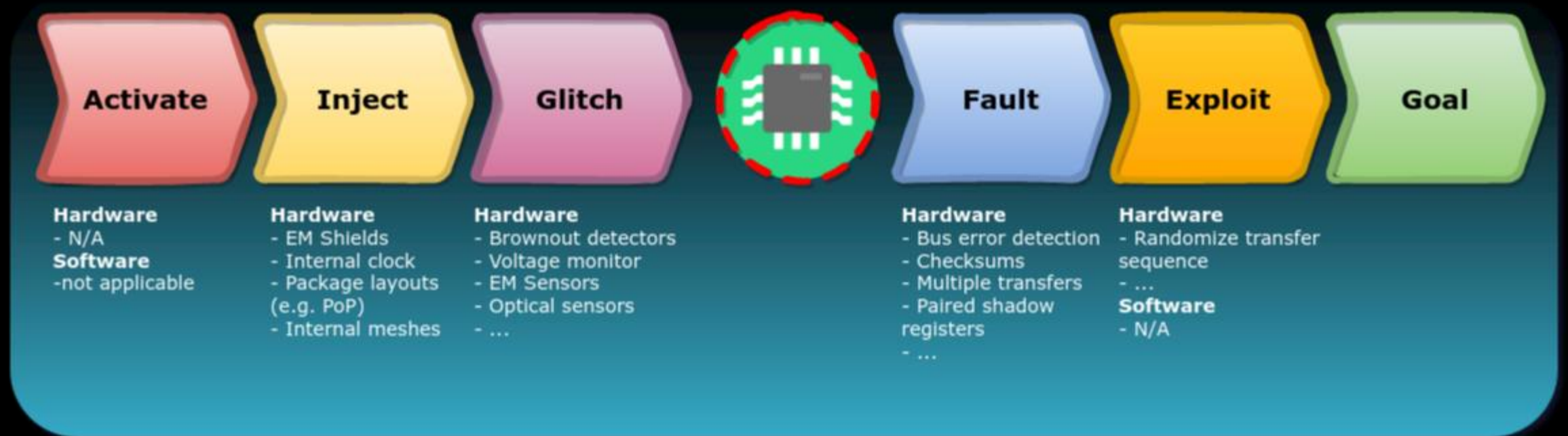
What can be done?

FIRM: Attack Analysis



Source: <https://raelize.com/posts/raelize-fi-reference-model/>

FIRM: Countermeasures



Source: <https://raelize.com/posts/raelize-fi-reference-model/>

Conclusions.

We have seen... (1)

- How OTP data transfers can be used to **compromise** SoC security
- FI attacks against **pure** HW implementations
- Bypass of critical SoC security features
 - Before any SW is **executed**
 - Before any CPU is released from reset
- Strong timing **correlations** allow for powerful attacks

We have seen... (2)

- FI (single glitch) attack to break ESP32 keys read **protection**:
 - No bruteforce required
- **Double** glitch attack to additionally bypass Secure Boot
- FI (single glitch) attack to:
 - Enable **JTAG**
 - Bypass Flash Encryption
 - Bypass secure boot

Final considerations

- FI attacks against OTP data are often **overlooked**
 - Immutability at rest does not imply integrity in transfer
- Design may not include adequate **protection**:
 - FI mostly used for targeting software
- Design-level **mitigations** are possible:
 - E.g. Breaking timing correlations, Bus level integrity checking

raelize

Thank you! Any questions!?

Niek Timmers
niek@raelize.com
[@tieknimmers](#)

Cristofaro Mune
cristofaro@raelize.com
[@pulsoid](#)