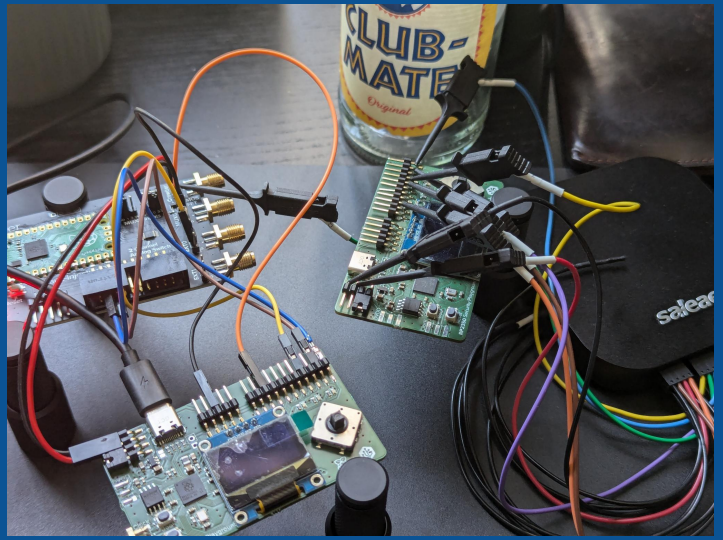
When boot vectors turn into attack vectors

Overcoming RP2350's secure boot chain with fault injection



Embedded Security Village @ Defcon33

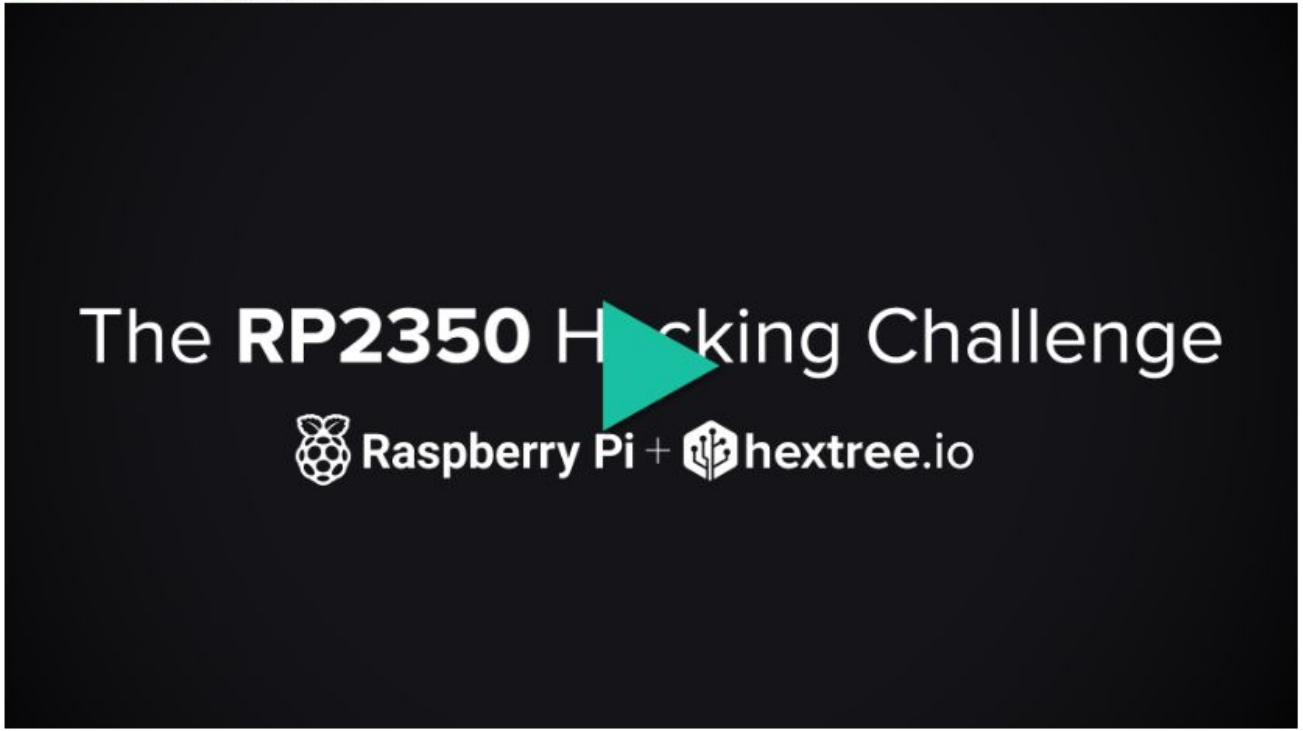
Context

README

RP2350 Hacking Challenge

Welcome to the Raspberry Pi RP2350 hacking challenge and bug bounty!

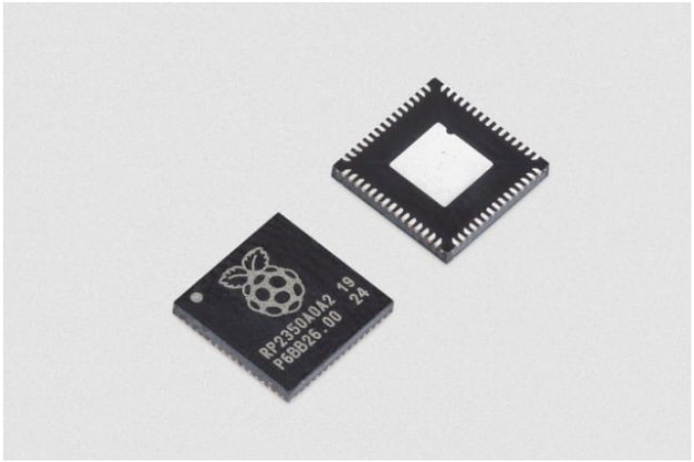
Watch our quick explainer video:

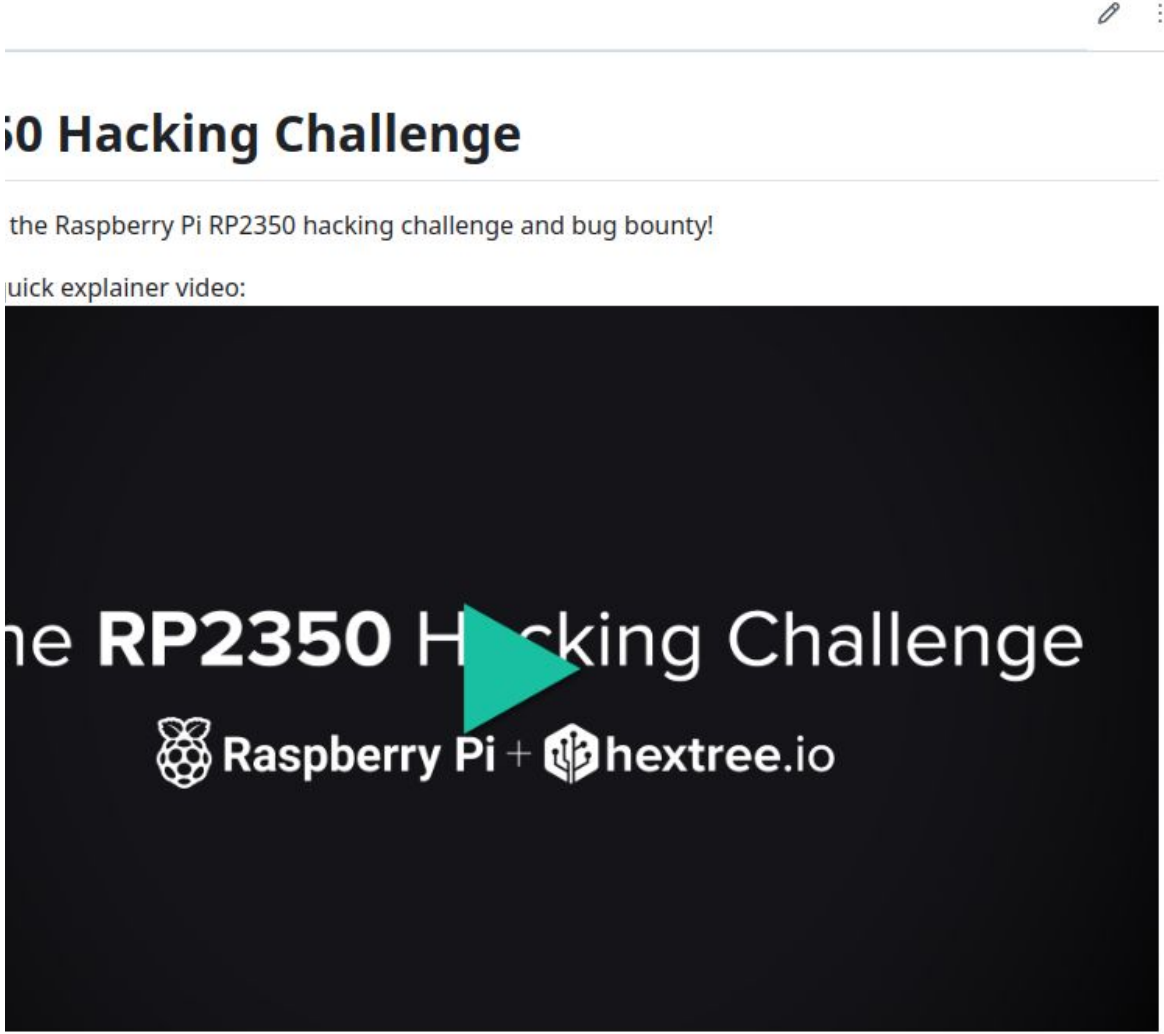


Security through transparency: RP2350 Hacking Challenge results are in

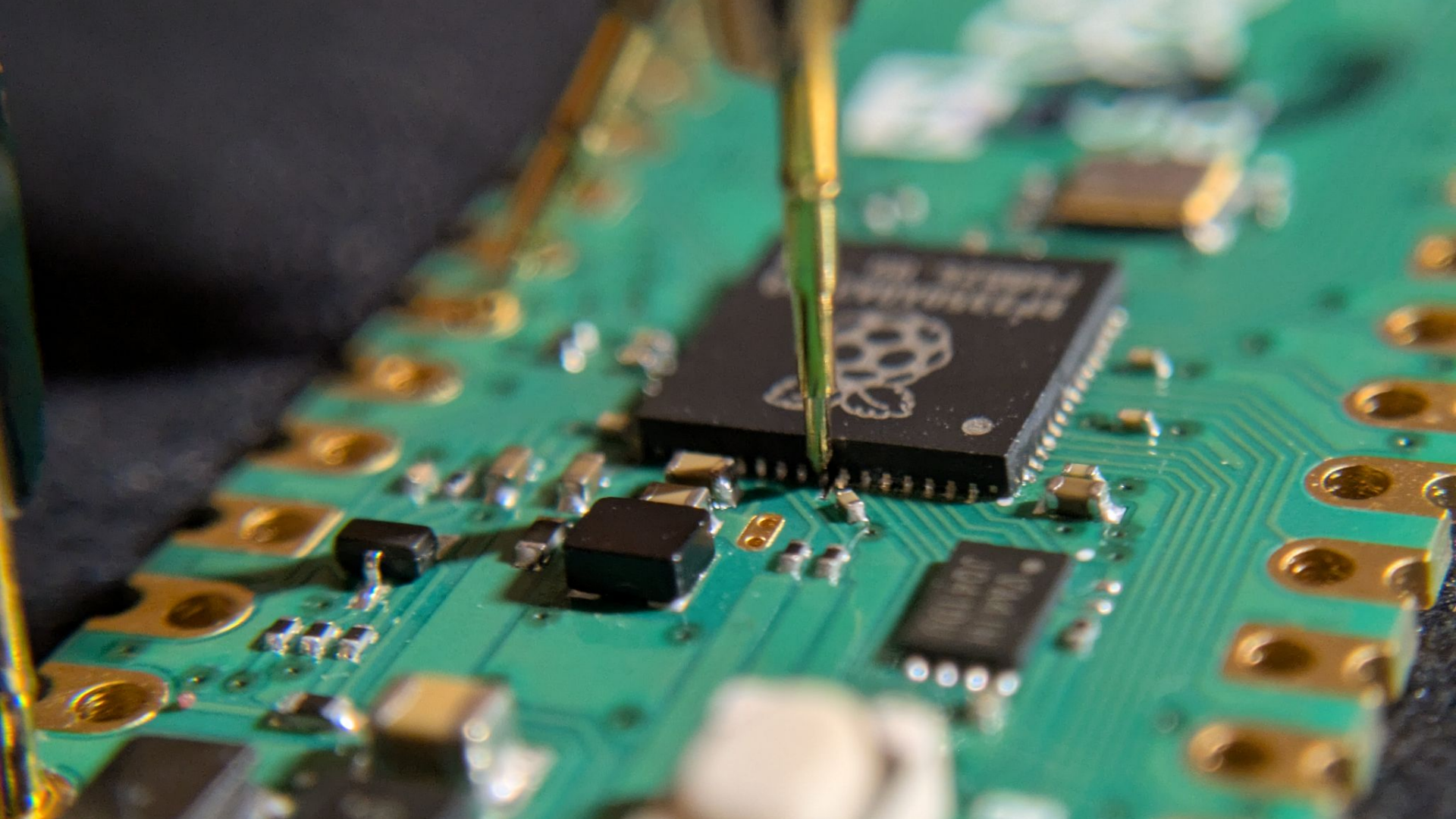
14th Jan 2025 Eben Upton 16 comments

We launched our second-generation microcontroller, RP2350, in August last year. Building on the success of its predecessor, RP2040, this adds faster processors, more memory, lower power states, and a security model built around Arm TrustZone for Cortex-M. Alongside our own Raspberry Pi Pico 2 board, and numerous partner boards, RP2350 also featured on the DEF CON badge, designed by Entropic Engineering, with firmware by our friend Dmitry Grinberg.

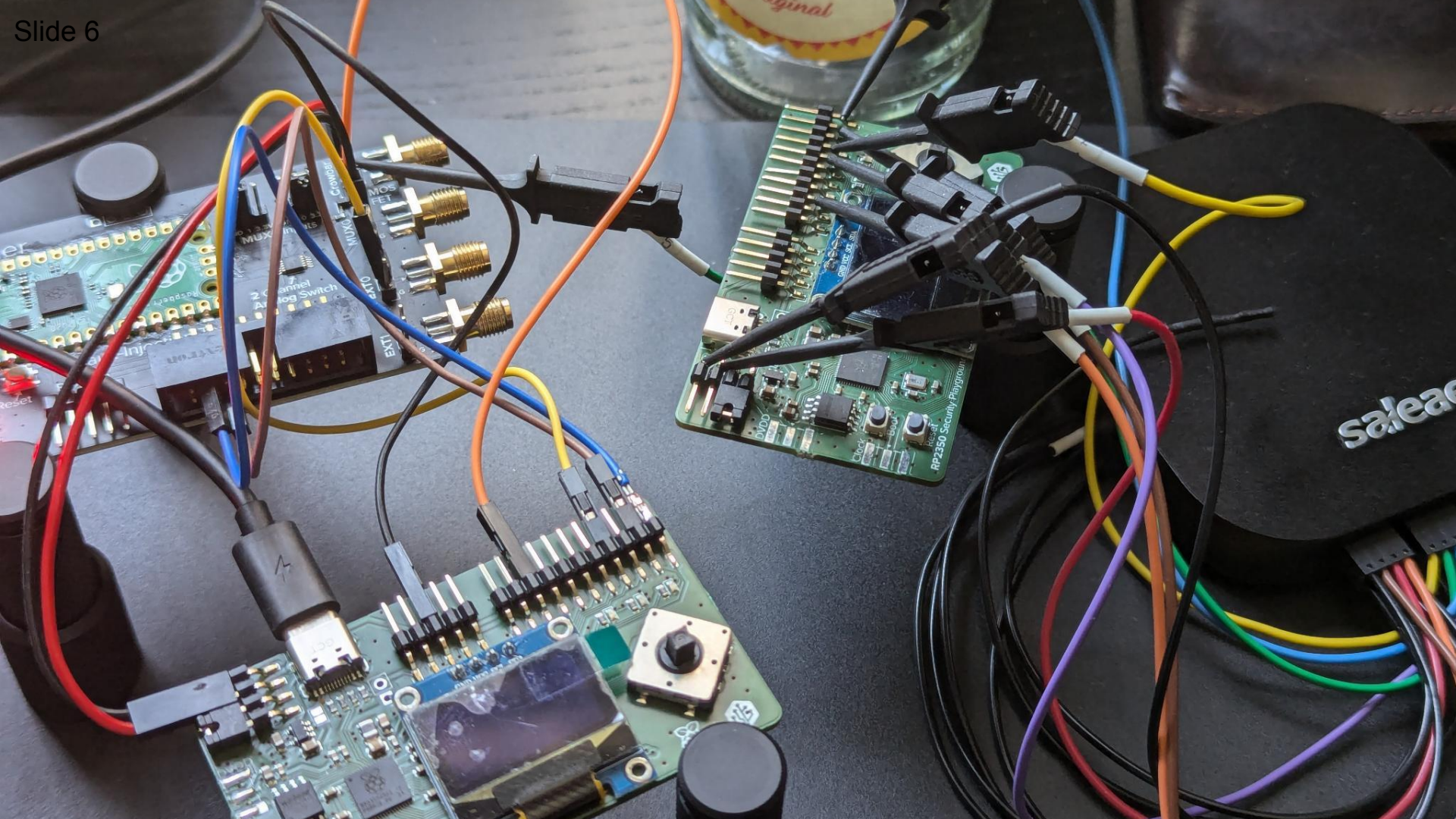


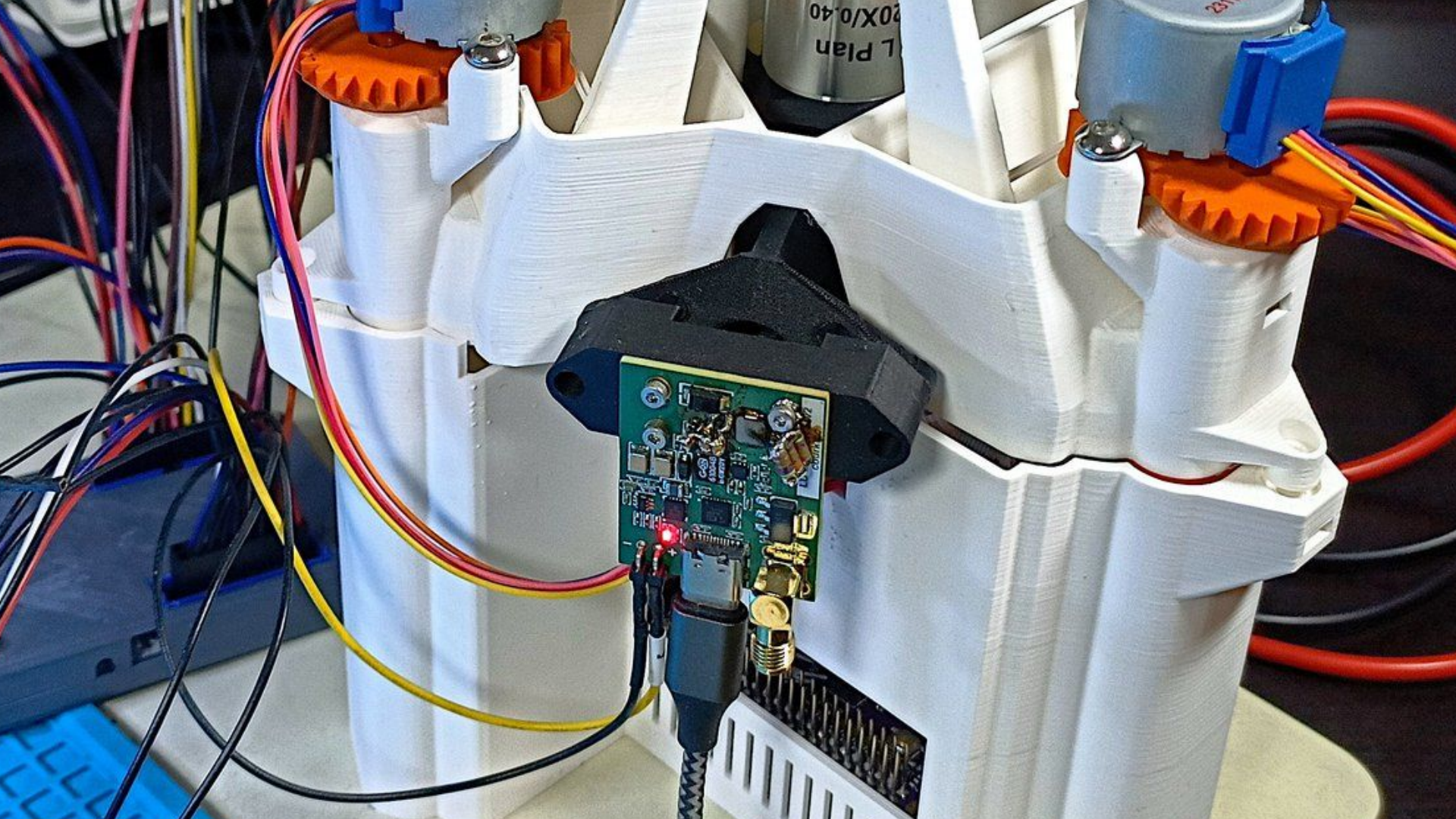


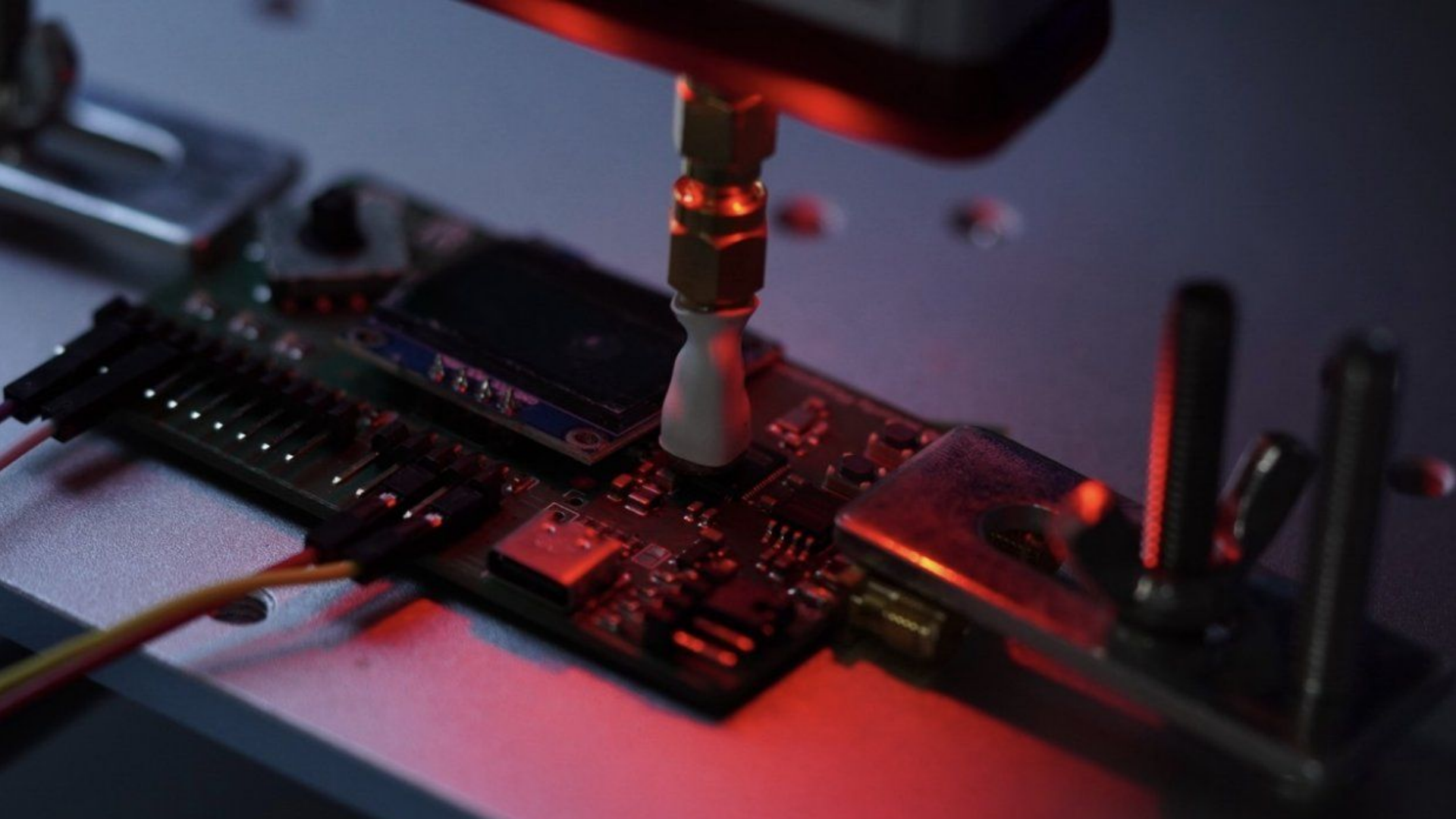
The Attacks



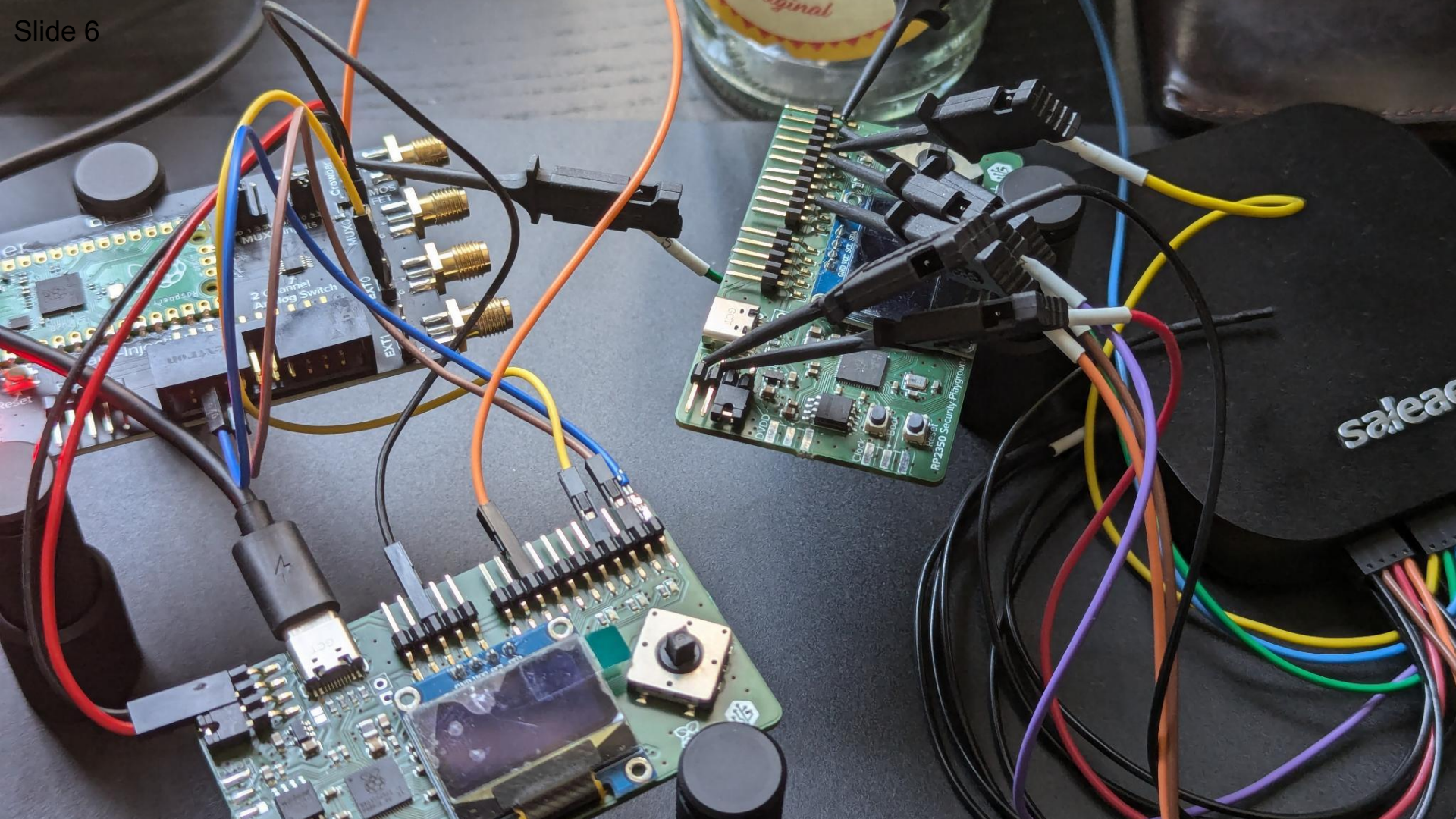
Slide 6





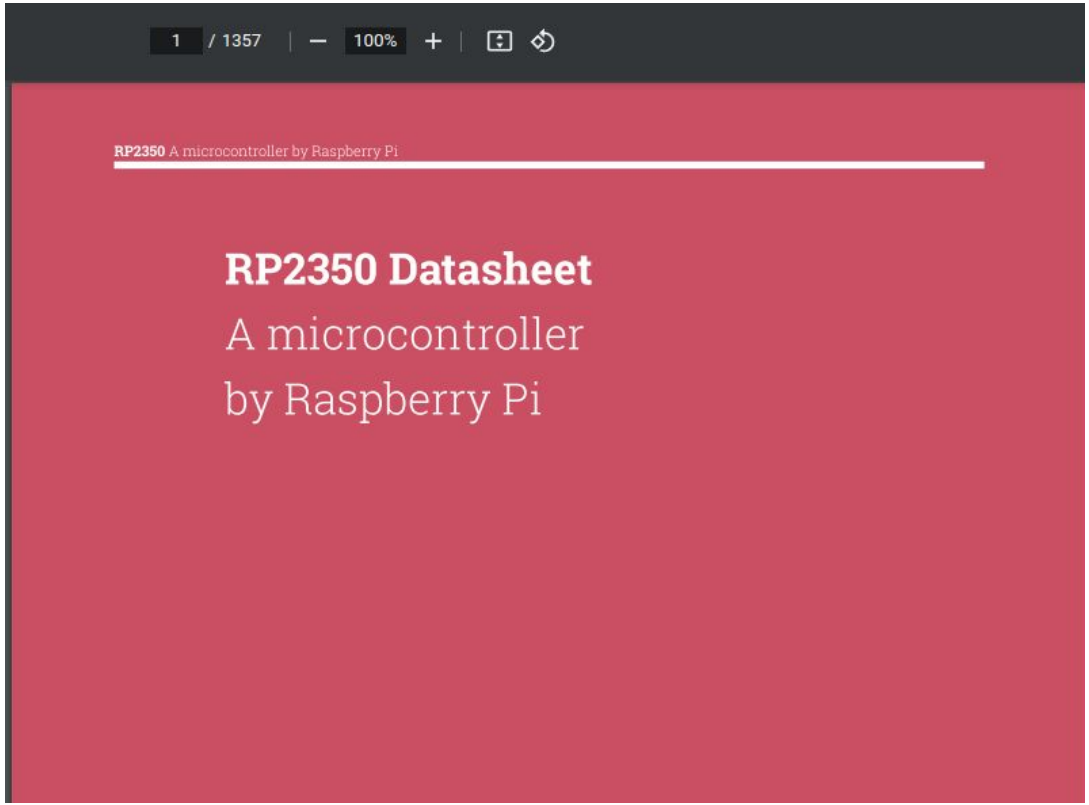


Slide 6



The Attacks

First Approach



First Intuition

5.2.3. POWMAN Boot Vector

POWMAN contains scratch registers similar to the watchdog scratch registers, which persist over power-down of the switched core power domain, in addition to most system resets. These registers allow users to install their own boot handler, and divert control away from the main boot sequence on non-POR/BOR resets. It recognises the following values written to BOOT0 through BOOT3:

- BOOT0: magic number `0xb007c0d3`
- BOOT1: Entry point XORed with magic `-0xb007c0d3` (`0x4ff83f2d`)
- BOOT2: Stack pointer
- BOOT3: Entry point

Use this to vector into code preloaded in RAM which was retained during a low-power state.

If either of the magic numbers mismatch, POWMAN vector boot does not take place. If the numbers match, the bootrom zeroes BOOT0 before entering the vector, so that the behaviour does not persist over subsequent reboots.

The POWMAN boot vector is permitted to return. The boot sequence continues as normal after a return from POWMAN vector boot, as though the vector boot had not taken place. There is no requirement for the vector to preserve the global pointer (`gp`) register on RISC-V. Use this to perform any additional setup required for the boot path, such as issuing a power-up command to an external QSPI device that may have been powered down (e.g. via a B9h power-down command).

The entry point (`pc`) must have the LSB set on Arm (the Thumb bit) and clear on RISC-V. If this condition is not met, the bootrom assumes you have passed a RISC-V function pointer to an Arm processor (or vice versa) and hangs the core rather than continuing, since executing code for the wrong architecture has spectacularly undefined consequences.

The linker should automatically set the Thumb bit appropriately for a function pointer relocation, but this is something to be aware of if you pass hardcoded values such as the base of SRAM: this is correctly passed as `0x20000001` on Arm (Thumb bit set) and `0x20000000` on RISC-V (no Thumb bit, halfword-aligned).

5.2.4. Watchdog Boot Vector

Watchdog boot allows users to install their own boot handler, and divert control away from the main boot sequence on non-POR/BOR resets. It recognises the following values written to the watchdog's upper scratch registers:

- SCRATCH4: magic number `0xb007c0d3`
- SCRATCH5: entry point XORed with magic `-0xb007c0d3` (`0x4ff83f2d`)
- SCRATCH6: stack pointer
- SCRATCH7: entry point

If either of the magic numbers mismatch, watchdog boot does not take place. If the numbers match, the Bootrom zeroes SCRATCH4 before transferring control, so that the behaviour does not persist over subsequent reboots.

Watchdog boot can also be used to select the bootrom's special one-shot boot modes, described in Section 5.2.4.1. The term **one-shot** refers to the fact these only affect the next boot (and not subsequent ones) due to the bootrom clearing SCRATCH4 each boot. These boot types are encoded by setting a special entry point (`pc`) value of `0xb007c0d3`, which is otherwise not a valid entry address, and then setting the boot type in the stack pointer (`sp`) value. Section 5.2.4.1 lists the supported values.

The watchdog boot vector is permitted to return. The boot path continues as normal when it returns: use this to perform any additional setup required for the boot path, such as issuing additional commands to an external QSPI device. On RISC-V the vector is permitted to use its own global pointer (`gp`) value, as the bootrom only uses `gp` during USB boot, which installs its own value.

With the exception of the magic boot type entry point (`0xb007c0d3`), the vector entry point `pc` must have the LSB set on Arm (the Thumb bit) and clear on RISC-V. If this condition is not met, the bootrom assumes you have passed a RISC-V function pointer to an Arm processor (or vice versa) and hangs the core rather than continuing.

5.2.4.1. Special Watchdog Boot Types

The magic entry point `0xb007c0d3` indicates a special one-shot boot type, identified by the stack pointer value:

`BOOTSEL`

Selected by `sp = 2`. Boot into BOOTSEL mode. This will be either UART or USB boot depending on whether QSPI `SD1` is driven high (default pull-down selects USB boot). See Section 5.2.8 for more details.

`RAM_IMAGE`

Selected by `sp = 3`. Boot into an image stored in SRAM or XIP SRAM. BOOTSEL mode uses this to request execution of an image it loaded into RAM before rebooting. See Section 5.2.5 for more details.

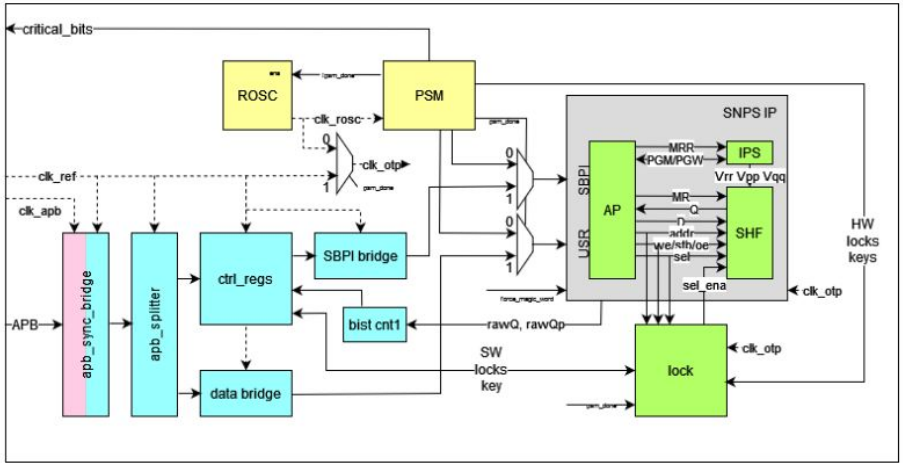
Relevant RP2350 Security Features

- Two cores (Arm+Risc-V), which can be disabled
- One time programmable (OTP) memory
- Glitch Detector
- Redundancy Co-Processor
- Secure Bootchain

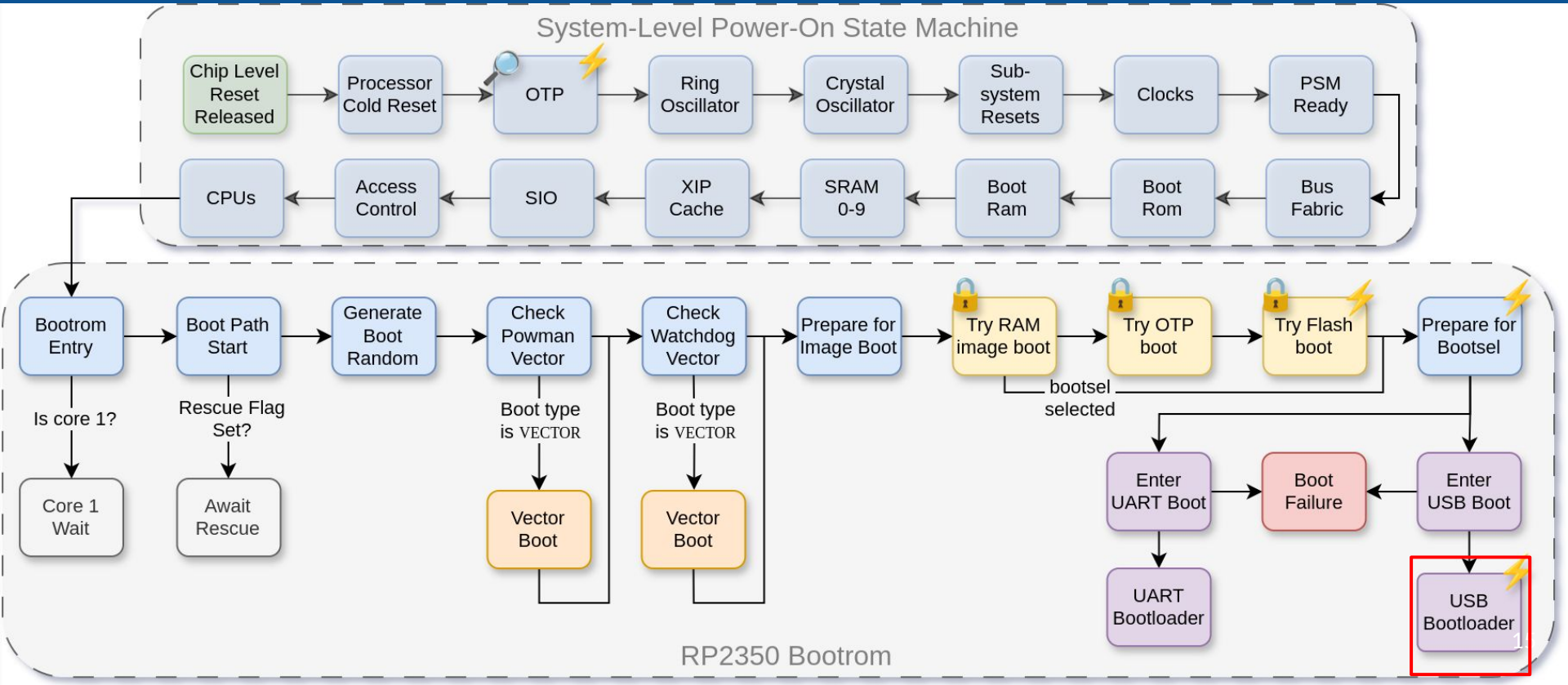
13.3. Background: OTP Hardware Architecture

This diagram shows the integration of the three Synopsys IP components, and the Raspberry Pi hardware added to make this all function in the context of RP2350's system and security architecture. More specifically:

- APB interface(s) to connect to the SoC
- Internal ring oscillator with clock edge randomisation
- Power-up state machine, running off the ring oscillator
- Lock shim, sitting between the SNPS RTL and the memory core (fuse)



RP2350 Boot Process



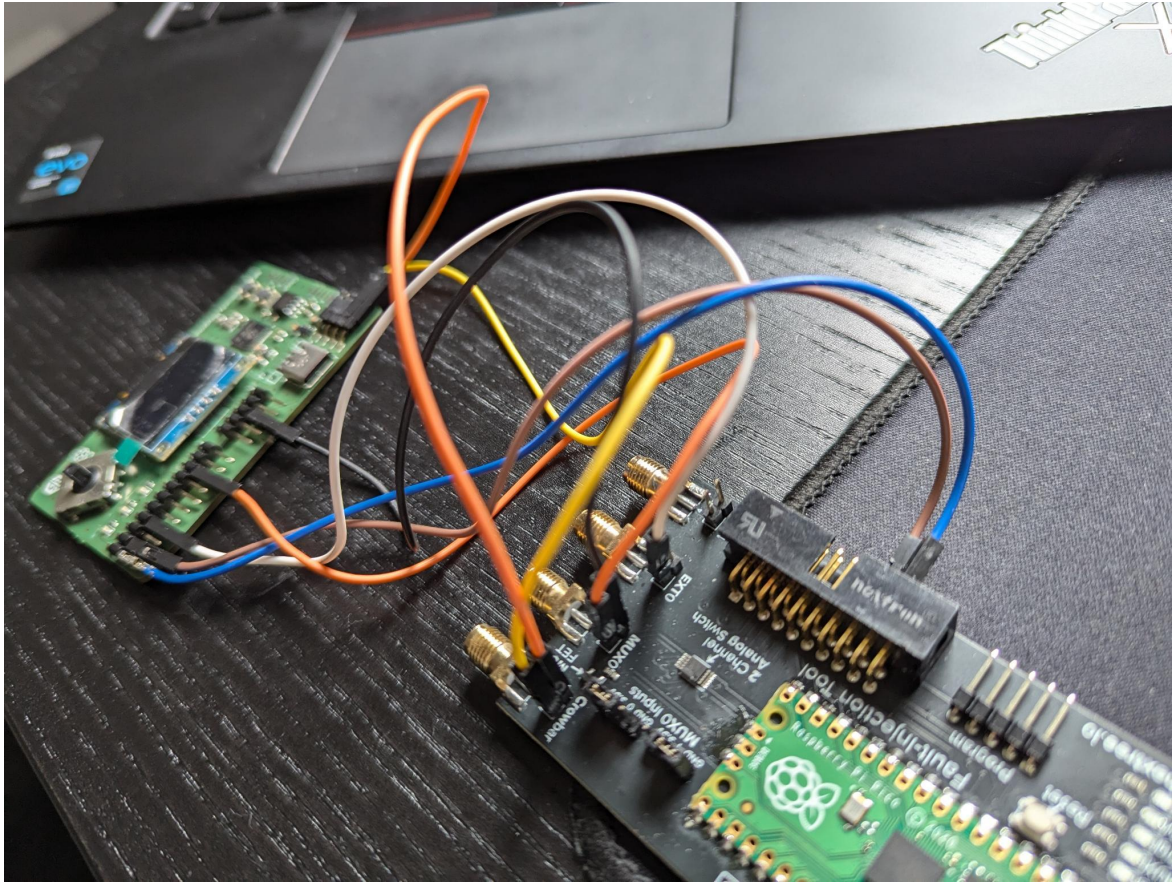
Challenge Setup

Disclaimer

For this challenge we will do the following persistent & irreversible changes to your RP2350:

- Writing bootkey0 (with a public key - or you can generate your own & build your own firmware)
- Enabling secure-boot via `crit1.secure_boot_enable` (but with public keys)
- Disable debug via `crit1.debug_disable`
- Overwrite & lock data in OTP ROW 0xc08
- *Enabling security will permanently disable both Hazard3 RISC-V cores (M33 cores will still be operable)*

First Steps: Faultier + RP2350 Security Playground

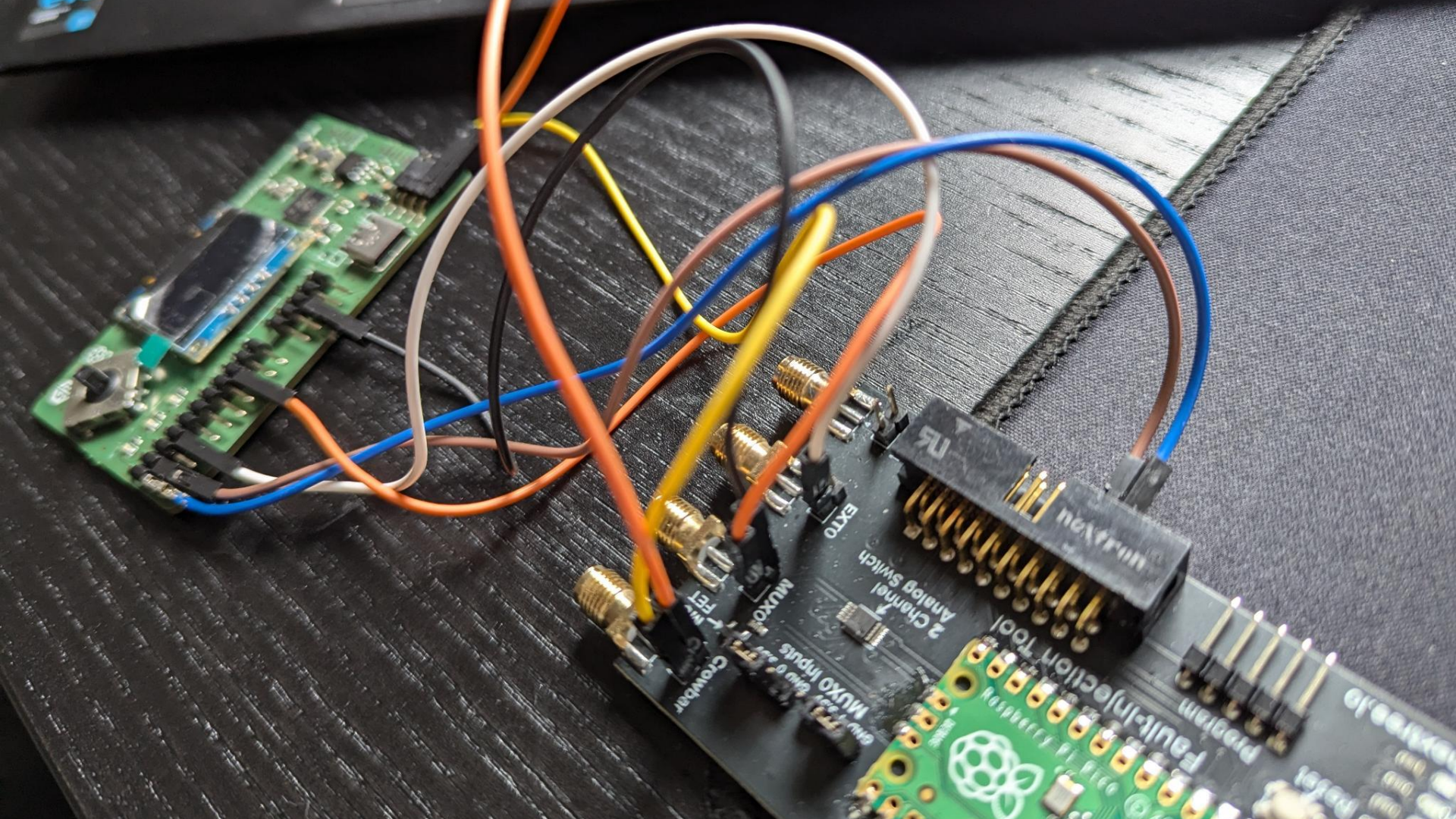


Glitch Detector Mode

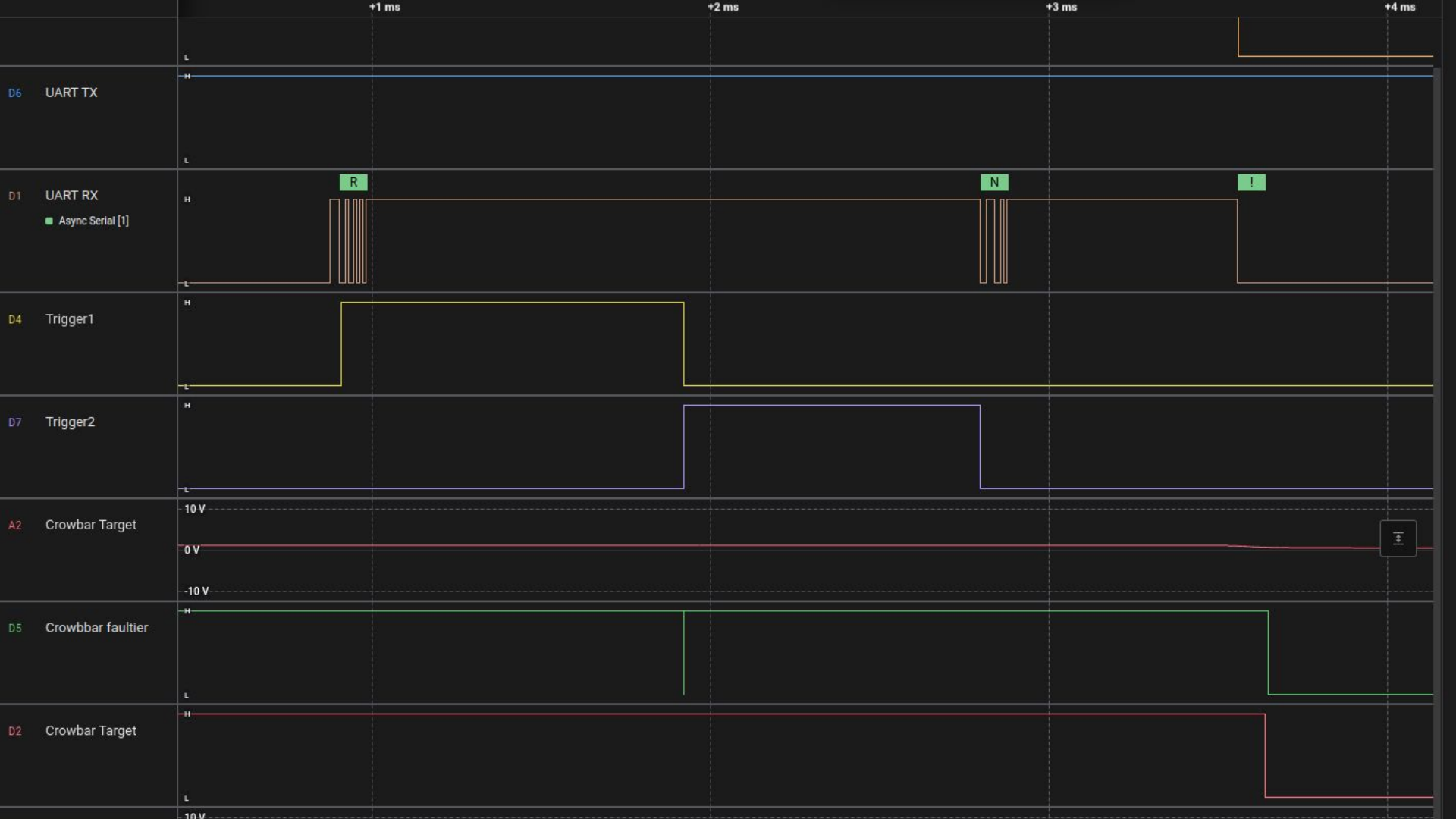
This example exposes a very simple nested loop as a glitch-target. On reset, the target will send "R" to the serial console (with TX on GPIO 12), on regular execution of the loop it will send "N", and on a successful glitch it will send "X".

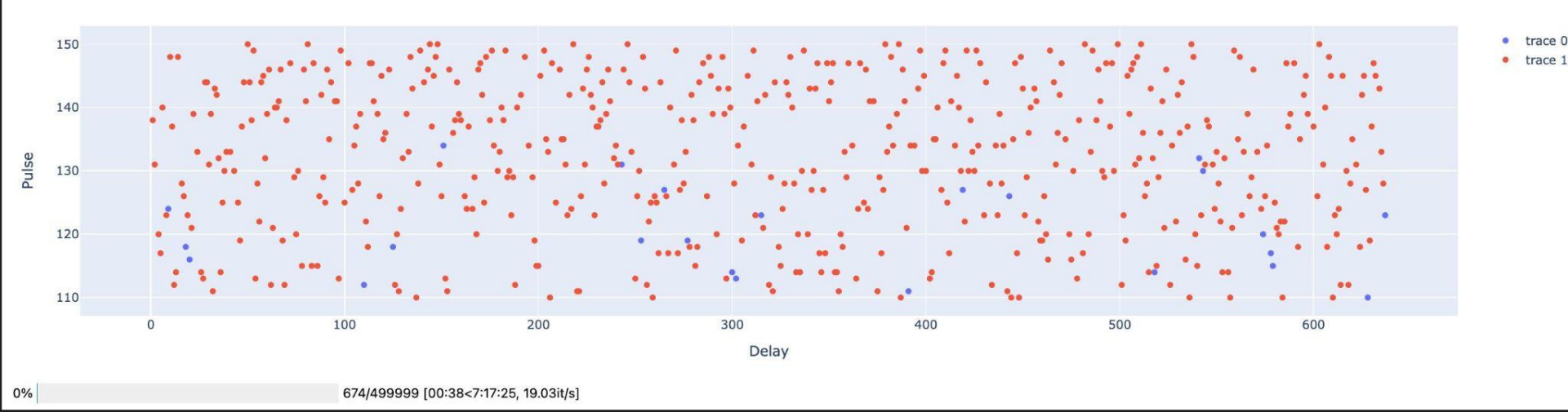
The firmware also activates the watchdog to ensure that an automatic restart occurs if the chip hangs.

```
for(i = 0; i < OUTER_LOOP_CNT; i++) {
    for(j=0; j < INNER_LOOP_CNT; j++){
        cnt++;
    }
}
if (i != OUTER_LOOP_CNT || j != INNER_LOOP_CNT || cnt != (OUTER_LOOP_CNT * INNER_LOOP_CNT)  ) {
    watchdog_update();
    // X indicates successful glitch
    uart_putc_raw(uart0, 'X');
} else {
    // N indicate4s regular execution
    uart_putc_raw(uart0, 'N');
}
```


EXT0
2 Channel
Analog Switch
MUXO
Crowbar
MUXO Inputs
Fault-Injection Tool
Raspberry Pi Pico

RP2350 Security Playground
DVDD
Boot
Reset

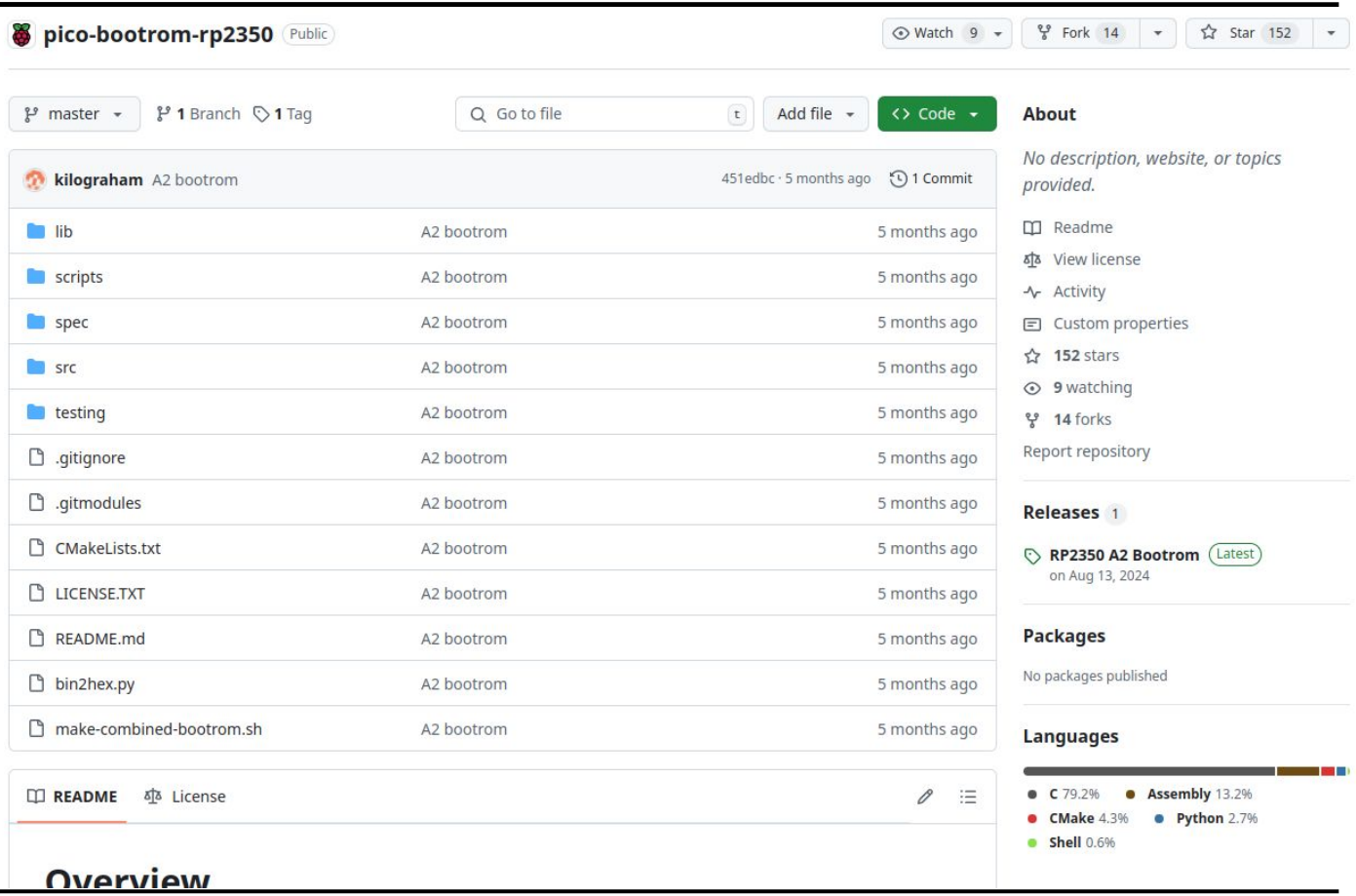

+1 ms
+2 ms
+3 ms
+4 ms
D6 UART TX
D1 UART RX
Async Serial [1]
R
N
!
D4 Trigger1
D7 Trigger2
A2 Crowbar Target
10 V
0 V
-10 V
D5 Crowbbar faultier
D2 Crowbar Target



0% 674/499999 [00:38<7:17:25, 19.03it/s]

```
Successful glitch - Delay: 9 Pulse: 124
Successful glitch - Delay: 18 Pulse: 118
Successful glitch - Delay: 20 Pulse: 116
Successful glitch - Delay: 110 Pulse: 112
Successful glitch - Delay: 125 Pulse: 118
Successful glitch - Delay: 151 Pulse: 134
Successful glitch - Delay: 243 Pulse: 131
Successful glitch - Delay: 253 Pulse: 119
Successful glitch - Delay: 265 Pulse: 127
Successful glitch - Delay: 277 Pulse: 119
Successful glitch - Delay: 300 Pulse: 114
Successful glitch - Delay: 302 Pulse: 113
Successful glitch - Delay: 315 Pulse: 123
Successful glitch - Delay: 391 Pulse: 111
Successful glitch - Delay: 419 Pulse: 127
Successful glitch - Delay: 443 Pulse: 126
Successful glitch - Delay: 518 Pulse: 114
Successful glitch - Delay: 541 Pulse: 132
Successful glitch - Delay: 543 Pulse: 130
Successful glitch - Delay: 574 Pulse: 120
```

Bootrom



pico-bootrom-rp2350 Public

Watch 9 Fork 14 Star 152

master 1 Branch 1 Tag Go to file t Add file Code

kilograham A2 bootrom 451edbc · 5 months ago 1 Commit

lib	A2 bootrom	5 months ago
scripts	A2 bootrom	5 months ago
spec	A2 bootrom	5 months ago
src	A2 bootrom	5 months ago
testing	A2 bootrom	5 months ago
.gitignore	A2 bootrom	5 months ago
.gitmodules	A2 bootrom	5 months ago
CMakeLists.txt	A2 bootrom	5 months ago
LICENSE.TXT	A2 bootrom	5 months ago
README.md	A2 bootrom	5 months ago
bin2hex.py	A2 bootrom	5 months ago
make-combined-bootrom.sh	A2 bootrom	5 months ago

README License

Overview

About

No description, website, or topics provided.

Readme
View license
Activity
Custom properties
152 stars
9 watching
14 forks
Report repository

Releases 1

RP2350 A2 Bootrom Latest
on Aug 13, 2024

Packages

No packages published

Languages

C 79.2% Assembly 13.2%
CMake 4.3% Python 2.7%
Shell 0.6%

USB Boot Loader

- Runs in NS

- Implements the interaction with Picotool

- Custom command packets, documented in Picotool and RP2350 manual
  - One of them: Reset
  - Also available as API call in bootroom 🤔

Reboot Types?

- `0x000d` : `REBOOT_TYPE_PC_SP` - reboot to a specific PC and SP. Note: this is not allowed in the `Arm-NS` variant.
  - `p0` - the initial program counter (PC) to start executing at. This must have the lowest bit set for Arm and clear for RISC-V
  - `p1` - the initial stack pointer (SP).

Reset Cmd Boot Flow

- NS: Receive command &
  jump to S entry with reboot-flag

- S: Sanitize parameters
  (including boot-type)
  - FI hardened!

- S: Jump to shared reset code
  - Verify boot type registers are matching
  - Execute requested boot

```
varm_misc.S:

// Redundant args are passed in r0/*sp, and we want them to mismatch when
   // bit 3 is set (set for all Secure-only reboot types).

   // first check that bit 3 isn't set (since that shouldn't be calling from NS)
   lsls r4, r0, #29
   bcs fail_reboot

   // now, try quite hard to clear bit 3 in r0, so that if bit3 were set in our
   // now-stacked 5th argument to s_varm_hx_reboot, the value in r0 won't match
   // and we'll get an rcp_ violation in the callee

   // 1. put r0 back together without bit 3
   lsrs r4, r4, #29
   lsrs r0, r0, #4
   lsls r0, r0, #4
   orrs r0, r4
   // 2. clear bit 3
   movs r4, #8
   bics r0, r4
   bl s_varm_hx_reboot
   b reboot_retur
```

```
int s_varm_hx_reboot
       00000578 f7 b5           push       {r0,r1,r2,r4,r5,r6,r7,lr}
       0000057a 14 fe 34 47     mrc2       p7,0x0,r4,cr4,cr4,0x1
       0000057e 01 94           str        r4,[sp,#0x4]=>__stack_canary_value
       00000580 08 9c           ldr        r4,[sp,#0x20]=>flags2
       00000582 44 ec 70 07     mcrr       p7,0x7,r0,r4,cr0
       00000586 00 25           movs       r5,#0x0
       00000588 02 26           movs       r6,#0x2
       0000058a 26 4c           ldr        r4,[DAT_00000624]                                = 400D8000h
       0000058c 76 42           rsbs       r6,r6
       0000058e 25 60           str        r5,[r4,#0x0]=>DAT_400d8000
       00000590 25 4d           ldr        r5,[DAT_00000628]                                = 40018000h
       00000592 ae 60           str        r6,[r5,#offset watchdog_disable]
       00000594 0f 25           movs       r5,#0xf
       00000596 32 36           adds       r6,#0x32
       00000598 05 40           ands       r5,r0 // R0 holds flags, so R5 now flags & 0xf (BOOT_TYPE)
       0000059a 06 40           ands       r6,r0 // R0 holds flags, so R5 now & 0x32 (BOOT_FLAGS:  REBOOT2_FLAG_REBOOT_TO_RISCV)
  -->  0000059c 0d 2d           cmp        r5,#0xd //0xd -> BOOT_TYPE_PC_SP
  -->  0000059e 09 d1           bne        LAB_000005b4 // branch to normal boot
       000005a0 6e b9           cbnz       r6,LAB_000005be // branch to RISCV handling(?)
                           LAB_000005a2                                  XREF[1]:     000005dc(j)
       /* BOOT_TYPE_PC_SP path */
       000005a2 22 4d           ldr        r5,[DAT_0000062c]                                = B007C0D3h
       000005a4 e5 61           str        r5,[r4,#offset wd_scratch_4]
       000005a6 e5 69           ldr        r5,[r4,#offset wd_scratch_4]
       000005a8 6d 42           rsbs       r5,r5
       000005aa 55 40           eors       r5,r2
       000005ac 25 62           str        r5,[r4,#offset wd_scratch_5]
       000005ae 63 62           str        r3,[r4,#offset wd_scratch_6]
       000005b0 a2 62           str        r2,[r4,#offset wd_scratch_7]
       000005b2 1a e0           b          LAB_000005ea
```

Attack strategy

- Load unsigned firmware in RAM
- Send USB Reset Command with PC/SP to attack code
- Glitch critical instruction

Challenges:

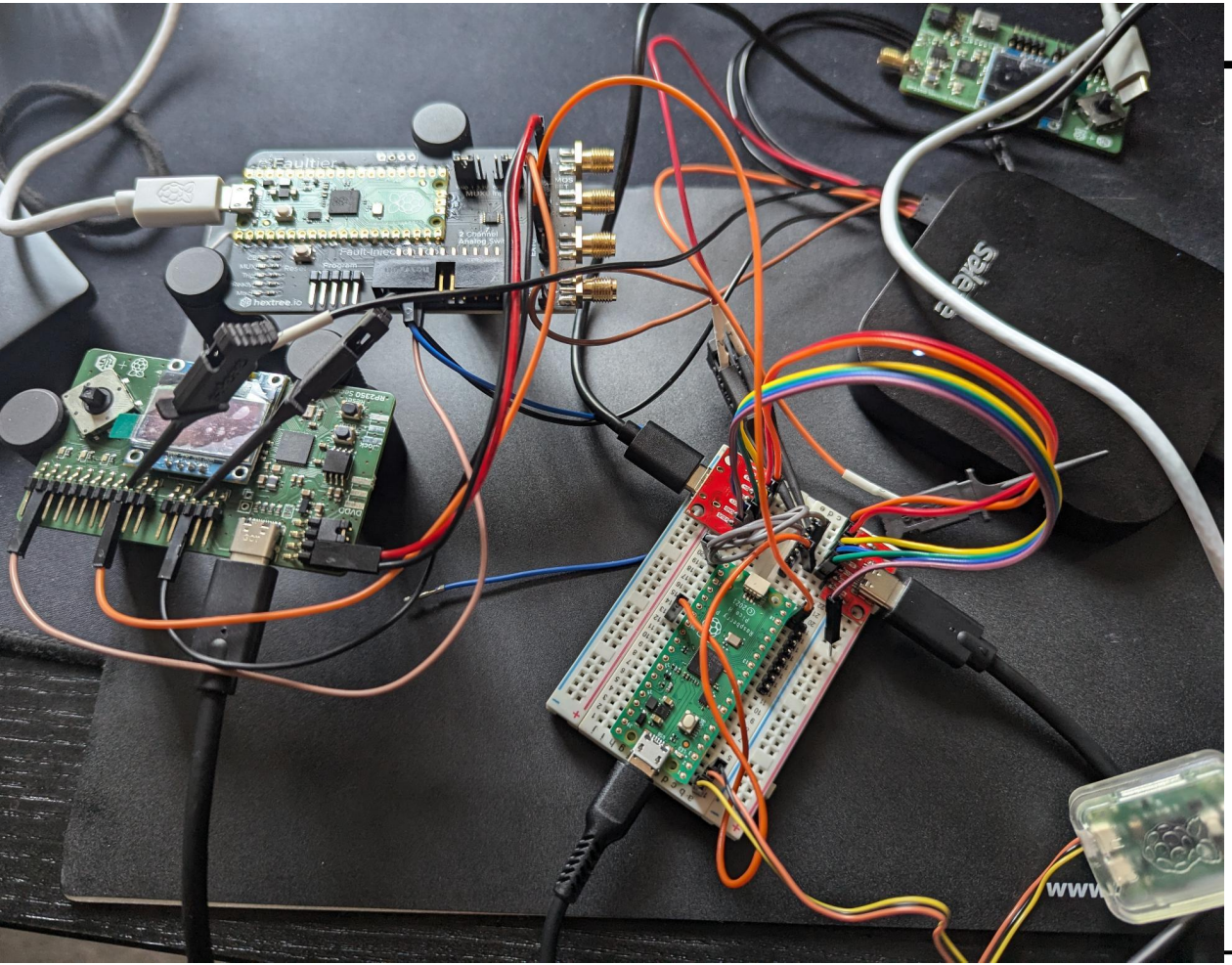
- What to use as attack code?
- Verify feasibility
- Bypass glitch detector
- How to trigger
- How to bypass to random-delay RCP

Prototyping Exploit Code

- Even if we have PC/SP control on boot:
  - Peripherals are not initialized
  - How do we get data out?

- Idea: Just manually run normal VTOR
  - Static variables in SRAM are great
  - No need to modify _entry or Pico build toolchain

```
int main() {
    static int was_here = 0;

    if (was_here == 0){
        uint32_t *vtor = 0xe000ed08;
        *vtor = 0x20000000;

        was_here = 1;
        ((void(*)(void))0x2000019f)();
    }


    init_uart();
    uart_putc_raw(uart0, 'X');
```

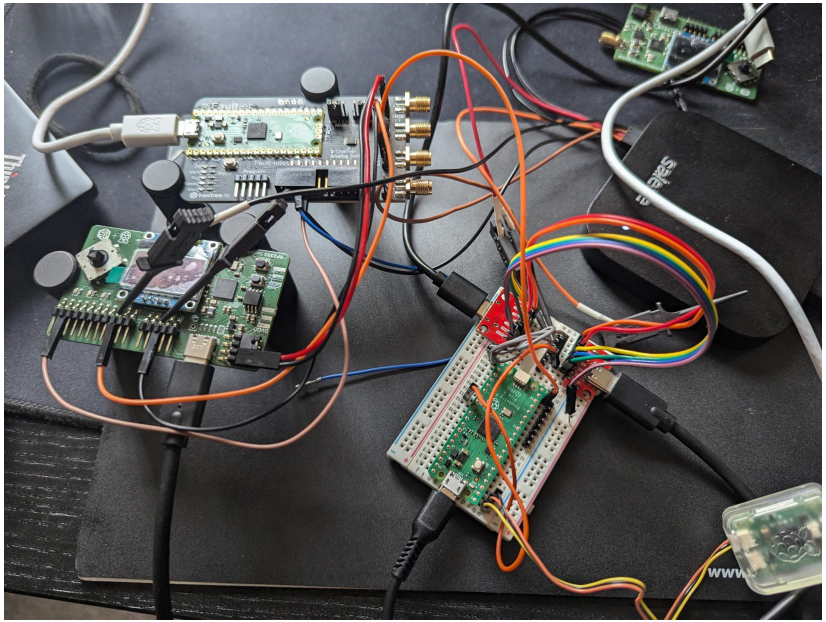


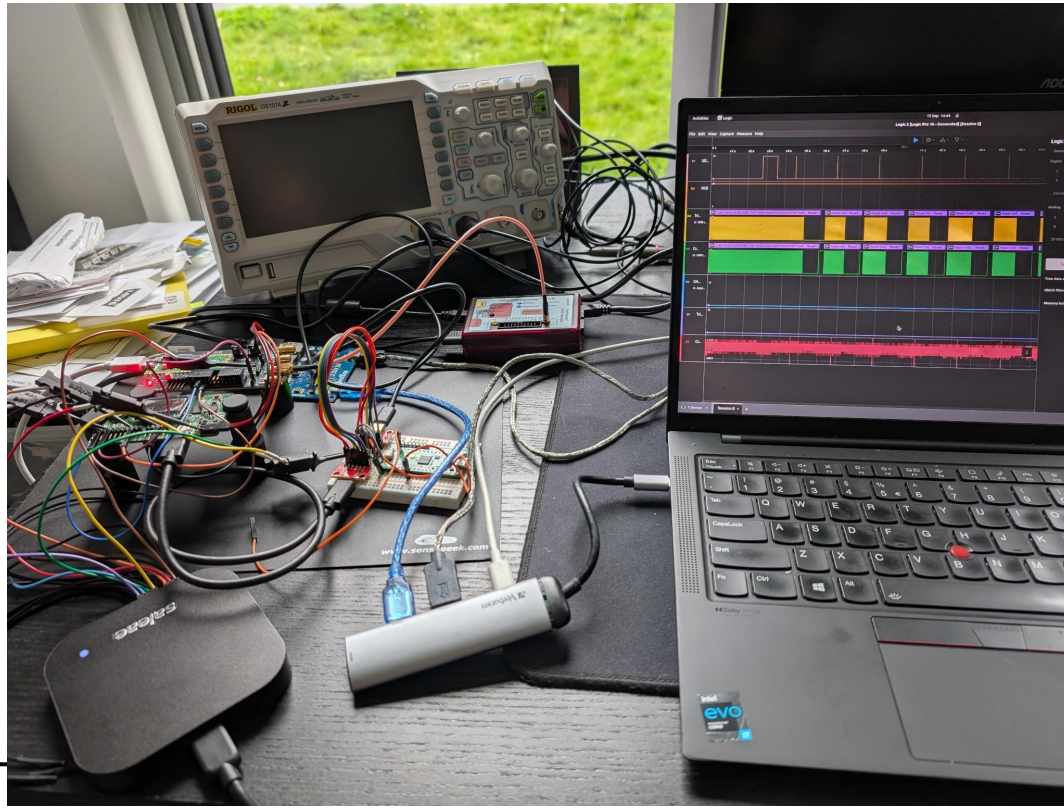
Verify feasibility (Glitch Simulation)

Bypassing the Glitch Detector

```
Successful glitch - Delay: 3917 Pulse: 150
Successful glitch - Delay: 3917 Pulse: 190
Successful glitch - Delay: 3917 Pulse: 230
Successful glitch - Delay: 3917 Pulse: 230
Successful glitch - Delay: 3917 Pulse: 270
Successful glitch - Delay: 3917 Pulse: 150
Successful glitch - Delay: 3917 Pulse: 190
Successful glitch - Delay: 3917 Pulse: 230
Successful glitch - Delay: 3917 Pulse: 270
Successful glitch - Delay: 3917 Pulse: 310
Glitch detector triggered - Delay: 3917 Pulse: 230
Glitch detector triggered - Delay: 3917 Pulse: 230
Glitch detector triggered - Delay: 3917 Pulse: 230
Glitch detector triggered - Delay: 3917 Pulse: 230
Glitch detector triggered - Delay: 3917 Pulse: 270
Glitch detector triggered - Delay: 3917 Pulse: 310
```

Triggering: Self-built trigger vs PhyWhisperer

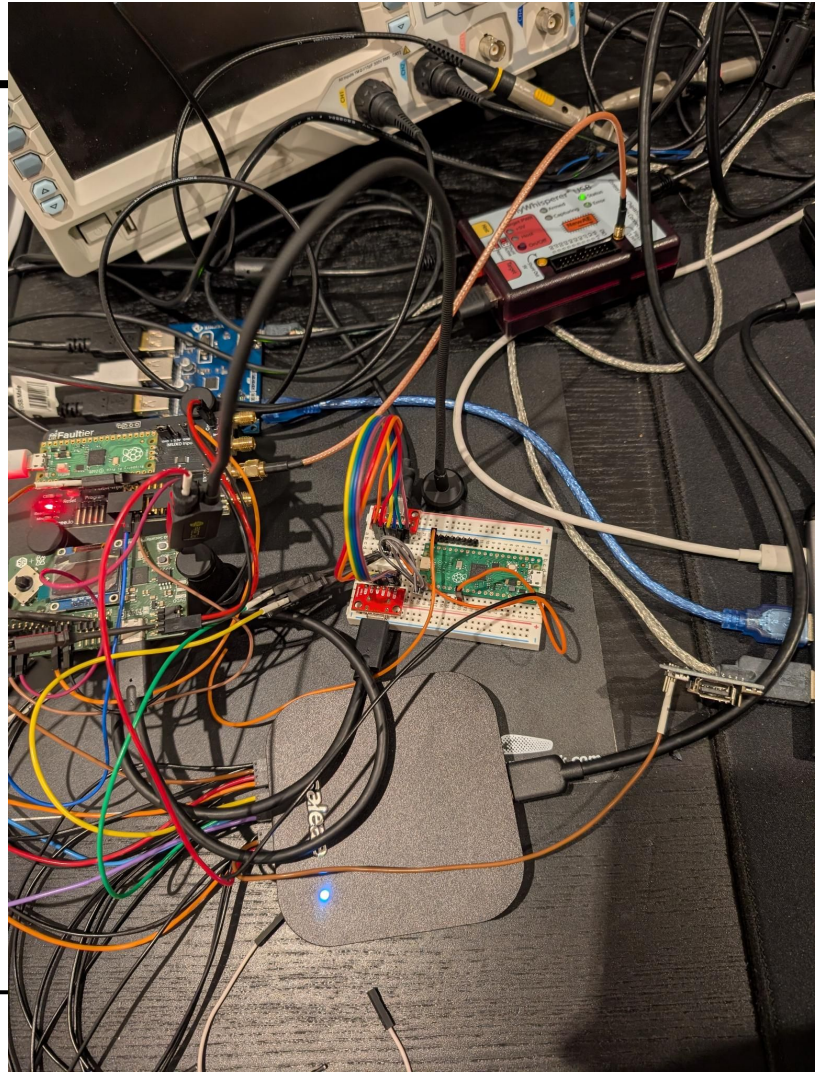




Bypassing Delay RCP

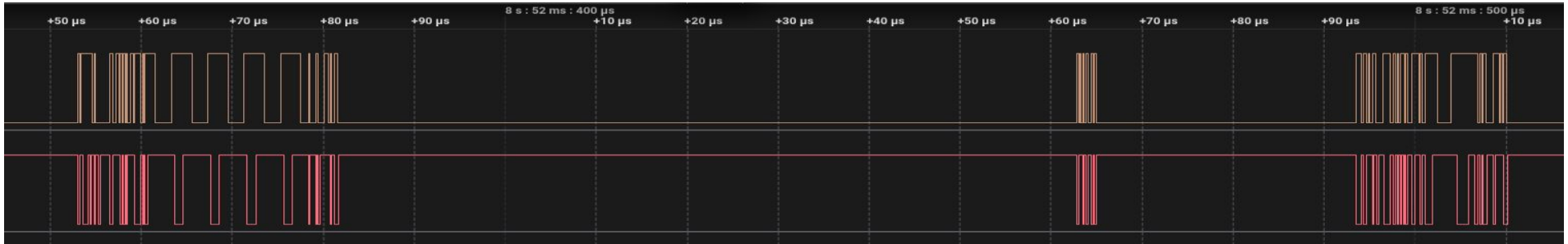
```
.cpu cortex-m33
.text
  .thumb

.global glitch_me
.thumb_func
glitch_me:
// param0: value to put into critical register
mcrr        p7,0x7,r0,r0,cr0
mov         r6, #0
movs        r5,#0xf
add         r6,r6, #0x32
and         r5, r0
and         r6, r0
cmp         r5,#0xd
bne         glitch_no_success1
cbnz        r6, glitch_no_success1

glitch_success:
        mov r0, #43
        bx lr
glitch_no_success1:
        mov r0, #0
        bx lr
glitch_no_success2:
        mov r0, #1
        bx lr
```

Putting it all together



Notable experience along the way

- Travel-safe experiment setup (Thanks PCBite)
- Used cables (Thanks Stacksmashing)
- LA Sample Rate for USB packets (Thanks David)
- “The front fell off”-reset button edition (include GPIO triggerable resets on your design!)



More information

https://github.com/bhamsec/woot25-rp2350-challenge