Performance-Efficient Cryptographic Primitives in Constrained Devices

Majed Alrowaithy

School of Computing
Newcastle University

This dissertation is submitted for the degree of
Doctor of Philosophy

November 2021

To my father, in loving memory.
To my mother, the origin of my success

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Majed Alrowaithy
November 2021

Acknowledgements

First, I praise God almighty for providing me with the strength, patience and determination and for all the blessings I have received to fulfil my ambition.

Second, I want to express my most sincere gratitude and appreciation to my supervisor, Dr Nigel Thomas, for his advice, guidance, encouragement and crucial insights throughout my PhD journey.

My most profound gratitude goes to my mother for her prayers and support, for my wife, who stands alongside me, sharing both joys and sorrows, and for my four kids, who have been the timeless source of joy and inspiration.

Furthermore, I want to thank my home country Saudi Arabia, the Institute of Public Administration in Riyadh, and the Saudi Arabian Cultural Bureau in London for their support and for granting me a full scholarship to pursue my PhD study.

Finally, I want to thank my friends and colleagues at the School of Computing at Newcastle University for their support and motivation.

Abstract

Resource-constrained devices are small, low-cost, usually fixed function and very limited-resource devices. They are constrained in terms of memory, computational capabilities, communication bandwidth and power. In the last decade, we have seen widespread use of these devices in health care, smart homes and cities, sensor networks, wearables, automotive systems, and other fields. Consequently, there has been an increase in the research activities in the security of these devices, especially in how to design and implement cryptography that meets the devices' extreme resource constraints.

Cryptographic primitives are low-level cryptographic algorithms used to construct security protocols that provide security, authenticity, and integrity of the messages. The building blocks of the primitives, which are built heavily on mathematical theories, are computationally complex and demands considerable computing resources. As a result, most of these primitives are either too large to fit on resource-constrained devices or highly inefficient when implemented on them.

There have been many attempts to address this problem in the literature where cryptography engineers modify conventional primitives into lightweight versions or build new lightweight primitives from scratch. Unfortunately, both solutions suffer from either reduced security, low performance, or high implementation cost.

This thesis investigates the performance of the conventional cryptographic primitives and explores the effect of their different building blocks and design choices on their performance. It also studies the impact of the various implementations approaches and optimisation techniques on their performance. Moreover, it investigates the limitations imposed by the tight processing and storage capabilities in constrained devices in implementing cryptography. Furthermore, it evaluates the performance of many newly designed lightweight cryptographic primitives and investigates the resources required to run them with acceptable performance. The thesis aims to provide an insight into the performance of the cryptographic primitives and the resource needed to run them with acceptable performance. This will help in providing solutions that balance performance, security, and resource requirements for these devices.

The major contributions of this thesis are as follows:

i. Performance evaluation of all cryptographic primitives implemented in .NET cryptographic library and all the symmetric ciphers, hash and MAC functions implemented in BouncyCastle library.

ii. Performance evaluation of all symmetric ciphers, Hash functions and MAC algorithms implemented in Crypto++, Botan, Nettle, Libgcrypt, Libtomcrypt, and OpenSSL cryptographic libraries.

iii. Identifying the different techniques for measuring the energy consumption of a computer program or part of it and using one of these methods to measure the energy consumption of all primitives from the previous bullets.

iv. Introducing a method for measuring the code size of an embedded program in isolation of the system libraries. This method helps in measuring the code size of the cryptographic primitives accurately.

v. Identifying different static and dynamic methods for measuring the maximum stack utilisation and heap allocation in embedded systems. These methods help in determining the RAM footprint of cryptographic primitives.

vi. Performance evaluation of the 32 round-2 candidates from the NIST lightweight cryptography competition. The evaluation included measuring the resource required for running these lightweight primitives in resource-constrained environments.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Introduction

Lightweight cryptographic primitives are cryptographic algorithms tailored for resource-constrained environments. Their main goal is to provide an adequate level of security using the least amount of computing resources with acceptable performance. There are different design choices and implementation approaches when designing lightweight primitives. These design choices must cope with the trade-offs between the primitive's security, performance, and implementation cost. Finding a balance between any two of these properties is not difficult, whereas optimising all the three at the same time is very challenging.

Lightweight primitives are not always built from scratch. On the contrary, most of the primitives we have seen are either built using components from existing ciphers or built as a modification of well-investigated ciphers such as AES, where the complexity of the original primitives is reduced. When building a lightweight primitive, the cryptography engineers use optimisation techniques, including using smaller block sizes and shorter key lengths, using simpler key schedules and round functions and utilising constructions that use elementary operations (binary AND, XOR, etc.) such as ARX paradigm. Since most constrained devices operate on minimal resources and can only afford to devote a small fraction of these resources to security, these optimisation techniques help keep the primitives' need of resources as low as possible and improve the primitives' throughput. However, they can significantly hinder security.

In this thesis, we will start by trying to understand the performance of the conventional cryptographic primitives. First, we will study the effect of the primitives' building blocks, implementation, and optimisation techniques on their performance. To achieve this, we will choose eight cryptographic libraries and evaluate their implementations of symmetric ciphers, hash functions, and MAC algorithms. We will first test the implementations on a powerful computer and later on constrained devices. After that, we will explore the limitations imposed by the resource constraints of the constrained devices in implementing cryptography. Finally, we will study the performance of the 32 round-2 candidates from the NIST lightweight cryptography competition. We will investigate their resource utilisation and their throughput.

1.2 Research Problem

Over the last decade, there has been a shift from using general-purpose computers to small, low power, battery-operated devices. This shift has brought new challenges to implementing cryptography in resource-constrained devices, especially in applying conventional cryptographic standards that are optimised originally for desktops and servers. The limited resources available to constrained devices are not sufficient to perform cryptographic operations that meet acceptable performance. There have been a growing number of research works to address this issue in the past few years. As a result, many new ciphers have been proposed; however, these ciphers suffer from either reduced security, low performance, or high implementation cost.

Our research will investigate, learn and understand the limitations imposed by the limited processing and storage capabilities in constrained devices in implementing security. It will also study the performance of some of the proposed lightweight ciphers and investigate the resources required to run them with acceptable performance. This will help in providing solutions that balance performance, security and resource requirements for these devices.

1.3 Objectives of the Research

1. **Understanding cryptographic primitives and their implementations:**
   We will review the available literature on the cryptographic primitives to understand their different designs, internal structure, implementation, and optimisation techniques.

2. **Investigate the performance of the primitives in the desktop environment:**
   We will test the implementations of different cryptographic schemes, including block ciphers, stream ciphers, hash and MAC functions, and public-key ciphers. For the tests, we will select cryptographic primitives from eight well known cryptographic libraries, which include; `.NET` cryptography, `BouncyCastle, Crypto++, Botan, Nettle`, `Libgcrypt, Libtomcrypt`, and `OpenSSL`. We aim to understand the performance of the different cipher constructions such as key schedules, round functions, and encryption and decryption operations and to study the effect of the implementation approaches and the efficiency of the optimisation technique.

3. **Study the resource-constrained devices:**
   We will review the available literature on resource-constrained devices. We will also study their different classes and the resource constraints each class has, and how these constraints impact cryptography implementation. We will learn how to develop an embedded program and how to communicate with constrained devices to upload our cipher implementation and receive its output. Finally, we will learn how to measure the cipher throughput and analyse its memory and storage usage on these devices.

4. **Study Lightweight Cryptography:**
   We will conduct a literature review to learn about lightweight cryptography and how

it differs from conventional cryptography. We will also study the different design choices that are taken when designing lightweight ciphers. Furthermore, we will examine the impact of these design choices on the performance of the ciphers and their resource requirements. Finally, we will look into the tradeoffs between the security, performance and implementation cost that should be considered when designing or choosing a lightweight cipher.

5. **Investigate the performance of the primitives in a constrained environment:**
We will select a broad range of lightweight cryptographic primitives and implement them on a highly resource-constrained device. We will study the impact of the extreme memory, storage and computing constraints on the performance of the primitives. We will also investigate the resource requirements for running the primitives with acceptable performance.

6. **Understanding energy consumption of the primitives: :**
We will survey the different methods used to measure the energy consumption of a computer program. We will then use the most suitable one to measure the energy consumption of a large number of cryptographic primitives when run in a resource-constrained environment. We aim to understand the effect of the cipher's internal structure, implementation approach and optimisation technique on its energy consumption.

7. **Explore the relationship between security and energy consumption**
We will investigate the effect of a cipher's key length and other security features on its energy consumption.

1.4 Contributions

The work carried out in this PhD research makes several contributions to the field of primitives performance evaluation. The main contributions of this research are as follows:

- Performance evaluation of all cryptographic primitives implemented in .NET cryptographic library and all the symmetric ciphers, hash and MAC functions implemented in BouncyCastle library.

- Performance evaluation of all symmetric ciphers, Hash functions and MAC algorithms implemented in Crypto++, Botan, Nettle, Libgcrypt, Libtomcrypt, and OpenSSL cryptographic libraries.

- Identifying the different techniques for measuring the energy consumption of a computer program or part of it and using one of these methods to measure the energy consumption of all primitives from the previous bullets.
- Introducing a method for measuring the code size of an embedded program in isolation of the system libraries. This method helps in measuring the code size of the cryptographic primitives accurately.
- Identifying different static and dynamic methods for measuring the maximum stack utilisation and heap allocation in embedded systems. These methods help in determining the RAM footprint of cryptographic primitives.
- Performance evaluation of the 32 round-2 candidates from the NIST lightweight cryptography competition. The evaluation included measuring the resource required for running these lightweight primitives in resource-constrained environments.

1.5 Thesis Structure

The thesis is divided into seven chapters, including this Introduction chapter. The following is a list of the thesis's chapters:

- **Chapter 1. Introduction:** This chapter discusses the research problems and outlines the research objectives. It also highlights the main contributions of the research. Furthermore, it gives a brief overview of the thesis chapters. Finally, it lists the publications produced throughout the course of the PhD.
- **Chapter 2. Background and Related Work:** This chapter introduces background information related to the different cryptographic techniques together with the performance benchmarking and the performance metrics. The chapter also discusses Lightweight Cryptography and its various design choices. Furthermore, it reviews the constrained devices, their computational capacities, energy consumption, resources limitations, and the restrictions and limitations they impose on running cryptographic primitives. Finally, the chapter discusses the related work of measuring the performance of cryptographic primitives.
- **Chapter 3. Investigating the Performance of `.NET` and BouncyCastle Cryptographic Libraries:** This chapter introduces the experiment we conducted to measure and evaluate the performance of the different cryptographic primitives implemented in

Microsoft .NET and BouncyCastle cryptographic libraries. The chapter also presents our findings on the results.

- **Chapter 4. Investigating the Performance of C and C++ Cryptographic Libraries:** This chapter presents a study on the performance comparison of several cryptographic primitives from six C/C++ cryptographic libraries to understand the performance cost of the primitives and find the most efficient primitive and library. The chapter also investigates the effect of the key length on the performance of variable key-length primitives.

- **Chapter 5. Performance of C and C++ Cryptographic Libraries under a Constrained Device:** This chapter investigates the performance of several cryptographic primitives when run under a device with limited computing capacity. The chapter aims to provide insight into the performance and energy consumption of the primitives when run under a resource-constrained environment. Additionally, the chapter studies the impact of the key and block size and number of rounds on the power consumption of the primitives. Finally, the chapter discusses three methods for measuring the energy consumption of a program code.

- **Chapter 6. Investigating the Performance of Lightweight Cryptography under Constrained Devices:** This chapter investigates the performance of the 32 round-2 candidates from the NIST lightweight cryptography competition. First, the chapter introduces the experiment we conducted to test the implementation of the candidates and study their throughput and their RAM and ROM footprint. In addition, the chapter describes the building process of an embedded program and explain how to develop and upload a program into constrained devices. Furthermore, it introduces a method to calculate the code size of binaries in isolation from the system libraries. Finally, The chapter describes the different regions of constrained device's memory and discusses static and dynamic analysis methods for investigating the memory usage of embedded software.

- **Chapter 7. Conclusion:** This chapter concludes the research thesis and summarises the main findings of our research. In addition, it will discuss future research directions for work in the area.

1.6 Related Publications

- Majed Alrowaithy and Nigel Thomas. "**Investigating the performance of cryptographic primitives in Microsoft .NET and BouncyCastle libraries.**" 34th UK Performance Engineering Workshop. Newcastle University, 2018
(Chapter 3)

- Majed Alrowaithy and Nigel Thomas. "**Investigating the performance of C and C++ cryptographic libraries.**" Proceedings of the 12th EAI International Conference on Performance Evaluation Methodologies and Tools. 2019.
(Chapter 4)

- Majed Alrowaithy and Nigel Thomas. "**Investigating the performance of lightweight cryptographic primitives in constrained devices**." 36th UK Performance Engineering Workshop. Online, 2020
(Chapter 6)

Chapter 2

Background and Related Work

Summary

This chapter provides an overview of the relevant background information and literature. Sections 2.1.1-2.1.4 discuss the different cryptographic techniques. Section 2.1.1 gives an overview of symmetric cryptography, which includes block ciphers and stream ciphers. This Section discusses the definition of the two types, their different structures, and how they operate. Section 2.1.2 discusses asymmetric cryptography and presents its different underlying computational problems. Section 2.1.3 introduces hash functions and discusses the several requirements for cryptographic hash functions. Message authentication code algorithms are explained in Section 2.1.4. Constrained devices and their classification are presented in Section 2.2.1. Section 2.2 discusses lightweight cryptography and its different design choices. Performance benchmarking and its five key criteria is explained in Section 2.3. Also, in this Section, we discuss microbenchmarks and the effect of compiler optimisation when writing microbenchmarks. Section 2.4 discusses performance metrics. Finally, in Section 2.5, we discuss related work of measuring the performance of cryptographic primitives.

2.1 Cryptography

Cryptography is a critical part of the security system. A system that consists of processes, policies, mechanisms and techniques to protect the data from unauthorised access, modification, disclosure, and destruction. Cryptography is the art and science of keeping data secure [65]. It is built heavily on mathematical theories and computer science practices. Cryptography encompasses mechanisms for exchanging information securely, techniques for ensuring the confidentiality and integrity of data, and protocols for authenticating users and machines. In addition, it guarantees the non-repudiation property where the sender/creator should not be able to falsely deny later his/her intentions in the creation or transmission of the information. In this thesis, we will refer to the cryptographic ciphers and algorithms as cryptographic primitives which include symmetric-key algorithms (block and stream ciphers), public key ciphers, hash and MAC functions, random number generator functions, digital signature algorithms, and others.

In the following sections, we will discuss the most common cryptographic techniques used today.

2.1.1 Symmetric cryptography

In symmetric cryptography, the sender and the receiver agree on a shared secret key. This single key is used for both encryption and decryption of the messages. A key is a string of characters that specify the transformation of a clear message (plaintext) into encrypted one (ciphertext), and vice versa. Keys have variable lengths which have a significant impact on the security and the performance of the cryptographic cipher. Longer keys improve the security but negatively impact the performance. The common key sizes that we have seen in our experiments are 128-bit (16 bytes), 192-bit (24 bytes), and 256-bit (32 bytes). Keys are usually generated using Random Number Generator (RNG) or Pseudo-Random Number Generator (PRNG) algorithms; both are probabilistic algorithms that produce keys chosen according to some distribution. Sometimes keys are derived from passwords or passphrases using functions such as HMAC Key Derivation Function (HKDF) or Password-Based Key Derivation Function (PBKDF). It is worth mentioning that the randomness of cryptographic keys is crucial for the security of cryptographic applications.

The process of disguising a message m (sometimes called plaintext or cleartext) to hide its substance is called *encryption*. The encryption function takes as input a key k and a message m, (m can be a text, image, audio or video file, or any stream of bits) and produces a ciphertext c.

$$Enc_k(m) = c$$

The reverse process of encryption is called *decryption* where the ciphertext is transformed to its original unencrypted form.

$$Dec_k(c) = m$$

Both operations should use the same key. In addition, encrypting a message and then decrypting its ciphertext should yields the original message. The following equation must hold true:

$$Dec_k(Enc_k(m)) = m$$

Symmetric-key algorithms are divided into two categories. They are as follows;

Block Ciphers

Block ciphers operate on a group of bits called *blocks*. The message is first divided into blocks of a predefined size (*block size*), and then each block is encrypted separately. When the message does not neatly divide into the exact block size (there is usually a short block at the end of the message), we fill up the short block with some regular pattern (null bytes, zeros, etc.). This process is called *padding*, and it must be reversible. The cipher should identify the original message from the padded one and remove the padded values after decryption. In block ciphers, the same message block will always produce the same ciphertext block if we use the exact key. Also, the ciphertext and the plaintext are always the same sizes. The typical block sizes we have seen in our experiments are 64-bits and 128-bits. The block size affects the security and performance (especially memory requirements) of the cipher. For security reasons, block ciphers usually are not used directly. Instead, they are used in what is called "*mode of operation*" [65] which include Cipher Block Chaining (CBC) [59], Electronic Codebook (ECB) [65], Cipher Feedback (CFB)[56], Counter (CTR)[65], and others. The mode of operation describes how to repeatedly apply a block cipher to encrypt or decrypt multiple blocks of data.

The designs of the block ciphers we studied in this thesis use one of the following three structures:

1. **Feistel Structure:**
   According to [148], most block ciphers are Feistel structures. In this structure, the message is divided into two equal-sized halves, L and R. In each round, the cipher Xor L with the results from the round function $f(K_i,R)$ and then swaps L and R, Fig 2.1. The left half L is XORed with the results from the round function $f(K_i,R)$ and then swapped with R, Fig 2.1. The process is repeated a fixed number of times, and the final output is the ciphertext. The main difference between the ciphers that implement the Feistel structure is the nature of the round function f and the number of rounds applied [58].

   The attractive feature of this structure is that decryption requires the same set of operation as encryption. So, we do not have to implement two different algorithms (for encryption and decryption), The Feistel structure makes it much easier to implement both functions together. In fact, in a hardware implementations, the same circuit is used to compute both the encryption and decryption [65]. Feistel structure also reduce the code size required to implement the ciphers to nearly a half as we have seen in lightweight cryptography in Chapter 6. Feistel ciphers' resistance to differential and linear cryptanalysis is another feature. This security strength is tied to their well-built

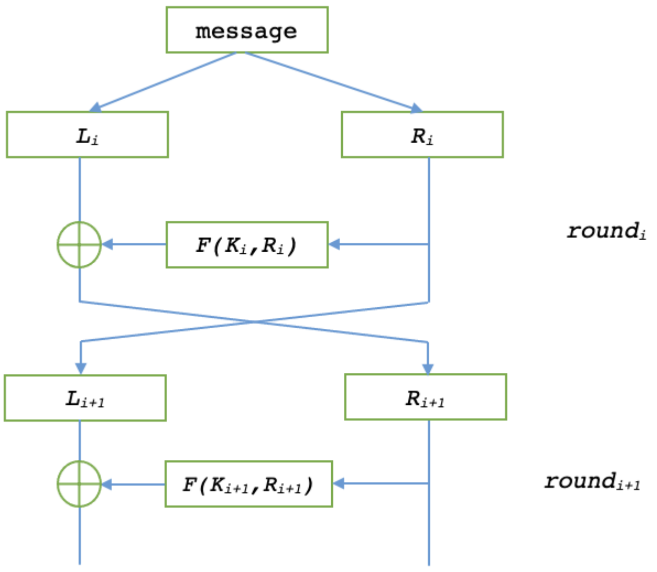


Fig. 2.1 Feistel Structure

substitution box [148]. Some examples of ciphers that use Feistel structure include DES, TwoFish, BlowFish, and TEA.

2. **Substitution-Permutation Networks (SPN)**
Confusion and diffusion are two important security features that Claude Shannon identified for making a secure cipher in his seminal 1945 paper [151]. The two features obscure the relationship between the plaintext, ciphertext and the key, making the linear and differential cryptanalysis very complicated [148]. The confusion property hides the relationship between the ciphertext and the key, making it difficult to find the key from the ciphertext. This is achieved using the **substitution** operation [135]. The diffusion property ensures that changing one bit in the plaintext will change multiple bits in the ciphertext. The **Permutation** operation is used to achieve diffusion. In substitution-permutation networks, ciphers are designed to apply several alternating substitution and permutation steps to provide an adequate level of confusion and diffusion [100]. Substitution involves using an S-box structure that maps an m-bit input to an n-bit output. An S-box is usually implemented using a look-up table [148]. The S-box size (usually referred to as $m \times n$ bit) affects the cipher's security and performance. Larger S-boxes are good for security but have large memory requirements. Permutation operations involve bit-shuffling to permute bits across S-box inputs [58]. Some examples of SPN ciphers include AES, DES, LED, PRESENT, Square, and Shark.

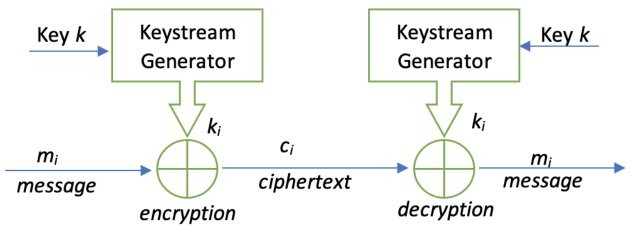


Fig. 2.2 Stream Cipher

3. **ARX Structure:**
   ARX based ciphers use simple operations which include modular Addition, word-wise Rotation, and eXclusive-OR operations [29, 150] and hence the name ARX. Unlike SPN's rounds which consist of substitution and permutations, ARX rounds use modular addition to provide confusion and rotation and XOR to provide diffusion [29]. The ARX simple operations can be efficiently implemented in software and hardware. As a result, ARX-based ciphers are among the best performers identified using FELICS benchmarking project [29, 52]. Some examples of ARX based ciphers include LEA, HIGHT, SPECK, Threefish, and FEAL.

Stream Ciphers

Stream ciphers are symmetric encryption ciphers that operate on a single bit or byte of data at a time. They use a stream of pseudorandom bits (the keystream) that are XOR'ed with the message's bits to generate the ciphertext. The pseudorandom keystream is generated from a random seed value using digital shift registers such as Linear Feedback Shift Registers (LFSRs). There are two types of stream ciphers; synchronous and asynchronous ciphers. In synchronous stream ciphers, the keystream is generated independently of the message or ciphertext, and it depends on the key alone [135]. Whereas in asynchronous ciphers, the keystream generation depends on the key as well as the previous ciphertext's bits [135]. Stream ciphers are typically faster than block ciphers and have lower hardware complexity [75]. The ciphers' bit manipulation operation is often time-consuming if implemented in software, so stream ciphers are more suitable for hardware implementation [148]. The ciphers are often used in applications where the amount of data is either unknown or continuous such as network streams. Another area where these ciphers are often used is in devices where low energy consumption is essential. Stream ciphers beat block ciphers in this aspect [99]. Some examples of stream cipher include RC4, Grain, PANAMA, Rabbit, Salsa20, and Trivium.

2.1.2 Asymmetric cryptography

In asymmetric cryptography (also called public-key cryptography), two keys are used. The public key for encryption and the private key for decryption. Public-key ciphers depend heavily on mathematics and their encryption/ decryption operations are very computationally intensive, which makes them much less efficient (by several orders of magnitude) than symmetric ciphers. Symmetric ciphers can encrypt data at a rate of about 100 to 1000 faster than asymmetric ciphers [135]. For this reason, the public-key ciphers are rarely used for bulk data encryption [148]. Furthermore, some ciphers are limited mathematically in how much data they can encrypt. For example, RSA can only encrypt data to a maximum amount of its key size minus the padding data [135]. In practical systems, public-key ciphers are almost always used in conjunction with symmetric ciphers [148]. In these hybrid cryptosystems, public-key ciphers establish secret keys and then encrypt them before sent over insecure channels. Later, symmetric ciphers use these keys for the actual encryption of the data.

Public-key ciphers are classified into three major categories based on their underlying computational problems [135, 49]. These categories are as follows:

- **Integer Factorisation**: the ciphers' scheme is based on the difficulty of factoring large numbers. RSA is the best-known algorithm in this category.
- **Discrete Logarithm**: the ciphers' scheme here is based on the difficulty of extracting discrete logarithms in finite fields. The ElGamal encryption algorithm, Diffie–Hellman key exchange, and Digital Signature Algorithm (DSA) are examples from this category.
- **Elliptic Curve**: This scheme is based on the algebraic structure of elliptic curves in finite fields. Elliptic Curve Digital Signature Algorithm (ECDSA) and Elliptic-curve Diffie–Hellman (ECDH) are well-known algorithms from this category.

Over the last decades, quite a few public-key ciphers have been proposed. Many of them are insecure, or impractical [148, 135]. For example, some of them have too large key sizes (in the range of megabytes) and some produce ciphertext that is much larger than the plaintext. Of the few secure and practical algorithms, some are only suitable for key generation and distribution (e.g. Diffie–Hellman key exchange). Some are suitable for encryption (e.g. RSA), and others suitable only for digital signature (DSA)[148] .

Unlike symmetric key ciphers, public-key ciphers provide non-repudiation in addition to confidentiality and integrity of the data.

2.1.3 Hashing

Hash functions (also called message digest algorithms) are mathematical functions that take an arbitrary-length input (called pre-image) and deterministically map it to a fixed-length output string [49, 148]. The output string is called a hash value, message digest, or message fingerprint. The typical sizes of hash values are 128-1024 bits [65]. Hash functions are used to preserve the messages' integrity; the sender first computes the hash for the message and then sends the hash value with the message to the recipient who will do the same and compare the hash he received with the hash he calculated. If the two hash values are equal, then the message has not been tampered with. Additionally, hash functions are sometimes used in pseudorandom number generators to generate several private keys from a single passphrase [65].

There are several requirements for a cryptographic hash function, as follows; [148, 58, 135, 49]

1. A cryptographic hash function must be a one-way function. It must be computationally infeasible to derive the original message from its hash value.
2. It must be computationally infeasible to create two different messages that hash to the same value. (called collision resistance property)
3. It must be infeasible to find two different messages that have the same hash values.
4. Hash function must be deterministic; the same message should always have the same hash value.
5. Regardless of the input size, the hash function should always produce a fixed-size hash value.
6. Hash function should be efficiently computable.

There are numerous, widely used hash functions. Examples include MD5, SHA256, SHA512, SHA-3, Whirlpool, and BLAKE3.

2.1.4 Message Authentication Code MAC

Although hash functions serve to guarantee message integrity, they do not solve the problem of message authentication. A malicious adversary can inject a new message or replace the original message and its hash value with new ones, without the receiver detecting that the messages did not originate from the intended sender [49]. To protect against this, we use a

Message Authentication Code (MAC). A MAC function (also called a *keyed hash function*) is a one-way hash function that takes as input a key and a variable-length message and produces a fixed-size MAC value (also called the *tag*) [65]. Upon receiving the message and the MAC value, the recipient verifies, using the same private key, whether the MAC is valid for the message received or not. If not, the recipient discards the message as unauthenticated [49]. Hash functions and MAC functions share the same properties we discussed in the previous section.

MAC functions can be constructed from block ciphers [85] as in CBC-MAC, AES-CMAC, OMAC, PMAC, and GMAC. They can also be built from dedicated hash functions [86] as in HMAC, or from universal hash function [87] as in UMAC, GMAC, VMAC, and Poly1305.

2.2 Cryptography in Constrained Devices

Over the last decade, we have seen a widespread use of Internet of Things IoT devices. IoT solutions are used in several emerging areas such as health care, smart homes and cities, sensor networks, wearables and automotive systems. A typical architecture of IoT solution consists of constrained end nodes (the “Things”) that are connected to gateways or routers which are connected to a cloud platform. Gateways and routers are powerful machines that act as an aggregation point for the connected constrained devices to coordinate their connectivity. Also, they pre-process, secure and route data to cloud servers. On the other hand, constrained devices are very limited in their computing capacities, and because of that, they have brought challenges in implementing security. The conventional cryptographic primitives are designed originally to work on servers and desktop environment which have a good computing capability. Running these primitives on IoT devices will result in either failure or unacceptable performance.

2.2.1 Constrained Devices

Constrained devices are small, low-cost, usually fixed function and very low-resource devices. These devices are constrained in terms of memory, computational capabilities, communication bandwidth and power. They are either battery-powered, or they harvest their energy from the environment. Examples of constrained devices include; RFID tags, wireless sensors, vibration detectors, smart cards and wearable devices.

In 2014, The Internet Engineering Task Force IETF published the RFC 7228 [33] that classifies constrained devices into three classes, as shown in Table 2.1. Class 0 devices are

Table 2.1 Classes of Constrained Devices (KB = 1024 bytes)

Name	Data Size (RAM)	Code Size (ROM)
Class 0	<10 KB	<100 KB
Class 1	$\sim$10 KB	$\sim$100 KB
Class 2	$\sim$50 KB	$\sim$250 KB

severely constrained and do not have the required resources to communicate directly and securely with the Internet. Instead, they access the Internet with the help of larger devices acting as proxies, gateways or servers. These constrained devices generally cannot be secured or managed comprehensively. They are typically pre-configured with a minimal data set.

Class 1 devices are highly constrained and cannot communicate directly with other nodes in the network employing a full protocol stack like HTTP, TLS, and XML based data representation. However, they are capable of running the IoT stack designed for constrained devices such as the Constrained Application Protocol (CoAP), UDP and lightweight security protocols such as DTLS.

Class 2 devices are less constrained than the previous two classes. They are generally capable of supporting most of the protocol stacks used on desktop machines. These devices can significantly benefit from using lightweight and energy-efficient protocols as using fewer resources for network protocols will leave more resources to applications. The RFC 7228 document did not consider constrained devices with capabilities beyond Class 2 as these devices can mainly use the exciting networking protocols unchanged.

2.2.2 Lightweight Cryptography

Cryptographic primitives are usually complex in terms of their internal operations, building blocks, computational overhead and memory usage. The widely used cryptographic primitives are either too large to fit on most of the constrained devices or too expensive to implement or highly inefficient when implemented. So, the demand for cryptographic primitives that can be efficiently implemented on these devices is very strong and growing. Lightweight cryptography is a research field that has developed in recent years to provide cryptographic solutions tailored for implementation in constrained environments.

The motivation for lightweight cryptography is to implement cryptography at a lower cost (less computing resource) with lower power consumption compared to the conventional cryptography. In lightweight cryptography, there is a trade-off between performance, security and cost. Finding a balance between any two is not difficult, whereas it is challenging to optimise all three properties at the same time. A cheap and efficient cipher may have a security drawback (e.g. side-channel attack); a side-channel-resilient and cheap one may have limited performance. Lightweight cryptography's role is to provide an adequate security level using the least amount of computing resources with acceptable performance.

In the past decade, there has been a significant amount of research done on lightweight cryptography; this includes efficient implementations of conventional cryptographic primitives on constrained devices and designing lightweight ciphers from scratch. As a result, a large number of lightweight cryptographic primitives have been proposed. Some of them have been standardised in ISO/ICE 29192 [88]. ISO/ICE 29192 is a standardisation project of lightweight cryptography that specifies lightweight cryptographic ciphers for confidentiality, authentication, identification, non-repudiation, and key exchange.

The two main approaches for implementing lightweight cryptographic primitives are [73, 60, 29]; first, efficiently implement or modify trusted and well investigated conventional ciphers such as AES and DES. DESL [106], a variant of DES, is an example of this approach where the round function uses only one S-box instead of eight as in DES. Second, design new lightweight ciphers from scratch. PRESENT [32], KTANTAN [48], and HIGHT [80] are examples of this approach.

Since most constrained devices operate on very limited resources and can only afford to devote a small fraction of these resources to security, the designers of the lightweight primitives must keep the primitives' need of resources as low as possible. Their design choices must cope with the trade-offs between security, performance and cost. These design choices are as follows [73, 60, 29, 118]:

Smaller block sizes and key sizes

Memory is the most expensive part of the implementation in constrained devices. So, lightweight ciphers usually use smaller block sizes (typically 64 bits or 80 bits) [60] and smaller key sizes (80 bits, 64 bits and less) [60] to save memory. In some implementation, the key is burned into the device to save some space on the memory. Using small block and key sizes has a security drawback and can lead to plaintext and key recovery attacks.

Simpler round functions

Round functions in lightweight primitives have less complexity compared to their counterparts in conventional primitives. For example, the substitution boxes, an essential part of many round functions in symmetric-key ciphers used to obscure the relationship between the key and the ciphertext, are smaller in lightweight primitives. The majority of these primitives use 4×4 S-boxes as opposed to 8×8 in the conventional primitives [29]. The downside of using a smaller S-Box is the inadequate security level we receive compared to a larger S-box. However, the later is costly in both hardware and software implementations. Many cryptographers consider 4×4 S-boxes the right choice that balances security and performance [29].

Some lightweight primitives do not use S-boxes. Instead, they use ARX (Addition, Rotation, XOR) paradigm to provide nonlinearity. The modular addition in ARX is extremely cheap in a software implementation [29]. As a result, the ARX-based lightweight primitives are among the best performers identified using FELICS benchmarking project [52]. Examples of such primitives include LEA [79], HIGHT [80], SPECK [20], and TEA [164].

The designers of lightweight primitives sometimes choose to use a lower number of rounds. Using a fewer rounds can improve the performance of the primitives. However, it has a negative impact on security. Choosing the right number of rounds requires a difficult balance between security and performance.

Simpler key schedule

Complex key schedule functions consume a substantial amount of resources such as RAM and battery. Therefore, lightweight primitives usually use simple key schedules that build the subkeys on the fly using simple operations [60]. These subkeys are generated by extracting bits from a key state which is updated in every round. In their effort to improve the performance, the cryptographic designers keep the update functions as simple as possible [60]. Using simple key schedules has a security drawback and can lead to attacks such as related key attack.

Minimal implementations

Some applications require the constrained device to support only one operation; encryption or decryption. Implementing only one operation will keep the source code small and minimise the resources needed to run the primitive. Hummingbird-2 [61] is an example of a lightweight primitive that adopts this strategy.

2.3 Performance Benchmarking

Benchmarking is a process of evaluating and comparing different systems or different components in a specific domain according to particular characteristics such as performance and security [160]. As discussed in [82, 160, 69], a good benchmark should fulfil five key criteria as follows:

- **Relevance**: the benchmark has to report something important. For the benchmark to be relevant, the metrics used should be meaningful and understandable by the audience. Also, the benchmark usefulness should be relevant for a longer time. Furthermore, the software features and hardware should be used in a way similar to consumer environments. Finally, the benchmark should cover all the relevant system aspects.

- **Repeatability**: This property means if we run the benchmark multiple times under similar conditions and against identical configuration, we should get the same (or nearly the same) results. It is essential to verify this property before trying to draw any conclusion from the benchmark results.

- **Fairness**: All systems and software should be fairly compared.

- **Verifiability**: There should be confidence that the benchmark results represent the actual performance of the system under test. Simple benchmarks are easy to verify (usually, they are self-verifying). However, Complex benchmarks have greater verification requirements, so the results are sometimes reviewed by auditors familiar with the benchmark and the system under test.

- **Cost-Effectiveness**: The cost of designing and running the benchmark should be economical.

Microbenchmark is a benchmark test that measures the performance of a very small unit of a program like a system call to the kernel of an operating system or measuring the time to execute an arithmetic algorithm versus an alternative implementation [132]. Microbenchmark is challenging to write correctly. One of the biggest challenges is the optimisation done by compilers. Compilers regularly try to minimise the program's execution time, code size, power consumption and memory usage using different optimisation techniques such as dead code elimination and loop unrolling [107]. Due to this optimisation, it is common that microbenchmarks give misleading results [134]. The risk is when these misleading results tempt the program developers to optimise the wrong part of the program.

```

using namespace std::chrono;
int main()
{
        unsigned long int f=0;

        auto start = high_resolution_clock::now();
        for(int x = 0; x< 10000; x++) {
                f= factorial(25);
10          }
11          auto end = high_resolution_clock::now();
12          auto duration = duration_cast<microseconds>
13                                      (end - start).count();
14
15          std::cout <<"elapsed time: "<< duration <<" microseconds, ";
16          std::cout<< ((double)duration/10000) <<" us per operation";
17
18          return 0;
19  }
20
21  unsigned long int factorial(int n)
22  {
23          if(n<0) {
24                  return -1;
25          }
26          else{
27                  unsigned int fact=1;
28                  for (int i = 1; i <= n; ++i) {
29                          fact *= i;
30                  }
31                  return fact;
32          }
33  }
```

Listing 2.1 Factorial Microbenchmarking

Listing 2.1 shows a microbenchmark test that measures the performance of a function that calculates the factorial of an integer. An intelligent compiler can figure out that the function inside the loop calculates the factorial for the same number. So it executes the loop only once, giving a misleading benchmarking result. To solve this, we should vary the numbers passed to the function as in line 13 in Listing 2.3.

Some compilers are clever about identifying and eliminating dead code; code that has no effect on the program's execution outcome. Microbenchmark programs do not always generate output which means some (maybe all) of the code can be optimised away. This optimisation will lead to inaccurate performance measurements. In Listing 2.1, the results from the factorial function are never used; the local variable *f* is a dead variable. Hence, a smart compiler will optimise away not only the call to the factorial function (line 9) but also the for loop [134]. The program will end up executing the code in Listing 2.2 instead.

```

using namespace std::chrono;
int main()
```

```
{
 auto start = high_resolution_clock::now();
 auto end = high_resolution_clock::now();
 auto elapsed = duration_cast<microseconds>(end - start).count();

 std::cout<< "elapsed time: " << duration << " microseconds, ";
 std::cout<< ((double)duration/10000) << " us per operation";

 return 0;
}
```

Listing 2.2 Factorial Microbenchmarking after optimising away the dead code

This issue can be solved by ensuring that all results from the code we are benchmarking are used (e.g. printing out the results to the screen) or saving the results in variables declared as *volatile* (see line 9 in Listing 2.3). C, C++, and Java offer *volatile* keyword to prevent their compilers from optimising any code that has to do with variables declared as *volatile* [110, 89].

```
1
2  using namespace std::chrono;
3  int main()
4  {
5          std::random_device rand_dev;
6          std::mt19937 rand_num_gen(rand_dev());
7          std::uniform_int_distribution<int> uniform_int_dist(1,30);
8
9          volatile unsigned long int f=0;
10
11         auto start = high_resolution_clock::now();
12         for(int x = 0; x< 10000; x++) {
13                 f= factorial(uniform_int_dist(rand_num_gen));
14         }
15         auto end = high_resolution_clock::now();
16         auto duration = duration_cast<microseconds>
17                                 (end - start).count();
18
19         std::cout <<"elapsed time: "<< duration <<" microseconds, ";
20         std::cout<< ((double)duration/10000) <<" us per operation";
21
22         return 0;
23 }
24
25 unsigned long int factorial(int n)
26 {
        if(n<0) {
                return -1;
        }
        else{
                unsigned int fact=1;
                for (int i = 1; i <= n; ++i) {
                        fact *= i;
                }
                return fact;
        }
}
```

Listing 2.3 Factorial Microbenchmarking

Other challenges when writing microbenchmark include; the effect of cache memory, compilers implementation and switches, operating system interrupts and schedulers, and garbage collection and JVM warmup (in Java).

2.4 Performance Metrics

To analyse the performance of the cryptographic primitives and investigate how they perform when run on different devices, we should first identify the relevant metrics that we should use. Some of the widely used metrics in the literature include; throughput, latency, power, and energy consumption [118, 167, 138]. Throughput is defined as the amount of data that can be processed (e.g., encrypted, decrypted, hashed, etc.) in a given amount of time. It is expressed as data size (bit, byte, KB, MB, etc.) over time (seconds, millisecond, etc.). Sometimes throughput is expressed as CPU cycles per byte. In many constrained devices, moderate throughput is more desirable than high throughput [118, 138]. The reason for this is that moderate throughput comes at the benefit of decreased power consumption as well as area (circuit size) [138]. Latency, a measure of delay, is especially relevant for real-time applications such as the airbag component in automotive systems. It is expressed as a time period or clock cycles. Power and energy consumption are also relevant metrics because constrained devices are usually battery-powered or harvest their energy from their surroundings. The power consumption metric is mostly used with devices that harvest their power from the environment [118] and is expressed in Watt. On the other hand, energy consumption, the power consumption over a given time, is more suitable for devices with a fixed amount of stored energy (battery-powered) [138]. This metric is calculated by multiplying the power consumption with the execution time of an operation [138]. The energy consumption is expressed in Joules and sometimes in Microjoule per bit.

In addition to the performance metrics, we should also consider the amount of resources required to run the cryptographic primitives. These resources include; ROM, which is used to store the primitive's code, RAM, used to keep the computations' intermediate values, CPU utilisation, and registers.

2.5 Related Work

A decent number of research papers has investigated the performance of cryptographic primitives and their resource requirements under desktop and server environment. For example, Lambrecht et al. in [105] evaluated the performance of several commonly used cryptographic primitives. They conducted a comparative analysis of primitives' implementations using two cryptographic libraries; the Java Cryptography Extension JCE and Cryptix extensions for Java. First, they evaluated the performance of the block ciphers implemented in the two libraries. Their evaluation is based on measuring the time required by each primitive to encrypt and then decrypt a plaintext of size 1137 bytes. This time is used to rank the block ciphers; a short execution time indicates a more efficient primitive. The researchers used a 128-bit encryption key size for all cryptographic primitives except for the ones that require a shorter key size which includes: DES (56 bits), Triple DES (112 bits), and Skipjack (80 bits). Out of the ten block ciphers tested, the study found that IDEA is the most efficient block cipher. The study also demonstrated that the execution time for the same block cipher varies between the two libraries, which means that primitive's implementation significantly impacts its efficiency. Lambrecht et al. also investigated if the primitive's mode of operation has a significant performance advantage. They used The Electronic Code Book (ECB) and Cipher Block Chaining (CBC) modes for this test. As indicated in their paper, neither modes show any substantial performance advantage. The researchers also investigated the performance of two public-key ciphers; RSA and Diffie Hellman. They evaluated the ciphers by measuring their average time to generate public encryption keys of different sizes and encrypt 1,137 bytes of data. The study shows RSA cipher outperforms Diffie Hellman's at all key lengths used in the experiment. Finally, Lambrecht et al. studied the impact of key size on the performance of signing a message and verifying its signature. The study shows that larger keys cause the signing time to increase drastically but have little effect on verification time.

Guillermo et al. in [67] conducted an empirical study to investigate the performance of four encryption ciphers; Rijndael, RC2, DES, and Triple DES. In addition to four message digest algorithms; MD5, SHA1, SHA256, and SHA512. The researchers used two different development environments and programming languages to test the primitives' implementation; MS Visual Studio and C# to test the .NET implementation. In addition to NetBeans

IDE, Java Cryptography Extension (JCE), and Java Cryptography API Cryptix for Java implementation of the primitives. The experiment was conducted on a computer with a 3.0 GHz CPU and 512 MB RAM. The computer runs three different operating systems, Microsoft Windows XP, Fedora Linux 5.0, and Sun Solaris 10. The size of the input data used for the test ranges from 256 MB to 8192 MB. The researchers set out three aims for their experiment as follows:

- To check if the commercial cryptographic APIs provide better performance than the open-source counterparts.
- To check if DES algorithm outperforms other encryption algorithms due to its simplicity
- To check if Rijndael algorithm provides consistent performance in a wide range of computing environment.

The results show that commercial cryptographic APIs (.NET implementation) of the block ciphers outperforms the open-source implementation (Java) under Windows environment. The speed of .NET implementation is almost twice the speed of Java implementation of the same ciphers except for Rijndael, which shows similar performance on both .Net and Java. The authors indicated that this similarity between the two implementations might be due to the flexibility and ease of implementation of Rijndael cipher. The results also show that DES, due to its simplicity, outperforms the rest of the block ciphers tested. However, this is only true for .Net implementation. As for the hash functions, .NET implementations outperform the Java implantations of the same algorithms except for the MD5. In the experiment, the authors tested running the Java implementation of the primitives on Linux to see if the results will be different than the results obtained from running them on Windows. The experiment shows that Java implementation of the block ciphers performs better under Windows platform. However, the results for the hash algorithms are too close to draw a definite conclusion. Lastly, the experiment shows that Rijndael algorithm provides similar performance across multiple frameworks and operating systems.

Montenegro et al. [122] present a study on the power consumption of cryptographic primitives in Android mobiles. The authors test the Java implementations of AES, DESede, RSA, DSA, and six HMAC based algorithms from BouncyCastle, SpongyCastle and IAIK cryptographic providers. The primitives' power consumption is evaluated in two smartphones (HTC and Nexus) with different hardware configurations. The authors first surveyed the available Android energy profiling tools, then decided on using PowerTutor as it provides method-level profiling. The study tries to answer three questions:

- **Q1:** Which cryptographic library provides the most energy-efficient implementation for the ten primitives tested?
- **Q2:** What is the energy cost associated with increasing the security level of the primitives?
- **Q3:** Which primitive configuration is more energy efficient; the default one chosen by the cryptographic providers or the one recommended by the security experts?

Answer to Q1:

- **Key generation:** RSA is the least energy-efficient primitive in terms of key generation. The primitive's implementation provided by IAIK is the least energy-efficient, which quadruples the consumption from the other providers.
- **Encryption/Decryption:** IAIK implementation of AES is the most energy-efficient for the encryption operation. The other two providers show similar performance. However, in the case of the decryption operation, BouncyCastle's implementation is the most energy-efficient, followed by SpongyCastle. IAIK's, on the other hand, show higher energy consumption.

  As for the DESede, SpongyCastle has the most energy-efficient implementation of the encryption operation. Conversely, IAIK's implementation of the decryption operation consumes the least amount of energy.

  The encryption and decryption operations of the RSA consume much less energy compared to the key generation. The most energy-efficient implementation is provided by SpongyCastle for the encryption operation and IAIK for the decryption operation.
- **Sign/Verify:** The cost of signing a message is much higher than the cost of the verification. This is true for both RSA and DSA primitives. SpongyCastle's implementations show a higher energy consumption compared to the other implementations.
- **Generating MAC:** BouncyCastle shows the lowest energy consumption for HMAC with SHA-1, SHA-384 and SHA-512. SpongyCastle shows lower energy consumption for the remaining MAC algorithms. On the other hand, IAIK shows high energy consumption for all MAC algorithms.

Answer to Q2:

- **Key length:** Increasing the AES key length from 128-bit to 192-bit does not incur extra energy consumption. However, this is not true for the increment of the key length from 192-bit to 256-bit. The situation is the same for the DESede primitive.

The cost of increasing the RSA's key length from 1024 to 2048 is about nine times for the IAIK, seven times for the BouncyCastle and six times for the SpongyCastle.

As for the DSA, the energy cost incurred by increasing the key length to 2048 is six times in IAIK and 13 times in SpongyCastle.

- **Encryption/Decryption:** BoucyCastle implementation of AES encryption operation shows the most energy consumption when increasing the key length. IAIK has the most energy-efficient implementation. On the other hand, IAIK's implementation of the decryption operation has the worst energy consumption, while BouncyCastle's implementation has the least energy consumption.

  As for the DESede, increasing the key length increases the energy consumption. All providers show a similar increase in energy consumption for the encryption operation. However, SpongyCastle shows more energy consumption for the decryption operation.

  The energy consumption of the RSA's encryption and decryption operations significantly increases when increasing the key size. SpongyCastle has the worst energy consumption implementation for the encryption operation, while IAIK and BouncyCastle show similar results. However, BouncyCastle's implementation has the worst energy consumption for the decryption operation, and IAIK has the most energy-efficient implementation.

- **Sign/Verify:** Increasing security negatively impacts the power consumption for both the signing and verification operations. When increasing the security options, the most energy-efficient primitives are DSA from IAIK for the signing operation and RSA from IAIK for the verification operation.

- **Generating MAC:** The results show that the recommended MAC algorithms are more expensive in terms of energy consumption than the non recommended ones.

Answer to Q3:

- **Key length:** Changing the default key length to a more secure one increases the energy consumption for the AES and RSA. The AES's implementation from the BouncyCastle and the RSA's implementation from IAIK has the highest increment in energy consumption. As for the DESede, the default key length matches the recommended one.

- **Encryption/Decryption:** changing the default configurations (mode of operation and block size) increases the energy consumption for both encryption and decryption operation. The IAIK implementation of the primitives shows less energy consumption compared to the other implementations.

- **Sign/Verify:** The results show that changing the default configurations of the primitives to more secure ones has a similar cost for signing the message and the verification.

Bingmann in [28], evaluates the performance of 15 different symmetric primitives from eight C++ open-source cryptographic libraries. The experiment is conducted on five different computers that run six distinct Linux distributions. Additionally, four different C++ compilers with different building flag combinations are used. Bingmann tries to test if the CPU attributes, distribution packaging and compiler flags affect the results. From the results obtained, The author notices that although all libraries contain approximately the same core cipher implementation, the performance of the libraries varies significantly due to the different code optimisation. Also, the results do not show any significant variations in the performance of the primitives when conducting the experiment on five computers with different CPUs. The most fragile primitive to the CPU architecture is Serpent, and the most robust one is CAST5. In addition, the results show that Intel's C++ compiler produces the most optimised code for all the tested primitives, followed by Microsoft Visual C++ 8.0 (16.5% slower) and GCC 4.1.2 (17.5% slower). Finally, the results show that Blowfish is the fastest primitive in the test and TDES and Safer+ are the slowest.

Instead of investigating the performance of individual cryptographic primitives, some studies investigated the performance of the cryptographic systems (e.g. secure communication protocols, virtual private networks VPNs) built on top of these primitives [13, 47, 155, 171]. Miranda et al. [121] present a measurement study on the energy consumption of the TLS transactions on a mobile phone. The TLS (Transport Layer Security) is a transport protocol used to establish a secure communication channel between two hosts. TLS uses symmetric ciphers (AES, SEED, ARIA and others), asymmetric ciphers (RSA, DH, ECDH, and others) and hash functions (SHA, HMAC, and others) to provide authentication of the communicating parties, confidentiality and integrity of the data. In this study, the authors run a client on a Nokia N95 mobile and establish TLS connections to different web services such as Google email and Facebook social network over a wireless connection (WLAN) and 3G network. Then, they measure the energy consumption of the TLS transactions using the Nokia Energy Profiler. The authors also configure the TLS to use AES for encryption, RSA and DHE for key generation and management, and SHA1 for digest generation. The results show interesting observations; we summarise them as follows:

- During the handshake phase, the amount of energy consumed varies significantly between the web services. This is due to the length of the server certificate and the round-trip delay, the RTT.
- Transactions over the 3G network have more significant energy consumption compared to those over WLAN.
- The energy required for the data transmission far exceeds the energy needed for the cryptographic primitives.

Zhao and Thomas [172] analyse the performance of a Key Distribution Centre (KDC); A trusted third party that provides shared sessions keys between two parties wishing to communicate securely. The KDC uses Needham-Schroeder's protocol for exchanging private keys securely. Needham-Schroeder's protocol is composed of a private key encryption algorithm and a public key algorithm. The authors use Markovian process algebra PEPA to model and study the performance of the KDC system. The requests and responses between the KDC and the communicating parties are defined as PEPA components, and the whole model is analysed as a Continues Time Markov Chain (CTMC).

In this study, the authors introduce a number of performance measures that can be evaluated, including the average response time of the KDC when serving requests from clients, the average utilisation at the KDC, average queue length, and the throughput. At the KDC, there is a competition for resources that dominate the protocol's performance, so any request for a session key will be queued along with other awaiting requests. If the scheduling algorithm used is first come, first served, then the service of any incoming request will depend on the service of all requests arrived earlier, the rate of the service, and the number of available servers.

In this study, the authors use the PEPA model to answer the following questions;

1. how many clients can a given KDC configuration support before its performance starts to degrade?
2. how much service capacity must be provided at a KDC to satisfy a given number of clients?

To answer the first question, The authors fix the rate at which the KDC issues new keys and the number of available servers. Then try to find the largest population size before the performance begins to degrade. To answer the second question, they fix the population size and service rate to find the optimal number of servers required.

The rate at which the user requests new keys (rekeying rate) and the service rate (the key issue rate) are significant in studying the trade-off between performance and security in KDC.

When varying these two rates, we can evaluate the average response time, the utilisation, and the security. The authors illustrate that increasing the security (by increasing rekeying rate and the key issue rate) will negatively impact the performance (increase response time and utilisation).

2.6 Conclusion

In this chapter, we introduced the essential background information related to cryptography and benchmarking. The topics covered in this chapter also include the four cryptographic schemes, including symmetric and asymmetric ciphers, hash and MAC algorithms. In addition to their design, usage and performance. We reviewed the constrained devices and thoroughly discussed lightweight cryptography and the different design choices taken by cryptographers. Furthermore, we studied the performance benchmarking, the criteria a good benchmark should fulfil, and some issues related to microbenchmarks and performance metrics. Finally, we reviewed the literature on the performance evaluation of the cryptographic primitives.

The surveyed research has several shortcomings, which we try to address in this dissertation. First, there is very limited experimentation conducted on the performance of cryptographic primitives on constrained devices and investigating their resource utilisation. Also, there is a lack of in-depth analysis of the effect of the different building blocks of primitives on their performance. Furthermore, there is a lack of a systematic approach to assess primitives' performance in small scale systems.

This thesis aims to advance the current state of the art by providing a methodical approach to assess resource utilisation in constrained devices. In addition, it seeks to provide extensive precious results on the performance of a wide range of lightweight and conventional cryptographic primitives. Furthermore, it studies the effect of primitives building blocks such as the key and block sizes, round functions, substitution boxes, etc., on their performance. The thesis also reviews the different techniques for measuring the energy consumption and memory utilisation of primitives.

There are still research gaps remaining, including studying the performance of asymmetric primitives on constrained devices, in-depth analysis of the performance of the hardware implementation of the primitives, the performance of cryptography on ultra-constrained devices such as RFID, and designing simple and efficient lightweight primitives building blocks.

Chapter 3

Investigating the Performance of Cryptographic Primitives in .NET and BouncyCastle

Summary

Security and performance are the two most important properties that decide the cryptographic primitives' choice when designing a secure system. There is usually a trade-off between them. To find an optimal trade-off means that we need to measure both using established measurement metrics and find a balance between them. This chapter presents a study on the performance comparison of several cryptographic primitives from `.NET` [119] and BouncyCastle [35] libraries aiming to find the most efficient primitive. The results show that `.NET`'s AES is by far the most efficient symmetric primitive tested. The results also show that `.NET`'s SHA-1 is the fastest hash function tested. Both primitives are about three times faster than the next fastest primitive in their category. HMACMD5 and RSA from `.NET` library are the most efficient MAC and asymmetric primitive respectively. `.NET` provides a high level of optimisation for almost all the primitives tested. The library, however, is very limited in the number of primitives it implements. BouncyCastle library, on the other hand, provides a broader range of cryptographic primitives. However, the C# version of the library is not efficient enough compared to `.NET` and suffers from poor documentation and examples.

3.1 Introduction

The high demand for data security has led to a rapid increase in the number of cryptographic tools. The search for the best tool, in terms of efficiency and level of security, is an active subject in the security communities. Although the emphasis has been put primarily on the level of security that each tool provides [28], there is a decent number of research papers on the efficacy of cryptographic techniques and primitives. Performance comparison of cryptographic tools, however, has received less attention from researchers [67]. This chapter investigates the performance of the cryptographic primitives implemented by Microsoft® under Cryptography namespace and by BouncyCastle C# library and presents a fair comparison between them. Primitives from four different cryptographic techniques are tested, which include; from symmetric cryptography: AES, DES, Triple DES, Blowfish, Twofish, IDEA, RC2, RC6, Camellia, Skipjack, and many others. From asymmetric cryptography: RSA. From hashing: SHA-1, SHA-2, SHA-3, RipeMD family, Tiger, Shake, SM3, and many others. Lastly, from message authentication code (MAC), HMAC, Poly1305, SkeinMac, CMAC, VmpcMac, and many others. The aim of the study is to investigate the performance cost of different cryptographic primitives under different settings and to conduct a performance comparison between them based on their execution times. The results will help in identifying the most suitable cryptographic primitives for applications with real-time constraints.

3.2 Cryptographic Libraries

There are very few C# cryptographic libraries. Most of what is available are either very limited in the number of implemented ciphers or dedicated to one type of cryptographic primitives such as Hash functions. `.NET` cryptography and BouncyCastle libraries are both well respected and mature cryptographic libraries that provide implementations for a good range of cryptographic primitives. The following two sections will give an overview of the two libraries.

3.2.1 `.NET` cryptography

Microsoft `.Net` offers implementations for several cryptographic primitives. All `.NET` cryptographic classes are located in the `System.Security.Cryptography` namespace. Under this namespace, there are specialized namespaces for various cryptographic primitives, including symmetric and asymmetric primitives, hash functions, key generation algorithms, key exchange algorithms, digital signatures algorithms, and random number generators. The base class for all symmetric primitives is `SymmetricAlgorithm` class, which is inherited by the following cryptographic primitives (abstract classes): AES, DES, RC2, Rijndael, and TripleDES. `AsymmetricAlgorithm` class is the base class for all asymmetric primitives. This class is inherited by RSA, Digital Signature Algorithm (DSA), Elliptic Curve Diffie-Hellman (ECDH), and Elliptic Curve Digital Signature Algorithm (ECDSA). The base for all hash functions is `HashAlgorithm` class which is inherited by SHA-1, SHA-256, SHA-384, SHA-512, RipeMD-160. In addition to KeyedHashAlgorithm for hash-based MACs (HMAC).

There are three different implementations for each primitive; First, Managed Code, where classes are written entirely in managed code (code whose execution is managed by Common Language Runtime, CLR). These classes have “Managed” as part of their names (e.g. RijndaelManaged). This implementation is available on all platforms that support the .NET Framework. It is, however, not certified by the US Federal Information Processing Standards (FIPS) and may be slower than the other two implementations [119]. The second implementation is a wrapper around the Windows Cryptography API (CAPI) implementation of primitives. These classes have "CryptoServiceProvider" as part of their names (e.g. RSACryptoServiceProvider). The CAPI implementation is available on older operating systems and is no longer being developed [119]. The third implementation is a wrapper around the Cryptography Next Generation (CNG), which is replacing the old CAPI. These classes have "CNG" as part of their names (e.g. AesCng). CNG implementation is available on Windows Vista and later.

3.2.2 BouncyCastle cryptography

BouncyCastle library is a collection of cryptographic APIs for both Java and C#. The library includes several block and stream primitives, public key algorithms, signature algorithms, key agreement protocols, hash and MAC functions, and random number generators. The cryptographic primitives were initially implemented in java, and later in 2006, C# APIs were added by the library designers. The current library version is 1.8.10, which was released on 16 February 2021.

3.3 Experiment setup

The performance of the cryptographic primitives is evaluated based on their execution time. The execution time includes the time for key generation (for public-key algorithms only), cipher initialisation, and encryption, followed by decryption of a buffer with a specific size. Inputs of sizes 1 MB, 4 MB, and 8 MB are used to test if the primitives provide consistent performance under different data loads. All tests are done on a machine running Windows® 10 professional on a virtual machine that has 4 gigabytes of RAM and 3.3 GHz Intel® Core™ i7-6567U (Skylake) CPU with a dual independent processor and 4 MB shared level 3 cache. All the experiments are repeated many times with negligible variance in the results. Microsoft Visual Studio® 2017 and .NET framework® ver. 4.6.1. are used to test the implementations of the primitives. We used BouncyCastle C# library version 1.8.4 (released in October 2018) to the implementation from The Legion of the Bouncy Castle®. BenchmarkDotNet library ver. 0.11.2 is used for benchmarking.

3.4 BenchmarkDotNet Library

BenchmarkDotNet [55] is a .NET open-source library that helps design and execute benchmarks and analyse the performance results. The library allows its users to compare different runtimes (e.g. .NET Framework, .NET Core, Mono and CoreRT) and check the performance difference between different processors architectures. The library uses Perfolizer statistical engine [4] to achieve high measurement precision. At the end of the experiment, the library presents a summary that shows a lot of useful statistical information, such as mean, median, standard deviation, minimum and maximum values, confidence interval and others. The summary also contains diagnostic information about the memory and information about the test environment. One of the features we found helpful is the automatic selection of the ideal number of iterations that the library does based on statistical metrics' values. Choosing the right number of iterations increases the measurement precision and saves time by reducing the total duration of the benchmark run. The library has few drawbacks. One of them is that it only supports C#, Visual Basic and F# languages.

Using the BenchmarkDotNet library is a straightforward and easy task; see Figure 3.1. First, we have to add to our project the two namespaces `BenchmarkDotNet.Attributes` and `BenchmarkDotNet.Running`. Second, we mark the methods we want to benchmark with `Benchmark` attribute. The library offers several *attributes* that helps with deep performance investigation. For a comprehensive list of these attributes, readers are advised to refer to [55]. Finally, we use `BenchmarkRunner.Run` method to run our code as in Figure 3.1.

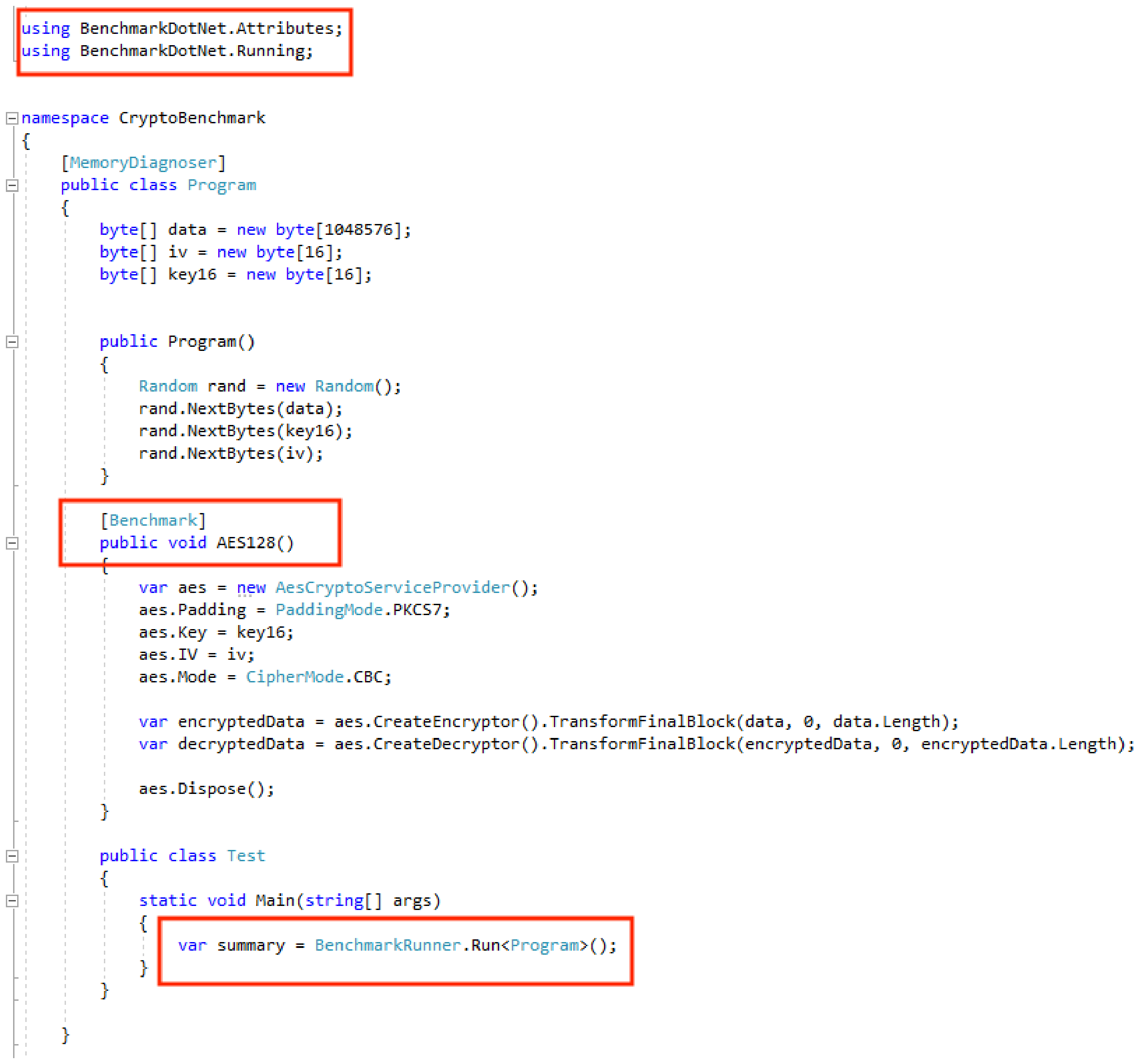
The library shows the results in a summary table (see Figure 3.2), displaying the essential statistics only. Users can add custom columns to this table using the attributes that the library provides. In addition to this table, the library shows all the available statistics for the functions under investigation; see Figure 3.3.

3.5 Comparative analysis of algorithms

The execution time of each primitive is used to calculate its throughput in megabyte per second. The throughput is then used to compare the performance of the primitive's implementation with its counterpart from the other library. The results are presented in the following five sections.

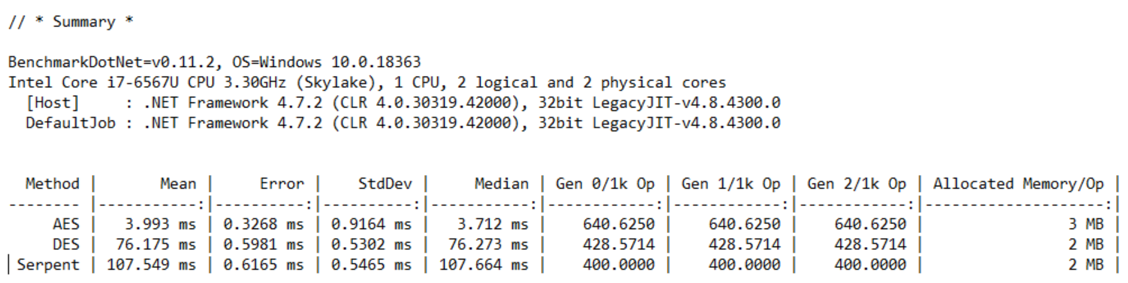
3.5.1 Block Ciphers

Block ciphers operate on fixed-size blocks of data, rather than one bit at a time as in stream ciphers. `.Net` provides implementations for a limited number of block ciphers, including



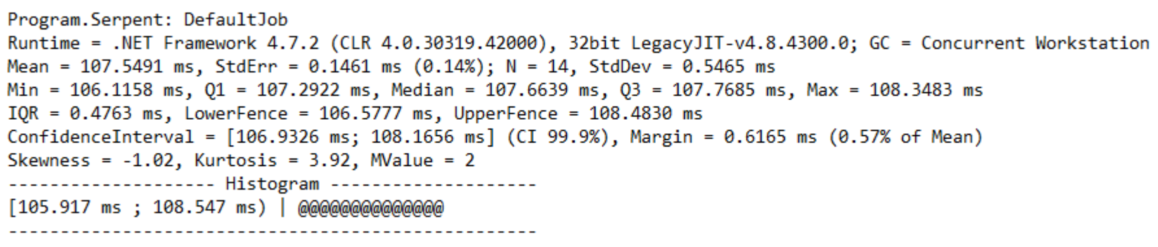
```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

namespace CryptoBenchmark
{
    [MemoryDiagnoser]
    public class Program
    {
        byte[] data = new byte[1048576];
        byte[] iv = new byte[16];
        byte[] key16 = new byte[16];


        public Program()
        {
            Random rand = new Random();
            rand.NextBytes(data);
            rand.NextBytes(key16);
            rand.NextBytes(iv);
        }

        [Benchmark]
        public void AES128()
        {
            var aes = new AesCryptoServiceProvider();
            aes.Padding = PaddingMode.PKCS7;
            aes.Key = key16;
            aes.IV = iv;
            aes.Mode = CipherMode.CBC;

            var encryptedData = aes.CreateEncryptor().TransformFinalBlock(data, 0, data.Length);
            var decryptedData = aes.CreateDecryptor().TransformFinalBlock(encryptedData, 0, encryptedData.Length);

            aes.Dispose();
        }

        public class Test
        {
            static void Main(string[] args)
            {
                var summary = BenchmarkRunner.Run<Program>();
            }
        }

    }
```

Fig. 3.1 Using BenchmarkDotNet library



```
// * Summary *

BenchmarkDotNet=v0.11.2, OS=Windows 10.0.18363
Intel Core i7-6567U CPU 3.30GHz (Skylake), 1 CPU, 2 logical and 2 physical cores
  [Host]     : .NET Framework 4.7.2 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.8.4300.0
  DefaultJob : .NET Framework 4.7.2 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.8.4300.0


  Method |       Mean |      Error |     StdDev |     Median | Gen 0/1k Op | Gen 1/1k Op | Gen 2/1k Op | Allocated Memory/Op |
-------- |-----------:|-----------:|-----------:|-----------:|------------:|------------:|------------:|--------------------:|
     AES |   3.993 ms | 0.3268 ms | 0.9164 ms |   3.712 ms |    640.6250 |    640.6250 |    640.6250 |                3 MB |
     DES |  76.175 ms | 0.5981 ms | 0.5302 ms |  76.273 ms |    428.5714 |    428.5714 |    428.5714 |                2 MB |
| Serpent | 107.549 ms | 0.6165 ms | 0.5465 ms | 107.664 ms |    400.0000 |    400.0000 |    400.0000 |                2 MB |
```

Fig. 3.2 Summary table from BenchmarkDotNet



```
Program.Serpent: DefaultJob
Runtime = .NET Framework 4.7.2 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.8.4300.0; GC = Concurrent Workstation
Mean = 107.5491 ms, StdErr = 0.1461 ms (0.14%); N = 14, StdDev = 0.5465 ms
Min = 106.1158 ms, Q1 = 107.2922 ms, Median = 107.6639 ms, Q3 = 107.7685 ms, Max = 108.3483 ms
IQR = 0.4763 ms, LowerFence = 106.5777 ms, UpperFence = 108.4830 ms
ConfidenceInterval = [106.9326 ms; 108.1656 ms] (CI 99.9%), Margin = 0.6165 ms (0.57% of Mean)
Skewness = -1.02, Kurtosis = 3.92, MValue = 2
-------------------- Histogram --------------------
[105.917 ms ; 108.547 ms) | @@@@@@@@@@@@@@
---------------------------------------------------
```

Fig. 3.3 Statistics from BenchmarkDotNet for Serpent function

AES, DES, Triple DES, and RC2. The BouncyCastle C# library, on the other hand, provides implementations for 20 block ciphers which include the well known and some of the less frequently-used ciphers. The two libraries provide implementations for the different cipher modes of operation. However, for this experiment, we only test the Cipher Block Chaining (CBC) mode. The results from the experiment are shown in Figures 3.4 - 3.6. For clarity, AES is split form the rest of the common ciphers and drawn in separate charts in Figure 3.4 and 3.5.

AES is one of, if not the most commonly used symmetric block ciphers today. The cipher was ratified as a standard by the National Institute of Standards and Technology (NIST) of the United States On November 26, 2001 [57]. NIST also started a Cryptographic Algorithm Validation Program (CAVP) for the cipher implementers to test their AES implementations for conformance to FIPS 197 (standards for AES implementation developed by the NIST). As of 2020, CAVP had validated over 5700 different AES implementations as conforming to FIPS 197 specifications [130]. AES has a fixed block size of 128 bits and supports keys of sizes 128-bit, 192-bit, and 256-bit. The key's size dictates its number of rounds; 10 rounds for a 128-bit key, 12 for a 192-bit key, and 14 for a 256-bit key. A small number of rounds usually work favourably with performance.

The widespread use of the AES cipher has led the processors' manufacturers such as Intel, AMD, and ARM to add encryption instruction set to their CPUs instructions to improve the cipher's performance and increase its security by making it more resilient to side-channel attacks. In our experiments, we found that all the AES implementations that are optimised to take advantage of Intel's AES-NI encryption instruction set show significant performance improvements over the completely software implementations.

`.NET` offers AES-NI implementation while BouncyCastle only provides a native implementation of AES. Due to this, the results show a big difference in their performance, as shown in Figure 3.4 and Figure 3.5. `.NET's` implementation is about 13 times faster than BouncyCastle's. In fact, `.NET's` AES is by far the fastest symmetric primitive in our test. Figure 3.4 and Figure 3.5 also show the impact of the key size on the performance. AES of

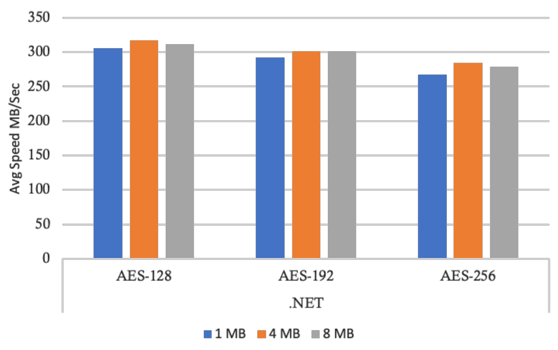


Fig. 3.4 Average speeds of .NET AES implementations.

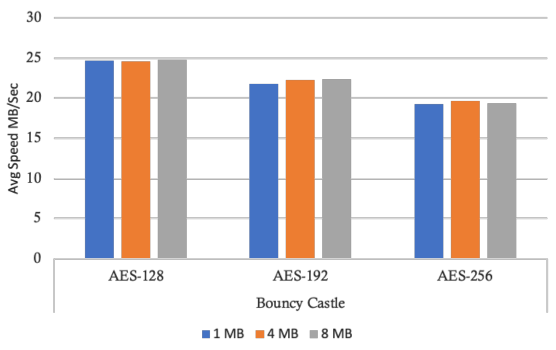


Fig. 3.5 Average speeds of BouncyCastle AES implementations.

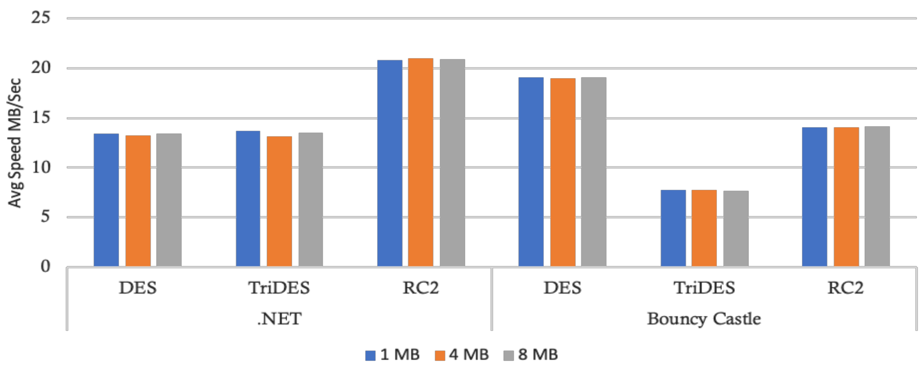


Fig. 3.6 Average speeds of the rest of the common block ciphers implementations.

128-bit key uses only ten rounds of scrambling operations compared to 14 rounds for 256-bit, so one might think that 128-bit AES should be about 40% faster. The results, however, show that the actual cost is not exactly linear in the number of rounds. The difference between 128-bit and 256-bit AES is only about 20% in BouncyCastle and $\approx$ 10% in `.NET`. The two figures also show that the cipher maintains a consistent speed for the three buffer sizes used. This consistency holds for all the tested primitives.

Figure 3.6 shows the results from testing the remaining common ciphers. RC2 from `.NET` is about 48% faster than its counterpart from BouncyCastle. The has a block of size 64-bit and consists of 18 rounds. It is based on a Feistel-like structure. DES, which consists of 16 rounds, has a block size of 64-bit, and a small key size of 64 bits only, is about 42% faster in BouncyCastle. One might think that DES is about three times faster than Triple DES. This is because Triple DES applies DES cipher three times to each data block. However, the results show that DES is only one and a half faster in BouncyCastle and surprisingly about the same in `.NET`. `.NET's` Triple-DES is 74% faster than BouncyCastle's.

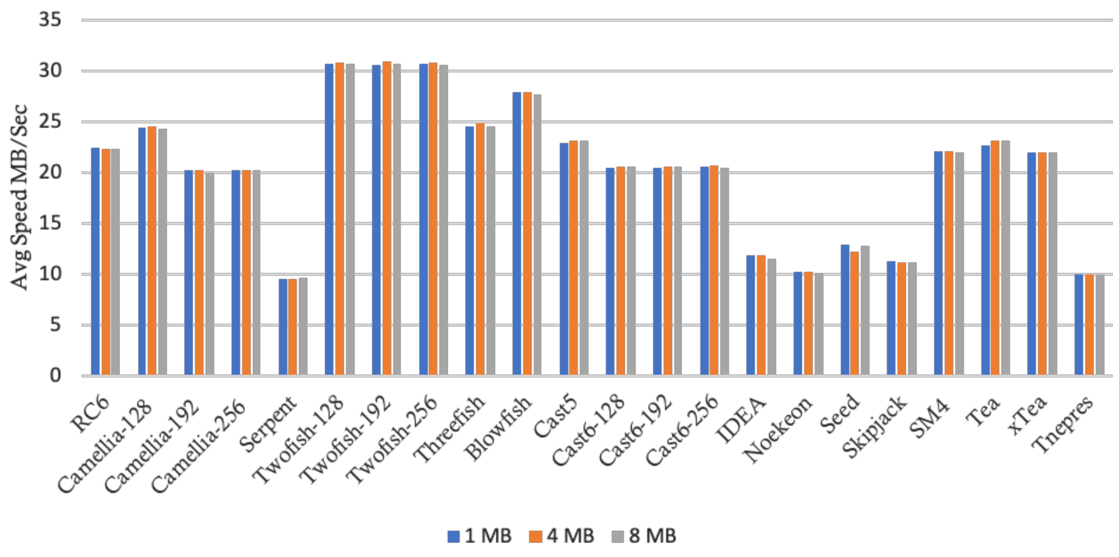


Fig. 3.7 Average speeds of the remaining BouncyCastle block ciphers implementations.

BouncyCastle provides more ciphers implementations than `.NET`. The results of testing the remaining block ciphers that are only found in BouncyCastle are shown in Figure 3.7. As can be seen in Figure 3.7, Twofish is the fastest block cipher in BouncyCastle at an average speed of $\approx$ 31 MB/s. The cipher has the same block and key size as AES. However, The number of rounds, 16, is higher than AES. Twofish has the same number of rounds for all the key sizes, making the cipher maintains a consistent speed for the three key sizes. Blowfish comes second at average speed $\approx$ 28 MB/s. The cipher is based on Feistel structure and has a smaller block size of 64-bit. It has 16 encryption rounds and support keys of size 32-448 bits [147]. The cipher has a very slow key set-up [112]. Threefish and Camellia of 128-bit key size come third at an average speed of $\approx$ 25 MB/s. Threefish supports keys of sizes 256-bit, 512-bit, and 1024-bit. The cipher's number of rounds is 80 rounds for the 1024-bit key and 72 rounds for both 256-bit and 512-bit keys [64]. Camellia is based on a Feistel structure and has the same block size as AES. It supports keys of sizes 128, 192, and 256 bits. However, the number of rounds is different than AES; 18 rounds for 128-bit key and 24 rounds for both 192 and 256-bit keys [6]. The cipher can be accelerated using hardware acceleration techniques [98]. However, these techniques are not used in C# BouncyCastle. Cast5, RC6, SM4, TEA, and XTEA ciphers have similar performance at an average speed of $\approx$ 22 MB/s. Serpent, IDEA, Noekeon, Seed, Skipjack, and Tnepres also have similar performance at an average speed of $\approx$ 10-13 MB/s. The slowest block cipher tested is Triple-DES from BouncyCastle at an average speed of $\approx$ 7 MB/s.

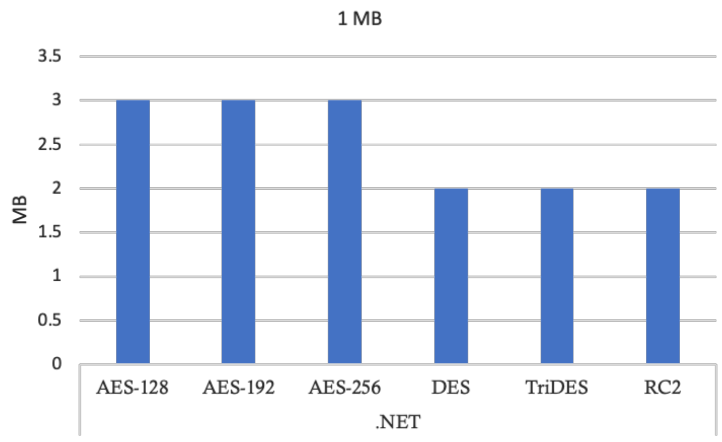


Fig. 3.8 Amount of memory utilised by `.NET`'s block ciphers for every 1 MB of data processed.

Figure 3.8 shows the amount of memory utilised by `.NET`'s block ciphers implementations for every 1 MB of data encrypted and decrypted. Comparing these results with BouncyCastle's in Figure 3.9 shows that `.NET`'s AES implementation uses more memory, 3MB, than in BouncyCastle, 2MB. On the other hand, BouncyCastle's Triple-DES implementation utilises 7 MB for processing 1 MB of data compared to 2 MB in `.NET`'s. BouncyCastle's CAST5 and CAST6 implementations also consume a large amount of memory for processing a small amount of data. RC2 and DES implementations utilise the same amount of memory on both libraries.

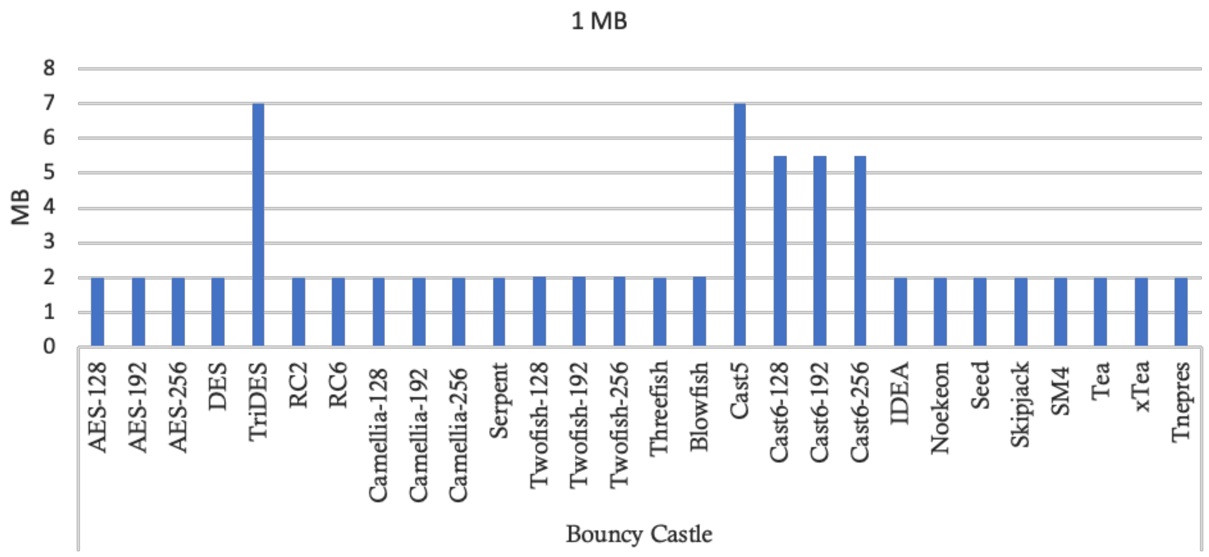


Fig. 3.9 Amount of memory utilised by BouncyCastle's block ciphers for every 1 MB of data processed.

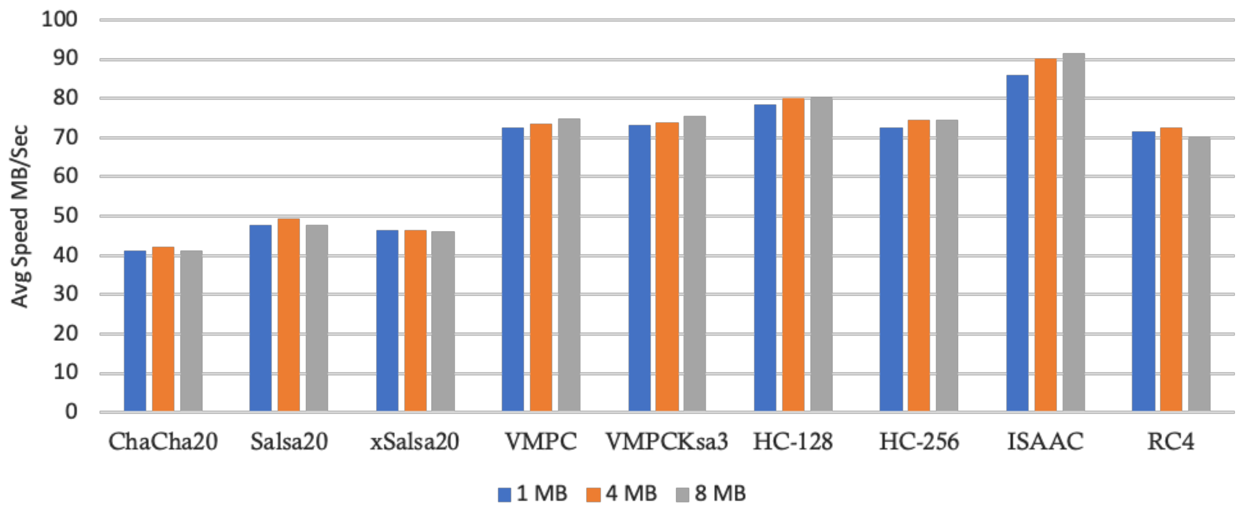


Fig. 3.10 Average speeds of BouncyCastle stream ciphers implementations.

3.5.2 Stream Ciphers

Stream ciphers are symmetric-key encryption ciphers that encrypt a single bit or byte (instead of blocks) of data at a time. Stream ciphers, in general, are faster than block ciphers [75]. `.NET` doesn't provide any implementation for any of the stream ciphers. On the other hand, BouncyCastle provides implementations for six stream ciphers in addition to their variations. Figure 3.10 shows the results of testing these ciphers. ISAAC is the fastest cipher at an average speed of $\approx$ 90 MB/s followed by VMPC and HC, which have similar performance at average speeds of $\approx$ 75-80 MB/s. RC4 also achieves similar performance at an average speed of $\approx$ 71 MB/s. RC4 was one of the most widely used stream ciphers [161]; however, it has some vulnerabilities and no longer considered secure [137]. Nevertheless, several attempts have been made to strengthen it [170]. Salsa and xSalsa come next with similar performance at an average speed of $\approx$ 47 MB/s. ChaCha20 is the slowest cipher at an average speed of $\approx$ 41 MB/s. The cipher is a modification of the Salsa20 cipher and uses 20 rounds and supports keys of size 128 and 256 bits [24].

The amount of memory utilised by the stream ciphers for every 1 MB processed is shown in Figure 3.11. Although ISAAC is the fastest stream cipher, it is inefficient in term of memory utilisation. The cipher consumes 4 MB of memory for every 1 MB of data. The rest of the stream cipher utilise the same amount of memory, 2MB, for every 1 MB of data.

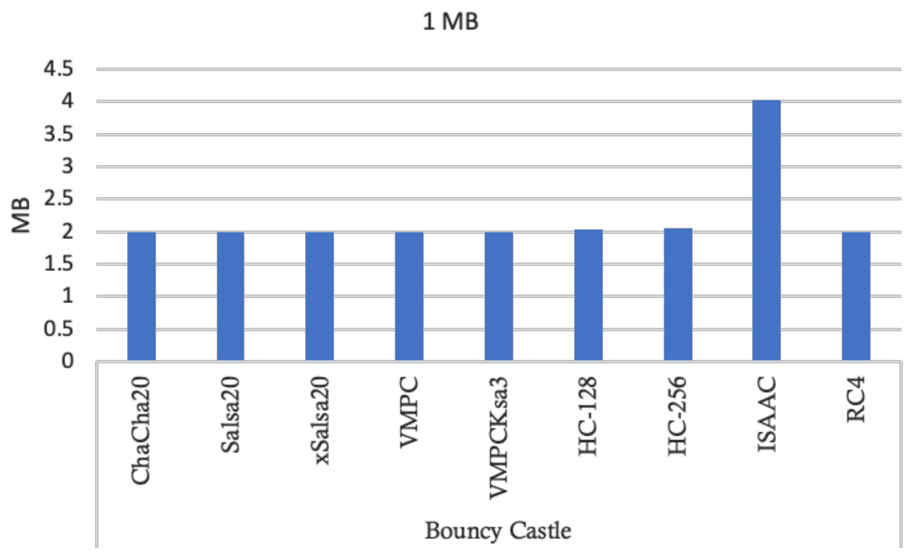


Fig. 3.11 Amount of memory utilised by BouncyCastle's stream ciphers for every 1 MB of data processed.

3.5.3 Asymmetric Algorithms

Asymmetric algorithms use two mathematically linked keys; a public key for encryption and a private key for decryption. Examples of asymmetric algorithms include RSA, ElGamal, Diffie–Hellman key exchange, Elliptic Curve Digital Signature ECDSA. `.NET` provides implementations for RSA for encryption and signing, Digital Signature Algorithm DSA and ECDSA for singing, and Elliptic-curve Diffie–Hellman ECDH for key generation. BouncyCastle provides implementations for RSA, ElGamal, DSA, ECDSA, and Naccache–Stern. In our experiment, we only test the performance of RSA. We calculate the average time it takes RSA to generate keys of sizes 1024-bit (minimum secure key length for RSA), 1536-bit, 2014-bit and then perform encryption followed by decryption of 117 bytes. The reason for using a small size input is that RSA only able to encrypt data to a maximum amount of its key size minus the padding data, 11 bytes for PKCS#1 v1.5 padding (the most common padding) [135]. So, RSA of key size 1024 bits will only encrypt 117 bytes (=128 bytes – 11 bytes). Figure 3.12 shows the results of testing RSA in the two libraries. The Figure shows that the time significantly increases as we increase the key length. This is especially true for BouncyCastle. The math library used by BouncyCastle for generating primes and testing them for primality is not as optimised as in `.NET`. The amount of memory utilised by RSA in both libraries is shown in Figure 3.13. BouncyCastle again is utilising more memory than `.NET`. BouncyCastle 1536-bit RSA uses $\approx$ 1.3 MB of memory compared to only 5.33 KB in `.NET`.

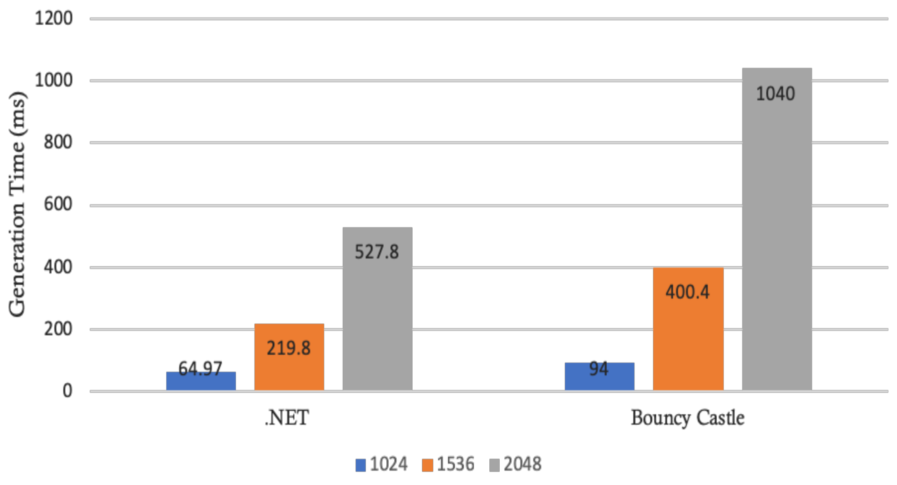


Fig. 3.12 Average time for key generation and encryption using RSA

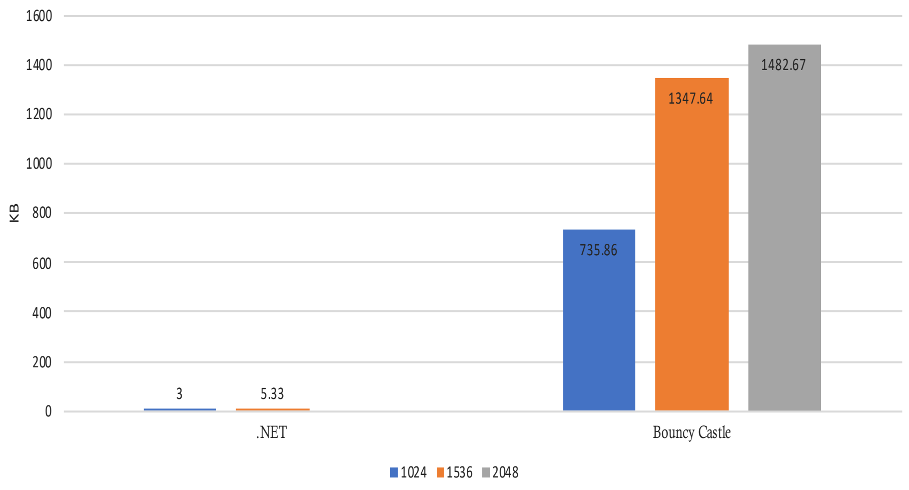


Fig. 3.13 The Amount of memory used for key generation and encryption using RSA

3.5.4 Hash Functions

Hash functions are mathematical algorithms that map data of an arbitrary length to small binary strings of a fixed length, known as hash values or digests. `.NET` provides implementations for SHA-1, SHA-256, SHA-384, SHA-512, and MD5. BouncyCastle provides a wider range of hash implementations which include; SHA-1, SHA-2 family, SHA-3, Blake, RipeMD family, Keccak, Shake, Skein, SM3, Tiger and Whirlpool. Figure 3.14 shows the results of testing the implementations of the common hash functions. `.NET's` SHA-1 at an average speed of $\approx$ 390 MB/s significantly outperforms the other hash functions in the two libraries. SHA-1, however, is no longer considered secure. `.NET` outperforms BouncyCastle for all the hash functions it implements except SHA-384 and SHA-512, which are about 50% slower than their counterparts. It is not clear why, as Microsoft does not provide any details about their primitives' implementations or offers any helpful detailed documentation. BouncyCastle is not good either. Figure 3.15 shows the results of testing the remaining BouncyCastle's hash functions. Tiger, RipeMD256, and RipeMD128 come second after SHA-1 at an average speed of $\approx$ 100 MB/s. SM3, Sake, SHA-224, and SHA-256 come next at average speed $\approx$ 75 MB/s followed by Skein, RipeMD160, Keccak, and SHA3 at average speed $\approx$ 62 MB/s. SHA-384, SHA-512, Blake2b, and RipeMD320 come next at average speed $\approx$ 53 MB/s. The slowest hash function is Whirlpool at average speed $\approx$ 7.5 MB/s. Both libraries use a minimal amount of memory to perform the hash computation, as seen in Figure 3.16. BouncyCastle, however, is better than `.NET`. Figure 3.17 show the amount of memory used by the rest of BouncyCastle hash functions for hashing 1 MB of data. For clarity, Blake and Gost3411 are omitted from the chart due to the large amount of memory they use. Blake2b and Blake2s use 1.14, 1.25 MB of memory, respectively, for hashing 1 MB data. BouncyCastle implementation of Gost3411 uses 18.88 MB of memory for hashing only 1 MB of data.

3.5.5 MAC Functions

MAC functions are similar to hash functions except that MAC functions use a private key to guarantee the authentication of the person who computes the MAC digest. MAC functions can be constructed from hash functions, block cipher, or two or more combined cryptographic primitives. `.NET` only provides implementations for MACs constructed from hash functions (Hash-based Message Authentication Code, HMAC), including HMACSHA1 HMACSHA256, HMACSHA384, HMACSHA512, HMACMD, and HMACRIPE160. BouncyCastle, on the other hand, provides implementations for MACs that are constructed from block ciphers (CMAC, GMAC), stream ciphers (VMPC-MAC) and hash functions (HMAC).

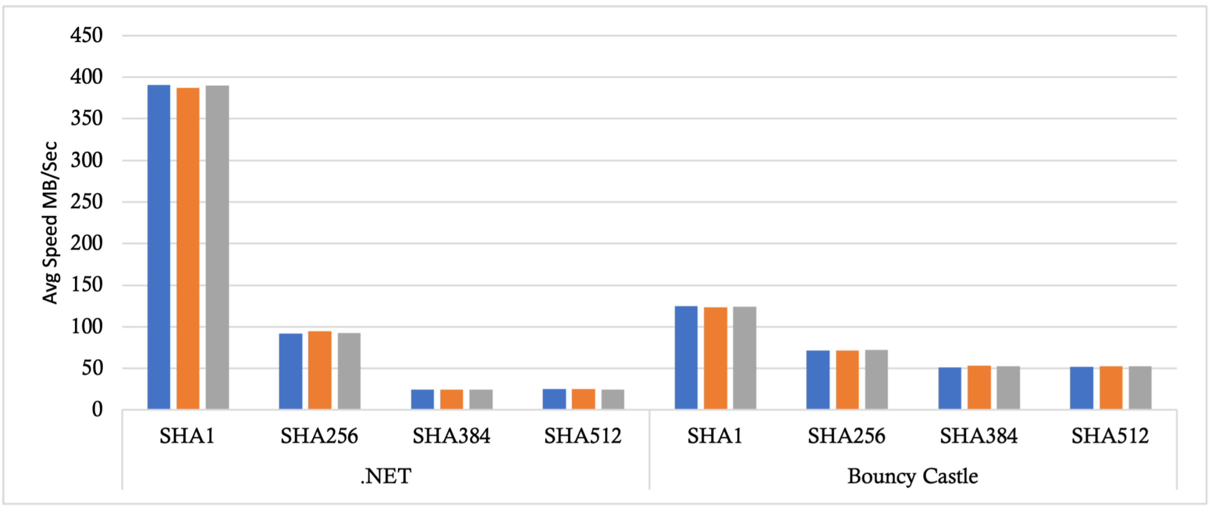


Fig. 3.14 Average speeds of the common hash functions implementations.

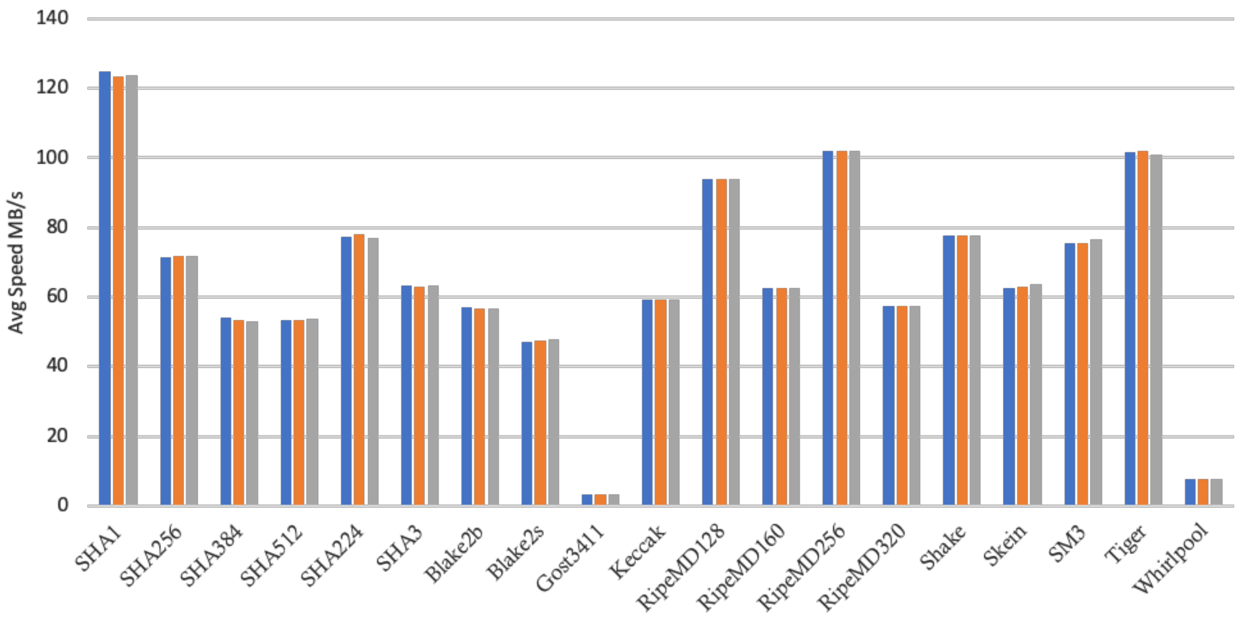


Fig. 3.15 Average speeds of BouncyCastle hash functions implementations.

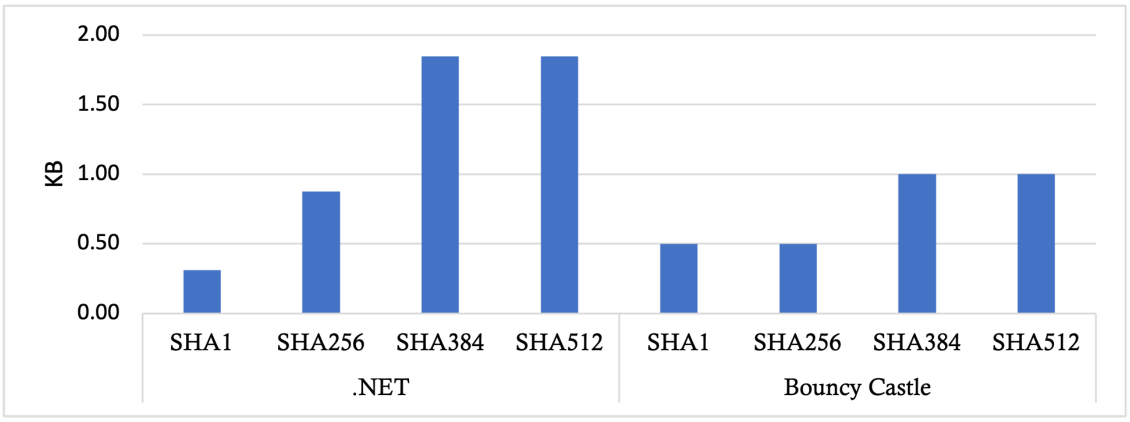


Fig. 3.16 The Amount of memory used for hashing 1 MB of data in kilobyte.

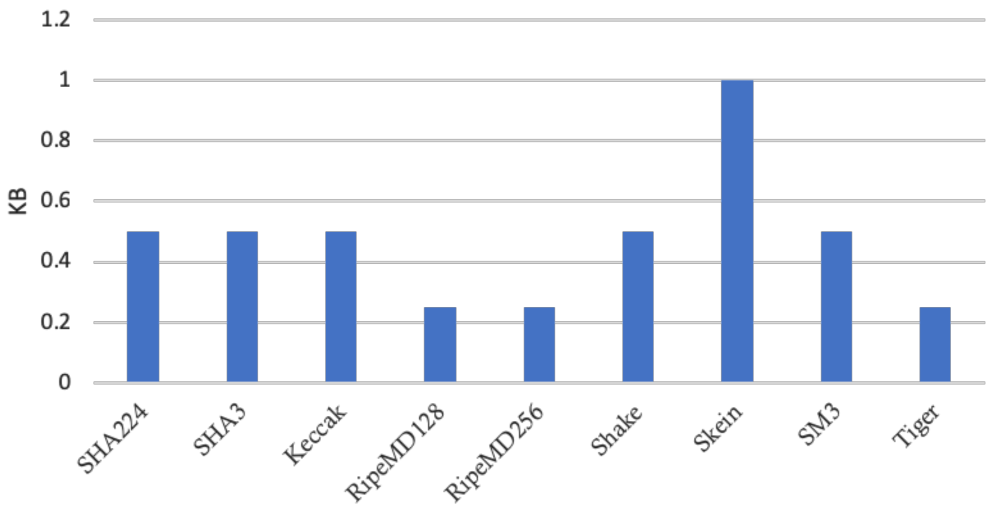


Fig. 3.17 The Amount of memory used by BouncyCastle hash functions for hashing 1 MB of data in kilobyte

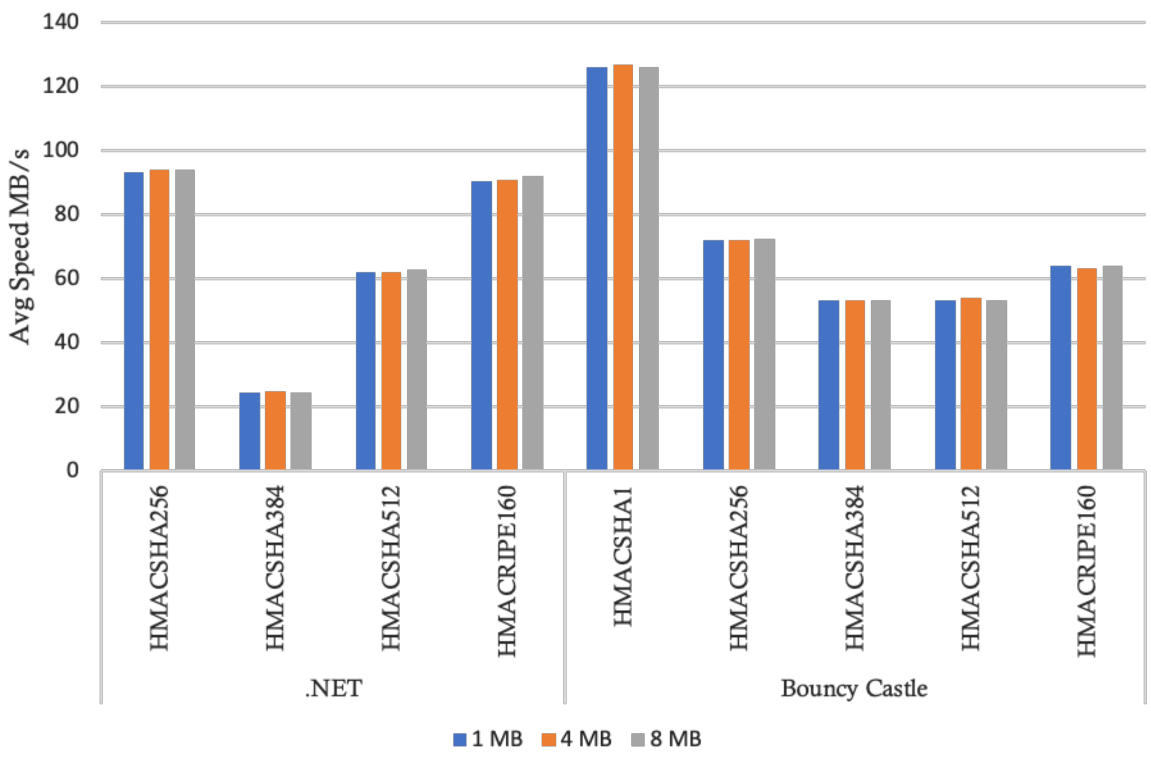


Fig. 3.18 The average speeds of the common MAC functions implementations.

Figure 3.18 shows the results of testing the common MAC functions. `.NET`'s HMACSHA1 and HMACMD5 are drawn in separate chart in Figure 3.19 for clarity. The two MAC functions are the fastest in our test. `.NET`'s HMACMD5 achieves a speed of $\approx$ 512 MB/s. BouncyCastle does not offer an implementation for this MAC. HMACSHA1, with an average speed of $\approx$ 394 MB/s, is three times faster than its counterpart in BouncyCastle. Except for HMACSHA-384, `.NET` outperforms BouncyCastle for all the MAC functions it implements.

Figure 3.20 shows the result of testing the remaining MAC functions from BouncyCastle. Poly1305 and SipHash at average speed $\approx$ 200 MB/s are the fastest BouncyCastle MAC functions. The stream and block cipher based MACs doesn't perform well compared to hash-based MAC functions. CMAC is the slowest MAC function at an average speed of $\approx$ 39 MB/s.

Figure 3.21 shows the amount of memory used by each MAC function to calculate the MAC digest for 1 MB data in kilobyte. All MAC functions consume a small amount of memory, $\approx$3 KB or less, except GMAC (not shown in the Figure), which uses 1.77 MB of memory to generate the MAC for a buffer of 1 MB only.

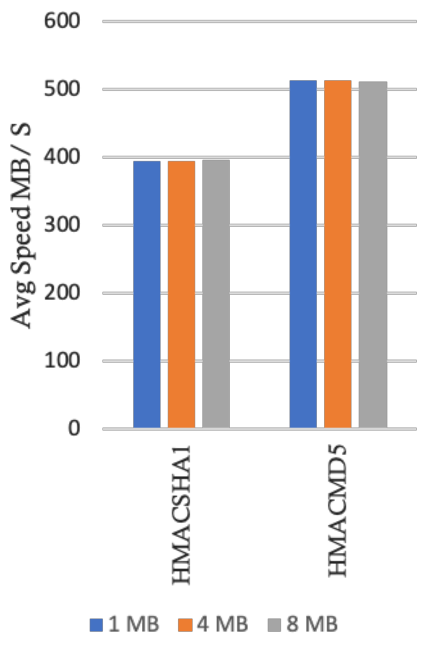


Fig. 3.19 Average speeds of `.NET`'s fast HMACSHA1 and HMACMD5 MACs.

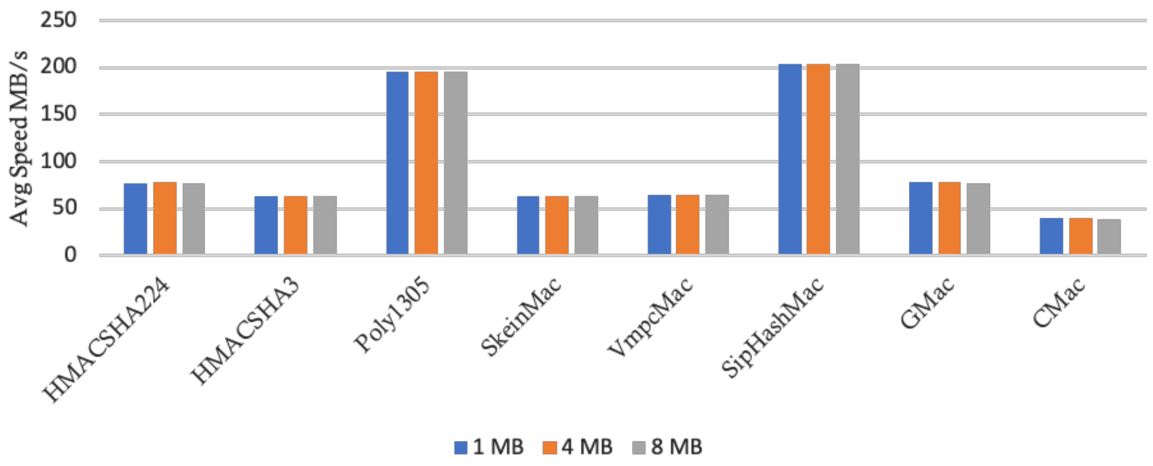


Fig. 3.20 Average speeds of the remaining BouncyCastle's MAC functions

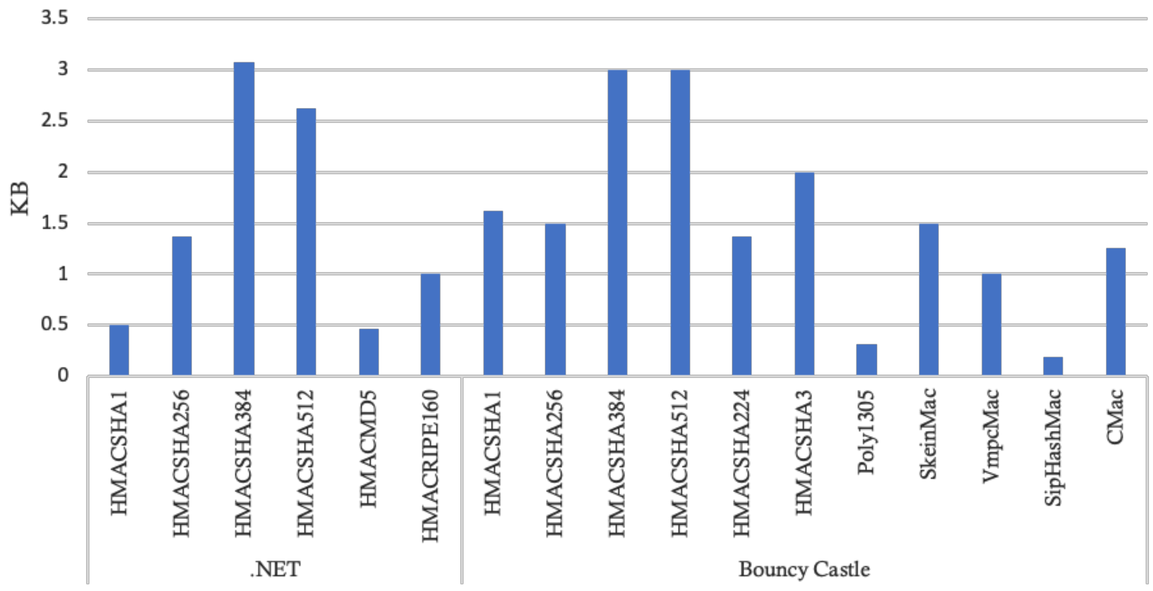


Fig. 3.21 The Amount of memory used by BouncyCastle MAC functions for hashing 1 MB of data in kilobyte

3.6 Conclusions

From the extensive list of the cryptographic primitives we tested, The results show that `.NET`'s AES (block cipher) at an average speed of ≈ 300 MB/s is by far the most efficient symmetric cipher tested. `.NET` provides high optimisation for the cipher by using AES-NI instruction sets. The results also show that `.NET`'s SHA-1 is the most efficient hash function tested. Both AES and SHA-1 are about three times faster than the next fastest primitive in their categories. `.NET`'s HMACMD5 at average speed ≈ 512 MB/s is the fastest MAC function. For asymmetric ciphers, the results show that `.NET`'s RSA significantly outperforms BouncyCastle's RSA for all key sizes tested. All primitives show consistent speeds for the three buffer sizes used.

It is worthwhile to mention that although speed tests give insight into primitives performance and are essential for further discussion, primitive selection cannot be solely based on them. Other parameters like perceived strength and security are crucial and should be considered.

Chapter 4

Investigating the Performance of C and C++ Cryptographic Libraries

Summary

This chapter presents a study on the performance comparison of several cryptographic primitives from six C/C++ open source libraries to understand the performance cost of the primitives and find the most efficient primitive and library. The study also investigates the effect of the key length on the performance of variable key-length primitives. The results show that AES is the most efficient block cipher tested. The results also show that ChaCha20 from OpenSSL is the most efficient stream cipher tested. The size of the key has some effect on the performance of the primitives. Smaller key sizes show better performance. However, this is not true for all the primitives with variable key lengths. Hash functions from the Blake2B family show an outstanding performance. OpenSSL and LibgCrypt libraries offer a higher level of optimisation for most of the primitives than the rest of the libraries. However, the LibgCrypt library suffers from poor documentation and a lack of examples.

4.1 Introduction

There are several open-source cryptographic libraries available for software developers to choose from. Some are comprehensive, and others are limited to few cryptographic primitives. Some provide an efficient implementation for the primitives, and others suffer from inefficient coding and resource utilisation. Some are highly modularised, making it easy to activate or deactivate certain library parts. Others are less modularised, making it difficult to remove unwanted ciphers and functionalities. We choose six mature and well maintained cryptographic libraries, four written in C and two written in C++ and evaluate their performance. The study investigates the primitives' performance within each library and compares their performance with their counterparts from other libraries.

The study aims to investigate and understand the performance cost of different primitives under different settings and eventually helps identify the most efficient primitive and library for developers to choose for their applications.

4.2 Cryptographic Libraries

This study investigates the performance of six general-purpose cryptography libraries that implement cryptographic primitives. All the libraries are free and can be used for both open and commercial applications. Except for LibgCrypt, all libraries, to some extent, provide a good level of documentation which includes API documentation and examples on how to use the APIs. Some libraries even provide manuals that describe their implementations and offer recommendations on using them securely and correctly. Unfortunately, some of the documentation provided is either outdated, incomplete, incomprehensive, or incorrect. Next, we will list the six libraries and give a brief overview of them.

Crypto++ [46] A free and open-source cryptographic library that provides implementations for a wide range of well-known and less frequently-used cryptographic primitives. The library is written in C++, and it fully supports 32-bit and 64-bit architectures for many major operating systems and platforms. The library uses preprocessor directives to detect the availability of the CPU's Instruction Sets Extensions (ISE), such as AES-NI, that are used in optimising the implementation of cryptographic primitives. If so, the library automatically activates the optimised version of the implementation.

Botan [113] Cryptographic C++ library that provides a wide variety of cryptographic primitives and protocols. The library is released under the permissive Simplified BSD license and can be freely used for both open and commercial applications. The German Federal Office for Information Security (BSI) conducted a project called "secure implementation of a universal crypto library" to provide an open-source cryptographic library that is secure, well-documented and suitable for "applications with increased security requirements" [36]. After reviewing and analysing the existing open-source libraries, the Botan library was selected as a suitable candidate for further development. The library then went through a rigorous examination process, after which all the identified security vulnerabilities and implementations problems were fixed, missing primitives were added, and all documentations were improved, extended and completed. The project was completed in early 2017. Botan 2.0 and later satisfies the requirement of BSI for securing applications [36].

OpenSSL [133] An open-source toolkit that implements the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols to secure communication over the Internet. It is also a general-purpose cryptography library that implements cryptographic primitives and provides various utilities. OpenSSL is written in C and provides wrappers for some other programming languages. The library is licensed under an Apache-style license, which means it can be freely used for commercial and non-commercial purposes.

LibgCrypt [101] A general purpose cryptographic library based on the code from GnuPG. It is written in C and licensed under the GNU Lesser General Public License and the GNU General Public License. The library provides functions for all cryptographic schemes, including symmetric and asymmetric ciphers, Hash and MAC functions, large integer functions, random numbers generators and key generation and management.

Nettle [124] A multi-platform, low-level cryptographic library. It provides simple interfaces for the developers to use. The library provides implementations for many block ciphers, few stream ciphers and all the hashes from SHA-2 and SHA-3 families. It is a small library written in C and licensed under the GNU Lesser General Public License.

LibTomCrypt [50] A comprehensive cryptographic library provides interfaces for most common ciphers, hash and MAC functions, public-key cryptography, and random number generators. The library is well documented, well maintained and easy to use. It is an open-source library written in C under the GNU General Public License.

4.3 Experiment setup

All tests are done on a machine running Ubuntu 16.04 on a virtual machine with 4 gigabytes of RAM and 3.3 GHz Intel® Core™ i7-6567U (Skylake) CPU with a dual processor and 4 MB shared level 3 cache. The performance of symmetric ciphers and hash functions are evaluated based on their throughput (in megabytes per second). The throughput is calculated by first measuring the execution time of a primitive then applying the following formula;

$$\textbf{Througput} = \frac{\textit{Data Size (MB)}}{\textit{Execution Time (Seconds)}}$$

The execution time includes (for symmetric ciphers) the time for cipher initialisation, encryption, and decryption of a buffer with a specific size. Inputs of sizes 1 MB, 4 MB, and 8 MB are used to test if primitives provide consistent performance under different data loads. As for asymmetric ciphers, the execution time includes the time for keys generation only. GNU G++ 7.3 is used to compile the code, and none of the optimisation flags are used in the test. Table 4.1 shows the versions of the libraries tested.

To measure the execution time of each primitive, we use the C++ `high_resolution_clock` clock in `std::chrono` namespace. This stopwatch clock is the most accurate as it has the shortest tick period provided by C++ [43]. Listing 4.1 shows how we use the `high_resolution`

Library	Version
Crypto++	5.6.5
Botan	2.3
OpenSSL	1.1.1
Libgcypt	1.8.0
Nettle	3.4
LibTomCrypt	1.18.1

Table 4.1 Cryptographic libraries' versions

_clock to measure the execution time. We first take a timestamp before executing the cipher and another timestamp right after the execution. Then we subtract the two timestamps to get the elapsed time. The results from each run are saved into a vector for further analysis.

```


 std::vector<double>  durationLog;
 for (int i = 0; i < 100; ++i)
{
        auto start = std::chrono::high_resolution_clock::now();
        //  code to measure ...
        testAES_256();
        auto end = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast
               <std::chrono::milliseconds>(end - start).count();

        durationLog.push_back(duration);
}

void testAES_256()
{
    // Encryption
    struct CBC_CTX(struct aes_ctx, AES_BLOCK_SIZE) enc_ctx;
    aes_set_encrypt_key(&enc_ctx.ctx, 32, key);
    CBC_SET_IV(&enc_ctx, iv);
    CBC_ENCRYPT(&enc_ctx, aes_encrypt, datalength, plaintext,
   ciphertext);

    // Decryption
    struct CBC_CTX(struct aes_ctx, AES_BLOCK_SIZE) dec_ctx;
    aes_set_decrypt_key(&dec_ctx.ctx, 32, key);
    CBC_SET_IV(&dec_ctx, iv);
    CBC_DECRYPT(&dec_ctx, aes_decrypt, datalength, ciphertext,
   decrypted_text);
}
```

Listing 4.1 Measuring the primitives' execution time

The benchmarking code was run in the Release build, not Debug build and without any attached debugger or profiler. The Debug version usually runs slower because of the debugging overhead and also, some optimisations are disabled in the Debug mode. All applications were closed (including the IDE), and nothing was left running in the background except the operating system processes. This is to minimise the system noise during the benchmark. The benchmarking tests are repeated at different times of the day, hoping to reduce the random noises' effect, if there are any.

Achieving stable and consistent results is not easy because of the benchmark noise. There are several sources for this noise; CPU frequency scaling is one of them. Modern CPUs change their frequencies up or down based on system workloads, temperature (to prevent the overheating of the CPU) and operating system power-saving policies. Also, a multi-core processor can turn off some of its cores to reduce power consumption. This has a significant impact on benchmarks results. For our experiment, We ensured that the CPU clock speed is the same for every benchmarking test. The Linux kernel allows its users to adjust the CPU frequency scaling and choose from four modes (called CPU governors). We selected the Performance Mode where the CPU runs at full clock speed.

In the experiment, we noticed that every time we start our benchmarking test, the ciphers' first few execution times are always higher than the others. This problem is caused by the "*cold start*" of the application where the system starts the loading of assemblies, initialises objects, populates the cache, etc. These activities add to the workloads and increase the time and spoil the benchmarking results. The problem also can happen when bringing the CPU out of power save mode. It is wise to exclude any readings from the cold start, which we did in this experiment. We observed that most of the tested ciphers reach a steady state after two to five iterations, so to be safe, we discarded the first ten calculations of the execution time as can be seen in Listing 4.2, Lines 4-7.

Choosing the ideal number of iterations is important to guarantee a high measurement accuracy and save time by reducing the experiment duration. After observing the execution times for many primitives from the different cryptographic schemes, we noticed that the measurements start to repeat after 20-30 iterations. For some primitives, this repetition happens even earlier. The only exception to this is some MAC functions that are very fast. For this experiment, we believe that with 100 iterations, we can make a reliable conclusion about the primitives' performance. The Standard Error (SE), which measures how far the true mean of the population from the average (sample mean) we calculated, can help choose the optimal number of iterations [77].

$$\textbf{SE} = \frac{\sigma}{\sqrt{n}} \quad \text{where } \sigma = \text{standard deviation and } n = \text{number of samples}$$

When the SE value is large, then we need to increase the sample size (number of iterations). SE is also used to indicate the uncertainty around our estimation of the mean; a smaller SE value shows a good estimate for the true mean. SE is implemented in Listing 4.2, Line 23.

Standard Deviation, which is a measure of how spread out the data, is used to indicate the benchmarks' accuracy. A large standard deviation means that data are spread out over a wider range and away from the mean [77]. It isn't easy to achieve a good level of accuracy or repeatable results with such a large spread. The standard deviation is implemented in Listing 4.2, Line 13-19.

$$\textbf{Standard Deviation} = \sqrt{\frac{\sum(x_i - \mu)^2}{N-1}} \quad \text{where } \mu = \text{mean and } N = \text{sample size}$$

Confidence Interval (CI) gives us a range of plausible values that we are reasonably sure our true mean lies in [77]. CI is also used to measure accuracy; when the CI range is too wide, then the average we calculated can not be trusted [3]. CI is implemented in Listing 4.2, Line 26-29. In the implementation, we used a confidence level of 95%.

$$\textbf{CI} = \bar{x} \pm z\frac{\sigma}{\sqrt{n}} \quad \text{where } \bar{x} = \text{sample mean and } z = \text{confidence level value (=1.960 for 95\%)}$$

In our experiment, we repeat each benchmarking test three times at different times of the day. If we find any variance in the results, we choose the results with a smaller standard deviation, tiny standard error, and a narrow confidence interval.

```
1
2  void calculate_stats(vector<double> vect)
3  {
4      // Discard the first 10 readings
5      vector<double> dLog;
6      for (int i=10; i < vect.size(); i++)
7      { dLog.push_back(vect[i]); }
8
9      // Calculating Mean
10     double mean = accumulate( dLog.begin(), dLog.end(), 0.0)/dLog.
   size();
```

```
    std::cout << "The Mean is "<< mean<< " ms "<< std::endl;

    // Calculating Standard Deviation
    double diff=0;
    for(std::vector<double>::const_iterator iter = dLog.begin(); iter
    !=dLog.end(); iter++ )
    {
        diff+=pow(*iter- mean, 2);
    }
    double stdDev = sqrt(diff / (dLog.size()-1));
    std::cout<<"Standard Deviation = "<< stdDev << endl;

    // Standard error
    double stdErr = stdDev / sqrt(dLog.size());
    std::cout<<"Standard Error = "<< stdErr<< endl;

    // Confidence Interval
    double ci_u = mean + 1.960 * stdErr;
    double ci_l = mean - 1.960 * stdErr;
    std::cout<<"Confidence Interval = ["<< ci_l << ", "<<ci_u<<"]"<<
   endl;
}
```

Listing 4.2 Measuring the mean, standard deviation and error, and confidence interval

4.4 Evaluation

Block size, round function, number of rounds, and key schedule function are the four design choices that significantly impact the cipher's performance. Ciphers with larger block size typically are faster [125]; a 128-bit block cipher requires half the number of execution cycles that a 64-bit block cipher needs to encrypt the same amount of data. Round functions consist of a set of operations for encryption and decryption. The complexity of these operations and how many times they are applied in a cipher impact its performance. Key schedule functions derive round keys from the private key. The complexity of these function negatively impacts the performance of the cipher.

4.4.1 Block Ciphers

Botan and LibTomCrypt have the widest range of block ciphers, 18 ciphers each, followed by Crypto++, 17 ciphers. Five block ciphers are common across the six libraries. These

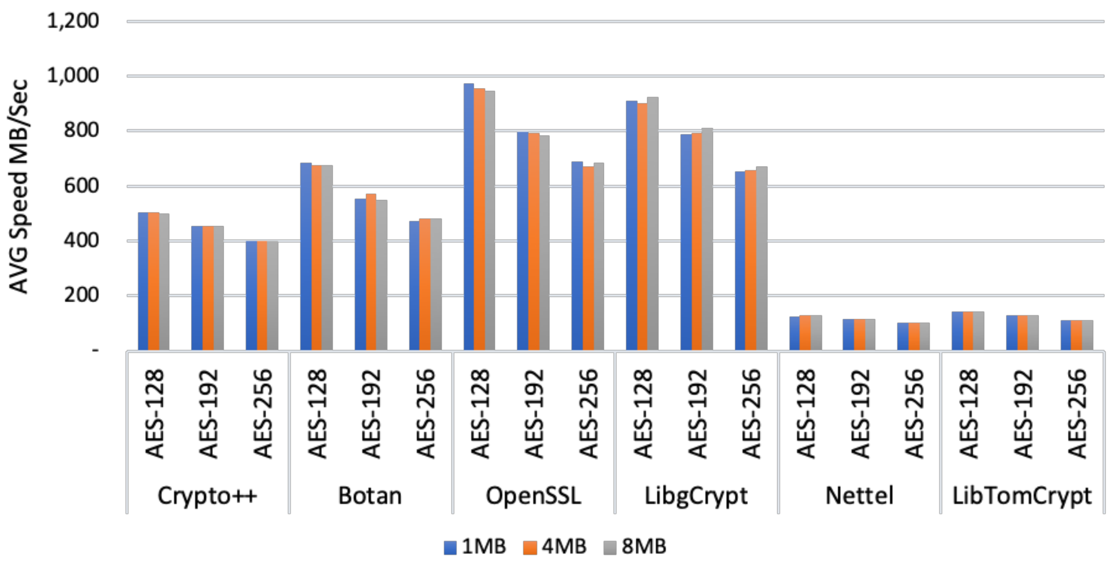


Fig. 4.1 Average speeds of AES implementations.

ciphers include; AES, Blowfish, CAST128, DES, and Camellia. The rest are common across two or three libraries or only found in one library. We tested all the block ciphers in Cipher Block Chaining (CBC) mode. The results of the testing are shown in Figures 4.1 and 4.2. For clarity, AES is split from the rest of the common ciphers and drawn in a separate chart in Figure 4.1.

Given that AES is the standard encryption cipher today, it is worth having a closer look at its performance. AES has a fixed block size of 128 bits which is an advantage. AES is based on a design known as a substitution–permutation network, and it is made of a succession of identical transformation rounds. The number of rounds is dictated by the key size used; 10 rounds for a 128-bit key, 12 for a 192-bit key, and 14 for a 256-bit key. Each round is a sequence of four inner transformations steps, including `SubBytes`; where each byte of the state is replaced with another from a lookup table, `ShiftRows`, `MixColumns`, and `AddRoundKey`; where each byte of the state is XORed with a byte of the subkey [44]. To speed up the execution of AES, the `SubBytes`, `ShiftRows`, and `MixColumns` operations are sometimes combined into a single round operation by transforming them into a sequence of table lookups [25] or using a Byte-oriented approach [115]. There are different AES implementation approaches. Some are hardware-specific (optimised to use hardware acceleration techniques, e.g. AES-NI, SSE, ARMv8-A), and others are generic software implementation. To speed up these implementations, there are several optimisation techniques such as lookup tables [44, 7, 149], bit-slicing [139, 116, 117] and byte-oriented approach [115]. Besides, today's processors from Intel, AMD and ARM contain an instruction set

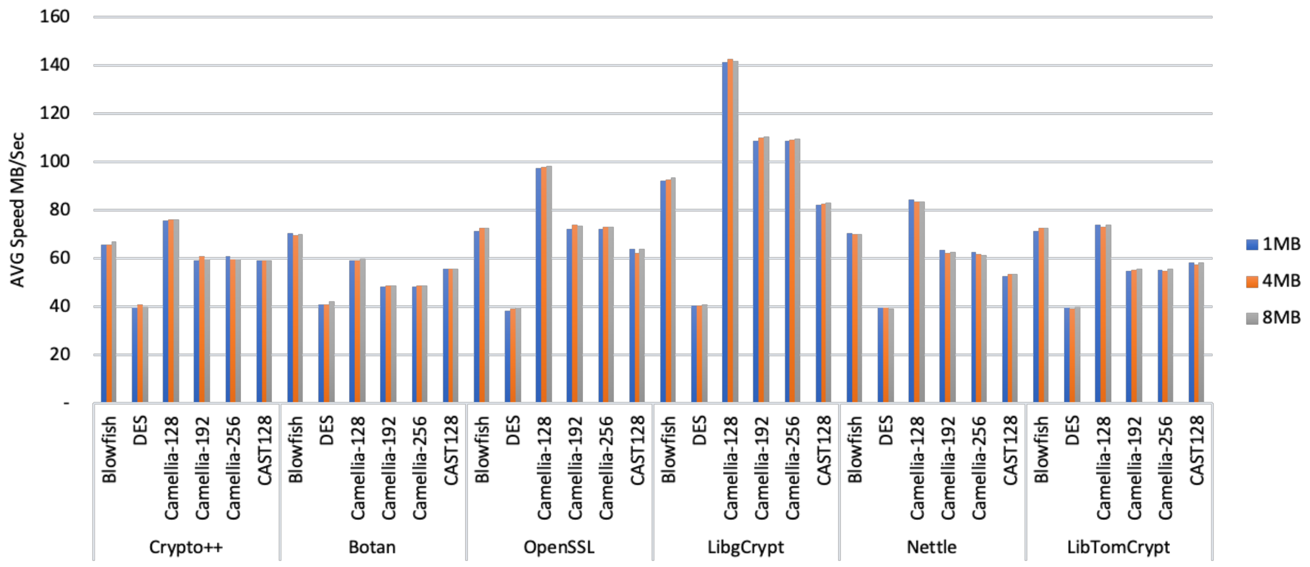


Fig. 4.2 Average speeds of the common block ciphers.

that provides hardware acceleration for AES. The instruction set, AES-NI, is designed to implement the cipher's performance-intensive operations using hardware and thus speed up the AES's execution. The AES-NI instruction set consists of four CPU's instructions for accelerating the encryption/decryption of a round and two instructions for the round key generation [145]. For the non-parallel mode of the cipher operations (e.g. CBC encryption), AES-NI can provide up to three times performance improvements over a complete software implementation. As for the parallel modes such as CBC decryption, AES-NI can provide ten times performance improvement over the software implementation [145].

Apart from the LibTomCrypt library, all the tested libraries support AES-NI in addition to some other hardware acceleration techniques, as seen in 4.2. The libraries default to AES-NI implementation if the CPU supports it. If not, they all have a fall-back mechanism on one of the other implementations.

	AES-NI	SSSE3/ SSE4.1	AVX/ AVX2	RdRand	VIA PadLock	ARMv8
Crypto++	✓	✓	✓	✓	✓	✓
Botan	✓	✓	✓	✓	x	✓
OpenSSL	✓	✓	✓	✓	✓	✓
LibgCrypt	✓	✓	✓	✓	✓	✓
Nettle	✓	x	x	x	x	✓
LibTom	x	x	x	x	x	x

Table 4.2 Hardware Acceleration support [165]

The libraries provide different optimisation levels; the most optimised AES implementations are found in OpenSSL and LibgCrypt. Both implementations achieved the highest speed in our test. It is worthwhile to mention that optimisations add complexity to the code and make it hard to understand and hard to maintain.

As can be seen in Figure 4.1, AES of all key sizes records the highest speed in the test except for Nettle and LibTomCrypt libraries. The highest speed achieved is by AES-128 from OpenSSL, which is around 960 MB/s. The cipher maintains a consistent speed for the three buffer sizes used. This consistency holds for all the tested primitives. The OpenSSL, which has the fastest AES implementation, provides a stitched AES CBC implementation using Function Stitching (a method of interleaving CPU instructions from two functions to optimise the cores' execution performance) [166] and also supports the multi-buffer processing techniques for AES CBC encryption [83]. Nettle's and LibTomCrypt's implementations are the slowest. This is because LibTomCrypt doesn't offer an AES-NI implementation. Nettle, on the other hand, has an AES-NI implementation but only in fat build. In non-fat builds, it has to be explicitly enabled. For this test, we use the native implementation, not the hardware optimised version.

Figure 4.1 also shows the impact of the key size on the cipher's performance. The difference between the performance of 128-bit AES and 256-bit AES is more pronounced ($\approx$ 25%). However, one might think that since the 128-bit AES uses only ten rounds compared to 14 rounds for 256-bit AES, 128-bit AES should be about 40% faster. The results show that the actual cost is not precisely linear in the number of rounds.

Figure 4.2 shows the results of the remaining block ciphers that are common across the six libraries. Camellia from LibgCrypt is the second-fastest common cipher for all key sizes. Camellia is based on a Feistel structure. It has the same block and key sizes as AES. However, the number of rounds is different; 18 for the 128-bit key and 24 for both 192 and 256-bit keys [6]. The cipher can be accelerated using CPU instruction sets designed for AES, such as AES-NI and AVX [98]. However, it is 7-11 times slower than AES, even using the hardware acceleration techniques. LibgCrypt's implementation of Camellia is about 31% faster than the second-fastest implementation of the cipher offered by OpenSSL. The third fastest block cipher is Blowfish from LibgCrypt at speed $\approx$93 MB/s. Blowfish is also based on Feistel structure but has a smaller block size, 64-bit. The cipher has 16 encryption rounds and supports keys of size 32-448 bits [147]. The cipher has a very slow key set-up, which negatively impacts its performance [112]. The Libraries show different speeds for the same ciphers except for DES, which shows almost a similar performance across all the different libraries.

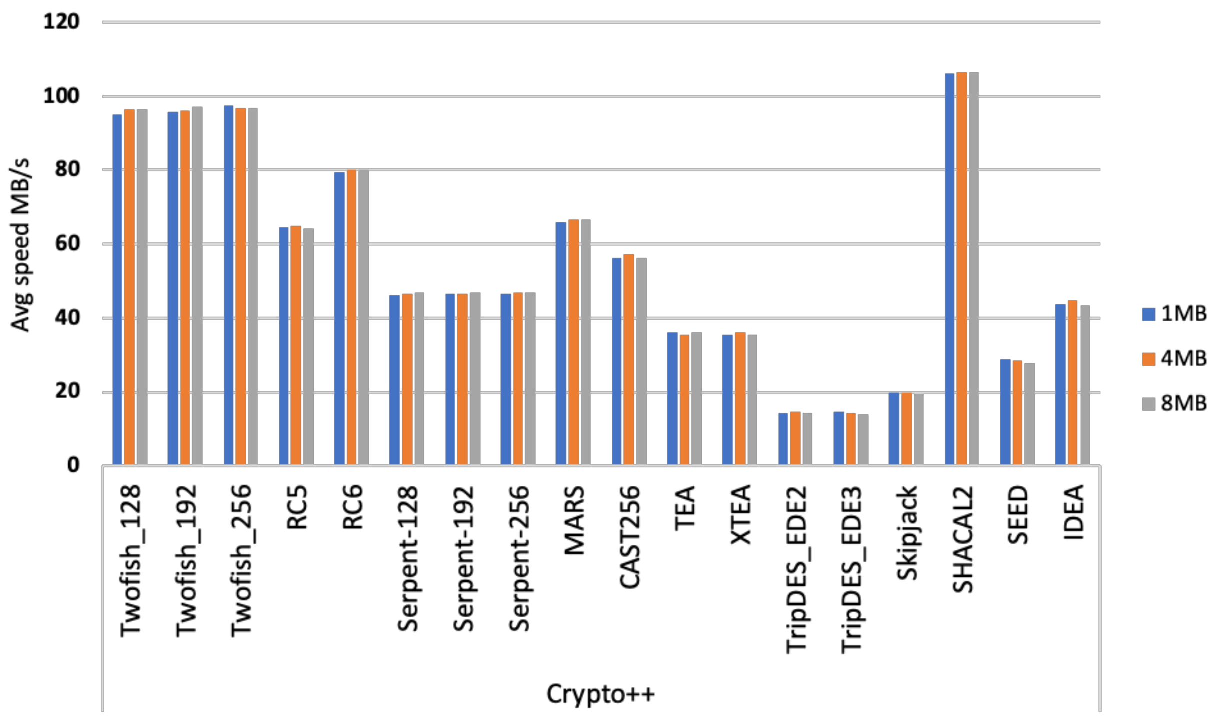


Fig. 4.3 Average speeds of Crypto++ block ciphers.

Some of the block ciphers are not common across all libraries. The results of testing these ciphers are shown in Figures 4.3-4.8. Figure 4.3 shows the block ciphers' average speeds in the Crypto++ library that are not common across all libraries. The TEA cipher is only available in the Crypto++ library. The rest are common across two or more libraries. SHACAL-2 is the fastest cipher at average speed ≈106 MB/s, and it comes second after AES in this library. SHACAL2 is also available in Botan; however, Botan's implementation is slower. Both libraries provide optimised implementation using SIMD and SHA-NI instruction sets. The cipher has a 256-bit block size and 64 rounds. It is based on SHA-256 and has a variable-length key of up to 512 bits[114]. Triple-DES cipher is the slowest at average speed ≈14 MB/s. The cipher's poor performance is due to its small block size (64-bit), its large number of rounds (48), and its bit-oriented operations that are not software friendly.

The average speeds of the remaining block ciphers found in Botan are shown in Figure 4.4. Threefish cipher at an average speed of ≈167 MB/s is the fastest block cipher in this library after AES. This cipher is only available in Botan. The cipher has wide block sizes (256, 512, and 1024 bits), making it useful for encrypting large blocks of data (e.g. in disk encryption). It supports keys of sizes 256-bit, 512-bit, and 1024-bit, and its number of rounds is 80 for the 1024-bit key and 72 for both 256 and 512-bit keys [64]. Botan provides an

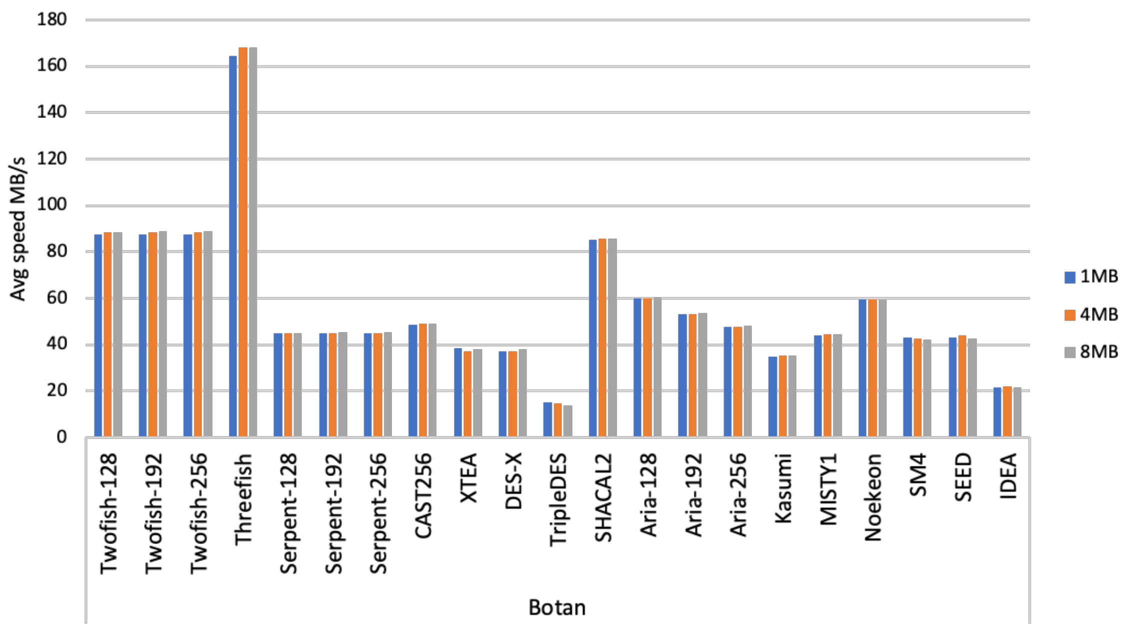


Fig. 4.4 Average speeds of Botan block ciphers.

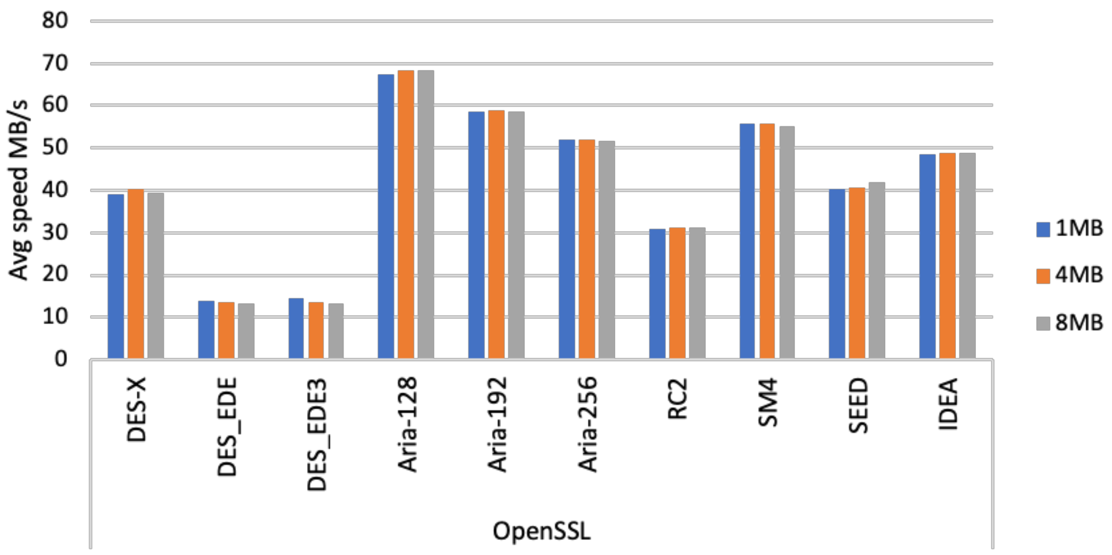


Fig. 4.5 Average speeds of OpenSSL block ciphers.

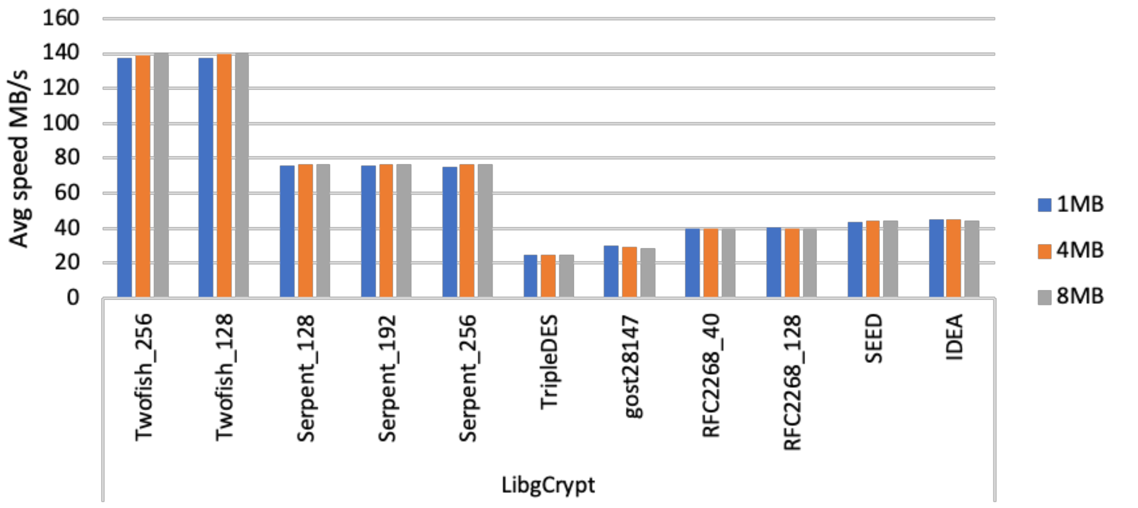


Fig. 4.6 Average speeds of LibGCrypt block ciphers.

optimized implementation for the cipher using AVX2. Triple DES at average speed ≈14 MB/s is the slowest in this library.

The results of testing the rest of the block ciphers from OpenSSL are shown in Figure 4.5. ARIA-128 at average speed ≈ 67MB/s comes fourth after Blowfish, Camellia and AES ciphers. ARIA is a South Korean cipher that shares similarities with AES. Both ciphers are based on substitution–permutation network, has the same block and key size, and encryption rounds [104]. However, there is a significant performance gap between the two ciphers. ARIA is available also in Botan, and the performance of the two implementations is comparable. Triple-DES cipher is the slowest at average speed ≈14 MB/s.

Figure 4.6 shows the average speeds of the remaining block ciphers from LibgCrypt. Twofish at average speed ≈139MB/s is the fastest ciphers after Camellia-128 (≈141MB/s) and AES. Twofish has the same block and key size as AES. The number of rounds, however, is 16 for all the key sizes. Crypto++'s and LibTomCrypt's implementation of the cipher are 30% slower. Botan's and Nettle's implementation are 36% slower. LibgCrypt's implementation is faster because it is written in assembly and used the AVX2 instruction set to speed up the execution of the cipher. Triple DES is the slowest cipher at average speed ≈ 24MB/s

The average speeds of the remaining block ciphers from the Nettle library are shown in Figure 4.7. Twofish cipher is the second-fastest cipher (after AES) at an average speed of ≈ 88 MB/s. Triple DES is the slowest at an average speed of ≈ 13 MB/s.

Figure 4.8 shows the average speeds of the remaining LibTomCrypt's block ciphers. RC5 is the fastest block cipher in this library at average speed ≈ $156MB/s$. RC5 is a simple cipher that has variable block sizes of 32, 64, and 128 bits. It supports keys of sizes 0 to 2040-bits

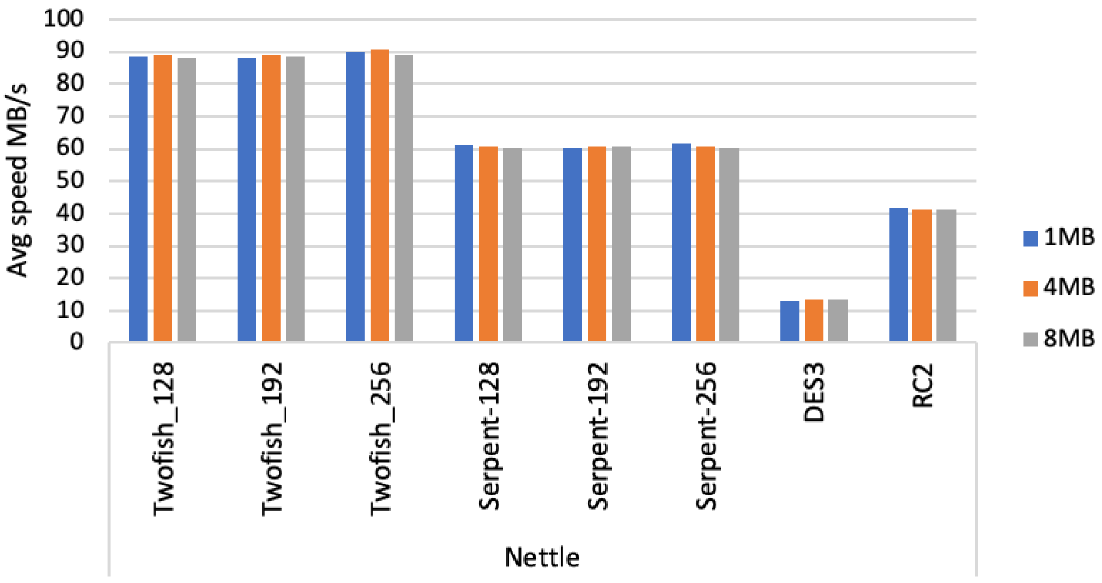


Fig. 4.7 Average speeds of Nettle block ciphers.

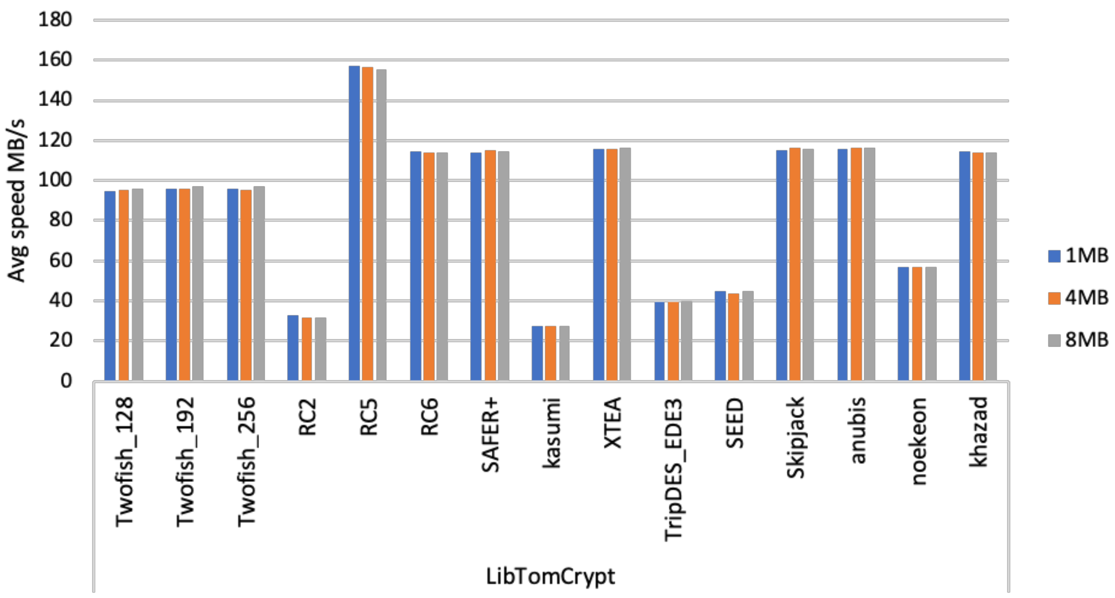


Fig. 4.8 Average speeds of LibTomCrypt block ciphers.

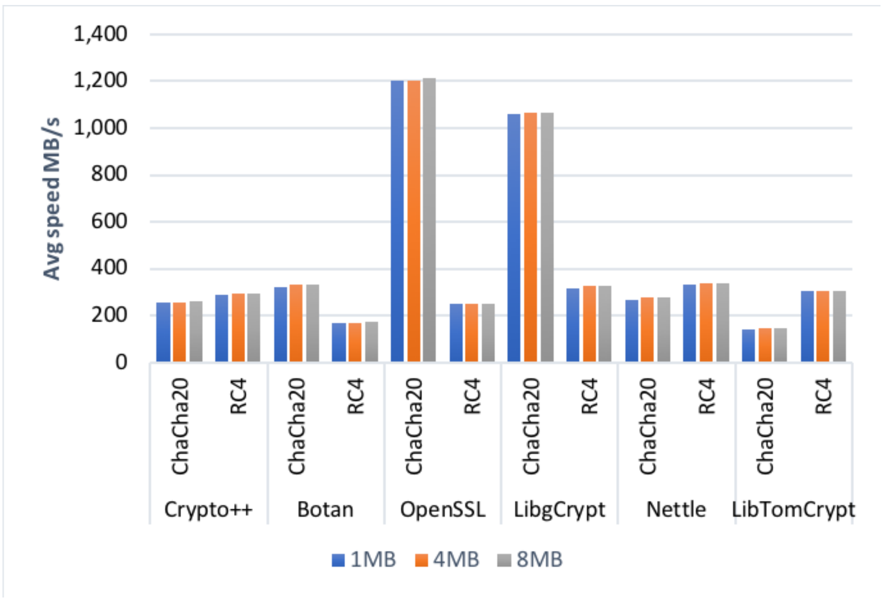


Fig. 4.9 Average speeds of the common Stream Ciphers.

and rounds from 1-255 [141]. For this test, we used a 64-bit block size, a 128-bit key and 12 rounds. Kasumi is the slowest cipher at an average speed of $\approx 28MB/s$. The cipher has a small block size and is better suited for hardware implementation.

4.4.2 Stream Ciphers

Stream ciphers are symmetric encryption ciphers that encrypt a single bit or byte of plaintext at a time. They use a stream of pseudo-random bits (Keystream) that is XOR'ed with the plaintext to generate the ciphertext. The pseudo-random keystream is generated from a random seed value using digital shift registers such as Linear Feedback Shift Registers (LFSRs). The seed value serves as the cryptographic key [58]. Stream ciphers typically are faster than block ciphers and have lower hardware complexity [75]. They are often used in applications where the amount of data is either unknown or continuous such as network streams.

In this experiment, six distinct stream ciphers are tested in addition to their variations. These ciphers include Salsa, ChaCha, Panama, RC4, Sober128 and Sosemanuk. Crypto++ has the widest range of stream ciphers compared to the other libraries. OpenSSL is the most limited library with only two stream ciphers implemented. RC4 and ChaCha-20 ciphers are the only common ciphers across all libraries. Until recently, RC4 was the most widely used cipher of all stream ciphers, especially in software [161]. The cipher has some vulnerabilities and is no longer considered secure [137]. However, a number of attempts have been made to

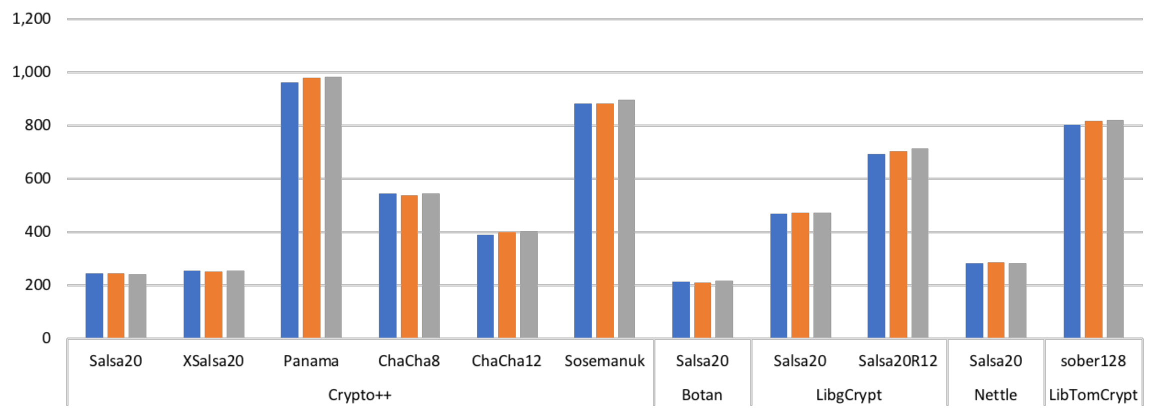


Fig. 4.10 Average speeds of Stream Ciphers that are not common across the libraries.

strengthen it [170]. The design of RC4 avoids the use of LFSRs which makes RC4 ideal for software implementation. Stream ciphers that are based on LFSRs are efficient in hardware but not so in software [58]. ChaCha cipher is a modification of the Salsa20 cipher. It aims to improve the diffusion per round and achieve the same or slightly better performance than Salsa20 [24]. ChaCha-20 uses 20 rounds and supports keys of size 128 or 256 bits [24]. Figure 4.9 shows the results of testing the two ciphers. The results show that ChaCha-20 significantly outperforms RC4 in OpenSSL, LibgCrypt, and Botan. RC4 outperforms ChaCha-20 in Crypto++, Nettle, and LibTomCrypt. OpenSSL's ChaCha-20 is the fastest stream cipher at an average speed of 1201 MB/s, followed by LibgCrypt's ChaCha-20 at an average speed of 1060 MB/s. The two libraries provide optimised implementations using assembly language and hardware acceleration techniques. OpenSSL provides an optimised implementation of the cipher for ARMv4, ARMv8, C64x+ core, PowerPC/AltiVec, s390x, x86, and x86_64. LibgCrypt provides optimised implementations for ARMv7 NEON, AVX2, SSE2 AMD64, and SSSE3 AMD64. The fastest RC4 is from the Nettle library at an average speed of 334 MB/s. The slowest stream cipher tested is LibTomCrypt's ChaCha-20 at average speed $\approx$144 MB/s.

The results of testing the remaining stream ciphers are shown in Figure 4.10. Panama from Crypto++ is the third-fastest stream cipher after ChaCha-20 from OpenSSL and LibgCrypt. Its average speed is 979 MB/s. The Salsa family does not perform well compared to the rest of the stream ciphers. This is true in all libraries except LibgCrypt.

4.4.3 Asymmetric Ciphers (Key Generations)

The libraries provide implementations for three public-key ciphers, including RSA, DSA, and ElGamal. RSA is the only common cipher across the six libraries. Libgcrypt provides

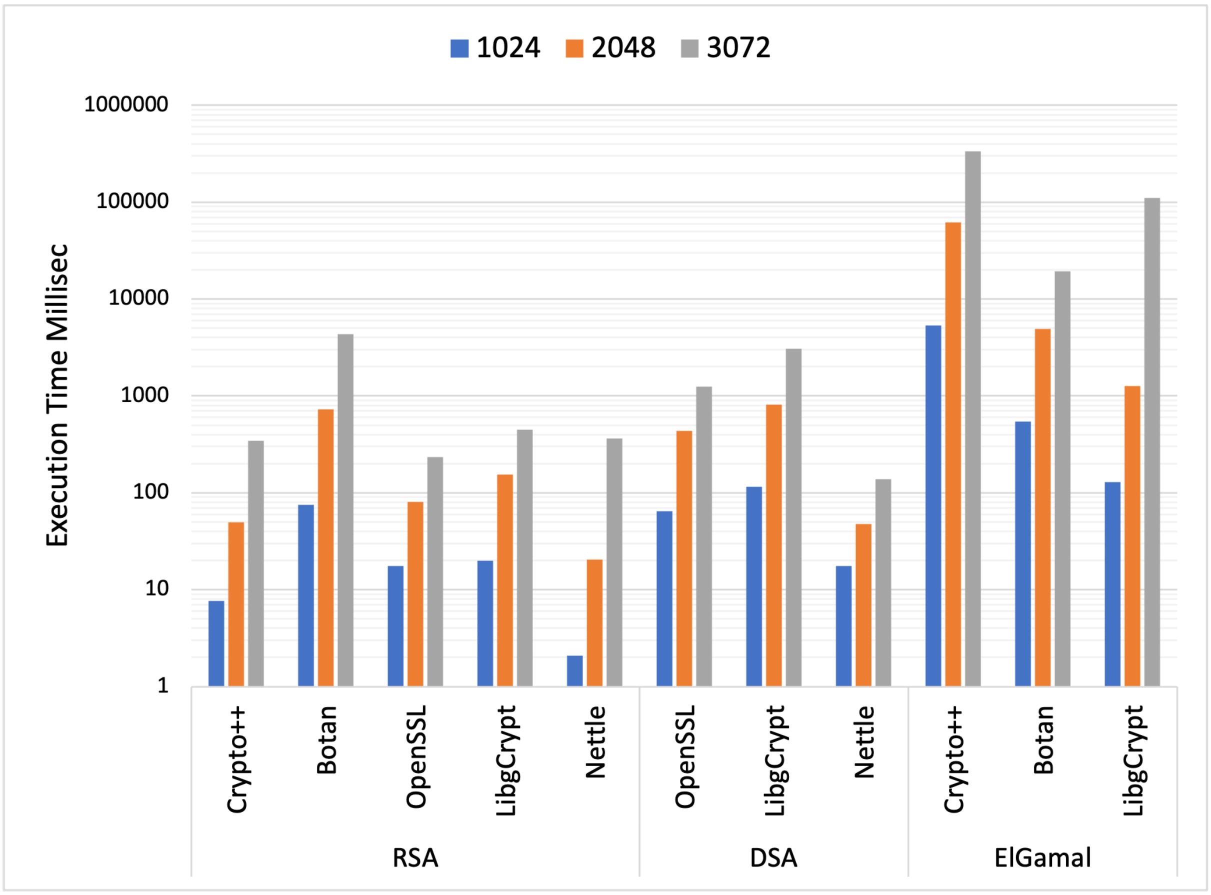


Fig. 4.11 Key generation times for variable key sizes

implementations for the three ciphers, while Crypto++ and Botan only offer RSA and ElGamal implementations. OpenSSL, Nettle and LibTomCrypt provide an implementation for RSA and DSA only. The experiment calculates the time it takes for each library to generate keys of sizes 1024, 2048, and 3072 bits. The encryption and decryption times are not calculated.

Figure 4.11 shows the average time for each library to generate keys of different sizes for the three public-key ciphers. The Figure shows ElGamal is the slowest cipher, and its implementation in Crypto++ is the least efficient. To generate a key, ElGamal needs to find a prime number p for which (p-1)/2 is a prime. This is harder and requires significantly more time than RSA, which requires finding two unrelated large primes to generate its key [46]. RSA cipher is the most efficient asymmetric cipher tested, and Nettle's implementation of the cipher outperforms the rest of the libraries except for the 3072-bit key size. DSA is slightly slower than RSA but much faster than ElGamal.

4.4.4 Hash Functions

There are 15 different hash functions implemented in the libraries in addition to their variations. The Botan library has the widest range of hash functions, including 12 distinct functions plus their variations.

Figure 4.12 shows the average speeds of the common hash functions. SHA1 from OpenSSL with an average speed of 860 MB/s is the fastest hash function across all libraries. OpenSSL offers a high level of optimisation for all hash implementations. As can be seen from Figure 4.12 all the hashes from this library outperform their counterparts from other libraries except for RIPEMD160.

Figures 4.13-4.18 shows the average speeds of the hash functions that are not common across all the libraries. Figure 4.13 shows the results from LibgCrypt. The library's implementations of hash functions outperform the other libraries' implementations except for OpenSSL. Libgcrypt implements ten different hash functions in addition to their variations. Blake2b family at an average speed of $\approx$ 774 MB/s outperforms all the hashes in this library except SHA1(825MB/s). Tiger family also perform better than the remaining hashes at an average speed of $\approx$ 590 MB/s. The slowest hash in this library is the Russian GOSTR3411_94 at an average speed of $\approx$ 44 MB/s.

OpenSSL has the fastest hash function across all libraries, SHA1, at an average speed of $\approx$ 860 MB/s. The library also has the slowest hash function, MDC2, at an average speed of 18 MB/s. As shown in Figure 4.14, the library's SHA-3 family did not perform well compared to the SHA-2 family. Blake2b-512 is the second fastest hash in the library at an average speed of $\approx$ 775 MB/s.

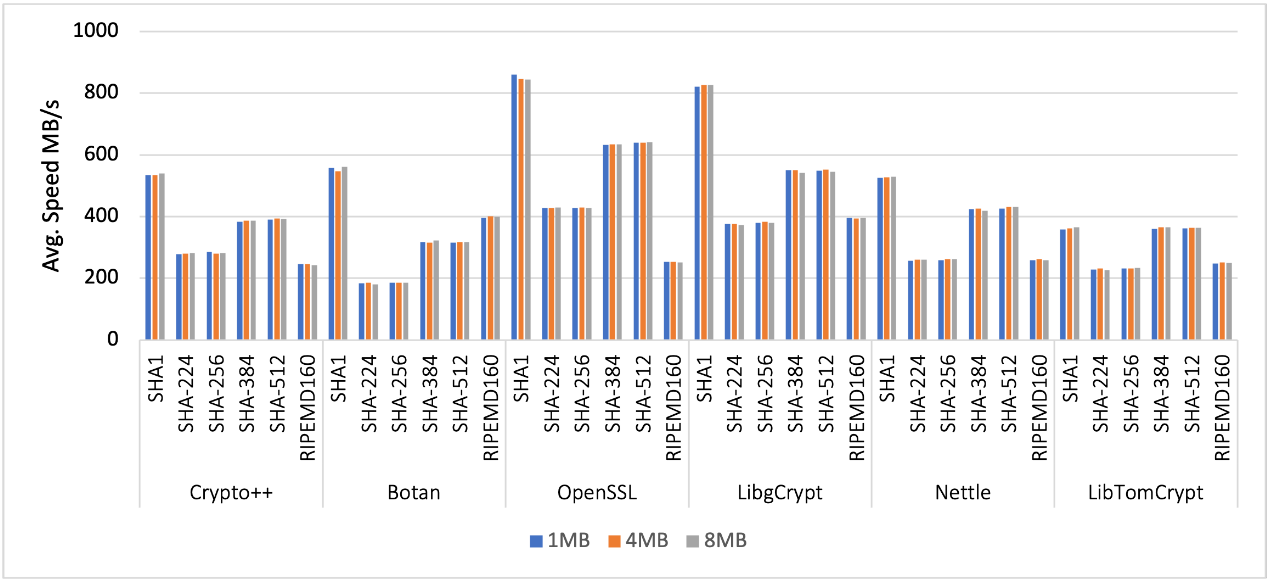


Fig. 4.12 Average speeds of the common hash functions

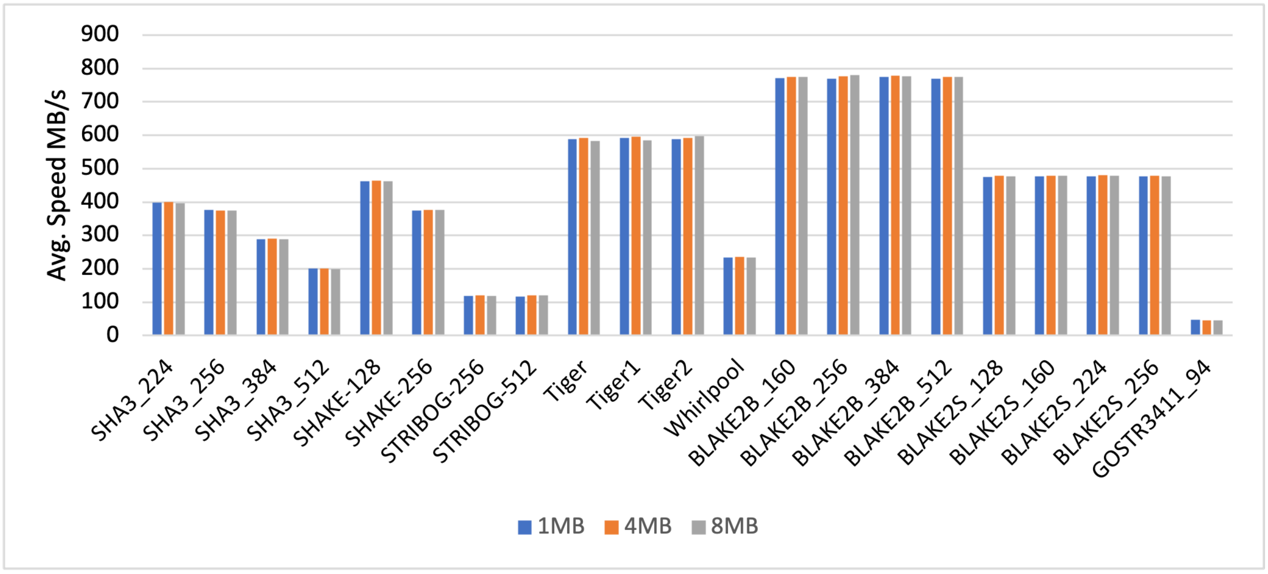


Fig. 4.13 Average speeds of LibgCrypt hash functions

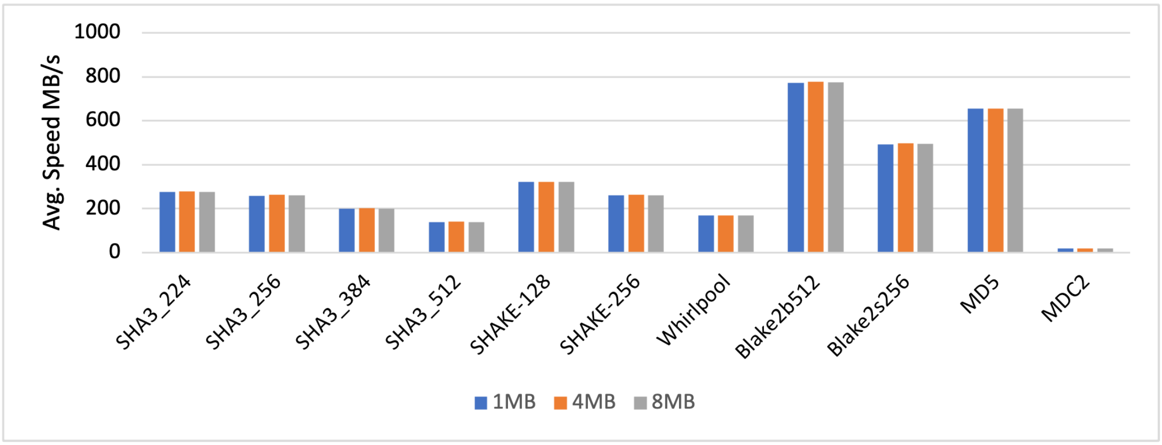


Fig. 4.14 Average speeds of OpenSSL hash functions

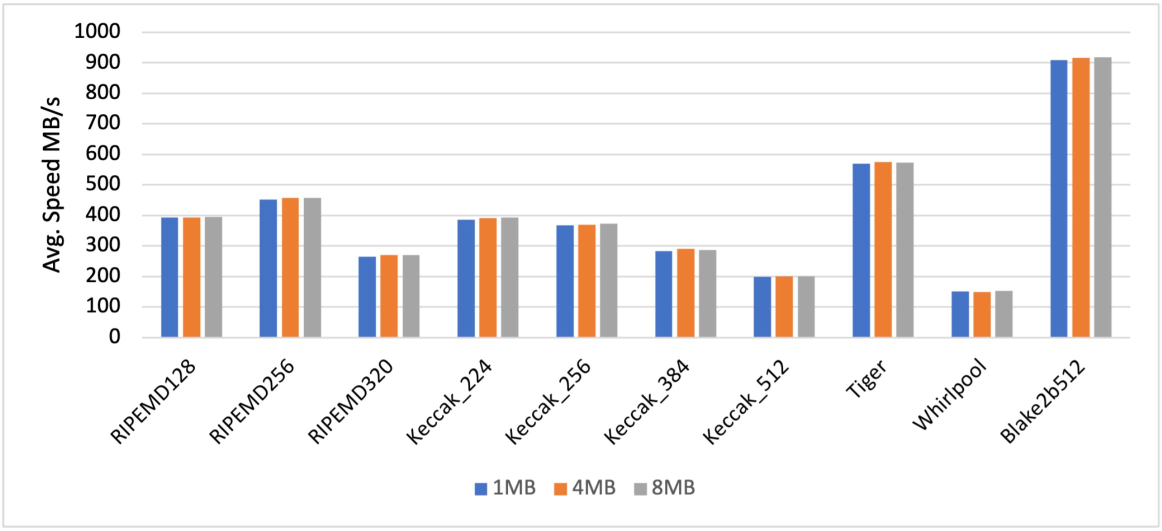


Fig. 4.15 Average speeds of Crypto++ hash functions

Figure 4.15 shows the results from Crypto++. The library's implementations of hash functions achieve an average performance compared to the other libraries' implementations. As can be seen from Figure 4.15, Blake2b-512 at average speed $\approx$ 914/s outperforms all the hashes in the library. At an average speed of $\approx$ 573/s, Tiger is the second-fastest functions in the library. Whirlpool is the slowest at an average speed of $\approx$ 150 MB/s.

Nettle library provides all the hashes from SHA-2 and SHA-3 family in addition to RipeMD. SHA-1 is the fastest hash in this library at an average speed of $\approx$ 526 MB/s. Functions from the SHA-2 family outperform the SHA-3 family. Figure 4.16 shows SHA3-224 at an average speed of $\approx$ 281 MB/s performs better than all of the SHA-3 functions.

Figure 4.17 shows the results of LibTomCrypt hash functions. Tiger at an average speed of $\approx$ 482 MB/s is the fastest hash in this library. Blake2B family outperforms the Blake2S

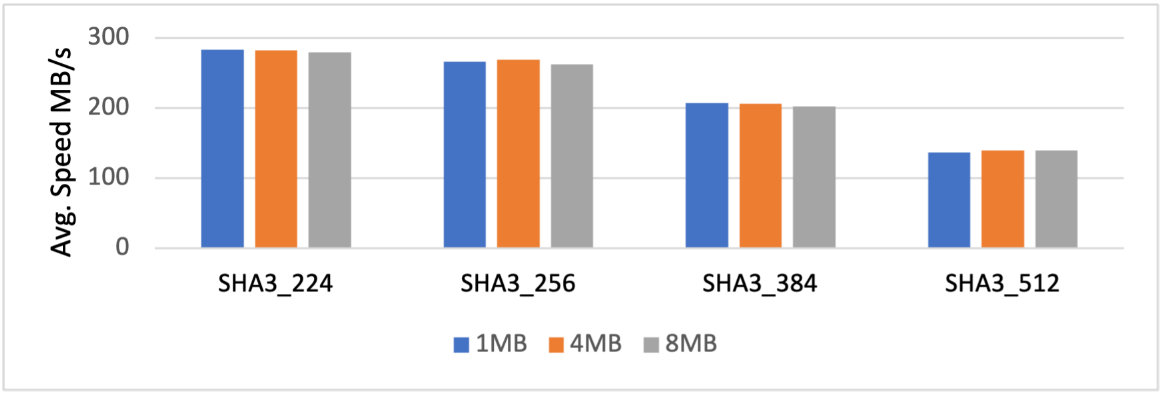


Fig. 4.16 Average speeds of Nettle hash functions

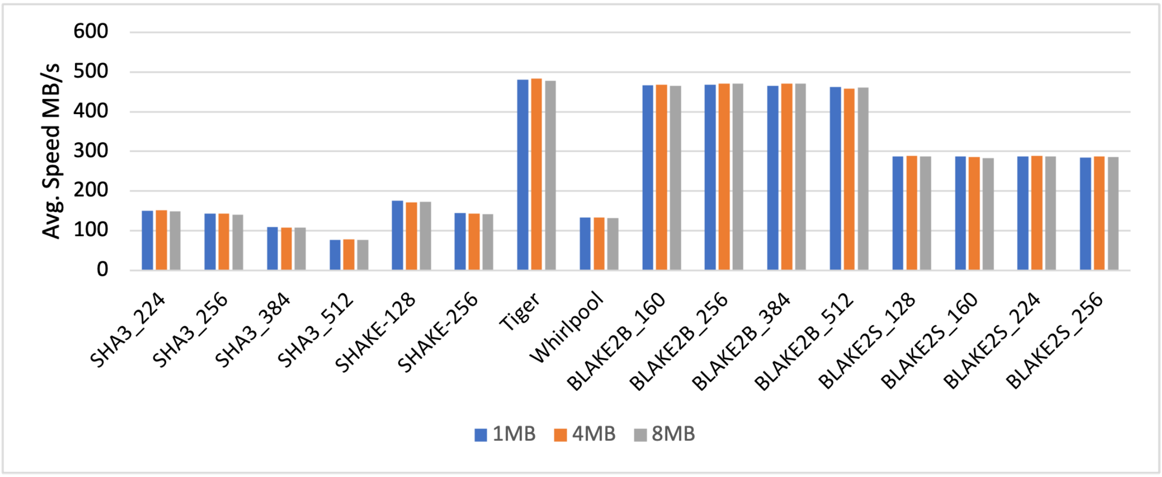


Fig. 4.17 Average speeds of LibTomCrypt hash functions

family and comes second at an average speed of ≈ 465 MB/s. Hash functions from the SHA-2 and the SHA-3 family don't perform well compared to their counterparts in other libraries.

Figure 4.18 shows the average speeds of the hash functions in the Botan library that are not common across all libraries. Blake2b-512 at an average speed of ≈ 713 MB/s outperforms the rest of the hashes in the library. Skein-512 at an average speed of ≈ 530 MB/s is the third after SHA-1. GOST3411 is the slowest at an average speed of ≈ 44 MB/s.

The top ten fastest hash functions in our test are shown in Figure 4.19, five of them are from the LibgCrypt library. Hashes from the Blake2B family show an outstanding performance; seven of these hash functions are among the best performant in our test. The fastest hash function tested is Blake2B512 from Crypto++ at an average speed of 914 MB/s.

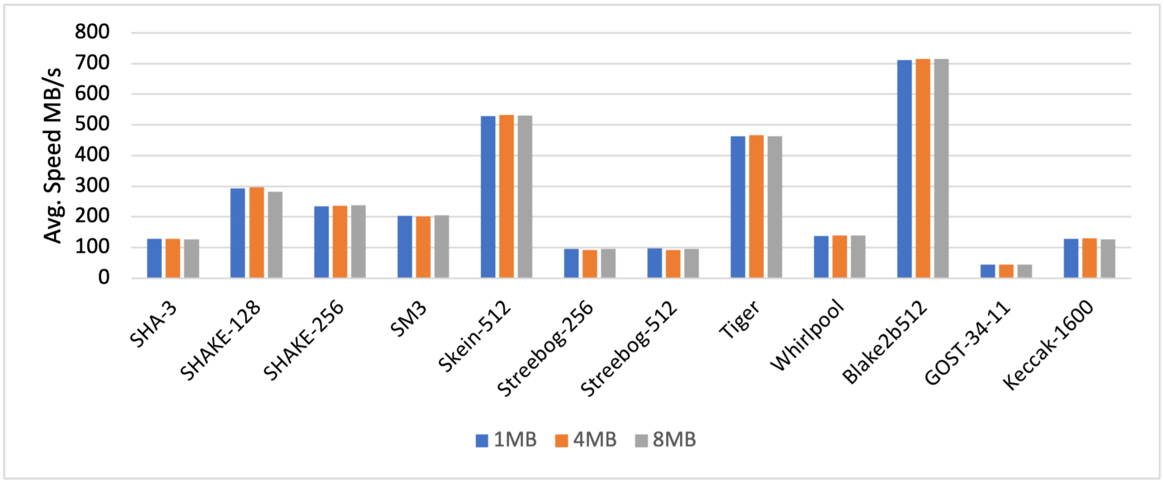


Fig. 4.18 Average speeds of Botan hash functions

4.5 Comparison with .NET Implementations

Table 6.18 shows a comparison between the performance of C# and the C/C++ implementation of the common block and stream ciphers. For each primitive, the table shows the most efficient implementation from .NET and C/C++ libraries. The results show that C/C++ implementations are much more efficient than those of C#. C and C++ are low-level languages that compile directly into machine code. The languages have advanced optimisation options available to their compilers. These optimisation techniques are utilised well by most of the tested C/C++ libraries, as we have seen in Section 4.4. On the other hand, C# is a bytecode based language that compiles first into Microsoft Intermediate Language, then the Just-In-Time (JIT) compiler generates the machine code. This is why C/C++ are typically faster than C#.

Although .NET implementations take advantage of the CPU acceleration techniques, the performance of its implementations is far behind. Unlike BouncyCastle, The .NET implementations are not published. So we could not study the optimisation techniques used. The C# version of the BouncyCastle, has the least efficient implementations. This is because the library provides reference implementations with no optimisation, unlike its Java version.

The memory utilisation of both .NET and BouncyCastle is relatively good. However, we did not measure the memory utilisation for the implementations from the C/C++, so no comparison is given.

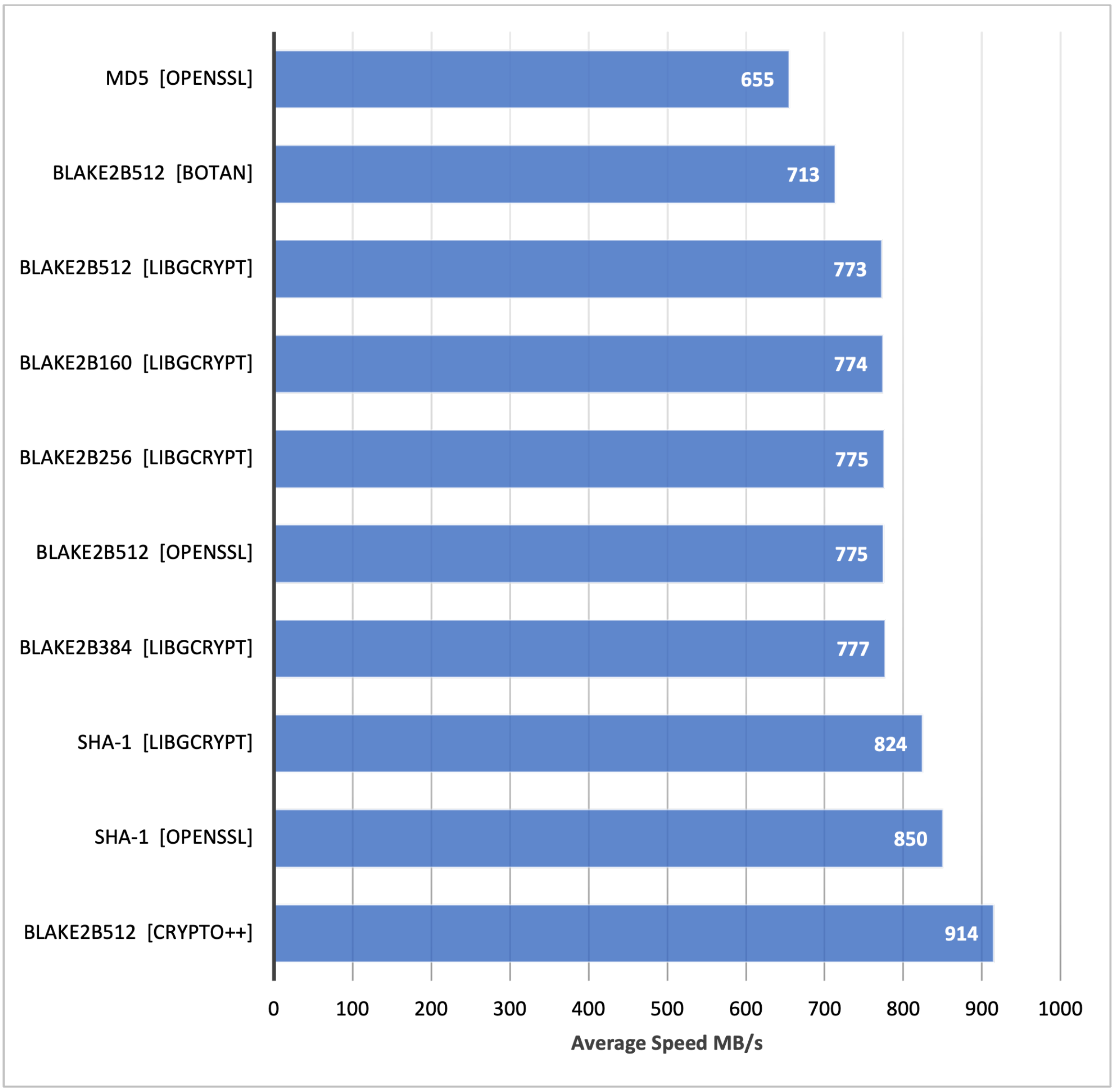


Fig. 4.19 Fastest hash functions across all libraries

4.6 Conclusion

In this chapter we evaluated the performance of an extensive list of cryptographic primitives from six C and C++ libraries. The results have shown many interesting observations which we summarise as follows:

- Optimisation adds complexity to the code making it hard to understand and maintain.
- Programming languages have a considerable impact on the performance of the cipher. For example, the NET's C# implementation of AES has an average speed of

Table 4.3 Comparison of .NET and C/C++ implementations' performance

Scheme	**Cipher**	**Key Size**	**Languages**	**Throughput MB/s**	**Library**
Block ciphers	AES	128-bit	C# .NET	317.46	Microsoft .net
			C/C++	985.22	OpenSSL
	DES	56-bit	C# .NET	19	BouncyCastle
			C/C++	≈ 40	all libraries
	3DES	112-bit	C# .NET	13.47	Microsoft .net
			C/C++	38.91	LibTomCrypt
	RC2	128-bit	C# .NET	20.95	Microsoft .net
			C/C++	42.05	Nettle
Stream Ciphers	ChaCha20	256-bit	C# .NET	42.15	BouncyCastle
			C/C++	1200	OpenSSL
	RC4	256-bit	C# .NET	72.45	BouncyCastle
			C/C++	350.72	LibGCrypt

300 MB/second, while OpenSSL's `C` implementation has an average speed of 960 MB/seconds

- Stream ciphers are faster than block ciphers. However, stream ciphers have a lengthy initialisation phase making them inefficient for encrypting a small amount of data
- The key size impacts the cipher's performance only if it dictates the cipher's number of rounds.
- The effect of the block size depends on the amount of data the cipher processed. If the cipher handles a large amount of data, then a larger block size means a faster cipher. However, if the amount of data is small, then a larger block size will negatively impact the cipher's speed. This is because of the block padding involved. It is worth mentioning that ciphers with larger block sizes use more memory and thus not suitable in a constrained environment.
- The primitives optimised to use the hardware acceleration techniques outperform their counterparts that use only software optimisation techniques.
- OpenSSL's AES is the fastest block cipher at an average speed of 960 MB/second, and OpenSSL's ChaCha-20 is the fastest stream cipher at an average speed of 1201 MB/seconds.

- Hash functions from the Blake2B family show an outstanding performance compared to the other hash functions.
- OpenSSL and LibgCrypt libraries offer a higher level of optimisation for most of the primitives than the rest of the libraries.

Chapter 5

Performance of C and C++ Cryptographic Libraries under a Constrained Device

Summary

This chapter investigates the performance of different cryptographic primitives when run under a device with limited computing capacity. We test the implementations of the block and stream ciphers from the six cryptographic libraries discussed in the previous chapter aiming to understand their performance and energy consumption when run under a resource-constrained device. We also examine the hardware optimised implementations of the primitives to understand their performance in the absence of any hardware acceleration support. Additionally, we study the impact of the key and block size and number of rounds on the power consumption of the primitives. The chapter also discusses three methods for measuring the energy consumption of a program code: using an instruction-level power model, using Linux command-line tools, and using power analyser devices.

5.1 Introduction

Constrained devices are small and low-resource devices constrained in memory, computational capabilities and power. They are often used as sensors or smart end nodes. These devices are either battery-powered, or they harvest their energy from the surroundings. Implementing cryptography on low power devices is challenging as the resources needed to run the cryptographic primitives with acceptable performance is not always available. In addition, the amount of energy stored on the constrained devices' batteries is minimal, so it is essential to conserve their power and avoid the frequent recharging or replacing of the batteries. In this chapter and the next one, we will measure and try to understand the performance of the conventional and lightweight cryptographic primitives when run on resource-constrained devices. We will also measure and investigate the energy consumption of the primitives when run on these devices.

This chapter is structured as follows; Section 5.2 discusses three methods for measuring the energy consumption of a program code; using a power model, using Linux tools and using multimeters. Section 5.3 presents the experiment setup and describes how we conducted our investigation. The results of evaluating the throughput and energy consumption of block and stream ciphers are presented in Section 5.4.1 and Section 5.4.2. Finally, we conclude the chapter in Section 5.5.

5.2 Measuring the Power Consumption of the Primitives

Electrical power (*P*) is the amount of electrical energy produced, absorbed, transferred or converted per unit of time [143]. The SI (International System of Units) unit of power is the *Watt*, which is equivalent to one *joule* per second. Power consumption is a metric often used with constrained devices that harvest their power from the environment [118]. Power is measured using the equation:

$$P(watts) = V(volts) \times I(amps)$$

V is the applied voltage and I is the amount of current drawn from the power source. Both quantities can be measured using a multimeter.

Energy consumption is the power consumption over a certain period and is expressed in *Joules*. It is more suitable for devices with a fixed amount of stored energy (e.g. battery-powered devices) [138]. Energy consumption is calculated using the equation:

$$E(joules) = P(watts) \times T(seconds)$$

T is the program execution time and is measured either using a high-resolution clock such as `high_resolution_clock` clock in C/C++ or using the equation [126, 154]:

$$T = N \times \tau$$

N is the number of clock cycles taken by a program to execute and can be measured using RDTSC (Read Time-Stamp Counter) instruction. τ is the CPU clock period and its value is published information. For cryptographic primitives, another way to find the number of clock cycle is using the following equation [159]:

$$\text{Cycles per byte} = \frac{\text{CPU clock speed in hertz}}{\text{Primitive's throughput in byte per second}}$$

Using this equation, one should note that the current CPU speed may not be the same as the advertised speed due to CPU frequency scaling. The Linux `lscpu`, `lshw` and `dmidecode` commands are helpful tools to check for the current CPU speed, as in listing 5.1.

```
  $ sudo dmidecode -t processor | grep Speed
Max Speed: 3312 MHz
Current Speed: 1600 MHz
```

Listing 5.1 Current CPU speed vs advertised speed

The energy consumption equation can be re-written as follows:

$$E = V \times I \times N \times \tau$$

V and τ values can be obtained from the processor's manual, and N can be calculated as discussed above. Whereas finding I requires extra effort, as we will see shortly.

Measuring Power Consumption Using Instruction-Level Power Model

Naik and Wei [126] present a methodology for analysing software energy usage based on a model that quantifies the energy cost of machine instructions. The authors calculate the energy cost of individual instructions, and by summing up these costs, they obtain the total energy consumption of a program. The authors first compiled a small C program into assembly language. Then, considering that different machine instructions draw different amounts of current and sometimes take a different number of clock cycles to execute, they multiplied the number of cycles the instruction takes by the current it draws to measure its energy cost. The authors obtained both quantities (the current and number of cycles) from the CPU manufacturer. The calculated costs, which are in *ampere-cycles*, are converted into *ampere-seconds* by dividing each cost by the CPU clock frequency in *cycle/seconds*. The sum of all instructions costs give the total energy cost of the program. The authors did not use the voltage measurements in their calculations of the energy, which is why the energy costs are in *ampere-second* and not *joules*. Although this methodology is useful to analyse the energy cost at the instruction level, it is not easy to use when we have a large program. Also, it requires we know the amount of the current drawn by each machine instruction and the number of cycles it takes, both of which are not always published by the processors' manufacturers.

Nikolaidis *et al.* [129] explain how to measure the instantaneous current drawn by the processor during the execution of each machine instruction using a devised measurement circuitry and an automated data acquisition setup. They also use an a high-speed oscilloscope to record the instantaneous supply voltage. By obtaining the voltage and the current for each clock cycle, they measure the energy consumption at a clock cycle.

Minaam *et al.* [120], calculate the energy consumption of some cryptographic primitives. The authors first calculate the number of clock cycles each primitive takes to encrypt one byte. Then they multiply the number of cycles obtained with the average current drawn by each clock cycle. The authors had to estimate the current drawn by each clock cycle as its value was not published for the CPU used for the test. The result, which is in *ampere-cycle*, is then converted into *ampere-seconds* by dividing it by the CPU clock frequency in *cycles/second*. Finally, the result is multiplied by the CPU operating voltage to get the energy cost of encrypting one byte in *joules*.

Measuring Power Consumption Using Linux Command-Line Tools

An easy but less accurate way of measuring the energy consumption of a program is to use the tools provided by some operating systems for measuring the power and energy consumption. The idea is to measure the energy consumption while the system is idle to establish a baseline. Then subtract the baseline consumption from the overall consumption measured while the program is running. The difference is the program energy consumption.

Linux supports Intel's RAPL (Running Average Power Limit), which offers a set of counters that provide information on energy and power usage of the CPU, RAM, and on-chip GPU [84]. RAPL uses a software power model for estimating the energy usage [163]. On some systems where there is a hardware support for RAPL, the power measurements are actual [51]. RAPL counters are updated every millisecond [84], and no timestamp is provided [70]. This makes obtaining useful results at small timescales difficult. As it is not indicated when the current measurement began, mapping the reading to executing code is almost impossible [51]. The accuracy of the power measurements estimation provided by RAPL has been measured and validated in different ways and on many different processors, see [51, 70, 71, 96, 144]. It appears that the accuracy can vary based on the type of processor and workloads. However, RAPL shows a promising accuracy, and its estimation is sufficiently good.

Powerstat [97] is one of the Linux tools that calculates the system's power consumption while running on a battery. By default, the tool waits 180 seconds for the battery readings to stabilise, then starts measuring the power consumption every 10 seconds for 300 seconds (30 samples), see Listing 5.2. The user can change the waiting time, the number of samples and the intervals.

```
    $ powerstat
Running for 300.0 seconds (30 samples at 10.0 second intervals).
Power measurements will start in 180 seconds time.
```

```

  Time     User  Nice   Sys  Idle    IO  Run Ctxt/s  IRQ/s  Watts
19:00:27   1.6   0.0   1.7  96.7   0.0    1    961    372   6.16
19:00:37   1.6   0.0   1.8  96.6   0.0    1    972    400   6.16
19:00:47   1.9   0.0   1.2  96.9   0.0    1    944    367   6.16
19:00:57   1.7   0.0   1.4  96.9   0.0    1    953    367   6.16
```

Listing 5.2 Output of the *powerstat* tool

At the end of a run, the tool displays the average and standard deviation of the measurements, see Listing 5.3. To calculate the energy consumption of a program, we should run *powerstat* twice; when the system is idle, and another time when our program is running. The difference between the two measurements is our program's power consumption. Multiplying the measured power consumption (*watt*) by the program's execution time gives the energy consumption in *joules*. The energy consumption of cryptographic primitives is usually measured per byte, so we divide the calculated energy consumption by the primitive's input size to get the energy consumption per byte (*joules/byte*).

```
-------- ----- ---- ---- ---- ----  ------ ----- ----- -----
 Average   1.7   0.0   1.4  96.9   0.0  1.4  985.3  393.2   5.96
 GeoMean   1.7   0.0   1.3  96.9   0.0  1.2  983.7  391.1   5.95
  StdDev   0.2   0.0   0.3   0.4   0.0  1.0   58.3   43.3   0.48
-------- ----- ---- ---- ---- ----  ------ ----- ----- -----
 Minimum   1.5   0.0   1.0  95.7   0.0  1.0  934.0  349.6   5.36
 Maximum   2.2   0.0   2.1  97.5   0.1  5.0 1118.2  494.3   6.79
-------- ----- ---- ---- ---- ----  ------ ----- ----- -----
Summary:
System:   5.96 Watts on average with standard deviation 0.48
```

Listing 5.3 Statistics from the *powerstat* tool

Another useful Linux tool for measuring energy consumption is **upower** [81]. The tool displays status and statistical information about the battery as seen in Listing 5.4. The *energy* (Line 14), and the *energy-rate* (Line 18) are the two relevant properties that used to measure the energy consumption. The *energy* property stores the amount of energy currently available in the battery in *Watt-hour*. To use this property to measure the energy consumption of a program, we should do the following:

1. Run the *upower* command and record the value of *energy*.

2. Wait for some time (e.g. 30 seconds), run the *upower* command again, and record the new *energy* value.

3. Subtract the two values of the *energy* property to get the system's energy consumption over the time period you have chosen. This value is the baseline energy consumption.

4. Start your program and run the *upower* command and record the value of *energy*.

5. Wait for the same time as in step 2, then run the *upower* command again and record the new *energy* value.

6. Subtract the two energy values to get the total energy consumption (the system + your program) over the time chosen.

7. Subtract the baseline energy consumption from the total energy consumption to get your program energy consumption over the time chosen.

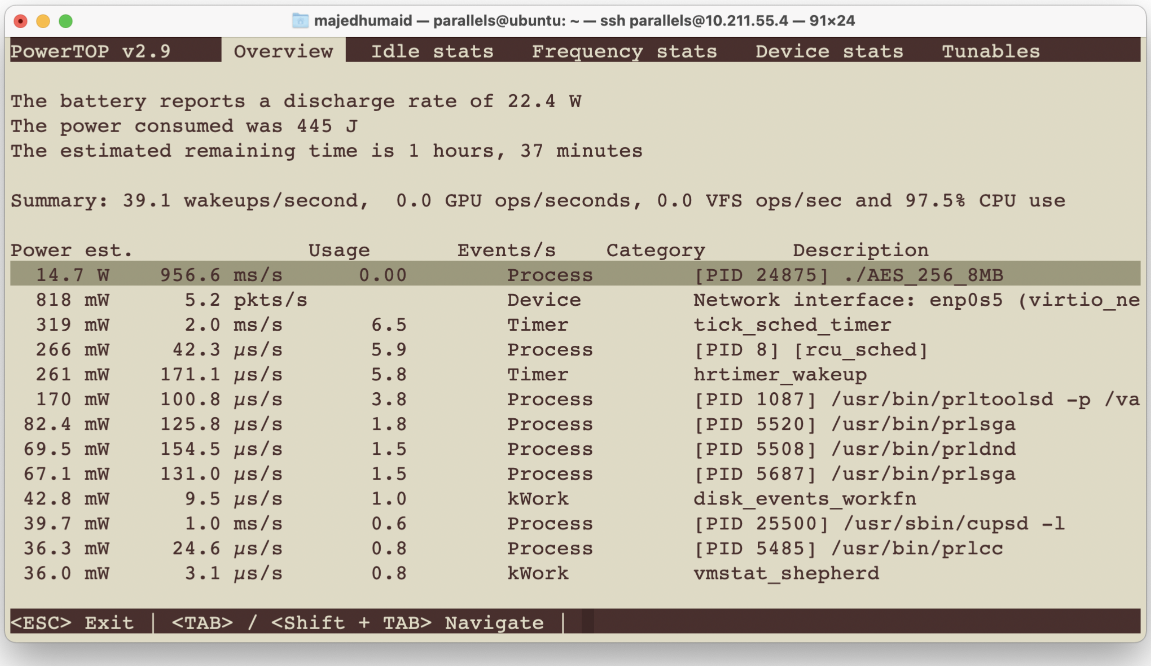
8. Divide your program's execution time by the time you selected in Step 2 and multiply the result by the difference you got from Step 7 to get your program's energy consumption in *Wh*.

```
$ upower -i /org/freedesktop/UPower/devices/battery_BAT0
  native-path:          BAT0
  vendor:               SMP
  model:                bq20z45
  power supply:         yes
  updated:              Sat 01 May 2021 04:50:30 PM BST
  has history:          yes
  has statistics:       yes
  battery
    present:             yes
    rechargeable:        yes
    state:               discharging
    warning-level:       none
    energy:              66.95 Wh
    energy-empty:        0 Wh
    energy-full:         100 Wh
    energy-full-design:  100 Wh
    energy-rate:         8.13 W
    voltage:             11.715 V
    time to empty:         8.2 hours
    percentage:            66%
    capacity:              100%
    icon-name:            'battery-full-symbolic'
```

Listing 5.4 Output of the *upower* tool

The *energy-rate* property stores the amount of energy that is being drained from the battery in *Watts*. This property should be read when the system is idle and again when our program is running. The difference between the two readings is our program's energy consumption in *Watts*. Multiplying the result by the program's execution time gives us the energy consumption in *joules*.

PowerTOP [140] is another Linux diagnostic tool that helps monitor the power usage of programs when the system is running on battery. A screenshot of *PowerTOP* is shown in Figure 5.1. The energy consumed since the start of the tool is displayed in *joules* in the second line. The *power est*imation column shows the instantaneous power consumption of every program and process currently running in the system in *Watts* or *Milliwatts*.


majedhumaid — parallels@ubuntu: ~ — ssh parallels@10.211.55.4 — 91×24

```
PowerTOP v2.9      Overview   Idle stats   Frequency stats   Device stats   Tunables

The battery reports a discharge rate of 22.4 W
The power consumed was 445 J
The estimated remaining time is 1 hours, 37 minutes

Summary: 39.1 wakeups/second,  0.0 GPU ops/seconds, 0.0 VFS ops/sec and 97.5% CPU use

Power est.              Usage       Events/s    Category       Description
  14.7 W     956.6 ms/s     0.00          Process        [PID 24875] ./AES_256_8MB
  818 mW       5.2 pkts/s                 Device         Network interface: enp0s5 (virtio_ne
  319 mW       2.0 ms/s       6.5         Timer          tick_sched_timer
  266 mW      42.3 µs/s       5.9         Process        [PID 8] [rcu_sched]
  261 mW     171.1 µs/s       5.8         Timer          hrtimer_wakeup
  170 mW     100.8 µs/s       3.8         Process        [PID 1087] /usr/bin/prltoolsd -p /va
 82.4 mW     125.8 µs/s       1.8         Process        [PID 5520] /usr/bin/prlsga
 69.5 mW     154.5 µs/s       1.5         Process        [PID 5508] /usr/bin/prldnd
 67.1 mW     131.0 µs/s       1.5         Process        [PID 5687] /usr/bin/prlsga
 42.8 mW       9.5 µs/s       1.0         kWork          disk_events_workfn
 39.7 mW       1.0 ms/s       0.6         Process        [PID 25500] /usr/sbin/cupsd -l
 36.3 mW      24.6 µs/s       0.8         Process        [PID 5485] /usr/bin/prlcc
 36.0 mW       3.1 µs/s       0.8         kWork          vmstat_shepherd

<ESC> Exit | <TAB> / <Shift + TAB> Navigate |
```

Fig. 5.1 Screenshot of *PowerTOP* tool

Perf [91] is a powerful profiling tool that is included in the Linux kernel. It provides a rich set of commands and options for collecting performance counters and analysing performance. The tool supports a list of measurable events coming from different sources. Some events come from the CPU and its Performance Monitoring Unit (PMU). These are called hardware events, and they differ depending on the CPU type and model. Other events are kernel counters and called software events [62]. A list of all supported events can be obtained using the command `perf list`.

The *power/energy-cores/* events are kernel's counters that stores the CPU cores power consumption information. In Listing 5.5, we used the command *stat* to read performance data from these counters. The *stat* command executes an application and collects performance counter statistics about it [92]. The results show the energy consumption of the executed program in *joules*

```
$ sudo perf stat -a -e power/energy-cores/ ./main
Performance counter stats for 'system wide':

             6.53 Joules power/energy-cores/

      0.562494102 seconds time elapsed
```

Listing 5.5 Output of the *perf* tool

TLP [108] is a Linux powerful command-line tool for advanced power management that helps in optimizing laptop battery life. The `tlp-stat` command [109], which is part of TLP, is used for collecting information from different hardware and kernel counters about the processor, battery, disk, and other objects. The data from the battery object has two relevant properties; *energy_now* and *power_now*. The *energy_now* keeps track of how much energy is currently available in the battery. The following steps show how to measure the energy consumption of a program:

1. Make sure the device is running on battery.
2. Run `tlp-stat -b` and record the energy currently stored in the battery.
3. Leave the system idle for a certain amount of time (e.g. 60 seconds), run `tlp-stat -b` again and record the amount of energy currently available in the battery.
4. The difference between the two readings is the energy consumption of the idle system (the baseline energy consumption).

5. Run `tlp-stat -b` and record the energy in the battery.
6. Run your program for the same period of time as in Step 2, then run `tlp-stat -b` and record the energy in the battery.
7. The difference between the two readings is the whole system energy consumption in mWh.
8. Subtract the difference in Step 7 from the baseline energy consumption to get your program energy consumption over the time chosen in Step 2.

The *power_now* property stores the instantaneous power consumption of the system in mW. The difference between the values of this property taken when the system is idle and when the program is running is the program's power consumption in mW.

```
$ sudo tlp-stat -b
--- TLP 1.1 --------------------------------------------

+++ Battery Status
/sys/class/power_supply/BAT0/manufacturer                  = SMP
/sys/class/power_supply/BAT0/model_name                    = bq20z45
/sys/class/power_supply/BAT0/cycle_count                   = (not
  supported)
/sys/class/power_supply/BAT0/energy_full_design            = 100000 [
  mWh]
/sys/class/power_supply/BAT0/energy_full                   = 100000 [
  mWh]
/sys/class/power_supply/BAT0/energy_now                    =  93160 [
  mWh]
/sys/class/power_supply/BAT0/power_now                     =   6821 [mW
  ]
/sys/class/power_supply/BAT0/status                        =
  Discharging

Charge                                                     =   93.2 [%]
Capacity                                                   =  100.0 [%]
```

Listing 5.6 The output of the *tlp-stat -b* command

Measuring Power Consumption Using Power Analysers

A third method of measuring the energy consumption of a program code is to use a multimeter, oscilloscope or specialised power analyser. These measurement devices measure the electrical properties, including the voltage and the current drawn from the power supply. By obtaining the measurements of these two properties, it is easy to measure the power consumption of the program code. The oscilloscope and the specialised power analyser offer more accurate results as their sampling rate is much higher than the multimeters. A higher sampling rate allows for a detailed analysis of the power traces of the cryptographic building blocks, such as round functions, key expansion and substitution box implementations. The two devices also provide their users with the ability to upload the results to a computer for further analysis of the program's power traces. The oscilloscopes and the specialised power analysers are very costly, making the multimeter a prefered measurement tool for many researchers. Another downside of using these measurement instruments is they require a fair amount of knowledge and experience with electronics and electrical engineering to use them.

USB multimeter is a simple and cheap alternative to regular multimeters. It does not require any knowledge of electronics to use. The device measures the power consumption as the electric power passes through it. The device, however, has a very slow sampling rate; most we have seen have a measuring speed of 0.5 times/second. With such speed, it is not feasible to measure the power consumption of individual building blocks of the primitives. Additionally, they only work with devices that draw power from an electrical outlet or an external battery.

5.3 Experiment Setup

The throughput and the energy consumption of the primitives are tested on a Raspberry Pi 3, Figure 5.2 [17]. The device has a Quad-Core ARM® Cortex®-A53 CPU running at 1.2 GHz with 32 kB Level 1 and 512 kB Level 2 cache memory. The Cortex®-A53 is a low-power processor that implements the Armv8-A instruction set. The Armv8-A has a Cryptography Extension that accelerates the AES cipher (encryption and decryption) and SHA-1, SHA-224 and SHA-256 hash functions. The Cryptography Extension, however, is optional [14] and not included in Raspberry Pi 3. ARM supplies the Cryptography Extension under an additional license to Cortex®-A53 processors [14]. Linux commands such as `lscpu`, `grep flags /proc/cpuinfo` or `sort -u /proc/crypto | grep module`; see Listing 5.7, can be used to check if a CPU has support for the cryptography extension.

```
$ sort -u /proc/crypto | grep module
   module       : aesni_intel
   module       : aes_x86_64
   module       : crc32_pclmul
   module       : crct10dif_pclmul
   module       : ghash_clmulni_intel
   module       : kernel
   module       : pcbc
```

Listing 5.7 CPU Cryptography extension support

Fig. 5.2 Raspberry Pi 3.

We used the GoogleBenchmark library [72] to measure the execution times of the primitives. The execution time includes the time of cipher initialisation, encryption and decryption of variable size buffers. Inputs of sizes 1 MB, 4 MB, and 8 MB are used to test if the primitives provide consistent performance under different data loads. The execution time is then used to calculate the primitives' throughput according to the following formula:

$$\textbf{Througput} = \frac{\textit{Data Size (MB)}}{\textit{Execution Time (seconds)}}$$

Library	Version
Crypto++	5.6.5
Botan	2.3
OpenSSL	1.1.1
Libgcypt	1.8.0
Nettle	3.4
LibTomCrypt	1.18.1

Table 5.1 Cryptographic libraries' versions

We used the six cryptographic libraries from chapter 4 in our benchmarking test. GNU GCC Raspbian 6.3.0 is used to compile the code, and none of the optimisation flags is used in the test. Table 5.1 shows the versions of the libraries tested.

The power and energy consumption of the primitives are measured using a USB digital power multimeter from COOWOO [42], Figure 5.3 [42]. The multimeter has one Type-A male USB connector that we plugged into the Raspberry Pi and one female Type-A USB port we connected to the power source. The multimeter measures voltage V and amperage I as the electric power passes through it. The power consumption ($V \times I$) is calculated and displayed on the multimeter screen in *watts*. Since the multimeter only calculates the overall power usage (Raspberry Pi + the primitive's implementation), we had first to find out how much power consumed by the Raspberry Pi (its OS, System on a Chip (SoC), and on-board peripherals). The result is then subtracted from the total power consumed during the execution of each primitive to obtain the primitive's power consumption. The measuring speed of the USB power multimeter is 0.5 times/second, which is slower than the execution times for almost all the primitives. For this reason, we run each primitive in a loop for a few seconds to measure its power consumption.

The energy consumption of a primitive is calculated in *joules* according to the following formula:

$$\text{Energy Consumption } (\textit{joules}) = \textit{The primitive's Execution Time (seconds)} \times \textit{Power (watts)}$$

We unplugged all the peripheral devices from the Raspberry Pi to reduce any noises when measuring the power consumption. We enabled the Wi-Fi and accessed the device via SSH terminal.

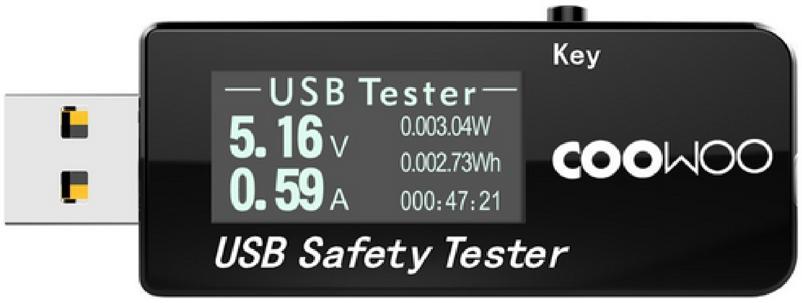


Fig. 5.3 USB digital multimeter

5.3.1 GoogleBenchmark Library

The GoogleBenchmark library [72] is an open-source library for benchmarking C++ code snippets. The library provides several functions that help with deep performance investigation. Furthermore, if the underlying architecture features a Performance Monitoring Unit, the library allows its users to request a list of performance counters such as CPU cycle count, number of instructions executed, and power state.

To use the library, we should first include the *benchmark* header file. Then register the function we want to benchmark using the *BENCHMARK* macro and use the BENCHMARK_MAIN function as shown in Listing 5.8.

```

#include <benchmark/benchmark.h>

void testAES_256()
{
    // Encryption
    struct CBC_CTX(struct aes_ctx, AES_BLOCK_SIZE) enc_ctx;
    aes_set_encrypt_key(&enc_ctx.ctx, 32, key);
    CBC_SET_IV(&enc_ctx, iv);
    CBC_ENCRYPT(&enc_ctx, aes_encrypt, datalength, plaintext, ciphertext);

    // Decryption
    struct CBC_CTX(struct aes_ctx, AES_BLOCK_SIZE) dec_ctx;
    aes_set_decrypt_key(&dec_ctx.ctx, 32, key);
    CBC_SET_IV(&dec_ctx, iv);
    CBC_DECRYPT(&dec_ctx, aes_decrypt, datalength, ciphertext, decrypted_text);
}

// Register the testAES_256 function as a benchmark
BENCHMARK(testAES_256);
```

```

BENCHMARK_MAIN();
```

Listing 5.8 Using *GoogleBenchmark* library

To run the benchmark, we should compile and link our program against the *benchmark* library in addition to *pthread* library as shown in Listing 5.9.

```
$ g++ botanTest.cpp -o botanTest  -I/pi/include -Lbenchmark/build/src -
   lbenchmark -lpthread -std=c++14
```

Listing 5.9 Compiling the benchmark program

The results from the benchmark test are displayed in a table that, by default, has four columns, as shown in Listing 5.10. The Benchmark column shows all the benchmarked functions. The execution times (both elapsed time and CPU time) are shown in the Time column and CPU column, respectively. The number of iterations for each benchmarked function is shown in the Iterations column. The library allows for a custom report generation.

```
  $ ./botanTest
2021-05-09 16:01:23
Running ./botanTest
Run on (4 X 1200 MHz CPU s)
Load Average: 0.00, 0.07, 0.11
-------------------------------------------------------
Benchmark            Time             CPU   Iterations
-------------------------------------------------------
AES128             905 ms          904 ms            3
AES192            1005 ms         1005 ms            3
AES256            1106 ms         1106 ms            3
Twofish128         424 ms          424 ms            4
Twofish192         423 ms          423 ms            4
Twofish256         423 ms          423 ms            4
```

Listing 5.10 *GoogleBenchmark* results report

The library runs each benchmarked function for a similar amount of time, long enough to obtain stable results. The time can also be adjusted manually. [72]. The number of iteration can be changed using the -benchmark_repetitions command-line option as in Listing 5.11. In this case, the library displays the mean, median, and standard deviation of the measurements.

```
  $ ./botanTest  --benchmark_repetitions=50 --
  benchmark_report_aggregates_only=true
2021-05-09 16:28:22
Running ./botanTest
Run on (4 X 1200 MHz CPU s)
Load Average: 0.00, 0.00, 0.00
-------------------------------------------------------------
Benchmark                     Time             CPU   Iterations
-------------------------------------------------------------
AES128_mean                 905 ms          905 ms           50
AES128_median               905 ms          905 ms           50
AES128_stddev             0.329 ms        0.218 ms           50
```

Listing 5.11 *GoogleBenchmark* statistics

Running the benchmark while the CPU frequency is set to a governor other than *Performance* can add unnecessary noise to the measurements. For example, the *on-demand* governor scales the CPU frequency dynamically according to the current load. To disable the CPU frequency scaling, we use the cpupower command as in Listing 5.12.

```
  $ sudo cpupower frequency-set --governor performance
Setting cpu: 0
Setting cpu: 1
Setting cpu: 2
Setting cpu: 3
```

Listing 5.12 Setting the CPU frequency

5.4 Evaluation

In the following two sections, we will discuss the performance of the symmetric primitives from the six cryptographic libraries from chapter 4. We will first discuss the primitives' throughput and then their power consumption.

The Cortex$^{®}$-A53 CPU we used in our experiment does not support any of the hardware acceleration techniques mentioned in Table 4.2 in chapter 4. So the testing results show the performance of the generic implementation of the primitives only and not the hardware optimised versions. The Linux command `lscpu | grep Flags` can be used to list all the features that the current CPU supports, as shown in Listing 5.13.

```
 pi@raspberrypi:~ $ lscpu | grep Flags
     Flags: half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva
idivt vfpd32 lpae evtstrm crc32
```

Listing 5.13 Checking the available CPU features

In the absence of hardware acceleration support, each library has a fallback mechanism for a software optimised implementation. The level of optimisation for the primitives' implementations varies between libraries as the level of experience, skills, and competencies varies between the development teams. Additionally, code optimisation requires performing difficult trade-offs choices (e.g speed versus code clarity and conciseness) and decisions, which are different among libraries. The level of optimisation is also different inside the same library. For example, some libraries, because of the overhead involved, only offer optimisation for the most commonly used ciphers.

5.4.1 Block Ciphers

We tested all the block cipher implementations from the six cryptographic libraries using the Cipher Block Chaining (CBC) mode of operation.

Primitives' Throughput

The throughput results are shown in Figures 5.4 and 5.5. For clarity, AES is split from the rest of the common ciphers and drawn in a separate chart in Figure 5.4. AES is the block cipher that benefits the most from the hardware acceleration techniques, as we have seen in chapter 4. The fastest AES (960 MB/s) was about six times faster than the next fastest block cipher (167 MB/s). However, with the absence of any hardware acceleration support, AES's performance becomes comparable to that of the other block ciphers, as shown in Figure 5.4.

OpenSSL implementation of AES outperforms the other libraries implementations of the cipher. Being one of the leading cryptographic libraries that have been prevalent across different architectures, it seems that the library developers have invested a lot of effort to optimise their implementations for the different types of processors. In the absence of

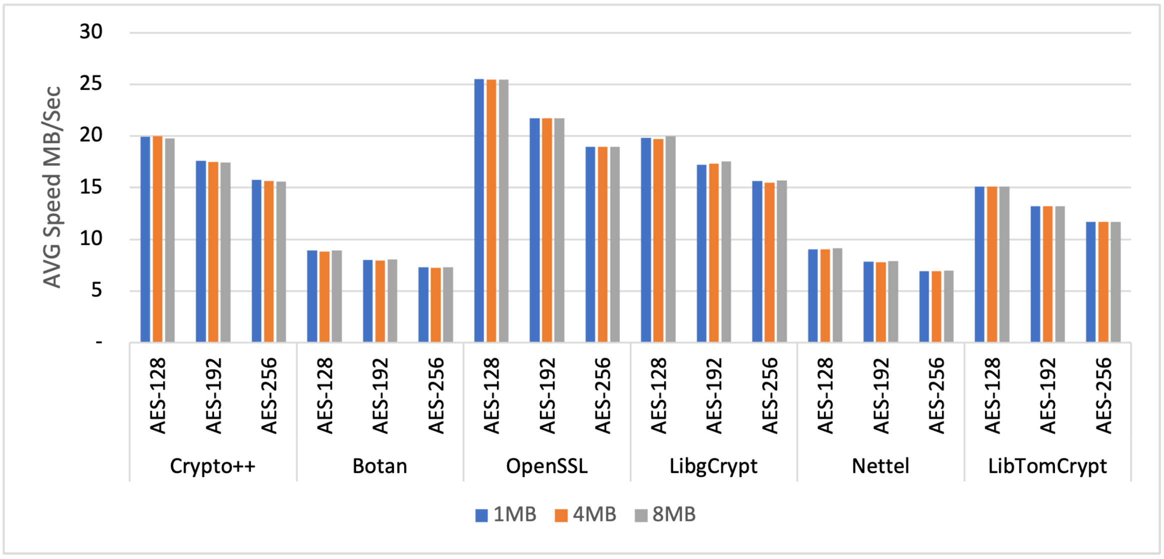


Fig. 5.4 Average speeds of AES implementations.

hardware acceleration features, the OpenSSL provides an optimised implementation for AES using assembly language, compressed substitution boxes, unrolled loops, and avoiding L1 cache aliasing [133]. LibgCrypt and Crypto++ implementations of AES come second with comparable performance. Both libraries provide optimised implementations using assembly code for the ARM architecture. Botan's and Nettle's implementations are the slowest. Both implementations have similar performance and are about three times slower than OpenSSL's. In the previous experiment, Botan has shown a fast implementation for AES using the SSSE3, SIMD and AES-NI instruction sets; however, none of these acceleration features is available or enabled for our ARM processor.

Figure 5.5 shows the performance of the remaining block ciphers that are common across the six libraries. Although Blowfish has a very slow key setup (it uses a large number of subkeys that must be pre-computed before any encryption or decryption [147]), the cipher outperforms all the common block ciphers including Camellia that was among the fastest block ciphers in the previous experiment. OpenSSL and LibgCrypt implementations of Blowfish are the fastest implementations. Both are optimised using ARM assembly code. In addition, OpenSSL uses the register keyword to assign live variables to registers instead of memory to make access to these variables as fast as possible. Camelia-128 and CAST-128 achieve comparable performance in all libraries except Botan and Nettle. Camellia's performance can be considerably improved using AES-NI and AVX instruction sets, as we have seen in the previous experiment. However, the absence of these feature in our ARM processor has negatively affected the cipher's performance. Camellia-128 is composed of fewer rounds (18 rounds) than Camellia-192 and Camellia-256 (24 rounds) and hence the

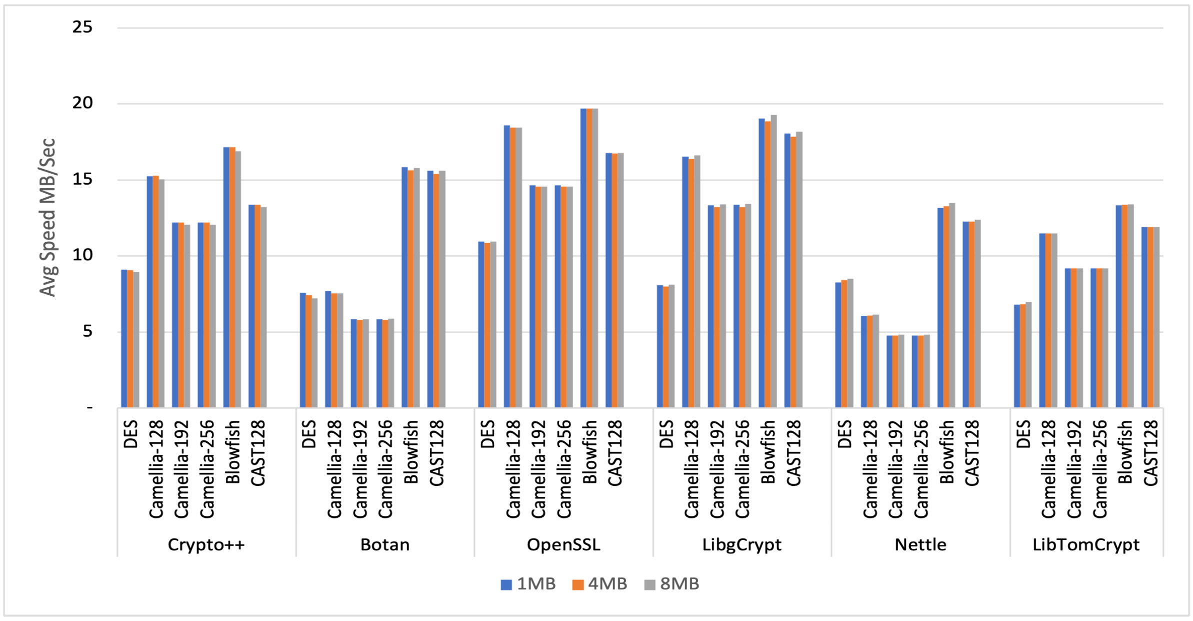


Fig. 5.5 Average speeds of the common block ciphers.

difference in the performance. Nettle's implementation of the Japanese cipher Camellia is the least efficient; it is three times slower than OpenSSL's. Nettle is one of the libraries that does not offer a good optimisation for the primitives that are less commonly used. CAST-128 is based on Feistel structure (as Camellia) and has 64 bits block size, 12 or 16 rounds and supports keys of size 40 to 128 bits [2]. OpenSSL and LibgCrypt provide optimisations for the cipher using ARM assembly code. It is no surprise that DES shows a weak performance; the cipher has a small block size (64-bits) and bit-oriented operations that are not software friendly.

Figures 5.6 to 5.11 show the performance of the block ciphers that are not common across all libraries. SHACAL-2, which is implemented only in Crypto++ (Figure 5.6) and Botan (Figure 5.7), is the second-fastest block cipher after AES. As we saw in the previous experiment, the cipher's implementation is optimised using SSSE3, SHA-NI, SIMD and AVX2 instruction sets. However, none of these features is available or enabled in our ARM processor, although the cipher is still performing well. SHACAL-2 is a 256-bit block cipher that is based on the SHA-256 hash function. The cipher is composed of 64 rounds and supports keys of length up to 512-bit [162]. Crypto++ implementations used parts of Botan's implementation with a little tweaking [46]. Both implementations have similar performance.

RC6 and SAFER+ are the third-fastest ciphers in our experiment. Both ciphers achieved the same speed. SAFER+ is only available in LibTomCrypt (Figure 5.11), while RC6 is available in LibTomCrypt and Crypto++ (Figure 5.6). RC6 is a very simple 128-bit block

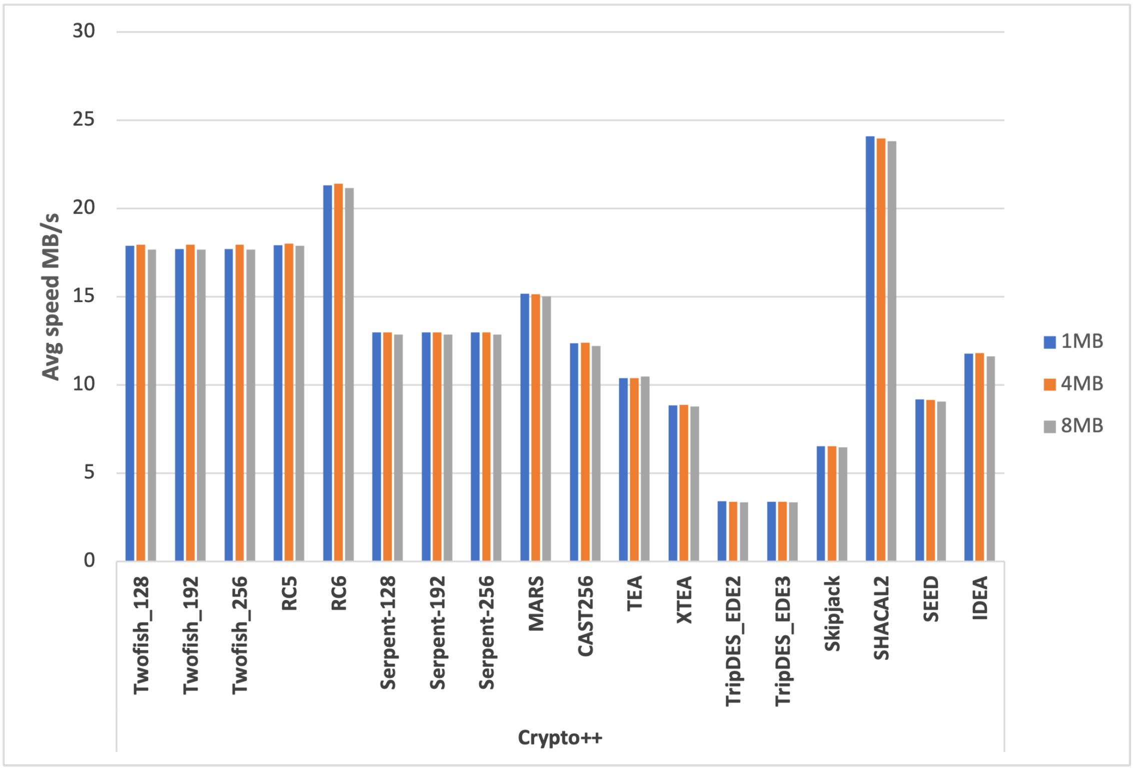


Fig. 5.6 Average speeds of Crypto++ block ciphers.

cipher [142] that is based on a Feistel structure. It is composed of 20 rounds and supports key sizes of 128-bit, 192-bit, and 256-bit. SAFER+ is a 128-bit block cipher consisting of 8-16 round functions and supports key sizes of 128-bit, 192-bit and 256-bit. SAFER+ key schedule is quite simple [123].

Twofish from LibgCrypt (Figure 5.9), and Botan (Figure 5.7), is the fourth-fastest cipher in our test. Both implementations have similar performance. In the previous experiment, the cipher was optimised using AVX2 acceleration instruction. However, in the absence of AVX2, the two libraries use pre-computed lookup tables to speed up the execution of the cipher. The pre-computed tables offer a good speed but consume a large amount of memory. Twofish is a 128-bit block size composed of 16 rounds and supports key sizes of 128-bit, 192-bit and 256-bit [149]. It is implemented in all the libraries except OpenSSL.

RC5 is the fifth-fastest cipher in our experiment. The cipher is only available in Crypto++ and LibTomCrypt. Both libraries offer implementations with similar performance. RC5 is a simple block cipher with variable block sizes of 32, 64, and 128 bits. It has a variable number of rounds that range from 1-255 and supports keys of sizes 0 to 2040-bits [141].

The slowest cipher in our test is Triple-DES. The cipher shows a similar performance across all the libraries that implement it. The cipher's performance is negatively affected by

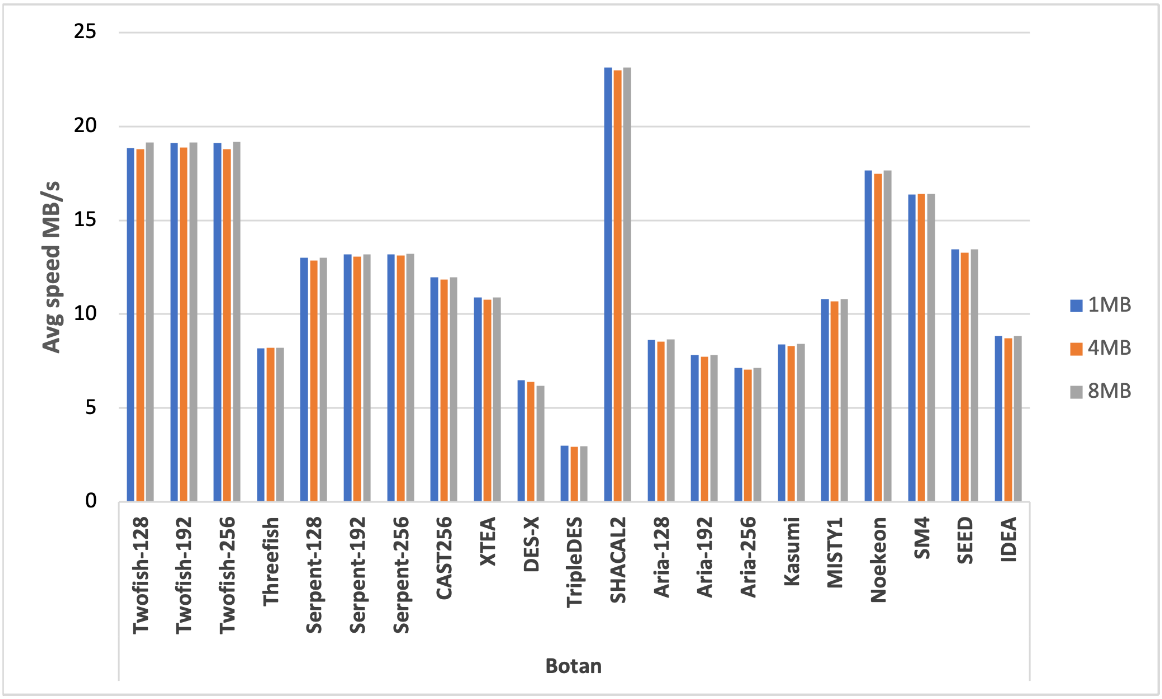


Fig. 5.7 Average speeds of Botan block ciphers.

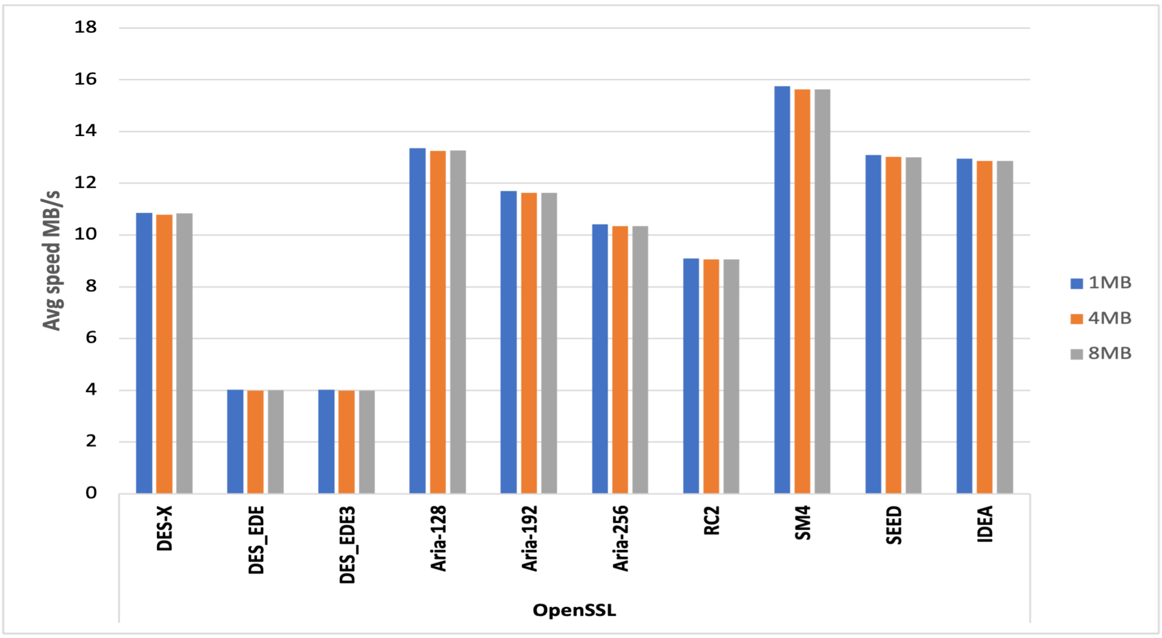


Fig. 5.8 Average speeds of OpenSSL block ciphers.

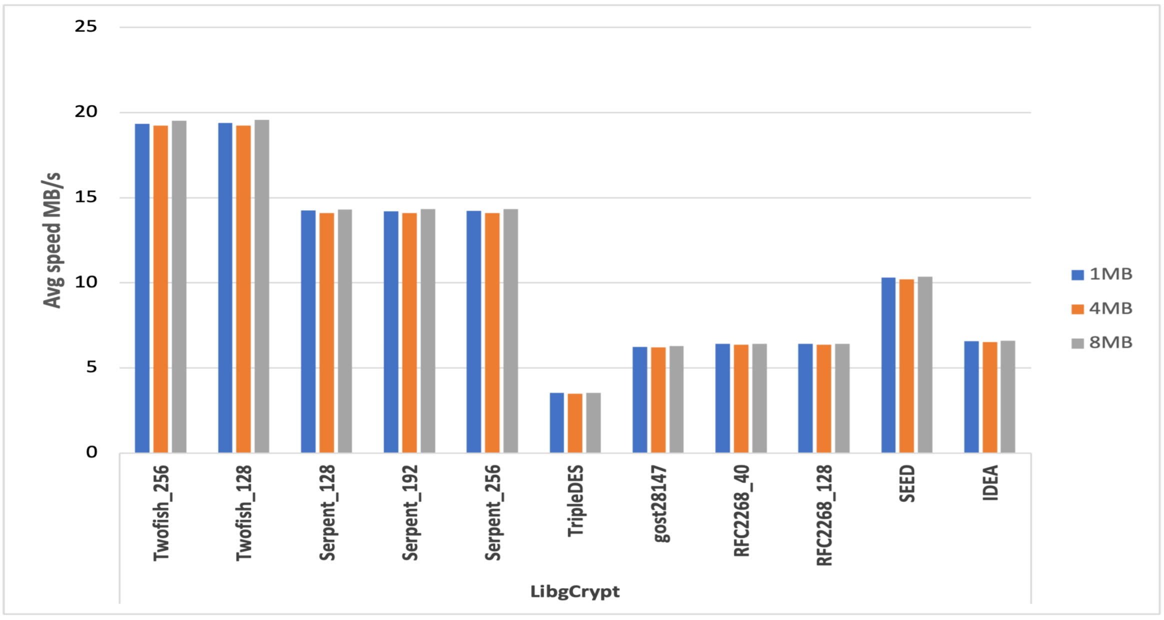


Fig. 5.9 Average speeds of LibGCrypt block ciphers.

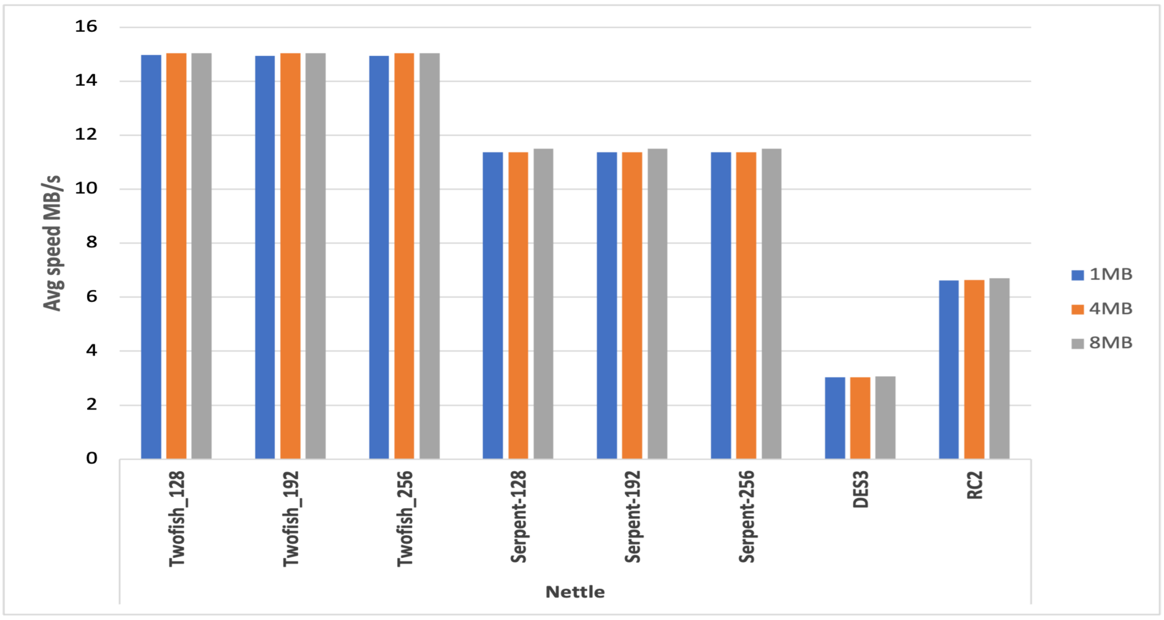


Fig. 5.10 Average speeds of Nettle block ciphers.

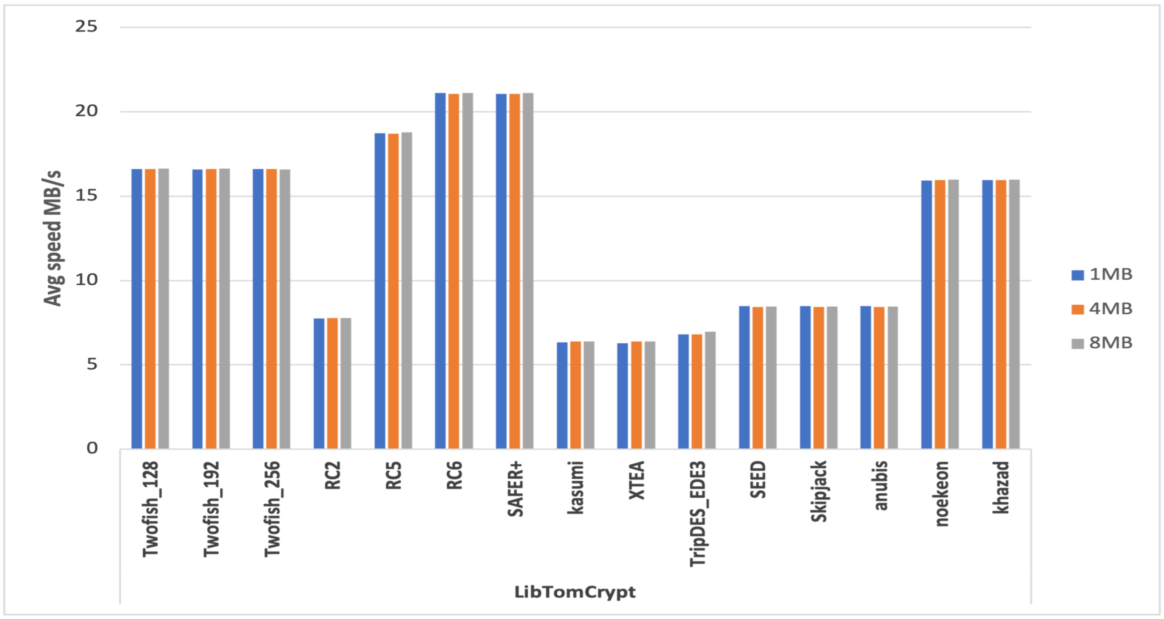


Fig. 5.11 Average speeds of LibTomCrypt block ciphers.

its small block size (64-bit) and large number of rounds (48) in addition to its bit-oriented operations that are not suited for a software implementation.

Energy Consumption

In this experiment, we measured the instantaneous electrical power (P) delivered to the Raspberry Pi by multiplying the amount of voltage applied (V) and the current drawn from the power supply (I); V and I are measured using the USB multimeter. We then subtract from P the baseline consumption (1.24 watt) to get the primitive's power consumption. After that we calculated the energy consumption (E) of each primitive according to the following formula:

$$\text{E}\ (joules) = Primitive\text{'}s\ Execution\ Time\ (seconds)\ \times Primitive\text{'}s\ Power\ Consumption\ (watts)$$

Energy consumption is a more relevant metric to evaluate the primitive's performance as it considers the primitive's execution time and measures the total power consumed during that time.

The majority of the block cipher power consumption comes from the implementation of its substitution box (S-Box) [1, 16, 19]. This is due to the size and complexity of S-Boxes. The key length and the block size of the primitive have no significance in its power consumption, as we see in this experiment. However, both properties have an impact on the speed of the execution and thus the energy consumption.

In this experiment, we found that most of the tested primitives have, to a certain degree, a comparable power consumption. This makes the execution time the most critical factor in determining the energy consumption of the primitives. Figure 5.12 shows the energy consumption of the common block ciphers in *μjoules* per byte. The figure shows that the fast implementations of AES from OpenSSL, LibgCrypt, and Crypto++ consumed the least amount of energy compared to the slow implementations from the remaining three libraries. OpenSSL, LibgCrypt, and Crypto++ offer optimised implementations using ARM assembly code, pre-computed AES look-up tables, and unrolled loops for AES rounds operations. These optimisation techniques speed up the execution of AES and consequently reduce its energy consumption. On the other hand, the remaining three libraries do not offer optimised implementations using ARM assembly code or unrolled loops. Nettle and LibtomCrypt use pre-computed AES look-up tables, while Botan builds the look-up tables dynamically (on the fly calculation), increasing the CPU workload. The poor optimisation from these three libraries led to increased execution times and energy consumption. The slowest common ciphers are Camellia-192 and Camellia-256 from Botan and Nettle. As a result, their implementations are the most costly among the common block ciphers. Figure 5.12 also shows the effect of the key size on the energy consumption of the cipher. When the size of the key dictates the number of rounds that the cipher must perform, as in AES and Camellia-128, then the key affects the cipher's energy consumption.

Figures 5.13 to 5.18 show the energy consumption of the remaining block ciphers. The figures show that Triple-DES and its variants are the most costly ciphers in all libraries except the LibTomCrypt, where XTEA is the least energy-efficient cipher. The figures also show that Botan and LibTomCrypt are the least efficient libraries in energy consumption, followed by LibgCrypt. OpenSSL and Crypto++ offered the most energy-efficient implementations.

5.4.2 Stream Ciphers

The number of stream ciphers implemented in the six libraries is limited compared to the number of block ciphers. The libraries only implemented the most frequently used stream ciphers, including ChaCha, Salsa, RC4, Panama, Sober-128 and Sosemanuk, in addition to their variations.

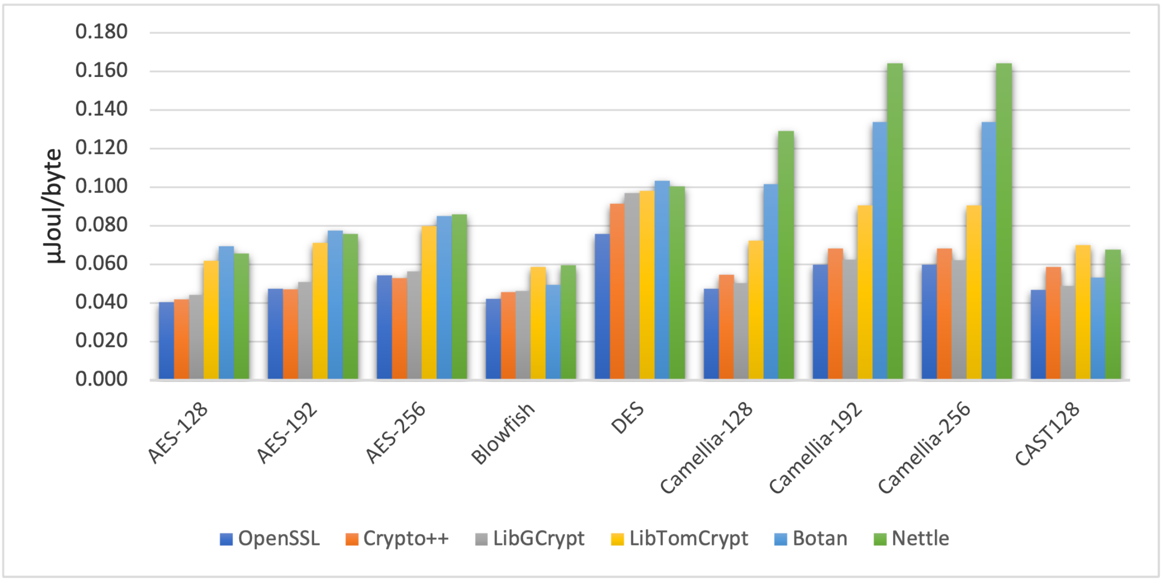


Fig. 5.12 Energy Consumption of the common block ciphers.

Primitives' Throughput

The stream ciphers' implementation performance can also be improved using the hardware acceleration techniques such as AVX2, SSSE3, ARMv8, and others. Surprisingly, all the ciphers that used the acceleration instruction sets in the previous experiment (ChaCha, Panama, Sosemanuk, and Sober-128) are still the fastest even with the absence of the acceleration features. The throughput results of the stream ciphers are shown in Figures 5.19 and 5.20.

Figure 5.19 shows the performance of the common stream ciphers from the six libraries. ChaCha20 is the fastest stream cipher in this experiment. The most optimised implementation is from OpenSSL, followed by LibgCrypt. Both libraries use ARM assembly code to improve the performance of the cipher's implementation. Crypto++'s implementation comes third at about half the speed of OpenSSL's. Crypto++ provides optimisation using AVX2, SSE2 and SIMD; however, none of these features is available for our processor. Nettle's and LibTomCrypt's implementations are the least efficient at speed 3.6 times slower than OpenSSL. ChaCha20 has a simple design based on ARX (Adition, Rotation, Xor) paradigm. It is composed of 20 rounds and supports keys of size 128 or 256 bits.

Figure 5.20 shows the performance of the stream ciphers that are not common across the six libraries. Salsa20r12 from LibgCrypt is the second-fastest stream cipher in our test. The cipher is a reduced round variant of Salsa20 that uses only 12 rounds instead of 20 and hence the difference in the performance between the two. The implementation from LibgCrypt is optimised using ARM assembly code.

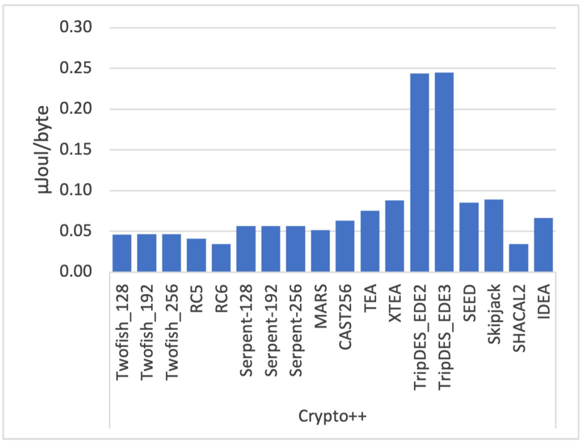


Fig. 5.13 Energy Consumption of Crypto++ block ciphers.

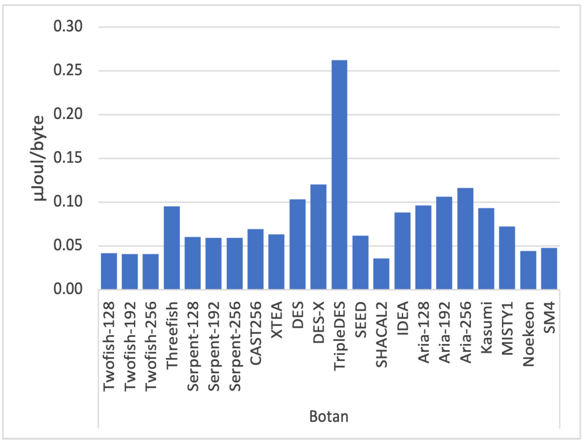


Fig. 5.14 Energy Consumption of Botan block ciphers.

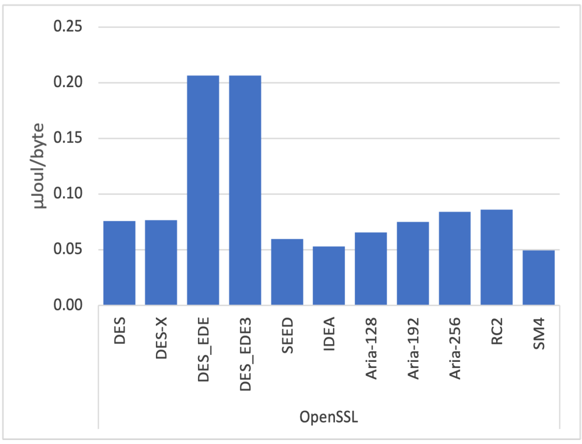


Fig. 5.15 Energy Consumption of OpenSSL block ciphers.

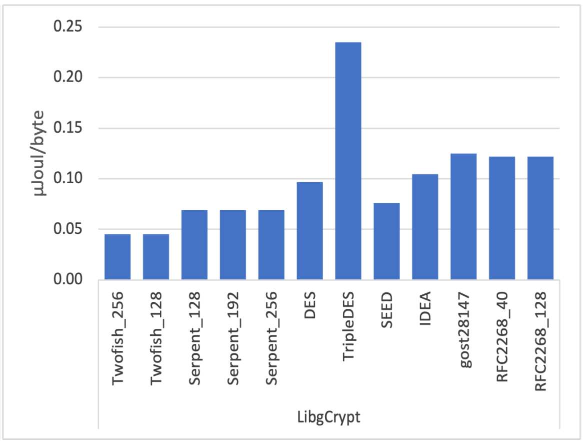


Fig. 5.16 Energy Consumption of LibgCrypt block ciphers.

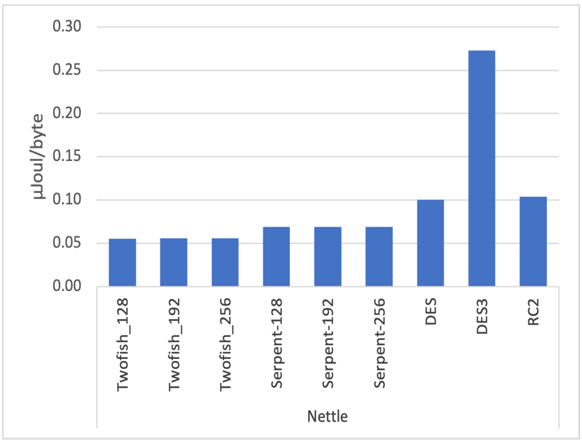


Fig. 5.17 Energy Consumption of Nettle block ciphers.

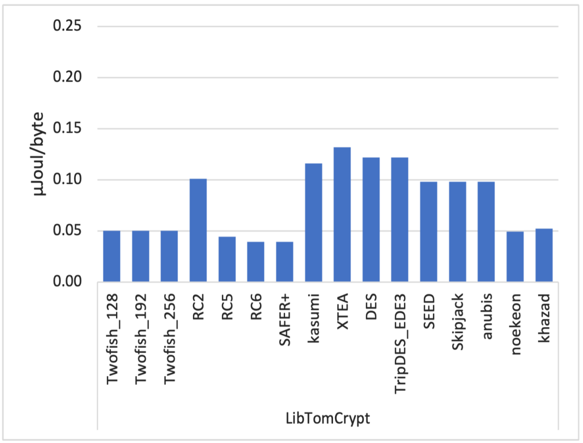


Fig. 5.18 Energy Consumption of LibTomCrypt block ciphers.

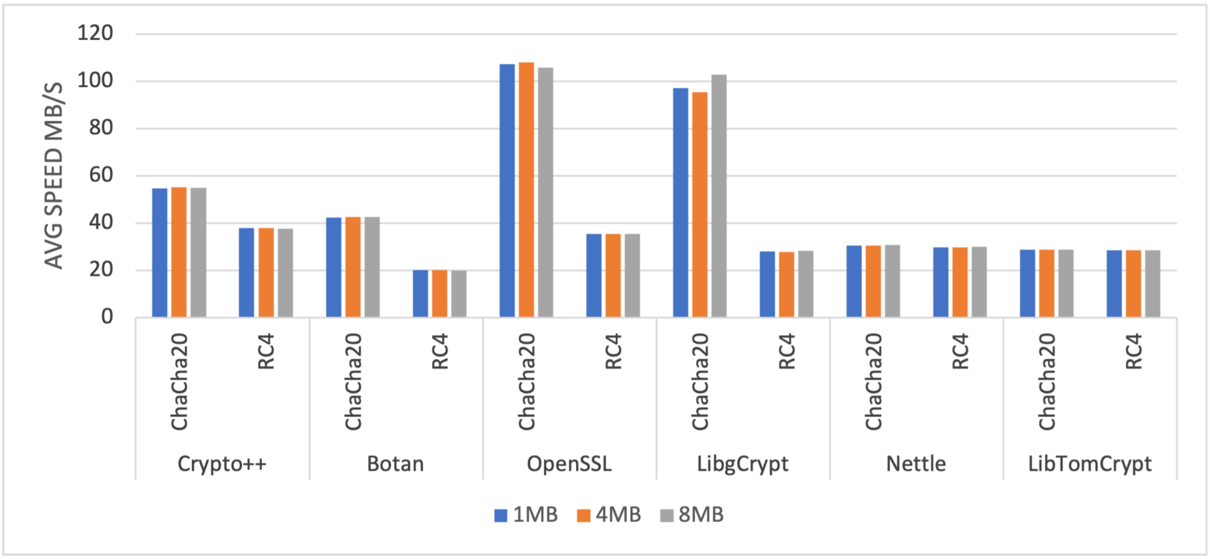


Fig. 5.19 Average speeds of the common stream ciphers.

Sosemanuk is the third-fastest stream cipher. This cipher is only available in Crypto++, which offers an optimised implementation using Intel's acceleration instructions. The cipher has a fast initialisation phase, a small internal state size and a small amount of static data [23]. These features help in improving the cipher performance.

Sober-128, which is available in LibTomCrypt, is the fourth-fastest stream cipher. It is a software-oriented cipher that supports a key up to 128 bits in length [74].

The fifth-fastest cipher is Panama which is only available in Crypto++. The library provides an optimised implementation for the cipher using Intel's acceleration instructions. Panama uses a key of size 256 bits and can be used as a stream cipher or a hash function. However, it is no longer suitable for cryptographic hash use after it has been broken [45].

Botan's implementation of RC4, Figure 5.19, is the slowest stream cipher in our test. RC4 uses key of sizes up to 2048 bits and has an internal state of size 2064 bits. The cipher is no longer considered secure and only used for backwards compatibility with legacy systems [137].

Energy Consumption

Stream ciphers encrypt a single bit (or byte) of data at a time. They use a stream of pseudo-random bits that are XOR'ed with the input data to generate the ciphertext. The pseudo-random bits are generated using digital shift registers with linear and non-linear feedback functions [58]. Therefore the power consumption of stream ciphers is strongly determined by the size of the shift registers and the complexity of the feedback functions [1].

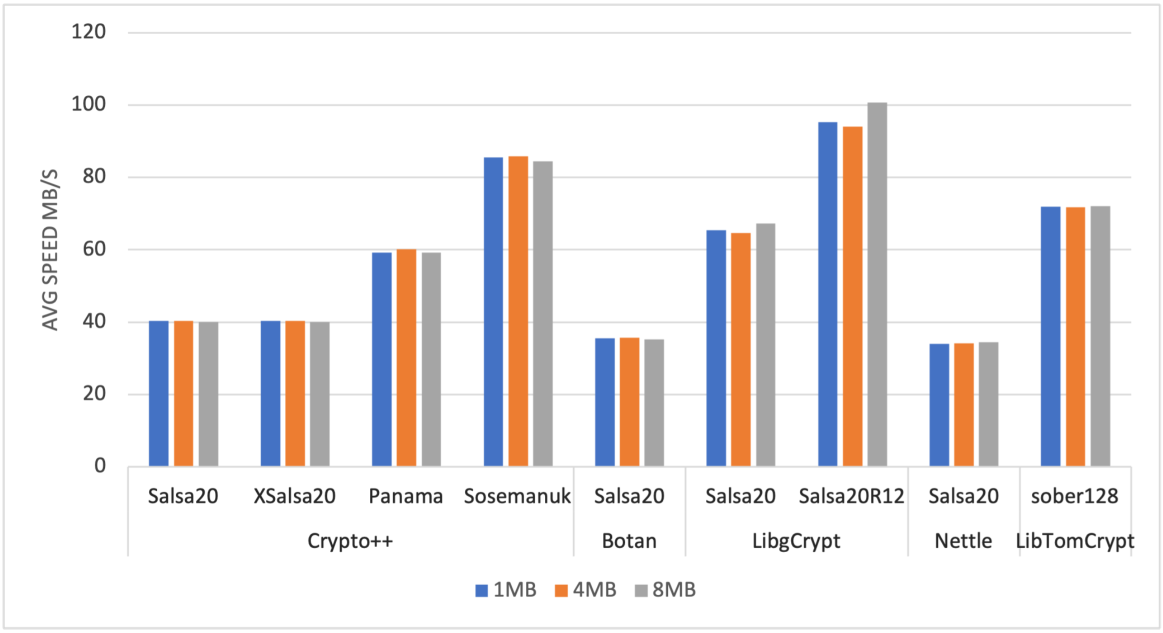


Fig. 5.20 Average speeds of stream ciphers that are not common across the libraries.

In our test, all the implementations of stream ciphers outperform the block ciphers implementations. The least energy-efficient stream cipher, which is Botan's RC4, consumes the same amount of energy, 0.034 microjoules, as the most efficient block cipher, Crypto++'s RC6.

Figure 5.21 shows the energy consumption of the common stream ciphers. Although RC4 has a larger state size than ChaCha20, it consumes less power ($\approx$ 0.75 watts) than that of ChaCha20 ($\approx$ 1.14 watts). This is due to the simplicity of the RC4 design. On the other hand, the ChaCha20 is more efficient in energy consumption due to its fast execution time.

Figure 5.22 shows the energy consumption of the remaining stream ciphers, which have to some degree a comparable power consumption. As a result, The energy consumption of the primitives relies strongly on their execution time. Salsa20 implementations from Botan and Nettle are the least energy-efficient ciphers after Botan's RC4. Sober-128 from LibTomCrypt and Sosemanuk from Crypto++ are the most energy-efficient stream ciphers.

5.5 Conclusion

In this chapter, we discussed three methods of measuring the energy consumption of a program code. We also conducted an experiment to study how cryptographic primitives' design and implementation techniques affect their throughput and energy consumption when run on a device with limited computing capabilities. The study also examined the

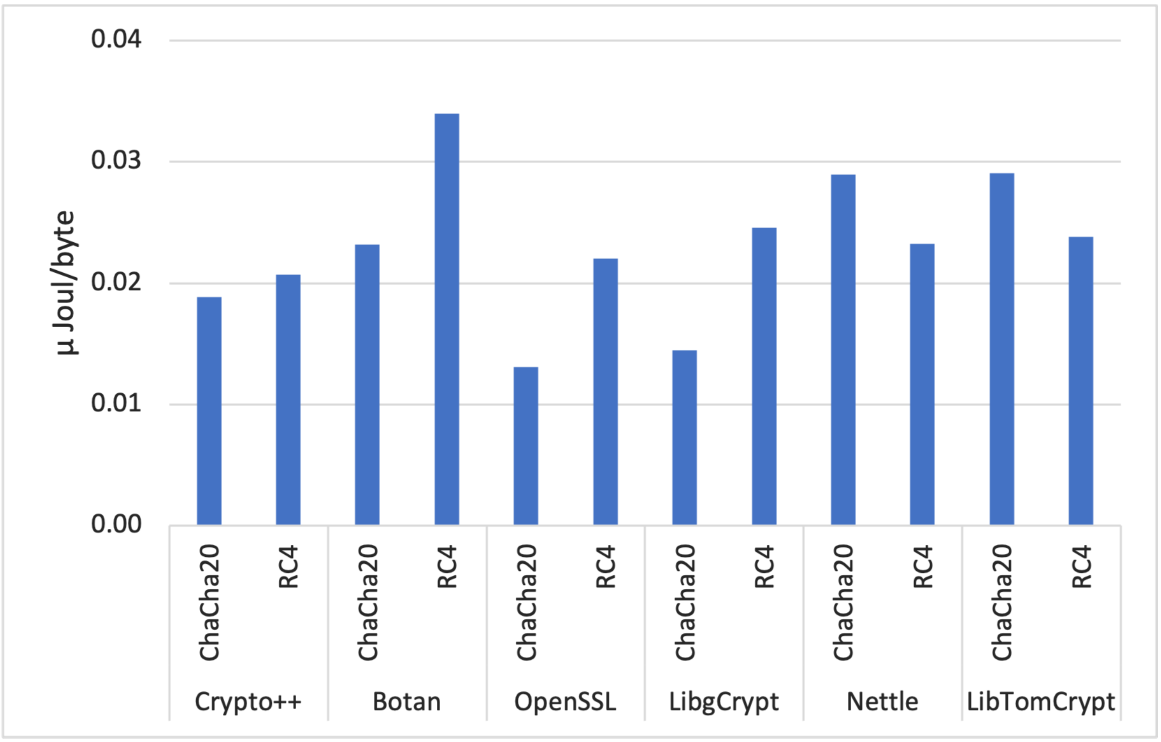


Fig. 5.21 Energy Consumption of the common stream ciphers.

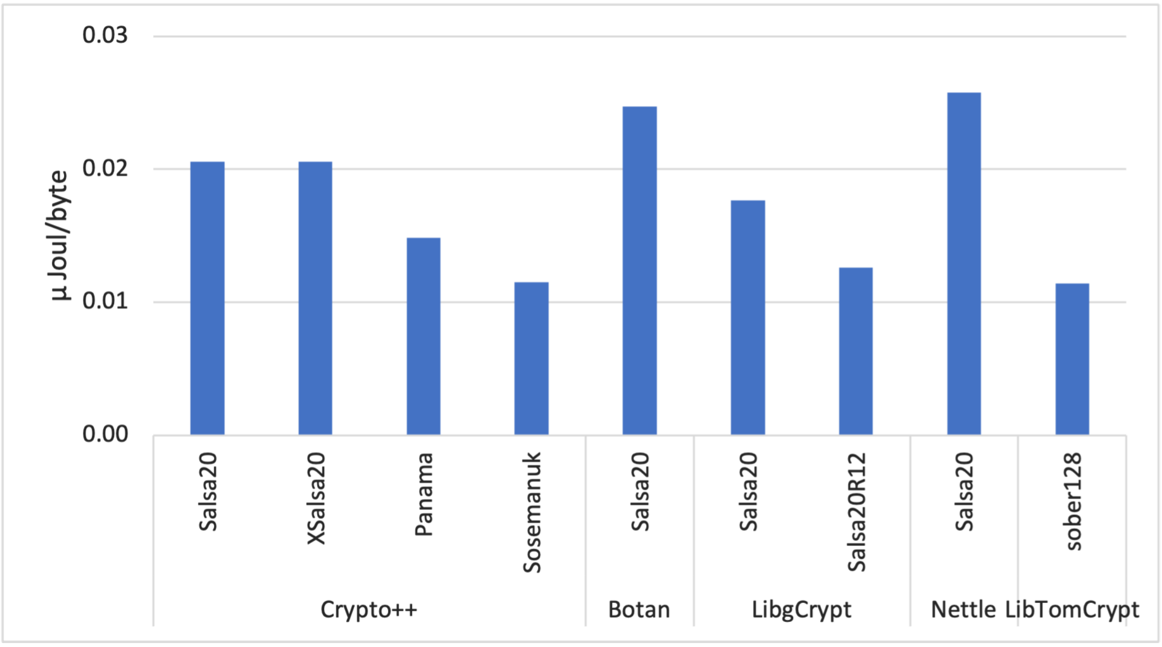


Fig. 5.22 Energy Consumption of the remaining stream ciphers.

performance of the six cryptographic libraries from Chapter 4 in the absence of the hardware acceleration instruction sets.

We summarise our findings from the experiment we conducted in this chapter as follows:

- Calculating the actual energy consumption of a primitive is difficult and requires using a specialised power analyser or instruction-level power model. On the other hand, estimating the energy consumption of a primitive using the command-line tools or cheap multimeters is not complicated.
- Measuring the program's energy usage at the machine instruction level, although accurate but requires knowing the number of clock cycles each instruction takes to execute and the currents it draws. Both quantities are not always published by the CPU manufacturers.
- We think energy consumption is a more relevant metric for evaluating the primitive performance since it considers the primitive's execution time and measures the total power consumed during that time
- Our experiment found that most of the tested primitives have to a certain degree, a comparable power consumption. This makes the primitive's execution time the most critical factor in determining the energy consumption of the primitives.
- The power consumption of the primitive depends on its design architecture and implementation. The complexity and size of the substitution box substantially impact the power consumption of block ciphers. The internal state size and the complexity of the feedback functions determine the stream ciphers' power consumption.
- The stream ciphers are more energy-efficient than block ciphers due to their simple design and fast execution times.
- The key length and the block size of the primitive have no significance in its power consumption. However, both properties have an impact on the speed of the execution and thus the energy consumption.
- All the stream ciphers optimised using the acceleration instruction sets in the previous experiment are still the fastest, even with the absence of these acceleration features.
- OpenSSL and LibgCrypt libraries offered the most optimised implementations for the block and stream ciphers.

Chapter 6

Investigating the Performance of Lightweight Cryptography under Constrained Devices

Summary

This chapter investigates the performance of the 32 round-2 candidates from the NIST lightweight cryptography competition. We conduct an experiment to test the implementation of the candidates and study their throughput and their RAM and ROM footprint. The experiment aims to investigate and understand the performance of the lightweight cryptographic primitives in a constrained environment and assess the amount of resources needed to run them. Additionally, the experiment tries to understand the limitations imposed by the embedded system's constraints on the performance of the cryptographic primitives. Furthermore, the chapter describes the building process of embedded software and explains how to develop a program on a computer and upload the compiled binaries into constrained devices. It also introduces a method to calculate the code size of a program by disassembling its binary files and examining the contents of their different sections. Finally, the chapter describes the different regions of constrained device's memory and discusses static and dynamic analysis methods for investigating the memory usage of embedded software.

6.1 Introduction

Over the last decade, we have seen wide adoption for IoT devices in several application domains ranging from industrial automation, smart home, traffic control, and healthcare. The widespread usage of these constrained devices and the security challenges they brought has resulted in a notable increase in research activities in IoT security. One of the active research areas is lightweight cryptography, due to the need to protect data being transferred between IoT devices. Biryukov and Perri [29], surveyed 117 lightweight cryptographic primitives. NIST (The U.S. National Institute of Standards and Technology) received 57 submissions in 2019 for their lightweight cryptography competition. Chapter 2, Section 2.2 discusses lightweight cryptography, its motivation, and the different design choices and implementation approaches. In this chapter, we evaluate the 32 round-2 candidates of the NIST competition in addition to their variations. We measure the throughput of 87 primitives and calculate their binary code size and RAM footprint.

This chapter is structured as follows: Section 6.2 presents NIST lightweight cryptography competition. Section 6.3 gives a brief introduction on working with constrained devices and describes the building process of a program on these devices. Section 6.3.1 presents a method for disassembling a program and measuring the code size of the program in isolation from the system libraries. Section 6.3.2 introduces static and dynamic analysis methods for measuring a program's memory consumption on embedded systems. The experiment setup is presented in Section 6.4. Section 6.5 discusses the results of evaluating the lightweight cryptographic primitives. Finally, we concluded the chapter in Section 6.6.

6.2 NIST Lightweight Cryptography Standardisation

NIST has started a project to solicit, evaluate, and standardise lightweight cryptographic primitives to be used on constrained devices where the standard cryptographic primitives can not run. In 2015, they held their first lightweight cryptography workshop to get public feedback on the requirements of lightweight cryptography applications and the constraints and limitations imposed by the constrained devices. The second workshop was held in 2016 for the same reason, in addition, to discuss recent works on designing lightweight cryptographic primitives and reviewing issues on the proposed standardisation process. In 2017, NIST published two reports; *Profiles for the Lightweight Cryptography Standardisation Process* [18], and *Report on Lightweight Cryptography* [118]. A call for submission of lightweight primitives was published by NIST in August 2018 (see [131]) in which NIST explained the submission requirements and the evaluation process. NIST requests that all the submitted primitives should implement the AEAD (Authentication Encryption with Associated Data) functionality and optionally implement hashing functionality and that the minimum key length should not be smaller than 128 bits. In February 2019, NIST received 57 candidate primitives for review. Later in April 2019, 56 of them were accepted as first-round candidates. The selection of the first-round candidate was based on the completeness and properness of the submitted primitives according to the requirements published in [131, 156]. In August 2019, NIST announced the second-round candidates. Thirty-two primitives were selected for the second round of the standardisation process (see Table 6.1). The selection was based on the primitive's maturity of design, security, performance and implementation cost [156]. NIST held two workshops in November 2019 and October 2020 to discuss the candidate primitives' design, security, implementation and performance. Recently, on March 29, 2021,

Table 6.1 NIST round-2 candidates and finalists (in bold)

ACE	**Elephant**	Gimli	KNOT	**Romulus**
ASCON	ESTATE	**Grain**	ORANGE	SAEAES
COMET	**GIFT-COFB**	HyENA	Oribatida	Saturnin
DryGASCON	ForkAE	**ISAP**	mixFeed	SKINNY
PHOTON-Beetle	**SPARKLE**	**Xoodyak**	Pyjamask	SPIX
SUNDAE-GIFT	**TinyJambu**	WAGE	Spook	SpoC
Subterranean 2.0		LOTUS and LOCUS		

NIST announced the ten finalists, shown in bold in Table 6.1. NIST has not yet published their report on the evaluation criteria and selection process for the finalists.

6.3 Working with Constrained Devices

Due to the limited resources available, constrained devices can not run a full-function operating system. They either run a lightweight operating system such as FreeRTOS, Mbed, and TinyOS or embedded Linux such as Yocto, Buildroot, Maemo, and Mobilinux (see [146] for a comparison of IoT operating systems). Extremely constrained devices such as the ones we used in our experiment do not run any operating system.

When developing an embedded software, developers usually write their code on a computer then use cross-platform toolchains such as AVR and ARM GNU toolchains to compile and deploy it to the constrained devices. Communicating with constrained devices for transferring data, commands, and debugging purposes are done using UART (Universal Asynchronous Receiver /Transmitter) interface, SPI (Single Peripheral Interface), USB or WiFi.

Building an executable embedded program involves several steps and using a suitable toolchain package (see Figure 6.1). The compilation is the first step in the building process. It involves generating the machine language instructions, the *opcode*, for the target platform and allocating memory for the program variables and the code. The memory allocation information is stored in ***Sections*** in the object files. Knowing these sections is essential when studying the memory and flash drive utilisation by the program code. The main sections are depicted in Figure 6.2 [103] . The ***text*** section (also called *code* section) stores the opcodes generated by the compiler, and its content ends up in the flash disk. The global and static variables that are uninitialised such as:

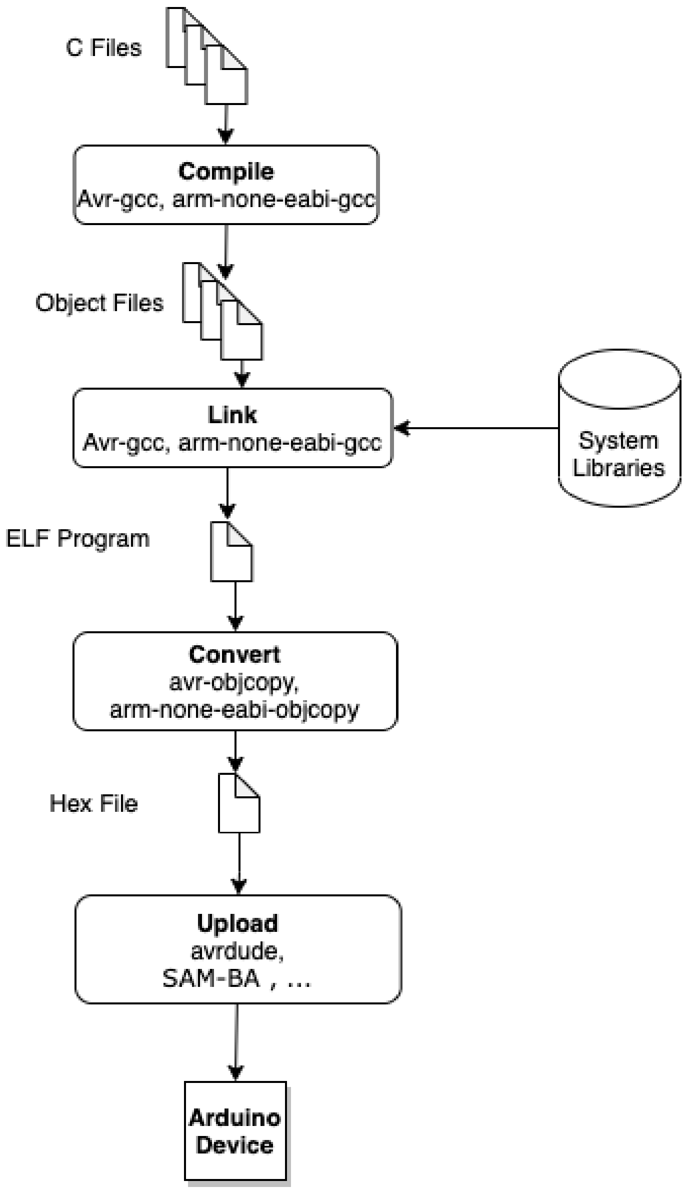


Fig. 6.1 Building an embedded C program

```
int myVar;
static int myVar1;
```

are stored in the ***bss*** section. This section is initialised with zero in the startup code. The global and static variables that are initialised by the programmers with predefined values are stored in the ***data*** section. Their constant values are initially stored in the flash drive in the ***rodata*** (read-only data) section and later copied to the ***data*** section during the program startup. In the following statement:

```
 uint32_t x = 40;
```

The variable *x* is not a constant, so it will live in the ***data*** section and occupy four bytes. The constant value "40" will occupy eight bytes; four bytes in the ***rodata*** section and four bytes in the ***data*** section. Some toolchains do not support a separate ***rodata*** section. In this case, the constant values are stored initially in the ***text*** section. Dynamic variables created by *malloc*, *calloc*, *realloc* calls, or the *new* keyword are stored in the ***heap*** region in memory. The local variables that are defined inside functions are stored in the ***stack*** region. The allocation of the stack and heap spaces are made by the linker and not the compilers.

Combining the different object files into a single executable program is done by a linker such as `avr-ld` or `arm-none-eabi-ld` during the linking process. The linker also concatenates similar sections (text, data, bss,...) from the object files. The output from the linking process is a platform-independent file, commonly **ELF** or **DWARF** files [38]. **ELF** and **DWARF** must be translated to a target-dependent format, typically *.hex* or *.bin*, before being uploaded onto the constrained device. Converting the executable program to target-dependent format is done using tools such as `avr-objcopy` and `arm-none-eabi-objcopy`. After generating the .hex or .bin files, we use an uploading tool specific to the microcontroller we have, such as `avrdude` for Atmel AVR microcontrollers and `SAM-BA` for Atmel SAM ARM core-based microcontrollers.

Developers usually use Integrated Development Environments IDEs applications when writing code for constrained devices. The IDEs help them to consolidate the different processes of developing a program. In addition, they increase the developers' productivity by providing automated build processes, finding the proper libraries and selecting the required toolchains. IDEs also help in using the correct build configuration parameters, selecting the appropriate communication protocol, setting its parameters and uploading the program onto the constrained device.

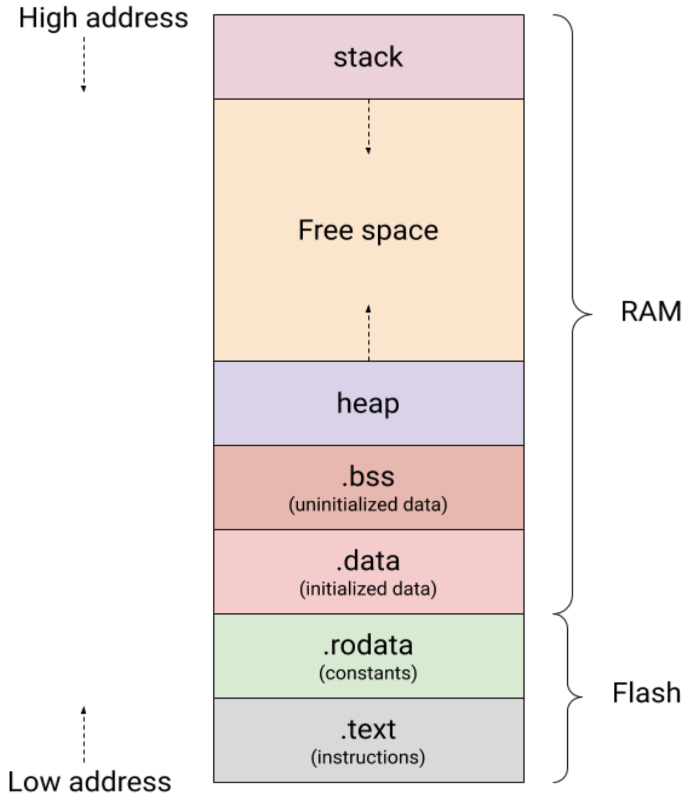
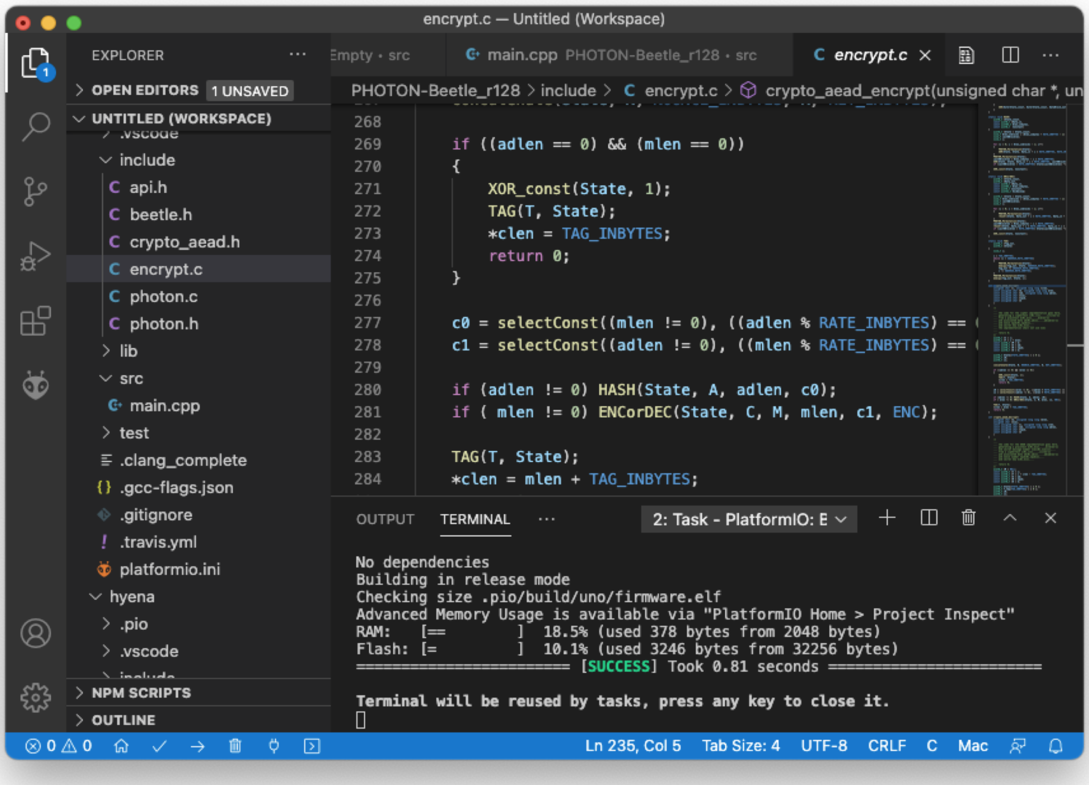


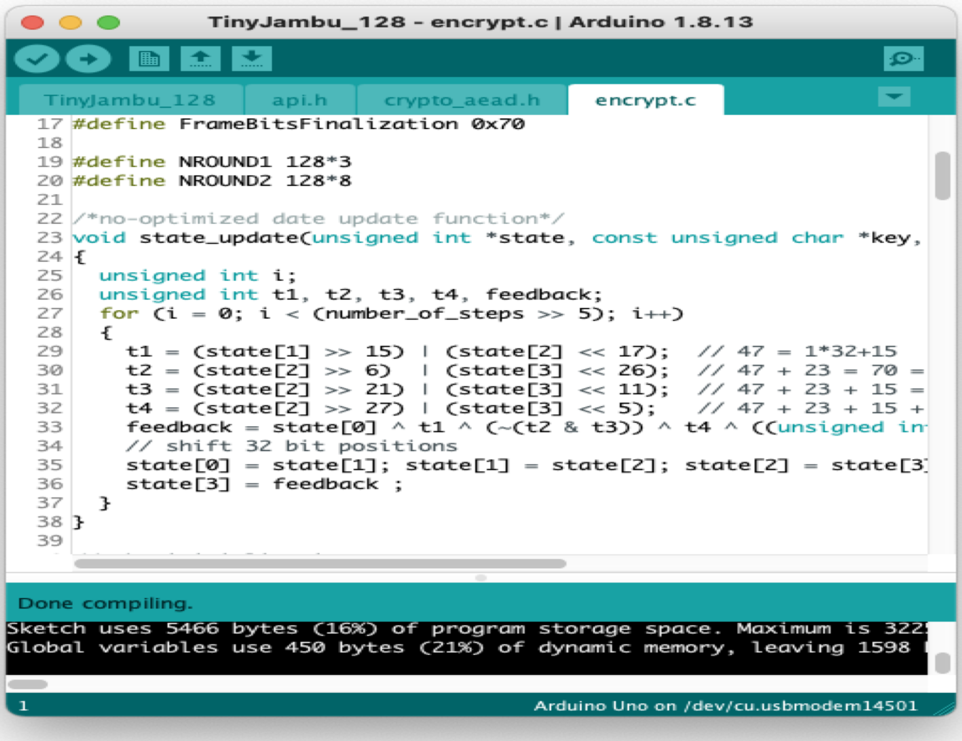
Fig. 6.2 Memory layout [103]



```c
    if ((adlen == 0) && (mlen == 0))
    {
        XOR_const(State, 1);
        TAG(T, State);
        *clen = TAG_INBYTES;
        return 0;
    }

    c0 = selectConst((mlen != 0), ((adlen % RATE_INBYTES) == (
    c1 = selectConst((adlen != 0), ((mlen % RATE_INBYTES) == (

    if (adlen != 0) HASH(State, A, adlen, c0);
    if ( mlen != 0) ENCorDEC(State, C, M, mlen, c1, ENC);

    TAG(T, State);
    *clen = mlen + TAG_INBYTES;
```

```
No dependencies
Building in release mode
Checking size .pio/build/uno/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:   [==        ]  18.5% (used 378 bytes from 2048 bytes)
Flash: [=         ]  10.1% (used 3246 bytes from 32256 bytes)
======================== [SUCCESS] Took 0.81 seconds ========================

Terminal will be reused by tasks, press any key to close it.
```

Fig. 6.3 Screenshot of PlatformIO IDE



```
#define FrameBitsFinalization 0x70

#define NROUND1 128*3
#define NROUND2 128*8

/*no-optimized date update function*/
void state_update(unsigned int *state, const unsigned char *key,
{
  unsigned int i;
  unsigned int t1, t2, t3, t4, feedback;
  for (i = 0; i < (number_of_steps >> 5); i++)
  {
    t1 = (state[1] >> 15) | (state[2] << 17);  // 47 = 1*32+15
    t2 = (state[2] >> 6)  | (state[3] << 26);  // 47 + 23 = 70 =
    t3 = (state[2] >> 21) | (state[3] << 11);  // 47 + 23 + 15 =
    t4 = (state[2] >> 27) | (state[3] << 5);   // 47 + 23 + 15 +
    feedback = state[0] ^ t1 ^ (~(t2 & t3)) ^ t4 ^ ((unsigned in
    // shift 32 bit positions
    state[0] = state[1]; state[1] = state[2]; state[2] = state[3]
    state[3] = feedback ;
  }
}

```

```
Done compiling.
Sketch uses 5466 bytes (16%) of program storage space. Maximum is 322
Global variables use 450 bytes (21%) of dynamic memory, leaving 1598
```

Fig. 6.4 Screenshot of Arduino IDE

There are a number of cross-platform IDEs. PlatformIO [136] (see Figure 6.3) and Arduino IDE [8] (see Figure 6.4) are among the most popular IDEs. Both of them are open-source and free to use. We used both of them in this experiment, and found PlatformIO to offer a wider range of features and capabilities, such as code debugging, static code analysing, unit testing and remote development. In addition, PlatformIO supports over 1000 boards from different manufacturers [136].

6.3.1 Measuring the Code Size

We saw in the previous section that the linker combines the different object files into an ELF executable program that contains machine code and the program sections (text, data, etc). ELF (Executable and Linking Format) [95] is widely used for executable files, relocatable object files, shared libraries, and core dumps. The design of ELF is a standard and is not limited to a specific processor, instruction set, or hardware architecture [95]. The file structure is shown in Figure 6.5 [78] ; The *section header table* lists all the sections in the binary. The *program header table* provides a segment view of the binary (as opposed to section view).

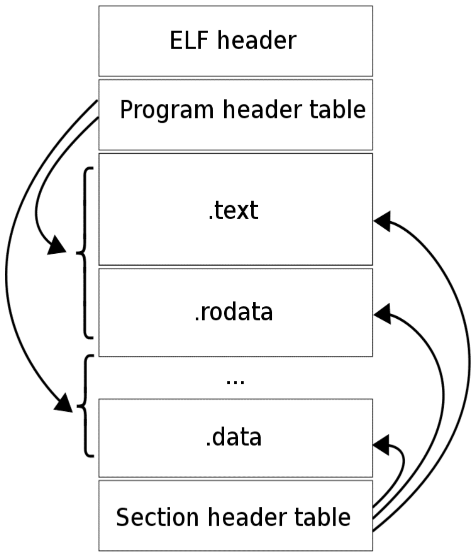


Fig. 6.5 ELF structure [78]

When loading ELF into a process for execution, the operating system and the dynamic linker uses the segment view to find the relevant code and data and select what should be loaded into virtual memory [5]. For a detailed description of ELF see [95] and Chapter 2 of [5].

Inspecting the text section of the ELF file shows all the elements (functions, objects, constants, ...) that will be copied into the flash drive and their sizes. `objectdump` [158] and `nm` [157] are the two command-line tools that can be used to show the content of the sections as in Listing 6.1 and 6.2.

```
$  ./arm-none-eabi-objdump firmware.elf | grep text

00002000 l    d  .text 00000000 .text
000064a0 l     O .text 00000000 __EH_FRAME_BEGIN__
000020b4 l     F .text 00000000 __do_global_dtors_aux
000020dc l     F .text 00000000 frame_dummy
00002380 l     F .text 0000008c _ZL4HASHPhPKhyh
0000240c l     F .text 000000a4 _ZL8ENCorDECPhS_PKhyhm
000024b0 l     F .text 00000018 _ZL3TAGPhS_
00005f1f l     O .text 00000040 _ZL12MixColMatrix
00002780 l     F .text 00000024 _ZL5utox8mPc
00002c68 l     F .text 000000c4 _ZN14USBDeviceClass6initEPEmm.part.7
00006062 l     O .text 00000012 _ZL21USB_DeviceDescriptorB
```

Listing 6.1 The output of the *objdump* command

The first column in the previous listing is the address of the element in the flash drive in hexadecimal value. The next is the flag; *l* is for the local elements. The type of the element, functions (*F*) or objects (*O*), is displayed in the third column. The fourth column is the name of the section in which the elements live followed by the size of the elements in hexadecimal. The names of the elements are displayed in the last column.

```
$  ./avr-nm --print-size --size-sort --radix=d firmware.elf

08389460 00000001 b timer0_fract
00008676 00000001 B USBDevice
00000348 00000002 t _ZN5Print5flushEv
00008272 00000004 T __ashrdi3
00003518 00000004 t __cxa_pure_virtual
08389461 00000004 b timer0_millis
08389465 00000004 b timer0_overflow_count
00000350 00000006 t _ZN5Print17availableForWriteEv
00009214 00000006 T __fp_nan
00008686 00000008 T __divsf3
00009374 00000008 T __mulsf3
08389316 00000008 b clen
08389180 00000008 b mcpylen
```

Listing 6.2 The output of the *nm* command

The `nm` command lists all the elements (also called symbols) from the different sections in ELF and other object files. The elements in Listing 6.2 are displayed in ascending order according to their size (`-size-sort`). We can use the option (`-size-sort -reverse-sort`) to display the largest elements first. `-radix=d` option displays the values of the elements in decimal. The first column in the previous listing shows the address of the element. The size of the elements is shown next in bytes, followed by the section (T and t for *text*, B and b for *bss*, D and d for *data*, and so on). The last column shows the names of the elements. To display the element's location (the file name and the line number), use the option `-l`.

We compute the code size of a program by adding the sizes of all elements in the *text* and *data* sections. The `size` [94] command-line tool is often used to calculate and report the section sizes in bytes as in Listing 6.3.

```
$   ./avr-size firmware.elf
```

```
  text       data       bss       dec       hex    filename
  9634        316       446     10396      289c    firmware.elf
```

Listing 6.3 The output of the *size* command

The `size` tool calculates the sizes of all elements in the *text*, *data*, and *bss* sections and displays the total size of each section. The sum of the three sizes is displayed in the **dec**imal and in **hex**adecimal format in fourth and fifth columns.

The code size of a program is the summation of *text* size + *data* size. The result is what the IDEs report as the code size (see Figure 6.3). However, this result is not the size of our code alone. It is the total size of our program's code and the system libraries of the constrained device. The system libraries are linked to our code by the IDEs automatically during the linking process. So we have to find a way to exclude the system libraries from the calculation and measure only the code size of our implementation.

One approach is to inspect the *text* and *data* sections of the ELF file and discard from the size calculation any element that does not belong to our implementation. This approach is what we follow in our experiment, and here how we implement it:

1. After compiling our code, we run the `nm` command as in Listing 6.2 on the resulting ELF file.

2. We copy all the elements that belong to Arduino's library from the *text*, *data* and *bss* sections to a text file (we name it `arduino_elements`).

3. For every primitive implementation, we use the `nm` command as in Step 1 and direct the output to a text file (we name it `imp_code_size`).

4. We run the python code in Listing 6.4 which we wrote to calculate the size of all elements that are in `imp_code_size` file but not in the `arduino_elements` file.

5. The result as shown in Listing 6.5, is the code size of the primitive implementation only.

```

arduino_secs = set()
with open('arduino_elements') as reader:
    for line in reader:
        contents = line.split()
        arduino_secs.add(contents[3])
```

```python

code_size = 0
text = 0
data = 0
bss  = 0
with open('imp_code_size') as reader1:
    for line in reader1:
        columns = line.split()
        if len(columns) == 4 and columns[3] not in arduino_secs:
            if columns[2].lower() == 't':
                text += int(columns[1])
            elif columns[2].lower() == 'd':
                data += int(columns[1])
            elif columns[2].lower() == 'b':
                bss += int(columns[1])
code_size = text + data
ram = data + bss
print('text = ', text, 'bytes')
print('data = ', data, 'bytes')
print(' bss = ', bss, 'bytes')
print('Code Size (text + data) = ', code_size, 'bytes')
print('RAM (bss + data) = ', ram, 'bytes')
```

Listing 6.4 Calculating the Code Size and RAM usage of a primitive implementation

```
% python calc_code_size.py
 text =  5196 bytes
 data =  262 bytes
  bss =  292 bytes
Code Size (text + data) =  5458 bytes
RAM (bss + data) =  554 bytes
```

Listing 6.5 The results from executing the python code in Listing 6.4

This approach of calculating only the code size of the primitives' implementation, although simple but not practical if we have a large implementation with hundreds of elements in the different sections of the ELF file. Another approach is to build the primitives' implementation outside the IDEs and to exclude the system libraries during the linking process. We have not tried this approach, and we do not know if the code will compile without any difficulties.

6.3.2 Measuring the Memory Consumption

The memory region consists of the *data* and *bss* sections in addition to the heap and the stack (see Figure 6.2). The RAM footprint of a program is determined by the size of the data and bss sections and the maximum stack and heap consumption. Calculating the size of *data* and *bss* sections is not difficult, as we have seen in the Section 6.3.1. Determining the maximum consumption of stack and heap requires extra effort as we will see shortly.

The Heap Memory

Heap is the area in memory where the dynamic memory allocation takes place. The operation is called dynamic allocation because the actual location and the size of the allocated space need not be known at the compile time [102]. A pointer must reference the dynamically allocated memory:

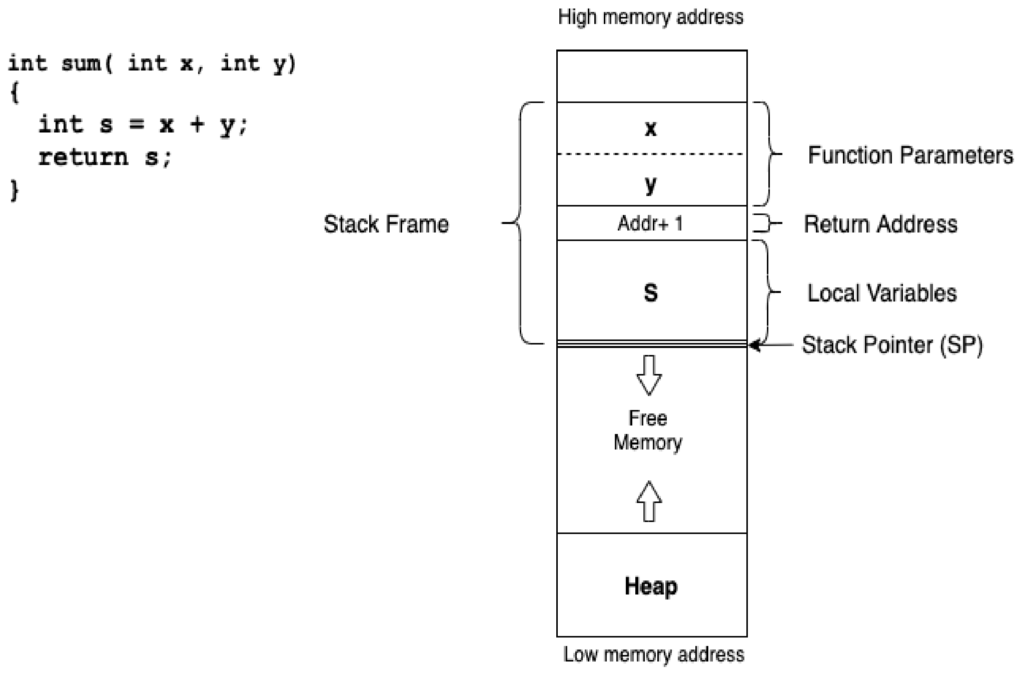
```
int *p1 = malloc(sizeof(int32_t)); // allocate 4 bytes at the heap memory
```

The heap area is managed by `malloc`, `realloc`, `free`, and `delete` operations. It starts at the end of the *bss* section and grows towards larger memory addresses. The memory used by the heap objects has to be explicitly reclaimed using the `free` or `delete` operators or by the run time system using garbage collectors [90]. The deallocation of the heap objects can lead to fragmentation which is a issue in heap memory.

The Stack Memory

The stack is a memory region where the local variables (variables defined inside a function) reside. It is usually located at the higher part of the memory, and it grows downward towards the heap region. The stack's current boundary is stored inside the stack pointer (`SP`). The memory address inside `SP` changes as the stack grows and shrinks. When this pointer meets the heap pointer (`__brkval`), the free memory is exhausted. The memory allocation in the stack takes place on contiguous blocks of the memory, unlike the heap where memory is allocated at random access [30]. Allocation occurs when a function is called, whereas deallocation happens when this function returns. Figure 6.6 depicts the layout of the stack memory, the figure is adopted from [102]. For a comprehensive comparison between the stack memory and the heap memory see [102].

Some compilers provide compilation options to help analyse the stack usage of a program. For example, both AVR and Arm GNU toolchains offer the `-fstack-usage` command-line option to make the compiler outputs per-function stack usage. The output is saved into a file



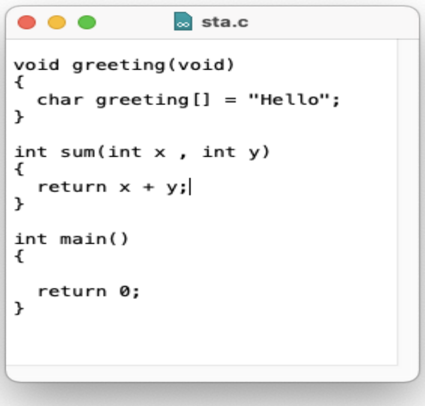
```
int sum( int x, int y)
{
  int s = x + y;
  return s;
}
```

Fig. 6.6 Memory Stack

that has the same name as the executable file appended to it `.su` extension (see Figure 6.7 and 6.8)

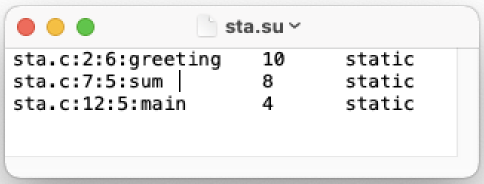
The output, as shown in Figure 6.8, is made up of three fields. The first field is the function name and its location (filename, line and column numbers). The second field is the number of bytes needed to be allocated for this function in the stack. The third field is the qualifiers for allocation pattern. These qualifiers are: *static* for constant allocation, which means a fixed number of bytes (as in Field 2) will be allocated for the function on its entry and deallocated on its exit and no stack adjustment is made during the function execution. The *dynamic* qualifier means, in addition to a constant allocation, a dynamic allocation may take place during the function execution for a sum greater than the number of bytes in Field 2. The *dynamic*, *bounded* qualifier means the allocation is the same as in dynamic allocation; however, the number of bytes allocated will never exceed the number of bytes in Field 2.

This static analysis of the stack has a problem: summing the number of bytes needed for allocating each function in the stack will not give us the real maximum stack usage of a program. This is because of the continuous allocation and deallocation of the stack memory as functions are regularly pushed and popped out of the stack. Thus, we get an overestimation of the maximum stack usage. To solve this, we need to inspect our code for all the possible execution paths and find the most expensive path in stack usage, then only consider the stack usage for the functions in this path.



```
void greeting(void)
{
  char greeting[] = "Hello";
}

int sum(int x , int y)
{
  return x + y;
}

int main()
{

  return 0;
}
```

Fig. 6.7 Stack Usage.



```
sta.c:2:6:greeting    10     static
sta.c:7:5:sum         8      static
sta.c:12:5:main       4      static
```

Fig. 6.8 `-fstack-usage` output

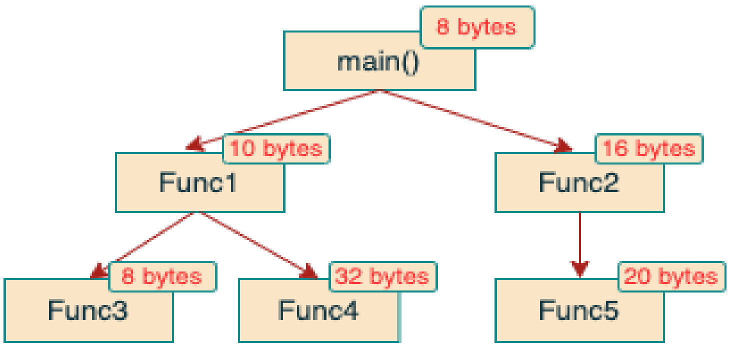


Fig. 6.9 Functions stack usage

Figure 6.9 shows the execution path of a program decorated with the stack usage for each function. To calculate the maximum stack usage, we identify the most expensive execution path which is `main -> func1 -> func4`, then add the stack usage of the functions in this path : 8 + 10 + 32 = 50 bytes.

Trying to find the most expensive execution path in a large program can be a tedious job. For this reason, some compilers offer help by providing the command-line option `-fcallgraph-info` to generate a call tree information for the program. This call graph makes it easy to have a visual look at all the possible execution paths of a program. The call graph output is generated in VCG format and can be viewed using any VCG viewers such as GraphViz. Each node (function) in the graph can be decorated with stack usage information using `-fcallgraph-info=su`, or with information about the dynamically allocated objects using `-fcallgraph-info=du`. Using the generated graph, you can iterate over all nodes in the call tree and calculate the stack usage for each node (function) along the execution path and find the highest stack usage. For more information about `-fstack-usage` and `-fcallgraph-info` see [68] and [34].

The stack usage can also be measured at runtime. The method involves subtracting the memory address at the top of the stack from the memory address at the bottom of the stack as follows:

```
int stack_size = RAMEND - SP;
```

`RAMEND` pointer holds the address of the end of the memory (where the stack begins), whereas the stack pointer (`SP`) holds the address of the top of the stack. The accuracy of the measurements depends on where you place the testing code as the stack grows and shrinks constantly. It is crucial to measure the stack at the deepest point of the call chain as this is where the stack is at its maximum size. Also, you should either measure the stack usage for every execution path to identify the maximum stack usage or find the most expensive execution path and measure its stack usage alone.

Measuring the heap size can be done using the GNU's `mallinfo2()` function [93], which returns a structure that contains memory allocation statistics:

```
struct mallinfo2 m_info;
m_info = mallinfo2();

int total_used_size    = m_info.arena;
int current_used_size  = m_info.uordblks;
int total_free_space   = m_info.fordblks;
```

The arena field stores the total size of the heap memory allocated so far in bytes. Its value increases on each successful allocation but doesn't decrease on deallocation. The uordblks field stores the size of the heap memory currently in use by our program in bytes. The size of the heap that is presently free is stored in the fordblks field. Unlike arena, the values of uordblks and fordblks change with every allocation and deallocation. For more information about the mallinfo2() function, see [93].

Unfortunately, the mallinfo2() is not supported by all systems. In the absence of this support, the easy way to measure the heap size is to use the comand-line option, -fcallgraph-info=du as described earlier. If the compiler used does not support that option, then another way is to inspect your code and count the size of all objects that will be allocated using malloc and related functions.

Suppose we are not interested in knowing the program's stack or heap usage and only interested in knowing the total memory usage of our program. In that case, we can measure the free memory and subtract the obtained value from the device's memory size to get the memory footprint of our program. The free memory is a region bounded by the top of the stack (SP) and heap (__brkval) (see Figure 6.2). The code in Listing 6.6, shows how to measure the free memory. This code is adopted from [66].

```

int free_memory () {
  extern int __heap_start, *__brkval;
  int top;   // This local variable will be placed on top of the stack.
  return (int) &top - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
}
```

Listing 6.6 Calculating the free memory

For devices that use ARM toolchain package, we measure the free memory using the following code [66]:

```

int free_memory () {
  extern "C" char* sbrk(int incr);
  int top;   // This local variable will be placed on top of the stack.
  return &top - reinterpret_cast<char*>(sbrk(0));
}
```

Listing 6.7 Calculating the free memory for ARM microchips

There are two problems with this method; the first is the fragmentation in the heap region. The constant allocation and deallocation leave the heap fragmented, with free spaces behind the top of the heap (`__brkval`). The second problem is that the size of the free memory changes throughout the program's execution. Therefore we must be careful where to evaluate the free memory, so we do not underestimate the program's memory usage.

A frequently used techniques by performance profiler applications to analyse the memory usage of programs is called *stack canary* and *stack painting*. The idea is to initialise (paint) the memory region containing the stack and heap with a known value at the very beginning of the program startup. Then later (after our program has run awhile or after we finish testing all program's functions), we walk the same memory addresses and count how many bytes have still the same initial value (and have not been overwritten by the expansion of the stack and the heap). The result is the free memory size which, when subtracted from the device memory size, gives the program's memory usage. Listing 6.8 shows the stack painting technique. The code is adopted from [153].

```
#define STACK_CANARY_MARKER 0xFC

extern char *__bss_end;
extern uint8_t  __stack;

extern uint8_t *__brkval;
extern uint8_t __heap_start;

void StackPaint(void) __attribute__((constructor));
void StackPaint(void)
{
  uint8_t *p = (uint8_t *)&__bss_end;
  while (p <= &__stack)
  {
    *p = STACK_CANARY_MARKER;
    p++;
  }
}

uint16_t StackCount(void) {
  const uint8_t *p = (reinterpret_cast<int>(__brkval)==0 ?
                     (uint8_t *)&__bss_end
                     :
                     (uint8_t *)__brkval);
  uint16_t count = 0;
  while (*p == STACK_CANARY_MARKER && p <= (uint8_t *)&__stack) {
```

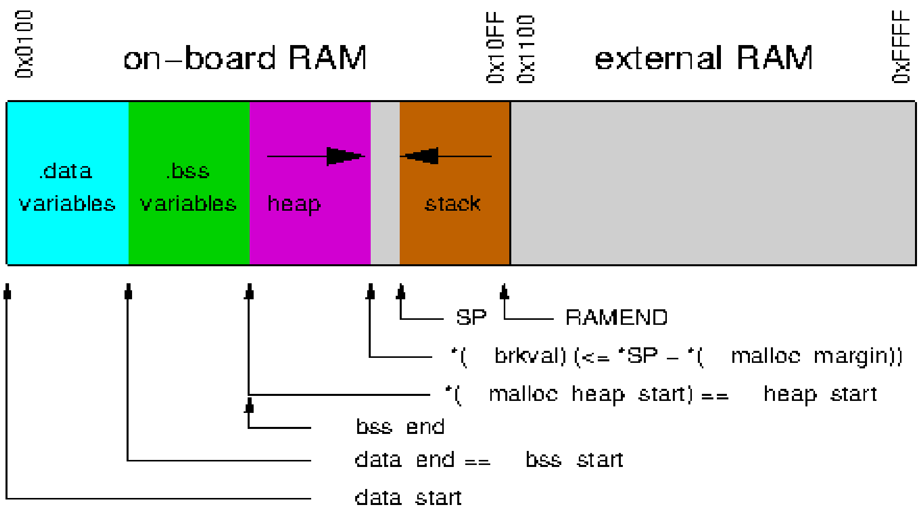


Fig. 6.10 Memory areas and their symbols[152]

```
        p++;
        count++;
      }
      return count;
    }
```

Listing 6.8 Stack Painting

The symbols in lines 3-7 in Listing 6.8 store the addresses of the different memory regions, as shown in Figure 6.10. The linker defines these symbols, and sometimes they differ from one linker to another. The stack starts at `__stack`, which is at the top of the memory, `RAMEND`. Thus, both `__stack` and `RAMEND` symbols hold the same memory address. However, this is not true for all memory layouts. We start filling the memory at `__stack` down until the heap start at `__heap_start`. For the memory we used, both `__heap_start` and `__bss_end` hold the same memory address. The constructor attribute at line 9, ensures that the `stackPaint()` method is executed at the program's startup before the stack is initialised and before the `main()` is called.

A word of caution before using the code in Listing 6.8, as constrained devices have different memory layouts, you should check the device's memory layout and identify where the stack and heap start and end. Also, you should be familiar with the different symbols defined by the linkers for the different memory regions. These symbols can be found in the linker file.

Table 6.2 The specification of the constrained devices used in the test

Board	Frequency	Flash	RAM	Compiler
ESP8266 NodeMCU WiFi	80 MHz	4MB	128 KB	Xtensa gcc toolchain
	32-bit microprocessor			
Arduino MKR WIFI 1010	48 MHz	256 KB	32 KB	Arm Embedded Toolchain
	SAMD21 Cortex®-M0+ 32bit ARM MCU			
Arduino Uno Rev3	16 MHz	32 KB	2 KB	AVR Toolchains
	ATmega328P AVR 8-bit			

Fig. 6.11 Arduino Uno Rev3 [10]

Fig. 6.12 ESP8266 [63]

Fig. 6.13 Arduino MKR 1010 [9]

6.4 Experiment Setup

For this experiment, we used three embedded devices, two from Arduino and one from Tensilica. The specifications of the three boards are shown in Table 6.2. We first started with the Arduino MKR 1010 [9] and the ESP8266 [63] devices to investigate the various methods of analysing the code size and memory usage. Then we moved to Arduino Uno [10], the most constrained device among the three, to study the implementation of NIST's second-round candidates and examine their throughput, code size and memory usage.

All three boards come with USB ports to connect the boards to the computer for uploading the binaries and receiving the program output. The USB ports are also used to power the devices. We used three compilers to compile our codes; for ESP8266, we used the Xtensa GCC Toolchain (`xtensa-lx106-elf-gcc`) version 5.2.0. We also used the GNU Arm Embedded Toolchain (`arm-none-eabi-gcc`) version 10.2.1 on Arduino MKR 1010 and the AVR Toolchain (`avr-gcc`) version 7.3.0 on Arduino Uno. We found that using IDEs to write code for a constrained device is very helpful. IDEs spare us dealing with many technical details, including setting the correct build configuration parameters, selecting the appropriate communication protocol and setting its parameters, and uploading the code onto the device. Because of that, we used the Arduino IDE (see Figure 6.4) and PlatformIO (see Figure 6.3) to write our benchmarking code. We uploaded the code to the boards using avrdude and SAM-BA tools.

We measure the execution times of the primitives using the `micros()` function [11] from the Arduino library. The function returns the number of microseconds since the Arduino board started running the current program. The microseconds are stored in a variable of type `unsigned long`. This data type store 32 bits and overflows (goes back to zero) after $2^{32} - 1$ microseconds, which is approximately 70 minutes [11]. The resolution of `micros()` on the board we used is four microseconds, which is good for the purpose we are using it for. The code in Listing 6.9, shows how we use the function. The library also provides another function which we did not use; `millis()` [12]. This function returns the number of milliseconds since the current program started. It overflows after approximately 50 days [12].

```

 unsigned long start;
 unsigned long elapsed;

 start = micros();

 for (count = 0; count < 50; ++count) {
   crypto_aead_encrypt(ct, &ct_len, message, m_len,
                       ad, ad_len, NULL, n_sec,
                       key);
   }

elapsed = micros() - start;
```

Listing 6.9 Measuring the execution time using `micros()` function

When calculating the execution times, we include the cipher initialisation, encryption and decryption operations. We used a 32-bytes plaintext, a 128-bit key, 128-bit nonce, and 32-byte associated data. The execution time is used to calculate the primitives' throughput according to the following formula:

$$\textbf{Througput} = \frac{\textit{Data Size (Bytes)}}{\textit{Execution Time (seconds)}}$$

We used the technique in Section 6.3.1 to measure the code size of the primitives. The method involves disassembling the ELF program of each primitive, using the `nm` command-line tool and the python code in Listing 6.4, which we wrote to calculate the various sections' sizes of the program.

The memory footprint of each primitive is measured using the stack painting technique discussed in Section 6.3.2 in addition to the `size` command-line tool. The `size` tool is used to calculate the size of the uninitialised global and static variables in section ***bss*** and the initialised variables in section ***data***.

Finally, the size of the plaintext, the ciphertext, the key, the nonce, and the associated data AD are all included in the calculation of the code size and the memory consumption of the primitives.

6.5 Evaluation

We evaluated the performance of all the second-round candidates and their variations with a total of 87 primitives. The variations differ from the primary algorithm in either the external parameters such as key size and nonce length or in internal parameters such as the number of rounds and state size. Table 6.3 shows the classification of the 32 second-round candidates. Some candidates provide hashing functionality (shown in bold in the table) in addition to AEAD. However, we only evaluate the AEAD primitives. Also, some submitters provided optimised implementations for their primitives in addition to the reference implementation required by NIST. However, we only tested the reference implementation of the candidates.

AEAD is a variant of the Authentication Encryption (AE) technique that provides confidentiality for the message and a way to check the message's integrity and authenticity [100]. The Associated Data (AD), sometimes called Authentication Data, in AEAD is a string that binds a ciphertext to a context, so replacing the ciphertext with another valid ciphertext (encrypted with the same key) or using it in a different context will be easily detected and rejected. The AD can be any context-dependent values such as record sequence number, addresses, and header information.

There are different implementations for the AEAD technique. However, typically an encryption function takes in as input a plaintext, a secret key, associated data AD, and nonce (unique value generated using a counter or a random number generator) and output a

Table 6.3 Classification of round-2 candidates. Bold primitives provide Hashing functionality in addition to AEAD)

Permutation Ciphers	**ACE**	**ASCON**	**DryGASCON**	Gimli
	ORANGE	Oribatida	**PHOTON-Beetle**	**SPARKLE**
	Elephant	SPIX	**Subterranean 2.0**	ISAP
	WAGE	**KNOT**	**Xoodyak**	SpoC
Block Ciphers	**Saturnin**	COMET	GIFT-COFB	HyENA
	Pyjamask	SAEAES	SUNDAE-GIFT	TinyJAMBU
	mixFeed			
Tweakable Block Ciphers	**SKINNY**	ForkAE	ESTATE	LOTUS
	Romulus	Spook	LOCUS	
Stream Ciphers	Grain-128AEAD			

ciphertext and an authentication tag. The decryption function takes in as input a ciphertext, a secret key, nonce, AD, and the authentication tag. It then outputs either a decrypted text or an error if the calculated authentication tag does not match the supplied one.

As discussed in Chapter 2, Section 2.2, lightweight cryptographic primitives are either built as a modification of well-investigated standards such as AES, where the complexity of the original primitives is reduced to make them IoT friendly or built using components from existing standards. Sometimes they are built from scratch. For the NIST competition, the primitives that leverage components from existing standards are favoured for selection.

The primitives from the second round can be classified based on their underlying construction into permutation cipher based, block cipher based, tweakable block cipher based, and stream cipher based (see Table 6.3). Candidates that are permutation cipher based build their AEAD primitives using constructions from permutation algorithms. The tweakable block cipher based candidates use components from the existing tweakable block ciphers to design their AEAD primitives. Tweakable block ciphers are the same as block ciphers, with the exception that they use an additional input to the encryption and decryption functions; the *tweak* [111]. In tweakable block ciphers, the plaintext is encrypted under the control of both the secret key and the *tweak*. Because block ciphers are inherently deterministic, changing the encryption key to achieve variability is highly recommended. However, changing the key for every encryption is expensive because of the costly key setup operations. Using a *tweak* (a cheap operation) in conjunction with the key is one solution to this problem. The *tweak* should be unique under the same key. Tweaks are similar to the initialisation vectors and nonce and can be generated using a counter or random number generator.

The results from testing the throughput of the primitives are shown in Figure 6.14 and 6.15. Due to the large number of variants, we only show the fastest variant from each candidate. In the figures, each bar shows the throughput of the encryption and the decryption operations separately. The bars are coloured according to the candidates' underlying cipher. Figure 6.14 shows the throughput of the six fastest primitives in bytes per second. The `sESTATE-Twe-AES-128v1`, a variant of ESTATE [40] candidate, significantly outperforms all other primitives. This AEAD primitive is developed using the ESTATE mode of operation in conjunction with the TweAES-128 block cipher. The designers of the primitive chose a fast underlying cipher, AES, and optimised it further by reducing its rounds to only six rounds. This optimisation increases the cipher's performance but reduces its security (although the designers argue otherwise). The cipher also has good design features that improve its performance even further, including **parallelisable** encryption and decryption, where the message blocks are processed in parallel and independent. The cipher uses a **single pass** scheme where the plaintext and the associated data, AD, are processed only once to generate the ciphertext and the authentication tag. The computation cost of a single pass is about half compared to two-pass schemes [41]. Another feature is **inverse-free**, where only the implementation of the encryption functions is needed, and no decryption call to the underlying cipher is required.

Although `Schwaemm128-128`'s throughput is less than half of that of the `sESTATE-Twe-AES-128v1`'s, it is the second-fastest primitive in our test. The primitive is an AEAD variant of the SPARKLE [21] algorithm. It uses the sponge construction [26] which is a simple and lightweight structure that doesn't require any key scheduling or decryption function implementation [41]. The primitive is based on the SPARX [53] cipher, which is an ARX-based block cipher. As discussed in Chapter 2, Section 2.2, ARX-based primitives are very efficient in constrained devices. `Schwaemm128-128` takes 128-bit nonce, a 128-bit key, outputs a 128-bit authentication tag, and has a small state size of 256 bits only.

`Ascon-128-av12` is the third-fastest primitive in our test. It is one of the three variants of the Ascon [54] candidate. The primitive design shares many features with SPARKLE and hence the comparable performance. Both are permutation-based and use Sponge construction. In addition, they are single pass and inverse free ciphers. Furthermore, the two primitives do not use a key schedule or key expansion functions, so there is no key setup cost when the key is changed [54]. Unlike SPARKE, the Ascon underlying permutation algorithm is ASCON-p. `Ascon-128-av12` has a state size of 320 bits and a block size of 128 bits. It takes a 128-bit nonce, a 128-bit key, and outputs a 128-bit authentication tag.

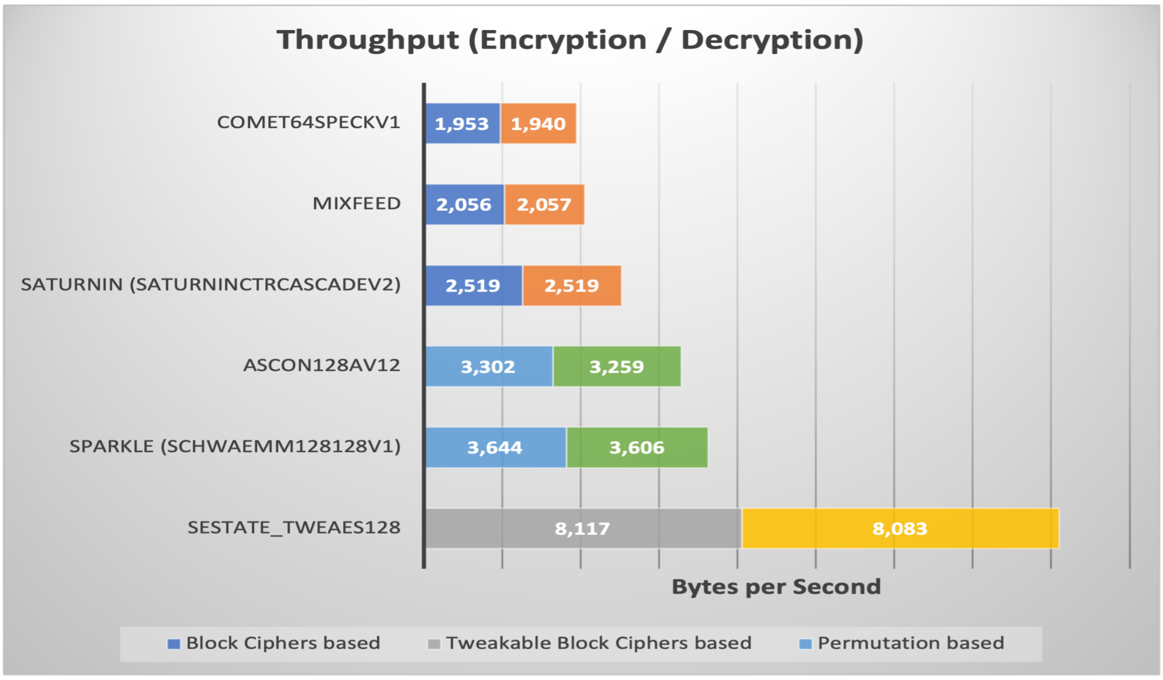


Fig. 6.14 The throughput of the six fastest primitives

`Saturnin-CTR-cascade-v2` is the fourth fastest primitive in our test. It is a variant of the block cipher Saturnin [37]. The primitive operates on 256-bit blocks, use a 256-bit key and has an internal state size of 256 bits. The nonce and the tag have variable lengths of up to 160 bits and 256 bits, respectively. It uses the counter CTR mode for encryption and a separate MAC (the Cascade construction [22]) for authentication. Unlike the previous three primitives, Saturnin requires two passes to generate the ciphertext and the authentication tag.

mixFeed [31] is the fifth-fastest primitive in our test. It is a block cipher based that employs a new version of the feedback paradigm called Minimally XORed Feedback mode, which the designers develop. The primitive uses a mixture of plaintext and ciphertext as feedback to a modified version of the AES-128 cipher, its underlying cipher. mixFeed is single-pass and inverse free, and its encryption and decryption function are parallelisable. It takes a 120-bit nonce, a 128-bit key, outputs a 128-bit authentication tag.

Figure 6.15 shows the throughput of the remaining candidates. `Oribatida-192-v11` [27] is the slowest primitive in our test, which can only encrypt/decrypt 29 bytes in one second. The primitive was also the slowest on our 32-bit ARM microcontroller. The design of the primitives is based on the SimP-192 permutation algorithm that is built around the Simon block cipher. Simon is optimised for hardware performance and not software friendly. The `Oribatida-192` takes a 64-bit nonce, a 128-bit key, outputs a 96-bit authentication tag.

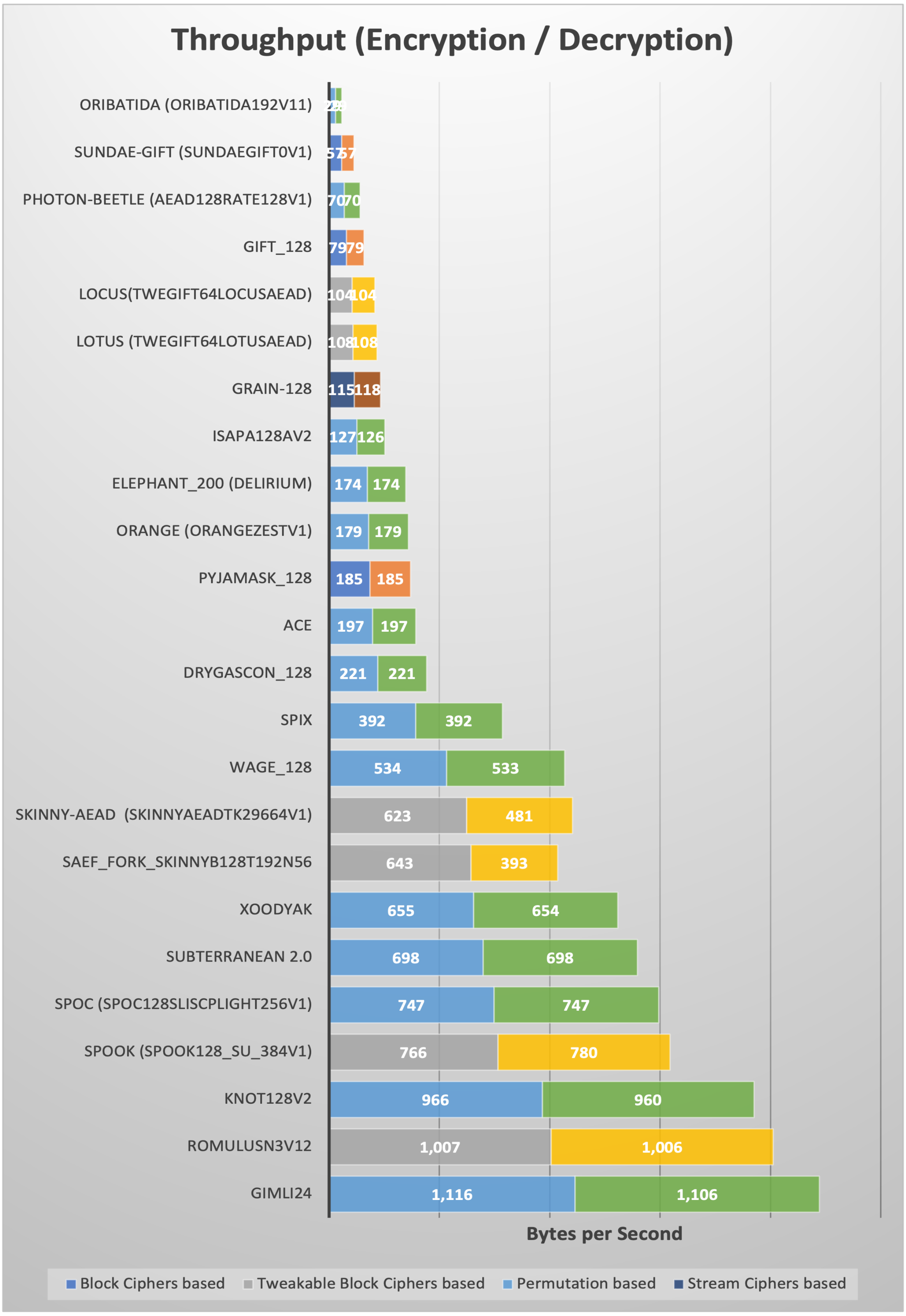


Fig. 6.15 The throughput of the remaining primitives

Sundae-GIFT-0v1, a variant of SUNDAE-GIFT [15] candidate, is the second slowest primitive. It is based on the GIFT-128 block cipher, which is an update to the PRESENT cipher. PRESENT is a hardware optimised cipher. Its bit permutation operation makes it software unfriendly. GIFT-128 and the primitives that are built based on it (e.g. Locus and Lotus) are among the slowest in our test

The RAM footprints of the candidates are shown in Figure 6.16. Due to the large number of variants, we only show the candidate's variant that consumes the least amount of memory. `Gimli-24` and `Ascon-128-av12` has the smallest memory footprint. `Ascon-128-av12` is defined using only bitwise Boolean functions, including AND, NOT, XOR and the bitwise rotation ROT. It does not require any table lookups, which saves memory. The primitive is based on the sponge construction, and it does not need to implement the inverse of the permutation or key schedule, masks, Galois field multiplications [54]. This, with the small state size, makes `Ascon-128-av12`'s RAM footprint minimum. `Gimli-24` is built on top of a permutation cipher called Gimli and has a small state size, resulting in its reduced memory consumption.

`Twe-GIFT-64-Locus-AEAD`, a Locus's variant and `Twe-GIFT-64-Lotus-AEAD`, a Lotus's variant, have the most significant memory usage. Lotus and Locus primitives are designed by the same team [39]. Both primitives are based on `TweGIFT-64`, a tweakable variant of the `GIFT-64-128` block cipher. Lotus uses four `TweGIFT-64` operations for every 16 bytes of input, while Locus uses two `TweGIFT-64` operations for every 8 bytes of input [39]. In addition, Locus requires both encryption and decryption operation of `TweGIFT-64`. All this increases the overall code size and memory consumption of the primitives.

Figure 6.17 shows the code sizes of the primitives. `TinyJAMBU-128` [169], and `Gimli-24` have the smallest code sizes among all candidates. The `TinyJAMBU` primitive is a small variant of JAMBU [168] cipher that is based on a 128-bit keyed permutation. `TinyJAMBU-128` takes a 96-bit nonce, a 128-bit key, and outputs a 64-bit authentication tag. It has a state size of 128 bits and a small message block size of only 32 bits. The cipher repeatedly reuses the keyed permutation component, which is implemented in only 16 lines of code. This helps in reducing the code size of the primitive. `Gimli` has a simple design that repeats a small number of operations over and over again. This helps in constructing an implementation with small code size.

`SAEAES-128-64-64-v1`, a variant of SAEAES [128], has the largest code size. SAEAES is based on AES and uses the block cipher mode of operation SAEB [127]. The `SAEAES-128-64-64-v1` variant takes a 128-bit key, a 64-bit nonce, outputs a 64-bit authentication tag. After investigating the primitive's code provided by the submitters, we found that their inefficient implementation of the underlying cipher, AES increases the code size by about 8

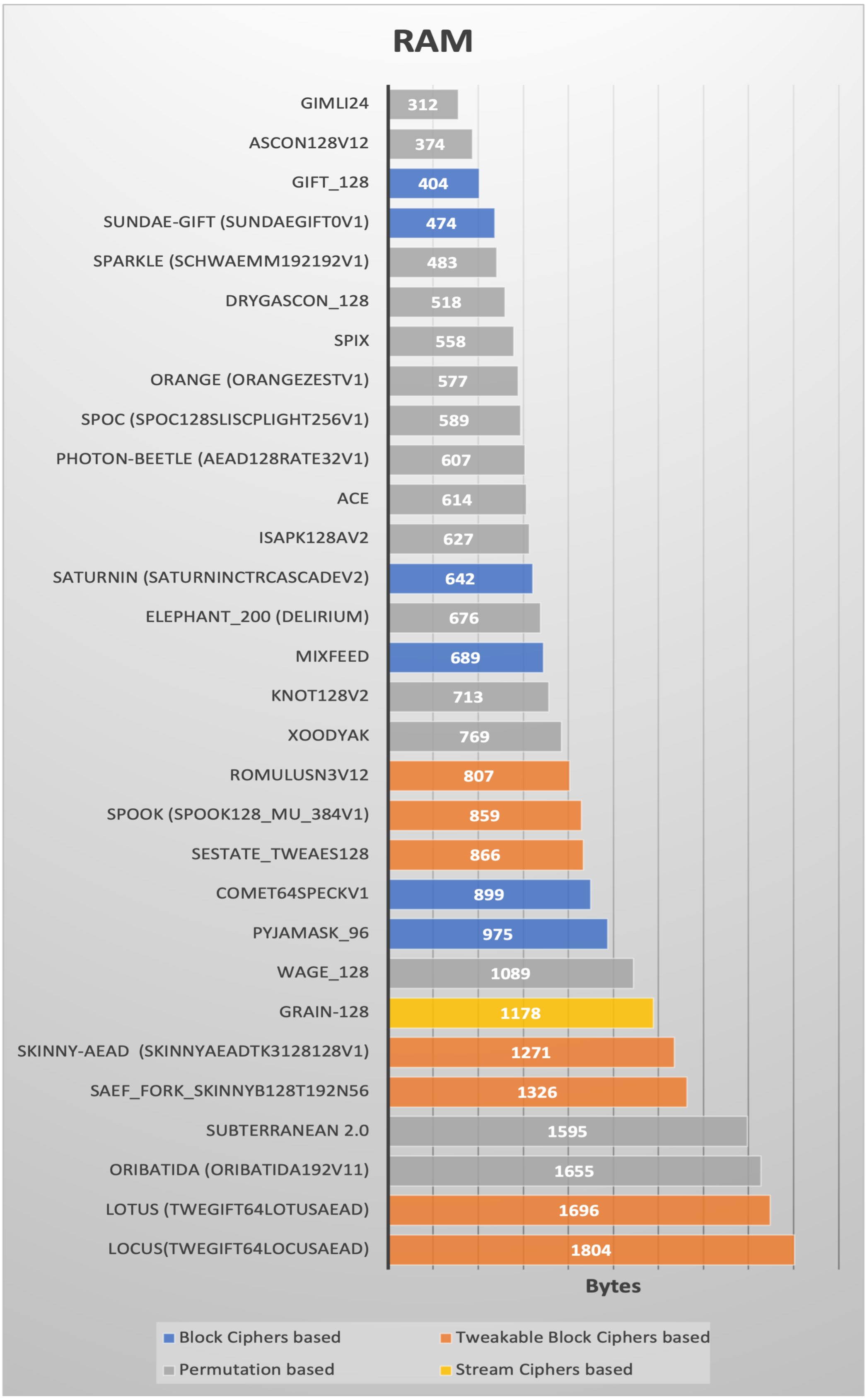


Fig. 6.16 The RAM footprint of the primitives

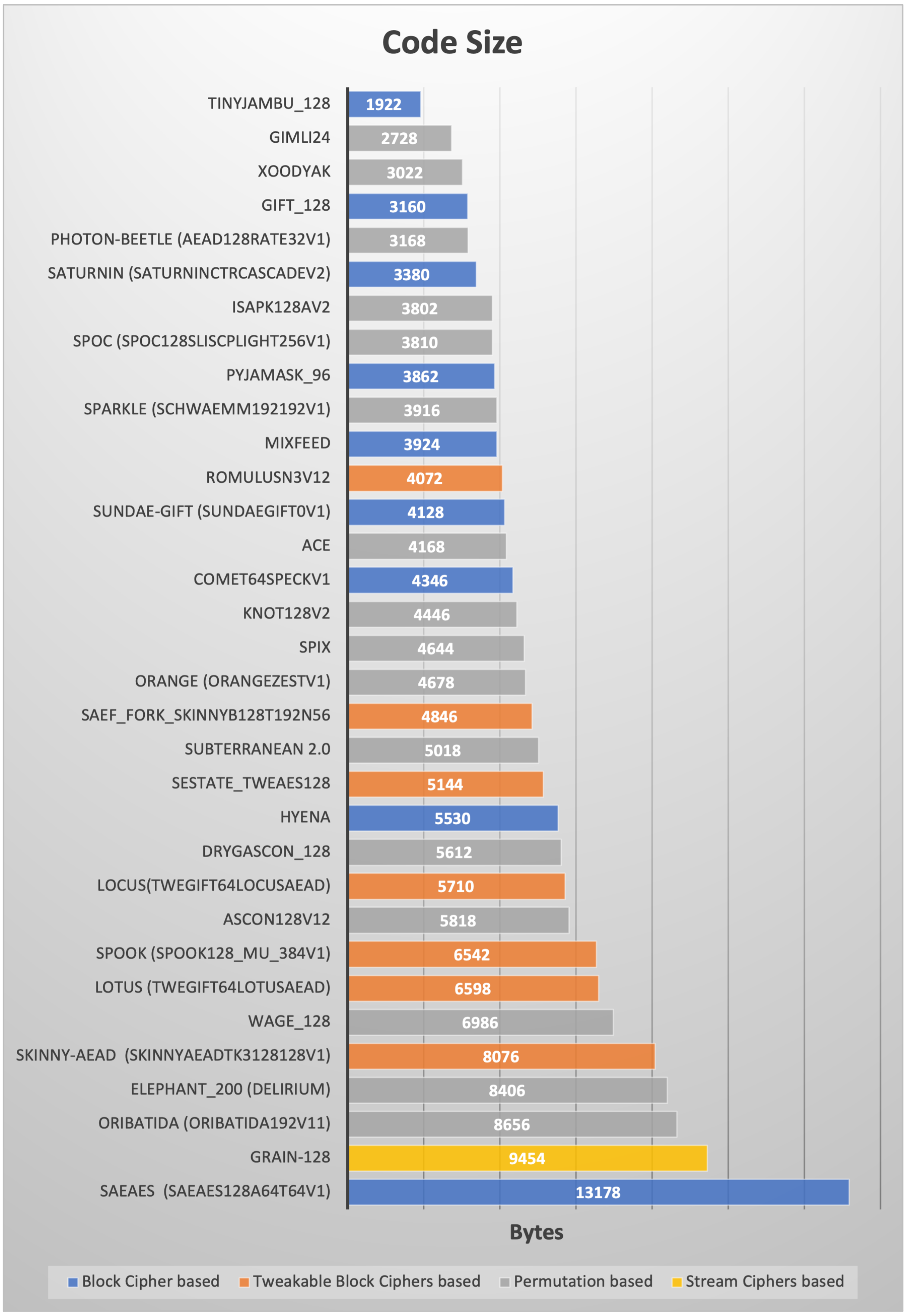


Fig. 6.17 The code sizes of the primitives

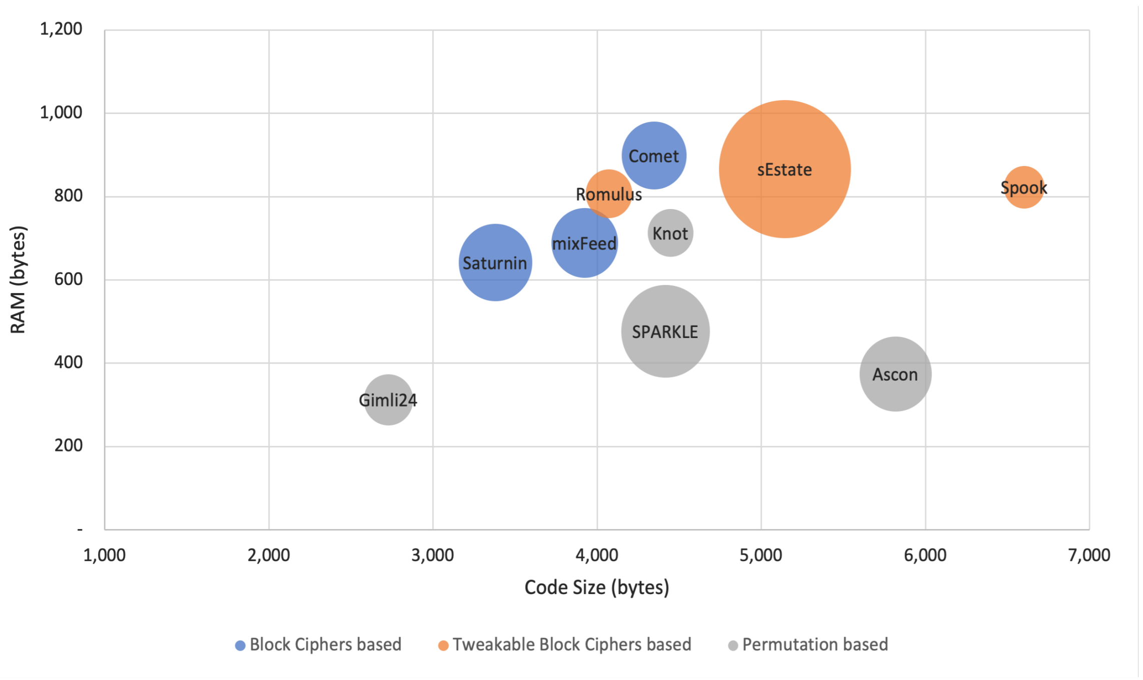


Fig. 6.18 Throughput vs ROM and RAM usage

Kilobytes. The code size of the cipher alone is about 5 Kilobytes which is around the average of the candidates code sizes.

`Grain-128-AEAD` [76] has the second-largest code size. It is the only authentication encryption candidate that is based on a stream cipher. `Grain-128-AEAD` adopts the design of `Grain-128` [75] and `Grain-v1` stream ciphers and consists of two building blocks; a 128-bit Linear Feedback Shift Register (LFSR) and a 128-bit Non-linear Feedback Shift Register (NFSR), which is an authenticator generator. It takes a 128-bit key and a 96-bit nonce and uses either 256 or 320 rounds. The cipher is bit-oriented that is optimised for hardware implementation.

Finally, we tried to find the best performing primitive that balances the three performance metrics. This was not an easy selection as primitives that are, for example, fast have a large RAM or ROM footprint. However, with the help of the graph in Figure 6.18, we see that `SPARKLE` can be a good candidate that balanced throughput (depicted as the bubble size), code size and RAM usage.

6.6 Conclusion

In this chapter, we evaluated the performance of the 32 round-2 candidates from the NIST lightweight cryptography competition. We measured their throughput and investigated the resource needed to run them. We also discussed how embedded programs are built and uploaded onto the constrained devices. In addition, we explained the different memory regions on constrained devices and how to measure memory consumption using static and dynamic methods. Finally, we introduced a method to measure the program binary size in isolation from the systems libraries.

The results of the experiment have shown many interesting observations. We summarise these observations as follows:

- The candidates are based on four cryptographic schemes, including block ciphers, tweakable block ciphers, permutation algorithms, and stream cipher. None of these schemes has shown any performance advantage over the others.
- Candidates with a small state size, a small message block size, do not require lookup tables, and are defined using only bitwise Boolean functions have shown a small RAM footprint.
- Candidates that do not require lookup tables, have a minimal implementation, and repeatedly reuse parts of the code have shown a small ROM footprint.
- Candidates that are single pass, inverse free, and feature parallelisable encryption and decryption, have shown a good throughput.
- sEstate is the fastest primitive tested. However, its RAM and ROM footprints are not among the minimum ones.
- Gimili-24 has the smallest code size and RAM usage.
- Trying to find a primitive that does well on all three performance metrics is challenging. For example, primitives that are optimised for speed incur extra code size or memory usage. Our experiment found that SPARKLE is the best performing primitive that balances the three metrics.

Chapter 7

Conclusion

Summary

This chapter summarises the work presented in this thesis and points to some directions for future work. The chapter also outlines the contributions and the limitations of our work.

7.1 Thesis Summary

In this thesis, we have studied the performance of a large number of cryptographic primitives. First, we investigated the effect of the primitives' building blocks, implementation, and optimisation techniques on their performance. After that, we explored the limitations imposed by the resource constraints of the constrained devices in implementing cryptography. Finally, we studied the performance of the 32 round-2 candidates from the NIST lightweight cryptography competition. We investigated their resource utilisation and their throughput.

This section summarises the work that has been carried out in this thesis and highlights the main contributions and the results obtained.

7.1.1 Investigating the Performance of `.NET` and BouncyCastle Cryptographic Libraries (Chapter 3)

In chapter 3, we conducted an experiment to evaluate the performance of all cryptographic primitives implemented in the Microsoft `.NET` cryptographic library and the C# version of the BouncyCastle library. In this experiment, we tested primitives from the four major cryptographic schemes, including block and stream ciphers, public-key ciphers, hash functions and MAC algorithms. The experiment has shown that the cipher's performance is not determined by its internal structure and building blocks alone. The cipher's implementation has a considerable impact on its performance; for example `.NET`'s implementation of SHA1 is about three times faster than BouncyCastle's. The experiment has also shown the significant effect of the hardware acceleration techniques on the cipher's performance. For example, the `.NET` implementation of AES, which take advantage of AES-NI instruction sets, is 12 times faster than BouncyCastle's native implementation.

From the extensive list of the cryptographic primitives we have tested, the results show that `.NET`'s AES is by far the most efficient symmetric cipher at an average speed of 300 MB/second. The `.NET`'s implementation of the primitive is accelerated using the AES-NI instruction set. As for the hash and MAC functions, the `.NET`'s SHA1 and HMAC-MD5 are the fastest hash and MAC functions, respectively. Finally, the `.NET`'s implementation of RSA primitive significantly outperforms its counterpart in the BouncyCaslte library.

7.1.2 Investigating the Performance of C and C++ Cryptographic Libraries (Chapter 4)

Chapter 4 presented a performance comparison of several cryptographic primitives from six C and C++ cryptographic libraries. The chapter aimed to understand the effect of the primitive's building blocks, different parameters (block size, key length, number of rounds, etc.) and various optimisation techniques on its performance. We first surveyed the available cryptographic libraries and then chose six well-known, open-source, and well-maintained libraries that offer implementations for a wide range of primitives from the different cryptographic schemes. After that, we benchmarked the implementation of each primitive with its counterparts from other libraries as well as primitives from the same cryptographic scheme

To write a robust benchmark test, one should first understand the problem and clearly define the benchmarking goals. Then choose the relevant metrics that correspond to the defined goals. Next, select the appropriate methodology and tools. Finally, conduct the experiment, analyse the results and draw conclusions based on the obtained data.

A sound benchmark should satisfy certain conditions, including repeatability, verifiability, fairness, and relevance. In addition, the benchmark should be non-invasive so that it will not affect the benchmark outcome. The latter condition is not easy to satisfy; however, we should minimise the effect of the measuring tools on the results or keep this effect in mind when analysing the results.

Repeatability is another condition that is not always easy to satisfy. We have seen in our experiments that writing a benchmark that produces stable and consistent results is challenging. One of the key challenges is the benchmark noise caused mainly by the operating system and the CPU. Therefore, during the benchmark, one should watch for the CPU frequency scaling where the CPU changes its frequency based on system workloads, temperature and operating system power-save policies. In addition, the CPU can turn off some of its core to reduce power consumption. When any of these happens, we get misleading benchmarking results. In Section 5.3.1 we explain how to minimise the noise from the CPU by setting the CPU governor mode to Performance Mode.

In our experiment, we have seen other issues that affect the accuracy of the benchmarking results and should be watched for. First, during the cold start of the application, the readings are higher than the others. Cold start happens when the system starts loading assemblies, initialising objects and populating the cache. It also occurs when bringing the CPU out of power save mode. Therefore, it is wise to exclude these readings from any calculations.

Another source of inaccuracy comes from the compiler when it optimises away part of our benchmarked code, giving misleading results. This often happens during micro-benchmarking and can be avoided, as we explain in Section 2.3.

Other sources of inaccuracy we have seen include running the benchmark code in a debug mode or under a busy system and using a stopwatch with insufficient accuracy or large timing resolution.

There are some statistical properties that help achieve a high measurement accuracy. For example, we use the *standard error* to measure how far the true mean is from our calculated average. A larger value indicates low accuracy and means we need to increase our number of iteration. In addition, we use the *standard deviation* to measure how spread out our results away from the mean. A huge spread indicates low benchmark accuracy. Furthermore, we use the *confidence interval*, which gives us a range of plausible values that we are reasonably sure our true mean lies in. If the confidence interval range is too wide, then our calculated average cannot be trusted.

The results from the extensive list of primitives tested show that primitives implementations written in C/C++ are highly optimised and more efficient than those written in C#. This makes C/C++ cryptographic libraries a top choice for applications with real-time constraints.

The results also show the stream ciphers are faster than block ciphers; however, their lengthy initialisation phase makes them inefficient for encrypting a small amount of data.

The key size significantly impacts the cipher's security. Therefore it is always advised to use larger key sizes. However, some application developers have concern that using larger key sizes will negatively impact the performance. In our experiment, we found that this is true only when the key size dictates the cipher's number of rounds, and even then, the effect is small and does not justify using smaller key sizes with symmetric ciphers.

The results also show that in applications that mainly process a larger amount of data (e.g. disk encryption), it is better to select ciphers with a larger block size (e.g. 128 bits). This is because ciphers with larger blocks can process bulky data faster than ciphers with smaller blocks.

Our results also show that ciphers optimised to use hardware acceleration techniques give better performance than their counterparts that do not utilise these techniques. However, each CPU offers different acceleration instructions sets, and some do not support these acceleration features. This should be considered as some primitive implementations are only optimised for one type of CPUs and may not work with other CPUs or with different versions of the same CPU. Nevertheless, most of the implementations we have seen have a fall-back mechanism on native implementation in the absence of the hardware acceleration features.

Finally, AES cipher shows an outstanding performance in the different implementations from all libraries tested even when no hardware acceleration is used. So, we think it should be a top choice for developers looking for an efficient cipher to use in desktop and server environments.

7.1.3 Performance of C and C++ Cryptographic Libraries under a Constrained Device (Chapter 5)

Chapter 5 investigated the performance of a large number of cryptographic primitives when run under a constrained environment. The chapter first discussed three methods for measuring the energy consumption of a computer program; using an instruction-level power model, using Linux command-line tools, and using power analyser devices. Then, it investigated the throughput of the primitives in addition to their energy consumption. The chapter also studied the impact of the key length, block size, and the number of rounds on the power consumption of the primitives. Additionally, it examined the performance of the primitives' hardware optimised implementations in the absence of any hardware acceleration support.

The first method of measuring the program code energy consumption we discuss in this chapter uses an instruction-level power model that quantifies the energy cost of a program machine instructions. This method calculates the energy cost of individual instructions by multiplying the number of cycles the instruction takes by the current it draws.

Although accurate, this method is not practical if we have a large program code, as in many cryptographic primitives. In addition, it requires a good knowledge of the assembly language. Also, it requires knowing the amount of current drawn by each machine instruction and the number of clock cycles it takes an instruction to execute. Unfortunately, both quantities are not published for every CPU model.

The second method we discuss is using Linux command-line tools. This method is not complicated as the first one. However, it is less accurate as the energy measurements are estimates provided by an embedded power model. This method works as follows: first, we measure the energy consumption while the system is idle to establish a baseline. Then, we measure the energy consumption while our program is running. After that, we subtract the two readings to get an estimate of our program energy consumption. For the systems that support Intel's RAPL (a set of counters that provide information on the energy consumption), Linux estimates the energy consumption based on information from RAPL's counters. The accuracy of the power measurements from these counters has been studied in many research papers. It appears that the accuracy can vary based on the type of the processor. However, RAPL shows a promising accuracy, and its estimation is sufficiently good. We recommend using more than one tool for estimating the energy to get a better estimation when using this method.

The third method is using power analysers such as multimeters and oscilloscopes. These measurement devices measure the electrical properties, including the voltage and the current drawn from the power supply allowing us to calculate the power consumption of the program code.

For a detailed analysis of the power traces of the different parts of the program code, one should use an oscilloscope or other specialised power analysers as they have a high sampling rate, offering more accurate results than the multimeters. However, these devices are expensive and require a fair amount of knowledge and experience with electronics and electrical engineering to use them.

We used a USB multimeter which measures the voltage and the current as the electric power passes through it to calculate the energy consumption of the primitives. These devices are simple to use and cheap alternatives to specialised power analysers. However, their slow sampling rates make them unsuitable for measuring the power consumption of the individual building blocks of the primitives.

From the experiment we presented in this chapter, we conclude that calculating the actual energy consumption of a primitive is complex and requires specialised tools or an instruction-level power model. On the other hand, estimating the energy consumption using multimeters or the operating system power model is simple; however, you need to reduce the measurement noise by turning off all other nonsystem applications so they do not negatively affect the results. In addition, you should take the readings many times and aggregate the results. The noise is usually random, so it spoils only some measurements but not all of them.

We also think that the energy consumption metric is a more relevant metric than the power consumption for evaluating the primitive performance since it considers the primitive's execution time and measures the total power consumed during that time.

Additionally, we found that most of the tested primitives have a comparable power consumption to a certain degree. This makes the primitive's execution time the most critical factor in determining the energy consumption of the primitives.

The results also show that the block ciphers with simple and small substitution boxes show small power consumption and thus are suitable for power-conscious systems. This is true for the stream ciphers with small internal sizes and simple feedback functions such as RC4.

Finally, reducing the key size in the hope of reducing the power consumption of a primitive is not wise, as the results show that the key size does not affect the power consumption of the primitive. However, the key size has an impact on the speed of a few primitives and thus their energy consumption.

7.1.4 Investigating the Performance of Lightweight Cryptography under Constrained Devices (Chapter 6)

Chapter 6 investigated the performance of the 32 round-2 candidates from the NIST lightweight cryptography competition. It presented the experiment we conducted to evaluate the implementation of the candidates and study their throughput and resource utilisation. In addition, it described the building process of an embedded program and explained how to develop and upload a program into embedded devices. Furthermore, the chapter introduced a method to calculate the code size of a program in isolation from the system libraries. This method helps in calculating the primitive code size with high accuracy. Finally, the chapter offered a description for the various sections of the constrained device's memory and explained different static and dynamic methods for investigating the memory usage of embedded software.

Writing a program for constrained devices is challenging due to their constraints, especially the RAM and disk constraints. In addition, these constraints make it challenging to attach a debugger or profiler (e.g. performance or memory profilers) to debug the code while it is running or to monitor its resource utilisation. For this reason, many developers use static analysis approaches to debug their code or to measure its RAM and ROM footprints before deploying it to the constrained devices. However, while measuring the code size statically can be accurate, measuring the RAM footprint of an application is only accurate if done dynamically. This is because the memory requirements change during the execution of the program.

Studying the memory and flash drive utilisation requires knowing the different sections of the ELF file, especially the *text*, *data* and *bss* and how the information is arranged in these sections. In addition, it requires knowing the different commands and compiler options used to inspect the various sections and display their contents. Furthermore, analysing the memory utilisation requires knowing the heap and stack areas of the memory, their boundaries and the different symbols assigned to them by the linkers.

The ELF file contains not only our code but also the code from the device system libraries. Therefore, we have to separate the two before calculating the RAM and ROM footprint. This is not easy as compilers sometimes add suffixes and prefixes to the names of the element in the ELF, making it difficult to recognise them. The other challenge is the large number of elements we need to inspect. For this reason, we introduced our method of calculating the code size and the memory's *data* and *bss* sections.

Calculating the heap and stack regions is another challenge as they constantly grow and shrink during the program's execution. In this chapter, we discussed different approaches to measure these two memory regions. However, each has its limitations. We think the stack painting is more practical and more accurate than the others. This method requires that we know the memory layout of the device and the different linker symbols for the various sections of the memory. These symbols and the memory layout are not the same for all constrained devices.

In this experiment, we looked at the primitives' characteristics and their impact on the performance, and we concluded that primitives with small state and block sizes, do not have lookup tables, and are defined using only bitwise boolean functions have low memory requirements and thus suited for devices with stringent memory constraints.

Additionally, primitives that do not require lookup tables, have a minimal implementation, and repeatedly reuse parts of the code are the right option for devices with minimal storage.

Furthermore, if we are concerned about throughput, then primitives that are single pass, inverse free, and feature parallelisable encryption and decryption operations have shown a good throughput.

Finally, trying to find a primitive that does well on all three performance metrics is challenging. For example, primitives that are optimised for speed incur extra code size or memory usage. The only primitive we found to show a good balance between the three metrics is `SPARKLE`.

7.2 Thesis Contribution

This thesis has contributed to the field of cryptographic primitive performance evaluation in several ways. The following are the thesis' key contributions:

- Performance evaluation of all cryptographic primitives implemented in .NET cryptographic library and all the symmetric ciphers, hash and MAC functions implemented in BouncyCastle library.

- Performance evaluation of all symmetric ciphers, Hash functions and MAC algorithms implemented in Crypto++, Botan, Nettle, Libgcrypt, Libtomcrypt, and OpenSSL cryptographic libraries.
- Identifying the different techniques for measuring the energy consumption of a computer program or part of it and using one of these methods to measure the energy consumption of all primitives from the previous bullets.
- Introducing a method for measuring the code size of an embedded program in isolation of the system libraries. This method helps in measuring the code size of the cryptographic primitives accurately.
- Identifying different static and dynamic methods for measuring the maximum stack utilisation and heap allocation in embedded systems. These methods help in determining the RAM footprint of cryptographic primitives.
- Performance evaluation of the 32 round-2 candidates from the NIST lightweight cryptography competition. The evaluation included measuring the resource required for running these lightweight primitives in resource-constrained environments.

7.3 Limitation and Future Work

In this thesis, we have looked at the non-functional properties of the implementations of the cryptographic primitives, principally throughput, memory usage and energy consumption. However, we have not considered security evaluation which is clearly a key component of the success of any cryptographic primitive. Furthermore, while we have considered the primitive's structure in order to attempt to explain the performance results derived, the measurements are based on its implementation, not the theoretical performance of the primitive itself. It is, therefore, possible that a more efficient implementation might be possible, which improves the results observed in this thesis.

The primitives submitted to the NIST competition provide authentication encryption and hashing functionalities. In our evaluation of the submitted primitives, we did not evaluate the performance of the hash functions. In addition, we did not evaluate any MAC algorithm or public-key cipher as none of them is in this competition.

Public-key ciphers depend heavily on mathematics. Their encryption and decryption operations are computationally intensive, making their requirements for resources much larger than that of symmetric-key ciphers. Thus, although they are in demand for the key management protocols in constrained environments, their high cost of implementation limited their use in these environments. Unfortunately, we have seen only a few papers that study the performance of these ciphers on constrained devices. So our work can be extended to evaluate their performance, especially those based on the algebraic structure of elliptic curves such as Elliptic Curve Digital Signature Algorithm (ECDSA) and Elliptic-curve Diffie–Hellman (ECDH). The Elliptic Curve ciphers use shorter keys, resulting in a relatively small RAM footprint and faster field arithmetic operations, making them more suitable for constrained environments than the other asymmetric ciphers.

A second limitation of our work is that we only considered the Cipher Block Chaining (CBC) mode of operation for the block ciphers as it is the most commonly used one. So, other block cipher modes such as the OFB, CFB, CTR, ECB, and XTS can be included for future work.

Another limitation of our work is that we could not measure the energy consumption of the NIST candidates as the power resolution of our multimeter is not good enough to capture the very small current and voltage readings. Measuring the energy consumption on constrained devices requires a high-resolution oscilloscope or specially designed measurement circuitry. Nevertheless, measuring the energy consumption of lightweight cryptography will provide valuable insight into their energy needs. This work can be considered for future research.

We evaluated the 32 round-2 candidates from the NIST competition on an 8-bit microcontroller. Investigating the performance of these primitives on a 16-bit or 32-bit architecture will likely give different results and provide more insight into the primitives' performance. This investigation can be carried out in future research.

Finally, in the future, there will always be new improved implementations, new optimisation techniques, new systems and hardware. These developments will lead to different performance results than the ones observed in this thesis. However, this thesis focuses on developing a long-lasting measurement method that will consistently give repeatable and accurate results across different operating systems and constrained devices.

References

[1] Acosta, A. J., Tena-Sánchez, E., Jiménez, C. J., and Mora, J. M. (2017). Power and energy issues on lightweight cryptography. *Journal of Low Power Electronics*, 13(3):326–337.

[2] Adams, C. M. (1997). Constructing symmetric ciphers using the CAST design procedure. *Designs, Codes and Cryptography*, 12(3):283–316.

[3] Akinshin, A. (2019). *Pro .NET Benchmarking*, pages [211–213]. Apress, 1st edition.

[4] Akinshin, A. and Dorokhov, A. (2021). perfolizer, performance analysis toolkit. [Online] https://github.com/AndreyAkinshin/perfolizer. Last accessed on Mar 23, 2021.

[5] Andriesse, D. (2018). *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*, page [52]. No Starch Press, Incorporated.

[6] Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., and Tokita, T. (2000). Camellia: A 128-bit block cipher suitable for multiple platforms, design and analysis. In *International Workshop on Selected Areas in Cryptography*, pages 39–56. Springer.

[7] Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., and Tokita, T. (2001). Specification of Camellia a 128-bit block cipher. *Nippon Telegraph and Telephone Corporation and Mitsubishi Electric Corporation*, 2.

[8] Arduino Editors (2021a). Arduino IDE. [Online] https://www.arduino.cc/en/software. Last accessed 4 Jun, 2021.

[9] Arduino Editors (2021b). ARDUINO MKR WIFI 1010. [Online] https://store.arduino.cc/arduino-mkr-wifi-1010. Last accessed 14 Jun, 2021.

[10] Arduino Editors (2021c). ARDUINO UNO REV3. [Online] https://store.arduino.cc/arduino-uno-rev3. Last accessed 14 Jun, 2021.

[11] Arduino Editors (2021d). micros(). [Online] https://www.arduino.cc/reference/en/language/functions/time/micros/. Last accessed 14 Jun, 2021.

[12] Arduino Editors (2021e). millis(). [Online] https://www.arduino.cc/reference/en/language/functions/time/millis/. Last accessed 14 Jun, 2021.

[13] Argyroudis, P. G., Verma, R., Tewari, H., and O'Mahony, D. (2004). Performance analysis of cryptographic protocols on handheld devices. In *Third IEEE International Symposium on Network Computing and Applications, 2004.(NCA 2004). Proceedings.*, pages 169–174. IEEE.

[14] ARM Editors (2014). ARM® Cortex®-A53 MPCore Processor Cryptography Extension. Technical report, ARM.

[15] Banik, S., Bogdanov, A., Peyrin, T., Sasaki, Y., Sim, S. M., Tischhauser, E., and Todo, Y. (2019). Sundae-gift. *Submission to Round*, 1.

[16] Banik, S., Bogdanov, A., and Regazzoni, F. (2015). Exploring energy efficiency of lightweight block ciphers. In *International Conference on Selected Areas in Cryptography*, pages 178–194. Springer.

[17] Barnes, R. (2016). Raspberry Pi 3: Specs, benchmarks and testing. [Online] https://magpi.raspberrypi.org/articles/raspberry-pi-3-specs-benchmarks. Last accessed 4 May, 2018.

[18] Bassham, L., Çalık, Ç., McKay, K., Mouha, N., and Sönmez Turan, M. (2017). Profiles for the lightweight cryptography standardization process. Technical report, National Institute of Standards and Technology.

[19] Batina, L., Das, A., Ege, B., Kavun, E. B., Mentens, N., Paar, C., Verbauwhede, I., and Yalçın, T. (2013). Dietary recommendations for lightweight block ciphers: Power, energy and area analysis of recently developed architectures. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 103–112. Springer.

[20] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., and Wingers, L. (2015). The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6.

[21] Beierle, C., Biryukov, A., dos Santos, L. C., Großschädl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q., and Biryukov, A. (2019). Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family. *NIST round*, 2.

[22] Bellare, M., Canetti, R., and Krawczyk, H. (1996). Keying hash functions for message authentication. In *Annual international cryptology conference*, pages 1–15. Springer.

[23] Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M., et al. (2008). Sosemanuk, a fast software-oriented stream cipher. In *New Stream Cipher Designs*, pages 98–118. Springer.

[24] Bernstein, D. J. (2008). ChaCha, a variant of Salsa20. In *Workshop record of SASC*, volume 8, pages 3–5.

[25] Bertoni, G., Breveglieri, L., Fragneto, P., Macchetti, M., and Marchesin, S. (2002). Efficient software implementation of aes on 32-bit platforms. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 159–171. Springer.

[26] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. (2007). Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer.

[27] Bhattacharjee, A., List, E., Lpez, C., and Nandi, M. (2019). The oribatida family of lightweight authenticated encryption schemes.

[28] Bingmann, T. (2008). Speedtest and comparison of open-source cryptography libraries and compiler flags. [Online] https://panthema.net/2008/0714-cryptography-speedtest-comparison/. Last accessed on Jun 01, 2018.

[29] Biryukov, A. and Perrin, L. P. (2017). State of the art in lightweight symmetric cryptography.

[30] Bisht, A. (2021). Stack vs Heap Memory Allocation . [Online] https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/. Last accessed 8 Jun, 2021.

[31] Bishwajit Chakraborty, M. N. (2019). mixFeed. *The Lightweight Cryptography NIST Competition*.

[32] Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J., Seurin, Y., and Vikkelsoe, C. (2007). PRESENT: An ultra-lightweight block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 450–466. Springer.

[33] Bormann, C., Ersue, M., and Keranen, A. (2014). Terminology for constrained-node networks. *Internet Engineering Task Force (IETF): Fremont, CA, USA*, pages 2070–1721.

[34] Botcazou, E., Comar, C., and Hainque, O. (2005). Compile-time stack requirements analysis with GCC. In *Proceedings of the 2005 GCC Developer's Summit*, page 93. Citeseer.

[35] Bouncy Castle Editors (2021). The Legion of the Bouncy Castle. [Online] https://www.bouncycastle.org/csharp/. Last accessed on Mar 20, 2021.

[36] BSI Editors (2020). BSI project: Development of a secure crypto library. [Online] https://www.bsi.bund.de/EN/Topics/Cryptography/CryptoLibrary/crypto_library_node.html. Last accessed on Apr 3, 2021.

[37] Canteaut, A., Duval, S., Leurent, G., Naya-Plasencia, M., Perrin, L., Pornin, T., and Schrottenloher, A. (2019). Saturnin: a suite of lightweight symmetric algorithms for post-quantum security. *The Lightweight Cryptography NIST Competition*.

[38] Carnie, G. (2012). The C build process. [Online] https://blog.feabhas.com/2012/06/the-c-build-process/. Last accessed 1 Jun, 2021.

[39] Chakraborti, A., Datta, N., JHA, A., Lopez, C., Nandi, M., and Sasaki, Y. (2019). LOTUS-AEAD and LOCUS-AEAD. *The Lightweight Cryptography NIST Competition*.

[40] Chakraborti, A., Datta, N., Jha, A., Mancillas-López, C., Nandi, M., and Sasaki, Y. (2020a). Estate: A lightweight and low energy authenticated encryption mode. *IACR Transactions on Symmetric Cryptology*, pages 350–389.

[41] Chakraborti, A., Datta, N., Jha, A., and Nandi, M. (2020b). Structural classification of authenticated encryption schemes.

[42] Coowoo Editors (2018). COOWOO USB Digital Power Meter Tester. [Online] http://www.coowootech.com/tools.html. Last accessed 8 May, 2018.

[43] CPPReference Team (2021). C++ Date and time utilities. [Online] https://en.cppreference.com/w/cpp/chrono. Last accessed 5 Apr, 2021.

[44] Daemen, J. and Rijmen, V. (2020). *The design of Rijndael. Information security and cryptography, 2nd edition*, pages [33–41]. Springer Berlin.

[45] Daemen, J. and Van Assche, G. (2007). Producing collisions for PANAMA, instantaneously. In *International Workshop on Fast Software Encryption*, pages 1–18. Springer.

[46] Dai, W. (2021). Crypto++ Library. [Online] http://www.cryptopp.com. Last accessed on Apr 3, 2021.

[47] Danilova, E. (2015). Comparing the performance of different vpn technologies with tls/ssl.

[48] De Canniere, C., Dunkelman, O., and Knežević, M. (2009). KATAN and KTANTAN—a family of small and efficient hardware-oriented block ciphers. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 272–288. Springer.

[49] Delfs, H. and Knebl, H. (2007). *Introduction to Cryptography Principles and Applications*, pages [34,108, 167, 168, 169]. Springer.

[50] Denis, T. S. (2020). LibTomCrypt. [Online] http://www.libtom.net/LibTomCrypt. Last accessed on Apr 3, 2021.

[51] Desrochers, S., Paradis, C., and Weaver, V. M. (2016). A validation of DRAM RAPL power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, pages 455–470.

[52] Dinu, D., Biryukov, A., Großschädl, J., Khovratovich, D., Le Corre, Y., and Perrin, L. (2015). Felics–fair evaluation of lightweight cryptographic systems. 128.

[53] Dinu, D., Perrin, L., Udovenko, A., Velichkov, V., Großschädl, J., and Biryukov, A. (2016). Design strategies for arx with provable bounds: Sparx and lax. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 484–513. Springer.

[54] Dobraunig, C., Eichlseder, M., Mendel, F., and Schläffer, M. (2016). Ascon v1. 2. *Submission to the CAESAR Competition*.

[55] dotnet (2021). BenchmarkDotNet, powerful .NET library for benchmarking. [Online] https://github.com/dotnet/BenchmarkDotNet. Last accessed on Mar 23, 2021.

[56] Dworkin, M. (2001). Recommendation for block cipher modes of operation: Methods and techniques. *NIST Special Publication.*, SP 800-38A:11.

[57] Dworkin, M., Barker, E., Nechvatal, J., Foti, J., Bassham, L., Roback, E., and Dray, J. (2001). Advanced encryption standard (AES), Federal Inf. Process. Stds. (NIST FIPS). [Online] https://doi.org/10.6028/NIST.FIPS.197. Last accessed on Mar 26, 2021.

[58] Easttom, W. (2021). *Modern Cryptography: Applied Mathematics for Encryption and Information Security*, pages [137,179, 190, 206]. Springer, New York.

[59] Ehrsam, W. F., Meyer, C. H., Smith, J. L., and Tuchman, W. L. (1976). Message Verification and Transmission Error Detection by Block Cchaining. *US Patent 4074066*.

[60] Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., and Uhsadel, L. (2007). A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6):522–533.

[61] Engels, D., Saarinen, M.-J. O., Schweitzer, P., and Smith, E. M. (2011). The hummingbird-2 lightweight authenticated encryption algorithm. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pages 19–31. Springer.

[62] Eranian, S. (2015). Linux kernel profiling with Perf. [Online] https://perf.wiki.kernel.org/index.php/Tutorial. Last accessed 3 May, 2021.

[63] Espressif Editors (2021). ESP8266, A cost-effective and highly integrated Wi-Fi MCU for IoT applications. [Online] https://www.espressif.com/en/products/socs/esp8266. Last accessed 14 Jun, 2021.

[64] Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., and Walker, J. (2010). The Skein hash function family. *Submission to NIST (round 3)*, 7(7.5):3.

[65] Ferguson, N., Schneier, B., and Kohno, T. (2013). *Cryptography engineering: design principles and practical applications*, pages [3,43, 52, 78, 89]. John Wiley & Sons.

[66] Flaga, M. and Mistry, S. (2016). Arduino library for Measuring and use less Free RAM. [Online] https://github.com/mpflaga/Arduino-MemoryFree. Last accessed 12 Jun, 2021.

[67] Francia III, G. A. and Francia, R. R. (2007). An empirical study on the performance of Java/. Net cryptographic APIs. *Information Systems Security*, 16(6):344–354.

[68] GCC Contributors (2021). GCC Developer Options. [Online] https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html. Last accessed 9 Jun, 2021.

[69] Gray, J. (1992). *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann Publishers Inc.

[70] Hackenberg, D., Ilsche, T., Schöne, R., Molka, D., Schmidt, M., and Nagel, W. E. (2013). Power measurement techniques on standard compute nodes: A quantitative comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204. IEEE.

[71] Hähnel, M., Döbel, B., Völp, M., and Härtig, H. (2012). Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17.

[72] Hamon, D. (2021). Google benchmark. [Online] https://github.com/google/benchmark. Last accessed on May 06, 2021.

[73] Hatzivasilis, G., Fysarakis, K., Papaefstathiou, I., and Manifavas, C. (2018). A review of lightweight block ciphers. *Journal of cryptographic Engineering*, 8(2):141–184.

[74] Hawkes, P. and Rose, G. G. (2003). Primitive Specification for SOBER-128. *IACR Cryptol. ePrint Arch.*, 2003:81.

[75] Hell, M., Johansson, T., and Meier, W. (2007). Grain: a stream cipher for constrained environments. *International Journal of Wireless and Mobile Computing*, 2(1):86–93.

[76] Hell, M., Johansson, T., Meier, W., Sönnerup, J., and Yoshida, H. (2019). An aead variant of the grain stream cipher. In *International Conference on Codes, Cryptology, and Information Security*, pages 55–71. Springer.

[77] Hinton, P. R. (2014). *Statistics explained*, pages [13–14, 57–58, 85–86]. Routledge.

[78] Hofmann, F. (2019). Understanding the ELF File Format. [Online] https://linuxhint.com/understanding_elf_file_format/. Last accessed 1 Jun, 2021.

[79] Hong, D., Lee, J.-K., Kim, D.-C., Kwon, D., Ryu, K. H., and Lee, D.-G. (2013). Lea: A 128-bit block cipher for fast encryption on common processors. In *International Workshop on Information Security Applications*, pages 3–27. Springer.

[80] Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B.-S., Lee, C., Chang, D., Lee, J., Jeong, K., et al. (2006). Hight: A new block cipher suitable for low-resource device. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 46–59. Springer.

[81] Hughes, R. (2010). UPower reference manual. [Online] https://upower.freedesktop.org/docs/. Last accessed 1 May, 2021.

[82] Huppler, K. (2009). The art of building a good benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 18–30. Springer.

[83] Intel Corporation (2015). Improving OpenSSL performance. [Online] https://software.intel.com/en-us/articles/improving-openssl-performance. Last accessed 3 Oct, 2018.

[84] Intel Editors (2016). Intel® 64 and IA-32 Architectures Software Developer's Manual. Technical report, Intel.

[85] ISO/IEC (2011a). Information technology — Security techniques — Message Authentication Codes (MACs) — Part 1: Mechanisms using a block cipher. *International Organisation for Standardisation, Geneva, Switzerland*, 9797-1:40.

[86] ISO/IEC (2011b). Information technology — Security techniques — Message Authentication Codes (MACs) — Part 2: Mechanisms using a dedicated hash-function. *International Organisation for Standardisation, Geneva, Switzerland*, 9797-2:39.

[87] ISO/IEC (2011c). Information technology — Security techniques — Message Authentication Codes (MACs) — Part 3: Mechanisms using a universal hash-function. *International Organisation for Standardisation, Geneva, Switzerland*, 9797-3:25.

[88] ISO/IEC (2012). Information technology — Security techniques — Lightweight Cryptography. *International Organisation for Standardisation, Geneva, Switzerland*, 29192-1-8:13.

[89] Jones, N. (2001). Introduction to the volatile keyword. [online] https://barrgroup.com/embedded-systems/how-to/c-volatile-keyword. Last accessed on Dec 19, 2020.

[90] Jones, R., Hosking, A., and Moss, E. (2016). *The garbage collection handbook: the art of automatic memory management*, pages [11–12]. CRC Press.

[91] Kerrisk, M. (2020a). Perf reference manual. [Online] https://man7.org/linux/man-pages/man1/perf.1.html. Last accessed 3 May, 2021.

[92] Kerrisk, M. (2020b). Perf stat reference manual. [Online] https://man7.org/linux/man-pages/man1/perf-stat.1.html. Last accessed 3 May, 2021.

[93] Kerrisk, M. (2021a). mallinfo2 - obtain memory allocation information. [Online] https://man7.org/linux/man-pages/man3/mallinfo.3.html. Last accessed 12 Jun, 2021.

[94] Kerrisk, M. (2021b). size - list section sizes and total size of binary files. [Online] https://man7.org/linux/man-pages/man1/size.1.html. Last accessed 4 Jun, 2021.

[95] Kerrisk, M. (2021c). elf - format of Executable and Linking Format (ELF) files. [Online] https://man7.org/linux/man-pages/man5/elf.5.html. Last accessed 4 Jun, 2021.

[96] Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K., and Ou, Z. (2018). Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26.

[97] King, C. (2017). powerstat. [Online] http://manpages.ubuntu.com/manpages/hirsute/man8/powerstat.8.html. Last accessed 1 May, 2021.

[98] Kivilinna, J. (2013). *Block ciphers: Fast implementations on x86-64 architecture*. PhD thesis, University of Oulu.

[99] Klein, A. (2013). *Stream ciphers*, page [11]. Springer.

[100] Knudsen, L. R. and Robshaw, M. (2011). *The block cipher companion*, pages [6, 82]. Springer Science & Business Media.

[101] Koch, W. (2021). LIBGCRYPT. [Online] http://gnupg.org/software/libgcrypt. Last accessed on Apr 3, 2021.

[102] Kokosa, K. (2018). *Pro .NET Memory Management For Better Code, Performance, and Scalability*, pages [12–26]. Springer.

[103] Koval, V. (2020). Analyze your firmware footprint with PlatformIO. [Online] https://piolabs.com/blog/insights/memory-analysis-part-1.html?utm_source=platformio&utm_medium=piohome. Last accessed 1 Jun, 2021.

[104] Kwon, D., Kim, J., Park, S., Sung, S. H., Sohn, Y., Song, J. H., Yeom, Y., Yoon, E.-J., Lee, S., Lee, J., et al. (2003). New block cipher: Aria. In *International Conference on Information Security and Cryptology*, pages 432–445. Springer.

[105] Lamprecht, C., van Moorsel, A., Tomlinson, P., and Thomas, N. (2006). Investigating the efficiency of cryptographic algorithms in online transactions. *International Journal of Simulation: Systems, Science & Technology*, 7(2):63–75.

[106] Leander, G., Paar, C., Poschmann, A., and Schramm, K. (2007). New lightweight des variants. In *International Workshop on Fast Software Encryption*, pages 196–210. Springer.

[107] Levesque, J. and Wagenbreth, G. (2010). *High performance computing: programming and applications*, page 37. CRC Press.

[108] linrunner (2021a). TLP Optimize Linux Laptop Battery Life. [Online] https://linrunner.de/tlp/. Last accessed 4 May, 2021.

[109] linrunner (2021b). tlpstat. [Online] https://linrunner.de/tlp/usage/tlp-stat.html. Last accessed 4 May, 2021.

[110] Lippman, S., Lajoie, J., and Moo, B. (2019). *C++ PRIMER.*, page [856]. Addison-Wesley.

[111] Liskov, M., Rivest, R. L., and Wagner, D. (2002). Tweakable block ciphers. In *Annual International Cryptology Conference*, pages 31–46. Springer.

[112] Lloyd, J. (2020). Botan Block Ciphers. [Online] https://botan.randombit.net/handbook/api_ref/block_cipher.html?highlight=block%20cipher. Last accessed on Mar 27, 2021.

[113] Lloyd, J. (2021). Botan: Crypto and TLS for Modern C++. [Online] http://botan.randombit.net. Last accessed on Apr 3, 2021.

[114] Lu, J. and Kim, J. (2008). Attacking 44 rounds of the shacal-2 block cipher using related-key rectangle cryptanalysis. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 91(9):2588–2596.

[115] Malbrain, K. (2009). Byte-oriented-AES: A public domain byte-oriented implementation of AES in C. https://code.google.com/archive/p/byte-oriented-aes/. Last accessed on Apr 3, 2021.

[116] Matsui, M. (2006). How far can we go on the x64 processors? In *International Workshop on Fast Software Encryption*, pages 341–358. Springer.

[117] Matsui, M. and Nakajima, J. (2007). On the power of bitslice implementation on intel core2 processor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 121–134. Springer.

[118] McKay, K., Bassham, L., Sönmez Turan, M., and Mouha, N. (2016). Report on lightweight cryptography. Technical report, National Institute of Standards and Technology.

[119] Microsoft Editors (2021). Microsoft .NET Framework Cryptography Model. [Online] https://docs.microsoft.com/en-us/dotnet/standard/security/cryptography-model. Last accessed on Mar 20, 2021.

[120] Minaam, D. S. A., Abdual-Kader, H. M., and Hadhoud, M. M. (2010). Evaluating the effects of symmetric cryptography algorithms on power consumption for different data types. *IJ Network Security*, 11(2):78–87.

[121] Miranda, P., Siekkinen, M., and Waris, H. (2011). TLS and energy consumption on a mobile device: A measurement study. In *2011 IEEE Symposium on Computers and Communications (ISCC)*, pages 983–989. IEEE.

[122] Montenegro, J. A., Pinto, M., and Fuentes, L. (2017). An empirical study of the power consumption of cryptographic primitives in android. *arXiv preprint arXiv:1705.03558*.

[123] Murphy, S. (1998). An analysis of SAFER. *Journal of Cryptology*, 11(4):235–251.

[124] Möller, N. (2018). Nettle - a low-level cryptographic library. [Online] http://www.lysator.liu.se/~nisse/nettle. Last accessed on Apr 3, 2021.

[125] Nadeem, A. and Javed, M. Y. (2005). A performance comparison of data encryption algorithms. In *Information and communication technologies, 2005. ICICT 2005. First international conference on*, pages 84–89. IEEE.

[126] Naik, K. and Wei, D. S. (2001). Software implementation strategies for power-conscious systems. *Mobile Networks and Applications*, 6(3):291–305.

[127] Naito, Y., Matsui, M., Sugawara, T., and Suzuki, D. (2018). SAEB: A lightweight blockcipher-based AEAD mode of operation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 192–217.

[128] Naito1, Y., Matsui1, M., Sakai1, Y., Suzuki1, D., Sakiyama, K., and Sugawara, T. (2019). SAEAES. *The Lightweight Cryptography NIST Competition*.

[129] Nikolaidis, S., Kavvadias, N., Laopoulos, T., Bisdounis, L., and Blionas, S. (2005). Instruction level energy modeling for pipelined processors. *Journal of Embedded Computing*, 1(3):317–324.

[130] NIST Editors (2021). AES development. [Online] https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development. Last accessed on Mar 26, 2021.

[131] NIST LwC Standardisation Team (2018). Submission requirements and evaluation criteria for the lightweight cryptography standardization process. [Online] https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf. Last accessed 15 Jun, 2021.

[132] Oaks, S. (2014). *Java Performance: The Definitive Guide: Getting the Most Out of Your Code*, pages [11, 12]. " O'Reilly Media, Inc.".

[133] OpenSSL Editors (2018). OpenSSL: Cryptography and SSL/TLS Toolkit. [Online] https://www.openssl.org. Last accessed on Apr 3, 2021.

[134] Oracle Corporation (2019). Benchmarking the Java HotSpot VM. [online] https://www.oracle.com/java/technologies/hotspotfaq.html.

[135] Paar, C. and Pelzl, J. (2010). *Understanding cryptography: a textbook for students and practitioners*, pages [30, 57, 154, 155, 294, 296]. Springer Science & Business Media.

[136] PlatformIO Editors (2021). PlatformIO IDE, A new generation toolset for embedded C/C++ development. [Online] http://platformio.org/platformio-ide. Last accessed 4 Jun, 2021.

[137] Popov, A. (2015). Prohibiting RC4 cipher suites. *Computer Science*, 2355(152-164):25.

[138] Poschmann, A. (2009). Lightweight cryptography - cryptographic engineering for a pervasive world. *IACR Cryptology ePrint Archive*, 2009:516.

[139] Rebeiro, C., Selvakumar, D., and Devi, A. (2006). Bitslice implementation of AES. In *International Conference on Cryptology and Network Security*, pages 203–212. Springer.

[140] Red Hat Editors (2010). Managing power consumption with PowerTOP. [Online] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/managing-power-consumption-with-powertop_monitoring-and-managing-system-status-and-performance. Last accessed 2 May, 2021.

[141] Rivest, R. L. (1994). The RC5 encryption algorithm. In *International Workshop on Fast Software Encryption*, pages 86–96. Springer.

[142] Rivest, R. L., Robshaw, M. J., Sidney, R., and Yin, Y. L. (1998). The RC6 block cipher. In *First Advanced Encryption Standard (AES) Conference*. Citeseer.

[143] Rob Lewis (2015). The Great "Power vs. Energy" Confusion. [Online] https://cleantechnica.com/2015/02/02/power-vs-energy-explanation/. Last accessed 26 Apr, 2021.

[144] Rotem, E., Naveh, A., Ananthakrishnan, A., Weissmann, E., and Rajwan, D. (2012). Power-management architecture of the intel microarchitecture code-named sandy bridge. *Ieee micro*, 32(2):20–27.

[145] Rott, J. (2010). Intel advanced encryption standard instructions (aes-ni). *Technical Report, Technical Report, Intel.*

[146] Sabri, C., Kriaa, L., and Azzouz, S. L. (2017). Comparison of IoT constrained devices operating systems: A survey. In *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pages 369–375. IEEE.

[147] Schneier, B. (1993). Description of a new variable-length key, 64-bit block cipher (Blowfish). In *International Workshop on Fast Software Encryption*, pages 191–204. Springer.

[148] Schneier, B. (2015). *Applied cryptography: protocols, algorithms, and source code in C*, pages [30, 198, 237, 347, 349, 461]. John Wiley & sons.

[149] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., and Ferguson, N. (1998). Twofish: A 128-bit block cipher. *NIST AES Proposal*, 15.

[150] Seo, H., Jeong, I., Lee, J., and Kim, W.-H. (2018). Compact implementations of arx-based block ciphers on iot processors. 17(3):1–16.

[151] Shannon, C. E. (1949). Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715.

[152] Team, A. D. (2016). Memory Areas. [Online] https://www.nongnu.org/avr-libc/user-manual/malloc.html. Last accessed 12 Jun, 2021.

[153] Ternan, M. (2007). AVRGCC: Monitoring Stack Usage. [Online] https://www.avrfreaks.net/forum/soft-c-avrgcc-monitoring-stack-usage?page=all. Last accessed 12 Jun, 2021.

[154] Tiwari, V., Malik, S., and Wolfe, A. (1994). Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445.

[155] Townsend, P. D. (1998). Experimental investigation of the performance limits for first telecommunications-window quantum cryptography systems. *IEEE Photonics Technology Letters*, 10(7):1048–1050.

[156] Turan, M. S., McKay, K. A., Çalik, Ç., Chang, D., Bassham, L., et al. (2019). Status report on the first round of the nist lightweight cryptography standardization process. *National Institute of Standards and Technology, Gaithersburg, MD, NIST Interagency/Internal Rep.(NISTIR)*.

[157] Ubuntu Editors (2019a). nm - list symbols from object files. [Online] http://manpages.ubuntu.com/manpages/impish/en/man1/nm.1.html. Last accessed 1 Jun, 2021.

[158] Ubuntu Editors (2019b). objdump - display information from object files. [Online] http://manpages.ubuntu.com/manpages/impish/en/man1/objdump.1.html. Last accessed 4 Jun, 2021.

[159] Viega, J. and Messier, M. (2003). *Secure programming cookbook for C and C++*, page 150. O'Reilly Media, Inc.

[160] Vieira, M., Madeira, H., Sachs, K., and Kounev, S. (2012). Resilience benchmarking. In *Resilience Assessment and Evaluation of Computing Systems*, pages 283–301. Springer.

[161] Villanueva, J. C. (2015). Stream Ciphers vs. Block Ciphers. [Online] https://www.jscape.com/blog/stream-cipher-vs-block-cipher. Last accessed on Mar 27, 2021.

[162] Wang, G. (2007). Related-key rectangle attack on 43-round SHACAL-2. In *International Conference on Information Security Practice and Experience*, pages 33–42. Springer.

[163] Weaver, V. M., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., and Moore, S. (2012). Measuring energy and power with papi. In *2012 41st international conference on parallel processing workshops*, pages 262–268. IEEE.

[164] Wheeler, D. J. and Needham, R. M. (1994). Tea, a tiny encryption algorithm. In *International workshop on fast software encryption*, pages 363–366. Springer.

[165] Wikipedia contributors (2018). Comparison of cryptography libraries. [Online] https://en.wikipedia.org/w/index.php?title=Comparison_of_cryptography_libraries&oldid=864698701. Last accessed on 2 Sep, 2018.

[166] Wolrich, G., Dixon, M., Locktyukhin, M., and Perminov, M. (2010). Fast Cryptographic Computation on Intel ® Architecture Processors Via Function Stitching. Technical report, Intel.

[167] Wolter, K. and Reinecke, P. (2010). Performance and security tradeoff. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 135–167. Springer.

[168] Wu, H. and Huang, T. (2014). Jambu lightweight authenticated encryption mode and aes-jambu. *CAESAR competition proposal.*

[169] Wu, H. and Huang, T. (2019). Tinyjambu: A family of lightweight authenticated encryption algorithms. *The Lightweight Cryptography NIST Competition.*

[170] Xie, J. and Pan, X. (2010). An improved RC4 stream cipher. In *2010 international conference on computer application and system modeling (ICCASM 2010)*, volume 7, pages V7–156. IEEE.

[171] Zhao, L., Iyer, R., Makineni, S., and Bhuyan, L. (2005). Anatomy and performance of ssl processing. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 197–206. IEEE.

[172] Zhao, Y. and Thomas, N. (2010). Efficient solutions of a pepa model of a key distribution centre. *Performance Evaluation*, 67(8):740–756.