Injecting Chaos: Fault Attack Techniques and Analysis via ChipWhisperer

Thesis submitted to the

Indian Institute of Technology, Bhilai

For award of the degree

of

Master of Technology

by

Chayan Pathak

ID: 12310630

Under the guidance of

Dr. Dhiman Saha

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY BHILAI

June 2025

I **dedicate** this thesis to my best friend,
Arunima Masanta, in recognition of her
unwavering support, loyalty, and encouragement.
Her presence has made this journey meaningful
and fulfilling.

Declaration

I Certify that

a. The work contained in the thesis is original and has been done by me under the guidance of my Supervisor;

b. The work has not been submitted to any other Institute for any degree or diploma;

c. I have followed the guidelines provided by the Institute in preparing the thesis;

d. I have conformed to ethical norms and guidelines while writing the thesis and;

e. whenever I have used materials (data, models, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis, giving their details in the references, and taking permission from the copyright owners of the sources, whenever necessary.

Date: ______________________________

Place: IIT Bhilai Chayan Pathak

Certificate

This is to certify that I have examined the thesis entitled **"Injecting Chaos: Fault Attack Techniques and Analysis via ChipWhisperer"**, submitted by **Chayan Pathak**(Roll Number: 12310630) a postgraduate student of Department of Computer Science and Engineering in partial fulfillment for the award of degree of Master of Technology. I hereby accord my approval of it as a study carried out and presented in a manner required for its acceptance in partial fulfillment for the Post Graduate Degree for which it has been submitted. The thesis has fulfilled all the requirements as per the regulations of the Institute and has reached the standard needed for submission.

Date:
Place: IIT Bhilai

Dr. Dhiman Saha
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bhilai

CERTIFICATE AGAINST PLAGIARISM

This is to certify that the thesis entitled **Injecting Chaos: Fault Attack Techniques and Analysis via ChipWhisperer**, submitted by **Chayan Pathak**, a bonafide record of the research work, done under my supervision is submitted to the Indian Institute of Technology Bhilai on **June 7, 2025** to meet the requirements for the award of **Master of Technology** degree.

The contents of the thesis have been verified through similarity check software and I am convinced that the thesis has no component of plagiarism.

Date:
Place: IIT Bhilai

Dr. Dhiman Saha
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Bhilai

I have personally verified the report and found no significant similarity in the thesis.

Date:
Place: IIT Bhilai

Dr. Anand Baswade
Convener, DPGC
Department of Computer Science and Engineering
Indian Institute of Technology Bhilai

Acknowledgements

I would like to express my sincere gratitude to my guide, Dr. Dhiman Saha, for his invaluable guidance, constant support, and encouragement throughout the course of my research. His expert advice and insightful feedback have been instrumental in the successful completion of this thesis.

I am deeply thankful to my parents for their unwavering love, patience, and motivation. Their constant belief in me has been my greatest strength.

I also extend my heartfelt thanks to my friends for their companionship, support, and encouragement during this journey.

My sincere appreciation goes to all the staff members at IIT Bhilai for providing a conducive environment and necessary facilities to carry out my research smoothly.

Finally, I am grateful to the members of the de.ci.phe.red Lab for their cooperation, insightful discussions, and friendly atmosphere that enriched my research experience.

Chayan Pathak

IIT Bhilai

COPYRIGHT TRANSFER CERTIFICATE

Title of the thesis:	**Injecting Chaos: Fault Attack Techniques and Analysis via ChipWhisperer**
Name of the Student:	**Chayan Pathak**

Copyright Transfer

Date:
Place: IIT Bhilai

Signature of the Student

Abstract

As embedded systems continue to play a vital role in secure communications and critical infrastructure, their vulnerability to physical attacks poses a growing security concern. Among these, fault injection attacks have gained prominence due to their effectiveness in bypassing cryptographic protections by introducing transient faults during execution. This thesis presents a comprehensive framework for setting up a practical fault injection laboratory using the open-source ChipWhisperer platform.

The work begins with the configuration and evaluation of various ChipWhisperer hardware modules, including scope and target boards such as the CWLite and CWHusky, alongside integrated targets. It further details the associated software environment, APIs, and firmware organization required to carry out repeatable and precise experiments. Clock and voltage glitching techniques are implemented to conduct real-time fault attacks in cryptographic routines.

To validate the setup, a real-world fault attack on AES-128 is reproduced, showcasing fault injection at critical rounds of encryption. Additionally, the thesis explores a custom attack against the BipBip cipher and initiates fault testing on the Kyber post-quantum cryptosystem, demonstrating the lab's versatility in evaluating both legacy and modern cryptographic schemes.

By documenting each stage from setup to execution this work offers a practical guide for researchers and security analysts interested in active hardware-based attacks. It also contributes experimental insights that can support the design of more resilient embedded systems in the face of evolving physical threats.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

In an increasingly interconnected world, embedded systems are deeply integrated into the fabric of modern life. From smart cards and smartphones to industrial control units and medical devices, these systems frequently handle sensitive operations such as cryptographic computations, secure communications, and user authentication. As a result, the security of embedded devices is not merely a technical concern, it is a critical requirement.

While traditional cybersecurity research has focused largely on software vulnerabilities, physical attacks on hardware have emerged as a powerful and practical threat vector. Among these, *fault injection attacks* have proven particularly effective. By deliberately introducing transient faults into a device during its operation, attackers can disrupt normal execution paths, bypass security mechanisms, and extract secret data such as cryptographic keys. These attacks are low-cost, non-invasive in many cases, and difficult to defend against without dedicated countermeasures.

This thesis focuses on the design, implementation, and evaluation of a fault injection laboratory based on the ChipWhisperer platform—a widely-used open-source toolkit for hardware security research. The core objective is to enable reproducible experiments that investigate the vulnerability of cryptographic algorithms under fault conditions and assess the effectiveness of various defenses.

Scope of the Work

Chapter 3 of this thesis introduces the complete setup of a fault injection laboratory using ChipWhisperer. It discusses the hardware components—including various scope and target boards—and explains how each contributes to the overall experimentation process. Special attention is given to popular configurations such as the ChipWhisperer-Lite, Nano, and Husky, which offer varying capabilities in terms of precision, flexibility, and ease of integration.

The software environment is covered in detail, providing a breakdown of the ChipWhisperer directory structure and its Application Programming Interfaces (APIs) for both scopes and targets. Readers are guided through the steps required to set up, configure, and run fault injection experiments using voltage and clock glitching techniques.

In Chapter 4, the thesis replicates a state-of-the-art diagonal fault attack on AES-128 to demonstrate how theoretical attack models translate into practical exploitation. By injecting faults at strategic points in the encryption process, the experiments show how an attacker could compromise a secure system, even with limited access to internal states.

Chapter 5 applies these methods to a lesser-known cryptographic algorithm, BipBip, to highlight how fault injection techniques can be generalized beyond well-studied targets. The chapter presents a full experimental pipeline—from attack strategy to results analysis—showcasing the lab's versatility.

Finally, Chapter 6 transitions the discussion to post-quantum cryptography (PQC), with a focus on Kyber, one of the leading candidates in the Post Quantum Cryptographic(PQC) standardization process of NIST . This chapter explores the challenges of preparing next-generation cryptographic algorithms for fault analysis and outlines a roadmap for future research.

Contributions Of The Thesis

The key contributions of this work are:

- A well documented, reproducible setup for conducting fault injection attacks using the ChipWhisperer platform.
- Implementation and evaluation of real-world fault attacks on AES-128, including both voltage and clock glitching methods.
- Novel fault analysis on BipBip cipher, contributing original experimental findings to the community.
- Initial groundwork toward fault injection studies on post-quantum cryptographic algorithms, using Kyber as a case study.

As physical attacks on embedded systems continue to evolve, the ability to experimentally evaluate their impact becomes increasingly important. This thesis aims to bridge the gap between academic theory and hands-on testing by offering a detailed guide to setting up and using a fault injection laboratory. The results not only validate known attack strategies but also offer a foundation for future explorations in hardware security—especially as we transition into the post-quantum era.

CHAPTER 2

Literature Survey

Fault injection attacks (FIAs) have emerged as a potent threat to embedded systems, exploiting hardware-level vulnerabilities through techniques such as voltage and clock glitching. These attacks can compromise cryptographic implementations by inducing faults that alter a devices normal execution flow. This section explores the major studies and developments in the field.

Non-Invasive Fault Injection Techniques

Mazumder et al. [SZFT23] present a detailed survey of non-invasive fault injection methods, including voltage, clock, and electromagnetic techniques. These methods allow adversaries to disrupt device operations without direct physical tampering, making them highly relevant in real-world attack scenarios.

Practicality of FIAs

Breier and Hou [BH22] examine the practicality of executing fault attacks across various architectures. Their work demonstrates that low-cost setups can achieve successful fault injections on widely used processors, raising concerns about the accessibility of such attacks.

Software-Level Implications

Yuce et al.[YSW20] focus on how FIAs affect embedded software systems. They detail how instruction and data flows can be corrupted, potentially bypassing authentication or exposing sensitive information.

Evaluation of Fault Injection Tools

Brito et al. [B+23] conduct a comparative evaluation of fault injection platforms, helping researchers select suitable tools for reliability testing of embedded systems under attack conditions.

Application to Post-Quantum Cryptography

Hermelink et al. [H+23] investigate the resilience of post-quantum algorithms, particularly Kyber and Dilithium, against both side-channel and fault injection attacks. This research is crucial as the cryptographic landscape shifts towards quantum-resistant designs.

CHAPTER 3

Setting Up a Fault Injection Lab with ChipWhisperer

In recent years, fault injection attacks have become a prominent area of study within the hardware security field due to their ability to compromise embedded systems in subtle yet powerful ways. By deliberately introducing faults during the execution of critical operations—particularly those involving cryptographic algorithms—attackers can exploit system weaknesses that remain hidden under normal conditions. As research into these attack vectors evolves, the importance of a dependable, adaptable, and thoroughly documented lab environment has grown significantly. A well-constructed experimental setup not only ensures consistent and reproducible results but also facilitates the design and validation of effective hardware-level countermeasures.

This chapter outlines the laboratory environment specifically built to conduct fault injection experiments using the ChipWhisperer platform. Known widely in the academic and professional security communities, ChipWhisperer offers a robust suite of tools tailored for side-channel analysis and glitch-based fault injection. One of its major strengths lies in its integrated approach—combining hardware modules with a powerful software interface to deliver a cohesive and highly controllable experimental experience.

ChipWhisperer [New25a] supports a range of techniques for fault injection, most notably Clock glitching and Voltage glitching fault injection. These methods can induce controlled errors in the behavior of a target device, often revealing vulnerabilities in cryptographic implementations or exposing unexpected execution paths. The platform's ability to fine-tune fault parameters, such as glitch width, offset, and repetition, is essential for conducting systematic attacks and for understanding how and when systems fail under abnormal conditions.

The lab setup described in this chapter includes a detailed walkthrough of the ChipWhisperer's hardware components, including the capture board, target devices, and power supply configurations. Special attention is paid to key operational aspects such as signal synchronization, trigger alignment, and the calibration of timing parameters, all of which are vital to the success of glitch-based attacks. Safety considerations, especially when dealing with fault injection at the physical level, are also addressed to ensure both operator protection and equipment longevity.

By documenting each step of the setup process, this chapter aims to provide a practical guide for researchers and practitioners who wish to replicate or build upon this work. Whether the goal is to explore new fault injection strategies, assess the resilience of embedded systems, or develop novel countermeasures, a clear understanding of the experimental foundation is critical. In doing so, this chapter contributes to the broader objective of advancing transparency, repeatability, and rigor in the field of hardware security research.

Techniques and analysis: Side-Channel Attacks Using ChipWhisperer

One of ChipWhisperer's key strengths lies in its ability to perform highly controlled fault injection techniques, most notably voltage glitching and clock glitching. With voltage glitching, the platform can momentarily drop or distort the power supply to a target device, causing it to behave unpredictably, potentially skipping instructions or bypassing security checks. Clock glitching, on the other hand, involves inserting short, precisely-timed disturbances into the device's clock signal. These glitches can disrupt the timing of operations, often triggering exploitable faults during critical processes like encryption or authentication. The fine-grained control over timing, width, and offset of these glitches makes ChipWhisperer exceptionally effective for exploring fault-induced vulnerabilities. Combined with its trace capture and analysis tools, these capabilities allow researchers to both induce and observe faults in real time, offering deep insight into how embedded systems respond under non-ideal operating conditions.

3.1 ChipWhisperer Hardware Platform

The ChipWhisperer platform is made up of specialized hardware designed to carry out side-channel analysis and fault injection on embedded devices. Its hardware setup is typically divided into two main parts: scope boards, which handle signal capture and glitching, and target boards, which run the code being tested. When used together, these components form a complete and flexible environment for testing the security of cryptographic systems and exploring hardware vulnerabilities.

3.1.1 Scope Boards (Capture Hardware)

Scope boards are at the heart of ChipWhisperer's side-channel capture capabilities. They serve as the interface between the host computer and the target device, enabling the collection of high-resolution power traces or electromagnetic emissions during cryptographic operations. These boards also provide fine-grained control over clock generation, triggering, and synchronization.

A popular example is the ChipWhispererLite(CWLite), an all-in-one board combining scope and target functionality, ideal for students and researchers. ChipWhispererHusky(CWHusky) offer higher sampling rates, more precise timing, and expanded features such as fault injection support. Key features of scope boards:

- Adjustable clock generation and distribution
- Trigger input/output for synchronization with DUT
- High-speed ADCs for capturing power consumption
- Communication interfaces (USB, serial, etc.)

Example usage scenario: Capturing power traces during AES encryption for differential power analysis (DPA).

3.1.2 Target Boards (Device Under Test - DUT)

Target boards, or Devices Under Test (DUTs), are embedded systems that execute the cryptographic or security-related code being analyzed. These boards are directly connected to the ChipWhisperer scope, allowing researchers to monitor power usage, apply glitches, and assess how the target responds to side-channel or fault injection attacks.

ChipWhisperer supports a variety of target boards suited for both beginners and advanced users. These targets range from basic microcontrollers to FPGA-based systems, making the platform highly flexible for different types of experiments.

Commonly used targets include:

- CW308 UFO Board: A modular baseboard [New16b] designed to support interchangeable target modules. It simplifies switching between different microcontrollers or chips without modifying the core setup.

- XMEGA Target: An Atmel XMEGA microcontroller board, commonly used for introductory experiments in side-channel analysis. It includes a reference AES implementation for easy testing.

- STM32F3 and STM32F4 Targets: ARM Cortex-M microcontrollers that are widely used in real-world applications. These targets are ideal for testing modern firmware under realistic fault or side-channel attack conditions.

- Artix-7 FPGA Target: A powerful platform [New16a] that allows the implementation and testing of custom cryptographic cores or hardware countermeasures. Suitable for advanced research in hardware security.

- SAM4S Target: Based on the ARM Cortex-M4 architecture, this board provides a more complex environment for evaluating side-channel resistance in higher-performance embedded systems.

Each of these targets includes features such as external clock input, adjustable power settings, and standard programming or debug interfaces (e.g., JTAG, SWD, UART). Together, they provide a robust environment for evaluating cryptographic implementations against a range of attack techniques.

3.1.3 Integrated Target on ChipWhisperer-Nano

The CWNano is a small and low-cost tool 3.1 used to learn about hardware security. It has both a target microcontroller and a measurement unit on a single board, making it easy to use. The built-in STM32F030F4P6 microcontroller has 16 KB of flash and 4kB of SRAM, suitable for running simple cryptographic algorithms. The board supports power analysis using an 8-bit ADC with a sampling rate of up to 20 MS/s. It also features basic voltage glitching through a crowbar method. The Nano connects to a computer via USB and is controlled using the ChipWhisperer software suite. It does not support clock glitching and has a limited flash size, but it is ideal for beginners, students, and hobbyists exploring side-channel analysis and embedded security. More details about this can be found in the official documentation [New25d].

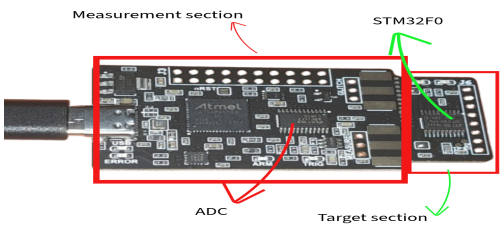


Figure 3.1: ChipWhisperer-Nano

3.1.4 Integrated Target on ChipWhisperer-Lite

The CWLite is a compact and low-cost hardware tool designed to help users learn about hardware security. It combines a power measurement system and a microcontroller target on a single board, making it easy to use. This device [New25e] can capture power traces with high speed and allows for glitching attacks like clock or voltage glitches. It is mainly used to perform and study side-channel attacks, such as breaking encryption by analyzing power usage. The board connects to a computer through USB and works with open-source ChipWhisperer software. Because of its simplicity and low price, it is widely used in education and research.
It is offered in four hardware options [New25c] to suit different learning and testing needs. The first version is the CWLite-XMEGA, which includes both the capture hardware and an ATxmega128D4 microcontroller on a single board. The second option, CWLITE-ARM, features an STM32F3 ARM Cortex-M4 microcontroller instead of the XMEGA, also on a single board as shown in figure 3.2.
The CWLite 2-Part version as shown in figure 3.3 separates the capture and target sections into two distinct boards, connected via SMA and 20-pin IDC cables. This modular design allows users to easily swap or upgrade target devices, enhancing flexibility for various testing scenarios. The capture board features a 10-bit ADC with a maximum sample rate of 105 MS/s, adjustable gain up to +55 dB, and supports both voltage and clock glitching with fine-grained control. The target board includes an ATxmega128D4 microcontroller, suitable for implementing and analyzing cryptographic algorithms. This two-part

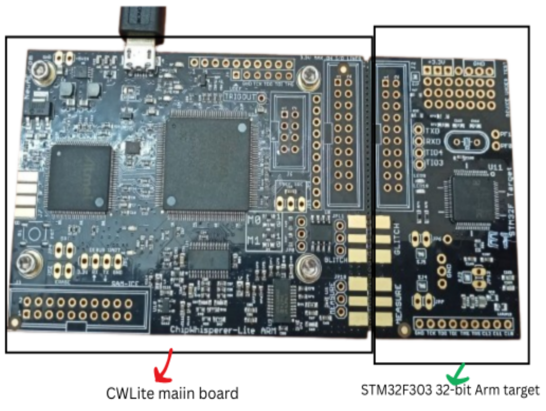


Figure 3.2: ChipWhisperer-Lite with 32-bit ARM target

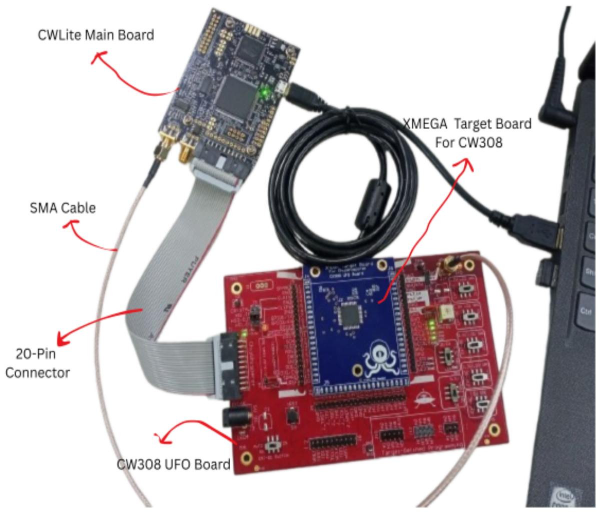


Figure 3.3: ChipWhisperer-Lite 2-Part version connections

configuration is ideal for users seeking a versatile and open-source platform for embedded hardware security research.
The CWLite Standalone is a specialized capture board designed for side-channel power analysis and fault injection experiments. Unlike other variants, it does not include an integrated target microcontroller, providing flexibility to connect external targets via standard interfaces. It is ideal for researchers and educators who wish to interface with custom or third-party targets while utilizing ChipWhisperer's powerful capture and analysis capabilities.

3.1.5 ChipWhisperer-Husky

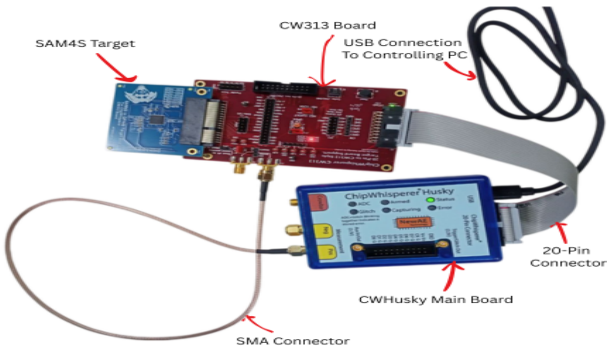


Figure 3.4: ChipWhisperer-Husky with SAM4S target

Together, the scope and target boards in the ChipWhisperer hardware ecosystem offer a comprehensive solution for conducting side-channel and fault injection attacks. Their modular, open-source nature makes them accessible to students, researchers, and engineers alike, and they are widely used in academic studies, industry evaluations, and security training environments.

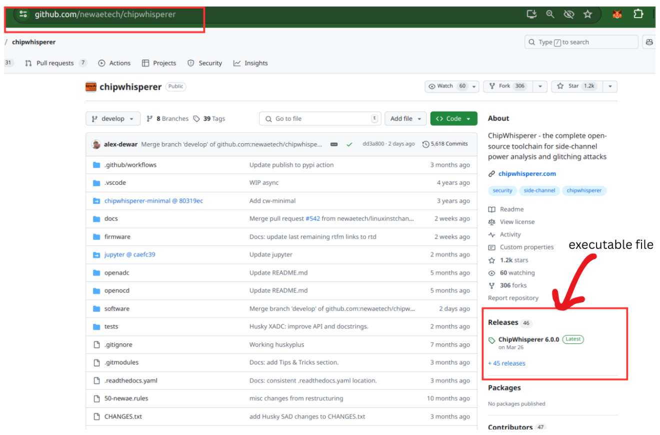
3.2 Software Environment

The ChipWhisperer platform includes a flexible and scriptable software environment built primarily around Python and Jupyter Notebooks. This environment is well-suited for side-channel analysis and fault injection research, offering full control of the hardware and reproducibility.

Setting up the ChipWhisperer software environment involves installing the necessary tools to interface with the hardware, control experiments, and analyze captured data. This section outlines the steps required to install the ChipWhisperer software stack using the recommended method.

For Windows :

Installing ChipWhisperer on Windows is straightforward with the provided ėxe file [New25a]. Once the download is complete, double-click the .exe file to launch the installer. If a security prompt appears, allow the installation to proceed. The ChipWhisperer Setup Wizard will open—follow the on-screen instructions by clicking "Next." Choose a destination folder or use the default option, then continue by clicking "Next" again. After the installation is finished, click "Finish" to complete the setup. The simplest way to start using



alex-dewar Merge branch 'develop' of github.com:newaetech/chipwhispe... dd3a800 · 2 days ago 5,618 Commits

.github/workflows	Update publish to pypi action	3 months ago
.vscode	WIP async	4 years ago
chipwhisperer-minimal @ 80319ec	Add cw-minimal	3 years ago
docs	Merge pull request #542 from newaetech/linuxinstchan...	2 weeks ago
firmware	Docs: update last remaining rtfm links to rtd	2 weeks ago
jupyter @ caefc39	Update jupyter	2 months ago
openadc	Update README.md	5 months ago
openocd	Update README.md	5 months ago
software	Merge branch 'develop' of github.com:newaetech/chipw...	2 days ago
tests	Husky XADC: improve API and docstrings.	2 months ago
.gitignore	Working huskyplus	7 months ago
.gitmodules	Docs: add Tips & Tricks section.	3 months ago
.readthedocs.yaml	Docs: consistent .readthedocs.yaml location.	3 months ago
50-newae.rules	misc changes from restructuring	10 months ago
CHANGES.txt	add Husky SAD changes to CHANGES.txt	3 months ago

About

ChipWhisperer - the complete open-source toolchain for side-channel power analysis and glitching attacks

chipwhisperer.com

security side-channel chipwhisperer

Readme
View license
Activity
Custom properties
1.2k stars
60 watching
306 forks
Report repository

Releases 46
ChipWhisperer 6.0.0 Latest
on Mar 26
+ 45 releases

Packages
No packages published

Contributors 47

Figure 3.5: Chipwhisperer official GitHub page

ChipWhisperer and access its tutorials is to open the ChipWhisperer application. You can find it in the Start Menu, the installation directory, or on your desktop if you chose to create a shortcut during setup.

for Linux:

To install ChipWhisperer on a Linux-based system, follow these steps [New25b]:

Step 1: Update System Packages

```
sudo apt update && sudo apt upgrade
```

Step 2: Install Required Dependencies

```
sudo apt install libusb-dev gcc-arm-none-eabi make git
 avr-libc gcc-avr avr-libc libusb-1.0-0-dev usbutils
python3 python3-venv
```

Step 3: Clone the Repository

```
cd ~/
git clone https://github.com/newaetech/chipwhisperer
cd chipwhisperer
```

Step 4: Set Up Python Virtual Environment

```
python3 -m venv ~/.cwvenv
source ~/.cwvenv/bin/activate
```

Step 5: Install USB Rules

```
sudo cp 50-newae.rules /etc/udev/rules.d/
sudo udevadm control --reload-rules
```

Step 6: Add User to Required Groups

```
sudo groupadd -f chipwhisperer
sudo usermod -aG chipwhisperer $USER
sudo usermod -aG plugdev $USER
```

Step 7: Initialize Submodules and Install Python Packages

```
git submodule update --init jupyter
python -m pip install -e .
python -m pip install -r jupyter/requirements.txt
```

Step 8: Reboot System
Once the setup is complete, reboot your system to apply changes.
Step 9: Launch Jupyter Notebooks

```
cd ~/chipwhisperer
jupyter notebook .
```

3.2.1 Directory Structure

The ChipWhisperer framework is a comprehensive toolkit for side-channel power analysis and glitching attacks. Its directory structure is designed to support both hardware interfacing and educational experimentation. Understanding this structure is key to navigating the project efficiently, especially when working with Jupyter notebooks or compiling firmware.

Here's a breakdown of the most relevant parts:

jupyter/ Directory :

One of the most user-friendly components of the ChipWhisperer project is the `jupyter/` directory. This folder contains a wide range of Jupyter Notebooks designed to guide users through hands-on tutorials and experiments.

These notebooks provide interactive lessons in topics such as: Side-channel analysis, Cryptographic attacks, Glitching techniques The content within the `jupyter/` directory is well-organized into subfolders, such as:

The Jupyter directory in ChipWhisperer includes several folders designed to support learning and experimentation. The `demos` folder contains simple, hands-on notebooks meant for beginners. These notebooks introduce key concepts like power analysis and fault injection through guided examples. The `courses` folder offers more structured content, similar to academic classes, with lecture-style explanations and lab exercises for deeper learning. The `setupscripts` folder provides example scripts and configurations that demonstrate different experiments and usage scenarios for ChipWhisperer hardware. Together, these resources help users understand and explore various aspects of side-channel analysis and hardware security.

firmware/ Directory:

The `firmware/` directory in the ChipWhisperer repository contains all the source code necessary to build firmware for a variety of supported target devices. This firmware is essential for running side-channel analysis and fault injection experiments, as it enables precise control and repeatable behavior during cryptographic operations on the Device Under Test (DUT).

The `firmware/` folder is organized into several subdirectories that correspond to specific microcontroller families, target boards, or cryptographic examples. `firmware/mcu` contains Cryptographic firmware examples such as AES and RSA with SimpleSerial protocol support for host-target communication.

3.2.2 Scope API

The `scope` object in ChipWhisperer is used to manage the capture and glitching operations of the hardware.The official documentation [New25f] provides a comprehensive overview of the scope API, detailing how to configure the hardware, capture traces, and perform glitching operations. The scope API is designed to be intuitive and flexible, allowing users to easily set up their experiments and interact with the ChipWhisperer hardware.

To create a `scope` object, the `chipwhisperer.scope()` function is the easiest approach to use. This function connects to a ChipWhisperer device and returns an instance of the appropriate scope object:

```
import chipwhisperer as cw
scope = cw.scope()
```

There are two types of `scope` : OpenADC for `CWLite/Husky` and CWNano for `CWNano`. The choice of scope depends on the specific hardware being used. The `scope` object provides various properties and methods to configure the hardware, capture traces, and perform glitching operations. Here are some of the key properties and methods:

`scope.adc.samples` **:**

This property sets the number of samples to capture by the ADC (Analog-to-Digital Converter). It is useful for determining the length of the captured trace. maximum number of samples for Lite is 244000 and for Husky is 131070

`scope.adc.timeout` **:**

Specifies the maximum duration (in seconds) the ADC will wait for a trigger signal before aborting the capture. This prevents indefinite waiting during a capture session.

$scope.clock.adc\_src$:

Determines the clock source for the ADC module. Generally it will be $clkgen\_x1$ or $clkgen\_x4$ which correspond to different clock frequencies.

`glitch.clk_src` **:**

This setting selects the clock source used by the glitch module's DCM (Digital Clock Manager). The DCM determines the timing of when glitches are produced. The available clock sources are:

- clkgen: Uses the output from the internal clock generator. *Note: This option is not supported on Husky.*
- pll: Uses the on-board Phase-Locked Loop (PLL) available on Husky devices. *Note: This is specific to Husky.*

Selecting the correct clock source is important for ensuring that glitches are timed correctly with the target device's operation.

scope.glitch.output :

This setting controls the type of signal produced by the glitch module. It defines how the regular clock and glitch pulses are combined to create the final output signal. The available output modes are:

- clock_only: Outputs only the normal clock signal, with no glitches applied.
- glitch_only: Outputs only the glitch pulses, ignoring the clock signal entirely.
- clock_or: The output is high when either the clock or the glitch pulse is high.
- clock_xor: The output is high only when the clock and glitch pulse differ.
- enable_only: The output stays high for a set number of full clock cycles. In this mode, `width` and `width_fine` have no effect, but `offset` and `offset_fine` are still used.

These modes are chosen based on the type of glitching attack:

- For clock glitching, use `clock_or` or `clock_xor`.
- For oltage glitching, use `glitch_only` or `enable_only`.

scope.glitch.trigger_src :

The glitch module supports four trigger modes:

- Continuous: Glitches are triggered constantly. Parameters like `ext_offset`, `repeat`, and `num_glitches` have no effect.
- Manual: Glitches are triggered manually via `manual_trigger()` or `scope.arm()`. Only `repeat` is relevant; others are ignored.
- ext_single: Triggers once per arming when a condition is met. Ignores further triggers until re-armed. `ext_single` provides a controlled, one-time glitch per arming cycle based on an external condition—making it useful for repeatable and precise fault injection experiments.
- ext_continuous: Triggers glitches repeatedly whenever the trigger condition is met, regardless of arm status.

scope.glitch.repeat :

`repeat` is a setting that controls how many glitch pulses are generated per trigger.

- If the output mode is `glitch_only`, `clock_or`, or `clock_xor`, each count in `repeat` represents a separate glitch pulse.
- If the output mode is `enable_only`, the glitch is a single pulse that lasts for `repeat` clock cycles.
- This helps in creating stronger glitches, especially for voltage glitching.
- On **CW-Husky**, if multiple glitches are used (`num_glitches > 1`), `repeat` should be a list with one value per glitch. Each value must be $\leq$ `ext_offset`$[i+1]+1$.
- On **CW-Lite/Nano**, only one glitch is supported, so `repeat` is a single integer.
- The value of `repeat` must be in the range [1, 8192], and it has no effect in `continuous` mode.

`scope.glitch.width :`

The `width` property defines how wide a single glitch pulse is and can be set using either a float or an integer. Its meaning depends on the type of hardware used. For CWHusky, the width is measured in phase shift steps. A value of 0 gives the smallest pulse, while the maximum is at half the total number of phase shift steps. Negative values are allowed and are interpreted as wrapping around the phase shift range, so $-x$ is treated the same as `total_steps` $-\,x$. The values also wrap around when going beyond the maximum.

For other devices like CWLite or CWNano, the width is given as a percentage of one clock cycle. You can set the pulse from about -49.8% to $+49.8\%$ of the cycle. A width of 0% will not work reliably. Negative widths behave like their positive counterparts but are applied to the opposite half of the clock cycle. This setting has no effect if the output mode is set to `enable_only`.

`scope.glitch.offset :`

The `offset` property sets the delay between the rising edge of the clock and the start of the glitch pulse. It can be a float or an integer, and its meaning depends on the hardware used.

For CWHusky, the offset is measured in phase shift steps. An offset of 0 means the glitch pulse starts exactly at the rising edge of the clock. At half of `scope.glitch.phase_shift_steps`, the glitch starts at the falling edge of the clock. We can use negative values, and $-x$ is treated the same as `scope.glitch.phase_shift_steps` $-x$. The value also wraps around, so $+x$ is the same as `scope.glitch.phase_shift_steps` $+x$.

For other devices like CWLite or CWPro, the offset is given as a percentage of one clock period. The glitch can start anywhere from -49.8% to $+49.8\%$ of the clock cycle. This allows us to move the glitch to any point within the cycle.

$scope.glitch.ext\_offset$:

The `ext_offset` property defines how many clock cycles the glitch module should wait after receiving a trigger before generating a glitch pulse. This delay is useful when you want to insert the glitch at a specific point in the target device's operation, such as during the execution of a particular instruction. On CW-Lite and CW-Pro, multiple glitches are not supported, so `ext_offset` is simply an integer representing the delay after the trigger to the single glitch. This setting has no effect if the trigger source is set to `manual` or `continuous`.

The `offset` property, on the other hand, controls the position of the glitch within a single clock cycle. While `ext_offset` determines *when* the glitch should start in terms of clock cycles after the trigger, `offset` adjusts the glitch *within* a chosen clock cycle—allowing very precise control relative to the rising or falling edge of the clock signal. In CW-Husky, `offset` is measured in phase shift steps, and in CW-Lite or CW-Pro, it is expressed as a percentage of one clock period. Together, these two properties provide both coarse and fine control over glitch timing.

$scope.arm()$:

The `arm()` function prepares the scope to start capturing data or performing a glitch when a trigger is received. This step is required before any capture or glitch attempt can begin. If the scope is set to `ext_single` mode, it will remain idle until it is armed and a valid trigger event occurs. Without calling `arm()`, the scope will not respond to trigger signals.

$scope.capture()$:

The `capture(poll_done=False)` function starts the process of capturing a trace. Before using this function, the scope must be armed using `arm()`. Once called, it waits for a trigger event. When the trigger occurs or a timeout is reached, the function stops the capture, disarms the scope, and retrieves the recorded data.

scope.get_last_trace() :

The `get_last_trace(as_int=False)` function returns the most recent trace captured by the scope. By default, it provides the data as a NumPy array of floating-point values scaled between `-0.5` and `0.5`. If the parameter `as_int` is set to `True`, the function returns the trace as raw integer values, which are the direct outputs from the ADC of the ChipWhisperer device. The resolution of these values depends on the hardware: for example, the ChipWhisperer-Lite uses a 10-bit ADC, the Nano uses 8-bit, and the Husky can use either 8-bit or 12-bit ADC data.

3.2.3 Target API

The `target` object in ChipWhisperer is used to manage the device under test (DUT) and perform operations such as loading firmware, executing commands, and capturing traces. It provides a high-level interface for interacting with the target device. ChipWhisperer provides two classes for UART communication:

- Simple Serial Target (default)
- Simple Serial V2 Target

The most straightforward way to create a target object in ChipWhisperer is by using the `cw.target` function. Here is a simple example:

```
import chipwhisperer as cw
scope = cw.scope()
try:
    if SS_VER == "SS_VER_2_1":
        target_type = cw.targets.SimpleSerial2
    else:
        target_type = cw.targets.SimpleSerial
except:
    SS_VER="SS_VER_1_1"
    target_type = cw.targets.SimpleSerial

try:
    target = cw.target(scope, target_type)
```

This code initializes the ChipWhisperer scope and sets up the Simple Serial target.

some useful methods and properties of the `Simple Serial V2 Target` object include:

$target.flush()$:

The `flush()` function clears all data currently stored in the serial buffer. This is useful when you want to discard any previous or unwanted serial communication data before starting a new operation. It helps ensure that the buffer only contains fresh data relevant to the current task.

$target.simpleserial\_write(cmd, data)$:

This function sends a command and associated data to a target device using the SimpleSerial protocol. This function is typically used when communicating with firmware that implements cryptographic functions such as AES encryption.

The `cmd` parameter is a one-character string that specifies the type of command. For example, using `'p'` tells the device to encrypt the given plaintext, while `'k'` sets the encryption key. These special cases internally map to specific command and sub-command values expected by the firmware.

The `data` parameter is a `bytearray` containing the actual data to be sent with the command. If no data is provided, a single byte `[0x00]` is sent by default. The optional `end` argument is reserved but not used in this implementation.

Example: To send a 16-byte plaintext to be encrypted using the AES block cipher, one might use the following:

```
target.simpleserial_write('p', bytearray([0x00]*16))
```

This line sends a block of 16 zero bytes to the device with the command `'p'`, which typically triggers AES encryption of the plaintext using a previously loaded key.

$target.simpleserial\_read\_witherrors(cmd, length)$:

The `simpleserial_read_witherrors()` function is used to read data from a device over a serial connection, especially when doing fault injection or glitch experiments. It tries to receive a full response from the device that includes a command, some data, and a special ending byte. If the response is incomplete, contains errors, or takes too long, the function will try one more time with a longer timeout to get whatever data is available. This is helpful when the target device behaves in unexpected ways due to glitches. The result is returned as a dictionary, which includes whether the response was valid, the decoded data if successful, the full raw output, and any return values if needed.

Example: Suppose we expect a 16-byte result from a previous command, such as an AES ciphertext. We can read it like this:

```
response = simpleserial_read_witherrors(cmd='r', pay_len=16)
if response['valid']:
    print("Received:", response['payload'])
else:
    print("Invalid response. Raw output:", response['full_response'])
```

In this example, the function tries to read 16 bytes from a response starting with the command `'r'`. If it fails due to glitches or timing issues, it still gives the raw data so we can analyze what went wrong.

$target.simpleserial\_wait\_ack(cmd, timeout = 1)$:

This function is used to wait for an acknowledgment (ack) or error message from the target device after sending a command. This is useful to confirm whether the target received and understood the previous instruction. You can set a timeout in milliseconds, which defines how long to wait for the ack. If the timeout is set to 0, the function will wait indefinitely until it receives a response. If no ack is received within the given time, the function returns `None`. Otherwise, it returns a code that indicates the result of the command.

3.3 Process of Clock/Voltage Glitching

Clock and voltage glitching are effective hardware fault injection techniques used to disrupt normal device behavior by introducing brief disturbances in timing or power. With tools like the ChipWhisperer, these attacks can be precisely controlled and analyzed to identify and exploit vulnerabilities in embedded systems.

3.3.1 Target Code Configuration with Simpleserial

The first step in preparing for a clock or voltage glitching attack is to configure the target code. This involves setting up the target device to respond to specific commands and to execute cryptographic operations that can be disrupted by glitches. The ChipWhisperer platform uses a SimpleSerial interface, which allows for easy communication between the host and the target. The target code is typically written in C and includes a main loop that listens for commands from the host. The following example illustrates a basic setup for the target code with simpleserial_v_2, which includes a command to perform cryptographic operation:

```
#include "simpleserial.h"

uint8_t function(uint8_t cmd, uint8_t scmd,
 uint8_t dlen, uint8_t* data) {
    //initializations
    //parsing of 40 bytes of data sent by simpleserial

    trigger_high();
        //perform cryptographic operation
    trigger_low();

    simpleserial_put('r', 5, result);
    return 0;
}
int main(void) {
    platform_init();
    init_uart();
    trigger_setup();
simpleserial_init();
    simpleserial_addcmd('b', 40, function);
    while(1)
        simpleserial_get();


  return 0;
}
```

The `function()` is a command handler invoked when a specific command ('b') is received over the serial interface. It expects 40 bytes of data, representing input for a cryptographic operations. The function performs the operation, triggers a high signal to indicate the start of processing, and then triggers a low signal to indicate completion. Finally, it sends back the 5 bytes result to the host.

3.3.2 Device Configuration

The second step is configuring the ChipWhisperer environment to match the target device. This involves specifying the correct hardware platform, communication protocol, and cryptographic target. These settings ensure that the glitching process can interact properly with the device under test.

For both CWLite and CWHusky devices, the `SCOPETYPE` should be set to `'OPENADC'`, which indicates that the OpenADC module is used for signal acquisition. The `PLATFORM` setting depends on the specific target hardware in use. For the CWLite-ARM board, the platform should be set to `'CWLITEARM'`. If using the CW308 baseboard with an STM32F3 target, the platform should be `'CW308_STM32F3'`. For CWHusky connected to a SAM4S target via the CW308 board, the platform should be `'CW308_SAM4S'`. When working with the CW308 baseboard paired with an XMEGA target, the appropriate platform is `'CW308_XMEGA'`. Additionally, if SimpleSerial version 2 is used for communication, the `SS_VER` must be set to `'SS_VER_2_1'` to ensure correct protocol compatibility.If using any `'CRYPTO_TARGET'`, it should be set. For example, if the crypto target is AES, it can be set to `'TINYAES128C'` as ChipWhisperer provides an implementation of AES-128.

3.3.3 Running the Setup Script

Once the target configuration is defined, the next step is to initialize the ChipWhisperer environment by running a setup script. This script is located in the `jupyter/Setup_scripts` directory of the ChipWhisperer installation. It loads all necessary modules, applies the appropriate configuration settings, and ensures that the scope and target are properly initialized for communication and glitching operations.

The following command is used to execute the generic setup script:

```
%run "{PATH to setup script}/Setup_Generic.ipynb"
```

This script sets up the scope, target, and communication interface automatically. It simplifies the initialization process by applying standard settings required for most experiments, allowing the user to focus on customizing parameters for the specific attack.

3.3.4 Compiling the Target Firmware

After setting up the device and selecting the appropriate target firmware, the next step is to compile the firmware to match the specified platform and cryptographic implementation. This ensures that the target device is running the correct code.

The compilation is typically performed using the `arm-none-eabi-gcc` compiler, which is suitable for ARM Cortex-M based targets. It is important that this compiler is installed and properly configured in the system's environment variables.

The following Bash command is used within a Jupyter notebook cell to compile the firmware using the selected platform, cryptographic algorithm, and SimpleSerial version:

```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET" "$SS_VER"
cd {PATH to the firmwire source directory}
make PLATFORM=$1 CRYPTO_TARGET=$2 SS_VER=$3
```

This command navigates to the firmware source directory and invokes the `make` utility with the relevant parameters. The result is a compiled firmware binary that can be flashed to the target device for use in glitching or side-channel analysis experiments.

3.3.5 Loading the Target Firmware

Once the firmware is compiled, it needs to be loaded onto the target device. This step ensures that the device is running the correct code for the glitching attack. The following command loads the compiled firmware onto the target:

```
    fw_path = "{PATH to the firmware source directory}" +
              "{compiled targate file}.hex"

    cw.program_target(scope, prog, fw_path)
```

3.3.6 Trace Capture Procedure

Once the scope and target are properly configured and the firmware is loaded, the next step involves triggering the cryptographic operation and capturing the power trace. First, the ADC clock source is set using `scope.clock.adc_src`, ensuring that the ADC is synchronized with the internal clock generator.

The `reboot_flush()` function is then called to reset the target device and flush any residual data from the communication buffer. After that, the scope is armed using `scope.arm()` to prepare it for capturing power data.

If the firmware needs any input then it is sent to the target using the SimpleSerial interface with the command `target.simpleserial_write('cmd` same as defined in the firmware', data). This command triggers the cryptographic operation on the target device.

The `scope.capture()` function is then called to record the power trace during the operation. If no trigger is detected, an error message is printed. Once the trace is successfully captured, it is retrieved using `scope.get_last_trace()`. By analysing the captured trace, one can observe the power consumption patterns during the cryptographic operation, which can be useful for identifying location of the attack that we want to perform.

Finally, the output (ciphertext) is read back from the target using `target.simpleserial_read_witherrors('r',size of data same as defined in the firmware)`. The captured waveform is then plotted using `cw.plot(wave)`, which displays the power trace for further analysis or glitching experiments.

3.3.7 Setting Up the Glitch Parameters

The next step is to configure the glitch parameters. This includes setting the glitch type, width, and offset and location of the glitch. The location of the glitch is specified in terms of the number of samples from the start of the trace, and can be set using `scope.glitch.ext_offset`. For voltage glitching, use `scope.glitch.output = "glitch_only"`, and for clock glitching, use `scope.glitch.output = "clock_xor"`. Set the other parameters like `scope.glitch.clk_src`, `scope.glitch.trigger_src` according to the options discussed in the previous section.

In the next step we can run the glitching attack by varying the width, offset and ext_offset parameters and analyzing the captured traces or data we can complete the attack.

3.4 Conclusion

Setting up a fault injection lab with ChipWhisperer combines specialized hardware and software components to enable precise glitching attacks. The step-by-step process from configuring target firmware to capturing traces ensures reliable and repeatable fault injection experiments. This setup provides a powerful foundation for exploring hardware security vulnerabilities effectively.

CHAPTER 4

Real-world Replication of State-of-the-art

The Advanced Encryption Standard (AES) has stood resilient for decades, becoming a cornerstone in modern cryptographic systems. However, side-channel attacks and fault analysis have emerged as powerful methods to compromise even the most robust ciphers, not by breaking the algorithm itself, but by exploiting its implementation. One such sophisticated technique is the Diagonal Fault Attack (DFA), a targeted fault analysis method designed to extract cryptographic keys from AES by injecting faults into specific portions of the cipher's internal state.

In this chapter, we reproduce the diagonal fault attack on AES as presented by Dhiman et al. in their 2009 paper titled "A Diagonal Fault Attack on the Advanced Encryption Standard" [SMC09]. Their work introduced a fault model where a single fault injected into a diagonal of the AES state matrix during the final rounds of encryption enables efficient recovery of the secret key.

To validate their proposed attack in a practical setting, we implemented the diagonal fault injection using two physical fault methods: clock glitching and voltage glitching, both facilitated by the ChipWhisperer Lite (CWLite) platform. Our goal is to observe diagonal fault patterns in the faulty ciphertexts and subsequently perform key recovery using the methodology described in the original paper.

This experimental reproduction helps bridge the gap between theoretical fault models and their real-world applicability, emphasizing the feasibility and efficiency of diagonal fault attacks under controlled glitching conditions.

4.1 Four Diagonals in the AES State Matrix

In the AES algorithm, the internal 128-bit state is arranged as a 4×4 matrix of bytes:

Diagonal 0 $[D_0]$

$\boxed{a_{00}}$	a_{01}	a_{02}	a_{03}
a_{10}	$\boxed{a_{11}}$	a_{12}	a_{13}
a_{20}	a_{21}	$\boxed{a_{22}}$	a_{23}
a_{30}	a_{31}	a_{32}	$\boxed{a_{33}}$

Diagonal 1 $[D_1]$

a_{00}	$\boxed{a_{01}}$	a_{02}	a_{03}
a_{10}	a_{11}	$\boxed{a_{12}}$	a_{13}
a_{20}	a_{21}	a_{22}	$\boxed{a_{23}}$
$\boxed{a_{30}}$	a_{31}	a_{32}	a_{33}

Diagonal 2 $[D_2]$

a_{00}	a_{01}	$\boxed{a_{02}}$	a_{03}
a_{10}	a_{11}	a_{12}	$\boxed{a_{13}}$
$\boxed{a_{20}}$	a_{21}	a_{22}	a_{23}
a_{30}	$\boxed{a_{31}}$	a_{32}	a_{33}

Diagonal 3 $[D_3]$

a_{00}	a_{01}	a_{02}	$\boxed{a_{03}}$
$\boxed{a_{10}}$	a_{11}	a_{12}	a_{13}
a_{20}	$\boxed{a_{21}}$	a_{22}	a_{23}
a_{30}	a_{31}	$\boxed{a_{32}}$	a_{33}

In the above matrices, each element a_{ij} represents the entry in the i^{th} row and j^{th} column of a 4×4 state matrix. This state matrix is commonly used in AES (Advanced Encryption Standard) to represent the internal data at various stages of the encryption or decryption process. The highlighted elements correspond to the entries located along one of the four diagonals of the matrix, indexed as D_0, D_1, D_2, and D_3, respectively.

4.2 Diagonal Fault Attack on AES-128 with Fault Injection at Round 8

In AES-128, the encryption process consists of 10 rounds, with each round transforming a 4×4 byte matrix called the state. Each round includes operations such as SubBytes, ShiftRows, MixColumns, and AddRoundKey. Notably, the AES state is updated in a predictable pattern, and faults injected in earlier rounds propagate through subsequent transformations in a structured way. Step-by-Step Fault Propagation:

Fault Introduction (Start of Round 8): A fault is injected into one or more bytes of one or more diagonals in the AES state matrix. This occurs just after the `AddRoundKey` operation of Round 7 and before the `SubBytes` transformation of Round 8.

SubBytes (Non-linear step): Each byte in the state, including any faulty bytes, undergoes substitution through the AES S-box. This non-linear transformation alters the faulty bytes unpredictably. However, at this stage, the fault remains localized to the originally affected bytes.

ShiftRows (Byte reordering): In this step, each row of the state matrix is cyclically shifted by a specific offset. As a result, the previously localized fault bytes—initially within a single diagonal—are now redistributed into different columns. This marks the beginning of spatial fault propagation.

MixColumns (Diffusion step): This transformation applies a fixed matrix multiplication over a finite field to each column of the state. Consequently, a single faulty byte in any column results in the corruption of all four bytes within that column. At this stage, the fault diffuses significantly across the state.

AddRoundKey : The transformed (and now faulty) state is XORed with the Round 8 key. The fault remains embedded and is now more spread out across the matrix.

Rounds 9 and 10: The corrupted state continues to evolve through the remaining AES rounds. Round 9 further propagates the fault via `SubBytes`, `ShiftRows`, and `MixColumns`. However, Round 10 omits the `MixColumns` step, causing the fault pattern to stabilize. The final faulty ciphertext thus reflects this structured propagation, which can be analyzed to extract internal state information. 4.1 is the Power Trace of the AES-128 running in the integrated target of CWLite-ARM. This clearly shows the AES128 rounds:

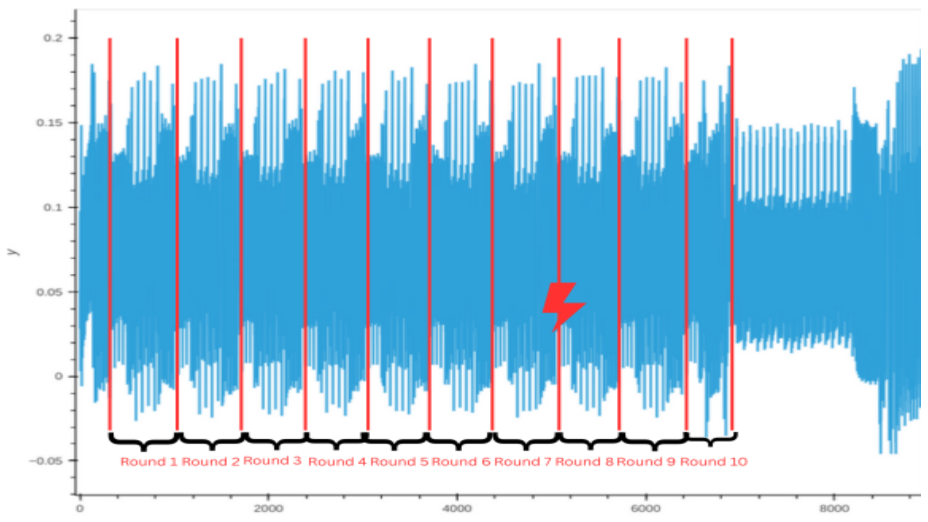


Figure 4.1: AES128 Power Trace of CWLite-ARM

4.3 Fault Injection at the Start of Round 8 using CW-Lite

To analyze the behavior of AES under fault conditions, a fault can be intentionally introduced at the beginning of Round 8 using the ChipWhisperer-Lite (CW-Lite) platform. Two common techniques supported by CW-Lite for fault injection are voltage glitching and clock glitching. These methods allow precise control over when and how a fault is introduced into the target device running AES.

4.3.1 Using Clock Glitching

We have identified a probable location(5080 to 5120 samples) of the starting of 8th round from the power trace of the AES-128 running in the integrated target of CWLite-ARM. Then we have set the parametsers for voltage glitching as follows:

```
scope.glitch.clk_src = "clkgen"
scope.glitch.output = "clock_xor"
scope.glitch.trigger_src = "ext_single"
scope.glitch.repeat = 1
scope.io.hs2 = "glitch"
```

Result

A glitch was introduced at location 5086 samples, which corresponds to the start of Round 8. The parametsers

```
scope.glitch.offset= 10
scope.glitch.width= 3
```

were responsible for this fault injection. This resulted obtaining a faulty ciphertext b'56 9b 6f 66 c9 41 96 f7 9c f1 49 44 29 13 04 e4' where the correct ciphertext without any fault would be b'f5 d3 d5 85 03 b9 69 9d e7 85 89 5a 96 fd ba af'.

Fault Propagation & Analysis

We have one correct ciphertext and one faulty ciphertext after the fault injection. As the key was already known to us we have compared the two ciphertexts to analyze the fault propagation using the AES-128 Decryption method. The following tables show the state of the AES matrix at various stages of the encryption process, both with and without the fault.

After The Fault Injection At Starting of 8th Round :

Without Fault

96	e9	e9	3c
71	87	61	89
6a	91	04	13
e4	c7	90	ff

With Fault

96	e9	e9	3c
71	a3	61	89
6a	91	04	13
e4	c7	90	ff

Difference

00	00	00	00
00	24	00	00
00	00	00	00
00	00	00	00

After The Completion of 8th Round(SubByte,ShiftRow,MixColumn,AddRoundkey(8)) :

Without Fault

0c	b7	3b	ad
77	b8	a0	c3
31	0a	19	d8
43	b0	70	eb

With Fault

2b	b7	3b	ad
4d	b8	a0	c3
2c	0a	19	d8
5e	b0	70	eb

Difference

27	00	00	00
3a	00	00	00
1d	00	00	00
1d	00	00	00

After The Completion of 9th Round(SubByte,ShiftRow,MixColumn,AddRoundkey(9)) :

Without Fault

c2	10	a5	54
df	31	da	c0
67	79	42	5d
9b	74	40	fa

With Fault

dc	52	13	6e
d0	73	1b	ec
68	bf	35	4b
8a	f0	f6	ec

Difference

1e	42	b6	3a
0f	42	c1	2c
0f	c6	77	16
11	84	b6	16

After The Completion of 10th Round(SubByte,ShiftRow,AddRoundkey(10)) :

Without Fault (ct)

f5	03	e7	96
d3	b9	85	fd
d5	69	89	ba
85	9d	5a	af

With Fault (ct_f)

56	c9	9c	29
9b	41	f1	13
6f	96	49	04
66	f7	44	e4

Difference

a3	ca	7b	bf
48	f8	74	ee
ba	ff	c0	be
e3	6a	1e	4b

Here we notice that the fault was injected at 1 bytes of the D_0 diagonal of the state matrix of 8th round. According to the fault model proposed by Dhiman et al.,[SMC09] the fault in the D_0 diagonal of the state matrix at the start of Round 8 propagates through the subsequent rounds, affecting the final ciphertext. The differences in the ciphertexts before and after the fault injection reveal how the fault has altered specific bytes, which can be exploited to recover parts of the secret key. If we consider Byte inter-relations at the end of ninth round corresponding to D_0 diagonal, we can express the following relation:

After 8th Round

f_1			
f_2			
f_3			
f_4			

After 9th Round

$2f_1$	f_4	f_3	$3f_2$
f_1	f_4	$3f_3$	$2f_2$
f_1	$3f_4$	$2f_3$	f_2
$3f_1$	$2f_4$	f_3	f_2

After 10th Round Shift Row

$2f_1$	f_4	f_3	$3f_2$
f_4	$3f_3$	$2f_2$	f_1
$2f_3$	f_2	f_1	$3f_4$
f_2	$3f_1$	$2f_4$	f_3

If we represent the 10^{th} round key as K_{10}, it can be expressed as:

k_{00}	k_{01}	k_{02}	k_{03}
k_{10}	k_{11}	k_{12}	k_{13}
k_{20}	k_{21}	k_{22}	k_{23}
k_{30}	k_{31}	k_{32}	k_{33}

Now we can frame 3 equations based on the differences in the ciphertexts before and after the fault injection to guess the key bytes K_{00},K_{13},K_{22} and K_{31} . The equations are as follows:

$$\begin{aligned}
\text{ISB}(\text{ct}[0] \oplus K_{00}) \oplus \text{ISB}(\text{ct}_f[0] \oplus K_{00}) &= \text{mul2}\,(\text{ISB}(\text{ct}[13] \oplus K_{13}) \oplus \text{ISB}(\text{ct}_f[13] \oplus K_{13})) \\
\text{ISB}(\text{ct}[13] \oplus K_{13}) \oplus \text{ISB}(\text{ct}_f[13] \oplus K_{13}) &= \text{ISB}(\text{ct}[10] \oplus K_{22}) \oplus \text{ISB}(\text{ct}_f[10] \oplus K_{22}) \\
\text{ISB}(\text{ct}[7] \oplus K_{31}) \oplus \text{ISB}(\text{ct}_f[7] \oplus K_{31}) &= \text{mul3}\,(\text{ISB}(\text{ct}[13] \oplus K_{13}) \oplus \text{ISB}(\text{ct}_f[13] \oplus K_{13}))
\end{aligned}$$

Similarly for keybytes K_{01},K_{12},K_{23} and K_{30} we can frame the following equations:

$$\begin{aligned}
\text{ISB}(\text{ct}[11] \oplus K_{32}) \oplus \text{ISB}(\text{ct}_f[11] \oplus K_{32}) &= \text{mul2} \cdot (\text{ISB}(\text{ct}[4] \oplus K_{01}) \oplus \text{ISB}(\text{ct}_f[4] \oplus K_{01})) \\
\text{ISB}(\text{ct}[1] \oplus K_{10}) \oplus \text{ISB}(\text{ct}_f[1] \oplus K_{10}) &= \text{ISB}(\text{ct}[4] \oplus K_{01}) \oplus \text{ISB}(\text{ct}_f[4] \oplus K_{01}) \\
\text{ISB}(\text{ct}[14] \oplus K_{23}) \oplus \text{ISB}(\text{ct}_f[14] \oplus K_{23}) &= \text{mul3} \cdot (\text{ISB}(\text{ct}[4] \oplus K_{1}) \oplus \text{ISB}(\text{ct}_f[4] \oplus K_{1}))
\end{aligned}$$

For keybytes K_{02},K_{11},K_{20} and K_{33} we can frame the following equations:

$$\begin{aligned}
\text{ISB}(\text{ct}[2] \oplus K_{20}) \oplus \text{ISB}(\text{ct}_f[2] \oplus K_{20}) &= \text{mul2} \cdot (\text{ISB}(\text{ct}[8] \oplus K_{02}) \oplus \text{ISB}(\text{ct}_f[8] \oplus K_{02})) \\
\text{ISB}(\text{ct}[15] \oplus K_{33}) \oplus \text{ISB}(\text{ct}_f[15] \oplus K_{33}) &= \text{ISB}(\text{ct}[8] \oplus K_{02}) \oplus \text{ISB}(\text{ct}_f[8] \oplus K_{02}) \\
\text{ISB}(\text{ct}[5] \oplus K_{11}) \oplus \text{ISB}(\text{ct}_f[5] \oplus K_{11}) &= \text{mul3} \cdot (\text{ISB}(\text{ct}[8] \oplus K_{02}) \oplus \text{ISB}(\text{ct}_f[8] \oplus K_{02}))
\end{aligned}$$

Similarly for keybytes K_{03},K_{10},K_{21} and K_{30} we can frame the following equations:

$$\begin{aligned}
\mathsf{ISB}(\mathsf{ct}[9] \oplus K_{12}) \oplus \mathsf{ISB}(\mathsf{ct}_f[9] \oplus K_{12}) &= \mathsf{mul2} \cdot (\mathsf{ISB}(\mathsf{ct}[6] \oplus K_{21}) \oplus \mathsf{ISB}(\mathsf{ct}_f[6] \oplus K_{21})) \\
\mathsf{ISB}(\mathsf{ct}[3] \oplus K_{30}) \oplus \mathsf{ISB}(\mathsf{ct}_f[3] \oplus K_{30}) &= \mathsf{ISB}(\mathsf{ct}[6] \oplus K_{21}) \oplus \mathsf{ISB}(\mathsf{ct}_f[6] \oplus K_{21}) \\
\mathsf{ISB}(\mathsf{ct}[12] \oplus K_{03}) \oplus \mathsf{ISB}(\mathsf{ct}_f[12] \oplus K_{03}) &= \mathsf{mul2} \cdot (\mathsf{ISB}(\mathsf{ct}[6] \oplus K_{21}) \oplus \mathsf{ISB}(\mathsf{ct}_f[6] \oplus K_{21}))
\end{aligned}$$

Here in AES, `mul2` and `mul3` denote multiplication by 2 and 3 in the finite field $\mathrm{GF}(2^8)$, used in the MixColumns step to achieve diffusion through field arithmetic. Solving these 12 equations allows us to recover the key bytes of 10th Round and thus the complete key of AES-128.

4.3.2 Using Voltage Glitching

Firstly we have identified a probable location(5090 to 5120 samples) of the starting of 8th round from the power trace of the AES-128 running in the integrated target of CWLite-ARM. Then we have set the parametsers for voltage glitching as follows:

```
scope.glitch.clk_src = "clkgen"
scope.glitch.output = "glitch_only"
scope.glitch.trigger_src = "ext_single"
scope.io.glitch_lp = True
scope.io.glitch_hp = True
```

Result

A glitch was introduced at location 5100 samples, which corresponds to the start of Round 8. The parametsers

```
scope.glitch.offset= -37.890625
scope.glitch.width= 37.109375
```

were responsible for this fault injection.

Analysis and Fault Propagation

We have one correct ciphertext and one faulty ciphertext after the fault injection. As the key was already known to us we have compared the two ciphertexts to analyze the fault propagation using the AES-128 Decryption method. The following tables show the state of the AES matrix at various stages of the encryption process, both with and without the fault.

After The Fault Injection At Starting of 8^{th} Round :

Without Fault

96	e9	e9	3c
71	87	61	89
6a	91	04	13
e4	c7	90	ff

With Fault

96	e9	e9	52
52	87	61	89
6a	91	04	13
e4	c7	90	ff

Difference

00	00	00	6e
23	00	00	00
00	00	00	00
00	00	00	00

After The Completion of 8^{th} Round(SubByte,ShiftRow,MixColumn,AddRoundkey(8)) :

Without Fault

0c	b7	3b	ad
77	b8	a0	c3
31	0a	19	d8
43	b0	70	eb

With Fault

0c	b7	3b	9e
77	b8	a0	75
31	0a	19	90
43	b0	70	6e

Difference

00	00	00	33
00	00	00	b6
00	00	00	48
00	00	00	85

After The Completion of 9^{th} Round(SubByte,ShiftRow,MixColumn,AddRoundkey(9)) :

Without Fault

c2	10	a5	54
df	31	da	c0
67	79	42	5d
9b	74	40	fa

With Fault

b4	11	6b	73
a9	32	a7	5e
fd	7b	f1	c3
77	75	f3	43

Difference

76	01	ce	27
76	03	7d	9e
9a	02	b3	9e
ec	01	b3	b9

After The Completion of 10^{th} Round(SubByte,ShiftRow,AddRoundkey(10)) :

Without Fault

f5	03	e7	96
d3	b9	85	fd
d5	69	89	ba
85	9d	5a	af

With Fault

5d	4b	9e	39
37	b2	67	b0
58	0b	58	2d
b2	7c	55	ab

Difference

a8	48	79	af
e4	0b	e2	4d
8d	62	d1	97
37	e1	0f	04

Here we notice that the fault was injected at 2 bytes of the D_3 diagonal of the state matrix of 8th round. According to the fault model proposed by Dhiman et al.,[SMC09] the fault in the D_3 diagonal of the state matrix at the start of Round 8 propagates through the subsequent rounds, affecting the final ciphertext.According to The Byte inter-relations corresponding to D_3 diagonal, we can express the following relation:

After 8th Round

			f_1
			f_2
			f_3
			f_4

After 9th Round

f_4	f_3	$3f_2$	$2f_1$
f_4	$3f_3$	$2f_2$	f_1
$3f_4$	$2f_3$	f_2	f_1
$2f_4$	f_3	f_2	$3f_1$

After 10th Round Shift Row

f_4	f_3	$3f_2$	$2f_1$
$3f_3$	$2f_2$	f_1	f_4
f_2	f_1	$3f_4$	$2f_3$
$3f_1$	$2f_4$	f_3	f_2

Again we can make the following equations based on the differences in the ciphertexts with or without the fault injection to guess the key bytes K_{00},K_{13},K_{22} and K_{31} . The equations are as follows:

$$\begin{aligned}
\mathsf{ISB}(\mathsf{ct}[0] \oplus K_{00}) \oplus \mathsf{ISB}(\mathsf{ct}_f[0] \oplus K_{00}) &= \mathsf{ISB}(\mathsf{ct}[13] \oplus K_{13}) \oplus \mathsf{ISB}(\mathsf{ct}_f[13] \oplus K_{13}) \\
\mathsf{ISB}(\mathsf{ct}[10] \oplus K_{22}) \oplus \mathsf{ISB}(\mathsf{ct}_f[10] \oplus K_{22}) &= \mathsf{mul3} \cdot (\mathsf{ISB}(\mathsf{ct}[13] \oplus K_{13}) \oplus \mathsf{ISB}(\mathsf{ct}_f[13] \oplus K_{13})) \\
\mathsf{ISB}(\mathsf{ct}[7] \oplus K_{31}) \oplus \mathsf{ISB}(\mathsf{ct}_f[7] \oplus K_{31}) &= \mathsf{mul2} \cdot (\mathsf{ISB}(\mathsf{ct}[13] \oplus K_{13}) \oplus \mathsf{ISB}(\mathsf{ct}_f[13] \oplus K_{13}))
\end{aligned}$$

Similarly for keybytes K_{01},K_{10},K_{23} and K_{32} we can frame the following equations:

$$\begin{aligned}
\mathsf{ISB}(\mathsf{ct}[11] \oplus K_{32}) \oplus \mathsf{ISB}(\mathsf{ct}_f[11] \oplus K_{32}) &= \mathsf{ISB}(\mathsf{ct}[4] \oplus K_{01}) \oplus \mathsf{ISB}(\mathsf{ct}_f[4] \oplus K_{01}) \\
\mathsf{ISB}(\mathsf{ct}[1] \oplus K_{10}) \oplus \mathsf{ISB}(\mathsf{ct}_f[1] \oplus K_{10}) &= \mathtt{mul3}\,(\mathsf{ISB}(\mathsf{ct}[11] \oplus K_{32}) \oplus \mathsf{ISB}(\mathsf{ct}_f[11] \oplus K_{32})) \\
\mathsf{ISB}(\mathsf{ct}[14] \oplus K_{23}) \oplus \mathsf{ISB}(\mathsf{ct}_f[14] \oplus K_{23}) &= \mathtt{mul2}\,(\mathsf{ISB}(\mathsf{ct}[4] \oplus K_{01}) \oplus \mathsf{ISB}(\mathsf{ct}_f[4] \oplus K_{01}))
\end{aligned}$$

For keybytes K_{02},K_{11},K_{20} and K_{33} we can frame the following equations:

$$\begin{aligned}
\mathsf{ISB}(\mathsf{ct}[8] \oplus K_{02}) \oplus \mathsf{ISB}(\mathsf{ct}_f[8] \oplus K_{02}) &= \mathtt{mul3}\,(\mathsf{ISB}(\mathsf{ct}[2] \oplus K_{20}) \oplus \mathsf{ISB}(\mathsf{ct}_f[2] \oplus K_{20})) \\
\mathsf{ISB}(\mathsf{ct}[15] \oplus K_{33}) \oplus \mathsf{ISB}(\mathsf{ct}_f[15] \oplus K_{33}) &= \mathsf{ISB}(\mathsf{ct}[8] \oplus K_{02}) \oplus \mathsf{ISB}(\mathsf{ct}_f[8] \oplus K_{02}) \\
\mathsf{ISB}(\mathsf{ct}[5] \oplus K_{11}) \oplus \mathsf{ISB}(\mathsf{ct}_f[5] \oplus K_{11}) &= \mathtt{mul2}\,(\mathsf{ISB}(\mathsf{ct}[2] \oplus K_{20}) \oplus \mathsf{ISB}(\mathsf{ct}_f[2] \oplus K_{20}))
\end{aligned}$$

For keybytes K_{03},K_{10},K_{21} and K_{30} we can frame the following equations:

$$\begin{aligned}
\mathsf{ISB}(\mathsf{ct}[9] \oplus K_{12}) \oplus \mathsf{ISB}(\mathsf{ct}_f[9] \oplus K_{12}) &= \mathsf{ISB}(\mathsf{ct}[6] \oplus K_{21}) \oplus \mathsf{ISB}(\mathsf{ct}_f[6] \oplus K_{21}) \\
\mathsf{ISB}(\mathsf{ct}[3] \oplus K_{30}) \oplus \mathsf{ISB}(\mathsf{ct}_f[3] \oplus K_{30}) &= \mathtt{mul3}\,(\mathsf{ISB}(\mathsf{ct}[6] \oplus K_{21}) \oplus \mathsf{ISB}(\mathsf{ct}_f[6] \oplus K_{21})) \\
\mathsf{ISB}(\mathsf{ct}[12] \oplus K_{03}) \oplus \mathsf{ISB}(\mathsf{ct}_f[12] \oplus K_{03}) &= \mathtt{mul2}\,(\mathsf{ISB}(\mathsf{ct}[6] \oplus K_{21}) \oplus \mathsf{ISB}(\mathsf{ct}_f[6] \oplus K_{21}))
\end{aligned}$$

4.4 Conclusion

In this experiment, we successfully demonstrated the use of CWLite for voltage glitching and Clock glitching to inject faults into the AES-128 encryption process. By analyzing the fault propagation through the rounds of AES, we were able to derive equations that allowed us to recover key bytes from the faulty ciphertext as mentioned in [SMC09]. These attacks can also be perfomed with the help of CWHusky with slight modification in the setting by making

```
scope.glitch.clk_src = "pll"
```

and then finding a suitable glitch location observing Power Trace, width and offset. We have observed few glitches in CWHusky as follows: Similar to the

Glitch Location	**Width**	**Offset**	**Effected Diagonal**
6123	50	10	D_3
6134	100	10	D_2,D_3
6134	500	10	D_2,D_3
6129	50	50	D_2

Table 4.1: Table showing location, width, offset, and whether the diagonal is affected

previous section, we can analyze the fault propagation and recover the key bytes by framing the equations from the faulty ciphertext and correct ciphertext. These results demonstrate the effectiveness of CWLite and CWHusky in performing fault injection attacks on AES-128 encryption. The ability to manipulate the encryption process through voltage and clock glitches provides valuable insights into the security vulnerabilities of cryptographic systems.

CHAPTER 5

Flipping Bits to Break BipBip

BipBip is a tweakable block cipher tailored for applications that demand fast decryption, particularly when implemented in Application-Specific Integrated Circuits (ASICs). These are specialized hardware platforms optimized for specific tasks, where minimizing latency—especially during the decryption phase—is often a critical requirement.

BipBip is designed with a relatively unconventional block size of 24 bits, which is smaller than the standard 64 or 128-bit blocks found in more widely used ciphers. It utilizes a 256-bit master key to ensure a high level of cryptographic security and supports a 40-bit tweak, which allows for additional variability in the encryption/decryption process without the need to change the key. This tweakable feature is particularly useful in modes of operation like tweakable block ciphers or authenticated encryption, where different tweaks can offer better resistance against certain attacks.

What sets BipBip apart from many conventional cipher designs is the way its specification is presented. Instead of focusing primarily on the encryption transformation—that is, the process of converting plaintext into ciphertext—BipBip emphasizes the decryption pathway, moving from ciphertext back to plaintext. This design perspective reflects a practical consideration: in many hardware-based systems, especially those that rely on ASICs, decryption is the performance bottleneck, often due to strict real-time constraints. Optimizing decryption can therefore lead to more efficient overall system performance, particularly in security-critical environments such as secure communications, embedded systems, and low-power IoT devices.

By designing with these hardware-centric priorities in mind, BipBip offers a lightweight yet secure option for scenarios where resource constraints and performance requirements must be carefully balanced.

5.1 High Level Decryption Structure of BipBip

Figure 5.1 taken from [BDD+22] is structured around three main components of BipBip's decryption: the datapath, the tweak schedule, and the key schedule.

The key schedule selects bits from the 256-bit master key K to produce the whitening key κ_0 and the tweak-round keys κ_i. These keys are used later in the tweak and datapath stages. For simplicity, the key schedule is not shown in the structural diagram.

The datapath begins by combining the ciphertext C with the whitening key κ_0. It then alternates between applying round functions R or R' and mixing in data-round keys k_i, eventually producing the plaintext P. These k_i keys are not taken directly from the master key but are derived through the tweak schedule.

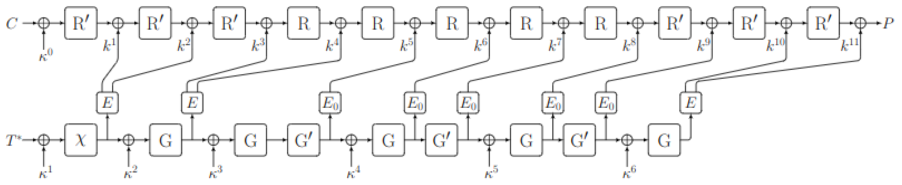


Figure 5.1: High Level Decryption Structure of BipBip

The tweak schedule initializes a 53-bit internal state using the padded 40-bit tweak T. It applies alternating rounds of functions G and G', combined with the tweak-round keys κ_i. At each stage, parts of the state are extracted to form the data-round keys k_i used in the datapath.

5.2 Proposed Attack on BipBip

According to the study by Dilip et al., the BipBip cipher shows a vulnerability to key recovery attacks when a single-bit fault is introduced at the input of the S-box in the 9th round. This fault, after passing through the permutation layer, affects only 2 out of the 6 bits in the corresponding S-box input of the next round. The authors used this limited diffusion property as a filtering technique

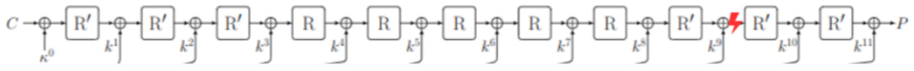


Figure 5.2: Proposed fault injection attack

to narrow down possible key candidates. In their approach, they guessed parts of the round key k_{11} relevant to the first S-box, performed partial decryption, and observed the resulting intermediate values. If the bits expected to be affected by the fault were non-zero, those key guesses were ruled out as invalid. Only the keys that produced zero in those specific bits were considered as potential correct keys.

5.3 Experimental Setup

First task was to add sempleserial to the BipBip cipher implementation so that communication can be established between the scope and target board.Then from the power trace of the BipBip cipher running on CWLite an approximat location for the attack was identified (10820,10890). For conduction a precise

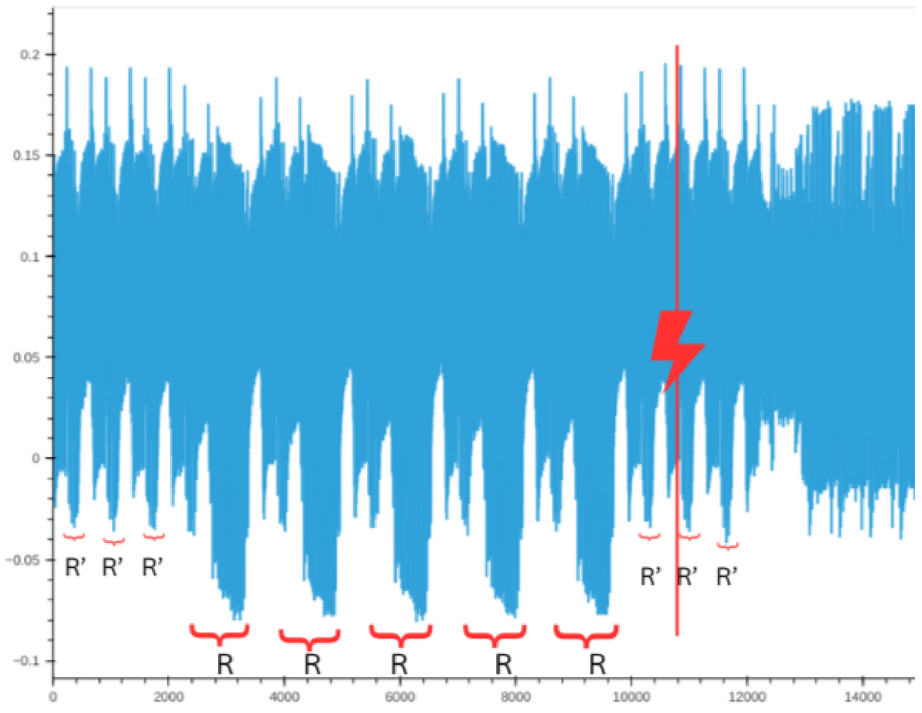


Figure 5.3: Power trace BipBip in CWLite-ARM

Clock glitch attack the following settings were used:

```
scope.glitch.clk_src = "clkgen"
scope.glitch.output = "clock_xor"
scope.glitch.trigger_src = "ext_single"
```

```
    scope.glitch.repeat = 1
    scope.io.hs2 = "glitch"
```

We have also done this glitch attack on the CWHusky board, and for that we used

```
    scope.glitch.clk_src = "pll"
```

and the location of the attack was set at by observing the power trace of the BipBip cipher running on CWHusky target.

5.4 Results & Analysis

Here we present the results of the clock glitch fault injection on the BipBip cipher. The table summarizes the parameters used for the fault injection and the resulting faulty plaintexts compared to the correct plaintexts after round 9.

According to the attack model proposed by Dilip et al., only 4 such fault plaintexts are required to recover the 11th round key k_{11}. The faulty plaintexts obtained from the clock glitch attack on BipBip were used to filter out incorrect key candidates. The results of the attack are summarized in Table 5.1. We have shown that the input of 9th round has only one bit flip in the obtained plaintexts compared to the correct one.

5.5 Conclusion

This work demonstrates how carefully targeted bit-flipping faults can compromise the security of the BipBip cryptographic implementation. By analyzing its decryption structure, an effective fault injection attack was executed. The experimental results confirm the attack's success, highlighting potential vulnerabilities and the importance of robust fault detection mechanisms in such systems.

Table 5.1: Summary of Clock Glitch Fault Injection Parameters and Results

Location	**Width**	**Offset**	**Plaintext**	**Faulty vs Correct state (Input of Round 9)**
10841	1.17	1.17	052aa7	**Faulty:** 011100 100111 111101 101111 **Correct:** 001100 100111 111101 101111
10842	1.95	1.17	6462d1	**Faulty:** 000100 100111 111101 101111 **Correct:** 001100 100111 111101 101111
10855	1.9	1.7	6f2a99	**Faulty:** 001000 100111 111101 101111 **Correct:** 001100 100111 111101 101111
10866	1.17	1.17	3c12bc	**Faulty:** 001101 100111 111101 101111 **Correct:** 001100 100111 111101 101111
10878	1.17	1.17	105bfc	**Faulty:** 001100 100101 111101 101111 **Correct:** 001100 100111 111101 101111
10887	1.9	1.17	1c52e4	**Faulty:** 001100 101111 111101 101111 **Correct:** 001100 100111 111101 101111

CHAPTER 6

Preparing PQC for Fault Analysis: A Kyber Implementation

In recent years, the global cryptographic community is facing a growing concern: the rise of quantum computing is threatning the security foundations of many widely used cryptographic protocols. Algorithms such as RSA, DSA, and ECC, which underpin much of today's secure communications, are susceptible to quantum attacks—most notably Shor's algorithm, which can factor large integers and compute discrete logarithms exponentially faster than classical algorithms. This looming threat has spurred the development of Post-Quantum Cryptography (PQC), which refers to cryptographic schemes believed to be resistant to attacks by quantum computers.

PQC does not depend on the number-theoretic problems that quantum computers can efficiently solve. Instead, it builds on hard mathematical problems such as lattice-based, hash-based, code-based, and multivariate polynomial problems. Among these, lattice-based cryptography has emerged as a strong candidate due to its balance between security, efficiency, and versatility. One of the most promising lattice-based schemes is Kyber, a key encapsulation mechanism (KEM) that is part of the PQC standardization process of NIST. Kyber is based on the Module Learning With Errors (MLWE) problem, which is widely considered secure even in the quantum setting.

The transition to PQC is not only a matter of replacing algorithms but also ensuring their robustness against a wide range of implementation attacks, including fault analysis. While PQC schemes may be theoretically secure, their implementations can still be vulnerable to side-channel and fault injection attacks—forms of active attacks where adversaries induce errors during computation to extract secret information. This report focuses on preparing a Kyber implementation to withstand fault analysis, exploring its vulnerabilities, and proposing countermeasures to secure its deployment in the post-quantum era.

6.1 Overview of Kyber

Kyber is a lattice-based key encapsulation mechanism (KEM) designed to provide secure key exchange in a post-quantum world. It is built on the Module Learning With Errors (MLWE) problem, which is believed to be hard even for quantum computers. Kyber offers several advantages, including efficient performance, small key sizes, and strong security guarantees. Kyber operates by encapsulating a symmetric key within a public key, allowing two parties to securely exchange keys without directly sharing them. The encapsulation process involves generating a random polynomial, encoding it with an error term, and then encrypting it using the recipient's public key. The recipient can then decrypt the encapsulated key using their private key, ensuring that only they can access the shared secret. The diagram 6.1 taken from [RCDB24] illustrates the high-level structure of Kyber's key encapsulation mechanism, showing the encapsulation and decapsulation processes. The security of Kyber relies on the hardness of the underlying MLWE problem, making it a strong candidate for post-quantum cryptography.

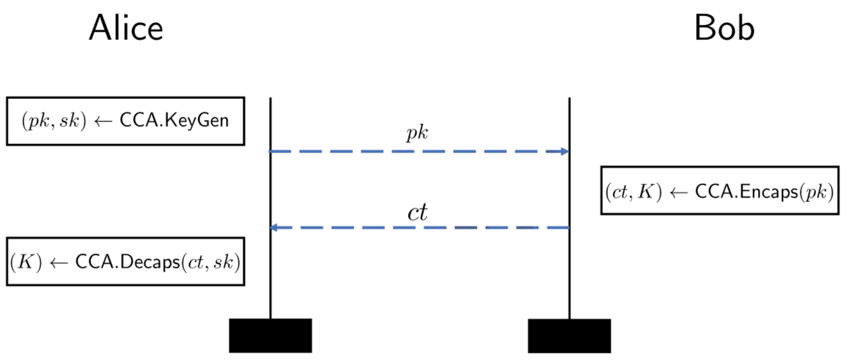

Alice
Bob
$(pk, sk) \leftarrow$ CCA.KeyGen
pk
$(ct, K) \leftarrow$ CCA.Encaps(pk)
ct
$(K) \leftarrow$ CCA.Decaps(ct, sk)

Figure 6.1: Overview of Kyber's Key Encapsulation Mechanism

6.2 Preparing the Kyber Implementation for Fault Analysis

The Kyber implementation utilized in this work originates from the official reference code submitted to the National Institute of Standards and Technology (NIST) as part of their Post-Quantum Cryptography (PQC) Standardization Project Round 3 [Nat20]. The reference implementation is written in portable C and primarily designed for algorithm validation and benchmarking on conventional computing platforms.

To adapt the Kyber implementation for hardware-based fault analysis, the codebase was modified to support the SimpleSerial v2 protocol. SimpleSerial provides a lightweight and deterministic serial communication interface, commonly used with side-channel and fault injection tools like ChipWhisperer. Integration involved encapsulating key Kyber operations, including `crypto_kem_keypair`, `crypto_kem_enc`, and `crypto_kem_dec`, within SimpleSerial command handlers. This modification enables consistent command-response interaction and reliable triggering for precise measurement and injection. As a result, the enhanced implementation is now fully compatible with ChipWhisperer, making it suitable for in-depth physical security testing of Kyber under fault-injection scenarios.

About the Author

Chayan Pathak is currently pursuing his M.Tech. in Computer Science and Engineering at the Indian Institute of Technology (IIT) Bhilai. Originally from Debra in Paschim Medinipur, West Bengal, Chayan brings together a solid academic background and hands-on industry experience as he explores the evolving world of technology.

He completed his B.Tech. in Computer Science and Engineering at B.P. Poddar Institute of Management and Technology. During his undergraduate studies, he gained practical experience in cybersecurity through a six-month internship at PwC India in Kolkata. This opportunity not only strengthened his technical foundations but also sparked a deeper interest in digital security and its real-world applications.

Outside of academics, Chayan is passionate about cricket and enjoys spending time listening to music interests that help him maintain a well-rounded and balanced lifestyle.

Bibliography

[B$^+$23] F. Brito et al. Evaluation of fault injection tools for reliability estimation of microprocessor-based embedded systems. *Microprocessors and Microsystems*, 96:104723, 2023. 6

[BDD$^+$22] Yanis Belkheyar, Joan Daemen, Christoph Dobraunig, Santosh Ghosh, and Shahram Rasoolzadeh. Bipbip: A low-latency tweakable block cipher with small dimensions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(1):326–368, Nov. 2022. 40

[BH22] Jakub Breier and Xiaolu Hou. How practical are fault injection attacks, really? Cryptology ePrint Archive, Paper 2022/301, 2022. 5

[H$^+$23] M. Hermelink et al. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (kyber, dilithium): Survey and new results. *ACM Transactions on Embedded Computing Systems*, 2023. 6

[Nat20] National Institute of Standards and Technology. Post-Quantum Cryptography Standardization: Round 3 Submissions. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions, 2020. Accessed: 2025-06-06. 47

[New16a] NewAE Technology Inc. *CW305 Artix FPGA Target Documentation*, 2016. Accessed: 2025-06-05. 10

[New16b] NewAE Technology Inc. *CW308 UFO Board Documentation*, 2016. Accessed: 2025-06-05. 10

[New25a] NewAE Technology Inc. *ChipWhisperer Documentation: Getting Started*, 2025. Accessed: 2025-06-05. 7, 13

[New25b] NewAE Technology Inc. *ChipWhisperer Documentation: Linux Installation*, 2025. Accessed: 2025-06-06. 14

[New25c] NewAE Technology Inc. *ChipWhisperer-Lite*, 2025. 11

[New25d] NewAE Technology Inc. *ChipWhisperer-Nano Target Board*, 2025. 10

[New25e] NewAE Technology Inc. *CW1173 ChipWhisperer-Lite Capture Documentation*, 2025. 11

[New25f] NewAE Technology Inc. *Scope API — ChipWhisperer Documentation*, 2025. 16

[RCDB24] Prasanna Ravi, Anupam Chattopadhyay, Jan Pieter D'Anvers, and Anubhab Baksi. Side-channel and fault-injection attacks over lattice-based post-quantum schemes (kyber, dilithium): Survey and new results. *ACM Trans. Embed. Comput. Syst.*, 23(2), March 2024. 46

[SMC09] Dhiman Saha, Debdeep Mukhopadhyay, and Dipanwita Roy Chowdhury. A diagonal fault attack on the advanced encryption standard. *IACR Cryptol. ePrint Arch.*, 2009:581, 2009. 29, 34, 36, 38

[SZFT23] Amit Mazumder Shuvo, Tao Zhang, Farimah Farahmandi, and Mark Tehranipoor. A comprehensive survey on non-invasive fault injection attacks. Cryptology ePrint Archive, Paper 2023/1769, 2023. 5

[YSW20] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault attacks on secure embedded software: Threats, design and evaluation. *arXiv preprint arXiv:2003.10513*, 2020. 5