

Evaluating "faulTPM: Exposing AMD fTPMs' Deepest Secrets"

Nathan Mossaad
RWTH Aachen

Mats Büscher
RWTH Aachen

Abstract

In this paper we reproduce the attack presented in faulTPM [1] on an AMD Ryzen 3600 (Zen 2) platform by reconstructing their hardware setup. However, we revealed some optimistic assumptions in their cost breakdown, showing that the actual cost is higher than their estimate. To compensate, we also propose an alternative hardware approach aimed at further cost reduction.

By successfully gaining arbitrary code execution on the AMD Secure Processor, we were able to recover the BitLocker Volume Master Key (VMK). Using that VMK, we were able to decrypt the BitLocker volume and defeat the protection measures provided by Microsoft BitLocker's default TPM-only configuration, proving the ineffectiveness of a hardware-only-backed security measure.

1 Introduction

Starting with Windows 11, Microsoft enforces TPM-secured BitLocker full-disk encryption for all new devices [2]. This is especially relevant for fields like digital forensics and data recovery, where the BitLocker Recovery Key is not available. Hans Niklas Jacob et al. present a method in "faulTPM: Exposing AMD fTPMs' Deepest Secrets" [1] for AMD Zen 2 and 3 systems to access TPM-protected BitLocker decryption keys, which are protected by the default protector chosen by Windows. Zen/Zen+ may be vulnerable through simpler code-execution paths, but their different fTPM NV format requires additional reverse engineering. Newer Zen systems are also vulnerable in theory; however, no successful attack has been publicized. Unfortunately there is no other public research on the AMD Secure Processor for Zen2 or later.

In the following, we will present their findings and evaluate their artifacts, thereby reproducing their results. Along with faulTPM, we relied on their previous work "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization," which includes necessary background on the physical attack.

Our efforts reproducing the faulTPM attack presented in [1] focus on decrypting a BitLocker-encrypted drive on an AMD Zen 2 platform. In doing so, we thereby evaluated the practicality, required hardware setup, and reproducibility of the attack using their provided artifacts.

We will start by introducing key components of TPM-protected BitLocker encryption and voltage glitching. Following this, we will explore details of attacking AMD's fTPM implementation: specifically, creating a payload, the required hardware setup, when and how to trigger the attack, and how to extract key material from the TPM. Furthermore, we describe our reproduction and finally discuss our findings, with an overview of our costs compared to the authors' estimation.

2 Background

Throughout this section, we will introduce all components involved in attacking the fTPM on AMD's Zen 2 platform. This includes the TPM, AMD Secure Processor, full-disk encryption with BitLocker, and the process of voltage glitching.

2.1 Trusted Platform Module (TPM)

A TPM is a secure crypto-processor defined by the Trusted Computing Group's TPM 2.0 specification [3]. Its implementations come in two forms: discrete TPMs (dTPMs) on a separate chip and firmware TPMs (fTPMs) that run in a trusted execution environment on the main processor. On AMD systems, the fTPM runs inside the AMD Secure Processor (ASP). This implies that the CPU does not communicate with a separate TPM chip over an exposed bus. This removes the vulnerability of a physical interface, but it also makes the fTPM implementation depend on the security of the ASP. faulTPM showed that this trust anchor can be attacked through both firmware and hardware faults [1].

TPMs are often used for full-disk encryption. Before the bootloader receives the disk decryption key, the TPM checks that the platform is in an expected state and only unseals

secrets when the policy matches. Because the fTPM’s non-volatile (NV) storage is limited, sensitive objects such as RSA keys are usually sealed instead of stored directly. The sealed data is encrypted with a TPM-resident key and can be released only when the configured conditions hold:

- Platform Configuration Register (PCR) values: PCRs record the boot chain as a sequence of measurements. Each extension hashes the current PCR value together with the new measurement, so any change to UEFI firmware, option ROMs, drivers, or the bootloader changes the final PCR value.
- User-provided PINs or passwords: These combine a retry counter with slow brute-force attempts. Windows, for example, increases the wait time after 32 failed attempts [4].

2.2 AMD Secure Processor (ASP)

AMD’s Secure Processor, originally introduced as the Platform Security Processor (PSP), is the security coprocessor used by recent AMD systems [5]. It is a dedicated ARMv7 core with its own SRAM and firmware. By acting as the root of trust for the SoC and the platform initialization process [1], it is the most vulnerable component in AMD’s security implementation. The ASP firmware and the UEFI firmware are stored together on a separate SPI flash chip.

The ASP starts before the x86 cores. It first loads a ROM from the SoC, which in turn loads firmware from the SPI flash chip. AMD signs that firmware with the so-called AMD root key (ARK), which is also stored on the SPI flash. The ASP verifies the hash of the ARK and after that the signature of the firmware before execution. Figure 1 summarizes this early boot flow and the trust relationships between ROM, flash, and the signed firmware [1]. The SoC also exposes a dedicated crypto coprocessor (CCP) that is available to both the ASP and x86 cores for cryptographic operations. It additionally includes the so-called local storage buffer (LSB) that is used to store sensitive data.

2.3 Full Disk Encryption (BitLocker)

Microsoft BitLocker protects data at rest by encrypting the full disk, including the operating system and installed applications. The bootloader unlocks the disk by obtaining the key material from the TPM, so decryption only happens after the platform has passed the configured checks [1].

BitLocker uses two layers of key material. The Full Volume Encryption Key (FVEK) encrypts the disk contents. The Volume Master Key (VMK) protects the FVEK, and BitLocker stores the VMK under multiple protectors so that more than one recovery path exists. On Windows 11, the initial setup can enable BitLocker automatically when a TPM is present and the user signs in with a Microsoft account.

In the TPM-only configuration, the VMK is sealed to PCR values that describe the measured boot chain. If the firmware, bootloader, or other measured components change, the PCR values change as well, and the TPM does not release the VMK. In that case, BitLocker falls back to the recovery key.

The TPM+PIN configuration adds a user secret to the same chain. In this paper, we focus on the TPM-only configuration because it is the one used in the attack path.

2.4 Voltage Glitching

Voltage glitching is a fault-injection technique that disturbs the supply voltage of an integrated circuit (IC) so that it executes outside its normal specifications [6]. ICs are only specified to operate correctly within a bounded range of supply voltage, clock stability, temperature, and electromagnetic conditions [7]. When the supply rail drops or spikes at the wrong moment, the processor can mis-fetch an instruction, skip a check, or corrupt a register or memory value [8].

This can be useful to an attacker because it may change control flow at a precise point in execution. During cryptographic code, a single skipped operation can expose secret-dependent behavior or produce output that depends on the plaintext or key material [7]. During security checks, the same fault can bypass a branch that enforces a policy or reach code that is normally unreachable if the fault lands inside a privileged path [9, 10].

3 AMD Secure Processor (ASP) Attacks

To access the chip-unique secret or seed (further explained in Section 3.2) that derives the NV storage keys used to decrypt sealed TPM objects, we want to execute custom code on the AMD-SP. On Zen 1, a ROM bootloader vulnerability already gives arbitrary code execution on the ASP. Because the ROM is not updatable, presumably all Zen 1 and Zen+ devices are affected [1]. On newer ASPs, the paper uses fault injection instead. The voltage is dropped at the point where the processor validates a modified ARK, which sometimes results in the ASP accepting the modified firmware.

The voltage regulator (VR) on the motherboard is directly reachable. It adjusts the SoC voltage on demand, and communication happens over SVI2, an I²C-like protocol.

Following faultTPM [1], the payload is stored on the SPI flash. It combines an ARMv7-A assembly bootstrap with a C payload, and it replaces the off-chip bootloader used during startup.

The payload outputs data by writing to the memory-mapped SPI interface, which is recorded by the logic analyzer. PSPTool [11] is used to replace the firmware, re-sign firmware components using the ARK, and keep the AMD-SP file system cryptographically consistent.

Figure 2 shows the attack setup. A Teensy is connected to CS for timing the attack, to SVD and SVC via a bus driver

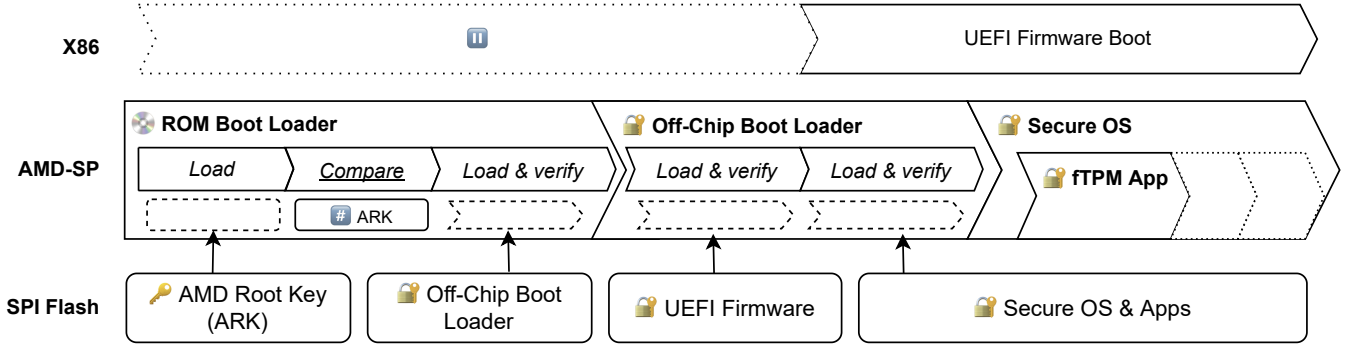


Figure 1: Early boot of a modern AMD CPU. A key denotes a public key, a pound sign a hash, and a lock a signed element. Reprinted from Ref. [1].

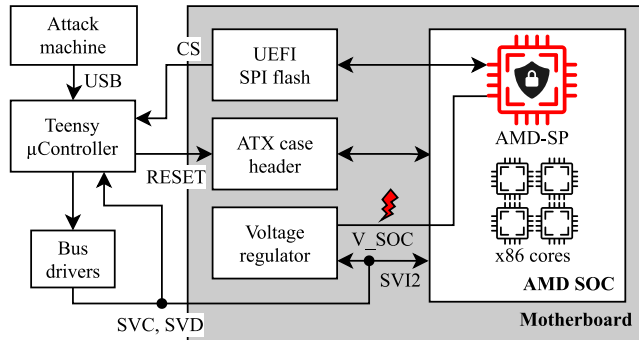


Figure 2: Schematic of the attack setup. Reprinted from Ref. [6].

to inject SVI2 packets, and to read the lines for detecting PC startup. The ATX reset header is connected to start a new attempt.

The timings need to be determined manually. After that, the attack is theoretically fully automated.

Figure 3 shows the SPI chip select (CS) line traces during the initial boot. The glitch has to land during ARK validation. In this setup, the validation window starts at the last CS low-to-high edge in the modified firmware and ends at the first CS high-to-low edge in the original image. Our own recording of the SPI CS and MOSI lines is depicted in Figure 7.

3.1 Parameter Determination

We have to determine precise parameters to correctly hit all events. These are noted in Figure 3, which shows the most important logic traces of an attack cycle. During the first few milliseconds, the CPU communicates with the VR over the SVC/SVD lines to set a stable voltage for the rest of the boot process. B1 marks the first chip select pulse, which continues until the CS count parameter is reached at B2.

The first SVI2 packet is injected at B3, shortly before the

ARK verification window, and it sets a low voltage. A second SVI2 packet follows at B4 and raises the voltage back to safe levels. B5 marks the lowest voltage reached during the attack. B6 and B7 show the difference between a successful and a failed attempt. If no continued SPI access occurs after a timeout, the PSP failed ARK verification. If SPI access continues after the timeout, the ROM bootloader accepted the ARK and continued loading the off-chip firmware. The ARK verification window itself is marked by C1 and C2.

The required voltage drop is achieved by setting a voltage identification (VID) value on the Teensy, which sends a packet to the VR over the SVI2 lines. The voltage of the first packet is set low enough that the AMD-SP cannot recover. A second, higher voltage level follows shortly after, but the window is so short that the voltage never reaches the initially requested level. The depth and shape of the voltage drop are therefore controlled by the duration, which minimizes the time spent at the lowest voltage level.

The attack parameters are the CS count, the delay, and the duration. The CS count determines how many chip selects to wait. The delay sets the timing of the voltage drop, and the duration sets its depth.

3.2 Unsealing VMKs

As described in 2.3, BitLocker stores a TPM-encoded sealed object in the unencrypted volume metadata. After decrypting the fTPM NV state, the attack extracts the relevant TPM wrapping keys and unseals that object to recover the VMK.

The NV state is stored on the BIOS flash. Sensitive data is encrypted, but the metadata and structure remain visible. Its entries use AES128-CTR for encryption and HMAC-SHA256 for integrity [1]. Those storage and integrity keys are derived from a 128-bit chip-unique secret in the CCP’s LSB 2.2. First, the chip-unique secret is decrypted with an AES-128 constant to obtain a seed. A key derivation function then derives two 256-bit values, one AES key for wrapping data and one HMAC key for wrapping data. Next, the fTPM application

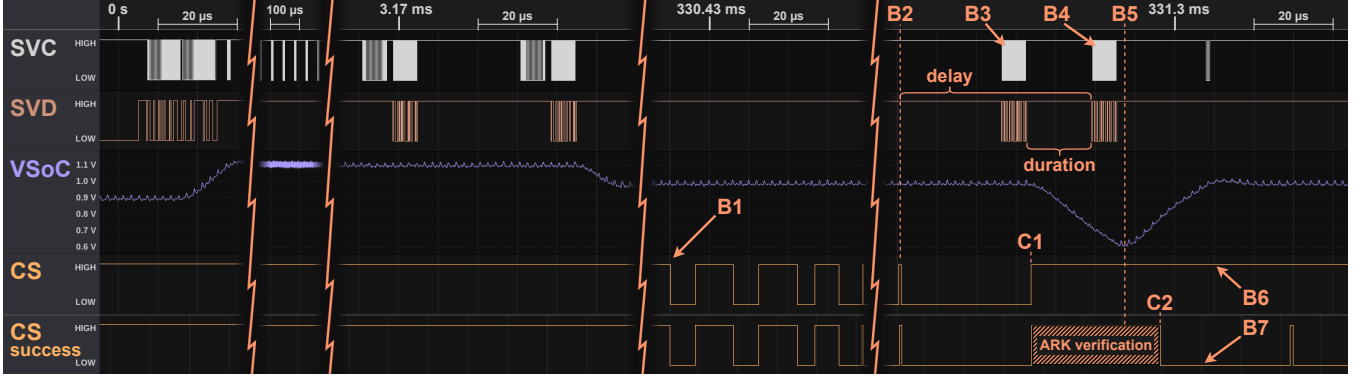


Figure 3: Logic traces of a complete Attack Cycle including CS traces of a successful and a failed attempt. Labels starting with “B” mark trigger events, the voltage drop injections, and the feedback mechanism. The CS edges marked with “C” are used to determine the initial window for the delay parameter. Simplified from Ref. [6].

signing key modulus and the fTPM application ID are mixed in, and the AES side is truncated to 128 bits while the HMAC side stays at 256 bits.

Once the payload is running, it writes the extracted seed to the SPI bus. With that seed, the fTPM state can be decrypted and TPM objects can be unsealed.

4 Reproduction

In faultTPM [1], a seven-step process is described for gaining access to BitLocker-encrypted data on an AMD Ryzen system. In the following, we will cover each of these steps in its own section. Furthermore, we present our findings and explain why we have deviated from their order in some places.

To reproduce their findings, we targeted a Ryzen 3600 on an Asrock H520M-HVS. This is a variation of the Asrock 520M-HDV with the only difference being the omission of a VGA port. We chose it because the authors verified it to be vulnerable to their fTPM attack and because of its good availability.

We created our own artifact repository [12] containing all patches, scripts, and supporting materials required to reproduce our results. The repository is organized into sub-repositories that fork the original authors’ codebases, with all modifications preserved as separate Git commits to clearly document our changes.

Overall, we found the original artifact to be largely complete, and it required only minor modifications to execute our attack. The primary challenge was the limited documentation, which made understanding the setup and workflow more time-consuming. However, because the source code for all components was available, we were able to resolve most ambiguities through code inspection.

4.1 Backup BIOS flash

We began by creating a backup of the SPI flash image using a recommended CH341-based programmer and standard programming software [13]. However, by directly connecting the programmer to our Winbond W25Q128JW, we slowly corrupted it, as it is a 1.8V part and our programmer operated with a supply voltage of 3.3V. Luckily, all errors were in non-critical sections, and we were able to re-flash a working BIOS with a 3.3V-to-1.8V level-shifter board. Fortunately, No permanent damage occurred.

4.2 Connect fault injection hardware and determine the attack parameters

Hans Niklas Jacob et al. present the concept of a targeted attacker, where they have “plenty of time; this attacker will open the case, will solder, and will use sophisticated hardware or software.” [1] We believe that this statement, while true, may understate the difficulty of actually running the attack successfully. In the following, we will present our findings on what worked and didn’t work when connecting each required component. Each component will be discussed separately.

4.2.1 Connecting the Logic Analyzer

We started by connecting a cheap 24MHz logic analyzer, using a clip as proposed in [1]. However, with the clip attached, we no longer booted due to signal integrity issues. To alleviate this, we opted to solder 0.2mm annealed copper wire to the flash chip’s pins, eliminating any connections via a breadboard and instead opting for jumpers directly into the logic analyzer. After that, we once again booted successfully.

When booting, our logic analyzer did not produce reliable captures. We only then noticed that our analyzer starts triggering at 2.0V, which was not satisfied by our 1.8V signals. Our

attempted solution was to boost the voltage to at least 2.0V with a level shifter (TXS0108E), which again resulted in a non-booting system. By soldering the TXS0108E directly to our SPI chip, we once again booted, though unreliably. When capturing the boot process with the 24MHz logic analyzer, we read an unreliable clock signal with only one to two samples registered per clock cycle. After rechecking all connected hardware, we came to the conclusion that this was because our clock speed was significantly higher than what the logic analyzer was capable of.

The solution was to acquire a new logic analyzer capturing at up to 400MHz, though 200MHz would be enough to reliably decode all MISO/MOSI traffic. Additionally, we had to shorten the soldered cables to about 5cm and plug the logic analyzer directly into the soldered ends (Figure 10). Even the included 15cm-long coax cables resulted in too much signal degradation, resulting in the system no longer booting. After successfully capturing the SPI clock signal with the new analyzer, we found it to be about 33.3MHz, proving that the proposed 24MHz logic analyzer was insufficient.

4.2.2 Attaching to SVI2

The voltage regulator on the H520M-HVS is a Richtek RT3667BQ, located near the CPU on the top left of the PCB. We opted to once again solder to the SVC and SVD lines using the same 0.2mm annealed wire because of the high cost of test probes. Unfortunately, we misread the datasheet and soldered our leads to the wrong pins; luckily, once again no permanent damage occurred. After noticing our mistake, we soldered the leads directly to some unpopulated pads, which expose SVC and SVD (see Appendix Figure 8 and Figure 9). Similarly to how we connected the first logic analyzer, we started off by using a breadboard to connect the SN74LVC (an open-drain output driver) with both our μ Controller and our VR data lines. Once again, this resulted in too much noise, and direct soldering between the SN74LVC and motherboard was required.

4.2.3 Attaching to Chip Select

Directly connecting our chip select line to our μ Controller resulted in missing reads, as the 1.8V signal of our SPI flash did not trigger the 3.3V SoC of our Teensy reliably. To solve this, we opted to step up the CS signals using another TXS0108E level shifter, which eliminated the reliability issues. To avoid requiring additional voltage regulation hardware, the ground and 1.8V input of the level shifter were directly connected to the corresponding pins on the SPI chip.

4.2.4 Reset

Connecting the reset trigger pin of the μ Controller to the motherboard was the only connection that did not cause any difficulty. We simply connected the μ Controller pin directly

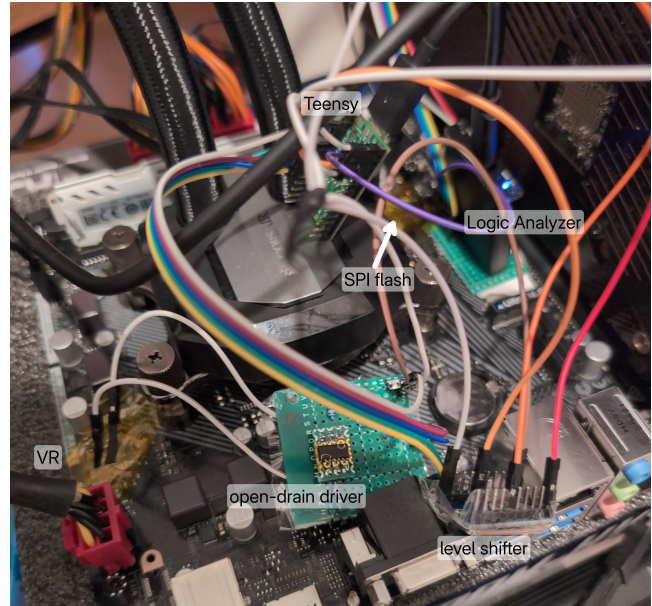


Figure 4: Full hardware setup, with annotations for relevant components. Reset header not visible. (Ground spider out of focus.)

to the corresponding header on the motherboard with some jumpers.

Note for those trying to reproduce our result that every time we tried using a breadboard, we had reliability issues due to the high frequency of our signals (except for the motherboard reset). However, we opted to use jumpers between the μ Controller and attached devices, allowing us to simply swap to the programmer when needed. Figure 4 shows all relevant connected components.

In the artifacts repository of Ref. [6], there is a detailed description of the parameter determination process. We slightly deviated from the proposed steps by first preparing the new payload. This simplifies parameter determination by not requiring an additional broken SPI flash image. We will elaborate on the process in Section 4.4.

4.3 Compiling and deploying the payload

In the following, we will discuss compiling and running the two most important software attack components: the Teensy firmware and the custom PSP firmware. This mostly worked, though smaller issues arose, which we will present along with their fixes.

4.3.1 Compiling the Teensy firmware

When trying to compile the Teensy firmware, multiple issues arose. First, due to Arduino IDE version changes, the file structure for downloaded components changed. To solve

this, we patched the Makefile to update the compiler path and Arduino core path. Second, the compile failed due to an unsupported use of `constexpr`; this was fixed by using `inline`. This is most likely due to us using a newer compiler version. After flashing the firmware to the Teensy, we could use a standard serial console to set all required parameters and arm an attack.

4.3.2 Compiling and patching the payload

We started out by verifying our SPI flash image from Section 4.1. Unlike with the Teensy firmware, the payload compiled without issues. To actually patch the payload into our flash image, we started out by using PSPTool to get the header structure of our specific UEFI. Unfortunately, our flash image had multiple different PSP firmware bootloaders for different supported CPU generations.

To find out the correct PSP firmware bootloader, we started out by executing a standard boot and recording the SPI flash chip’s MOSI with our logic analyzer. Then we had to convert the MOSI hex dump into the Saleae comma-separated values (CSV) format. This was necessary, as PSPTrace [14] does not support our binary dump directly. By running PSPTrace on our CSV and flash image, we obtained a list of accessed firmware components. This resulted in us finding out that our CPU used Directory 6 Entry 1, which makes sense as this entry is for Zen 2. Finally, we could use PSPTool to overwrite the PSP bootloader and re-sign it with a newly written ARK.

4.4 Parameter Determination

The following section is based on Ref. [6] and its artifacts, where a complete description is provided. In our reproduction, parameter determination focused on the two main glitch parameters, namely delay and duration. We first constrained the delay window to align the glitch with the ARK verification phase and then refined the duration window to increase the probability of successful glitching.

4.4.1 Delay Window

To determine the delay window, we started off by flashing a firmware with an invalid ARK onto the flash; this was our modified image. Our goal was to have the fault occur somewhere in the ARK verification window. This was achieved by setting the duration to zero and the delay to slightly after the last CS pulse. In our case, we chose the third-to-last CS pulse with a higher delay instead of the last pulse. This was necessary, as our motherboard and CPU combination required a very high duration window.

After getting the CS pulse count, we could flash the original firmware back onto the flash. Using the original firmware, we always successfully booted. This way, we could find the upper bound of the delay, resulting in an approximately 2000-tick-wide window.

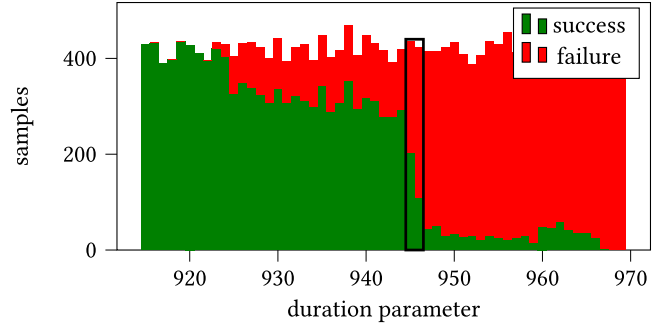


Figure 5: Attack samples for duration parameters in the transition window between always succeeding and always failing. The attacks target the original firmware image on the AMD Epyc 72F3 CPU. The final duration window is marked in black and contains the values deemed most likely to cause a fault by the refinement process. Reprinted from Ref. [6].

4.4.2 Duration Window

The duration window required a lot more trial and error. We simply tested different values until we found the range where we fully crashed the AMD-SP with a probability of about 50%. However, the authors’ recommended voltage level of 0xA0 never achieved a successful glitch with our target system. We needed an even lower target for the SoC voltage to drop enough and chose 0xC0, which sets the VR to almost fully cut power. According to the authors, this took them ≈ 30 min; it took us ~ 5 h to find semi-reliable values after familiarizing ourselves with the methods.

Throughout testing, we found some strange behavior where a combination of delay window and duration window sometimes worked, but other times did nothing. We decided to investigate this, and in our testing, we could not reproduce Figure 5. Instead of large drops in certain ranges, we found a linear correlation with higher durations during the same cold boot. However, separate cold boots shifted this correlation. Figure 6 shows two attack sample sets, each from a single cold boot. We currently have no theory on why it might make a difference whether the target is reset or booted after a full shutdown within a couple of seconds. We speculate that the nonlinear correlation in Figure 5 could be because of a lack of separate cold boots when gathering the data.

4.5 Start logic analyzer capture

Starting the logic analyzer should be trivial, but our logic analyzer is only supported via sigrok [15]. When using PulseView, a GUI from the sigrok project, with a trigger channel, there is a memory leak that eventually results in the OS killing the application. Therefore, we used sigrok-cli to capture the SPI data into a binary file to be analyzed by PulseView at a later time.

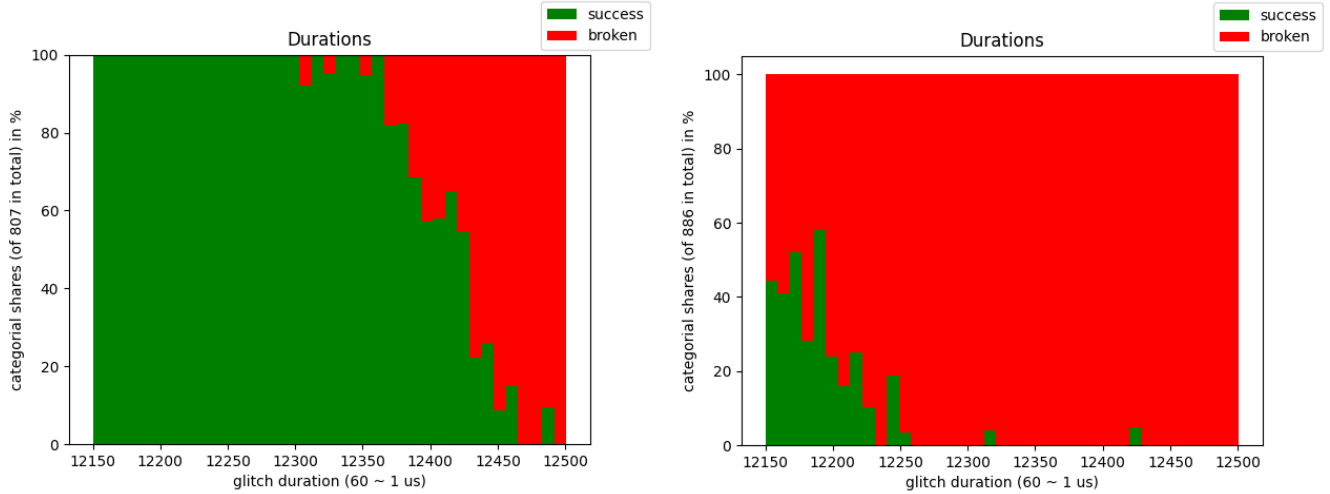


Figure 6: Attack samples similar to Figure 5 but represented as percentages. Samples are all from a single cold boot.

4.6 Start the attack cycle on the target machine

After repeated attacks, we at least once glitched into a state where a second ARK readout and verification was performed, which failed. This resulted in a false positive attack and should not be used for refinement of the delay, as it was much smaller than all other successful attempts.

4.6.1 Refining parameters

Hans Niklas Jacob et al. were able to refine the parameters to a window of about ± 50 for delay and ± 10 for duration. We could not find these tight tolerances; our final glitch values resulted in windows of ± 105 for delay and ± 125 for duration, most likely due to our cold boot variations. Across all of their successful attempts, they reached a rate of 17/15459 ($\approx 0.11\%$); we only reached a rate of 16/303029 ($\approx 0.00528\%$). With their final windows, they reached a rate of successful attempts of 6/3467 ($\approx 0.173\%$); we only reached 6/41628 ($\approx 0.0144\%$). Their average waiting time was 13.4 min (Zen 1) and 46.5 min (Zen 3). We reached a highly variable time of about 30 min to 60 min for our final parameters on Zen 2, after optimizing the attack script by reducing unnecessary waiting.

4.7 Parse and decrypt the NV storage and extract TPM protector

By using the provided instructions along with `amd_ftpm2_unseal.py` from the artifacts repository, with its sub-repositories (PSPTool and amd-nv-tool), we were able to create the BitLocker VMK. To achieve this, we utilized `uv` [16], as this simplified the setup due to the requirement of using Python 3.8. This was necessary, as

certain crypto-related functions were removed in Python 3.9. We created the necessary `uv` configuration, created a virtual environment, and installed the specified versions of both PSPTool and amd-nv-tool and all other dependencies. Unfortunately, the Python scripts were not compatible with our image due to some entries being duplicated in our BIOS. By adding a patch that simply ignores all but the first entry, we successfully derived the VMK.

4.8 Unseal VMK using TPM protector

Instead of booting Linux on our target, we decided to plug the target’s SSD into our host using a SATA-to-USB adapter. The final steps to gain access to the C: partition involved mounting the disk, decrypting it with the VMK using dislocker, and finally mounting the encrypted partition.

5 Cost of attack

One of our goals from the beginning was to find out if the claimed \$195 cost, excluding \$75 for test probes, was possible. Table 1 shows our costs. The biggest issue was that the cheap logic analyzer was not sufficiently fast, thereby requiring a much more costly logic analyzer. Furthermore, the cost of soldering equipment was not included, though we assumed that this would be available to the attacker, e.g., by visiting a local hackerspace [17] or makerspace. The actual cost will be higher than the estimate due to shipping, taxes, and spare components.

5.1 A new cheaper setup and implementation

We wanted to prove that a much cheaper attack using a Raspberry Pi Pico (with an RP2040 μ Controller) is possible. We

Table 1: Our cost (estimated) breakdown compared to the costs of FaultPM [1]. Additional consumables is an estimated catch-all for used wire, solder, flux, (Kapton) tape etc. consumed.

Description	FaultPM costs	Our costs	Costs using pico
Teensy 4.0 Development Board	\$20	\$28	-
Raspberry Pi Pico	-	-	\$4
SPI flash programmer	\$15	\$7*	-
Logic Analyzer	\$15	\$63	\$7**
Driver IC and additional resistors	\$5	\$2	\$2
Flash chip test clip	\$5	-	-
Test probes with spring-loaded pins	\$120	-	-
Controllable Power Relay	\$15	-	-
Level shifter	-	\$1	\$1
1.8V LDO for flashing	-	-	\$1
Jumper wires	-	\$2	\$2
Additional consumables	-	\$5	\$5
Total	\$195 (\$75 excl. probes)	\$108	\$22 (\$15 without logic analyzer)

*includes clip and 1.8V adapter **Cheap 24MHz logic analyzer optional, but included for convenience

developed ftpm-glitcher [18], a proof-of-concept firmware and CLI tool that provides much faster iteration compared to the authors’ Teensy-based firmware. It is possible to do over 30 attempts per second compared to the around three attempts that we managed to achieve with small modifications to the Teensy firmware and test scripts. Using this, we could reliably achieve glitching times of less than five minutes with about 6 500 attempts per successful glitch, distributed across our five attempts.

Our firmware utilizes the SPI controller of the RP2040 μ Controller and a level shifter to directly read and decode the SPI MOSI line. Using this, we can eliminate the need for a logic analyzer entirely. Additionally, SVI2 communication is implemented using one of the RP2040’s PIO blocks. Furthermore, the CH341-based programmer can be replaced with an appropriate serprog [19] flash image and a cheap 1.8 V power supply (e.g., an LDO). A full copy of our custom firmware is included in our artifacts [12]. A schematic of the setup is also included in Figure 11, as well as a photograph in Figure 12. By omitting the need for an additional logic analyzer, programmer and the need for an expensive Teensy, we reduced our costs to about \$15 or \$22 with an optional cheap analyzer, used for debugging purposes.

6 Conclusion

Using the hardware and fault-injection techniques from FaultPM [1], we glitched a Zen 2 CPU and compromised the AMD Secure Processor fTPM implementation. The attack depended on a 1.8 V SPI flash chip and a 33 MHz clock. At those data rates, signal integrity became the main constraint, which forced careful layout, direct soldering to the motherboard, and a more expensive logic analyzer.

Overall, we found the artifact and reproduction guides detailed and complete. The main gap was usage documentation for PSPTool [11], where we had to inspect the code directly to use the `-replace-file` feature.

Finally, we were able speed up and reduce the cost of the attack from \$195 to \$15 by replacing the Teensy 4.0 with a Raspberry Pi Pico and eliminating the need for an expensive logic analyzer.

References

- [1] Hans Niklas Jacob, Christian Werling, Robert Buhren, and Jean-Pierre Seifert. faultpm: Exposing amd ftpms’ deepest secrets. <https://arxiv.org/abs/2304.14717>, 2023.
- [2] Microsoft. Bitlocker drive encryption in windows 11 for oems. <https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-bitlocker>, 2026. Accessed: 2026-06-21.
- [3] Trusted Computing Group. Tpm 2.0 library specification. <https://trustedcomputinggroup.org/resource/tpm-library-specification/>, 2026. Accessed: 2026-06-21.
- [4] Microsoft. Bitlocker countermeasures. <https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/countermeasures>, 2026. Accessed: 2026-06-20.
- [5] Roger Lai. Amd security and server innovation. <https://uefi.org/sites/default/files/>

- [resources/UEFI_PlugFest_AMD_Security_and_Server_innovation_AMD_March_2013.pdf](#), 2013. Accessed: 2026-06-21.
- [6] 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. <https://arxiv.org/abs/2108.04575>, 2021.
- [7] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer’s apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.
- [8] A. Djellid-Ouar, G. Cathebras, and F. Bancel. Supply voltage glitches effects on cmos circuits. In *International Conference on Design and Test of Integrated Systems in Nanoscale Technology, 2006. DTIS 2006.*, pages 257–261, 2006.
- [9] Yifan Lu. Injecting software vulnerabilities with voltage glitching. *CoRR*, abs/1903.08102, 2019.
- [10] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling pc on arm using fault injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35, 2016.
- [11] Christian Werling et al. Psptool. <https://github.com/PSPReverse/PSPTool>, 2026. Accessed: 2026-06-21.
- [12] Nathan Mossaad, Mats Büscher. Artifacts. https://github.com/Nathan-Mossaad/ftpm_glitch_reproduction, 2026. Accessed: 2026-07-14.
- [13] Mikhail Medvedev et al. Imsprog. <https://github.com/bigbigmdm/imsprog>, 2026. Accessed: 2026-07-14.
- [14] Christian Werling et al. Pspttrace. <https://github.com/PSPReverse/PSPTTrace>, 2026. Accessed: 2026-06-21.
- [15] Sipeed et al. Add sipeed slogic analyzer driver support. <https://github.com/sigrokproject/libsigrok/pull/275>, 2026. Accessed: 2026-06-21.
- [16] uv contributors. uv docs. <https://docs.astral.sh/uv/>, 2026. Accessed: 2026-06-21.
- [17] List of hacker spaces. https://wiki.hackerspaces.org/List_of_Hacker_Spaces, 2026. Accessed: 2026-06-21.
- [18] Nathan Mossaad, Mats Büscher. ftpm-glitcher. <https://github.com/Nathan-Mossaad/ftpm-glitcher>, 2026. Accessed: 2026-07-19.
- [19] Serprog (overview). https://www.flashrom.org/supported_hw/supported_prog/serprog/overview.html, 2026. Accessed: 2026-06-21.

Appendix

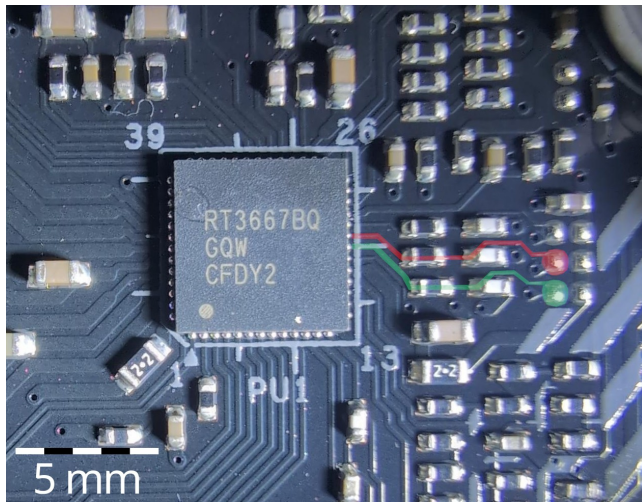


Figure 8: Voltage regulator (RT3667BQ) on the H520M-HVS with highlighted SVD (red) and SVC (green) lines and unpopulated pads.

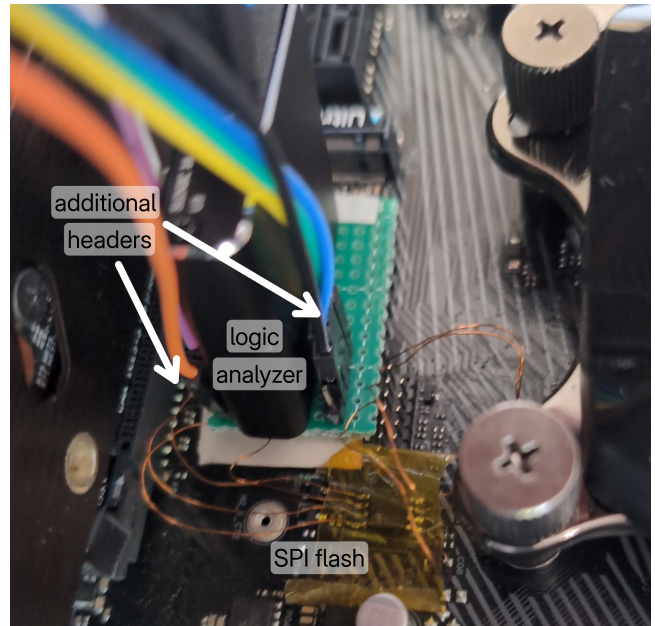


Figure 10: SPI flash (W25Q128JW) soldered with 0.2mm copper wire to a distribution circuit board. Logic analyzer directly connected to headers. Additional headers for programmer and glitch hardware.

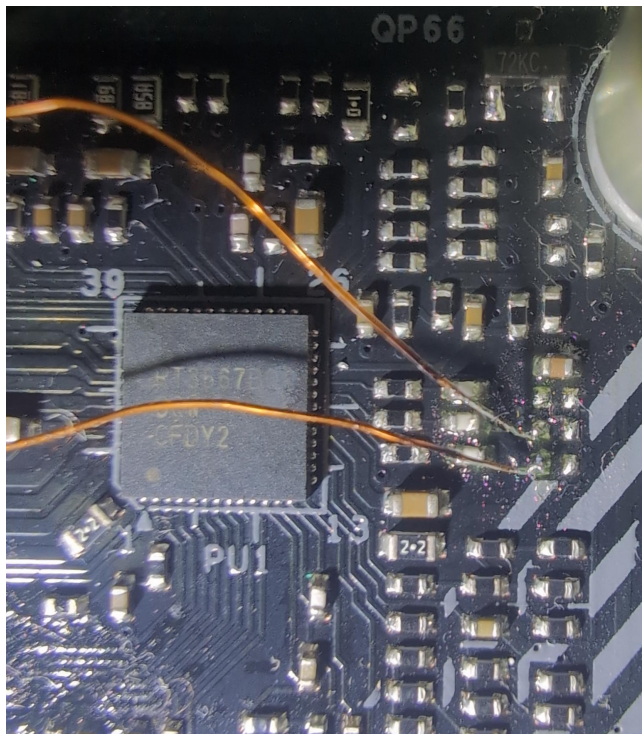


Figure 9: Voltage regulator (RT3667BQ) with leads attached to SVD and SVC.

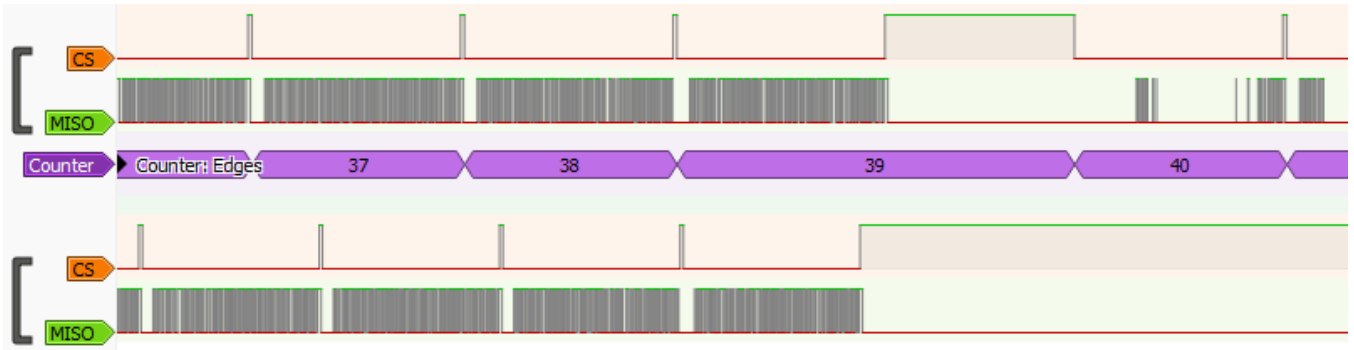


Figure 7: SPI bus traces during the initial boot. CS and MISO lines only. The upper part depicts SPI bus activity for the original flash image. The lower part shows the corresponding SPI signals for a flash image with a manipulated ARK (no communication after the 39th chip select). Image and description replicated from Ref. [6].

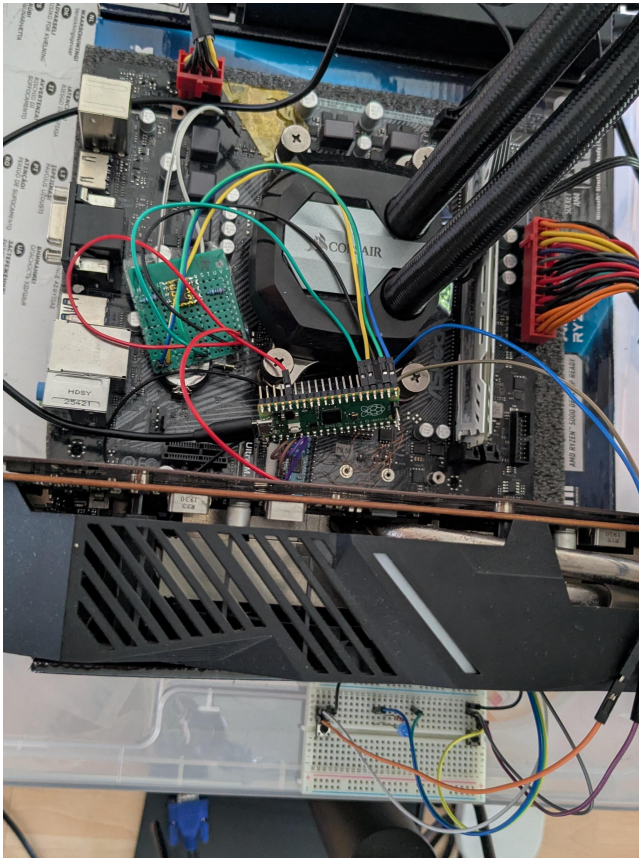


Figure 12: Full Pico hardware setup, as described by the schematic in Figure 11.

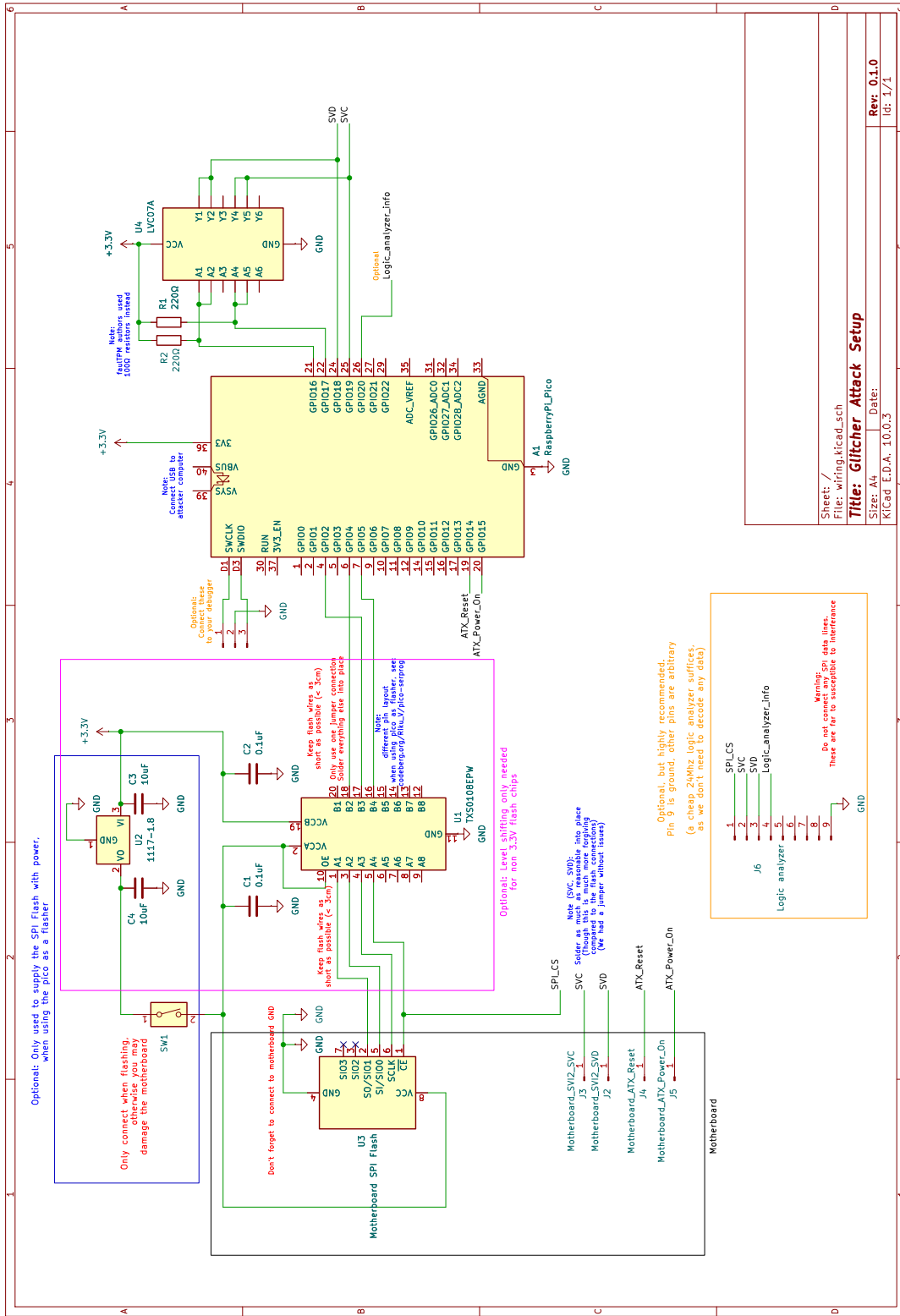


Figure 11: Schematic of the Pico glitch setup.