Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Energy Consumption Analysis of Lightweight Cryptographic Algorithms in Server Environment

Author: Sushmita Thakur (VU - 2752951, UvA - 14113430)

1st supervisor:	Dr. Francesco Regazzoni (University of Amsterdam, The Netherlands)
daily supervisor:	Dr. Francesco Regazzoni
	Dr. Subhadeep Banik (University of Lugano, Switzerland)
2nd reader:	Dr. Anuj Pathania (University of Amsterdam, The Netherlands)

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

February 18, 2024

Abstract

Cryptographic algorithms have varying resource requirements depending on their complexity, intended application, and the specific characteristics of the targeted computing environment. As they are computationally heavy, understanding the impact of energy consumption is vital for energy efficiency in computing. In this experiment, we investigate energy consumption across various cryptographic algorithms, specifically lightweight ciphers such as LED, Piccolo, PRESENT, GIFT, but also AES, and assess their energy efficiency in a remote server environment.

The objective is to gain insights into the energy consumption patterns of selected algorithms, also aiming to identify the optimal trade-off between energy consumption and performance. By utilizing a monitoring tool to measure the energy consumed by the algorithms during their execution, we monitor data for each algorithm under different configurations. We systematically collect the energy data across various key sizes and implementation types and varying compiler optimization levels.

Our experiment yields detailed insights into the energy consumption of lightweight ciphers, we observe how energy usage varies with different key sizes and the impact of implementation choices, such as table-based, vperm, and bitslice methods. We also learn that compiler optimization levels play a crucial role, offering opportunities for tailored configurations that balance energy efficiency and performance across varying cryptographic scenarios. The collected energy and performance data are used to assess an optimal choice of implementation type in these lightweight ciphers.

Contents

Chapter 1

Introduction

Within the broader field of cryptography, there is a growing interest in understanding the resource implications of cryptographic algorithms, especially in resource-constrained devices. Lightweight ciphers are specially made to work well in devices that don't have a lot of computing power or memory, like embedded systems and IoT devices. The deployment of cryptographic techniques expands to diverse domains. Similarly, Cryptographic research spans various dimensions, including security, efficiency, and adaptability to different computing environments. Understanding the energy usage of cryptographic algorithms can help in optimizing resources, enabling informed algorithm selection, hence balancing security with efficiency.

In this thesis, we focus on the energy consumption patterns of cryptographic algorithms, particularly in the context of lightweight ciphers, and compiler optimization levels. We compare the consumed energy level with different configurations (multiple combinations of key sizes and implementation types) and we also compare it with 3 different implementations of the widely adopted encryption standard AES, a non-lightweight cipher. A server communicates with multiple devices communicates with multiple devices and has complex infrastructure, our objective is to understand the implications of energy use in a server environment considering these specific factors. In pursuing this, our objective is to address the following research questions:

RQ1: Do different key sizes, implementation types, and compiler optimization levels of the same algorithm, yield varying energy consumption levels?

RQ2: Is there a balance point (a sweet spot) between energy efficiency and performance for lightweight cryptographic algorithms?

1. INTRODUCTION

We initiate by examining the prerequisites for conducting this study, and compiling a list of essential tools, applications, and any other resources necessary to establish the experiment. We seek cryptographic algorithm code implementations that either come with various implementation types or allow for modification to accommodate different implementations and key sizes. We implement each of these algorithms with various implementations on a remote server, gathering energy data through an energy monitoring tool that reads consumption values from the preexisting sensors in the server's processor. Subsequently, we collect the data and develop scripts to analyze it, determining the average consumption for each configuration. Mainly our contributions are -

- We provide energy usage patterns across varying key sizes (64/80/128/192/256 depending upon the algorithm), and implementation types (table based/vperm/bitsliced) for LED, Piccolo, and PRESENT. For AES, we study a non-optimized version of AES implementation, a lightweight one, and a bitsliced implementation.
- We provide the study of the effects of compiler optimization levels (such as levels 1,2,3, loop unrolling etc.) for GIFT cipher on energy usage levels with g++ compiler.
- We provide a comprehensive analysis for finding the trade-off between energy efficiency and performance in cryptographic algorithms.

While various parameters contribute to judging algorithm performance such as speed, throughput, etc., our particular emphasis on energy consumption stems from the importance of efficiency in contemporary computing environments. Through the comparative analysis research, our work contributes to a deeper understanding of the relationships between energy consumption, algorithm design, and compiler optimizations, thereby advancing knowledge in the field of cryptographic research.

Roadmap: The subsequent sections of the thesis are structured as follows: In Chapter 2, we cover the basics of cryptographic algorithms and block ciphers, explaining the types, the implications of different key sizes and implementations, and how we analyze energy. Chapter 3 presents an overview of similar work done by others in the field. Then, Chapter 4 gives details about how we design the experiment, and Chapter 5 explains the implementation details of the experiment. We look at the results in Chapter 6, make comparisons, and discuss the results in Chapter 7. Chapter 8 briefly looks at any issues that could affect our experiment. Finally, we conclude and offer ideas for future research in Chapter 9.

Chapter 2

Background

2.1 Cryptographic Algorithms

Cryptography is the practice and study of techniques for securing communication and data from unauthorized access. Cryptographic algorithms are mathematical procedures or rules designed to secure information by transforming it into an unreadable format, known as ciphertext, using encryption [1]. The primary role of cryptographic algorithms is to ensure the confidentiality, integrity, and authenticity of data during transmission or storage. In encryption, a key is used to convert plaintext (original data) into ciphertext, and decryption employs a complementary key to revert the ciphertext back to its original form. This process safeguards sensitive information from unauthorized access or manipulation, forming the foundation of secure communication and data protection in various domains, including information technology, finance, and communication networks. While the algorithms are efficient for security they also produce significant computational overhead [2] [3].

2.2 Types of Cryptography

Symmetric and asymmetric algorithms are two main types of cryptographic algorithms with distinct approaches to encryption and decryption. Symmetric algorithms use a single key for both encryption and decryption, providing efficiency. Asymmetric algorithms involve a pair of keys—public for encryption and private for decryption—offering key distribution advantages but requiring more computation [4].

Block ciphers are a type of Symmetric cryptographic algorithm that processes data in fixed-size blocks, typically chunks of a fixed number of bits [5]. These algorithms encrypt or decrypt data in blocks, and each block is treated independently during the encryption or

decryption process. The fixed block size distinguishes block ciphers from stream ciphers, which operate on individual bits or bytes.

2.3 Influence of Key Size

The size of the key in cryptographic algorithms directly affects the security level, with longer keys providing higher security. The security of many cryptographic algorithms relies on the complexity and strength of their keys. Increasing the key size makes it exponentially more challenging for an attacker to perform a successful brute-force attack, where they systematically try all possible keys to decrypt encrypted data.

However, it's important to note that the relationship between key size and security is not linear. As key size increases, the computational requirements for encryption and decryption also increase. In [6] the relationship between key size and energy consumption for a symmetric encryption (Si) is described as -

$$Energy\_cost_{(Si)} = Key\_setup(Si) + Energy\_per\_byte(Si) * Data\_size \tag{2.1}$$

Therefore, a balance must be achieved between achieving the desired level of security and the practicality of implementing cryptographic algorithms.

2.4 Anotomy of a Block Cipher

In a block cipher, the same key is used for the encryption and decryption of each block. Key scheduling transforms the original key into a series of round keys, essential for the multiple rounds of encryption. The core of a block cipher is the round function, applying various cryptographic operations like substitution (called s-boxes), permutation, and mixing to enhance security [5]. A block cipher typically consists of these key components, organized in a specific structure. Not all block ciphers include the exact set of components mentioned below, as the design of a block cipher can vary based on the cryptographic algorithm and its specific goals. However, these components represent common elements found in many modern block ciphers as depicted in figure 2.1, especially those following widely accepted design principles.

- **Input Block:** The data to be encrypted or decrypted is divided into fixed-size blocks. The block size is a critical parameter, often 64 or 128 bits.

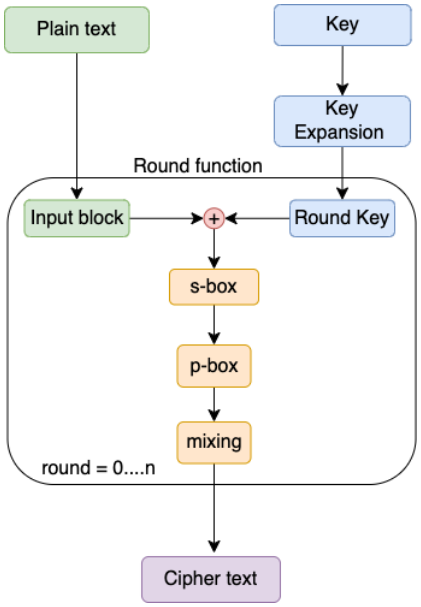

Figure 2.1: Basic structure of a block cipher

- **Key Expansion:** The secret key provided by the user is expanded into a set of round keys. The key expansion algorithm generates a unique key for each encryption round.

- **Round Function:** The round function is the core operation performed in each encryption or decryption round. It typically includes a combination of substitution, permutation, and mixing operations to introduce confusion and diffusion [1].

- **Number of Rounds:** The block cipher operates on the data for a specified number of rounds. The number of rounds contributes to the overall security of the cipher. Common choices include 10, 12, or 16 rounds.

- **Substitution-Permutation Network (SPN):** Many modern block ciphers, such as AES, use the SPN structure. It involves multiple rounds of substitution (S-box), permutation (P-box), and mixing operations.

[1] In the context of block ciphers, confusion refers to the property that makes the relationship between the plaintext and the ciphertext complex and difficult to analyze. It is mainly achieved through substitution. Diffusion refers to the property that a small change in the input (plaintext or key) should result in a significant change in the output (ciphertext). In other words, diffusion ensures that changes in one part of the input block spread throughout the entire output block during encryption, making the relationship between input and output complex and nonlinear.

S-box (Substitution Box): The S-box replaces input bit patterns with predefined output patterns. This nonlinear operation adds confusion to the encryption process. The S-box layer stands out as one of the most critical and resource-intensive operations within the cipher [7].

P-box (Permutation Box): The P-box rearranges the order of bits within a block. This permutation operation helps in spreading the influence of each input bit across the entire block, contributing to diffusion.

- **Mixing Operations:** Operations like XOR or modular addition are used to mix bits within a block. Mixing operations contribute to the overall confusion and diffusion in the encryption process.
- **Output Block:** The final block, after undergoing all the rounds, represents the encrypted or decrypted data.
- **Decryption Process:** In the case of symmetric ciphers, the decryption process is essentially the reverse of encryption. The same round keys are often used in reverse order, and inverse operations are applied.

2.5 Relevant Ciphers

Here, we briefly explain the block ciphers that are part of this study.

2.5.1 AES

AES [8] is a widely adopted symmetric block cipher, selected by the U.S. National Institute of Standards and Technology (NIST) as the standard for encrypting sensitive information. It operates on fixed-size blocks and supports key lengths of 128, 192, or 256 bits. AES has a strong security track record and is used in various applications worldwide. Despite being a secure standard, AES proves unsuitable for devices constrained by hardware limitations [9].

2.5.2 LED

The LED [10] block cipher excels in compact hardware supporting 64 and 128-bit key sizes, with the smallest silicon footprint among similar ciphers. Notably, LED addresses three key objectives. First, it introduces an ultra-light (virtually non-existent) key schedule,

streamlining the hardware requirements. Second, LED demonstrates resilience to related-key attacks, supported by straightforward yet compelling AES-like security proofs. Finally, despite its hardware efficiency, LED strives to uphold a balanced performance profile for software implementation.

2.5.3 PRESENT

PRESENT [11] is a lightweight block cipher designed for applications with limited computational resources. It has a small key size and compact S-box design. It operates as an SP-network (Substitution-permutation layer) with 31 rounds, supporting a block length of 64 bits and key lengths of 80 and 128 bits. Importantly, this is by the design objectives of hardware-oriented stream ciphers within the eSTREAM project.

2.5.4 Piccolo

Piccolo [12] is a 64-bit block cipher that supports both 80 and 128-bit keys, emphasizing efficiency and security for low-resource environments. Through innovative design and implementation techniques, Piccolo achieves a high level of security while maintaining a notably compact hardware implementation. It demonstrates resilience against various analyses, including recent related-key differential attacks and meet-in-the-middle attacks [12]. The cipher's energy consumption efficiency, evaluated by energy per bit, is also remarkable. This positions Piccolo as a competitive ultra-lightweight block cipher along with LED and PRESENT, suitable for extremely constrained environments such as RFID tags and sensor nodes.

2.5.5 GIFT

GIFT [13], characterized by a straightforward design and 128-bit key size, establishes itself as one of the most energy-efficient ciphers available today. Its simplicity, primarily composed of Sboxes and bit-wiring, ensures optimal performance in various scenarios, from area-optimized hardware implementations to swift software execution on high-end platforms. GIFT's robust design is further validated through a comprehensive analysis, providing strong bounds against state-of-the-art cryptanalysis, particularly concerning differential and linear attacks.

2.6 Implementation Methods

Different implementation types for cryptographic algorithms refer to various ways these algorithms can be realized in software or hardware. For example, a software implementation might involve coding the algorithm in a high-level programming language, while a hardware implementation might involve designing a custom circuit or using specialized hardware components. Similarly, in a table-based implementation, tables or arrays are precomputed to enhance processing speed, as often seen in software implementations. On the other hand, bitsliced implementations involve parallelizing bitwise operations for improved efficiency, commonly employed in hardware designs. We briefly discuss the relevant types here -

- **Basic Implementation:** A basic cryptographic implementation involves a direct translation of the algorithm's mathematical description into code. It serves as a fundamental representation for understanding the workings of the algorithm but is typically not optimized for performance or efficiency. This type of implementation is often used for educational purposes, providing a clear and comprehensible representation of the cryptographic processes involved.

- **Lightweight:** Lightweight cryptographic implementations are designed for resource-constrained environments, such as IoT devices or embedded systems. These algorithms prioritize efficiency in terms of speed, code size leading to reduced computational complexity and small energy footprint [14]. While lightweight algorithms may sacrifice some level of security for enhanced performance in constrained environments, they play a critical role in applications where resource limitations are a primary consideration.

- **Vperm:** A Vperm (Vector Permutation) implementation leverages vector operations, capitalizing on modern processors' Single Instruction, Multiple Data (SIMD) capabilities. By processing multiple data elements simultaneously, Vperm implementations significantly enhance the speed and efficiency of cryptographic operations. This type of implementation is especially relevant for optimizing algorithms on platforms with advanced parallel processing capabilities.

- **Table-Based:** Table-based implementations of cryptographic algorithms utilize precomputed tables to store intermediate values. These tables help expedite computations, leading to improved performance. Commonly employed in symmetric ciphers,

this approach enhances the efficiency of cryptographic operations by reducing the need for on-the-fly calculations, making it a valuable optimization technique.

- **Bitsliced:** Bitsliced implementations operate on multiple bits of data simultaneously, using bitwise operations to achieve high parallelism. This approach is particularly effective in symmetric-key cryptographic algorithms. Bitsliced implementations are known for their resistance to certain side-channel attacks, making them a valuable choice in security-sensitive scenarios where efficient parallel processing is crucial.

2.7 Energy Analysis Process on a Server

Several components in a server contribute to energy consumption. The central processing unit (CPU) is a major player, as it executes instructions and performs computations. Memory modules (RAM) also consume power, especially during data access and retrieval. Storage devices, such as hard disk drives (HDDs) or solid-state drives (SSDs), contribute to energy usage when reading or writing data. Power supplies, responsible for converting electrical input into usable power for the server components, are crucial in the energy equation. Networking components, including network interface cards (NICs) and switches, consume power during data transmission. Additionally, cooling systems, like fans and air conditioning, play a role in maintaining optimal temperature levels, impacting overall energy consumption. On top of this, the applications being executed contribute to the energy usage on a server. While the power drawn by hardware components can be static, the power used by software components is dynamic.

2.7.1 Measurement, Tools & Terminologies

Measuring energy consumption involves quantifying energy consumption in watts and joules. Watts represents the rate of energy transfer or consumption per unit of time, indicating the server's instantaneous power usage. Joules, on the other hand, measures the total energy expended over a specific duration [15].

$$EnergyConsumption(Joules) = Power(Watts) * Time(Seconds) \tag{2.2}$$

Figure 2.2 shows the basic components of an energy measurement setup.In the context of energy measurement on servers, the "system under test" refers to the specific server or infrastructure being evaluated for power consumption. The "power measuring framework" encompasses the tools and methodologies employed to measure and monitor energy usage

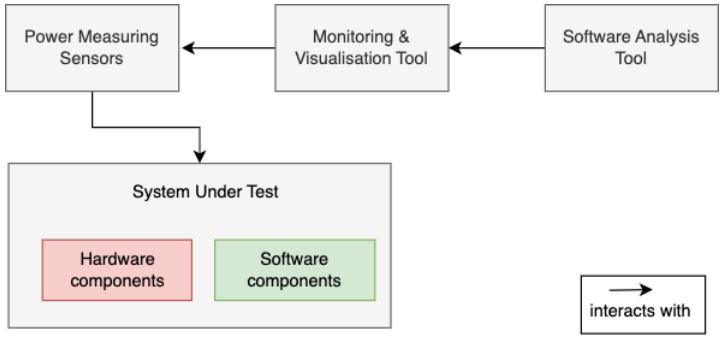

Figure 2.2: Basic components in an energy measurement setup

in this system. This framework typically involves the deployment of power meters, intelligent power strips, or server management software. The "sampling rate" represents how frequently measurements are taken, indicating the rate at which data points on power consumption are recorded over time. A higher sampling rate provides more granular insights into energy usage patterns but may result in increased computational overhead. Balancing the sampling rate is crucial to capture meaningful data without causing excessive system impact.

The process involves the installation and deployment of measurement tools, such as power meters and server management software, to continuously monitor and quantify real-time power consumption. The collected data undergoes detailed analysis to recognize patterns and trends.

2.7.2 Energy Monitoring for Cryptographic algorithms

The energy consumption of block ciphers is significantly influenced by their structural components. The complexity of the Substitution Box (S-box), permutation operations, mixing operations, and key expansion processes all contribute to computational demands, impacting energy efficiency. The number of rounds in a block cipher, a critical factor for security, directly affects energy consumption, with more rounds typically requiring higher computational resources. Hardware implementations, such as ASICs (Application-Specific Integrated Circuit) or FPGAs (Field-Programmable Gate Arrays), also play a role, as optimizing the design for specific architectures can enhance energy efficiency. Efficient inverse operations for decryption in symmetric ciphers further contribute to overall energy optimization. A general research objective is to explore algorithmic adjustments and optimizations to strike a balance between security and energy efficiency. This involves

recognizing the interconnected role of each structural element in determining the energy profile of block ciphers.

The energy analysis process for cryptographic algorithms encompasses a detailed examination of power consumption patterns during their execution. This involves employing energy monitoring tools to measure and record the amount of power consumed by the algorithm over time. Grasping these patterns enables the recognition of energy-efficient implementation strategies.

Chapter 3

Related Work

3.1 On Energy Analysis of Symmetric and Asymmetric Algorithms

In Potlapally *et al.* [6] energy consumption in battery-powered systems is evaluated by measuring power usage during data transfer using the SSL protocol and within the timeframe of cryptographic operations. This analysis provides insights into variations in energy consumption patterns across asymmetric algorithms, hash algorithms, and the impact of key size on both asymmetric and symmetric algorithms. The study suggests that by adjusting key size and the number of rounds, cryptographic algorithms (CAs) can be designed to operate more energy efficiently. The experimental setup for secure client-server communication involves a client connected to a LAN through a wireless access point, while the server, a PC, is wired to the LAN. The research aims to conclude optimizing energy usage in cryptographic operations, contributing to the development of energy-efficient security protocols. While this study offers foundational knowledge, it is important to acknowledge that it predates (2006) recent research advancements in the field. Our experiment is focused on symmetric block ciphers and their varying implementation while this paper involves energy analysis for different symmetric and asymmetric algorithms with one specific implementation type for each algorithm.

Wu *et al.* [16] evaluated three widely used encryption algorithms — AES, Blowfish, and GOST by comparing their performance across different data block sizes. Key expansion time and encryption/decryption speed are measured, and simulation results are presented to showcase the effectiveness of each algorithm. The selected algorithms, namely Blowfish,

AES with a 256-bit key length, and GOST with a 256-bit key length, undergo a comprehensive comparison. Several conclusions are drawn from the analysis: Blowfish exhibits superior performance regardless of plaintext size, key expansion is a potential bottleneck for Blowfish, GOST and Blowfish have similar key expansion times, and decryption is the most time-consuming operation for AES. Although the research yields impressive results, its scope is constrained by the limited number of algorithms evaluated. Also, it specifically examines the performance of algorithms executed on a smartphone, providing a unique perspective tailored to the mobile computing environment hence distinguishing from our study on a server environment.

3.2 On Energy Analysis of Lightweight Algorithms

Aslan *et al.* [9] focus on analyzing lightweight cryptographic algorithms, including PRESENT, CLEFIA, Piccolo, PRINCE, and LBLOCK. PRESENT and CLEFIA are standardized lightweight block cipher algorithms, that meet ISO/IEC 29192-2:2012 requirements. Piccolo, PRINCE, and LBLOCK, while not standardized, remain relevant in IoT applications, being suitable for hardware implementation. Our experiment differs from this context as we, along with the lightweight ciphers, also have AES as a part of our comparative analysis to give the energy perspective for a non-lightweight encryption technique while also making compiler-level energy monitoring.

In the work by Banik *et al.* [7], while the primary objective is to introduce a novel lightweight block cipher named Midori, the authors additionally conduct a comprehensive energy usage comparison with various other lightweight ciphers. This provides valuable insights into the efficiency and performance of Midori in relation to its counterparts. For optimized energy usage in Midori, they propose optimal cell-permutation layers designed to enhance diffusion speed and increase the number of active S-boxes in each round with minimal implementation overheads. They evaluate the energy consumption of Midori along with some other ciphers - AES, NOEKEON, SIMON, PRESENT, and PRINCE. Our works, however, mainly differ from them by the experimental environment and motivation choices of the algorithm. We perform the execution and measurement on a remote server to make comparisons with varying implementation methods.

3.3 On Energy and Performance Trade-off

Datsios *et al.* [17] look into the performance of three cryptographic algorithms – DES, AES, and RSA. Through thorough testing, the researchers measure power and energy consumption across different setups. The results yield valuable insights into the power and energy trade-offs associated with diverse architectural styles. Their analysis centers around three key parameters crucial to power consumption in modern embedded general-purpose processors. These parameters include the parallelism of the core, specifically the issue width and size of the execution window, as well as voltage and frequency switching within the core. Additionally, they examine the impact of the size of the last-level cache (LLC). Notably, the findings highlight that, with specific parameter values, cryptographic operations can be efficiently performed, achieving a balance between performance and power consumption. In our experiment, we refrain from getting into the underlying hardware architecture. Instead, we establish a baseline for the power drawn by the hardware in an ideal state for a specific configuration. The subsequent analysis focuses on isolating the impact of different cryptographic algorithms on the system, aiming to provide insights into their individual performances without considering variations in hardware settings.

Zhou *et al.* [18] introduces a novel FPGA utilization approach, whereas Haleem *et al.* [19] presents a framework, for emphasizing security performance maximization with optimized energy levels. These studies deviate from our research in terms of their respective areas of emphasis. The former concentrates on hardware optimization, particularly FPGA usage, while the latter provides a mathematical model to assess security-throughput tradeoffs, validated exclusively against the Rijndael cipher (AES).

Paper	Tested Ciphers
[6]	AES, RSA, DSA, Hash functions
[16]	AES, Blowfish, GOST
[9]	PRESENT, CLEFIA, Piccolo, PRINCE, LBLOCK
[7]	AES, NOEKEON, SIMON, PRESENT, PRINCE
[17]	AES, DES, RSA
[18]	AES, RSA, DES, SHA
[19]	AES

Table 3.1: Algorithms tested in the presented related work

Chapter 4

Design

We divide this study into three main parts — experimental setup, implementation & data collection, and finally - analysis. This chapter details the environment and conditions under which our experiments were conducted. It includes information about the hardware and software used, and any specific configurations, settings, or parameters.

4.1 Experimental Setup

To facilitate this study, we require specific components as integral parts of our energy analysis process. These components include - a server with sufficient computational power to execute the algorithms, a power sensor for measuring power consumption, a monitoring tool capable of accessing and storing data from the sensors, an optional but preferred visualization tool to make the monitoring and analysis convenient and finally a test suite that includes the application whose energy footprint we want to analyze. The subsequent section elaborates on each component and the used tool in detail. Figure 4.1 offers a comprehensive overview of our setup, illustrating the interconnected components involved in the measurement process.

4.1.1 Server

A server is utilized for several strategic reasons for this study. Primarily, servers offer centralized processing capabilities, efficiently handling the computational demands of the experiment while also providing an isolated clean environment. We use an Ubuntu OS with intel x86_64 architecture and model Intel(R) Xeon(R) (i7) CPU E5-2660 v2 @ 2.20GHz with a total of 20 physical cores. There are two sockets - *socket0* and *socket1* where each has 10 physical cores.

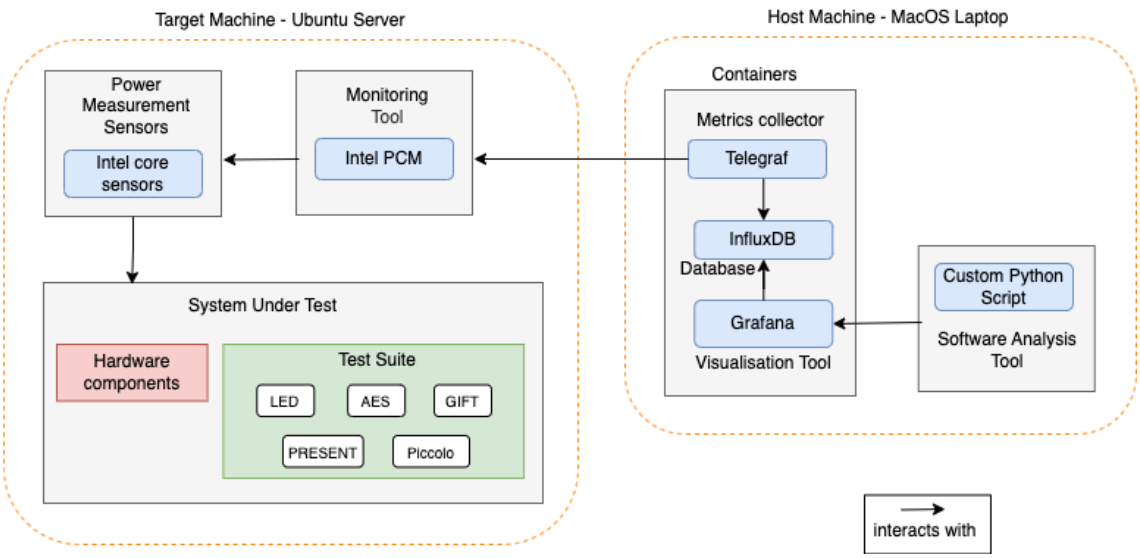

Figure 4.1: Experimental setup

4.1.2 Power Measurement & and Monitoring tool - Intel PCM

The advantageous aspect of this study lies in the intrinsic capabilities of Intel processors, particularly the provision for monitoring performance events within the processors themselves. We use the Intel PCM [20] [1] tool that uses the built-in processor sensors to get the CPU's energy utilization values. The PCM tool utilizes real-time data sourced from the performance monitoring units (PMU) integrated into Intel processors. Notably, the Xeon E5-2660 v2 is built upon the Sandy Bridge EP architecture. Within the Sandy Bridge architecture, a feature known as the Running Average Power Limit (RAPL) was introduced, and seamlessly integrated into the Power Monitoring Unit.

The RAPL [22] interface stores power consumption limits across different domains on a processor within a specified time frame. The power measurement logic in RAPL uses activity counters and predefined weights to capture accumulated energy, storing this information in 32-bit Machine State Registers (MSRs) [23] [24]. The Package (PKG) domain of RAPL measures the energy consumption of the entire processor socket, encompassing cores, integrated graphics, and uncore components such as last-level caches and the memory controller as depicted in Figure 4.2. The energy readings for the Package domain (PKG) are stored in the MSR_PKG_ENERGY_STATUS register. RAPL updates the counters every 1ms. Energy is quantified in units of Joules for PP0/PP1 (powerplane 0

[1] Intel PCM Repository - https://github.com/intel/pcm

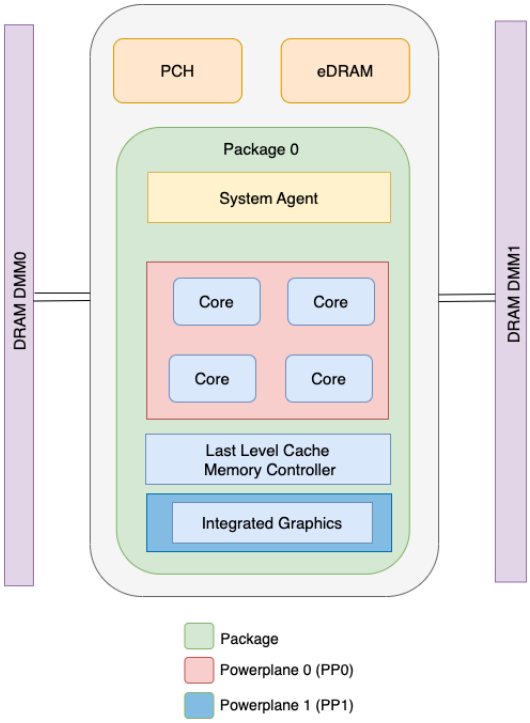

Figure 4.2: Power domains supported by RAPL [21]

and 1, Figure 4.2) RAPL Domains, as specified in the Intel processor architecture manual [25] for the Xeon E5-2600 processor family in section 15.10.1. While RAPL has the drawback of temporal resolution being limited to 1 millisecond, making it incapable of resolving events shorter than 1ms [23], this granularity is acceptable within the requirements of our experiment. Despite the fact that RAPL do not measure the actual power supplied and instead uses performance counters to calculate it, it has a reasonable accuracy level [21]. Khan *et al.* [21] underlines the high correlation between RAPL readings and plug power, promising accuracy with minimal performance overhead. Since no external power meter is required to be installed, the RAPL feature proves to be a convenient, and reliable feature.

On top of this, the Intel PCM tool comes with command-line utilities that can be used to connect with graphical interfaces, such as the Grafana dashboard for visualization and InfluxDB for storing the collected data. Additionally, the Intel PCM tool provides several other matrices along with energy consumption such as Instruction per cycle, L2-L3 cache misses, etc. that provide important information about performance.

4.1.3 Visualization & Data Storage tool - Grafana & InfluxDB

Grafana [26] [1] is an open-source platform designed for monitoring and observability, providing powerful visualization. To utilize Grafana, we first need to set it up and configure a data source. This involves specifying where Grafana should retrieve data for visualization. Grafana supports various data sources, such as InfluxDB, Graphite, Prometheus, and more. Once a data source is connected, you can create dashboards within Grafana and select the desired visualizations, such as graphs or tables. Grafana fetches this data in real-time at specified intervals, in our case, every 2 seconds. The PCM collector exposes the energy data over HTTP in JSON format that can be collected by calling the API via Telegraf and stored in an InfluxDB database. Telegraf is a server-based agent designed to collect and transmit metrics and events originating from data sources etc. The setup involves running both the database and Grafana as Docker containers on the host machine and accessing the Intel PCM tool on the target machine (server) via an SSH connection. This integration allows Grafana to dynamically query the database, extract relevant data, and update visualizations in real-time on the dashboard. Figure 4.3 shows an example of energy consumption visualization on the Grafana dashboard.

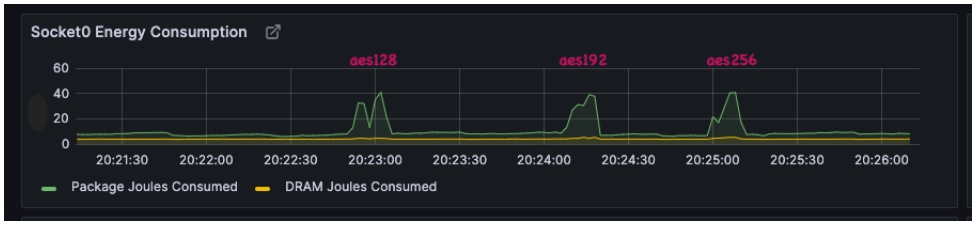

Figure 4.3: Example of Grafana dashboard for a test process execution

4.2 Test Suite Design

Our test suite is a comprehensive collection of cryptographic algorithms designed to assess the energy implications of their implementation styles.

4.2.1 Criteria for Algorithm Selection

The parameters for selecting algorithms in this study include:

[1] Intel PCM Grafana Integration - https://github.com/intel/pcm/tree/master/scripts/grafana

- **Uniform High-level Language Implementation:** All chosen algorithms must be implemented in the same programming language, ensuring consistency and facilitating comparative analysis.
- **Multiple Implementation Types:** Each algorithm should have more than one available implementation, promoting diversity and allowing for a comprehensive evaluation of different approaches.
- **Testability for Different Key Sizes:** Preferably, each implementation type should be testable with various key sizes, enhancing the versatility of the study and accommodating different cryptographic requirements.
- **Scalability for Customization:** In cases where an algorithm lacks availability for multiple key sizes, it should exhibit scalability, enabling straightforward customization to accommodate diverse key sizes.
- **Minimal Dependency on Additional Software:** Selected algorithms should be executable without the need for installing additional software applications, aside from the compiler, streamlining the experimental setup and ensuring practical implementation.

4.2.2 Final Test Suite

In this experiment, all code implementations in the test suite are chosen to be in the C language, except GIFT, which is implemented in C++. GIFT serves a distinct purpose in the experiment and is not subjected to direct comparison with algorithms implemented in C.

We choose the first three algorithms - LED, Piccolo, and PRESENT from the Cryptolib [27] code implementation provided by the authors of SAC 2013 [28]. LED, PRESENT, and Piccolo adopt a 64-bit architecture characterized by the iteration of a round function, incorporating a 4-bit Sbox and a linear diffusion layer. The key schedule in these ciphers is intentionally straightforward. The rationale behind selecting Cryptolib lies in its lightweight implementation and the availability of several variations in the implementation of the three aforementioned algorithms – including table-based, vperm, and bit-sliced designs tailored for different key lengths and parallelism levels. They offer the first bit-sliced implementation of LED and claim this to be the best-performing one for this cipher [28]. Cryptolib also provides cycles per byte data for each execution which is critical to this study for evaluating performance. To ensure comprehensive comparisons, we employ

three distinct implementations of AES. The first is a non-optimized version, serving as a benchmark for AES energy consumption. The second represents a lightweight attempt at AES, while the third is a bit-sliced implementation. Additionally, to assess the influence of compiler configurations, GIFT, a lightweight cipher with two variations (GIFT64 and GIFT128, denoting block size), is included, maintaining a fixed key size of 128 bits. This approach ensures a diverse set of implementations for a nuanced evaluation of energy consumption across cryptographic algorithms. Table 4.1 provides an overview of the final set of algorithms selected for this experiment.

Algorithm	**Implimentation types**	**Key sizes**	**Repo.**	**Paper**
LED	table, v-perm, bitsliced	64, 128	[27]	SAC 2013 [28]
PRESENT	table, v-perm, bitsliced	80, 128	[27]	SAC 2013 [28]
Piccolo	table, v-perm, bitsliced	80, 128	[27]	SAC 2013 [28]
AES - basic	Non-optimized	128, 192, 256	[29]	
AES - tiny	light-weight	128, 192, 256	[30]	
AES - bitsliced	bitsliced	128	[31]	
GIFT	light-weight	64-128, 128-128	[32]	CHES 2017 [13]

Table 4.1: Overview of the Test Suite

4.3 Compiler selection and Configurations

For compiling our collection of algorithms we use GCC and G++ compilers. These two compilers are part of GCC (GNU Compiler Collection) which is a widely used open-source compiler suite that includes compilers for various programming languages, with the most common being GCC for C and G++ for C++. It is renowned for its extensive command-line options hence providing us with the desired control over the research. As the default compiler in many systems, it requires no additional installations and benefits from strong community support. The use of GCC and G++ in research helps with the standardization of results and ensures comparability and reproducibility. The following 8 levels of compiler optimization are employed in this study, table 6.5 lists the configurations with their flags.

- **No Optimization:** Compiles code without applying any optimization techniques, prioritizing faster compilation over execution speed or code size.
- **Level 1 Optimization:** Basic optimizations focused on minimizing code size and improving execution speed at the cost of more extensive compile time.
- **Level 2 Optimization:** Enhanced optimizations beyond Level 1, aiming to strike a balance between code size and execution speed.
- **Level 3 Optimization:** Aggressive optimizations, prioritizing maximum execution speed over code size and requiring longer compilation times.
- **March=native:** Compiler generates code specifically tailored for the host machine's architecture, leveraging all available features and instructions.
- **Function Inlining:** Incorporates the body of a called function directly into its caller, potentially improving performance by reducing function call overhead.
- **Loop Vectorization:** Transforms loops to use SIMD (Single Instruction, Multiple Data) instructions, enhancing parallel execution and performance.
- **Loop Unrolling:** Expands loop iterations, potentially improving performance by reducing loop control overhead and allowing better compiler optimization opportunities.

ID	Type	Command with corresponding flags
#**0**	Basic	g++ <fileToCompile.cpp> -o <outputExecutable>
#**1**	Level-1 optimization	g++ -O1 <fileToCompile.cpp> -o <outputExecutable>
#**2**	Level-2 optimization	g++ -O2 <fileToCompile.cpp> -o <outputExecutable>
#**3**	Level-3 optimization	g++ -O3 <fileToCompile.cpp> -o <outputExecutable>
#**4**	March Native	g++ -march=native <fileToCompile.cpp> -o <outputExecutable>
#**5**	Function Inlining	g++ -finline-functions <fileToCompile.cpp> -o <outputExecutable>
#**6**	Loop Vectorization	g++ -ftree-vectorize <fileToCompile.cpp> -o <outputExecutable>
#**7**	Loop Unrolling	g++ -funroll-loops <fileToCompile.cpp> -o <outputExecutable>

Table 4.2: Compiler Configurations

Chapter 5

Implementation

Once the energy monitoring setup is established on the server, we establish the baseline power consumption. To achieve this, we ensure no processes are running other than PCM tool and record the energy consumption during this idle state for five minutes. The energy consumption range during this period is 6.19 - 10.3 joules with an average of 8.08 joules. Subsequently, we prepare the test suite for the final executions, the code files are uploaded through an SFTP and then all network connections are closed. Next, we initiate the energy monitoring process and collect relevant data.

5.1 Modifications in the algorithms

In the initial phase of our experiment, we prepare all algorithms for execution, ensuring sufficient data by running 50000 iterations of each configuration for all the algorithms.

For CryptoLib algorithms (LED, PRESENT, Piccolo) compiled with gcc, we introduce the following alterations in the code files of the Cryptolib folder -

- A custom bash script file *run_iterations.sh* is added to execute the resulting binary file from executing *make* file, for the intended number of iterations. We control all the variations of an algorithm from this file and pass each specific configuration argument with the executable file name argument.

```
1
2   EXECUTABLE="./check_all_ciphers Piccolo128 bitslice16 -samples=32"
3    ITERATIONS=50000
4
5    # parameters for all the variations
6    # LED64 table
7    # LED64 vperm
```

```
    # LED64 bitslice32
    # LED128 table
    # LED128 vperm
    # LED128 bitslice32

    # PRESENT80 table
    # PRESENT80 vperm
    # PRESENT80 bitslice16
    # PRESENT128 table
    # PRESENT128 vperm
    # PRESENT128 bitslice16

    # Piccolo80 table
    # Piccolo80 vperm
    # Piccolo80 bitslice16
    # Piccolo128 table
    # Piccolo128 vperm
    # Piccolo128 bitslice16

    cd bin
    for ((i = 1; i <= ITERATIONS; i++)); do
        echo "Running iteration $i"
        $EXECUTABLE
    done
    # save the performace data into relevant file
    mv perf_data.txt piccolo128bitslice_perf.csv
```

Listing 5.1: Example of run_iterations.sh from CryptoLib

- For getting the performance data we add a small code to write the sum of total cycles/byte taken for encryption and key scheduling, into a file into the *helper.h* function.

```
FILE* fptr = fopen("perf_data.txt", "a");
fprintf(fptr, "%f\n", (key_schedule_average + encrypt_average));

```

Listing 5.2: Storing performance data from helper.h

For the AES algorithms (AES-basic, AES-tiny, and AES-bitsliced), given their makefile-based implementation, we implement the following enhancements:

- For AES-basic and AES-tiny, we add the iterations in main file (*main.c* and *aes.c* respectively) the main code file is replicated with different key values (*aes128.c, aes192.c, aes256.c*) so that a dedicated makefile could be created for each key value

(e.g., MAKE128 for a make file created with a key value of 128 bits). These files are then compiled to create corresponding executable files. For AES-tiny the executables are in *.elf* format (eg. aes128.elf etc.).

- For AES-bitslied since we have 128-bit implementation only, we use the default make file to compile the code and then add the bash script file *run_ iterations.sh* to add the intended number of iterations.
- The decryption component is excluded, recognizing that not all algorithms offer both encryption and decryption functionalities simultaneously.

For GIFT, the following preparation is done -

- Since we are testing it for compiler configurations, to find a fair number of iterations after which the energy readings stay the same and do not increase with the number of iterations. We first try to find a stability point for increasing the number of iterations 100,200,500 ...and so on for GIFT64.
- Once the stability point is established i.e. 10000 iterations, it is added into the main files (*GIFT64-128_ cipher.cpp* and *GIFT128-128_ cipher.cpp*) that executes encryption.
- Different binary files with different key and compiler configurations are made, provided as flags in the g++ compiler command.
- Each command represents a different configuration and outputs an executable that represents that configuration. For eg. gift64_3 represents that it is a GIFT implementation with 64 bits block size, 128 bits key size and that it is the 3rd compiler (#3) configuration type which is defined in table 6.5.

The compilation commands for the GIFT algorithm, organized by the configuration types outlined in 6.5, are as follows-

GIFT64

1. g++ GIFT64-128_cipher.cpp -o gift64_0
2. g++ -O1 GIFT64-128_cipher.cpp -o gift64_1
3. g++ -O2 GIFT64-128_cipher.cpp -o gift64_2

4. g++ -O3 GIFT64-128_cipher.cpp -o gift64_3
5. g++ -march=native GIFT64-128_cipher.cpp -o gift64_4
6. g++ -finline-functions GIFT64-128_cipher.cpp -o gift64_5
7. g++ -ftree-vectorize GIFT64-128_cipher.cpp -o gift64_6
8. g++ -funroll-loops GIFT64-128_cipher.cpp -o gift64_7

GIFT128

1. g++ GIFT128-128_cipher.cpp -o gift128_0
2. g++ -O1 GIFT128-128_cipher.cpp -o gift128_1
3. g++ -O2 GIFT128-128_cipher.cpp -o gift128_2
4. g++ -O3 GIFT128-128_cipher.cpp -o gift128_3
5. g++ -march=native GIFT128-128_cipher.cpp -o gift128_4
6. g++ -finline-functions GIFT128-128_cipher.cpp -o gift128_5
7. g++ -ftree-vectorize GIFT128-128_cipher.cpp -o gift128_6
8. g++ -funroll-loops GIFT128-128_cipher.cpp -o gift128_7

5.2 Experiment Execution

The experiment is initiated through the following steps:

- The server is accessed via SSH from the host machine, and the Intel PCM is launched in detached mode.

```
1    sudo pcm-sensor-server -d
2
```

- Docker containers (grafana, telegraf, influxdb) are initiated on the host machine with the use of *start.sh* file.
- A secure shell (SSH) connection is established from the host machine to the target machine, utilizing the IP address and port number 9738 exposed by the Intel PCM tool.

```
    sudo sh start.sh http://145.100.131.36:9738

```

- Upon a successful connection, the Grafana dashboard becomes accessible at http://localhost:3000/ on the host machine.

Subsequently, the binary files of the algorithms in the test suite are sequentially executed, and the data collection is performed from the Grafana dashboard. This step involves running each algorithm individually to observe and record its impact on power consumption.

The data can be conveniently downloaded from the Grafana dashboard in either CSV or JSON format. The query executed to retrieve the readings from InfluxDB is as follows.

```
SELECT mean("Sockets_0_Uncore_Uncore Counters_Package Joules Consumed")
    FROM "http" WHERE time >= now() - 5m and time <= now() GROUP BY time(2s
    ) fill(null)
```

Listing 5.3: Query for collecting energy data

The query retrieves the average energy consumption over 2-second intervals in package joules for the CPU socket. Grafana groups the data at a sampling rate of every two seconds, representing the mean energy consumption for each 2-second interval.

As each algorithm implementation is executed on the server, the energy data is downloaded individually from Grafana in CSV format. Subsequently, custom Python code is employed to process and analyze these files, yielding the final results discussed in the following section.

Chapter 6

Results

This section comprises all the energy consumption readings we collected from our experiment starting with energy consumption followed by performance data from CryptoLib algorithms. The execution takes place on the server and one by one the corresponding energy data for each is collected from the Grafana board.

6.1 Energy Consumption

The tables 6.1 to 6.5 present energy consumption data for all the test suite ciphers across different key sizes and implementation methods for LED, PRESENT, Piccolo, AES, and GIFT. Notably, the blue-highlighted values denote the minimum energy consumption, while the red-highlighted values indicate the maximum within a specific configuration. The numerical entries in each cell represent energy consumption measured in joules, with values paired by a '/' symbol representing average and maximum values. From the collected energy consumption values for these ciphers with key sizes 64, 80, and 128 across different implementation methods, several conclusions are drawn below. In the context of 'Key Size Impact,' we examine how energy consumption varies when employing the same implementation method with different key sizes. Meanwhile, in 'Implementation Method Comparison,' we assess how energy consumption differs for a particular key size across various implementation methods.

6.1.1 LED

Refer Table 6.1

- Key Size Impact:
  - For the table implementation, the energy consumption increases from 20.2 joules

(key size 64) to 22.8 joules (key size 128). For the vperm implementation, the energy consumption increases from 23.6 joules (key size 64) to 23.9 joules (key size 128). For the bitsliced implementation, the energy consumption increases from 22.1 joules (key size 64) to 23.1 joules (key size 128). This suggests that, for all implementations, a larger key size results in higher energy consumption.

- Implementation Method Comparison:
  - For key size 64, the table implementation has the lowest energy consumption (20.2 joules), followed by bitsliced (22.1 joules) and vperm (23.6 joules).
  - For key size 128, the bitsliced implementation has the lowest energy consumption (23.1 joules), followed by table (22.8 joules) and vperm (23.9 joules).

For LED cipher, the table-based implementations look to be the most energy-efficient for both key sizes. The vperm implementation tends to have higher energy consumption compared to the other methods.

LED (joules)					
Key size = 64			**Key size = 128**		
table	**vperm**	**bitsliced**	**table**	**vperm**	**bitsliced**
20.2/23.7	23.6/24.3	22.1/24.2	22.8/24.3	23.9/24.4	23.1/24.2

Table 6.1: Results of LED cipher's energy consumption for varying Key-sizes and implementation methods

6.1.2 PRESENT

Refer Table 6.2

- Key Size Impact:
  - For the table implementation, the energy consumption decreases from 23.2 joules (key size 80) to 20.1 joules (key size 128). This suggests that, for the table implementation, a larger key size results in lower energy consumption.
  - For the vperm implementation, the energy consumption increases from 22.2 joules (key size 80) to 23.7 joules (key size 128). This indicates that, for the vperm implementation, a larger key size leads to higher energy consumption.
  - For the bitsliced implementation, the energy consumption increases slightly from

21.0 joules (key size 80) to 23.2 joules (key size 128). This implies that, for the bitsliced implementation, there is a modest increase in energy consumption with a larger key size.

- Implementation Method Comparison:
  - For key size 80, the bitsliced implementation has the lowest energy consumption (21.0 joules), followed by vperm (22.2 joules) and table (23.2 joules).
  - For key size 128, the table implementation has the lowest energy consumption (20.1 joules), followed by bitsliced (23.2 joules) and vperm (23.7 joules).

When comparing implementation methods, bitsliced exhibits lower energy consumption for key size 80, while the table implementation is more energy-efficient for key size 128. The vperm implementation tends to have higher energy consumption compared to the other methods.

PRESENT (joules)					
Key size = 80			**Key size = 128**		
table	**vperm**	**bitsliced**	**table**	**vperm**	**bitsliced**
23.2/24.8	22.2/24.7	21/24.6	20.1/23.3	23.7/24.4	23.2/24

Table 6.2: Results of PRESENT cipher's energy consumption for varying Key-sizes and implementation methods

6.1.3 Piccolo

Refer Table 6.3

- Key Size Impact:
  - For the table implementation, the energy consumption increases from 20.7 joules (key size 80) to 23.8 joules (key size 128). For the vperm implementation, the energy consumption decreases from 24.0 joules (key size 80) to 21.6 joules (key size 128). For the bitsliced implementation, the energy consumption increases from 21.4 joules (key size 80) to 22.9 joules (key size 128). Hence, a larger key size results in higher energy consumption except for vperm.

- Implementation Method Comparison:
  - For key size 80, the table implementation has the lowest energy consumption (20.7 joules), followed by bitsliced (21.4 joules) and vperm (24.0 joules).
  - For key size 128, the vperm implementation has the lowest energy consumption (21.6 joules), followed by bitsliced (22.9 joules) and table (23.8 joules)

For Piccolo cipher, the table-based implementations are the most energy-efficient for 80-bit key sizes. The vperm implementation tends to have good energy consumption compared to the other implementation types for key size 128. Bitsliced stays in between the other two methods for both key sizes.

Piccolo (joules)					
Key size = 80			**Key size = 128**		
table	**vperm**	**bitsliced**	**table**	**vperm**	**bitsliced**
20.7/24.2	24/25.4	21.4/24.3	23.8/24.6	21.6/24.5	22.9/24.4

Table 6.3: Results of Piccolo cipher's energy consumption for varying Key-sizes and implementation methods

6.1.4 AES

Refer Table 6.4

- Key Size Impact:
  - Among the basic implementation type, the energy consumption increases with the key size, peaking at AES-basic-192.
  - Among the tiny implementation type, there is a slight increase in energy consumption with the key size, with AES-tiny-256 having the highest value.

- Implementation Method Comparison:
  - For the key size of 128, the tiny implementation type consumes the least energy (18.1 joules), followed by bitslice (21.3 joules), and basic (27 joules).
  - For the key size of 192, tiny again consumes the least (18.3 joules), followed by basic which consumes the most (29.8 joules).
  - For the key size of 256, tiny consumes the least (19.3 joules), followed by basic which again consumes the most (27.7 joules).

The conclusions drawn from these values suggest that the AES-tiny implementation type generally exhibits lower energy consumption compared to the basic and bitslice types. The basic implementation type shows the highest energy consumption for key size 192, while the tiny implementation has the highest energy consumption at 256 across key sizes.

AES (joules)						
basic			**tiny**			**bitsliced**
128	**192**	**256**	**128**	**192**	**256**	**128**
27/41	29.8/39	27.7/40.7	18.1	18.3	19.3	21.3/23.1

Table 6.4: Results of AES cipher's energy consumption for varying Key-sizes and implementation methods

6.1.5 GIFT

From the collected energy consumption data for the GIFT64 and GIFT128 algorithms in Table 6.5 across different compiler configurations and optimizations, the following observations could be made -

ID	**Type**	**GIFT64 (joules)**		**GIFT128 (joules)**	
		Avg	**Max**	**Avg**	**Max**
#0	Basic	17.6	22.5	21.2	31.5
#1	Level-1 optimization	16.7	23.4	16.4	37.8
#2	Level-2 optimization	14.6	22.7	22.3	29.7
#3	Level-3 optimization	16.2	21.8	23.6	29.6
#4	March Native	17.6	25.5	26	34.2
#5	Function Inlining	16.5	22	28.5	39.7
#6	Loop Vectorization	16.9	29.8	18.2	30.8
#7	Loop Unrolling	16.4	25.6	26.9	39.1

Table 6.5: Results of GIFT cipher's energy consumption for varying compiler configurations

- Compiler optimizations significantly influence energy consumption. Notably, level-2 optimization tends to result in the lowest energy consumption for GIFT64, while function inlining and loop vectorization show mixed impacts across configurations.

- Generally, GIFT128 exhibits higher energy consumption than GIFT64 across various compiler configurations and optimization levels. This is expected as larger block sizes often require more computational resources.

- Level-2 optimization consistently achieves low energy consumption for GIFT64, but its impact varies for GIFT128, where level-1 optimization sometimes performs better. This suggests that the optimal optimization level depends on the block size and specific characteristics of the algorithm.

- The March Native configuration tends to increase energy consumption, particularly for GIFT128. This could be attributed to architecture-specific optimizations that may not align with the algorithm's characteristics.

- Function inlining and loop vectorization show varying impacts on energy consumption. While function inlining tends to reduce energy consumption for GIFT64, it increases it for GIFT128. Loop vectorization also has mixed effects on energy consumption. Loop unrolling tends to have a positive impact on energy consumption, particularly for GIFT128 where it reduces both average and maximum values.

6.2 Performance Results

In the context of cryptographic algorithms, including block ciphers, such as AES (Advanced Encryption Standard) or other symmetric encryption algorithms, the performance is often evaluated based on how quickly the algorithm can process data. "Cycles per byte" is a metric used to measure the performance of a cipher algorithm, particularly in terms of computational efficiency. It represents the number of clock cycles required by the processor to process one byte of data through the encryption or decryption algorithm. Hence, a lower cycle per byte value indicates better performance. It is a useful metric because it provides a standardized measure that is independent of specific hardware characteristics. Since a larger key size enhances the complexity of a cryptographic algorithm, contributing to increased security, achieving a larger key size while maintaining a low cycle per byte value is considered optimal for performance. The performance data for each algorithm of

CryptoLib for multiple variations is analyzed for average cycles/byte values are presented in Table 6.6, 6.7, and 6.8.

LED - perf (cycles/byte)					
Key size = 64			Key size = 128		
table	vperm	bitsliced	table	vperm	bitsliced
128.9	73.8	38.42	187.5	112.8	50.54

Table 6.6: LED cipher's performance (cycles/byte)

PRESENT - perf (cycles/byte)					
Key size = 80			Key size = 128		
table	vperm	bitsliced	table	vperm	bitsliced
222.8	130.2	40.3	225.04	133.8	45.54

Table 6.7: PRESENT cipher's performance (cycles/byte)

Piccolo - perf (cycles/byte)					
Key size = 80			Key size = 128		
table	vperm	bitsliced	table	vperm	bitsliced
156.2	71.94	34.3	201.1	89.1	36.5

Table 6.8: Piccolo cipher's performance (cycles/byte)

A consistent pattern emerges in performance values upon examination of various implementations (table, vperm, bitslice) for LED, PRESENT, and Piccolo across key sizes (64/80/128). Specifically, the bitslice implementation consistently exhibits the lowest cycles per byte, while the table-based implementation demonstrates the highest. The vperm implementation falls between these two extremes. These results also align with the performance results obtained for use case - 4 in SAC 2013 Paper [28] for an XEON X5650 @ 2.67GHz. The consistently lower cycles per byte for bitslice implementations suggest more efficient processing of data compared to table-based implementations. This efficiency

may be advantageous in scenarios where computational resources or processing speed are critical factors. Bitslicing processes data by representing each bit of a word independently, allowing for parallel bitwise operations. This inherent parallelism aligns well with modern processors that excel at parallel computation. Table-based implementations often exhibit lower efficiency compared to bitslice counterparts due to the inherent reliance on table lookups and more complex operations. In a table-based approach, the algorithm typically involves using precomputed tables to substitute complex operations like S-box substitutions or other non-linear transformations. These table lookups introduce memory access overhead and potential cache misses, especially when the tables are large or irregularly accessed. The observed pattern may also reflect a design trade-off in favor of efficiency (bitslice) or simplicity (table). Vperm implementations, consistently falling between bitslice and table values, may be seen as a balanced compromise between efficiency and simplicity. Depending on the specific application requirements, vperm implementations might provide a reasonable trade-off between performance and ease of implementation.

Chapter 7

Discussion

7.1 Table-based Vs Vperm Vs Bitslice

The analysis of the energy consumption data reveals a clear distinction between lightweight and non-lightweight implementations of cryptographic algorithms. It is evident that lightweight implementations consistently consume significantly less energy than their basic, unoptimized counterparts. This can be seen clearly in Figure 7.1 Table 6.4 and Table 7.1 that a non-optimized version of AES (AES128, AES192, and AES256 in blue bars) consumes a comparatively high amount of energy. Among the lightweight ciphers, the lowest average energy usage, approximately 20 joules, is observed for table-based implementations of LED, Piccolo, and PRESENT. On the other hand, vperm implementations exhibit the highest average energy consumption among lightweight ciphers, reaching up to 24 joules.

Algorithm	**No. of Rounds**	**Table(basic for AES)** (joules)	**Vperm(tiny for AES)** (joules)	**Bitslice** (joules)
LED64	32	20.2	23.6	22.1
LED128	48	22.8	23.9	23.1
Present80	31	23.2	22.2	21
Present128	31	20.1	23.7	23.2
Piccolo80	25	20.7	24	21.4
Piccolo128	31	23.8	21.6	22.9
AES128	10	27	18.1	21.3
AES192	12	29.8	18.3	-
AES256	14	27.7	19.3	-

Table 7.1: Energy Consumption Data in joules

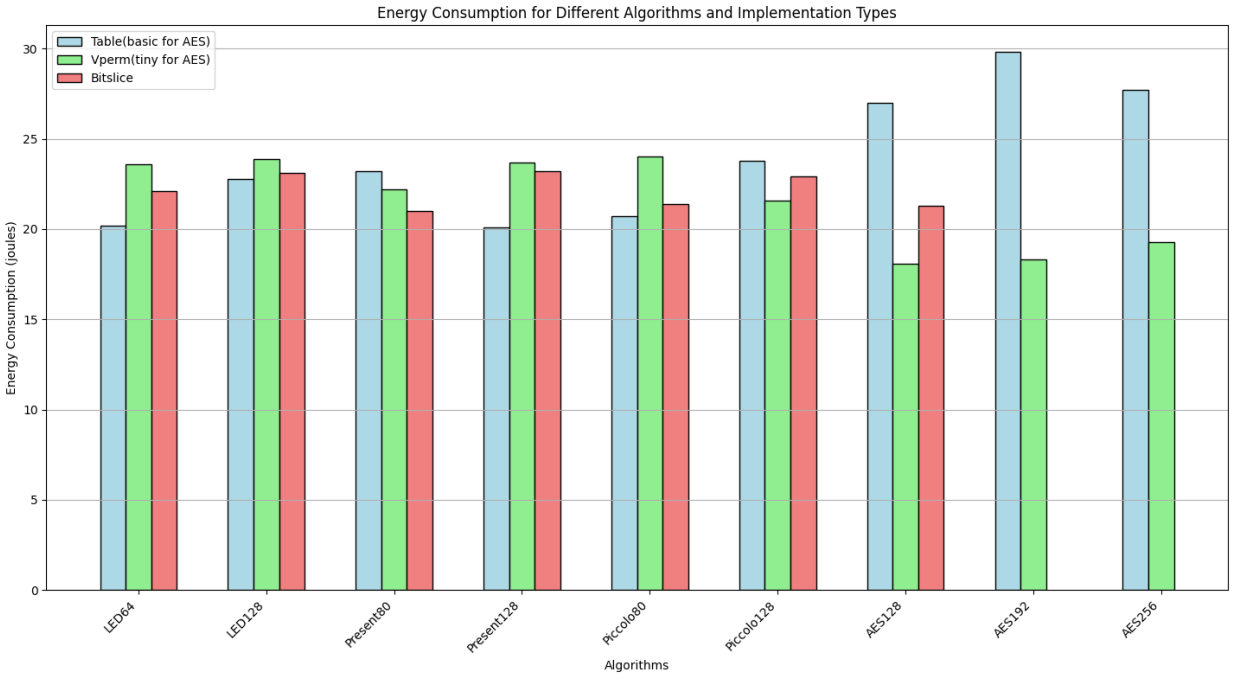

Figure 7.1: Effects of implementation types on LED, PRESENT, Piccolo, and AES

However, it is highlighted by Potlapally et al. [6], that the use of table lookups and similar techniques, while enhancing performance, often results in increased memory accesses. This increased memory usage can have implications for energy efficiency, even though the energy consumption on memory is not explicitly observed in the provided data. The study suggests that aggressive utilization of table lookups, loop unrolling, and similar measures might inadvertently lead to decreased energy efficiency due to the associated rise in memory accesses. Therefore, the seemingly lower energy requirements of table-based methods may be accompanied by considerations related to memory usage. For these particular table-based implementations, Benadjila *et al.* [28] also confirms that table lookups can be performed to achieve the round transformation. With this, larger tables result in a reduction of operations during the round function, but they also bring about increased latencies in table lookups. In essence, the trade-off involves a decrease in computational workload at the expense of higher lookup latencies. PRESENT128 and LED64 show the best average energy levels for table-based (20.2 joules), Piccolo128 for vperm (21.6 joules), and PRESENT80 (21 joules) for bitslice.

In contrast to lightweight ciphers, the non-lightweight algorithms, such as the basic

AES implementation, demonstrate a notably higher energy consumption, peaking at 29.8 joules. Interestingly, the AES-tiny 128-bit implementation stands out as the most energy-efficient option, achieving the lowest observed energy consumption of 18 joules, showcasing the efficiency gains achieved through lightweight design and optimization strategies. The AES-tiny code in our test suite functions in CBC mode, offering a good level of security, particularly in comparison to simpler modes such as Electronic Codebook (ECB) mode. The observed energy efficiency, coupled with the utilization of CBC mode, suggests a well-considered implementation and design choice. The bitslice AES implementation stays on a relatively low level of energy consumption, only slightly higher than the AES-tiny implementation, and marginally lower than the AES-basic implementation. This observation reinforces the trend that bitslice implementations exhibit notable efficiency in energy consumption across different cryptographic algorithms.

7.2 Smaller vs Bigger key-sizes or Block sizes

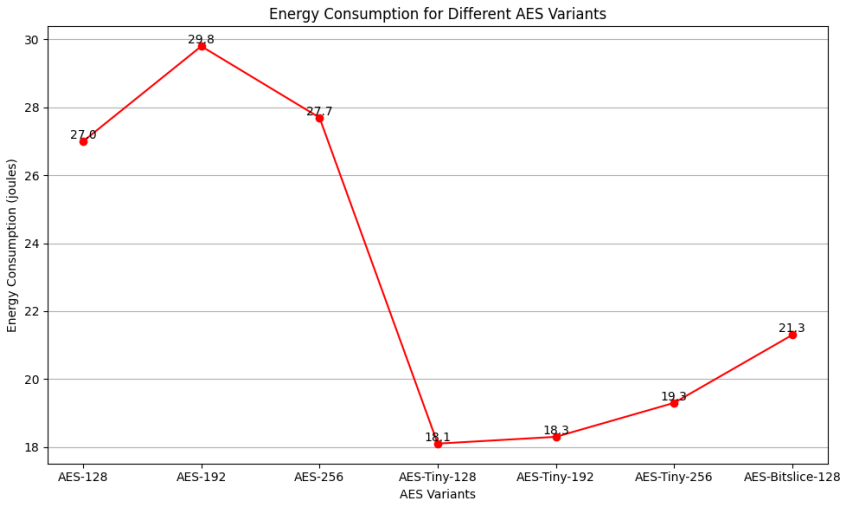

Figure 7.2: AES energy consumption pattern

Typically, larger key sizes introduce heightened computational complexity to cryptographic operations, potentially leading to increased energy consumption—a correlation supported by our experimental data. This distinction becomes particularly evident in the case of PRESENT Figure 7.1 Table 7.1, where the number of rounds remains consistent for both key sizes (80 and 128). Despite this uniformity in the number of rounds, the energy consumption for PRESENT128 is notably higher than that for PRESENT80 for vperm and

bitslice methods. The energy data for AES is visualized in Figure 7.2 which shows a clear impact of key sizes on consumption levels. For instance, within the AES algorithm, the AES-tiny-128 implementation underscores the efficiency of smaller key sizes, consuming an average of 18.0 joules compared to AES-basic-128's 27.0 joules. In the LED algorithm, the LED128-table implementation consumes more energy (22.8 joules) than the LED64-table (20.2 joules). This pattern is also present for varying block sizes in the GIFT algorithm's energy usage, for eg. in the "Function Inlining" configuration in Figure 7.3 Table 6.5, GIFT128 consistently demands more energy than GIFT64, exemplified by averages of 28.5 joules and 16.5 joules, respectively. These instances underscore the general trend wherein larger key sizes and block sizes tend to correlate with higher energy consumption.

7.3 Impact of Compiler Optimization

As evidenced by the GIFT algorithm data in figure 7.3, different compiler optimizations, such as level-1, level-2, and level-3 optimizations, native marching, function inlining, loop vectorization, and loop unrolling, exhibit varying impacts on energy consumption. Notably, the GIFT64 algorithm demonstrates lower energy consumption with level-2 optimization, suggesting that certain compiler configurations might be more effective for specific algorithmic structures. Function inlining tends to increase energy usage, likely due to the expanded code size, while loop vectorization and loop unrolling showcase nuanced effects, with GIFT64 benefiting from loop unrolling but GIFT128 experiencing an increase in energy consumption.

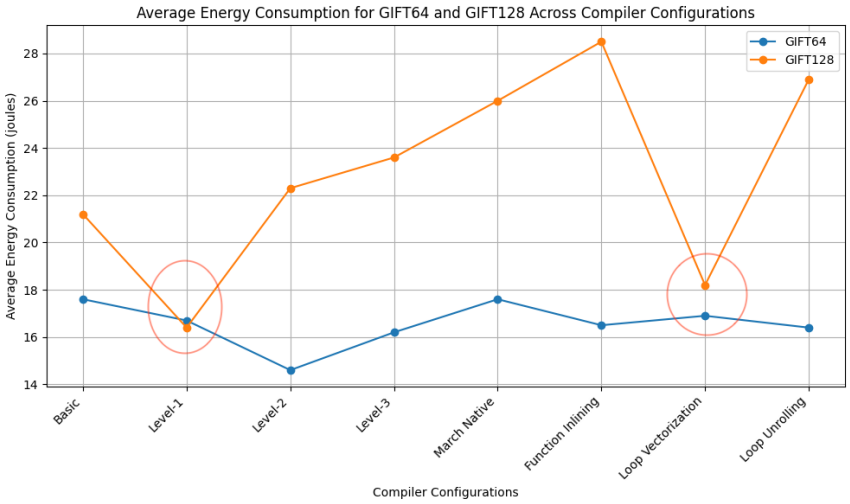

Figure 7.3: Enery consumption for varying compiler optimizations on GIFT

A noteworthy observation emerges from Figure 7.3 indicated by the red circles, where level-2 optimization demonstrates nearly identical energy consumption for both GIFT64 and GIFT128. Similarly, the energy consumption during loop vectorization closely aligns for GIFT64 and GIFT128. These outcomes highlight the relationship between compiler configurations and energy efficiency. This emphasizes that the algorithm's compilation strategies can be tailored for better energy efficiency depending on algorithmic characteristics and key sizes.

7.4 Answering the Research Questions

7.4.1 RQ1

Q: Do different key sizes, implementation types, and compiler optimization levels of the same algorithm, yield varying energy consumption levels?

Certainly. This analysis of cryptographic algorithms, reveals that different key sizes, implementation types, and compiler optimization levels do indeed yield varying energy consumption levels. For instance, key sizes exhibit distinct impacts on energy efficiency, with larger keys generally leading to increased energy consumption due to heightened computational complexity. Moreover, the choice of implementation type, such as table-based, vperm, or bitsliced, influences energy usage, with table-based more often demonstrating lower energy consumption compared to vperm. Bitslice implementation generally shows consumption levels in between table-based and vperm for algorithms LED, PRESENT, and Piccolo and also in varying implementations of AES. Additionally, compiler optimization levels, exemplified by level-2 optimization in GIFT, showcased unexpected similarities in energy consumption between different key sizes. This collective evidence affirms that key sizes, implementation types, and compiler optimizations contribute to varied energy efficiency outcomes.

7.4.2 RQ2

Q: Is there a balance point (a sweet spot) between energy efficiency and performance for lightweight cryptographic algorithms?

The balance point is often characterized by an optimal compromise between minimizing energy consumption and achieving efficient performance. Figure 7.4 shows a summary view of all the performances of LED, PRESENT, and Piccolo in comparison with each other. While there is no universal "sweet spot" applicable to all scenarios, certain algorithms,

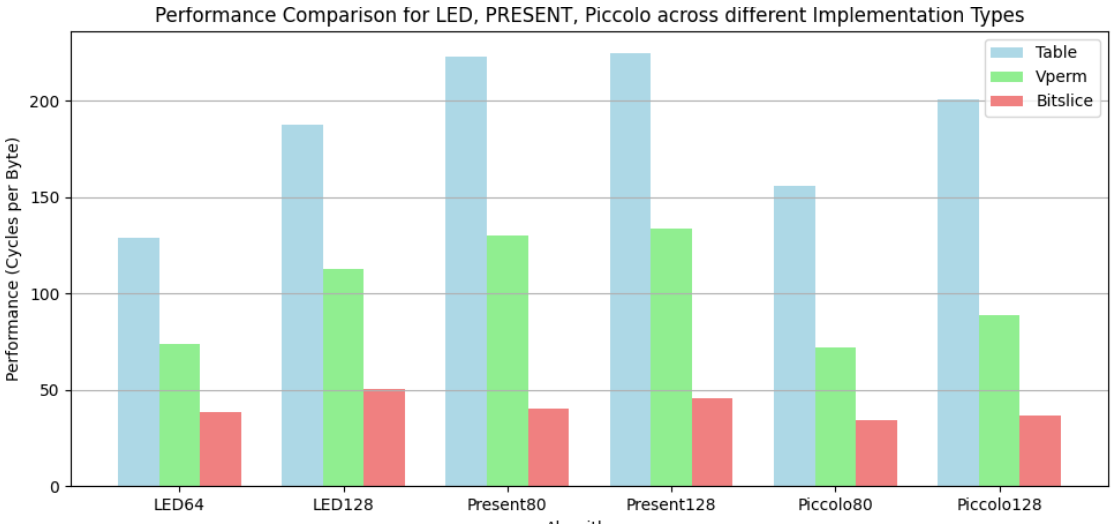

Figure 7.4: LED, PRESENT, Piccolo performance summarization

like Piccolo80 and LED64, exhibit characteristics that align well with a balanced trade-off between energy efficiency and performance depicted in figure 7.5 and 7.7.

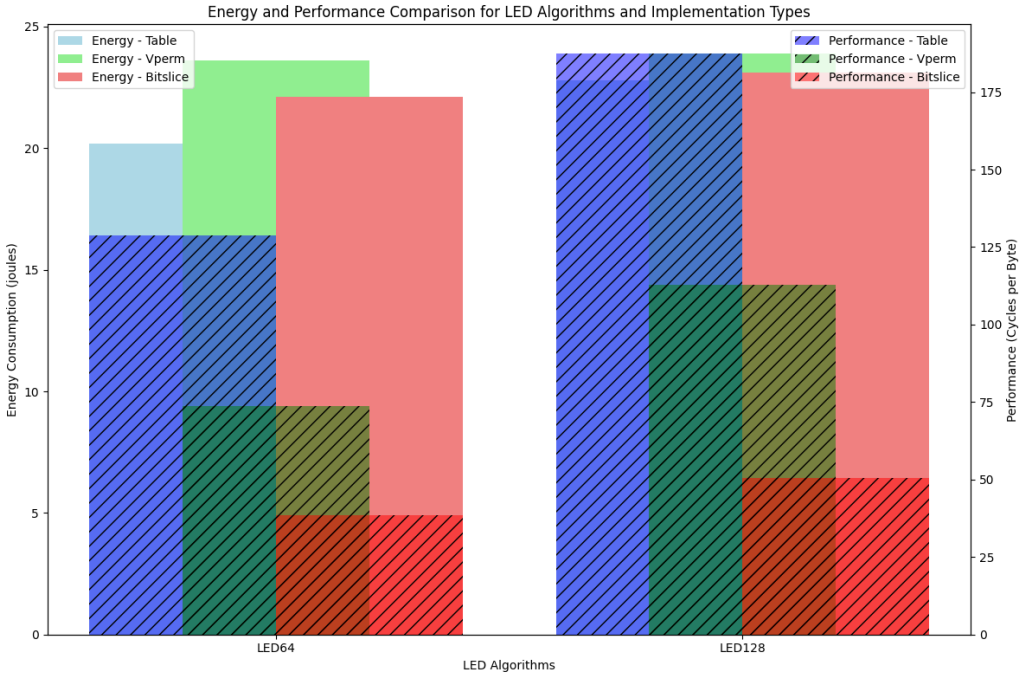

Figure 7.5: LED - Energy vs Performance

For LED128, there is a noticeable increase in cycles per byte compared to LED64, and also with the cost of slightly higher energy consumption as shown in figure 7.5. In con-

trast, PRESENT has relatively high energy consumption and comparatively moderate performance, suggesting a potential trade-off that may not be as balanced as it is for LED, displayed in figure 7.6. Also, there is not much difference in performance levels between PRESENT80 and PRESENT128 for all three implementation types. For Piccolo Figure 7.7, both vperm and bitslice seem to have good balance points between energy efficiency and performance with bitslice offering better balance than vperm. Between Piccolo80 and Piccolo128, table-based and vperm show better performance for Piccolo80, while bitslice the cycles required per byte remain in close range for the two key sizes.

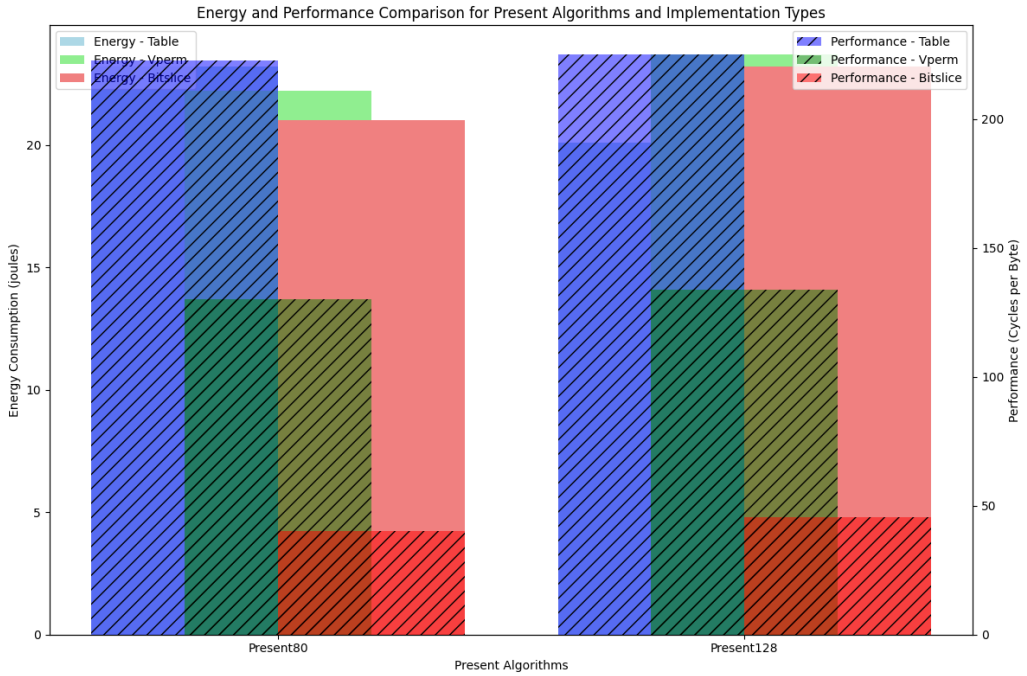

Figure 7.6: PRESENT - Energy vs Performance

In conclusion, vperm implementations show much better performance than table-based, while bitslice has a clear pattern of being the best performing one among the three. Considering that the table-based method might also have increased energy usage from the memory overhead from the table lookups, the bitslice method seems to come out on top for providing energy and performance balance. This could be because a significant portion of the encryption cost stems from the Sboxes, but the bitslice representation enables simultaneous computation of multiple Sboxes within a few clock cycles [28]. Hence, bringing improving the number of clock cycles required for performing the computation.

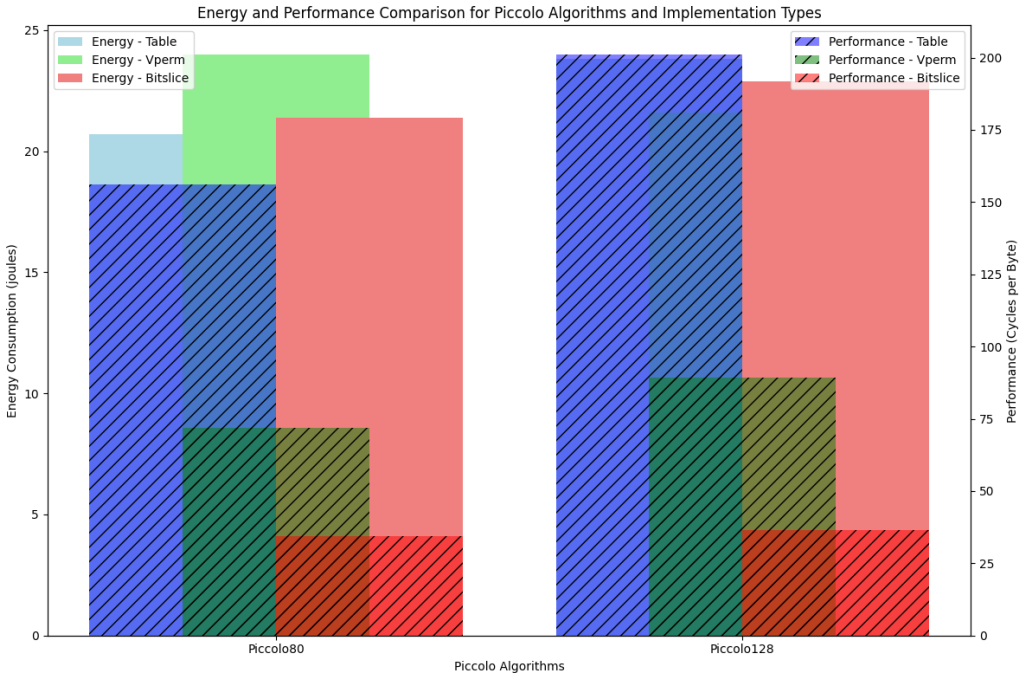

Figure 7.7: Piccolo - Energy vs Performance

Chapter 8

Threats To Validity

The threats to the validity of an experiment can be categorized into different types. We discuss here the potential threats according to the classification framework proposed by Wohlin *et al.* [33]

8.1 Internal Validity

In the experimental setup where code implementations of cryptographic algorithms are sourced from the internet and implemented in the same programming language, internal validity is threatened by some factors - differences in code quality, incomplete documentation, and variations in the expertise and coding styles of the original authors may introduce inconsistencies. The adaptability and scaling of implementations for different variation types and key sizes need careful consideration to maintain uniform experimental conditions. While we ensured that external code dependencies were close to none, potential biases from unanticipated variability and the sources' reliability could impact the experiment's internal validity.

8.2 External Validity

In the context of experimenting on a remote server with Ubuntu OS and Intel Xeon core, the study's external validity may face constraints. We covered varieties by including multiple lightweight block ciphers (LED, PRESENT, Piccolo, GIFT), AES, and multiple key sizes, implementations, and compiler configurations. However, the generalizability of the findings to broader systems, operating environments, or hardware architectures may be limited due to the specificity of the chosen setup. The remote server configuration with Intel Xeon core

does not represent the full spectrum of computing environments. Additionally, focusing on specific lightweight block ciphers and chosen compiler configurations for one algorithm might restrict the external applicability of the results to a broader range of cryptographic techniques and compiler optimizations.

8.3 Construct Validity

The chosen metrics, energy consumption in joules and cycles per byte for performance, provide quantifiable measures for comparing and analyzing the performance and efficiency of the cryptographic algorithms. We encountered a limitation in the Intel PCM setup, constrained by the 2-second mean power requirement. If we categorize analysis into three levels: firstly, monitoring energy consumption across various algorithms; secondly, examining energy consumption for different implementations and key sizes within each algorithm; and thirdly, exploring energy consumption at the sub-operation level (sub-bytes, shift rows, SP network, etc.). We were able to achieve the following depth -
LEVEL 1 - Attained with diverse algorithms.
LEVEL 2 - Attained by exploring various implementations of the same algorithms.
LEVEL 3 - Not attained, due to the rapid execution of sub-operations (such as sub-bytes, shift rows, SP network, etc.) occurring within less than a second, and our monitoring tool cannot capture data at such fine temporal granularity.

The experiments are performed on an intel processor server without modifying any default settings. However, as pointed out in [28] the Turbo Boost technology on the Intel CPU might cause dynamic upscaling of the processor leading to local overclocking. "Local overclocking" refers to the temporary increase in the clock frequency of a specific processor core. The turbo boost setting needs to be manually disabled to avoid this situation. Hence this experiment can be re-performed to see if performance results could be more consistent than achieved in this study.

8.4 Conclusion Validity

Our setup provides distinct results for energy efficiency and performance, aligning with the expected behavior and supporting our assumptions. While the evaluation of execution speed across different design choices for the algorithms is not measured in this study, it presents an avenue for future research.

Chapter 9

Conclusion

In conclusion, this study contributes a thorough examination of energy consumption patterns, including key sizes (64/80/128/192/256 bits) and implementation types (table-based, vperm, and bitsliced) for LED, Piccolo, PRESENT, and various configurations of AES. Additionally, the research explores the impact of compiler optimization levels on energy usage in the GIFT (68-128, 128-128) cipher. This investigation yields valuable perspectives in energy consumption and performance trade-offs for these lightweight block ciphers. Notably, the data indicates that lightweight implementations tend to exhibit lower energy consumption, with bitslice methods consistently outperforming other competing algorithms across different algorithms and key sizes. PRESENT128 and LED64 exhibit the most favorable average energy levels in table-based configurations when table lookup overhead is not considered. Meanwhile, Piccolo128 demonstrates optimal energy efficiency with vperm implementation, PRESENT80 leads in bitslice configuration, and AES's tiny version shows the lowest energy consumption among all. However, the average energy consumption range remains relatively close among these lightweight ciphers for a large number of execution iterations. Key size impacts and implementation method comparisons shed light on energy efficiency variations that can used for security considerations, optimizing computational complexity, and energy efficiency in cryptographic algorithms. Additionally, the examination of compiler configurations in the context of the GIFT algorithm underscores the importance of optimization techniques in minimizing energy usage. Our findings also indicate the possibility of achieving a balanced trade-off between energy consumption and performance in LED, PRESENT, and Piccolo algorithms. While there is no one-size-fits-all algorithm, in terms of energy consumption, the findings can be employed for carefully selecting cryptographic algorithms, implementation methods, and compiler configurations

based on the specific requirements and constraints of a given application and computing environment.

Given the focus on lightweight ciphers, the learnings from this study can be used to develop entirely new lightweight cryptographic algorithms or enhance existing ones. For example, one possible future work can be implementing a bitslice version of block cipher, comparing it with the non-bitslice version, and observing the energy efficiency and performance gain. Also, a deeper analysis can be performed for implementation choices, including the impact of Sbox design, table size, and levels of parallelism, which could further enhance our understanding of the factors influencing overall cryptographic algorithm behavior. Furthermore, investigating parameters such as memory consumption and cache latencies can contribute to a more holistic evaluation alongside the energy evaluation presented in this experiment. Finally, considering the duration of execution as a metric for assessing the speed of cryptographic algorithms can offer valuable perspectives on overall performance.

References

[1] Shawkat K. Guirguis Omar G. Abood. **A Survey on Cryptography Algorithms**. 2018. 3

[2] Neil Daswani and Dan Boneh. **Experimenting with Electronic Commerce on the PalmPilot**. Berlin, Heidelberg, 1999. Springer-Verlag. 3

[3] W. Freeman and E. Miller. **An experimental analysis of cryptographic overhead in performance-critical systems**. In *MASCOTS '99. Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 348–357, 1999. 3

[4] W. Stallings. *Cryptography and Network Security.* Prentice Hall, 1995. 3

[5] Christophe Cannière, Alex Biryukov, and Bart Preneel. **An introduction to Block Cipher Cryptanalysis**. *Proceedings of the IEEE*, **94**:346 – 356, 03 2006. 3, 4

[6] N.R. Potlapally, S. Ravi, A. Raghunathan, and N.K. Jha. **A study of the energy consumption characteristics of cryptographic algorithms and security protocols**. *IEEE Transactions on Mobile Computing*, **5**(2):128–143, 2006. 4, 12, 15, 38

[7] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. **Midori: A Block Cipher for Low Energy**. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 411–436, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. 6, 13, 15

[8] **Advanced Encryption Standard**, 2023. [Online; accessed 5-Jan-2023]. 6

REFERENCES

[9] Bora Aslan, Füsun Yavuzer Aslan, and M. Tolga Sakalli. **Energy Consumption Analysis of Lightweight Cryptographic Algorithms That Can Be Used in the Security of Internet of Things Applications**. *Security and Communication Networks*, **2020**:8837671, Nov 2020. 6, 13, 15

[10] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. **The LED block cipher**. **6917**, pages 326–341, 01 1970. 6

[11] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. **PRESENT: An Ultra-Lightweight Block Cipher**. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 7

[12] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. **Piccolo: An Ultra-Lightweight Blockcipher**. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 342–357, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 7

[13] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. **GIFT: A Small Present**. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 321–345, Cham, 2017. Springer International Publishing. 7, 21

[14] Okamura Toshihiko. **Lightweight cryptography applicable to various IoT devices**. *NEC Technical Journal*, **12**(1):67–71, 2017. 8

[15] Luís Cruz. **Tools to Measure Software Energy Consumption from your Computer**, 2021. (Last accessed August 2023). 9

[16] Jiehong Wu, Ilia Detchenkov, and Yang Cao. **A study on the power consumption of using cryptography algorithms in mobile devices**. In *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 957–959, 2016. 12, 15

[17] C. Datsios, G. Keramidas, D. Serpanos, and P. Soufrilas. **Performance and power trade-offs for cryptographic applications in embedded processors**. In *IEEE International Symposium on Signal Processing and Information Technology*, pages 000092–000095, 2013. 14, 15

[18] Boyou Zhou, Manuel Egele, and Ajay Joshi. **High-performance low-energy implementation of cryptographic algorithms on a programmable SoC for IoT devices**. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2017. 14, 15

[19] Mohamed Haleem, Chetan Mathur, Rajarathnam Chandramouli, and Koduvayoor Subbalakshmi. **Opportunistic Encryption: A Trade-Off between Security and Throughput in Wireless Networks**. *IEEE Transactions on Dependable and Secure Computing*, **4**(4):313–324, 2007. 14, 15

[20] **Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization**, 2022. [Online; accessed 5-Jan-2024]. 17

[21] Kashif Khan, Mikael Hirki, Tapio Niemi, Jukka Nurminen, and Zhonghong Ou. **RAPL in Action: Experiences in Using RAPL for Power Measurements**. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, **3**, 01 2018. 18

[22] Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. **Measuring Energy and Power with PAPI**. In *2012 41st International Conference on Parallel Processing Workshops*, pages 262–268, 2012. 17

[23] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. **The yin and yang of power and performance for asymmetric hardware and managed software**. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, page 225–236, USA, 2012. IEEE Computer Society. 17, 18

[24] Daniel Hackenberg, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt, and Wolfgang E. Nagel. **Power measurement techniques on standard compute nodes: A quantitative comparison**. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204, 2013. 17

[25] **Intel Architecture Manual**, 2022. [Online; accessed 5-Jan-2024]. 18

[26] **Grafana Integration with Intel PCM**. [Online; accessed 5-Jan-2024]. 19

[27] **Lightweight-crypto-lib**. [Online; accessed Oct-2023]. 20, 21

[28] Ryad Benadjila, Jian Guo, Victor Lomné, and Thomas Peyrin. **Implementing Lightweight Block Ciphers on x86 Architectures**. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, pages 324–351, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 20, 21, 35, 38, 43, 46

[29] **AES in C**. [Online; accessed Oct-2023]. 21

[30] **Tiny - AES**. [Online; accessed Oct-2023]. 21

[31] **Bitsliced - AES**. [Online; accessed Oct-2023]. 21

[32] **GIFT Cipher**. [Online; accessed Oct-2023]. 21

[33] Claes Wholin, Per Runeson, Martin Host, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. **Experimentation in software engineering: an introduction**. *Massachusetts: Kluwer Academic Publishers*, 2000. 45