Assembly Level Clock Glitch Insertion Into An XMega MCU

Nigamantha Gopala Chakravarthi

Follow this and additional works at: https://engagedscholarship.csuohio.edu/etdarchive

Part of the Electrical and Computer Engineering Commons

How does access to this work benefit you? Let us know!

Recommended Citation
https://engagedscholarship.csuohio.edu/etdarchive/917

This Thesis is brought to you for free and open access by EngagedScholarship@CSU. It has been accepted for inclusion in ETD Archive by an authorized administrator of EngagedScholarship@CSU. For more information, please contact library.es@csuohio.edu.
ASSEMBLY LEVEL CLOCK GLITCH INSERTION INTO AN XMEGA MCU

NIGAMANTHA GOPALA CHAKRAVARTHI

Bachelor of Engineering in Electronics and Communications Engineering
Anna University, Chennai
May 2014

Submitted in partial fulfillment of requirements for the degree
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
at the
CLEVELAND STATE UNIVERSITY
August 2016
We hereby approve this thesis for

Nigamantha Gopala Chakravarthi

Candidate for the Master of Science in Electrical Engineering degree for the

Department of Electrical Engineering and Computer Science

and the CLEVELAND STATE UNIVERSITY

College of Graduate Studies by

__________________________________
Thesis Chairperson, Dr. Chansu Yu

__________________________________
Thesis Committee Member, Dr. Nigamanth Sridhar

__________________________________
Thesis Committee Member, Dr. Haodong Wang

Student’s Date of Defense (08/04/2016)
Dedicated to Atthi thatha
ACKNOWLEDGEMENT

I would like to express my gratitude to all those who gave me their support to complete this thesis. First of all, I owe my sincere thanks to my supervisor and department Chairperson Dr. Chansu Yu, whose constant guidance and valuable suggestions helped me during my research work, experimental set-up and writing of this thesis. This is my first time working on research work, hence I sincerely thank him for his patience during my learning process. I would like to thank Santosh Desiraju, alumni at Cleveland State University, who helped me in my initial stages of research. I would also like to thank Abhishek K. Upadhyay, alumni at Cleveland State University, who helped me with formatting of my thesis writing. I would like to thank Dr. Nigamanth Sridhar and Dr. Haodong Wang for serving on my thesis committee. I am grateful to Ms. Melanie Caves and Ms. D Lydia, our office secretaries, whose timely help in coordinating various official activities was essential in expediting my work. I want to acknowledge great help provided by ChipWhisperer technical support team whose timely response over the doubts I had helped me in carrying out my research work successfully.

Finally, I would like to thank my parents, grandparents, relatives and friends for their invaluable encouragement and moral support that helped me in completing my thesis successfully. I would like to dedicate this thesis to my grandfather Mr. N C Seshadri.
ASSEMBLY LEVEL CLOCK GLITCH INSERTION INTO AN XMEGA MCU

NIGAMANTHA GOPALA CHAKRAVARTHI

ABSTRACT

This thesis proposes clock-glitch fault injection technique to inject glitches into the clock signal running in a microcontroller unit and studying its effects on different assembly level instructions. It focuses mainly on the effect of clock glitches over the execution, sub-execution and pre-execution cycles of the test instructions and also finds the delay between the actual position of glitch insertion and the trigger being set for the glitch insertion. The instructions used in this work are provided by Atmel which classifies them according to their type of operation. These instructions are here further grouped depending on the number of clock cycles they require for their execution. Each group of instructions are tested for their behavior towards clock glitches being injected at different places in and surrounding their execution cycle. This thesis utilizes the ChipWhisperer-Lite board (CW1173) for performing the whole experiment by controlling the target device, providing clock as well as clock glitches with appropriate properties at appropriate position to the target device. The Atmel AVR XMEGA 128D4U is used as the target device (CW303) that uses an external clock of frequency 7.37MHz generated by the main board. The Capture software, provided by the ChipWhisperer, is used for establishing the hardware connection between the main board and the target board. The clock glitches are designed and triggered through the Capture software.
TABLE OF CONTENTS

ABSTRACT........................................................................................................................... v

LIST OF TABLES.................................................................................................................. ix

LIST OF FIGURES................................................................................................................ x

CHAPTERS

I. INTRODUCTION................................................................. 1
   1.1. Scope......................................................................................................................... 1
   1.2. Research Question.................................................................................................... 2
   1.3. Proposed Work.......................................................................................................... 3
   1.4. Organization............................................................................................................. 4

II. BACKGROUND OF STUDY................................................................. 5
   2.1. Hardware Attacks and Analyses............................................................................. 6
   2.2. Clock Glitch Attack.................................................................................................. 8
   2.3. Clock Glitch Attack Examples using ChipWhisperer-Capture......................... 10
      2.3.1. Manually triggered glitch................................................................................ 11
      2.3.2. Automatically triggered glitch........................................................................ 12
      2.3.3. Real time example: Password Authentication............................................... 14

III. LITERATURE REVIEW................................................................. 16
   3.1. Works related to General Fault Injection Attacks................................................ 16
3.2. Works related to Clock Glitch Attacks.......................................................... 18

IV. EXPERIMENTAL SETUP AND DESIGN....................................................... 22

4.1. ChipWhisperer-An Overview................................................................. 22

4.1.1. ChipWhisperer-The Software............................................................. 22

4.1.2. ChipWhisperer-The Hardware.......................................................... 23

4.2. XMEGA-The Target Device................................................................. 25

4.2.1. Flash Memory..................................................................................... 27

4.2.2. Data Memory..................................................................................... 27

4.3. Glitch generation and External trigger.................................................... 28

V. IMPLEMENTATION AND PERFORMANCE STUDY.................................... 32

5.1. Clock-Glitch Attack on Single-Cycle Instructions.................................... 33

5.1.1. Trigger Delay...................................................................................... 33

5.1.2. Single-Cycle Instructions................................................................. 36

5.1.2.A. Glitch in Test Instruction Execution Cycle...................................... 37

5.1.2.B. Glitch in Clock Cycle previous to Test Instruction Execution Cycle... 38

5.2. Clock-Glitch Attack on Multi-Cycle Instructions.................................... 39

5.2.1. Two-Clock Cycle Instructions............................................................ 41

5.2.1.A. Glitch in Test Instructions 2nd Execution Cycle.......................... 43

5.2.1.B. Glitch in Test Instruction 1st Execution Cycle.............................. 43

5.2.1.C. Glitch in Clock Cycle previous to Test Instruction Execution Cycle... 43

5.2.2. Three Clock-Cycle Instructions......................................................... 44
5.3. Clock-Glitch Attack on Caesar Cipher Algorithm – Test Case 47

VI. CONCLUSION AND FUTURE WORK 50

6.1. Conclusion 50

6.2. Future work 51

REFERENCES 52

APPENDICES 56

A. Trigger Delay Estimation 57

A. Five NOP Instructions Insertion 58

B. Six NOP Instructions Insertion 61

C. Seven NOP Instructions Insertion 65

B. ChipWhisperer Clock-Glitch Example-Codes 70
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>CW-Lite 20-pin Connector</td>
<td>24</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary of observations made by testing the effect of clock-glitch insertion in the clock cycle executing the single-cycle test instructions and in the clock cycle prior to the test instruction execution</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>Atmel AVR XMega assembler instructions categorized depending on the type of operation and number of execution clock-cycles used</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>Summary of observations made by testing the effect of clock-glitch insertion in the clock cycle executing the double-cycle test instructions and in the clock cycle prior to the test instruction execution</td>
<td>44</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of observations made by testing the effect of clock-glitch insertion in the clock cycle executing the triple-cycle test instructions and in the clock cycle prior to the test instruction execution</td>
<td>46</td>
</tr>
<tr>
<td>A.1</td>
<td>Prediction based on analysis of how single-cycle ALU instructions get affected due to clock glitch at various offsets assuming different trigger delays</td>
<td>64</td>
</tr>
<tr>
<td>A.2</td>
<td>Observed output due to clock glitch insertion</td>
<td>65</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Hardware Attacks-Classification</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Glitch insertion in single-clock cycle in a train of clock pulses</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Clock glitch attack on a target running cryptographic algorithm</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Manually glitched output for glitch_infinite () function</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Glitch Explorer showing externally triggered glitch output for glitch_1()</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>Glitch Explorer showing the output for glitch_3() function</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Glitch generation using High-frequency signal</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Glitch generation using phase-shifted signals</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>ChipWhisperer-Lite Board (CW1173) along with XMEGA target device (CW303)</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Experimental Setup</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>XMEGA D4 Block Diagram</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Clock-glitch generation</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Glitch insertion into Clock</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Routing of trigger to Glitch module</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>Addition operation in assembly immediately after Trigger set</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Addition in assembly after 6 nop instructions</td>
<td>34</td>
</tr>
<tr>
<td>5.3</td>
<td>Parallel instruction fetches and execution in 8-bit Atmel XMEGA</td>
<td>35</td>
</tr>
<tr>
<td>5.4</td>
<td>Single-Cycle ALU execution in 8-bit Atmel XMEGA</td>
<td>35</td>
</tr>
<tr>
<td>5.5</td>
<td>Depiction of 6 nop clock cycles inserted between Trigger set and ALU insertion execution</td>
<td>36</td>
</tr>
<tr>
<td>5.6</td>
<td>Glitch insertion in positive and negative halves of single-cycle test instruction execution cycle and in nop execution cycle that is previous to test instruction execution cycle</td>
<td>37</td>
</tr>
</tbody>
</table>
5.7 Depiction of 6 nop clock cycles inserted between Trigger set and a two-clock cycle test instruction execution

5.8 Glitch insertion in positive and negative halves of double-cycle test instruction 2\textsuperscript{nd} execution cycle, 1\textsuperscript{st} execution cycle and in nop execution cycle that is previous to test instruction execution cycle

5.9 Depiction of 6 nop clock cycles inserted between Trigger set and a three-clock cycle test instruction execution

5.10 Caesar Cipher Encryption Algorithm

A.1 Depiction of 5 nop clock cycles inserted between Trigger set and ALU (inst) execution, and single cycle ALU (inst) execution sub-cycles

A.2 Clock glitch Insertion at negative and positive halves of 5th and 6th clock cycles respectively assuming 5clock cycle trigger delay

A.3 Clock glitch Insertion at negative and positive halves of 6th and 7th clock cycles respectively assuming 6clock cycle trigger delay

A.4 Clock glitch Insertion at negative and positive halves of 7th and 8th clock cycles respectively assuming 7clock cycle trigger delay

A.5 Depiction of 6 nop clock cycles inserted between Trigger set and ALU (inst) execution, and single cycle ALU (inst) execution sub-cycles

A.6 Clock glitch Insertion at negative and positive halves of 5th and 6th clock cycles respectively assuming 5clock cycle trigger delay

A.7 Clock glitch Insertion at negative and positive halves of 6th and 7th clock cycles respectively assuming 6clock cycle trigger delay

A.8 Clock glitch Insertion at negative and positive halves of 7th and 8th clock cycles respectively assuming 7clock cycle trigger delay

A.9 Depiction of 7 nop clock cycles inserted between Trigger set and ALU (inst) execution, and single cycle ALU (inst) execution sub-cycles
A.10 Clock glitch Insertion at negative and positive halves of 5th and 6th clock cycles respectively assuming 5clock cycle trigger delay…………………………………….62

A.11 Clock glitch Insertion at negative and positive halves of 6th and 7th clock cycles respectively assuming 6clock cycle trigger delay…………………………………….63

A.12 Clock glitch Insertion at negative and positive halves of 7th and 8th clock cycles respectively assuming 7clock cycle trigger delay…………………………………….64
CHAPTER I

INTRODUCTION

This chapter gives a brief introduction about the scope, aim and the motivation for the research work being carried out for this thesis. The section 1.1 discusses about the scope and section 1.2 explains the research question behind carrying out this thesis work. The main proposed work has been established in the section 1.3 and the section 1.4 outlines the thesis organization.

1.1 Scope

With the huge development in electronic systems in the recent decades, the electronic devices have acquired more important role in day-to-day life. From the point of entertainment to the point of storing and transferring personal and other secret data for business or other purposes, these electronic gadgets and devices play a huge role. Even though these make life more comfortable and easier, the security of the information to be protected is still under concern. The secret data from these devices, especially from those running cryptographic algorithm, can be acquired by an attacker by performing either software or hardware attack on them. Since the software is built over the hardware, it is very much important to protect the hardware from any vulnerability and must be made secured from any kinds of attacks.
These kinds of attacks are of two types: Side-Channel Attacks and Fault Injection Attacks. The fault injection is a method by which an attacker can retrieve the secret data by making changes in the intrinsic parameters of the hardware or via any other external perturbations and then studying the observed outputs [17]. The internal changes can be induced through voltage or clock glitches. These glitches affect the fetching and execution processes of certain instructions present in the cryptographic algorithm running in a target device depending on the type and position of the injected glitch. The output given by the algorithm then depends on the effect of the glitches over those instructions.

This thesis uses a platform to insert clock glitches in a target device running simple assembly codes and helps in understanding the behavior of various assembly instructions towards clock glitching. These instructions are categorized depending on their behavior and the results can be used for designing more secure cryptographic algorithm.

1.2 Research Question

The challenge posed by this research is to find a method by which the faults are introduced in a target device running simple programs, further by which different assembly instructions can be tested for finding how they are getting affected. This should be done by observing the faulty outputs generated due to the injected faults. The clock glitches are employed to induce faults in the target device. The aim is to find the effect of clock glitches over the assembly instruction execution and, in particular, sub-execution stages. This aim is set keeping in mind when a target device running cryptographic algorithm is attacked through fault injection, it is necessary to know how the important instructions that involve memory read, write and storage are getting affected, since the secret data can be leaked by attacking such instructions in particular. For this, suitable glitch parameters are selected and the glitches are injected at different positions during and surrounding the test instruction execution cycle. For doing so, the overall design should be in such a way that the glitch is triggered automatically such that it is inserted at the exact desired spot in the clock cycle, thus, affecting the test instruction execution. This raises another question: would there be any delay associated with the glitch trigger and insertion process? If yes, then how much
would that be and how to rectify that delay such that the glitch injection is exactly at the desired point?

Hence, this research work is mainly confined to a scenario when the details of the cryptographic algorithm running in a device is known, how the security key can be obtained via sophisticated insertion of clock glitches attacking particular assembly instructions by studying their behavior towards glitches. This study mainly helps in redesigning several cryptographic algorithms such that they are made less vulnerable to any kind of fault injection attacks.

1.3 Proposed Work

This thesis proposes clock-glitch fault injection technique to inject glitches into the clock signal running in a microcontroller unit and hence studying its effects on different assembly level instructions. Although the purpose of this research work looks similar to the work done by the authors of [20], this work differs with respect to the methodology and the hardware being utilized to carry out the experiments. This work focusses mainly on the effect of clock glitches over the execution, sub-execution and pre-execution cycles of the test instructions and also finds the delay between the actual position of glitch insertion and the trigger being set for the glitch insertion. The instructions used in this work are provided by Atmel which classifies them according to their type of operation. These instructions are here further grouped depending on the number of clock cycles they require for their execution. Each group of instructions are tested for their behavior towards clock glitches being injected at different places in and surrounding their execution cycle, by modifying the glitch attributes like width, offset etc. The thesis has described the required background, set-up and methodology for performing the clock-glitching attack over the assembly instructions.

This thesis utilizes the ChipWhisperer-Lite board (CW1173) for performing the whole experiment by controlling the target device, providing clock as well as clock glitches with appropriate properties at appropriate position to the target device. The board includes Spartan-6 ZTEX LX-9 FPGA and the Atmel AVR XMEGA 128D4U target device (CW303). The physical connection between the target and the FPGA is pre-made. A clock signal with frequency of 7.37MHz is generated by the main board and supplied as an external clock to the target device.
The Capture software, provided by the ChipWhisperer, is used for establishing the hardware connection between the main board and the target board. The clock glitches are designed and triggered through the Capture software.

1.4 Organization

This section lists the rest of the chapters in this thesis and gives a brief about what these chapters discuss about.

Chapter II-Background of Study provides the necessary background for understanding the subsequent chapters. It explains the various basic concepts, terminologies and methodologies involved in general Hardware attacks as well as in Clock-glitch attack in particular. The existing literature related to the general fault injection attacks \cite{21} \cite{24} \cite{25} and clock glitching attacks \cite{20} \cite{22} \cite{23} have been discussed in the Chapter III-Literature Review. The Chapter IV-Experimental setup and Design details the environment of the experiments conducted for this thesis and also explains the equipment and devices that were utilized for conducting the experiments. The Chapter V-Implementation and Performance Study discusses the main work that has been proposed in this thesis. It explains how different assembly level instructions were tested for the effect of clock-glitch attack on them and describes the difference in behavior of these instructions towards glitch attack. Finally, the Chapter VI-Conclusion and Future Works summarizes the thesis and provides the possible future works that can be implemented using clock-glitching technique that has been described in this work. The various program codes and miscellaneous concepts related to the proposed work have been given in the Appendix section.
CHAPTER II

BACKGROUND OF STUDY

Hardware is the root of computation and communication. It is the enabler of any software, algorithm, or communication protocols. All the computation will eventually be carried out by hardware, i.e., by the processor, and hence, it is an efficient option to implement cryptography. But, by varying some parameters in the devices running cryptography, the secret data is more likely to be leaked. This is possible by injecting faults in the device clock, varying the device power or even by detecting the electromagnetic leakages that take place during run of the cryptographic algorithm. The faults are found to be induced in the device by unusual conditions of close, physical environment of the device running cryptographic algorithm. The study about the various possible hardware attacks, analyses and its counter-measures is termed as “Hardware Security”.

This chapter discusses the basic concepts like hardware attacks, types and analyzing methods, that are necessary to understand the subsequent chapters. This covers a brief introduction about Hardware Security and explains mainly about clock-glitch attack. The Section 2.1 provides information about the various hardware attack analysis types, which is followed by explanation about clock-glitch attack in Section 2.2. The Section 2.3 explains clock-glitching at different levels of attack with the help of an example. It also explains various features of the ChipWhisperer Capture software, which is the basic tool used for injecting clock-glitches into the target device.
2.1 Hardware Attacks and Analyses

This section discusses in short about the various types of hardware attacks and analysis methods. Hardware security is a serious emerging concern in chip designs and applications. The hardware attacks cover all forms of attacks against cryptographical devices relying on the physical realization of the algorithms in hardware or software\textsuperscript{[9]}. Due to the globalization of the semiconductor design and fabrication process, integrated circuits are becoming increasingly vulnerable to Passive and Active hardware attacks.

![Hardware Attacks Classification](image)

*Figure 2.1: Hardware Attacks-Classification [9]*

The figure 2.1 given above, depicts the various types of hardware attacks and classifies them mainly as active and passive attacks. Passive attacks, or Side-Channel Analyses, on the hardware devices result in secret information leakage without affecting the system, but active attacks affect the functioning of the system either entirely or partially. It might even cause catastrophic system failures. These hardware attack types as shown in figure 2.1 are explained below:

- **Side-Channel Attacks**: The unhidden information, such as the time taken for a particular instruction execution, power consumed during the execution of a particular function in the code and electromagnetic waves radiated from the device during the run of a cryptographic algorithm can be observed and some special attack designs can be implemented to modify these properties and retrieve secured information from them\textsuperscript{[13]}. These unhidden properties are known as Side-Channels and hardware attacks made using them are hence termed as
the Side-Channel Attacks. There are different types of methods that are used to analyze the side channel power, time and electromagnetic radiations. Of these, Simple Power Analysis (SPA), Differential Power Analysis (DPA) and Correlation Power Analysis (CPA) are some of the main side-channel power analysis techniques.

- **Reverse Engineering:** Mostly all the products that are available in the market comes with a documentation that contains the list of components which are used by the manufacturers to design these products. Hence, using this basic information, an attacker can design and implement specific method to reach his goal, which can be either a security-breach or for any other research-purpose. These methods involve analysis of hardware used and reconstruction. Specific components for each product, which do not include documentation are not usually manufactured and made available in the markets since it involves huge costs\(^{14}\).

- **Fault Injection:** Generating faults and injecting them into the device running cryptographic algorithm, such that it affects the program flow and execution, due to which the secret data can be found comparatively easily is known as a Fault Injection Attack. Injecting glitches into the clock that runs the whole device or into the power supply given to the device affects the execution of different instructions differently. By studying this, one can understand the program flow and hence retrieve the secret information by providing the glitches accordingly\(^{12}\). The main concept behind the research topic of this thesis falls under this category of hardware attack, that is, clock glitching to be particular.

Fault Injection Attacks can be further classified into three different categories as listed below:

- **Invasive attack:** It is the category of fault injection attacks in which the attacker can have direct electrical access to the internal components by physically probing the system’s components by either using simple or high-tech techniques \(^{15}\). This involves chip depackaging and hence physically damaging the hardware device.

- **Non-invasive attack:** It is the category of fault injection attacks in which the attacker can externally observe and manipulate the target device without causing any physical harm to
it [16]. Bus-snooping and pin-probing [17] are some of the methods that fall under this category of side-channel analyses.

- **Semi-invasive attack:** It is the third category of fault injection attacks that falls in between the first two types, in which, the attacker can still have direct electrical access to the internal components, but with the restriction of causing no damage to the hardware [15][18]. There are different types of glitch attacks that fall under this category [17] and Clock-glitching is one among them.

### 2.2 Clock Glitch Attack

The maximum clock frequency specified by the manufacturer of a device is the reciprocal of the minimum possible time by which the clock signal is assured to be reaching every component of the device properly. A clock glitch is a sudden increase of the system clock frequency for a very short period of time [19]. If a device is forced to operate at a frequency greater than the prescribed maximum frequency only for a particular period of time, the CPU would not execute correctly only for that particular period and the instructions being executed in that period might output undesired values. These instructions would get affected depending on various factors, such as, the time duration for which the clock frequency is increased (Glitch Width), the position of the normal clock cycle during which the glitch is being injected (Glitch Offset) and the number of the instruction execution cycle. But the subsequent instructions would be processed correctly since the clock signal comes back to the normal frequency.

**Figure 2.2** illustrates the insertion of a glitch into the positive half of a one of the clock cycles out of a train of clock pulses. It also indicates the glitch features like glitch width and glitch offset in the enlarged view of the glitched clocked cycle. In a processor, say if the instructions start executing when the rising edge of the clock pulse is reached, then at least that entire clock cycle time would be allotted for its execution, i.e., till the rising edge of the next clock cycle is reached. But, when a glitch is inserted in the positive half of the execution cycle as shown in figure 2.2, the remaining portion of the clock cycle after the glitch is misinterpreted to be the next clock cycle since the next rising edge has been reached, that would invoke fetching of the next instruction and
execution of the previously fetched instruction. This may lead to either skipping of the instruction being executed or the instruction may get partially executed leaving out improper output depending on the position of glitch insertion.

![Glitch insertion](image)

Figure 2.2: Glitch insertion in a single clock cycle in a train of clock signal pulses. It also shows an enlarged view of the glitched clock cycle indicating the glitch width and glitch offset.

When this concept of injecting glitches into the CPU clock, such that a particular instruction execution gets affected, is utilized in a proper manner for getting the secret key from a device running cryptographic algorithm, then it is known as Cryptographic Clock-Glitch Attack. In this case, the instructions running in a program that handles some very important steps of encryption/decryption are targeted, such that the hidden information can be found either directly or can be estimated depending on the result it outputs.

The figure 2.3 shows attacking a target device running a cryptographic algorithm through clock glitching, such that it gives out faulty output. Inserting glitches between the several regular
clock cycles, such that it seems like the frequency of the device clock has been increased for a short span of time and hence the executing operation gets affected. Therefore, comparing the expected and the faulty outputs, some information regarding the secret data can be retrieved, which would lead to the retrieval of the whole secret key/data.

Figure 2.3: Clock glitch attack on a target running cryptographic algorithm. The output generated with normal clock (Expected Output) and that with a glitched clock (Faulty Output) are compared to get some information about the hidden data, using which the whole secret data can be found. [17]

The next section explains how clock-glitch attack is implemented using the ChipWhisperer testbed system and how insertion of clock glitches at different places of a program running simple functions and hence affecting different instruction, either randomly or purposefully, would affect the output.

2.3 Clock Glitch Attack Examples using ChipWhisperer-Capture

For this research work, a software system was used for designing the clock glitch and injecting it into the clock provided to the target device at a particular point of code. The features of a glitch like, width, offset, trigger offset and number of glitches to be inserted, are decided over here before even inserting it into the clock. This software is known as the ChipWhisperer-Capture. This section would introduce various features of the capture software along with an example, that are used to calibrate glitch features.
The ChipWhisperer contains several example codes. Of those examples, the code for clock glitch attack termed as “glitch-simple” is used here that helps in understanding how clock glitches can be injected using ChipWhisperer-Capture software and how the target under attack behaves due to a glitch at different specific positions. The code (See Appendix B.1) is a simple plain program showing different ways of using the clock-glitch insertion technique to disturb the main task. Each of the simple functions given in the code help in understanding the several features of the Capture software and to learn how to use them. The steps involved in setting-up the connection between the FPGA and the target device through Capture software have been explained in the section 4.2 of ChipWhisperer git documentation [5]. The sub-sections 2.3.1, 2.3.2 and 2.3.3 discuss about the various simple glitch functions present in the example code and show how different types of commands are getting affected differently due to clock-glitching. This also progressively shows how to select the glitch width, offset and trigger offset values in order to induce a successful fault in the target device.

2.3.1 Manually triggered glitch

The first function given in the code is glitch_infinite() that counts the number of nested loop iterations and prints it (See Appendix B.2). After all the required connections are made and an external clock of 7.37 MHz is supplied to the target device through the FPGA, which is configured in the Capture software, the glitch width and offset values are selected. The glitch width, offset and the type of trigger can be selected from the Glitch Module of the Capture software as shown in the figure 2.4 given below. For this example, the glitch width is taken to be 2.5% of the CPU clock (~3.4ns) and the glitch offset is taken to be 22% of the CPU clock (~29.92ns).

The glitch is inserted manually through the Capture software when the program is already running in the target. Since the function contains two nested loops each adding one to itself for 500 iterations, thus the external loop counting up to 250000, if the function is getting executed perfectly, one can expect it to print the output in the below given format:

\[
250000 \quad 500 \quad 500
\]

But if the glitch is inserted at a random position when the nested loop given in the function is
getting executed, the output may differ. This may be caused because of either skipping of a loop, incorrect addition in iterating the variables or unexpected exit of the program flow.

The figure 2.4 shows the screenshot taken when manually inserted glitches affected the given code at some point and hence produces wrong results.

2.3.2 Automatically triggered glitch

The next function given in the example code is glitch1() (See Appendix B.3). This function runs a simple infinite ‘while’ loop and the glitch being inserted would exit this loop, thus printing “1234”. This example function introduces many new features of the Capture software. In this case, the glitch to be inserted is going to be externally triggered. That is, the trigger is set in the given function itself which is represented by “trigger_high()” command in the code. When this line of command gets executed, the FPGA should automatically insert a glitch into the next clock cycle.
that affects the immediate C code command. For this, the glitch trigger mode given in the Glitch module of the Capture software is changed to “Ext Trigger: Single-Shot” from being in “Manual” mode as indicated in the figure 2.5. Now, the glitch width is retained to be 2.5% of the CPU clock, i.e., 3.4ns, but the offset is varied within a range of (0 to 50) % of the clock period. To do this, one should know how to use the Glitch Explorer, which is described in the section 6.2.7 [6] of ChipWhisperer git documentation.

In the glitch explorer, the normal response and the successful response should be specified in order to differentiate between the actual output and an expected glitched output. In this case, the normal response is “hello\nA”, whereas, a successfully glitched response should be “hello\nA1234”. When the Capture Multi button is clicked in the software, glitch would be injected each time with different offset value ranging between 0 and 50% of the clock period. The successful glitches would be highlighted in green in the Glitch Explorer window as shown in the figure 2.5 given below.

Figure 2.5: The Glitch Explorer showing externally triggered glitch output for glitch1(). The glitched faulty outputs are highlighted in green, present in the Glitch Explorer window. The Glitch Trigger mode is selected to be Ext. Trigger: Single Shot and in encircled in red present under the Glitch Module of the Capture Software. The figure also labels the Capture-Multi button encircled in red in the top-left corner of the window, which is used to start the automatically triggered glitch-insertion process and hence output the results in the Glitch Explorer window.
On repeating this by varying the glitch width and offset values giving different ranges, the optimal glitch width and offset range for which there are maximum number of successful glitches can be figured out. For this particular hardware set, maximum successful glitched outputs were observed for the glitch width and offset of 2.5% (3.4ns) and 7% (9.5ns) of clock period respectively.

2.3.3 Real time example: Password authentication

The last function given in the example considered is glitch3() (See Appendix B.4) which is used to check a password for entry authentication. The glitch being inserted in the password checking loop would skip the checking condition, thus breaking the authentication irrespective of the password entered. For this again, glitch width of 2.5% and glitch offset of 7% CPU clock are selected, since maximum successful glitches were observed for this value set in the previous glitch function execution.

![Image](image.png)

Figure 2.6: Glitch Explorer showing the output for glitch3() function. The external trigger offset, encircled in red in the left side of the figure, is varied between 0 to 200 in order to fine-tune the glitch and the faulty output is received for a trigger offset value of 14, as encircled in the Glitch Explorer window. The faulty output is highlighted in green.
The function works well if the password is given as “touch” and the authentication is denied for any other entries. While testing the code for successful glitches using Glitch Explorer, the password cannot be manually entered. For this, the password is given in the “Go Command” option under the “Target Connection Settings” tab in the Capture software, such that, each time when an input (password in this case) is prompted, the value stored here would be automatically entered and the authentication testing function would then be getting executed. Now, for testing the efficiency of the glitch attack, some random word is stored in the “Go Command”, so that the normal response in the Glitch Explorer should always be “Denied\nPassword:” until unless a glitch inserted is successfully disturbing the execution of the authentication testing function. Now, in the Glitch Explorer, the Ext Trigger offset is varied with in a range of 0 to 200 and the glitches are inserted when the external trigger is set after some delay corresponding the trigger offset. The figure 2.6 given above shows the Glitch Explorer window showing the glitch attempts for varying trigger offsets.
CHAPTER III

LITERATURE REVIEW

In this chapter, the existing works related to general fault injection attacks have been discussed in the section 3.1 that includes the work on creating a side-channel attack and analysis toolbox which is the backbone of this thesis work. All the hardware and software for attacking the target device by providing clock glitches of specified widths and offsets have been carried out using this work as a basis. The section 3.2 covers the various works that are existing in the literature related to clock glitching attacks to be specific.

3.1 Works related to general Fault Injection attacks

The Cryptographic algorithm implementations continue to proliferate in consumer products because of the increasing demand for secured transmission of highly secret information. Although the current standard cryptographic algorithms proved to withstand exhaustive attacks, their hardware implementations are still prone to side channel attacks like, power analysis and fault injection attacks.

The authors of [25] focus on fault injection attacks that require very easily available low-cost equipment and a very less amount of time for their implementation. This paper provides a comprehensive description of the various types of fault injection attacks on the devices running cryptographic algorithm and the countermeasures that have been developed against such attacks.
The authors in this paper first start by briefly reviewing the widely used cryptographic algorithms like DES, AES, SNOW 3G and RSA, which is then followed by classification of the various currently known fault injection attacks into low cost ones and high cost ones. The low cost fault injection attacks are those that with a low budget a single attacker can mount, whereas, the high cost fault injection attacks require highly skilled attackers with a huge budget for equipment used for the attack. The authors have then listed the attacks that have been developed for the commonly used ciphers and then they have indicated those attacks among them that have been successfully used in practice. They have then presented the known countermeasures against the fault injection attacks that they have described previously and it includes intrusion detection and fault detection. The survey has been concluded by the authors with a discussion on the interaction between fault injection attacks and power analysis attacks.

The authors of [24] claim that optical fault injection is an affordable method because of the practical security tradeoffs made by chip manufacturers and card vendors, even though the fault injection can be made very trivial by employing a combination of various countermeasures. In their paper, the latest developments in optical fault injection into some secure microcontrollers have been detailed. By developing different fault injection mechanisms, the authors of [24] have experimentally shown that the latest protected smart cards are still vulnerable to fault injection attacks. A strong light source like photo flash or a laser beam is used to generate optical faults in devices. Since the electronic devices like smart cards, processors and MCUs are made up of semiconductors that are very sensitive to light, when these are exposed to optical pulses, random faults would be induced, which could even switch the transistors. It is even possible to target certain portions of a chip so accurately by using a precision stage and a focused laser beam, such that the induced fault affects some cryptographic operation, which would lead to leakage of secured data. To attain a higher security level, the authors emphasize that the sensitivity of the hardware to optical perturbation should be decreased by the chip. The authors further claim that a presumption of single or double successful perturbations should be made during the development and hence they have advised to use more than two verification steps.

With the newest fault injection methods, the authors of [24] claim that the fault injection can be more carefully tuned, but the effect of perturbation cannot be fully controlled. Even though additional challenges like positioning and finding correct parameter locations would be created,
the authors are still experimenting with two lasers for injecting faults at two different locations on the chip, one in the CPU and the other one in the cryptographic accelerator.

The authors in [21] have introduced a hardware security analysis platform that deals with a side-channel attack and includes analog capture hardware, target device, capture software and analysis software. This system is completely self-contained and requires no additional hardware or software besides a standard computer. The hardware uses a synchronous capture method that greatly reduces the required sample rate, the data storage requirement, and improves synchronization of traces. Beyond side-channel analyses, the hardware can also be used for other types of fault inject attacks, like Voltage glitching and clock glitching attacks. It also provides modules to adjust the parameters of glitches like, glitch width, offset and triggering conditions. A clock glitch module, that is present in the system, can insert glitches into a target clock using two adjustable delay lines that are built into the FPGA present in the system itself. The clock signal to the target device can be either externally provided by the system FPGA or generated in the target system itself. When a trigger for inserting the glitch into the target clock is given by the software, the glitch being generated in the FPGA would be injected into the target clock depending on the glitch parameters that are set in the software. Once a configuration is set, it can be utilized multiple times on the same target device by even varying each glitch parameter one after the other. The results are stored as power traces, that can be analyzed later using either the Analyzer software provided by this system or even by any other external trace analyzing software. This paper by the authors of [21] forms the base for the research work been carried out for this thesis.

3.2 Works related to Clock Glitch attacks

The literature about fault analysis describes the various fault injection mechanisms, like glitches and lasers, and cryptography analytic methods in order to exploit the faults based on some assumed fault model. The work done by the authors of [20] narrows the gap between the process of injecting a fault into a device and the process of using a technique to analyze the outputs generated due to these faults that affects the cryptographic implementation in the device. They have thoroughly analyzed how injection of clock glitches can affect a commercially low-cost processor. They have used the legacy 8-bit AVR MCUs for the purpose of their experimentation,
that are based on modified Hardware architecture which has two-stage of pipeline. That is, both the data bus and the instruction bus can be accessed in a single clock cycle. As a result, the authors observed that the effects of fault injection on these devices were more complex than commonly reported in the literature and also they stated that injecting an exploitable fault is hard as compared to injecting a random fault. Various terminologies were used by the authors of [20], like Tg: glitch period, Tn: the normal clock period and the difference between Tn and Tg gives the remaining time available in the clock period after the glitch insertion. The glitch period was first taken as 125 ns for which the target device functioned correctly, i.e., the insertion of the glitch had no effect on the device operation. The glitch period was then decreased gradually until 15 ns, which was set as the minimum glitch width that they can produce and inject.

The authors of [23] have designed a clock-glitch generator and that they have integrated in an FPGA for evaluating the effects of faults being injected using clock glitches into the target device and their countermeasures on cryptographic modules. The generator can be embedded in a single FPGA without the need of any external instrument, like a pulse generator, a variable power supply, etc. The authors express that this kind of an integration helps in conducting reliable and reproducible glitch insertion experiments. The glitch generator that they have proposed in this paper exploits all the clock management capabilities that are common in modern FPGAs and generates clock signal with temporal voltage spike. They have made the various features of a glitched clock, like the shape, time of insertion and duration, to be configurable at the run time. In this paper, the authors have conducted fault injection experiments on Side-channel Attack Standard Evaluation Board (SASEBO) to examine the characteristics of the clock-glitch generator that they have designed and proposed. They have also demonstrated an application of their glitch generator in safe-error attack against an RSA processor and as a result, they have found that the minimum duration of a clock glitch that the generator can produce, in order to inject faults into a program running in the devices, is about 0.17 ns.

The authors of [20] have also explained two different ways of producing the glitch and inserting it into the clock. The figure 3.1 and figure 3.2 illustrates these mechanisms. In the first mechanism as depicted in the figure 3.1, the authors have made their FPGA to produce two clock signals, one with a nominal frequency as desired by the target device and the other with high frequency depending on the glitch width that they wanted to generate. A selection line is used to
select a portion of the high-frequency signal in order to inject a glitch. When the selection signal is low, the signal with nominal frequency is fed as clock into the target and the high-frequency signal is fed into the target when the selection signal goes and remains high.

In the second mechanism represented by the figure 3.2, the authors have used a nominal clock signal along with two different phase shifted versions of the signal with same period and using the combination of these three signals the output clock signal is produced and fed into the device as device clock. The output clock remains same as the nominal clock till the selection signal is low.
When the selection signal becomes high, a glitch is generated by logical combination of these signals and injected into the output clock signal.

As a result of their experiment, the authors of [20] expressed that the instructions can be replaced rather than perfectly skipped, and that the effects of faults are deterministic and reproducible. Even though they were able to identify faults with stable behavior, the lack of knowledge they had about the chip's internal working, had limited them from providing a detailed explanation about characterization of the faults they observed. Whereas, the results obtained by the authors of [22] lead to a number of practical guidelines that would be essential when planning experimental studies on fault injection in FPGA based circuits. In their work, the authors have implemented an FPGA based test bed injecting faults using clock glitches, to result in setup and hold violations. Clock signals of different frequencies and phase shifts were generated using the Xilinx Digital clock manager (DCM) component. An AND-OR type multiplexer logical implementation of the clock switch was used by the authors in [22]. The design of selecting the high-frequency signal when the selection line goes high, is similar to the first mechanism used by the authors in [20]. The authors in [22] have used the pre-build serial International Data Encryption (IDEA) algorithm as the test encryption algorithm. The difference between the outputs of the encryption module, one with fault and the other without any fault injection, is represented as a ratio of used clock frequency and maximum frequency of operation reported by the synthesis tool. The authors have used Xilinx Spartan family FPGA board connected to a PC through serial communication has been used for hardware level testing.
CHAPTER IV

EXPERIMENTAL SETUP AND DESIGN

This chapter describes the details of the experiment environment. A relatively new FPGA-Target MCU combinational device called ChipWhisperer-Lite (CW-Lite) has been used mainly for the purpose of generating and injecting clock glitches into the target MCU, apart from being an external clock source. The Atmel Xmega 128D4U microcontroller unit is used as a target and this target device is already attached to the CW-Lite board. The Section 4.1 briefs completely about the ChipWhisperer and explains various software provided by them. It also gives an overview about the general ChipWhisperer hardware and CW-Lite in particular. This is followed by the Section 4.2 that discusses about the Xmega target device. It gives the main features of the target MCU and discusses more about its memory architecture. The last section of this chapter, Section 4.3 covers the method of Clock-Glitch generation, insertion and providing external trigger.

4.1 ChipWhisperer- An Overview

This section gives a brief introduction about ChipWhisperer and the details about the software and hardware that are used for this thesis work are handled by the next two sub-sections.

ChipWhisperer is an open-source side-channel attack and analysis platform for hardware and embedded security research, designed and developed by Colin O’Flynn and Zhizhang (David) Chen [1]. It is a complete side-channel analysis toolbox that includes the analog capture hardware,
the target device and the capture and analysis software. Glitch and fault attacks can be implemented and analyzed on this system apart from the side-channel attacks. This eliminates the difficulty in comparing the results of any hardware attacks on different platforms, by providing a generalized standard platform. The system has a very modular design and has more extensive publically available codes for computer control.

4.1.1 ChipWhisperer-The Software

The ChipWhisperer provides two different software for capturing the traces from the device under test (DUT) and using these saved traces for analysis. The process of connecting the CW-lite with the target device, injecting glitches, capturing the traces and saving them is done with the help of the ChipWhisperer-Capture Software, whereas the process of analyzing the traces saved by the Capture software is done with the help of the ChipWhisperer-Analyzer Software. The various features and functionalities of the Capture software, with the help of clock-glitch examples, have been discussed in the Section 2.2: Background of Study. The whole software is implemented entirely in Python and hence, it provides a simple GUI, can be easily interfaced with other languages like C/C++ and MATLAB, and has a large collection of modules which provide functionality such as cryptographic functions, plotting, numeric computations, low-level IO, and smart card interfaces.

4.1.2 ChipWhisperer-The Hardware

The ChipWhisperer system is designed to work on several FPGA boards, all based on the Spartan-6 FPGA [2]. But the discussion is restricted only up to CW-lite board (CW1173), since this research work is based completely on this particular board that uses Spartan-6 ZTEX LX9 FPGA. The CW-Lite comes with the Atmel AVR XMEGA 128D4U target board (CW303) attached with it. The target device can be detached and the board can be connected to any other target. The complete ChipWhisperer-Lite board is shown in the figure 4.1[26]. As it can be seen from the figure 4.1, the CW-lite includes both the clock and glitch generating FPGA (CW-Lite Main Board) and the target device under attack (XMEGA Target Board) attached together. By default, all the connections between the main board and the target board are made by the 20-pin target connector, in which, the clock signal to the target from the FPGA is provided through the pin 6 that is named
as FPGA-HS2. The various other connections between the main and target boards are given in the Table 4.1[27] shown below.

![Diagram of ChipWhisperer-Lite Board (CW1173) with XMEGA target device (CW303) (Source: CW Documentation [26]). Figure 4.1: The figure shows 20-Pin Target Connector encircled in red, which makes hardware connection between the FPGA and the Target. The figure also indicates Pin 6 and Pin 16, that are used to output clock/glitch signal to the target and trigger signal from the target to the FPGA respectively. A brief description of all the 20-pins have been given in the table 4.1 given below.

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Dir</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+VUSB (5V)</td>
<td>O</td>
<td>Not Connected on ChipWhisperer-Lite.</td>
</tr>
<tr>
<td>2</td>
<td>GND</td>
<td>O</td>
<td>System GND.</td>
</tr>
<tr>
<td>3</td>
<td>+3.3V</td>
<td>O</td>
<td>+3.3V to Target Device, can be turned off, 200mA available.</td>
</tr>
<tr>
<td>4</td>
<td>FPGA-HS1</td>
<td>I/O</td>
<td>High Speed Input (normally clock in).</td>
</tr>
<tr>
<td>5</td>
<td>PROG-RESET</td>
<td>I/O</td>
<td>Target RESET Pin (AVR Programmer).</td>
</tr>
<tr>
<td>6</td>
<td>FPGA-HS2</td>
<td>I/O</td>
<td>High Speed Output (normally clock or glitch output).</td>
</tr>
<tr>
<td>7</td>
<td>PROG-MISO</td>
<td>I/O</td>
<td>SPI input: MISO (for SPI + AVR Programmer).</td>
</tr>
<tr>
<td>8</td>
<td>VTARGET</td>
<td>I</td>
<td>Drive this pin with desired I/O voltage in range 1.5V-5V.</td>
</tr>
<tr>
<td>9</td>
<td>PROG-MOSI</td>
<td>I/O</td>
<td>SPI output: MOSI (for SPI + AVR Programmer).</td>
</tr>
<tr>
<td>10</td>
<td>FPGA-TARG1</td>
<td>I/O</td>
<td>TargetIO Pin 1 – Usually UART TX or RX.</td>
</tr>
<tr>
<td>11</td>
<td>PROG-SCK</td>
<td>I/O</td>
<td>SPI output: SCK (for SPI + AVR Programmer).</td>
</tr>
<tr>
<td>12</td>
<td>FPGA-TARG2</td>
<td>I/O</td>
<td>TargetIO Pin 2 – Usually UART RX or TX.</td>
</tr>
<tr>
<td>13</td>
<td>PROG-PDIC</td>
<td>I/O</td>
<td>PDI Programming Clock (XMEGA Programmer), or CS pin (SPI).</td>
</tr>
<tr>
<td>14</td>
<td>FPGA-TARG3</td>
<td>I/O</td>
<td>TargetIO Pin 3 – Usually bidirectional IO for smartcard.</td>
</tr>
<tr>
<td>15</td>
<td>PROG-PDID</td>
<td>I/O</td>
<td>PDI Programming Data (XMEGA Programmer).</td>
</tr>
<tr>
<td>16</td>
<td>FPGA-TARG4</td>
<td>I/O</td>
<td>TargetIO Pin 4 – Usually trigger input.</td>
</tr>
<tr>
<td>17</td>
<td>GND</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>+3.3V</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>GND</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>+VUSB (5V)</td>
<td>O</td>
<td>Not Connected on ChipWhisperer-Lite.</td>
</tr>
</tbody>
</table>

Table 4.1: CW-Lite 20-pin Connector (Source: CW Documentation [27]). Pin 6 and Pin 16 are encircled in red. The clock signal to the target is provided by the FPGA via Pin 6. When the glitch trigger is set via Pin 16, the Pin 6 transfers the glitched clock to the target from the FPGA.
The power supply of +3.3V to the target device is given by the main board through pin 3, whereas, power to the main FPGA board is given from a desktop computer through a micro-USB connection.

![Figure 4.2: Experimental set-up](image)

The complete experimental set-up, that includes CW-lite, target device and a desktop computer with CW-Capture software running on it, is shown in the figure 4.2. The Capture software is first opened in the computer through Linux terminal and all the basic specifications representing the type of device to be connected are set. The CW-Lite is then connected to the system and the connection between the software and the hardware is then established.

The next section deals with the features and the memory architecture of the XMEGA target device.

### 4.2 XMEGA-The Target Device

The CW-Lite comes with XMEGA 128D4U as a target device. The AVR XMEGA D is a family of low-power, high-performance and peripheral rich CMOS 8/16-bit microcontrollers based on the AVR enhanced RISC architecture \[3\]. It is an entry-level 8-bit microcontroller targeting the power-conscious applications. The important features of this device are as listed below \[4\]:

- Operating Voltage
- 1.6 – 3.6V

- Operating frequency
  - 0 – 12MHz from 1.6V
  - 0 – 32MHz from 2.7V

- Nonvolatile program and data memories
  - 128KBytes of in-system self-programmable flash
  - 8KBytes boot section
  - 2KBytes EEPROM
  - 8KBytes internal SRAM
  - 4 general purpose registers

The Atmel AVR architecture has two main memory spaces, the data memory that stores the data, and the program memory that stores the executable program code as well as the data. The figure 4.3 [4] shows the entire block diagram of XMEGA D4 target device.

![Figure 4.3: XMEGA D4 Block Diagram](Source: AVR XMEGA D4 Datasheet [4]). The clock signal externally provided by the FPGA is received by the XMEGA Target via XTAL/TOSC1 as indicated in the figure.
The program memory includes the flash memory (128KB+8KB+8KB) that is sub-divided into 3 sections, and the data memory comprises the internal SRAM (8KB), the I/O memory (4KB) and the EEPROM (2KB) for nonvolatile data storage. Since this thesis is concerned with the effects of glitch insertion into the clock, such that it affects the execution of different types of assembly instructions, that also include memory mapping, addressing and branching instructions based on the stored values, hence, it is very much important to know the basic memory architecture of the target device. It is also important to know how fast an instruction and other operands during instruction execution are fetched, that depends on the place of the main program and the data storage. The upcoming sub-section 4.2.1 and 4.2.2 briefs about the Flash Program Memory and the Data memory sections of the XMEga MCU.

4.2.1 Flash Memory

The XMEGA D4 target device consists of an on-chip, in-system reprogrammable, linearly addressed flash memory for program storage. The AVR CPU instructions are 16 or 32 bits wide and hence each flash location is 16 bits wide. This memory is sub-divided into three sections as listed below:

- Application Section (128KB)
- Application Table Section (8KB)
- Boot-loader Section (8KB)

The Application section stores the executable application code, whereas, the Application Table section, which is actually a part of the Application section, is allowed to store data. The application code can also reside in the table section if it is not used for data storage. The Boot-loader section stores the boot loader software that allows the store program memory (SPM) instruction to initiate programming. The SPM instruction is used to write to the flash from the application software.

4.2.2 Data Memory

The data memory of XMEGA D4 target device is organized as one continuous memory section and as in the case of flash memory, the data memory is also sub-divided into three section as listed below:
I/O Memory (4KB)
Internal SRAM (8KB)
Optionally memory-mapped EEPROM (2KB)

The load (LD/LDS/LDD) and store (ST/STS/STD) instructions can access all the I/O memory locations since these instructions are used to transfer data between the 32 registers in register file and the I/O memory. The 2KB of EEPROM is used for non-volatile data storage. It is also accessible using the load and store instructions only when it is memory mapped. In other case, EEPROM is addressable in a separate data space as default.

As far as the memory access time is concerned, it takes one full CPU clock cycle for either reading from or writing to an I/O memory location. While, write to SRAM takes one clock cycle and read from SRAM takes two full CPU clock cycles. This time difference in reading and writing a data would affect the time required for the execution of particular instructions, which might hence show a different reaction towards clock-glitch attacks.

4.3 Glitch generation and External trigger

This section discusses how a glitch is generated, inserted into an outgoing clock pulse and how this process is triggered externally via the code running in the target device.

The figure 4.4[8] given below depicts the overall idea behind the clock glitch generation. A system clock (which can come from either the ChipWhisperer FPGA or the Target Device) is used to generate the glitches. In this case, a 7.37 MHz clock is generated by the Spartan-6 FPGA present in the ChipWhisperer-lite main board and this is provided as an external clock to the Xmega target device. As it can be seen clearly, the input clock coming from the FPGA is phase shifted twice. These shifts depend on the glitch-width and glitch-offset values that are inputted through the Capture software. After taking NOT of the 1st phase shifted pulse train, the two phase shifted clock pulses are allowed to pass through an AND gate. The output of the AND gate would produce the Glitch Stream with the desired glitch width and offset values. This glitch stream, thus produced, would be inserted into the clock once the Enable pin is set high. (Note: The process of setting up
these values and conducting simple experiments have been discussed in the Section 2.2-
Background of Study).

![Figure 4.4: Clock-glitch generation (Source: CW Documentation [8]). Once the Glitch Enable pin is set, the
glitch becomes ready and is inserted into the input clock as shown in the figure 4.5.]

The Glitch Enable pin shown in figure 4.4 is set and the glitch is inserted into the clock going
to the device under test (DUT), as shown in the figure 4.5[8] given below, at an appropriate position
depending on the glitch offset specified in the Capture software.

![Figure 4.5: Glitch insertion into the Clock (Source: CW Documentation [8]). The glitch generated is XORed with the input
clock, as highlighted in the figure, to produce a glitched clock signal and this signal is provided to the target device.]

The generated glitch from the figure 4.4 is XORed with the input clock present in the figure
4.5, to produce a glitched clock train. The number of consecutive clock cycles to be glitched can
also be chosen by providing corresponding value for the Repeat option given in the Glitch module
of the Capture software. If the repeat value is 1, then only 1 clock pulse would be glitched, but if
the repeat value is more than one, say 10, then 10 continuous clock cycles would be inserted with
glitches of same width and offsets that are already mentioned in the Glitch module.

The Glitch module of Capture software also provides other options of operations that can be
performed between the clock and the glitch streams. Depending on the application, either the clock
alone or the glitch stream alone can be chosen by selecting the appropriate option from the glitch module. In case of selecting a glitched clock irrespective of the offset value, the Clock XOR option is chosen as explained earlier. But the option that involves OR Gate operation between the clock and the glitch stream would be meaningful only if the operation is carried out between a glitch with positive amplitude and a negative clock cycle. In all other cases, only the original clock signal would be selected.

![Figure 4.6: Routing of trigger to Glitch module (Source: CW Documentation [29]).](image)

The figure 4.6[29] shows various sources through which the Glitch Enable pin shown in figure 4.4 can be set, i.e., the glitch insertion process can be triggered. It can be set either manually or externally by giving a trigger set command in the code running in the target device. This option can be chosen through Capture software from the Glitch module.

The SAD trigger is one of the sources that is used to trigger glitch based on a pattern in the analog waveform that comes from the target device performing a cryptographic or authentication operation[28]. But, at present the manufacturers have not built the SAD trigger into the CW1173: ChipWhisperer-Lite Board. In the case of the Digital IO Pattern matching trigger, when a particular pre-set digital data pattern is observed in a running code, the trigger is set and hence the glitch would be injected. It is another kind of providing an external trigger for glitch insertion into the running clock.
But, with respect to the experiments conducted in accordance to this thesis, a trigger setting function (trigger_high()) is called in the execution program running in the target device. Whenever this function gets executed, the glitch is triggered and this trigger is routed to the glitch module in the Capture software through the External line Select Matrix, that acts as the trigger source as shown in Figure 4.6. There would be different options available in the Glitch module of Capture software for selecting the type of Glitch trigger needed for the specific kind of application. For this purpose, External Trigger: Single-Shot is chosen and when the program is allowed to run, the glitch enable pin shown in Figure 4.4 is set whenever the trigger set command in the code is encountered, as mentioned earlier.
CHAPTER V

IMPLEMENTATION AND PERFORMANCE STUDY

This chapter discusses about how different assembly level instruction sets are attacked by injecting clock glitch at various points of clock and describes how different instructions behave differently due to glitch insertion. The glitches are introduced not only during the execution period of the particular instruction considered, but also a clock period before and after the execution phase to check how the instruction under test is getting affected due to the glitch, even though it is not directly affecting it. The instructions may take different number of clock cycles for their execution, hence, clock glitch is inserted into all the execution cycles, one at a time, to check the sub-execution level effect of clock-glitch on an assembly instruction.

Section 5.1 deals with the main idea behind how an instruction is actually tested completely for its effect due to clock-glitch, by considering a very basic single-cycle ADD instruction. This section also discusses on correcting the delay between the time of setting up the trigger for inserting the glitch and the time when the glitch is actually getting injected. This difference in time has been coined as the Trigger Delay. This value differs from device to device.

Section 5.2 discusses clock-glitch attack on instructions that take two or more clock cycles for their execution. It is divided into sub-section 5.2.1 that deals with two clock cycles instructions and sub-section 5.2.2 that deals with three clock cycle instructions. At last, the Section 5.3 uses the results obtained from its previous sections to implement clock glitch attack on a simple Caesar cipher algorithm.
5.1 Clock-Glitch Attack on Single-Cycle instructions

5.1.1 Trigger Delay

The assembly instruction for addition operation “ADD” is first considered for clock glitch attack since it is a very simple single cycle assembly instruction. When the trigger set command is given just before the instruction in assembly code, it was observed that this instruction gets executed perfectly and the corresponding correct output was printed. This implies, the instruction under attack is not getting affected due to the clock glitch insertion if this instruction is just immediate to the trigger set as shown in figure 5.1.

![Figure 5.1: Addition operation in assembly immediately after Trigger set. The Z-register here is used to address Port A of XMega device for setting and clearing the glitch trigger which is received by the FPGA through the pin 16 as shown in Table 4.1 of Chapter IV, representing the CW-Lite 20-pin connector.](image)

A delay between setting the clock glitch insertion trigger and the point of actual glitch insertion is then assumed and termed as Trigger Delay, as mentioned earlier. In order to estimate this, a single cycle instruction is first considered (ADD instruction in this case) and number of NOP (no operation) instructions are inserted between the trigger set command and the instruction under attack. These NOP instructions are added one by one in order to find the glitch effect and the delay. When exactly 6 NOP instructions were inserted as shown in figure 5.2, the Glitch explorer of ChipWhisperer Capture software showed the instruction getting skipped successfully for several attempts, because of the glitch being injected into the clock cycle, that affects the instruction execution either directly or indirectly. This means that we can very well assume the Trigger delay to be 6 clock cycles, since each NOP instruction takes one clock cycle for its execution and instruction skips are achieved when 6 such NOP instructions are inserted.
For the code given in figure 5.2, if the ADD instruction skips due to the clock glitch, then the register r24 will retain its previous value ($20_{(10)}: \text{x14}_{(\text{Hex})}$). If the instruction is not getting affected due to the glitch, then the register r24 will output the addition result ($10+20=30_{(10)}: \text{x1e}_{(\text{Hex})}$). If the instruction is getting attacked due to clock glitch but does not get skipped, then the register r24 may store some random value. The outputs are observed due to the clock glitch insertion with a clock width of 2.5% of the clock period and at 2 different offset positions (Set 1 and Set 2) as described later. The observation is done by inserting 5, 6 and 7 nop instructions between the trigger set command and the single-cycle ALU instruction. This is done for both the ADD instruction as well as for the AND instruction. It is to be noted that the blank places in the table represents that no effect of clock glitch was seen on the instruction under attack.

Now, Consider the figure 5.3 and the figure 5.4 [7] given below that depict the 2-stage pipeline concept being implemented in 8-bit Atmel XMEGA (target) device and the internal timing concept for the instruction execution which is explained using a single clock cycle ALU operation. The execution of the ADD instruction, being a single cycle ALU operation, would also undertake similar sub-cycles for its operands fetching, ALU operation execution and writing the results back as shown in figure 5.4. Also, as shown in figure 5.3, the ADD instruction would be fetched during the execution of the 6th NOP instruction. This means, while executing the code shown in figure 5.2, if the trigger delay was actually 5 clock cycles, the glitch would have been injected in the 6th cycle, where execution of 6th NOP and fetching of ADD instruction are taking place. This could
have affected the execution of ADD instruction (indirectly) that is meant to be executed in the following clock cycle. But, if the delay is 6 clock cycles as assumed earlier, then the glitch would have been injected in the 7th clock cycle where the ADD instruction is getting executed. In this case, we need to know which part of the ADD instruction execution (operand fetch, operation execute or result write back) is getting affected (directly) such that the instruction has got skipped. This leads to an uncertainty over the assumption of trigger delay to be only 6 clock cycles.

Figure 5.3: Parallel instruction fetches and execution in 8-bit Atmel XMEGA

Figure 5.4: Single cycle ALU execution in 8-bit Atmel XMEGA

In order to confirm the exact value of this trigger delay, which is a very important factor that affects the entire study, an experiment was conducted (See Appendix A). Different cases where considered with different trigger delay assumptions, the corresponding assembly code were executed and the results were noted and analyzed. By comparing the actual outputs and the various possible outcomes for different cases, the actual delay between the point of trigger set and the point of actual clock glitch insertion (i.e., the Trigger Delay) was confirmed to be Six Clock Cycles.
5.1.2 Single-Cycle Instructions

As it is mentioned in the previous sub-section, it takes six full clock cycles to inject a glitch into the clock after a glitch trigger is set. In order to find the effect of a clock glitch over different stages of a test instruction execution, as depicted in figure 5.4, the position of the test instruction with respect to the position of the glitch trigger is varied such that the glitch gets inserted into the clock either before or during the test instruction execution cycle. To achieve this, the following clock glitch insertion positions are considered:

- Glitch in Test Instruction Execution Cycle (A)
- Glitch in clock cycle previous to Test Instruction Execution Cycle (B)

The figure 5.5 shows the CPU clock diagram and fetch-execution cycles of trigger set (0\textsuperscript{th} clock cycle), 6 NOP instructions (1-6 clock cycles), single cycle ALU instruction (ADD, 7\textsuperscript{th} clock cycle) and trigger clear (9\textsuperscript{th} clock cycle). It also shows the sub-cycles involved in the execution of the ALU instruction that includes Register Operand Fetch (ROF), ALU operation execution and Result Write back (RWB).

![Figure 5.5: Depiction of 6 nop clock cycles inserted between Trigger set (0\textsuperscript{th} clock cycle) and ALU instruction execution (7\textsuperscript{th} clock cycle) and single cycle ALU instruction execution sub-cycles.](image)
For glitch insertion in all the above mentioned cases, a glitch width of 2.5% of the CPU clock (7.37 MHz) is considered and the code is run for two sets of glitch offsets:

- **Set 1**: (-50 to 0) % of CPU clock, in this the glitch is inserted in the negative half of the previous clock cycle [8].

- **Set 2**: (0 to 50) % of CPU clock, in this the glitch is inserted in the positive half of the present clock cycle [8].

The above mentioned two sets of offsets (Set 1 and Set 2) would be applied through the glitch module of ChipWhisperer Capture software. Now, the code is run and the results are analyzed for the two different glitch positions, as mentioned earlier.

![Diagram](image)

**Figure 5.6**: Glitch insertion in positive (Set 2) and negative (Set 1) halves of single-cycle test instruction execution cycle (A) and in nop execution cycle that is previous to test instruction execution cycle (B).

### 5.1.2.A Glitch in Test Instruction Execution Cycle

The block (A) in the figure 5.6 shows the insertion of clock glitch into the test instruction execution cycle. As mentioned above, the glitches are injected one at a time for both the sets of glitch offsets (Set 1 and Set 2). This is done by employing 6 NOP instructions between the trigger
set and the actual glitch insertion, such that the glitch gets injected either in the 2\textsuperscript{nd} half or in the 1\textsuperscript{st} half of the test instruction execution cycle depending on the glitch offset applied.

When a glitch is inserted into the 2\textsuperscript{nd} half of the instruction execution cycle using a Set 1 offset, the ALU operation would have already completed, but the Result Write Back operation (RWB) would be still executing. Thus, the glitch inserted would affect the RWB, such that, it would either get skipped or would write some random value back into the register, leading to the register r24 either retaining its previous value or storing some other random value. In this case, the RWB was observed to get affected for smaller offsets (between 0 to -5\% of clock period).

For the Set 2 offsets case as shown in block (A) of figure 5.6, the glitches are injected over the 1\textsuperscript{st} half of the ALU instruction execution cycle, thus, due to the limited time available for its execution, the entire ALU operation would get skipped for almost any value of offset used. This can be seen very much clearly from the Table A.2 given in Appendix A that shows the observed results for two single-cycle instructions (ADD, AND).

### 5.1.2.B Glitch in clock cycle previous to Test Instruction Execution Cycle

For examining the effect of clock-glitch inserted during the pre-execution phase of the test instruction, 6 NOP instructions are added between the trigger set command and the ALU instruction, and, the glitches with Set 1 offsets as shown in the block (B) of figure 5.6 are injected into the 2\textsuperscript{nd} half of the ALU instruction fetching cycle, that is just before the actual ALU execution cycle. The portion of the clock cycle immediately after the glitch acts as a new clock cycle, since the rising edge would have been reached due to the glitch. In case of more negative offsets (towards –ve 50\%), there would be enough time available for the ALU execution to take place. But, since the ALU execution would start in the rising edge of this negative glitch and should end while encountering the next rising clock edge, when the offset values get closer to 0\%, the limited time would not be enough for the execution of the test instruction and as a result the entire instruction would get skipped.

In the case of Set 2 offset (block (B) of figure 5.6) glitch insertions, even though the glitch is in the previous clock cycle to the test instruction execution, the portion of the NOP clock cycle shown in figure 5.6 after the glitch acts as a new clock cycle, where the ALU execution should
take place. This means, the execution of the ALU instruction would take place earlier due to the skipping of its previous nop instruction execution. When the offset value increases (towards 50%), the total time period of the new clock cycle decreases, hence decreasing the time availability for the ALU instruction execution. This would lead to skipping of the test instruction execution for glitches inserted with larger offset values as that can be clearly seen from the Table A.2 given in Appendix A.

<table>
<thead>
<tr>
<th>Glitch Offset</th>
<th>Glitch Cycle</th>
<th>Test instruction Execution</th>
<th>Cycle previous to Test Instruction Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>Glitch Cycle</td>
<td>Glitch affects RWB of Test Instruction for smaller offsets</td>
<td>Test Instruction gets skipped only for smaller offsets</td>
</tr>
<tr>
<td>Set 2</td>
<td>Test Instruction gets skipped for any offset value</td>
<td>Test Instruction gets skipped only for larger offsets</td>
<td></td>
</tr>
</tbody>
</table>

*Table 5.1: Summary of observations made by testing the effect of clock-glitch insertion in the clock cycle executing the single-cycle test instructions and in the clock cycle prior to the test instruction execution.*

All the single-cycle instructions provided by Atmel AVR XMega assembler, irrespective of their type, were tested for the effect of clock-glitch attack on them, in a similar manner as explained in this section. All these instruction are found to behave similarly as explained in the sub-sections 5.1.2.A and 5.1.2.B and it has been summerised in the Table 5.1 given above.

### 5.2 Clock-Glitch Attack on Multi-Cycle Instructions

The below given Table 5.2 shows the various assembler instructions that take different number of clock cycles for their execution and these instructions are categorized according to the type of operation for which they are used.

The previous section discussed about the effect of clock-glitch over single execution cycle instructions as those shown in the Table 5.2 given below. This section talks about how the instructions that require two or more number of clock-cycles for their execution react towards clock-glitch insertion at different positions in the clock surrounding the instruction execution period.
<table>
<thead>
<tr>
<th>Type of Operation</th>
<th>Instructions</th>
<th>Number of Execution Clock-Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic and Logic instructions</strong></td>
<td>ADD, SUB, SUBI, ADC, SBC, SBCI, AND, ANDI, OR, ORI, EOR, COM, NEG, CBR, INC, DEC, SER, CLR</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ADIW, SBIW, MUL, MULS, MULSU, FMUL, FMULS, FMULSU</td>
<td>2</td>
</tr>
<tr>
<td><strong>Data Transfer instructions</strong></td>
<td>MOV, MOVW, LDI, IN, OUT</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>LAC, LAT, LAS, XCH, ST, PUSH, POP</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>LDS, LD, STS, ST, LPM, ELPM</td>
<td>3</td>
</tr>
<tr>
<td><strong>Branch instructions</strong></td>
<td>CP, CPC, CPI</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RJMP, JMP, EJMP, RCALL, ICALL</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>BRBC, BRBS, BREQ, BRNE, BRCS, BRCC, BRSH, BRLO, BRMI, BRPL, BRGE, BRLT, BRHS, BRHC, BRTS, BRTC, BRVS, BRVC, BRIE, BRID</td>
<td>2/1*</td>
</tr>
<tr>
<td></td>
<td>SBRC, SBRS, SBIC, SBIS, CPSE</td>
<td>3/2/1**</td>
</tr>
<tr>
<td></td>
<td>JMP, CALL</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RET, RETI</td>
<td>4</td>
</tr>
<tr>
<td><strong>Bit and Bit-test instructions</strong></td>
<td>LSL, LSR, ROL, OR, ASR, SWAP, BSET, BCLR, BST, BLD, SEC, CLC, SEN, CLN, SEZ, CLZ, SEI, CLI, SES, CLS, SEV, CLV, SET, CLT, SEH, CLH, NOP, SLEEP, WDR, BREAK</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SBI, CBI</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.2: Atmel AVR X-Mega assembler instructions categorized depending on the type of operation and number of execution clock-cycles they use.

* 2 clock-cycles if condition is true, 1 clock-cycle if condition is false.
** 3 clock-cycles if condition is true and the instruction skipped is 2 words, 2 clock-cycles if condition is true and the instruction skipped is 1 word, 1 clock-cycle if the condition is false.
5.2.1 Two Clock-Cycle Instructions

The below given figure 5.7 depicts the CPU clock diagram and fetching-execution cycles of trigger set command (0\textsuperscript{th} clock cycle), 6 NOP instructions (1-6 clock cycles), double-cycle assembler instruction (7\textsuperscript{th} and 8\textsuperscript{th} clock cycles) followed by another NOP instruction (9\textsuperscript{th} clock cycle) and trigger clear command (10\textsuperscript{th} clock cycle).

![Figure 5.7: Depiction of 6 NOP clock cycles inserted between Trigger set (0\textsuperscript{th} clock cycle) and a two clock-cycle test instruction execution (7\textsuperscript{th} and 8\textsuperscript{th} clock cycles).]

As in the case of single-cycle instructions, different instructions belonging to different categories of operation, that require 2 clock cycles for their execution were tested to check how a clock-glitch insertion before and during the instruction execution has influence on them. To achieve this, the number of NOP instructions between the trigger set command and the double-cycle test instruction were modified such that clock glitch can be inserted at different positions as shown in the figure 5.8 and these glitch positions are listed below:

- Glitch in Test Instruction 2\textsuperscript{nd} Execution Cycle (A)
- Glitch in Test Instruction 1\textsuperscript{st} Execution Cycle (B)
- Glitch in clock cycle previous to Test Instruction Execution Cycle (C)
A two clock-cycle instruction ADIW (Add Immediate to Word) is now considered here as the test instruction in order to explain the general behavior of all the other two clock cycle instructions available as listed in the Table 5.1. This particular ALU instruction is actually used to add an immediate value, ranging between 0 to 63, to a register pair and place the result in the register pair. Hence, for testing purpose, 0x14 (20\(_{10}\)) was added to the register pair r24-r25 that already stores 0x14 in it, such that the actual output should be 0x28 (40\(_{10}\)), while the register pair r24-r25 should retain its original value 0x14 if the instruction completely gets skipped.

Figure 5.8: Glitch insertion in positive (Set 2) and negative (Set 1) halves of double-clock cycle test instruction 2\(^{nd}\) execution cycle (A), 1\(^{st}\) execution cycle (B) and in nop execution cycle that is previous to test instruction execution cycle (C).
5.2.1.A Glitch in Test Instruction 2nd Execution Cycle

The behavior of the double-cycle test instruction towards glitch insertion with Set 1 offsets into the 2nd half of the 2nd execution cycle is similar to that in the case of glitch insertion into the 2nd half of a single-cycle instruction execution clock cycle. The glitch inserted affects the RWB for smaller offsets, such that, it would either get skipped or would write some random value back into the register. But, when glitches were injected into the 1st half of the 2nd execution cycle with Set 2 offsets as indicated in block (A) of figure 5.8, the instruction get skipped for initial offset values, but starts to produce wrong results as offsets increase gradually. This is because the RWB process has got affected due to the glitch injection.

5.2.1.B Glitch in Test Instruction 1st Execution Cycle

When the clock glitch was applied with Set 1 offsets into the 2nd half of the double-cycle test instruction’s 1st execution cycle as represented by block (B) of figure 5.8, the execution got skipped once for an offset of -5.5% of clock period thus the register pair retaining its original value of 0x14 and also produced wrong results (produced 0x10 (16_{10}) as the output) for offsets close to -5.5% of clock period. But it was observed that the outputs produced in the case of inserting glitch with Set 2 offsets into the 1st half of the 1st execution cycle of a two-cycle test instruction is similar to that in the case of glitch insertion into the 1st half of any single-cycle instruction execution clock cycle. That is, the entire test instruction execution skips for almost any value of offset used due to the limited time available for its execution.

5.2.1.C Glitch in clock cycle previous to Test Instruction Execution Cycle

The case of glitch insertion into the clock cycle that just comes before a two-cycle test instruction execution cycle is entirely similar to the case of a single-cycle instruction execution as explained in the sub-section 5.1.2.B. The test instruction execution gets skipped for glitch inserted into the 1st half of the previous clock cycle with only smaller Set 1 offsets, whereas, it gets skipped for only larger Set 2 offsets when a glitch is applied into the 2nd half of the previous clock cycle and this happens for the same reason as described in the sub-section 5.1.2.B.
Table 5.3: Summary of observations made by testing the effect of clock-glitch insertion in the clock cycles executing the double-cycle test instructions and in the clock cycle prior to the test instruction execution.

<table>
<thead>
<tr>
<th>Glitch Cycle</th>
<th>Test instruction 2\textsuperscript{nd} Execution Cycle (A)</th>
<th>Test instruction 1\textsuperscript{st} Execution Cycle (B)</th>
<th>Cycle previous to Test Instruction Execution (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glitch Offset</td>
<td>Glitch affects RWB of Test Instruction for smaller offsets</td>
<td>Test Instruction rarely skips but produces wrong output for smaller offsets</td>
<td>Test Instruction gets skipped only for smaller offsets</td>
</tr>
<tr>
<td>Set 1</td>
<td>Test Instruction skips for smaller offsets and RWB gets affected for higher offsets</td>
<td>Test Instruction gets skipped for any offset value</td>
<td>Test Instruction gets skipped only for larger offsets</td>
</tr>
<tr>
<td>Set 2</td>
<td>Test Instruction skips for smaller offsets and RWB gets affected for higher offsets</td>
<td>Test Instruction gets skipped for any offset value</td>
<td>Test Instruction gets skipped only for larger offsets</td>
</tr>
</tbody>
</table>

5.2.1 Two Clock-Cycle Instructions

The other two-clock cycle instructions that show similar behavior as that of ADIW are SBIW, MUL, MULS, MULSU, FMUL, FMULS and FMULSU as far as Arithmetic and logic instructions are concerned. A similar experiment was conducted using other types of two-clock cycle instructions as well. It was observed that all these instructions were behaving similar to what has been explained in this sub-section 5.2.1 and this general behavior has been summarized in the Table 5.3 given above.

5.2.2 Three Clock-Cycle Instructions

The below given figure 5.9 depicts the CPU clock diagram and fetching-execution clock cycles of the trigger set command (0\textsuperscript{th} clock cycle), 6 NOP instructions (1-6 clock cycles), triple clock cycle assembler instruction (7-9 clock cycles) followed by a trigger clear command (10\textsuperscript{th} clock cycle).

The LDS instruction (Load Direct from Data Space) which is a Data Transfer instruction was first used to test the reaction of clock-glitch attack on it. This instruction is used to load one byte of data from the data space. By data space, here it means a combination of register files, I/O memory and internal SRAM. The instruction execution takes two clock cycles if it accesses I/O
memory, whereas, it takes one extra clock-cycle, i.e., a total of three clock-cycles for its execution if it accesses the internal SRAM. Hence, for the purpose of testing its behavior due to glitch attack when the instruction takes three execution cycles, LDS was coded to load one byte of data from the SRAM.

![Figure 5.9: Depiction of 6 nop clock cycles inserted between trigger set and a three clock-cycle test instruction execution](image_url)

As explained in the previous sections, here also different cases were considered depending on the relative position of the test instruction with respect to the trigger set command, and the glitches were applied accordingly. These cases are not listed here since these are very similar to the cases listed in the sub-section 5.2.1.

When the glitch is inserted in the negative half of the clock cycle previous to the LDS execution and in the positive half of the actual LDS execution cycle, the instruction gets skipped from being executed mostly for small Set 2 offsets and for small Set 1 offsets. This is because of the same reasons discussed in the sub-section 5.1.2.A and 5.1.2.B. When the glitch is inserted in the negative half of the 1st execution cycle with Set 1 offsets, instruction execution is very rarely getting affected, whereas, when the glitch is injected in the positive half of the 2nd execution cycle
with Set 2 offsets, the execution is getting affected mostly for small offset values. This may be because of reading wrong data from SRAM due to glitch insertion.

When the negative half of the 2nd and the positive half of the 3rd execution cycles are injected with clock glitch at different offsets, the actual output (0x00) is getting replaced by an irrelevant value (0x27) only for small values of Set 1 offsets, whereas, this output replacement takes place for any Set 2 offset values, thus entirely affecting the loading process in LDS execution. In the case when glitches with both the sets of offset values are inserted in the negative half of the final execution cycle and the positive half of the next clock cycle, the Result Write Back section (RWB) gets affected that produces constantly a wrong result 0x02 in place of the actual output 0x00.

Table 5.4: Summary of observations made by testing the effect of clock-glitch insertion in the clock cycles executing the triple-cycle test instructions and in the clock cycle prior to the test instruction execution.

<table>
<thead>
<tr>
<th>Glitch Offset</th>
<th>Glitch Cycle</th>
<th>Test instruction 3rd Execution Cycle (A)</th>
<th>Test instruction 2nd Execution Cycle (B)</th>
<th>Test instruction 1st Execution Cycle (C)</th>
<th>Cycle previous to Test Instruction Execution (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>Glitch affects RWB of Test Instruction for smaller offsets</td>
<td>Test Instruction outputs irrelevant values for small offsets</td>
<td>Test Instruction gets affected rarely for any offset</td>
<td>Test Instruction gets skipped only for smaller offsets</td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td>RWB gets affected for all offset values</td>
<td>Wrong values getting loaded from SRAM for smaller offsets</td>
<td>Test Instruction gets skipped for any offset value</td>
<td>Test Instruction gets skipped mostly for smaller offsets</td>
<td></td>
</tr>
</tbody>
</table>

The other three-clock cycle Data transfer instructions that showed similar behavior as that of LDS instruction towards clock-glitch attack are LD (Load Indirect), STS (Store Direct), ST (Store Indirect), LPM (Load Program Memory) and ELPM (Extended Load Program Memory). Apart from these Data transfer instructions, there are some Branching instruction as shown in the Table 5.1 that take 3 clock cycles for their execution depending on the branching condition, whose
behavior is similar to that explained in this sub-section when they utilize all the 3 cycles for their execution. The Table 5.4 given above summarizes the general behavior of all these triple-clock cycle instructions towards insertion of clock glitches into their pre-execution and execution cycles.

5.3 Clock-Glitch Attack on Caesar Cipher Algorithm - Test case

This section deals with a case study of attacking the XMEGA target device running a simple Caesar cipher algorithm using clock-glitching technique as explained in the previous sections. It deals with a more real-time application of employing the method of clock-glitching as discussed so far that uses the results obtained on attacking different types of assembly instructions to more accurately attack an encryption algorithm running in the target MCU. The aim here is not to obtain any secret data, but to show how a glitch insertion into a particular instruction being executed in any cryptographic algorithm would have its influence on the cipher text thus obtained.

The Caesar cipher is one of the earliest known and one of the simplest encryption algorithm. It is a type of substitution cipher in which each letter in the plaintext is shifted some fixed number of places down the alphabet. For example, with a right shift of 1, the alphabet ‘A’ would be replaced by the alphabet ‘B’, ‘B’ would become ‘C’, etc., whereas with a left shift of 2, ‘C’ would be replaced by ‘A’, ‘D’ would be replaced by ‘B’ and so on. Here, the length for which the shift process is taking place is considered as the key, and the negation of the encryption key is used to decrypt an encrypted message. For example, if the plain text “Message” is encrypted using Caesar cipher algorithm with ‘+3’ as the encryption key, then the cipher text would be “Phvvdjh” which can be then decrypted using the decryption key ‘-3’ to get back the plain text.

For the purpose of analyzing the effect of clock glitch over the cipher text generated using Caesar cipher algorithm, a C program that implements the desired algorithm is written and downloaded into the Atmel XMEGA target device connected to the ChipWhisperer-Lite main board. Since the algorithm is already known, the “trigger_high()” function that externally triggers the glitch insertion process is called just before the actual encryption step inside the code as shown in the figure 5.10 given below. As it can be seen clearly from the figure 5.10, the glitch trigger is called only while encrypting the upper case alphabets. This means, the glitch thus inserted would
affect those instructions such that the upper case alphabets in the plain text would be wrongly encrypted.

```c
void caesar_encrypt(char src[100], int key, int dist) {
    for (int i=0; i < dist; i++) {
        if (src[i] >= 'A' && src[i] <= 'Z') {
            src[i] = (char)(((src[i] + key - 'A' + 26) % 26) + 'A'); // Encryption as per key
        } else if (src[i] >= 'a' && src[i] <= 'z') {
            src[i] = (char)(((src[i] + key - 'a' + 26) % 26) + 'a'); // Encryption as per key
        }
    }
}
```

Figure 5.10: Caesar Cipher Encryption Algorithm. External glitch trigger set function, as encircled in red, is called inside the encryption block just before the main encryption step such that the glitch is inserted accordingly thus affecting the encryption process.

Now, consider the case of “Message” as the plain text and ‘+3’ to be the encryption key. The expected cipher text after encryption is “Phvvdjh”. But, when a clock glitch is inserted as mentioned before, the encryption of the upper case letter ‘M’ of the plain text alone would get affected resulting in producing any other letter or even any garbage value in place of the actual cipher letter ‘P’. This resulting output depends on exactly which instruction is getting affected and also on the position of the glitch being inserted with respect to that instruction. If the glitch trigger command in assembly language is adjusted in such a way that the glitch with Set 1 offset gets inserted into the execution cycle of the “add” instruction that adds the key to the plain text, then the plaintext ‘M’ might get replaced with any value other than ‘P’ since, according to the Table 5.1, the glitch at the described position could affect the RWB of the single-cycle “add” instruction resulting in producing some random output. If a glitch with Set 2 offset gets inserted into the execution cycle of the “add” instruction, then, again according to the Table 5.1, the instruction execution would get skipped, due to which the plaintext ‘M’ would not get encrypted and hence leading the code to producing the complete cipher text to be “Mhvvvdjh” in place of “Phvvdjh”.

48
Likewise, by adjusting the trigger set command in assembly version of the given code, the encryption of any plaintext can be attacked through clock-glitch insertion technique and hence affecting the cryptographic system. This method can be implemented on target device running any other cryptographic algorithms like DES, AES, RSA etc., and with little more efforts the secret information can also be retrieved.
CHAPTER VI

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The research presented in this thesis paper was performed to utilize an existing but comparatively new platform for performing clock-glitch attack on an 8-bit microcontroller, such that, the effect of a fault injection over different assembly level instructions can be observed. It is very important to know the effect of clock glitches over several instructions since some of these instructions might constitute a cryptographic function and thus, one can understand how an attacker could accordingly design his attack to retrieve the secret information from the device. By understanding this, the cryptographic algorithm can be modified such that the constituted instructions can be made more secure to such kinds of fault injection and hence the cryptographic data can be protected from such attacks.

This research used ChipWhisperer-lite board, that includes both clock and glitch generating FPGA and the Target device. ChipWhisperer is basically an open-source side channel attack and analysis platform for hardware security research. The target device consists of Atmel AVR XMEGA 128D4U MCU, which is an 8-bit microcontroller. A clock frequency of 7.37MHz is supplied to the target via Spartan-6 ZTEX LX-9 FPGA present in the ChipWhisperer-lite main board.
The basic hardware configuration, the parameters of the clock glitch and the type of glitch insertion trigger are set with the help of the Capture software provided by the ChipWhisperer. After a basic test, the glitch width was chosen to be 2.5% of the clock period, i.e., ~3.4ns, and two sets of glitch offsets were considered, one from -50% to 0% and other from 0% to 50% of the clock period. An external glitch trigger was set and it was initiated via the test code running in the target device. Different assembly level instructions were first grouped according the number of their execution clock cycles and were tested for the effect of clock glitch on them with the glitches been placed either before, during or after the instruction execution. The faulty outputs of each instruction with different cases of glitch insertion were observed and were compared with the actual expected outputs. The outputs affected by the glitch insertion have been reasoned and have been explained in the Chapter V.

6.2 Future Work

The experiments in this research have been performed on a target device running simple codes whose outputs directly correspond to the exact purpose of the test instruction being used. The results obtained here and the platform used in this research can be used in the near future to effectively attack the target device running a cryptographic algorithm to obtain the secret information. Also the attack can be performed over other devices that run at high-speed clock frequencies which are actually being used in different sectors that involve secret data storage and transfers.
REFERENCES

[1] https://hackaday.io/project/956-chipwhisperer-security-research


[27] https://wiki.newae.com/CW1173_ChipWhisperer-Lite#20-Pin_Connector

APPENDICES
APPENDIX A

TRIGGER DELAY ESTIMATION

A delay between setting the clock glitch insertion trigger and the point of actual glitch insertion is then assumed and termed as Trigger Delay.

In order to find the actual trigger delay and to know how the glitch placed at different offsets of clock cycles would affect the add instruction, an experiment is conducted. In this, first a particular number of nop instructions are inserted between the trigger set and the single cycle ALU instruction and then different trigger delay values are assumed. Since in previous case, many successful instruction skips due to glitch were observed while inserting 6 nop instructions, hence the trigger delay is presumed to be either 5, 6 or 7 clock cycles. The different cases considered here are listed below:

A. Five nop instructions insertion
   a. Five clock cycle trigger delay assumption
   b. Six clock cycle trigger delay assumption
   c. Seven clock cycle trigger delay assumption

B. Six nop instructions insertion
   a. Five clock cycle trigger delay assumption
   b. Six clock cycle trigger delay assumption
   c. Seven clock cycle trigger delay assumption

C. Seven nop instructions insertion
   a. Five clock cycle trigger delay assumption
   b. Six clock cycle trigger delay assumption
   c. Seven clock cycle trigger delay assumption

Let us now test the code by inserting 5, 6 and 7 nop instructions and infer the result by considering all the above given assumptions.
A. Five NOP Instructions Insertion

The figure A.1 shows the CPU clock diagram and fetching-execution of trigger set, 5 nop instructions, single cycle ALU instruction (add) and trigger clear. It also shows the sub-cycles involved in the execution of the ALU instruction that includes Register Operand Fetch (ROF), ALU operation execution and Result Write back (RWB).

For glitch insertion in all the cases, a glitch width of 2.5% of the CPU clock (7.37 MHz) is considered. The code is run for two sets of glitch offsets:

- **Set 1**: (-50 to 0) % of CPU clock, in this the glitch is inserted in the negative half of the previous clock cycle [8].
- **Set 2**: (0 to 50) % of CPU clock, in this the glitch is inserted in the positive half of the present clock cycle [8].

Now, the code with 5 nop instructions between the trigger set command and the add instruction is run assuming that the trigger set takes 5 clock cycles to take effect and the glitch is inserted in the 6th clock cycle. This code is tested for both the sets of glitch offsets as explained earlier and is shown in figure A.2, and the results are noted down. These results are then used to check the reliability of our assumption of 5 clock cycle trigger delay.

The results are also used to check for the possibilities of either 6 or 7 clock cycle trigger delays, as sown in figure A.3 and figure A.4.

A.a. Five clock cycle trigger delay assumption

As mentioned earlier, the figure A.2 shows the insertion of the clock glitches assuming 5 clock cycle trigger delay, while inserting 5 nop instructions between the trigger set and add instruction. The glitches are injected one at a time with a glitch width of 2.5% of clock cycle and for both the sets of glitch offsets (Set 1 and Set 2).
In this case, for the Set 1 offsets glitch insertion (A.a.1 in figure A.2), the ALU instruction will not get affected for more negative offsets (towards -50% offsets) because the instruction would still have enough time to fetch the register operands, to perform ALU execution and then to write back the result (RWB) into the corresponding register. But when the offsets become closer to 0%, the time available for the main ALU operation and RWB decrease, leading it to get skipped.
Since the ALU operation itself skips, there would not be any effect seen on the output register r24. Even in the intermediate case, when the ALU is performed but RWB is skipped, this would not affect r24 since the ALU output has not been written back into the register.

When the Set 2 offsets are used for inserting the glitches, the whole ALU instruction should get skipped due to the limited time availability for its execution.

A.b. Six clock cycle trigger delay assumption

The figure A.3 shows the insertion of the clock glitches assuming 6 clock cycle trigger delay, while inserting 5 nop instructions between the trigger set and add instruction. As in the previous case, the glitches are injected one at a time for both the sets of glitch offsets (Set 1 and Set 2).

![Figure A.3: Clock glitch Insertion at negative and positive halves of 6th and 7th clock cycles respectively assuming 6clock cycle trigger delay](image)

For this, when glitches are inserted using Set 1 offsets, the ALU operation would have already completed, but the result write back operation would either get skipped or would write some random value back into the register, leading to the register r24 either retaining its previous value or storing some other random value. In this case, the RWB would get affected for smaller offsets (close to 0%).

But for the Set 2 offset case, since the ALU instruction would have got executed completely in the last cycle, introducing a glitch here would not affect the ALU instruction.
A.c. Seven clock cycle trigger delay assumption

The insertion of the clock glitches assuming 7 clock cycle trigger delay, while inserting 5 nop instructions between the trigger set and add instruction is shown below in figure A.4. The glitch insertion and code testing processes are similar to as in the previous cases. As one can infer from the figure A.4, since for either sets of the offsets applied, the position of glitch insertion is irrelevant to actual ALU instruction execution cycle.

Hence one can very well observe that the ALU instruction performs smoothly and the glitch insertion would have no effect on it.

B. Six NOP Instructions Insertion

The figure A.5 shows the CPU clock diagram and fetching-execution of trigger set, 6 nop instructions, single cycle ALU instruction (add) and trigger clear. It also shows the sub-cycles involved in the execution of the ALU instruction that includes Register Operand Fetch (ROF), ALU operation execution and Result Write back (RWB). The configuration of the glitch to be inserted is the same as the previous case. The same 2 sets of offsets (Set 1 and Set 2) would be applied through the glitch module of ChipWhisperer Capture software.
Figure A.5: Depiction of 6 nop clock cycles inserted between Trigger set and ALU (inst) execution, and single cycle ALU (inst) execution sub-cycles

Now, 6 nop instructions would be inserted between the trigger set and the ALU instruction under test, the code would be run and the results would be analyzed for the 3 different trigger delay assumptions, as mentioned earlier.

**B.a. Five clock cycle trigger delay assumption**

The figure A.6 shows the insertion of the clock glitches while inserting 6 nop instructions between the trigger set and add instruction and assuming 5 clock cycle trigger delay. The glitches are injected one at a time with a glitch width of 2.5% of clock cycle and for both the sets of glitch offsets (Set 1 and Set 2).

For the Set 1 offsets glitch insertion (B.a.1 in figure A.6), the ALU instruction occurs a clock cycle after the position where the glitch is assumed to be injected. Hence, this should show no effect on the ALU operation.
But in the case of Set 2 offsets (B.a.2 in figure A.6) glitch insertion, even though the glitch is in the previous clock cycle to the ALU instruction execution, the portion of the 6th clock cycle shown in figure A.6 after the glitch acts as the 7th clock cycle, where the ALU execution would take place. When the offset value increases (towards 50%), the total time period of the new 7th clock cycle decreases, hence decreasing the time availability for the ALU instruction execution. This would lead to skipping of the ALU instruction execution for glitches inserted with larger offset values.

**B.b. Six clock cycle trigger delay assumption**

In the case of assuming the trigger delay to be 6 clock cycles and inserting 6 nop instructions between the trigger_high() and the ALU instruction, the glitches with Set 1 offsets (B.b.1 in figure A.7) would be injected at the ALU instruction fetching cycle, that is just before the actual ALU execution cycle. In case of more negative offsets (towards -50%), there would be enough time available for the ALU execution to take place. But, since the ALU execution would start in the rising edge of this negative glitch and should end while encountering the next rising clock edge, when the offset values get close to 0%, the limited time would not be enough for its execution and as a result the entire ALU instruction would get skipped.
Figure A.7: Clock glitch insertion at negative and positive halves of 6th and 7th clock cycles respectively assuming 6 clock cycle trigger delay

For the **Set 2** offsets case (B.b.2 in figure A.7), the glitches assumed to be injected exactly over the ALU instruction execution cycle, thus, due to the limited time availability for execution, the entire ALU operation would get skipped for almost any value of offset used.

**B.c. Seven clock cycle trigger delay assumption**

This case, as depicted in the figure A.8, is similar to the case discussed in the Section A.b and should show the same results as in that section.

The **Set 1** offset glitch insertion (B.c.1 in figure A.8) would affect the RWB for offsets closer to 0% of CPU clock and hence some random value would be stored in the output register r24. Whereas, there would be no effect on the ALU instruction seen for the **Set 2** offsets glitch insertion (B.c.2 in figure A.8) since the glitch is assumed to be applied only after the complete execution of the ALU instruction.
C. Seven NOP Instructions Insertion

The figure A.9 shows the CPU clock diagram and fetching-execution of trigger set, 7 nop instructions, single cycle ALU instruction (add) and trigger clear. It also shows the sub-cycles involved in the execution of the ALU instruction that includes Register Operand Fetch (ROF), ALU operation execution and Result Write back (RWB).

Now, 7 nop instructions would be inserted between the trigger set and the ALU instruction under test, the code would be run and the results would be analyzed for the 3 different trigger delay assumptions, as mentioned earlier.

C.a. Five clock cycle trigger delay assumption

The figure A.10 depicts the scenario where the clock glitch with the 2 sets of offsets (Set 1 and Set 2) are injected when 7 nop instructions are inserted between the trigger set and the ALU instruction, assuming that the trigger set command takes 5 clock cycles to take effect. Since in either case, as anyone who sees the figure A.10 could realize, the ALU operation is occurring much after the time of glitch insertion, neither the set 1 offset nor the set 2 offset glitch insertion would affect the instruction execution.
Figure A.9: Depiction of 7 nop clock cycles inserted between Trigger set and ALU (inst) execution, and single cycle ALU (inst) execution sub-cycles

Figure A.10: Clock glitch Insertion at negative and positive halves of 5th and 6th clock cycles respectively assuming 5clock cycle trigger delay

**C.b. Six clock cycle trigger delay assumption**

The glitch insertion, in the case of the Set 1 offsets, as indicated by the arrow C.b.1 in the figure A.11, should not affect the ALU instruction execution, since this glitch is assumed to be inserted a clock cycle before the actual execution of the instruction under test.
But the case of Set 2 offsets (C.b.2 in figure A.11) glitch insertion is the same as that explained in the Section B.a. Even though the glitch is in the previous clock cycle to the ALU instruction execution, the portion of the 7th clock cycle shown in figure A.11 after the glitch acts as the new 8th clock cycle, where the ALU execution should take place. This means, the execution of the ALU instruction would take place early due to the skipping of its previous nop instruction (nop 7). When the offset value increases (towards 50%), the total time period of the new 8th clock cycle decreases, hence decreasing the time availability for the ALU instruction execution. This would lead to skipping of the ALU instruction execution for glitches inserted with larger offset values.

![Figure A.11: Clock glitch insertion at negative and positive halves of 6th and 7th clock cycles respectively assuming 6clock cycle trigger delay](image)

**C.c. Seven clock cycle trigger delay assumption**

The concept depicted by the figure A.12 is very much similar to that explain in the Section A.a and the Section B.b, except for a small change that in this case 7 nop instructions have been inserted between the trigger set command and the instruction under test, and the trigger delay is presumed to be 7 clock cycles.

For the Set 1 offsets case, represented by C.c.1 in figure A.12, the ALU instruction would get skipped for smaller offsets (close to 0% from left), whereas, for the Set 2 offsets case, represented by C.c.2 in the above figure, the instruction under test should get skipped for almost all the offset values. The reason for this is the same as that is explained in the Section A.a and the Section B.b.
Conclusion

An experiment was conducted in ChipWhisperer Capture software using ChipWhisperer-lite board and the XMEGA target device to find the actual trigger delay between the point of trigger set and the actual insertion of clock glitch into the XMEGA assembly instruction clock cycle. This is performed also to know how the glitch placed at different offsets of clock cycles would affect a single-cycle ALU instruction. For this, different cases where considered with different trigger delay assumptions, the corresponding assembly code were executed and the results were noted and analyzed. The below given table (Table A.1) summarizes the presumptions made in Sections A through C.

<table>
<thead>
<tr>
<th>Trigger delay Assumption</th>
<th>5 nop instructions</th>
<th>6 nop instructions</th>
<th>7 nop instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Set 1</td>
<td>Set 2</td>
<td>Set 1</td>
</tr>
<tr>
<td>5 clock cycle delay</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Skips for smaller offsets</td>
<td></td>
<td>Not affected</td>
</tr>
<tr>
<td>6 clock cycle delay</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Affects RWB for smaller offsets</td>
<td>Not affected</td>
<td>Skips for smaller offsets</td>
</tr>
<tr>
<td>7 clock cycle delay</td>
<td>Not affected</td>
<td>Not affected</td>
<td>Affects RWB for smaller offsets</td>
</tr>
</tbody>
</table>

Table A.1: Prediction based on analysis of how single-cycle ALU instruction gets affected due to clock glitch at various offsets assuming different Trigger delays
Comparing the Table A.1, that describes the possible results, with the Table A.2, that shows the actually observed results, one can prove that the actual delay between the point of trigger set and the point of actual clock glitch insertion (i.e., the Trigger Delay) is Six Clock Cycles.
APPENDIX B

CHIPWHISPERER CLOCK GLITCH EXAMPLE-CODES

B.1 main() function of glitch-simple.c code

```c
int main(void){

    platform_init();
    init_uart();
    trigger_setup();

    /* Uncomment this to get a HELLO message for debug */
    putch('h');
    putch('e');
    putch('l');
    putch('l');
    putch('o');
    putch('
');
    _delay_ms(20);

    while(1){
        glitch_infinite();    //Different glitch functions are called here
    }

    return 1;
}
```

B.2 glitch_infinite() function of glitch-simple.c code

```c
void glitch_infinite(void)
{
    char str[64];
    //Declared volatile to avoid optimizing away loop.
    //This also adds lots of SRAM access
    volatile uint16_t i, j;
    volatile uint32_t cnt;
    while(1){
        cnt = 0;
        for(i=0; i<500; i++){
            for(j=0; j<500; j++){
                cnt++;
            }
        }
        sprintf(str, "%lu %d %d\n", cnt, i, j);
        uart_puts(str);
    }
}
```
void glitch1(void)
{
    led_ok(1);
    led_error(0);

    //Some fake variable
    volatile uint8_t a = 0;

    putch('A');

    //External trigger logic
    trigger_high();
    trigger_low();

    //Should be an infinite loop
    while (1) {
    
    }
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);

    uart_puts("1234");

    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);
    led_error(1);

    //Several loops in order to try and prevent restarting
    while(1){
    
    }
    while(1){
    
    }
    while(1){
    
    }
    while(1){
    
    }
    while(1){
    
    }
void glitch3(void)
{
    char inp[16];
    char c = 'A';
    unsigned char cnt = 0;
    uart_puts("Password:");

    while((c != 'n') & (cnt < 16)){
        c = getch();
        inp[cnt] = c;
        cnt++;
    }

    char passwd[] = "touch";
    char passok = 1;
    trigger_high();
    trigger_low();

    //Simple test - doesn't check for too-long password!
    for(cnt = 0; cnt < 5; cnt++){
        if (inp[cnt] != passwd[cnt]){
            passok = 0;
        }
    }

    if (!passok){
        uart_puts("Denied\n");
    } else { 
        uart_puts("Welcome\n");
    }
}