Universiteit Leiden
The Netherlands

Opleiding DSAI

Evaluating the robustness of neural networks in the presence of fault injection attacks: simulations versus measurements

Joris Duurt Cornelis Alkema

Supervisors:
Nele Mentens & Nuša Zidarič

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

21/08/2024

Abstract

This thesis investigates the effect of fault injection attacks, specifically clock glitch attacks on neural networks implemented on embedded hardware. Using the ChipWhisperer (CW) platform, it explores how successful clock glitches can be carried out. It analyzes their impact on the internal behaviour of a multi-layer perceptron (MLP) classifier trained on the MNIST dataset. A practical setup for investigating the effect of clock glitches on neural network execution and performing successful attacks on the hidden layer and activation functions is proposed. The research encounters challenges in precisely determining glitch injection locations and reproducing attacks in debugging sessions. It concludes that the fault model used for reproducing clock glitches might not be as straightforward as skipping instructions. Despite these limitations, the study provides a research setup for fault injections on the internal CW target, and insights into clock cycle glitches on an MLP.

Contents

1 Introduction

AI models are becoming more popular in everyday use. Consequently, there is an increasing need to deploy AI models closer to the edge, such as on embedded computing systems. Examples of these AI implementations are neural networks, which are used in various fields, such as computer vision, image analysis, autonomous driving and personal assistants. When deployed on embedded devices, these neural networks are vulnerable to fault attacks, where computer components are exposed to conditions outside their intended operation. These attacks are to disrupt, steal model info or change the output of the neural network.
Due to the growing use of neural networks, the research into the security of these implementations is increasing, and such is the research into fault attacks of various implementations. The hardware setups for these types of research are expensive, complicated and target-dependent. Simulations are possible but often forfeit physical accuracy. How many bits and what bits are flipped, or what instructions are corrupted and or skipped during clock/voltage glitching is not yet straightforward and easy to derive.
This thesis considers hardware implementations of clock glitches on the ChipWhisperer platform. Fault attacks on a neural network are done and it is investigated how these faults come into effect deep within the network. It is then tried to analyze how these faults affect the program flow. The analysis is focused on a multi-layer perceptron (MLP) on an MNIST dataset k-classification problem. Multiple successful faults and their internal effects on the network are investigated. This Thesis has the following main contributions:

- It proposes a practical setup for analysing the effect of internal clock glitches on the execution of a neural network.
- It performs successful clock glitch attacks on the hidden layer and discusses the effect on the internal behaviour of the neural network.

1.1 Problem Definition

This thesis explores the following research questions:

1. How are successful clock glitches carried out on an embedded neural network?
2. How do successful clock glitches affect instructions in an embedded neural network?

2 Background

In this section, the concepts and relevant topics for this thesis are described. The theory behind fault attacks is defined as well as the working of multi-layer perceptrons. Simulations for fault attacks are also described.

2.1 Multi-Layer Perceptrons

A Multi-Layer Perceptron (MLP) is a name for a feed-forward artificial neural network, consisting of connected neurons with a nonlinear activation function. From now on, these will be referred to as "Neural networks" mostly. Neural networks consist of an input layer, which takes in the input to the network, a set of hidden layers, which abstract and model information, and an output layer, where the output is presented. For a fully connected network, which will be used in this thesis, each layer is connected to all the neurons in the subsequent layer. The connections are defined by a 'weight' value, which is used when calculating the output of a neuron. A weighted summation of all the node's inputs and their respective weights defines the node's value. A diagram is shown in Figure 1.

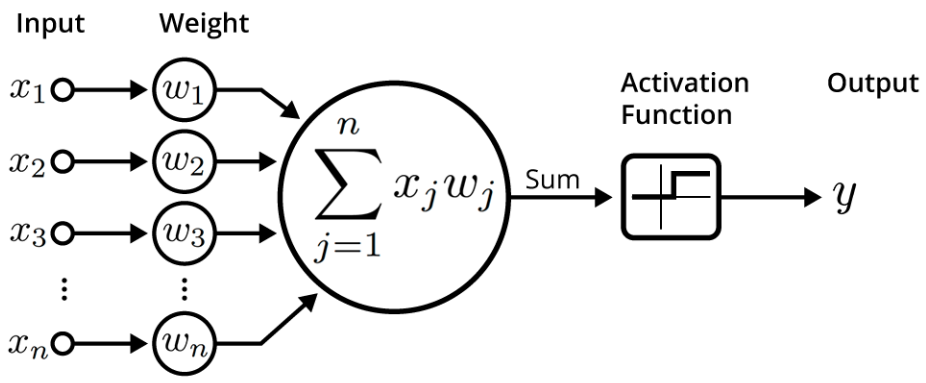

Figure 1: Illustration of a neuron in a neural network using a step function as an activation function.[McC21]

Each neuron is a node with a non-linear activation function. These functions have different effects on the neural network and presumably respond differently to fault attacks. Commonly used activation functions such as TanH, Relu, Prelu and Sigmoid are shown in Figure 2.

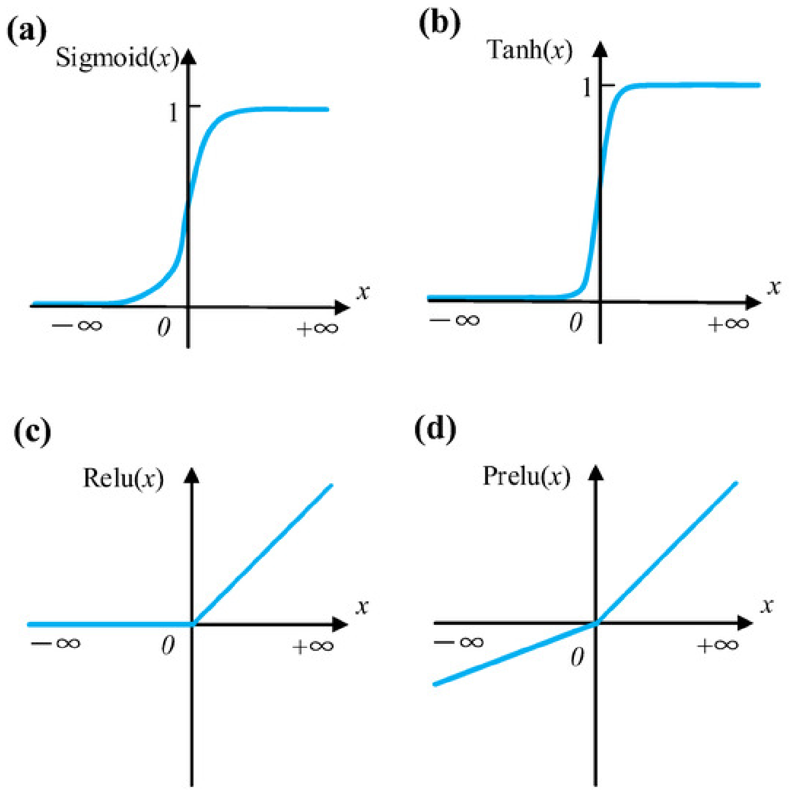

Figure 2: Several typical activation function images: (a) sigmoid, (b) tanh, (c) relu, and (d) prelu. [WXD+21]

These layers of connected neurons form a network, defined by the size and amount of these layers as well as the activation functions used and the connections between layers. An example of such a network is shown in Figure 3.

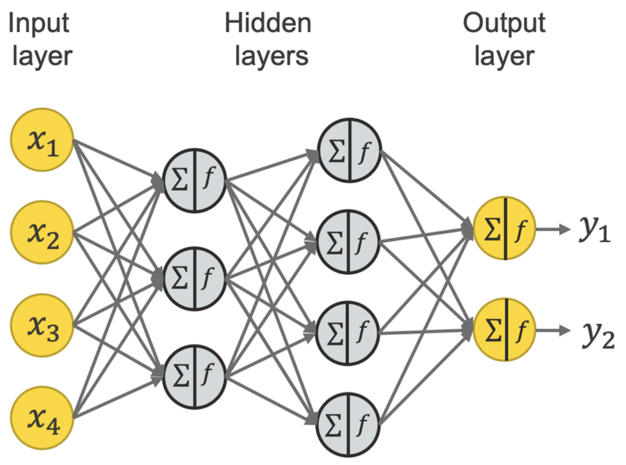

Figure 3: A fully connected 4-layer neural network with 4 inputs and 2 outputs.[Mel]

The training of these neural networks optimizes the internal parameters (in this case, the weights of the connections). This is done by the following steps:

1. Randomly initialize the weights of all nodes
2. Feed a batch of input-output pairs through the network (and produce outputs).
3. Quantify the error between the input and output using an error function (such as mean square error).
4. Backpropagate to compute the gradients of the loss function concerning the input parameters.
5. Update the weights.
6. Repeat steps 2-5 until the network no longer improves (or the stopping criterion is met).

For a network doing classification, a soft-max function is used to pick the option the neural network is most 'confident' about.

2.2 Fault injections (FI)

Fault injection (FI) attacks are techniques where computer components are exposed to conditions outside of their intended operation conditions, to cause behaviour in the system where secrets are exposed or computations are distorted [BECN+06]. FI attacks are often used to target encryption algorithms like Advanced Encryption Standard (that, for example, ensures the secure implementation of the start-up chain on computers). There exist many types of strategies to attack hardware. An example of a fault injection attack changing an image output classification task is shown in Figure 4. This is the type of attack that will be implemented and researched in this thesis.

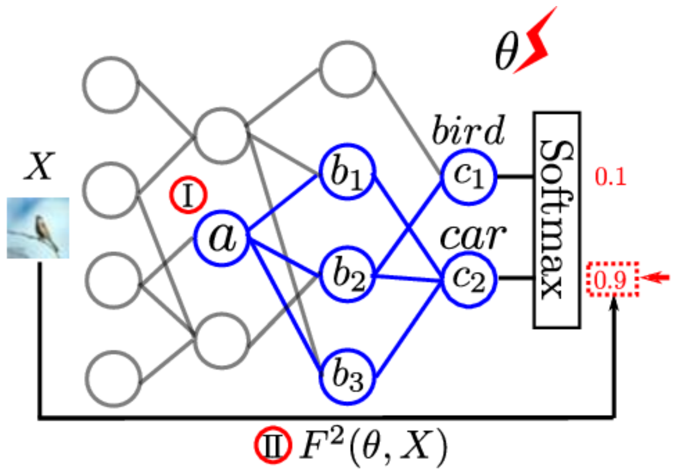

Figure 4: Fault attack in the hidden layer of a neural network causing the image of a bird to be classified as a car. [LWLX17]

2.2.1 Clock glitches

The target CPU runs instructions on a clock, executing an instruction in each clock cycle. The CPU is built according to a (maximum) clock cycle speed and can complete one step of the program's computation at every cycle. If the clock speed is too high, and the CPU receives a new high clock edge too soon, the CPU will not be able to complete its instructions, and undefined behaviour will occur. This is the theory behind clock glitching, where a shortened clock cycle is injected into the target's clock. The goal is to skip an important instruction, which could be used to allow the attacker to gain access to something which should not be possible. This can be achieved by allowing the program counter to advance, a register that stores a pointer to what instruction the CPU is currently executing. When this happens while not allowing the CPU to complete the current instruction, the instruction is effectively skipped altogether and the CPU moves to the next instruction. This can be powerful in the bypassing of authentication, by skipping the comparison of two values and letting the CPU continue in the branch where they are assumed to be equal (when in fact they were not, but the CPU never checked this). Consider a microcontroller with an instruction pipeline like shown in Figure 5.

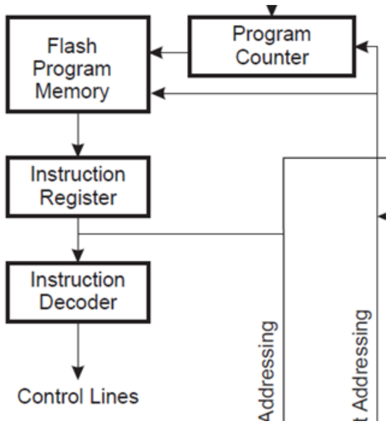

Figure 5: Excerpt of the instruction pipeline taken from the Atmel AVR ATMega328P datasheet[Tec]

Rather than loading each instruction from the FLASH memory and performing an entire execution each cycle, the system described above has a pipeline to speed up this process.[Tec] It decodes an instruction and at the same time retrieves the next one. So when a given instruction x is executed, the CPU also loads operation x+1 into the instruction register. With a modified clock, it can then be possible for a situation to occur where the CPU does not have enough time to perform instruction x but does load the next, essentially skipping instruction x at the start of the next cycle.

A diagram showing the disruption of loading and executing instructions during a clock glitch is shown in Figure 6.

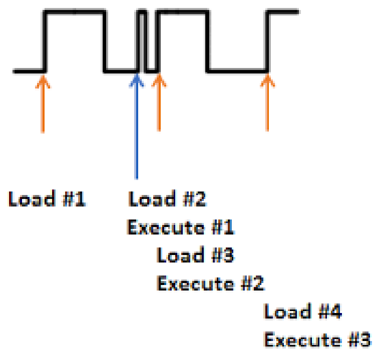

Figure 6: Clock glitch disrupting execution of CPU [Tec]

2.3 Simulating Fault Attacks

Some fault attacks can be simulated, for example, laser-induced bit flips can be simulated by flipping bits in the hidden layer values, to simulate this kind of attack and their effect. This can be used to infer if the bit flip changes the output result (and thus if this weight is significant for the classification). Together with stuck-at-zero faults, where a bit-value is changed to zero, regardless of its original value, these techniques can be used to uncover model weights, as is done in [Ren]. For clock glitches, the instructions that are skipped are implementation-dependent. However given that these layers are long loops of calculating the values of a neuron, which is mostly multiplication followed by activation functions, it is investigated how fault attacks can change these computations. In Section 4.3.3 the assembly of our neural network implementation is analysed. It is hypothesised that successful clock glitches have one of the following effects on the network:

- Biasing the computation towards certain values, by utilizing old values for new computations.
- Reducing the effective number of iterations, by skipping the incrementing of the loop counter.
- Causing the algorithm to consider certain inputs multiple times.
- Skipping the processing of certain inputs entirely.

3 Related Work

This Section provides an overview of relevant work in adversary attacks on neural networks, focusing on attacking embedded systems by using fault injection attacks. Multiple goals exist for these attacks: Recording model assets (such as model parameters, layer sizes and hidden layer activation functions), changing output classifications and denial of service. Multiple techniques are used to reach these goals, such as using lasers to induce bit-flips or clock/voltage glitches to induce precise faults to change results. Stuck at faults also exist (in which case a bit will stay at its current state), which are used by [Ren] to recover model assets. These different fault injection techniques are reviewed in [TG22] and also highlight that neural networks have graceful degradation under faults. Further research into this robustness of neural networks [ABR$^+$19] and [THG17] also show that NNs inherently offer partial fault tolerance, meaning that they are better resistant against faults than other computer algorithms. However, most research underlines that fault injections still can be very effective at attacking neural networks. Real-world bit flip attacks where evaluated in [HFK$^+$19] and [THG17] by using lasers to flip bits in the neural network's hidden layer to affect weights and neuron values. How clock glitch attacks can be used to misclassify images was further evaluated against deep learning accelerators in [LCZL20]. However, a deeper analysis of how clock glitches caused these effects was not explored. Instead, the effects on the results were analysed. Since hardware implementations are costly and need the required targets and tooling, the simulation of fault injection attacks was also investigated by [AB23]. The goal of the simulation of fault attacks is to find possible vulnerabilities and effects that these have, but they don't attempt to recreate the exact internal behaviour that real-world fault attacks would have on the architecture of the CPU, instead using bit flips and instructions skips to simulate faults. Much research has been done into the techniques and effects of adversarial attacks on neural networks and cryptography algorithms, but most simulation techniques do not look at how these faults come into effect in the algorithm and do not get a deeper insight into how these faults distort the computation. A simulation methodology was introduced recently [GS21] that uses hardware simulation to obtain physically accurate software fault evaluation results. However, the drawback is that accurate low-level configurations are needed of the target CPU to perform these simulations. No relevant work was found with the attempt to create more realistic clock glitch attack simulations without using hardware-specific simulation.

4 Methods

In this section, the devices and techniques that were used in the research Section of this thesis are explained. The ChipWhisperer platform, its workings and the definition of glitch parameters are explained in Section 4.1 to 4.1.3. The Neural networking implementation used is discussed in Section 4.2. The experimental setup is shown in Section 4.3.

4.1 ChipWhisperer (CW) platform

The ChipWhisperer ecosystem is an open-source, low-cost solution to perform side-channel analysis and fault attacks on embedded systems. [1] The ChipWhisperer consists of two main parts. The 'capture' side, contains all the analysing and fault-generating hardware, and the 'target' side, has the target CPU to be attacked. This is shown in Figure 7.

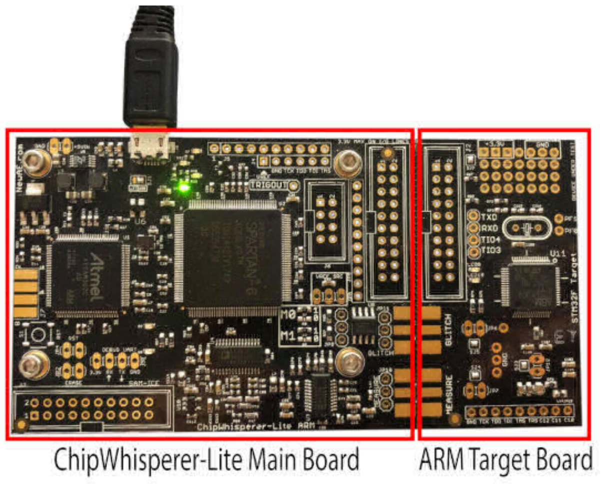

Figure 7: The ChipWhisperer-lite capture (called main board here) vs target side. [RC22]

4.1.1 Clock glitch hardware

The ChipWhisperer has hardware on board to modify the target's clock cycle. First a "glitch stream" is generated as shown in Figure 8. This is done by taking an input clock, which can either be from the capture board or the target CPU and shifting it to create a signal that is synced to the target CPU.

[1] Leiden University provided a ChipWhisperer-Lite for this thesis.

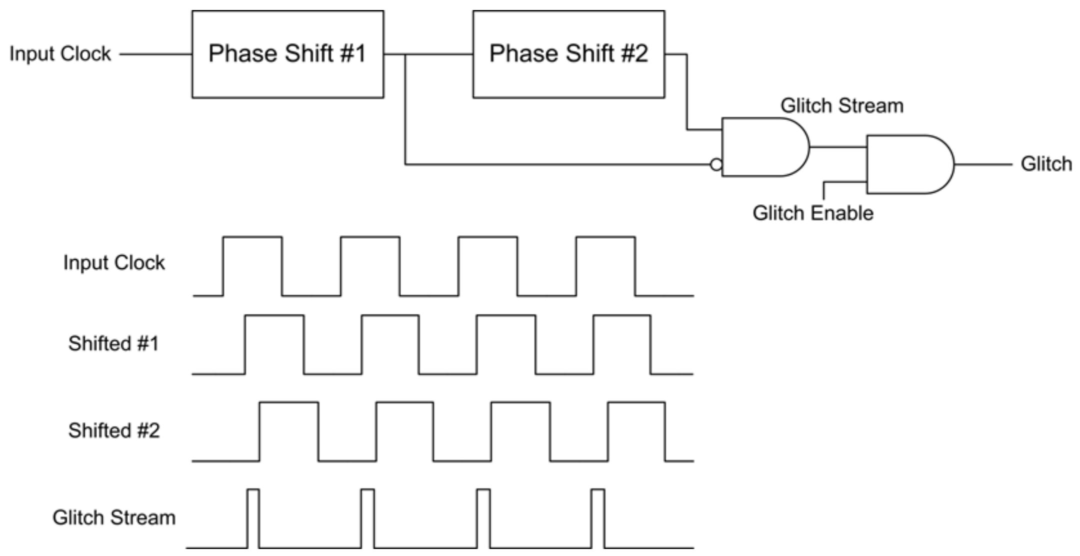

Figure 8: Generation of glitches [Tec]

This glitch stream is then injected into the target (referenced as 'Device Under Test' DUT) in Figure 10.[Tec]

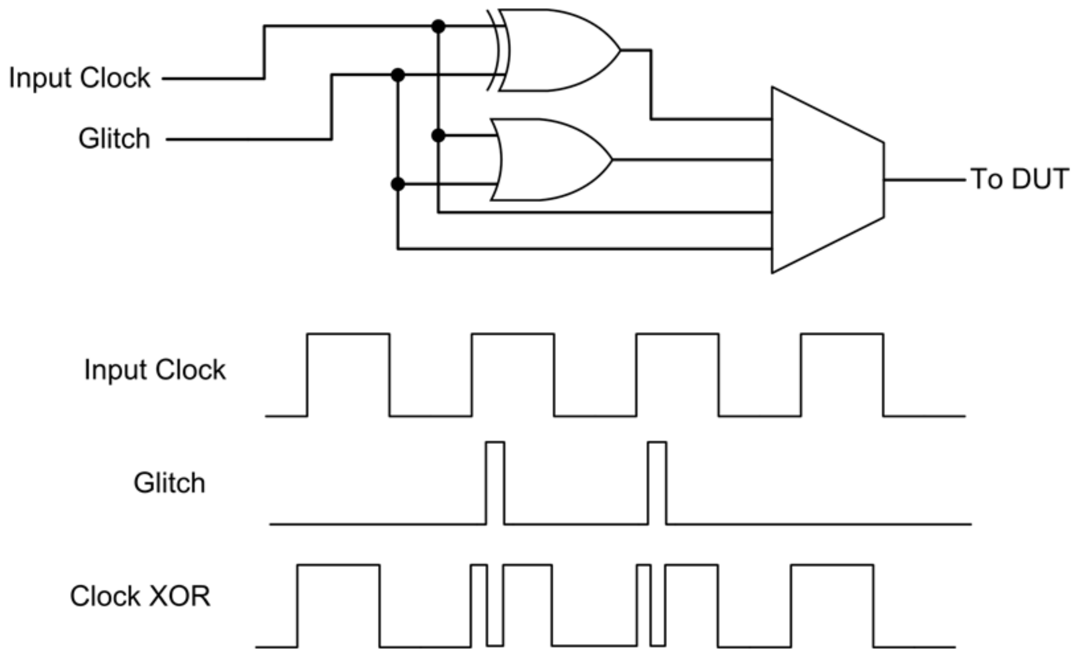

Figure 9: Glitch multiplexed into target's clock. [Tec]

4.1.2 Clock glitches

With the glitch enable line, the ChipWhisperer can precisely set the location and amount of glitches, and by changing the phase shifts of the glitch stream, the 'width' of the glitch can be modulated as well. As the glitch is an XOR operation on the target's clock, the location of a glitch is referred to as the "offset" or "glitch delay" as it describes how long the ChipWhisperer waits after the rising edge of the clock cycle to insert the glitch (bringing the signal low again) and the width describes how long the signal is kept low for. These are the set of glitch parameters used for clock glitching:

- ”width” The width of a single glitch pulse.
- ”offset” The offset from a rising clock edge to a glitch pulse rising edge. Described as the ”glitch delay” in Figure 10.
- ”ext_offset” How long (measured in the number of cycles) does the glitch module wait between a trigger and a glitch.
- ”num_glitches” The number of glitch pulses to generate per trigger.

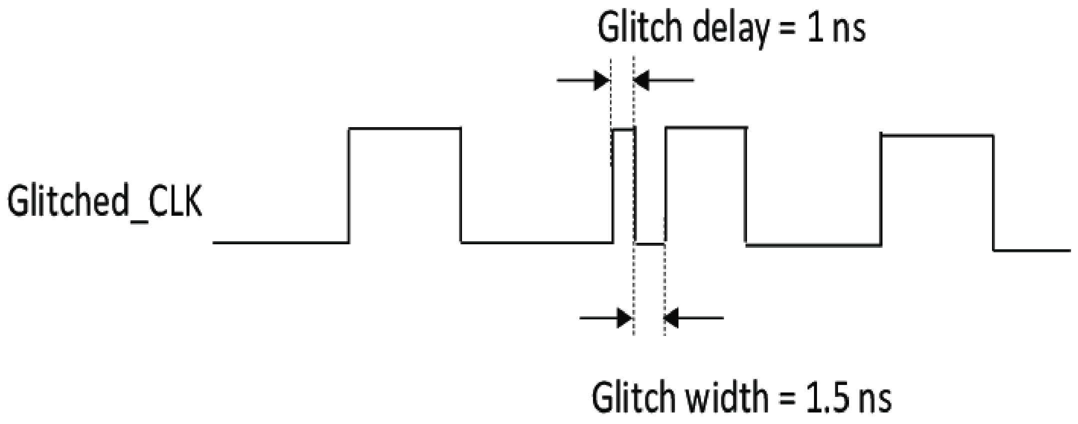

Figure 10: Glitch delay and width parameters. [BN19]

4.1.3 Communication with CW

The C code implemented on the target communicates with the capture board of the ChipWhisperer to time glitches and report the classification results. This was done using the ”Simple-serial” protocol to communicate between both boards. The code featured a ”trigger” which would signal to the capture side of the ChipWhisperer that it can now start clock glitching. The place of this trigger in the code was very consequential for the experiment’s findings as this combined with the ”ext_offset” setting affects the locations where the glitches are injected. The trigger was placed just ahead of the first pass in the neural network, where the input values of the pixel were multiplied by the weights of the hidden layer. At the end of the ’predict’ function, the values of all output nodes were sent to the ChipWhisper capture board using the simple serial protocol.
The trigger mode for the glitch controller was set to ”ext_single”, meaning glitches are only sent out for the first trigger. This was useful since the trigger was used in a loop, and only one neuron was to be attacked instead of the whole layer. This was done because multiple glitches were more likely to crash the target.

4.2 NN implementation

The Neural Network(NN) implemented on the ChipWhisperer is a 2-layer fully connected NN written in ANSI C forked from GitHub [Kro21]. The weights of both layers were trained on a desktop system and hard coded in a custom implementation to be run on the ChipWhisperer. This was done to simplify the code as much as possible and reduce errors during the experimental phase. The forward pass of the neural network implemented is shown in the appendix, Listing 5. The Simple serial protocol version 2_1 was used throughout.
For the input, a simple Python script was taken from [ney17] and adapted to extract one example from the MNIST test set to be hardcoded in the network. It is included in the appendix, Listing 7. The code was compiled using the standard Makefile.inc included in the ChipWhisperer software, and flashed to the target using the Python code shown in Listing 1.

Listing 1: Python code run to flash target

```
SCOPETYPE = 'OPENADC'
PLATFORM = 'CWLITEARM'

%%bash
make PLATFORM=CWLITEARM CRYPTO_TARGET=NONE SS_VER=SS_VER_2_1

import ChipWhisperer as cw
scope = cw.scope()
target_type = cw.targets.SimpleSerial2
target = cw.target(scope, target_type)
scope.default_setup()

cw.program_target(scope, cw.programmers.STM32FProgrammer, "MLP-CWLITEARM.hex")
```

4.3 Experimental setup

The ChipWhisper was connected to a Windows system over a micro-USB cable using the Chip Whisper software version 5.7.0. The hardware firmware was updated to version 0.64 by using the firmware update script included under the ChipWhisperer / jupyter folder from the ChipWhisper folder.

4.3.1 DWT

The ARM cortex-m4 CPU inside the target featured a 'Data Watch Point and Trace Unit' (DWT). "The DWT is an optional debug unit that provides watchpoints, data tracing, and system profiling for the processor." [ARM10] It was needed for this thesis as it features a clock cycle counter which can be used to accurately track the elapsed cycles of the CPU clock for an executed instruction. This can in turn be used to count the number of cycles[Sty17] after a trigger so that the exact location of the glitch can be determined and analysed. The registers used and their addresses as well as the values they were set to is shown in Table 1. These are used to then enable the DWT and enable the clock cycle counter as is shown in Listing 9. Then the clock count can be read from address $0xE0001004$ using GDB.

Name	Adress/value	Description
DWT_CTRL	0xE0001000	Control Register
DWT_CYCCNT	0xE0001004	Cycle Count Register
DWT_DEMCR	0xE000EDFC	Debug Exception and Monitor Control Register
DWT_CYCCNTENA_BIT	1UL<<0	DWT enable bit
TRCENA_BIT	1UL<<24	Trace enable bit

Table 1: The registers/enable bits used and their address/value for the ARM Cortex-M4 [ARM10]

4.3.2 GDB

GDB is the GNU Project Debugger. "It allows you to see what is going on 'inside' another program while it executes – or what another program was doing when it crashed" [Fou]. GDB was utilised to analyze the program during execution and to simulate glitches by skipping instructions. GDB-multiarch was used because normal GDB does not support the ARM instruction set used for this target. Source code for some commands from the ChipWhisperer HAL (hardware abstraction layer) had to be pulled in so GDB-multiarch got to view the instructions inside those functions as well. This was needed for the trigger functions since the exact moment the trigger pin went high was needed to calculate glitch injection positions. The __mulsf3 and __aebdi functions used in the calculation of the hidden layer values also needed to be analysed to simulate faults inside these functions. The ieee754-sf.S file was retrieved from the libgencc source code version 10.3.0 and pulled into the source directory so that GDB-multiarch could disassemble these functions.
Commands used to analyze and emulate faults where:

- **step**, to advance the CPU one instruction.
- **jump +x**, where x is the number to advance the PC register. The CPU uses 2-byte and 1-byte instructions, meaning +2 and +1 skip that instruction respectively.

- **print (u*0xE0001004)** to print the current value of the DWT CYNNT register.
- **print y** to print the value of the output layer
- **layout src**, to view the current position regarding the C source code.
- **breakpoint trigger_high()** to set a breakpoint at the trigger function or **breakpoint mlpCW.c:xxx** where *xxx* denotes a line number according to where a breakpoint is to be set.

4.3.3 Analysing ASM

In Listing 2 GDB-multiarch was used to disassemble the prediction function. This ASM was compiled using the optimization option '-s', which denotes optimizing for size. This was standard in the makefile included in the ChipWhisperer project. The triggers can be seen on +40 and +132. In the appendix, Listing 4, the respective C source code for this assembly can be seen.

Listing 2: multiplication and activation function of the hidden layer

```
0x080007f4 <+40>:   bl      0x80009d4 <trigger_high>
0x080007f8 <+44>:   mov     r5, sp
0x080007fa <+46>:   mov.w   r9, #1065353216  ; 0x3f800000
0x080007fe <+50>:   ldr.w   r4, [r5], #4
0x08000802 <+54>:   ldr.w   r11, [pc, #104]  ; 0x800086c <predict+160>
0x08000806 <+58>:   mov     r10, r8
0x08000808 <+60>:   movs    r7, #0
0x0800080a <+62>:   ldr.w   r1, [r11], #4
0x0800080e <+66>:   ldr.w   r0, [r10], #4
0x08000812 <+70>:   bl      0x80003e4 <__mulsf3>
0x08000816 <+74>:   mov     r1, r0
0x08000818 <+76>:   mov     r0, r4
0x0800081a <+78>:   bl      0x80001d4 <__aeabi_fadd>
0x0800081e <+82>:   adds    r7, #1
0x08000820 <+84>:   cmp.w   r7, #784         ; 0x310
0x08000824 <+88>:   mov     r4, r0
0x08000826 <+90>:   bne.n   0x800080a <predict+62>
0x08000828 <+92>:   add.w   r0, r0, #2147483648      ; 0x80000000
0x0800082c <+96>:   bl      0x80014fc <expf>
0x08000830 <+100>:  mov     r1, r9
0x08000832 <+102>:  bl      0x80001d4 <__aeabi_fadd>
0x08000836 <+106>:  mov     r1, r0
0x08000838 <+108>:  mov     r0, r9
0x0800083a <+110>:  bl      0x800054c <__divsf3>
0x0800083e <+114>:  add.w   r6, r6, #784     ; 0x310
0x08000842 <+118>:  cmp.w   r6, #7840        ; 0x1ea0
0x08000846 <+122>:  str.w   r0, [r5, #-4]
0x0800084a <+126>:  add.w   r8, r8, #3136    ; 0xc40
0x0800084e <+130>:  bne.n   0x80007fe <predict+50>
0x08000850 <+132>:  bl      0x80009e2 <trigger_low>
```

4.3.4 Injecting Clock Glitches

The Fault attacks were carried out using the Python code shown in the appendix, Listing 6. Scope.capture() is used to set the CW capture board into a state of waiting for the trigger. Before the glitching, a baseline result was generated of a test image (referred to as prev_value).
The following steps were in the glitching process:

1. Update glitch parameters (width, length, offset)

   Timeout: If the trigger was still high from the last glitch, it meant our target crashed. Record it as a crash and reboot the target.

2. Then an 'a' is written over a simple serial to start the predict function and wait for the trigger with scope.capture()

   Timeout: If the target does not send a trigger, record it as a timeout.

3. Inject glitches after trigger

   Invalid return: If the target responds with an invalid value, record it as a crash and record the parameters.

4. On receiving the response, compare it to prev_values.

   Different: A successful glitch. Record the width, scope and offset.

   The same: Record it as normal and continue.

In Figure 11 an example plot of how different clock glitch parameters generate successful or unsuccessful clock glitch attacks is shown.

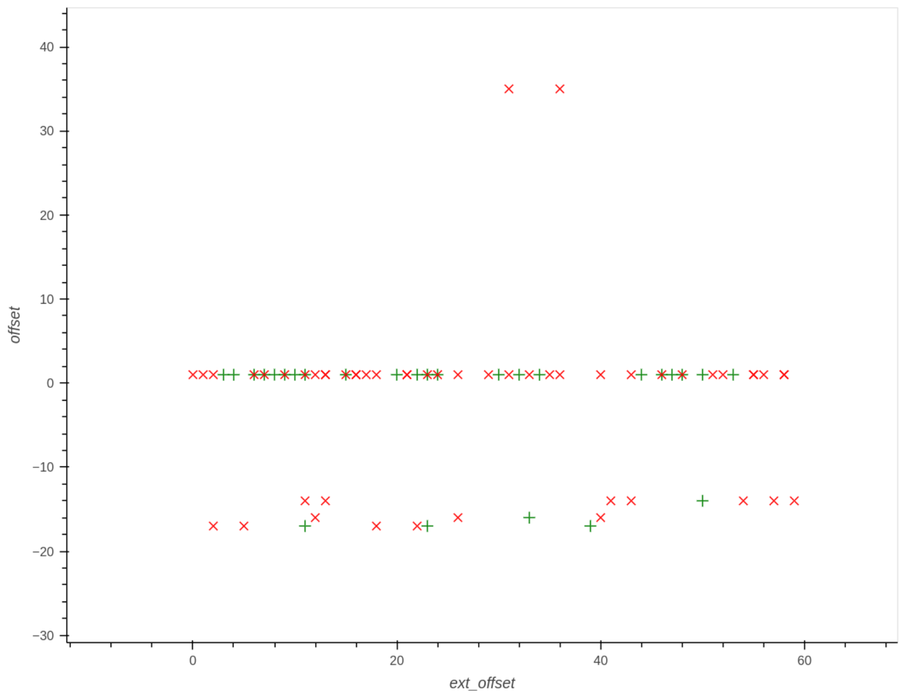

Figure 11: Glitchplot of the parameters for clock glitch attacks, Width is stacked. A reset is shown with a red cross, and a successful attack changing the neural network's output is shown with a green plus. The red/green 'stars' that can be seen are sets of parameters where some width settings failed and some succeeded. Only 2D plots were possible, so widths had to be stacked.

4.3.5 Debugging target

The exact location where the glitch was injected in the code must be found to analyse the attacks. After trying multiple failing techniques further described in Section 5.1 and 6 it was decided to read values from the ARM target running the NN directly using a debugger running on the ChipWhisperer directly. This means that GDB can be used directly on the program that is being run on the target chip of the ChipWhisperer. The ChipWhisperer target was debugged by utilising the Multi-Protocol Synchronous Serial Engine (MPSSE) interface of the board [New]. With some additional connections between the target and the mainboard, a remote GDB session can be used to debug the target chip. This was needed to determine the value of the cycles passed after the trigger signal to analyse what instruction was skipped or corrupted by the fault attack. The following extra connections were made between the non-standard JTAG connector on the CW-lite side [Inc18a] and the JP13 connector[Inc18b] at the bottom of the target using header pins and DuPont cables, as shown in Figure 12. [2]

- Connected SCK to TCLK/SWDCLK
- Connected PDID to TMS/SWDIO
- Connected GND on the ChipWhisperer to GND on the target
- Connected MISO to TDO
- Connected MOSI to TDI

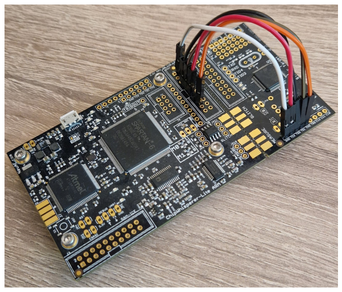
Figure 12: The target and capture board are connected using female cables and soldered headers

After this, openOCD was used to connect to the chip whisper in debugging mode using the following command .[3]

```
./run_openocd.sh lite swd -- -f target/stm32f3x.cfg
```

This command first puts the device in MPSSE mode, and then re-enumerates and reconnects the device, allowing a remote GDB session to attach using the following command from a different terminal:[4]

```
joris@JorisPC:~$ gdb-multiarch
(gdb) target extended-remote localhost:3333
```

Now that GDB was connected to the target using a remote session a breakpoint was set at the trigger high function and returned the DWT CYCCNT at the register at memory address 0xE0001004 after stepping some instructions, as can be seen in Listing 3.

[2]See Section 5.1 for potential recommendations on the debug connection protocol.

[3]on Linux, cw_openocd.cfg had to be modified with 'ftdi_command' instead of 'ftdi command'

[4]Please note that when using the virtual machine implementation of ChipWhipserer, localhost will not work. The IP address assigned to the networking adapter used by the virtual machine solution has to be used.

Listing 3: Using GDB to find skipped instructions(truncated long file paths and function parameters for readability)

```
(gdb) monitor reset halt
(gdb) load MLP-CWLITEARM.elf
(gdb) break trigger_high()
(gdb) continue
continuing.
(gdb) Breakpoint 15, trigger_high () at ../...
109         HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12, SET);
(gdb) print/u *((volatile uint32_t*)0xE0001004)
$1 = 2
(gdb) step
HAL_GPIO_WritePin (...) at ../...
801         if(PinState != GPIO_PIN_RESET)
(gdb) print/u *((volatile uint32_t*)0xE0001004)
$2 = 7
(gdb) step
803           GPIOx->BSRR = (uint32_t)GPIO_Pin;
(gdb) print/u *((volatile uint32_t*)0xE0001004)
$3 = 8
```

5 Results

In Section 5.1, the results of the proposed research setup for analysing clock glitches and verifying results is shown. In Section 5.2 shows the results of fault attacks on the neural network and how they affect output classifications. Finally, in Section 5.3, the results are discussed on how these glitches affected the CPU on the instruction level.

5.1 Research setup

The extra connections made on the ChipWhisperer as described in Section 4.3.5 were tested first using male-to-male DuPont cables without soldering. While this did not result in a usable connection for a GDB session without crashing due to communication errors, it did show that the implementation would work over the JTAG protocol. The SWD protocol, supported by the target chip, uses fewer pins but does not work this way. However, when all pins were connected in a reliable way using soldered headers the SWD protocol did work and has notable improvements over the JTAG protocol, as well as using 3 pins less. Therefore, the SWD protocol should be used instead of the JTAG protocol and extra pins as described in [New] do not have to be soldered unless needed.

Ultimately, the thesis did not succeed in analysing the exact effect of clock cycle glitches on the program execution. This was due to problems with finding the exact location where the glitch was injected in the code by the ChipWhisperer glitch controller, and the fault model assumed for clock glitches. This was concluded after skipping instructions inside the remote GDB session following the offset parameters from the real-world results of clock glitch attacks (further discussed in 5.3) did not match up results. Multiple techniques to try to tackle this problem were tried:

- Emulating the Target ARM cortex M4 in RENODE, A CPU simulator. While emulating the target was successful, emulating the DWT was not, and RENODE did not feature any single clock-stepping options, meaning that the glitch location could not be determined. Ultimately emulators like RENODE and QEMU are not clock cycle accurate and are not the right tool for research like this according to [Oli13] and [Zie21].

- Calculating the clock cycle count by hand by the use of the technical ARM Cortex-M4 manual [ARM10]. Due to possible pipeline refills, which can range from 1 to 3 cycles depending on instruction alignment, width, and early address speculation, the exact cycle count may vary. This variability resulted in many possible injection sites for glitches that were a further (amount of cycles) away from the trigger. Multiple possibilities were evaluated but none resulted in the expected results. The further a glitch was injected, the larger the range of possible glitch injection sites was.

- It was tried to find a reliable set of parameters for a clock glitch attack and attach the debugger to investigate register values during the attack. Unfortunately, the MPSSE connection for debugging cannot be used during a connection needed for glitching, so a debug session cannot be run during glitching. This prohibited analysis the analysis of CPU registers during a glitch. The values of the hidden layer were however further investigated in Section 5.3.

To create the best possible insight into successful fault attacks, GDB was used with a remote session on the target hardware over an MPSSE connection to investigate the effect of skipping instructions. During the setup of debugging on the ChipWhisperer target, the stm32f3x.cfg configuration file included with OpenOCD ver. 0.11.0+dev did not seem to work on Linux but did work on Windows. Due to possible problems with the LUSB driver on Windows[New], the need was created to debug on Linux, so the script was changed to work on Linux, with the changes in the appendix, Listing 5.

This however featured some limitations in the type of glitches that could be investigated:

- Some instructions take more than 1 cycle. To advance to a clock cycle count x, while currently positioned at cycle y (given that $x > y$) a single instruction step can advance us to cycle count z, where $z > x$. This means the glitch started halfway through an instruction, which cannot be simulated using the proposed technique.
- Jumping or setting the PC register to a new instruction means that only glitches that end up at the end of an instruction can be simulated.

5.2 Fault Attacks

A random digit was extracted from the MNIST test set to import as input image. digit used is a zero and is shown in Figure 13. To target the hidden layer, the following trigger setup was used:

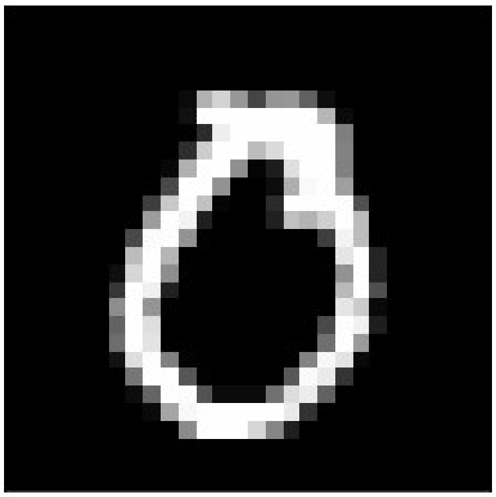

Figure 13: Graphical representation of MNIST zero digit[unk17].

```
// h := w'
for (int j = 0; j < H; j++) {
    for (int i = 0; i < X; i++) {
        trigger_high();
        h[j] += w[j * X + i] * input[i];
    }
    h[j] = 1.0f / (1.0f + expf(-h[j]));
}
trigger_low();
```

The output that the neural network provided on the output layer while not being under attack can be seen in table 2 and 3 as the 'baseline' row. A sweep of clock glitches was run with the following parameters to discover what kind of results were possible with different sets of glitch parameters:

```
gc.set_range("width", -35, 5)
gc.set_range("offset", -35, 35)
gc.set_range("ext_offset", 0, 60)
gc.set_global_step(1.0)
```

The results of this sweep can be seen in Figure 14 .

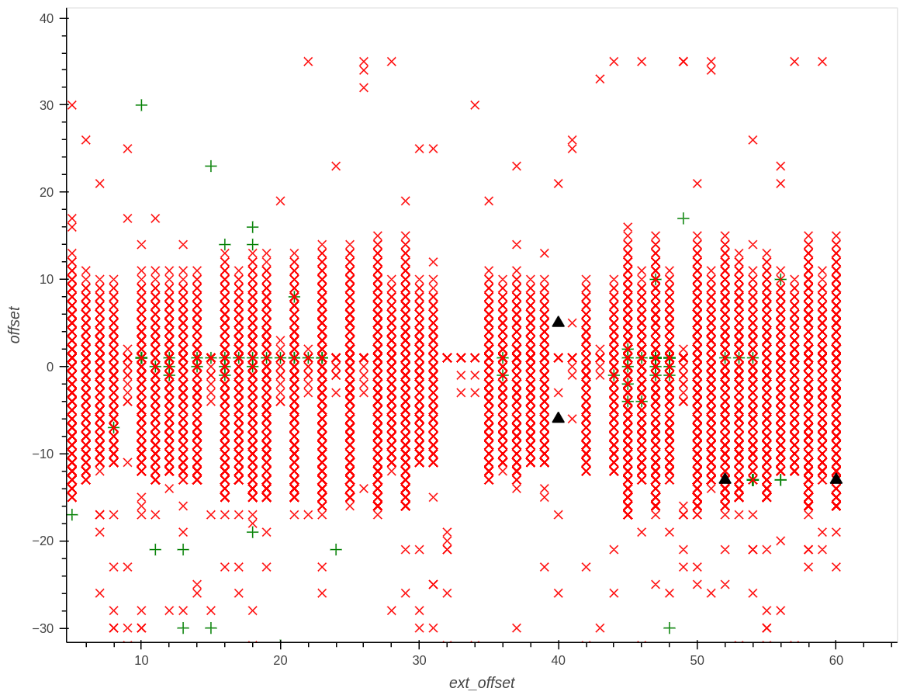

Figure 14: Glitchplot of the parameters for clock glitch attacks, widths are stacked. A reset is shown with a red cross, and a successful attack changing the neural network's output is shown with a green plus. The black triangle indicates a time out of the target.

As can be seen, this resulted in numerous glitches that ended up changing the network output. Most glitches do not end up changing the network's final output classification, they do not have a large enough effect on the output layer for the maximum value to not be the zero node (which is the correct class of the image that was input). The output of the glitch loop in these cases has a small difference on one of the output nodes, as can be seen in Table 2.

Width	Offset	Ext_Offset	Node 0	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8	Node 9
Baseline			***9.83***	*-9.47*	*2.00*	*0.87*	*-3.68*	*3.18*	*0.74*	*-1.08*	*-0.67*	*-2.16*
-25.0	1.17	12	**9.221**	-11.954	2.983	-0.839	-0.061	1.710	1.433	2.375	-5.125	-0.129
-28.12	-21.09	39	**9.52**	-10.71	2.49	0.01	-1.87	2.44	1.09	0.65	-2.90	-1.14

Table 2: Glitch Parameters and Node Outputs compared to baseline. Minor changes to the output layer and final classification output were not affected.

A portion of the glitches caused the entire output layer to stay at zero, and a small amount of glitches changed the output layer values so much so that the final output classification changed to another class, shown in Table 3.

Width	Offset	Ext_Offset	Node 0	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8	Node 9
Baseline			***9.83***	*-9.47*	*2.00*	*0.87*	*-3.68*	*3.18*	*0.74*	*-1.08*	*-0.67*	*-2.16*
-21.09	1.17	7	**0.0**	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
-28.12	-21.09	60	9.83	-9.47	2.00	0.87	-3.68	**131075.19**	0.74	-1.08	-0.67	-2.16

Table 3: Glitch Parameters and node Outputs compared to baseline. Empty output and major output layer values change changing the final classification output to digit 5.

Most glitches could be reproduced using the same set of glitch parameters about 5 to 10% of the time. This was not an issue since the main focus was to take a closer look at the effect of these glitches on the CPU, and less so on the reproducibility. As explained in Section 5.1, it was also attempted to execute glitches while connected to the debugging interface. For this, a reproducible set of glitch parameters was needed. A small range was selected around a glitch found in the initial sweep and the step size was set from 1.0 to 0.1, to achieve more accuracy on where the glitch exactly was reproducible the best. The 'tries' setting was set to 5, and glitches that could be repeated 3-4 times were deemed reproducible enough to be used for this approach. These were ultimately not needed as the approach was not possible, as was explained in Section 5.1.

5.3 Effects on CPU

GDB was used to analyze the program during execution and to simulate glitches by skipping instructions, under the assumption that successful clock glitches skip whole instructions without interacting with another part of the program and/or execution. This was done by setting a breakpoint at the trigger-high function, stepping to the exact line where the UART pin is set to high. This is presumed to be from where the glitch controller starts counting down the ext_offset. by reading from the DWR CYNNT register the current clock cycle count can be found (referred to as 'the start point'). After stepping an instruction it can be read again, to check the difference from the start point. By advancing instructions until the difference between the current count and the start point is the same as the ext_offset of a successful glitch the injection site of the glitch can be determined. By using the disas command the following instructions can be seen. After looking up the cycle count of the coming instructions a deduction can be made for how many instructions need to be skipped to replicate the glitch. Either Jump can be used with the byte size * the number of instructions that need to be skipped, or the PC register can be advanced to an instruction to where the glitch ended. The assignment of a variable or value of a computation can then be read by printing them.
Due to the issues further described in Sections 5.1 and 6 it was not possible to reconstruct any successful clock glitch attacks that changed the NN output in a GDB session, and it could therefore not be assumed any analysis was under the right conditions to be accurate to the real world working of these glitches. It is presumed that the NN output could not be replicated because of the incorrect assumption about the fault model of clock glitches being that they cause completely isolated instructions skips. Following this suspicion, the hidden layer outputs were investigated by returning them directly instead of the output layer during glitching, using a changed hidden layer computation shown in the appendix, Listing 8. Some code was added to only trigger on input values that were not equal to zero, as the first few values of the image are zero, and we also want to get glitches where the image value where not zero without using very long delays (ext_offsets). It seemed that the value of the hidden layer values was affected in many different ways under different glitches. The following results were seen in the hidden layer values, with an analysis of the __aeabi_fadd function and what instructions were possibly skipped to cause these results:

- Sign flipping

  Skipping sign handling:

  ```
  0x080004fc <+64>: negne r0, r0
  or
  0x0800050c <+80>: negne r1, r1
  ```

  Skipping either of these could cause sign flips, explaining negative values where positive ones were expected or vice versa.

- Inf / -inf or Nan values

  Skipping overflow checks:
  ```
  0x0800054c <+144>: bcs.n 0x80005f2 <__aeabi_fadd+310>
  ```

  Skipping this branch could prevent proper handling of overflow, potentially explaining the appearance of infinity or very large values.

  Skipping special case handling:
  ```
  0x080005fc <+320>: mvns.w r2, r2, asr #24
  0x08000600 <+324>: itet ne
  0x08000602 <+326>: movne r0, r1
  0x08000604 <+328>: mvnseq.w r3, r3, asr #24
  0x08000608 <+332>: movne r1, r0
  ```

  Skipping instructions here could cause incorrect handling of special cases like NaN or infinity.

- Off py powers of 2

  Skipping the rsbs instruction or the subsequent itttt:
  ```
  0x080004d6 <+26>: rsbs r3, r2, r3, lsr #24
  0x080004da <+30>: itttt gt
  ```

  This could cause incorrect exponent alignment, leading to values that are off by powers of 2.

- Extremely large or small values

  Normalization:
  ```
  0x08000540 <+132>: lsrs r0, r0, #1
  0x08000542 <+134>: mov.w r1, r1, rrx
  0x08000546 <+138>: add.w r2, r2, #1
  ```

  Skipping these could lead to unnormalized results, explaining very large or small values.

- All zero's

  Skipping loop control:
  ```
  0x08000d4a <+122>: cmp.w r10, #7840 ; 0x1ea0
  ```

  If operations deciding of the value of R10 are skipped this could cause the entire outer loop to be skipped and all values to stay at zero.

While offsets were found that could cause glitches in these locations, none did exactly match. This is either caused by limitations explained in 6, or an incorrect assumption that clock glitches only cause complete instruction skips and do not affect other values or cause half-finished results/ other undefined behaviour.

6 Discussion

The ChipWhisperer used in this thesis featured an STM32F303RD target, with an ARM cortex M4 processor. This processor has 64K RAM, which poses some limitations for the NN implementation. Since the weights for the network need to be in RAM, this limited the size of the hidden layer. This lowered the achieved accuracy for the network on the test set, but given that this does not impact research into the analysis of fault attacks in neural networks, other techniques to mitigate the strained hidden layer size were not investigated (weight streamlining and/or half precision). It should however be noted that a smaller network can change how fault attacks affect the output as the network is more concentrated (and therefore less resilient against faults, as it is less distributed.)

There are several possible problems with the proposed research setup that ultimately caused problems in finding the exact locations and reproducing clock glitch attacks.

- According to 'Jean-Pierre', one of the employees at the ChipWhisperer project "There is a small and fixed latency from when the mainboard gets the trigger to when it issues the glitch (in addition to scope.glitch.ext_offset.)" [via the official discord forum.] It was considered if lowering the ChipWhisperer capture board clock could lower this latency, but JP also stated that the latency is some clock cycles of the glitch modules' source clock, which would normally be the capture board system clock. This means that lowering this would also result in the same latency. It was tried to switch to using the ChipWhisperer's target clock as the source clock for the glitch module by changing the 'clk_src' variable in the glitch controller from "clkgen" to "target". Using the same set of parameters no successful glitches could be found.
- As per JP, The target's execution pipeline can be disturbed by the glitch. This can also add some cycles to the CPU execution, that would not be present in the GDB session, and would therefore not match the simulation.

Because of the potential latency and timing issues mentioned above, it was tried to find a set of glitch parameters close to the trigger. Manually calculate the glitch's possible locations, and try offsets of this possible location. Due to the pipeline refills, this resulted in multiple possible instructions, and all were evaluated, but none created the NN output that was also returned at the time of glitching.

- The fault model used was under the assumption that clock glitches would affect the CPU by skipping instructions, and voltage glitches would cause undefined behaviour in the processor. But it might be so that clock glitches don't always skip instructions but can also cause undefined behaviour, such as finishing an instruction halfway. This was hard to verify since a debug session could not be run during glitching to verify register results.
- An attempt was made to return register values at the correct clock cycle using the DWT without using a debugger, but by utilising the simple serial protocol during glitching. This however did not seem practical since the code to return the PC Register could only be placed at a certain position in the c code which meant that it would be almost impossible to position the code at the correct clock cycle. (the __mulsf3 function takes 20-30 cycles, and there are two of them in the $h[j] += w[j*X+i]*input[i]$; line. This function is probably exactly where the glitch takes place, and the C code to return the PC register of the instructions

that the glitch would skip could only be placed above or below this, not inside this function, therefore making it useless for the analysis.

6.1 Answers to research questions

The research carried out in this thesis leads to the following answers to the research questions:

How are successful clock glitches carried out on an embedded neural network?

A successful research method to carry out successful clock glitches on a neural network is proposed, by using the ChipWhipserer's glitch controller and modifying the neural network implementation with the simple serial protocol and trigger commands.

How do successful clock glitches affect instructions in an embedded neural network?

Due to the problems discussed in Sections 5.1 and 6, what exact instructions were affected could not be concluded and verified by reproducing them over a GDB session. Judging from the classification results from successful clock glitches, assumptions can be made about which instructions were affected in some cases. None of the offsets found by actual glitches could be used to reproduce these results, however.

7 Conclusions and Future research

7.1 Future research

For future research, a closer look can be taken at how clock glitches affect a CPU's execution of instructions at the architecture level, to investigate how a better fault model can be proposed for simulating realistic clock glitches. Research into using a ChipWhisperer in combination with actual embedded devices running neural networks or breaking cryptographic implementations could also be very interesting and pose a challenging research subject. Examples of such devices are cameras with face recognition or hardware wallets/authentication devices.

7.2 Conclusions

This thesis has explored the application of clock glitch attacks on neural networks implemented on embedded hardware, focusing on an MLP trained for MNIST digit classification. While it was successfully demonstrated that clock glitches can indeed affect the output of the neural network, precise analysis of how these glitches affect specific instructions proved challenging due to various factors, including potential latency in the ChipWhisperer platform and complexities in the CPU's execution pipeline.

Research highlights the vulnerability of embedded neural networks to fault attacks, demonstrating that carefully timed clock glitches can change classification results. However, the study also highlights the complexity of analyzing such attacks at the instruction level, particularly when dealing with real hardware rather than simulations.

The difficulties encountered in reproducing successful glitches in debugging sessions suggest that initial assumptions about clock glitches primarily causing instruction skips may be oversimplified. observations indicate that glitches might also yield undefined behaviour in the processor, complicating the fault model.

Despite these challenges, this work provides a foundation for future research in this area. It demonstrates a practical setup for conducting clock glitch attacks on embedded neural networks and highlights areas where further investigation is needed. Future work should focus on developing more accurate fault models for clock glitches, exploring their effects at the architectural level of the CPU, and investigating the resilience of different neural network architectures and activation functions to such attacks.

In conclusion, while this thesis has not fully resolved all questions regarding the precise effect of clock glitch attacks on embedded neural networks, it has made steps toward the usability and deeper understanding of these attacks and how to gain insight in them.

References

[AB23] Asmita Adhikary and Ileana Buhan. Sok: Assisted fault simulation. In Jianying Zhou, Lejla Batina, Zengpeng Li, Jingqiang Lin, Eleonora Losiouk, Suryadipta Majumdar, Daisuke Mashima, Weizhi Meng, Stjepan Picek, Mohammad Ashiqur Rahman, Jun Shao, Masaki Shimaoka, Ezekiel Soremekun, Chunhua Su, Je Sen Teh, Aleksei Udovenko, Cong Wang, Leo Zhang, and Yury Zhauniarovich, editors, *Applied Cryptography and Network Security Workshops*, pages 178–195, Cham, 2023. Springer Nature Switzerland.

[ABR$^+$19] Manaar Alam, Arnab Bag, Debapriya Basu Roy, Dirmanto Jap, Jakub Breier, Shivam Bhasin, and Debdeep Mukhopadhyay. Enhancing fault tolerance of neural networks for security-critical applications, 2019.

[ARM10] ARM. Cortex-M4 Technical Reference Manual (Revision R0P0). Technical Report ARM DDI 0439B (ID030210), ARM, 3 2010.

[BECN$^+$06] 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.

[BN19] Talal Bonny and Qassim Nasir. Clock glitch fault injection attack on an fpga-based non-autonomous chaotic oscillator. *Nonlinear Dynamics*, 96, 05 2019.

[Fou] Free Software Foundation. GDB: The GNU Project Debugger.

[GS21] Jacob Grycel and Patrick Schaumont. Simplifi: Hardware simulation of embedded software fault attacks. *Cryptography*, 5(2), 2021.

[HFK$^+$19] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 497–514, Santa Clara, CA, August 2019. USENIX Association.

[Inc18a] NewAE Technology Inc. CW1173 ChipWhisperer-Lite - NewAE Hardware Product Documentation, 2018.

[Inc18b] NewAE Technology Inc. CW303 Arm Target - NewAE Hardware Product Documentation, 2018.

[Kro21] Krocki. GitHub - krocki/MLP-C: Multi-layer perceptron in C, 1 2021.

[LCZL20] Wenye Liu, Chip-Hong Chang, Fan Zhang, and Xiaoxuan Lou. Imperceptible misclassification attack on deep learning accelerator by glitch injection. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[LWLX17] Yannan Liu, Lingxiao Wei, Bo Luo, and Q. Xu. Fault injection attack on deep neural network. *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 131–138, 2017.

[McC21] Nick McCullum. Deep Learning Neural networks explained in Plain English, 4 2021.

[Mel] Kathrin Melcher. A Friendly Introduction to [Deep] Neural Networks — KNIME.

[New] NewAE. Debugging with ChipWhisperer — ChipWhisperer 5.7.0 documentation.

[ney17] tyler neylon. A function to load numpy arrays from the MNIST data files., 2017.

[Oli13] Bryan Olivier. Can you check performance of a program running with Qemu Simulator?, 7 2013.

[RC22] Anca Rădulescu and Marios Choudary. Side-channel attacks on masked bitsliced implementations of aes. *Cryptography*, 6:31, 06 2022.

[Ren] Shuaizhen Ren. Exploring fault injection attacks against embed- ded neural networks.

[Sty17] Erich Styger. Cycle Counting on ARM Cortex-M with DWT, 1 2017.

[Tec] NewAE Technology. Introduction to Glitch Attacks - ChipWhisperer Wiki.

[TG22] Shahin Tajik and Fatemeh Ganji. Artificial neural networks and fault injection attacks. In *Security and Artificial Intelligence*, Lecture Notes in Computer Science, pages 72–84. Springer International Publishing, Cham, 2022.

[THG17] Cesar Torres-Huitzil and Bernard Girau. Fault and error tolerance in neural networks: A review. *IEEE Access*, 5:17322–17341, 2017.

[unk17] 3.3. The MNIST Dataset — conx 3.7.9 documentation, 2017.

[WXD$^+$21] Dongcheng Wang, Yanghuan Xu, Bowei Duan, Yongmei Wang, Mingming Song, Huaxin Yu, and Hongmin Liu. Intelligent recognition model of hot rolling strip edge defects based on deep learning. *Metals*, 11(2), 2021.

[Zie21] Piotr Zierhoffer. How to benchmark some algorithms for Cortex-M architecture, 2 2021.

Appendix

Listing 4: C implementation of forward pass (truncated variables)

```
/* input size */
#define X 784
/* hidden size */
#define H 10
/* output size */
#define Y 10

float w[X * H] = {...};
float v[H * Y] = {...};
float input[X] = {...};

//header for use with simpleserial commands.
int8_t predict(uint8_t cmd, uint8_t subc, uint8_t len, uint8_t *in) {
    float h[H] = {0};
    float y[Y] = {0};

    // h := w'
    trigger_high(); //start glitching
    for (int j = 0; j < H; j++) {
        for (int i = 0; i < X; i++) {
            h[j] += w[j * X + i] * input[i];
        }
        h[j] = 1.0f / (1.0f + expf(-h[j]));
    }
    trigger_low(); //signal no crash

    // y := vh
    for (int k = 0; k < Y; k++) {
        for (int j = 0; j < H; j++) {
            y[k] += v[k * H + j] * h[j];
        }
    }

    //communicate output layer to CW
    simpleserial_put('r', Y * sizeof(float), (uint8_t*)y);

    return 0x00;
}

int main(void) {
    platform_init();
    init_uart();
    trigger_setup();
    simpleserial_init();
    //add a command to run predict function, we pass no arguments.
    simpleserial_addcmd('a', 0, predict);
    while(1):
        //wait for commands
        simpleserial_get();

    return 0;

}
```

Listing 5: Changes to TPIU creation in OpenOCD stm32f3x.cfg file for linux implementation

```
tpiu create $_CHIPNAME.tpiu -dap $_CHIPNAME.dap -ap-num 0 -baseaddr 0xE0040000

lappend _telnet_autocomplete_skip _proc_pre_enable_$_CHIPNAME.tpiu
proc _proc_pre_enable_$_CHIPNAME.tpiu {_targetname} {
        targets $_targetname

        # Set TRACE_IOEN; TRACE_MODE is set to async; when using sync
        # change this value accordingly to configuretrace pins
        # assignment
        mmw 0xe0042004 0x00000020 0
}

$_CHIPNAME.tpiu configure-event pre-enable
    "_proc_pre_enable_$_CHIPNAME.tpiu $_TARGETNAME"
```

This original configuration was changed to:

```
# TPIU configuration
proc stm32f3x_tpiu_init {} {
    # Set TRACE_IOEN; TRACE_MODE is set to async; when using sync
    # change this value accordingly to configuretrace pins assignment
    mmw 0xe0042004 0x00000020 0
}

# Configure TPIU
# Adjust TRACECLKIN and trace port frequency as needed
set _TRACECLKIN_FREQ 72000000
set _TRACE_FREQ 4000000

tpiu config external uart off $_TRACECLKIN_FREQ $_TRACE_FREQ

# Add TPIU initialization to reset-init event
$_TARGETNAME configure -event reset-init {
    stm32f3x_default_reset_init
    stm32f3x_tpiu_init
}
```

Listing 6: Clockglitching loop with prev_values truncated for readability

```
broken = False
for glitch_setting in gc.glitch_values():
 #set params
 scope.glitch.offset = glitch_setting[1]
 scope.glitch.width = glitch_setting[0]
 scope.glitch.ext_offset = glitch_setting[2]

 if scope.adc.state:
  # can detect crash here (fast) before timing out (slow)
  gc.add("trigger_still_high")

  reset_target(scope)
  target.flush()

 scope.arm()

 # Do glitch loop
 target.simpleserial_write('a', bytearray([]))
 ret = scope.capture()

 loff = scope.glitch.offset
 lwid = scope.glitch.width

 if ret:
  gc.add("time_out")

  reset_target(scope)
  target.flush
 else:
  val = target.simpleserial_read_witherrors('r',40,glitch_timeout=50)
  if val['valid'] is False:
   gc.add("reset")
  else:
   if val['payload'] is None:
    #print(val['payload'])
    continue

   # Convert the byte array to a list of integers
   data = list((val['payload']))

   # Convert the list of integers to bytes
   bytes_data = bytes(data)

   # Unpack the bytes into floating-point numbers
   # Assuming each float is 4 bytes (32 bits) and there are 10 floats
   floats = struct.unpack('10f', bytes_data)
```

```
    is_same, changed_indices = compare_float_lists(prev_values, floats)

    if not is_same:
     gc.add("success")
     print(scope.glitch.width, scope.glitch.offset, scope.glitch.ext_offset)
    else:
     #print("normal")
     gc.add("normal")
print("Done glitching")
```

Listing 7: Helper script for extracting examples of the MNIST dataset.

```
import struct
import numpy as np

def read_idx(filename):
    with open(filename, 'rb') as f:
        zero, data_type, dims = struct.unpack('>HBB', f.read(4))
        shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
        return np.frombuffer(f.read(), dtype=np.uint8).reshape(shape)

def save_examples_with_labels(images_file, labels_file, output_file, num_examples=10)
    images = read_idx(images_file)
    labels = read_idx(labels_file)

    with open(output_file, 'w') as f:
        for i in range(num_examples):
            image = images[i]
            label = labels[i]
            # Save the flattened pixel values in a C-compatible format
            flattened_pixels = ','.join(str(px) for row in image for px in row)
            f.write(f"float example_{i + 1}[X] = {{{flattened_pixels}}};\n")
            f.write(f"int label_{i + 1} = {label};\n")

# Usage
save_examples_with_labels('data/t10k-images-idx3-ubyte', 'data/t10k-labels-idx1-ubyte
```

Listing 8: target implementation to read hidden layer values

```
float h[H] = {0};
float h_old[H] = {0};
float y[Y] = {0};

// h := w'
for (int j = 0; j < H; j++) {
    for (int i = 0; i < X; i++) {
        if (input[i] != 0){
            trigger_high();
        }
        h_old[j] += w[j * X + i] * input[i];
    }
    h[j] = 1.0f / (1.0f + expf(-h_old[j]));
}
trigger_low();

// y := vh
for (int k = 0; k < Y; k++) {
    for (int j = 0; j < H; j++) {
        y[k] += v[k * H + j] * h[j];
    }
}

simpleserial_put('r', Y * sizeof(float), (uint8_t*)h_old);
```

Listing 9: target implementation to read hidden layer values with DWT cycle counter activated. Code reused from Listing 8 is omitted.

```
// DWT register definitions
#define DWT_CONTROL             (*((volatile uint32_t*)0xE0001000))
#define DWT_CYCCNTENA_BIT       (1UL<<0)
#define DWT_CYCCNT              (*((volatile uint32_t*)0xE0001004))
#define DEMCR                   (*((volatile uint32_t*)0xE000EDFC))
#define TRCENA_BIT              (1UL<<24)

// Function prototypes
void InitCycleCounter(void);
void ResetCycleCounter(void);
void EnableCycleCounter(void);
void DisableCycleCounter(void);
uint32_t GetCycleCounter(void);

uint8_t predict(uint8_t cmd, uint8_t subc, uint8_t len, uint8_t *in) {
    float h[H] = {0};
    float y[Y] = {0};

    // Enable DWT cycle counter
    InitCycleCounter();
    ResetCycleCounter();
    EnableCycleCounter();

...


void InitCycleCounter(void) {
    DEMCR |= TRCENA_BIT;
}

void ResetCycleCounter(void) {
    DWT_CYCCNT = 0;
}

void EnableCycleCounter(void) {
    DWT_CONTROL |= DWT_CYCCNTENA_BIT;
}

void DisableCycleCounter(void) {
    DWT_CONTROL &= ~DWT_CYCCNTENA_BIT;
}

uint32_t GetCycleCounter(void) {
    return DWT_CYCCNT;
}
```