NAME OF INTERN	Yeoh Wei Yang
NAME OF DSO SUPERVISOR	Tan Sze Yan, Xing Yihuan
DATE OF INTERNSHIP	15 Jan 2024 - 31 May 2024

1. Overview of internship assignment

Abstract

Smart and connected devices are prevalent and integral to many parts of our daily lives. To ensure the integrity and authenticity of the software running on these devices, manufacturers utilize hardware-based security in microcontrollers to protect their intellectual property. However, research has shown that hardware security is not foolproof, and can be defeated with hardware-based attacks, such as voltage fault injection. The more notorious examples of hardware security breaches would be the Apple Airtag [2] and the Trezor One cryptocurrency wallet security breaches [1].
As electronic development boards become cheaper and more accessible over the years, this project aims to explore the cost asymmetry in fault injection through the use of such boards to coordinate and inject the voltage fault.

1. Introduction

Side channel attack is a security exploit that aims to gather information from or influence the program execution of a system by measuring or exploiting the indirect effects of the system or its hardware. Side channel attacks can be either passive or active. Passive attacks exploit information that is being leaked by the device, such as power consumption and electromagnetic emissions. Active attacks are more commonly referred to as fault injection attacks. These attacks influence the system with internal or external stimuli. There are 4 main types of fault injection attacks: voltage, clock, electromagnetic and optical. These attacks aim to extract sensitive information in devices, such as cryptographic keys or device memory, by measuring and/or manipulating the hardware. In this project, I will be looking into the use of low-cost hardware, such as the RP2040 microcontroller, in order to conduct voltage fault injection on microcontrollers, which have been increasing in usage due to the rise in popularity in IOT devices. While voltage fault injection has been researched on, most literature have utilized specialized hardware such as ChipWhisperer or the Field Gate Programmable

Array (FPGA) to conduct these attacks. As such, this project aims to implement this attack using commercial, off-the-shelf microcontrollers that are lower in cost and more readily available to the public. Additionally, I will also be utilizing other tools to assist in this project:, namely Ghidra for static analysis of the firmware, and oscilloscope for side channel analysis.

2. Preliminaries

Voltage fault injection.

Voltage fault injection, otherwise known as voltage glitching, is the act of momentarily dropping the supplied voltage to a device during specific operations to cause a misbehavior in the target. This momentary drop in voltage supplied to the target device usually has a duration in the nanoseconds range, and that it must occur at very specific timings during the target's code execution. The figure below illustrates the power trace of a typical voltage fault injection set up.

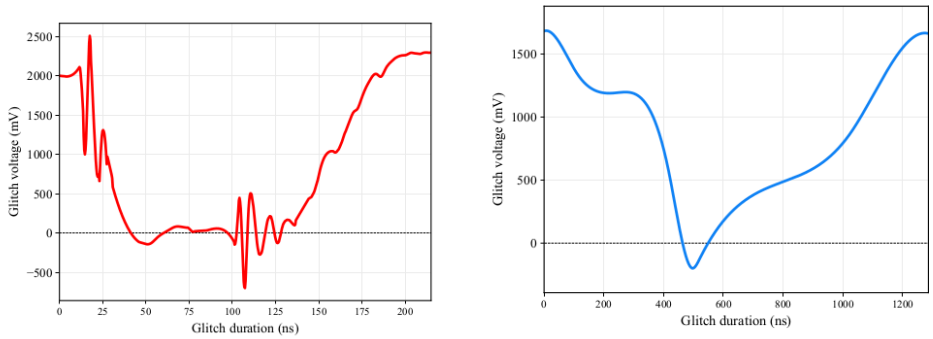

Figure 1. Power Trace of a Voltage Fault Injection Attack [3]

Being able to successfully glitch the target allows an attacker to compromise its security: eg, skipping of instructions, corrupting data, changing program flow, etc. As a result, there are various security implications, including extraction of memory, privilege escalation, extracting of cryptographic keys, etc.

Setup. A typical VFI setup would be in Figure 2 below.

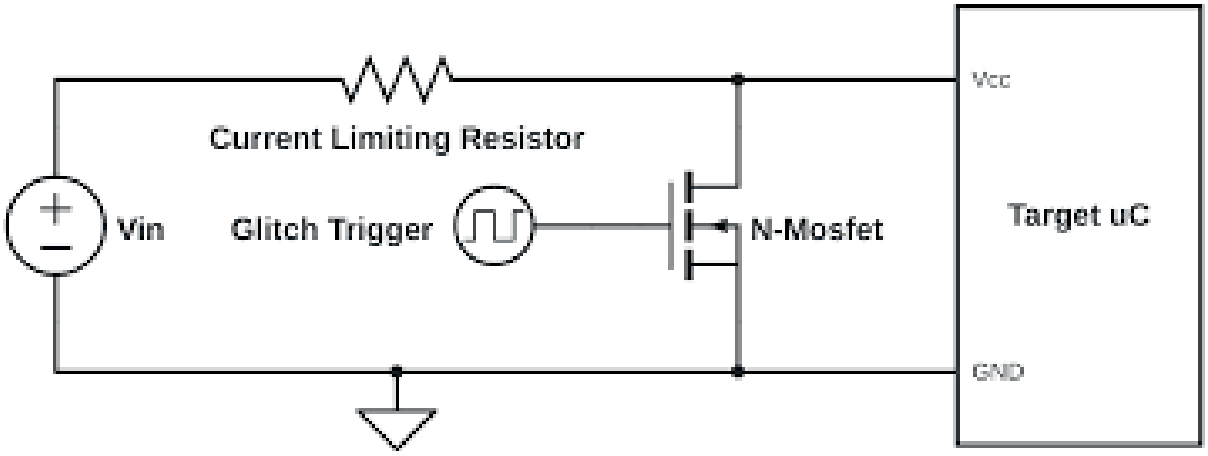

Figure 2. Standard voltage fault injection setup [3]

My setup makes slight modifications to the above setup, further elaborated in the section below. However, most essential components are still present: the fault injector, the target microcontroller, and the MOSFET.
The fault injector typically waits for a set amount of time or a trigger such as a rising edge before asserting the glitch.
The MOSFET is used to short the voltage to ground in a short period of time, meeting the strict timing constraint.
The target microcontroller would be the device under attack. Typically, these microcontrollers would integrate certain voltage regulation components, such as capacitors, providing a fixed and stable power supply to the processor. In order to improve the chances of glitching, we would need to remove such components or completely isolate the chip by removing it from the current PCB and transferring it to a standalone PCB.

Programming interfaces / Tools used.

The fault injector of choice in this project would be the Raspberry Pi Pico, a low cost microcontroller. The main program flow would be written in MicroPython, and the time sensitive code would be written in PIO assembly.
The target microcontroller in this project would be the STM32F103RB. There are 2 types of code that will be run on the microcontroller: The user configurable code that can be programmed onto the STM32 via the STM32CubeIDE . And the fixed bootloader code that is pre-written by ST, living in the system memory of the device.
A P&N MOSFET will be utilized to short the voltage to ground to meet timing constraints.

Ghidra will be utilized in order to statically analyze the bootloader code of the STM32 in order to better understand program flow and in turn, when to glitch the device.
An oscilloscope will be utilized to visualize the glitch, as well as observe the power trace in order to perform side channel analysis.

Overview of progression.

The main objective of this project would be to utilize voltage fault injection to skip key security checks and eventually dump out the flash memory of the microcontroller efficiently. To achieve this, I will be tackling this in 3 main stages:

1) Programming the STM32 with user code and performing voltage fault injection. Being able to control various parameters would help with getting the basis of fault injection ready.
2) Skip key security checks in the STM32's bootloader mode. This is code written by the manufacturer and thus, would be of higher difficulty but allows us to dump out memory
3) Write shell-code to the STM32 and execute it with the help of voltage fault injection. This would be the most optimal way to dump out memory of the STM32 using the least amount of glitches possible.

Terminology used.

Glitching: An alternate term used interchangeably with voltage fault injection
Delay Duration: The amount of time that the fault injector waits from the trigger before asserting the glitch
Glitch Duration: The amount of time that the glitch is asserted for
Glitching parameters: Delay and Glitch durations

3. In-depth look at the circuit/setup

Circuit Diagram.

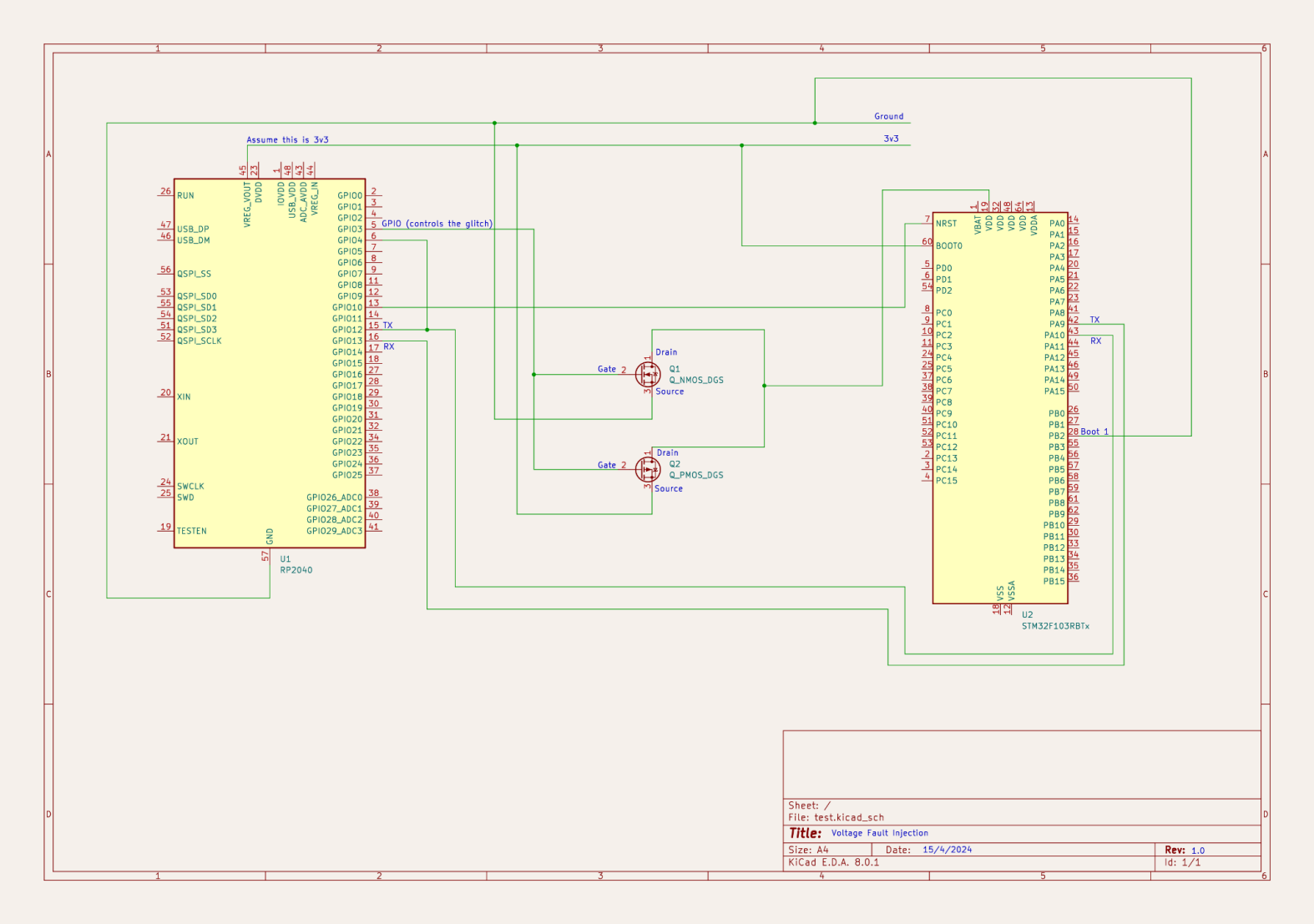

Figure 4. Circuit Diagram

Raspberry Pi Pico, Fault Injector.

The Raspberry Pi Pico consists of 2 key components, the main program and the Programmable Input/Output (PIO). A small note would be that the main program can set the PIO program to be active or inactive. When the PIO is active, it runs in parallel with the main program.

Main Program.
The main program will be responsible for the initialization and control flow of the fault injection setup. A few notable initializations are as follows:

- Pins 3&4: Utilized by PIO. Pin 3's corresponding register value will also be initialized to increase its output drive strength from 4mA (default) to 12mA, and to set its slew rate to fast. Both these changes will decrease the rise and fall time of the glitch waveform, helping to meet the time constraints

- Pins 12 and 13: Responsible for the UART communication between the target microcontroller and itself. It will have a baud rate of 115200, 8 data bits, even parity, and 1 stop bit.

- An on-chip file that is used to record the various glitches that occurred during the program flow.

Glitching the user code will be slightly different from the bootloader mode. For simplicity, this section will assume that we are trying to glitch the user program. The main program flow would be as follows:

1) Initialize the above
2) Set the PIO state machine to be active
3) In a nested "for" loop, vary both the delay and glitch duration. This step allows us to search for the optimal parameters to glitch the target device. Each loop is ran for approximately 5 seconds. During this process, the main program would need to communicate these 2 pieces of information with the PIO program. However, due to certain hardware limitations, this process is slightly more complicated and thus, will be elaborated further down below
4) When a glitch occurs, 2 scenarios can occur:
    a) If a crash happens, the program will cut the power to the STM32 and assert the NRST, causing the STM32 to reset itself
    b) The output will be different from the expected output due to skipping of instructions or data corruption. Therefore, the abnormal output, along with the corresponding delay and glitch durations, will be written to a file for analysis.

(The corresponding code used here would be voltage_test2.py)

Programmable Input/Output (state machine / PIO).
The PIO is responsible for the time-sensitive glitching process.
A few important characteristics of the PIO is as follows:

- It can run up to a clock frequency of 125 Mhz. I will be running it at 100 Mhz for easier calculations later on.
- Each instruction written in PIO takes exactly one clock cycle
- This means that each instruction takes 10ns to execute, allowing us to meet the timing constraint for the glitching process
- It contains 4 registers that can be utilized, more on this in the section below

The program flow would be as follows:

1) It takes in input from the TX FIFO register, dictating the glitch and delay duration
2) It moves data to the respective registers before waiting for a rising edge on Pin 4
3) Once a rising edge is detected, it waits for a short amount of time (delay duration), before asserting the glitch with Pin 3. Figure 5 shows the waveform for the glitch process
4) It then resets Pin 3 back to its default value, before delaying for a short while to allow the STM32's voltage to stabilize back to its operating voltage.
5) Loops back to 1)

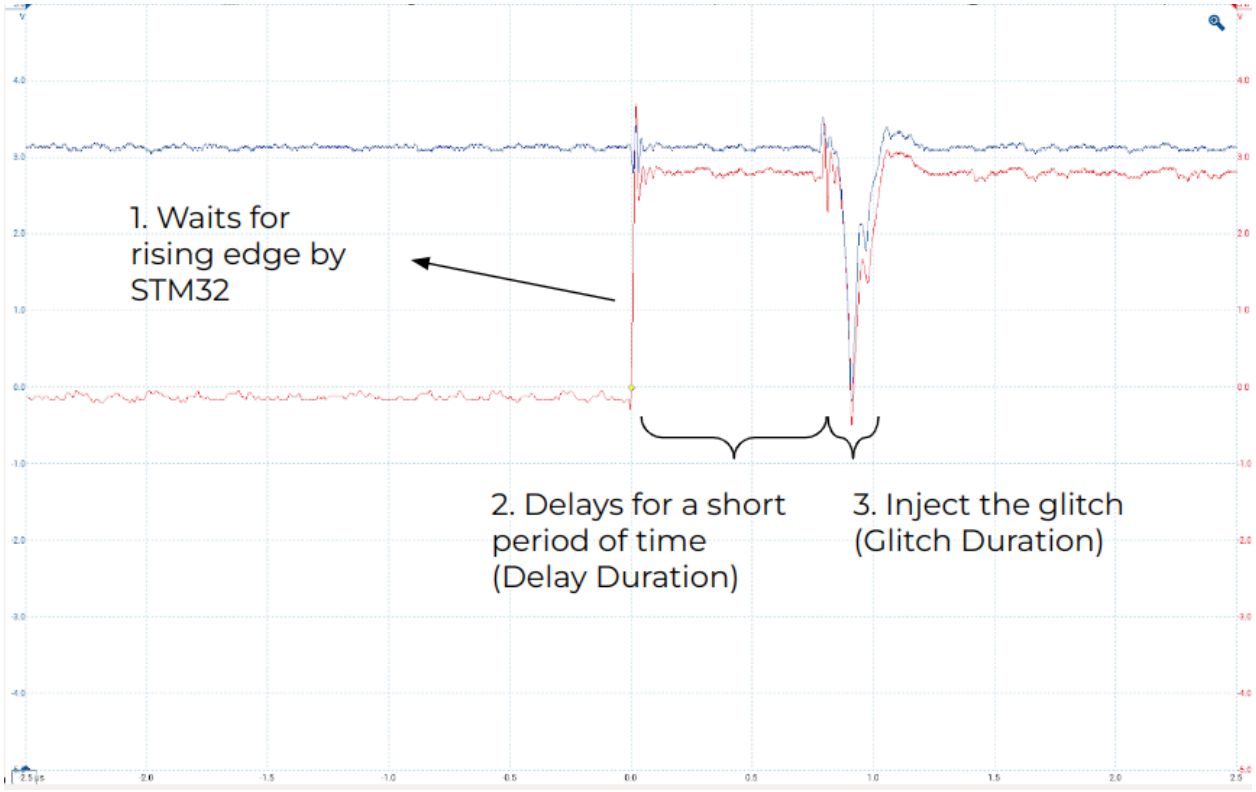

Figure 5. Glitch waveform

Raspberry Pi Pico Limitations and Workarounds.

1) Communication between main program and PIO
A few key limitations are as stated below:

- The PIO contains 4 32-bit registers, X, Y, TX FIFO and RX FIFO.
- While the registers are all 32-bit, only the least significant 5 bits are actually being used as data bits in the PIO.
- When data is being pulled from the TX FIFO of the main program to the RX FIFO of the PIO, data at the TX FIFO of the main program is lost.
- Due to the difference in operating speeds between the PIO and main program, the main program is unable to keep up with the PIO and send the required values continually
- Decrementing the X and Y registers modifies the values

In short, there is a space limitation as there are not enough registers compared to the number of variables being used.

Workaround:

- Since the PIO is only utilizing 5-bits of data at a time, we can keep the values to be used to a 5-bit limit.
- Then we concatenate these 5-bit values into one 32-bit register and pass that to the PIO
    - Bits 0 to 4: Short delay cycles
    - Bits 5 to 9: Long delay cycles
    - Bits 10 to 14: Glitch cycles
    - Bits 15 to 31: Unused for now. Can be edited for further use later if needed
- The PIO can then utilize Logical Shifting to shift out the appropriate values at the appropriate timings to be utilized, resolving the space issue.

2) The PIO is running at 100Mhz.
The PIO is only accurate up to 10ns. While this is accurate enough to conduct the glitch, finer control of the glitch is not possible. As such, fine tuning to get the truly optimal parameters is limited.

3) A delay of 140ns is always present due to noise

4) Delay duration, with my current implementation, only goes up to 10680ns. Any longer would require some changes to the PIO's code.
3 & 4 are not very major limitations, just points to take into account when doing calculations in the main program.

P & N MOSFET.

Looking at the circuit diagram in Figure 5, the MOSFET is wired in a way that it forms a crowbar circuit, enabling fast draining and supplying of voltage. While most literature has mentioned that a single N MOSFET is sufficient, my testing has shown otherwise. Using only one type of MOSFET does not meet the timing constraints.

Wires.

Manipulating the voltage of a device is very sensitive to external factors. Therefore, the noise included in breadboards and normal wires adds in a layer of inconsistency. Moving the wires around in the breadboard will completely change the glitching parameters. Therefore, wire wrapping is utilized on the crucial components that are involved in the glitching process. This improves the consistency of the glitching parameters greatly and is a crucial component in the circuit.

STM32F103RB.

The STM32 has 2 different codes that I am interested in glitching: user code and bootloader mode. In this section, I will only be mentioning user code as it sets up the basis for glitching the bootloader code later on.

The user code contains:

- HAL_GPIO_WritePin, enabling me to set the rising edge on whichever section of code I want to glitch
- An “If” conditional that always returns false and the program flow would never enter
- A “for” loop that runs for 5 to 1000 cycles, depending on how the user sets it.
- Changing the clock frequency of the STM32

```
131       // Original code
132       //HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, 1);
133       i = 0; j = 0; ctrl = 0;
134
135       if (flag != 0){
136           //Prints only once when first booted
137           //Allows us to know if the STM has restarted when voltage has d
138           sprintf(tx_buffer, "Inside of IF statement\n\r");
139           HAL_UART_Transmit(&huart1, tx_buffer, 30, 10);
140           HAL_UART_Transmit(&huart2, tx_buffer, 30, 10);
141
142           while (1){
143               HAL_Delay(100);
144           }
145       }
146
147
148       HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, 1);
149       for (i = 0; i < 10; i++){
150           for (j = 0; j < 100; j++){
151
152               ctrl++;
153
154           }
155       }
156       HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, 0);
157
158       sprintf(tx_buffer, "i = %i j = %i ctrl = %i \n\r", i, j, ctrl);
159       HAL_UART_Transmit(&huart1, tx_buffer, 30, 10);
160       HAL_UART_Transmit(&huart2, tx_buffer, 30, 10);
161       HAL_Delay(50);
162
163   }
164   /* USER CODE END 3 */
```

Figure 6. User code of the STM32

However, the STM32 has several capacitors on the board to regulate the voltage being supplied. This inhibits the ability to glitch the STM32. Therefore, after studying the schematics, I identified the key capacitors to be removed and desoldered them from the board.

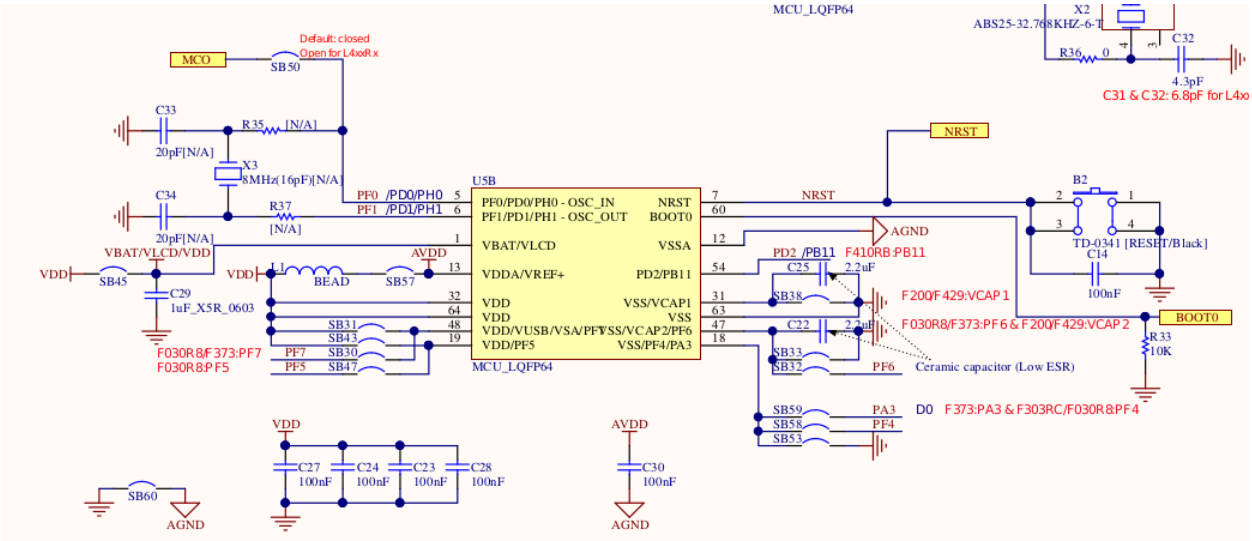

Figure 6. Schematic diagram of STM32. Removed C23, C24, C27, C28, C29

4.1 Observations (Glitching user code)

Firstly, I attempted to glitch the “for” loop of the user code as it has the largest surface area and largest probability that a glitch can occur. After tuning the parameters, I was able to observe the effects of glitching: Corruption of data, skipping of instructions, and more commonly, crashing of the STM32.

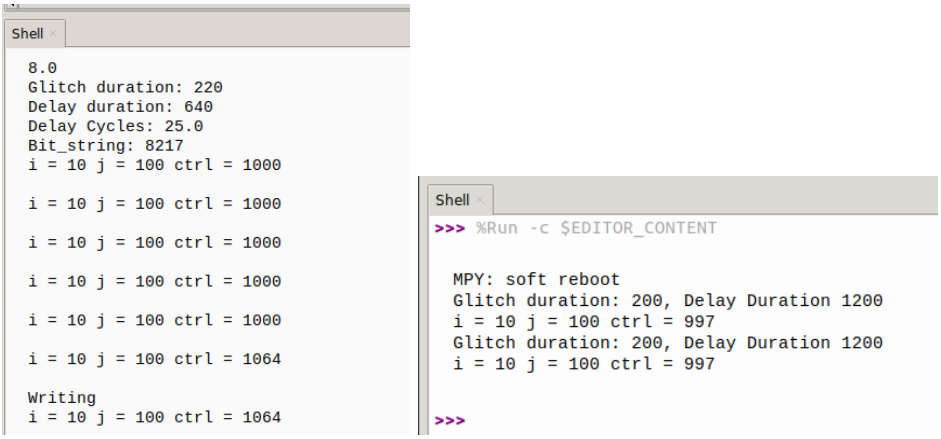

Figure 7. Corruption of data (left), Skipping of instructions (right)

Afterwards, I attempted to glitch into the “if” statement, which the program flow will usually never reach. However, by tuning the glitch parameters correctly, the conditional check was skipped and the program entered the IF statement.

```
0
7.0
Glitch duration: 200
Delay duration: 980
long delay cycle: 1
Delay Cycles: 10
Bit_string: 7210
i = 5 j = 0 ctrl = 5

i = 5 j = 0 ctrl = 5

i = 5 j = 0 ctrl = 5

i = 5 j = 0 ctrl = 5

i = 5 j = 0 ctrl = 5

i = 5 j = 0 ctrl = 5

i = 5 j = 5 ctrl = 5

Writing
Inside of IF statement
```

Figure 8. Glitching into the IF statement

When conducting the glitch, I noticed that there is a small allowance of time available for glitching the same instruction. Therefore, the next step was to look into the optimal parameters for glitching. The process was as such: Find an area to glitch, run the program for 1000 iterations and find the glitch success rate, change only the delay duration for sample taken. (Only delay duration was changed as there was only 1 relatively stable glitch duration at the point of testing.)
After obtaining the results, I plotted them on a graph as shown in Figure 9 below.

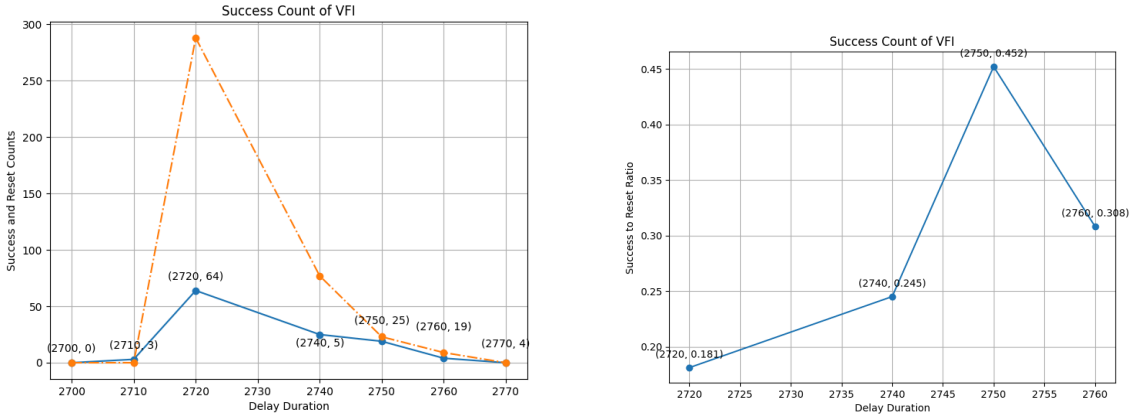

Figure 9. Successful glitch counts (left), ratio of success to crash(right)

From the graph(left) above, the timing with the highest glitch count also had the highest amount of crashes. As such, a better schematic should be used, and for that, I turned to success to crash ratio, taking into account only points that have a success rate of 1% or higher. The results are plotted as shown on the graph on the right. Looking at the plots above, it is reminiscent of a normal distribution graph, with one point peaking higher than others. A small limitation would be the lack of data points, but with the Pi Pico being unable to obtain more accurate reading, it has to be a compromise.

Another variable that was tested was the operating frequency of the STM32. While this was not extensively tested, preliminary tests shows that running at a higher frequency does improve the success rates of glitching. Similarly, this test was also done over 1000 iterations.

Frequency	Successful Glitches	Crashes
24 Mhz	20	24
64 Mhz	33	0

Table 1. Glitch success rates, varying frequency

While the exact mechanisms of how voltage fault injection works is unknown, there has been research explaining this phenomenon. A paper by Zussa et al., 2013 [4] gives a compelling argument to why voltage fault injection works. A summary would be as such:

- On a digital logic level, data is released from a register bank on a clock's rising edge, processed through the logic, before being latched onto the next register on the next rising edge
- The clock period must be longer than the maximum data propagation time through the D Flip-Flops, the logic circuits, and other parameters. Otherwise, data being read will be incorrect
- In voltage fault injection, it artificially increases the propagation delay through these logic gates without causing the device to reset. This causes the wrong data to be latched on during the rising edge of the clock.
- A higher clock frequency would decrease the maximum propagation time through the logic gates, therefore, glitching would have a higher chance of success.

Defending against Voltage Fault Injection.

Measures to defend against Voltage Fault Injection can be done on both the software and hardware side. In nccgroup's article about defending against fault injection attacks[5], various measures were given, such as resiliency through redundancy and random delays.

To test this, I used a nested IF statement and tested the success rates of glitching. This now makes glitching into the IF statement harder as there is a need to skip 2 conditional checks OR corrupt data. The initial tests showcases that these software measures do work, drastically reducing the chances of fault injection. Furthermore, this implementation has holes to exploit. By improving on this code, to cover for the aforementioned cases, the success rates of fault injections would likely see a significant drop.

Single IF Conditional	Nested IF Conditional
63 Glitches	3 Glitches

Table 2. Glitch success rates of IF conditionals

On the hardware side, various circuitry is present to detect sudden fluctuations in voltage. These include Brown-out Detection and tunable replica circuits. Both are able to detect fluctuations in current and cause a hardware reset, stopping fault injection from occurring. [5]

However, both software and hardware each have their own limitations.
On the software side, redundant checks and random delays reduce the code readability and increases the time/space required to execute/store code, which might not be ideal if the device is time sensitive. Additionally, the compiler can optimize out these crucial redundant checks.

For hardware, addition of these circuits would increase the manufacturing costs of the microcontrollers. Furthermore, each circuit can only defend up to two different fault injection attacks, and there are 4 classes of fault injection attacks, not including side channel analysis. There would also be an issue of space limitation on the microcontroller. With microcontrollers designed to be small and cost effective, additional of these circuitry will only serve to increase the silicon space and drive up the cost of production, making these hardware changes not ideal.

4.2 Observations (Glitching bootloader mode)

The objective of this project is to dump out the flash memory content of the STM32 microcontroller using voltage fault injection. To look into how to achieve this, we first look into the documentation provided by ST.

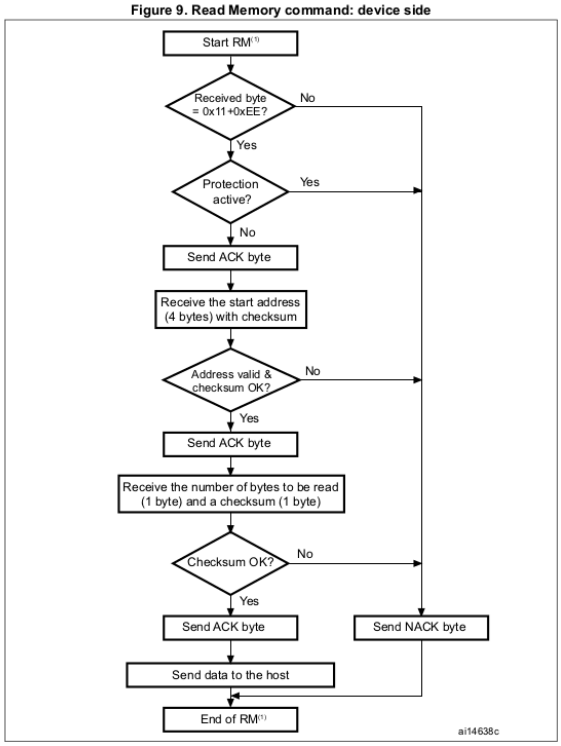

Figure 10. Read Memory flowchart

On the STM32, there is a register containing a bit known as the Read Protection bit (RDP). When in bootloader mode, using commands such as Read Memory would cause the STM32 to check against the RDP bit. If the RDP bit is set, it prevents memory from being accessed. Most programmed devices would have this bit set to prevent information from being leaked. Writing a 0 to the RDP bit causes a mass erase of memory, protecting the integrity of the memory. Looking at the program flowchart, we can see exactly where the RDP bit is being checked. This is a key check that we want to skip in order to extract the memory of the STM32 microcontroller.

To improve the chances of glitching the check, we first look into Ghidra to do static analysis on the bootloader firmware. After identifying the memory region associated with the bootloader code, I used the ST-link debugger to dump out the bootloader code on a STM32 that is not programmed yet. Putting the file into Ghidra, I was able to analyze the bootloader code and determine the line of code that was doing the checks.

```
uint read_memory_command(void)

{
  int iVar1;
  uint uVar2;
  uint uVar3;
  uint uVar4;
  uint uVar5;
  uint uVar6;

  iVar1 = RDP_Get();
                    /* r0 has a value of 0x1 now
                       cbnz comapres register with 0. If not equals to 0, jump */
  if (iVar1 == 0) {
    send_data_through_UART(0x79);
    uVar2 = read_uart();
    uVar3 = read_uart();
    uVar4 = read_uart();
    uVar5 = read_uart();
    uVar6 = read_uart();
                    /* Checksum by XOR-ing the address bits */
    if (uVar6 == (uVar2 ^ uVar3 ^ uVar4 ^ uVar5)) {
      send_data_through_UART(0x79);
      return uVar2 << 0x18 | uVar3 << 0x10 | uVar4 << 8 | uVar5;
    }
  }
  send_data_through_UART(0x1f);
  return 0x55555555;
}
```

```
                        **************************************************************
                        *                          FUNCTION
                        **************************************************************
                        undefined RDP_Get()
             undefined          r0:1           <RETURN>
                        RDP_Get                                         XREF[5]




       1ffff132 c0 49           ldr         r1,[->Peripherals::FLASH]
       1ffff134 00 20           movs        r0,#0x0
                        Load in Option Bytes
       1ffff136 c9 69           ldr         r1,[r1,#offset FLASH.OBR]
                        RDPRT, WDG_SW, nRST_STOP, nRST_STDBY are shifted in
       1ffff138 89 07           lsls        r1,r1,#0x1e
       1ffff13a 00 d5           bpl         LAB_1ffff13e
       1ffff13c 01 20           movs        r0,#0x1

                        LAB_1ffff13e                                    XREF[1]
       1ffff13e 70 47           bx          lr
```

Figure 11. Ghidra analysis of bootloader code

This still leaves one issue: the reference point to start the glitching process.
Luckily, the STM32 requires the Pi Pico to send a specific sequence of bytes before the STM32 carrying out certain commands. We can utilize the rising edge of the UART communication as a reference point for the glitch.
Furthermore, looking at the power trace of the STM32, a slight fluctuation in power trace is observed at a certain point between the end of Pico's transmission and start of STM32’s transmission. This power fluctuation is due to the STM32’s power consumption when accessing its flash memory. Therefore, by analyzing fluctuations in the power trace, we can determine the accepted range of time to successfully glitch the RDP check.

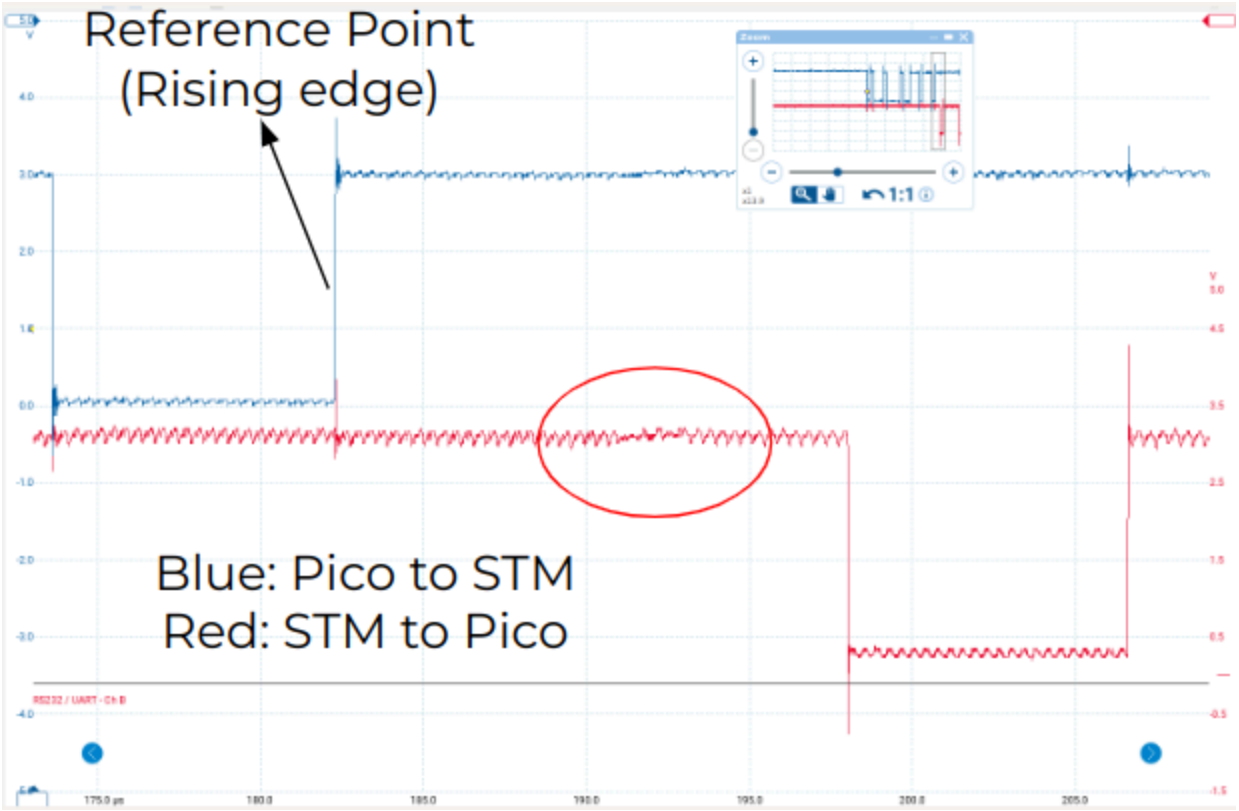

Figure 12. Reference point and Power trace fluctuation

After calculating and tuning the glitch parameters, the RDP bit was skipped and the memory was dumped out.

```
Glitch duration: 180
Delay duration: 10460
Bit_string: 7140

31
31
31
31
31
31
Glitched
121
79040008eb0402431a1b68551a2b60091ff9d1bcf1040f51d14e488
0454ed1e00702d004f0fe04a41cfff71bfdfff713fdfff70bfd484b
186940f010001861fff7fffc64b109eb040208eb0405101b0088291
bc0b20880fff7f3fca41ef6d1186920f0100018617920fff7defcc6
f80c7d24e0fff75efd01283ff411af7920fff7d3fcfff7aefdff280
5d1fff7aafda8b9fff7ddfd12e005ad441c2946fff7affdaa287ff4
fcae4cb14ff0006c15f8010b0ceb8020fff7bafd641ef7d1792016e
77920fff7b1fcfff78cfd05ad441c2946fff795fdaa287ff4fcae00
223cb1012115f8013b01fa03f00243641ef8d1fff7b2fc19483521c
1601046fff721fd11e0
Traceback (most recent call last):
  File "<stdin>", line 228, in <module>
NameError: name 'quit' isn't defined
```

Figure 13. Dumping of Memory with RDP bit set

One issue with this method of using the Read Memory command would be that it only dumps out memory in blocks of 255 bytes. Dumping out the whole flash memory using this method would take a significant amount of glitches and time. Therefore, a more optimal way to dump out the memory is needed.

Looking at the other bootloader commands available, the “Write Memory” and “Go” commands stand out. Write memory allows the user to write memory directly to the flash or RAM of the STM32 while the Go command allows the user to jump to a specific address in memory and execute from there. Taking reference from the concept of shellcode, the plan going forward would be to write executable code as a payload and execute it using the Go command, dumping out the memory of the STM32.

Shellcode.
Firstly, we will use Ghidra to analyze the various peripherals that have been initialized and de-initialized. Doing so will allow us to save code space when writing the payload later on.

```
                    /* Set up RCC */
*(undefined4 *)puVar4 = 0xaaaa;
puVar4 = RCC_Peripheral;
*(undefined4 *)(RCC_Peripheral + 4) = 0x100000;
*(undefined4 *)(reserved(?) + 0x60) = 1;
puVar13 = RCC_Peripheral;
do {
                    /* Waits for PLL clock ready flag to be unlocked
                       */
} while (-1 < *(int *)puVar4 << 6);
                    /* Sets the Clock Configuration Register

                       Feeds system clock into PLL
                       PLL Multiplication factor set to PLL input clock x6 (output freq must not
                       exceed 72Mhz)
                       */
*(undefined4 *)(RCC_Peripheral + 4) = DAT_1ffff46c;
                    /* APB2 Peripheral Clock Enable Register to enable USART1 and IO Port A */
*(undefined4 *)(puVar13 + 0x18) = 0x4004;
puVar4 = Alt_Function_IO;
Peripherals::SCB.VTOR = 0;
                    /* Pointer Pointing towards address of GPIO Port A

                       All pins but Pin 9 are in Output mode, GPIO Open-Drain

                       Pin 9 is Alt Function Push Pull, Output mode of Max Speed of 50Mhz */
*(undefined4 *)(Alt_Function_IO + 0x804) = 0x4444_44B4;
puVar13 = PTR_USART1.DR_1ffff478;
                    /* Setting CTRL to 4 indicates a systick reset

                       LOAD: loads the value into the register and when it hits 0, triggers
                       exception request and countflag is activated(?) */
Peripherals::STK.CTRL = 4;
Peripherals::STK.LOAD_ = (int)puVar4 << 3;
do {
                    /* Sets bit 15 of GPIO PortA, Input Direct Register, the first bit that is not a
                       Reserved bit, to the MSB */
} while (-1 < *(int *)(puVar4 + 0x808) << 0x15);
do {
} while (*(int *)(puVar4 + 0x808) << 0x15 < 0);
do {
} while (-1 < *(int *)(puVar4 + 0x808) << 0x15);
Peripherals::STK.CTRL = 5;
do {
                    /* Baud Rate Detection and Set up on USART1 */
} while (*(int *)(puVar4 + 0x808) << 0x15 < 0);
```

FIgure 14. Analysis of bootloader code 1

```
165                     /* Go Command */
166       puVar5 = (undefined4 *)read_memory_command();
167       cVar2 = checks_for_readable_address();
168       puVar4 = base_addr_RCC;
169       if ((cVar2 == '\x02') || (cVar2 == '\x03')) {
170                     /* Initialize the register for RCC, Clock Configuration Register */
171         *(uint *)(base_addr_RCC + 4) = *(uint *)(base_addr_RCC + 4) & DAT_1ffff74c;
172                     /* Initialize the RCC, Clock Control Register */
173         *(uint *)puVar4 = *(uint *)puVar4 & DAT_1ffff750;
174         *(uint *)puVar4 = *(uint *)puVar4 & 0xfffbffff;
175         *(uint *)(puVar4 + 4) = *(uint *)(puVar4 + 4) & 0xff80ffff;
176                     /* Reset SysTick control and status register */
177         Peripherals::STK.CTRL = 0;
178                     /* End of RCC, Systick, and Interrupt initialization */
179         enable_IRQ_Interrupts();
180                     /* Load from r4 +4 into r5.
181
182                        R5 will be the starting address of the user code
183
184                        Therefore, at mem address r4 +4, should contain the starting address of
185                        malicious code
186                         */
187         pcVar11 = (code *)puVar5[1];
188         Checks_for_priviledge_mode(*puVar5);
189         (*pcVar11)();
190         goto Start_Of_Loop;
191       }
192     }
```

Figure 15. Analysis of bootloader code 2

4.3 Optimal Extraction of Flash Memory

At first, I attempted to use assembly and jump to the send_uart function but was unable to do so. Instead, my supervisor recommended writing the payload in C, giving me a link to the bare-metal implementation https://github.com/fcayci/stm32f1-bare-metal.

Payload.

Using this, I modified the makefile, linker script, and the C code.

For the makefile and linker script, it was changed such that the code will be written to the RAM region of the STM32. At first, I was considering writing to the Flash memory but there would be uncertainties: the area of flash might already have been written to, and with how the flash works, the payload will not overwrite the existing code. Writing and executing from RAM solves the issue and is the best option.

In the C code provided by the GitHub repository, I first modified the main program such that the main functionality would be to print out the contents of the flash memory. Next, I modified the system clock being used as the options provided by the GitHub repository were not suitable. Additionally, I once again utilized Ghidra to analyze the bootloader code, identifying what was initialized and de-initialized in order to cut down as much code as possible. Eventually, the payload includes:

- Initialization of the system clock
- Initialization of GPIO alternate function output (for USART communication)
- Initialization of USART1 Baud Rate Register (to account for the change in clock frequency)
- A while loop to dump out the flash memory of the STM32

After compiling the code and verifying that it works (with a second ST32 board, RDP bit not set), I utilized an online .bin file to hex string converter to obtain the hexadecimal string to be used.

Dumping of STM32 memory using payload (with RDP bit set)

In this section, I would need to glitch both the Write Memory and Go Commands. Both commands check the RDP bit before carrying out their functionality. Therefore, we will aim to skip this check using voltage fault injection. The area to glitch can be done using power trace analysis as stated in 4.2.

Small note that the PIO code for glitching the Write Memory command has some slight changes to accommodate for the longer delay required.

After making some changes to the micropython code on the Raspberry Pi Pico and various tests, I was able to glitch both commands, executing the payload, and dumping out the flash memory of the STM32.

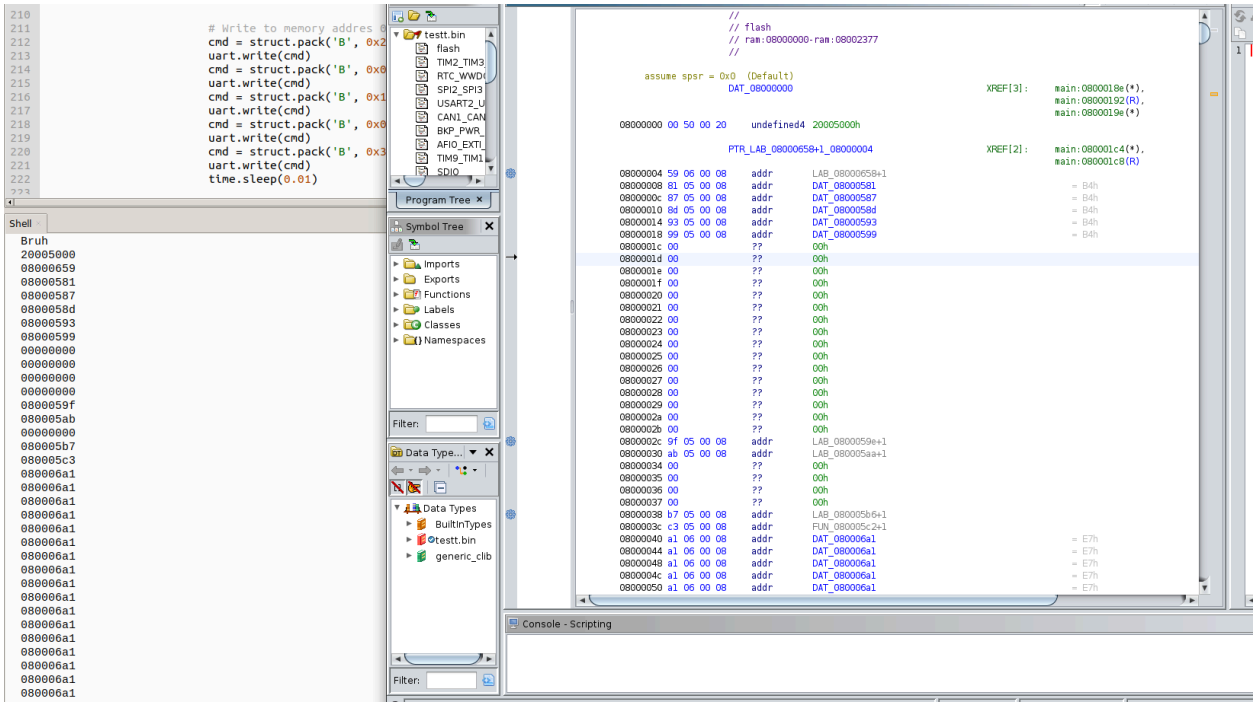

Figure 16. Dumping out Flash memory & comparing to Ghidra

All relevant code is compiled onto my GitHub, specifically, the week 13 folder which contains the working code for the payload and memory extraction.

5. Future work

Moving on, I would firstly be trying to overclock the Raspberry Pi Pico to run at beyond the recommended operating frequency. This would help to address one of the limitations stated in section 3.
Next, I will be looking into another chipset, and how to defeat its security mechanisms with voltage fault injection.

References and citations (IEEE)

[1] K. Zetter, “Cracking a $2 million crypto wallet,” The Verge, https://www.theverge.com/2022/1/24/22898712/crypto-hardware-wallet-hacking-lost-bitcoin-ethereum-nft (accessed Apr. 10, 2024).

[2] Colin O’Flynn https://circuitcellar.com/author/colin-oflynn/, “AirTag Teardown and security analysis,” Circuit Cellar, https://circuitcellar.com/research-design-hub/design-solutions/airtag-teardown-and-security-analysis/ (accessed Apr. 10, 2[1] C. Bozzato, R. Focardi, and F.

[3] Palmarini, “Shaping the glitch: Optimizing Voltage Fault Injection attacks,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 199–224, Feb. 2019. doi:10.46586/tches.v2019.i2.199-224

024).

[4] L. Zussa, J.-M. Dutertre, J. Clediere, and A. Tria, “Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism,” *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, Jul. 2013. doi:10.1109/iolts.2013.6604060

[5] “Software-based Fault Injection Countermeasures (part 2/3),” NCC Group Research Blog, https://research.nccgroup.com/2021/07/08/software-based-fault-injection-countermeasures-part-2-3/ (accessed Apr. 11, 2024).

2. How the project / research can be applied to DSO/Division's mission and vision?

With the increasing usage of IOT devices in our daily lives, it increases the surface area of attack for malicious hackers. Furthermore, microcontrollers are becoming more accessible in terms of availability, cost and usage. As such, there is a need to improve the security aspect of our embedded and IOT devices, to not only defend against software attacks, but hardware attacks as well. My project gives a proof of concept that such attacks are possible with components that are readily available to the general public. As such, it will provide a basis for researchers to improve the security of embedded/IOT devices such that newer devices are able to be more resilient against malicious attackers.

3. Challenges faced during your internship

Firstly, I would like to address the technical challenges that I faced when doing my project. The first challenge I faced was my lack of intricate electrical circuit knowledge. Electrical terms such as crowbar circuit and parasitic currents were rather foreign to me. Furthermore, abnormal and inconsistency due to electrical noise was something that I did not account for at the early stages of the project. In a project that can be affected by such sensitivities, it took me some time to rectify it. It was only when I seeked my supervisor's clarifications that I got my answers to these queries.

The next challenge I faced was my inexperience in linking what I have learnt in theory to actual practice. For example, I have learnt how flash memory works in my computer architecture classes. However, I was unable to make the link between it and the fluctuations observed on the power trace. These kinds of scenarios whereby I have learnt something in classes before but unable to relate it to real-life occurrence happens rather often and is something that I am trying to improve on.

Excluding technical challenges, I also faced other challenges as well. As this is my first internship experience, this experience and environment is completely new to me. Adapting to the workplace, the people and the work itself took some time. But gradually, I started to get used to the culture here. Everything seems less daunting and more manageable.

The last challenge I want to work on would be my communication skills. I am a soft-spoken person and find it hard to speak up on multiple occasions. This is evident during my weekly reviews with my supervisors, where I would usually stutter a lot and

sometimes, am not able to fully convey my thoughts properly. I have been trying to work on this by making notes before attending our weekly reviews but I do acknowledge that I still need to work on this area.

4. Lessons learnt from the tasks assigned to you

Firstly, I was able to learn about static analysis using Ghidra. During my project, I needed to have a better understanding of the bootloader firmware that was present in the STM32 microcontroller and therefore, it was suggested that I utilized Ghidra to do firmware analysis. It took some time to get used to using Ghidra for static analysis but eventually, after some effort, I was able to demystify the firmware. I believe this skillset is rather important as it gives me a deeper understanding of the embedded system and how microcontrollers work at the bare-metal level.

Secondly, I was able to gain hands-on experience with working with hardware related tools. During the internship, I was able to work with oscilloscopes, soldering, and interfacing various components together. I was able to learn from my supervisors on how to operate these devices and picked up a few skillsets along the way.

I was also able to gain a better understanding of the architecture of microcontrollers on a deeper level. Notably, would be my supervisors encouraging me to have a deeper look into the UART architecture of microcontrollers. While most textbooks offer a high level overview of the topic, looking into how specific microcontrollers implement it in practice along with the firmware analysis on Ghidra, it provided me with a more practical outlook on the topic.

Lastly, this project also helped me to hone my problem solving skills. While working on the project, I experienced multiple issues. As such, I had to utilize the resources I have to try to resolve the issues, be it reading documentation, asking my fellow interns, or

learning from videos. Each new problem encountered was a new learning experience for me and help me to think deeper and more critically, honing my problem solving skills.

5. Positive experiences in DSO

Firstly, the environment for interns was conducive for working and learning. We are given a good amount of space and equipment to start. I still remember the tech support and supervisors were very patient during my first week when I was having issues logging into my DSO email account. Furthermore, any equipment needed for my project or access to the lab are readily provided by my supervisors.

Speaking of whom, they are very pleasant to work with. During our weekly reports, they would give constructive feedback on my work and push me to learn more about certain interesting aspects of my project. Instead of giving out the solutions, they will give hints on what to do and leave it to me to figure the rest out, making this a invaluable learning experience.

The interns as a group are also very inclusive and friendly to each other, often going for meals together. I was also able to find a group to play table tennis with during our spare time.

I also appreciate the multiple events held by DSO to celebrate the holidays and the talks given by the full timers to give us a general overview on various topics that they specialize on.

Overall, the people and the environment makes the experience I have at DSO a positive one.

6. Areas of improvement

While I do enjoy working on my project, I also felt that the work done was rather isolated. Whilst some components of the intern's project do overlap, most of the work done is on an individual scale. To address this, I would like to suggest making some projects pair / group work. This would give interns the experience of working with others, building up collaborative skills such as communication skills and more technical software skill sets such as collaborating on GitHub.

7. Changes you would recommend

As stated in 6., I would recommend having more collaborative work, either within the interns or with the supervisors so that the project feels more meaningful.

8. Whether you initial expectations has been met

Coming into DSO, I did not know what to expect as it was my first internship. However, I was gladly surprised at the conducive environment provided for interns.

There were various amenities other than the office: a library, gym, recreational room, etc. The various other options provided interns with areas to explore outside of working hours and make use of our sports hour. Personally, I have recently been utilizing my sports hour to play table tennis with some of my fellow interns. All of these places provide interns a place to relax after work or during our lunch breaks.

The full-timers also sometimes hold a sharing session, sharing on certain topics relating to their expertise. The most recent sharing session was about cloud security. These sharing sessions are a nice way to introduce interns to various other topics aside from our project, granting us exposure to other fields.

As for working on the project itself, the workload was manageable. Although I did encounter issues that I could not resolve by myself at certain points, my supervisors were always present and responsive whenever I reached out to them for help. With their help, I was able to learn more about hardware architecture and the security implementations that comes along with them. Furthermore, this internship has given me firsthand industry experience and gave me insights on hardware security.

9. Thoughts on your internship supervisor

My internship supervisors have been very nice and helpful to me throughout my internship journey.

During weekly reviews, they would be attentive and give helpful advice to enable me to make progress in my project. Usually, they would point me in the right direction but not explicitly give the answer, giving room for self-learning. They would also push me to think deeper as to why certain anomalies in hardware occur and investigate the reasoning behind them. All these made for very fruitful feedback sessions, allowing me to learn as I work on the project.

My supervisors were also very responsive whenever I needed help, equipment or access to the lab. Communication with them is is always clear and efficient.

With their overall experience and professionalism, it provided a conducive learning environment and I am grateful for their help so far.

10. Thoughts on your future career

As for my future career, my end goal would be to focus on hardware security. However, I do recognise that it is a niche field and requires a level of skill in hardware / embedded systems. Therefore, my idea of progression would be to first gain experience in these fields before I move onto hardware/embedded system’s security.