Evaluation of lightweight data integrity algorithms for the Internet of Things

TC Mangole

orcid.org/0000-0003-2008-8110

Dissertation accepted in fulfilment of the requirements for the degree *Master of Science in Engineering Sciences with Computer and Electronic Engineering* at the North-West University

Supervisor:	Prof ASJ Helberg
Co-supervisor:	Dr KK Nair

Graduation:	November 2022
Student number:	23829923

DEDICATION

I dedicate this research dissertation to my late father, Mr. Botiti Mosweu Mangole. Without your teachings, I doubt I would have embarked on this journey.

ACKNOWLEDGEMENTS

I would like to express my special thanks and gratitude to everyone who participated in making this research successful. Firstly, I would like to thank God for giving me the potential and the strength to complete this research.

I am extremely grateful to my supervisors, Prof. A.S.J Helberg and Dr. K.K Nair, for their invaluable advice, continuous support, and patience during my Master's degree. Their immense knowledge and plentiful experience have encouraged me throughout this research project.

My gratitude extends to my colleagues at the Council of Scientific and Industrial Research (CSIR) for technical support and brain-storming sessions.

My appreciation also goes out to my mother, Mrs. M.E. Mangole, for always believing in me; my siblings, nephews, and nieces for the prayers; and for always being there for me.

I must not forget my friends for their encouragement and moral support throughout the completion of this project.

Thank you so much! *Ke leboga go menagane!*

ABSTRACT

Advances in ubiquitous computing have led to the use of the Internet of Things (IoT). IoT interconnects various physical objects, such as computers, mobile phones, vehicles, and sensors, to the internet. In an IoT platform, sensors and actuators monitor, collect, and process the data from their environment. Since the IoT is made up of devices with limited resources, such as limited memory, computation power, and processing speed, primitive cryptographic algorithms are not practical for them. Due to this, lightweight cryptographic algorithms are used in IoT systems to provide the necessary security.

The selection of a cryptographic algorithm is critical in a resource-constrained environment for improving performance and security. With the widespread use of resource-constrained devices that should be secure, it is difficult to development lightweight cryptographic hash functions that are able to meet security requirements while also providing good performance.

A lightweight *cryptographer/designer* has to deal with the trade-off between security, cost, and performance. Only two of these, security and low costs, security and performance, or low costs and performance, can be easily implemented, but not all three at the same time. To balance the trade-off between security, cost, and performance, lightweight hash functions with smaller internal state sizes, smaller output sizes, and short message sizes should be used. With many lightweight algorithms being researched and developed, it is not easy for researchers and designers to know which one they should choose for their application. Performance evaluation of hash functions is necessary, as it updates the body of knowledge and gives academics a guideline for selecting lightweight hash functions that are secure and can perform well for their applications' needs. This study aims to update the body of knowledge on the latency and memory consumption performance of two widely used lightweight hash functions called SPONGENT and PHOTON. Extensive research around SPONGENT, which is a family of thirteen variants, and PHOTON, which is a family of five variants, has been done. They are also listed as standardised lightweight hash functions in ISO/IEC 29192 Part 5. The Raspberry Pi 3 Model B was used to test and evaluate SPONGENT and PHOTON in two test cases: Test Case 1 analyses and evaluates the performance when the data log file used as input to the hash increases in size. Test case 2 studies how adapting the capacity/input rate affects performance.

Keywords—Internet of Things (IoT), lightweight cryptographic algorithms, constrained devices, data integrity, SPONGENT, PHOTON hash function

PUBLICATION DERIVED FROM THIS DISSERTATION

Portions of this work were published in the ICTAS 2022 conference proceedings, as shown in the citation below:

T. Mangole, A. S. J. Helberg and K. K. Nair, "**Resource Usage Evaluation of the PHOTON Hash Function**"*2022 Conference on Information Communications Technology and Society (ICTAS)*, 2022, pp. 1-6, doi: 10.1109/ICTAS53252.2022.9744686.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 Introduction

The technological advancement of pervasive computing has paved the way for massive communication of various devices. This has led to a new technology known as the Internet of Things (IoT). This technology includes human beings, computers, vehicles, smartphones and devices hereby referred to as “things”; connected to the Internet to collect data, transmit it, process it and make (real-time) decisions. [1] Explained how radio frequency identification (RFID) as the main technology for IoT allows microchips to transmit the identification information to a reader through wireless communications. The IoT is shaped by various traditional technologies such as RFID and Wireless Sensor Networks (WSNs), which is a group of sensors that monitor and control physical or environmental conditions communicating over the network [2]. WSNs can be widely used in real-time systems.

Several researchers [1],[2], show that Radio Frequency Identification (RFID) and Wireless Sensor Networks (WSNs) are the most widely used technologies for IoT. [3] Describes RFID as “a technology that allows microchips to transmit the identification information to a reader through wireless communication.” Furthermore, WSN is made up of sensors with small volume, low cost, and low electricity consumption; that monitor and control environmental conditions [3]. The WSNs can be extensively used to develop real-time systems.

In general, the architecture of the IoT is made of three layers, namely [3]: the perception layer, also known as the physical layer, collects data through the use of "things" such as sensors and actuators; the network layer, also known as middleware, receives data from the perception layer and transmits it to the users portal (e.g. web application, cloud, etc.). The application layer presents data to the end-users in a human-readable format such as a webpage. Figure 1-1 shows a generic IoT architecture.

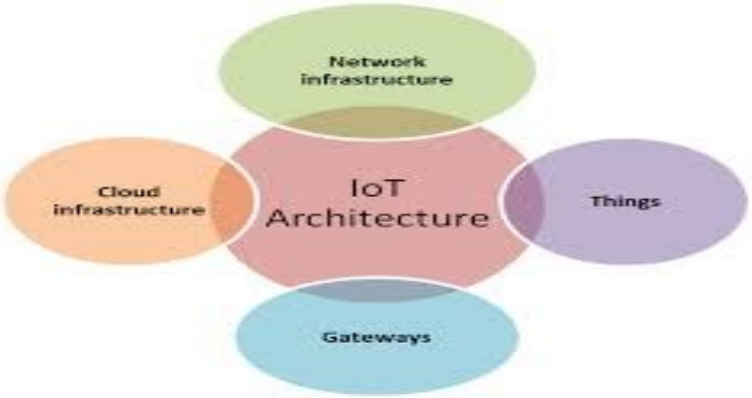

Figure 1-1: A generic context diagram depicting the IoT architecture [4]

The components depicted in Figure 1-1 are grouped as follows:

- Things – these are the objects/ “things” found in the perception layer, they include sensors, smart phones. These objects do not necessarily need to be managed with human interaction.
- Gateways – a device or application that translates data into the format required by the network. They serve as a portal to give the “things” access to the internet.
- Network infrastructure – transmits the data that was collected by “things” and also controls the data flow, such as ZigBee, Wi-Fi, 5G [5].
- Cloud infrastructure – a virtualized software or hardware such as servers and personal computers used to store data.

The study conducted by Gartner Inc. estimated that devices that exclude PCs, tablets, and smartphones, participating in IoT, were expected to increase to 26 billion by 2020 , compared to 2009, when they were about 0.9 billion [6]. Furthermore, the 2015 Hype Cycle for emerging technologies showed that IoT was at the peak of inflated expectations [7]. In contrast, ABI Research [8] estimated that by 2020, 30 billion devices will be wirelessly connected to the IoT platform. This shows that IoT applications are expected to increase tremendously. Despite these trends, the study conducted by Hewlett Packard in 2014 revealed that 70% of the devices most used in the IoT have severe vulnerabilities [9], [10]. Additionally, the 2019 Hype Cycle for IoT shows IoT security is at the peak of inflated expectations [11]. [12] explained that these vulnerabilities are the result of a lack of data and transport encryption techniques, insecure web interfaces, inadequate software protection, and inadequate authorisation. As a result, cryptographic algorithms are used to address these security issues.

The need to address security challenges is of high priority so that the adoption of IoT by its stakeholders - consumers, businesses, and government can be successful. In order for IoT to be adopted and be sustained the new regulatory approaches to ensure privacy and security should be implemented [6]. Information security has properties/requirements that should be implemented in order to have a secured application. These requirements are data Confidentiality, Integrity, and Availability known as the CIA triad – a reference model made of information security principles [13],[14]. During data transmission from a sender to the intended receiver the following should happened to ensure security:

- Confidentiality – Only authorised users and processes should be able to access the data. This is achieved by encryption/decryption, which is a process of scrambling a message so that the authorised recipient can de-scramble it.
- Integrity – involves maintaining the consistency, accuracy and trustworthiness of the data sent to the recipient. Data integrity can be achieved using primitive hash functions such as MD5, SHA-3, and RC6.
- Availability – the services and data should be always available to authorised users. Disaster recovery plan and backup can mitigate threats that compromise data availability.

Trends have shown that resource-constrained devices are expected to increase in numbers, requiring to be protected together with the applications in which they are deployed such as IoT. Cryptographic algorithms have been used to secure data. Due to complex mathematical operations, large internal state sizes, and long keys, primitive hash functions like MD5 and the SHA-3 family are resource intensive, particularly in power and memory [15], [16]. In resource-constrained devices utilised in IoT applications, such hash functions are infeasible [15]. To mitigate this, lightweight cryptography [17] has been introduced as a subset of cryptography that allows primitive hash functions to fit into resource-constrained IoT devices.

The selection of a cryptographic algorithm is critical in a resource-constrained environment for improving performance and security. Several notable studies [18]–[20] have been conducted to balance the performance, security, and cost of lightweight cryptographic algorithms, as well as to analyse lightweight cryptographic algorithms for "smart" applications [21]. Implementing lightweight hash functions using smaller internal state sizes, output sizes, and message sizes is one way to achieve a balance between security, cost, and performance [15].

1.2 Problem Statement

The widespread use of resource-constrained devices is making the development of lightweight cryptographic hash functions which are secure while also providing good performance difficult

[22]. The survey by [23], highlights that there are more than 100 lightweight cryptographic algorithms developed, with only 9 which are lightweight hash functions. The exponential deployment of resource-constrained devices will allow academia to continue researching and developing more lightweight cryptographic algorithms to address emerging security concerns. With so many lightweight algorithms to choose from, how does one choose one hash function over the other? The analysis and evaluation of existing lightweight hash functions is necessary to find out how they perform in resource-constrained devices. Performance evaluation of hash functions is necessary, as it updates the body of knowledge and gives academics a guideline for selecting lightweight hash functions that are secure and can perform well for their applications' needs.

1.3 Research Motivation

This work intends to analyse and evaluate the performance of widely used lightweight hash functions. The findings of the memory and latency performance analysis reveal that the optimal combination for the individual application and the IoT platform envisaged may be realized.

1.4 Research Questions

The following research questions arise and are intended to be answered in this study:

1. Which popular lightweight cryptographic hash functions are available for resource-constrained nodes?
2. What are measures that can be used to evaluate the performance of the hash functions in resource-constrained devices?
3. How do lightweight cryptographic hash functions compare in terms of the identified measures?

1.5 Research Objectives

The following research objectives are set to be able to achieve the research goal for this study:

1. Identify typical constraints of the IoT node;
2. Identify measures for evaluating the performance of lightweight hash functions;
3. Identify two widely used lightweight cryptographic hash functions used for data integrity in IoT nodes;
4. Implement the identified lightweight hash functions; and
5. Evaluate the performance of the identified lightweight hash functions.

1.6 Research Approach

The targeted goal of this study is to evaluate the performance of two widely used cryptographic data integrity algorithms used in resource-limited IoT devices and IoT applications. The study followed an empirical approach. Empirical research is a data-driven study that results in conclusions that can be validated by observation or experiment. The ability of the experimenter to modify one of the variables to investigate its effects is one of the primary characteristics of empirical research [24].

A quantitative approach was followed to achieve the research objectives stated. The quantitative method employs numerical measurements or counts. The approach to quantitative methods is based on structured and validated data collection instruments [25]. Methods such as simulation and experimentation could be used to collect data for a quantitative study. Simulation, as indicated in [24], entails the creation of an artificial environment in which relevant information and data can be generated. Experimentation allows the researcher to "observe the effect of a given intervention," but tools and equipment can be costly to purchase and may also be difficult to explain the findings [26].

In order to be able to address the research questions and meet the objectives, there is a need to have a systematic approach. The approach taken towards the completion of this research is given below:

1. Conduct a background and literature review to achieve research objectives 1 to 3. This section surveyed the state-of-the-art research studies to provide the foundation for the research goal.
2. Determine the equipment for the data collection and implementation environment.
3. Validate and implement the identified hash functions using the determined equipment.
4. Evaluate the performance given the standard, i.e., original and modified implementation parameters.
5. Draw conclusions and make recommendations.

1.7 Delimitations, Limitations and Assumptions

The identified hash functions are limited to two commonly used version families, and it is assumed that they are secure and do not have any known security vulnerabilities.

1.8 Dissertation Layout

The rest of the dissertation is structured as follows:

Chapter 2: Presents the literature review which outlines the current-state-of-art and the work done by other researchers within the same research field. It starts by giving the background on IoT, security challenges in IoT applications, and reviews the existing lightweight hash functions.

Chapter 3: Gives the background for research design and explains how the problem will be solved.

Chapter 4: Provides the research implementation details and experimental setup and procedure.

Chapter 5: Provides the results analysis and discussions.

Chapter 6: Concludes the research study by providing answers to the research questions posed in Chapter 1 and gives direction on the possible future work.

CHAPTER 2 LITERATURE STUDY

2.1 Introduction

This chapter provides a background of the research for this study. It begins with a report on the background of the IoT, including IoT application domains. The security challenges in IoT applications are also highlighted. The main focus of this chapter is on the current literature on the performance of lightweight cryptographic hash functions utilised in resource-constrained devices used in IoT platforms.

2.2 Background

The use of the internet has evolved enormously, enabling traditional computer devices, sensors, and actuators to communicate through wired/wireless networks and share data and information with the aim of achieving a common goal [14]. According to [14], [27], the IoT was originally formulated by Kevin Ashton in 1999 with the aim of leveraging the functionality of Radio Frequency Identification Devices (RFID) to electronically identify and interact over the internet [28].

There is no standard definition of IoT, but different definitions of IoT are provided in [28], [29]. For the purpose of this study, Kranenburg's IoT description is adopted as *"a dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual "things" have identities, physical attributes, virtual personalities, and use intelligent interfaces and are seamlessly integrated into the information network"* [30]. IoT components include actuators, sensors, the Global Positioning System (GPS), laser scanners, RFID, Wireless Sensor Networks (WSNs), mobile phones, and people [31]. These components could constantly collect data around their environment using sensors and actuators, process the collected data, and store it either in the cloud or on a server. Integration of these components and the communication protocols serves as the foundation of IoT, allowing them to communicate with each other over the internet.

Due to the high demand for IoT applications, there is a need for appropriate security implementation [32]. Lightweight cryptographic hash functions are among the major research work addressing such a need. It is mandatory for researchers in the field of cryptography to improve and enhance the strength of primitive cryptographic algorithms. A lightweight cryptographer has to deal with the trade-off between security, cost, and performance.

Only two of these, security and low costs, security and performance, and low costs and performance are easily implemented [33], and not all three simultaneously.

2.3 Application Domains

Various sectors are adopting IoT as they are starting to realise its benefits. Applications of IoT can be found in almost any field one can think of, as shown in [29]. For example, in the smart city domain, possible applications could be traffic congestion monitoring, building management, and smart energy. The smartness of the devices (i.e. the ability to remotely identify, locate, track, monitor, and manage) makes this possible [3]. The various protocols, such as Near Field Communication, Bluetooth Low Energy, and ZigBee, are used to track and monitor their environment, while sensors are used to detect distinct variables such as temperature, pressure, and altitude. IoT is expected to make our daily lives much more convenient, for example through home automation allowing home residents to switch on the kettle, geyser, and TV when the lights go on, book appointments, and make calls automatically. Nonetheless, these systems can still be hacked, so security is highly important.

2.4 Generic IoT architecture

Compared to the well-known Open Systems Interconnection (OSI) model, the IoT architecture has either three [31] or more layers [28], [34]. Hinai and Singh [35] explain that IoT layers differ according to the "roles they play and the devices operating on them." In this study, the IoT architecture has three layers, as depicted in Figure 2-1.

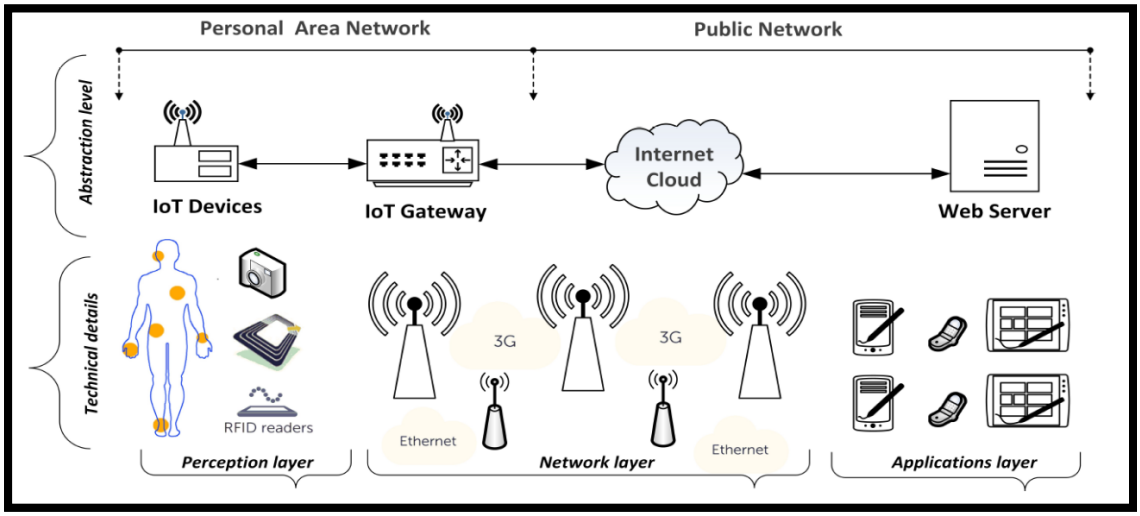

Figure 2-1: A generic IoT architecture [36]

- **Perception layer**

This is the layer where the data might originate. It is made up of sensors, RFID tags, WSNs, and cameras, to name a few, collecting and sensing the information of objects anytime, anywhere [31].

- **Network layer**

The collected data in the perception layer is transmitted through wireless and wired networks. The technologies involved include, internet, Wi-Fi, WiMax, Bluetooth, Zigbee, mobile communication (such as 3G, General Packet Radio Service (GPRS), LTE, 5G) and gateways are found in this layer for aggregating, filtering and moving data between sensors and other devices [35].

- **Application layer**

This is a graphical user interface between the applications and the end users. It offers IoT services tailored to the needs of end users [28]. It uses the received information to make control decisions, allowing the achievement of intelligent processing.

2.5 Security Challenges

2.5.1 Information security goals

Swamy, Jadhav, and Kulkarni [14], demonstrate that the significant security goals of the IoT are to afford a reliable connection, proper authentication techniques, and provide confidentiality about the data to each connected device. The typical information security triad model advocating data Confidentiality, Integrity, and Availability (CIA) is still applicable in IoT. A breach of any of the security goals could result in enormous damage to the system.

Regardless of the potential of IoT and the benefits realised by this technology, there are still many security challenges that might hinder the successful adoption of IoT [37]. The IoT applications are connected to the internet, which has its own security issues that are still not entirely solved. Therefore, these security challenges are exacerbated in the IoT space. The comprehensive analysis of security attacks and threats prone to each IoT layer by Hinai and Singh [35] proves there is a need for security solutions.

The extensive adoption of IoT technologies and services is substantially dependent on the security issues caused by the complexity, deployment, and mobility of the application [1]. Furthermore, Hinai and Singh [35] have indicated security concerns across the architecture layers related to CIA such as unauthorised access, data breach, and threats in data manipulation. The enabling IoT technologies should be secured; however, it is complex due to billions of

interconnected devices increasing the potential threats. The 2014 Open Web Application Security Project (OWASP) Internet of Things project led by Miessler and Smith has identified the top ten vulnerabilities that could be exploited if countermeasures are not implemented as presented by [38]. Table 2-1 summaries the IoT architecture layers susceptible to vulnerabilities.

Table 2-1: OWASP top ten vulnerabilities in IoT architecture [38]**,** [39]

Vulnerability	Perception layer	Network layer	Application layer
Insecure Web Interface		✓	✓
Insufficient Authentication/Authorisation	✓	✓	✓
Insecure Network Services		✓	
Lack of Transport Encryption/Integrity Verification	✓	✓	
Privacy Concerns	✓	✓	✓
Insecure Cloud Interface			✓
Insecure Mobile Interface		✓	✓
Insufficient Security Configurability	✓	✓	✓
Insecure Software/Firmware	✓		
Poor Physical Security	✓	✓	

2.6 Data Integrity as Information Security Goal

The focal point of this subsection is to explore data integrity as one of the security goals in the context of the Internet of Things. It is argued in [40] that data can be corrupted by software bugs or configuration errors. Furthermore, it is highlighted that the primitive approaches to proving the integrity of data are done either by a client or a third party. Data integrity as one of the information security requirements is imperative, as recommended in the International Telecommunication Union Telecommunications Standardisation Sector (ITU-T Y.2066) [41]. ITU-T Y.2066 recommends common requirements for the Internet of Things independently of any application domain. ITU-T is an ITU sector that develops standards and makes recommendations for the telecommunications industry.

The common requirements of the IoT specified in this recommendation are classified into the categories of non-functional requirements, application support requirements, service requirements, communication requirements, device requirements, data management requirements, and security and privacy protection requirements.

The functional requirements from the ITU-T Y.2066 recommendation relevant to this study are as follows [41]:

- Data validation is required through integrity checking and life cycle management to ensure high availability and reliability of data.
- Communication security is required to prevent unauthorized access to data, ensure data integrity, and protect data privacy-related content during data transmission or transfer in IoT.

Implementing information security requirements can significantly contribute to the successful deployment of IoT.

2.6.1 Hash functions

A hash function is a non-reversible mathematical function that takes a message of arbitrary length and produces a fixed length output known as a message digest or hash value.

The security properties of a hash function are as follows [42], [43]:

- Preimage resistance (one-way): It should be *hard* (i.e., *computationally infeasible*) to determine the input message (preimage) corresponding to a given hash value. This property implements the *one-way characteristic* of a hash function.
- Second preimage resistance (weak collision resistance): Given the hash value and the input message, it should be infeasible to get another input message with the same given hash value.
- Collision resistance (strong collision resistance): It should be *computationally infeasible* to produce the same hash value given two input messages.

Some of the primitive/traditional hash functions that have been used are MD5 (Message Digest 5) and SHA-x family.

Message Digest 5 (MD5) is a cryptographic algorithm that “takes as input a message of arbitrary length and produces as output a 128-bit *fingerprint* or *message digest* of the input” [44], [45]. MD5 was designed by Ron Rivest in the early 1990s after its predecessor, MD4, was identified to have security vulnerabilities. However, in 2010, the CMU Software Engineering Institute considered MD5 to be unsuitable for further use [46]. Applications like Secure Sockets Layer (SSL) certificates or digital signatures could not be secured by MD5. This led to the development of a cryptographic algorithm known as the Secure Hash Algorithm (SHA-x family).

The first SHA version to be implemented was SHA-1, followed by SHA-2, and then SHA-3. SHA-1 was designed in 1995 by the United States National Security Agency (NSA) to replace MD5. It was officially deprecated by NIST in 2011 due to the security weaknesses shown in various analyses and theoretical attacks. In February 2017, the Shattered.io team [47] revealed the first collision attack for full SHA-1; the team also reported that all systems that rely on SHA-1 are vulnerable. This attack proved that SHA-1 was no longer useful, and the industry should address this. Since SHA-2 has mathematical properties similar to those of SHA-1, this weakened SHA-2 and it ultimately experienced preimage and collision attacks [48]. In 2015, NIST approved SHA-3 as a replacement for SHA-2.

SHA-3, as the latest member of the Secure Hash Algorithm family, has been widely adopted to protect data integrity. NIST reported that none of these approved hash functions are suitable for constrained environments, primarily because they have large internal-state size requirements [15].

Traditional hash functions have been used in non-resource-constrained systems such as desktops/servers. Because of this, these hash functions cannot be implemented on resource-constrained devices used by IoT applications. The traditional hash functions consume a lot of resources as they have complex mathematical operations. This led to the research and implementation of lightweight hash functions [39].

2.6.2 Lightweight cryptography

Lightweight cryptography is a subfield of cryptography that aims to provide solutions tailored to resource-constrained devices [12]. The resource-constrained devices used in IoT have limited computational power, energy, and memory which make it infeasible to execute existing traditional hash functions [49]. Lightweight cryptography aims to find solutions to this issue [17]. The selection of a cryptographic algorithm plays a critical role in resource-constrained applications for improving performance and security. Several significant studies [18]–[20] have been conducted to balance the performance, security, and cost of lightweight cryptographic algorithms, as well as to analyse lightweight cryptographic algorithms for "smart" applications [21].

Recent developments in the field of lightweight cryptography have led to a need for NIST to standardise lightweight algorithms. The NIST Report [15] outlines the design considerations of algorithms implemented for constrained environments. NIST will use these considerations as guidelines to evaluate the designs of lightweight cryptographic algorithms submitted for standardisation.

Saarinen and Engels [50], emphasise the importance of implementation types, which are software and hardware. For each implementation type, there are hardware and software-specific metrics to be used to evaluate lightweight cryptographic algorithms. The categories of constraints for lightweight cryptography are also listed in part one of the International Standard ISO/IEC 29192 [51]. Additionally, the authors in [17] and [15] classified the performance metrics according to the implementation type as follows:

For **hardware implementation,** the memory complexity is described in terms of gate area, which can be stated in terms of logic blocks for field-programmable gate arrays (FPGAs) or Gate Equivalents (GEs) for ASIC implementations. The time complexity is measured in terms of throughput rate, i.e., bits per second for a particular frequency, which is usually 100 GHz, and the latency, which is the time taken for a process to finish.

For **software implementation,** the time complexity of an algorithm is measured according to the number of clock cycles per byte, also known as the latency. The memory complexity is measured according to the size of the RAM necessary to carry out the computation and the ROM space required to store the algorithms, e.g., flash memory. Lastly, power consumption estimates the power utilised based on the hardware technology. This metric is universal to both hardware and software, and the unit used is Watts.

The challenge in designing lightweight hash functions and other primitive cryptographic algorithms such as block ciphers and message authentication codes lies in achieving the balance between security, performance, and cost requirements [52] shown in Figure 2-2.

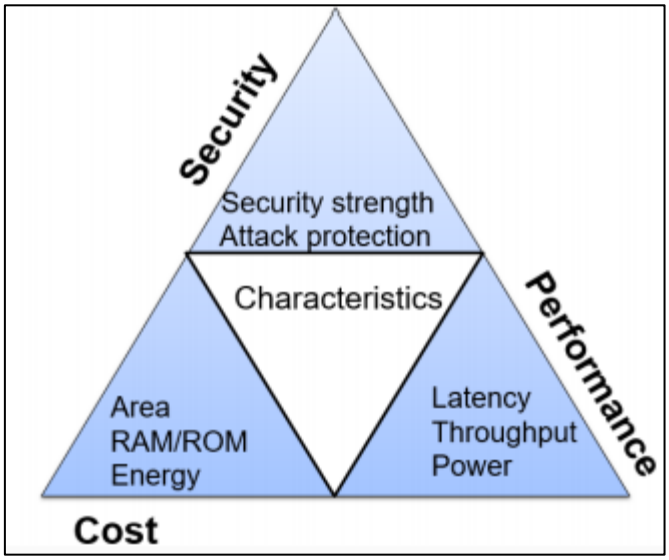

Figure 2-2: Metrics used in lightweight cryptographic algorithms [15]

In the field of lightweight cryptography, it is quite complex to find a balance between security, cost, and performance [53]. This limits the targeted application due to different requirements that should be met, therefore, making the lightweight algorithm specific to the application (not one size fits all) [54]. Devices with limited resources such as RFID, microcontrollers, sensors, and actuators may not necessarily process the traditional algorithms that were initially implemented for devices such as mobile phones and computers, necessitating the need for lightweight cryptography [3]. This study refers to resource-constrained devices as embedded devices lacking computer power, such as microcontrollers ranging from 4-bit to 64-bit [55], [15] with low CPU power and limited memory. These microcontrollers have a wide range of performance attributes depending on the hardware specifications [15]. Other studies, like [56], [57] have used the Raspberry Pi as a resource-constrained device to evaluate lightweight cryptography algorithms. The Raspberry Pi falls within a more resource constrained range compared to sensors and RFIDs. As a result, performing a latency and memory usage analysis on a larger system, such as the Raspberry Pi 3 Model B with 64-bit CPU and 1 GB RAM (more specifications are listed in section 4.2), allows the selection of a resource constrained node and hash variant, whereas a very small system (e.g., 8-bit and 16-bit microcontroller) would have provided a failure/operate result.

Many lightweight hash functions have been developed. The following section outlines the most common hash functions.

2.6.2.1 Existing lightweight cryptographic hash functions

Whirlwind [58] was proposed in 2010 to address the cryptanalytic attacks developed on the algorithms submitted for the NIST SHA-3 competition. Whirlwind is a software implementation hash function, and its design is based on the Whirlpool cipher, which uses the sponge construction. Sponge construction is an algorithm with finite internal state that takes an input of a bit stream of any length and processes it to produce an output bit stream of any length. Whirlwind is designed according to the Wide Trail strategy to offer higher security while keeping performance at the same level. The length of the internal state is 1024 bits. However, the performance of the software implementation is not exceptionally fast, but it compares favorably with the 512-bit versions of SHA-3 candidates submitted to the SHA-3 competition. The authors do not provide an explanation of the security strength levels of the three main security properties of the hash function.

PHOTON [59], proposed in 2011, is a lightweight hash function family of five variants with permutation sizes of 100, 144, 196, 256, and 288 bits, computing hash digests of lengths of 80, 128, 160, 224, and 256 bits, respectively. PHOTON is a hardware-based implementation with sponge construction and internal permutation comparable to the Advanced Encryption Standard (AES) design strategy with the addition of a new mixing layer mechanism. PHOTON variants

include a variety of security levels ranging from 64 bit preimage resistance to 128 bit collision resistance. The design of PHOTON was designed to keep the internal memory size as low as possible. The authors claimed that PHOTON is faster than other existing lightweight hash functions known in 2011.

SPONGENT [60], proposed in 2013, is a lightweight hash function family of thirteen variants with permutation sizes of 88, 136, 176, 240, and 272 bits, computing hash digests of lengths of 88, 128, 160, 224, and 256 bits, respectively. The proposed algorithm is a hardware-based implementation; its different variants use sponge construction with a PRESENT-type permutation. PRESENT is a compact hardware-oriented 64-bit block cipher supporting 80 and 128-bit keys [61]. Different SPONGENT variants offer three types of security: full preimage and second-preimage resistance; reduced second-preimage resistance; reduced preimage and second-preimage resistance; and accommodating applications with various security requirements. The SPONGENT variants can be used in security applications such as lightweight signature schemes, RFID security protocols, and random number generation. SPONGENT was implemented on an ASIC implementation environment, allowing low gate area and low-to-high range throughput, offering better performance.

Light-Weight One-way Cryptographic Hash Algorithm (LOCHA) [62], was proposed in 2014 to address the energy-constrained nodes used in WSNs. The hashing scheme used the binary representation of the input message. They assumed that the input message consists of characters belonging to the 96 printable ASCII character set. The hash value length is fixed at 96 bits, which is comparatively lower than Whirlwind. LOCHA maintains all three security properties: preimage resistance of 96 bits, second-preimage resistance, and collision resistance. The security levels of second-preimage resistance and collision resistance were not stated due to "space limitation" in their publication. The performance analysis was compared to SHA-1 and MD5, which are primitive hash functions that are not suitable for resource-constrained nodes. Although LOCHA produces a fixed and small length of the hash value, the authors compared their performance analysis with SHA-1 and MD5, although SHA-1 was deprecated by NIST in 2011.

Hash-One [33], proposed in 2016, is a hardware-based implementation and uses sponge construction that produces a hash value of 160 bits. The motive behind the design of Hash-One was to keep things simple, yet secure. Hash-one provides a security level of 80 bits against collision and second-preimage attacks with minimal computational complexity and area requirements.

2.7 Summary

This chapter presented the current literature on lightweight cryptography. The limitations of the resources used in the IoT and the performance measures for hardware and software implementations were discussed. From the literature review, it was noted that IoT applications are susceptible to various vulnerabilities, which could breach the CIA. This chapter gave the background of the IoT, including discussions of the existing lightweight algorithms used to ensure data integrity.

This chapter concludes by identifying SPONGENT and PHOTON as lightweight hash functions that are evaluated in this study. The identified hash functions are widely used and have been included in the international standard ISO/IEC 29192. SPONGENT and PHOTON are hardware-based hash functions, but in this study a software-based measurement is followed since many IoT devices will be implemented in a software solution. The software measurements, i.e., the memory and latency of these selected hash functions, are not fully researched as indicated by literature review. The next chapter will outline the systematic methods used to implement and evaluate SPONGENT and PHOTON in a resource-constrained device.

CHAPTER 3 METHODOLOGY

3.1 Introduction

This chapter introduces the experimental design and methodology used to answer the research questions posed in Chapter 1. The chapter provides a detailed description of the research design, data evaluation method, experiment objectives, setup, and methods used to test the identified cryptographic hash functions.

From the lightweight hash functions reviewed in Chapter 2, only SPONGENT and PHOTON published their reference implementation code. The experiments in this study were carried out using reference implementation codes in C++ and C for SPONGENT and PHOTON, respectively, obtained from the official websites of the SPONGENT[1] and PHOTON[2] designers. The versions of the downloaded code for SPONGENT and PHOTON are versions 1.0 and 20131205 respectively. Using reference implementation code assures the developers that the code follows the specification offered by the original designers and conforms to the specified behaviour.

3.2 Performance Measures

The research design evaluates the performance of the proposed lightweight hash functions in a constrained node. In addition, the research design involves the process of collecting data, analysing, interpreting, and identifying the best performing hash functions in IoT applications. The following measures were identified with the guidance of the lightweight cryptography implementation metrics in section 2.6.2.

Latency - Time taken for the task/instruction to be completed is also known as the response time and is measured in clock cycles or in milliseconds (ms). In this study, latency is defined as the time (in ms) taken to hash the data log file and print the hash digest as an output. Low latency is preferred because it means that the hash function is operating efficiently [63].

Memory - In IoT applications, there is a need to store the data and information in order to make some kind of decision. Memory consumption can be measured when the hash function is running or the code size of the hash function to determine if it will fit into the resource-constrained node. There are various ways of addressing this parameter, such as reducing code size by storing lookup tables generated in RAM, or maintaining a small memory footprint compared to code size [64]. This study will evaluate the memory consumption, which is the amount of physical memory,

[1] https://sites.google.com/site/spongenthash/
[2] https://sites.google.com/site/photonhashfunction/downloads

that is, RAM, used when the hash function is running. This is achieved by passing the executable application to a Linux built-in command known as */usr/bin/time*. This command gives the maximum resident set size, which is the amount of memory the process (i.e., the hash function program) utilises at its peak in kilobytes (kB).

3.2.1 Proposed hash functions

In the literature review, SPONGENT and PHOTON lightweight cryptographic hash functions were presented. The original SPONGENT and PHOTON reference implementation codes were obtained online. It is imperative to validate the implementation code to check if it is working as expected and was not tampered with to produce different results compared to the original publications. This was done by generating and comparing test vectors as explained in section 3.5.1.

SPONGENT [60] is a family of hash functions with thirteen variants. The various variants are referred to as *SPONGENT-n/c/r* where *n* is the hash size, *c* is the capacity, and *r* is the output rate. The remaining bits after the message blocks have been absorbed are referred to as the capacity *c*. SPONGENT uses sponge construction with PRESENT-like permutation. A sponge construction is defined as a "iterated design that takes an input with variable length and produce an output of an arbitrary length based on the permutation operating on a state of a fixed number of *b* bits" [60]. PRESENT is a compact (uses low gate area) hardware-oriented 64-bit block cipher supporting 80 and 128-bit keys targeting applications requiring moderate security levels (i.e. 80-bit) [61]. The architecture of PRESENT allows the calculation of round keys in SPONGENT variants. It encrypts and decrypts the input message in SPONGENT. PRESENT was developed specifically for resource constrained devices and is listed in ISO/IEC 29192 as a lightweight block cipher.

SPONGENT variants that have the same hash output are referred to as SPONGENT-n. This gives us five groups of SPONGENT variants, viz., SPONGENT-88, SPONGENT-128, SPONGENT-160, SPONGENT-224, and SPONGENT-256. The SPONGENT-88 is designed for extremely restricted applications that require low security preimage security levels. SPONGENT-128, SPONGENT-160 are designed for highly constrained applications with low to middle requirements for collision security. Lastly, SPONGENT-224 and SPONGENT-256 are compatible with standard embedded applications with their parameters corresponding to those of primitive hash functions SHA-2 and SHA-3. For each SPONGENT variant, the following security requirements were met:

- Full preimage and second-preimage security: Offers the standard security requirements for an n-bit output size hash function, which are collision resistant; preimage and second-preimage resistant. For this category, $r = n$ and $c = 2n$ to obtain SPONGENT-088-176-

088, SPONGENT-128-256-128, SPONGENT-160-320-160, SPONGENT-224-448-224 and SPONGENT-256-512-256. The collision resistance is $2^{n/2}$, and both preimage and second-preimage are both 2^n.

- Reduced second-preimage security: Offers reduced preimage and second-preimage resistances for small messages *r* for variants with $n \approx c$ which are SPONGENT-088-80-008, SPONGENT-128-128-008, SPONGENT-160-160-016, SPONGENT-224-224-016, and SPONGENT-256-256-016. The collision resistance is $2^{c/2}$; preimage and second-preimage are 2^{n-r} and $2^{c/2}$ respectively.
- Reduced preimage and second-preimage security: For applications where only the collision security is needed. To reduce both the preimage and the second-preimage security capacity, *c* and *r* are set to $c = n$ and $r = n/2$ respectively, to obtain SPONGENT-160-160-80, SPONGENT-224-224-112, and SPONGENT-256-256-128. The collision resistance is $2^{c/2}$; preimage and second-preimage are both $2^{n/2}$.

Table 3-1 shows a summary of the parameters of the SPONGENT variants. The internal state *b* is known as the width. The width of the sponge construction for both SPONGENT and PHOTON is represented as the sum of the output rate and the capacity, i.e., $b = r + c$. From Table 3-1, it can be seen that the capacity *c* depends on the security requirement to be achieved, as explained at the beginning of this subsection. The number of rounds depends on the block size *b*. For detailed functionality of SPONGENT, please refer to the pseudo-code in Figure 4-2.

Table 3-1: SPONGENT parameters [60]

	***n* (bit)**	***b* (bit)**	***c* (bit)**	***r* (bit)**	**Number of rounds**
SPONGENT-088-080-008	88	88	80	8	45
SPONGENT-088-176-088	88	264	176	88	135
SPONGENT-128-128-008	128	136	128	8	70
SPONGENT-128-256-128	128	384	256	128	195
SPONGENT-160-160-016	160	176	160	16	90
SPONGENT-160-160-080	160	240	160	80	120
SPONGENT-160-320-160	160	480	320	160	240
SPONGENT-224-224-016	224	240	224	16	120
SPONGENT-224-224-112	224	336	224	112	170
SPONGENT-224-448-224	224	672	448	224	340
SPONGENT-256-256-016	256	272	256	16	140
SPONGENT-256-256-128	256	384	256	128	195
SPONGENT-256-512-256	256	768	512	256	385

PHOTON [59] is a family of hash functions with five variants; each variant is defined by the hash output size *n*, its input bitrate *r*, and its output bitrate *r'*. The variants are formatted as *PHOTON-n/r/r'*. The input message divided into blocks after padding are referred to as the *r* bit rate. The *r'* bit rate is an output of the internal state in each iteration. PHOTON uses an extended sponge function in which the input rate (*r*) may vary from the output rate of the output side (*r'*). If *r'* is greater than *r*, the time taken by the squeezing process will be low. Otherwise, the pre-image bound will be greater [65].

The internal state of a sponge construction is divided into two distinct parts: the r-bit rate (represented as *c* for SPONGENT and *r* for PHOTON) decides how fast the input message is processed and how fast the final digest is produced; and the c-bit capacity determines the security level [23]. PHOTON is based on AES for permutation computed in serialised way. AES is a block cipher processing data block-by-block with internal state of 128 bit. Unlike with PRESENT used by SPONGENT, AES is a traditional cipher suited for both software and hardware implementations. AES is responsible for calculating all round keys in PHOTON variants. For detailed functionality of PHOTON, please refer to the pseudo-code in Figure 4-3.

All five PHOTON variants are designed to offer reduced second-preimage security. Since both SPONGENT and PHOTON are sponge- based hash functions, the PHOTON variants used similar formulas as SPONGENT – for reduced second-preimage security which is: the collision resistance is $2^{c/2}$; preimage and second-preimage are 2^{n-r} and $2^{c/2}$ respectively.

Table 3-2 shows the PHOTON parameters. PHOTON has a fixed number of rounds of 12, unlike with SPONGENT, where SPONGENT number of rounds depends on the block size of each variant. The PHOTON security level lower bounds are determined by $min\{\ 2^{n/2},\ 2^{c/2}\}$ for collision resistance, $min\{\ 2^{n},\ 2^{c/2}\}$ for second-preimage and $min\{\ 2^{\min\{n,t\}}, \max\{\ 2^{\min\{n,t\}-r'},\ 2^{c/2}\}\}$ for preimage resistance where *t* is the internal state width $t = b = r + c$.

For SPONGENT the security level bounds for preimage resistance are determined by $min\{\ 2^{\min\{n,b\}}, \max\{\ 2^{\min\{n-r,c\}},\ 2^{c/2}\}\}$. The bounds for modified SPONGENT and PHOTON are given in Table 5-8 and Table 5-9.

Table 3-2: PHOTON parameters

	n (bit) = c (bit)	b (bit)	r (bit)	r' (bit)	Number of rounds
PHOTON-80-20-16	80	100	20	16	12
PHOTON-128-16-16	128	144	16	16	12
PHOTON-160-36-36	160	196	36	36	12
PHOTON-224-32-32	224	256	32	32	12
PHOTON-256-32-32	256	288	32	32	12

PHOTON variants are designed for small messages (output rate *r'* ranges from 16 to 36 bits) their width *b* values are more than the width *b* of SPONGENT variants with small input rate *r* (i.e. SPONGENT-088-80-008, SPONGENT-128-128-008, SPONGENT-160-160-016, SPONGENT-224-224-016, and SPONGENT-256-256-016). Hash functions designed for small messages are slow to process large data but use less memory [66].

3.3 Assumptions of the study

This study focuses on the performance of lightweight cryptographic hash functions. Therefore, it is assumed that these hash functions are secure and do not have any known security vulnerabilities.

3.4 Limitations of the study

“Virtually all research has its limitations” [67],[68]. The following are the identified limitations in this study:

- Due to budget limits, the experiment for data collection was done using M3 node from FIT IoT LAB (described in section 4.2) as it was available for free.
- The proposed lightweight hash functions are limited to the popular family versions based on sponge construction with publicly available reference implementation code.

3.5 Validation and Reliability

Validity and reliability are the measures of the quality of the quantitative study [69]. "Validity refers to the extent to which a test measures what we actually wish to measure" [24]. This study will use

various sources from the literature review to ensure the validity of the hash functions and comparative method.

Reliability refers to the ability of the study to be repeatable. It measures the stability and consistency of the study. Experiments will be used to increase the reliability of the study.

3.5.1 Validation of hash functions

Test vectors are a set of inputs that is used to test a system and verify the software. The SPONGENT reference code is written in C++, and in C for PHOTON. The following subsection gives a step-by-step description of how each hash function was verified.

- **SPONGENT**

SPONGENT reference implementation code was validated using the test vectors in the test environment.

The SPONGENT designers were emailed to provide us with test vectors and hash values. The provided test vectors were for SPONGENT-088-080-008, SPONGENT-128-128-008, SPONGENT-160-160-016, SPONGENT-224-224-016, and SPONGENT-256-256-016. The downloaded reference implementation code was compiled on the Linux command line using the *gcc* compiler. When the reference code was compiled, it used the original input message *"Sponge + Present = Spongent*". This gave us an output of test vectors, message in hexadecimal format, plain text, and the hash value for each variant, which were compared to the ones provided by the designers, and they verified the correct functioning of the reference code on the experimental platform.

- **PHOTON**

The downloaded reference code was compiled on Ubuntu command line using the *gcc* compiler. When compiling the code the original input message was *"The PHOTON Lightweight Hash Functions Family"*. The test vectors, input message, and hash value were printed as output on the command line.

The PHOTON designers uploaded the test vectors online[3] where they were downloaded together with the reference code in a text file. Additionally, the designers included the test vectors in their full version of their publication [70], so these were also compared to the results obtained when compiling the downloaded reference code and the test vectors in text files. The test vectors in a text file, in a published paper, and the results of the compiled reference code matched. The comparison of the test vectors from the above-mentioned sources verified that the underlying

[3] https://sites.google.com/site/photonhashfunction/downloads/testvectors.zip?attredirects=0&d=1

algorithm components were original as published by the designers of SPONGENT and PHOTON respectively.

Various software and hardware tools used for the performance evaluation in this study are summarised in Table 3-3.

Table 3-3: Summary of hardware and software tools

Evaluation Hardware		
Node	Raspberry Pi 3 Model B	
OS	Raspbian 10 Buster	
Kernel	5.10.17	
Software		
	SPONGENT	PHOTON
Implementation language	C++	C
Code version	1.0	20131205
Compiler	g++ 8.3.0	gcc 8.3.0

3.5.2 Validation of the comparison method

With a rapidly emerging lightweight cryptographic algorithms to protect IoT applications, it can be difficult for a researcher to choose one hash function over the other for their work. Comparative research tends to expand our knowledge about a particular work, in this case, about the best performing lightweight hash function. The comparative method is used to identify, analyse, and explain similarities and differences between the SPONGENT and PHOTON performance. FIT IoT LAB M3 node was used to collect the readings using the ambient light and pressure sensors built on the node. This node was accessed remotely where the data collection experiment was run for 30 minutes with the interval of a second (a default setting by the FIT IoT LAB admin) and the readings where streamed to a text file during the experiment. This text file was copied to a local Raspberry Pi 3 Model B which was used as an IoT node to execute the proposed hash functions. This file was used as an input message for SPONGENT and PHOTON. Only Raspberry Pi 3 Model B was used as an evaluation platform for evaluating the performance of the proposed hash functions. The number of runs were ten for SPONGENT and PHOTON, respectively, in order to average out possible variations that might occur in single readings and to provide a more accurate estimation of latency and memory consumption. The findings were presented in a manner to determine which variant best performs when the data log file size increases and when the input

parameter was modified. The results obtained in this study were also compared to the unmodified SPONGENT and PHOTON.

A number of studies [57], [71]–[76] compared implementation of lightweight algorithms in software and hardware. This suggests that a comparative approach is commonly used in the lightweight cryptography research field. The criteria used for comparison differ, as they depend on the research questions of a particular study. Some studies, like [65], compared six popular lightweight sponge-based hash functions using six different parameters, including security, area requirement, digest, throughput, power requirement, and cycle. In [56], they benchmarked not only hash functions but included symmetric block ciphers on the Raspberry Pi model 3 platform and compared their results with the Arduino benchmark results provided in the literature. It is evident that using a comparative method will assist in determining the best performing hash function between SPONGENT and PHOTON variants, ultimately answering a posed research question seeking to answer how the proposed lightweight cryptographic hash functions compare in terms of the identified measures. In this study, we only compared the performance of popular family versions of hash functions in terms of latency and memory consumption on one evaluation platform. As stated in Chapter 6 as future work, the work in this study can be extended to multiple platforms.

3.6 Summary

The purpose of this chapter was to outline the research methods that were followed to achieve the objectives and answer the research questions studied. In this chapter, the research design, method, and data collection to be followed to achieve the objectives of the study were discussed. The reference implementation codes obtained from the hash functions publications were verified using test vectors before they were used. An overview of the validation of the comparison method was presented. This gave evidence from the literature view that this study is following the right way to compare the hash functions. The next chapter presents the implementation of the proposed hash functions.

CHAPTER 4 RESEARCH IMPLEMENTATION

4.1 Introduction

This chapter will test and evaluate the performance of the identified lightweight hash functions used to maintain data integrity in IoT applications, which are SPONGENT and PHOTON. The description of the implementation environment is outlined, followed by a description of the data logging method used to collect monitoring data, and pseudo-code of the hash functions. Finally, the experimental procedure, setup, and parameters are discussed.

4.2 Test Environment

In this study, two environments are used for experiments: the M3 IoT node from FIT IoT LAB [77] and the Raspberry Pi 3 Model B. FIT IoT LAB is a large-scale, open-access multiuser testbed, with its architecture located in six sites across France, Europe. It offers a total number of 1786 nodes, with different functionalities and architecture. The FIT IoT LAB is accessible via a web portal or command-line tools (using SSH as the front-end). One of the advantages of using a testbed is that it can cut costs for researchers since the hardware infrastructure has been implemented, which ultimately reduces the labour resources needed to set up the environment. However, the testbed experimental results “are heavily affected by the testing environment that is often highly random and uncontrollable” [78].

FIT IoT LAB M3 node is used in this study for data collection of the readings as it has set of sensors to be taken advantage of. For experiments, the sensors used are ambient light (based on ISL29020) and atmospheric pressure (based on LPS331AP) sensors as shown in Figure 4-1.

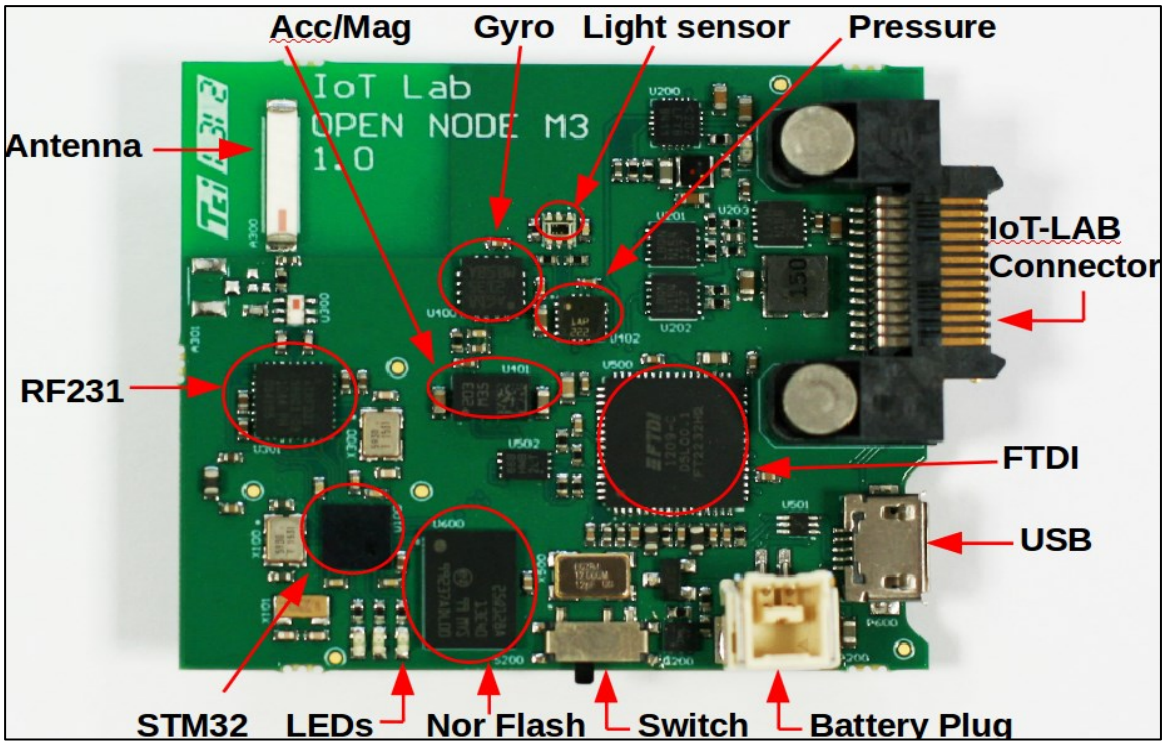

Figure 4-1: Schematic diagram of M3 IoT node [79]

The M3 IoT Node has a maximum microcontroller frequency of 72 MHz and a readily available Contiki-NG[4] Operating System (OS). The chosen node(s) is sequentially selected depending on the availability of the nodes in the testbed. The goal of the experiment is to collect data using sensors from the M3 node that will be used as input message for evaluating the identified hash functions. The process to configure Contiki-NG on the node was done on the testbed. The files required to compile Contiki-NG for the IoT LAB-M3 platform were available only on the testbed itself. The process of setting up the OS is provided by the lab administrator and is found on the FIT IoT LAB website[5]. The compiled code was stored in the SSH profile of the FIT IoTLAB author and was uploaded whenever the data collection experiment was created on the selected M3 node as a firmware.

The Raspberry Pi 3 Model B (hereinafter refered to as Raspberry Pi) was used to implement the SPONGENT and PHOTON hash functions. The Raspberry Pi had the Raspbian running as OS. It is a quad core ARM Cortex-A53 with 1.2 GHz 64-bit CPU and 1 GB RAM. Raspberry Pi has 40 extended General-Purpose Input/Output (GPIO) that can be used to connect sensors and other

[4] Contiki-NG is an open source, real time OS for resource-constrained devices in IoT with low memory footprint. See: https://github.com/iot-lab/iot-lab/wiki/Contiki-support
[5] https://www.iot-lab.info/legacy/tutorials/contiki-ng-compilation/index.html

external devices. However, due to financial constraints, the M3 node was used for data readings as it had freely available sensors.

As mentioned at the beginning of this study, the M3 node was identified to be used for data collection and evaluation of the SPONGENT and PHOTON as it has more limited resources than the Raspberry Pi 3 Model B.

To do this, we needed to merge the makefiles of the reference implementation codes of the hash functions and the Contiki-NG code loaded on M3 as firmware. The firmware alone was compiled following a tutorial provided by FIT IoT LAB administrators. The reference implementation code for SPONGENT is written in C++, for PHOTON it is C, and for Contiki-NG it is C++. To implement and execute the hash functions on the M3 node, we needed to merge the makefiles of the implementation code and the Contiki-NG, compile them together, and upload them on M3 when starting the data collection experiment. The code merge did not succeed as there was no official documentation guiding this process. Due to time constraints, we used the M3 node for data collection and the evaluation of the lightweight hash functions was done on a Raspberry Pi 3 Model B. The next section presents the testing methodology in the described testing environments.

4.3 Testing Methodology

4.3.1 Data collection

Data logging is the process of collecting and storing data over a period of time. The data log is accessible via a network card from the M3 node. The network card of a node is used to access the sensors using general purpose input/output (GPIO) pins from the M3 IoT Node. This is enabled by using the command line tools on the testbed Secure Shell (SSH) frontend; the command and the port number are as follows:

nc m3-<node network address> 20000

where the <*node network address*> is a variable with dynamic number that changes based on the node used and *20000* is a fixed port number.

When using the above-mentioned command, the three sensors of the M3 node are accessed. The temperature, ambient light, and pressure sensor readings can be obtained by choosing the corresponding prompt option (i.e., *–t*, *–l*, or *–p*) on the command line. The data from the sensors is then printed on the command line or written to a file, such as a text file. For multiple devices deployed in an experiment, a better and more efficient tool to use is a "*serial_aggregator"*. *serial_aggregator* is a Python-based script provided by the FIT IoT LAB to automate data logging

of more than one node. This tool prevents the user from having to use multiple terminals for each node deployed and connected. The difference in the data logging using these two tools is that *serial_aggregator* appends a timestamp and a node identifier to each deployed node of the experiment and makes it easy to pipe the data to a file.

The ambient light and atmospheric pressure sensors readings of the M3 node were collected using *serial_aggregator* for the duration of the experiment in the lab and the data was written to a text file. This text file was then copied to the Raspberry Pi outside of the lab using the Secure Copy Protocol (SCP) command to be used locally. The local text file results (hereinafter referred as data log file) in the Raspberry Pi was passed as a message input to be hashed by the SPONGENT and PHOTON hash functions respectively as explained in the experimental procedure in section 4.3.4.

4.3.2 Pseudo-code

Before the identified hash functions can be used, they must be understood. This is accomplished through the use of pseudo-code, which is a simplified representation of the code. Pseudo-code is a description of how an algorithm (hash function) should work that is independent of a programming language or programming knowledge. Furthermore, it is intended to be easily understood by a human rather than a machine. It provides a high-level overview of an algorithm.

The reference implementation code for SPONGENT and PHOTON was analysed by the author to write the pseudo-code to describe how the hash functions work in non-technical language. For both hash functions, the pseudo-codes for the original reference implementation code and the modified code are shown in Figure 4-2 and Figure 4-3. In the modified code version (shown in blue), we added the timestamp function used for latency measurements and changed the capacity/input rate as described in section 4.3.4.2. Figure 4-2 illustrates the pseudo-code for SPONGENT original code and modified code, respectively.

```
Algorithm: Modified SPONGENT
Input: message as a string
Output: hashvalue

/* Pseudo-code for SPONGENT with input message read from log file and
getTime() to append the timestamps. */

Function Init(state, hashVal):
    Check the hash size if it is any of 88, 128, 160, 224, 256
    Set the first nSBox bytes of the memory pointed by current value of
state to 0
    Initialize the number of bits hashed so far to 0
    // hashsize/8 is defined as rate/8
    Set the first byte hashsize/8 of the memory pointed by current hashvalue
of state to 0
return SUCCESS

Function Absorb(*state):
 Declare int counter
 for counter is 0 to rate in bytes do
    Assign the current state value at index counter to the results of XORed
states message block at index counter
    Permute(state)
return SUCCESS

Function Permute(*state):
    Check the version of the state output
    Add counter values
    Apply sBoxLayer layer
    Apply pLayer

Function Squeeze(*state):
    Copy the current state value from the pointed location to state's
message block directly to the memory
    Permute (state)
return SUCCESS

Function Pad(*state):
    Declare integer byte index and assign it remaining byte of data length
    Declare integer bit position and assign it bit position in the last byte
    if bit position is not null then
        Make the unoccupied bits of the messageblock 0
    if bit position is not null then
        Add single 1-bit to the current messageblock
    else
        Assign 0x80 to current messageblock
    while byteIndex is not equals to R_SizeInBytes do
        // RateSizeInBytes is the rate in bytes
        Increment byteIndex
        Add 0-bits to the current state messageblock
    Permute(state)
return SUCCESS

Function getTime():
    Import class template std::chrono::duration to get time interval
    Get the duration in nanoseconds
 return Duration

Function SpongentHash(*data, databitlen,*hashval):
    Declare int start_time and assign it the returned value of getTime()
    Declare state variable
```

```
    Declare results variable
    Assign the returned value of Init(&state, hashval) to varible results
    if results is not SUCCESS then
        return results
    //Absorb available message blocks
    while databitlen is greater or equals to the rate do
        Copy the parameter data to the current state's message block memory
        Absorb(&state)
        Assign the difference of databitlen and the rate to databitlen
        Assign the sum of parameter data and R_SizeInBytes to the parameter
data
    if databitlen is greater than 0 then
        // Pad the remaining bits before absorbing
        Copy the data parameter to the first byte of current state's
messasage block memory
        Assign the databitlen to the remaining bits variable
    else if databitlen is equal to 0 then
        Covert 0 to unsinged char and fill it into each of the first
R_SizeInBytes of the current messageblock
        Assign the databitlen to the remaining bits variable
    Pad(&state)
    Absorb(&state)
    Assign the sum of bits of the state hashed so far and the rate to the
bits of the state hashed so far
    // Squeeze data blocks
    while the bits of the state hashed so far is less than the hash size do
        Squeeze(&state) Copy the state's messageblock from the
            pointed location to the parameter hashval directly to the
            memory
        Assign the sum of the bits of the state hashed so far and the rate
            to the bits of the state hashed so far
        Assign the sum of the hashval and the R_SizeInBytes to the hashval
    Copy the current state value from the pointed location to hashval
directly to the memory
    Assign the sum of the hashval and the R_SizeInBytes to the  hashval
    Declare int end_time and assign it the returned value of getTime()
    Subtract start_time from end_time to get the time taken to execute this
hash method
return SUCCESS

Function readInputMessage():
    Declare counter int
    Declare databitlen and assign it 216
    Declare hashval[hashsize/8] and initialize it to 0
    // ---------------Start logfile message----------
    Determine the file size and declare it
    Open the file stream buffer
    Declare string variable file_contents to store all the lines read
    Declare string variable text_read to iterate through the multiple file
lines
    while file is readable and can get the file lines do
        Check if the last character is not an end of file
        Read the file lines and assign text_read to file_contents
    Declare int variable file_content_len and initialize it to 0
    Get the size of the file_contents and assign it to file_content_len
    Declare char variable message with array length of file size
    Copy up to file_size characters from the string pointed to by
file_contents to message
    // ---------------End logfile message------------
    Print the message as a string
    Print the message in hex format
```

```
    for counter is 0 and less than databitlen divided by 8 do
        Print the message at index counter
    SpongentHash(message, databitlen, hashval)
    for counter is 0 and less than hash size divided by 8 do
        Format the hashval at index counter to atleast 2 hex digit precision
and print it

 Function Main:
    readInputMessage();
 return 0
```

Figure 4-2: Pseudo-code for SPONGENT with modifications in blue

The original SPONGENT reference implementation code was using the static message: *SPONGE + PRESENT = SPONGENT*. This message was used to verify the algorithm using test vectors as discussed in Chapter 3. To be able to read the log file containing the sensor readings, the original reference implementation code was modified. Figure 4-3 illustrates the pseudo-code for PHOTON original code and modified code.

The original PHOTON reference implementation code was using the static message: *The PHOTON Lightweight Hash Functions Family*. This message was used to validate the algorithm using test vectors, as discussed in Chapter 3.

The SPONGENT and PHOTON codes were modified to pass the data log file containing the sensor readings as an input message. The modification did not alter the core operations of the hash function; it simply included the opening of a data log file, reading all the lines, and storing that in a locally declared variable. This variable was assigned to the message. During the code implementation (modification), the original fixed block length had to be changed to accommodate the length of the new input message (contents of the log file). Since the data file had multiple lines, a local variable in the code was declared and initialized to an array with the array length the same as the log file size explained in the experiment procedure in section 4.3.4.1 and section 4.3.4.2. This variable was then assigned to an input message variable used in the hash method; converting the multiple lines into a single line file.

```
Algorithm: Modified PHOTON
Input: message as a string
Output: hashvalue

/* Pseudo-code for PHOTON with input message read from log file and
getTime() to append the timestamps. */
//Global variables
Define S 8 // symbolic constant
Define S 4 // symbolic constant
// Note: The Round is 12
Declare int DEBUG variable and initialize it to 0
Declare unsigned long long MessBitLen variable and initialize it to 0

Function PrintState(state):
    if !DEBUG then
        return
    Declare counter i
    Declare counter j
    // is [D][D] not supposed to be refered as first row first column
    for counter i is 0 and less than first column index D do
        for counter j is 0 and less than second column index D do
            Format the state at counter i and counter j to atleast 2 hex
digit precision and print it

Function printDigest(byte* digest):
    Declare counter i
    for counter i is 0 and less than DIGESTSIZE/8 do
        Format the digest at counter i to atleast 2 hex digit precision and
print it

Function Permutation(state[D][D], ROUND):
    // state is a d x d matrix
    Declare int counter
    for counter is 0, and less than ROUND do
        if DEBUG then
            Print counter
        /* The internal state is represented as a (d x d) matrix of s-bit
cells and each round is defined
            as  the application of 4 operations (AddKey, SubCell, ShiftRow
and MixColumn): */
        /* AddKey() simply consists of adding fixed values to the cells of
the internal state.
            The internal state is bitwise XORed with a round-dependent
constant (RC) (generated with an LFSR);
            The two constants (RC and ROUND) are both XORed with the first
column of the (d x d) internal state.
            In this operation, only the first column is permuted while other
columns are left unchanged.
        */
        AddKey(state, counter)
        PrintState(state)

        /* This is a Cell substitution operation. The S-box is applied to
each s-bit nibble of the internal
        state (i.e the PRESENT S-box if s= 4,the AES S-box if s= 8) where s
is the size of the cell.
        Each cell of the state is replaced by a corresponding cell  from the
nonlinear S-Boxes
        */
        SubCell (state)
        PrintState(state)
```

```
        // Nibble row i of the internal state is cyclically shifted by i
positions to the left
        ShiftRow (state)
        PrintState(state)
        /* The columns of the internal state are independently multiplied
with the predefined matrix based on the
        dimension size d. Linearly mixes all the columns
        */
        independently.
        MixColumn(state)
        PrintState(state)

Function WordXorByte(state[D][D],const byte*str, int BitOffSet, int
WordOffSet, int NoOfBits):
    Declare counter i and assign it to 0
    while counter i is less than NoOfBits do
        state[(WordOffSet+(i/S))/D][(WordOffSet+(i/S))%D] ^= GetByte(str,
BitOffSet+i, min(S, NoOfBits-i)) << (S-min(S,NoOfBits-i))
        Assign the sum of counter i and S to counter i

Function Init(state[D][D]):
    Declare counter i
    Declare counter j
    Assign the MessBitLen 0
    for counter i is 0 and less than row at index D do
        for counter j is 0 and less than column at index D do
            Assign the state[counter i][counter j] to 0
    Declare byte presets variable with length of 3
    Assign presets at first index the result of shifting DIGESTSIZE to the
right by two bits & constant value 0xFF
    Assign presets at second index the result of RATE & 0xFF
    Assign presets at third index the result of RATEP & 0xFF
    WordXorByte(state, presets, 0, D*D-24/S, 24)

Function CompressFunction(state[D][D], const byte* mess, int BitOffSet):
    WordXorByte(state, mess, BitOffSet, 0, RATE)
    Permutation(state, ROUND)

Function Squeeze(state[D][D], byte* digest):
    Declare int counter i and initialize it to 0
    while 1 do
        WordToByte(state, digest, i, min(RATEP, DIGESTSIZE-i))
        Assign the sum of counter i and RATEP to counter i
        if counter i is greater or equals to DIGESTSIZE then
            break
        Permutation(state, ROUND)

Function hash(byte* digest,const byte* mess, int BitLen):
    Declare int start_time and assign it the returned value of getTime()
    Declare 2 dimensional state array
    Round up RATE in bytes to return the smallest integral value and add
constant value 1 to the rounded byte
    Declare array padded with the length same as the returned above results
    Init(state)
    Declare int MessIndex and initialize it to 0
    while MessIndex is less or equals to the difference of BitLen and the
RATE do
        CompressFunction(state, mess, MessIndex)
        Assign the sum of MessIndex and RATE to MessIndex
    Declare int counter i
    Declare int counter j
```

```
    for counter i is 0 and is less than sum of rounded up RATE in bytes and
1 do
        Assign the padded at index counter i 0
    Subtract MessIndex from BitLen and convert it to a byte by dividing by
8.0. Round up the results to
    return the smallest integral value. Then assign the results to counter j
    for counter i is 0 and counter i is less than counter j do
        Convert MessIndex to bytes by dividing it by 8.
        Then add counter i to the MessIndex in bytes.
        This is a new index to be  used to access the mess parameter.
        Assign the results (i.e. mess[messageIndex in bytes+1]) to array
padded at index counter i
    Assign hexadecimal 0x80 to array padded at index counter i
    CompressFunction(state, padded, MessIndex&0x7)
    Squeeze(state, digest)
    Declare int end_time and assign it the returned value of getTime()
    Subtract start_time from end_time to get the time taken to execute hash
method

Function getTime():
    Declare a Structure timeval holding an interval broken down into
milliseconds
    Declare int time_now
    Get the structure value in milliseconds and assign the value to time_now
return time_now

Function readInputMessage():
    Initialize DEBUG to 0
    Declare byte digest[DIGESTSIZE/8]
    // ---------------Start logfile message----------
    Declare a file pointer
    Determine the file_size and declare it
    Declare variable file_contents[file_size] to store the lines read
    Declare int length_per_line and assign it 0
    Open the log file
    while the file character is not an end of file and length_per_line is
less than file_size do
        Read the file characters and assign them to
file_contents[length_per_line]
    Reset the file_contents[length_per_line] = 0
    Declare char message and assign the file_contents to it
    // ---------------End logfile message------------
    hash(digest, (byte*) mess, 8*strlen(mess))
    printDigest(digest)

Function Main:
    readInputMessage()
return 0
```

Figure 4-3: Pseudo-code for PHOTON with modifications in blue

4.3.3 Performance measures

Due to the nature of an IoT application that has restricted resource capabilities and one of its core functions is to collect a multitude of data, the resources of such applications should be utilised properly. To reiterate, the security of the hash function is out of scope, therefore, the author assumed that the proposed hash functions are secure and do not have any security vulnerabilities

or would not introduce any vulnerabilities. The performance measures considered in the performance of the hash functions are latency and memory; and they are defined as follows:

Latency – defined as the time (in ms) taken to hash the data log file and print the hash digest as an output. Low latency is preferred as this means that the algorithm is operating efficiently. To get the latency, we need timestamps at the beginning of the hash method referred to as *start_time* and end referred to as *end_time* (just before the return which prints the hash output) of the hash function. Subtracting the *start_time* from *end_time* gives you the latency of the hashing function.

Memory – The resource-constrained devices have limited RAM. The Raspberry Pi 3 has 1GB of RAM as mentioned in section 4.2. The usage of RAM for an algorithm needs to be less for a resource-constrained device to ensure that devices with constrained memory will be able to execute the proposed hash functions. Linux built-in command */usr/bin/time* is used to measure memory consumption when the program runs. This command gives the maximum resident set size; which is the amount of memory the process (i.e. the hash function program) utilised at its peak in kilo bytes (kB). To run this, the full command to use in the Linux terminal is */usr/bin/time –v ./ExecutableApplicationName*.

4.3.4 Experiment procedure

The initial step is to collect data from the sensor readings. This was done by the data log scripts *serial_aggregator* offered by FIT IoT LAB. The sensor readings are ambient light and pressure. The data collection experiment was done in the FIT IoT LAB as explained in Section 4.3.1, and it ran for 30 minutes with a one-second interval. The 30-minute log file contained raw data like timestamps, node id, ambient light and pressure values. To increase the file data readings, a new experiment was started with the required longer duration. This will allow us to have different variable length messages.

The following subsequent sections outlines the experiment procedures for the two test cases evaluated in this study.

4.3.4.1 Test case 1: The effect of the log size on the latency and program memory consumption

This test scenario examines the effect of data log file size on latency and memory consumption of PHOTON and SPONGENT variants. The test case used log sizes ranging from 100 kB to 1 MB, separated into ten data points. The actual file sizes utilised in the experiments were 100.3 kB, 200.5 kB, 300.8 kB, 400.7 kB, 500.6 kB, 600.6 kB, 700.6 kB, 800.6 kB, 900.6 kB, and 1.1 MB; these files were produced based on the single file imported from FIT IoT LAB into the Raspberry Pi as mentioned in section 4.3.1. A Python script was created to read the local data log file and

recursively copy and paste its content until the specified log size is reached for all ten sizes. During this stage the uniqueness of the contents of the newly generated data log files was not a priority as the focus was on getting the correct file size. Each data log was then passed as a message to be hashed by the SPONGENT and PHOTON hash functions.

4.3.4.2 Test case 2: The effect of the input rate on the latency and program memory consumption

The original input rate of the reference implementation code was modified to evaluate the effect of one of the variants' parameters known as the input rate. This parameter is called the capacity *c* for SPONGENT variants, which is defined as the remaining bits after the input message has been absorbed, while it is called the input rate for PHOTON. However, capacity *c* and input rate are both a second parameter in the formats SPONGENT-n/*c*/r and PHOTON-n/*r*/r' and they represent the rate of the message blocks that the hash function is processing. To accommodate the different names of the two hash functions, this parameter will be denoted as capacity/input rate (symbol: *cr*) in this test case. The sizes used were 128, 160, and 176 bits. It is vital to note that modifying the capacity/input rate did not alter the hash functions' underlying structure, like permutations and bitwise operations. However, the second preimage and collision security levels are affected.

In the test cases described above, the SPONGENT code was compiled using *g++* and PHOTON using *gcc* compilers to obtain the executable application, also known as binary file, as an output. Research implementation answers the following questions when comparing the performance of SPONGENT and PHOTON in resource-constrained devices such as the M3 node and Raspberry Pi:

- How long does the hash function take to compute hash values given a batch of collected data?
- How does a different data log file size affect the performance, specifically the latency and memory?
- Is there a correlation between latency and program memory?
- How much memory space is utilised by the hash function, in terms of the node's RAM?

4.4 Summary

This chapter discussed the implementation environment used to evaluate the proposed hash functions, SPONGENT and PHOTON. The data collection process was described, as well as the pseudo-code of the proposed lightweight hash functions. The latency and memory consumption of the program were identified as the performance measures used to evaluate and compare the

performance of the best performing lightweight hash function. The next chapter will present and discuss the performance evaluation results of SPONGENT and PHOTON.

CHAPTER 5 DATA ANALYSIS AND DISCUSSIONS

5.1 Introduction

This chapter presents graphical and statistical results of the latency and memory consumption performance; the discussion of the data analysis and its findings. The chapter first gives an overview of the data analysis protocol, reiteration of the data collection process, and finally the results of the effect of log size (test case 1) and the effect of capacity/input rate bit length on the performance measurements (test case 2).

5.2 Data Analysis Protocol

Chapter 3 explained the rationale and purpose of using quantitative research design. In this study, the experimental method was applied to acquire the performance measurements of the lightweight hash functions. The captured data from the methodology explained in detail in Chapter 3 are presented, analysed, described, and interpreted in a systematic manner.

Data analysis is a process of systematically applying statistical and logical techniques to describe and evaluate data. The analysis process aims to identify trends and relations according to the research objectives.

5.2.1 Data collection process

As mentioned in the previous Chapter, the data log file was produced in the FIT IoT LAB, where the experiment to collect the light and atmospheric pressure readings was conducted. During this experiment the M3 node was used to record the light and atmospheric pressure readings. The "*serial_aggregator*" script was used to write the output (sensor readings) to a text file for the duration of the experiment in the lab. This text file was then copied to the Raspberry Pi outside the lab using the Secure Copy Protocol (SCP) command to be used locally. The local text file results (hereinafter referred as data log file) in the Raspberry Pi was passed as a message input to be hashed by the SPONGENT and PHOTON respectively.

5.2.2 Performance measurements

The path and filename of the log file were included in SPONGENT and PHOTON. For each algorithm, the log file was opened, read, and processed as a message to be hashed. The input rate parameter for SPONGENT variants is the capacity *c,* which depends on the security requirement needed for the applications as explained in Chapter 3, while for PHOTON variants

the input rate is *r* and is equal to *n* for each variant. The parameter comparison of SPONGENT and PHOTON variants are shown in Table 3-1 and Table 3-2, respectively.

5.2.3 Compiling the code

A Make utility tool was used to automate the setup rules to compile source code files, generate and execute the executable binary files for SPONGENT variants, as shown in Figure 5-1.

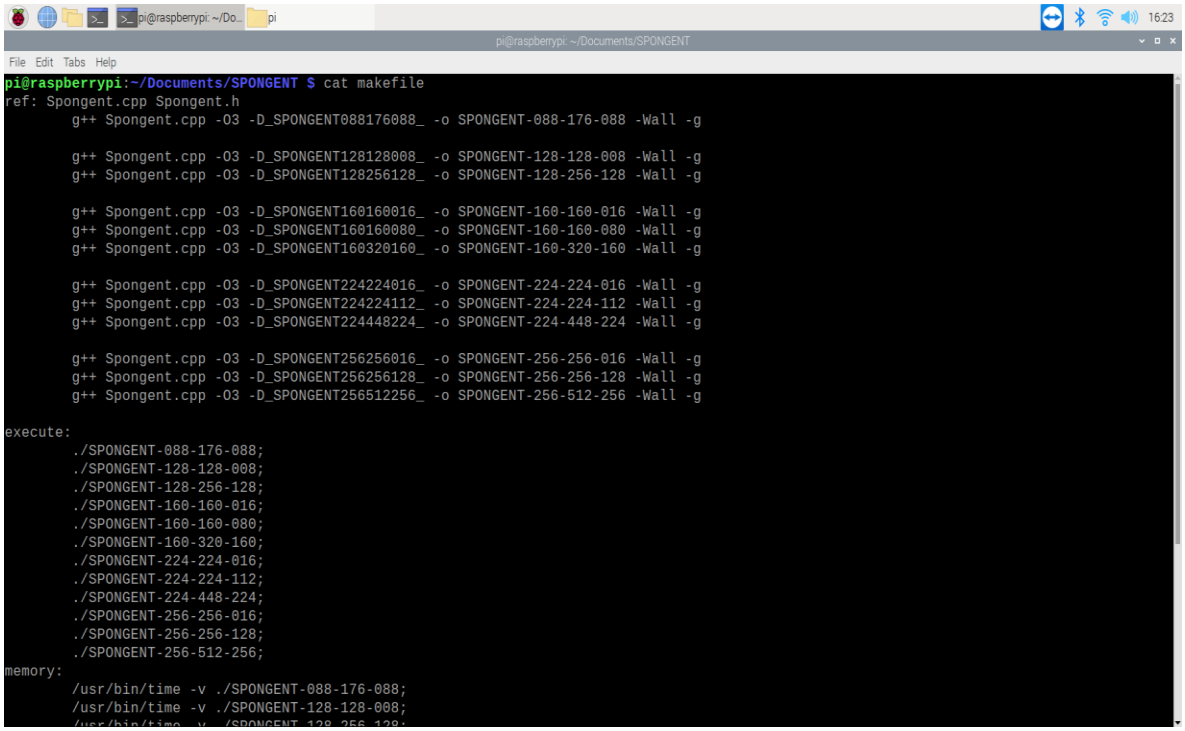

Figure 5-1: Illustration of the code compilation make rules

This applies for PHOTON, except that it is using *gcc* to compile the code. PHOTON had its own *makefile* to generate the binary files and execute them for results capturing.

5.2.4 Capturing the results

To obtain the latency and memory size consumption, the executable binary files of SPONGENT were executed. Figure 5-2 shows an example of the results output for latency and memory consumption when the SPONGENT *makefile* was executed. For each SPONGENT and PHOTON variant, the latency and maximum memory size used were recorded.

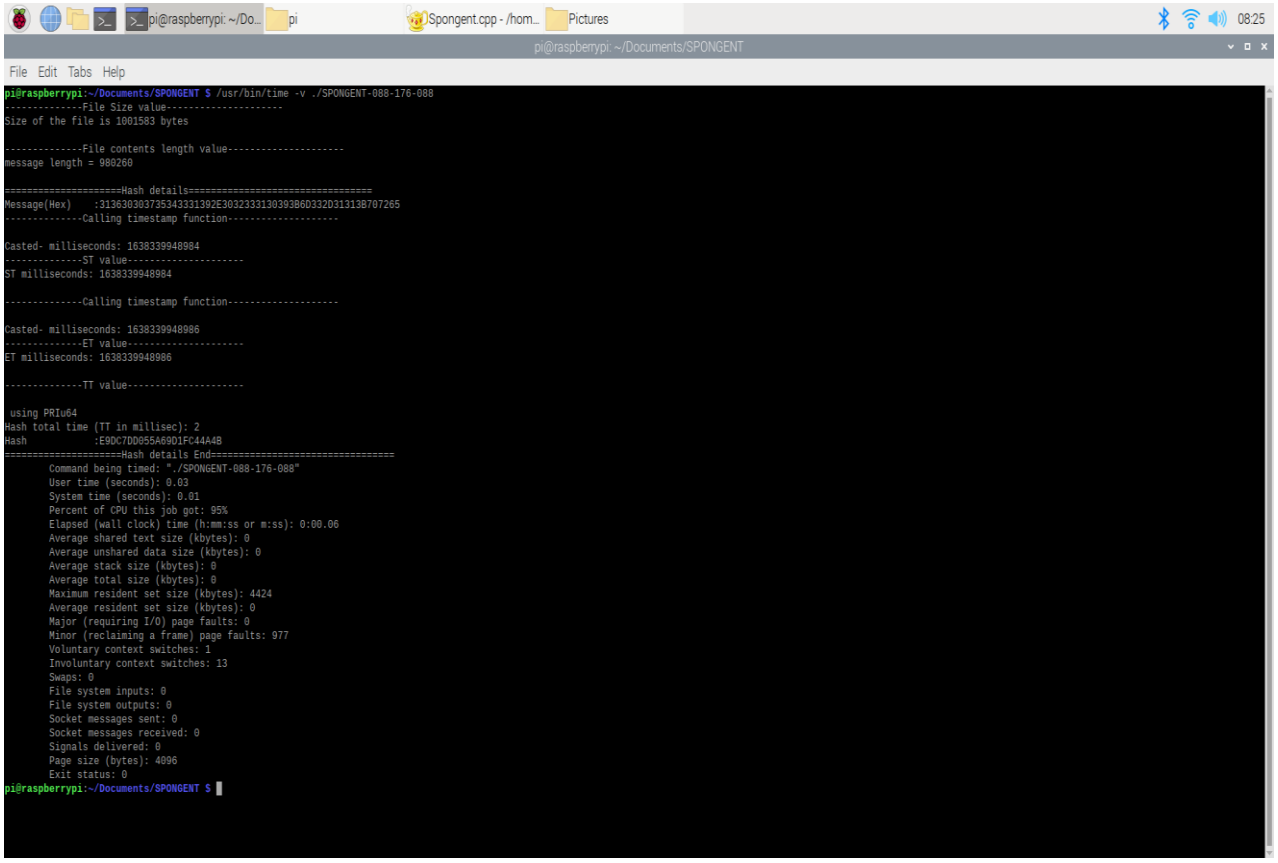

Figure 5-2: Screenshot of the computer terminal printing the results

5.3 Results and evaluation

The experiments were conducted in two test cases. Test case 1: analyses and evaluates the performance when the data log file used as input to the hash increases in size. Test case 2 studies how adapting the capacity/input rate affects performance.

5.3.1 Test case 1: The effect of the log size on the latency and program memory consumption

In this test case, the latency and memory consumption performance of the PHOTON and SPONGENT variants are evaluated using different data log file sizes. The log sizes ranged from 100 kB to 1 MB and were separated into ten data log files. The actual file sizes were 100.3 kB, 200.5 kB, 300.8 kB, 400.7 kB, 500.6 kB, 600.6 kB, 700.6 kB, 800.6 kB, 900.6 kB, and 1 MB. During discussions, file sizes are referred to as whole numbers for ease of reference. The goal of this test case was to evaluate how the size of the data log file affected latency and program memory performance. For PHOTON and SPONGENT, respectively, each data log file was run ten times (number of runs); this made a total of 50 samples for five PHOTON variants and 130

for thirteen SPONGENT variants per data log file. The average results were calculated and used for the graphical and statistical results for PHOTON and SPONGENT. The next section presents the results of test case 1, starting with PHOTON.

Figure 5-3 shows the latency of the five PHOTON variants when they were hashing ten different data log files. PHOTON-80-20-16 is the fastest variant to across all data log file sizes, followed by PHOTON-160-36-36, while PHOTON-224-32-32 has the highest latency.

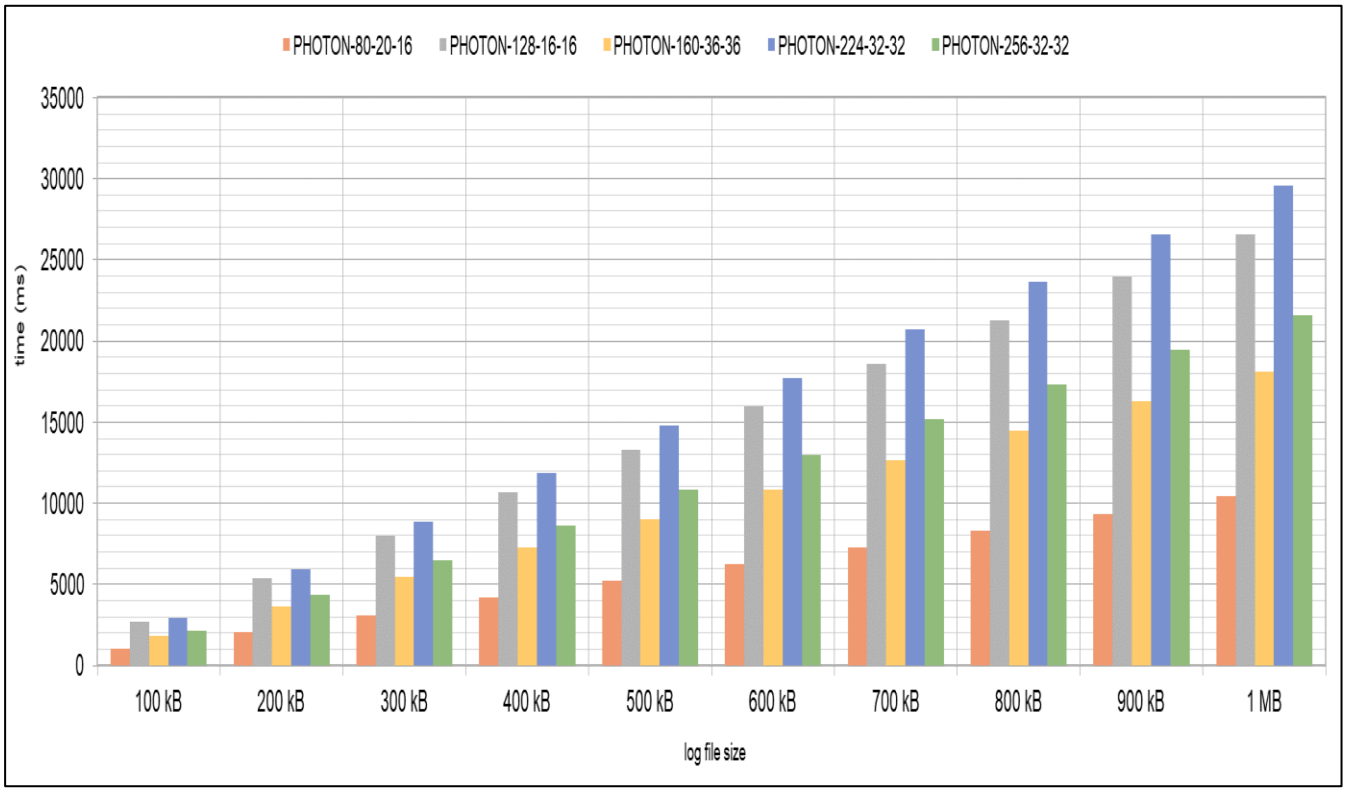

Figure 5-3: The effect of log file size on the average latency of PHOTON

Figure 5-4 shows the effect of log size on the memory consumption of PHOTON. PHOTON-160-36-36 which is the second best variant in Figure 5-3 consumes the least overall memory, while PHOTON-128-16-16 is the second best variant in terms of memory performance. It can be seen that the program memory usage increases as the data log file increases. The program utilised a slightly higher memory when hashing the 200 kB rather than 100 kB for the smallest variant PHOTON-80-20-16. It can be concluded that PHOTON-80-20-16 is best suited for low latency applications. The memory consumption is the same within limits for all variants per file size.

	100 kB	200 kB	300 kB	400 kB	500 kB	600 kB	700 kB	800 kB	900 kB	1 MB
PHOTON-80-20-16	1527.6	1631.6	1721.6	1832.4	1939.2	2003.6	2136.8	2218	2322.8	2411.2
PHOTON-128-16-16	1536.8	1617.2	1731.2	1853.6	1928	2019.2	2102.8	2206.4	2305.6	2426.4
PHOTON-160-36-36	1526.4	1644.4	1740.8	1815.6	1935.6	2036.4	2114.4	2220.8	2320.8	2415.2
PHOTON-224-32-32	1545.2	1629.2	1730.8	1849.6	1923.6	2007.6	2128	2216	2328.8	2414.4
PHOTON-256-32-32	1549.2	1636	1732.4	1814.4	1934.4	2026	2118.8	2212	2315.6	2418

Figure 5-4: The effect of log size on the average memory consumption of PHOTON

Compared to PHOTON, the following section presents the results of SPONGENT's test case 1. As it can be seen in Figure 5-5, the SPONGENT variants have low overall latency compared to PHOTON (see Figure 5-3). The difference between Figure 5-3 and Figure 5-5 could be caused by the different underlying structure and design of the hash functions such as the permutation method of each hash function.

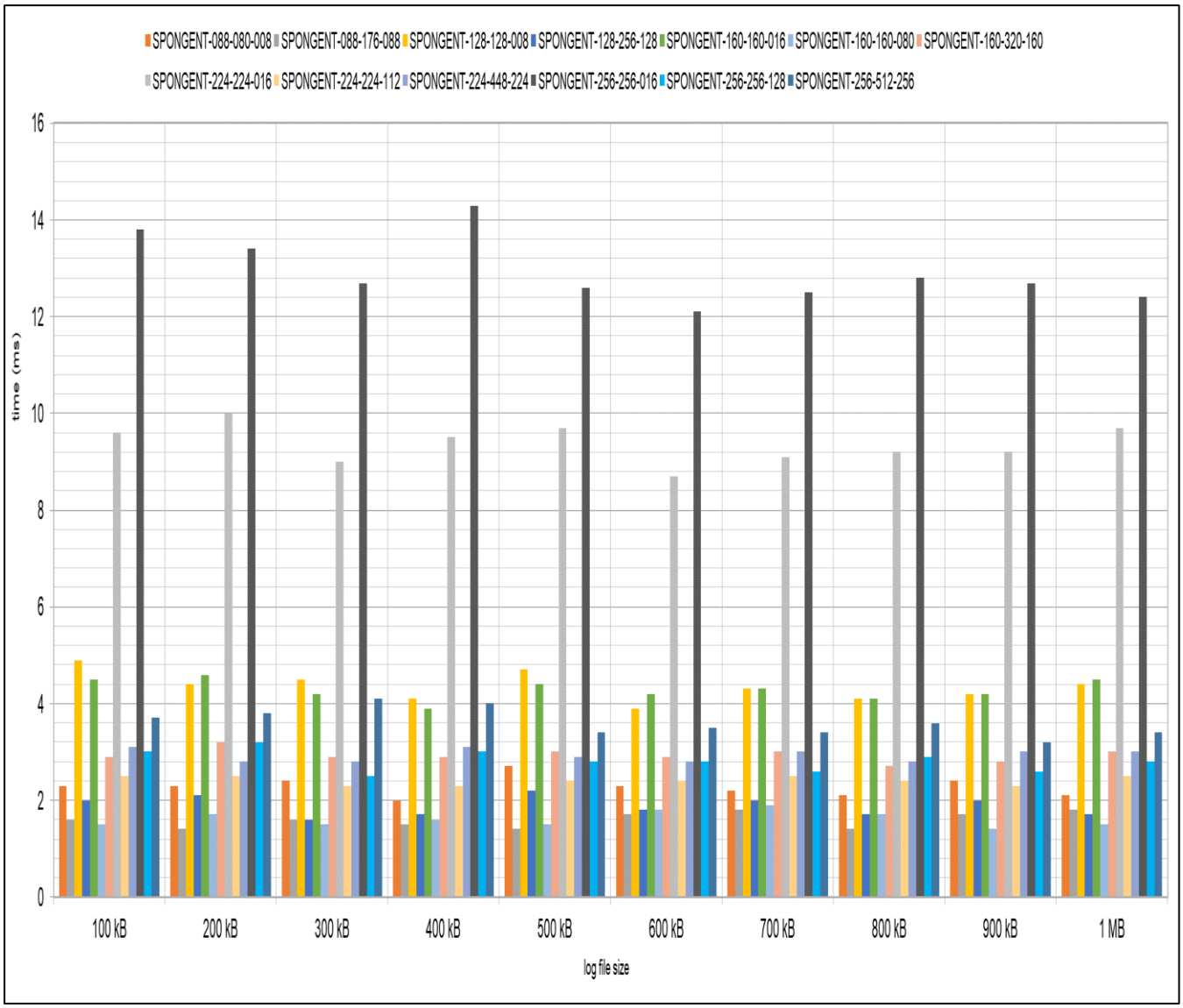

Figure 5-5: The effect of log size on the average latency of SPONGENT

The descriptive statistical analysis of the latency measurements for SPONGENT in Figure 5-5 is presented in section 5.3.1.1.2.

SPONGENT is using PRESENT-type permutation (permutations are fixed key), while PHOTON is using AES-like permutation as unkeyed permutation. All PHOTON variants have 12 rounds in each iteration, while for SPONGENT the number of rounds depends on the block size of the internal state.

SPONGENT-088-176-088 and SPONGENT-160-160-080 have the lowest latency across the range of file size. SPONGENT-256-256-016 has the highest overall latency, as shown in Figure 5-5. From each SPONGENT-n group, the variants with low output rate, i.e. SPONGENT-088-080-

008, SPONGENT-128-128-008, SPONGENT-160-160-016, SPONGENT-224-224-016 and SPONGENT-256-256-016 have higher latency than other variants in the same SPONGENT-n group. It is interesting to observe the significant different latency changes for the SPONGENT-160 group; where SPONGENT-256-256-128 decreased by over 9 ms compared to SPONGENT-256-256-016 which has the highest latency with a 400 kB data log file which had the largest decrease of about 11.30 ms.

Figure 5-6 shows that the maximum memory consumption increases as the data log file size increases. The larger the data log file the more memory it requires as it might be taking longer to complete the hash generation process.

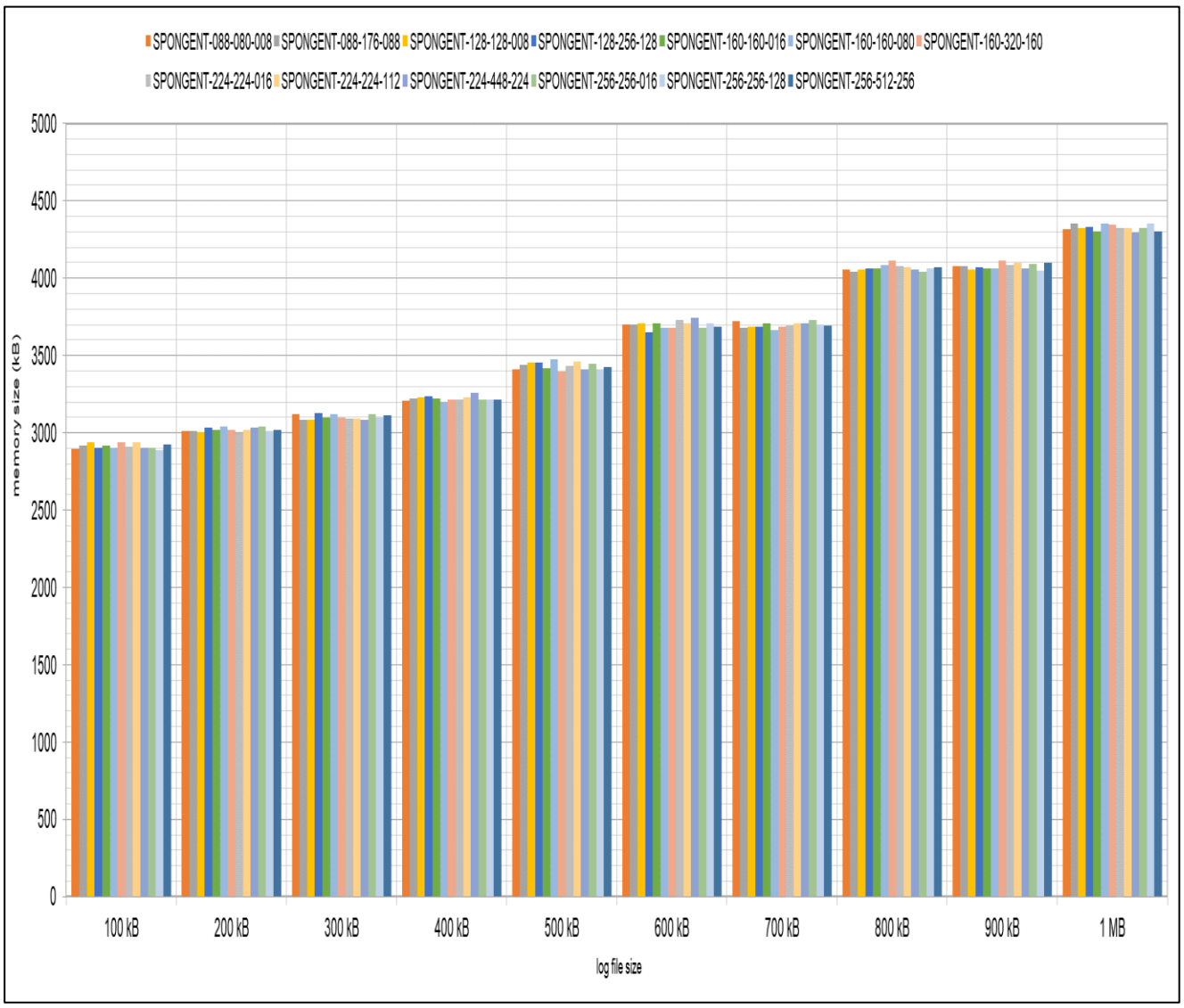

Figure 5-6: The effect of log size on the memory consumption of SPONGENT

Table 5-1 shows the security levels for the original SPONGENT and PHOTON variants' matching parameters in Table 3-1 and Table 3-2, respectively. Since PHOTON was designed for small messages, for fair comparison only SPONGENT variants with low output rates are compared to PHOTON variants in Table 5-1. Table 5-1 highlights the preimage (as pre), second-preimage (as 2nd pre), and collision resistance (as col) security levels for SPONGENT and PHOTON using the unmodified original parameters. PHOTON has a high internal state width *b*, and the hash output *n* bits for both SPONGENT and PHOTON are the same, with the exception of SPONGENT-088-080-008 and PHOTON-80-20-16, which are 88 and 80 bits, respectively. SPONGENT variants input rate bits, denoted as *c*, are higher than PHOTON variants (represented as *r*).

Table 5-1: Security levels of SPONGENT and PHOTON variants with small output rate

	n (bit)	b (bit)	c (bit)	r (bit)	Number of rounds	pre	2nd pre	col
SPONGENT-088-080-008	88	88	80	8	45	80	40	40
SPONGENT-128-128-008	128	136	128	8	70	120	64	64
SPONGENT-160-160-016	160	176	160	16	90	144	80	80
SPONGENT-224-224-016	224	240	224	16	120	208	112	112
SPONGENT-256-256-016	256	272	256	16	140	240	128	128
	n = c (bit)	b (bit)	r (bit)	r'(bit)	Number of rounds	pre (n - r')	2nd pre (c/2)	col (c/2)
PHOTON-80-20-16	80	100	20	16	12	64	40	40
PHOTON-128-16-16	128	144	16	16	12	112	64	64
PHOTON-160-36-36	160	196	36	36	12	124	80	80
PHOTON-224-32-32	224	256	32	32	12	192	112	112
PHOTON-256-32-32	256	288	32	32	12	224	128	128

5.3.1.1 Descriptive analysis

Descriptive statistics were used to describe the measured results for test case 1. Descriptive statistics is imperative for quantitative data analysis and simply describes what is or what the data show. The data dispersion was analysed to find the spread of the values around the central tendency. The mean, standard deviation, and confidence intervals of the measured results

presented above were calculated to confirm the trends visible in the graphs presented. Descriptive analysis was performed per row for each variant for both PHOTON and SPONGENT.

5.3.1.1.1 PHOTON analysis

From Table 5-2, PHOTON-80-20-16 has the lowest standard deviation of 3146.50 ms, while PHOTON-224-36-36 has a high standard deviation of 8947.20 ms. The confidence intervals were calculated for a 95% confidence level. PHOTON-128-16-16 and PHOTON-224-32-32 have higher mean latencies. The length of the hash value, the input and output rates for PHOTON-256-32-32 are two times that of PHOTON-128-16-16, which makes PHOTON-256-32-32 have less mean latency than PHOTON-128-16-16. The increased input and output rates allow for faster data log file processing.

Table 5-2: Descriptive statistics for the latency of PHOTON variants using 10 log file sizes

	PHOTON-80-20-16	**PHOTON-128-16-16**	**PHOTON-160-36-36**	**PHOTON-224-32-32**	**PHOTON-256-32-32**
Mean latency (ms)	5727.63	14637.97	9953.21	16274.53	11894.58
Standard Error	995.01	2541.32	1728.58	2829.35	2067.71
Standard Deviation	3146.50	8036.37	5466.24	8947.20	6538.68
Confidence Level (95.0%)	2250.87	5748.87	3910.31	6400.44	4677.49
CI Upper bound	7978.50	20386.84	13863.52	22674.97	16572.07
CI Lower bound	3476.76	8889.10	6042.90	9874.09	7217.09

The statistics of the memory consumption data set for the PHOTON variants are indicated in Table 5-3. PHOTON-256-32-32 has the lowest standard deviation of 293.30 kB, while the largest deviation of 298.37 kB was observed by PHOTON-80-20-16.

Table 5-3: Data statistics for PHOTON memory consumption using 10 log file sizes

	PHOTON-80-20-16	PHOTON-128-16-16	PHOTON-160-36-36	PHOTON-224-32-32	PHOTON-256-32-32
Mean memory (kB)	1974.48	1972.72	1977.04	1977.32	1975.68
Standard Error	94.35	93.17	93.65	93.28	92.75
Standard Deviation	298.37	294.63	296.14	294.97	293.30
Confidence Level (95.0%)	213.44	210.77	211.84	211.01	209.81
CI Upper bound	2187.92	2183.49	2188.88	2188.33	2185.49
CI Lower bound	1761.04	1761.95	1765.20	1766.31	1765.87

5.3.1.1.2 SPONGENT analysis

In Table 5-4, SPONGENT-224-224-112 has the lowest standard deviation of 0.09 ms. The two variants with the lowest latency SPONGENT-088-176-088 and SPONGENT-160-160-080 have the same deviation of 0.16 ms. A high deviation of 0.69 ms occurred in SPONGENT-256-256-016.

Determining the deviation of the maximum memory consumption by SPONGENT in Table 5-5, shows little deviation of 487.36 kB by SPONGENT-224-448-224 and high deviation of 506.27 kB by SPONGENT-160-320-160.

Table 5-4: Data analysis for SPONGENT latency using 10 log file sizes

	Mean latency (ms)	Standard Error	Standard Deviation	Confidence Level (95.0%)	CI Upper bound	CI Lower bound
SPONGENT-088-080-008	2.28	0.06	0.20	0.14	2.42	2.14
SPONGENT-088-176-088	1.59	0.05	0.16	0.11	1.70	1.48
SPONGENT-128-128-008	4.35	0.09	0.30	0.21	4.56	4.14
SPONGENT-128-256-128	1.88	0.06	0.20	0.15	2.03	1.73
SPONGENT-160-160-016	4.29	0.07	0.21	0.15	4.44	4.14
SPONGENT-160-160-080	1.61	0.05	0.16	0.11	1.72	1.50
SPONGENT-160-320-160	2.93	0.04	0.13	0.10	3.03	2.83
SPONGENT-224-224-016	9.37	0.12	0.39	0.28	9.65	9.09
SPONGENT-224-224-112	2.41	0.03	0.09	0.06	2.47	2.35
SPONGENT-224-448-224	2.93	0.04	0.13	0.09	3.02	2.84
SPONGENT-256-256-016	12.93	0.22	0.69	0.49	13.42	12.44
SPONGENT-256-256-128	2.82	0.07	0.21	0.15	2.97	2.67
SPONGENT-256-512-256	3.61	0.09	0.29	0.21	3.82	3.40

Table 5-5: Data analysis for SPONGENT memory consumption using 10 log file sizes

	Mean memory (kB)	Standard Error	Standard Deviation	Confidence Level (95.0%)	CI Upper bound	CI Lower bound
SPONGENT-088-080-008	3552.00	156.72	495.59	354.52	3906.52	3197.48
SPONGENT-088-176-088	3553.24	157.38	497.68	356.02	3909.26	3197.22
SPONGENT-128-128-008	3554.00	154.60	488.89	349.73	3903.73	3204.27
SPONGENT-128-256-128	3556.16	154.45	488.41	349.39	3905.55	3206.77
SPONGENT-160-160-016	3552.92	154.91	489.85	350.42	3903.34	3202.50
SPONGENT-160-160-080	3559.88	156.91	496.18	354.95	3914.83	3204.93
SPONGENT-160-320-160	3560.88	160.10	506.27	362.16	3923.04	3198.72
SPONGENT-224-224-016	3556.84	158.82	502.23	359.28	3916.12	3197.56
SPONGENT-224-224-112	3565.28	155.97	493.23	352.84	3918.12	3212.44
SPONGENT-224-448-224	3555.00	154.12	487.36	348.63	3903.63	3206.37
SPONGENT-256-256-016	3558.84	154.91	489.88	350.44	3909.28	3208.40
SPONGENT-256-256-128	3549.36	158.61	501.58	358.81	3908.17	3190.55
SPONGENT-256-512-256	3554.16	155.07	490.37	350.79	3904.95	3203.37

SPONGENT variants have less overall latency as indicated by the mean ranging from 1.59 to 12.93 ms when PHOTON variants ranged from 5727.63 to 16274.53 ms. However, the PHOTON variants have a smaller memory footprint ranging from 1972.72 to 1977.32 kB than those of SPONGENT with a range of 3565.28 to 3549.36 kB. It is evident that SPONGENT was designed to have faster processing capabilities and PHOTON for less memory consumption.

For some variants, their performance increases as the data log file size increases, while for other variants, it decreases for PHOTON and SPONGENT. The excel data analysis toolpak plugin was used to calculate skewness statistic to determine the symmetry of the measured results for test case 1. Skewness is a measure of the symmetry of the probability distribution of values from the mean. The skewness value can be positive, zero, or negative. Positively skewed data has a mean greater than the median, while negatively skewed data has a mean less than the median, and zero-skewed data has the same mean as the median. We calculated the skewness of the measured results in test case 1 to confirm the observed non-linearity relationship between the performance measurements when the data log file increases for some variants. Table 5-6 shows the skewness of the measured latency and memory consumption.

Table 5-6: The skewness of data log file sizes on the performance of SPONGENT

	Latency	Memory
SPONGENT-088-080-008	0.767	0.188
SPONGENT-088-176-088	-0.004	0.250
SPONGENT-128-128-008	0.452	0.206
SPONGENT-128-256-128	0.141	0.242
SPONGENT-160-160-016	-0.268	0.186
SPONGENT-160-160-080	0.620	0.264
SPONGENT-160-320-160	0.362	0.309
SPONGENT-224-224-016	-0.101	0.188
SPONGENT-224-224-112	-0.223	0.195
SPONGENT-224-448-224	0.144	0.142
SPONGENT-256-256-016	1.061	0.196
SPONGENT-256-256-128	0.181	0.227
SPONGENT-256-512-256	0.501	0.221

Most of the SPONGENT variants are positively skewed towards the mean, with the exception of SPONGENT-088-176-088, SPONGENT-160-160-016, SPONGENT-224-224-016, and SPONGENT-224-224-112, which have negative skewness for latency performance. These variants with negative skew for latency confirms that the latency fluctuated as the data log file sizes increased. All SPONGENT variants have a positive skewness for memory consumption. The skewness statistics confirm that the maximum memory consumed by SPONGENT increased as the data log sizes increased.

Table 5-7: The skewness of data log file sizes on the performance of PHOTON

	Latency	Memory
PHOTON-80-20-16	0.004	-0.023
PHOTON-128-16-16	-0.002	0.009
PHOTON-160-36-36	-0.001	-0.020
PHOTON-224-32-32	0.002	0.021
PHOTON-256-32-32	-0.002	0.037

Table 5-7 shows the distribution of latency and memory consumption from the mean for PHOTON variants. PHOTON-80-20-16 and PHOTON-224-32-32 are the only two variants out of five variants with positive skew. PHOTON-128-16-16 and PHOTON-256-32-32 have the same skew of -0.002 ms. In terms of the memory consumption, PHOTON-80-20-16 and PHOTON-160-36-36 have a negative skew, which confirms that the memory consumption measurements have a mean less than the median. The memory consumption fluctuated significantly as the data log file size increased.

In summary, SPONGENT has been shown to have a small overall latency compared to PHOTON, when hashing the ten data log files. PHOTON, on the other hand, uses less memory than SPONGENT. It can be concluded that SPONGENT variants are better suited for IoT applications that require faster data processing, whereas PHOTON is better suited for IoT devices with limited memory. SPONGENT is designed to provide different levels of security and to accommodate scenarios with extremely limited and low preimage security requirements; highly constrained applications with low and middle collision resistance requirements; and applications requiring full preimage and second preimage security requirements. The last category is for applications that are not resource-constrained and require the maximum security levels, which IoT applications are not. As a result, many IoT applications will rarely use the last category.

5.3.2 Test case 2: The effect of the input rate on the latency and program memory consumption

In test case 2, we modified the second predefined input rate size parameter for the algorithms; for SPONGENT-n/*c*/r and PHOTON-n/*r*/r'. By predefine, we mean the original sizes that the designers of the hash functions published. To investigate the impact of the hash function parameters on the performance, we studied how adapting one of the three parameters affected the performance. This parameter is called the capacity for SPONGENT variants, which are the remaining bits after the input message has been absorbed, while it is called the input rate for PHOTON. In this test case, this parameter will be referred to as the capacity/input rate (symbol: *cr*) to accommodate the different names from the two hash functions. To be able to benchmark the performance of PHOTON and SPONGENT, the input rates used were 128, 160, and 176 bits. Compared to the unmodified parameters, the selected input rates are high for PHOTON variants and SPONGENT-088-080-008, SPONGENT-128-128-008 when the capacity/input rate is 160 and 176 bits. For the original SPONGENT-160-160-016 and SPONGENT-160-160-080, the selected input rate is increased when we use a capacity/input rate of 176 bits. The SPONGENT variants that originally had an input rate of more than 128 bits, had their new capacity/input rate decreased when using the selected input rates. The input rate and capacity parameters define the trade-off between speed and security of a hash function. Higher rates result in a faster but less secure lightweight cryptographic hash function [80].

By changing the capacity/input rate parameter, we defined our own variants. However, the core underlying structure (sponge construction initialisation, number of rounds, and permutation function) of the hash functions was not modified as indicated by the pseudo-codes in Figure 4-2 and Figure 4-3. When changing the capacity/input rate, we updated the versions of the variants to match the new format. For example, SPONGENT-n/*c*/r is now represented as SPONGENT-n/*cr*/r and PHOTON-n/*r*/r' as PHOTON-n/*cr*/r'. To illustrate using SPONGENT, the smallest SPONGENT variant with capacity/input rate set to 128 bits changes to SPONGENT-088-128-088 from SPONGENT-088-176-088. This new variant has the same underlying structure and operations as the original SPONGENT-088-176-088. The same logic applies to PHOTON variants. Unlike in test case 1, the data log file size in test case 2 is fixed at 500 kB. 500kB was chosen because it is the average file size of the ten data log files used in test case 1.

The modification of the capacity/input rate parameter has an impact on the second-preimage and collision resistance security levels of the hash function. For SPONGENT, the security levels of only the variants that offer reduced second-preimage security using the formula $2^{c/2}$ as explained in section 3.2.1 are affected. The details of the changed security levels are in Table 5-8 and Table 5-9 for SPONGENT and PHOTON respectively. Table 5-8 shows the changed security levels (in

bits) of the variants with adapted capacity/input rate where “Pre” is the preimage resistance, “2nd pre” is the second-preimage resistance and “Col” is the collision resistance.

Different font colors are used to highlight the security levels that have increased (green), remained the same (blue) or decreased (red) after modifying SPONGENT, relative to the unmodified variants.

The security levels of variants with security requirements that use a capacity/input rate in their formulas were the only ones affected by the modification of the capacity/input rate. Table 5-8 shows that when the capacity/input rate was 128 bits, the second-preimage and collision bits of SPONGENT-088-080-008 increased to 64 bits; the security levels of SPONGENT-128-128-008 did not change as the unmodified capacity/input rate was 128.

Table 5-8: SPONGENT security levels for modified capacity/input rate

	Unmodified			**128 bits**			**160 bits**			**176 bits**		
	Pre	**2nd pre**	**Col**	**Pre**	**2nd pre**	**Col**	**Pre**	**2nd pre**	**Col**	**Pre**	**2nd pre**	**Col**
SPONGENT-088-080-008	80	40	40	80	64	64	80	80	80	80	88	88
SPONGENT-128-128-008	120	64	64	120	64	64	120	80	80	120	88	88
SPONGENT-160-160-016	144	80	80	144	64	64	144	80	80	144	88	88
SPONGENT-224-224-016	208	112	112	208	64	64	208	80	80	208	88	88
SPONGENT-256-256-016	240	128	128	240	64	64	240	80	80	240	88	88

When the capacity/input rate was 160 bits, SPONGENT-224-224-016 and SPONGENT-256-256-016 decreased the second-preimage and collision resistance from 112 to 80 bits and 128 to 80 bits, respectively. The capacity/input rate set to 176 bits lowered SPONGENT-224-224-016 and SPONGENT-256-256-016 second-preimage and collision resistance to 88 bits, but SPONGENT-088-080-008, SPONGENT-128-128-008 and SPONGENT-160-160-016 increased.

PHOTON designers extended the sponge construction to provide trade-offs between preimage security and small message hashing speed. PHOTON variants have ideal collision resistance and

reduced second-preimage resistance, similar to SPONGENT variants that belong to the category of reduced second-preimage security requirements.

In Table 5-9 the security levels are not shown per modified capacity/input rate like in Table 5-8. This is because the second-preimage and collision resistance security levels did not change for PHOTON variants like SPONGENT variants. SPONGENT was originally designed to use the capacity *c*, which is the remaining bits after the message blocks have been absorbed as the capacity/input rate. The PHOTON parameter that is modified in test case 2 is not used in the formulas that determine the second-preimage and collision resistance security levels. This made the security levels for modified PHOTON variants the same as those of the unmodified PHOTON shown in Table 5-9. This was significant as we improved the latency (see Figure 5-21) and memory consumption for some variants (see Figure 5-22) without affecting the original second-preimage resistance and collision security levels.

Table 5-9: PHOTON security levels for modified capacity/input rate

	Pre	**2^{nd} pre**	**Col**
PHOTON-80-20-16	64	40	40
PHOTON-128-16-16	112	64	64
PHOTON-160-36-36	124	80	80
PHOTON-224-32-32	192	112	112
PHOTON-256-32-32	224	128	128

5.3.2.1 SPONGENT evaluation results

This sub-section gives an overview of the SPONGENT results and findings for investigating the effect of the capacity/input rate on the latency and the memory consumption of program at run time. The next section will start with latency results thereafter present the memory consumption.

5.3.2.1.1 Latency measurement

The first set of results were conducted with the unmodified parameters. This was the original reference code implementation published by its authors. We will start with presenting the results for unmodified capacity/input parameter, then when the capacity/input parameter is 128 bits, 160 bits and lastly 176 bits for latency measurement and in the same order for memory the results will be in the next section.

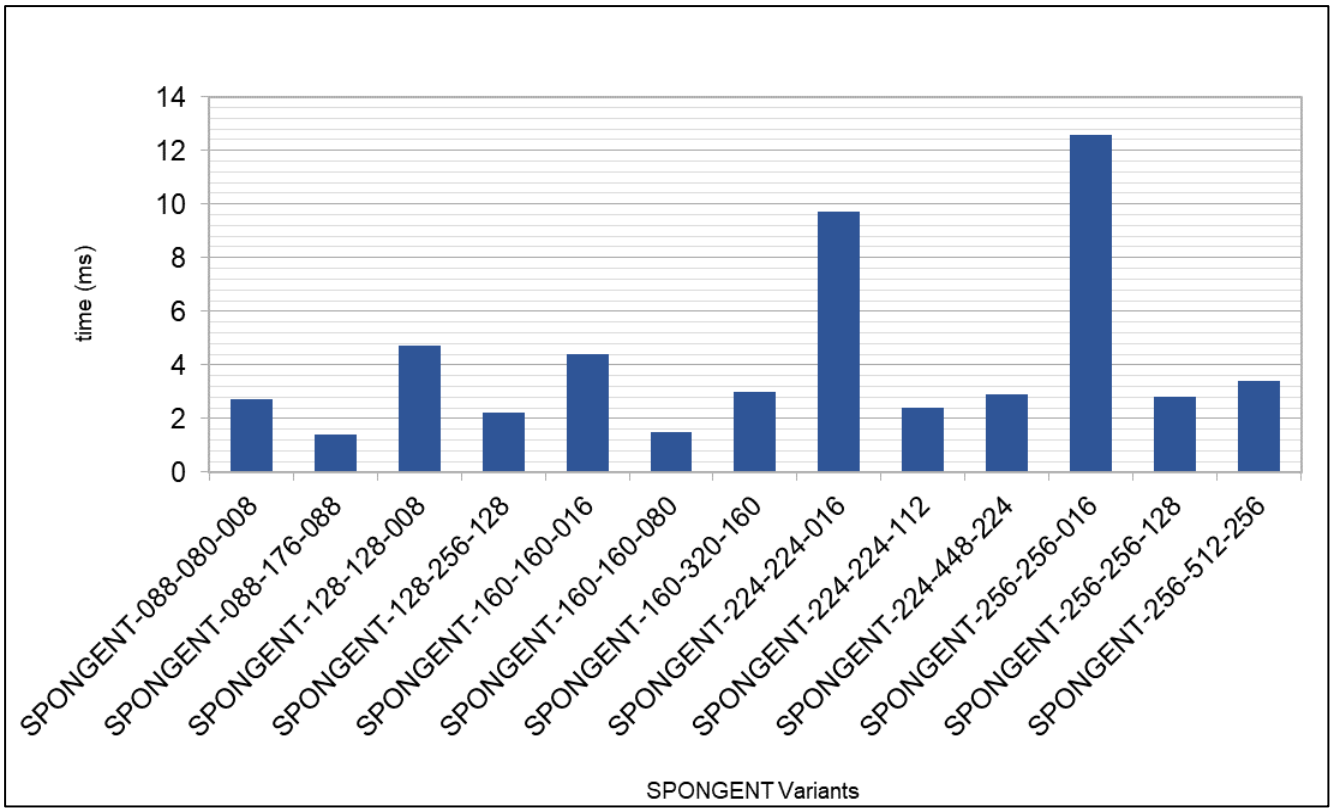

Figure 5-7: Latency of unmodified capacity/input rate parameter of SPONGENT 500 kB file log

As mentioned during the discussion of the results of test case 1, in each SPONGENT-n group the SPONGENT variants with a small output rate have a higher latency than other variants in the same group, as shown in Figure 5-7. SPONGENT-088-176-088 has the lowest latency of 1.4 ms followed by SPONGENT-160-160-080 at 1.5 ms while SPONGENT-256-256-016 has the highest latency of 12.6 ms to hash the readings in a 500 kB data log file.

According to SPONGENT designers [60], the SPONGENT variants with a hash output size of 88 bits, i.e., SPONGENT-088-080-008 and SPONGENT-088-176-088, are designed for extremely restricted scenarios and low preimage security requirements. The low latency performance of SPONGENT-088-176-088 confirms that this variant is more suitable for IoT applications with little to no processing delay.

Test case 2-1: Performance of SPONGENT at 128 bits

When setting the capacity/input rate to 128 bits, the overall latency as indicated in Figure 5-8 decreased compared to Figure 5-7. SPONGENT-128-128-008 has the lowest latency of 0.8 ms,

followed by SPONGENT-088-128-088 at 1.1 ms. The SPONGENT-256-128-016 is the slowest variant, with 7.7 ms.

Surprisingly, SPONGENT-128-128-008 in Figure 5-8 decreased by 3.9 ms compared to SPONGENT-128-128-008 in Figure 5-7, even though the unmodified input rate was also 128 bits. One would think that this variant was modified and unverified. This was ruled out by verifying SPONGENT-128-128-008 by generating the variant's test vectors and the hash value using the original input message and comparing them with the ones provided by the hash function designer as detailed in section 3.5.1. They passed the verification, so the cause of the difference is unknown as the rate was 128 bits in both cases.

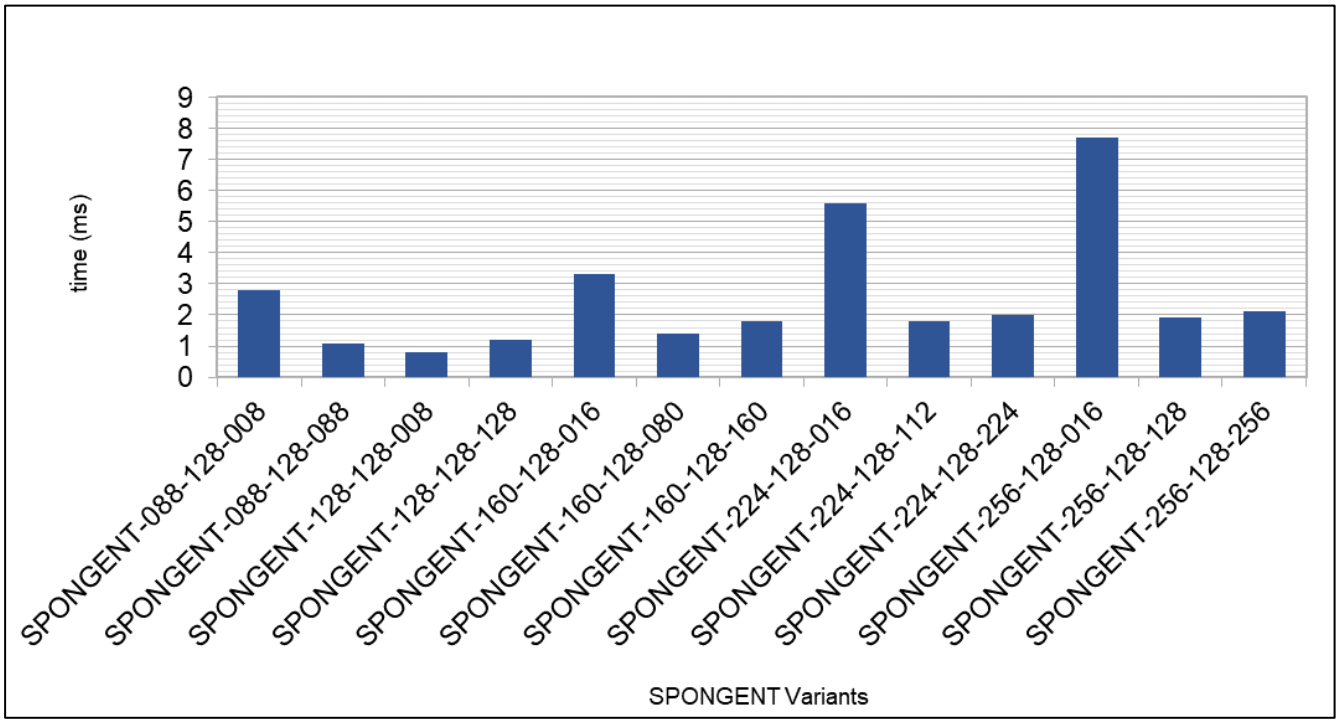

Figure 5-8: Latency of SPONGENT with capacity/input rate parameter of 128 bits

Test case 2-2: Performance of SPONGENT at 160 bits

As shown in Figure 5-9, SPONGENT-128-160-008 is the most latency-efficient variant (1 ms), followed by SPONGENT-088-160-088 with 1.5 ms, and SPONGENT-256-160-016 with the highest latency of 8.2 ms. When the capacity/input rate parameter was set to 160 bits, the overall latency increased as indicated in Figure 5-9 compared to the case of 128 bits in Figure 5-8.

All the SPONGENT variants have higher latency, with the exception of SPONGENT-224-160-112 and SPONGENT-224-160-224, which decreased by 6% and 10%, respectively. SPONGENT-

128-160-128 had the biggest increase of 50%, with SPONGENT-256-160-256 experiencing the smallest increase of 5%. The major contribution to the increase was caused by the increase in capacity/input rate from 128 to 160 bits, while the output rate did not change.

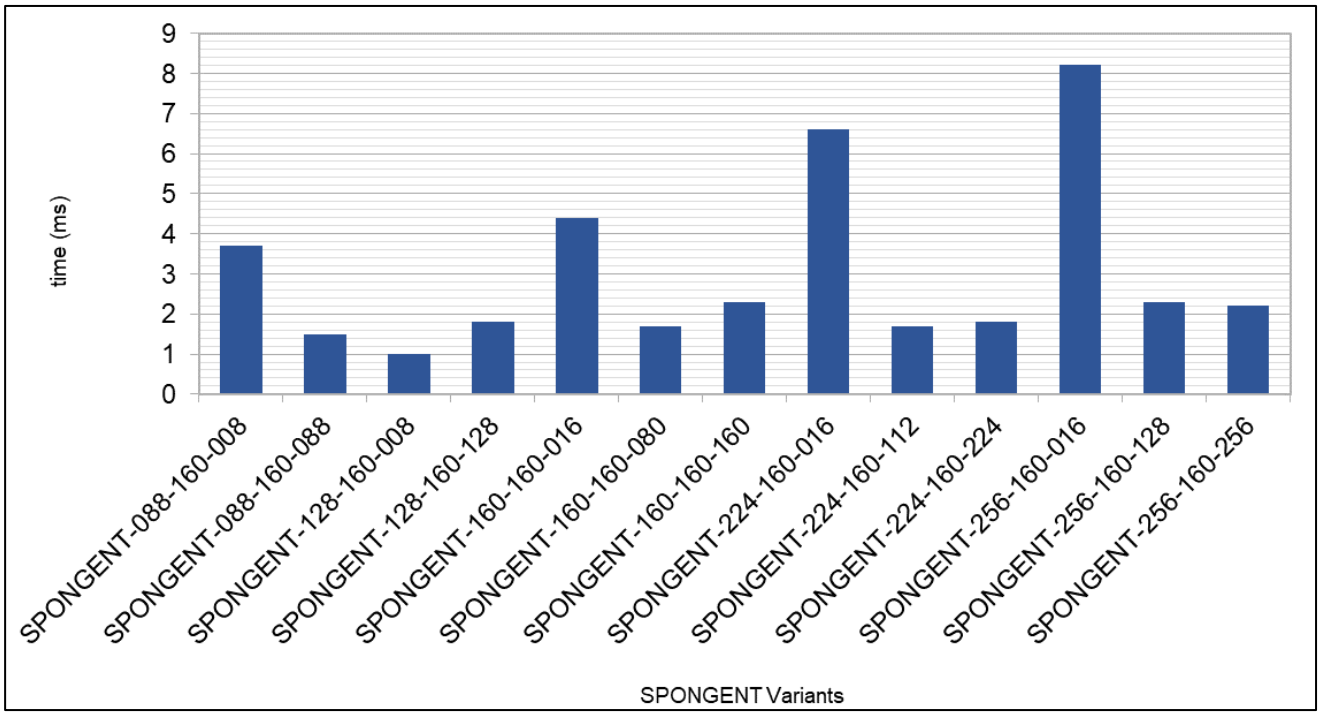

Figure 5-9: Latency of SPONGENT with capacity/input rate parameter of 160 bits

Test case 2-3: Performance of SPONGENT at 176 bits

As shown in Figure 5-10, SPONGENT-128-176-008 and SPONGENT-128-176-128 have the lowest latency of 1.3 ms, with SPONGENT-088-176-088 and SPONGENT-160-176-080 at 1.4 ms. SPONGENT-256-176-016 has the highest latency of 9.1 ms. Looking at 160 bit and 176 bit cases, SPONGENT-224-160-112 in Figure 5-9 and SPONGENT-224-176-112 in Figure 5-10 is the only variant with the same latency. In that case, SPONGENT-224-160-112 was expected to have less latency than SPONGENT-224-176-112 as the length of the hash value and output rate are the same, but the input rate is less than that of SPONGENT-224-176-112.

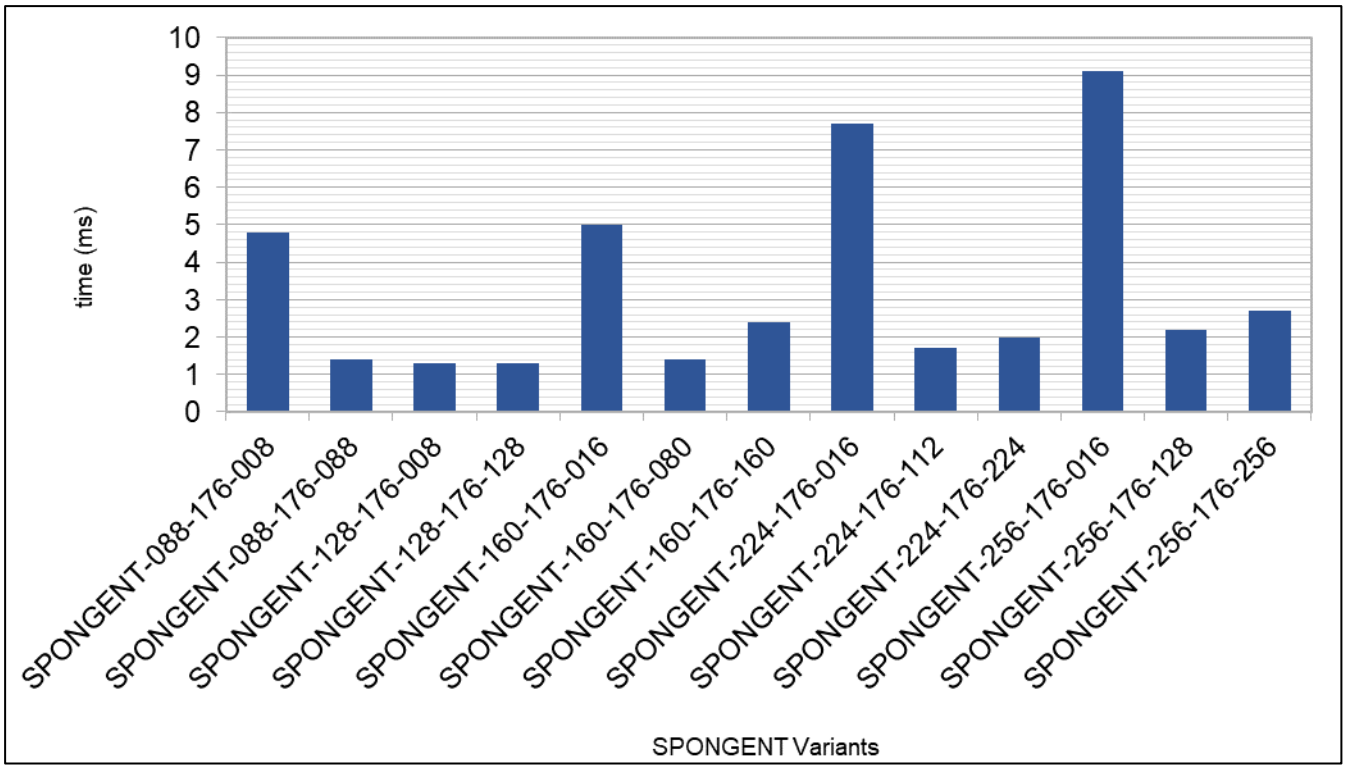

Figure 5-10: Latency of SPONGENT with capacity/input rate parameter of 176 bits

For some SPONGENT variants, the capacity/input rate in test case 2 was the same as the unmodified used in test case 1. The descriptive statistics are presented to confirm the trends noted in the SPONGENT latency performance. In an attempt to benchmark common variants between test case 1 and test case 2, the measurements of the 500 kB data log from test case 1 were used, as test case 2 used a fixed file size (also 500 kB). By common variants, we refer to the SPONGENT variants that have the same input/capacity parameter for test case 1 and test case 2, which are SPONGENT-088-176-088, SPONGENT-128-128-008, SPONGENT-160-160-016 and SPONGENT-160-160-080. Table 5-10 shows the statistics of four common variants in test case 1 and 2.

SPONGENT-088-176-088 and SPONGENT-160-160-016 had zero deviation between the results of the 500 kB on both test cases. SPONGENT-160-160-080 deviated by 0.14 ms and SPONGENT-128-128-008 by 2.76 ms.

Table 5-10: Descriptive statistics of the latency of common SPONGENT variants

	SPONGENT-088-176-088	SPONGENT-128-128-008	SPONGENT-160-160-016	SPONGENT-160-160-080
Mean latency (ms)	1.40	2.75	4.40	1.60
Standard Error	0.00	1.95	0.00	0.10
Standard Deviation	0.00	2.76	0.00	0.14
Confidence Level (95.0%)	0.00	24.78	0.00	1.27
CI Upper bound	1.40	27.53	4.40	2.87
CI Lower bound	1.40	-22.03	4.40	0.33

95% of the latency falls between the following confidence intervals:

- SPONGENT-088-176-088: exactly 1.40 ms
- SPONGENT-128-128-008: 27.53 and 22.03 ms
- SPONGENT-160-160-016: exactly 4.40 ms
- SPONGENT-160-160-080: 2.87 and 0.33 ms

SPONGENT-128-128-008 has a wide spread of latency measurements. As explained previously, we are not aware of the possible cause of this behaviour.

5.3.2.1.2 Memory consumption measurement

Memory consumption is the amount of maximum memory that SPONGENT and PHOTON utilise throughout their execution. Due to the limitations of memory in IoT devices, it is important that these algorithms consume less memory. For one, the Raspberry Pi used during the experiments had 1 GB of RAM, while other memory-limited devices could have less than one (1) GB; therefore, the hash functions should be able to run on this node. This subsection presents the memory utilised by these two hash functions.

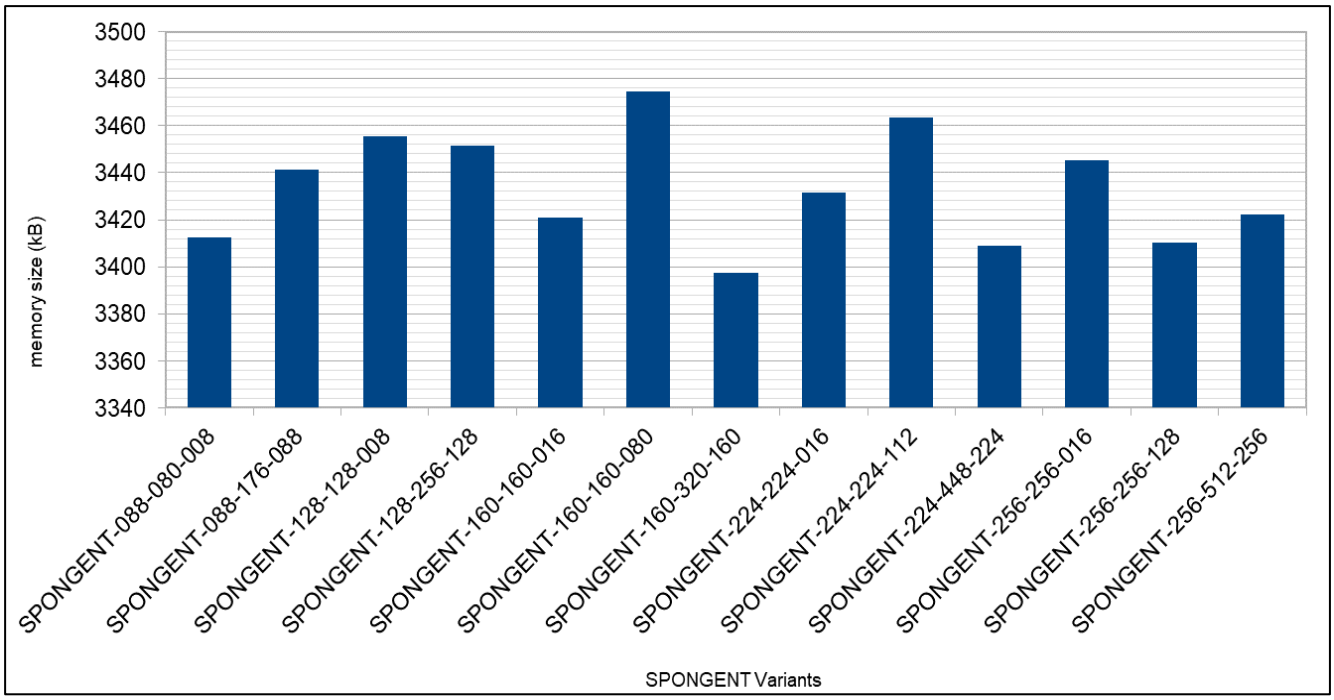

Figure 5-11: Memory consumption of unmodified capacity/input rate parameter of SPONGENT

Figure 5-11 shows the performance measurements of program memory where the parameters of the SPONGENT variants were unmodified. When the parameter was unmodified as shown in Figure 5-11, SPONGENT-160-320-160 used the smallest memory of 3397.6 kB, followed by SPONGENT-224-448-224 with 3408.8 kB. SPONGENT-160-160-080 is the second fastest unmodified as shown in Figure 5-7, however it traded-off that with the memory consumption; it consumed 3474.4 kB.

Test case 2-1: Performance of SPONGENT at 128 bits

As shown in Figure 5-12, SPONGENT-160-128-160 and SPONGENT-088-128-088 used less memory, 3402 kB and 3412 kB, respectively. SPONGENT-160-128-160 has the smallest memory footprint in both the 128 bit case and the unmodified case as shown in Figure 5-11 and Figure 5-12 even though in Figure 5-11 it increased by 4.4 kB.

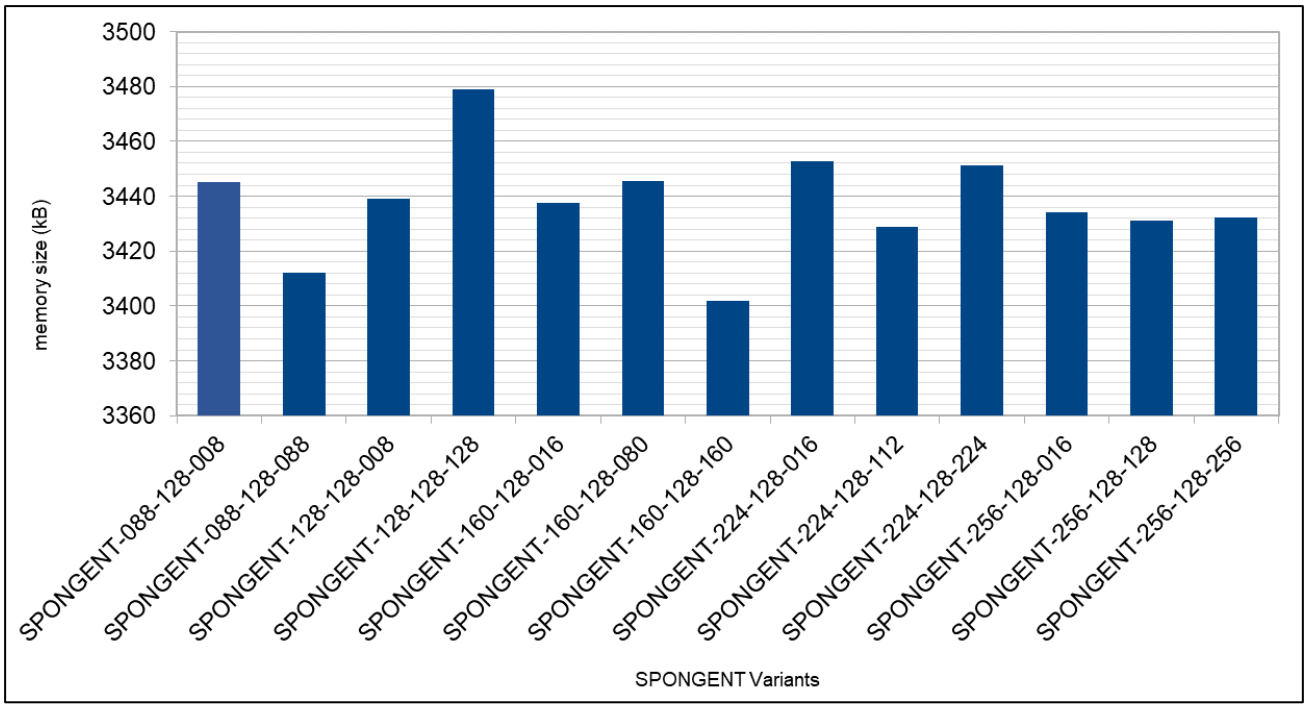

Figure 5-12: Memory consumption of SPONGENT with capacity/input rate parameter of 128 bits

Test case 2-2: Performance of SPONGENT at 160 bits

SPONGENT-160-160-160 utilised little memory of 3381.6 kB, followed by SPONGENT-256-160-128 with 3417.6 kB when the capacity/input rate was set to 160 bits as shown in Figure 5-13, while SPONGENT-160-160-080 used the largest memory of about 3480.4 kB.

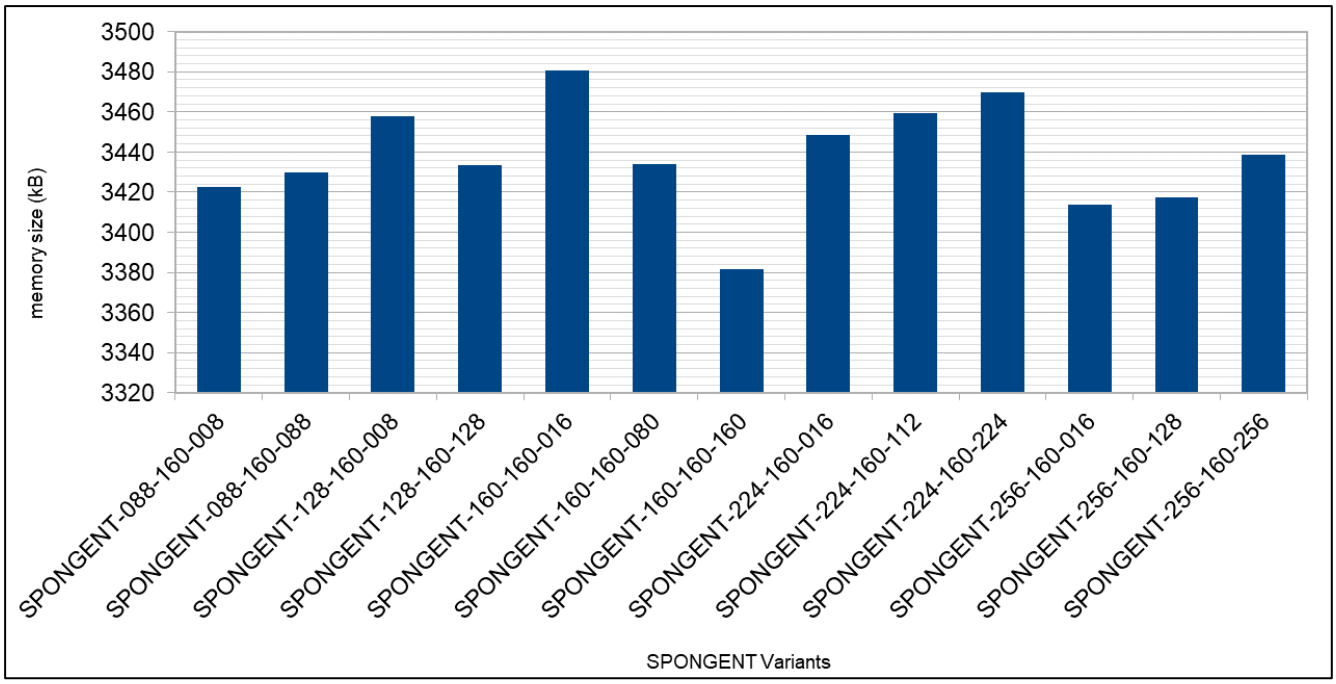

Figure 5-13: Memory consumption of SPONGENT with capacity/input rate parameter of 160 bits

Test case 2-3: Performance of SPONGENT at 176 bits

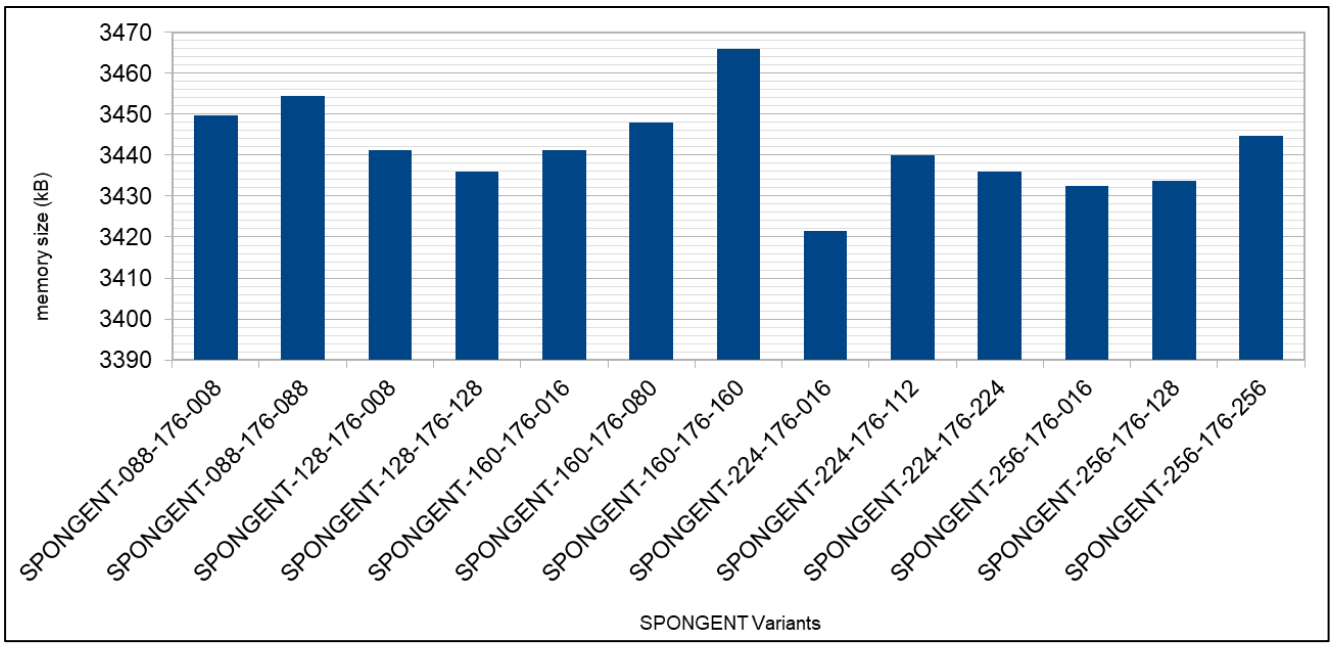

Figure 5-14: Memory consumption of SPONGENT with capacity/input rate parameter of 176

In Figure 5-14 overall memory consumption is approximately the same size as the overall memory in Figure 5-13 when the parameter was set to 160 bits. SPONGENT-224-176-016 had memory consumption of 3421.6 kB, followed by SPONGENT-256-176-016 with 3132.4 kB while SPONGENT-160-176-016 used the largest memory of 3466 kB.

For some SPONGENT variants, the capacity/input rate in test case 2 was the same as the unmodified used in test case 1, referred to as common variants. The descriptive statistics are presented to confirm the trends noted in the SPONGENT memory performance. In an attempt to benchmark some common variants between test case 1 and test case 2 (also 500 kB)., the measurements of the 500 kB data log from test case 1 were used, as test case 2 uses a fixed file size. Table 5-10 shows the statistics of four common variants in test case 1 and 2.

Table 5-11: Descriptive statistics of the memory consumption of common SPONGENT variants

	SPONGENT-088-176-088	**SPONGENT-128-128-008**	**SPONGENT-160-160-016**	**SPONGENT-160-160-080**
Mean memory (kB)	3447.80	3447.40	3450.60	3454.20
Standard Error	6.60	8.20	29.80	20.20
Standard Deviation	9.33	11.60	42.14	28.57
Confidence Level (95.0%)	83.86	104.19	378.64	256.67
CI Upper bound	3531.66	3551.59	3829.24	3710.87
CI Lower bound	3363.94	3343.21	3071.96	3197.53

SPONGENT-088-176-088 had little deviation of 9.33 kB, and SPONGENT-160-160-016 had wide fluctuation (42.14 kB) of the maximum memory consumption. 95% of the maximum memory consumption falls between the following confidence intervals:

- SPONGENT-088-176-088: 3531.66 and 3363.94 kB
- SPONGENT-128-128-008: 3551.59 and 3343.21 kB
- SPONGENT-160-160-016: 3829.24 and 3071.96 kB
- SPONGENT-160-160-080: 3710.87 and 3197.53 kB

A summary of the SPONGENT latency and memory consumption in test case 2 is shown in Figure 5-15 and Figure 5-16, respectively. The *x*-axis shows the SPONGENT variants with unmodified capacity/input rate values for easy reference.

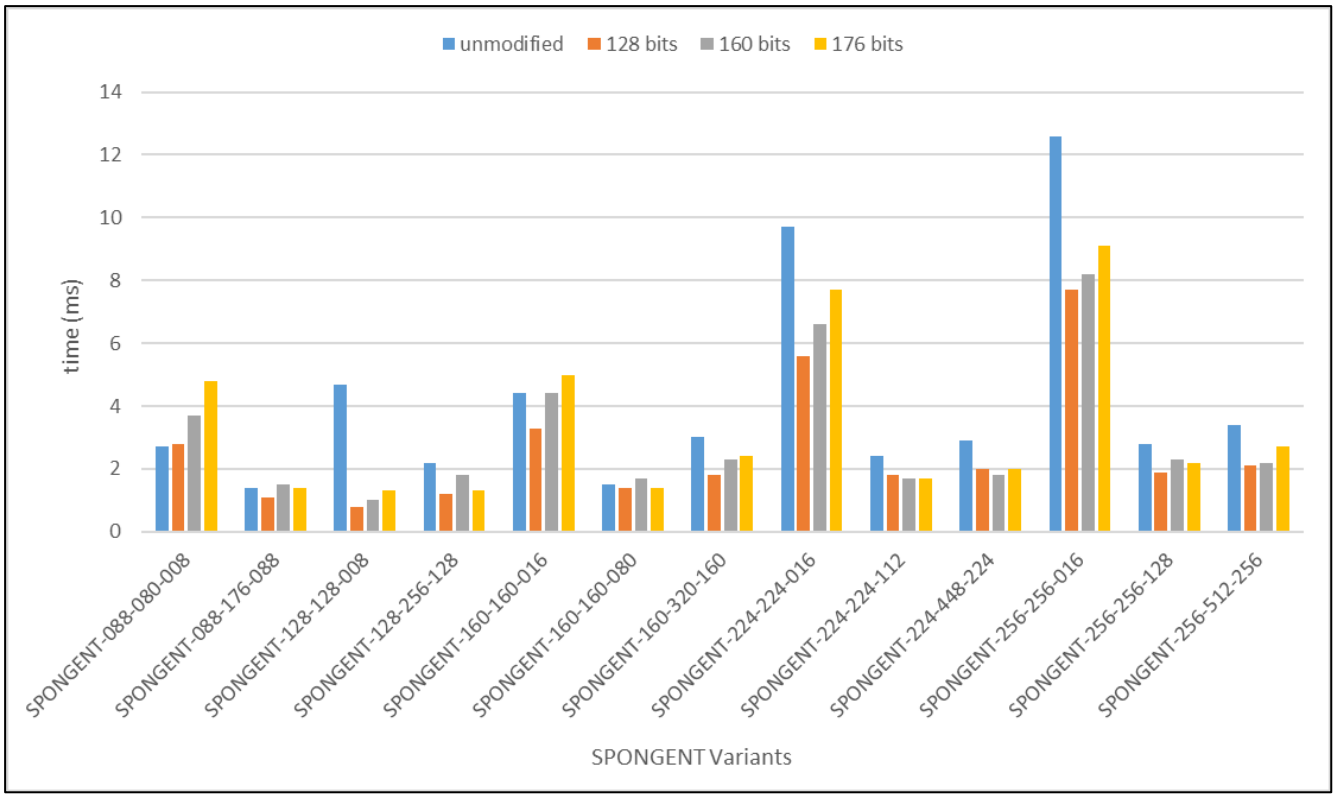

Figure 5-15: Summary of SPONGENT latency performance measurements for test case 2

From Figure 5-15, it is evident that the SPONGENT variants with unmodified capacity/input rate are generally slower to hash the 500 kB data log file contents, except for the SPONGENT-088 variants. SPONGENT-088-080-008 originally had the capacity/input rate parameter value of 80 bits (unmodified), changing it to 128, 160 and 176 bits introduced more delay, and hence these variants are slower to hash compared to the unmodified variant. Increasing the capacity/input rate for variants with small output rate (8, 16 88 bits) introduced further delay for those variants as the hash function spends more time processing the larger block message.

SPONGENT-128-*cr*-088 had a consistent overall latency of 0.8 ms, 1 ms and 1.3 ms when the capacity/input rate was set to 128, 160, and 176 bits outperforming other variants. SPONGENT-088-*cr*-088 was the second fastest variant at capacity/input rate of 128, 160 and 176 bits. In a SPONGENT-n group the variant with the small output rate has high latency than their counterparts in the same group.

Even though SPONGENT-128-*cr*-088 have the same latency for the selected three capacity/input rates, with the memory consumption it is not the case as shown in Figure 5-16. SPONGENT-160-*cr*-160 seemed to be the best in terms of the memory consumption performance where the *cr* (capacity/input rate) was unmodified (320 bits), 128 bits and 160 bits. The largest variants

SPONGENT-224-176-016 and SPONGENT-256-176-016 outperformed their counterparts only in the case where the *cr* was 176 bits.

From Figure 5-15 and see Figure 5-16, it is evident that for some variants less latency does not necessarily decrease the memory consumption. It is concluded that for SPONGENT variants the change in latency does not depend on the change in memory consumption and vice versa.

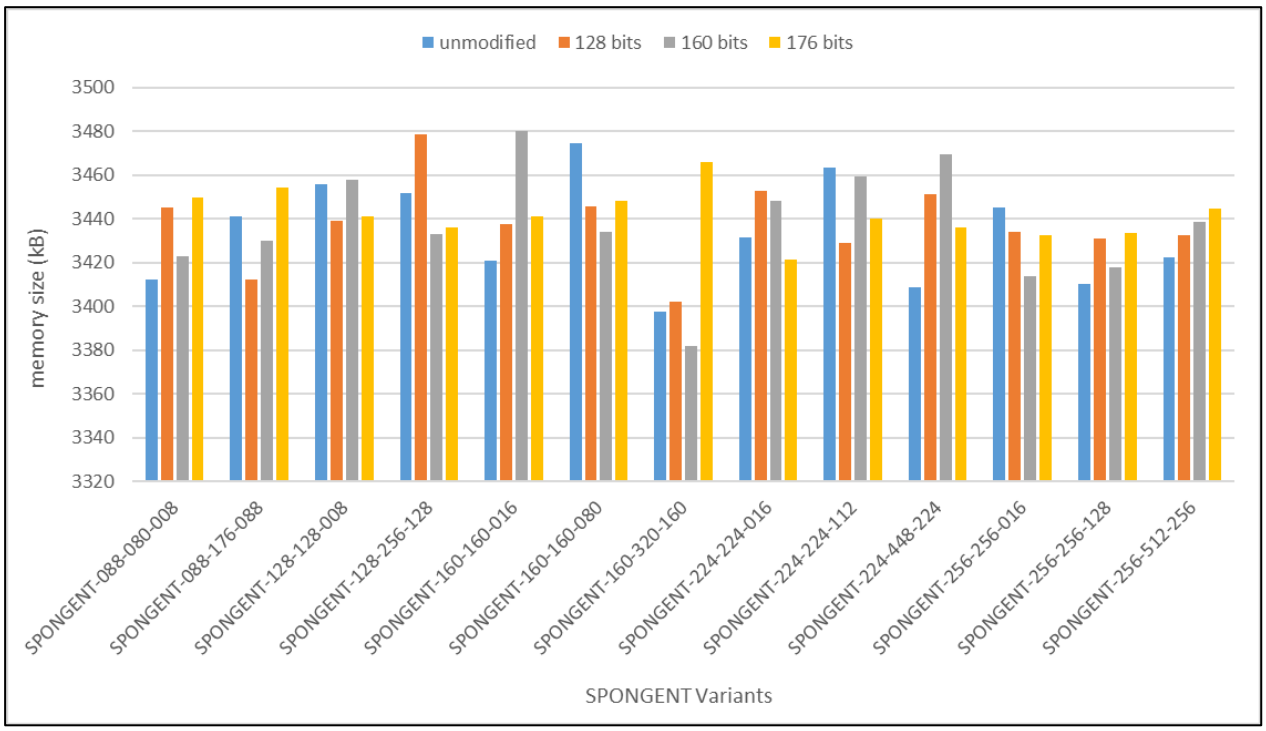

Figure 5-16: Summary of SPONGENT memory consumption performance measurements for test case 2

5.3.2.2 PHOTON evaluation results

This subsection gives an outline of the SPONGENT results and findings for investigating the effect of capacity/input rate on program latency and memory consumption at run time. PHOTON was originally designed to use capacity/input rate less, i.e., 16, 20, 32, and 36 bits than the hash value, unlike with SPONGENT variants. The capacity/input rate is the same as the output rate as listed in Table 3-2. To be able to evaluate and compare PHOTON performance with SPONGENT, PHOTON variants where the capacity/input rate was larger than the hash value were omitted. The following section will begin with latency results and then present memory consumption results.

5.3.2.2.1 Latency measurement

From Figure 5-17, with the unmodified capacity/input rate, PHOTON-80-20-16 is the fastest variant at 5207.6 ms, followed by PHOTON-160-36-36 at 9059.7 ms, while PHOTON-224-32-32 has the highest latency of 14809.6 ms. The variant PHOTON-256-32-32 outperforms PHOTON-224-32-32 by 27%, with longer hash value.

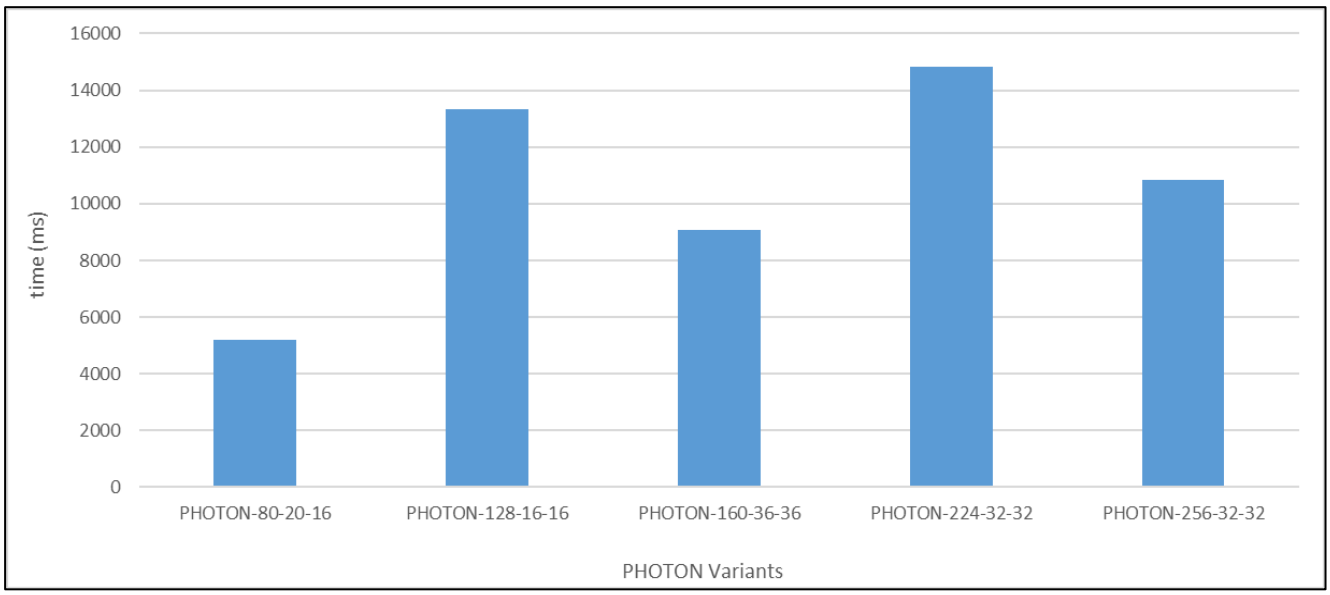

Figure 5-17: Latency of unmodified capacity/input rate parameter of PHOTON

Test case 2-1: Performance of PHOTON at 128 bits

PHOTON latency increases with the parameters (hash value) of the variants (PHOTON-128-128-16 to PHOTON-224-128-32), as shown in Figure 5-18. However, the latency of the PHOTON-256-128-32 variant does not follow this upward trend. From Figure 5-18, when the input rate parameter is 128 bits, PHOTON-128-128-16 has the lowest latency of 1681.8 ms, making it the fastest variant, while PHOTON-224-128-32 has the highest latency of 3711.1 ms, which is 67% more than PHOTON-128-128-16.

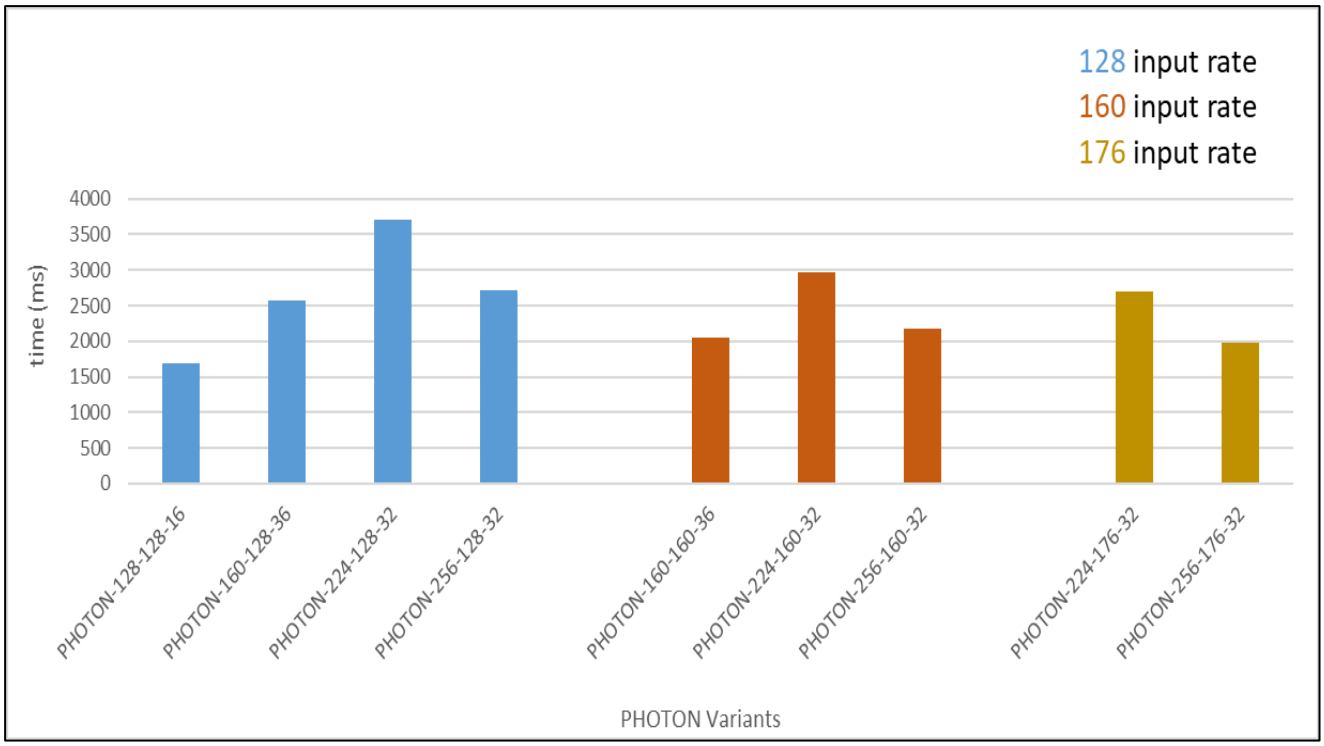

Figure 5-18: Latency of PHOTON with modified capacity/input rate parameter for 128, 160 and 176-bit rate cases

Test case 2-2: Performance of PHOTON at 160 bits

As shown in Figure 5-18, PHOTON-160-160-36 has the lowest latency of 2050.5 ms, while the variant with the highest latency is PHOTON-224-160-32 at 2975.5 ms, exceeding the latency of the largest variant, PHOTON-256-160-32 by 27%. It was interesting to observe that PHOTON-160-160-36, PHOTON-224-160-32, and PHOTON-256-160-32 decreased by 20.1%, 19.8%, and 20.2%, respectively, when the capacity/input rate increased from 128 bits to 160 bits.

Test case 2-3: Performance of PHOTON at 176 bits

As shown in Figure 5-18, PHOTON-256-176-32 has a lower latency of 1983.1 ms, while PHOTON-224-176-32 was 2697.9 ms. PHOTON-224-176-32 and PHOTON-256-176-32 decreased the latency by 9.3% and 8.7%, respectively, compared to PHOTON-224-160-32 and PHOTON-256-160-32 in test case 2-2 above.

5.3.2.2.2 Memory consumption measurement

From Figure 5-19, unmodified capacity/input rate PHOTON-224-32-32 utilised the smallest memory of 1923.6 kB, while PHOTON-80-20-16 is 1939.2 kB. The smallest variant which is

PHOTON-80-20-16 is designed to fit extremely resource-constrained devices however the memory resource usage is at the highest peak. This is a trade-off between the latency and memory consumption as PHOTON-80-20-16 is the fastest variant as shown in Figure 5-17.

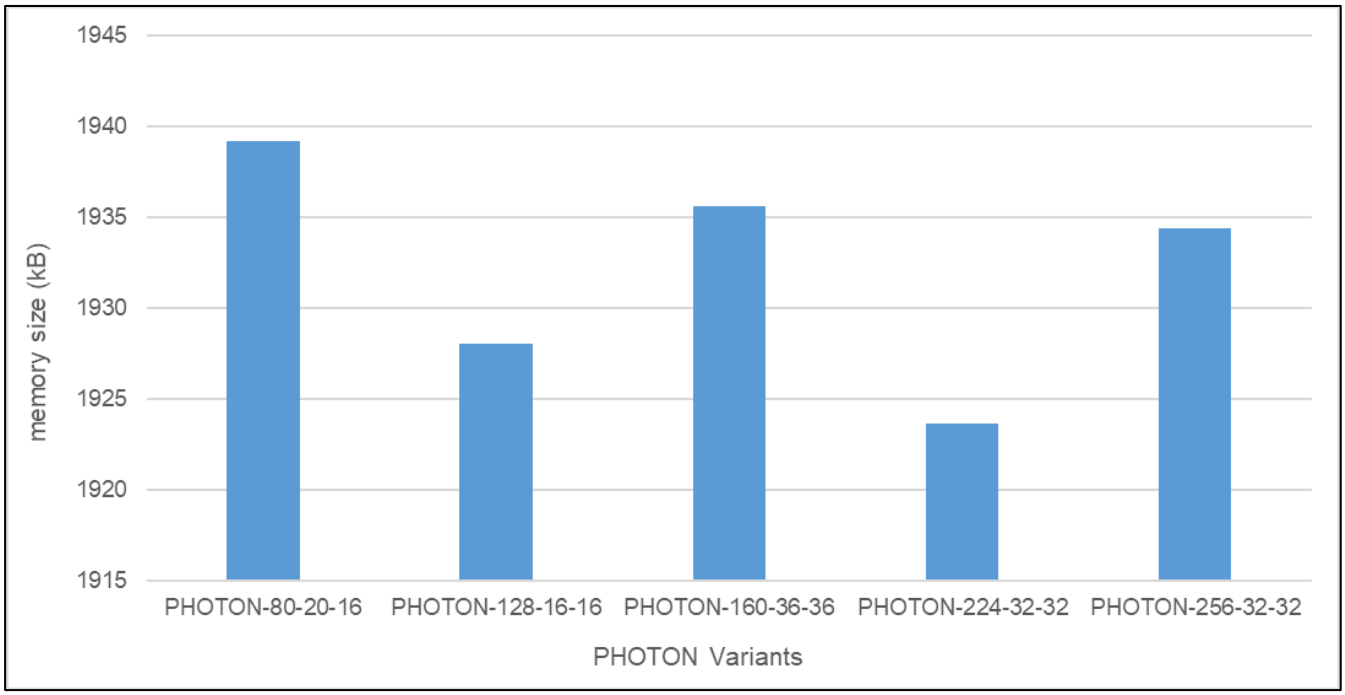

Figure 5-19: Memory consumption of PHOTON at unmodified capacity/input rate parameter

Test case 2-1: Performance of PHOTON at 128 bits

From Figure 5-20, when the parameter is set to 128 bits, PHOTON-256-128-32 uses little memory of 1925.6 kB, followed by PHOTON-128-128-16 at 1923.2 kB, which is more by just 2.4 kB. PHOTON-160-128-36 has the highest memory consumption of 1938 kB.

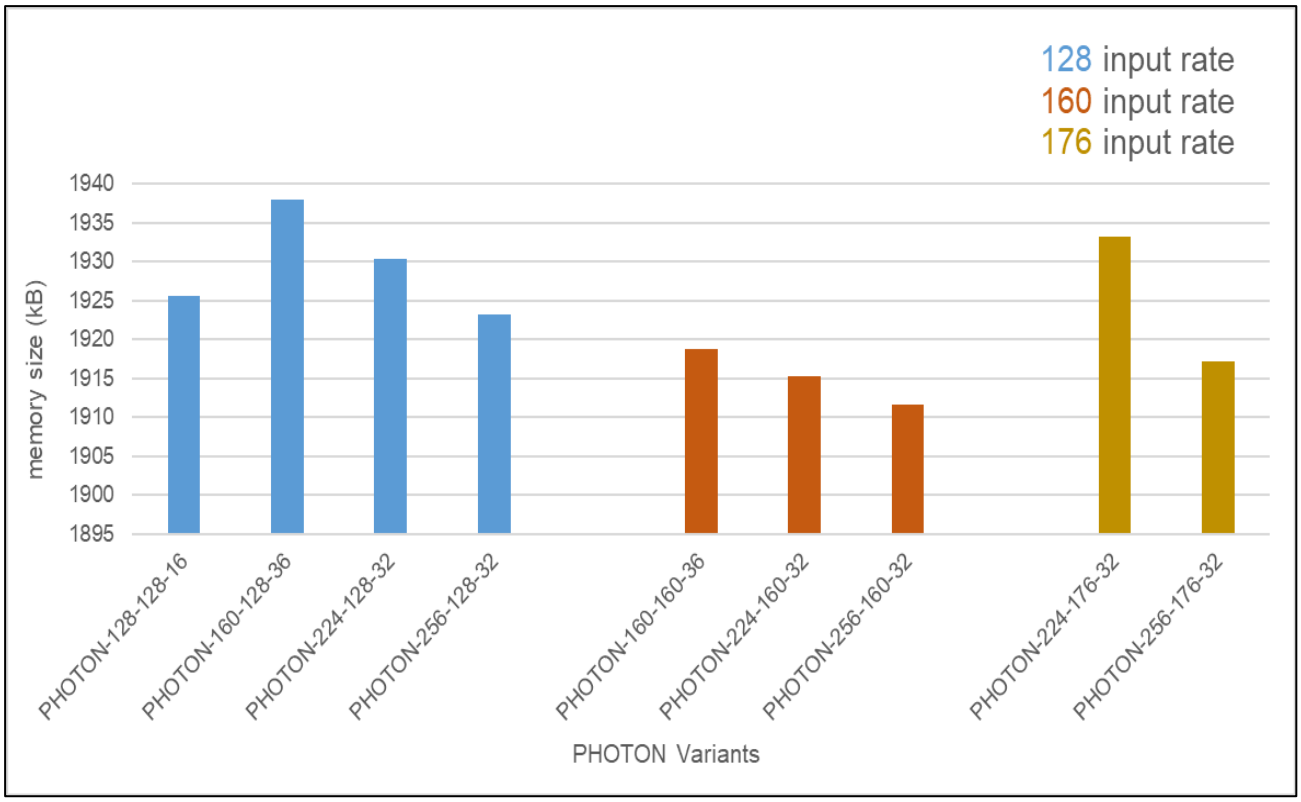

Figure 5-20: Memory consumption of PHOTON at capacity/input rate parameter for 128, 160 and 176-bit rate cases

Test case 2-2: Performance of PHOTON at 160 bits

As shown in Figure 5-18, PHOTON-256-160-32 is the second fastest variant when capacity/input is 160 bits, and in Figure 5-20, the same variant has little memory consumption of 1911.6 kB. Compared to when the capacity/input was 128 bits, the PHOTON-160-160-36, PHOTON-224-160-32, and PHOTON-256-160-32 decreased the memory consumption by 1%, 0.8%, and 0.6%, respectively. This confirms that modifying the capacity/input rate allows the PHOTON variants to have less memory usage.

Test case 2-3: Performance of PHOTON at 176 bits

When the capacity/input rate was set to 176 bits, only two PHOTON variants were evaluated. The memory consumption being measured by PHOTON-256-176-32 is 1917.2 kB and PHOTON-224-176-32 is 1933.2 kB, which both increased by 0.9% and 0.3% in the 160 bit case.

To conclude on PHOTON test case 2, Figure 5-21 and Figure 5-22 compare the results of test case 2 with the unmodified variants implemented. For easy reference, the x-axis labels are based on the unmodified parameters. Figure 5-21, clearly shows that PHOTON variants with the

unmodified capacity/input parameter take the longest time to execute the hash, whereas their maximum memory consumption is lower when executing. Increasing the capacity/input rate parameter drastically improved the latency of the variants. The higher the capacity/input rate value, the lower the latency. PHOTON-256-32-32 has less overall latency across 160 and 176 bit cases.

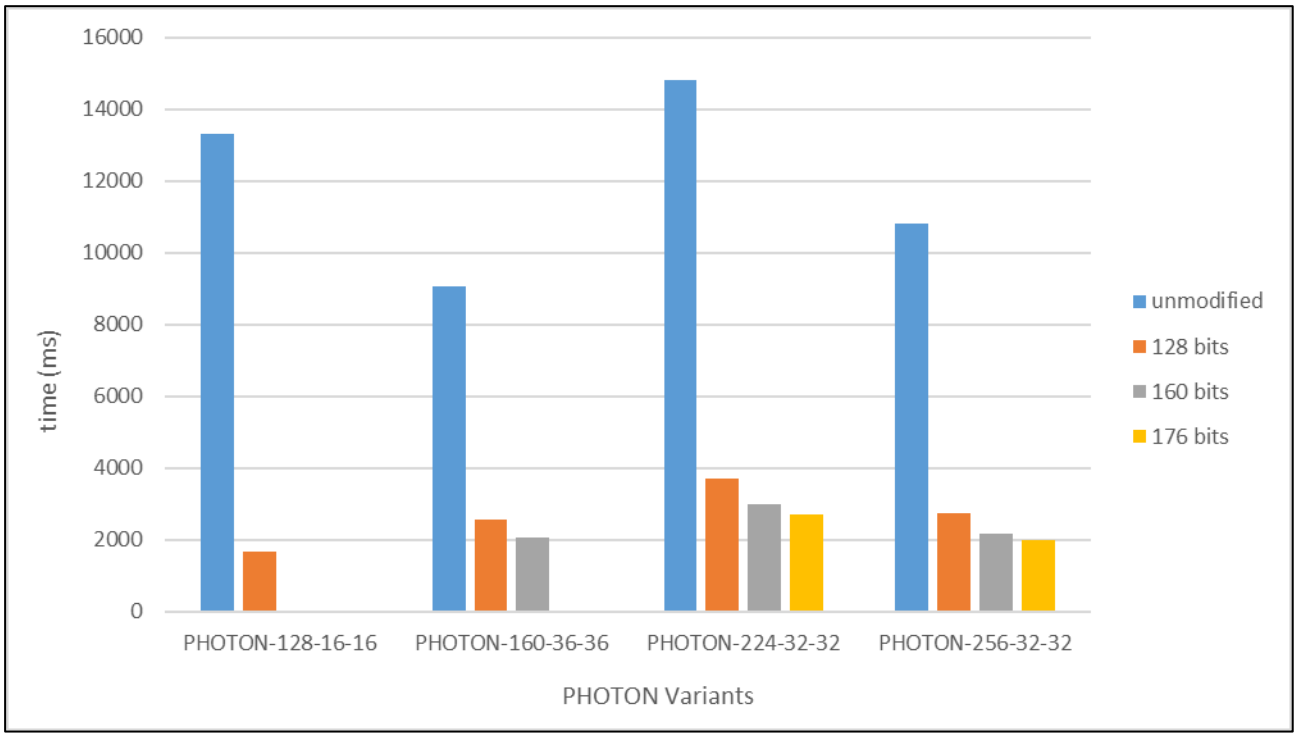

Figure 5-21: Summary of PHOTON latency performance measurements for test case 2

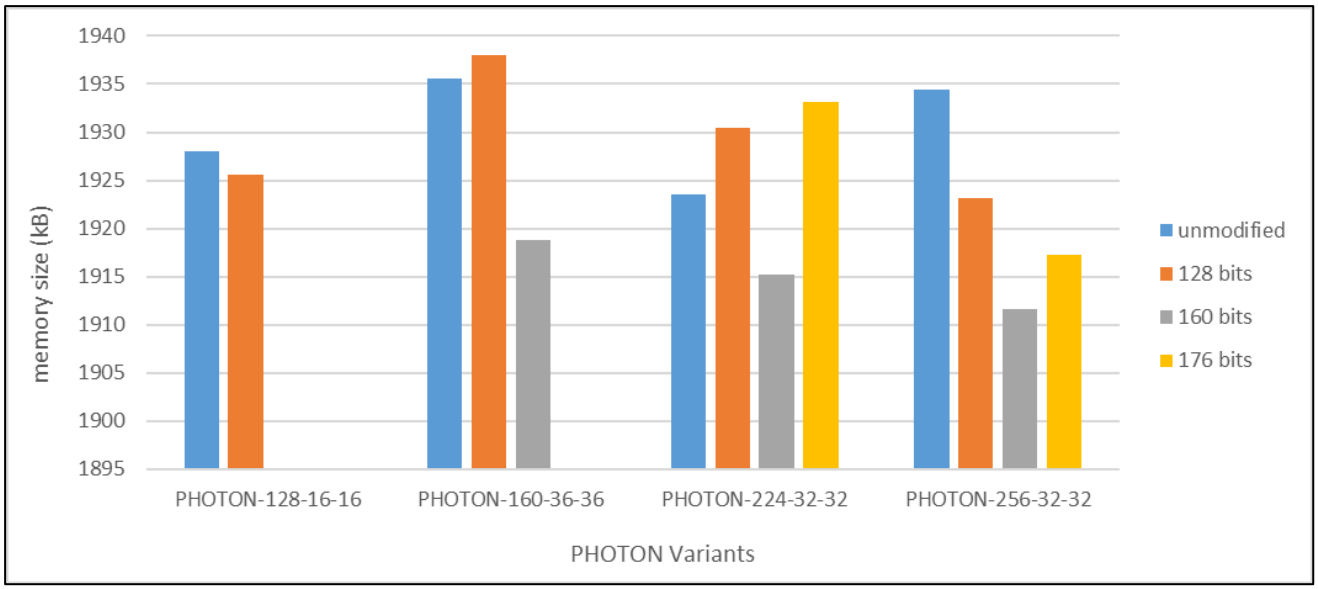

Figure 5-22: Summary of PHOTON memory consumption performance measurements for test case 2

PHOTON variants that have capacity/input rate greater than the length of the hash value were omitted as explained in the beginning of test case 2. From Figure 5-22, the memory consumption of the unmodified PHOTON-128-*cr*-16 and PHOTON-256-32-32 is larger than the modified variants.

PHOTON-224-160-32 and PHOTON-256-160-32 seem to be best suited for constrained devices as it has lower latency and consumes less memory compared to other variants. It was also observed that increasing the capacity/input rate decreased the latency. However, the decrease in latency did not decrease the memory consumption for some variants. The latency and memory consumption of PHOTON-160-160-36 and PHOTON-256-160-32 were improved when the capacity/input rate was 160 bits.

Unlike SPONGENT, changing the capacity/input rate parameter had no effect on the PHOTON variants' original security levels. The capacity bits of PHOTON variants are identical to the hash value bits, i.e. $n = c$. The input rate is denoted by r in PHOTON-n-r-r'. The PHOTON security levels are calculated using the following formulas: $2^{n-r'}$ for preimage security and $2^{c/2}$ for both second-preimage and collision resistance. r has no effect on the PHOTON variants' security levels. The unchanged security levels for the various input rates used are shown in Table 5-9.

For some SPONGENT variants, the latency and memory consumption were improved with the tradeoff of security. This is valuable insight if the hardware architecture supports it.

5.3.3 Conclusion on Test case 2 results

PHOTON is designed for applications requiring standard security levels. In addition, it is for small messages as the output rate is very small (16 bits to 36 bits). Compared to SPONGENT, the SPONGENT variants are aimed at achieving three types of security requirements. Firstly, full preimage and second-preimage where the output rate is the same as the hash bits (*n*). These are the largest five variants, where each variant is from the SPONGENT group based on the hash size. To accommodate applications that do not require high security, the designers implement the second security requirement, where the *n* is approximately equal to the capacity bits and the output *r* is very small. Here, the second-preimage has been reduced, and at a relatively small rate r, the loss of preimage security is limited.

Concluding test case 2, it was observed that for some variants, when the input parameter is increased, it decreases the time to hash the log file. When the parameters are decreased, the latency will be high. Choosing the best performing hash function is influenced by various factors. The performance of each hash function is dependent on the hash design and the environment in which they are evaluated.

5.4 Summary

This chapter presented the detailed analysis, interpretation, and findings. The SPONGENT and PHOTON latency and memory measurements were captured. The evaluations were done in two test cases: 1) to investigate the effect of log file size on latency and memory consumption and 2) the effect of the capacity/input rate on latency and memory consumption. For test case 1, an analysis of the results was conducted after presenting the mean, standard deviation, and confidence interval bounds for a 95% confidence level. For test case 2, it was observed that PHOTON has the worst overall latency performance compared to SPONGENT but uses less memory than SPONGENT. Only PHOTON variants with capacity/input greater than or equal to the length of the hash value were evaluated. PHOTON-160-176-36 is suited for devices with limited memory and that require fast processing time. The change in capacity/input rate for test case 2 reduced the security levels for some of the SPONGENT variants evaluated in this study. The next chapter will present the conclusions by answering the research questions from Chapter 1 and give directions for future work on this study.

CHAPTER 6 CONCLUSIONS AND FUTURE WORK

6.1 Introduction

This chapter provides a summary of the ideas, results, and findings presented in this study. The conclusion is presented in terms of the research questions, research objectives and the contributions made to the field of lightweight cryptography in the IoT.

6.2 Research Synopsis

At the beginning of this study, the problem that was posed was which lightweight data integrity algorithms perform better in IoT applications. Specifically, it was aimed at answering the following research questions:

- **Which commonly used lightweight cryptographic hash functions is robust and efficient for resource-constrained nodes?**

This study identified SPONGENT and PHOTON as two commonly family version lightweight hash functions. SPONGENT is a family of 13 variants developed to offer three security requirements and accommodate different types of constrained devices. The hash values of the SPONGENT variants range from 88 to 256 bits with five groups of SPONGENT variants, viz. SPONGENT-88, SPONGENT-128, SPONGENT-160, SPONGENT-224, and SPONGENT-256. PHOTON is a family of 5 variants for small messages and reduced second preimage resistance. PHOTON uses an extended domain sponge function where the input rate (r) may vary from the output rate (r').

After the development work in this study was initiated, there have been recent developments of lightweight hash functions after SPONGENT and PHOTON were proposed. LNMNT hash [63], which was proposed in 2021, is a 128 bit hash which was developed to utilise fewer rounds to minimize power consumption, memory utilisation and execution time. It uses the new Mersenne Number Transform to implement the diffusion property.

[65] presented an analysis of six (KECCAK, HASH-ONE, QUARK, PHOTON, SPONGENT and GLUON) various sponge construction-based lightweight hash functions and their suitability to the IoT applications. The metrics used for the evaluation include security, area requirement, digest, throughput, power requirement, and the cycle needed to execute. Their evaluation followed hardware-based implementation, and 0.13 μm and 0.18 μm ASIC was used as an implementation environment. Their conclusion shows that each hash function has its own achievements, where SPONGENT-88 has the lowest area requirement, SPONGENT-256 has the highest preimage resistance, and KECCACK-400 achieves the highest throughput.

Another work that followed hardware-based implementation is [81], which aimed at improving the area performance trade-off of the small variant PHOTON, PHOTON-80-20-16. It was implemented on Xilinx and Altera Field Programmable Gate Array (FPGA) devices, which offer low cost and high processor speed. Their results show that the improved PHOTON-80-20-16 offers a speed performance rate of 10.26% to 51.01% compared to the original hardware implemented PHOTON-80-20-16 [59]. [82] proposed a 1024 bit lightweight hash function for big data and IoT applications, that employs linear transformation, and bit permutation functionalities. It uses the similar construction used in MD5 and SHA-1. This hash function has a long hash value compared to the SPONGENT and PHOTON variants.

In terms of software implementation, [83] provided the benchmarks for eight primitives on a low-power and resource-limited computational device, and outlined an execution model for these primitives under intermittent powering. The metrics used to benchmark the primitive hash functions are clock cycles and the memory footprint comprised of ROM and RAM usage. A 16-bit MSP430FR5969 microcontroller (MCU) was used for performance evaluations.

SPONGENT and PHOTON were chosen for evaluations as they are a popular family version, which allowed them to be included in part 5 of ISO/IEC 29192 and their reference implementation code which is publicly available to the researchers.

- **What are the measures which can be used to evaluate the performance of the hash functions in resource-constrained devices?**

This research question aimed to determine the measures that can be used to fairly compare the performance of SPONGENT and PHOTON hash functions. Measures were selected based on guidelines discovered during the literature review in Chapter 2 under the performance implementation types, which are hardware and software. Latency and memory consumption are software implementation based measures which were chosen as performance measures to evaluate the proposed hash functions.

The works reviewed in the previous research question [59], [65], [81], [82] show that there are many hardware based implementation hash functions. Software implementation has not been fully explored. This is also highlighted by the work of [83], which motivates the need to explore the software-based implementation of hash functions, which we also followed in this study. The work of [83] evaluates the software-based implementation of traditional primitive hash functions, with SHA-3 being the only sponge construction-based hash function, while our work evaluates two sponge construction-based hash functions, which were originally implemented as hardware-based but were evaluated as software implementation in this study. This approach was followed because the hardware-based hash functions will be used in IoT applications that offer software

solutions. The identified performance measures, i.e., latency and memory, reflect the software implementation performance metrics, which enable software simulation, including real-world implementation of IoT nodes that make up an IoT ecosystem running the proposed hash functions.

- **How does lightweight cryptographic hash functions compare in terms of latency and memory consumption performance?**

The evaluation of lightweight hash functions in devices with limited resources depends on various factors, such as the experimental platform and the design of the actual hash function [60], [65], [54]. When investigating the effect of the data log file size on the performance of SPONGENT and PHOTON, SPONGENT has better overall latency than PHOTON. SPONGENT-088-176-088 is the fastest variant in 160 and 176 bit cases, while it is second fastest in 128 bit cases, after SPONGENT-128-128-008. The fastest SPONGENT variant is not the outperforming variant when it comes to memory consumption. SPONGENT-160-320-160, SPONGENT-160-128-160, and SPONGENT-160-160-160 lead the performance of memory consumption in unmodified, 128 and 160 bit test cases. In terms of PHOTON variants, the fastest variant is PHOTON-160-36-36 in unmodified, 128 and 160 bit cases, whereas PHOTON-256-160-32 has the lowest memory consumption. PHOTON has the highest overall latency performance compared to SPONGENT, but is designed to consume less memory than SPONGENT. Lightweight hash functions with larger number of rounds shows high memory utilisation and time of execution.

In test case 2, an improvement in the memory consumption was realised when the capacity/input rate was changed to 160 bits for PHOTON. PHOTON-160-160-36, PHOTON-224-160-32, and PHOTON-256-160-32 used less memory by 0.9%, 0.4%, and 1.2%, respectively, than their unmodified counterpart variants. This is significant as we managed to improve the memory consumption without reducing the PHOTON security levels.

6.3 Addressing the Research Objectives

This section addresses the study's research objectives and how they were attained. The research objectives for this study were as follows:

1. Identify typical constraints of the IoT node;
2. Identify measures for evaluating the performance of lightweight hash functions;
3. Identify two widely used lightweight cryptographic hash functions used for data integrity in IoT nodes;
4. Implement the identified lightweight hash functions; and
5. Evaluate the performance of the identified lightweight hash functions.

To achieve objectives 1 and 2, a detailed literature review was conducted and presented in Chapter 2. The focus was on understanding the limitations of the nodes used in IoT applications; latency and memory were identified as performance measures for the software implementation of lightweight hash functions. The IoT architecture and technologies were highlighted in Chapter 2; the main properties of a secured hash function and the measures for both hardware and software implementation types were discussed. Chapter 2 also addressed objective 3, where the commonly used lightweight cryptographic hash functions were reviewed. This study identified SPONGENT and PHOTON as widely used lightweight hash functions used to ensure data integrity. These two hash functions are standardised in part 5 of ISO/IEC 29192 as lightweight cryptographic hash functions, and their reference implementation code was made public.

Chapter 3 outlines the methods that were followed to answer the research questions. Research equipment and a description of the proposed hash function were addressed. The validation of the proposed hash functions and the comparative method were discussed. The SPONGENT and PHOTON reference implementation codes were obtained from the official designers' websites. Upon downloading the code, the test vectors were generated and compared to those of the designers to validate the hash functions. In addition, the hash functions were executed with the original short message that came with the code to generate the hash values that were compared with those provided by the designers.

Research objective 4 served to detail the experimental setup and procedures used to complete this work, introducing the virtual lab—the FIT IoT LAB made of constrained nodes. The M3 node from the FIT IoT LAB was selected for data collection since it has sensors. The atmospheric pressure and ambient light sensors of the M3 node were used to sense the environment and write the data readings to a text file on the SSH server of the lab where the M3 node is connected. The text file was copied to a local Raspberry Pi 3 Model B running the evaluation of the proposed hash functions. At the beginning of this study, the author planned to use the FIT LAB M3 node for data collection and evaluation of hash functions. However, merging the reference implementation codes of the hash functions and the Contiki-NG code loaded on M3 as firmware did not work. The reference implementation code for SPONGENT is written in C++, for PHOTON it is C, and for Contiki-NG it is C++. To implement and execute the hash functions on the M3 node, we needed to merge the *makefiles* of the implementation code and the Contiki-NG, compile them, and upload them on M3 when starting the data collection experiment. There were limited publicly available guidelines and documentation from the FIT IoT LAB administrator to merge the reference code and Contiki-NG codes and upload them as firmware on the M3 node. It was decided to use the Raspberry Pi 3 Model B to setup the evaluation experiment. The experiment was carried out in two different test cases. Test case 1 analyzed and evaluated the performance of SPONGENT

and PHOTON when the data log file used as input to the hash increases in size from 100 kB to 1 MB. Test case 2 studies how adapting the capacity/input rate affects the performance of hash functions represented as SPONGENT-*n/c/r* and PHOTON-*n-r-r'* when a 500 kB data log file size was used as an input message.

Chapter 5 provided an in-depth analysis of the performance of SPONGENT and PHOTON in terms of latency and memory consumption for test case 1 and test case 2. The latency is the time taken by the hash function to execute the hash method and output the hash value, whereas the memory consumption is the amount of maximum memory utilised by the program (i.e. the hash function) while it is running. The results were presented in Chapter 5, where research objective 5 was achieved, including the discussions of the SPONGENT and PHOTON performance measurements.

6.4 Validation and Verification

Validation is the process of assessing the evaluation of the proposed algorithm to check whether it meets all the requirements. Validation ensures that the model is implemented correctly [84]. This study validates and verifies that the requirements of this study are met. The goal of this study was to evaluate the performance of SPONGENT and PHOTON lightweight hash functions when implemented on a resource-constrained device such as a Raspberry Pi. It was noted that some variants of SPONGENT and PHOTON compromise latency for memory consumption and vice versa. Modifying the capacity/input rate for SPONGENT and PHOTON reduced the overall latency of their variants when compared to the original unmodified variants. However, when the capacity/input rate of the PHOTON variants was 160 and 176 bits, the first two smallest variants, PHOTON-80-20-16 and PHOTON-128-16-16, were incompatible.

Verification ensures that the simulation or the experimental procedures were performed correctly [84]. This study verifies that when the capacity/input parameter was modified, the performance of some variants of the SPONGENT and PHOTON improved. For some variants, high latency does not correlate with lower memory consumption. The improved overall performance did reduce the security levels of some variants of SPONGENT; this employed the trade-off between security and performance. The comparison of lightweight cryptographic hash functions depends on various factors, such as their core design, the technology used for implementation, and the parameterization of the variants, which impacts the performance [60].

6.5 Research Contributions

There is rapid development and adoption of IoT applications. These applications need to be secured, and an evaluation of their performance is necessary. This study updates the state-of-

the-art literature on lightweight cryptography with a focus on the performance of hash functions. Researchers will benefit from knowing how the parameters (specifically capacity/input rate) influence the latency and memory consumption of SPONGENT and PHOTON and how the parameters of other lightweight hash functions can be modified.

SPONGENT and PHOTON are hardware-based hash functions and have been evaluated in hardware-based architectures as indicated by the literature review. The software measurements, i.e., the memory and latency of these selected hash functions, are not fully researched. This gap was addressed by following a software-based measurement approach to evaluate the performance since many IoT devices will be implemented as software solutions.

6.6 Limitations and Future work

The security of the hash function is out of scope. Therefore, the author assumed that the proposed hash functions are secure and do not have any security vulnerabilities or would not introduce any vulnerabilities by adapting the capacity/input rate.

Due to resource limitations, only the Raspberry Pi was used to execute the hash functions. In the future, the evaluation of variants with modified capacity/input rate could be evaluated on different platforms or nodes with more limited resources than the Raspberry Pi.

This study evaluated only the capacity/input rate. Perhaps changing other parameters like the length of the hash value and output rate together with the capacity/input rate could be worked on in the future. Cryptanalysis of the modified variants is another work that can be done.

BIBLIOGRAPHY

[1] L. Da Xu, W. He, and S. Li, “Internet of things in industries: A survey,” *IEEE Trans. Ind. Informatics*, vol. 10, no. 4, pp. 2233–2243, 2014,

[2] L. M. R. Tarouco *et al.*, “Internet of Things in healthcare: Interoperatibility and security issues,” in *IEEE International Conference on Communications*, 2012, pp. 6121–6125.

[3] R. Mahmoud, T. Yousuf, F. Aloul, and I. Zualkernan, “Internet of things (IoT) security: Current Status, Challenges and Prospective Measures,” in *The 10th International Conference for Internet Technology and Secured Transactions (ICITST 2015)*, 2015, pp. 336–341.

[4] A. Banafa, “IoT and Blockchain Convergence: Benefits and Challenges,” *IEEE Internet of Things*, 2017. https://iot.ieee.org/newsletter/january-2017/iot-and-blockchain-convergence-benefits-and-challenges.html (accessed Oct. 17, 2017).

[5] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, “A Survey on Application Layer Protocols for the Internet of Things,” *Trans. IoT Cloud Comput.*, vol. 3, no. 1, pp. 11–17, 2015.

[6] Gartner Inc, “Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020,” 2013. https://www.gartner.com/newsroom/id/2636073 (accessed May 26, 2018).

[7] A. Chaudhuri, “Cyber Threat Mitigation of Wireless Sensor Nodes for Secured,” *Trust. IoT Serv. EDPACS*, vol. 54, no. 1, pp. 1–14, 2016,

[8] ABIresearch, “More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020 | ABI Research,” *ABI Research*, 2013. https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne/ (accessed May 26, 2018).

[9] HP Company, “Internet of Things Research Study,” 2014.

[10] HP Company, “HP News - HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack,” Jul. 29, 2014. https://www.hp.com/us-en/hp-news/press-release.html?id=1744676#.YX6FX55BxPb (accessed Sep. 12, 2017).

[11] I. L. Memon, S. Memon, J. Ahmed, R. Ahmed, and A. Sattar, “FLA-IoT: Virtualization

Enabled Architecture for Heterogeneous Systems in Internet of Things," *Int. J. Adv. Comput. Sci. Appl.*, vol. 11, no. 4, pp. 360–366, 2020,

[12] I. Lee and K. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises," *Bus. Horiz.*, vol. 58, no. 4, pp. 431–440, Jul. 2015,

[13] R. de Oliveira Albuquerque, L. Villalba, A. Orozco, F. Buiati, and T.-H. Kim, "A Layered Trust Information Security Architecture," *Sensors*, vol. 14, no. 12, pp. 22754–22772, Dec. 2014,

[14] S. N. Swamy, D. Jadhav, and N. Kulkarni, "Security threats in the application layer in IOT applications," in *Proceedings of the International Conference on IoT in Social, Mobile, Analytics and Cloud, I-SMAC 2017*, Feb. 2017, pp. 477–480.

[15] K. A. Mckay, L. Bassham, M. S. Turan, and N. Mouha, "NISTIR 8114 Report on Lightweight Cryptography," U.S, 2017.

[16] B. T. Hammad, Y. A. Abbas, N. Jamil, M. E. Rusli, and M. R. Z`aba, "FPGA Implementation of DLP-PHOTON Hash Function," *Int. J. Futur. Gener. Commun. Netw.*, vol. 10, no. 12, pp. 71–78, Dec. 2017,

[17] W. Julian Okello, Q. Liu, F. Ali Siddiqui, and C. Zhang, "A survey of the current state of lightweight cryptography for the Internet of things," in *2017 International Conference on Computer, Information and Telecommunication Systems (CITS)*, Jul. 2017, pp. 292–296.

[18] M. Katagi and S. Moriai, "Lightweight Cryptography for the Internet of Things," 2008.

[19] C. Alippi, A. Bogdanov, and F. Regazzoni, "Lightweight Cryptography for Constrained Devices," in *2014 International Symposium on Integrated Circuits (ISIC)*, 2014, pp. 144–147.

[20] S. T. Patel and N. H. Mistry, "A survey: Lightweight cryptography in WSN," in *2015 International Conference on Communication Networks (ICCN)*, Nov. 2015, pp. 11–15.

[21] S. Kaur and S. Kaur, "Comparative Analysis of Lightweight Cryptography Algorithms for Smart Grids," in *2017 4th International Conference on Signal Processing, Computing and Control (ISPCC)*, Sep. 2017, pp. 564–567.

[22] M. Knežević, V. Nikov, and P. Rombouts, "Low-Latency Encryption – Is 'Lightweight = Light + Wait'?," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7428, no. 258754, Springer,

Berlin, Heidelberg, 2012, pp. 426–446.

[23] A. Biryukov and L. Perrin, “State of the Art in Lightweight Symmetric Cryptography,” Esch-sur-Alzette, 2017.

[24] C. Kothari, *Research Methodology : Methods and Techniques*, Second. New Delhi: New Age International, 2004. Accessed: Jun. 18, 2018. [Online]. Available: http://dspace.utamu.ac.ug:8080/xmlui/bitstream/handle/123456789/181/Research Methodology - Methods and Techniques 2004.pdf?sequence=1

[25] C. McCrindle, “Choosing and Using Quantitative Research Methods and Tools.” University of Pretoria, pp. 1–28. [Online]. Available: https://www.up.ac.za/media/shared/624/choosing-and-using-quantitative-research-methods-and-tools.zp119932.pdf

[26] E. Hofstee, “The Method,” in *Constructing a Good Dissertation: A Practical Guide to Finishing a Masters , MBA or PhD on Schedule*, A. Denniston, Ed. Johannesburg, South Africa: EPE, 2006, p. 128.

[27] I. Andrea, C. Chrysostomou, and G. Hadjichristofi, “Internet of Things: Security vulnerabilities and challenges,” in *The 3rd IEEE ISCC 2015 International Workshop on Smart City and Ubiquitous Computing Applications*, 2015, pp. 180–187.

[28] B. Mostefa, “A survey of Wireless sensor network Security in the context of Internet of Things,” in *2017 4th International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, 2017, pp. 1–8.

[29] F. K. Shaikh, S. Zeadally, and E. Exposito, “Enabling Technologies for Green Internet of Things,” *IEEE Syst. J.*, vol. 11, no. 2, pp. 983–994, Jun. 2017,

[30] J. P. Conti, “The Internet of things,” *Commun. Eng.*, vol. 4, no. 6, pp. 20–25, Dec. 2006,

[31] H. Verma and R. K. Chahal, “A review on security problems and measures of Internet of Things,” in *International Conference on Intelligent Computing and Control Systems ICICCS 2017*, 2017, pp. 71–76.

[32] Q. Jing, A. V. Vasilakos, J. Wan, J. Lu, and D. Qiu, “Security of the Internet of Things: perspectives and challenges,” *Wirel. Networks*, pp. 2481–2501, 2014,

[33] P. M. Mukundan, S. Manayankath, C. Srinivasan, and M. Sethumadhavan, “Hash-One: a lightweight cryptographic hash function,” *IET Inf. Secur.*, vol. 10, no. 5, pp. 225–231, 2016,

[34] D. Navani, S. Jain, and M. S. Nehra, “The Internet of Things (IoT): A Study of Architectural Elements,” in *2017 13th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*, Dec. 2017, pp. 473–478.

[35] S. Al Hinai and A. V. Singh, “Internet of things: Architecture, Security challenges and Solutions,” in *2017 International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions) (ICTUS)*, 2017, pp. 1–4.

[36] B. Ali and A. Awad, “Cyber and Physical Security Vulnerability Assessment for IoT-Based Smart Homes,” *Sensors*, vol. 18, no. 3, p. 817, Mar. 2018,

[37] K. K. Nair, E. Dube, and S. Lefophane, “Modelling an IoT testbed in context with the security vulnerabilities of South Africa,” in *2017 3rd IEEE International Conference on Computer and Communications, ICCC 2017*, 2018, vol. 2018-Janua, pp. 244–248.

[38] OWASP, “OWASP Internet of Things Project,” *Creative Commons Attribution-ShareAlike*, 2018. https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project (accessed May 16, 2018).

[39] S. Li, T. Tryfonas, and H. Li, “The Internet of Things: a security point of view,” *Internet Res.*, vol. 26, no. 2, pp. 337–359, Apr. 2016,

[40] I. Zikratov, A. Kuzmin, V. Akimenko, V. Niculichev, and L. Yalansky, “Ensuring data integrity using blockchain technology,” in *2017 20th Conference of Open Innovations Association (FRUCT)*, Apr. 2017, pp. 534–539.

[41] ITU-T Study Group 20, “Recommendation ITU-T Y.2066 Common requirements of the Internet of things,” Geneva, Switzerland, Edition 1, 2014.

[42] P. Rogaway and T. Shrimpton, “Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance,” in *Fast Software Encryption. FSE 2004. Lecture Notes in Computer Science, vol 3017*, W. Roy, B., Meier, Ed. Berlin, Heidelberg.: Springer, 2004, pp. 371–388.

[43] D. Toz, “Cryptanalysis of Hash Functions, (Ph.D. dissertation), Faculty of Eng. Science,” Leuven, Belgium, 2013. [Online]. Available: https://lirias.kuleuven.be/retrieve/221186

[44] R. Rivest, “The MD5 Message-Digest Algorithm,” Cambridge, USA, RFC1321, Apr. 1992.

[45] V. IMMANUEL and D. CHIDAMBARAM, “Secure Data Access Control for Multi–Authority

Cloud Storage," *Int. J. Emerg. Technol. Comput. Sci. Electron.*, vol. 13, no. 1, pp. 629–631, 2015,

[46] A. P. Jamdar, M. B. Bhangire, S. G. Shahari, and K. G. Matere, "An Efficient Framework for Database Forensic Analysis," *Int. J. Eng. Sci. Comput.*, vol. 7, no. 6, pp. 12634–12637, 2017, Accessed: Sep. 15, 2018. [Online]. Available: http://ijesc.org/

[47] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," in *Advances in Cryptology – CRYPTO 2017*, 2017, pp. 570–596.

[48] A. Joaquim, M. L. Pardal, and M. Correia, "vtTLS: A vulnerability-tolerant communication protocol," in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, Oct. 2016, pp. 212–215.

[49] M. Kumar, "A Light Weight Cryptographic Hash Algorithm for Wireless Sensor Network," *Int. J. Eng. Res. Gen. Sci.*, vol. 4, no. 1, pp. 66–78, 2016, Accessed: Sep. 09, 2018. [Online]. Available: www.ijergs.org

[50] M.-J. O. Saarinen and D. Engels, "A Do-It-All-Cipher for RFID: Design Requirements." Accessed: Sep. 09, 2018. [Online]. Available: https://pdfs.semanticscholar.org/79b0/7edd7dbec199d612fcf534401851cf08fe0b.pdf

[51] Joint Technical Committee ISO/IEC JTC 1, "ISO/IEC 29192-1: Information technology — Security techniques — Lightweight cryptography Part 1 : General," Switzerland, 2012.

[52] B. T. Hammad, N. Jamil, M. E. Rusli, and M. R. Z`aba, "A survey of Lightweight Cryptographic Hash Function," *Int. J. Sci. Eng. Res.*, vol. 8, no. 7, pp. 806–814, 2017, [Online]. Available: https://www.ijser.org/researchpaper/A-survey-of-Lightweight-Cryptographic-Hash-Function.pdf

[53] A. Y. Poschmann, "LIGHTWEIGHT CRYPTOGRAPHY Cryptographic Engineering for a PervasiveWorld," Ruhr-University Bochum, 2009.

[54] T. Mangole, A. S. J. Helberg, and K. K. Nair, "Resource Usage Evaluation of the PHOTON Hash Function," in *2022 Conference on Information Communications Technology and Society (ICTAS)*, Mar. 2022, pp. 1–6.

[55] C. Lachner and S. Dustdar, "A performance evaluation of data protection mechanisms for resource constrained IoT devices," in *Proceedings - 2019 IEEE International Conference on Fog Computing, ICFC 2019*, Jun. 2019, pp. 47–52.

[56] M. El-Haii, M. Chamoun, A. Fadlallah, and A. Serhrouchni, “Analysis of Cryptographic Algorithms on IoT Hardware platforms,” in *2018 2nd Cyber Security in Networking Conference (CSNet)*, Oct. 2018, pp. 1–5.

[57] A. Fotovvat, G. M. E. Rahman, S. S. Vedaei, and K. A. Wahid, “Comparative Performance Analysis of Lightweight Cryptography Algorithms for IoT Sensor Nodes,” *IEEE Internet Things J.*, vol. 8, no. 10, pp. 8279–8290, May 2021,

[58] P. Barreto, V. Nikov, S. Nikova, V. Rijmen, and E. Tischhauser, “Whirlwind: a new cryptographic hash function,” *Des. Codes Cryptogr.*, vol. 56, no. 2–3, pp. 141–162, Aug. 2010,

[59] J. Guo, T. Peyrin, and A. Poschmann, “The PHOTON Family of Lightweight Hash Functions,” *Adv. Cryptol. – CRYPTO 2011*, vol. 6841, pp. 222–239, 2011,

[60] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede, “SPONGENT: The Design Space of Lightweight Cryptographic Hashing,” *IEEE Trans. Comput.*, vol. 62, no. 10, pp. 2041–2053, Oct. 2013,

[61] A. Bogdanov *et al.*, “PRESENT: An Ultra-Lightweight Block Cipher,” in *International workshop on cryptographic hardware and embedded systems*, 2007, pp. 450–466.

[62] A. R. Chowdhury, T. Chatterjee, and S. DasBit, “LOCHA: A Light-weight One-way Cryptographic Hash Algorithm for Wireless Sensor Network,” in *The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014)*, 2014, pp. 497–504.

[63] R. M. Fisher, “INTERNET OF THINGS: SECURE APPLICATION IN INDUSTRIAL WIRELESS SENSOR NETWORKS,” University of Pretoria, 2015.

[64] J. Grossschadl, S. Tillich, C. Rechberger, M. Hofmann, and M. Medwed, “Energy Evaluation of Software Implementations of Block Ciphers under Memory Constraints,” in *2007 Design, Automation & Test in Europe Conference & Exhibition*, Apr. 2007, pp. 1–6.

[65] D. N. Gupta and R. Kumar, “Sponge based Lightweight Cryptographic Hash Functions for IoT Applications,” in *2021 International Conference on Intelligent Technologies (CONIT)*, 2021, pp. 1–5.

[66] T. Meuser, L. Schmidt, and A. Wiesmaier, “Comparing Lightweight Hash Functions-PHOTON & Quark,” 2015. Accessed: May 25, 2022. [Online]. Available:

https://download.hrz.tu-darmstadt.de/pub/FB20/Dekanat/Publikationen/CDC/2015-07-06_TR_PhotonQuark.pdf

[67] M. Saunders, P. Lewis, and A. Thornhill, “Writing and presenting your project report,” in *Research Methods for Business Students*, 5th ed., Pearson Education Limited, 2009, p. 538. [Online]. Available: https://eclass.teicrete.gr/modules/document/file.php/DLH105/Research Methods for Business Students%2C 5th Edition.pdf

[68] P. Chetty, “Limitations and weakness of quantitative research methods,” *Projectg Guru*, 2016. https://www.projectguru.in/publications/limitations-quantitative-research/ (accessed Nov. 28, 2018).

[69] R. Heale and A. Twycross, “Validity and reliability in quantitative studies,” *Evid. Based Nurs.*, vol. 18, no. 3, pp. 66–67, Jul. 2015,

[70] J. Guo, P. Thomas, and A. Poschmann, “The PHOTON Family of Lightweight Hash Functions - full version,” *Adv. Cryptol. – CRYPTO 2011*, vol. 6841, 2011, Accessed: Oct. 22, 2021. [Online]. Available: https://sites.google.com/site/photonhashfunction/downloads/Reference-Implementation.zip?attredirects=0&d=1

[71] S. Surendran, A. Nassef, and B. D. Beheshti, “A Survey of Cryptographic Algorithms for IoT Devices,” in *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, May 2018, pp. 1–8.

[72] L. P. I. Ledwaba, G. P. Hancke, H. S. Venter, and S. J. Isaac, “Performance Costs of Software Cryptography in Securing New-Generation Internet of Energy Endpoint Devices,” *IEEE Access*, vol. 6, pp. 9303–9323, 2018,

[73] M. Fritter, N. Ould-Khessal, S. Fazackerley, and R. Lawrence, “Experimental Evaluation of Hash Function Performance on Embedded Devices,” in *2018 IEEE Canadian Conference on Electrical and Computer Engineering*, 2018, pp. 1–5.

[74] K. Deshpande and P. Singh, “Performance Evaluation of Cryptographic Ciphers on IoT Devices,” in *International Conference on Recent Trends in Computational Engineering and Technologies (ICTRCET)*, Dec. 2018, pp. 1–6. [Online]. Available: http://arxiv.org/abs/1812.02220

[75] N. Nabeel, M. H. Habaebi, N. A. Che Mustapha, and M. R. Islam, “IoT Light Weight (LWT)

Crypto Functions," *Int. J. Interact. Mob. Technol.*, vol. 13, no. 04, p. 117, Apr. 2019,

[76] K. N. Pallavi, V. R. Kumar, and S. Srikrishna, "Comparative Study of Various Lightweight Cryptographic Algorithms for Data Security Between IoT and Cloud," in *2020 5th International Conference on Communication and Electronics Systems (ICCES)*, Jun. 2020, no. Icces, pp. 589–593.

[77] C. Adjih *et al.*, "FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed," in *2015 IEEE 2nd World Forum on Internet of Things (2015 WF-IoT)*, Dec. 2015, pp. 459–464.

[78] A. G. Dludla, A. M. Abu-Mahfouz, C. P. Kruger, and J. S. Isaac, "Wireless Sensor Networks Testbed: ASNTbed," in *2013 IST-Africa Conference and Exhibition*, 2013, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6701766

[79] "M3 Open Node," *FIT IoT Lab*, 2013. https://www.iot-lab.info/hardware/m3/ (accessed Mar. 14, 2020).

[80] M. Borowski, "The sponge construction as a source of secure cryptographic primitives," in *2013 Military Communications and Information Systems Conference (MCC 2013)*, 2013, pp. 1–5.

[81] M. O. A. Al-Shatari, F. A. Hussin, A. A. Aziz, G. Witjaksono, and X. T. Tran, "FPGA-Based Lightweight Hardware Architecture of the PHOTON Hash Function for IoT Edge Devices," *IEEE Access*, vol. 8, pp. 207610–207618, 2020,

[82] Z. A. Al-Odat, E. M. Al-Qtiemat, and S. U. Khan, "An Efficient Lightweight Cryptography Hash Function for Big Data and IoT Applications," in *2020 IEEE Cloud Summit*, Oct. 2020, pp. 66–71.

[83] Y. Su, Y. Gao, O. Kavehei, and D. C. Ranasinghe, "Hash Functions and Benchmarks for Resource Constrained Passive Devices: A Preliminary Study," in *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2019)*, 2019, pp. 1020–1025.

[84] R. G. Sargent, "An introduction to verification and validation of simulation models," in *2013 Winter Simulations Conference (WSC)*, Dec. 2013, pp. 321–327.