Replicant: A Primer on Fault Injection Attacks

Matthew Alt

VoidStar Security LLC

Outline

- Introduction / Goals
- Fault Injection Overview
- Target Overview
- The Attack
  - Hardware
  - Software
- Conclusion

Intro / `whoami`

- `whoami?`
  - Matthew Alt
  - @wrongbaud
- Security researcher for Caesar Creek Software
  - Previously @ MIT Lincoln Laboratory, Revo Technik/STASIS Engineering
  - Recovering ECU tuner
- Offer training/consulting through VoidStar Security LLC

Presentation Goals

- Provide fault injection overview and beginners guide
- Review steps taken to replicate public fault injection attacks
  - Hardware/Software components
  - Problems encountered along the way
- Replicate RDP2 -> RDP1 bypass

Fault Injection Overview

- Fault injection involves purposefully introducing an error on a target
- Undefined behavior!= hard fault
  - We want to modify the behavior but **not** crash the target
- This involves injecting a high voltage pulse or draining the voltage from a power source on the target.

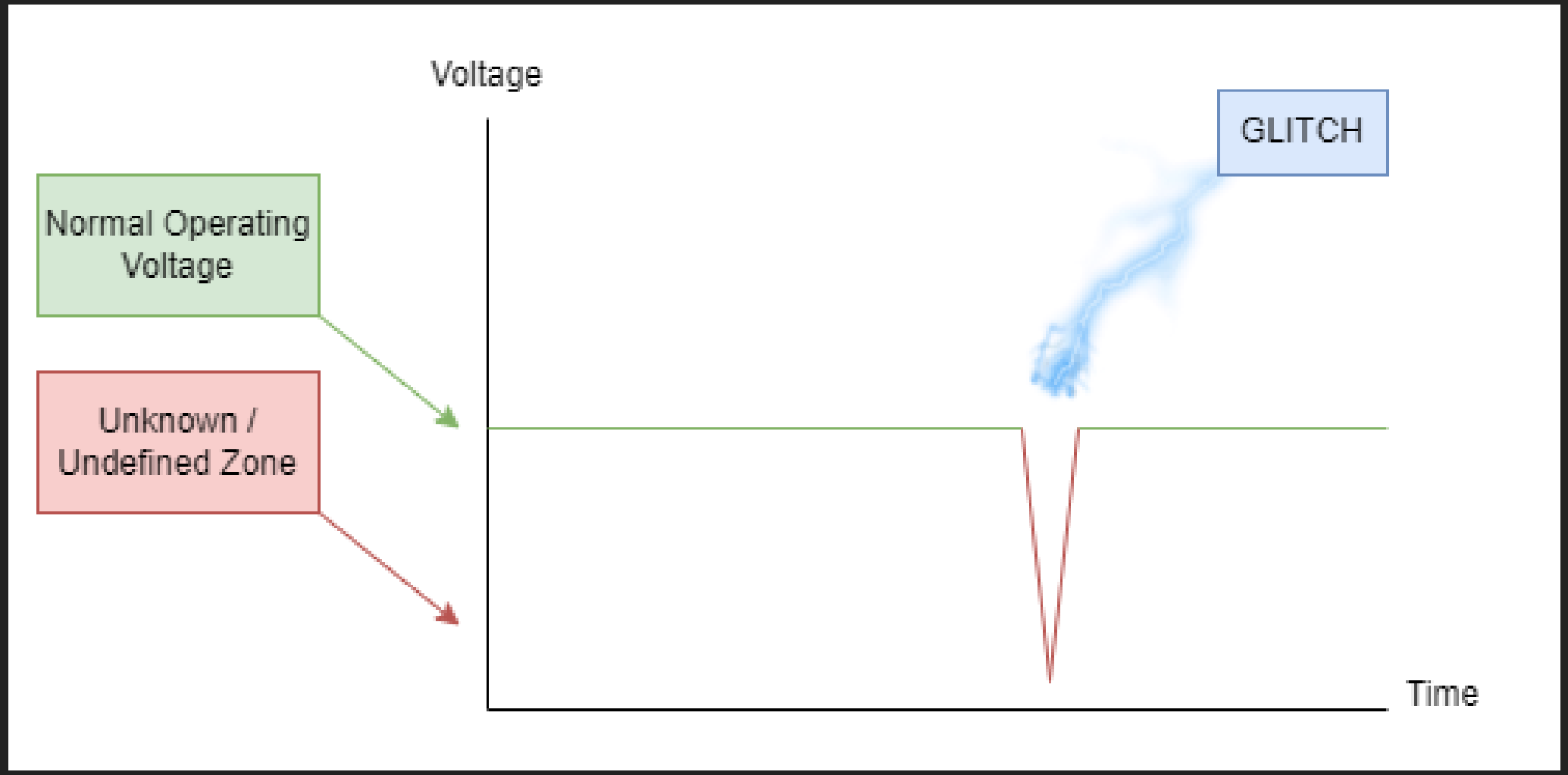
Fault Injection Overview

- By causing momentary voltage modulations, we can force our 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.

Voltage Glitching

- Voltage glitching involves targeting the power source of the entire system.
- By briefly cutting the power to a target system, we can modify its behavior/performance.
- Our goal is to reduce the voltage for a short enough time such that the processor enters an undefined state
- Processors have features to mitigate this, requiring component removal or injection methods.

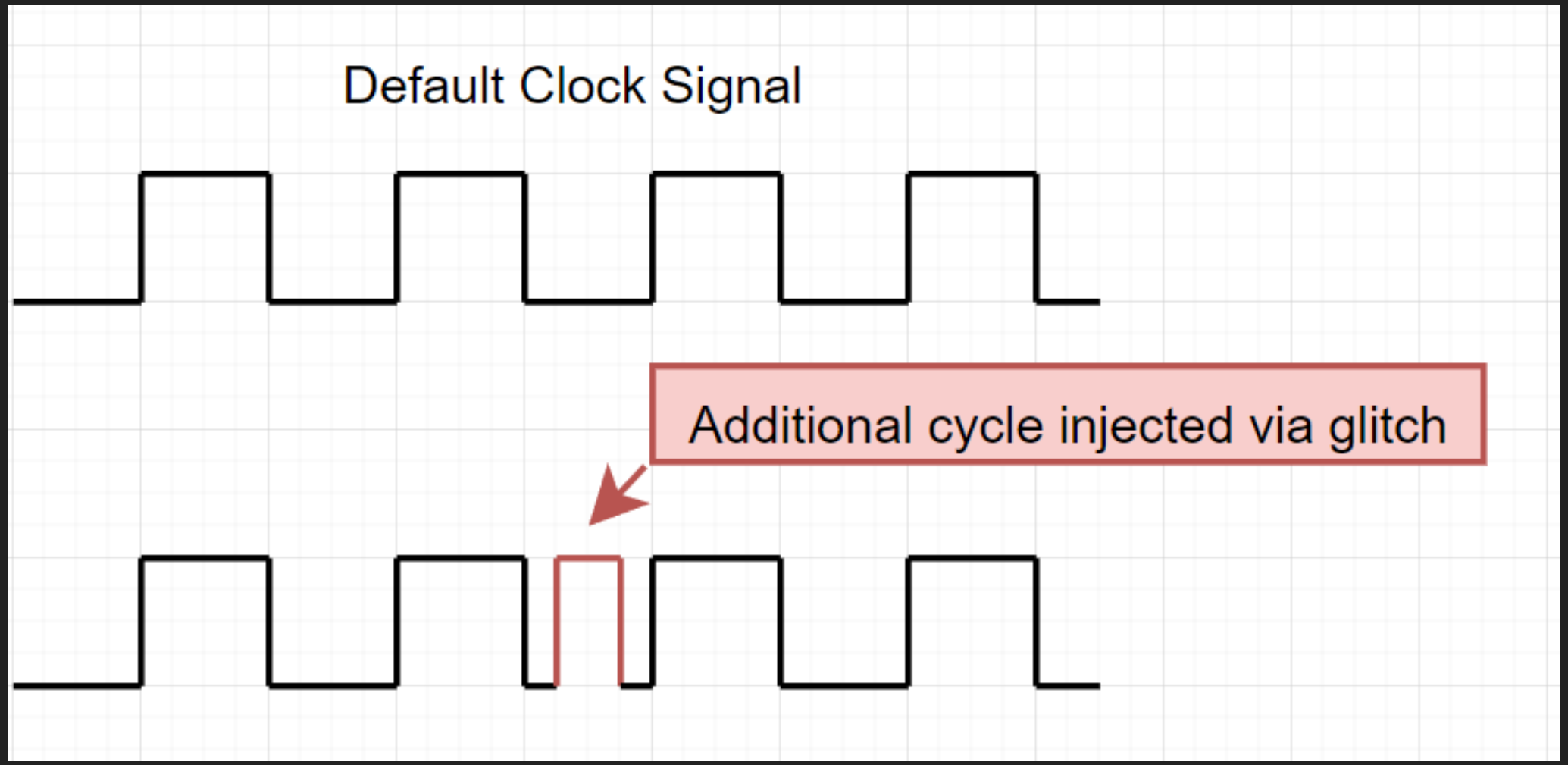
Voltage Glitching



Clock Glitching

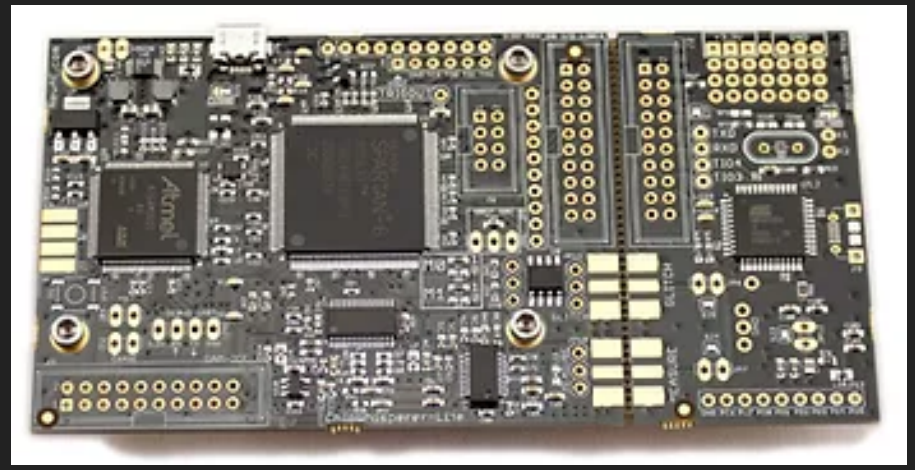
- Clock glitching is done by targeting the external clock of a target CPU
- Additional clock cycles are injected onto the clock line
- With clock glitching, our goal is to skip or modify instructions.

Clock Glitching



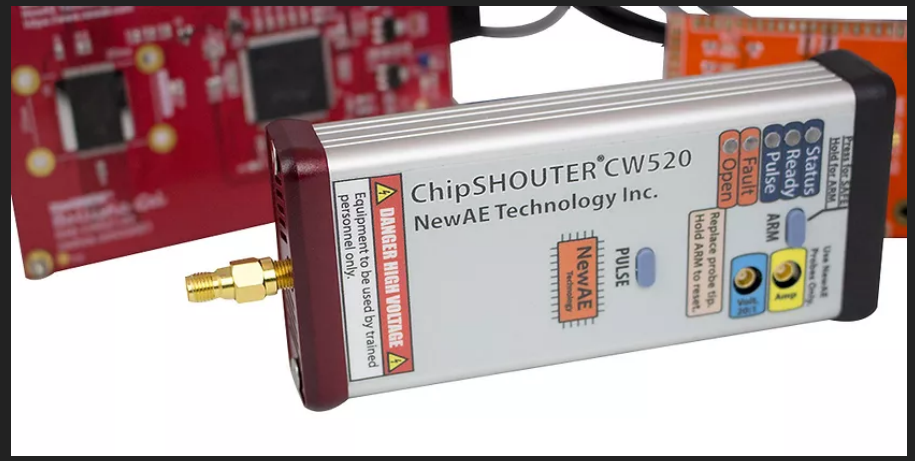
Tools: ChipWhisperer

- Open source fault injection platform
- Designed by Colin O'Flynn of NewAE Technology
- Excellent resource for learning about fault injection
  - Tutorials
  - Forums
  - Documentation



Tools: ChipShouter

- Electromagnetic Fault Injection (EMFI) platform
- Used to generate electromagnetic pulses to disrupt platform operation
- Developed by NewAE Tech

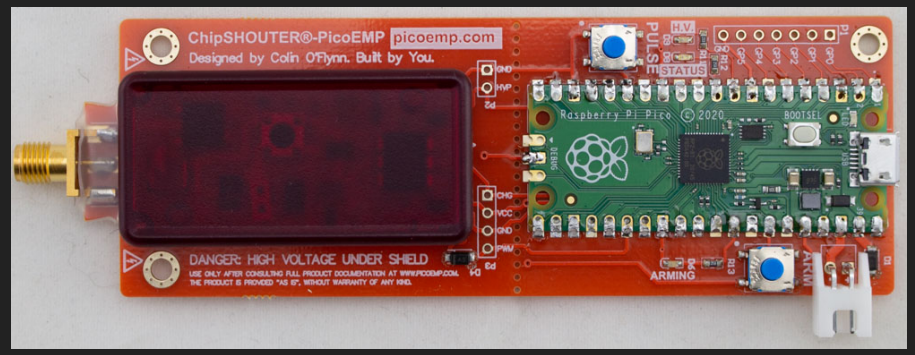


Electromagnetic Fault Injection

- EMFI attacks involve generating a large electric field targeted at a specific region of an integrated circuit
- This field can cause hardware to fail, resulting in potential bit-flips and other undefined behavior.
- Tools for this include the PicoEMP or chipshouter

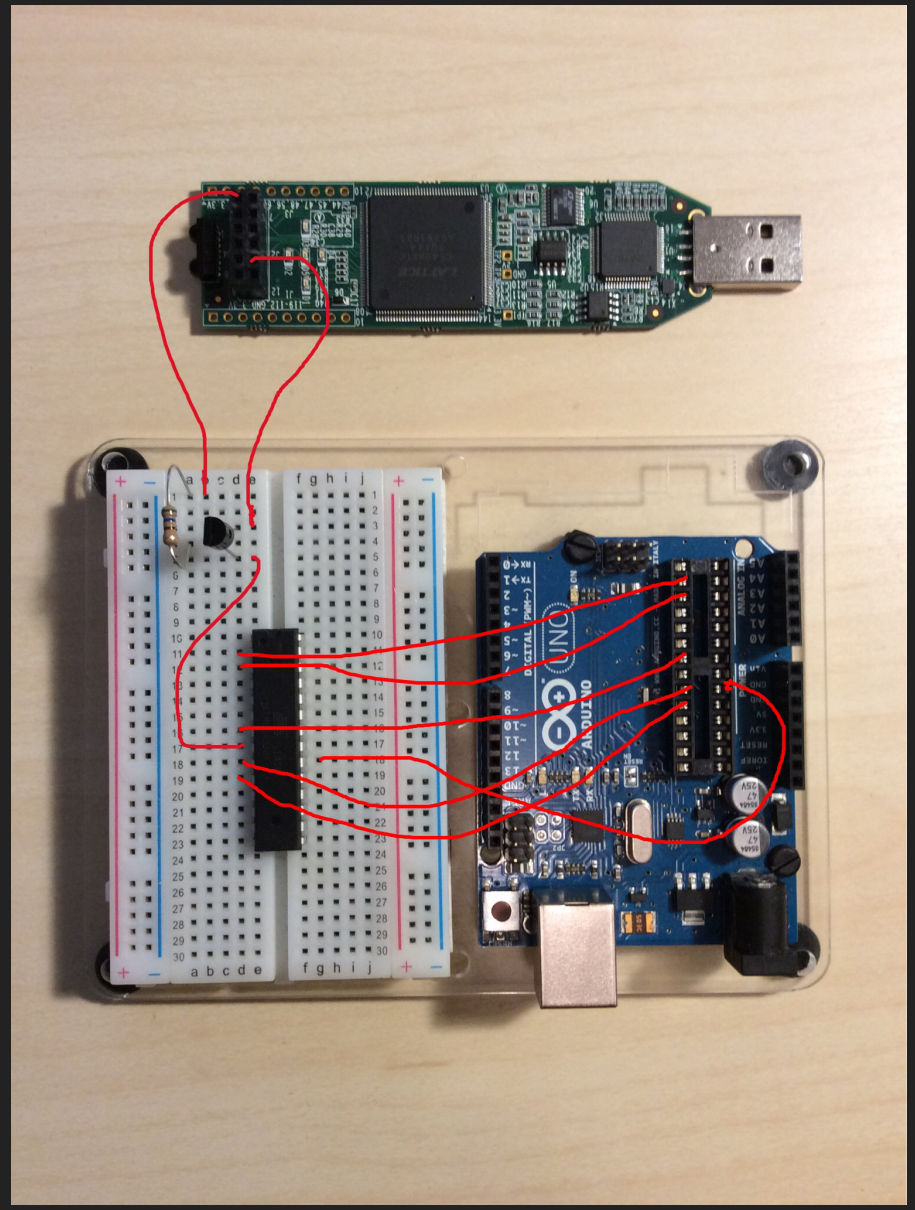
Tools: PicoEMP

- Low-cost Electromagnetic Fault Injection (EMFI) tool
- Designed specifically for self-study and hobbyist research
- Relatively cheap to build and easy to use



Tools: Custom / Low Cost

- The overall workflow for a glitch is relatively simple
  - Detect trigger event (reset line, UART packet, etc)
  - Trigger glitch!
- Can be done with a development board and patience
  - IceStick Glitcher
  - Pocket Glitcher



Target Overview

- The target for this work is the Trezor One wallet.
- This is a popular low-cost wallet that is built around the STM32F2 microcontroller.
- Trezor's hardware and software are both open source
  - Provides us access to the hardware diagrams and the firmware source

STM32 Security Overview

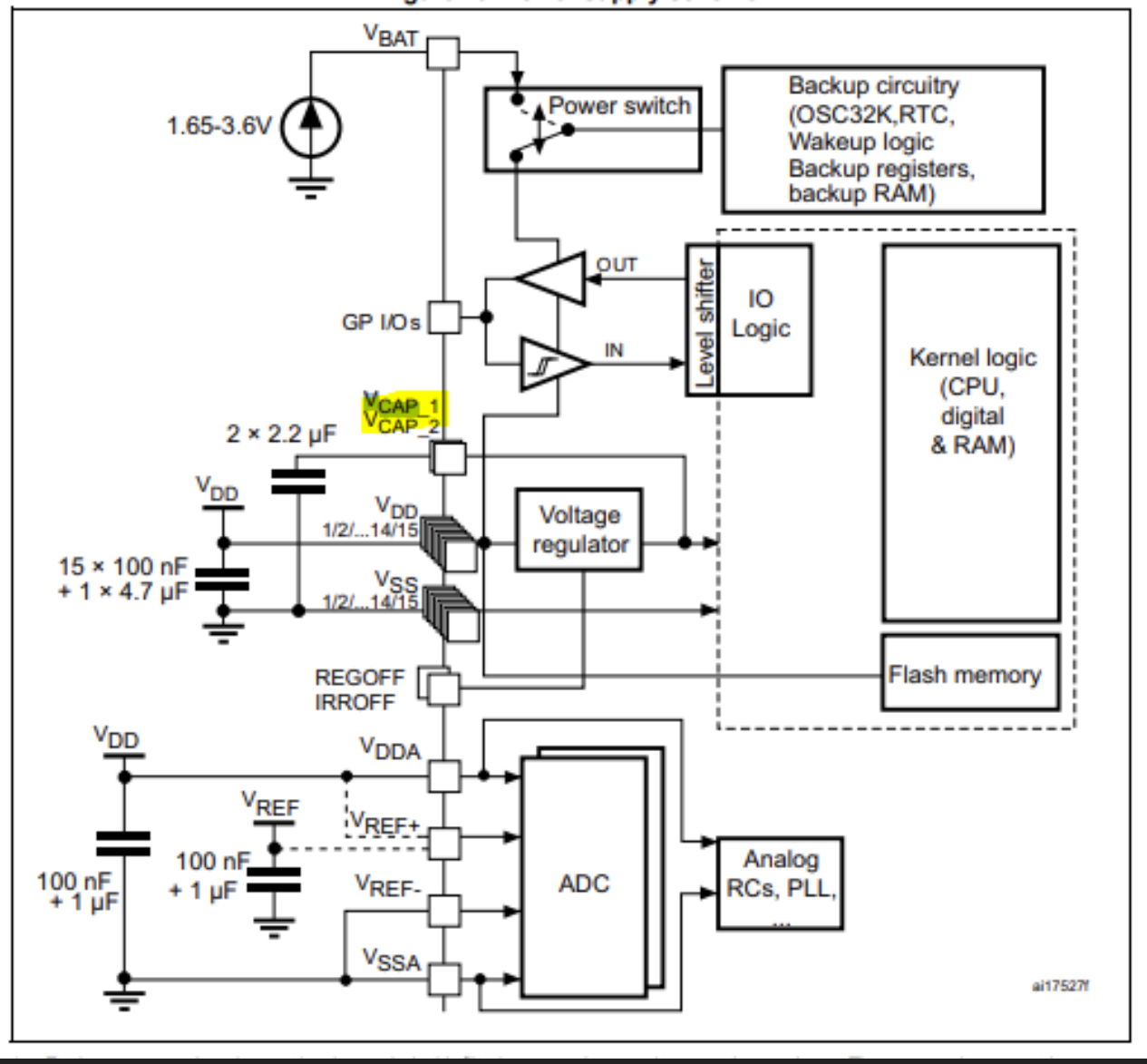
- The STM32 has multiple levels of "Read-out protection" (RDP)
- RDP 0: Flash unlocked, all-flash/ram is accessible via the debug interface
- RDP 1: Flash locked; you can connect a debugger and read out RAM/peripherals, but not flash.
- RDP 2: Flash locked, RAM reads locked, debug interface locked

STM32 Power Management/ Regulation

- Within any microcontroller, there are multiple power domains
  - Power Domain: Shared power source
- Used for powering various chip peripherals and internal operations and comparators.
- We will be targeting the internal voltage regulator.
  - Exposed via `VCAP_1` and `VCAP_2`

6.1.6 Power supply scheme

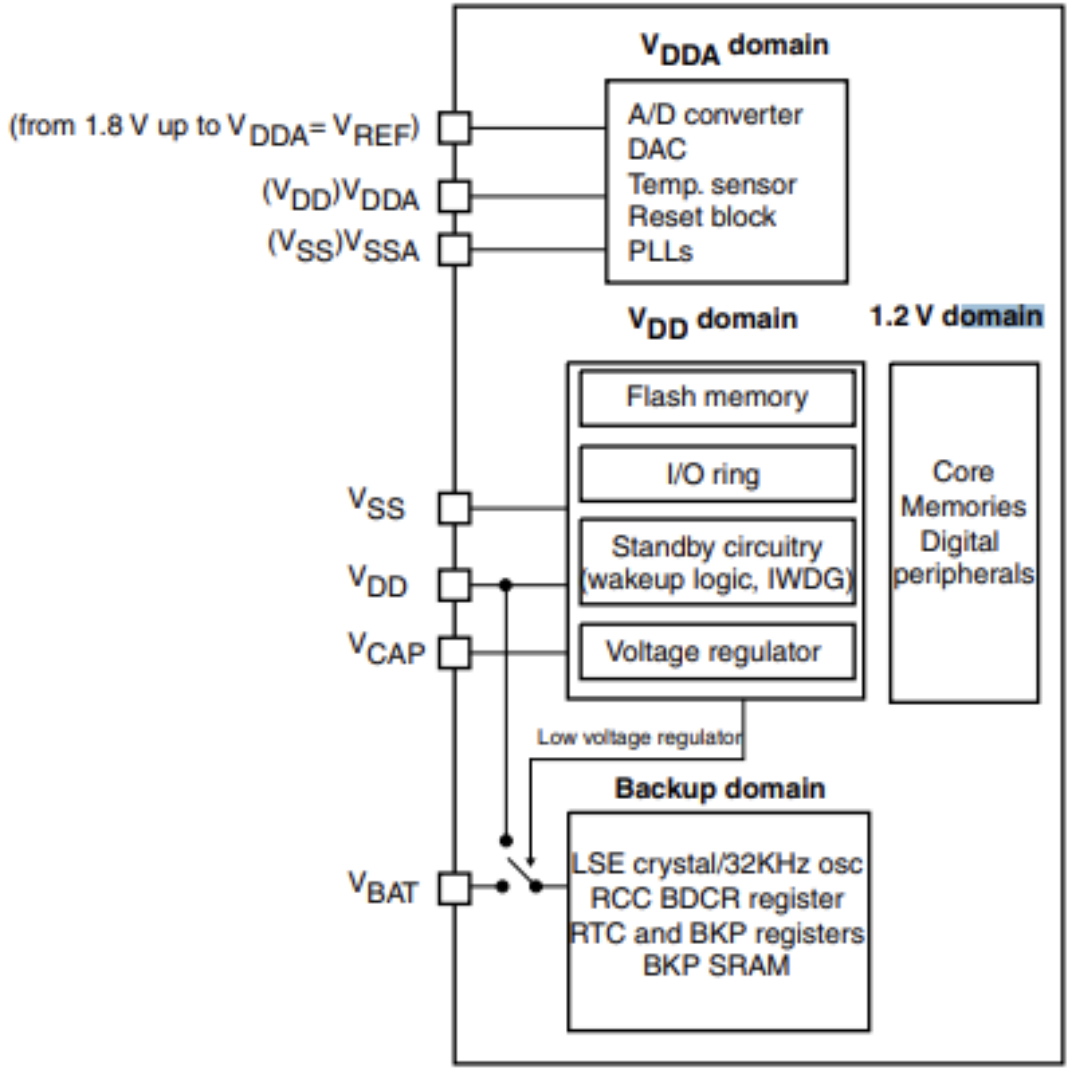
Figure 19. Power supply scheme



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!

Figure 1. Power supply overview



STM32 Power Management/ Regulation

- We want to attempt to modify the RDP state of the device from RDP2 to RDP1,
- We will do this by **glitching** the voltage on the `VCAP_1` / `VCAP_2` lines

If we can modify the behavior of the internal voltage regulator, we can potentially alter the processor's behavior.

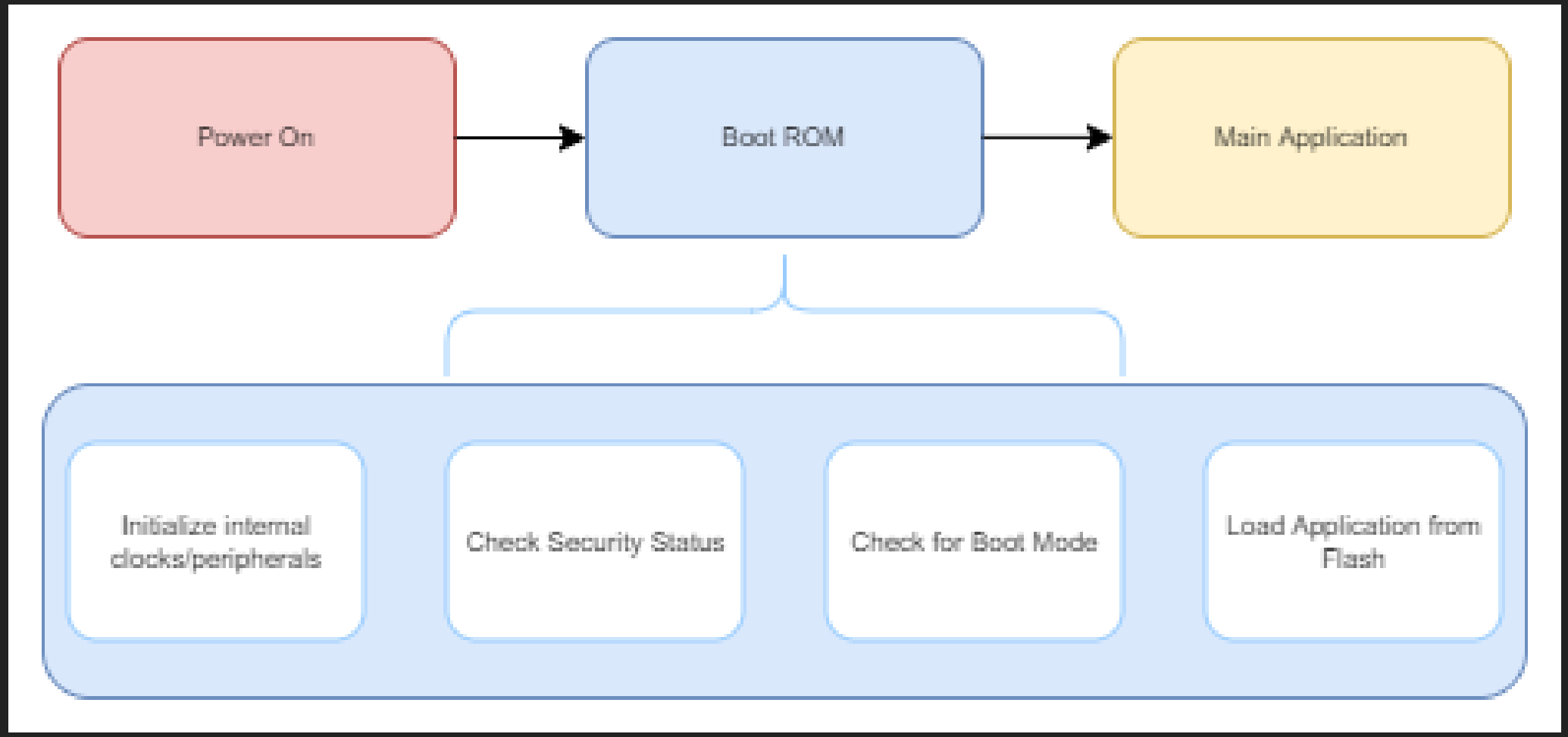
The Attack: Overview

- We are replicating the attack presented in the chip.fail research
- The attack targets the boot ROM of the STM32F2 series microcontroller
- This is the same attack used by Joe Grand and also documented by Kraken Labs

chip.fail

- It was discovered that at approximately 170uS after startup, the boot ROM checks and sets the RDP settings
- Checks whether the chip is in RDP2 or RDP1
- Via voltage glitching, this check can be modified
  - Allows the user to force the chip to boot with RDP1 settings
- RDP1 allows for SRAM access!
  - In old firmware versions this would allow for the recovery keys to be read out!

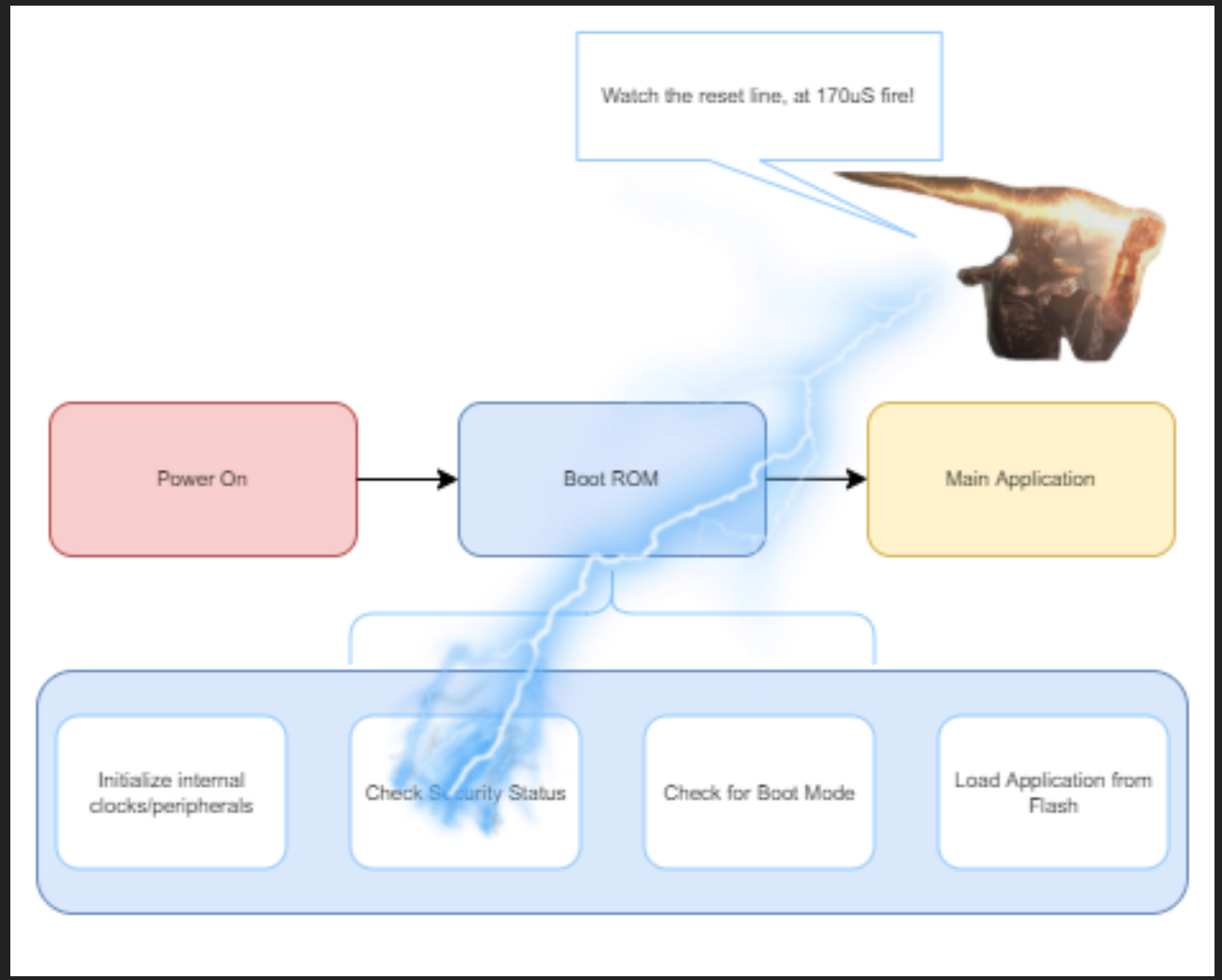
The Attack: Overview



The Attack: Overview

1. Power on the wallet
2. When the RESET line is asserted, begin the countdown to the glitch
3. At 170 microseconds, pull VCAP low
4. Test RDP bypass via SWD
5. Read SRAM from the target device

The Attack: Overview



The Attack: Next Steps

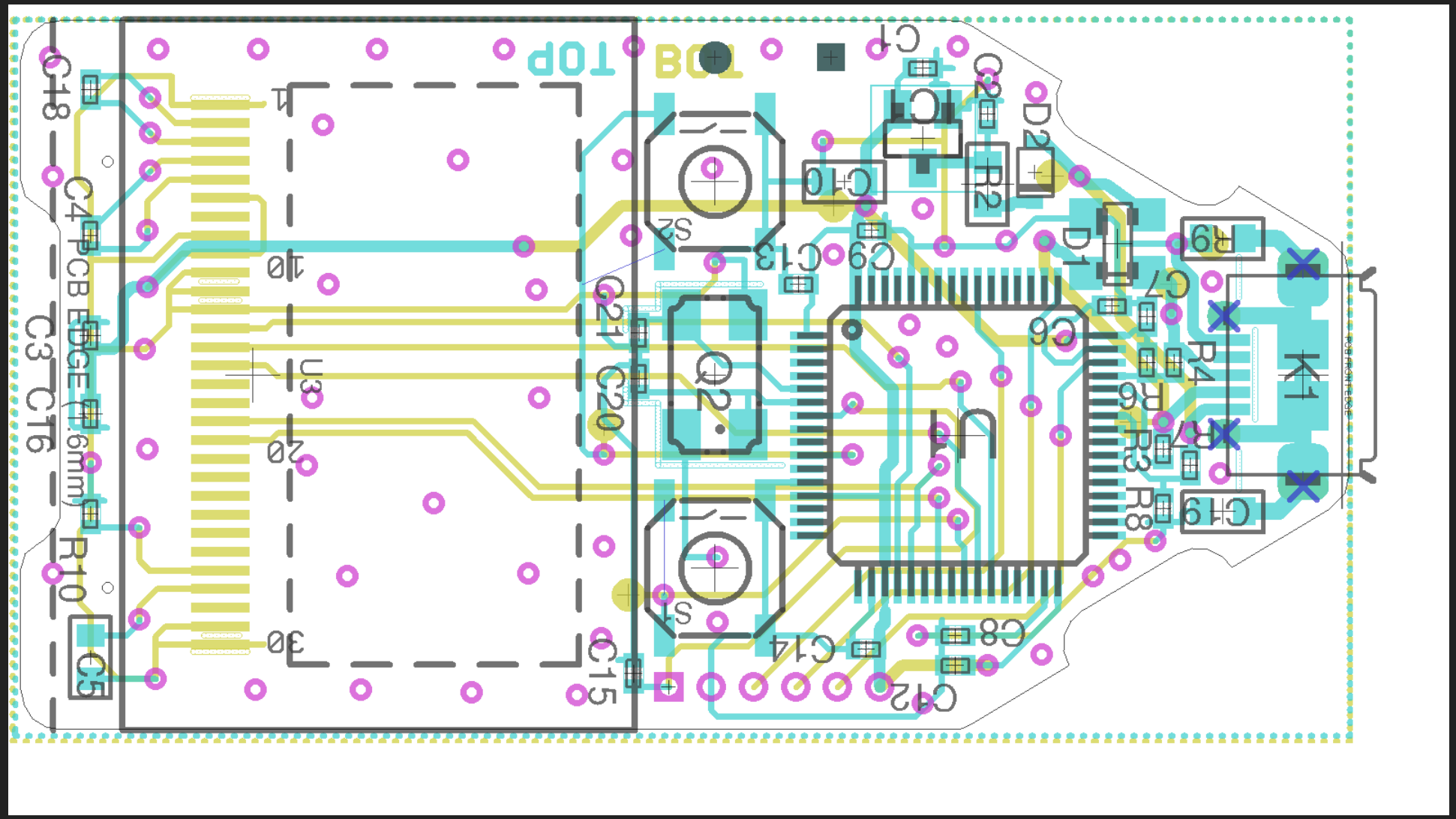
- What does an attack like this look like at the signal level?
- How do we know when the processor has started executing the boot ROM?
- How would one proceed if analyzing a new target/power trace?

Let's start by removing a few components from our target and looking at a power trace.

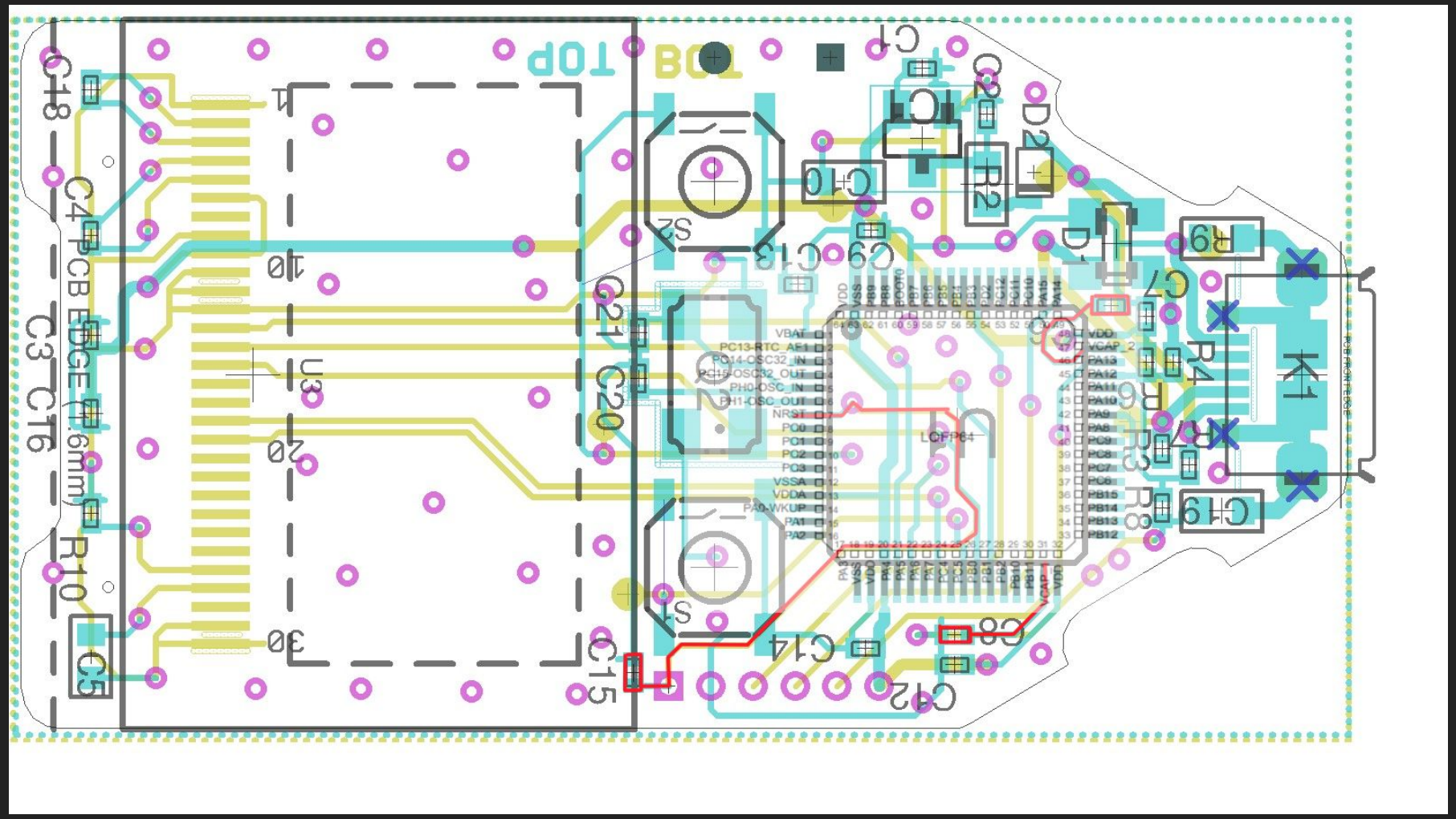
The Attack: Hardware

- Capacitors are often used on voltage busses to help filter and regulate a power supply
  - This is the last thing we want!
- To increase the chances of a successful glitch, we will remove the capacitors on:
  - `VCAP_1`
  - `VCAP_2`
  - `RST`

Trezor One: Schematic



Trezor One: Component Removal

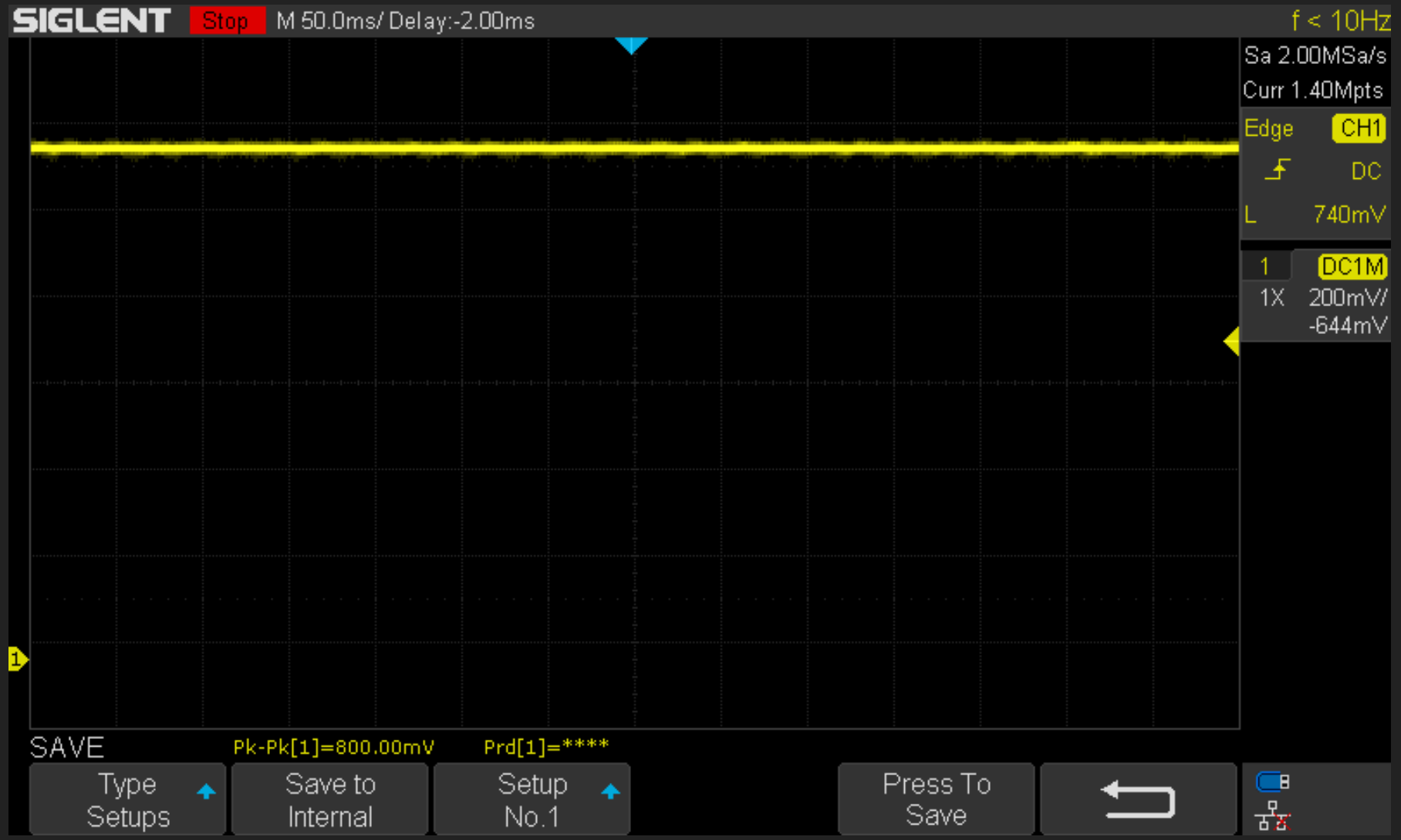



8.0
AMM2 E8C

Hardware: Capturing Power Traces

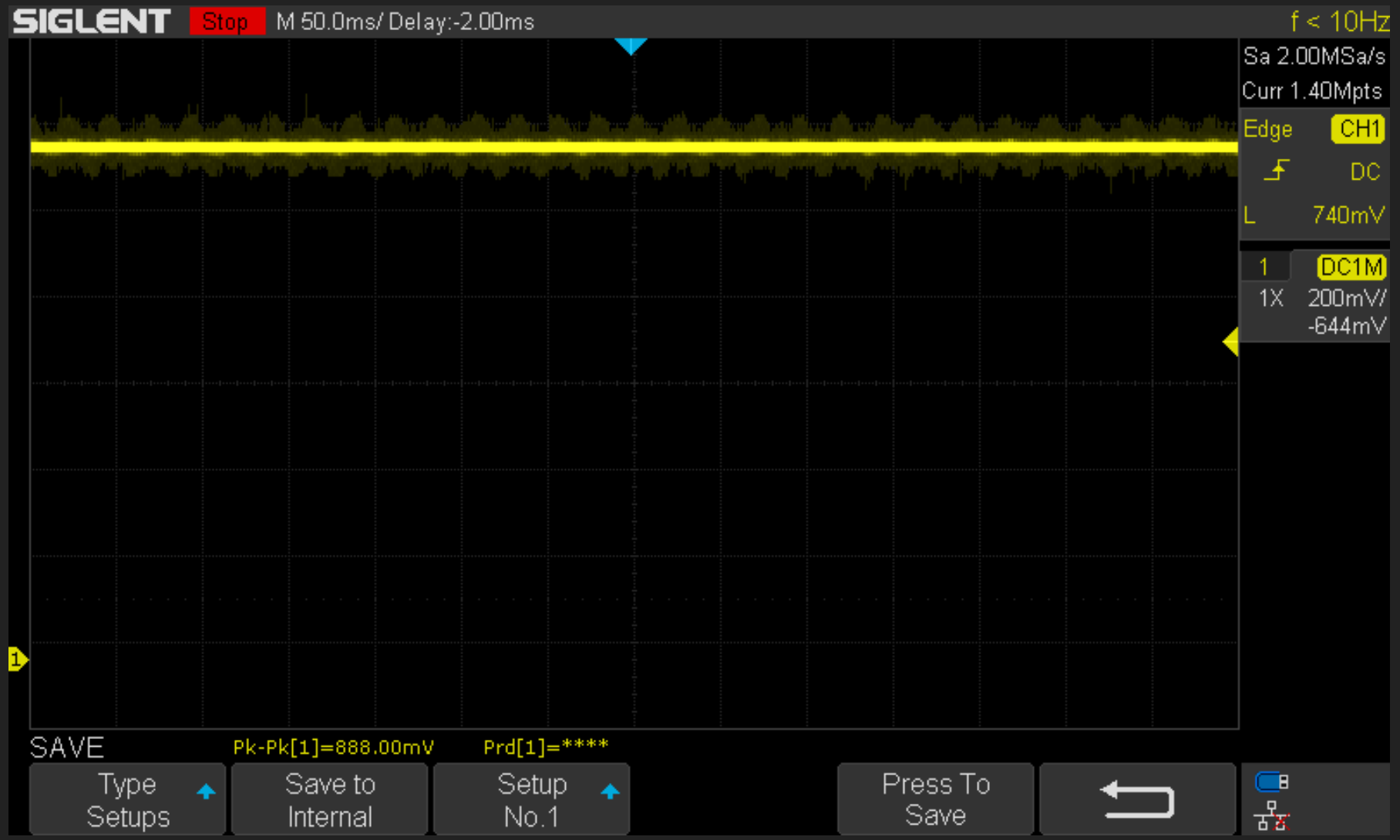
- We need to analyze the power consumption of the device to confirm the appropriate offsets
  - Done with an oscilloscope
- Captures will trigger on the `RESET` line of the Trezor
  - This indicates that the boot ROM has begun
- Capacitor removal is necessary for *noisy/loud* power traces

Hardware: Capturing Power Traces



Voltage on the `VCAP` line with the external capacitors

Hardware: Capturing Power Traces

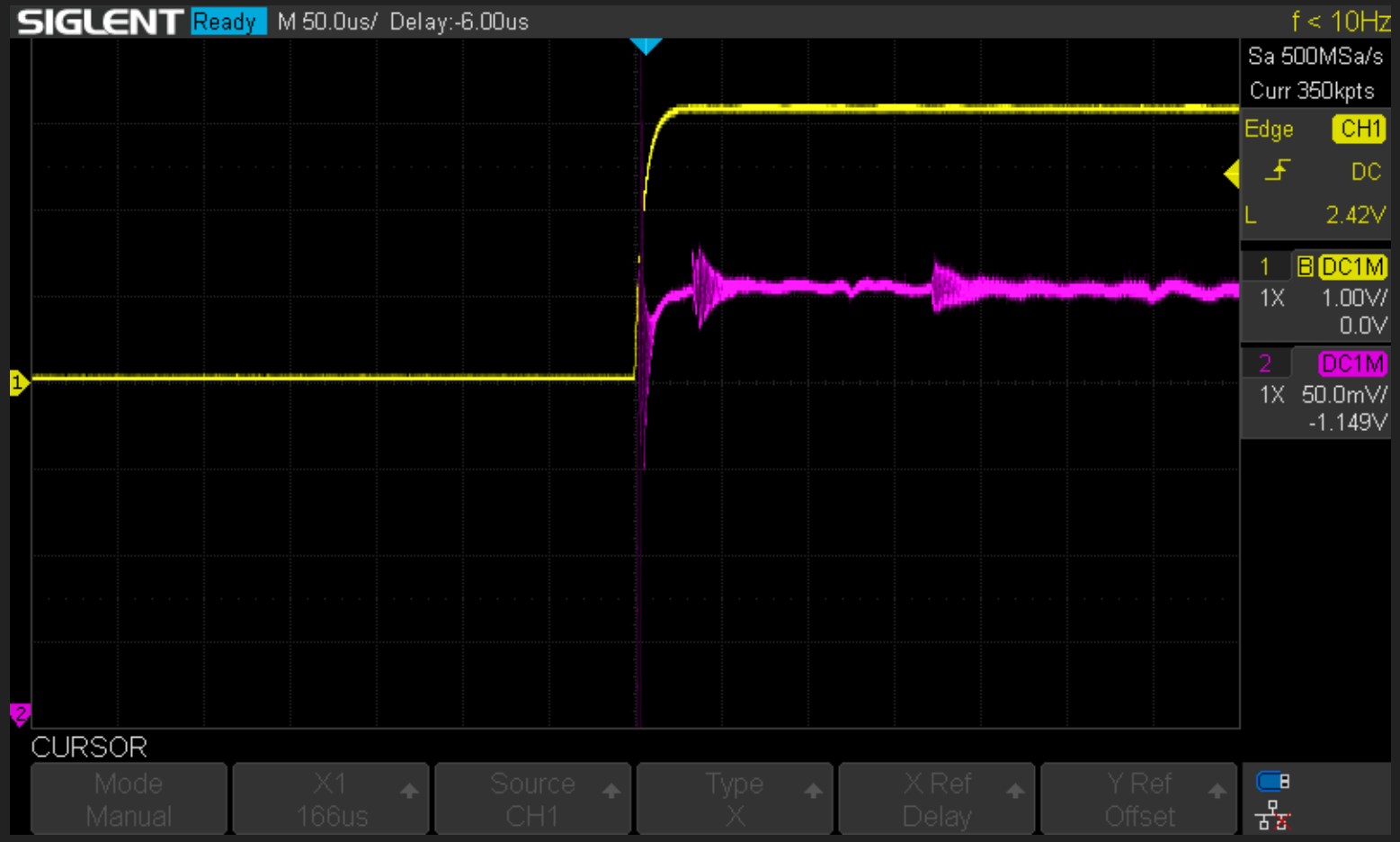


Voltage on the `VCAP` line without the external capacitors

Hardware Capturing Power Traces

- The bootrom begins execution when the system reset line hits the 3.3V threshold.
- By monitoring the reset line, we can determine when the boot ROM begins execution
- We will use this as our trigger for our glitch.

Power Trace: Analysis



The pink line represents the voltage on the `VCAP` line, the yellow line is `RESET` line

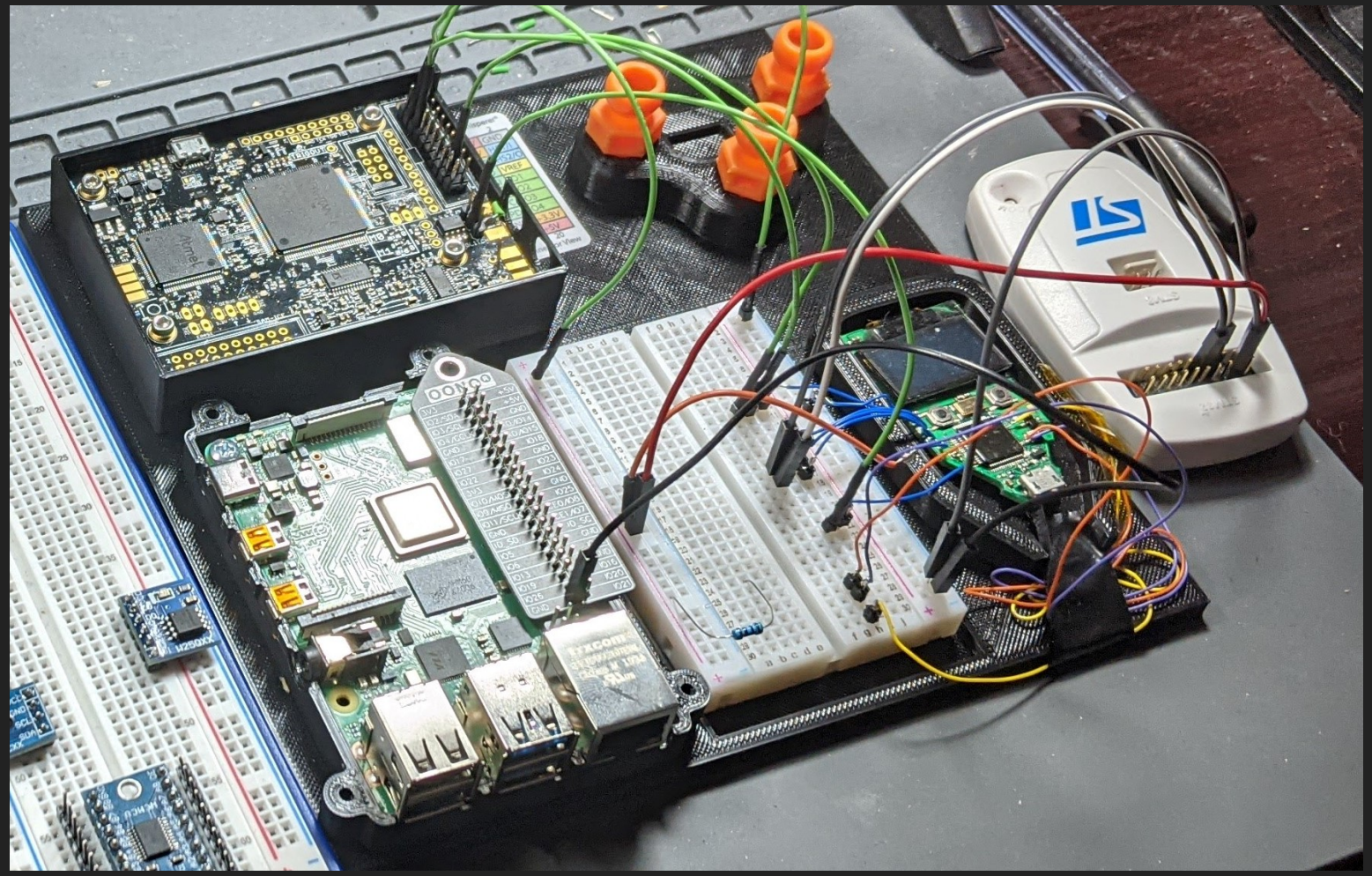
Power Trace: Analysis

Power Trace: Analysis

The Attack: Hardware

- Now that we have reviewed the power trace, we need to set up the attack
- We will need the following:
  - A way to trigger on the reset line (ChipWhisperer)
  - A glitching tool (ChipWhisperer)
  - A way to detect *if* SWD has been re-enabled (STLink)
  - Computer to control it all (Raspberry Pi)

The Attack: Hardware



ChipWhisperer Roles

- The CW is responsible for:
  - Powering the target
  - Detecting the reset line as a trigger
  - Counting down to the proper time for a glitch
  - Triggering the glitch!
  - Resetting the device
- The STLink is used to detect if SWD is enabled
- The Pi is used to control it all via python!

Wiring Tables: CW to Trezor

ChipWhisperer Pin Number / Usage	Trezor Pin
(14) / `FPGA-TARG4`	`RST`
(5) / `PROG-RESET`	`RST`
`Glitch Out`	`VCAP1`
(3) / `+3.3V`	`VCC`
(2) / `GND`	`GND`

Wiring Tables: STLink to Trezor

STLink Header	Trezor Pin
GND	VDD
SWCLK	PA14
SWDIO	PA13
VTREF	VCC

The Attack: Software

- Now that all the appropriate components are connected, we need to control them
  - Done in python using CW APIs
  - STLink can be used via python as well
- NewAE tutorials are a great place to start
  - We used them as a skeleton for our code

The Attack: Software

```
if(drinks < 3):
    review_code();
else:
    poorly_review_code();
```

Note: For those visiting these slides at a later date, the code is much better described in the blog post

Shaping The Glitch

- The three variables we use to help shape our glitch are the `width`, `repeat`, and `offset` variables.
- `width`: How wide to make the glitch. This is the percentage of one period.
- `repeat`: The number of clock cycles to repeat the glitch. Higher values increase the number of instructions that can be glitched but often increase the risk of crashing the target.
    - **Note:** a higher repeat typically results in a **stronger** glitch
- `offset`: Where in the output clock to place the glitch.

Placing the Glitch

- The `ext_offset` defines how many clock cycles the FPGA will wait after triggering before glitching
- Since we specified a clock rate of 100 MHz, 15000 clock cycles equate to ~150 microseconds.
- **After** the `RESET` line goes high on the Trezor, we will start counting down from our `ext_offset` value
    - When it hits zero - we will glitch the `VCAP` line!

Now we wait

Debugging the Glitch

- After running the attack for some time with no results, we began troubleshooting our setup.
- We needed to ensure:
  - The glitch placement was correct
  - The detection code (STLink) was properly detecting the SWD port being enabled

Debugging the Attack: USB

```
'''
swd_check
Use the link to attempt to connect via SWD
'''
def swd_check():
    global dev
    import swd
    pc = 0
    try:
        dev = swd.Swd()
        pc = dev.get_version().str
    except:
        del swd
        pass
    return pc
```

We tested this subroutine with a development board, everything seemed OK!

Debugging the Attack: USB

- Everything seemed OK when testing with the development board
- There was one problem:
  - We were testing if the SWD probe worked on the *first* attempt
  - Our code required many many attempts!
- After testing a second attempt, it failed!
  - Even though the tested device was unlocked

Bug Number 1: STLink Re-enumeration

- The STLink needed to be re-enumerated after each failed SWD probe
  - AKA: For EACH glitch attempt
- We fixed this by writing a C program that would reset the device via `sysfs` between glitch attempts
- We modified a simple c program to reset the STLink device after each glitch attempt.
- Surely this was what was stopping us - on to glitching!

Bug Solution: Force reset

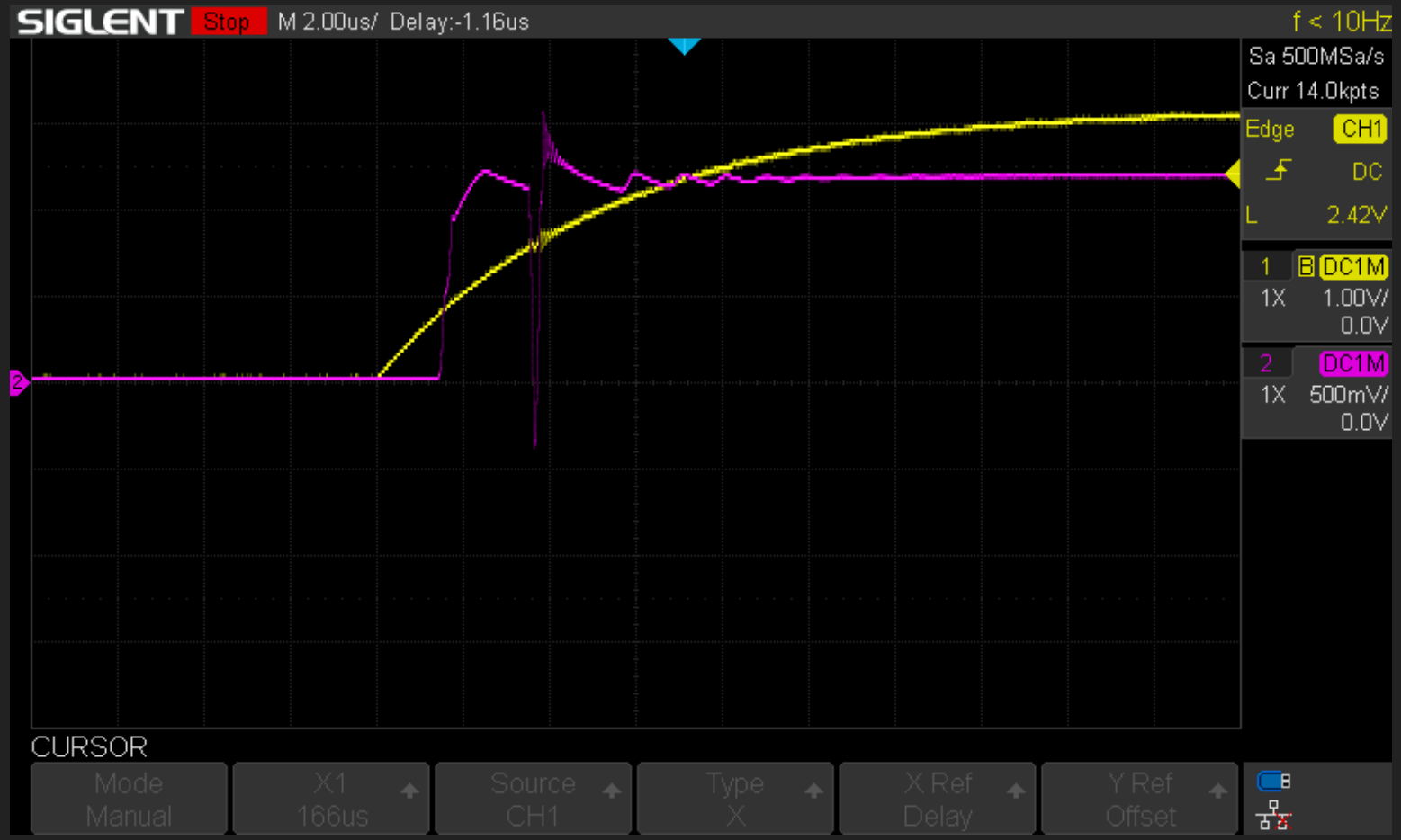
```
'''
swd_check
Use the link to attempt to connect via SWD
'''
def swd_check():
    global dev
    import swd
    pc = 0
    try:
        # Reset the STLink
        os.system(f"sudo /home/pi/glitch/replicant/python/usbreset {usb_path}")
        dev = swd.Swd()
        pc = dev.get_version().str
    except:
        del swd
        pass
    return pc
```

Now we wait

Bug Number 2: Triggering

- Unfortunately, we were still not seeing results after a few more days of testing
- We decided to get the scope back out and inspect our glitch.
- We examined the glitch with an `ext_offset` of 16000,17000, and 18000, and it *looked* reasonable to the naked eye.
- With an `ext_offset` of 0, we saw the following:

Bug Number 2: Triggering

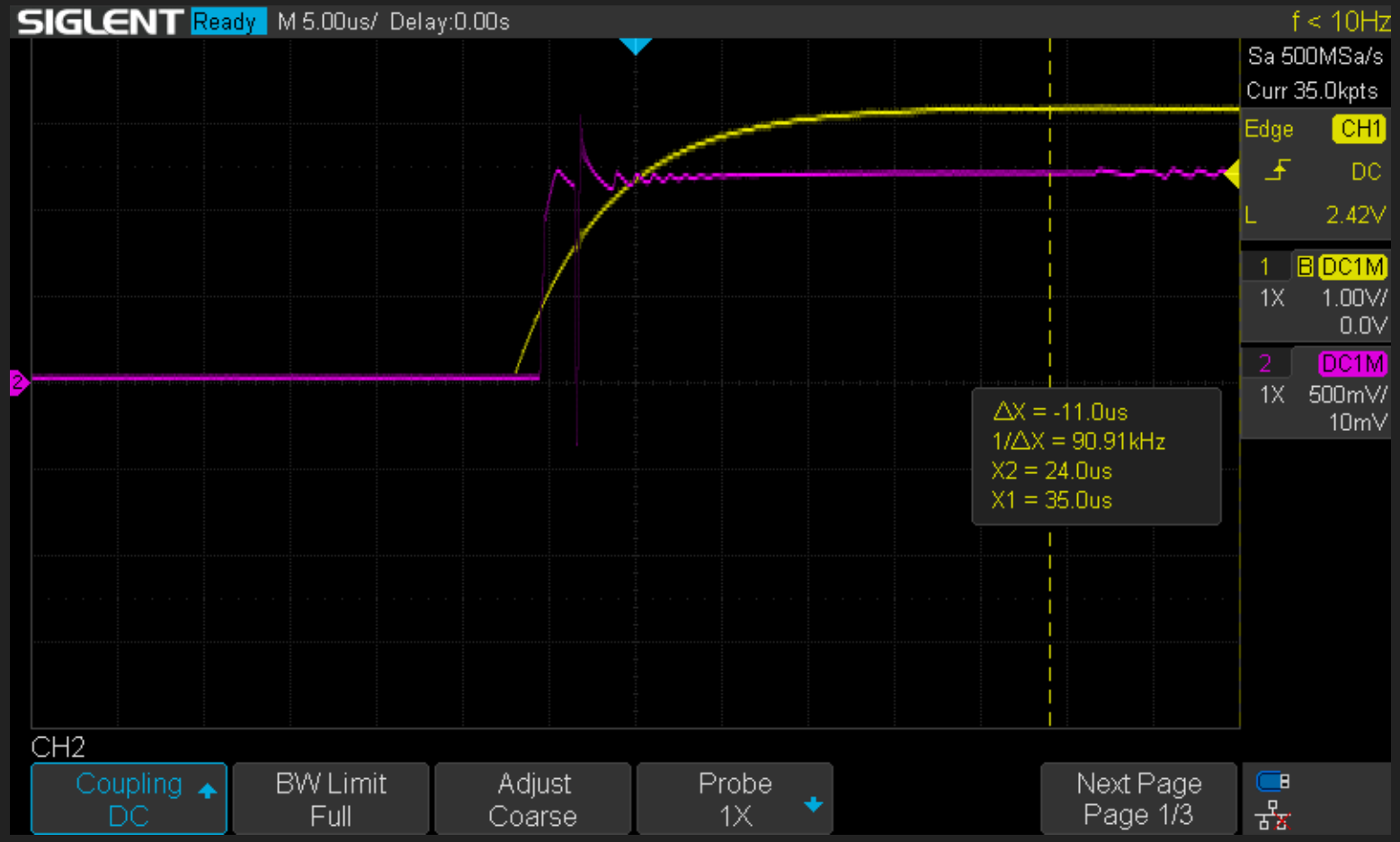


Pink = Glitch, Yellow = Reset. Does anything look odd?

Bug Number 2: Triggering

- The ChipWhisperer is triggering before the reset line reaches 3.3V!
- This early triggering was causing our glitch to start counting down before the reset line was fully asserted.
- The glitch was triggering about 20 microseconds too early.

Bug Number 2: Triggering



Bug Number 2: Triggering

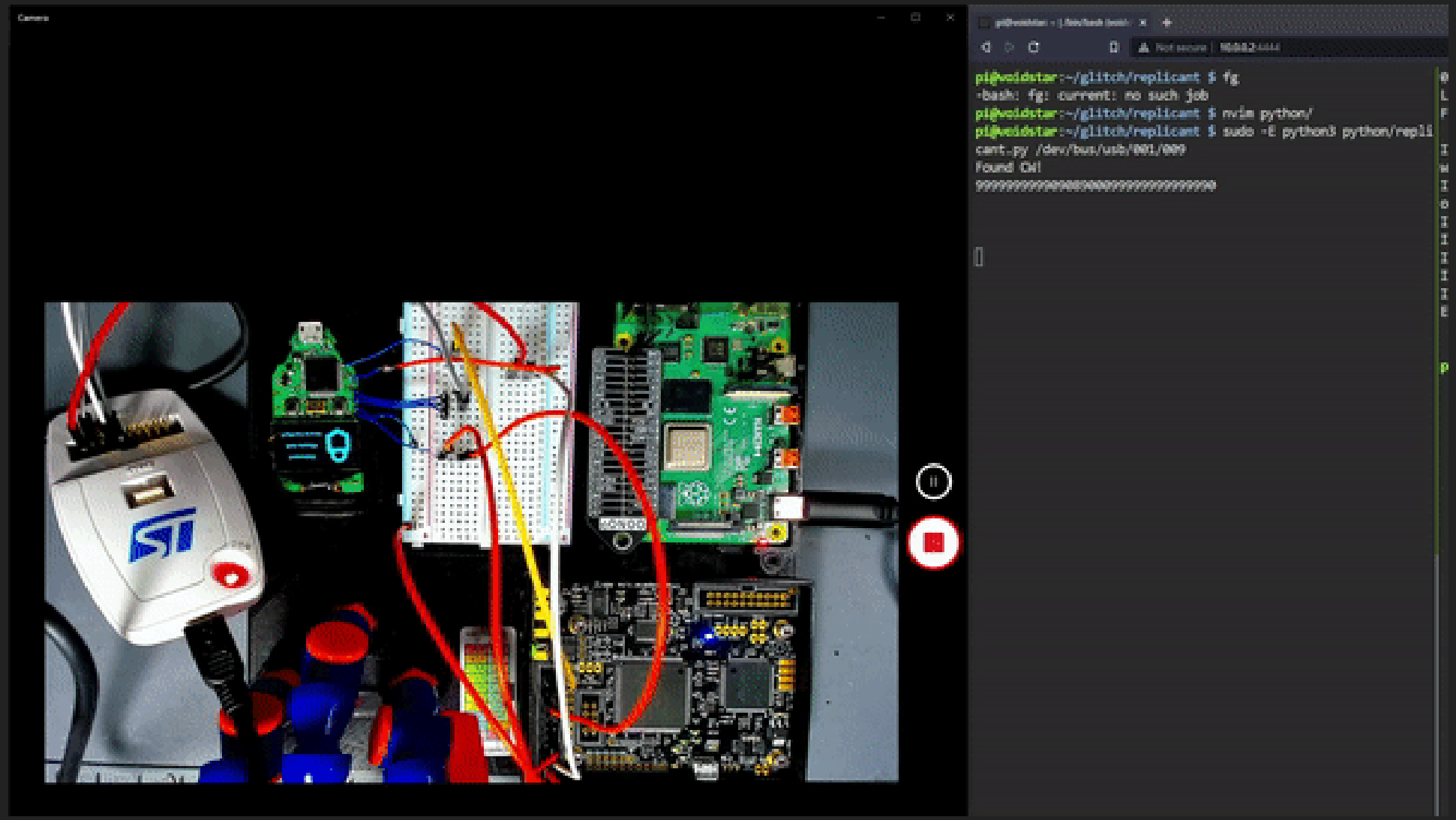
- We determined we were triggering approximately 24 microseconds too early using the oscilloscope.
- Changed `ext_offset` range to be between 17000 and 20000 and left that running over the weekend...

Now we wait

Success!

```
pi@voidstar:~/glitch/replicant $ sudo -E python3 python/replicant.py /dev/bus/usb/001/004
Found CW!
Success! -- offset = -44.921875, width = 39.84375, ext_offset = 19752
successes = 1, failures = 0, offset = -44.921875, width = 39.84375, ext_offset = 19752
Done glitching
```

When we returned to the office on Monday, the STLink LED was green, meaning it had successfully accessed the device via SWD!


Camera
pi@voidstar:~/glitch/replicant $ fg
-bash: fg: current: no such job
pi@voidstar:~/glitch/replicant $ nvim python/
pi@voidstar:~/glitch/replicant $ sudo -E python3 python/repli
cant.py /dev/bus/usb/001/009
Found CH!
HONOR

Using the Glitch

We can now use OpenOCD to read the SRAM region with the glitch.

```
pi@voidstar:~/glitch/replicant $ ./run_openocd.sh
Open On-Chip Debugger 0.10.0+dev-01514-ga8edbd020-dirty (2022-03-01-19:24)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info: auto-selecting first available session transport "hla_swd". To override use 'transport select <transport>'.
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 11111 for telnet connections
Info : clock speed 1000 kHz
Info : STLINK V2J29S7 (API v2) VID:PID 0483:3748
Info : Target voltage: 3.259749
Info : stm32f2x.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f2x.cpu on 3333
Info : Listening on port 3333 for gdb connections
Info : accepting 'telnet' connection on tcp/11111
```

Using the Glitch

With OpenOCD running, we can connect to it via telnet and read out SRAM:

```
pi@voidstar:~/glitch/replicant $ telnet localhost 11111
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
> dump_image sram2.bin 0x20000000 0x1FFFFFFF
```

Using the Glitch

```
pi@voidstar:~/glitch/replicant $ hexdump -n512 -C sram2.bin
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000090  78 77 00 08 90 77 00 08  ff ff 00 00 13 70 00 08  |xw...w.......p..|
000000a0  1f 70 00 08 26 70 00 08  24 00 00 00 28 00 00 00  |.p..&p..$...(...|
000000b0  00 01 04 00 01 00 00 00  00 00 00 00 00 01 57 49  |..............WI|
000000c0  4e 55 53 42 00 00 00 00  00 00 00 00 00 00 00 00  |NUSB............|
000000d0  00 00 00 00 80 c3 c9 01  00 87 93 03 00 00 00 00  |................|
000000e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000200
```

A struct is present in the RAM dump that can be found at the following lines of code.

Using the Glitch

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000090  78 77 00 08 90 77 00 08  ff ff 00 00 13 70 00 08  |xw...w.......p..|
000000a0  1f 70 00 08 26 70 00 08  24 00 00 00 28 00 00 00  |.p..&p..$...(...|
000000b0  00 01 04 00 01 00 00 00  00 00 00 00 00 01 57 49  |..............WI|
000000c0  4e 55 53 42 00 00 00 00  00 00 00 00 00 00 00 00  |NUSB............|
000000d0  00 00 00 00 80 c3 c9 01  00 87 93 03 00 00 00 00  |................|
000000e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00010630  00 00 00 00 00 00 00 00  6c 0a 01 20 00 00 00 00  |........l.. ....|
00010640  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00010650  00 00 00 00 00 00 00 00  00 00 00 80 80 00 00 00  |................|
00010660  00 80 80 00 80 80 80 80  00 00 80 80 00 00 00 80  |................|
00010670  80 80 00 00 00 80 00 00  80 80 80 00 00 80 80 00  |................|
00010680  80 80 00 00 00 80 80 00  00 80 80 80 00 00 80 80  |................|
00010690  80 80 80 00 80 80 80 80  00 00 00 00 80 80 00 80  |................|
000106a0  80 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
:
```

Reviewing our Glitch

- With our setup, the glitch hit around 197 microseconds after the ChipWhisperer triggered.
- In the chip.fail work; their offset was roughly 170 microseconds.
  - Ours had a delay of 24 microseconds, placing it in a similar range to the previous research
  - (197-24 = 173).
- This offset range was repeatable, and we could consistently trigger the glitch in the 194-197 microseconds range.

Glitch Outcomes

- We now can read out the wallet's SRAM
- On older firmware versions this allowed for keys to be recovered
  - Allowing access to the wallet's contents
- This bug has been fixed and the keys are no longer available in RAM.

Technical Lessons Learned

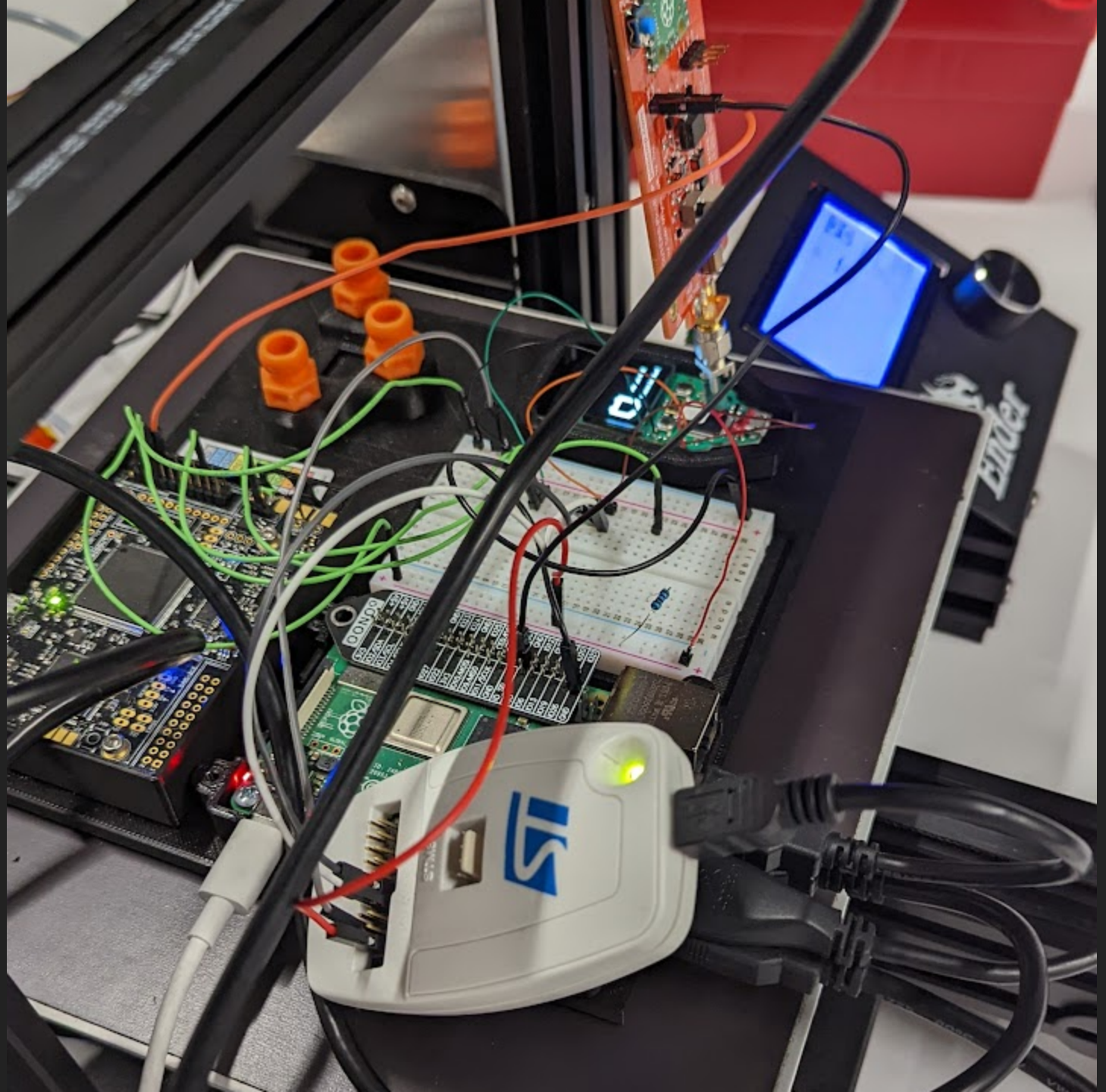
- Fault injection is difficult and progress is hard to measure
- Double and triple check your timings with an Oscilloscope
  - Microseconds matter
  - Millivolts matter
- Test your glitching loop with an unlocked device first

Conclusion

- Many technical hurdles that can occur when attempting to replicate a fault injection attack
- Demonstrated the process of using fault injection to bypass an RDP check
- Small details matter when replicating a FI attack, and these details can change based on your hardware setup.

Questions?

- Reach out on twitter with any questions
- Code / resources can be found here



References / Resources

- Thomas Roth (stacksmashing) FI Talk
- wallet.fail
- chip.fail
- Colin's Blog
- Chipwhisperer Tutorials
- Kraken Blog
- Joe Grand's Wallet Hacking
- STM32F2 Power Information
- STM32F2 Datasheet

Tools Used

1. Chipwhisperer Lite
2. Python environment capable of installing the chipwhisperer package
    - `pip install chipwhisperer`
3. Trezor One
4. STLink Programmer
5. Oscilloscope
    i. We used a Siglent Technologies SDS1104X-E
6. Soldering equipment for component removal, etc