Security through Transparency: Tales from the RP2350 Hacking Challenge

Marius Muench
University of Birmingham

Aedan Cullen
Independent

Kévin Courdesses
Independent

Thomas 'stacksmashing' Roth
Hextree

Andrew Zonenberg
IOActive

Abstract

Security of Microcontroller Units (MCUs) is crucial for the modern computing landscape, as they often provide a root-of-trust to larger systems or are embedded in safety-critical applications. To prevent attackers from running unsigned code, many MCUs implement secure boot mechanisms.

One such MCU is the recently released RP2350, which combines secure boot with additional hardware features to protect against fault injection attacks. In this paper, we demonstrate the possibility of fault injection and secret extraction attacks *despite* the presence of dedicated countermeasures.

We showcase five different attacks that break the secure boot guarantees of a locked down RP2350 chip. Our attacks leverage voltage, electromagnetic, and laser fault injection techniques. They allow us to bring back disabled CPU cores and debugging ports, boot unverified firmware images, bypass signature verification checks, and provide unprivileged access to sensitive data. We further demonstrate direct extraction of antifuse memory contents using focused ion beam passive voltage contrast. To improve the MCU security landscape, we propose potential mitigations against our attacks and share our lessons learned with the community.

1 Introduction

Modern society is increasingly dependent on the broad availability of computing systems. While consumers usually consider computers as laptops, desktops, or mobile devices, there is a complex layer of *invisible* embedded devices pervading nearly every aspect of computing. Wi-Fi chipsets, security co-processors, memory controllers, trusted platform modules, SIM cards, and car ECUs—all are examples of embedded systems regulating important aspects of our life.

Although these systems differ in size, complexity, and purpose, they are typically based on Microcontoller Units (MCUs) and run custom firmware. Given the safety- and security-critical nature of many application domains, recent MCUs tend to provide secure boot capabilities. Typical secure boot implementations verify the authenticity of the embedded system's firmware to prevent attackers from running custom, or modified, code. The widespread deployment of secure boot and verified firmware updates underpins their importance. Current surveys estimate that one in three embedded developers utilize secure boot and secure firmware update functionalities in their designs [4].

Unfortunately, most secure boot implementations rely on the fact that the underlying hardware always executes as intended. In the context of embedded systems, this may not always be true. In many cases, attackers may have physical access to the target device allowing for *Fault Injection (FI)* attacks. These attacks can lead to instruction skips, register value corruptions, or other unforeseen side-effects, ultimately subverting the guarantees of secure boot implementations. Building on fault injection capabilities, a variety of previous work demonstrates successful attacks against secure boot loaders on embedded devices (e.g., [13, 15, 37, 44, 48]).

Unsurprisingly, chip manufacturers may implement fault injection mitigation schemes to reduce or eliminate the attack surface. Potential countermeasures include duplication of critical sections, error detection/correction mechanisms, or custom glitch detectors to detect physical attacks [3].

One recently released MCU that incorporates both a secure boot implementation and hardware-level mitigations against FI attacks is the RP2350 from Raspberry Pi Ltd [31]. Among other features, its FI-hardened bootrom, glitch detectors, and redundancy coprocessor are all aimed at mitigating physical attacks. To assess whether these security measures indeed thwart fault injection attacks, the chip manufacturer took an unconventional route: After initial internal and third-party audits, Raspberry Pi announced the *"RP2350 Hacking Challenge"* [23] together with the public release of the chip. This challenge promised a monetary reward for the first demonstration of a *break*, i.e., a successful attack bypassing secure boot and allowing execution of unsigned code to extract secrets from the chip.

In this paper, we highlight the attacks we found over the course of this challenge. In total, we found five different

attacks, leading to an instructive showcase on bypassing contemporary MCU security measures. Our attacks cover voltage, laser, and electromagnetic fault injection, as well as a direct antifuse secret readout using Focused Ion Beams (FIBs). Each attack targets a different stage of the chip's secure boot implementation, including the hardware power-on state machine, the signature verification of signed firmware, and a secondary USB bootloader. All of our attacks have been verified and acknowledged by Raspberry Pi Ltd, and four of them resulted in published errata [31].

Based on our findings, we discuss potential mitigations against the presented attacks and reflect on lessons learned. In particular, we discuss the potential impact of pre-boot attacks, the effectiveness of the hardware mitigations employed by the RP2350, and the dormant dangers of complex bootloader implementations. Furthermore, we elaborate on the practicality of low-cost laser FI platforms and our overall experiences with the RP2350 Hacking Challenge.

In summary, we provide the following contributions:

- We provide an overview of the RP2350 security architecture and its hardened secure boot implementation that takes advantage of hardware countermeasures against FI attacks.
- We show five hands-on proof-of-concept attacks subverting secure boot on the RP2350. Our attacks allow us to either run unsigned code or directly extract secrets stored on the chip.
- We discuss mitigations against our attacks and provide practical insight into the lessons learned.

2 The RP2350 Security Architecture

The RP2350 is a dual core, dual-architecture MCU with 520 KiB of on-chip SRAM and 8 KiB of One-Time Programmable Memory (OTP). Both cores implement an Arm Cortex-M33 and RISC-V Hazard3 processor, and the processor used on each core is selected during boot. In the following, we provide an overview of the RP2350 security architecture and discuss security-relevant hardware and bootrom features[1].

2.1 Overview & Security Goals

The RP2350 uses a hardened bootrom coupled with hardware features to implement three high-level security goals [31]:

(G1) Prevent an attacker from running unauthorized code. This is achieved via a secure boot implementation embedded in the boot ROM.

(G2) Prevent an attacker from reading user code and data. This is achieved by encrypting a secure-boot-enabled firmware and adding a decryption stage to the firmware image. The symmetric decryption keys are stored in OTP.

(G3) Isolate trusted and untrusted software to reduce the impact of compromised applications. This is achieved by leveraging the ARMv8-M Security Extensions of the Cortex-M33 processor in the user firmware in conjunction with RP2350-specific extension to associate DMA channels, GPIOs, and peripherals to the different security domains.

The root of trust for all three goals is the secure boot implementation. To protect against attacks, the RP2350 implements additional hardware features such as a redundancy coprocessor and glitch detectors, which we will highlight in Section 2.2. The configuration for the secure boot process, including the required key material, is stored in OTP. Enabling secure boot will disable the RISC-V cores, as the bootrom implements secure boot only for the Arm cores.

We visualize the power-on and boot process in Figure 1. The main component for secure boot is the signature verification of firmware images to be booted, which is carried out in the 🔒-steps of the boot process. In Section 4, we will showcase various FI attacks subverting secure boot (⚡) and extracting secrets from the OTP (🔎).

2.2 Hardware Features

Cortex-M33 Security Features. The Cortex-M33 processor used in the RP2350 implements TrustZone-M. Thus, it provides a Secure and a Non-Secure world and executes always either in Secure or Non-Secure execution state. The non-secure world can enter into the secure world using `Secure Gateway` instructions, leading to a switch of security domains only at well-defined entry locations.

Memory regions and their access permissions are configured via the Security Attribution Unit (SAU), while peripherals are assigned to the different security domains through bus filtering. Additionally, bootrom API functions may be only available to selected security domains.

One-Time Programmable Memory. The RP2350 includes 8 KiB of OTP organized in 4096 rows of 16 data and 8 parity/ECC bits. Upon chip manufacturing, all bits are initially 0 and can be programmed to 1 by the user. The OTP memory is implemented as an antifuse array using the Synopsys SHF NVM.[2]

Alongside the antifuse array, the OTP block also integrates the following components:

[1] We exclude the description of further software-level protections, as these are application specific and independent of the RP2350's secure boot implementation.

[2] `dwc_nvm_ts40 IP family`.

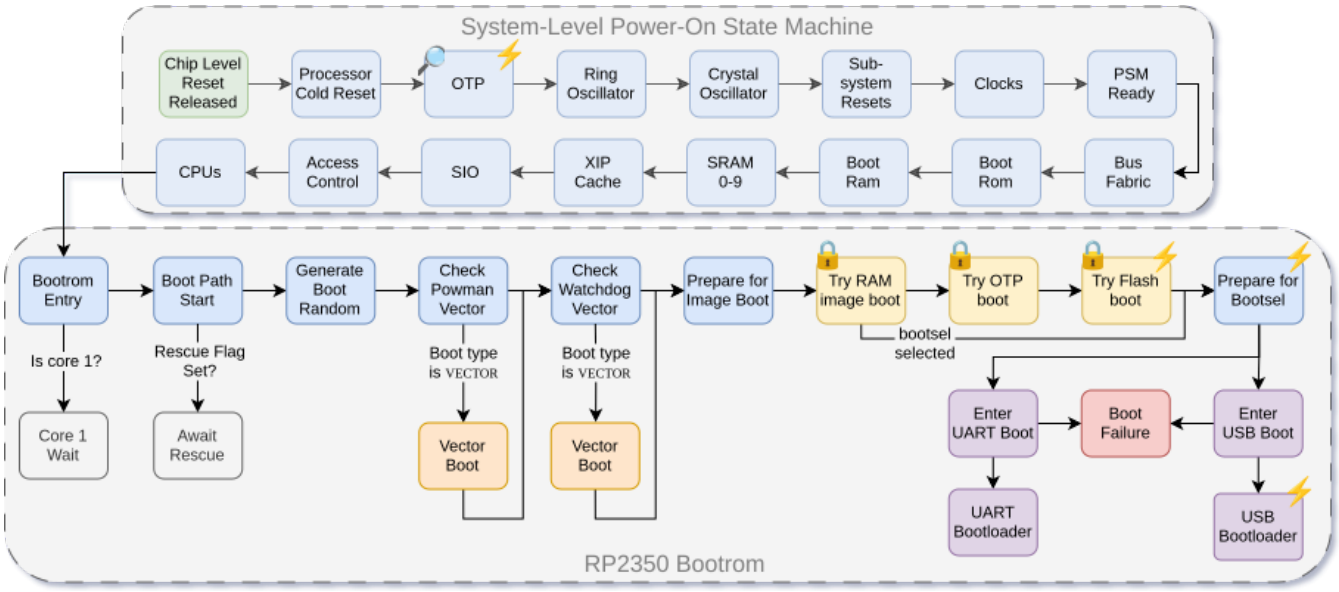

Figure 1: The RP2350 power-up and boot process. The upper parts are executed in hardware by the system-level power-on state machine, while the lower parts are implemented in the boot ROM. Procedures marked with a 🔒 verify the image's signature if secure boot is enabled. ⚡ indicates locations we targeted with our fault injection attacks. 🔎 shows the target for secret extraction.

Power Supply. The Integrated Power Supply (IPS) provides on-chip programming and read voltages for the antifuse array. It is supplied by the external `USB_OTP_VDD` pin.

State Machine. The OTP Power-On State Machine (OTP PSM) is responsible for autonomously performing reads to obtain configuration bits during the reset process of the device.

Ring Oscillator. The OTP PSM is clocked by a local, randomized ring oscillator to hinder external analysis or FI attacks. This oscillator is disabled after the automatic read sequence is complete.

The OTP PSM functions as the hardware root of trust of the system. It reads out all settings needed to establish the early state of the device, called *critical bits*, soon after cold reset. Resulting signals then configure multiple security-related mechanisms, including enabling secure boot, allowing or denying debug access, and selecting the processor architecture. We detail the critical OTP registers in Table 2 and Table 3 in the Appendix.

Glitch Detector. To protect against FI attacks, the RP2350 employs four identical glitch detection circuits in different physical locations. They are based on a pair of D flip-flops that toggle on each clock cycle, with one being combined with a programmable delay on its input path. Under normal operation, the values of these flip flops are always identical. Differing values indicate manipulation of the clock or rapid changes in the core supply voltage. Thus, as soon as a detector senses different values, a system reset is performed.

The glitch detectors are disabled by default and can be enabled via two interfaces: (1) the `GLITCH_DETECTOR_ENABLE` flag in OTP, or (2) through the corresponding Memory-Mapped Input/Output (MMIO) register. An additional `SENSITIVITY` register allows the delay to be configured from 75% to 120% of the minimum system clock period, allowing the glitch detectors to trigger even on smaller fluctuations. To avoid manipulation of the glitch detector setup from untrusted software, all associated MMIO registered are only readable and writeable from the Secure world.

Redundancy Coprocessor. Both Cortex-M33 processors of the RP2350 are equipped with a Redundancy Coprocessor (RCP) instance. To mitigate FI and Return-Oriented Programming (ROP) attacks, an RCP instance provides the following hardware-assisted security features:

Stack Canaries. The RCPs utilize a per-boot random seed to derive stack canaries. They provide explicit instructions for retrieving and validating the canary value. The stack canaries aim to mitigate ROP attacks.

Instruction Delays. RCP instructions can optionally induce a pseudorandom 0-127 cycle delay. Adding delays at critical points complicates FI attacks due to unpredictable timings.

Sequence Count Checking ensures that a number of instructions happen in the right order. Once initialized, each

subsequent check asserts correctness of the sequence counter and increments it by one.

Integer & Boolean Validation. The RCPs allow asserting desired boolean relationships between two registers, as well as checking register values for specific magic constants. This mitigation can hamper FI attacks by ensuring that register values have not been manipulated in unforeseen ways (e.g. by instruction skipping).

Panic. In a non-recoverable error state, the processor can issue an RCP panic instruction, preventing the processor from executing further instructions.

The underlying RCP instructions are heavily used in the critical sections of the RP2350 bootloader to harden the system and provide strong security guarantees for its secure boot implementation.

2.3 The RP2350 Bootrom

The bootrom provides the secure boot implementation for the RP2350 and resides in the chip's mask ROM. Its implementation is publicly available, both in source and in binary form, allowing for transparent security analysis by third parties [29].

Boot Sequence. We depict the boot sequence in the lower part of Figure 1. The bootrom starts by advising core 1 to wait, while clearing the boot RAM and continuing to execute the main boot logic on core 0. It then checks whether the rescue flag is set, before generating a boot-time random value using the RP2350's True Random Number Generator (TRNG). This value is used to seed the RCP. Afterwards, the boot sequence checks the power manager and watchdog scratch registers to check whether a *vector boot* was requested (i.e., a boot with a set `pc`/`sp` pair). In the case of a vector boot, the target code is executed *without* image verification and, upon its return, the bootrom continues with the boot sequence.

Next, the bootrom prepares for booting regular images and checks for the presence of a valid image in RAM, OTP, or flash memory depending on the requested boot type. In the case that secure boot is enabled, an image must contain a matching signature to be considered valid. If a valid image is found, it is entered, effectively terminating the bootrom's execution and transferring control to the code of the image.

Additionally, if a BOOTSEL request was detected or flash boot is unsuccessful, bootrom execution will enter the BOOTSEL stage, which provides the choice between two interactive bootloaders. If either of these are selected, but disabled (e.g. via OTP configuration), the boot sequence reaches the *Bootrom Failure* stage. Otherwise, the selected bootloader is started and allows for further user interaction.

Signature Verification. The RP2350 uses a secp256k1 ECDSA elliptic curve cipher for the creation and verification of signatures. The signature of an image is a signed SHA-256 hash over the image's contents, and the fingerprints of allowed public keys are stored in OTP. Images are expected to include both the signature and the public key used for signing.

To verify the signature, the bootrom first creates the image's SHA-256 hash. Then, it verifies that the signature embedded in the image correctly decrypts to a hash under the specified public key, and that the decrypted value is equal to the calculated hash. Lastly, the bootrom verifies that the public key used matches one of the fingerprints stored in OTP. Only if all four checks succeed is the signature considered valid.

Table 1: Security-relevant bootrom API functions reachable from the Non-Secure execution state.

API Function	Short Description	NS	S
`flash_op`	Perform a flash read, erase, or program operation	✓	✓
`otp_access`	Writes or reads data to/from OTP	✓	✓
`reboot`	Resets the RP2350 and uses the watchdog facility to restart	✓	✓
`secure_call`	Call a Secure method from Non-Secure code	✓	

Bootrom APIs. The bootloader implements multiple API functions available to both Secure (S) and Non-Secure (NS) execution states; we show security-relevant bootrom API functions in Table 1. To call an API function available to both S- and NS-states, software in NS needs to enter S by the defined *Secure Gateway* trampoline. If required, the Secure world then carries out additional sanitization for the passed parameters.

Bootloaders. The bootrom provides two different bootloaders, which are selected in the BOOTSEL stage. Both bootloaders run in NS and use bootrom APIs to execute security-critical operations, such as rebooting or accessing the OTP.

The first bootloader option is a minimalistic UART bootloader, which implements a simple serial protocol to read from and write to RAM. Additionally, it provides an *exec* command, which will trigger a reboot to a RAM image, using the corresponding bootrom API call.

The second bootloader option is a full-fledged USB bootloader, implementing a USB mass storage device, as well as a picoboot interface. The former implements a simple device that, upon receiving an image file, writes it to flash memory and triggers a reboot. The picoboot interface, on the other hand, is designed for interaction with the *picotool* software running on a host machine. It allows interactively loading images to flash and RAM, accessing the OTP, and rebooting the RP2350 into different boot modes.

3 The RP2350 Hacking Challenge

The RP2350 Hacking Challenge ran for a period of 4.5 months between August and December 2024. At its core is an RP2350 with secure boot enabled and confidential secrets stored in its OTP.

In the following, we describe the assumed attacker model for the challenge, and describe the steps required to set up an RP2350 chip for the challenge.

3.1 Attacker Model

The RP2350 Hacking Challenge assumes a strong attacker that is capable of physically tampering with the chip. Following typical fault injection scenarios, this includes applying abnormal voltage levels, providing an attacker-supplied clock, tampering with peripherals connected to external buses, or applying high-energy pulses to specific regions of the chip.

However, the attacker is *not* able to control or alter the firmware running on the target. Under normal operation, all executed code either resides in the unmodifiable bootrom or is part of a signed firmware verified by the MCU's secure boot implementation. Any potential flaw in the bootrom and protected firmware may be exploited by the attacker.

Finally, the challenge assumes that the RP2350 was correctly locked down and properly configured with secure boot, following the steps described in the next subsection. In other words, supply chain attacks and attacks that rely on weak configurations or compromised cryptographic keys are explicitly out of scope.

3.2 Setup

The RP2350 configuration used for the challenge is described in a public GitHub repository [30]. To prepare a pristine chip for the challenge, one needs to enable secure boot and *"lock"* the chip (i.e., disable debug access, enable the glitch detectors, disable unused boot keys, and prevent later changes to the critical OTP registers). In detail, this involves the following steps:

1. Generate a public/private key pair on the secp256k1 elliptic curve. These keys will be used for signing and verifying benign firmware.
2. Write a 128-bit secret value to OTP row 0xc08. Extracting this secret is the goal of the attacker.
3. Write the sha256 value of the public key to the `bootkey0` locations of the OTP. This key must then be marked as valid in the `boot_flags` OTP register.
4. Enabling secure boot by setting the `crit1.secure_boot_enable` bit in OTP. This automatically disables the RISC-V cores and enforces signature verification before booting an image.
5. Lock down the chip by permanently disabling debug access, marking all other bootkeys as invalid, and enabling the glitch detectors at their highest sensitivity level. All of these measures are implemented by setting the corresponding flags in OTP and utilizing OTP page locks to prevent later changes.

After this initial setup and lockdown of hardware, a custom firmware needs to be compiled, signed, and flashed to the target chip. This firmware is expected to be booted once before the target is subjected to attacks. It first checks the page lock state for OTP page 48 that contains the 128-bit secret stored in row 0xc08. If the page is not locked down yet, the firmware will permamently disable Non-secure access to the page, effectively permitting only software running in Secure state to read the secret.

Afterwards, access to the page is further restricted to make the secret completely inaccessible until the next reboot using the OTP software locks. Finally, the firmware spins in an infinite idle loop, minimizing the potential attack surface as much as possible.

4 Attacks

Over the course of the challenge, we found five different attacks. In this Section, we summarize the key ideas and attack vector for each. All of these attacks have been confirmed by Raspberry Pi Ltd to be successful *breaks*, effectively fulfilling the objective of the challenge.

4.1 Attacking the OTP PSM

Overview. The outputs of the OTP PSM must be unconditionally trusted by many other aspects of the system, making it a valuable target. This FI attack causes it to read incorrect critical bits, placing the device in an insecure RISC-V configuration with full-privilege debug access despite the true secure boot settings.

Context & Background. Correct readout of bits within the OTP antifuse array relies on a correctly-functioning IPS. However, an attacker can control the input rail to the IPS on the `USB_OTP_VDD` pin to create power faults.

The RP2350 incorporates an obstacle to `USB_OTP_VDD` manipulation in the form of *guard reads*, which are reads of known fixed data temporally interspersed with the meaningful reads performed by the state machine. Guard reads intend to detect IPS power faults by identifying mismatches in this known *guard word* data and restarting the OTP PSM if any are found. However, an attacker may hope to collect precise knowledge of the guard reads and avoid or defeat them.

`USB_OTP_VDD` Instrumentation. To understand the detailed behavior of the OTP PSM, we measure the current into

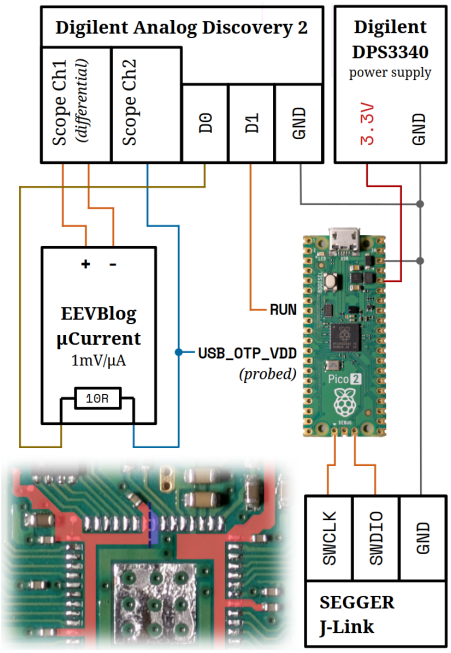

Figure 2: OTP PSM experiment setup. The inset shows where USB_OTP_VDD (blue, pin 53) is isolated from adjacent 3.3V power (red).

USB_OTP_VDD over time as an indication of OTP read activity. Furthermore, for FI, it is necessary to turn USB_OTP_VDD on and off with precise timing relative to that current waveform. A Raspberry Pi Pico 2 board is modified to isolate the USB_OTP_VDD pin from the main 3.3V I/O supply by cutting a trace between pins 53 and 54, and a test setup is constructed as shown in Figure 2 to allow this measurement and control.

IPS Fault Sweep. We sweep a 50µs-long USB_OTP_VDD power outage over a range of 0 to 600 µs after a characteristic current spike found near the start of the OTP PSM. 100 ns steps are used with 10 attempts at each offset. This process is initially performed on a *non-secured* device so that a debug probe can be connected to check for anomalies. We plot results in Figure 6 in the Appendix for the interested reader.

We observe a complete failure to boot the Cortex-M33 processors when the fault is positioned near 245-275 µs after the start of the OTP PSM (marked in Figure 6 with ⚡). Logging the available debug access ports on each fault attempt reveals that the device unexpectedly selects RISC-V mode in this scenario.

Guard Word Identification. Connecting a debugger to the non-secured chip, we check the contents of the OTP CRITICAL register in this failure state and notice that it has the value 0x00030033. Through additional experimentation, we find that the 24-bit guard word happens to be 0x333333.

Seeing that guard data has corrupted critical bits, we realize that all reads, including guards, have a persistence behavior across IPS power failures: the last-successfully-read data is interpreted as the result of subsequent reads, as long as the power remains off. Therefore, an attacker can carry forward latched guard data to overwrite subsequent meaningful reads.

Critical bits happen to be packed into CRIT0 and CRIT1 OTP rows in positions which result in both ARM_DISABLE and RISCV_DISABLE being set when overwritten with 0x333333. The RP2350 processor-selection logic is designed to choose RISC-V under this conflict. Further, the position of the DEBUG_DISABLE bit means that it happens to be cleared. On a locked-down chip, this combination should defeat all security features of the device.

Important OTP PSM reads, such as CRIT0 and CRIT1 reads, also involve a voting process across 8 redundant OTP rows which are read sequentially (with interspersed guard reads, of course). However, since all guard reads use the same data, we can extend the fault for as long as necessary without triggering a guard mismatch. We hypothesize that performing this OTP PSM power outage on a *secured* chip will result in the same unfortunate corruption of critical bits.

Verification. As hypothesized, the override of OTP CRIT0 and CRIT1 row data with 0x333333 during OTP PSM readout causes even a locked-down chip to exit reset with the security-unaware RISC-V cores selected and the DEBUG_DISABLE bit cleared. As a result, we can perform arbitrary bus reads and writes with full privilege over the RISC-V debug interface, allowing us to dump the OTP secret and run unverified firmware with no restrictions.

Following our initial public dissemination of this attack [14], it was independently replicated using different equipment [20], demonstrating its simplicity and amenability to many low-cost tools. Success is not difficult even without automation: adjusting the offset of the 50µs-wide fault in steps of 5 µs and *manually* triggering 10 attempts at each start position is a feasible attack method.

4.2 Forcing a Vector Boot

Overview. This attack injects faults during the reboot logic to boot an unsigned firmware image, effectively bypassing secure boot. For this attack, we verified the individual components in isolation before confirming our findings with Raspberry Pi Ltd.

Context & Background. The RP2350 supports reboot via the reboot() bootrom API call, which is exposed to both NS and S execution states. This API function takes a boot type as an argument to distinguish, for instance, between normal reboots and reboots to RAM images.

One special boot type is REBOOT_TYPE_SP_PC, which is only available to the secure world and instructs the bootrom

```
s_from_ns_varm_api_reboot:
  // Shadows r0 and sets up stack
  push {r0, r1, r4, lr}
  // ...
  // Verify bit 3 is not set (NS-reboot types)
  lsls r4, r0, #29
  bcs fail_reboot
  // Clear bit 3 in r0, to protect against FI
  lsrs r4, r4, #29
  lsrs r0, r0, #4
  lsls r0, r0, #4
  orrs r0, r4
  movs r4, #8
  bics r0, r4
  // If bit 3 was set, r0 and *sp would mismatch now.
  // This would trigger a rcp violation in this function.
  bl s_varm_hx_reboot
  b reboot_return
```

Listing 1: Annotated excerpt of `varm_misc.S`. The code ensures that `s_varm_hx_reboot` is not called with secure boot types from non-secure mode.

upon reboot to execute the code at the specified `pc`. This boot type is implemented by writing magic values alongside the desired `pc` and `sp` into watchdog scratch registers. The values in these registers persist over reboots and are checked by the bootrom to determine the boot type.

`REBOOT_TYPE_SP_PC` forms a security hazard, as the check for this boot type happens early-on in the boot process *before* signature verification of the target firmware. However, the only way an attacker can call the `reboot()` API function in a secure boot enabled system is by using the built-in USB bootloader implementing picoboot. As this bootloader executes in NS, the boot type is not available under normal execution but it can be reached via fault injection.

Vulnerable Instruction Sequence. The RP2350 bootloader uses a common code path for all reboot types. However, when `reboot()` is called from NS, the secure gateway redirects execution first to the hand-written `s_from_ns_varm_api_reboot` assembly stub, shown in Listing 1. This stub ensures that bit 3 of the boot flags parameter is not set, as this would indicate a secure-only boot type (e.g., `REBOOT_TYPE_SP_PC`). Additionally, to mitigate FI attacks skipping this check, the boot flags argument is shadowed onto the stack and bit 3 is explicitly cleared in `r0` before calling `s_varm_hx_reboot`.

We show a disassembled and annotated version of this function in Listing 2. This function implements the `reboot()` bootloader API call. Early on, it uses the RCP to verify that the flags value shadowed on the stack is equal to the value stored in the argument register. Then, after additional setup, the function validates the requested boot type and jumps to the corresponding code path. Unfortunately, the logic for checking whether the requested reboot type is `REBOOT_TYPE_SP_PC` is done via two instructions, marked with ⚡. Skipping either of these instructions will force a reboot in this mode, making them a prime target for a FI attack.

```
s_varm_hx_reboot:
00000578  push  {r0,r1,r2,r4,r5,r6,r7,lr}
0000057a  mrc2  p7,0x0,r4,cr4,cr4,0x1
0000057e  str   r4,[sp,#__stack_canary_value]
00000580  ldr   r4,[sp,#flags2]
// RCP equality check of r0 (flags) and r4 (flags2)
00000582  mcrr  p7,0x7,r0,r4,cr0
00000586  movs  r5,#0x0
00000588  movs  r6,#0x2
// disable watchdog temporarily
0000058a  ldr   r4,[DAT_00000624] // WATCHDOG_BASE
0000058c  rsbs  r6,r6
0000058e  str   r5,[r4,#0x0]=>DAT_400d8000
00000590  ldr   r5,[DAT_00000628] = 40018000h
00000592  str   r6,[r5,#0x8]=>DAT_40018008
00000594  movs  r5,#0xf
00000596  adds  r6,#0x32
00000598  ands  r5,r0 // extract boot_type
0000059a  ands  r6,r0
// boot_type == BOOT_TYPE_PC_SP (0xd) ?
0000059c  cmp   r5,#0xd                    ⚡
0000059e  bne   reboot_type_not_pc_sp      ⚡
000005a0  cbnz  r6,LAB_000005be
do_pc_sp_boot
000005a2  ldr   r5,[DAT_0000062c] // = B007C0D3h
000005a4  str   r5,[r4,#0x1c] // WD_SCRATCH_4: magic
000005a6  ldr   r5,[r4,#0x1c]
000005a8  rsbs  r5,r5
000005aa  eors  r5,r2
000005ac  str   r5,[r4,#0x20] // WD_SCRATCH5: pc ^ magic
000005ae  str   r3,[r4,#0x24] // WD_SCRATCH6: sp
000005b0  str   r2,[r4,#0x28] // WD_SCRATCH7: pc
000005b2  b     trigger_reboot
```

Listing 2: Disassembly for s_varm_hx_reboot. Skipping instructions marked with ⚡ will trigger a reboot with attacker controlled pc and sp value.

Attack Strategy. Assuming a capability to successfully skip one of the vulnerable instructions, designing an attack strategy is straightforward. First, an attacker preloads a malicious, unverified firmware to RAM using picoboot. The attacker then issues a benign reboot command but sets the boot parameters to meaningful `pc`/`sp` values if the reboot type was misinterpreted as `REBOOT_TYPE_PC_SP`. During execution of the reboot logic, the attacker glitches one of the two critical ⚡-instructions, forcing the bootloader to execute the wrong reboot logic, booting to the unverified, attacker-controlled firmware.

Verification. To verify this attack, we created four isolated tests. First, we verified the conceptual correctness of the attack strategy. For this, we used a non-locked down chip with debug access enabled and skipped the target instruction via an attached debugger. This test allowed to successfully boot an attacker-controlled firmware.

Second, we tested whether we can glitch the target instructions *without* the presence of additional FI mitigations (i.e., without RCP delays or enabled glitch protectors). We created a custom firmware re-implementing the critical logic of the reboot path and flashed it to a *RP2350 Security Playground Demo* board. To closely mimic the behavior of USB bootloader which runs from a 48 MHz clock, we also change the

system's clock source to the 48 MHz USB PLL. We used the Faultier fault injection tool [3] to inject voltage faults through crowbar glitching [27] and observed successful glitches.

Third, we enabled the glitch detector at the highest sensitivity level and attempted attacks on the same test code. Although the glitch detector did detect most FI attempts, we still observed successful glitches. We hypothesize that on a 48 MHz clock source, we can still inject glitch pulses short enough to go unnoticed by the glitch detection circuits but long enough to inject faults disrupting the target's execution.

Our fourth test adds RCP delays to our proof-of-concept victim code path. Between issuing a reboot command via the USB bootloader and the vulnerable instruction sequence, the bootloader executes four RCP instructions leading to a random delay between 0-508 clock cycles. We added four such instructions to our verification code sample and could still verify successful glitches, as random instruction delays do not conceptually mitigate FI attacks. Instead, they just reduce the likelihood of skipping one of the target instructions.

4.3 Attacking Signature Verification with Laser Fault Injection

Overview. In our third attack, we leverage laser FI to inject highly localized faults undetected by the glitch detectors. Inspired by recent efforts in the community [5, 40], we built our own laser FI platform to avoid the use of expensive and highly specialized equipment.

As an attack vector, we select the signature verification scheme in combination with hotswapping of flash memory. This allows us to create a mismatch between the cryptographically verified and the actual executed firmware.

Context & Background. The core logic of the RP2350's secure boot implementation resides inside `s_varm_crit_ram_trash_verify_block`. This function first loads the target firmware from flash into RAM, then creates a sha256 hash over it, before using this hash as input for the signature verification. After successful verification, the RAM-loaded firmware is then executed.

Since the RP2350 supports external QSPI flash, an attacker can change the contents read from the flash between accesses, for instance by hotswapping to a secondary flash or providing a custom QSPI implementation dynamically changing its contents. Assuming that successful fault injection can trick the bootloader to reread the firmware from flash during hash construction, an opportunity for attack arises. The attacker first supplies their malicious firmware to be loaded into RAM and then supplies the unmodified signed firmware to the hashing function. As the cryptographic verification passes on the constructed hash, the attacker-controlled RAM loaded firmware is executed.

[3] https://1bitsquared.com/products/faultier

```
1  s_varm_crit_ram_trash_verify_parsed_blocks:
2  //...
3  00001d02: 32 00         movs   r2,r6 // size
4  00001d04: 21 00         movs   r1,r4 // src  ⚡
5  00001d06: 0f a8         add    r0,sp,#0x3c // sha context
6  00001d08: 02 f0 49 fb   bl     sb_sha256_update_32
7  // ...
```

Listing 3: Potential FI target to force hash construction from flash. Skipping the ⚡-instruction forces hashing of the memory pointed to by `r1`.

```
1  s_varm_crit_mem_copy_by_words:
2  0000346c: 19 fe 3a c7   mrc2   p7,0x0,r12,cr9,cr10,0x1
3  00003470: 10 b5         push   {r4,lr}
4  00003472: 03 46         mov    r3,r0
5  00003474: 22 b1         cbz    r2,LAB_00003480
6  00003476: 1a 44         add    r2,r3
7      LAB_00003478:
8  00003478: 10 c9         ldmia  r1!,{r4}
9  0000347a: 10 c0         stmia  r0!,{r4}
10 0000347c: 90 42         cmp    r0,r2
11 0000347e: fb d3         bcc    LAB_00003478
12     LAB_00003480:
13 00003480: c0 1a         subs   r0,r0,r3
14 00003482: 09 fe 3a c7   mcr2   p7,0x0,r12,cr9,cr10,0x1
15 00003486: 10 bd         pop    {r4,pc}  ⚡
```

Listing 4: Another potential FI target to force hash construction from flash. Mutating the ⚡-instruction forces `r4` to hold an attacker-controlled value that is later used as source for hash construction.

Vulnerable Instruction Sequences. We identified three possible fault injection scenarios that result in the firmware being read from flash, rather than RAM, during hashing:

(S1) Listing 3 shows the disassembled code immediately before calling the hashing function `sb_sha256_update_32`. This code moves the source pointer for hashing into `r1` (L.3). At the point of execution, `r1` points to the end of the firmware that resides in flash. Thus, skipping the ⚡-instruction results in hash construction over flash contents.

(S2) During our initial tests, we observed that our injected faults would result in hash construction of data residing at flash address 0. While we do not know the exact cause for this, we believe one possibility could be a corruption of the `r1` register value before entering `sb_sha256_update_32`.

(S3) The register `r4` holds the pointer to the firmware in RAM in Listing 3. It is previously restored in the epilogue of `s_varm_crit_mem_copy_by_words` shown in Listing 4. By mutating this instruction through FI, `r4` may not be restored and still hold its previous value: the last word of the initial firmware read from flash. By carefully setting up the attacker-controlled firmware, this word could be a pointer to flash.

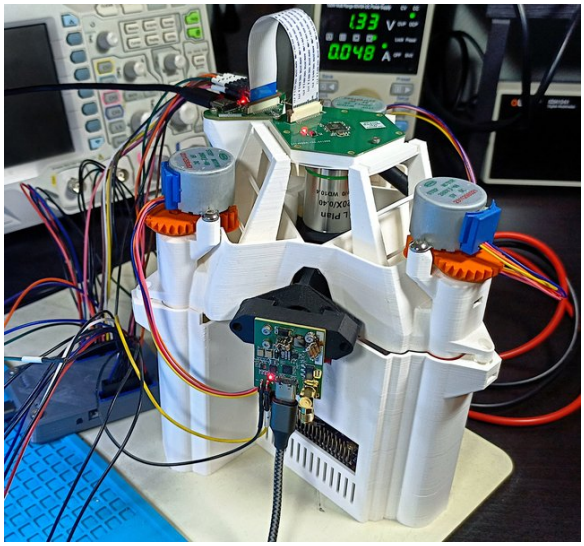

Figure 3: Our custom Laser Fault Injection Platform.

In our attack, we use two flashes and swap from the first flash to the second after the firmware has been loaded to RAM (i.e., we detected the last read in `s_varm_crit_mem_copy_by_words`).

The first flash holds a tampered version of the original firmware with two modifications. First, we manipulate the reset handler to hold shellcode diverting execution to a fixed located in flash. Second, we changed the last word of the firmware to point to a flash memory address to support (S3).

The second flash holds our main attack code (at the location targeted by the shellcode) and three copies of the original, unmodified firmware at different locations. Each copy corresponds to one of the scenarios, so that all three are supported by a single attack attempt.

Preparing the Target. To succeed with Laser FI, we need to expose the target while still ensuring operationality. We desoldered the RP2350 from a Raspberry Pi Pico 2 board and exposed its backside using a low-cost engraving tool that functions similarly to a Dremel. We then re-soldered the chip onto a custom carrier board which leaves the chip's backside exposed. We connect the carrier board to a custom I/O-board holding both QSPI flashes, SWD debug test points, a USB connector for communication, and a pin header for directly driving the attack. Both the I/O- and carrier board are mounted together in our *Laser Fault Injection Platform*.

FI Platform. We show our platform in Figure 3. The platform combines the carrier and I/O board with a positioning stage and includes an optical subsystem for imaging and laser FI. All mounting points for the platform are 3D printed and we use the ice40 FPGA embedded in the Glasgow Interface Explorer [4] as the main driver for the attack. The positioning stage is based on the OpenFlexure Delta Stage [24] and controlled through a Raspberry Pi single-board computer. We applied minor modifications to accommodate for our optical subsystem and mounting of our custom electronics.

For imaging, we follow the *Infra-Red, In-Situ* imaging approach introduced by Huang [2]. We use an 1070nm infrared LED for illumination and a *Raspberry Pi Camera v2 NoIR* for capturing images of the die. We further developed a custom laser driver board to send short, high-power light pulses. We used a 1064nm laser diode rated at 3W, which, when overdriven, is powerful enough to inject faults into the target RP2350 chip.

Verification. With the Laser FI platform in place, we first verified our fault injection capabilities using toy firmware with simple glitch-loops. After finding areas on the die yielding to changes in control-flow, we attempted to attack the secure boot implementation as described.

While additional fine-tuning was necessary of our attack setup was necessary (e.g., due to mechanical drifts in the positioning stage), we could observe successful glitches. These led to execution of the attacker-controlled firmware, ultimately leaking the OTP secret.

4.4 OTP Read Double-Instruction Fault

Overview. The RP2350 supports locking the OTP to different states which are forward rolling only. The bootrom leverages this in the boot process to secure sensitive OTP contents before jumping to one of the interactive bootloaders. However, we found that a double glitch in the critical path for locking down the OTP can prevent the actual locking. As a result, an attacker can leverage the interactive USB bootloader to directly read sensitive OTP data.

Vulnerable Instruction Sequence. The vulnerable code path lies in the custom configuration of the lock-state for PicoBoot. Essentially, OTP pages can be configured to be readable by firmware (in secure and non-secure mode), but not by the picoboot `otp` commands. The detailed lock-progressions (Figure 7) and states (Figure 8) can be found in the Appendix.

The configuration for this is stored in the OTP. The bootrom loads this configuration on each boot and uses the `SW_LOCK` registers to configure access permissions at runtime, before dropping into the interactive bootloader.

The implementation of loading the values from OTP and configuring the `SW_LOCK` registers resides in `s_varm_crit_nsboot`, as shown in Listing 5. For loading the lock values from OTP, this function calls `s_otp_advance_bl_to_s`, which reads OTP memory and returns a 3-way majority vote

[4] https://glasgow-embedded.org/

```
for(; page < NUM_OTP_PAGES; page++) {
  uint32_t sw_lock =
     s_otp_advance_bl_to_s_value(0xff, page);
  uint32_t sw_lock2 =
     s_otp_advance_bl_to_s_value(0xff, page2);
  hx_assert_equal2i(sw_lock, sw_lock2);
  otp_hw->sw_lock[page] = sw_lock;
  uint32_t sw_lock_verify = otp_hw->sw_lock[page];
  // ...
  page2 += (sw_lock_verify & sw_lock) == sw_lock;
}
```

Listing 5: Excerpt of `s_varm_crit_nsboot` for initializing OTP locks. The OTP lock values are read twice and compared using the RDP before writing the `otp_hw->sw_lock` register.

of the values. To harden against FI attacks, the call to `s_otp_advance_bl_to_s` is performed twice, and the two reads are compared by involving the RCP via `hx_assert_equal2i`.

When inspecting the disassembly for this code path, we found that if both calls to `s_otp_advance_bl_to_s_value` are skipped, all comparisons will succeed, as both arguments to the `hx_assert_equal2i` will be set to 0xFF. However, this would also mean that the value 0xFF will be written to the `SW_LOCK`—locking the pages completely. We show the according assembly in Listing 7 in the Appendix.

Instead, we found a different instruction sequence for mounting our attack. The end of `s_otp_advance_bl_to_s_value`, shown in Listing 6, performs two bitshifts. The first one shifts the result value in `r0` 28 bits to the left, and the next one shifts it 28 bits back to the right, effectively clearing the upper-most bits. If the ⚡-instruction was skipped, then a return value locking down `SW_LOCK` would be:

$$0b1111\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000$$

instead of:

$$0b0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1111$$

During testing, we found that such writes are simply ignored by the OTP locking hardware. Thus, OTP secrets which should be protected against reads from picoboot become suddenly accessible to the attacker, including the secret as set up for the challenge.

However, as the value `sw_lock` value is read twice, a double-glitch/double-skip of the ⚡-instruction is required. To make matters more complex, we also need to bypass two random delays generated by the sequence counting instructions at the exit entry of `s_otp_advance_bl_to_s_value`. This means that there are 254 different possible delay durations. However we tested the statistical distribution of these delays. Our tests indicated that an assumed delay value of 127 cycles may still be sufficient to carry out a successful attack.

Fault Injection Setup & Verification. For this attack, we leveraged EMFI. We carefully mapped the RP2350 to find

```
// r0: ignored, r1: otp_page
s_otp_advance_bl_to_s_value:
// RCP count set
00003380 89 ee 10 07  mcr  p7,0x4,r0,cr9,cr0,0x0  ☃
00003384 08 4a        ldr  r2, =0x4013FE04
00003386 c9 00        lsls r1,r1,#0x3
00003388 50 58        ldr  r0,[r2,r1] //
0000338a 01 0b        lsrs r1,r0,#0xc
0000338c 02 03        lsls r2,r0,#0xc
0000338e 11 43        orrs r1,r2
00003390 00 09        lsrs r0,r0,#0x4
00003392 01 40        ands r1,r0
00003394 0c 20        movs r0,#0xc
loop_entry:
00003396 08 43        orrs r0,r1
00003398 09 0a        lsrs r1,r1,#0x8
0000339a fc d1        bne  loop_entry
0000339c 00 07        lsls r0,r0,#0x1c
0000339e 00 0f        lsrs r0,r0,#0x1c  ⚡
// RCP count check
000033a0 a9 ee 30 07  mcr  p7,0x5,r0,cr9,cr0,0x1  ☃
000033a4 70 47        bx   lr
```

Listing 6: Assembly code for `s_otp_advance_bl_to_s_value`. When skippingthe ⚡-instruction, an insecure lock-word is returned. ☃-instructions induce a random delay.

locations which inject faults with a high probability while not triggering the glitch detector. By repeatedly attempting the attack, we could, indeed, observe successful double glitches and extract the OTP secret.

4.5 FIB/PVC based antifuse data extraction

Overview. Our last attack takes a different approach. Rather than attempting to defeat secure boot in some way and then reading fuse data via the processor memory bus, we simply read the secret directly from the fuse memory array. In the following, we provide a summary of our approach with more details published independently in [49].

Difficulty of Extraction. Antifuse memory is widely considered to be a challenging target for extraction because of the dimensional scales involved: modern antifuse bitcells such as those used on the RP2350 (Figure 4) use transistor gate dielectric as the programming element, with a thickness of only about 1.5 nm [10] on the TSMC 40nm node. The breakdown area is thus too small to directly image under an optical microscope or Scanning Electron Microscope (SEM). Scanning probe techniques (STM/AFM) and TEM cross sectioning have the necessary resolution, but are extremely slow which makes them impractical for extracting more than a handful of data bits.

Theory of Operation. Passive Voltage Constrast (PVC) is a common IC failure analysis technique [9] which can visualize electrical connectivity and leakage paths through differences in charging when a sample is scanned with a particle beam. Previous research has shown that some types of flash memory can be extracted by PVC [11] but there were no examples

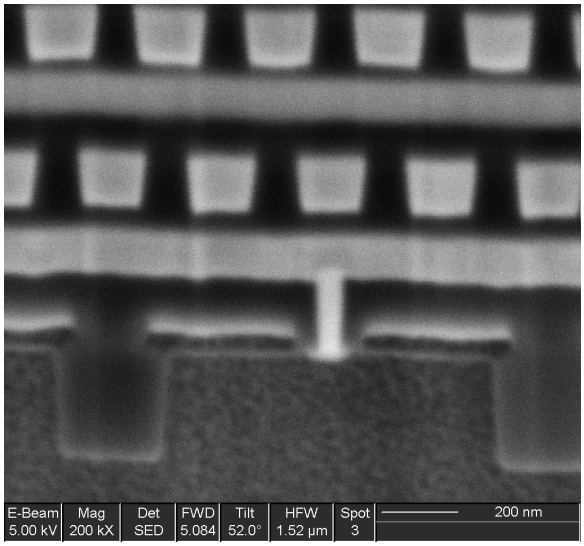

Figure 4: FIB cross section of two RP2350 fuse bitcells sharing a common bitline.

of successful data extraction from antifuses via PVC in the literature.

We discovered that the RP2350 antifuse bitcell could be read by PVC with a positively charged particle beam in a FIB. The particle beam injects positive charge carriers into the polysilicon wordline (partially turning on the bitcell transistor as V_{gs} increases), but leakage rapidly increases as transistors turn on which results in an equilibrium state holding the wordline close to the transistor V_t.

The particle beam also injects positive charge into the bitline contacts. If the fuse is intact, the contact will charge to a fairly high voltage due to a lack of a leakage path (the transistor is on, but there is no path from the channel to the wordline) and appear dark in the PVC image. If the fuse is blown, the injected charge will rapidly dissipate into the wordline and the contact will appear light under PVC.

Attack Process & Verification. We programmed an RP2350 sample with test vectors in the fuse memory. Then, we decapsulated and mechanically polished the sample to metal 1 and placed it in a dual-beam SEM/FIB.

We used the ion beam at high current (300 pA) to mill away metal 1 over each bank of the fuse array to expose the contacts over each pair of fuse bitcells. Next, we followed up with a cleanup pass using very low beam current (10 pA) and XeF_2 etch chemistry to remove redeposited debris. After milling was complete, the electron beam was scanned over the area to neutralize any positive charges which had built up.[5]

[5]Some FIB systems have a low energy *"flood gun"* electron emitter for charge neutralization, however the authors' system lacks this feature. With a dual beam platform, it is straightforward to perform charge neutralization on demand, by scanning the electron beam over positively charged areas.

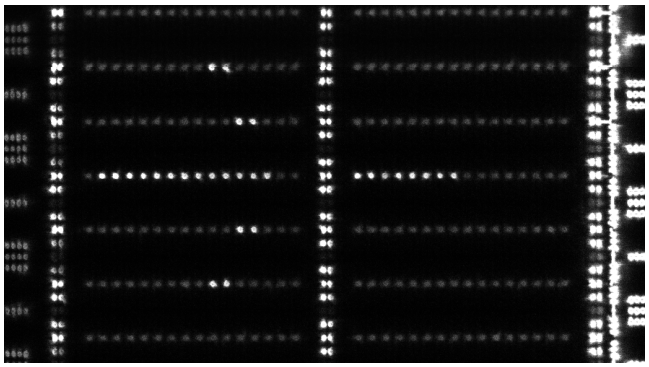

Figure 5: Location of the challenge key in fuse memory, with an arrow burned into adjacent fuses to highlight its location. Blown (logic 1) fuses appear light in this view, while unprogrammed (logic 0) appear dark.

We then used the ion beam at low current (10 pA) to image the fuse memory array, directly displaying pairs of fuse values[6] as light or dark contacts. Figure 5 shows a closeup of the challenge key (eight consecutive bits in the right column, all programmed to 1 in this example).

5 Mitigations

After reporting, Raspberry Pi Ltd confirmed all findings presented in this paper and added errata E16, E20, E21, and E24 to the RP2350 datasheet, covering all but one of the identified issues [31]. While a subset of the attacks can be fixed by workarounds, others remain unaddressed for the time being.

In this section, we discuss the workarounds and fixes suggested by Raspberry Pi Ltd and provide additional suggestions on how to mitigate the found issues in the future.

Attack 1: Faulting the OTP PSM. This issue fundamentally requires a silicon revision and so remains unfixed; Raspberry Pi Ltd notes that it will likely be addressed in a future version of the RP2350.

Guard reads should not contain values that downgrade security guarantees when misinterpreted as critical-bit data. An aggressive mitigation would store security settings in OTP as multi-bit magic values which correspond to *enabled* and *disabled* states. However, this comes at a space cost.

A simpler mitigation would rotate multiple distinct guard words, but might remain susceptible to short attacks against only meaningful reads. Randomization might reduce but not eliminate this concern. We do not view the currently-claimed ring oscillator randomization as an impediment.

[6]Each bitline contact is shared by two bitcells (with addresses 32 words apart in logical address space), as seen in Fig. 4. The basic PVC technique will show the contact as bright if at least one fuse is blown and dark if both are intact. We have ideas for extending the technique to separate the bitcell values, but have not yet been successful.

A straightforward response would be to add glitch detection (or some robust form of power-good sensing) on `USB_OTP_VDD`.

Attack 2: Unverified Vector Boot. This attack can be mitigated in two ways through additional OTP configuration, further hardening the system. First, booting from watchdog registers can be disabled by setting the `boot_flags0.disable_watchdog_scratch`. This disables vector reboot requests issued by the `reboot` bootrom API. Second, one can configure the OTP to disallow booting into the USB bootloader alltogether, effectively eliminating the attack surface.

Attack 3: Faulting the Signature Verification. Similar to the OTP PSM attack, no fix is available at the time of writing and Raspberry Pi Ltd notes that this attack is likely to be addressed in a future version of the RP2350.

To fix this issue, an update of the bootrom code is required. Potential mitigations could include in-lining hash construction in the initial copy-to-RAM loop (e.g. copy and hash one block at a time), or by adding FI-hardened checks to enforce that hashing is done only over RAM contents.

Attack 4: OTP Read Double Fault. Two workarounds are provided for this issue. First, the USB bootloader could be disabled via OTP configuration, similar to the mitigation of attack #2. Second, sensitive data could be protected with OTP access keys locking the data with page-level granularity. To read the sensitive data, application-level unlocking of the pages is required, which is not implemented by the Non-Secure USB bootloader.

Attack 5: Antifuse data extraction. At the time of writing, no erratum has been assigned for this issue. To mitigate this attack, we suggest employing a bit *„chaffing"* approach, where sensitive data are stored only in the lower or upper half of an OTP page. The other half-page is then filled with complementary bit values impeding PVC based extraction (all bit pairs will read as logic 1).

6 Lessons Learned

The attacks we found, alongside our interactions with Raspberry Pi Ltd, provided us with valuable insights, which we will share in this section. We hope that our observations will be useful for the community and further increase MCU security in the future.

Security through Transparency. We want to positively highlight the attitude of Raspberry Pi Ltd towards securing their products. Hosting the RP2350 Hacking Challenge incentivized third parties, including us, to independently verify the MCU's security guarantees. Especially crucial for this was the safe environment created for this task: The challenge explicitly allowed breaking the product's security without legal repercussions and provided a clear point of contact. We hope that similar challenges and security programs will become more popular for hardware components in the future.

Besides this, it is noteworthy that after the initial disclosure period, Raspberry Pi Ltd *transparently* communicated about the found weaknesses and attack vectors. This includes even attacks for which no mitigation is available. We believe that informing end users of the risks and limitations of proposed counter measures is important. This transparency allows iteration over designs and helps all vendors to create more secure products.

Pre-Boot Attacks. One particularly noteworthy attack presented in this paper is the attack against the OTP PSM. Typical fault injection attacker models assume that an attacker aims to change the behavior of executed software (e.g. by skipping instructions or introducing register-level faults) [48]. However, as our work shows, fault injection can also target digital or mixed-signal IP used for early initialization, and lead to severe consequences requiring deeper changes than simple mask ROM updates. This has also been demonstrated by other recent work that, for instance, unlocks hardware-locked flashes [26] or re-enables debugging ports [22, 35].

Thus, we argue that fault injection attacker models should also include capabilities to maliciously influence hardware behavior *before* any code is executed.

Effectiveness of Hardware Mitigations. Another interesting point is the effectiveness of the used hardware security features. While certainly raising attack complexity, three of our attacks successfully bypassed the RP2350's glitch detectors. The other two attacks were completely unaffected by the glitch detectors, as they compromise the chip's security without tampering with the glitch-protected core voltage domain.

The other main hardware mitigation, the redundancy coprocessor, was likewise unable to fully mitigate our attacks. While we did not need to bypass the generated stack canaries, the instruction delays, sequence count checking, and integer validation features reduced the available attack surface. Sequence count checking and integer validation required us to carefully select our targets for fault injection, hardening significant portions of the bootrom's code.

The random instruction delays impede the success rate of our attacks, but they cannot completely stop all attacks. Further, during our experiments, we noticed that the instruction delays would stall the coprocessor and pause the associated core. This leads to a measurable side-channel, allowing attackers to utilize observed instruction delays as a trigger condition. While we did not use this primitive in our attack, we note that the instruction delays may be a curious case where a mitigation negatively impacts a system's overall security.

The Dangers of Complicated Boot Paths. The code residing in the RP2350's bootrom is, arguably, complex. The combined Secure- and Non-Secure parts of the bootrom require 32kb and implement various convenience features. The prime example for this is the USB bootloader that allows interaction with the OTP or triggering different types of reboot—both features which we leveraged in our attacks.

We thus argue that for hardened and safety-critical systems, bootrom and bootloader implementation should be kept to a minimum, only providing the necessary features for a secure boot implementation. Convenience features could then be implemented by a signed second-stage bootloader flashed by benign parties when required.

Laser FI on a Budget. Instruments for Laser Fault Injection have traditionally been expensive, posing a significant financial challenge to attackers. Typical setups are estimated to exceed a price of $100,000 [7]. However, recent efforts in the community have demonstrated that Laser FI setups can be constructed at a much lower price point [5, 19, 40].

Our study independently confirms that building low-cost laser FI platforms is practical. We built our setup for less than $800, and are confident that further optimizations (e.g. by using different laser diodes) could further reduce the costs to below $500. To ease replication of our platform and re-use in other scenarios, we open source all our design files and provide a detailed write-up at [12].

7 Related Work

Fault Injection & Secret Extraction. A large body of work demonstrates attacks on MCUs using fault injection techniques [7, 17, 39, 42, 46, 48]. Closely related to our work, Delvaux et al. [15] bypass secure boot of the FI hardened ESP32 V3 MCU. Furthermore, recent work has demonstrated attacks against TrustZone-M, which can potentially undermine the security of secure bootloader implementations [34, 36]. Askeland et al. systematically analyze the security guarantees of glitch detectors [3], and different low-cost laser fault injection platforms have been described recently [5, 19, 48]. In contrast to these works, we provide the first study attacking the recently released RP2350 MCU implementing dedicated FI mitigations in hardware.

Another line of work focuses on firmware extraction by leveraging fault injection (e.g. [25, 38, 41]) or invasive hardware attacks (e.g. [21, 47]). Although this is often a required step to attack integrated products, our security analysis relies on the openly available RP2350 bootloader. Rather than extracting firmware, our attacks aim to inject attacker-controlled firmware on a secure-boot enabled MCU.

Software Security of Bootloaders. Recent studies investigate the security of bootloaders from a software perspective. For instance, Redini et al. [32] leverage symbolic execution to find vulnerabilities in bootloaders and Wang et al. [45] propose a fuzzing framework to find memory safety issues. While according vulnerabilities were in-scope for the RP2350 Hacking Challenge, our efforts focused on more invasive physical attacks. This decision is deliberate as we did not find obvious memory corruption vulnerabilities during initial static analysis. Additionally, the hardening provided by the used software isolation and RCP features would make exploitation difficult, even if vulnerabilities were present in the bootrom.

Attacking Secure Boot on Desktop Systems. While our work focuses on the secure boot implementation present in a popular microcontroller, various attacks show that secure boot implementations for desktop systems are also susceptible to compromise. For instance, LogoFail [43] presents a practical attack against UEFI firmware, Hudson & Bosch show a TOCTOU against Intel's Boot Guard [6], and Buhren et al. [8] attack the AMD security processor with voltage fault injection. These works demonstrate the relevance of attacks as reported in our work beyond the context of MCUs.

Hardware Security Challenges. In contrast to classic Capture-the-Flag (CTF) competitions, security challenges targeting hardware components under a physical attacker model are rather rare [28]. Most notably in the space are the Hack@DAC event series [16], the Capture the Signal CTF [1], hardwear.io's hardpwn competition [18], and the discontinued Rhme challenge series by Riscure [33].

We hope that the positive results of the RP2350 Hacking Challenge will create additional interest in organizing hardware security challenges. As shown by our work, such challenges lead to an overall improvement of the chip's security and provide valuable insights to chip manufacturers and the security community alike.

8 Conclusion

In this paper, we investigated the security guarantees of the RP2350, an FI-hardened MCU implementing secure boot. Despite the implemented and active countermeasures, we found five attacks subverting the security guarantees of the chip. All attacks were found within the scope of the RP2350 Hacking Challenge and are confirmed as legitimate *breaks* by the vendor.

We proposed several mitigations against our findings, some of which may be implemented in subsequent iterations of the RP2350 design. We further discussed insights resulting from our attack implementations. We hope that our observations will benefit the community and help to secure future generations of microcontrollers.

Acknowledgments

We would like to thank Raspberry Pi Ltd for hosting the RP2350 Hacking Challenge, their support throughout it, and their feedback on this paper.

Availability

We provide a meta-repository linking to the code and additional information for all attacks described in this paper at https://github.com/bhamsec/woot25-rp2350-challenge/releases/tag/v1.0.

References

[1] Jonathan Andersson, Marco Balduzzi, and Federico Maggi. Capture the Signal (CTS). cts.ninja, 2018.

[2] Andrew 'bunnie' Huang. Infra-red, in-situ (iris) inspection of silicon. *arXiv preprint arXiv:2303.07406*, 2023.

[3] Amund Askeland, Svetla Nikova, and Ventzislav Nikov. Who watches the watchers: Attacking glitch detection circuits. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024.

[4] Aspencore. The Current State of Embedded Development. embedded.com, 2023.

[5] Sam Beaumont and Larry Trowell. Laser Beams & Light Streams: Letting Hackers Go Pew Pew Building Affordable Light-Based Hardware Security Tooling. Blackhat USA, blackhat.com, 2024.

[6] Peter Bosch and Trammell Hudson. Now You See It... TOCTOU Attacks Against BootGuard. Hack In The Box Amsterdam, https://conference.hitb.org, 2019.

[7] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? *IEEE Access*, 10, 2022.

[8] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[9] AN Campbell, JM Soden, JL Rife, and RG Lee. Electrical biasing and voltage contrast imaging in a focused ion beam system. In *Presented at the 21st International Symposium for Testing and Failure Analysis*, pages 5–10, 1995. https://www.osti.gov/biblio/106667.

[10] Hsien-Chin Chiu, Min-Li Chou, Chun-Hu Cheng, Hsuan-Ling Kao, and Cheng-Lin Cho. Effect of body bias and temperature on low-frequency noise in 40-nm nmosfets. *Microelectronics Reliability*, 78:267–271, 2017.

[11] Franck Courbon, Sergei Skorobogatov, and Christopher Woods. Reverse engineering flash eeprom memories using scanning electron microscopy. In *International Conference on Smart Card Research and Advanced Applications*, pages 57–72. Springer, 2016.

[12] Kévin Courdesses. Laser Fault Injection on a Budget: RP2350 Edition. https://courk.cc, 2025.

[13] Ang Cui and Rick Housley. BADFET: Defeating modern secure boot using Second-Order pulsed electromagnetic fault injection. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[14] Aedan Cullen. Hacking the RP2350. In *38th Chaos Communication Congress (38C3)*, 2024.

[15] Jeroen Delvaux, Cristofaro Mune, Mario Romero, and Niek Timmers. Breaking Espressif's ESP32 V3: Program Counter Control with Computed Values using Fault Injection. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2024.

[16] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HardFails: Insights Into Software-Exploitable Hardware Bugs. In *USENIX Security Symposium*, 2019.

[17] Travis Goodspeed. *Microcontroller Exploits*. No Starch Press, 2024.

[18] Hardwear.io. HardPwn USA 2023: Google hails record haul of device vulnerabilities. hardwear.io, 2023.

[19] Martin S Kelly and Keith Mayes. High precision laser fault injection using low-cost components. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020.

[20] Matthias Kesenheimer. Glitching the Raspberry Pico with a Raspberry Pico. https://mkesenheimer.github.io, 2025.

[21] Thilo Krachenfels, Tuba Kiyan, Shahin Tajik, and Jean-Pierre Seifert. Automatic Extraction of Secrets from the Transistor Jungle using Laser-Assisted Side-Channel Attacks. In *USENIX Security Symposium*, 2021.

[22] Limited Results. nRF52 Debug Resurrection (APPROTECT Bypass). limitedresults.com, 2023.

[23] Raspberry Pi Ltd. RP2350 Hacking Challenge at DEF CON 2024. www.raspberrypi.com, 2024.

[24] Samuel McDermott, Filip Ayazi, Joel Collins, Joe Knapper, Julian Stirling, Richard Bowman, and Pietro Cicuta. Multi-modal microscopy imaging with the openflexure delta stage. *Optica Express*, 30(15), 2022.

[25] Johannes Obermaier, Marc Schink, and Kosma Moczek. One exploit to rule them all? on the security of drop-in replacement and counterfeit microcontrollers. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[26] Johannes Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller's firmware protection. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[27] Colin O'Flynn. Fault injection using crowbars on embedded systems. *Cryptology ePrint Archive*, 2016.

[28] Paolo Prinetto, Gianluca Roascio, and Antonio Varriale. Hardware-based capture-the-flag challenges. In *2020 IEEE East-West Design & Test Symposium (EWDTS)*. IEEE, 2020.

[29] Raspberry Pi. RP2350 A2 Bootrom. github.com, 2024.

[30] Raspberry Pi. RP2350 Hacking Challenge. github.com, 2024.

[31] Raspberry Pi Ltd. RP2350 Datasheet - A microcontroller by Raspberry Pi. Technical report, 2024.

[32] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. BootStomp: On the Security of Bootloaders in Mobile Devices. In *USENIX Security Symposium*, 2017.

[33] Riscure. Welcome to RHme3, the world's first automotive CTF. Archived via web.archive.org, 2018.

[34] Thomas Roth. TrustZone-M(eh): Breaking ARMv8-M's Security. In *36th Chaos Communication Congress (36C3)*, 2019.

[35] Thomas Roth, Fabian Freyer, Matthias Hollick, and Jiska Classen. Airtag of the clones: Shenanigans with liberated item finders. IEEE, 2022.

[36] Xhani Marvin Saß, Richard Mitev, and Ahmad-Reza Sadeghi. Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M. In *USENIX Security Symposium*, 2023.

[37] Michael Scire, Melissa Mears, Devon Maloney, Matthew Norman, Shaun Tux, and Phoebe Monroe. Attacking the nintendo 3ds boot roms. *arXiv preprint arXiv:1802.00359*, 2018.

[38] pcy Sluys, Lennert Wouters, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth security evaluation of the nintendo dsi gaming console. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 2023.

[39] Xi Tan, Zheyuan Ma, Sandro Pinto, Le Guan, Ning Zhang, Jun Xu, Zhiqiang Lin, Hongxin Hu, and Ziming Zhao. SoK:Where's the "up"?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2024.

[40] Janne Taponen. Laser Fault Injection for The Masses. Fractal, blog.fraktal.fi, 2024.

[41] Jan Van den Herrewegen, David Oswald, Flavio D Garcia, and Qais Temeiza. Fill your boots: Enhanced embedded bootloader exploits via fault injection and binary analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021.

[42] Sebastian Vasile, David Oswald, and Tom Chothia. Breaking all the things—a systematic survey of firmware extraction techniques for iot devices. In *International Conference on Smart Card Research and Advanced Applications*. Springer, 2018.

[43] Yegor Vasilenko, Alex Ermolov, Sam Thomas, Anton Ivanov, Fabio Pagani, and Alex Matrosov. LogoFAIL: Security implications of image parsing during system boot. Blackhat EU, blackhat.com, 2023.

[44] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adele Morisset, and Sebastien Ermeneux. Laser-induced fault injection on smartphone bypassing the secure boot. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2017.

[45] Jianqiang Wang, Meng Wang, Qinying Wang, Nils Langius, Li Shi, Ali Abbasi, and Thorsten Holz. A comprehensive memory safety analysis of bootloaders. 2025.

[46] Jasper van Woudenberg and Colin O'Flynn. *The Hardware Hacking Handbook. Breaking Embedded Security with Hardware Attacks*. No Starch Press, 2022.

[47] Yuanzhe Wu, Grant Skipper, and Ang Cui. Cryomechanical ram content extraction against modern embedded systems. IEEE, 2023.

[48] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault attacks on secure embedded software: Threats, design, and evaluation. *Journal of Hardware and Systems Security*, 2018.

[49] Andrew D Zonenberg, Antony Moor, Daniel Slone, Lain Agan, and Mario Cop. Extraction of Secrets from 40nm CMOS Gate Dielectric Breakdown Antifuses by FIB Passive Voltage Contrast. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2025.

A Appendix

Table 2: OTP_DATA `CRIT0` (row `0x038`; eight copies.)

Bits	Name	Description
23:2	Reserved	
1	RISCV_DISABLE	Prevent RISC-V processor selection
0	ARM_DISABLE	Prevent ARM processor selection (with higher priority than `RISCV_DISABLE` if both are set)

Table 3: OTP_DATA `CRIT1` (row `0x040`; eight copies.)

Bits	Name	Description
23:7	Reserved	
6:5	GLITCH_DETECTOR_SENS	Increase the sensitivity of the glitch detectors from their default
4	GLITCH_DETECTOR_ENABLE	Arm the glitch detectors to reset the system if an abnormal clock/power event is observed
3	BOOT_ARCH	Set boot architecture (0=ARM, 1=RISC-V). Lower priority than `ARM_DISABLE`/`RISCV_DISABLE`.
2	DEBUG_DISABLE	Disable all debug access
1	SECURE_DEBUG_DISABLE	Disable Secure debug access
0	SECURE_BOOT_ENABLE	Enable boot signature enforcement. Selects ARM, overriding `BOOT_ARCH` but not `ARM_DISABLE`.

```
loop_entry:
00003f34 01 99         ldr    r1,[sp,#0x4]
00003f36 ff 20         movs   r0,#0xff
00003f38 ff f7 22 fa   bl     s_otp_advance_bl_to_s_value ⚡
00003f3c 21 00         movs   r1,r4
00003f3e 05 00         movs   r5,r0
00003f40 ff 20         movs   r0,#0xff
00003f42 ff f7 1d fa   bl     s_otp_advance_bl_to_s_value ⚡
00003f46 40 ec 70 57   mcrr   p7,0x7,r5,r0,cr0
00003f4a 01 9b         ldr    r3,[sp,#0x4]
00003f4c 40 4a         ldr    r2,[page]
00003f4e 9b 00         lsls   r3,r3,#0x2
00003f50 9b 18         adds   r3,r3,r2
00003f52 1d 60         str    r5,[r3,#0x0]=>OTP_BASE
00003f54 1b 68         ldr    r3,[r3,#0x0]=>OTP_BASE
00003f56 2b 40         ands   r3,r5
00003f58 5b 1b         subs   r3,r3,r5
00003f5a 5a 42         rsbs   r2,r3
00003f5c 53 41         adcs   r3,r2
00003f5e e4 18         adds   r4,r4,r3
00003f60 01 9b         ldr    r3,[sp,#0x4]
00003f62 01 33         adds   r3,#0x1
00003f64 01 93         str    r3,[sp,#0x4]
00003f66 40 2b         cmp    r3,#0x40
00003f68 e4 d1         bne    loop_entry
```

Listing 7: Assembly for the code shown in Listing 5. Skipping ⚡-instructions will lead to writing 0xff to the OTP software locks.

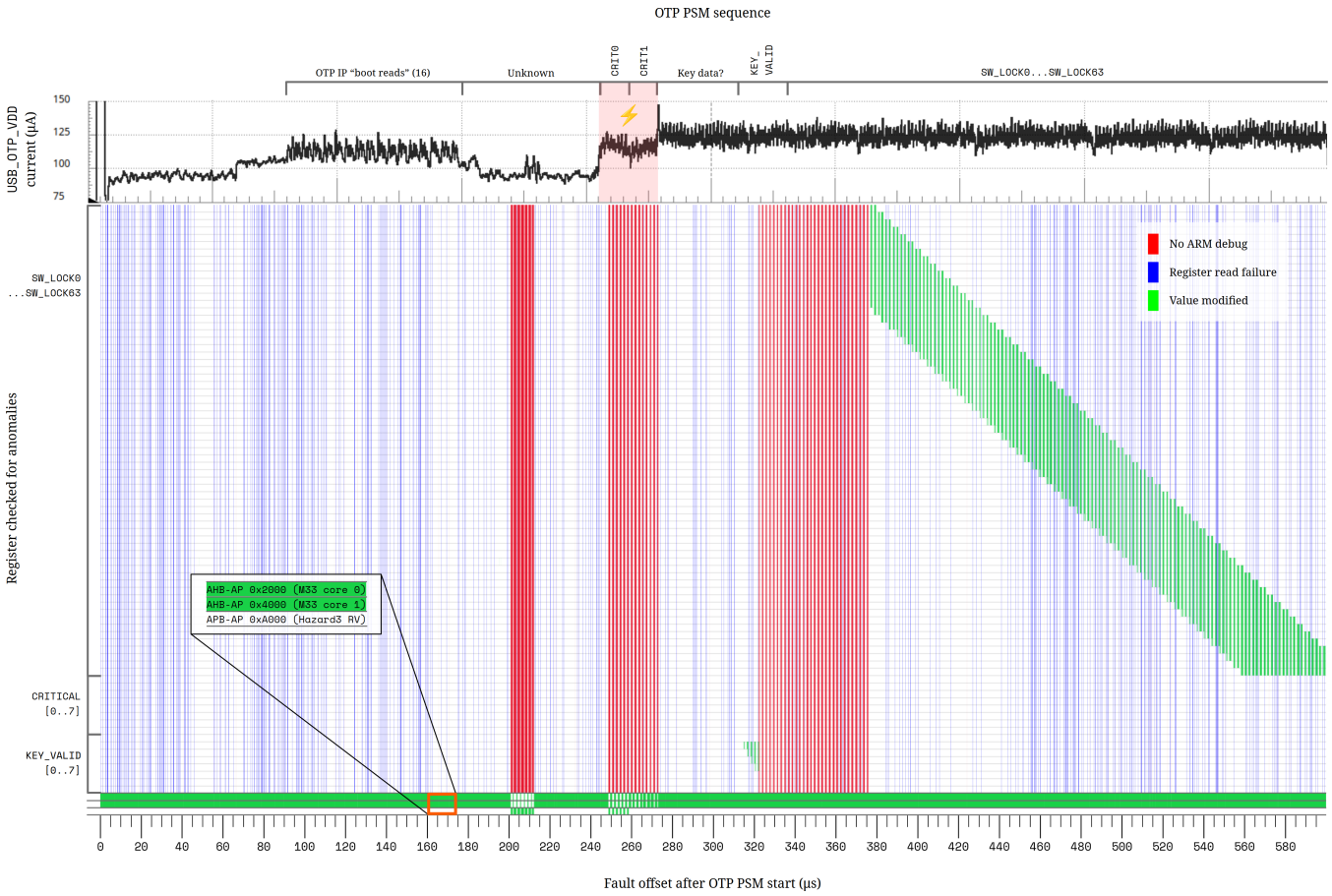

Figure 6: `USB_OTP_VDD` current trace and anomalous behavior under 50 µs faults on a non-secured device. Beginning a fault near 245-275 µs carries forward a guard read from just before the `CRIT0`/`CRIT1` reads. Beginning the fault near 200 µs captures guard data from an earlier initialization procedure (an undocumented implementation detail of the OTP, marked "Unknown" at the top.) Note that 50 µs is a sufficiently long period to span the distance between this initialization procedure and the location of the meaningful `CRIT0`/`CRIT1` reads. Failure to debug the ARM cores when faults are positioned during `KEY_VALID` reads is due to accidental enabling of debug keys.

Due to hardware constraints (Section 13.3.1), read and write restrictions are not orthogonal: it's impossible to disallow reads to an address without also disallowing writes. So, the progression of locking for a given page is:

0. Read/Write
1. Read-only
2. Inaccessible

Figure 7: The lock progressions supported by the RP2350 according to its datasheet [31, Chapter 13.5.1].

9:8	Secure lock state (thermometer code 0 → 2)
11:10	Non-secure lock state (thermometer code 0 → 2)
13:12	PicoBoot lock state (thermometer code 0 → 2) or software-defined use if PicoBoot OTP is disabled

Figure 8: The lock states supported by the RP2350 according to its datasheet [31, Chapter 13.5.3].