

Architecture-aware Timing Fault
Injection Attack for Predictable
Control of AVR microcontrollers

K.D.T Perera
2025



Architecture-aware Timing Fault Injection Attack for Predictable Control of AVR microcontrollers

K.D.T Perera
Index No: 20001292

Supervisor: Prof. Kasun De Zoysa
Co-supervisor: Dr. Asanka P. Sayakkara

June 2025

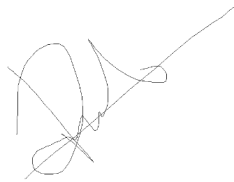
Submitted in partial fulfillment of the requirements of
the B.Sc in Computer Science Final Year Project



Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name: : K.Divaka Tharinda Perera



14-06-2025

Signature & Date

This is to certify that this dissertation is based on the work of
Mr. K.Divaka Tharinda Perera
under my supervision. The thesis has been prepared according to the format
stipulated and is of acceptable standard.

Supervisor Name : Prof. Kasun De Zoysa



2025/06/25

Signature & Date

Co-supervisor Name : Dr. Asanka P. Sayakkara



2025-06-26

Signature & Date

Contents

1	Introduction	5
1.1	Research Questions	7
1.2	Aims and Objectives	8
1.2.1	Aims	8
1.2.2	Objectives	8
2	Related Work	9
2.1	Fault Injection Attacks on Microcontrollers	9
2.2	Vulnerability of ATmega328P	10
2.3	Previous Work	11
2.4	Research Gap	12
3	Methodology and Evaluation Plan	12
3.1	Evaluation Plan	13
3.2	Experimental Setup	14
4	Experiments and Results	16
4.1	Case Study - A faulty clock signal	16
4.2	UART Communication - <code>Serial.println()</code>	17
4.3	I2C and SPI Communication Protocols	20
4.4	Pausing function - <code>delay()</code>	22
4.5	Exploiting Clock Glitching in UART Communication	24
4.5.1	Hypothesis	24
4.5.2	Experimental Setup	25

4.5.3	Observations - Boolean Values	26
4.5.4	Effect on Integer Values	34
4.5.5	Effect on ASCII Characters (Text Data)	36
4.6	Precision Instruction-Level Glitching Using Additional Glitching Circuit	37
4.6.1	Experimental Setup	38
4.6.2	Bit-wise XOR Operation	41
4.6.3	Bit-wise OR Operation	44
4.6.4	Bit-wise AND Operation	46
4.6.5	Bit-wise NOT Operation	49
4.6.6	Addition & Subtraction Operations	52
5	Discussion	56
5.1	Research Question 1	61
5.1.1	UART-Level Fault Injection (Low Precision Clock Glitching)	62
5.1.2	Instruction-Level Fault Injection (Precise Glitching using ESP32)	63
5.2	Research Question 2	65
5.2.1	UART-Level Fault Injection (Low Precision Clock Glitching)	65
5.2.2	Precised Instruction-Level Glitching	66
6	Conclusion	68
6.1	Future Directions	70

List of Figures

1	Setup to inject timing fault attacks	14
2	Faulty clock signal and target clock signal	16
3	Resulting clock signal	17
4	Hardware experimental setup	18
5	Software experimental setup	18
6	Accurate logs are only presented within 15.692456823 MHz and 16.409110402 MHz range	19
7	Hardware setup for both I2C and SPI protocol. The timing fault attack is applied to the master device.	20
8	I2C sender sketch with application of delay()	23
9	I2C receiver sketch with ability to measure elapsed time	23
10	Time taken for one second delay over frequency	24
11	Experiment Setup of the hypothesis scenario	25
12	Logic Analyzer view of original bit pattern of False data packet - 0x00	27
13	Logic Analyzer view of glitched bit pattern of False data packet during the fault injection - 0x80	27
14	Hardware Setup For Precised Clock Glitch Injection	39
15	Relationship between minimum failure probability and the ac- tual hamming weight of the result: Addition	55
16	Relationship between minimum failure probability and the ac- tual hamming weight of the result: Subtraction	55

Listings

1	8bit data packet representing FALSE with LSB first	32
2	Targeted C Code	40
3	Targeted C Code With XOR Operation	41
4	Targeted C Code With OR Operation	44
5	Targeted C Code With AND Operation	46

List of Tables

1	Result of false data bits in contrast to the glitch frequency . .	29
2	Result of true data bits in contrast to the glitch frequency . .	29
3	Probability of faulty result in each Attempt: XOR	43
4	Probability of faulty result in each Attempt: OR	45
5	Probability of faulty result in each Attempt: AND	48
6	Probability of faulty result according to the hamming weight: ADDITION	53
7	Probability of faulty result according to the hamming weight: SUBTRACTION	54

1 Introduction

In the present world, across various platforms, from consumer electronics to industrial machinery, the importance of microcontroller security has increased significantly. ATmega328P has become one of the most widely used microcontroller among those embedded systems due to its power efficiency, versatility and balance of performance (Corporation (2016)). Because of this widespread adoption of this microcontroller also makes them attractive target for malicious attacks. Fault injection is playing a major role in such malicious attacks.

Fault injection attacks introduce faults into a system to alter its intended behavior, data corruption, unauthorized access or complete system failure (Breier & Hou (2022)). These kind of attacks represent critical threat to the reliability and security of embedded systems. Exploiting vulnerabilities in the hardware or software of microcontrollers, attackers can manipulate the control flow of programs running on these devices with severe consequences, particularly in safety critical systems.

Despite the recognized danger, there is limited understanding of the specific impacts of fault injection attacks on the control flow of programs within an 8-bit AVR microcontroller called “ATmega328P”. This research proposal aims to investigate the effects of timing fault attacks on the program control flow of ATmega328P microcontroller. This project proposes a systematic analysis on how timing fault attacks disrupt normal operation of the program control flow aiming to identify potential vulnerabilities that can provide predictable control of the program control flow.

In an era where embedded systems play a critical role in security-sensitive applications, ensuring the reliability and integrity of microcontroller communications is more important than ever. The ATmega328 microcontroller, widely used in embedded devices due to its simplicity and versatility, often relies on the UART (Universal Asynchronous Receiver/Transmitter) protocol for serial communication. However, this protocol by-design lacks robust integrity checking mechanisms, making it susceptible to physical-layer attacks. This thesis investigates a novel class of hardware attack based on clock glitching, whereby the microcontroller’s clock is momentarily altered to introduce timing faults during UART communication. The hypothesis posed is that such glitching can change critical transmitted values—specifically boolean signals used for authentication—from false to true, thereby allowing unauthorized access to secure systems.

Through controlled experiments, the findings confirm that at specific glitch frequencies, clock glitching can successfully flip boolean values transmitted over UART. For instance, a false value represented by 0x00 was observed to flip to values like 0x80, 0xC0, or 0xE0 when the glitching clock was tuned between 17 MHz and 23 MHz, effectively turning a rejection into an acceptance signal. Conversely, a true signal (0x01) could be flipped to 0x00 at frequencies near 1.87 MHz, achieving denial-of-service through intentional rejection of valid credentials. This behavior was not isolated to boolean data types. Further experiments showed that integer values transmitted over UART were similarly affected severely impacting the integrity of numerical computations or commands. In addition to numerical data,

ASCII character transmissions were also vulnerable which could result in corrupted or maliciously altered messages being interpreted by the receiving system. The cumulative evidence from boolean, integer, and character-based experiments strongly supports the hypothesis: UART-based communication on microcontrollers like the ATmega328 can be intentionally disrupted using clock glitching to produce deterministic and exploitable outcomes. This thesis explores the implications of these findings for embedded system security.

In summary this research study will employ combination of simulation and theoretical analysis with empirical testing in order to provide an understanding of the timing fault injection attacks and their implications. The objectives of this research involving characterizing the nature of timing fault attacks specific to ATmega328p microcontroller and its program control flow including assessing the impact of these attacks on the control flow of programs.

1.1 Research Questions

1. **To what extent does timing fault attacks disrupt program execution sequence of ATmega328P microcontroller?**

This research investigate how timing fault attacks effectively impact on program control flow and its execution by observing and analyzing the capabilities of timing fault attacks. This evaluates the extent of disruption causing to the program execution due to the vulnerability of ATmega328P microcontroller. Answering this research question is crucial for improving the reliability of embedded systems used in AT-

mega328P microcontroller.

2. How to conduct timing fault attacks to generate predictable program execution behaviors on ATmega328P microcontrollers?

This research also involving in exploring the methods for executing timing fault attacks in program control flow on ATmega328P microcontroller in a way which those effects produce predictable changes in program execution. This study consisting of selecting fault injection method with designing experimental setups and analyzing the impact of timing fault attacks on program execution flow.

1.2 Aims and Objectives

1.2.1 Aims

1. Evaluating the impacts of timing fault attacks on the reliability and integrity of control flows of the embedded programs in ATmega328P microcontroller.
2. Identifying the suitable timing fault injection methods to make impacts and change program control flow in ATmega328P microcontroller.

1.2.2 Objectives

1. Review and analyzing existing timing fault injection attacks focusing on effects of those attacks in program control flow.
2. Perform systematic experiments in order to evaluate the impact of timing fault attacks on program control flow in ATmega328P microcon-

troller.

3. design and implement a controlled experimental setup to inject timing faults to make impacts on program control flows in ATmega328P microcontroller.
4. Perform systematic experiments to evaluate the impact of timing fault attacks on different program control flows in different conditions.

2 Related Work

2.1 Fault Injection Attacks on Microcontrollers

Fault injection attacks can be used to exploit vulnerabilities in microcontrollers by causing both transient and permanent faults through various methods.

- **Clock glitching:** disrupting the synchronization clock in the microcontroller in order to induce timing errors (Bonny & Nasir (2019)). Misleading synchronizing clock of a circuit causes instruction execution, data storing and retrieving have wrong behaviors.
- **Voltage glitching:** applying changes in the voltage of the power supply in order to cause errors in microcontroller processing. Gnad et al. (2017) has shown capability of voltage glitching on making a field programmable gate array (FPGA) circuit crash within milliseconds. Not only that but also successful voltage injection can cause erroneous re-

sults from Advance Encryption Standard (AES) process on FPGA circuits.

- **Electromagnetic interference:** disturbing the microcontroller’s operation using electro-magnetic fields (Dumont et al. (2019)). Ordas et al. (2017) has shown the effectiveness of the electro-magnetic fault attacks on bit set and bit reset actions on FPGA circuits. Also Delarea & Oren (2022) has described the possibility of electro-magnetic fault injection on skipping instructions of ARM embedded systems.
- **Laser fault injection:** altering the state of the microcontroller’s internal circuits using a focused laser beams (Colombier et al. (2021)). Successful laser fault injection can cause bit flipping intentionally on FPGA circuits (He et al. (2016)) and ARM embedded systems (Vasselle et al. (2017)).

Due to the capability of above attacks in disrupting the processing flows of microcontrollers, these attacks can be used to disrupt the program control flow in such microcontrollers.

2.2 Vulnerability of ATmega328P

Unlike other advanced microcontrollers, the ATmega328P lacks built-in security features to detect and mitigate fault injection attacks. Also its popularity and applications increase the likelihood of targeted fault injection attacks. Specially due to the open nature of Arduino ecosystem, it provides potential

attackers with resources to study and exploit its ATmega328P microcontroller. Banerjee et al. (2022) has stated that how accurate the power-based side channel attack to extract AES keys on Atmega328 microcontroller, which shows the vulnerability of Atmega328 microcontroller. Sanjaya et al. (2024) has also shown the vulnerability of Atmega328 microcontroller when it comes to physical layer supply voltage coupling.

2.3 Previous Work

Researches have demonstrated that the employing of fault injection attack types including clock glitching, voltage glitching and electro-magnetic fault injection can cause significant threat in microcontroller operation. Breier & Hou (2022) shows the possibility of employing such attacks on architectures including ARM, Intel, AMD and FPGA with discussing the practicality and cost-efficiency. Furthermore, O’Flynn (2016) has demonstrated that voltage glitching has significant impact on in-built registers which effects the program instructions when it comes to 8-bit AVR microcontrollers. Also O’Flynn (2016) provide evidence on the feasibility of fault injection attacks on 8-bit AVR microcontrollers like ATmega328P. Tehranipoor et al. (2023) has shown that applying faults using voltage fault injection on FPGA which causes erroneous results on AES encryption process. With clock glitching, Bonny & Nasir (2019) has demonstrated that how the generated bits stream getting distorted after a successful clock glitching attack on FPGA.

2.4 Research Gap

As from the previous research works, we can conclude that there have been enough research work done to assess and evaluate the hardware level effects on ATmega328P microcontroller with fault injection techniques. However, the specific effects on the program control flow within the ATmega328p microcontroller remain under-explored. This research gap includes finding out the capability of producing erroneous results on critical program control flows like security gateways and conditional flows. According to this research gap, this research study will introduce timing faults on to a program control flow operating in an ATmega328P microcontroller using clock glitching attacks and observe the effectiveness of those attacks.

3 Methodology and Evaluation Plan

As mentioned above, the main purpose of this research is to investigate the impact of timing fault attacks to program control flow within ATmega328P microcontroller as well as determining methods to generate predictable faulty program execution behaviors using such fault injection attacks. After successfully gathering existing well established knowledge on timing fault attacks on other similar microcontrollers and vulnerabilities of ATmega328P microcontroller, this research study is proposing a methodology to answer above research questions with systematic experimental setup and systematic result evaluations.

3.1 Evaluation Plan

1. **"To what extent does timing fault attacks disrupt program execution sequence of ATmega328P microcontroller?"** To answer the first research question, this research study will evaluate the experiments to identify the extent of disruptions in program execution flow with the impact of timing fault attacks. The metrics including number of program crashes, types of program anomalies and frequency of control flow deviation will be taken into account when conducting this evaluation. Other than statistical analysis of recorded execution traces, this evaluation plan will also include comparison of fault and fault free execution paths and ability of reproducing the identified disruptions across multiple trials.
2. **"How to conduct timing fault attacks to generate predictable program execution behaviors on ATmega328P microcontrollers?"**

In order to answer second research question, the evaluation will be focusing on the predictability of behaviors of program execution flow during and after the timing fault injection. This evaluation will consists of metrics including consistency in program behavior under repeated identical fault injection attack and ability of inducing specific disruptions to the program execution flow in a reliable way followed by analysis of program execution behaviors to identify predictable patterns. This evaluation will be concluded with the demonstration of ability to predict and reproduce specific disruptions to the program execution using timing fault attacks.

3.2 Experimental Setup

Main objective of this experimental setup is to establish a controlled environment for conducting timing fault experiments on program execution and its control flow within ATmega328P microcontroller. During the experiment, this research study will be using Arduino Uno development board which consists an ATmega328P microcontroller.

Hardware setup will be consisting of enough Arduino Uno development boards with their ATmega328P microcontrollers. Apart from microcontrollers, the following hardware devices will be used in the experiments.

1. Function generator as a synchronization clock controlling device in order to introduce timing faults on the target device.
2. A monitoring device to capture program execution traces.
3. Oscilloscope to observe the faulty clock signals.

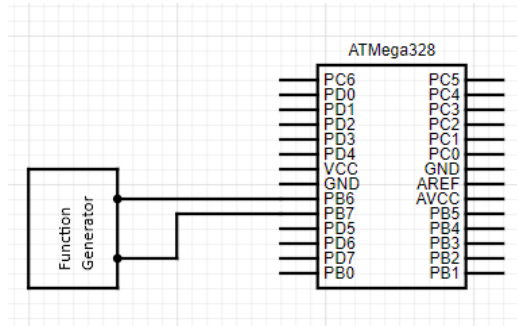


Figure 1: Setup to inject timing fault attacks

PB6 and PB7 are the pins which are connected to a crystal oscillator to maintain the clock signals through out the entire microcontroller. As displayed in Figure 1 above, the PB6 and PB7 pins will be bypassed and connected to a function generator so that those pins will be connected to the clock signal generator of the Arduino Uno board and the function generator at the same time. Using this setup we can generate the desired clock signals through the function generator without modifying the original development board. If we don't need to generate any timing faults, we need to keep the function generator to produce a 16 MHz clock signal so that the external clock signal will overlap with the recommended clock signal generated by the Arduino Uno development board. This will make no faults on the microcontroller.

Software setup will be consisting embedded systems covering range of control flows constructs focusing on following aspects,

- Arithmetic operations
- Bit-wise operations
- UART/Serial communication
- I2C communication
- SPI communication
- Delays

4 Experiments and Results

4.1 Case Study - A faulty clock signal

According to the experimental setup stated above, the operational clock signal of the entire microcontroller will be the combination of the clock signal generated by the Arduino Uno development board and the faulty clock signal generated by the function generator.

Motivation - Observe the what will be the resulting clock signal generated by the combination of the faulty clock signal and the operational clock signal supplied by the Arduino Uno development board.

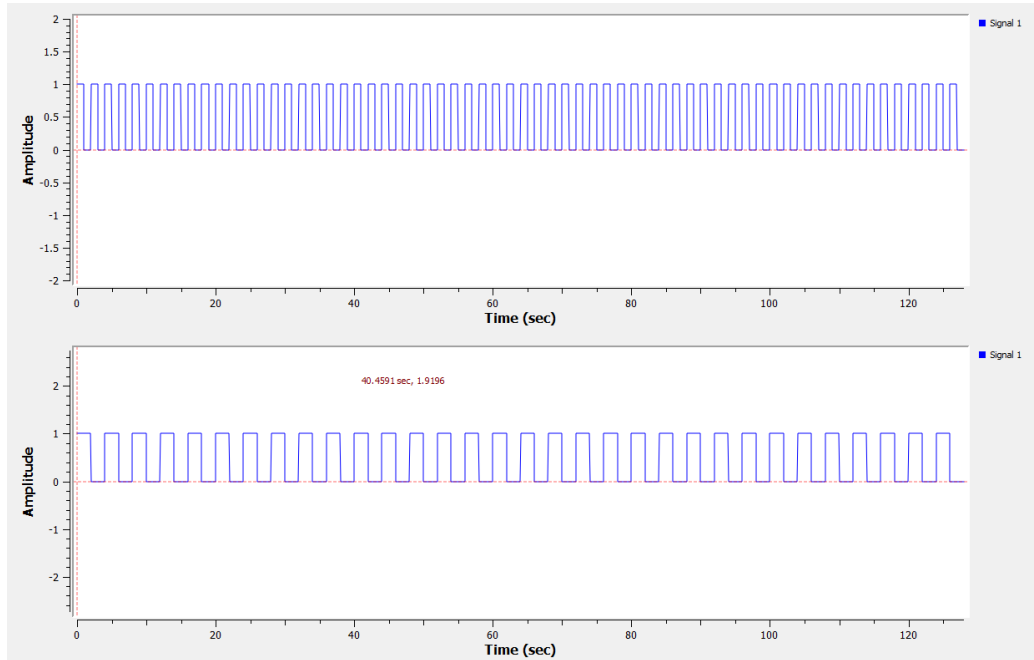


Figure 2: Faulty clock signal and target clock signal

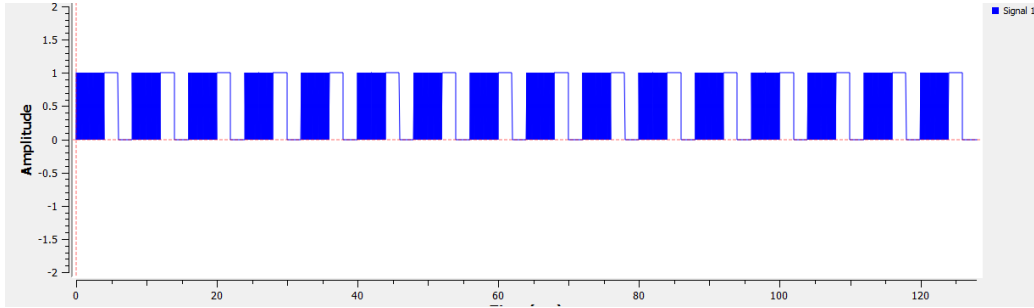


Figure 3: Resulting clock signal

This study has been conducted using the GNU Radio Companion simulator and the results have been verified by a single oscilloscope with two connected function generator outputs representing exactly the same input clock signals used in simulation. According to this study and the results (Figure 2 and Figure 3) we can conclude that the resulting clock signal of the combination of two different clock signals will be the OR operation between those clock signals.

This section focuses on identifying the potential vulnerabilities of the ATmega328 microcontroller in order to lay down the foundation and objectives of this research's direction.

4.2 UART Communication - `Serial.println()`

Motivation - Since ATmega328 microcontroller doesn't support any additional debugging options, `Serial.println()` method used with UART communication is considered as a major method for observations, logs and debugging. So it is crucial to make sure `Serial.println()` method is not effected by the timing fault attack we are injecting, for accurate logs.

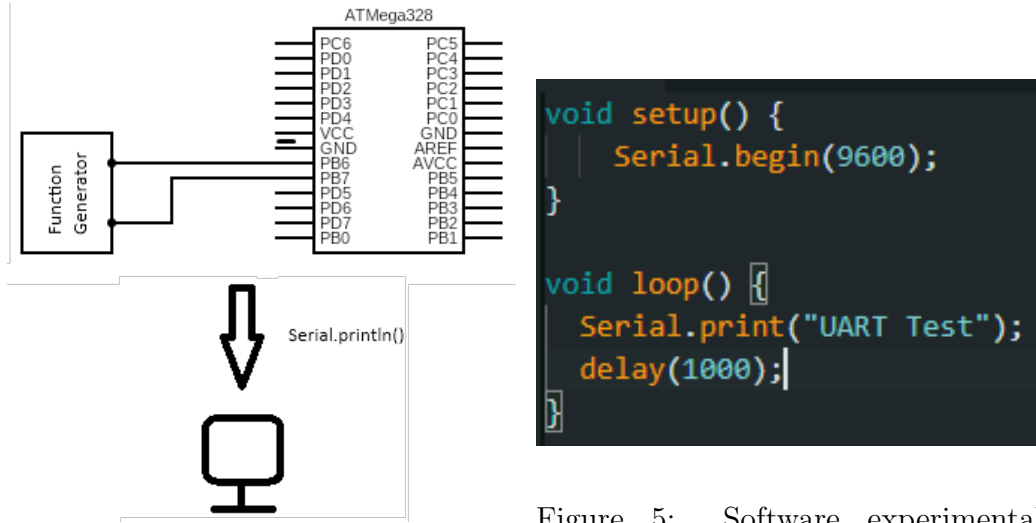


Figure 4: Hardware experimental setup

Figure 5: Software experimental setup

This experiment has been conducted by applying varying clock frequencies from 1 MHz to 25 MHz and observing whether the embedded system provides accurate logs according to the software experimental setup (See Figure 5).

Results and Conclusion - During this experiment it has been observed that the logs are not accurate if the glitch clock frequency is not within 15.692456823 MHz and 16.409110402 MHz range. So we can conclude that we cannot expect accurate and valid logs when we are applying timing fault attacks with frequencies which are not from the above mentioned range.

The results of this experiment are leading us to find an accurate and a valid method for log observation when we are applying timing fault attacks. Furthermore, this vulnerability exposes a further experiments to identify the

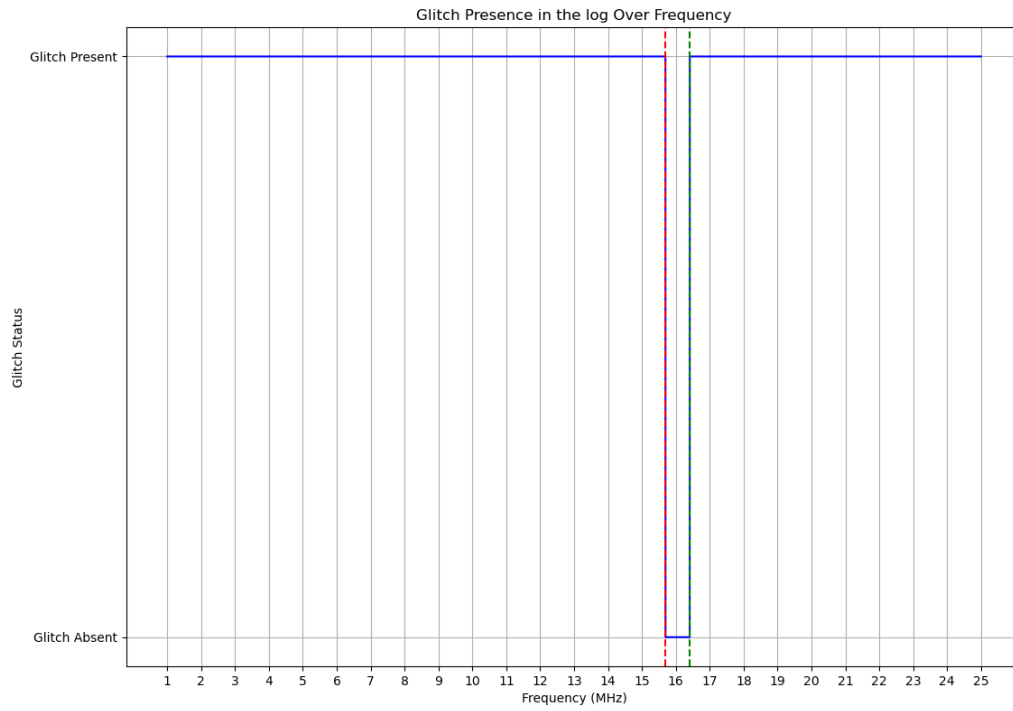


Figure 6: Accurate logs are only presented within 15.692456823 MHz and 16.409110402 MHz range

potential answers for the research questions in latter sections.

4.3 I2C and SPI Communication Protocols

Motivation - Identifying the effects of timing fault attacks when sending and receiving data using I2C and SPI communication protocols used in embedded systems. Also this experiment is subjected to an aim of finding a solution to replace `Serial.println()` method for log monitoring.

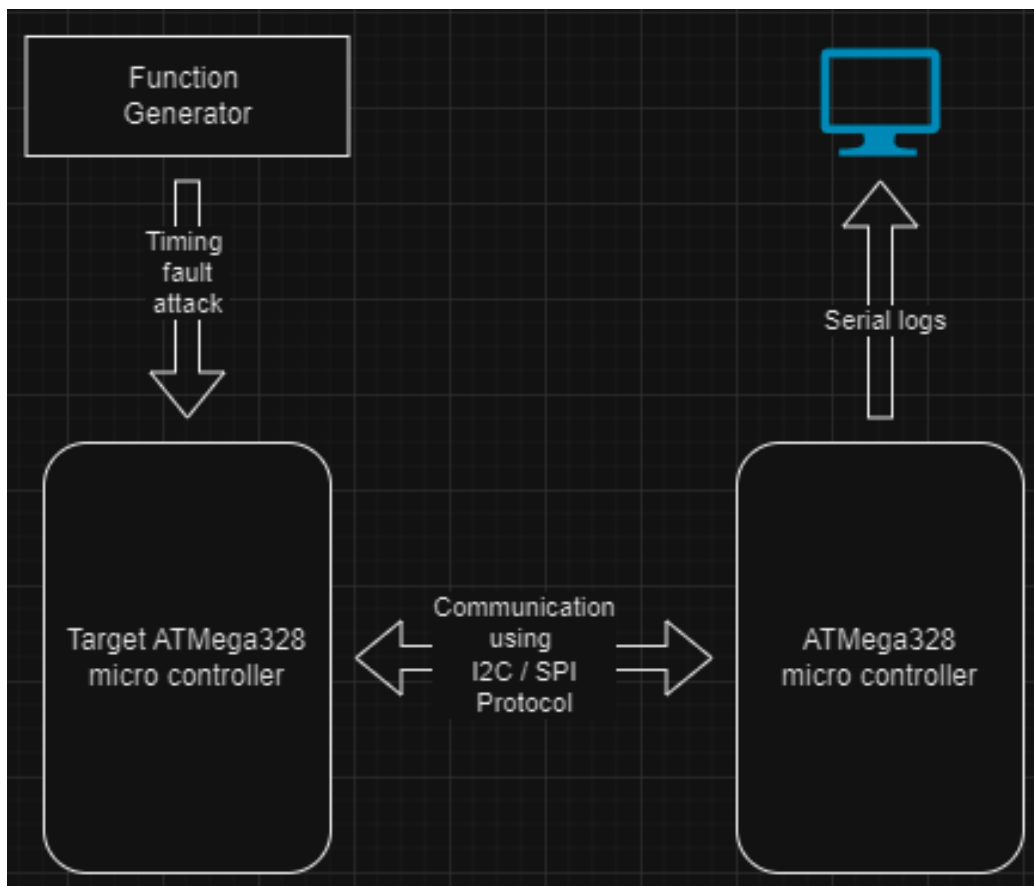


Figure 7: Hardware setup for both I2C and SPI protocol. The timing fault attack is applied to the master device.

Observations - For both I2C and SPI communication protocols, the data

that are communicating are not getting changed or effected by inaccuracies relevant to the varying glitch clock signals (from 1 MHz to 25 MHz) generated by the function generator. Reason for this behavior is that the both I2C and SPI protocols are operating using the same clock synchronization clock signal for both sender and receiver.

Conclusion - We can conclude that when communicating using both I2C and SPI protocols and upon applying timing fault attacks on sender is not affecting the data we are communicating. Due to this behavior, the hardware setup (Figure 7) can be used as a solution as a `Serial.println()` for observation of logs and outputs.

4.4 Pausing function - `delay()`

Motivation - The `delay()` function in Arduino is crucial for timing control. Exploring how clock glitching affects `delay()` on an Arduino is fascinating because it explores the impact of timing disruptions on code execution stability. Since `delay()` depends on precise clock cycles to pause the program accurately, introducing glitches can reveal vulnerabilities in timing-sensitive tasks. By observing how glitches alter `delay()` intervals, we can gain insights into how fault injection attacks could interfere with or even control program flow—key knowledge for developing more robust, secure embedded systems.

Hardware setup - Use the same hardware experimental setup used in the previous experiment (see figure 7).

```

#include <Wire.h>

void setup() {
  Wire.begin(); // Join I2C bus as master
  Serial.begin(9600); // Start Serial for debugging
}

void loop() {
  bool numberToSend = 1; // Example number to send

  Wire.beginTransmission(8); // Address of the slave
  Wire.write(numberToSend); // Send the number
  Wire.endTransmission(); // Stop I2C transmission

  Serial.print("Sent: ");
  Serial.println(numberToSend); // Print the sent number

  delay(1000); // Wait for 1 second before sending again
}

```

Figure 8: I2C sender sketch with application of delay()

```

#include <Wire.h>

int receivedNumber;
unsigned long lastReceiveTime = 0; // Store the time of the last message
unsigned long currentTime = 0; // Store the current time
float timeDifference = 0;
int count = 1; // Store the time difference in seconds

void setup() {
  Wire.begin(8); // Join I2C bus with address #8
  Wire.onReceive(receiveEvent); // Register event handler for receiving
  Serial.begin(9600); // Start Serial for debugging
}

void loop() {
  // Do nothing in the main loop
}

// Function that executes whenever data is received
void receiveEvent(int howMany) {
  currentTime = millis(); // Record the current time in milliseconds

  // Calculate the time difference in seconds (convert ms to s)
  timeDifference = (currentTime - lastReceiveTime) / 1000.0;

  // Store the time of this message for the next calculation
  lastReceiveTime = currentTime;

  // Read and store the received number
  while (Wire.available()) {
    receivedNumber = Wire.read();
    Serial.print(count++);
    Serial.print(" Received (" + String(timeDifference) + "): ");
    Serial.println(receivedNumber); // Print the received number
  }
}

```

Figure 9: I2C receiver sketch with ability to measure elapsed time

This experiment is using the alternative hardware setup used in "I2C and SPI Communication Protocols" experiment. The sender is sending digit "1" in every second and the receiver is receiving the digit "1" and calculate the time it taken for the delay.

Observations - With recommended 16 MHz clock signal, the time taken for the one second delay is exactly a second. But when the clock frequency getting increased, the time taken for the one second delay is getting decreased and when the clock frequency decreases the time taken for one second delay is getting increased.

Conclusion - So the observation is negative correlation between the delay and the glitch frequency. This is a crucial observation as the timing fault

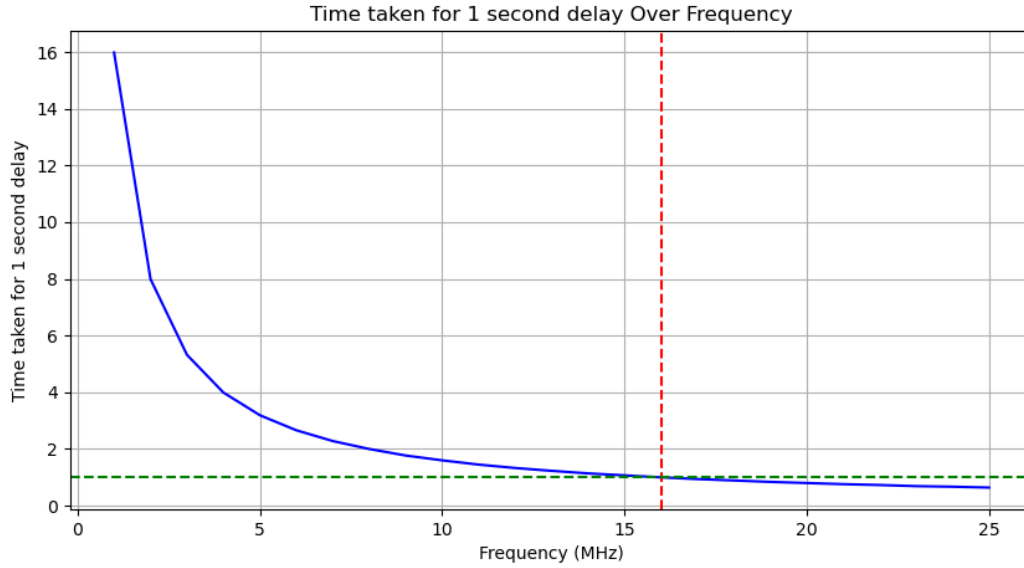


Figure 10: Time taken for one second delay over frequency

attacks can interfere the amount of time control of the embedded system specially when it requires precise pausing situations.

4.5 Exploiting Clock Glitching in UART Communication

4.5.1 Hypothesis

This section presents a hypothesis on the vulnerability of UART-based communication between an ATmega328 microcontroller and a door lock system. The microcontroller is responsible for verifying user credentials and transmitting an authentication result—true (unlock) or false (deny access)—to the door lock. However, due to the nature of UART synchronization, a clock glitch attack can induce faults in data transmission. This hypothesis explores

the possibility of altering a transmitted false signal into true, thereby granting unauthorized access to the door lock. The analysis is based on observed behaviors of clock glitching on ATmega328, where controlled glitches can selectively modify transmitted data bits.

4.5.2 Experimental Setup

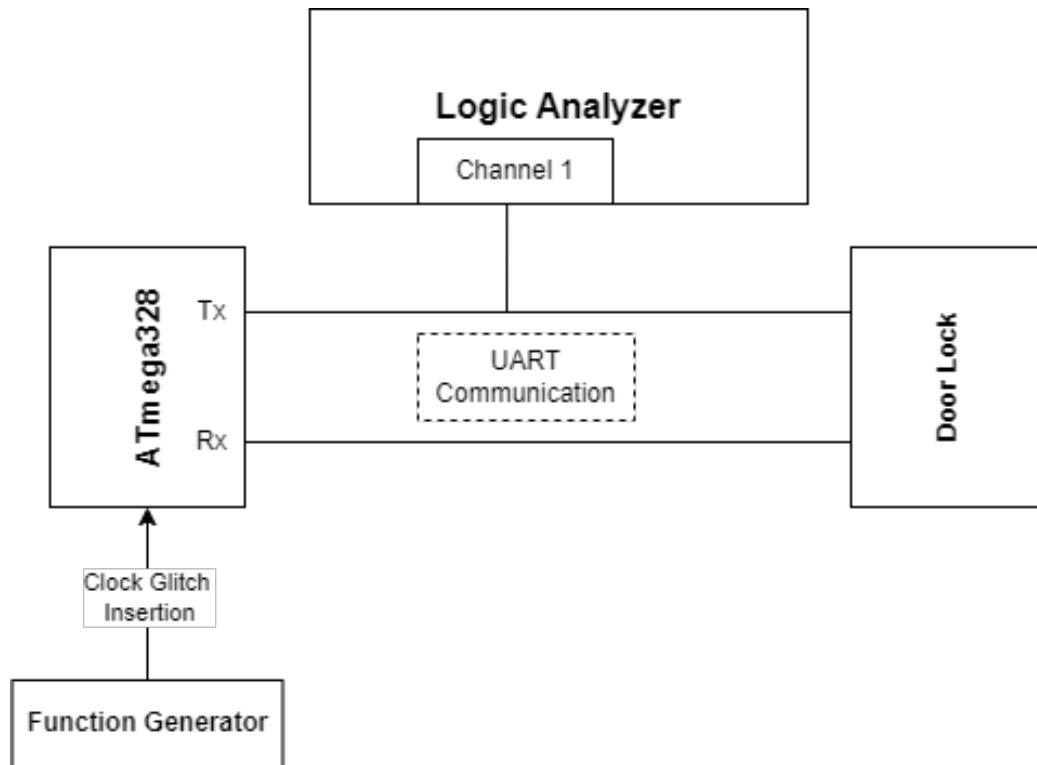


Figure 11: Experiment Setup of the hypothesis scenario

The experimental setup consists of an ATmega328 microcontroller which is responsible of sending the relevant signal to the door lock system using UART protocol, a door lock system as a receiver of the authentication from

ATmega328, a logic analyzer in order to monitor the data packets transferring in between the two devices and a function generator used to inject glitch clock pulses.

The ATmega328 microcontroller is used to send the 'true' or 'false' according to the authorization. This will be sent to the door lock system. In this experiment, a logic analyzer has been used to monitor the communication between the devices. Using the logic analyzer we can inspect the data packets transferring through the channel. In this experiment, the specific scenario has been simulated. This scenario includes an unauthorized access and the a communication which the microcontroller is sending 'false' to the door lock system. During this communication, a function generator has been used to insert a glitched clock signal with precise control of the frequency.

4.5.3 Observations - Boolean Values

Under normal operation, the microcontroller verifies user credentials and transmits either a true (unlock) or false (deny access) signal to the door lock. However, controlled clock glitching can disrupt this transmission, causing unauthorized access. The findings demonstrate that:

- When the glitching clock frequency exceeds 17 MHz, a false signal is altered and received as true, allowing unauthorized entry.
- When the glitching clock frequency is 1.87 MHz, a true signal is corrupted and received as false, denying legitimate access.

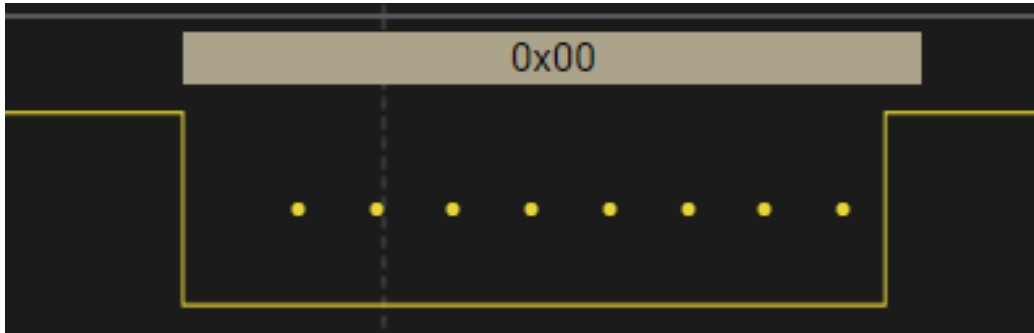


Figure 12: Logic Analyzer view of original bit pattern of **False** data packet - 0x00

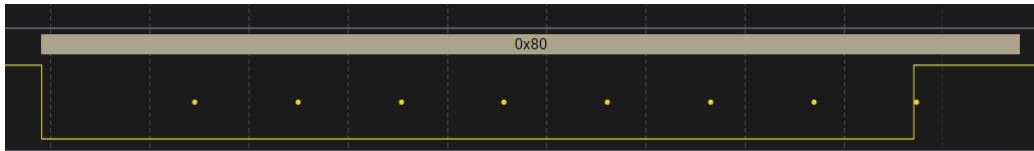


Figure 13: Logic Analyzer view of glitched bit pattern of **False** data packet during the fault injection - 0x80

Figure 12 demonstrates the logic analyzer observation of bit pattern representing the **false** boolean hex value(0x00). This observation is taken where there is no fault injection(no glitched situation). Since the receiving device is referring to the boolean values, this value is considered as the **false** boolean value at the receiving end.

Figure 13 is the logic analyzer observation of data transmission during the fault injection. It demonstrates the logic analyzer observation of bit pattern representing a **true** boolean hex value(0x80). Since the receiving device is referring to the boolean values, this value is considered as the boolean value **true** at the receiving end.

During both scenarios, The microcontroller sends the **false** boolean value. But in the glitched scenario, due to the clock glitch, the UART communication has lost its synchronization. Due to this failed synchronization, the receiving device is receiving a wrong data packet.

Since the UART communication doesn't share a common clock signal between the sender and receiver, the sending device and receiving device need to operate in 16Mhz clock frequency. Due to this the sending device assumes that receiver is receiving the data packet with 16Mhz frequency, and the sending device sends 8bit data in 16Mhz frequency and LSB first. Also the receiving device assumes that the sending device is sending 8bit data in 16Mhz frequency, so the receiving device captures the data with 16Mhz frequency. During the idle time, the transmission channel stays at logic level high. When the sender sends data, it makes the channel low to indicate the starting of the data transmission and sends data in 8 bits. When the sender completes the 8bit data transmission, it will again set the channel to logic level high. During this transmission if we insert a clock glitch to the sender, it will alter the working clock frequency of the sending device and the sending device and receiving device will not be working with the same clock frequency. Hence losing the synchronization during the data transmission. In the experiment scenario, the clock glitch makes the sending device to send data with more than 16Mhz frequency. But still the receiving device works with 16Mhz. Since the sending device makes the channel high after transmitting the 8bits, the receiving device captures that high signal as the last data bit. Hence, the receiving device receives the 0000 0000 (0x00)

data as 0000 0001 (0x80 - this represents true in boolean values). This makes the receiving device capture the data as **true** even-though the sending device sends the **false**.

Glitch Frequency (Mhz)	16	17 - 19	20 - 22	23	24 - 25
Observations	0x00	0x80	0xC0	0xE0	Error

Table 1: Result of **false** data bits in contrast to the glitch frequency

Glitch Fre- quency (Mhz)	16 - 13	12 - 11	10	9 - 8	7	6	5	4 - 2	1.86 - 1.89	1
Observa- tions	0x01	0x03	0x02	0x06	0x0E	0x0C	0x1C	0xF0	0x00	Error

Table 2: Result of **true** data bits in contrast to the glitch frequency

Table 1 illustrates the effect of clock glitching at various frequencies on the transmission of a false data bit in a UART communication system involving an ATmega328 microcontroller and the digital door lock. The primary objective of this experiment is to analyze how different glitching frequencies impact the integrity of the false signal, which is originally represented as 0x00 in hexadecimal format. By observing how the received data changes

under different glitch frequencies, this table provides critical insights into data corruption patterns and their implications for security.

The first row of the table represents the applied glitch frequencies in megahertz (MHz), ranging from 16 MHz up to 24–25 MHz. Note that injecting 16Mhz glitched clock is not inducing any faults on the microcontroller and it is considered as the no-fault scenario. Each frequency range corresponds to an experimental scenario where clock glitching was introduced to the system while transmitting a false signal. The second row displays the observed data received at the receiving end, which highlights how the original 0x00 (false) value was altered due to the effects of clock glitching.

Analysis of Observations:

- At 16 MHz: The received value remains 0x00, indicating that at this frequency, the glitching does not significantly affect data integrity. This suggests that the transmission is still stable, and no bit flips occur at this point.
- At 17–19 MHz: The received value changes to 0x80, which indicates that a significant bit flip has occurred. Specifically, in an 8-bit binary representation, 0x80 corresponds to 10000000, meaning that the most significant bit (MSB) was flipped from 0 to 1, while the remaining bits remained unchanged. This marks the first clear indication that clock glitching at this frequency range can introduce errors into the transmission.
- At 20–22 MHz: The received value is 0xC0, which in binary is 11000000.

Compared to 0x80, this shows that another bit (the second MSB) has also flipped, suggesting that increasing the glitch frequency causes more bits in the transmitted data to be altered. This indicates a progressive corruption of the originally transmitted false signal.

- At 20–22 MHz: The received value is 0xC0, which in binary is 11000000. Compared to 0x80, this shows that another bit (the second MSB) has also flipped, suggesting that increasing the glitch frequency causes more bits in the transmitted data to be altered. This indicates a progressive corruption of the originally transmitted false signal.
- At 20–22 MHz: The received value is 0xC0, which in binary is 11000000. Compared to 0x80, this shows that another bit (the second MSB) has also flipped, suggesting that increasing the glitch frequency causes more bits in the transmitted data to be altered. This indicates a progressive corruption of the originally transmitted false signal.
- At 23 MHz: The received value is 0xE0, which in binary is 11100000. This pattern demonstrates further bit corruption, with three of the most significant bits being flipped. The increasing error rate suggests that as the glitch frequency rises, more bits within the byte are affected, reinforcing the idea that clock glitching directly influences UART transmission errors.
- At 24–25 MHz: The system records an "Error," indicating that at this frequency, the glitching becomes so severe that the communication fails entirely. The receiver is unable to interpret any meaningful data,

likely due to excessive synchronization issues between the sender and receiver. This suggests that beyond a certain frequency threshold, the UART transmission mechanism is entirely disrupted, preventing the reception of even a corrupted message.

Listing 1 demonstrates how the bit shifts occur according to the glitch frequency during the UART communication.

```

16Mhz: 0000 0000 -> LSB to MSB -> 0000 0000 (0x00)
17Mhz: 0000 0001 -> LSB to MSB -> 1000 0000 (0x80)
20Mhz: 0000 0011 -> LSB to MSB -> 1100 0000 (0xC0)
23Mhz: 0000 0111 -> LSB to MSB -> 1110 0000 (0xE0)

```

Listing 1: 8bit data packet representing FALSE with LSB first

The table 2 illustrates the impact of various glitch frequencies on the integrity of the data bits representing TRUE during UART communication between an ATmega328 microcontroller and the digital door lock. This table specifically investigates how different clock glitching frequencies affect the reception of a true signal, which is originally represented as 0x01 in hexadecimal format. By analyzing the received data under different glitching conditions, this experiment provides insight into the extent of data corruption and its implications for system security.

The first row of the table categorizes the applied glitch frequencies, ranging from 16–13 MHz down to 1 MHz, with a critical intermediate range of

1.86–1.89 MHz, which has been experimentally observed to cause significant disruptions in UART communication. The second row lists the observed data received at the receiving end when the original transmitted value was 0x01. These hexadecimal values indicate how the transmission was altered due to clock glitching effects.

Examining the results, it is evident that higher glitch frequencies introduce minor corruption. At 16–13 MHz, the received data remains 0x01, indicating no disruption. However, as the glitch frequency decreases, more significant alterations emerge. For instance, at 12–11 MHz, the received value changes to 0x03, while at 10 MHz, it shifts to 0x02, demonstrating that specific bits of the original data are being affected. Further reductions in frequency introduce more pronounced distortions, such as 0x06, 0x0E, and 0x0C at intermediate frequencies, showing progressive bit flips and increasing data corruption.

A crucial observation occurs at the 4–2 MHz range, where the received data is 0xF0. This suggests substantial corruption of the original 0x01 value, potentially flipping multiple bits within the transmitted byte. The most significant finding, however, is at the 1.86–1.89 MHz range, where the received value is 0x00. This indicates a complete inversion of the original 0x01 (true) into 0x00 (false), confirming that a glitching clock at approximately 1.87 MHz can fully alter the authentication signal and potentially deny legitimate access.

At 1 MHz, the system records an "Error," meaning that at this low frequency, the glitching disrupts communication so severely that the receiver

fails to interpret any valid data. This likely results from a breakdown in the UART synchronization mechanism, leading to complete transmission failure.

Overall, this table provides concrete experimental evidence that clock glitching can systematically alter transmitted data during UART communication. The ability to flip a 0x01 (true) to 0x00 (false) at a specific frequency confirms the hypothesis that carefully timed glitches can exploit vulnerabilities in asynchronous communication protocols like UART.

The findings from Tables 1 and 2 demonstrate that clock glitching can systematically alter the integrity of transmitted UART data packets, flipping true (0x01) to false (0x00) and vice versa. However, this phenomenon is not exclusive to boolean values; it extends to any 8-bit data packet, including integer values and ASCII characters. Since UART transmits data in byte-sized (8-bit) packets, any numerical or textual information represented in 8-bit format is vulnerable to similar corruption patterns when exposed to clock glitches at specific frequencies. Due to these findings, the experiments have been extended into another step by considering the integers and ASCII characters to illustrate how clock glitching can affect data types other than Boolean values.

4.5.4 Effect on Integer Values

This experiment has been conducted by considering a scenario where an ATmega328 microcontroller is transmitting integer values over UART. Normally, an integer is represented in an 8-bit binary format before being transmitted. However, due to clock glitching, bit flips occur, altering the transmitted value

into an unintended number. The following scenarios illustrate this effect:

1. Scenario 1: Integer 25 (0x19 in Hexadecimal, 00011001 in Binary)

- If no glitching occurs, the receiving end correctly interprets the transmitted data as 25.
- However, at 17–19 MHz, based on the pattern observed in Table 1, a bit flip occurs in the most significant bit (MSB), altering the binary representation to 10011001 (0x99 in hex), which corresponds to 153 instead of 25.
- At 20–22 MHz frequency, the second MSB flips as well, resulting in 11011001 (0xD9), changing the value to 217 instead of 25.
- When the glitch occurs at 1.86–1.89 MHz, the entire packet has been corrupted to 0x00, resulting in the integer being received as 0.

2. Scenario 2: Integer 200 (0xC8 in Hexadecimal, 11001000 in Binary)

- Normally, 200 is transmitted as 0xC8. At 16 MHz, no changes occur, and 200 is correctly received.
- At 23 MHz frequency, the third MSB has been flipped, resulting in 11101000 (0xE8). This has been received as 232.
- In the worst-case scenario, at 1.86–1.89 MHz, the entire packet has been flipped to 0x00, resulting in 0 being received.

These experimental scenarios show that a malicious glitch insertion or accidental glitch can dramatically change numerical values in a way that

disrupts calculations, control commands, or authentication mechanisms that rely on numerical data.

4.5.5 Effect on ASCII Characters (Text Data)

ASCII characters are also transmitted as 8-bit data packets, making them equally vulnerable to corruption. If a clock glitching applied to a UART-based communication channel transmitting text, individual characters can be altered, leading to unintended modified messages.

1. Scenario 1: Character 'A' (0x41 in Hexadecimal, 01000001 in Binary)
 - Normally, sending 0x41 results in 'A' being received.
 - At 17–19 MHz, a bit flip alters the binary representation to 11000001 (0xC1), which corresponds to 'Á' instead of 'A'.
 - At 23 MHz, another bit flip changes 0x41 to 0xE1 (11100001), resulting in 'á' instead of 'A'.
 - At 1.86–1.89 MHz, a full corruption flips all bits to 0x00, making the character unreadable and registering as a null byte.
2. Scenario 2: Word "HELLO" (ASCII: 0x48 0x45 0x4C 0x4C 0x4F)
 - If an attacker applies a 20–22 MHz glitch, individual bits might be flipped, changing "HELLO" (0x48 0x45 0x4C 0x4C 0x4F) to 0xC8 0xE5 0xCC 0x4C 0x0F, which displayed as "ÈeÌL"— completely distorting the original message.

- A 1.87 MHz glitch flips characters into null bytes (0x00), making the entire message unreadable.

This type of glitching attack has significant security implications, particularly in communication systems, authentication protocols, and control signals. If a system is relying on specific text-based commands ("OPEN_DOOR" or "SET_TEMP 25"), clock glitching could alter or erase critical words, leading to security vulnerabilities or operational failures.

4.6 Precision Instruction-Level Glitching Using Additional Glitching Circuit

While the previous UART glitching experiments revealed significant vulnerabilities in asynchronous serial communication, they were based on a low-precision clock glitch injection method. This technique, although effective in corrupting UART-transmitted data, lacked fine-grained control over where the glitch was applied in the program's execution. The primary limitation of this approach is its indiscriminate nature as it affects the entire ATmega328 microcontroller's execution environment, not a specific instruction. Consequently, while successful at altering UART packets, this method cannot reliably target particular parts of the program control flow without potentially causing widespread instability or unintended faults elsewhere in the application.

To overcome this limitation and extend the scope of clock glitching to instruction-level precision, a new approach was developed using an ESP32-based glitching circuit. This advanced method introduces a significant im-

provement in control and reliability. In this design, the ATmega328 microcontroller is programmed to emit a digital signal at the moment a specific instruction or critical operation is about to be executed. This digital output line is connected to an interrupt pin on the ESP32.

When the designated signal is triggered, it causes an interrupt on the ESP32, which immediately activates its glitching routine. The ESP32 then generates a noisy glitched clock signal and injects it into the ATmega328's clock line. This allows the glitch to be introduced synchronously with the execution of a specific instruction, dramatically increasing the accuracy and repeatability of the attack.

This high-precision setup brings numerous advantages over the previous UART-based method. First, it ensures targeted instruction fault injection, allowing experiments to explore the security of specific code paths such as arithmetic and bit-wise operations. Second, it minimizes the collateral impact on the rest of the program, reducing the risk of crashing the system and increasing the chances of successful, stealthy exploitation.

By narrowing the attack surface from system-wide UART faults to specific instructions, this method bridges the gap between theoretical side-channel vulnerabilities and practical exploitation techniques. It opens the door to a more granular analysis of embedded control flow security.

4.6.1 Experimental Setup

The experimental setup consists of following components,

1. **Glitching Device:** Responsible for producing the clock glitch signal

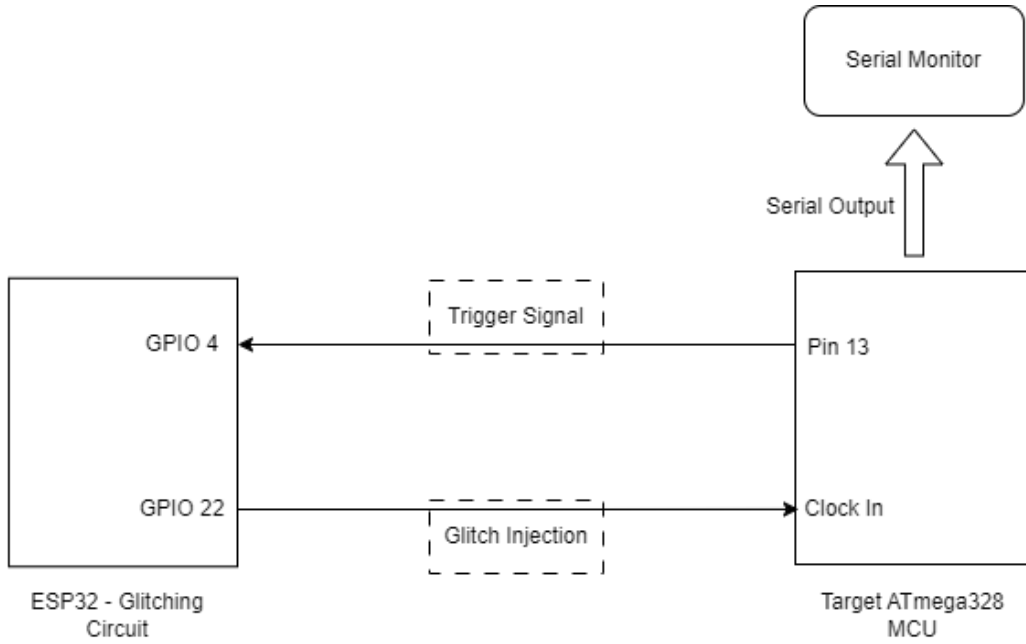


Figure 14: Hardware Setup For Precised Clock Glitch Injection

upon an output interrupt. This glitch signal will be eventually injected into the clock-in interface of the target ATmega328 MCU. ESP32 development board has been used to develop this device. GPIO pin 4 will be used to capture the digital trigger signal. Changes in this pin will trigger an interrupt. When the input is HIGH, the interrupt function is producing a glitch signal using I2S protocol using GPIO pin 22. This pin is directly connected to the clock-in interface of the target MCU. When the input is LOW the interrupt function stops producing the glitch signal.

2. **ATmega328 Microcontroller:** This is the target device that executes the specific program control flow. This MCU responsible for producing a digital signal just before the targeted instruction. It sends

a digital HIGH signal using pin 13 just before starting critical section in the code. Once after the critical section, it sends digital LOW signal using the pin 13. This signal is triggering an interrupt in the glitching device. Refer to the Listing 2.

```
void setup(){
    pinMode(13, OUTPUT);
}

void loop(){
    triggerFunc(true); // Trigger the signal
    delayMicroseconds(10);

    // Affected Code

    triggerFunc(false); // Turning trigger signal off
}

void triggerFunc(bool trigger){
    if(trigger){
        digitalWrite(13, HIGH);
    }else{
        digitalWrite(13, LOW);
    }
}
```

Listing 2: Targeted C Code

3. Programs including arithmetic operations and bit-wise operations. Mainly focusing on bit-wise XOR, OR, AND, NOT and arithmetic operations addition and subtraction.

However, using this method, we cannot have a precise control over the glitched clock signal produced by the glitching device as it always generate a random noise. This limitation causes a failure in interpreting a relationship between the properties of the glitched signal and the observations in the program execution flow.

4.6.2 Bit-wise XOR Operation

```
uint8_t key = 0x5A;
uint8_t data = 0xC3;

void loop(){
    triggerFunc(true); // Trigger the signal
    delayMicroseconds(10);
    // Critical section begin
    uint8_t encrypted = data ^ key; // XOR operation
    // Critical section end
    triggerFunc(false); // Turning trigger signal off
}
```

Listing 3: Targeted C Code With XOR Operation

To further explore the capabilities of precise instruction-level clock glitching, a focused experiment was conducted on arithmetic logic unit (ALU) operations—in particular, the bitwise XOR operation—executed within the ATmega328 microcontroller. XOR is a fundamental operation commonly used in many embedded systems for cryptographic transformations, checksums, conditional logic, and bit manipulation tasks. As such, its reliability is crucial in maintaining the functional correctness and security posture of a

system.

The experiment aimed to determine whether the newly developed glitch injection device could influence the integrity of XOR operation results by applying glitches at the moment of execution. The setup was configured such that the ATmega328 would trigger a digital signal prior to executing an XOR instruction, allowing the glitching device to synchronize the clock glitch with this specific instruction.

During this test, two 8-bit values were XORed repeatedly, and the result was observed under glitch conditions. The key finding from this experiment was a consistent yet asymmetrical fault behavior: the glitch was able to flip the least significant bit (LSB) of the XOR result from 1 to 0, but not the other way around (it could not flip 0 to 1). This asymmetry suggests that the glitch induces a fault at a very low hardware level, likely during the write-back or propagation of logic values in the processor’s register file. One plausible explanation is that the glitch introduces a timing violation that causes the logic high state (1) to decay or be misread as a logic low (0).

To quantify the fault occurrence, the experiment was repeated across 10 independent runs, with each run consisting of 100 iterations of the XOR operation. The number of successful LSB faults was recorded in each run, and the corresponding probabilities were calculated. The observed fault rates are provided in Table 3.

These results demonstrate a relatively consistent fault rate, fluctuating only slightly between 34% and 39% across trials. When aggregated, the overall average fault probability for the LSB flip from 1 to 0 was calculated to be

Attempt(100 turns)	Probability
1	36
2	34
3	36
4	36
5	37
6	39
7	38
8	39
9	36
10	36

Table 3: Probability of faulty result in each Attempt: XOR

36.7%. This percentage provides a statistical indication of the reliability and reproducibility of this fault injection method. While not deterministic, the glitching method is highly repeatable and probabilistically effective, making it a viable tool for targeted fault attacks, especially when paired with timing-based retry or brute-force strategies.

The fact that the glitch only flips from 1 to 0 suggests that any security-critical logic relying on XOR operations—such as authentication routines, checksum validation, or even simple flag checks—could be vulnerable under certain fault conditions. For instance, a condition like: `if ((key^mask) == expected_value)` could evaluate incorrectly if the result’s LSB is altered, possibly allowing unauthorized access or bypassing critical checks.

4.6.3 Bit-wise OR Operation

```
uint8_t data1 = 0x5A;
uint8_t data2 = 0xC3;

void loop(){
    triggerFunc(true); // Trigger the signal
    delayMicroseconds(10);
    // Critical section begin
    uint8_t result = data1 | data2; // OR operation
    // Critical section end
    triggerFunc(false); // Turning trigger signal off
}
```

Listing 4: Targeted C Code With OR Operation

Building on the earlier success of fault injection in XOR operations, a follow-up experiment was conducted to analyze the effects of instruction-level clock glitching on another common bitwise operation: the logical OR. This instruction, like XOR, is frequently used in embedded systems for control logic, masking, and configuration settings. Evaluating the susceptibility of OR instructions to glitching is essential to understanding the broader impact of clock manipulation on general-purpose computing in microcontrollers.

In this test, the OR operation was performed between two 8-bit values in such a way that the least significant bit (LSB) of the result would be deterministically 1 in normal operation. The objective was to observe whether glitching the clock at the precise moment of OR execution would lead to a corruption in the LSB. Consistent with findings in the XOR experiment, the results revealed an asymmetrical fault pattern: the glitch could flip the LSB

from 1 to 0, but not from 0 to 1. This strongly suggests that the fault is induced during signal propagation or register write-back stages, where timing violations caused by clock distortion result in the logic high (1) value not being properly latched, while zero values remain stable.

To statistically evaluate the reliability and repeatability of this glitching technique on OR operations, 100 iterations were performed across 10 separate attempts. The glitch frequency and timing window remained consistent throughout, optimized based on earlier experimentation. The following are the measured probabilities of the LSB being flipped from 1 to 0 in each batch of 100 OR operations. The observed fault rates are provided in Table 4.

Attempt(100 turns)	Probability
1	36
2	37
3	37
4	36
5	36
6	36
7	36
8	39
9	39
10	36

Table 4: Probability of faulty result in each Attempt: OR

The resulting average probability of fault occurrence was calculated to be 36.8%, indicating a highly consistent yet non-deterministic behavior. This

frequency of success places the OR instruction within the same vulnerability range as the previously tested XOR instruction, further validating the effectiveness of the glitch injection method.

From a security and correctness perspective, the implications of this finding are profound. OR operations are frequently used to set flags, enable bits, or combine control values. A fault that consistently flips 1 to 0 can undermine these operations and lead to unintended consequences such as disabled interrupts, broken configurations, or bypassed logical conditions. For example:

```
status |= 0x01; // Ensure the lowest bit is set
```

Under normal execution, this line ensures that the LSB of status is always 1. However, if glitching occurs and causes the result to be stored as with $\text{LSB} = 0$, the condition meant to be guaranteed fails silently, potentially compromising the control flow or system behavior.

4.6.4 Bit-wise AND Operation

```
uint8_t data1 = 0x5A;
uint8_t data2 = 0xC3;

void loop(){
    triggerFunc(true); // Trigger the signal
    delayMicroseconds(10);
    // Critical section begin
    uint8_t result = data1 & data2; // OR operation
    // Critical section end
    triggerFunc(false); // Turning trigger signal off
```

```
}
```

Listing 5: Targeted C Code With AND Operation

Following the systematic exploration of fault injection effects on bit-wise XOR and OR operations, a third phase of experiments was conducted to investigate the susceptibility of the bit-wise AND operation to instruction-level clock glitching. Bit-wise AND is one of the most fundamental operations in embedded programming, widely used for masking operations, enforcing flags, range-checking, and low-level hardware manipulation. Due to its prevalence in control logic and critical decision-making routines, any vulnerability in this operation under fault conditions could have serious implications for program reliability and security.

In this experiment, the ESP32-based glitch injection circuit—designed for high-precision fault insertion—was again utilized. The ATmega328 microcontroller was configured to output a digital signal prior to executing a specific AND instruction. This trigger was received by the ESP32, which then injected a clock glitch precisely during the instruction execution window. The goal was to analyze whether the glitch could affect the least significant bit (LSB) of the result computed by the AND operation.

To ensure consistent testing conditions, operands were selected such that the AND operation would always produce an LSB of 1 under normal circumstances (e.g., $0x5A \ \& \ 0xC3 = 0x42$). The results showed a consistent fault pattern similar to previous experiments: the glitch could change the LSB of the result from 1 to 0, but never from 0 to 1. This asymmetry strongly suggests that the glitch does not interfere with the computation logic of the

operation itself, but rather with the post-processing stages—most likely during the write-back or propagation of the result into the register file. These timing faults are sufficient to cause incorrect values to be captured or stored, particularly affecting logic highs (1), which are more vulnerable to degradation in short or unstable clock pulses.

To quantify the reliability and frequency of this induced fault, the experiment was run ten times, with each batch consisting of 100 executions of the targeted AND operation under glitching conditions. The probabilities of the LSB being flipped from 1 to 0 in each attempt are stated in the Table 5.

Attempt(100 turns)	Probability
1	35
2	36
3	36
4	38
5	38
6	39
7	38
8	38
9	39
10	39

Table 5: Probability of faulty result in each Attempt: AND

These results indicate an increasing consistency in the glitching mechanism’s ability to induce faults. The average probability of the LSB being flipped from 1 to 0 across all trials was calculated to be 37.6%, slightly

higher than the probabilities observed in XOR (36.7%) and OR (36.8%) experiments. This steady increase might be attributed to the fact that AND operations often result in tighter logic paths with less signal margin, making them more susceptible to high-frequency clock distortions.

From a practical perspective, this vulnerability has serious implications. Consider the following example in embedded C:

```
status = 0x01; // Clear all bits except the LSB
```

This operation is typically used to preserve or isolate a single control bit, such as a user input or a hardware flag. If a glitch flips the result's LSB from 1 to 0, the logic relying on this condition may falsely interpret that the bit was unset. This could disable critical subsystems, trigger error handling unnecessarily, or prevent expected actions from being taken—especially in systems without fault detection or redundancy.

Furthermore, AND operations are often used to restrict access or filter permissions. For example:

```
if ((user_flags & ACCESS_MASK) == REQUIRED_FLAG)
```

A glitch that alters the LSB of such a condition could mislead the system into thinking access is not granted—even when it should be—resulting in denied service, logic errors, or unintentional bypasses of valid states.

4.6.5 Bit-wise NOT Operation

To complete the analysis of basic bit-wise operations under fault injection, the bitwise NOT operation was also subjected to precise clock glitching experiments using the ESP32-controlled glitching circuit. Bit-wise NOT, often

represented as the complement operation (`~` in C/C++), is a unary operator that inverts all bits of an 8-bit operand. It plays a vital role in creating bit masks, toggling states, and implementing low-level logic in embedded systems. Due to its widespread use, it was critical to examine whether this operation exhibits any vulnerability when subjected to glitch-induced timing faults.

The experiment followed the same methodological framework used for XOR, OR, and AND operations.

Test cases were selected such that the expected output of the NOT operation would have a well-defined and testable LSB. For instance, inverting `0xFE` should yield `0x01`, ensuring the LSB of the output is 1. Similarly, inverting `0xFF` should yield `0x00`, allowing assessment of whether the glitch could flip the result in either direction.

Surprisingly, no faults were observed in any of the test cases, regardless of the operand or the targeted glitch parameters. The NOT operation consistently produced the correct result, even under repeated and aggressive glitching attempts. Over the course of ten separate trials, each consisting of 100 NOT operations, not a single instance of LSB corruption or output deviation was recorded. This suggests that the bit-wise NOT instruction in the ATmega328 is either inherently more robust to clock faults or is executed within an internal micro-architectural stage that is less sensitive to timing anomalies.

Several technical factors may contribute to this resilience:

1. Unary Simplicity: Unlike binary operations (AND, OR, XOR), which

involve multiple operands and potential carry/propagation paths, the NOT operation deals with a single operand and may have a shorter or more direct execution path in the microcontroller’s arithmetic logic unit (ALU).

2. Execution Speed: The NOT instruction might be executed in fewer cycles or with less dependency on pipeline stages, reducing the window of opportunity for a glitch to cause a timing violation.
3. Instruction Timing Alignment: It is possible that the NOT instruction’s execution timing did not align well with the injected glitch pulse, making it less likely for the clock distortion to coincide with a critical logic transition.

These observations underscore the non-uniform impact of clock glitching across different instruction types. While XOR, OR, and AND operations showed measurable and repeatable vulnerabilities in their least significant bit outcomes, the NOT instruction remained entirely unaffected under identical glitching conditions. This indicates that instruction-level susceptibility to fault injection varies based on operation complexity, timing, and micro-architectural implementation.

From a security and reliability perspective, this result is encouraging. It demonstrates that not all operations are equally vulnerable to glitching attacks and that some instructions—particularly unary operations like NOT, may inherently possess greater resistance to such forms of fault injection. However, this also highlights a potential vector for attackers to selectively

target more vulnerable operations, focusing their efforts on specific instructions that exhibit asymmetric or frequent fault behavior.

4.6.6 Addition & Subtraction Operations

To extend the scope of instruction-level fault injection analysis beyond logical bit-wise operations, further experiments were carried out on arithmetic operations, specifically addition and subtraction. Arithmetic instructions are deeply integrated into the control flow and logic of virtually every embedded system application, from sensor readings to timing calculations, encryption routines, and more. Any vulnerability in the correctness of arithmetic results caused by clock glitching can pose severe risks to system reliability, data integrity, and security mechanisms.

As in previous instruction-targeted glitching experiments, a glitching circuit built with an ESP32 was used to inject noisy clock faults into a target ATmega328 microcontroller. The glitch injection was triggered precisely at the point where addition or subtraction instructions were executed, using a digital output signal from the MCU. Each experiment consisted of 100 executions of either an addition or subtraction operation under glitching conditions. The operands were systematically selected to generate results with varying Hamming weights—that is, the number of binary 1s in the 8-bit result. Hamming weight was chosen as a parameter to examine whether the number of logic high bits in the result had any correlation with susceptibility to glitch-induced faults. Note that the operands have been selected in an order that it does not overflow the result over the 8 bits.

The Table 6 demonstrates the results those were obtained from repeated tests across a full spectrum of Hamming weights in the result of the addition operations.

Original Hamming Weight of The Result	Probability of Failure (%)
0	16 - 17
1	19 - 23
2	26 - 29
3	34 - 37
4	38
5	38
6	41 - 42
7	46 - 49
8	52 - 57

Table 6: Probability of faulty result according to the hamming weight: ADDITION

The results clearly exhibit a positive correlation between Hamming weight and the probability of fault. Results with a lower number of 1s were less likely to be corrupted by glitching, while higher Hamming weights—especially 6 to 8—demonstrated significantly higher fault probabilities. For instance, when the result had a Hamming weight of 8 (i.e., all bits set to 1, such as 0xFF), the failure probability peaked between 52% and 57%. This suggests that glitching is more effective when the target instruction is expected to produce multiple high logic levels, likely due to increased switching activity and power consumption within the microcontroller’s arithmetic logic unit

(ALU), making it more susceptible to timing faults.

A parallel set of experiments was conducted on the subtraction operation under identical glitching conditions. The results, organized by Hamming weight of the result, stated in Table 7

Original Hamming Weight of The Result	Probability of Failure (%)
0	16 - 17
1	19 - 22
2	26 - 29
3	33 - 37
4	38
5	38
6	40 - 43
7	46 - 49
8	53 - 57

Table 7: Probability of faulty result according to the hamming weight: SUBTRACTION

As observed in the addition experiment, the subtraction results also follow a similar trend of increasing failure probability with increasing Hamming weight. While the lower weight results (e.g., Hamming weights 0–2) saw failure rates of less than 30%, results with high bit density (weights 6–8) reached failure probabilities up to 57%. The most vulnerable results were again those with Hamming weight 8, showing a remarkably high error rate ranging between 53% and 57%.

These results provide strong evidence that the bit-level structure of an

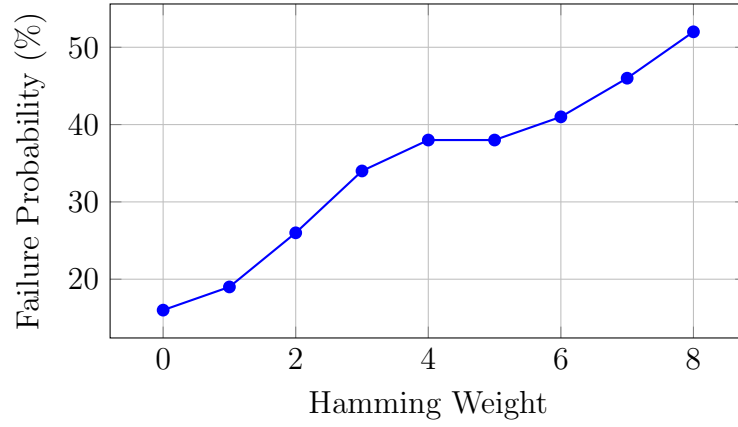


Figure 15: Relationship between minimum failure probability and the actual hamming weight of the result: Addition

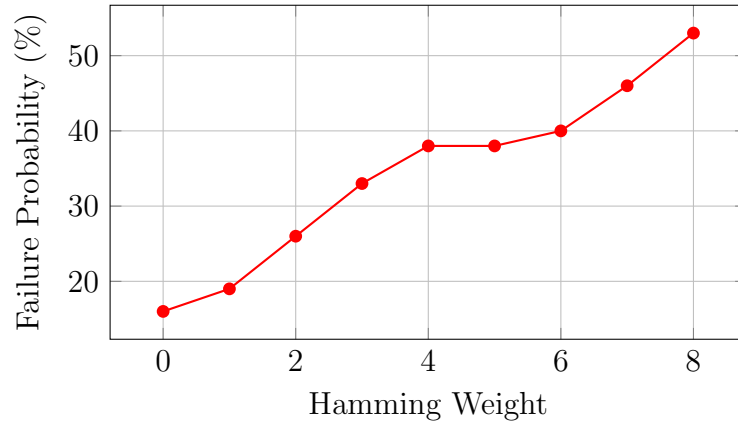


Figure 16: Relationship between minimum failure probability and the actual hamming weight of the result: Subtraction

arithmetic result directly influences the susceptibility of the operation to clock glitching. The correlation between higher Hamming weights and greater failure probability indicates that the cumulative effect of multiple 1s in the result increases the chance of propagation delays or logic instability during

the write-back phase of the instruction cycle. Since clock glitching shortens or misaligns the clock pulse during execution, it is likely to interfere with critical timing windows, particularly in cases of heavy signal switching as in high-Hamming-weight outputs.

The data also revealed variations in the probability range for certain Hamming weights (e.g., weights 1, 2, 3, 6, 7, and 8). To better understand this variability, additional tests were conducted to identify minimum and maximum bounds of fault probability for each case. These supplementary experiments confirmed that even within a given Hamming weight, slight differences in operand values, glitch pulse alignment, and system state could influence the probability of fault, underscoring the probabilistic and non-deterministic nature of fault injection at this level.

5 Discussion

The experimental results provide strong evidence that controlled clock glitching can systematically alter UART data transmission, leading to severe security vulnerabilities. The findings demonstrate two critical exploit scenarios:

1. Flipping a false Signal to a true Signal (Unauthorized Access):
 - Under normal conditions, when the ATmega328 microcontroller sends a false signal to the door lock, it is transmitted as 0x00 over UART.
 - However, at specific glitching frequencies (17–19 MHz, 20–22 MHz, and 23 MHz), the received value changes from 0x00 to progres-

sively corrupted values like 0x80, 0xC0, and 0xE0, indicating that bits within the byte are being flipped.

- This means that a carefully timed glitch could potentially modify a false signal into a value that is incorrectly interpreted as true, allowing unauthorized access to a secured system.
- Another critical finding is that at very high glitching frequencies (24–25 MHz), the transmission fails entirely, suggesting that excessive interference can break synchronization altogether, making the system unstable and unreliable.

2. Flipping a true Signal to a false Signal (Denial of Access):

- According to Table 2, when a true signal (originally represented as 0x01) is transmitted, clock glitching at specific frequencies disrupts the transmission.
- A key observation is that when the glitching clock is 1.86–1.89 MHz, the received value becomes 0x00, meaning that the original true signal has been fully inverted to false.
- This means an attacker could strategically apply a glitch at this frequency to prevent legitimate users from gaining access to a secured system by flipping authentication approvals into denials.
- Moreover, other glitch frequencies result in various corrupted values (0x03, 0x02, 0x06, etc.), suggesting that while full inversion occurs at 1.87 MHz, partial corruption could also cause unpredictable behavior in the system.

The initial hypothesis suggested that by inducing clock glitches during UART communication between an ATmega328 microcontroller and a door lock system, it might be possible to manipulate transmitted authentication signals—specifically flipping a false signal to true—thereby granting unauthorized access. Through systematic experimentation, this hypothesis has been confirmed as true. By applying a clock glitch at a frequency above 17 MHz, the false signal, originally intended to deny access, is corrupted and received as true at the door lock, effectively unlocking it without valid credentials. Conversely, when the glitching clock is set to 1.87 MHz, the corruption occurs in the opposite manner, flipping a true signal into false, potentially locking out legitimate users. These findings provide concrete evidence that clock glitching can successfully interfere with UART data integrity, breaking the fundamental trust in the microcontroller’s decision-making process. The observed behavior aligns with known vulnerabilities of asynchronous communication protocols like UART, where disruptions in timing can miss-align data frames, leading to unintended bit flips. As a result, the hypothesis is no longer theoretical but an established fact—unauthorized access through clock glitching is indeed achievable. This confirmation emphasizes the need for security enhancements, such as stronger data integrity verification methods, glitch-resistant hardware designs, or alternative communication protocols that mitigate such attacks. This confirms that synchronization failures in UART communication, induced via clock glitching, can lead to security breaches.

Furthermore, the latter experiments prove that well-controlled clock glitch

can induce faults and alter the 8bit data. That represents the vulnerability of data types not only Boolean but also Integer and ASCII characters under clock glitch injection. The above experiments also illustrate the practical aspect of applying clock glitch in order to alter the data bits in a predictable and reproducible manner. As an example if we want to change the '0000 0010' data to '1100 0010' when they are transmitting through UART communication, we can simply apply a glitchy clock signal between 20-22 MHz into the sending device.

Following the preliminary fault injection experiments that revealed the vulnerabilities of the ATmega328P microcontroller to low-precision clock glitching, this research took a major step forward by designing and implementing a highly targeted fault injection mechanism. This new approach leverages an ESP32-based external glitching device to inject carefully timed glitches in synchronization with specific instruction executions of the target microcontroller. The transition from general system-level disruption (as observed in UART faults) to precise instruction-level attacks enabled a much deeper analysis of program control flow behavior and fault impact.

Unlike the earlier method—where clock glitches were injected continuously or blindly throughout UART communication—this setup is trigger-based and deterministic. The ATmega328P was programmed to emit a digital signal through one of its GPIO pins just before executing the instruction under test. This digital signal served as an external interrupt for the ESP32, which was pre-programmed to immediately generate a short-duration clock glitch and feed it to the ATmega328P. This method ensures the glitch is

aligned precisely with the instruction of interest, minimizing collateral effects and isolating the fault to a single operation.

This method was rigorously tested across a variety of bitwise and arithmetic operations:

- Bitwise XOR, OR, and AND: In each of these logic operations, it was observed that glitching caused a consistent fault pattern where the Least Significant Bit (LSB) of the result was flipped from 1 to 0, but not from 0 to 1. This asymmetry suggests that glitches tend to cause logic gate malfunctions in a single direction, possibly due to timing violations in the propagation of high-state logic. The fault was probabilistic, with failure rates hovering between 36%–38% across trials, depending on the operation.
- Bitwise NOT: Interestingly, this operation was completely immune to glitching in all 100 tests. This might indicate that the NOT instruction in AVR assembly is either executed too quickly or lacks enough transitional states to be affected by brief glitch pulses.
- Addition and Subtraction: These arithmetic operations exhibited a clear correlation between hamming weight and fault probability. As the number of 1s in the result increased, so did the failure rate, peaking at over 57% for addition and 55% for subtraction. This suggests that more logic transitions (i.e., more switching bits) increase the susceptibility to timing faults, likely because higher logic density places greater strain on internal propagation delays and setup/hold times during execution.

One of the most significant outcomes of this experiment is the demonstrated control over fault injection timing, which is a key milestone in transitioning from observational fault research to strategic exploitation. By aligning glitch pulses with specific instruction boundaries, the attacker can intentionally influence the result of targeted operations. This is a considerable advancement over the low-precision UART attack, where the glitch could only probabilistically flip a bit during communication but had no awareness of internal instruction execution states.

This newfound control implies that timing fault attacks are not just a side effect of electrical instability but a repeatable and engineerable threat. The experiment validates the potential for developing an automated glitching system that can learn and adapt based on target behavior, paving the way for programmable fault-injection engines used in advanced reverse engineering or security research.

In detailed discussion with critical evaluation within the context of research questions are presented in the following subsections.

5.1 Research Question 1

To what extent do timing fault attacks disrupt the program execution sequence of the ATmega328P microcontroller?

The first research question focuses on evaluating the vulnerability of the ATmega328P microcontroller to timing fault attacks and the extent of disruption they cause to the program control flow and execution correctness. The extensive set of experiments, ranging from system-level UART commu-

nication to low-level arithmetic and logic operations, collectively paints a comprehensive picture of how significantly program behavior can be altered under fault conditions.

5.1.1 UART-Level Fault Injection (Low Precision Clock Glitching)

The initial experiments using clock glitching during UART communication revealed that fault injection can manipulate the boolean data transmitted between two devices. Specifically:

- When the transmitted value was FALSE (0x00), and the glitching clock was increased above 17 MHz, the receiving end erroneously interpreted it as TRUE (0x01).
- When TRUE (0x01) was transmitted and the glitching clock was slowed down to 1.87 MHz, the receiver read it as FALSE (0x00).

This effectively demonstrated that timing faults at the communication level could bypass authentication mechanisms or binary decisions (e.g., unlocking a door). This indicates a disruption in program behavior at the system level, since the microcontroller's logic assumes a correct communication medium while it's unknowingly altered by timing faults.

Additionally, this effect was proven not limited to boolean values. The glitching resulted in predictable bit-level changes in integer values and ASCII characters, most often affecting the Least Significant Bit (LSB). For instance:

- Integer values like 0x04 were received as 0x05 or vice versa.

- ASCII characters like 'A' (0x41) were received incorrectly due to altered LSBs.

These results indicate fundamental disruptions in data integrity, showcasing that not only program control flow but also data correctness is compromised, which can lead to cascading failures or logic branches being incorrectly triggered.

5.1.2 Instruction-Level Fault Injection (Precise Glitching using ESP32)

To overcome the non-specific and system-wide nature of the UART glitching method, the study developed a more precise method using an ESP32, capable of injecting faults at specific instruction boundaries by reacting to triggers generated from the target ATmega328. This significantly improved the control and reproducibility of timing fault attacks.

In this controlled environment, various bitwise and arithmetic operations were studied. The results from these operations directly correlate with how execution correctness at the instruction level can be altered:

- XOR, OR, AND Operations: All three operations exhibited the same bit flip behavior, where the LSB could be flipped from $1 \rightarrow 0$, but not from $0 \rightarrow 1$. These flips did not occur every time, but with a measurable probability:
 - XOR: $\sim 36.7\%$
 - OR: $\sim 36.8\%$

- AND: ~37.6%

This behavior implies partial but deterministic corruption in logical results, which can affect conditional branches, flags, or logical masking within the program flow.

- NOT Operation: Contrary to the above, the NOT operation was completely resilient to clock glitching under the same conditions. This suggests that not all instructions are equally vulnerable, likely due to differences in micro-architectural implementation (unary vs binary logic).
- Arithmetic (ADD and SUB): These operations demonstrated data-dependent fault behavior, with higher failure probabilities associated with higher Hamming weights in the result:
 - Addition: Up to 57% failure at Hamming weight 8
 - Subtraction: Up to 55% failure at Hamming weight 8

This confirms a clear relationship between switching complexity and glitch susceptibility, indicating that the more bits set to 1, the greater the chance the operation result is corrupted. Since arithmetic outcomes often determine control flow (e.g., counters, pointer calculations, loop limits), this directly contributes to unintended jumps, logic errors, or buffer overruns.

5.2 Research Question 2

How to conduct timing fault attacks to generate predictable program execution behaviors on ATmega328P microcontrollers?

The second research question investigates how fault attacks can be carried out in a way that produces repeatable and controllable effects making it a crucial criterion for real-world exploits.

5.2.1 UART-Level Fault Injection (Low Precision Clock Glitching)

Controlled Bit Manipulation in UART Communication:

One of the most revealing findings of this work is the realization that UART data packets can be deliberately corrupted in a predictable manner using clock glitching, even without targeting specific instructions. UART communication, by nature, relies heavily on synchronization between the transmitter and receiver based on a fixed baud rate. Introducing controlled glitches at this level affects the sampling of individual bits, which in turn can flip the value of the received byte.

In our experiments, the ATmega328P transmitted boolean and integer values over UART, which were intercepted and faulted by clock glitches introduced during transmission. The findings showed that:

- At frequencies above 17 MHz, transmitted values such as FALSE (0x00) were consistently received as TRUE (0x01)
- At frequencies around 1.87 MHz, transmitted values like TRUE (0x01)

were received as FALSE (0x00)‘

This pattern was not random. The injection of glitch pulses during UART bit transmission had a bit-specific effect, particularly on the Least Significant Bit (LSB). A glitch could change the LSB from 0 to 1 or from 1 to 0, depending on the frequency and timing of the fault. This opened the door to controlled bit flipping in UART communication, extending the attack’s utility beyond booleans to 8-bit integers and ASCII characters.

For example:

- Sending 0x02 (binary 0000 0010) with a well-timed glitch could be received as 0x03 (0000 0011)
- Sending 'A' (ASCII 0x41, 0100 0001) could be faulted and received as '@' (0x40, 0100 0000)

If we change the perspective from implemented data types to hardware level 8-bit data packet, the above described results demonstrate the capability of flipping the bits in the LSB end. This behavior provides a level of bit-level predictability and controllability, enabling the capability to craft data-level payload manipulation over serial interfaces using only timing faults without modifying the program code or physical memory directly providing solid evidence to answer the 2nd research question.

5.2.2 Precised Instruction-Level Glitching

The employed method with additional glitching circuit introduces an accurate method to inject high precised glitches for specific instructions with

demonstrated effects on the program execution within ATmega328 micro-controller. This method provided the following advantages.

- Capability to triggering fault injection on specific instructions.
- Enabled studying instruction-level vulnerabilities.
- Allowed analysis of repeated patterns and fault probabilities.
- Produced controlled and statistically predictable errors.

By integrating the ESP32-triggered glitch injection with signal timing from the ATmega328P, this method introduces programmable precision, which dramatically increases the reliability and repeatability of fault attacks. This setup transforms what is usually seen as random behavior into an experimentally verifiable and manipulable phenomenon. This evolution in methodology fulfills the requirement for controlled fault injection, enabling researchers and adversaries alike to reproduce specific errors with known probability distributions.

For example: Knowing that XOR fails 36.7% of the time, or that an addition result with Hamming weight 8 fails 57% of the time, allows probabilistic modeling of outcomes and potential for intentional glitch exploitation.

Additionally, the directionality of faults (e.g., $1 \rightarrow 0$ but not $0 \rightarrow 1$) further adds to the predictability and reduction in search space for successful attack vectors.

The integration of controlled glitch frequency in UART-level system faults and noisy disruptions in instruction-level injection demonstrates that timing

fault attacks can be effectively controlled and replicated to produce targeted, bit-specific, and instruction-specific behavior changes in the ATmega328P microcontroller. This level of control satisfies the criteria outlined in the second research question, proving that predictable program execution deviations are not only possible but also engineerable through precise glitch injection methods.

6 Conclusion

Through a series of meticulously designed experiments and fault injection methodologies, this research has demonstrated that timing fault attacks can significantly disrupt the program execution behavior of the ATmega328P microcontroller, both at the system-level communication protocols and at the instruction-level computation logic. The experiments revealed that by carefully introducing glitches either during data transmission (UART-based) or during specific instruction execution (logic and arithmetic operations), the normal behavior of the program can be altered in repeatable and measurable ways.

In the case of UART communication, low-precision clock glitching was able to modify transmitted boolean values, integers, and ASCII characters by altering individual bits in the data packets. It was proven that certain glitch frequencies can systematically flip the Least Significant Bit (LSB) of a byte, converting 0x00 to 0x01 and vice versa. This directly disrupted the logic upon which program decisions are made. For example, allowing unauthorized access to a door lock system based on faulty logic generated by timing-based

interference.

Further, transitioning to a higher-precision, instruction-targeted fault injection setup using an ESP32-based circuit enabled this study to explore the depth of vulnerability within the ATmega328P’s instruction pipeline. The results from XOR, OR, AND, ADD, and SUB operations consistently showed that clock glitches could flip the LSB of computation results from 1 to 0, although not vice versa. Importantly, these disruptions did not occur randomly but followed a statistically observable pattern, with overall fault probabilities averaging around 36%–38% for logic operations and climbing up to 57% for arithmetic operations involving higher hamming weights.

Notably, certain instructions such as the bitwise NOT operation remained resilient against glitching, indicating that while the ATmega328P is vulnerable to timing faults, not all parts of the instruction set or execution paths are equally susceptible. This finding further highlights the importance of instruction-level understanding in analyzing microcontroller vulnerabilities.

Taken together, the extent of disruption caused by timing fault attacks on ATmega328P is substantial. They can interfere with both data integrity and control flow logic, leading to unintended execution paths, incorrect computations, and security breaches. These attacks are achievable using affordable hardware setups and can exploit common communication protocols and computational operations. As a result, this research underscores the need for more robust fault-tolerant design practices and hardware-level protection mechanisms in embedded systems based on the ATmega328P microcontroller.

What distinguishes this work is the ability to reproduce these faults in a controlled and targeted fashion. By correlating glitch frequency and timing with predictable outcomes, this research not only confirms the extent of disruption (Research Question 01) but also lays the groundwork for developing fault injection strategies that yield consistent, predictable behavior modifications (Research Question 02). This capacity for precision fault control elevates timing fault attacks from being seen as random anomalies to becoming engineerable, repeatable threats, with broad implications in embedded system security.

In conclusion, the ATmega328P microcontroller is demonstrably vulnerable to timing fault attacks that can compromise both data and control flow integrity. Moreover, with the appropriate fault injection techniques, such attacks can be made predictable and systematic, thus fulfilling both the exploratory and methodological objectives outlined in this thesis. These insights make a significant contribution to understanding fault models in embedded systems, highlighting both risks and potential countermeasures. Your thesis not only proves vulnerabilities, but also sets a foundation for developing robust software-hardware co-design strategies in fault-sensitive applications.

6.1 Future Directions

The findings of this research not only confirm the susceptibility of the ATmega328P microcontroller to timing fault attacks but also open a variety of avenues for deeper exploration in the domain of embedded systems security.

While this study has successfully demonstrated the capability to induce and control timing faults at both the communication and instruction levels, there remain several compelling future directions to further this line of research:

- Improved clock glitch injection device:

The precise clock glitch injection device implemented using ESP32 introduced faults and affected instruction execution. However, this method can only generate random noise. We don't have precise control over the properties of the clock glitch signal. This leads to a major limitation when understanding and analyzing the relationship between the results and the generated glitching clock properties. Due to this factor, future researches can focus on improving this method by adding the controllability of the glitching signal and analyze the relationship between the glitching properties and the observations or even can open a window for precised method with better reproducibility and better controllability.

- Cross-Architecture Analysis Although the experiments focused on the ATmega328P, expanding this methodology to other microcontroller families such as:

- ARM Cortex-M based microcontrollers (STM32, NRF52)
- PIC and MSP430 series
- RISC-V based architectures

Would validate whether the observed vulnerabilities are architecture-

specific or generalizable across platforms. This also supports the development of cross-platform fault models and countermeasures

- Security Countermeasures and Mitigation Techniques As fault attacks become more precise and repeatable, future work should also investigate software and hardware-level protections including:
 - Instruction duplication and checksumming
 - Redundant computation with majority voting
 - Clock and power glitch detectors
 - Secure bootloaders with integrity checks

Implementing and evaluating these countermeasures on the same experimental setup could provide valuable insight into their effectiveness and potential overhead.

References

- Banerjee, U., Ho, L. & Koppula, S. (2022), ‘Power-based side-channel attack for aes key extraction on the atmega328 microcontroller’, *arXiv preprint arXiv:2203.08220* .
- Bonny, T. & Nasir, Q. (2019), ‘Clock glitch fault injection attack on an fpga-based non-autonomous chaotic oscillator’, *Nonlinear Dynamics* **96**, 2087–2101.
- Breier, J. & Hou, X. (2022), ‘How practical are fault injection attacks, really?’, *IEEE Access* **10**, 113122–113130.
- Colombier, B., Grandamme, P., Vernay, J., Chanavat, É., Bossuet, L., de Laulanié, L. & Chassagne, B. (2021), Multi-spot laser fault injection setup: New possibilities for fault injection attacks, *in* ‘International Conference on Smart Card Research and Advanced Applications’, Springer, pp. 151–166.
- Corporation, A. (2016), ‘Atmega328/p: 8-bit avr microcontroller with 32k bytes in-system programmable flash’. Data Sheet.
URL: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_datasheet.pdf
- Delarea, S. & Oren, Y. (2022), ‘Practical, low-cost fault injection attacks on personal smart devices’, *Applied Sciences* **12**(1), 417.
- Dumont, M., Lisart, M. & Maurine, P. (2019), Electromagnetic fault injec-

- tion: How faults occur, *in* ‘2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)’, IEEE, pp. 9–16.
- Gnad, D. R., Oboril, F. & Tahoori, M. B. (2017), Voltage drop-based fault attacks on fpgas using valid bitstreams, *in* ‘2017 27th International Conference on Field Programmable Logic and Applications (FPL)’, IEEE, pp. 1–7.
- He, W., Breier, J., Bhasin, S., Jap, D., Ong, H. G. & Gan, C. L. (2016), Comprehensive laser sensitivity profiling and data register bit-flips for cryptographic fault attacks in 65 nm fpga, *in* ‘Security, Privacy, and Applied Cryptography Engineering: 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings 6’, Springer, pp. 47–65.
- O’Flynn, C. (2016), ‘Fault injection using crowbars on embedded systems’, *Cryptology ePrint Archive* .
- Ordas, S., Guillaume-Sage, L. & Maurine, P. (2017), ‘Electromagnetic fault injection: the curse of flip-flops’, *Journal of Cryptographic Engineering* **7**, 183–197.
- Sanjaya, S., Jayasena, A. & Mishra, P. (2024), ‘Information leakage through physical layer supply voltage coupling vulnerability’, *arXiv preprint arXiv:2403.08132* .
- Tehranipoor, M., Nalla Anandakumar, N. & Farahmandi, F. (2023), Volt-

age glitch attack on an fpga aes implementation, *in* ‘Hardware Security Training, Hands-on!’, Springer, pp. 219–234.

Vasselle, A., Thiebeauld, H., Maouhoub, Q., Morisset, A. & Ermeneux, S. (2017), Laser-induced fault injection on smartphone bypassing the secure boot, *in* ‘2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)’, IEEE, pp. 41–48.