PREVENTING CRYPTOGRAPHIC ATTACKS USED ON THE INTERNET OF THINGS

Mr. Khumbelo Difference Muthavhine

A thesis submitted for the degree of Doctor of Engineering in Electrical Engineering

University of South Africa, Department: Electrical Engineering

May 2022

DECLARATION

I, Mr. Khumbelo Difference Muthavhine, declare that the work I am submitting for assessment or evaluation contains no sections that have been copied in whole or in part from any other source unless explicitly specified or identified in quotation marks and with detailed, complete, and accurate referencing.

Khumbelo Difference Muthavhine	Date

Papers Published From this Thesis

Journal Papers

i. K. D. Muthavhine and M. Sumbwanyambe, "Securing IoT Devices against Differential-Linear (DL) Attack used on Serpent algorithm", MDPI, Future Internet, pp. 1-34, 2022.

ii. K. D. Muthavhine and M. Sumbwanyambe, "Preventing Differential Cryptanalysis Attacks Using a KDM Function and the 32-Bits Output S-Boxes on AES Algorithm Found on Internet of Things devices", MDPI, Cryptography, pp. 1-33, 2022.

Conference Papers

i. K. D. Muthavhine and M. Sumbwanyambe, "Reconstruction of DES in Order to Reduce Memory Constraints Found on IoT Devices," 2021 International Conference on Artificial Intelligence, Big Data, Computing and Data Communication Systems (icABCD), pp. 1-7, 2021.

ii. K. D. Muthavhine and M. Sumbwanyambe, "Modifying Cast algorithm in order to Increase Encryption Strength and to Reduce Memory Limitations," 2021 International Conference on Artificial Intelligence, Big

Data, Computing and Data Communication Systems (icABCD), pp. 1-7, 2021.

iii. K. D. Muthavhine and M. Sumbwanyambe, "Conversion of Clefia Algorithm to Decrease Memory Restrictions Encountered on IoT by Applying CMA Method," 2021 International Conference on Artificial Intelligence, Big Data, Computing and Data Communication Systems (icABCD), pp. 1-7, 2022.

Abstract

Cryptographic attacks on Internet of Things (IoT) devices are not highly considered by the users of IoT. Most cryptographic algorithms commonly used on IoT devices are vulnerable to cryptographic attacks. Cryptography attacks refer to mathematical procedures to crack the secret key of the algorithm used on IoT devices. More needs to be done to prevent attacks on cryptographic algorithms used on IoT devices. The objectives of this study are: (i)To use the Khumbelo Difference Muthavhine (KDM) function to prevent Differential Cryptanalysis (DC) attacks in the AES algorithm used on IoT devices. (ii) To apply the Blocker function to prevent Differential-Linear Cryptanalysis (DL) attacks in the Serpent algorithm used on IoT devices. (iii) To use the Khumbelo function to prevent Linear Cryptanalysis (LC), DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in the Camellia algorithm used on IoT devices. (iv) Applying the Khumbelo function to protect IoT against LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks. The KDM, Khumbelo, and Blocker functions prevented cryptographic attacks since all 8 x 8 S-Boxes were changed to 8 x 32 S-Box depending on the particular chapter. The analysis produced remarkable results in preventing cryptographic attacks from IoT devices using the KDM, Khumbelo, and Blocker functions. The objectives of the study was to block the construction of distiguishers. Distinguishers are used by intruders as first step to conduct any cryptographic attack. Once the construction

of distiguishers are blocked, therefore no attacks would be established. The study managed to block construction of distiguishers to 0% probability compared to (i)50% of LC attacks, (ii) 50% of DL attacks, and (iii) 50% of DC attacks. The 8 x 32 S-Box was expected to build distinguishers from 2^8 x 2^{32} = 256 x $4,294,967,296$ matrices with $1,099,511,627,776$ elements. Due to computational space, an ordinary computer could not compute 256 x $4,294,967,296$ matrices.

Acknowledgments

First and foremost, I would like to express my gratitude to Prof. Sumbwanyambe Mbuyu of the Department of Electrical Engineering at the University of South Africa (UNISA). The professor's office was always open for consultation when I had problems or questions about my experiments, results, or research. Finally, I would like to convey my thanks to my wife, Mrs. Tintswalo Audrey Mabasa Muthavhine, and my children (Frankie Muthavhine, Lulama Muthavhine, Mukoni Muthavhine, and Khuliso Muthavhine), and my father (Mr. Daniel Madume Muthavhine) for providing me with the necessary support and motivation throughout my research and studies. This achievement would not have been possible without their assistance. Thank you so much to you all. Mr. Khumbelo Difference Muthavhine

Contents

List of Figures

List of Tables

Abbreviations and Glossary

AE Avalanche effect

AES Advanced Encryption Standard

Bit_Perm Bit Permutation Function

DASH7 Alliance Protocol (D7A) which is an open-source wireless sensor and actuator network protocol

DDT Differential Distribution Table

DES Data Encryption Standard Algorithm

DH Diffie-Hellman

DL Differential-Linear Cryptanalysis Attack

DLCT Differential-Linear Connectivity Table

DSA Digital Signature Algorithm

ECC Elliptic Curve Cryptographic Algorithm

ECDH Elliptic-curve Diffie–Hellman

ECDSA Elliptic Curve Digital Signature Algorithm

FEAL Fast Data Encipherment Algorithm

F-function Feistel function

HMAC Hash Message Authentication Code

IDEA International Data Encryption Algorithm

KDM Khumbelo Difference Muthavhine's Function

LAT Linear Approximation Table

LC Linear Cryptanalysis Attack

M_AES New Modified AES

M_Cast New modified Cast

Mag_Serpent Magic Serpent

MD5 Message Digest Algorithm 5

MDCs Maximum Differential Characteristics

MMB Modular Multiplication-Based Block Cipher

mod modular operation

Mod_Blowfish New Modified Blowfish

MPC Modification Process of Cast

P-array Permutation array

RC4 Rivest Cipher 4 algorithm also known as ARC4 or ARCFOUR

RC5 or RC-5 Rivest Cipher 5 algorithm

RC6 Rivest Cipher 6 algorithm

Rec_Algorithms Rectified Algorithms

Rec_DES Rectified DES Algorithm

RSA Rivest Shimar Aglemen Algorithm

SAC Strict Avalanche Criterion

S-Box Substitution Box

SHA-2 Secure Hash Algorithm 2

SHA-3 Secure Hash Algorithm 3

SPA Simple Power Analysis attack

TEA Tiny Encryption Algorithm

TPS Target Partial Subkeys

List of Basic Mathematical Operators and Symbols

▲	Different percentage memory
$\Xi_{\wp}$	Plaintext Difference of XOR
$\oplus$	XOT operator
$\triangle$	Difference of XOR
$+$	Addition modular ten
$<<$	Shifting of bits
$<<<$	Rotation of bits
ζ	Ciphertext
$F - function$	Fiestel function
for	for loop of C++ program
$GF(2^{32})$	Galois Field of order 2^{32}
$GF(p^q)$	Galois Field of order p^q
$InPer$	Initial Permutation
mod	Modulo oparetor
$\rho\tau$	High enough probability
$S(X)$	S-Box operating in an input integer X
$SB_i(x)$	S-Box number i operating in an input integer x

Chapter 1

Introduction

1.1 Background of Study

Security on the Internet of Things (IoT) devices is not highly considered by the users during the design and implementation of cryptographic algorithms [1]. Most cryptographic algorithms commonly used on IoT are vulnerable to attacks such as Linear Cryptanalysis (LC), Differential Cryptanalysis (DC), Differential-Linear cryptanalysis (DL), boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher [2]. Most well-known cryptographic algorithms are not resistant to these attacks mentioned above [3]. Little has been done to prevent attacks on IoT devices [1]- [2]. This study proposes a novel approach that incorporates the following:

i. The Khumbelo Difference Muthavhine (KDM) function prevents DC attacks using the Advanced Encryption Standard (AES) algorithm on IoT devices.

ii. The Blocker function prevents DL attacks using the Serpent algorithm on IoT devices.

iii. The Khumbelo function prevents LC, DC, DL, boomerang, truncated

differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in the Camellia algorithm IoT devices.

1.1.1 Exploration of IoT

IoT is now ubiquitous computing that connects people and devices (objects or things) to enable newly established networks [4] - [6]. IoT is a new paradigm that connects numerous devices depending on the purpose to be achieved by the established network [7], [8]. Many objects/things, such as sensors, cameras, and others, are interconnected to the internet, and to each other [9] - [12]. These objects/things communicate through the internet, bringing a new level of involvement to users and leading to different definitions of IoT [7], [13].

1.1.2 Devices (Things, Objects, Machines)

IoT devices appear in a diversity of configurations and capacities. The most commonly used devices are sensors, smart cards and microcontroller-platforms [1]- [3]. These are several devices that can be classified as the pave of IoT [1]- [3].

1.1.2.1 Sensors

IoT employs a wide range of sensors to analyze vast amounts of data or information [7], [15], [16]. The sensors could be used to measure speed, temperature, energy, animal behavior (including human behavior), weather, water content, environmental conditions, changes in individual performance, and many other things [7], [15], [16]. IoT sensors need several aspects, including low cost, acceptable accuracy, acceptable speed, acceptable selectivity, repeatability, and excellent resolution. The accuracy of Global Positioning System (GPS) sensors, for example, are driving the growth of IoT, such as in driverless cars [17].

1.1.2.2 Smart Cards

The smart card technology is expected to be used in various applications, particularly on IoT devices, to enhance security features such as access control, the ability to perform multiple tasks, and the management security upgrade [18]. The two types of smart cards are contact and contactless smart cards [19]. A contact smart card is a device that is physically connected to the card reader and has a serial connection to a chip on the card's surface [20]- [24]. Data and card status information are communicated via these physical contact points [20], [24].

A contactless smart card is a device that communicates with a reader via radio frequencies (RF) antennas over contactless links [20]- [24]. Data and information are communicated using an electromagnetic signal from the chip and RF antennas [20]- [24].

Smart cards are required for various functions, depending on the user. These functions include security identification, health applications, payment applications, telecommunication, and others. Identity cards, passports, and driver's licenses are now converted to smart cards to perform multiple functions [19]- [22]. For IoT authentication, all information describing a person is embedded on a smart card [21] - [24]. Other examples include medical records, which are now kept on smart cards rather than hard copy files [24]. Also, in the banking sectors, payment transactions are handled with contact and contactless credit and debit cards [20]– [24]. Subscriber Identity Modules (SIM) cards are used by cell phones to identify a contact number, make calls, and store contacts [20]– [24].

Smart cards use RFID chip. RFID has been identified as an important device for enabling IoT functionality [25]. RFID is used in various techniques to accomplish various tasks such as identification, recording, and controlling other objects used on IoT [26]. RFID devices are wireless microchips that are divided into two parts: tags (transmitters/responders) and readers (transmitters/receivers) [26], [27]. RFID identifies different tags in a specific area

without needing a line of sight or human intervention [28]. Data and information stored on RFID tags can be renewed and updated in real-time [29]. RFID tags have recently been upgraded to serve as barcode substitutes for wireless reading and updating applications [30].

1.1.2.3 Microcontroller-Platform

Microcontroller-platform hardware has reprogrammable flash memory, which is required for embedding security applications on IoT [31]. One example of microcontroller-platform is Mica2 hardware. Mica2 hardware assists other devices like sensors by preventing data, information, and signals from being reported as outputs during IoT communication [32]. Mica2 hardware can also be used as a base station of IoT [33], [34]. Mica2 hardware is used by many IoT researchers worldwide because it interfaces with sensors, communicates with programming boards, stores sensor data logs, and manages resources [35]. The code stored on the Mica2 hardware can be reused [31]– [37]. Most researchers prefer Mica2 hardware over other programmable hardware devices because of these reusable features [4], [35].

1.1.3 Relationship between IoT and Cryptography

AES, DES, Camellia, and Serpent algorithms are commonly used on IoT devices [7]. AES is used to secure IoT sensors and contactless smart cards [38], [39]. For example, the AES algorithm is used on neural networks as represented in [40]. The modified DES algorithm has been used on neural networks to avoid attacks [41]. The AES and DES algorithms are used on inference engines to secure information as represented in [42]. Serpent and Camellia algorithms are used on platforms such as Big Data Analytics companies [43]. DES and AES algorithms are used in Big Data Analytics found in the cloud [44].

More often than not, there is a powerful connection between IoT and cryptographic algorithms [1]- [2]. The most common cryptographic algorithms used on IoT devices are AES, DES, Camellia, and Serpent. Because

they are the most commonly used, the literature review will concentrate on them while ignoring the less commonly used algorithms. Table 1.1 lists all four algorithms and their applications on IoT.

1.1.3.1 The AES Algorithm

Rijndael created the AES cryptographic algorithm for electronic data security. AES was submitted to the National Institute of Standards and Technology (NIST) [47], [48]. The AES specification was published on the Federal Information Processing Standards (FIPS) in 1997 and was later adopted by NIST [49], [50]. AES was found to be a block cipher capable of encrypting/decrypting 128-bit blocks with keys of 128, 196, or 256 bits [51].

1.1.3.2 The Camellia Algorithm

In 2000, three companies, Telephone Corporation, Nippon Telegraph, and Mitsubishi Electric Corporation, developed the Camellia algorithm [52]. In 2000, ISO/IEC JTC 1/SC 27 examined it as a candidate algorithm for an international encryption standard [53]. Following analysis, the ISO/IEC adopted Camellia [54]. Camellia employs 128-bit block lengths and key lengths of 128, 192, and 256 bits [55], [56]. The Camellia encryption and decryption follow the same procedure, but the order of the sub-keys is reversed during the decryption process [57].

1.1.3.3 The DES Algorithm

DES is a symmetric-key algorithm used to encrypt electronic data [58]. In 1974, IBM and the US government collaborated to create DES [59]. The National Bureau of Standards (NBS) analyzed the algorithm after it was developed to see if DES could secure data and information used by the United States of America (USA) government [60]. Because of flaws discovered in the length of the key using brute-force attacks, NBS slightly modified DES in 1976. In 1977, the United States Federal Information Processing Standard (FIPS) published DES. DES employs a 64-bit block and a key of 56 bits [61].

Table 1.1: Algorithms and their applications on IoT

Name of the Algorithms	**Algorithm Deployment on IoT**
AES	Sensors and contactless smart cards use AES for encryption [7], [38], [39]. Chang *et al.* [40] showed that the AES algorithm is used on neural networks. Prajapat *et al.* [42] indicated that AES algorithms are used on IoT devices when deployed with Artificial Intelligence (AI). Rodríguez-Lera *et al.* [45] showed that AES algorithms are used on IoT devices when deployed on Robotics and, to some extent, humanoids.
Camellia	Sensors and contactless smart cards use Camellia algorithms for encryption [7], [39]. Aoki *et al.* [43] indicated that Big Data Analytics companies use Camellia algorithms. Aoki *et al.* [43] demonstrated that Camellia algorithms are used on IoT platforms such as Robotics and, to some extent, humanoids.
DES	Neural networks use DES to secure most of its devices [7], [38]. Alallayah *et al.* [41] modified the DES algorithm used on neural networks to avoid attacks. Prajapat *et al.* [42] indicated that DES algorithms are used on neural network devices when deployed on Artificial Intelligence. Rodríguez-Lera *et al.* [45] showed that 3DES algorithms are used on IoT devices deployed on Robotics and, to some extent, humanoids.
Serpent	Sensors also use Serpent for encryption [7], [46]. Aoki *et al.* [43] indicated that the Serpent algorithm is used by Big Data Analytics deployed with Robotics.

To complete encryption, DES requires 16 changes [58], [59]. The encryption process is identical to the reverse (decryption) process, but subkeys are used in reverse order [60], [61].

1.1.3.4 The Serpent Algorithm

The Serpent was created in 1998 by Lars Knudsen, Ross Anderson, and Eli Biham [62], [63]. They intended to submit the algorithm as a candidate for the DES [64]. The Serpent is derived from the DES algorithm; the S-Boxes Serpent is derived from DES' S-Boxes and has a new structure that gives the attacker diffusion [65]. The Serpent requires a block length of 128 bits and a key length of 128, 192, or 256 bits [62].

1.2 Problem Statement

Concerns have been presented about the security of IoT algorithms against cryptographic attacks. The source of concern is various attacks (such as LC, DC, DL cryptanalysis, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher) used by intruders on IoT cryptographic algorithms to discover certain keys [68]- [80]. An attacker can quickly use cryptographic algorithms with short output bit lengths of an S-Box to create probability tables on algorithms to guess the secret encryption key [[81], pp. 21]. The output sizes of most well-known algorithms are S-Boxes, which are less than 32 output bits. AES and Camellia, for example, have eight output bits [[48], pp.16], [[54], pp.18]. DES and Serpent have four output bits [[58], pp. 13-14], respectively [[64], pp. 3]. [[66], pp, 8]. If the output issues are not appropriately addressed, the problem of attacks can jeopardize the entire security of the IoT system. This study focuses on solving attacks (LC, DC, DL cryptanalysis, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher). The objectives were applied to prevent the cryptographic attacks:

i. The study uses the KDM function to prevent DC attack in the AES

algorithm used on IoT devices.

ii. The study uses the Blocker function to prevent DL attack in Serpent algorithm used on IoT devices.

iii. The study uses the Khumbelo function to prevent LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in Camellia algorithm used on IoT devices.

Table 1.2: Algorithm Attacks and Solutions

Name of the Algorithms	**Type of the Attacks**	**Novel Solution Used**
AES	DC attack. Refer to chapter three of this study.	The KDM function. Refer to chapter three of this study.
Camellia	LC, DC, DL cryptanalysis, and boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks. Refer to chapter five of this study.	The Khumbelo function. Refer to chapter five of this study.
Serpent	DL attack. Refer to chapter four of this study.	The Blocker function. Refer to chapter four of this study.

1.3 Research Questions

i. How to develop the KDM function that will prevent DC attacks in the AES algorithm used on IoT devices?

ii. How to develop the Blocker function that will prevent DL attacks in the Serpent algorithm used on IoT devices?

iii. How to develop the Khumbelo function that will prevent LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in the Camellia algorithm used on IoT devices?

1.4 Research Objective

i. To develop the KDM function that will prevent DC attacks in the AES algorithm used on IoT devices.

ii. To develop the Blocker function that will prevent DL attacks in the Serpent algorithm used on IoT devices.

iii. To develop the Khumbelo function that will prevent LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in the Camellia algorithm used on IoT devices.

1.5 Hypothesis

The research hypothesis, which will also serve as the statement of the research, reads:

H1 It is possible to develop the KDM function that will prevent DC attacks in the AES algorithm used on IoT devices.

H2 It is possible to develop the Blocker function that will prevent DL attack in the Serpent algorithm used on IoT devices.

H3 It is possible to develop the Khumbelo function that will prevent LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in Camellia algorithm used on IoT devices.

1.6 Research Methodology

In this study, the research methodology to be used will be an experimental simulation-type research method that will be conducted as follows:

i. Developing the KDM function that will prevent DC attack in the AES algorithm used on IoT devices. IoT devices encrypt data stored and transmitted during communication using an AES algorithm [82]- [95]. The AES algorithm is frequently subjected to DC attacks [77], [96]. There has been little progress in preventing DC attacks, particularly on an AES algorithm [77], [96]. The goal of this research is to prevent DC attacks. The novel approach of using a KDM function and replacing the 8 x 8 S-Boxes with the 8 x 32 S-Boxes prevents DC attacks on an AES algorithm. A KDM function is a newly developed mathematical function that was coined and used on purpose in this study. Except for this study, no researcher has ever created, defined, or used a KDM function. A KDM function contains many mathematical modulo operators. A KDM function creates a new 32-Bit S-Box suitable for the new Modified AES algorithm and confuses the attacker. These mathematical modulo operators are also irreversible. The study managed to prevent a minimum of 70% of DC attacks on AES and a maximum of 100% on a Simplified DES. Because no S-Box is used as a building block, the attack on the new Modified AES algorithm is 0%.

ii. Developing the Blocker function that will prevent DL attacks in the Serpent algorithm used on IoT devices. The DL characteristic's probability produces a Differential-Linear Connectivity Table (DLCT), which is necessary for DL attack [97], [98]. The DLCT is a probability table that offers many opportunities for an attacker to deduce the cryptographic keys for any algorithm, including Serpent, found on IoT devices [97]. To attack algorithms, an attacker first builds DLCT utilizing building blocks like Substitution-Boxes (S-Boxes), which are common in the

structures of many algorithms [98]. This study aims to protect IoT devices from DL attacks that target the Serpent algorithm using three magic numbers that are mapped onto a newly created mathematical function called Blocker. The Blocker is incorporated into Serpent's infrastructure before being placed on IoT devices. To replace the original Serpent S-Boxes with 4-Bits-output, new S-Boxes with 32-Bits-output were created for this study. The architecture of Serpent also included the innovative S-Boxes. The Blocker function and magic numbers, were influential in this investigation. The findings indicate that an algorithm with an S-Box made up of 4-Bits-output is more prone to attack than an algorithm with an S-Box made up of 32-Bits-output. Three magic numbers and 32-bit output S-Boxes were used in the study to create a novel blocking technique that prevented the development of DLCT and DL assaults. The innovative strategy successfully protected IoT devices' installed Serpent algorithms from DL attacks.

iii. Developing the Khumbelo function that will prevent LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in the Camellia algorithm used on IoT devices. Camellia is one of the cryptographic algorithms implemented on many Internet of Things (IoT) devices [99]. However, an intruder uses the Substitution Box (S-Box) distinguisher to attack her Camellia cipher [100]. A distinguisher is a table that provides the probability of guessing the algorithm's secret key [99]. Distinguisher's features are used in most attacks. The most well-known characteristic features are the Linear Approximation Table (LAT), Difference Distribution Table (DDT), and Differential Linear Connections Table (DLCT) [100]. This work focuses on preventing these attacks by deflecting the construction of an S-Box distinguisher with a new function called the Khumbelo. The Khumbelo function prevented distinguisher construction by lowering the construction probability. The Khumbelo function successfully re-

duced the attack probability of LAT (54.6875 percent to 0 percent), DDT (1.5625 percent to 0 percent), and DLCT (50.0000 percent to 0 percent). The Khumbelo function is generated using a 4-Byte output S-Box instead of Camellia's original 1-Byte output S-box. Also, the Khumbelo function consists of many modulo operators. New 4-Byte output S-boxes and modulo operators confuse and block intruders to build distinguishers. After successfully embedding the Khumbelo function in the traditional camellia, the newly modified camellia was coined K_Camellia.

To summarize the research methodology and the casual relationship between objectives and questions, refer to Table 1.3 and Figure 1.1. Refer to chapter three of this study for more information about the DC attack and KDM function. Refer to chapter four of this study for more information about the Blocker function DL attack. Refer to chapter five of this study for more information about the Khumbelo function LC, DC, DL cryptanalysis, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks.

1.7 Benefits

The practical implementations of the funding can be applied in the real world if one wants to prevent one of the following attacks using the following methods, respective to the particular algorithms:

i. The KDM Function has been found to be more effective in preventing DC attacks in AES.

ii. A Blocker Function has been found to be more effective in preventing LC attacks in the Serpent.

iii. A Khumbelo function has been found to be more effective in preventing LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in the Camellia.

Table 1.3: The casual relationship among research objectives research questions and research methodology

Research Questions	**Research Objective**	**Research Methodology**
How to use the KDM function to prevent DC attack in the AES algorithm used on IoT devices?	To use the KDM function to prevent DC attack in the AES algorithm used on IoT devices.	Using the KDM function to prevent DC attack in the AES algorithm used on IoT devices.
How to use the Blocker function to prevent DL attack in Serpent algorithm used on IoT devices?	To use the Blocker function to prevent DL attack in Serpent algorithm used on IoT devices.	Using the Blocker function to prevent DL attack in Serpent algorithm used on IoT devices.
How to use the Khumbelo function to prevent LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in Camellia algorithm used on IoT devices?	To use the Khumbelo function to prevent LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in Camellia algorithm used on IoT devices.	Using use the Khumbelo function to prevent LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in Camellia algorithm used on IoT devices.

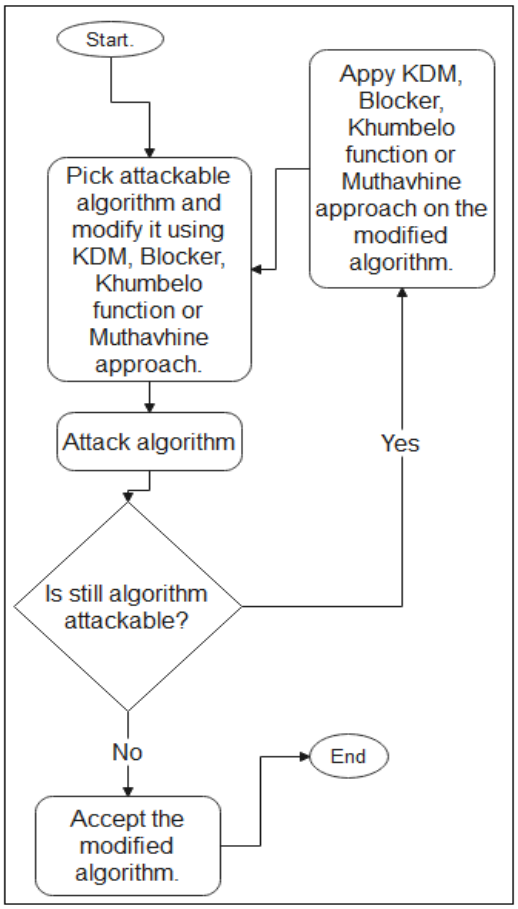


Figure 1.1: A flowchart diagram representing general research methodology

1.8 Limitations and Drawbacks

Apart from the methods used in this study, there are limitations and drawbacks that can expose IoT devices to other attacks. The few limitations and drawbacks are as follows:

i. A powerfully encrypted piece of data can be problematic to access, even for a lawful user, during a critical period of decision-making. IoT devices can be attacked and rendered inoperable by an intruder.

ii. The use of cryptography only cannot ensure the security of information. Other methods of attack prevention are required.

iii. The new algorithms proposed in this study do not protect against the threats and vulnerabilities arising from poor structure, technique, and design. These must be addressed through appropriate planning and establishing a protective IoT network before the algorithms are implemented.

iv. The new algorithms proposed in this study, like any other known algorithms, are not immune to brute-force attacks.

1.9 Significance of the Study

After the study, enhancing and probing the security of three cryptographic algorithms commonly on IoT will benefit the community, developers, and other researchers in securing their communication, data, and sensitive information stored on devices. Three cryptographic algorithms refer to (AES, Camellia, and Serpent). For layout of the study, refer to Figure 1.2. The proposed future work will give other researchers and developers of the cryptographic algorithm to focus on improving, installing, and implementing IoT security during their future studies and research. It is believed that the research investigation of the security issues surrounding the IoT will add yet another dimension to strengthening security like authentication,

authorization, confidentiality, non-repudiation, availability, and privacy of data used on IoT. Few attacks, such as the LC, DC, DL cryptanalysis, and boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks, will be prevented or made more difficult for the intruder to use and crack the four algorithms.

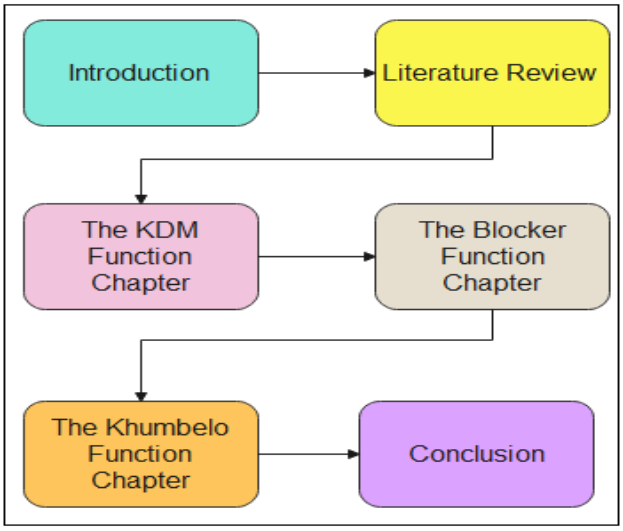


Figure 1.2: Flowchart diagrammatic representation of thesis chapters

1.10 Outline of Final Thesis

The following chapters will comprise the thesis: given in Figure 1.2:

CHAPTER 1. Introduction: This chapter will contain introductory background information, problem statements, research objectives, research questions, definitions, and limitations.

CHAPTER 2. Literarature Review: This chapter will include a grounded in a comprehensive literature review and theoretical basis for the study.

CHAPTER 3. The KDM Function : This chapter will present how KDM function prevented DC attack on AES algorithm.

CHAPTER 4. The Blocker Function: This chapter will present how Blocker function prevented DL attack on Serpent.

CHAPTER 5. The Khumbelo Function: This chapter will present how Khumbelo Function prevented the boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks on Camellia.

CHAPTER 6. Conclusion: This chapter will include a summary of the research findings, the significance of the contribution, and recommendations for future research.

Chapter 2

Literature Review

2.1 IoT Communications and Connections

IoT is a network of networks that connects devices, things, objects, and machines to the internet [105], [82]- [95]. The IoT media (means of connections) are network layers, wireless connectivity, protocols, device-to-device communication models, device-gateway communication models, back-end data sharing communication models, device-to-cloud communication models, and small connected devices [3]. In this section, a review of the previously mentioned attacks is analyzed to investigate the security of IoT communications and connections of devices.

2.1.1 Overviews of the IoT Communication Modes

When IoT is compared to the traditional internet, the traditional internet requires physical links [106]. IoT requires wireless connectivity to establish protocols, network layers, communication models (back-end data sharing communication, device-to-cloud communication, device-to-device communication, and device-gateway communication), and tiny devices interconnected to each other with vast storage, high speed, and adaptability to cloud computing [3].

2.1.1.1 IoT Protocols

For IoT devices to function correctly, several protocols inverted by the International Organization for Standardization (ISO) stack are used, as well as the implementation of robust cryptographic algorithms to drive the security of these protocols [107] - [108]. Many protocols are included in the IoT stack, including the Constrained Application Protocol (CoAP), which is used for messaging; the Infrastructure Protocol, which is used for networking; and the Identification Protocol, which is used to identify the user [109]. Other application protocols, such as the Message Queuing Telemetry Transport Protocol (MQTT) and Advanced Message Queue Protocol (AMQP), are managed by the application protocol; refer to Figure 2.1. The discovery protocol is intended to identify the web and neighboring devices. In addition, other protocols, such as the Data Protocols or the Representational State Transfer Protocol (REST), are used [111]. REST is intended for data management and control, such as the web socket [111]. Aside from the protocols mentioned above, many surplus protocols in Device Management Protocols (DMP) provide ways to control devices. The Semantic Protocol is intended to provide web services, while the Stomp Protocol is meant to manage text-based messaging [112]. Because of space constraints on IoT devices and energy supply constraints from batteries, these protocols are designed to save power, use less memory, and reduce computational time [107], [112]. As a result of these limitations or restrictions, dealing with the IoT security problem is difficult because most security applications require a large amount of space and time to run properly after installation [107] – [112]. Extensible Messaging and Presence Protocol (XMPP) is an accessible communication protocol for instant messaging (IM), existence information, and directory of contacts management [110]. Hypertext Transfer Protocol (HTTP) is a protocol that is used to retrieve resources such as Hypertext Markup Language (HTML) documents [107], [112]. HTTP is the foundation of all data exchange on the Web and is a client-server protocol, meaning the recipient

initiates requests, typically the Web browser [111].

2.1.1.2 IoT Architecture

IoT architecture is built using several stack layers [114]. Nonetheless, the security of these layers necessitates robust cryptographic algorithms [115]. On the one hand, IoT researchers construct different network layers for IoT architecture by subdividing some of the main layers into sublayers [116]. For example, a user supporting layer from the traditional internet has been subdivided into the application, presentation, and session layers on the IoT by [117]. Refer to Figure 2.2. On the one hand, other new layers are still being developed by [116]. In recent years, many investigators have become interested in IoT layers, particularly its architecture, consisting of three layers: the network, perception, and application [114] – [116].

The network layer manages the connection of IoT networks, such as wireless or wired networks, while the application layer manages all IoT applications [115]. The perception layer is intended to collect, bring in, and process data from the IoT communications [116].

2.1.1.3 Model of Machine-to-Machine Communication

The Machine-to-Machine (M2M) communication model refers to a group of two or more devices that are directly interconnected to establish communication [3]. This model typically does not require the implementation of a gateway, cloud computing, or servers [118]. These devices communicate via Bluetooth, SHAREit, Z-Wave, or ZigBee managed by data-link protocal [119]. Refer to Figure 2.1. The security and trust of device-to-device communication are dependent on the devices pairing PINs for authentication. The pairing PINs on both or all devices should be the exact [120]. No internet protocol is required on the device-device communication model [118]– [120].

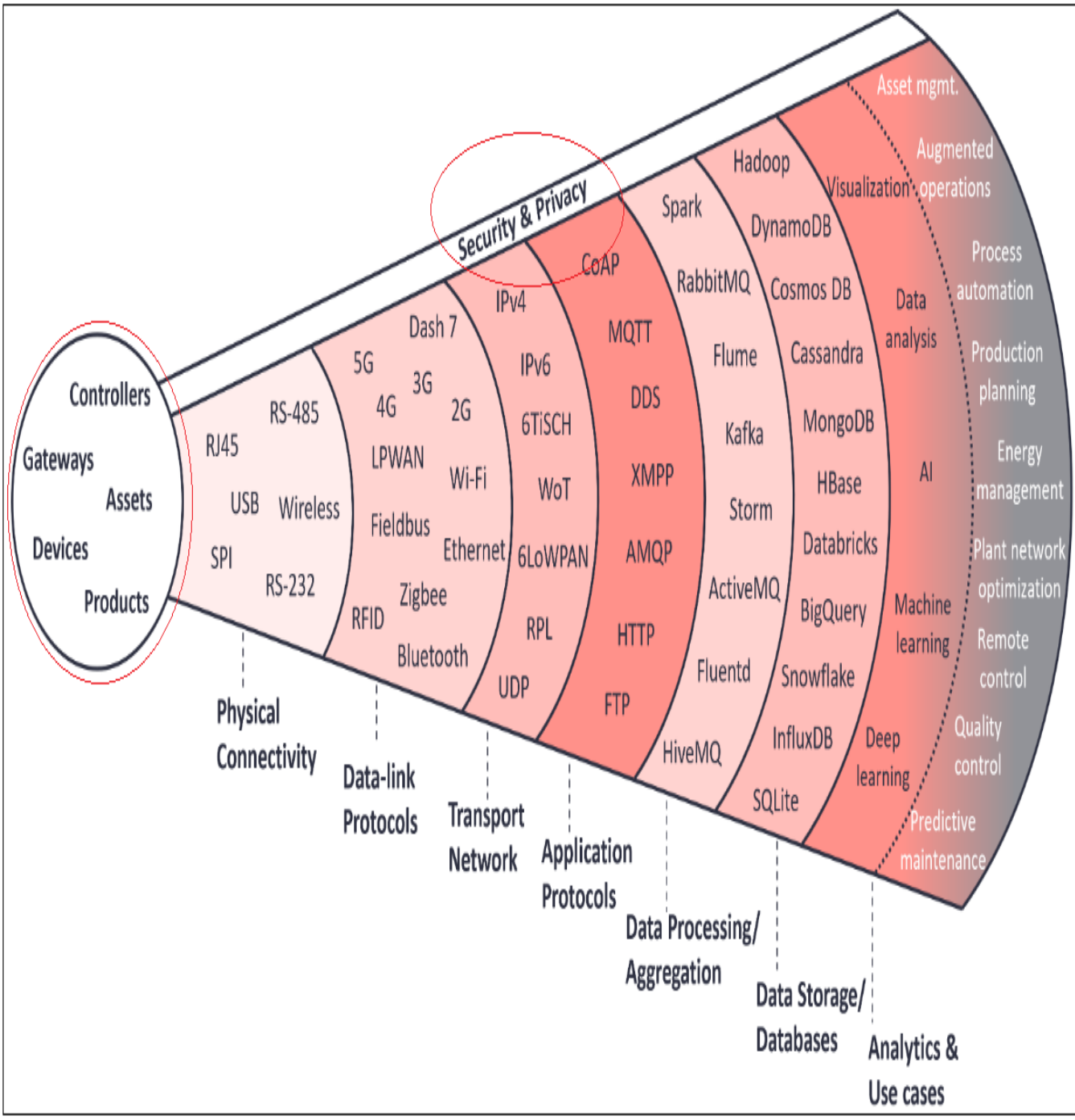


Figure 2.1: IoT Protocols with TCP/IP models [121]

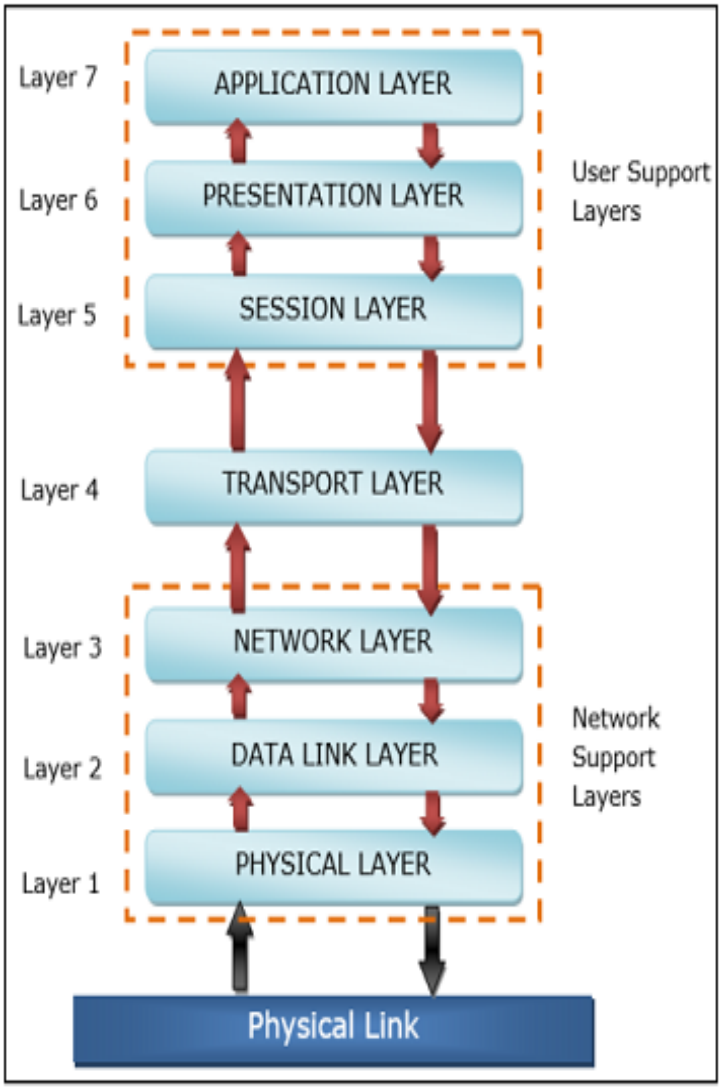


Figure 2.2: Subdivisions of layers derived from main layers [117]

2.1.1.4 Model of Device-to-Cloud Communication

In a device-to-cloud communication model, IoT devices are connected directly to an internet cloud service, such as an internet service provider (ISP), to establish communication [122]. This approach uses existing channels of communication mechanisms, such as Ethernet, mobile phone, service providers, or Wi-Fi connections, to establish contact and communication between devices via the internet network, which later connects to cloud computing [123]. The internet service provider (ISP) is responsible for the security and privacy [124]. Communication may be hampered if the service provider disappears or the hosting provider decides to stop [124].

2.1.1.5 Model of Device-to-Gateway Communication

Device-to-Gateway communication model is referred to as device-to-gateway communication when connected to IoT gateways as a bridge to connect cloud computing and services [3]. The IoT gateway's function bridges the communication gap between IoT devices, other equipment, systems, sensors, cloud computing, and services [118]. IoT gateways also organize internal processing, application requirements, and storage solutions [124]. Data security or confidentiality is determined by the website searched, visited, called, or browsed during communication [123]. If an attacker breaches or hacks information on a website, it can turn it into a dangerous platform. They could create or use a bogus webpage to hack or phish information from the user [124].

2.1.1.6 Communication Model for Back-End Data Sharing

The back-end data sharing communication model introduces a communication architecture that allows end-users to assign and evaluate data from a cloud computing service in conjunction with data from other devices and sources [118], [124]. The Internet Protocol (IP) is not required for back-end data-sharing communication [123]. The application service providers are responsible for the security, and trust [125]. Assume the application

service provider terminates the services or installs software on a platform with an open back door. In that case, all information and data from end users could be manipulated, read, exploited, and transferred to unauthorized recipients [119]. IoT devices use cryptographic algorithms to secure communication [3].

2.2 Cryptographic Algorithms Theory

The mathematical art of converting plaintext (readable message, communication, or data) to ciphertext (unreadable scrambled message, transmission, or data) and vice versa is known as cryptography [126]. Converting plaintext to ciphertext is known as encryption, and converting ciphertext back to plaintext is known as decryption [7], [127] . In layman's terms, cryptography is a required practice to protect commutations, information, and data from hackers, intruders, and attackers by making everything unreadable, inaudible, and inaccessible [3].

In most cases, this art of encryption and decryption is accomplished by using a mathematical procedure known as a cryptographic algorithm [2]. Historically, encryption was done with pen-and-paper methods based on letter substitutions and shifting numbers, such as in Vigenère and Steganography encryption [7]. Modern cryptographic algorithms employ digital cryptographic systems such as symmetric, asymmetric, and hash ciphers [127].

2.2.1 Symmetric Encryption System

For secure communication, the symmetric encryption system (algorithm) uses only one key to secure data from sender to receiver, and vice versa [7], [127]. Symmetric encryption is defined as uniform or symmetric by Kaur *et al.* [128] because it uses only one invariable cryptographic key for encryption and decryption. Examples include: the AES, DES, Camellia, Serpent [129]. They both use the same encryption key to secure communication. The symmetric encryption system is also known as a block cipher [128].

2.2.2 Asymmetric Encryption System

An asymmetric Encryption System (algorithm) is a type of encryption method in which the key used to decrypt some data or information is different from the key used to encrypt some data or information [128]. Asymmetric encryption is also known as public-key encryption [129]. Examples include the Elliptic Curve Cryptographic Algorithm (ECC), Rivest Shimar Aglemen (RSA), Diffie-Hellman (DH), and Digital Signature Algorithm (DSA), and other asymmetric encryption algorithms [7], [130].

2.2.3 Steganography Encryption System

A steganography encryption system displays data or information on other mediums or platforms discretely [130]. These mediums are typically visible objects to humans [7]. Photographs, videos, audio files, and paper documents are examples of such items [131]. The use of unnoticeable/invisible ink to write messages as hidden text on a picture or piece of paper that contains visible text, and the receiver will disregard the visible text in favor of the invisible one written by invisible ink using a candle or other mechanism, is a simple example of steganography [132]. Another example is when the video is played, in which the video images, conveyed by an authorized receiver, differs significantly from a video viewed by an ordinary, or unauthorized person [7]. The videos' trees, grasses, and pavements could convey messages, which could be read as plain text or Morse code (for example, dots and dashes, shorts and longs) [7].

2.2.4 The Vigenère Encryption System

The Vigenère Encryption System is a method of encryption in which each alphabetical letter is substituted or shifted a predetermined number of times [133]. A letter A would be shifted three times to become a letter E in a vigenère encryption of shift 3, a letter Y would become a letter C, a letter B would become a letter F, and so on [134]. Before communicating, the sender

and receiver must agree on how to handle shifting [133]. Vigenère encryption was used before the invention of computers [135]. Vigenère encryption is done by the CAESAR algorithm [7], [133]. Because of technological advancements, no one uses vigenère encryption or the CAESAR algorithm for top-secret messages [7], [133], [135].

2.2.5 Hashing (Authenticated) Encryption System

The Hashing (Authenticated) Encryption System is a method of encryption that generates a unique fixed-length signature for each communication, data, or information set [136]. Minor changes to that communication, data, or information would be easily be detected because each hash is unique to that communication, data, or information [7]. An unauthorized person cannot crack, reverse, or decipher a communication, data, or information that has been encrypted with Hashing (Authenticated) Encryption [126]. It is straightforward to determine whether hashed communication, data, or information received has been tempered with [7]. Hashing (Authenticated) encryption is used primarily to prevent, detect, or investigate whether an intruder has tampered with communication [136]. Hash encryption is demonstrated by Secure Hashing Algorithms (SHA-2 and SHA-3), Message Digest Algorithm 5 (MD5), Race Integrity Primitives Evaluation Message Digest (RIPEMD), and BLAKE2 algorithms [126], [136].

2.3 Related Work of DC attack on AES

Biham and Keller [137] first presented the DC attack on the AES-128 and reduced it to five rounds, in this work, the secret key was recovered before the final round. The DC attack was later refined by Cheon *et al.* [138] to find six rounds using 291.5 favorite plaintext pairs and time complexity of 2^{122} for AES-192 and AES-256, Raphael and Phan [139] achieved to attack both AES-192 and AES-256 reduced it to seven rounds, respectively [140]. in this work, the secret key was recovered before the final round. The DC

attack required 292 (AES-192) and 292.5 (AES-256) chosen-plaintext pairs with time intricacies of 2186 (AES-192) and 2250.5 (AES256) [140]. Overall, the better DC attack thus far succeeded in breaking through seven rounds of AES-192 and AES-256 [140].

Lacko-Barto˘sova [141] used the DC attack on two rounds of AES, with a overall complexity calculated on a three-round AES attack. The DC attack relied on detecting noticeable bitwise differences in the secret key. Data complexity of the DC attack was 227, where 8 bits of the private key were recovered, in this work, the secret key was recovered before the final round [141].

Jakimoski and Desmedt [142] used a related-key DC attack to AES's 192-bit secret key modification. Jakimoski and Desmedt [142] also indicated that on four rounds, at least 25 active bytes of the private key were used in the DC attack. In this work, the secret key was recovered before the final round. Jakimoski and Desmedt [142] demonstrated that a truncated DC could be used to improve the attack. Within the case, the number required of plaintext/ciphertext pairs might be 281, corresponding to a computational burden of 286. Utilizing an impossible related-key DC attack, Jakimoski and Desmedt [142] claimed to break a 7-round with computational complexity of 2116 and 2111 plaintext/ciphertext pairs. The attack on 8-round, required complexity of about 2183 encryptions and 288 plaintext/ciphertext pairs [142]. In this work, the secret key was recovered before the final round. In this work, the secret key was recovered before the final round.

Hu and He [143] utilized a new property of MixColumns Transformation and constructed a new 4-round possible DC attack path. Hu and He [143] added 1-round and 3-round of possible DC attack paths before and behind the approach. Additionally, Hu and He [143] constructed a new 7-round impossible DC attack path. Hu and He [143] used the approach to analyze 64-bit initial keys of 7-round AES-192, which required 271 pairs of selected plaintexts, approximately 272 memory cells, and around 2135 encryption and

decryption computations. Finally, recovering the secret keys [143]. In this work, the secret key was recovered before the final round.

Rouquette and Solnon [151] showed that based on the complete distribution ratio and complexity that occurred, Mini-AES algorithms had been vulnerable to DC attack [151]. The greatest DC attack characterization is the one that uses a single active S-Box and has a distribution proportion of 8/16. [151]. Rouquette and Solnon [151] used the probability of guessing the secret key was calculated using the distribution ratio of 8/16. In this work, the secret key was recovered before the final round.

2.4 Related Work of DL attacks on Serpent Algorithm

Anderson *et al.* [152] attacked the Serpent algorithm using the DL and Differential-Linear Connectivity Table (DLCT) table. Compton *et al.* [153] developed the Simple Power Analysis attack (SPA) to attack an 8-bits smart card encrypted by Serpent. The results showed that Serpent key generation was weaker to a side-channel attack because of a linear feedback shift register (LFSR). LFSRs are common in most cryptographic algorithms. Thus, suggestions were given that Serpent's LFSRs should be carefully modified to reduce attacks. Bar-On *et al.* [97] developed a new tool called DLCT to attack Serpent's secret keys. The tool was used to calculate the probability of the secret key. In this work, the probability of secret key was recovered before the final round. Canteaut *et al.* [98] analyzed the DLCT to get absolute indicators of Serpent's secret key weakness. According to the results proposed by [98], the DLCT approach method was found to be similar to the auto-correlation spectrum entities. Conclusion was drawn that DLCT was nothing else but Auto Correlation Table (ACT). Further on, Canteaut *et al.* indicated that the ACT spectrum was invariable under any equivalence similarities and was not invariant under changes. Biham *et al.* [155] attacked the Serpent algorithm using the DL attack with the aid of the DLCT tool.

In this work, the secret key was recovered before the final round [155].

2.4.1 Related Work of Attacks on the Camellia cipher

The Camellia cipher can be attacked using the boomerang, according to Yap, Khoo, and A. Poschmann [156]. Yap, Khoo, and A. Poschmann [156] managed to attack Camellia using boomerang, the boomering attack revealed the secret key before round four. Lee, Hong, Lee, Lim, and Yoon [157] introduced a truncated differential cryptanalysis attack of adjusted Camellia reduced to 7-round and 8-round. They [157] recognized the 8-Bit key on 7-round and the 16-Bit key on 8-round, with 3*281 and 3*282 plaintext, respectively. The other building block of the boomerang attack is the truncated differential cryptanalysis attack [158], [159].

Bai and Li [160] successfully attacked the 11 rounds of Camellia-128, 11 rounds of Camellia-192, 12 rounds of Camellia-192, and 14 rounds of Camellia-256 using impossible differential cryptanalysis attacks and with the time complexities of $2^{123.6}$, $2^{121.7}$, $2^{171.4}$ and $2^{238.2}$, respectively.

Wu, Zhang, and Feng [161] used the relationship between the subkeys and the number of Camellia rounds, combined with several novel approaches in the secret key retrieval technique, to improve the impossible differential attack up to 12 rounds of Camellia-128 and 16 rounds of Camellia-256. They [161] were successful in attacking 12-round and 16-round plaintexts of 2^{65} and 2^{89}, respectively.

Lu, Wei, Pasalic, and Fouque [162] characterized the infrequent 5-round and 6-round impacts of Camellia and eventually used them to attack 10, 11, and 10 rounds of Camellia using meet-in-the-middle attacks to discover the 128-Bit key, 192-Bit key, and 256-Bit key, respectively.

Using the zero-correlation linear cryptanalysis attack, Liu, Sun, Wang, Varici, and Gu [163] explored Camellia's security. As a result of the analysis, particular weak keys were found. According to Liu, Sun, Wang, Varici, and Gu [163] proposed some unique properties of the $FL/(FL)^{-1}$ functions in

Camellia.

Liu, Sun, Wang, Varici, and Gu [163] constructed the first known eight rounds of zero-correlation linear distinguisher of Camellia with $FL/(FL)^{-1}$ layers for the described weak keys because it covered the same number of rounds as the best-known zero-correlation linear distinguisher for Camellia without $FL/(FL)^{-1}$ layers. According Liu, Sun, Wang, Varici, and Gu [163] claimed that $FL/(FL)^{-1}$ layers could not virtually prevent zero-correlation linear cryptanalysis attacks for specific weak keys.

2.4.2 Chapter Summary

This chapter discussed IoT communication, cryptographic algorithms, confirmation of difference attacks, symmetric, asymmetric, steganography, the Vigenère Encryption System, and hash fuction. The aim was to give a literature review of the research study.

Chapter 3 discusses the design and development of KDM function to prevent DC attacks on AES. Chapter 4 discusses the development of Blocker function to prevent DL attacks on the Serpent algorithm for IoT devices. Chapter 5 discusses the design and development of Khumbelo function on the Camellia algorithm to prevent attacks on IoT devices.

Chapter 3

The Design and Development of KDM Function to Prevent DC attacks on AES

The chapter that follows is based on published work by:

i. K. D. Muthavhine and M. Sumbwanyambe, "Preventing Differential Cryptanalysis Attacks Using a KDM Function and the 32-Bits Output S-Boxes on AES Algorithm Found on Internet of Things devices", MDPI, Cryptography, pp. 1-33, 2022.

   Website: *https://www.mdpi.com/2410-387X/6/1/11*

   Status: Published.

   Publisher: MDPI.

ii. K. D. Muthavhine and M. Sumbwanyambe, "Reconstruction of DES in Order to Reduce Memory Constraints Found on IoT Devices," IEEE, pp. 1-7, 2021.

   Website: *https://ieeexplore.ieee.org/document/9519312*

   Status: Published.

Publisher: IEEE.

Abstract: IoT devices encrypt data stored and transmitted during communication using an AES algorithm [82]- [95]. The AES algorithm is frequently subjected to DC attacks [77], [96]. There has been little progress in preventing DC attacks, particularly on an AES algorithm [77], [96]. The goal of this research is to prevent DC attacks. The novel approach of using a KDM function and replacing the 8 x 8 S-Boxes with the 8 x 32 S-Boxes prevents DC attacks on an AES algorithm. A KDM function is a newly developed mathematical function that was coined and used on purpose in this study. Except for this study, no researcher has ever created, defined, or used a KDM function. A KDM function contains many mathematical modulo operators. A KDM function creates a new 32-Bit S-Box suitable for the new Modified AES algorithm and confuses the attacker. These mathematical modulo operators are also irreversible. The study managed to prevent a minimum of 70% of DC attacks on AES and a maximum of 100% on a Simplified DES. Because no S-Box is used as a building block, the attack on the new Modified AES algorithm is 0%.

3.1 Background on Preventing DC Attacks with a KDM Function on the AES

Without a doubt, cryptographic algorithms such as AES are used by IoT devices and platforms to ensure the safety and confidentiality of highly classified information and data. [82]- [95].

As a result, new services provided by IoT devices must be adequately secured using robust cryptographic algorithms such as AES [105]. Simultaneously, as the improvement of security and privacy on IoT devices is observed as increasingly surpassing the use of solid cryptographic algorithms such as AES, the more attackers create and improve different techniques of attacking the distinct reliable algorithms [77], [96]. Most common algorithms,

such as AES, are being attacked using various mathematical methods, such as DC attacks [105]- [151]. For example, four-round AES is attacked with DC attacks [105]. AES has been used to secure data in online transactions, and smart cards in other IoT devices [105]. An exhaustive research attack was compared to an AES attack utilizing DC attacks on fewer rounds variant [140]. DC attacks have proven to be more effective than comprehensive research attacks, which are regarded as the upper bound attack in cryptography [140], [147].

Today, the AES algorithm is still used to secure sensitive information and data stored on IoT devices [82]- [95]. AES has been discovered to aid in establishing IoT sensor communication security in various IoT devices such as intelligent energy grids, Machine to Machine (M2M) communications, buildings, and data computing devices [84]. To ensure the safety of sensor nodes as IoT devices, communications are encrypted using the AES algorithm [85]. One of the IoT devices that uses an AES to secure data privacy and protection is the PRISEC module of the UbiPri middleware [86].

The newly created 32 output bit S-Boxes protect AES DC attacks on IoT devices because it contains many mathematical modulo operators. A KDM function creates a new 32-Bit S-Box suitable for the new Modified AES Algorithm and confuses the attacker. Furthermore, the majority of mathematical modulo operators are irreversible.

The primary concern is the DC attack used by trespassers on IoT devices to identify the cryptographic keys of an AES algorithm. The DC attack is launched against an AES [105]- [151]. For example, the DC attack has been tested on the Mini-AES algorithm [146]. The experiment revealed more than half of the secret keys. To decode the secret key, AES has been attacked using an algebraic DC attack [145]. The study conducted by [77] conveyed the basic principle that DC attack has been quested the strong chance of suitable events of plaintext pair differences and ciphertext pair differences generated in the deciding round. Lacko-Barto˘sova [141] demonstrated a DC two-round

AES attack with a complexity approach of a three-round AES attack. Lacko-Barto˘sova [141] has also demonstrated that the DC attack is dependent on the support of remarkable bitwise text differences. Grassi [144] used the DC attack and the "multiple-of-8" rule to attack five-round AES. According to Tunstall [105], the first attack is a four-round AES DC attack controlled to a differential completed with a high probability. The second strike is a five-round AES Square attack that takes 237.5 seconds of time complexity and 28 combinations of ciphertexts to break an AES cryptographic keys [105].

Even though AES is vulnerable, it is used on IoT devices. For example, IoT devices need an AES algorithm to encrypt information and transfer it to the next layer of security, known as the Message Queuing Telemetry Transport protocol [95]. The Message Queuing Telemetry Transport protocol (ISO/IEC PRF 20922) is an ISO standard for transferring encrypted information. The secret message was decrypted on the receiver side using the AES algorithm [95]. As IoT devices aggregated on recurring links related to regulating traffic across optimal links [94], the VMware SD-WAN Edge holds VMware SD-WAN Dynamic Multipath Optimization (DMPO) and an Extensive Application Recognition.

Furthermore, traffic is routed to other VMware SD-WAN Edges of different departments, personal data hubs, academic institutions, and workplaces, with AES for secure communication [94]. According to Sophia *et al.* [93], the department of health is a growing threat to patient safety worldwide. An e-healthcare Remote Clinical Sensor Network aids in collecting vital body information from private terminals via sensors such as IoT devices. The recommended method is based on policy initiatives of implementing a secured key and encoding it with an AES [93].

With all of this information, this research aims to recover an AES from DC attacks and secure all IoT devices and data using an AES algorithm. If not adequately investigated, a DC attack can destroy the complete security of IoT devices and users. Little research has been done to increase the number

of output bits on S-Boxes to combat the DC attack [105]- [151]. This research focuses on retaliating against a DC attack on an AES.

3.1.1 An AES Algorithm

The AES algorithm is a 128-bit symmetrical cryptosystem that is widely and commonly used on IoT devices [164] - [167]. An AES has four major phases called functions, which are as follows: *Substitute Byte* (*SubByte*), *Shift Rows* (*ShiftRows*), *Mix Columns* (*MixColumn*) and finally *Add Round Key* (*AddRoundKey*) [164] - [167]. Three of these four major functions have reciprocals, namely: *Inverse Mix Columns* (*InvMixColumn*), *Inverse Substitute Byte* (*InvSubByte*) and *Inverse Shift Rows* (*InShiftRows*). The only function that does not have an inverse is *Add Round Key* (*AddRoundKey*) [7], [164]. The main functions are used in the encryption operation, and the inverses are used in the decryption [164] - [167]. The decryption AND encryption processes are depicted in Figure 3.1.

The first step or function in the encryption process is *SubByte*. An AES algorithm employs a Substitution-Box in this function (S-Box). S-Box is a look-up table with inputs and outputs in the form of bytes [7], [164]. Using an AES S-Box, each input byte is replaced by a different unconventional byte in the *SubBytes* step [164] - [167]. Assume, as shown in Figure 3.2, that the input byte is $c000$ in hexadecimal notation, $c0 = x$ for a row, and $00 = y$ for a column. $c000$ is replaced by ba when examined from an AES S-Box on Figure 3.2, where x and y intersect.

During the decryption process, an AES S-Box is used in reverse. When inverse an AES S-Box is used, the step is called *InvSubBytes* and is the third step in decryption. *InvSubByte* is the direct inverse of *SubByte*. Please see Figure 3.2.

An AES converts a string of plaintext (input) into a 4×4 matrix; the matrix after the replacement or substitution is referred to as an AES's state. Note that a state is the output of each AES step or function. *MixColumns*

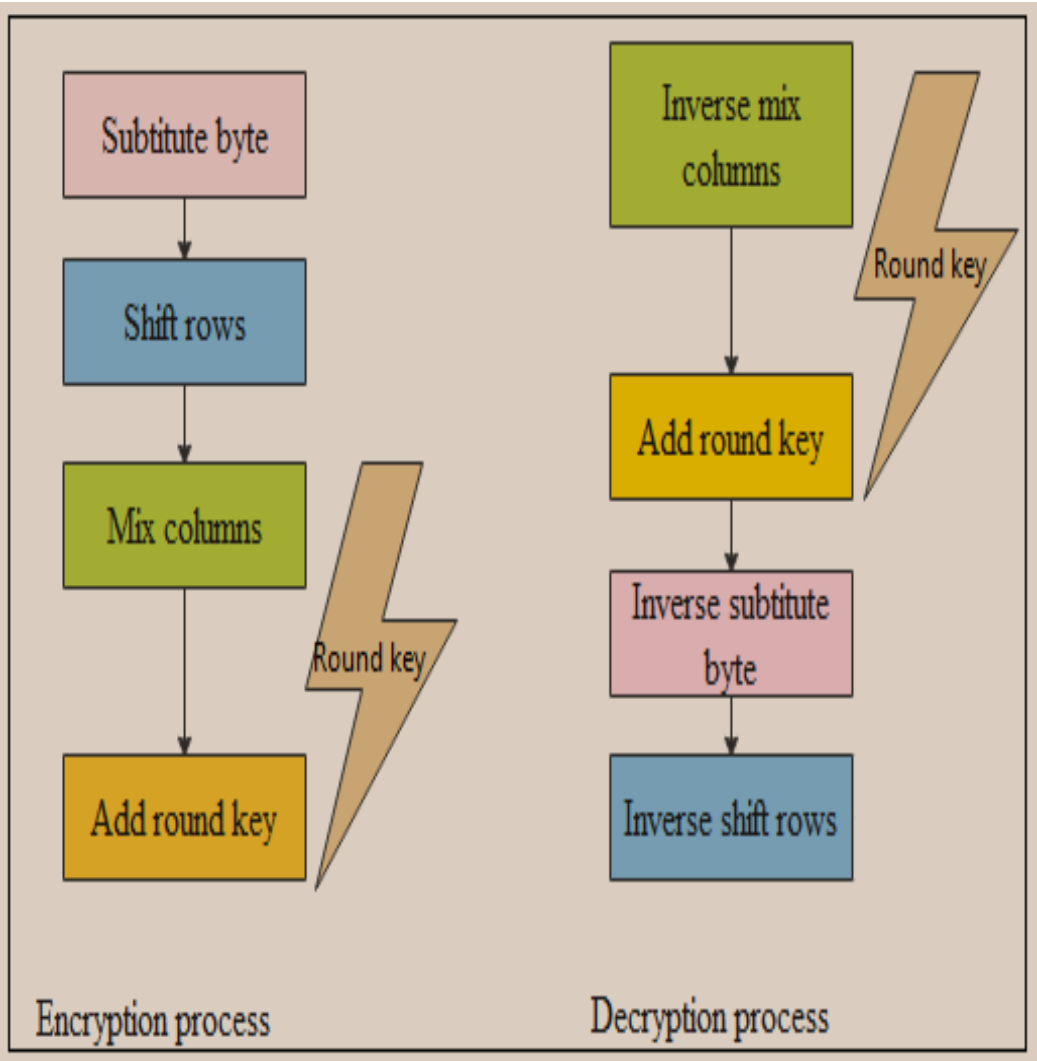
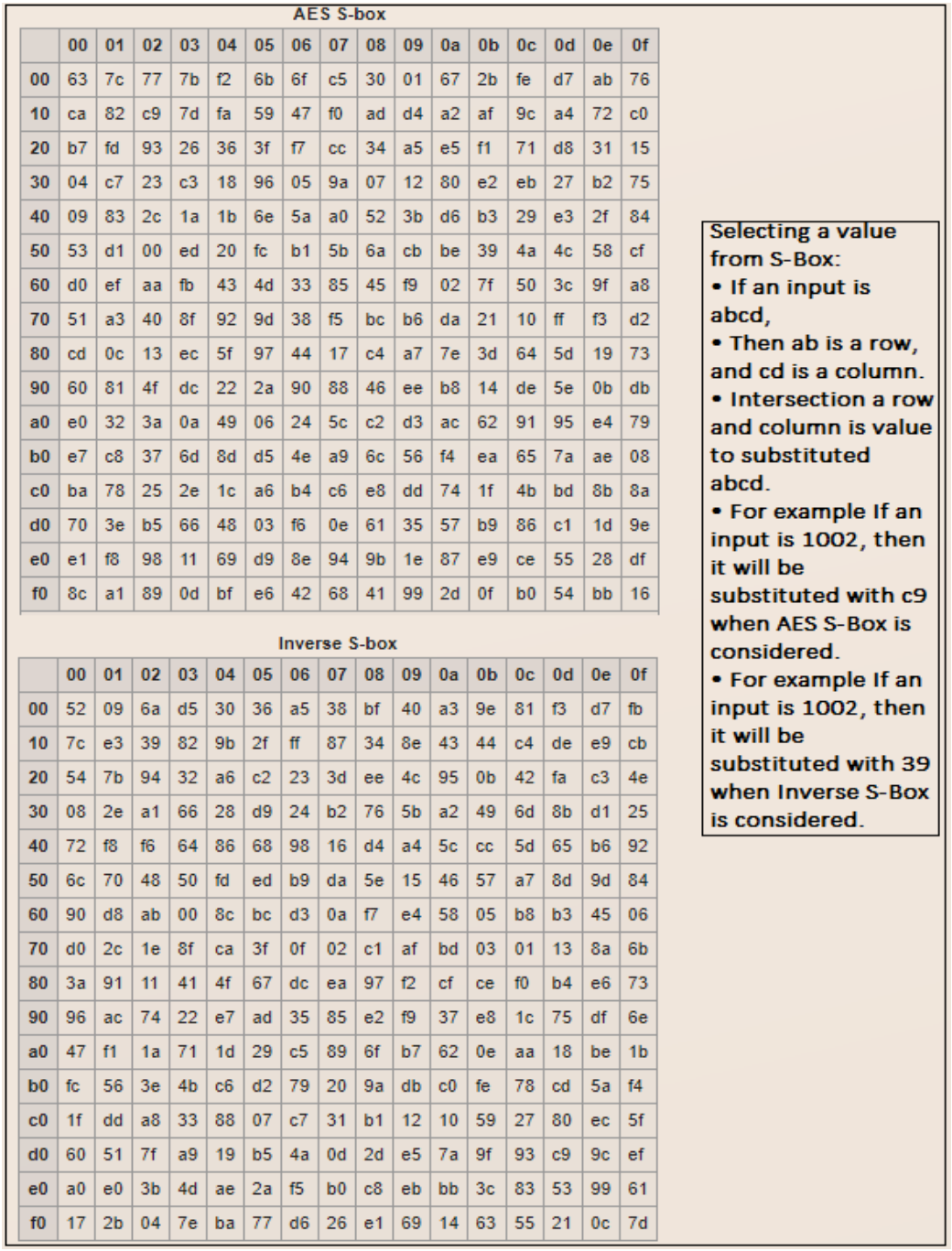


Figure 3.1: AES Decryption and Encryption Processes [164]



AES S-box

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Inverse S-box

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
10	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
20	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
30	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
40	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
50	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
60	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
70	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
80	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
90	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a0	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b0	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c0	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d0	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e0	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f0	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Selecting a value from S-Box:
- If an input is abcd,
- Then ab is a row, and cd is a column.
- Intersection a row and column is value to substituted abcd.
- For example If an input is 1002, then it will be substituted with c9 when AES S-Box is considered.
- For example If an input is 1002, then it will be substituted with 39 when Inverse S-Box is considered.

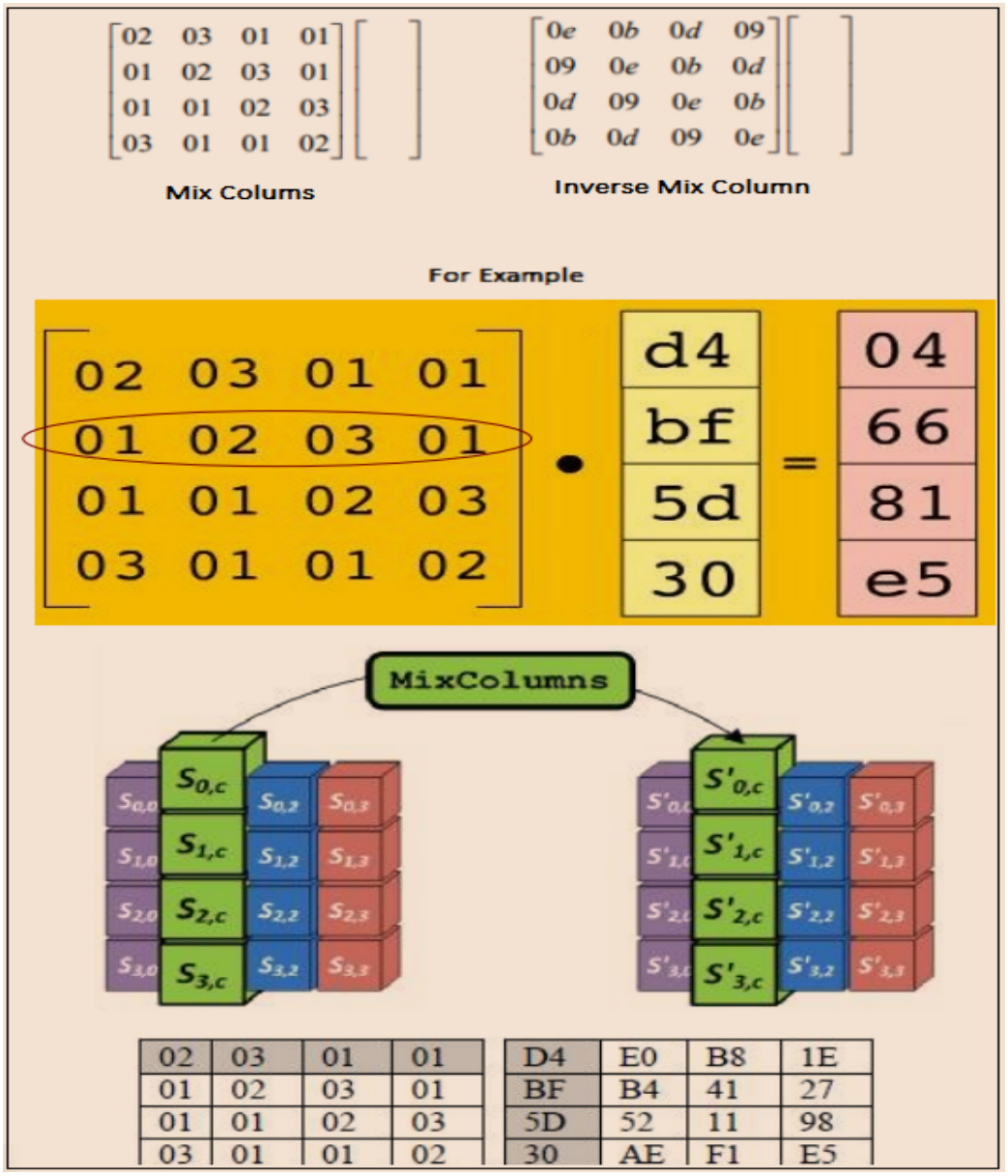
Figure 3.2: AES InveSubBytes and SubBytes with an Example [167]

is another critical function that keeps the state running. The mixing method, also known as $MixColumns$, is a multiplication method for combining matrix rows and columns. Using matrix transformation, each 8-bit entity of a row is multiplied by each 8-bit entity of the state column. Simply put, each row of the matrix transformation is used to multiply each column of the state [164] - [167].

The multiplication outputs are XORed to produce a distinct state. $InvMixColums$ is the inverse transformation of $MixColumn$. $InvMixColumns$ is obtained during the decryption process [7], [164]. The size of the states is always the same, which is a $4mathrmtimes4$ matrix. Please see Figure 3.3.

Add Round Key ($AddRoundKey$) is the final function or step of an AES during the encryption process. In contrast to other functions, $AddRoundKey$ does not have an inverse. The $AddRoundKey$ method is used in both the encryption and decryption processes. During the $AddRoundKey$ operation, either the state produced by $MixColumns$ or the state produced by $InvMixColumns$ is XORed with the state of key [7], [164]. Refer to Figure 3.4 for more information.

AES supports three original key sizes: 192-Bits, 128-Bits, and 256-Bits [164] - [167]. The encryption method consists of 10 rounds of key modification for a 128-bit key, 14 rounds for a 256-bit key, and 12 rounds for a 192-bit key [7], [164]. All subkeys are generated from an initial key; the size of the initial key determines the number of subkeys generated. Subkeys are employed during the encryption and decryption processes [164] - [167]. Figure 3.5 depicts the mathematical steps in generating subkeys.



02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

D4	E0	B8	1E
BF	B4	41	27
5D	52	11	98
30	AE	F1	E5

Figure 3.3: AES Inverse Mix Columns and Mix Columns [164]

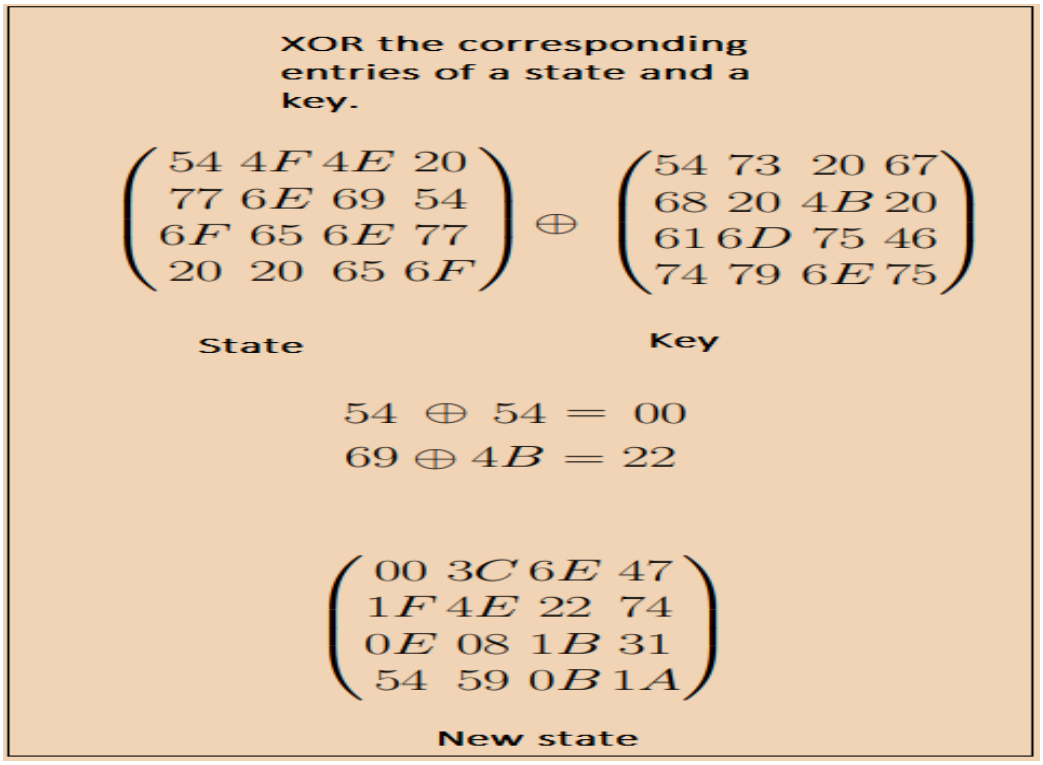


Figure 3.4: An AES's Key Addition Process [167]

To generate the subkey:

1. Use an initial 128-Bits key
2. For example, if initial 128-Bits key is defined as
   {54, 68, 61, 74, 73, 20, 6D, 79, 20, 4B, 75, 6E, 67, 20, 46, 75},
   Divide it into four words: W_0, W_1, W_2 *and* W_4
   {54, 68, 61, 74, 73, 20, 6D, 79, 20, 4B, 75, 6E, 67, 20, 46, 75},
   W_0 = {54, 68, 61, 74}, W_1 = {73, 20, 6D, 79}, W_3 = {20, 4B, 75, 6E} and
   W_3 = {67, 20, 46, 75}.
3. Calculated other words as follow:
   $W_i = W_{i-1} \oplus W_{i-4}$
   Where *i* is not a multiple of 4.
4. If *i* is a multiple of 4,
   - Rotate W_3 = {67, 20, 46, 75} left shift once to get a new W_3
   - W_3 = {20, 46, 75, 67}
   - Apply (S-Box), then output of S-Box = {B7, 5A, 9D, 85}
   - Add a round constant Rcon [i] = {01, 00, 00, 00} which yields:
     $g(W_3)$ = {B6, 5A, 9D, 85}
5. *Calculated other words as follow:*
   $W_4 = W_0 \oplus g(W_3)$, $W_5 = W_4 \oplus W_1$, $W_6 = W_5 \oplus W_2$ and $W_7 = W_6 \oplus W_3$

i	1	2	3	4	5	6	7	8	9	10
Rcon[i]	[01]	[02]	[04]	[08]	[10]	[20]	[40]	[80]	[1b]	[36]
	[00]	[00]	[00]	[00]	[00]	[00]	[00]	[00]	[00]	[00]
	[00]	[00]	[00]	[00]	[00]	[00]	[00]	[00]	[00]	[00]
	[00]	[00]	[00]	[00]	[00]	[00]	[00]	[00]	[00]	[00]

Figure 3.5: Key Scheduling in AES [164]

3.1.2 The DC Attack

The DC attack takes advantage of the high probability of specific events of plaintext differences and differences into the final round of the cipher [105]- [151]. Consider the following algorithmic program with plaintext $\wp = [\wp_1, \wp_2, ..., \wp_n]$ and ciphertext $\zeta = [\zeta_1, \zeta_2, ..., \zeta_n]$ [77]. Expect two inputs to the cipher, $\wp^{'}$ and $\wp^{"}$ with additive outputs $\zeta^{'}$ and $\zeta^{"}$, respectively. The input difference is calculated using $\Xi\ \wp = \wp^{'} \oplus \wp^{"}$, the symbol $\oplus$ shows XOR bitwise operator, and thus $\Xi\wp_i = \wp_i^{'} \oplus \wp_i^{"}$, correspondingly to the output difference where, $\Xi\zeta = \zeta^{'} \oplus \zeta^{"}$ and $\Xi\zeta_i = \zeta_i^{'} \oplus \zeta_i^{"}$ [96]. TTo apply the DC attack, the trespasser must recover the high differential probabilities of each S-Box used in the specific algorithm [105]- [151]. The intruder then calculates the outputs of high S-box differential probabilities that affect the known-plaintext difference $\Xi\wp = \wp^{'} \oplus \wp^{"}$ in relation to the ciphertext difference $\Xi\zeta = \zeta^{'} \oplus \zeta^{"}$ [96]. Furthermore, the intruder generates Difference-Distribution tables for each S-Box for input difference $\Xi\wp$ and output difference $\Xi\wp$ in order to reveal the differential characteristic. Because of the size of both input and output bits, most S-Boxes implemented on various algorithms are weak [105]. Because of S-Box's weakness, the attacker may easily observe the high difference probabilities of pair $(\Xi\wp_i, \Xi\zeta_i)$ of $(1/(2^n))$, where n is the bit number used as an output [77], [96]. The intruder examines all distinct pairs of input $\wp_i$ and output ζ_i of an S-Box, where i represents the $i - th$ bit of the $\wp_i$ and ζ_i, respectively. The high difference probabilities of each pair $(\Xi\wp_i, \Xi\zeta_i)$ are combined and used from round to round, utilizing S-Boxes as an independent building block of the particular algorithm. Assume that the differential characteristic for the second last round has a high enough probability $\rho\tau$. In that case, it is simple to find specific bits of the key or subkey used on the last round subkey by XORing all the potential keys of all affected nonzero difference bits TPS (Target Partial Subkeys) and operating one round backward through S-Boxes. The intruder requires $1/\rho\tau$ known plaintext-ciphertext pair differences [77], [96].

The intruder examines the difference pairs of the S-Boxes found in the cryptographic algorithm in the DC attack. For example, suppose the 4×4 S-Box is shown in Table 3.1 with plaintext $\wp = [\wp_1, \wp_2, \wp_3, \wp_4]$ and ciphertext $\zeta = [\zeta_1, \zeta_2, \zeta_3, \zeta_4]$ [77], [96].

Table 3.1: Simplified DES S-Box

$\wp$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
S($\wp$) = ζ	**4**	**E**	**D**	**1**	**2**	**F**	**B**	**8**	**3**	**A**	**6**	**C**	**5**	**9**	**0**	**7**

All difference pairs of an S-Box illustrated in Table 3.1, $(\Xi\wp_i, \Xi\zeta_i)$, can be examined and the probability of $\Xi\zeta_i$ given $\Xi\wp_i$ can be calculated by considering ciphertext pairs $(\wp^{'}, \wp^{"})$ such that $\Xi\wp = \wp^{'} \oplus \wp^{"}$ [105], [148]. For a 4×4 S-Box such as the one shown in Table 3.1 the intruder only considers all $16 = (2^4)$ values for $\wp^{'}$ and then the value of $\Xi\wp_i$ shows the value of $\wp^{"}$ to be $\wp^{"} = \wp^{'} \oplus \Xi\wp$ [77], [96].

Using the 4×4 S-Box shown in Table 3.1, the intruder can calculate the probability values of $\Xi\zeta$ for each plaintext pair $(\wp^{'}, \wp^{"} = \wp^{'} \oplus \Xi\wp)$ [105], [149]. Table 3.2 shows the binary values of $\wp$, ζ, and the ciphertext values for $\Xi\zeta$ for given plaintext pairs $(\wp, \wp \oplus \Xi\wp)$ are presented in for $\Xi\wp$ values of $1011_{binary\ number}$, $1000_{binary\ number}$, and $0100_{binary\ number}$.

Table 3.2 depicts the last three columns of $\Xi\zeta$ values for the $\wp$ value row and the specific $\Xi\wp$ value column [105]- [145]. The intruder can see from Table 3.2, that the occurrence number of $\Xi\zeta = 0010_{binary\ number}$ for $\Xi\wp = 1011_{binary\ number}$ is 8 over 16 possible values, so the probability = 8/16; the occurrence number of $\Xi\zeta = 1011_{binary\ number}$ given $\Xi\wp = 1000_{binary\ number}$ is 4 over 16; and the occurrence number of $\Xi\zeta = 1_{binary\ number}$ given $\Xi\zeta = 0100_{binary\ number}$ is 0 over 16 [105]- [151].

The intruder tabularizes the entire data for the 4×4 S-Box shown in Table 3.1 in a Difference-Distribution Table with columns representing $\Xi\zeta_{hexadecimal}$ and the rows representing $\Xi\wp$ values [96], [149].

Table 3.2: S-Box Difference Pairs of 4 x 4

$\wp$	ζ	$\Xi\zeta$ $\Xi\wp = 1011$	$\Xi\zeta$ $\Xi\wp = 1000$	$\Xi\zeta$ $\Xi\wp = 0100$
0000	1110	**0010**	**1101**	**1100**
0001	0100	**0010**	**1110**	**1011**
0010	1101	**0111**	**0101**	**0110**
0011	0001	**0010**	**1011**	**1001**
0100	0010	**0101**	**0111**	**1100**
0101	1111	**1111**	**0110**	**1011**
0110	1011	**0010**	**1011**	**0110**
0111	1000	**1101**	**1111**	**1001**
1000	0011	**0010**	**1101**	**0110**
1001	1010	**0111**	**1110**	**0011**
1010	0110	**0010**	**0101**	**0110**
1011	1100	**0010**	**1011**	**1011**
1100	0101	**1101**	**0111**	**0110**
1101	1001	**0010**	**0110**	**0011**
1110	0000	**1111**	**1011**	**0110**
1111	0111	**0101**	**1111**	**1011**

Table 3.1 gives the Difference-Distribution Table for the 4×4 S-Box illustrated in Table 3.3 [77], [144]. Table 3.3 depicts the occurrence number of each element corresponding ciphertext difference $\Xi\zeta$ value given the plaintext difference $\Xi\wp$ [105], [140], [141], [96]. Aside from the specific cases of $(\Xi\wp = 0, \Xi\zeta = 0)$, the intruder can see that the highest value in Table 3.1 is 8, corresponding to $\Xi\wp = B_{hexidecimal}$ and $\Xi\zeta = 2_{hexidecimal}$ [105]- [148].

Table 3.3: DES Difference-Distribution Table (DDT)

Input Difference $\Xi\wp$	**Output Difference** $\Xi\zeta$															
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	0	0	0	2	0	2	4	0	4	2	0	0
2	0	0	0	2	0	6	2	2	0	2	0	0	0	0	2	0
3	0	0	2	0	2	0	0	0	0	4	2	0	2	0	0	4
4	0	0	0	2	0	0	6	0	0	2	0	4	2	0	0	0
5	0	4	0	0	0	2	2	0	0	0	4	0	2	0	0	2
6	0	0	0	4	0	4	0	0	0	0	0	0	2	2	2	2
7	0	0	2	2	2	0	2	0	0	2	2	0	0	0	0	4
8	0	0	0	0	0	0	2	2	0	0	0	4	0	4	2	2
9	0	2	0	0	2	0	0	4	2	0	2	2	2	0	0	0
A	0	2	2	0	0	0	0	0	6	0	0	2	0	0	4	0
B	0	0	8	0	0	2	0	2	0	0	0	0	0	2	0	2
C	0	2	0	0	2	2	2	0	0	0	0	2	0	6	0	0
D	0	4	0	8	0	0	0	4	2	0	2	0	2	0	2	0
E	0	0	2	4	2	0	0	0	6	0	0	0	0	0	2	0
F	0	2	0	0	6	0	0	0	0	4	0	2	0	0	2	0

As a result, the probability of $\Xi\zeta = 2_{hexidecimal}$ knowing an arbitrary pair of plaintext values that satisfy $\Xi\wp = B_{hexidecimal}$ is 8/16 [77], [96]. On the other hand, the smallest value in Table 3.1 is 0 and occurs for various pairs. In this situation, the probability of the $\Xi\zeta$ value happening knowing the $\Xi\wp$ value is 0. With all of this information, the intruder can discover the algorithm with the highest percentage of secrete bits key using a similar S-Box to the one defined in Table 3.1 [77], [96]. The few remaining secret key bits are discovered through simple mathematical and statistical analysis, as well as trial and error.

3.1.3 Development of the KDM Function

A new function known as a KDM is included in this study. Please see Figure 3.6.

A KDM function is a new C++ function that is only used to increase DC attack blockage on an AES algorithm that is required on IoT devices. This function is obtained after modifying the S-Boxes of an AES algorithm to generate the 32-bit output S-Boxes. A KDM function's primary function is to ensure that newly 32-bit output S-Boxes are compatible with an AES cipher infrastructure.

In layman's terms, a KDM function coordinates the use of all newly 32-bit output S-Boxes during the encryption and decryption processes of the newly adjusted AES algorithm. A KDM function is used to create a new 32-Bit S-Box suitable for the new Modified AES Algorithm and confuses the attacker because it contains many mathematical modulo operators. Furthermore, the majority of mathematical modulo operators are irreversible. Newly generated 32-bit output S-Boxes will not be set in algorithms without a KDM function. This KDM function has specific properties that ensure the DC attack is blocked. These are the properties:

i. In contrast to S-Boxes, where a look-up table with determined inputs and outputs is used, the production of a KDM function is not deter-

mined.

ii. The output of a KDM function is hidden and calculated, unlike the output of an AES S-Box, which is visible on a look-up table.

iii. A KDM function cannot be changed. If an output of a KDM function is identified, this does not imply that an input can be reverse-calculated and recovered. The reason for this is that a KDM function is made up of a large number of modular operators.

iv. Constant numbers used in a KDM function (such as $Muthavhine$, $Khumbelo$, and $Difference$) are unfactorizable. Refer to Figure 3.6.

v. All of the functions that make up a KDM function are non-linear.

vi. A KDM function's input is 32 bits long, and the attacker cannot simply construct the 2^{32} Difference-Distribution Table using a computer.

vii. A KDM function takes the input of 32-bit S-Boxes and processes it. After that, the modified AES algorithm generates a unique output value. The attacker is perplexed because a new specific output value is unpredictable.

viii. $state32hold$ is the output of 32 bits S-Boxes. This output is passed to the KDM function, which returns an unpredicted variable called $Khumbelo$. Refer to Figure 3.6.

ix. Because S-Boxes are mathematically protected and unchangeable in a KDM function, all functions in an AES algorithm that call S-Boxes must call or use a KDM function after implementing a KDM function.

x. The 32-bit output S-Boxes are tamper-proof thanks to a KDM function. M_AES will not produce the expected results if the positions of the 32-bit output S-Boxes are changed or the 32-bit S-Boxes are replaced.

```
int KDM_Function(state32hold)
{
 int i,j;
 uint64_t TempState =0, Muthavhine=4294967296, T=0,V=0, X=0;
 uint64_t Khumbelo = 041760427607,
 Difference= 040035532523;
 uint64_t Mod[6]= {256604724, 40037230360,
  7779667,4294968531,0273,4};
 for(i=0,j=0;i<4&&j<4;i++,j++)
 {
 T = state32hold[j][i] * (bool)(state32hold[j][i]/ Muthavhine);
 V = Muthavhine * (bool)(Muthavhine / state32hold[j][i]);
 state32hold[j][i] = T+V;
 }
 for(i=0,j=0;i<4&&j<4;i++,j++)
 {
 if (state32hold[j][i] > Muthavhine)
 { Khumbelo =    (Mod[0] ^ Khumbelo);
   Difference=   (Mod[2] ^ Muthavhine) % Khumbelo;
   Muthavhine =      (Mod[2] ^ Difference) % Mod[3];
 }
  else
    {
    Muthavhine = (state32hold[j][i]<<Mod[4]) % Khumbelo;
    Khumbelo = (state32hold[j][i] <<Mod[5]) % Difference;
    Difference = ((state32hold[j][i]% Khumbelo) <<Mod[4]);
    Khumbelo = (Muthavhine ^ Difference)% Mod[2];
    Difference = ((Muthavhine % Khumbelo)^ Mod[0]);
    Muthavhine = (Khumbelo ^ Difference)% Muthavhine ;
    }//Modular operators (%) make variable to be unknown,
  } // invisible and irreversible to an intruder.
  for(i=0,j=0;i<4&&j<4;i++,j++)
  { while (Khumbelo)
   {
    TempState = (~state32hold[j][i]) & Khumbelo;
    state32hold[j][i]=_abs64(state32hold[j][i] ^ Khumbelo);
statehold[j][i]=(state32hold[j][i]/(Mod[2]^Muthavhine))^Mod[4];
    Khumbelo = (TempState << 1);
   }
    TempState = Khumbelo ^ TempState;
    Khumbelo = Khumbelo % Muthavhine;
   }
}
```

Figure 3.6: A KDM Function for Creating a New 32-S-Box for the Modified AES Algorithm

A KDM function contains many mathematical modulo operators. This study employs a KDM function to create a new 32-Bit S-Box suitable for the new Modified AES Algorithm and confuse the attacker.

Furthermore, the majority of mathematical modulo operators are irreversible. Unlike the traditional S-Boxes used in most AES algorithms, a KDM function is more resistant to DC attacks. A KDM function successfully blocks the DC attack of a recently modified AES algorithm while also being appropriate for newly 32-bit S-Boxes. A KDM function is built mathematically as follows:

Assign: $Muthavhine = 4294967296$, $Khumbelo = 4559351687$ and $Difference = 4302746963$

Create the first for both i and j less than four, where both i and j range form 0 to 4,

do: assign

$T = state32hold[j][i] \times (\frac{state32hold[j][i]}{Muthavhine})$. where $state32hold[j][i]$ is an input of a KDM function from 32-bit S-Box.

do: assign

$V = Muthavhine \times (\frac{Muthavhine}{state32hold[j][i]})$

Change the value of $state32hold[j][i]$, to be the value of $T+V$ by assigning $state32hold[j][i] = T + V$.

Close the first for loop.

Create an array of six elements called $Arraof6$ and assign to as $Arraof6 = 256604724, 40037230360, 7779667, 4294968531, 0273, 4$ where $Arraof6_0$ is the first element of $Arraof6$ defined as $Arraof6_0 = 256604724$, $Arraof6_1 = 40037230360$,...,$Arraof6_5 = 4$.

Create the second for loop both i and j less than four, where both i and j range form 0 to 4.

Recall the value of $state32hold[j][i]$ calculated from the $first$ for loop.

Compare the value of $state32hold[j][i]$ to the value of $Muthavhine$.

Create condition one: if $state32hold[j][i]$ is greater than $Muthavhine$,

then do: assign

$Khumbelo = Arraof6_0 \oplus Khumbelo$

$Difference = (Arraof6_2 \oplus Muthavhine)\ modulo\ (Khumbelo)$.

Where *modulo* operation is the mathematical operator that returns the remainder of a division $(Arraof6_2 \oplus Muthavhine)$ divided by $Khumbelo$.

do: assign

$Muthavhine = (Arraof6_2 \oplus Difference)\ modulo\ (Arraof6_3)$.

Close condition one.

Recall the value of $state32hold[j][i]$ calculated from the $first$ for loop.

Compare the value of $state32hold[j][i]$ to the value of $Muthavhine$.

Create condition two: if $state32hold[j][i]$ is less than or equal to $Muthavhine$, then

do: assign

$Muthavhine = (state32hold[j][i] <<< Arraof6_4)\ modulo\ (Khumbelo)$.

Where $<<<$ is left circular shifting of the bits, for instance, 5 in decimal = 0101 in binary. If 0101 is left-shifted by 1, then 0101 will be 1010 in binary, which equals 10 in decimal or A in hexadecimal.

do: assign

$Khumbelo = (state32hold[j][i] <<< Arraof6_5)\ modulo\ (Difference)$.

$Difference = (state32hold[j][i]\ modulo\ Khumbelo)$

$<<< Arraof6_4)$.

$Khumbelo = (Muthavhine \oplus Difference)\ modulo\ (Arraof6_2)$.

$Difference = Muthavhine \oplus Khumbelo + Arraof6_0)$.

$Muthavhine = (Khumbelo \oplus Difference)\ modulo\ (Muthavhine)$.

Close condition two and the second for loop.

Create the third for loop where i and j are less than four, where both i and j range from 0 to 4.

Recall all the returned values calculated from the first and second for loops. If the value returns to variable $Khumbelo$, greater than 0, then create a variable $TempState$.

do: assign

$TempState = NOT(state32hold[j][i])ANDKhumbelo.$

Where NOT and AND are bitwise operators. Note that NOT return negative number increased by 1 if an input is a positive integer. For instance, $NOT(2) = -3$, $NOT(5) = -6$, $NOT(10) = -11$ and so on.

do: assign

$state32hold[j][i] = |(state32hold[j][i] \oplus Khumbelo|$, where $|x|$ means absolute operator. An absolute operator changes every negative value to be positive. For instance, $|-x| = |x| = x$.

do: assign

$statehold[j][i] = (\frac{state32hold[j][i]}{Arraof6_2 \oplus Muthavhine}) \oplus Mod4.$

do: assign

$Khumbelo = TempState <<< 1.$

Note that the expression of $Khumbelo = TempState <<< 1$ always reduces the value of $Khumbelo$ until $Khumbelo$ is less than 0. It also checks if $Khumbelo$ is greater than 0. If $Khumbelo$ is greater than 0, repeat the third for loop until $Khumbelo$ is less than 0.

Else do: assign

$TempState = Khumbelo \oplus TempState$

$Khumbelo = Khumbelo($ $modulo$ $(Muthavhine))$

Send or return the new value of $statehold[j][i]$ to be used by other AES functions or building blocks

Close the third for loop.

Close a KDM function.

A KDM Function takes 32-bit output value from S-Box as $state32hold[j][i]$ and returns a new value $state32hold[j][i]$ value as an output. A KDM Function also makes $Muthavhine$ value, $Difference$ value, and $Khumbelo$ value unfactorizable polynomials, and then modular operators are used for confusion and diffusion to block reverse engineering for intruders. The modular operator ($modulo$) changes the value of the variables inside a KDM Function.

The modular operator also gives a confusing input range when intruders reverse back a KDM Function to guess the correct information used in that event. Refer to Figure 3.7.

The value of $Muthavhine$, $Difference$, and $Khumbelo$ are also constantly kept unfactorizable polynomial variables non-linear and cumbersome to construct Difference Distribution Table using any machine. Modular operators also make variables unknown, invisible, and irreversible to intruders. A KDM function makes a new 32-Bit S-Box suitable for the new Modified AES Algorithm and confuses the attacker since it comprises many mathematical modulo operators. Additionally, most mathematical modulo operators are irreversible. For more mathematical features of a KDM function and C++ comments, refer to Figure 3.6. For more detail on a KDM function and flowchart, refer to Figure 3.7.

3.2 Methodology for Preventing DC Attacks Using a KDM Function on AES

The primary goal of this study was to protect an AES algorithm discovered on IoT devices from a DC attack. The original 8-Bits-output S-Box and inverse Box of an AES algorithm was replaced in this study with newly generated 32-Bits-output S-Boxes. For the suitability of newly generated 32-Bits-output S-Boxes, a unique mathematical function called KDM was developed. The newly generated 32-Bits-output S-Boxes were inserted into an AES algorithm to obtain a more desirable encryption and decryption process with DC attack protection. A KDM function has been used to establish a new 32-Bit S-Box appropriate for the innovative Modified AES Algorithm and confuse the intruder because it encompasses many computational modulo operators, refer to Figure 3.7.

Furthermore, the majority of mathematical modulo operators are irreversible. After embedding newly generated 32-Bits-output S-Boxes and a KDM function in an AES's infrastructure, a new modified AES algorithm

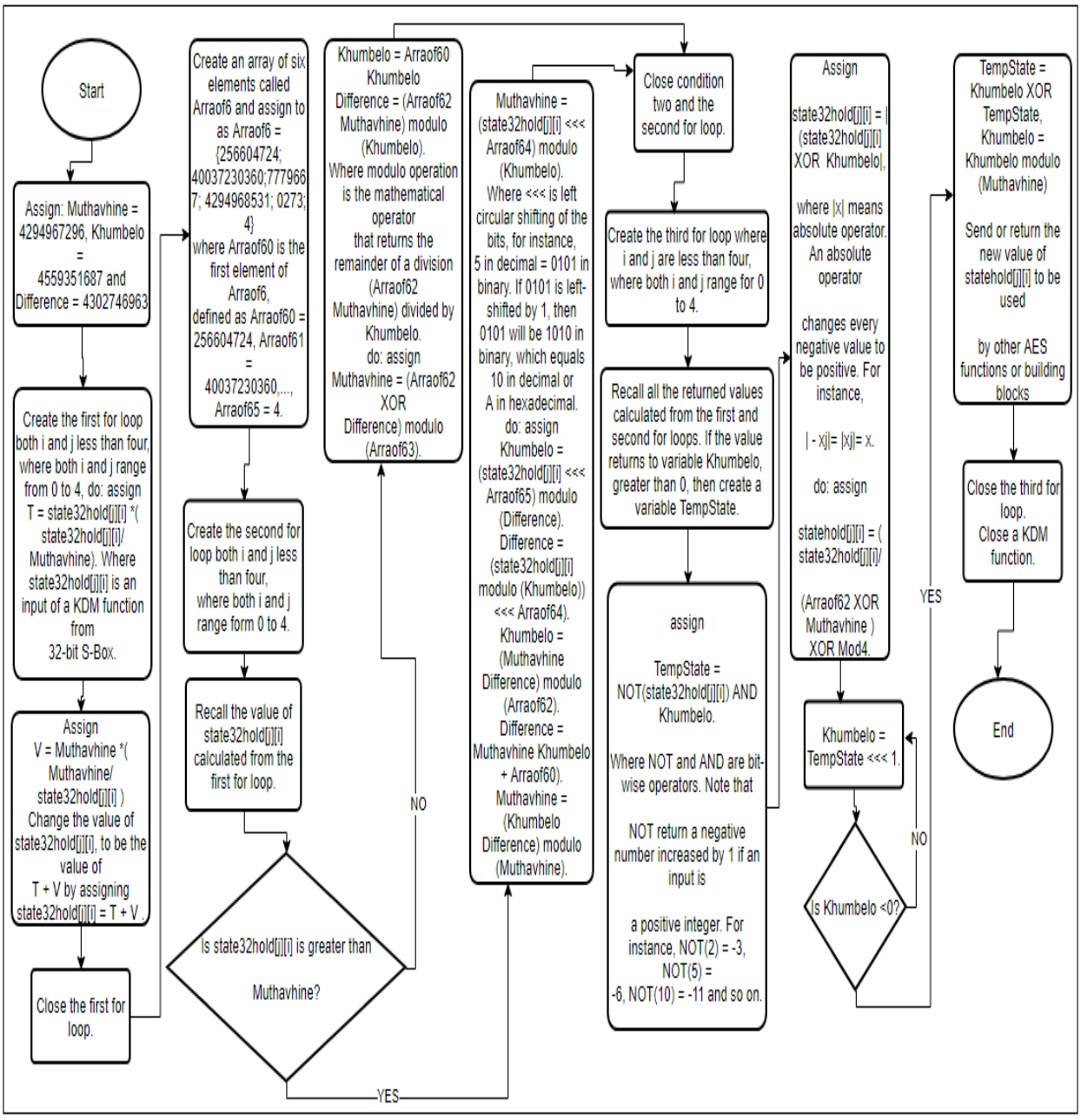


Figure 3.7: A KDM function flowchart

was developed. The recently modified AES algorithm, with freshly generated 32-Bits-output S-Boxes and a KDM function, was dubbed M_AES in this study. The mode of operation of M_AES was significantly different from that of an original AES algorithm because the strength, encryption process, and resistance to DC attacks were substantially more significant than that of an original AES algorithm found on IoT devices. The study was carried out as follows:

i. IoT devices (such as mobile phones, contactless payments, Machine to Machine (M2M), and sensors) were used to discover an original AES algorithm.

ii. Using test vectors from the literature review, the precision of an original AES algorithm was confirmed and inspected.

iii. All procedures implemented on an original AES algorithm throughout DC attacks were evaluated using C++.

iv. The existing AES 8-Bits-output S-Box and reciprocal S-Box were transformed into the newly established 32-Bits-output S-Boxes.

v. Using C++, the new KDM function was integrated inside an original AES algorithm infrastructure. Refer to Figure 3.6.

vi. That all the other functions in the original AES algorithm that used S-Boxes and the invertible of 8-Bits-output were altered to use KDM functions with new fully 32-Bits-output S-Boxes as feedback. For instance, if

$$Output = \zeta = SB_i(\wp) \tag{3.1}$$

Note: $SB_i(\wp)$ on Equation 3.1 is 8-Bits-output S-Box. Equation 3.1 is substituted using Equation 3.2.

$$KDM_function(SB_i(\wp), Khumbelo), \tag{3.2}$$

$SB_i(\wp)$ on Equation 3.2 is a newly 32-Bits-output S-Box because an AES S-Box and its inverse were converted to give new 32-Bits-output S-Boxes.

vii. The M_AES algorithm was used to reconstruct the possibility of DC attacks. If the DC attacks still were effective after the new fully integrated 32-Bits-output S-Box and a KDM function. Steps (iii) and (iv) were repeated if it was still possible.

viii. Only when DC attacks were prevented in steps (iii), (iv), and (v) a new M_AES algorithm was accepted as a M_AES algorithm, which was embedded with the newly 32-Bits-output S-Box and a KDM function. As a result, the M_AES algorithm was discovered to be resistant to DC attacks. Refer to Figure 3.9.

After DC attacks, the research methodology used a more obstreperous Difference-Distribution Table to prevent attackers from discovering AES keys. The M_AES algorithm's security is determined by the size of the S-Boxes output bits and a KDM function. AES's S-Box and its inverse's original output bits were low (8-Bits). Intruders can easily compromise such an algorithm.

A newly generated 32-bit output S-Box and its inverse were used to replace all 8-bit output S-Boxes, increasing the size of output bits from 8 to 32-bit. The M_AES algorithm made output bits more resistant to DC attacks. Experiments revealed that the new 32-bit output S-Box and its inverse effectively blocked DC attacks. Simultaneously, a KDM function is used to create a new 32-Bit S-Box suitable for the new Modified AES Algorithm and confuse the attacker due to many mathematical modulo operators. Furthermore, the majority of mathematical modulo operators are irreversible.

The research methodology was described using the schematic diagram shown in Figure 3.8.

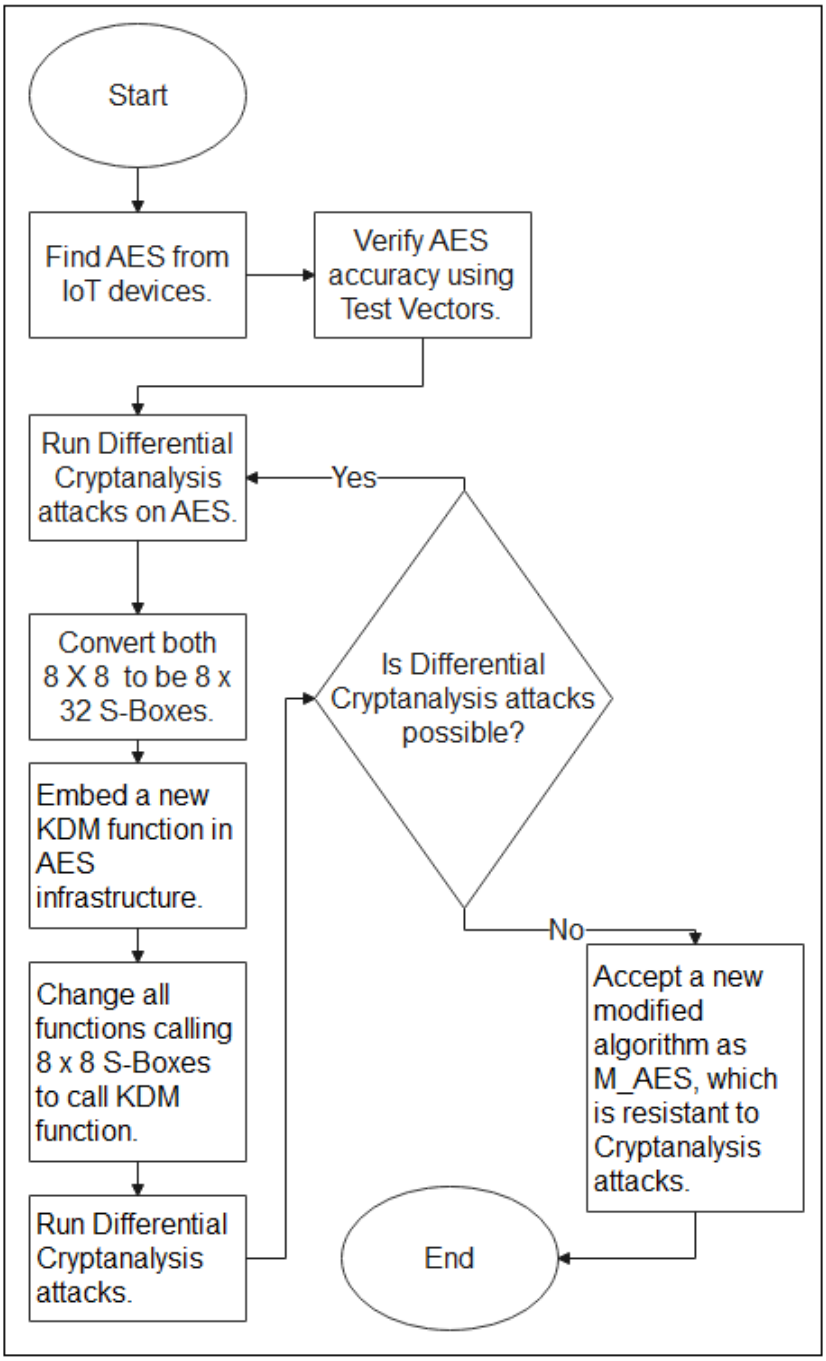


Figure 3.8: AES Research Methodology Schematic Diagram

The results successfully prevented the construction of the Difference-Distribution Table and resulted in a complex process for performing DC attacks on the M_AES algorithm. Please see Figure 3.9. The difference between Figure 3.1 and Figure 3.9 was the unique 32-bit output S-Box, exact reverse 32-bit output S-Box, and a KDM function.

The S-Box and its inverse of the AES were found to be 8 x 8, indicating that they had 8-Bits-inputs and 8-Bits-outputs, respectively. The study discovered that using these S-Box descriptions, it was simple to create a Difference-Distribution Table. Back to our example, the 4 $\times$ 4 S-Box shown in Table 3.1 produced a Difference-Distribution Table of 2^4 x 2^4 illustrated in Table 3.3 with high high-probability components of detecting secret key bits. When an S-Box has X-Bits of inputs and Y-Bits of output, its Difference-Distribution Tables are typically a 2^X x 2^Y matrix. As a result, the Difference-Distribution Table shown in Table 3.3, was shown to be 2^4 x 2^4.

In this research, C++ program was designed to generate a Difference-Distribution Table of 2^4 x 2^4, as shown in Table 3.3, by using Equation 3.2. Using the 4 $\times$ 4 S-Box illustrated in Table 3.1, the code proved simple to attack any algorithm. Furthermore, the code stated that it is to build the Difference-Distribution Table of 2^8 x 2^8, using the 8 x 8 AES Box and its inversed defined in Figure 3.2.

To guard against DC attacks, a new 32-bit output S-Box and its inverse were created to replace the 8 x 8 AES Box and its inverse defined in Figure 3.2.

For example, an AES S-Box in Figure 3.2 was replaced by a new 32-bits output AES S-Box in Appendix A, Figure A.1. Figure 3.2 inverse S-Box was replaced with the new 32-bits output AES inverse S-Box found in Appendix A, Figure A.2.

The KDM function was designed to work with the new 32-bit output S-Box and its inverse in a new M_AES algorithm. Because it contains many mathematical modulo operators, a KDM function creates a new 32-Bit S-Box suitable for the new Modified AES Algorithm and confuses the attacker. Furthermore, the majority of mathematical modulo operators are irreversible. A new 32-Bit S-Box was immune to DC attacks. Please see Figure 3.9. When comparing Figure 3.1 and Figure 3.9, the M_AES algorithm in Figure 3.9 was more resistant to DC attacks than the AES algorithm in Figure 3.1.

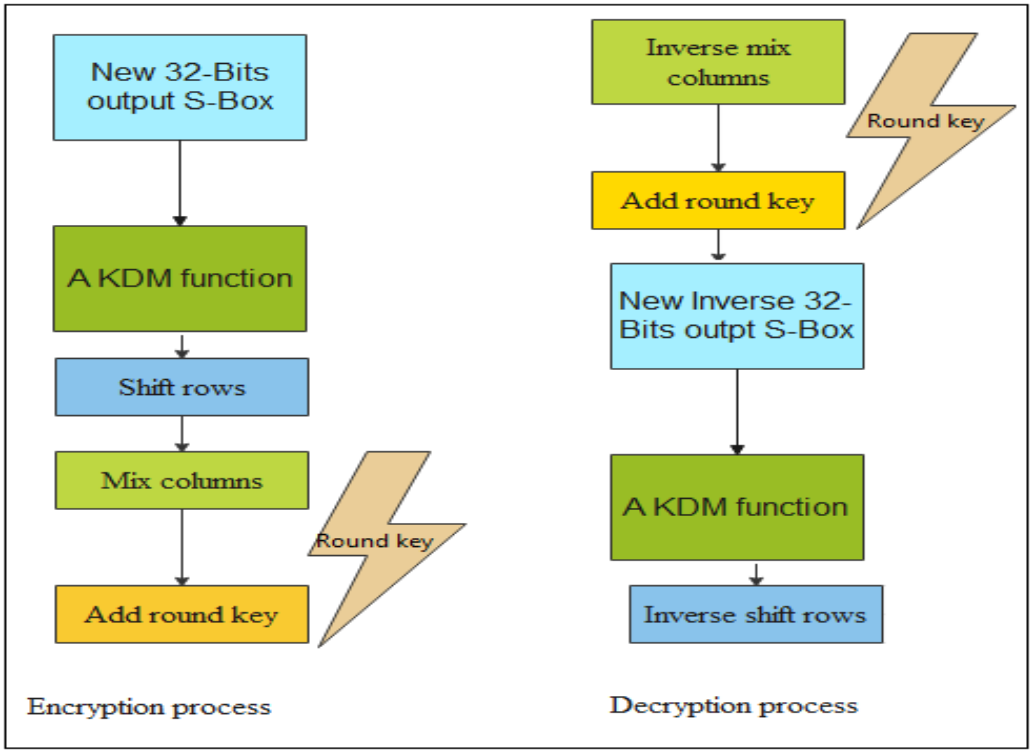


Figure 3.9: New Algorithm Modified AES (M_ AES) with Encryption and Decryption Process

3.3 Results and Analysis of Preventing DC Attacks Using a KDM Function on AES

The results on an AES demonstrated that the DC attack was feasible. The estimation of the S-Boxes was the most critical component in determining all possible outcomes of the DC attack. An AES's S-Box was 8 x 8, demonstrating that 8 bits were input and 8 bits were yielded. Difference-Distribution Table discovered that making the Difference-Distribution Table with an 8 x 8 AES S-Box was simple.

The consider wrote a C++ program to generate the Difference-Distribution Table of the 4 x 4, 6 x 4, 8 x 8, and 8 x 32 S-Boxes. The validation of code was attempted using a rearranged 4 x 4 DES S-Box given in Table 3.1, a 6 x 4 DES S-Box given in [168] page 12 and 13, an 8 x 8 AES S-Box given in Figure 3.2, and a newly developed 8 x 32 S-Box of M_AES algorithm given in Appendix A, Figure A.1.

The purpose of approving the code was to confirm that the composing C++ exploratory yield Difference-Distribution Table was correct compared to the theoretical yields.

Figure 3.15 showed the C++ Difference-Distribution Table of a 4 x 4 S-Box, and the substances were the same as in Table 3.3. As a result, the C++ Difference-Distribution Table of 4 x 4 carried out good results. The C++ Difference-Distribution Table of 4 x 4 S-Box execution time was 0.2815 seconds. Refer to Figure 3.15. The 4 x 4 S-Box Difference-Distribution Table is a matrix of 2^4 x 2^4 = 16 x 16 matrix with 256 substances.

Figure 3.10 showed a C++ Difference-Distribution Table of 6 x 4, and the substances were the same as in the hypothetical Difference-Distribution Table shown [168] page 12 and 13. As a result, the C++ Difference-Distribution Table of 6 x 4 carried out good results. The C++ Difference-Distribution Table of 6 x 4 S-Box execution time was 1.2100 seconds. Refer to Figure 3.10. It should be noted that the 6 x 4 S-Box Difference-Distribution Table is a

framework of 2^6 x 2^4 = 64 x 16 matrix with 1024 substances.

The test is carried out on the 8 x 8 AES S-Box shown in Figure 3.14. It is worth noting that the Difference-Distribution Table of 8 x 8 AES S-Box is a 2^8 x 2^8 = 256 x 256 framework with 65536 substances. Five pages are required to show a complete unmistakable 256 x 256 network as the Figure like Figure 3.15 and Figure 3.10. As a result, Figure 3.14 has dots to show that it is a massive 256 x 256 framework. The C++ Difference-Distribution Table of 8 x 8 S-Box execution was 23.6800 seconds. Refer to Figure 3.14.

The investigation is carried out on a recently created 8 x 32 S-Box of the M_AES algorithm, as shown in Appendix A, Figure A.1. After three hours, the program crashed before the Difference-Distribution Table was executed. No machine or computer can compute the Difference-Distribution Table of the 2^8 x 2^{32} = 256 x 4294967296 network, which is expected to contain the 1099511627776 substances. It was absurd to perform the DC attack on a newly created 8 x 32 S-Box of the M_AES algorithm shown in Appendix A, Figure A.1 without a DDT.

The primary substance of the Difference-Distribution Table of 4 x 4 S-Box was integer 16, (2^4) because S-Box required four bits as the most crucial parameter. Refer to Figure 3.15 16 is a byte represented in binary as 00010000. If each component of the 4 x 4 S-Box Difference-Distribution Table is treated as a byte, the memory required to construct the 4 x 4 S-Box Difference-Distribution Table was 8 bits x 256 = 256 bytes. The number of substances shown on a 4 x 4 S-Box Difference-Distribution Table is 256. A machine or computer is capable of handling 4096 bytes.

The primary substance of the Difference-Distribution Table of 6 x 4 S-Box was integer 64, (2^6) because S-Box required six bits as the most elevated parameter. Refer to Figure 3.10 64 is a byte represented in binary as 001000000. If each 6 x 4 S-Box Difference-Distribution Table substance is treated as a byte, the memory required to construct the 6 x 4 S-Box Difference-Distribution Table was 8 bits x 1024 = 1024 bytes. The number

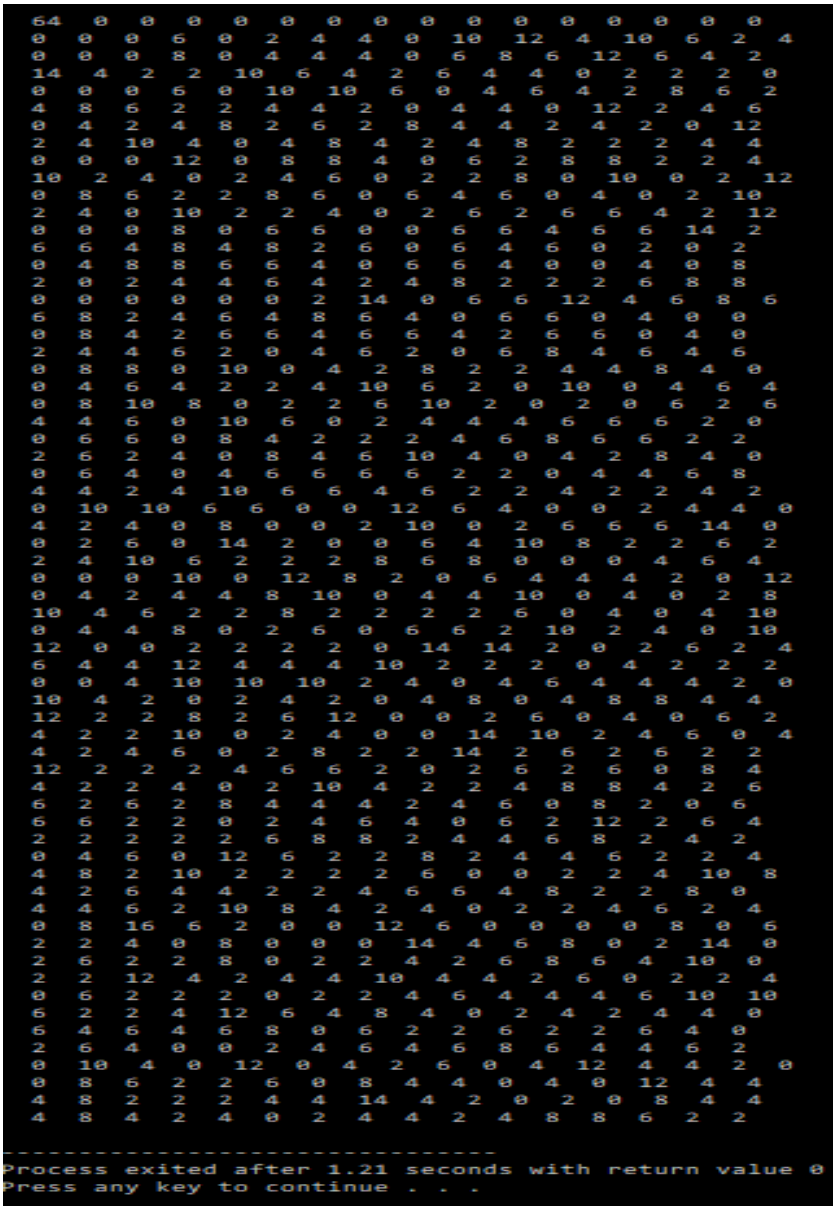


Figure 3.10: C++ DDT experiment with 6 x 4 DES S-Box

of substances shown on a 6 x 4 S-Box Difference-Distribution Table is 1024. One thousand twenty-four machines or computers can easily handle 1024 bytes.

The primary substance of the Difference-Distribution Table of 8 x 8 S-Box was integer 256, (2^8) because S-Box required eight bits as the most elevated parameter. Refer to Figure 3.14 256 is a two-byte word represented in binary as 0000000100000000. If each 8 x 8 S-Box Difference-Distribution Table substance is treated as a word, the memory needed to construct an 8 x 8 S-Box Difference-Distribution Table is 16 bits x 65536 = 131072 bytes. The number of substances shown on an 8 x 8 S-Box Difference-Distribution Table is 65536. A machine or computer can handle 131072 bytes.

The study predicted that the Difference-Distribution Table of 8 x 32 S-Box would have the primary integer as 4294967296, which is (2^{32}) because S-Box required thirty-two bits as the most critical parameter. 4294967296 is a triple-word made up of five bytes that are represented in binary as 00000000100000000000000000000000.

If each 8 x 32 S-Box Difference-Distribution Table substance were treated as a triple-word, the memory required to build an 8 x 32 S-Box Difference-Distribution Table would be 40 bits x 1099511627776 = 5497558138880 bytes. The anticipated number of substances shown on an 8 x 32 S-Box Difference-Distribution Table was 1099511627776. A machine or computer does not appear to be capable of handling a computation memory of 5497558138880 bytes of each substance with ease.

As a result, the 8 x 32 S-Box C++ Difference-Distribution Table program was smashed before execution. All of the discoveries are listed in Table 3.4, 3.5, and 3.7

The comparison of the findings was illustrated graphically in Figure 3.11, 3.12 and 3.13.

The Difference-Distribution Table of an AES S-Box was a 2^8 rows x 2^8 columns table with a high probability of determining a key. To create the

Table 3.4: The End Product of Creating a Difference-Distribution Table (DDT)

Size of S-Box	**Time Taken (in Seconds) to Create Difference-Distribution Table (DDT)**	**Number of entities required**	**Memory (in bytes) needed**
4 x 4	0.2815	256	256
6 x 4	1.2100	1024	1024
8 x 8	23.6800	65536	131073
8 x 8	∞	1099511627776	5497558138880

Table 3.5: The feasibility of creating a Difference-Distribution Table before and after applying a Novel Approach of using a KDM function and 32-bit S-Boxes

Name of Algorithms	**Before a Novel Approach of using a KDM function and 32-bits S-Boxes was Applied**	**After a Novel Approach of using a KDM function and 32-bits S-Boxes was Applied**
AES	Construction of Difference-Distribution Table was feasible.	Construction of Difference-Distribution Table was infeasible due to memory limitation of a computer.

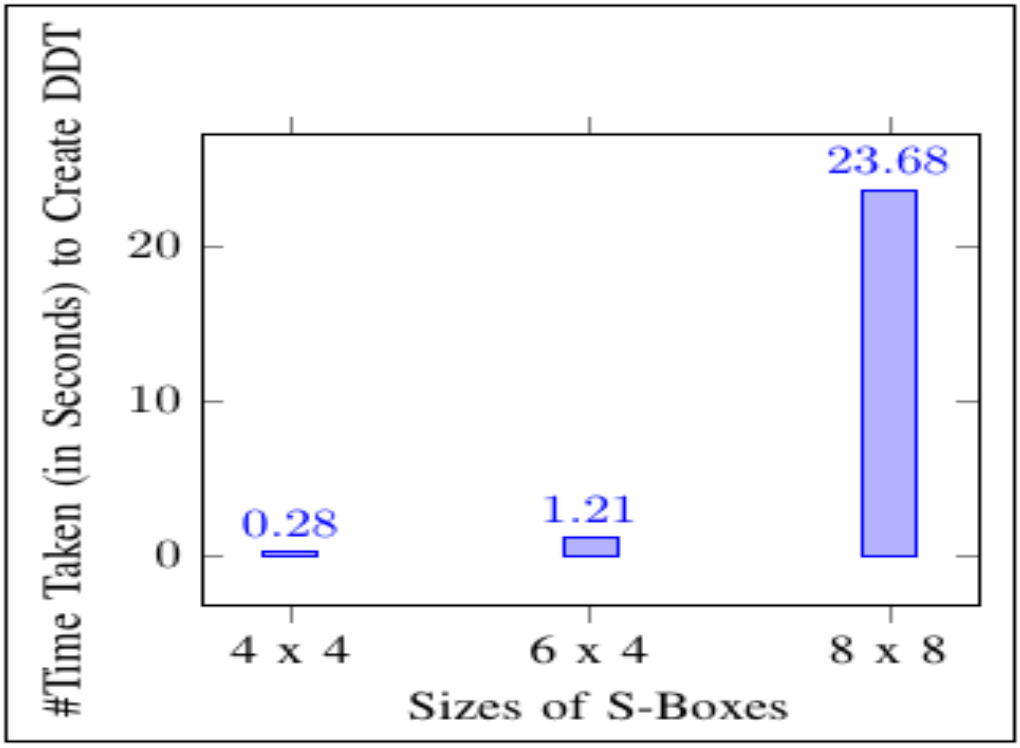


Figure 3.11: DDT Experimental Time Required

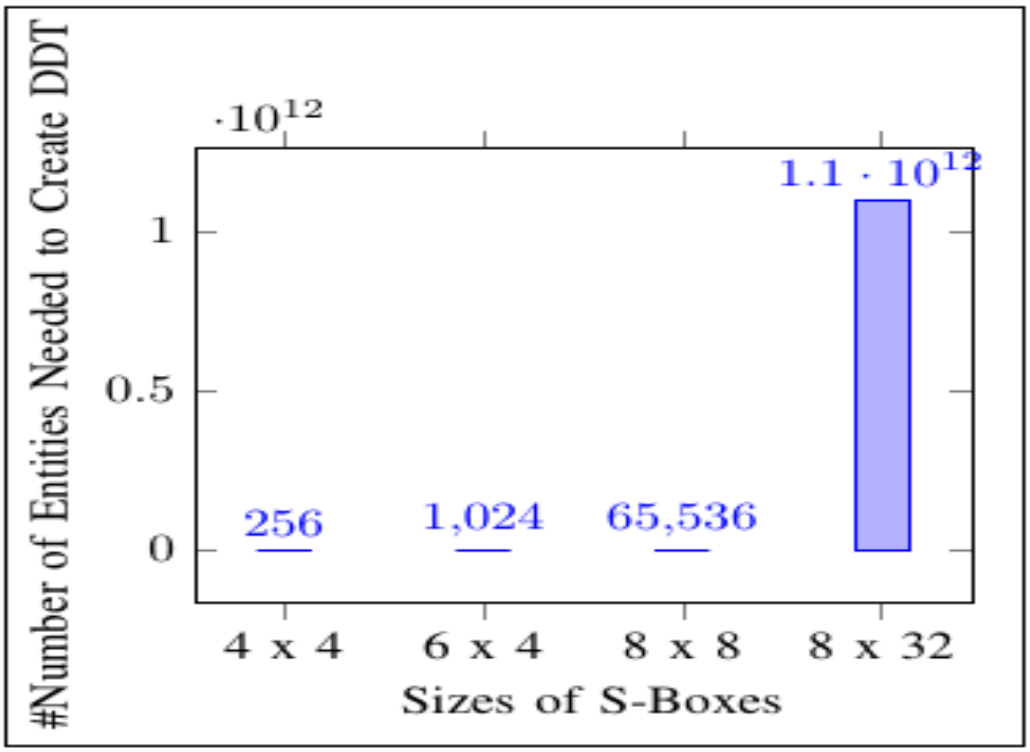


Figure 3.12: Number of Entities to Create the Experimental DDT

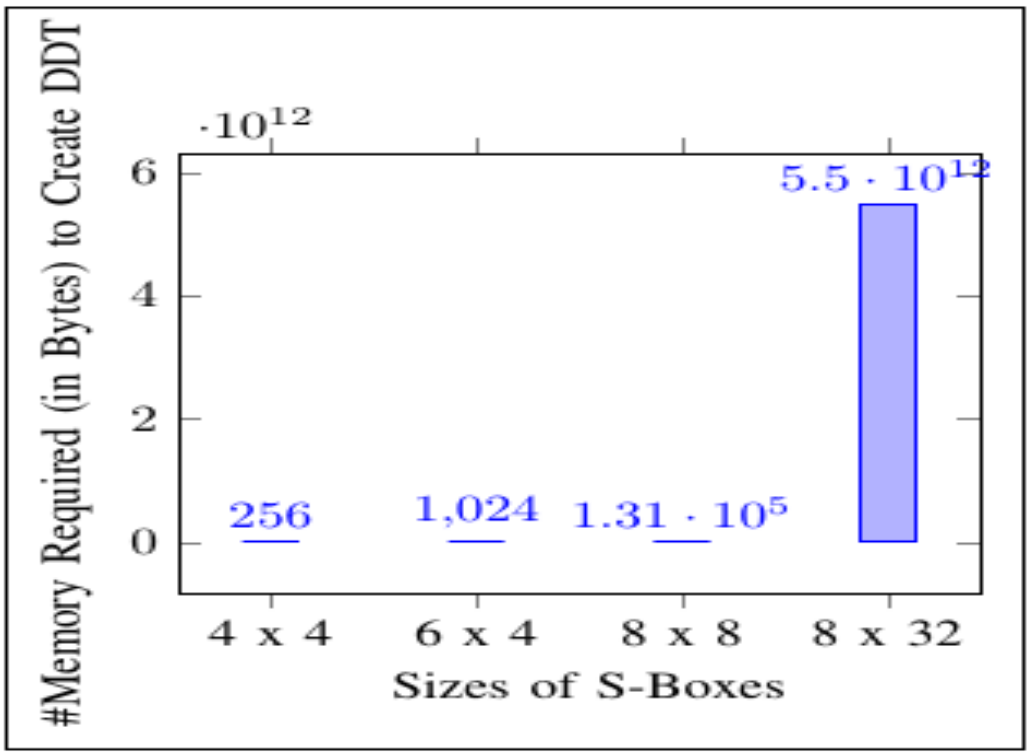


Figure 3.13: Memory Required to Create Experimental DDT

Difference-Distribution Table of the 8 x 8 AES S-Box, a C++ program was written. After investigating the strategy, it was determined that using the Difference-Distribution Table to attack an AES algorithm was feasible. To anticipate the DC attack on a new 32-Bit S-Box, recently produced 32 yield bits S-Boxes were used on an AES found on IoT gadgets.

Furthermore, the novel approach of converting 8 x 8 S-Boxes to 8 x 32 S-Boxes effectively fragments DC attacks on an AES algorithm. In this study, a KDM function is scientifically created, coined, and used on purpose. A KDM function was never recently delivered, characterized, or used by any analyst except in this case. Because it contained numerous scientific modulo operators, a KDM function was used to create a new 32-Bit S-Box suitable for the modern AES Algorithm. Furthermore, the majority of numerical modulo operators were irreversible.

The Difference-Distribution Table of 8 x 32 AES S-Box was created using a C++ program. The code crashed before constructing a Difference-Distribution Table of a new S-Box, which was supposed to be a 2^8 x 2^{32} matrix. Because a computer has limited memory compared to the desired memory to develop a Difference-Distribution Table of a new S-Box, it was impossible to form a Difference-Distribution Table of a new 8 x 32 AES S-Box with a yield of 32-bits.

The preliminary trial was to use a cluster of size $2^{32} = 4294967296$; however, it appeared that input $2^8 = 256$ had to be included. Due to computer memory constraints avoided creating a Difference-Distribution Table of 8 x 32 AES S-Box due to computer memory constraints. The program for creating a Difference-Distribution Table of a new S-Box fizzled before completion due to the amount of memory required by a computer to run, display, and execute a 256 x 4294967296 matrix. The calculation of 2^{32} x 256 required more than 2^{64} memory allocation, which is impractical when using a computer. Due to memory limitations distributed on a computer, the investigation concluded that it was illogical to form a table or network of 256 x 4294967296.

Memory boundaries were 2^{64} in Microsoft (Hp) and Mac (Apple) computers, causing Difference-Distribution Table problems for the DC attack. It was unreasonable to induce probabilities of calculating a key of 32-bits yielding S-Box. As a result, the anticipated Difference-Distribution Table development using 32-bits recently yielded S-Boxes and a KDM function, which were numerically created, coined, and purposefully used in this study.

Except for this ponder, no analyst has recently built, characterized, or used a KDM function. A KDM was created to test the validity of newly created 32-Bits-output S-Boxes within the recently altered AES algorithm. The study employs a KDM function to generate a new 32-Bit S-Box appropriate for the new Modified AES Algorithm and perplex the attacker because it contains numerous scientific modulo administrators. Furthermore, the vast majority of numerical modulo administrators are irreversible. Refer to Figure 3.6 for more learning, almost a KDM function.

It was discovered that there was no Difference-Distribution Table and thus no DC attack. As a result, this study increases the assurance of an AES against a DC attack.

The C++ executable record of a Difference-Distribution Table described in Table 3.3 was also performed in Figure 3.15 to confirm that all tactics of the DC assault utilizing the Difference-Distribution Table were carried out.

Simplified-DES and AES were the targets of the experimental DC attack by the consider. The investigation in this paper provides clarifications in the form of discrete rounds of a sensible DC attack. Repeating the same preparation for each round of an all-out assault makes up the remaining components.

3.3.1 Results of DC attack on Simplified-DES

The calculation would have eliminated the *key* necessary if it had used the distinction of a ciphertext combination of ciphertext, leaving us with no knowledge of the *key*: $ciphertext_a = plaintext_a \oplus key$ The calcula-

tion would have eliminated the *key* necessary if it had used the distinction of a ciphertext combination, leaving us with no knowledge of the *key*: $ciphertext_a \oplus ciphertext_b = plaintext_a \oplus key \oplus plaintext_b \oplus key$ $ciphertext_a \oplus ciphertext_b = plaintext_a \oplus plaintext_b$.

The contrast between the plain and ciphertext appears to be the same when overworked.

Keep in mind that the simplified DES algorithm is not linear. In this way, the distinctions between plaintext and ciphertext differ from one another. In simplified DES, the *key* value determines the distinction in a plaintext match for a certain distinction in a ciphertext combination. From the Difference-Distribution Table given in Table 3.3, $plaintext_a \oplus plaintext_b = \Xi\wp$ $ciphertext_a \oplus ciphertext_b = \Xi\zeta$ The calculation obtained the output and input values from Table 3.3 under the guidance of the Difference-Distribution Table. On occasion, when $\Xi\wp = 12$ and $\Xi\zeta = 3$, the conceivable of *key* occurence 2. That is $\Xi\wp = 6 \oplus 10$ or $\Xi\wp = 10 \oplus 6$. There are potentially two input sets: $(6, 10)$ and $(10, 6)$. Consider input combining $(6, 10)$, then $plaintext\_a = 6$, $plaintext\_b = 10$ and assume then $ciphertext\_1 = 3$ and $ciphertext_b = 0$, therefore $\Xi\zeta = 3$. If the input difference of 4 x 4 S-Box is denoted by $H = H_a \oplus H_b$, let's assume that $H_a = plaintext_a \oplus key$ and $H_b = plaintext_b \oplus key$. From the above inquiry, it appears that the *key* has no bearing on the input contrast esteem because it is the same constant esteem, so: $\Xi\wp = H = 6 \oplus 10 = 12$, meaning $H = 12 = 4 \oplus 8$ if $\Xi\zeta$ is accepted to be the result of utilizing the Difference-Distribution table. $H = H_a \oplus H_b = 4 \oplus 8) = 12$.

$key = H \oplus \Xi\wp$. Therefore, $key = H_a \oplus plaintext_a$ and $key = H_a \oplus plaintext_b$. Subtituting the values $key = H_a \oplus plaintext_a = 4 \oplus 6 = 2$, and $key = H_a \oplus plaintext_b = 4 \oplus 10 = 14$. Alternatively $key = H \oplus \Xi\wp$. Therefore, $key = H_b \oplus plaintext_a$ and $key = H_b \oplus plaintext_b$. Subtituting the values $key = H_b \oplus plaintext_a = 8 \oplus 6 = 14$, and $key = H_a \oplus plaintext_b = 8 \oplus 10 = 2$. In this manner, two conceivable *key* values are found, namely

2 and 4. Each key is tested to give the value of $\Xi\zeta$, the one that gives the same esteem of combine is the correct key. In this case, 2 is the proper tried key. Therefore $key = 2$. With this data, the ponder affirmed that the Simplified-DES is crackable utilizing a DC attack. The DC attack oversaw the break of both rounds of Simplified DES, utilizing a ciphertext pair of 2^{10} with a time complexity of 2^{16}. At that point, the same method was utilized on DES. Allude to Table 3.6 and Figure 3.16.

3.3.2 Results of DC attack on DES

The study utilized an input combining $\Xi\wp$ to DES S-Box as $(1, 35)$ where $\Xi\wp = Plaintext_1 \oplus Plaintext_2 = 1 \oplus 35$, therefore $\Xi\wp = 34$. Suppose $\Xi\zeta = D$. $\Xi\wp = 34$, regardless of the key value, because $H_a = Plaintext_1 \oplus key$ and $H_b = Plaintext_2 \oplus key$, therefore $H = H_a \oplus H_b$ $H = (Plaintext_1 \oplus key) \oplus (Plaintext_2 \oplus key)$ $H = Plaintext_1 \oplus Plaintext_2$ $H = \Xi\wp$. Also $H_a = \Xi\wp \oplus key$ and $key = H \oplus Delta\wp$. Utilizing the Difference-Distribution Table given in Figure 3.10, the conceivable key occurance is 8, which are $\{07, 11, 17, 1D, 23, 25, 29, 33\}$.

In case the same method was rehashed when input combines $\Xi\wp$ to DES S-Box as $(21, 15)$, but still keeps $\Xi\wp = 34$ since $21 \oplus 15 = 34$, and changes $\Xi\zeta = 3$ rather than utilizing $\Xi\zeta = D$. Utilizing the Difference-Distribution Table given in Figure 3.10, the possible key occurence is 6, which are $\{00, 14, 17, 20, 23, 34\}$. The accurate key esteem ought to be visible in both of these bunches: $\{07, 11, 17, 1D, 23, 25, 29, 33\}$ and $\{00, 14, 17, 20, 23, 34\}$ which $\{17, 23\}$ either 17 or 23 is the right key value. Each key is tried to provide the esteem of $\Xi\zeta$, the one that gives the same esteem of match is the proper key. In this case, 17 is the correct tried key. The DC attack managed to break all 16 rounds of DES utilizing a ciphertext pair of 2^{14} with a time complexity of 2^{58}. At that point, the same method was utilized on AES. Allude to Table 3.6 and Figure 3.16.

3.3.3 Results of DC attack on AES

The thinker makes a comparison in a single byte. Eight bits make up a byte. Then, $2^8 = 256$ potential ciphertexts should be generated. The final subkey can be verified using the Difference-Distribution Table theories shown in Figure 3.14 once all 256 possible ciphertexts have been generated. By examining the conditions of the final subkey byte-by-byte, theories are tested. The investigation examined whether the inputs added together increased the final circular to zero. The calculation gives the specific subkey back.

Given that an arbitrary conveyance has an input-combined break-even with a probability of 1/256 from the Difference-Distribution Table theories given in Figure 3.14, the study also expects one additional off-base hypothesis byte-by-byte.

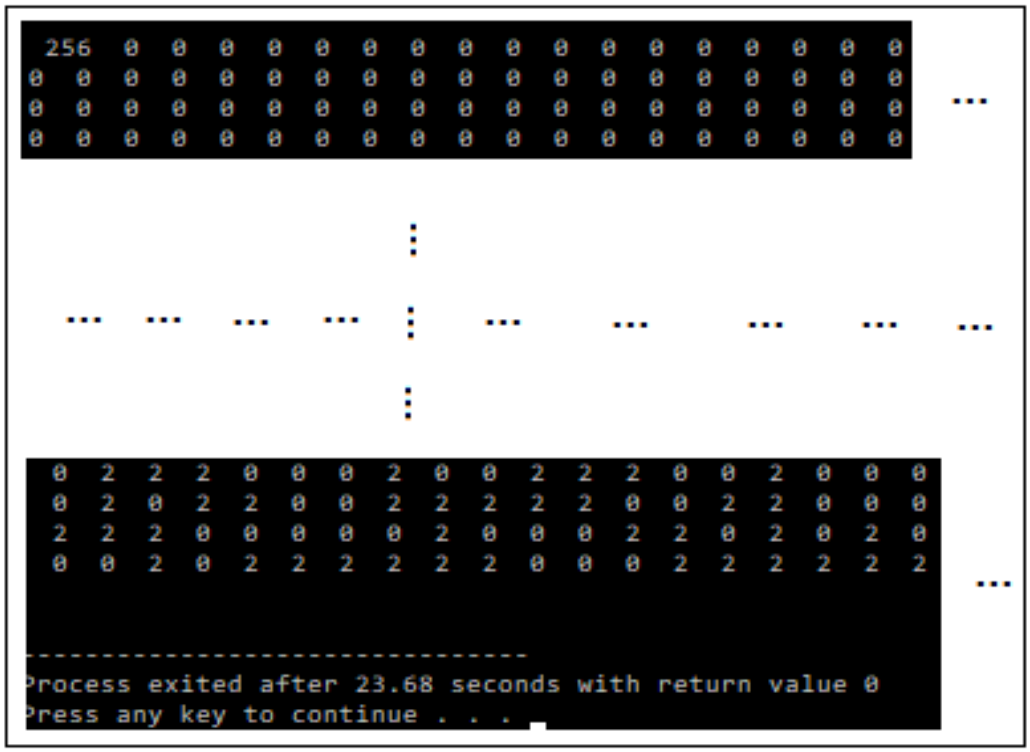


Figure 3.14: C++ DDT experiment with 8 x 8 AES S-Box

The investigation led to an anticipated increase in the number of key hypotheses for the final subkey of 2^{16}. The DC attack used a 2^{92} ciphertext combination with a 2^{186} time complexity and could decipher seven out of ten rounds. Make references to the Table 3.6 and Figure 3.16.

3.3.4 Results of DC attack on M_AES

Due to memory restrictions on those machines and computers, M_AES used a new 32-bit S-box that failed to execute the C++ Difference-Distribution Table from those multiple machines and computers. The Difference-Distribution Table of the 2^8 x 2^{32} = 256 x 4294967296 matrix, which is predicted to contain 1099511627776 entities, does not appear to be computed by any machine or computer. It would have been absurd to attempt the DC attack on the recently created 8 x 32 S-Box of the M_AES algorithm shown in Appendix A, Figure A.1 without the Difference-Distribution Table. Due to a new 32-bit yield S-Box that prevented the construction of the Difference-Distribution Table due to machine memory restrictions, no round out of 16 was broken using the DC attack. Make references to the Table 3.6 and Figure 3.16.

Table 3.6: DC Attack Outcomes

Name of Algorithm	**Time Complexity**	**Ciphertext Pairs**	**Number of Round in %**
Simplified DES (S-DES)	2^{16}	2^{10}	2 out 2 or 100%
DES	2^{58}	2^{14}	16 out 16 or 100%
AES	2^{186}	2^{92}	7 out 10 or 70%
M_AES	∞	∞	0 out 10 or 0%

The possible creation of Table 3.3 was evaluated and composed in workable C++ code for approval, testing, and validation. Refer to Figure 3.15

and Table 3.3. Figure 3.15 and Table 3.3 had the same probability components. The test Difference-Distribution Table was executed by running a C++ Difference-Distribution Table code, and Figure 3.15 was the theoretical Difference-Distribution Table. That was done to verify and validate that creating a Difference-Distribution Table was done using all of the DC attack's tactics against an AES.

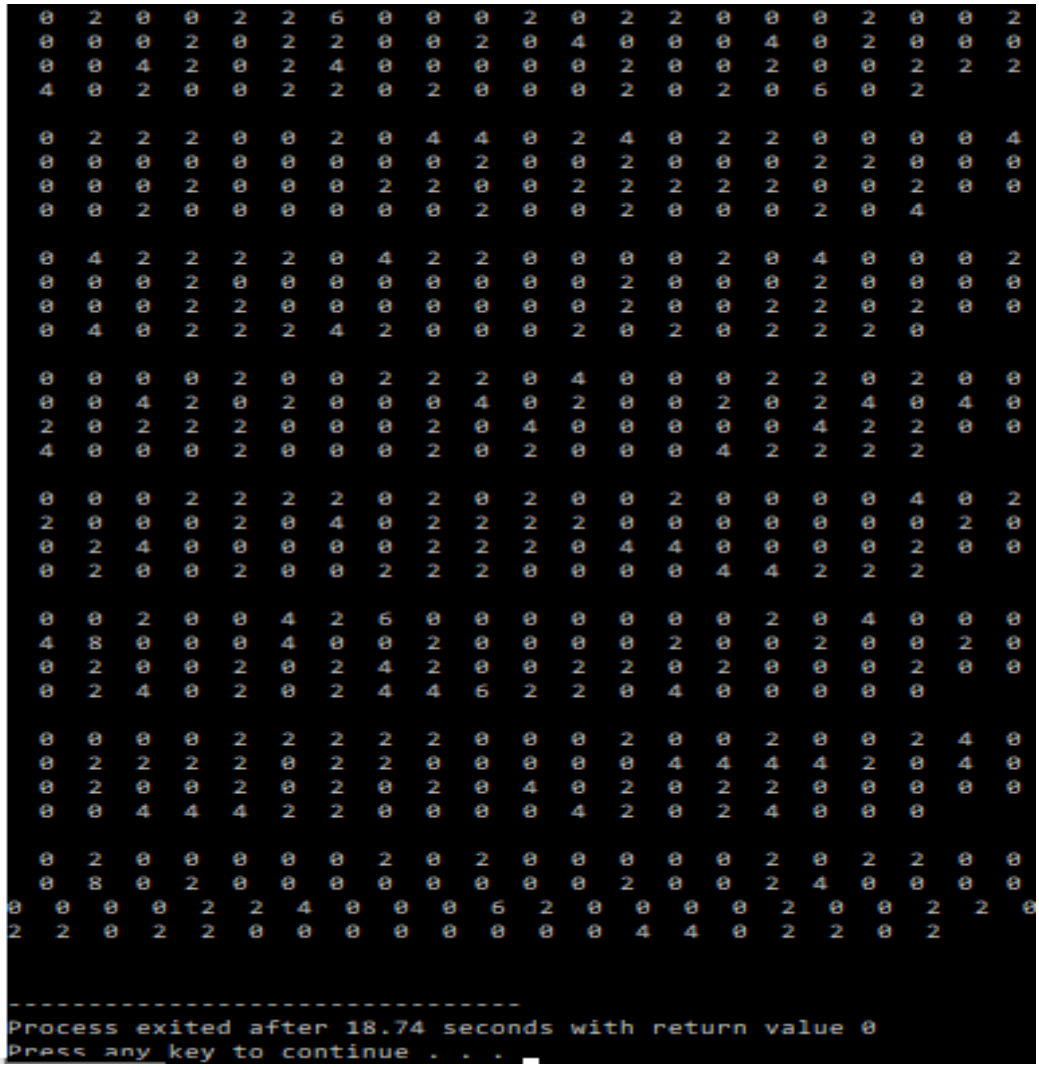


Figure 3.15: C++ DDT of 4 x 4 Simplified DES experiment

To determine whether the DC attack was feasible, the code was connected to AES and M_AES. Table 3.5 and Table 3.7 contain all the discoveries. Table 3.5 occurred throughout the Difference-Distribution Table's growth, and S-Boxes were connected after a novel approach that used 32 bits.

The study used a KDM function to build a new 32-Bit S-Box that is appropriate for the new Modified AES Algorithm and confuses the attacker because it has many numerical modulo operators. Additionally, the majority of scientific modulo operators are irreversible. Table 3.7 arose as a result of the discovery of crucial bits following a novel approach that connected 32-bit S-Boxes and a KDM function.

In this investigation, M_AES was created utilizing the new 8 x 32 S-Boxes and was resistant to a DC assault. The developer used a KDM function to create a new 32-Bit S-Box suitable for the new Modified AES Algorithm. It would confuse the attacker since it has many numerical modulo operators. Additionally, the majority of modulo operators are irreversible. The new 8 x 32 S-Boxes and KDM function were used to unscramble and scramble the new M_AES successfully. Ask makes it possible to access the latest M_AES code.

Before using a KDM function and the new 8 x 32 S-Boxes, it appeared from the C++ code that the DC attack was feasible against a standard AES on numerous rounds. However, it emerged in C++ code that the DC attack on M_AES was successfully averted after using a KDM function and the unique 8 x 32 S-Boxes. The limited memory of a computer made it challenging to construct a 2^{32} rows and columns Difference-Distribution Table architecture. Contain all of the discoveries. Figure 3.11, 3.12 and 3.13 were used to illustrate the comparison of the discoveries.

The Avalanche Effect is an acceptable attribute of algorithms in cryptography [169]. The yield bits must change if one input bit is modified (flipped). In robust algorithms, such a slight change to the plaintext or key should result in an extreme contrast in the ciphertext [169].

Table 3.7: Results of key bits finding before and a Novel Approach of employing a KDM function and 32-bits S-Boxes was applied

Name of Algorithms	**Before a Novel Approach of using a KDM function and 32-bits S-Boxes was Applied**	**After a Novel Approach of using a KDM function and 32-bits S-Boxes was Applied**
AES	The key was discovered in many round.	No key bits were discovered or detected in all rounds of an AES.

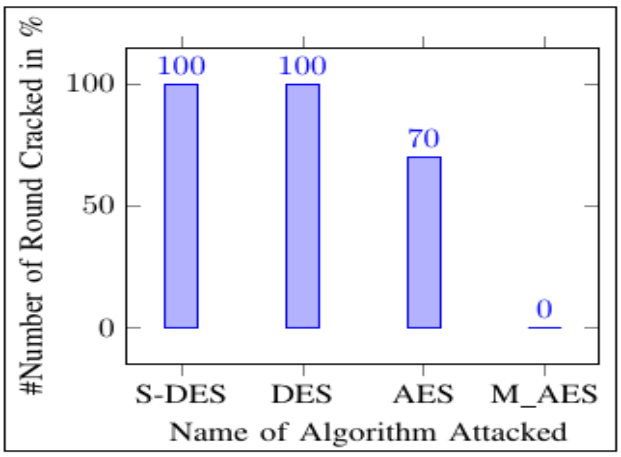


Figure 3.16: Number of Rounds Cracked during the Experimental Diferential Crypatanalysis Attack

The Strict Avalanche Criterion (SAC), a method that advances the Avalanche Effect, is used to evaluate the encryption strength of the algorithm [170].

If a single input bit—from either the plaintext or the key changes the ciphertext with a probability of 50 percent or greater, the SAC is considered satisfied. This work applied the Avalanche Effect to S-DES, DES, AES, and M_AES to produce SAC. Since the Avalanche Effect of M_AES on both the key and plaintext was about 50 percent probability compared to S-DES and DES, it appears that the AES and a newly produced M_AES method had a far better SAC property than S-DES and DES. Refer to Table 3.8 and Figure 3.17 to Figure 3.25.

The same image could be encrypted and decrypted using all methods (S-DES, DES, AES, or M_AES). However, encrypted images were not identical. Please see Figure 3.27.

Table 3.8: Avalanche Effect of Key and Plaintext Bit was Flipped

Name of Algorithm	**Plaintext Avalanche Effect in Percentage**	**Key Avalanche Effect in Percentage**
Simplified DES (S-DES)	25	25
DES	60.4003	44.2138
AES	50.0488	50.2807
M_AES	49.9023	50.2807

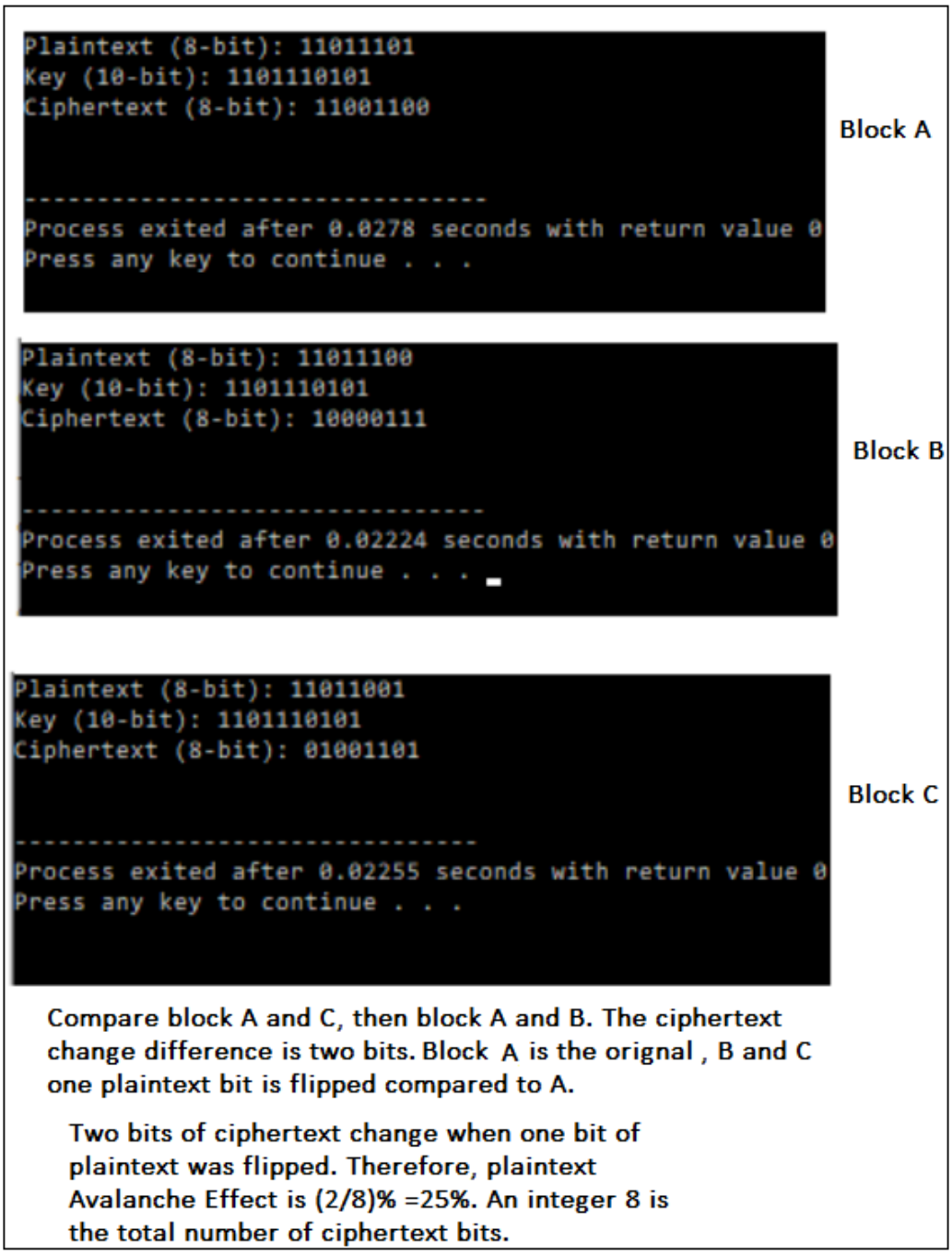


Figure 3.17: S-DES Plaintext Avalanche Effect in Experimental Results

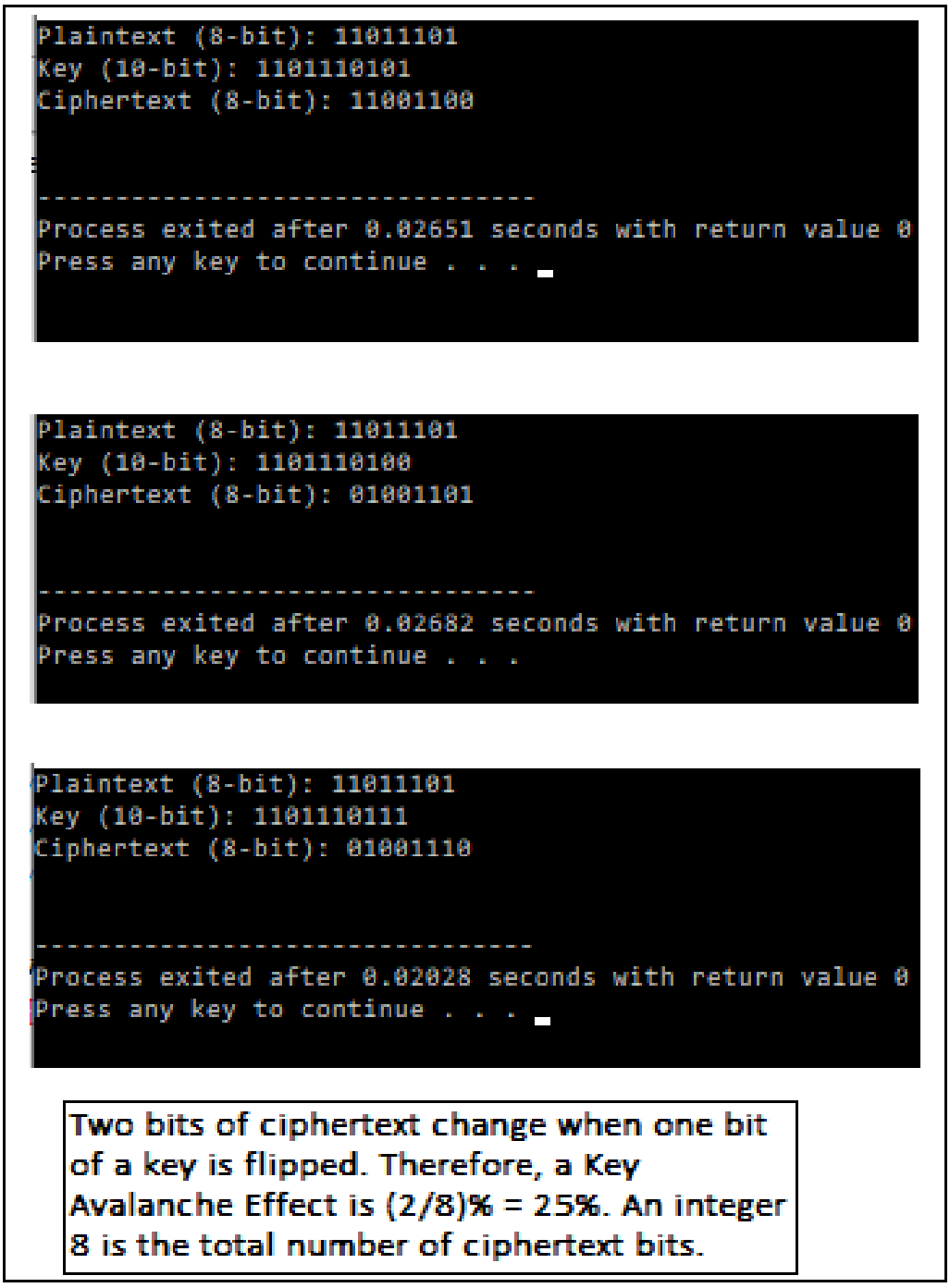


Figure 3.18: S-DES Key Avalanche Effect Experimental Results

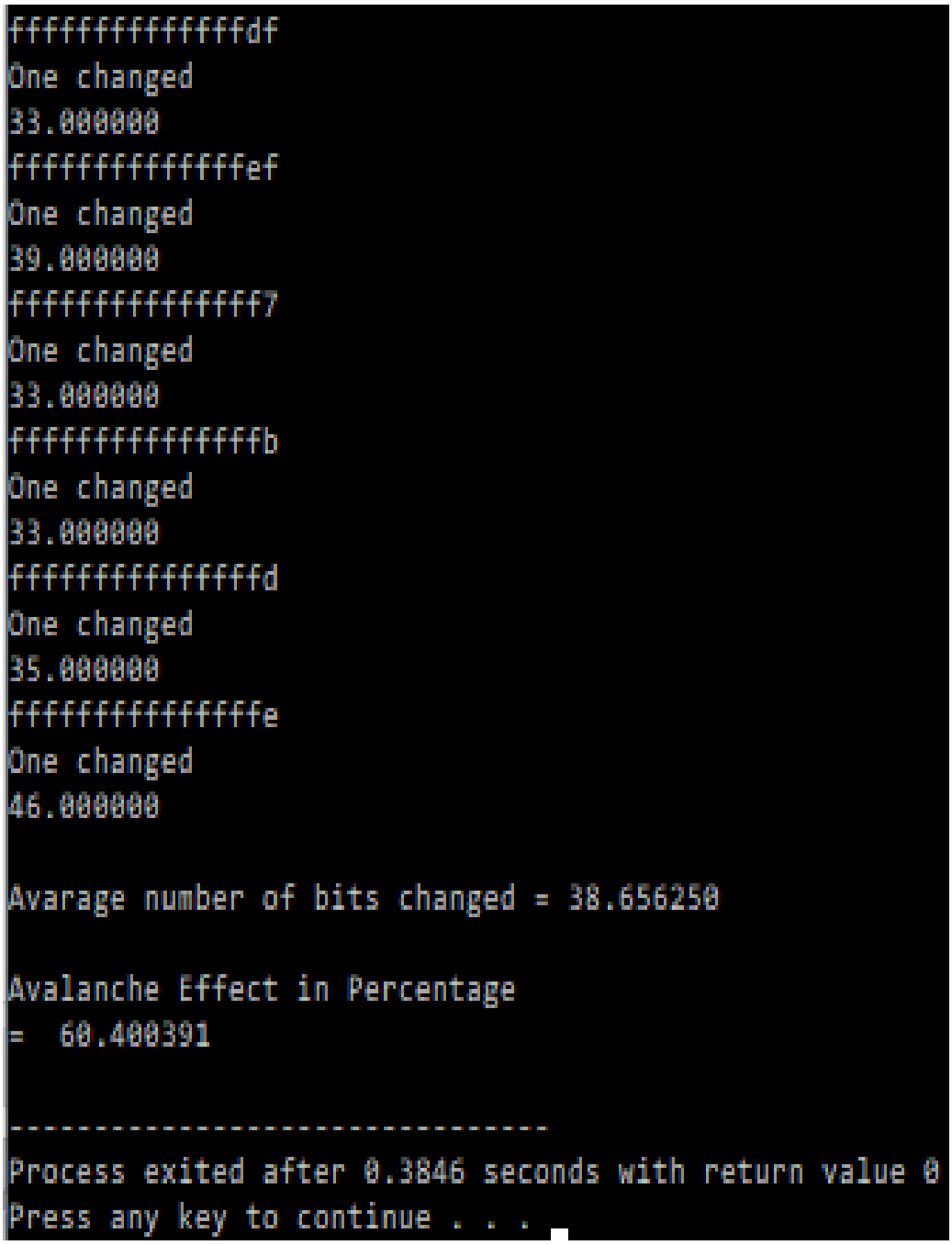


Figure 3.19: DES Plaintext Avalanche Effect Experimental Results

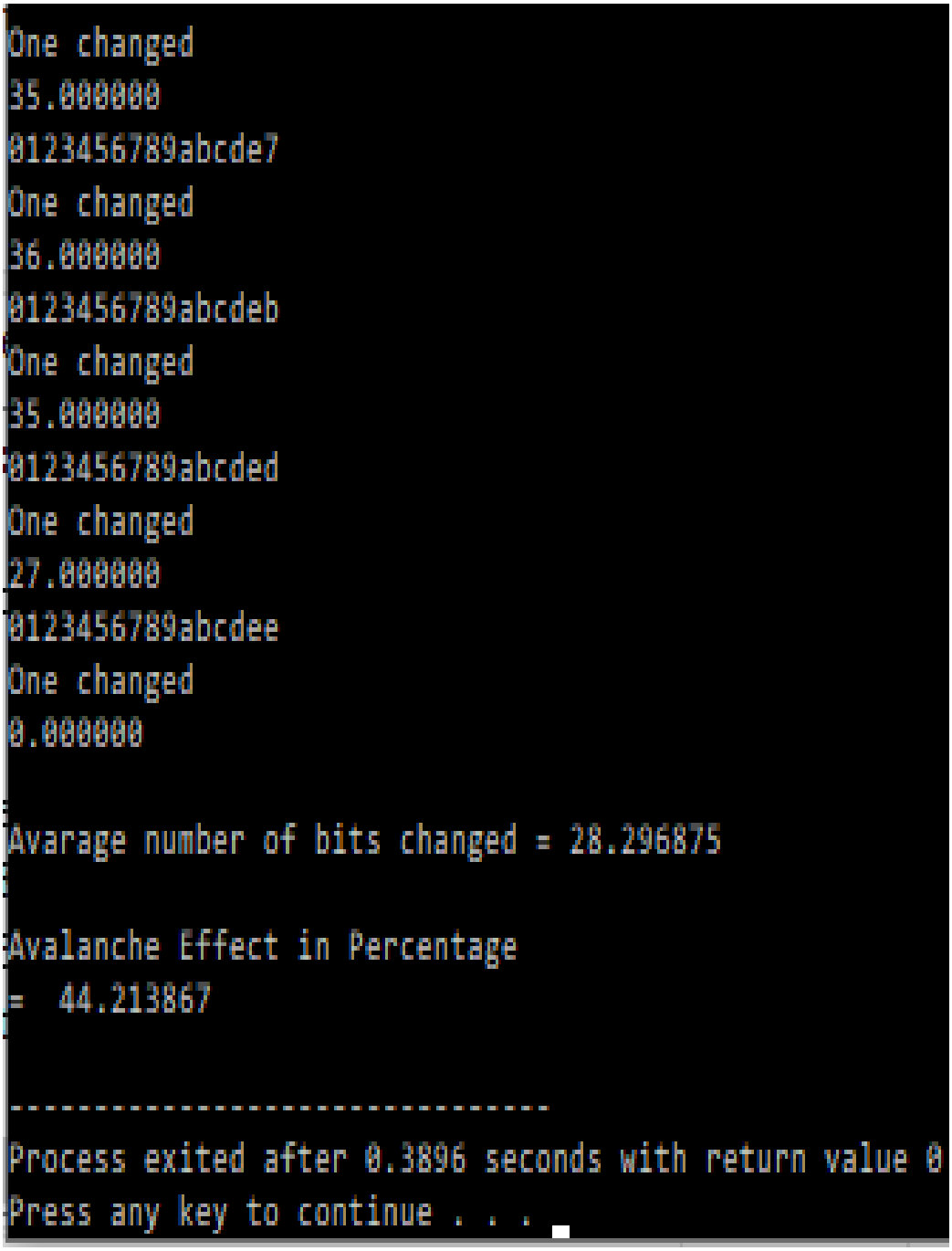


Figure 3.20: DES Key Avalanche Effect Experimental Results

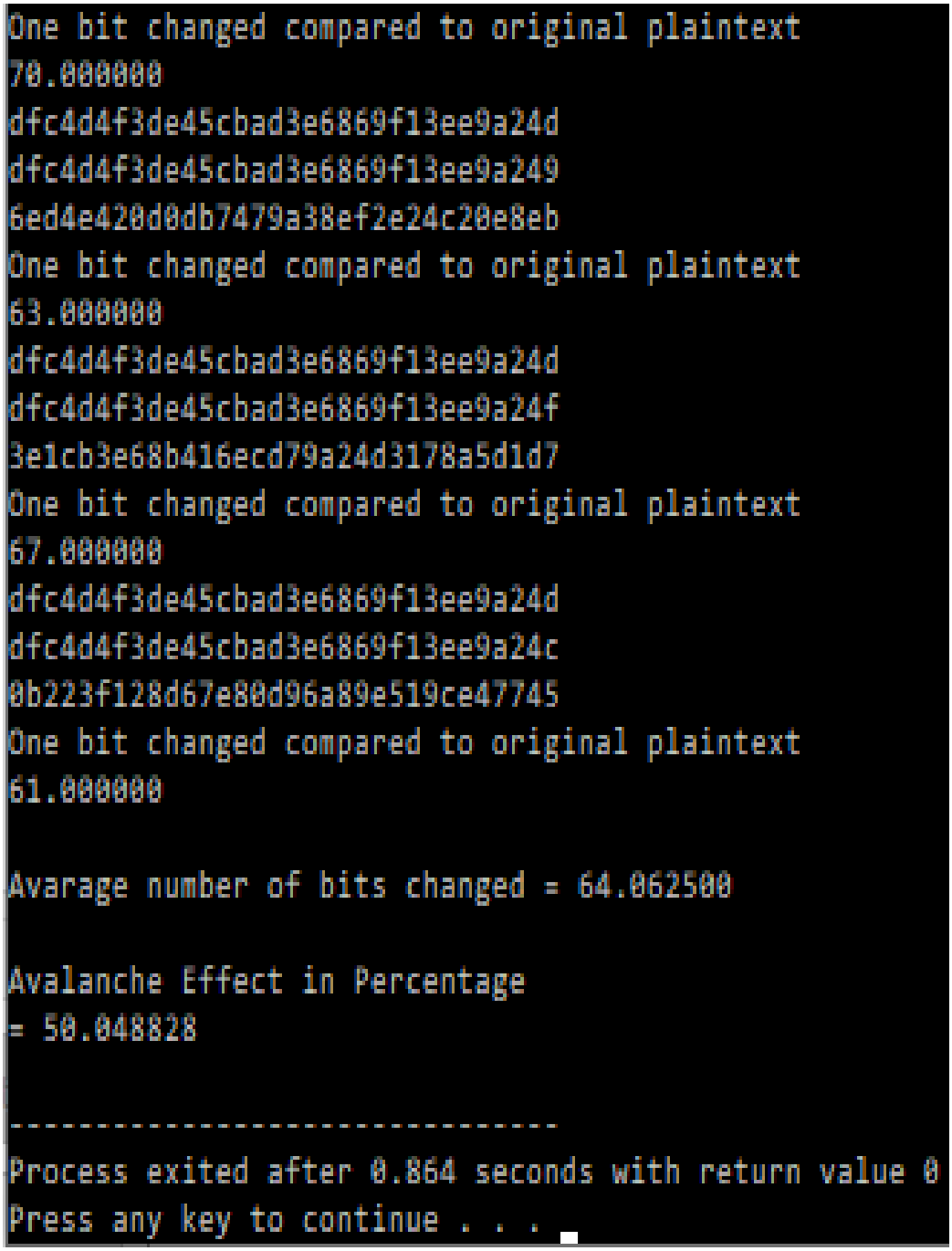


Figure 3.21: AES Plaintext Avalanche Effect Experimental Results

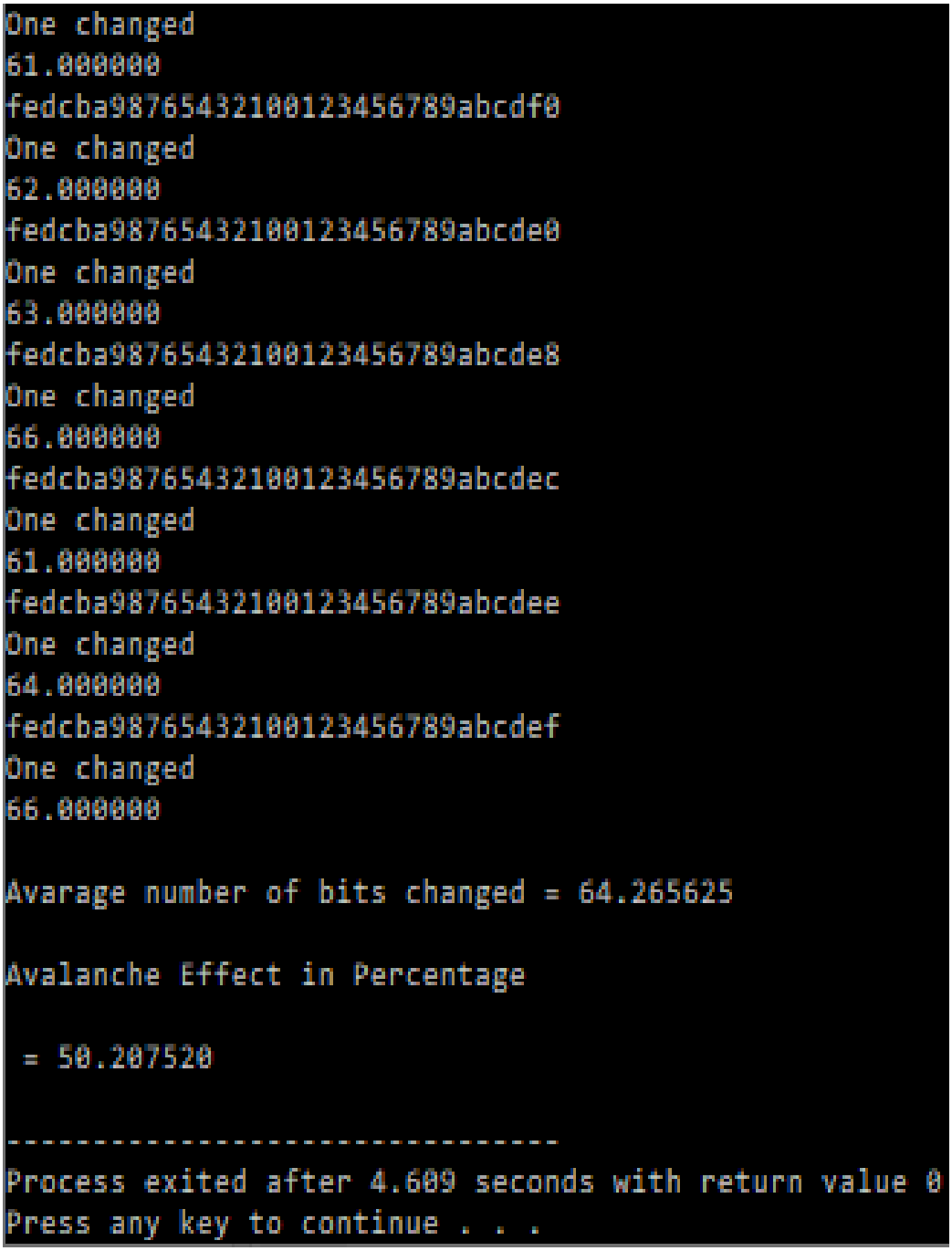


Figure 3.22: AES Key Avalanche Effect Experimental Results

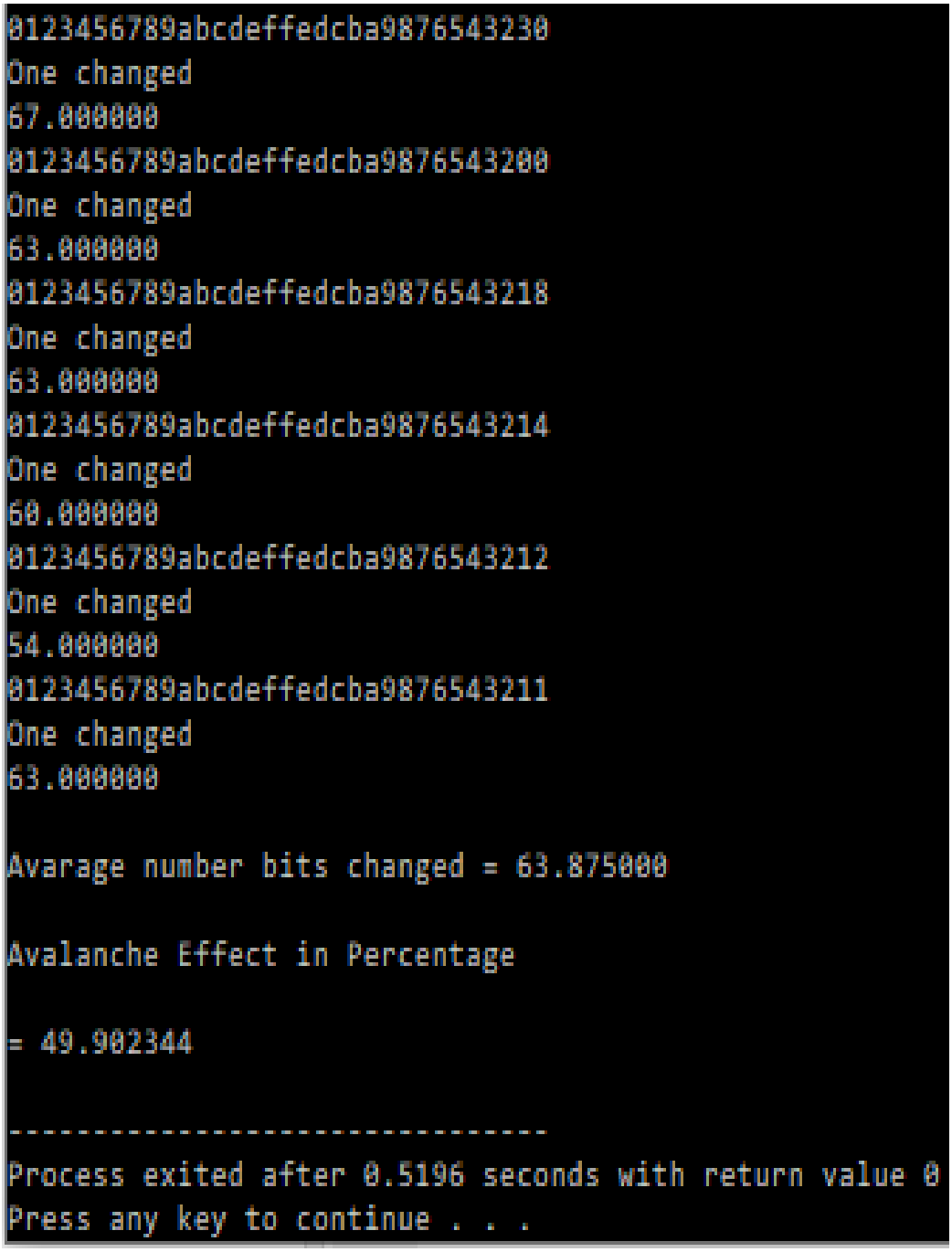


Figure 3.23: M_AES Plaintext Avalanche Effect Experimental Results

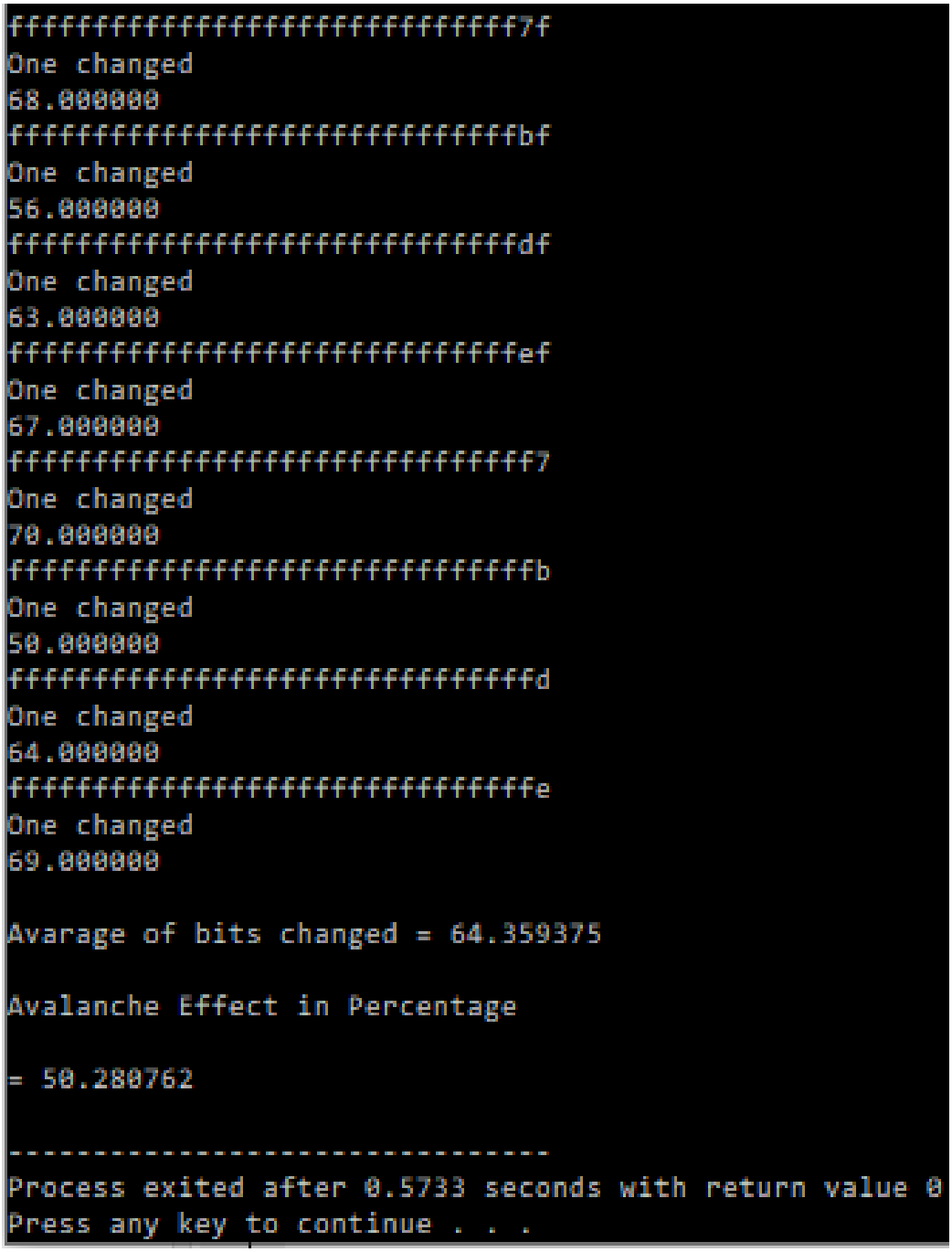


Figure 3.24: M_AES Experimental Key Avalanche Effect

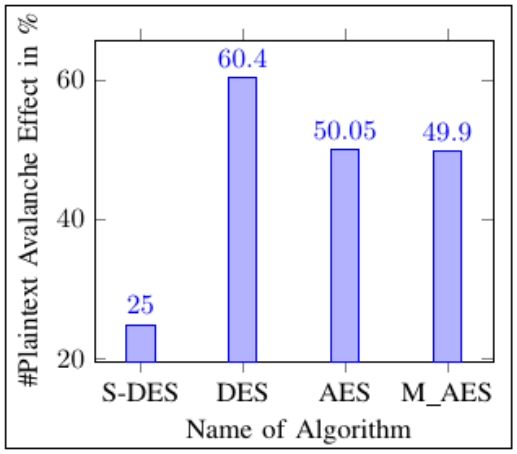


Figure 3.25: Plaintext Avalanche Effect Experimental Analysis in Percentage

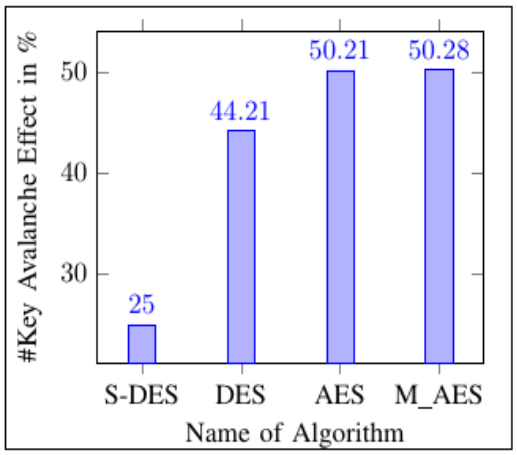


Figure 3.26: Key Avalanche Effect Experimental Analysis in Percentage

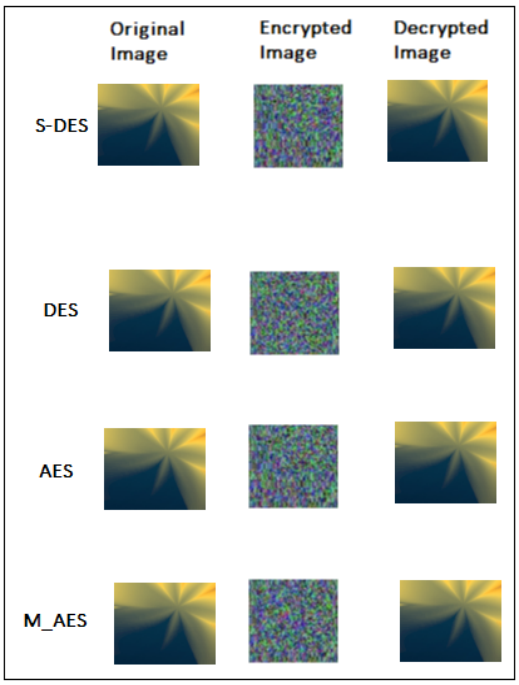


Figure 3.27: All Algorithm Image Encryption

3.4 Summary of Preventing DC Attacks Using a KDM Function on AES

The study has confirmed that creating a Difference-Distribution Table of 32-bits produce S-Box is not feasible. It has been suggested that there is no DC attack and no Difference-Distribution Table. Therefore, they concluded that the unique method of using a KDM function for the logic of repeatedly producing 32 bits yields an S-Box, and communication on an AES may foresee the DC attack.

A new 32-Bit S-Box that is appropriate for the new Modified AES Algorithm and confuses the attacker since it has a lot of numerical modulo operators was created by the study using a KDM function. Additionally, the majority of scientific modulo operators are irreversible. The cutting-edge technique can securely encrypt any private data sent and stored by the Internet of Things devices.

Chapter 4

The Design of Blocker Function to Prevent Differential-Linear Attacks on the Serpent

The chapter that follows is based on published work by:

i. K. D. Muthavhine and M. Sumbwanyambe, "Securing IoT Devices against Differential-Linear (DL) Attack used on Serpent algorithm", MDPI, Future Internet, pp. 1-34, 2022.

   Website: *https://www.mdpi.com/1999-5903/14/2/55*

   Status: Published.

   Publisher: MDPI.

ii. K. D. Muthavhine and M. Sumbwanyambe, "Modifying Cast algorithm in order to Increase Encryption Strength and to Reduce Memory Limitations," IEEE, pp. 1-7, 2021.

   Website: *https://ieeexplore.ieee.org/document/9519349*

   Status: Published.

   Publisher: IEEE.

Abstract: The Differential-Linear Attack (DL) attack was created by Langford and Hellman [97] - [98]. Calculating the DL characteristic's probability produces a Differential-Linear Connectivity Table (DLCT), which is necessary for DL attack [97], [98]. The DLCT is a probability table that offers many opportunities for an attacker to deduce the cryptographic keys for any algorithm, including Serpent, found on IoT devices [97]. To attack algorithms, an attacker first builds DLCT utilizing building blocks like Substitution-Boxes (S-Boxes), which are common in the structures of many algorithms [98]. This study aims to protect IoT devices from DL attacks that target the Serpent algorithm using three magic numbers that are mapped onto a newly created mathematical function called Blocker. The Blocker is incorporated into Serpent's infrastructure before being placed on IoT devices. To replace the original Serpent S-Boxes with 4-Bits-output, new S-Boxes with 32-Bits-output were created for this study. The architecture of Serpent also included the innovative S-Boxes. The Blocker function and magic numbers, were influential in this investigation. The findings indicate that an algorithm with an S-Box made up of 4-Bits-output is more prone to attack than an algorithm with an S-Box made up of 32-Bits-output. Three magic numbers and 32-bit output S-Boxes were used in the study to create a novel blocking technique that prevented the development of DLCT and DL assaults. The innovative strategy successfully protected IoT devices' installed Serpent algorithms from DL attacks.

4.1 Background of Securing IoT Devices against DL Attack used on Serpent algorithm

Unaware of it or not, the IoT has significantly disrupted people's lives in modern times, despite the non-technologically inclined who have started to make use of the services, comfort, support, and critical insights that contribute to [171], [172], [174]. Given the relationship between IoT devices and smart thermostats, home hubs, remote door locks, and numerous app-

controlled gadgets, it's likely that everyone is already aware of their importance in daily life [175].

In actuality, IoT is improving in quality for both daily use and manufacturing. It is improving people's lives in various ways and will continue to do so in the appropriate manner [176]. The concerns that people are aware of are that IoT has been resolving issues without understanding they were issues until the solution was performed in a miraculous way [174].

IoT enables users to work more intelligently, live more creatively, and have complete control over their lives [174], [175], [176]. IoT devices also provide confidential assistance for our welfare. In addition to customers' smart home appliances, IoT is a vital technology in business and industry since it gives companies a real-time view into their internal activities [175]. IoT provides insights into a wide range of processes, including machine production, supply chains, and logistics from the warehouse to the customer's door [174], [176].

Businesses can automate processes and spend less money on labor thanks to IoT. Additionally, IoT reduces waste, improves service delivery, makes it less expensive to produce and transport products, and increases client transaction transparency [175]. The IoT allows businesses to reduce costs, increase security, and improve quality throughout the process, resulting in a win scenario for both stakeholders and clients. As a result, manufacturing client assets is expensive, transportation is more common, and businesses can grow, energizing the administration and expressing the sense of security the customer can bring to the bank [175], [176]. Even though IoT helps the community and manufacturers, there are challenges associated with its deployments, such as concerns over privacy and the security of sensitive information.

The IoT poses privacy challenges similar to other digital technologies that generate and acquire data, particularly radio-frequency identification and cloud computing. IoT devices have proliferated, and users lack understanding of how to manage data [176].

Users should take into account the potential fiscal and social impact of a possible digital protection occurrence regarding the availability, integrity, or data confidentiality in the data operation while carrying out an action that depends on digital technologies, including the IoT [174]. These values have the potential to deplete resources (for example, by disrupting transactions), endanger reputation (for instance, by disclosing private information or causing website damage), or alter the business environment (for example, through deprivation of innovation) [174], [175], [176].

One of the most significant issues is privacy, which includes the methods through which one might get personal information. Users assume that the actions are spy-related due to the tracking, validating, and validating of devices and the collection of private information from various sources [175]. Furthermore, it is easy to recognize characters who have been abducted, lost, or have been in an accident [174], [175], [176]. It is also a hassle for anyone who needs to preserve their privacy [176].

One of the issues the IoT frequently faces is security, according to [174]. For secure localization in public spaces, transmitting them unsecured and sacrificing cyber-attacks, affordable and low-cost broadband connections and Wi-Fi capabilities in various devices are needed [174], [175]. To provide security, IoT distinguishes three key components: authentication, access control, and secrecy focused on IoT operations [176]. IoT enables consistent data sharing between similar devices. To safeguard data collected, used, stored, and transferred using IoT devices, a robust cryptographic technique is required [174].

The Serpent algorithm is one of the most frequently requested algorithms for IoT devices. A primary issue is a new tool called DLCT, which exploited Serpent to learn the covert encryption keys used by intruders to safeguard the data gathered and stored on IoT devices. The DL attack is the process of obtaining the key via DLCT [97], [98]. All data encrypted with Serpent on IoT devices can be easily accessible to attackers once they crack and find

the Serpent's key. This assault has the potential to take advantage of all IoT users and devices. A cryptographic technique like Serpent has a limited number of output bits (4-Bits), which can be found on the S-Boxes, making it simple for an adversary to exploit it [81].

If the DL attack is not considered correctly, it could compromise all IoT devices' security. Much hasn't been done to protect the Serpent algorithm from DL assaults. To make it challenging to build DLCT using the S-Boxes of Serpent algorithm, this study focuses on protecting Serpent from DL attack by creating a new additional function called Blocker, employing three magic numbers, and designing new 32-Bits-output S-Boxes. Analysis has shown that the DL attack begins with DLCT. Therefore, it is thought that preventing the building of DLCT will make a DL attack unfeasible.

Cryptographic algorithms are now used without hesitation by IoT devices to store and transmit confidential information [171] - [154]. While IoT device security is increasingly improving through robust cryptographic algorithms, more attackers develop various methods of attacking the notably powerful algorithms [154]. The Serpent is an algorithm created by Eli Biham and Lars Knudsen. Like any other algorithm, Serpent is examined to see if it is vulnerable to the DL attack. Langford and Hellman developed and tested the DL attack on Serpent [97] - [98].

This research focuses on securing the Serpent algorithm found on IoT devices against DL attacks by developing a new function called Blocker using three magic numbers. The first magic number is $Q = 4302746963$, the second is $P = 4559351687$, and the third is $M = 4294967296$, all of which are mapped on Blocker and will be inserted into Serpent's architecture. To replace the original 4-Bits-output S-Boxes, new 32-Bits-output S-Boxes were created. In this study, the newly developed 32-Bits-output, S-Boxes, and Blocker function successfully secured the Serpent algorithm by preventing the construction of a probability table called DLCT, which is used during the DL attack process.

This study also introduces a newly generated function called Blocker. A Blocker Function takes a 32-bit output value from S-Box and returns a new value *statehold* as an output. A Blocker Function also renders polynomials with P, M, and Q values unfactorizable. To prevent attackers from reverse engineering, random numbers and XOR operators are used to add complexity and confusion. The XOR operator and $rand()$ are used to change the values of variables within a Blocker Function.

When invaders reverse back a Blocker Function to calculate the exact information used in that situation, the random numbers and XOR operators also provide a problematic input range. To construct DLCT using any machine or computer, the values of M, P, and Q are constantly maintained as unfactorizable polynomial variables that are non-linear and difficult to reverse. Intruders can use random numbers and XOR operators to create hidden, unseen, and unchangeable variables.

A Blocker function generates a one-of-a-kind 32-Bit S-Box suitable for the new Magic Serpent Algorithm. Because it contains many mathematical random numbers and XOR operators, a Blocker Function distracts the attacker. Furthermore, the majority of mathematical XOR operators and random numbers are irreversible. Figure 4.4 contains additional mathematical characteristics of a Blocker function as well as C++ explanations.

4.1.1 Serpent Algorithm

The Serpent is a cryptographic algorithm, a block cipher that encrypts and decrypts a 128-Bit data block using different key sizes, such as 128, 192, and 256-Bits [177]. The Serpent is made up of three main components. The mathematical function used to construct an algorithm is the building block. These three major building blocks are as follows:

i. $InPer$ denotes the initial permutation. Using Equation 4.1, the function of $InPer$ is to reorder an original plaintext order before the encryption process (1). Where $Original_{Plaintext}$ is the $InPer$ input. A

multiplication operator is represented by the symbol " $*$ ". $Mod(127)$ is a mathematical modulus of 127, and $Output_{InPer}$ is an output of $InPer$. Please see Equation 4.1.

$$Output_{InPer} = (Original_{Plaintext} * 32) mod(127) \tag{4.1}$$

ii. Serpent's 32-round function comprises subkeys (key mixing), eight S-Boxes, and linear transformation. The 32-round-function in Figure 4.1 is mathematically explained by a mathematical expression.

iii. Serpent has a function called Final Permutation $InPer^{-1}$, which is the inverse of Initial Permutation $InPer$.

During the encryption process, Serpent employs eight 4 x 4 S-Boxes. These S-Boxes and their inverses are defined in Table 4.1 - Table 4.8. For example, if $SB_0(X)$'s input is $0 = X$, the output is 3, and $SB_0(0) = 3$. If the $SB_1(X)$ input is $1 = X$, the output is 12, and $SB_1(1) = 12$. If the $SB_7(X)$ input is $2 = X$, the output is 15, $SB_7(1) = 15$, and so on. The same is true for the inverses. Please see Figure 4.1. Serpent necessitates 33 of 128 bits. Before encryption begins, subkeys are generated from an original key provided by the user. The user can specify a key length of 128, 192, or 256 bits. The original 128-Bits key is used in this study to demonstrate how other 33-Bits subkeys are generated using the mathematical expression shown in Figure 4.2.

Table 4.1: Serpent's first S-Box was defined as $SB_0(X)$

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SB_0(X)$	3	8	F	1	A	6	5	B	E	D	4	2	7	0	9	C
$InvSB_0(X)$	D	3	B	0	A	6	5	C	1	4	4	7	F	9	8	2

The encryption part of Serpent consists of 32 rounds. The first input is the plaintext $B_0 = P$. In each round $i \varepsilon \{0, 1, 2, 3, ..., 29, 30, 31\}$, there are three building blocks which are mathematical functions.

1. The Key Mixing: In each round, a 128-Bits subkey K_i, is bitwise XORed with intermediate plaintext of B_i.
2. The Substitution boxes(S-Boxes): The inputs of $K_i \oplus B_i$ are divided into four 32-Bits words. The S-Box is applied to these four 32-Bits words to yield new substituted four 32-Bits words by using $S_i(K_i \oplus B_i)$.
3. The Linear Transformation: Each output of $S_i(K_i \oplus B_i)$ are mixed using linear transformation as follows:
   $X_0, X_1, X_2, X_3 = S_i(K_i \oplus B_i)$
   $(X_0 <<< 13) = X_0$
   $(X_2 <<< 3) = X_2$
   $(X_0 \oplus X_2 \oplus X_1) = X_1$
   $(X_2 \oplus X_3 \oplus (X_0 << 3)) = X_3$
   $(X_1 <<< 1) = X_1$
   $(X_3 <<< 7) = X_3$
   $(X_0 \oplus X_1 \oplus X_3) = X_0$
   $(X_3 \oplus X_2 \oplus (X_1 << 7)) = X_2$
   $(X_0 <<< 5) = X_0$
   $(X_2 <<< 55) = X_2$
   $X_0, X_1, X_2, X_3 = B_{1+i}$
   Where an operator $<<<$ is a rotation. An operator $<<$ is a shift. In the last round, the linear transformation is XORed with the key mixing as follows: $B_i = S_{i-1}(K_{i-1} \oplus B_{i-1}) \oplus K_i$.

Figure 4.1: Serpent's 32-Round Function [177]

Table 4.2: Serpent's second S-Box was defined as $SB_1(X)$

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SB_1(X)$	F	C	2	7	9	0	5	A	1	B	E	8	6	D	3	4
$InvSB_1(X)$	5	8	2	E	F	6	C	3	B	4	7	9	1	D	A	0

Table 4.3: Serpent's third S-Box was defined as $SB_2(X)$

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SB_2(X)$	8	6	7	9	3	C	A	E	C	1	E	4	0	B	5	2
$InvSB_2(X)$	C	9	F	4	B	C	1	2	0	3	6	D	5	8	A	7

Table 4.4: Serpent's fourth S-Box was defined as $SB_3(X)$

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SB_3(X)$	0	F	B	8	C	9	6	3	D	1	2	4	A	7	5	E
$InvSB_3(X)$	0	9	A	7	B	E	6	D	3	5	B	2	4	8	F	1

Table 4.5: Serpent's fourth S-Box was defined as $SB_4(X)$

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SB_4(X)$	1	F	8	3	C	0	B	6	2	5	4	A	9	E	7	D
$InvSB_4(X)$	5	0	8	3	A	9	7	E	2	C	B	6	4	F	D	1

Table 4.6: Serpent's sixth S-Box was defined as $SB_5(X)$

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SB_5(X)$	F	5	2	B	4	A	9	C	0	3	E	8	D	6	7	1
$InvSB_5(X)$	8	F	2	9	4	1	D	E	B	6	5	3	7	C	B	0

All the time, w_i will stand for a 32-Bits word, b_i will stand for a single bit with an original key

$w_i = b_{32i}b_{32i+1}b_{32i+2}...b_{32i+30}b_{32i+31}$.

$w_{-8}w_{-7}, ...w_{-2}w_{-1} = b_{-256}b_{-255}...b_{-3}b_{-2}b_{-1}$, from which the subkeys are:

$w_0w_1w_2w_3...w_{130}w_{131} = b_0b_1b_3b_3...b_{4222}b_{423}$

is developed by using the repetition of

$w_i = (w_{i-1} \oplus w_{i-3} \oplus w_{i-5} \oplus w_{i-8} \oplus \phi \oplus i) <<< 1$

hither $\oplus$ is bitwise-XOR operator, i is the 32-Bits depiction of the integers i, ϕ is the first 32-Bits of the binary depiction of the Golden Ration fractional part (which is $9E3779B9$ in hexadecimal), and $<<<$ 11 depicts a left rotation shift by 11-Bits position.

Succeeding, words

$k_0, k_1, k_2k_{130}, k_{131}$

are calculated using the eight S-boxes as follows:

S-box $S_{(3-j)mod8}$

is used 32 times and implemented in parallel to

$w_{4j+3}w_{4j+2}w_{4j+1}w_{4j}$ to find $k_{4j+3}k_{4j+2}k_{4j+1}k_{4j}$.

The subkeys needed by Serpent's encryption algorithm are:

$K_j = IP(k_{4j}k1 + 4jk_{2+4j}k3 + 4j)$, where

for $j = 0, 1, 2, 3, ...30, 31, 32$.

Figure 4.2: Serpent's Key Generation [177]

Table 4.7: Serpent's seventh S-Box was defined as $SB_6(X)$

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SB_6(X)$	7	2	C	5	8	4	6	B	E	9	1	F	D	3	A	0
$InvSB_6(X)$	F	A	1	D	5	3	6	0	4	9	E	7	2	C	8	B

Table 4.8: Serpent's eighth S-Box was defined as $SB_7(X)$

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SB_7(X)$	1	C	F	0	E	8	2	B	7	4	C	A	9	3	5	6
$InvSB_7(X)$	3	0	6	D	9	E	F	8	5	C	B	7	A	1	4	2

4.1.2 DL Attack

The DL attack involves a probability table called DLCT and using S-Boxes to guess the keys [97]- [98]. An attacker selects input pairs $(\wp_1, \wp_2)$ from an S-Box and uses Equation 4.2 to construct DLCT from the output pairs (ζ_1, ζ_2). There is Xi, which is calculated as $\wp_1 \oplus \wp_2$ and Λ, which is calculated as $\zeta_1 \oplus \zeta_2$ from Equation 4.1. The dot multiplication operator is used to indicate that bits are multiplied rather than entire bytes.

$$DLCT(\Xi, \Lambda) = \sum_{SB_i(x)\epsilon[1,0]} (-1)^{\Lambda.(SB_i(x) \oplus SB_i(x \oplus \Xi))} \tag{4.2}$$

It has already been stated that Equation 4.1 is used to construct DLCT using S-Box: for example, if the first Serpent S-Box is defined by Table 4.1, which has 4-Bits-input and 4-Bits-output, then the DLCT will be a 2^4 x 2^4 matrix. Generally, if an S-Box has N-bit of input and M-bit of output, its DLCT will be a 2^N x 2^M matrix when built. As a result, the DLCT of the first Serpent S-Box defined in Table 4.1 is said to be 2^4 x 2^4. The DLCT of Serpent's first S-Box is constructed using Equation 4.1 and is shown in Table 4.9.

Table 4.9: The DLCT of the Serpent's first S-Box $SB_0(X)$

$\Xi\backslash\Lambda$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
1	8	0	-4	0	-4	-4	0	4	0	-4	0	0	0	4	0	0
2	8	0	0	0	-4	0	0	-4	-8	0	0	0	4	0	0	4
3	8	-4	0	0	4	-4	0	-4	0	0	-4	0	0	4	0	0
4	8	0	0	-8	0	0	0	0	-8	0	0	8	0	0	0	0
5	8	4	0	0	0	0	-4	0	0	0	4	0	-4	0	-4	-4
6	8	-4	-4	0	0	0	0	0	8	-4	-4	0	0	0	0	0
7	8	0	4	0	0	0	-4	0	0	4	0	0	-4	0	-4	-4
8	8	-4	0	0	-4	0	-4	4	0	0	-4	0	0	0	4	0
9	8	0	0	-8	0	0	0	0	0	0	0	0	0	0	0	0
A	8	0	-4	0	4	0	-4	-4	0	-4	0	0	0	0	4	0
B	8	0	0	0	-4	0	0	-4	0	0	0	-8	4	0	0	4
C	8	0	4	0	0	-4	0	0	0	4	0	0	-4	-4	0	-4
D	8	-4	-4	8	0	4	4	0	0	-4	-4	0	0	-4	-4	0
E	8	4	0	0	0	-4	0	0	0	0	4	0	-4	-4	0	-4
F	8	0	0	0	0	4	4	0	0	0	0	-8	0	-4	-4	0

An attacker can use probability theory to guess the key with Table 4.9. In Table 4.9, the highest number is eight (8). The probability of predicting a key is 8/16, which is about the same probability of guessing the head side of a coin. In other words, using DLCT, an attacker can easily attack an algorithm. The attacker examines the relationship between $\zeta_1.\Lambda$ and $\zeta_2.\Lambda$. The key is discovered if the correction is high using DLCT. While the DLCT's primary application is to discover a more accurate key investigation of the DL attack, it can also be used to advance DL attacks to the next advanced level. This study demonstrated that attackers could use the DLCT to select the differential for ζ_1 and the linear approximation for ζ_2 in such a way that the correlation between ζ_1 and ζ_2 is exposed to the attackers' advantage [97]. This research focuses on securing Serpent found on IoT devices against DL attacks by making DLCT difficult to build with S-Box.

4.1.3 The Magic Number

The anti-design of using a constant integer directly to an algorithm's source code is called the magic number. The magic number is used to break one of the oldest coding functions [172]. The magic number makes it more difficult for an attacker to modify and analyze the source code [173]. When the same constant is applied to one section of an algorithm's source code without the derivative, attackers are more confused [172], [173].

4.1.4 The numerous DL attacks on Serpent Algorithm

Anderson *et al.* [152] used the DL attack and DLCT table to attack the Serpent algorithm. Compton *et al.* [153] developed the Simple Power Analysis attack (SPA) to attack an 8-bit Serpent-encrypted smart card. Because of a linear feedback shift register, the results showed that Serpent key generation was resistant to a side-channel attack (LFSR). Most cryptographic algorithms used LFSRs; it was suggested that Serpent's LFSRs be carefully modified and guesstimated to reduce attacks. To attack Serpent's secret keys, Bar-On *et al.* [97] created a new tool called DLCT. Canteaut *et al.* [98]

analyzed DLCT observations to obtain absolute indicators of Serpent weaknesses. Canteaut *et al.* [98] expanded on the DLCT and DL attack analytic results. Canteaut *et al.* [98] improved on the observations about DLCT and DL attack. Canteaut *et al.* [98] discovered that the DLCT approach method was similar to the auto-correlation spectrum entities, leading to the conclusion that DLCT was nothing more than an Auto Correlation Table (ACT). The ACT spectrum was invariable under any equivalence similarity and was not invariant under alterations, according to Canteaut *et al.* [98]. With the aid of the DLCT tool, Biham *et al.* [155] attacked the Serpent algorithm using the DL attack. Undoubtedly, the DL attack and DLCT table may be used to attack the Serpent algorithm. Refer to the study's literature review for more details on the Serpent attack.

4.2 Methods of Securing Serpent against DL Attack

The primary goal of this research was to protect Serpent from DL attacks on IoT devices. The original 4-Bits-output S-Boxes in Serpent were replaced with newly generated 32-Bits-output S-Boxes. A new mathematical function called Blocker was created using three magic numbers. To improve encryption and decryption while resisting DL attacks, Serpent's infrastructure was upgraded with new 32-bit output S-Boxes and a Blocker.

A new modified Serpent was created by inserting new 32-Bits-output S-Boxes and a Blocker into Serpent's infrastructure. In this study, the modified Serpent with new S-Boxes and Blocker was dubbed Magic Serpent (Mag_Serpent). Because the encryption process, strength, and resistance to the DL attack were more substantial than that of an original Serpent found on IoT devices, the functionality of Mag_Serpent was discovered to be very different from that of an original Serpent. The study was conducted as follows:

i. The Serpent was collected using IoT devices (like sensors, smart cards, and 8-Bits processors).

ii. The correctness of Serpent was checked and tested using test vectors provided by Serpent developer reports.

iii. During the DL attack process, all of the procedures implemented on Serpent were tested and analyzed in C++.

iv. Serpent's original 4-bit output S-Boxes were all modified with newly generated 32-bit output S-Boxes.

v. Three magical numbers have been used to generate a new function called Blocker, which was then implemented in the Serpent infrastructure using C++. Refer to Figure 4.4.

vi. All original Serpent functions that called S-Boxes with 4-bit output were changed to call Blocker functions with 32-bit output S-Boxes. As an example, if

$$Output = SB_i(x) \tag{4.3}$$

Note: $SB_i(x)$ on Equation 4.3 is an S-Box with four bits of output. Equation 4.3 is omitted in favor of Equation 4.4.

$$Blocker(SB_i(x), Output) \tag{4.4}$$

$SB_i(x)$ Since all original 4-Bits-output S-Boxes were replaced with new 32-Bits-output S-Boxes, on Equation 4.4 is a 32-Bits-output S-Box. On key generation, defined in Figure 4.2, the Golden ratio $\phi = 9E3779B9$ was also replaced by magic number $M = 4294967296$.

vii. The possibility of a DL attack was examined to see if it was still effective after applying or inserting new S-Boxes and a Blocker. If it was still feasible, steps (iii) and (iv) were repeated.

viii. If the DL attack was stopped at steps (iii), (iv), and (v), a new algorithm with new 32-Bits-output S-Boxes was introduced, and the Blocker was accepted as a Mag_Serpent. As a result, it was discovered that Mag_Serpent is immune to DL attacks.

The research methodology was used to determine how to make DLCT more challenging to build to prevent attackers from discovering Serpent's keys following a DL attack. Serpent's security is already stated to be dependent on the size of the output bits of the S-Boxes. The original output bits of Serpent's S-Boxes were short (4-Bits).

This type of algorithm is easy for attackers to exploit. To increase the size of the output bits and protect Serpent from DL attacks, all 4-Bits-output S-Boxes were replaced with newly generated 32-Bits-output S-Boxes. The new 32-Bits-output S-Boxes successfully prevented DL attacks, while the Blocker function prevented DLCT construction. A schematic diagram was used to summarize the research methodology in Figure 4.3. The results successfully prevented the DLCT from being built, resulting in a complicated process for carrying out the DL attack on Serpent.

The Serpent's S-Boxes were found to be 4 x 4, indicating that they had 4-Bits-inputs and 4-Bits-outputs. The experiment discovered that it was simple to construct DLCT using these types of S-Boxes. The original Serpent's S-Boxes' DLCT were 2^4 x 2^4 matrix tables with high-probability elements for discovering secret keys. In general, if an S-Box has N bits of input and M bits of output, its DLCT, when built, will be a 2^N x 2^M matrix. As a result, the DLCT of Serpent's first S-Box was stated to be 2^4 x 2^4 in Table 4.1. Using Equation 4.1, C++ program code was written to construct DLCT of the original first S-Boxes defined in Table 4.1.

It is simple to attack Serpent with DLCT, as discussed by Bar-On *et al.* [97] and Canteaut *et al.* [98]. To combat the DL attack, new 32-Bits-output S-Boxes were developed to replace Serpent's original S-Boxes. Table 4.1 - Table 4.8 were replaced, for example, with Table 4.10 - Table 4.17. Using the

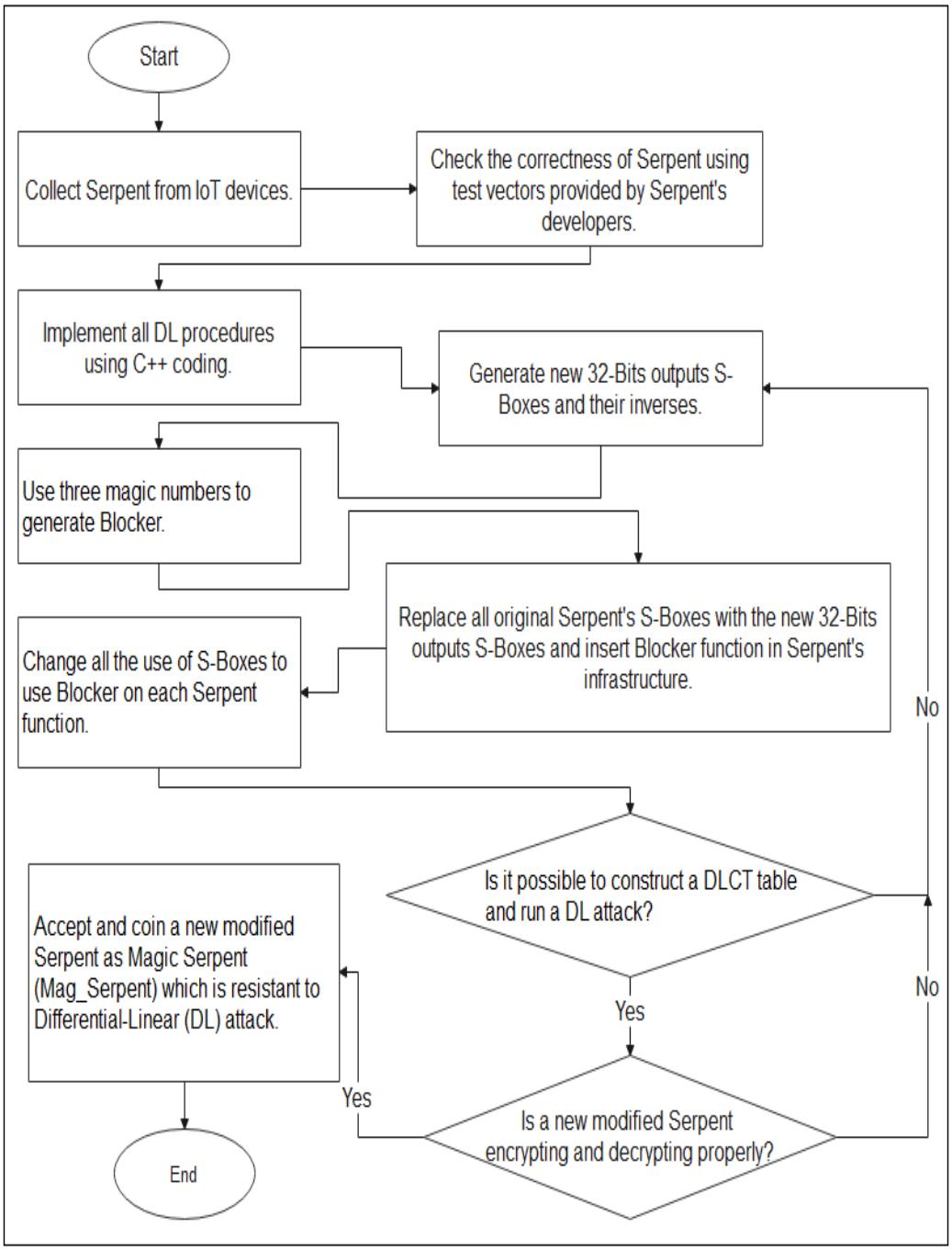


Figure 4.3: Serpent Research Methodology Schematic Diagram

C++ code shown in Figure 4.4, the Blocker function was created from three magic numbers. The magic numbers were $Q = 4302746963$, $P = 4559351687$, and $M = 4294967296$.

Table 4.10: New Generated 32-Bits-output S-Box to replace Table 4.1

X	$SB_0(X)$	$InvSB_0(X)$
0	411264f80	411264f80
1	91377da1f	10fc22f87
2	1016b6cf64	7128a6f79
3	21038e4da	e15c964be
4	b146544c5	a13ee8f72
5	7128a6f79	f16401a11
6	61213ba26	1016b6cf64
7	c14dbfa18	91377da1f
8	f16401a11	61213ba26
9	e15c964be	d1552af6b
A	5119d04d3	c14dbfa18
B	310af9a2d	8130124cc
C	8130124cc	b146544c5
D	10fc22f87	21038e4da
E	a13ee8f72	5119d04d3
F	d1552af6b	310af9a2d

Table 4.10 - Table 4.17 were composed experimentally in C++ programs and are represented by Appendix B, Figure B.1 and Appendix B, Figure B.2. In C++, Appendix B, Figure B.1 denoted all new 32-bit S-Boxes, while Appendix B, Figure B.2 denoted all new 32-bit S-Box inverses.

Table 4.11: New Generated 32-Bits-output S-Box to replace Table 4.2

X	$SB_1(X)$	$InvSB_1(X)$
0	1016b6cf64	1016b6cf64
1	d1552af6b	b146544c5
2	310af9a2d	21038e4da
3	8130124cc	e15c964be
4	a13ee8f72	61213ba26
5	10fc22f87	411264f80
6	61213ba26	7128a6f79
7	b146544c5	10fc22f87
8	21038e4da	5119d04d3
9	c14dbfa18	a13ee8f72
A	f16401a11	f16401a11
B	91377da1f	8130124cc
C	7128a6f79	310af9a2d
D	e15c964be	d1552af6b
E	411264f80	91377da1f
F	5119d04d3	c14dbfa18

Table 4.12: New Generated 32-Bits-output S-Box to replace Table 4.3

X	$SB_2(X)$	$InvSB_2(X)$
0	91377da1f	91377da1f
1	7128a6f79	1016b6cf64
2	8130124cc	310af9a2d
3	a13ee8f72	a13ee8f72
4	411264f80	5119d04d3
5	d1552af6b	21038e4da
6	b146544c5	e15c964be
7	1016b6cf64	f16401a11
8	e15c964be	c14dbfa18
9	21038e4da	7128a6f79
A	f16401a11	61213ba26
B	5119d04d3	411264f80
C	10fc22f87	8130124cc
D	c14dbfa18	d1552af6b
E	61213ba26	b146544c5
F	310af9a2d	10fc22f87

Table 4.13: New Generated 32-Bits-output S-Box to replace Table 4.4

X	$SB_3(X)$	$InvSB_3(X)$
0	10fc22f87	61213ba26
1	1016b6cf64	10fc22f87
2	c14dbfa18	91377da1f
3	91377da1f	411264f80
4	d1552af6b	b146544c5
5	a13ee8f72	a13ee8f72
6	7128a6f79	8130124cc
7	411264f80	f16401a11
8	e15c964be	310af9a2d
9	21038e4da	d1552af6b
A	310af9a2d	c14dbfa18
B	5119d04d3	7128a6f79
C	b146544c5	5119d04d3
D	8130124cc	1016b6cf64
E	61213ba26	e15c964be
F	f16401a11	21038e4da

Table 4.14: New Generated 32-Bits-output S-Box to replace Table 4.5

X	$SB_4(X)$	$InvSB_4(X)$
0	21038e4da	10fc22f87
1	1016b6cf64	a13ee8f72
2	91377da1f	b146544c5
3	411264f80	8130124cc
4	d1552af6b	c14dbfa18
5	10fc22f87	f16401a11
6	c14dbfa18	7128a6f79
7	7128a6f79	e15c964be
8	310af9a2d	411264f80
9	61213ba26	61213ba26
A	5119d04d3	d1552af6b
B	b146544c5	310af9a2d
C	a13ee8f72	5119d04d3
D	f16401a11	91377da1f
E	8130124cc	1016b6cf64
F	e15c964be	21038e4da

Table 4.15: New Generated 32-Bits-output S-Box to replace Table 4.6

X	$SB_5(X)$	$InvSB_5(X)$
0	1016b6cf64	1552af6b
1	61213ba26	a13ee8f72
2	310af9a2d	1016b6cf64
3	c14dbfa18	5119d04d3
4	5119d04d3	c14dbfa18
5	b146544c5	f16401a11
6	a13ee8f72	21038e4da
7	d1552af6b	310af9a2d
8	10fc22f87	10fc22f87
9	411264f80	411264f80
A	f16401a11	7128a6f79
B	91377da1f	e15c964be
C	e15c964be	61213ba26
D	7128a6f79	91377da1f
E	8130124cc	b146544c5
F	21038e4da	8130124cc

Table 4.16: New Generated 32-Bits-output S-Box to replace Table 4.7

X	$SB_6(X)$	$InvSB_6(X)$
0	8130124cc	61213ba26
1	310af9a2d	91377da1f
2	d1552af6b	310af9a2d
3	61213ba26	f16401a11
4	91377da1f	1016b6cf64
5	5119d04d3	7128a6f79
6	7128a6f79	d1552af6b
7	c14dbfa18	411264f80
8	f16401a11	c14dbfa18
9	a13ee8f72	5119d04d3
A	21038e4da	8130124cc
B	1016b6cf64	a13ee8f72
C	e15c964be	21038e4da
D	411264f80	e15c964be
E	b146544c5	b146544c5
F	10fc22f87	10fc22f87

Table 4.17: New Generated 32-Bits-output S-Box to replace Table 4.8

X	$SB_7(X)$	$InvSB_7(X)$
0	21038e4da	e15c964be
1	e15c964be	411264f80
2	1016b6cf64	c14dbfa18
3	10fc22f87	10fc22f87
4	f16401a11	b146544c5
5	91377da1f	7128a6f79
6	310af9a2d	61213ba26
7	c14dbfa18	d1552af6b
8	8130124cc	21038e4da
9	5119d04d3	f16401a11
A	d1552af6b	5119d04d3
B	b146544c5	8130124cc
C	a13ee8f72	1016b6cf64
D	411264f80	a13ee8f72
E	61213ba26	91377da1f
F	7128a6f79	310af9a2d

4.2.1 A Blocker Function

A new function known as a Blocker is introduced in this study. Please refer to Figure 4.4. A Blocker function is a brand-new C++ function solely responsible for developing DL attack blockage on the Serpent algorithm, which is required on IoT devices. This function is called after the S-Boxes of the Serpent algorithm have been transformed to produce the 32-bit output S-Boxes. The primary purpose of a Blocker function is to ensure that newly 32-bit output S-Boxes are compatible with the Serpent algorithm infrastructure. In layman's terms, a Blocker function governs all-new 32-bit output S-Boxes that are efficiently used throughout the encryption and decryption processes of the newly modified Serpent algorithm. A Blocker function provides a 32-bit S-Box compatible with the new Magic Serpent Algorithm. A Blocker Function confuses the intruder because it contains many mathematical random numbers. In addition, the vast majority of mathematical random numbers are irreversible. Unless a Blocker function is used, newly generated 32-bit output S-Boxes will not be placed in algorithms. This Blocker function has unique properties that prevent DL attacks. These are the characteristics:

i. Unlike S-Boxes, which use a look-up table with defined inputs and outputs, the output of a Blocker function is not fixed.

ii. Unlike the Serpent S-Boxes, where the output is visible on a look-up table, the output of a Blocker function is hidden and calculated.

iii. The presence of a Blocker function is unavoidable. If a Blocker function output that does not represent an input is recognized, it can be estimated and retrieved in reverse. A Blocker function is composed of several random numbers and XOR operators.

iv. Magic numbers (such as P, Q, and M) used in a Blocker function are unfactorizable. Refer to Figure 4.4.

v. A Blocker function is made up of non-linear functions.

vi. A Blocker function's input is 32 bits long, and an intruder cannot easily create the DLCT of 2^{32} using a computer or any processor because it requires a large amount of memory.

vii. A Blocker function takes as input the output of 32-bit S-Boxes and manipulates it. The Magic Serpent algorithm is then used to generate an excellent output value. Intruders are confused because a new distinct output value is unpredictable.

viii. The expression $state32hold$ determines the output of 32-bit S-Boxes. This output is passed to a Blocker function, which returns an unknown variable named *statehold*. Refer to Figure 4.4.

ix. Because S-Boxes in a Blocker function are mathematically preserved and unalterable, all functions in the Serpent algorithm that recall S-Boxes must identify or employ a Blocker function after executing a Blocker function.

x. A Blocker function prevents tampering with the 32-bit output S-Boxes. Assume that the 32-bit output S-Boxes' positions are changed or that the 32-bit S-Boxes are displaced. In that case, Mag_Serpent will not produce the expected results.

This study employs a Blocker function to generate a new 32-bit S-Box suitable for the new Magic Serpent Algorithm and distract the attacker because it contains many mathematical random numbers and XOR operators. In addition, the vast majority of mathematical random numbers and XOR operators are irreversible. To increase robustness against DL attacks, a Blocker

function has been added to the traditional S-Boxes used in Serpent algorithms. A Blocker function improves the suitability of new 32-bit S-Boxes and prevents a DL attack by a newly developed Magic Serpent algorithm. A Blocker function is mathematically defined as follows:

Assign: $M = 4294967296$,

do: Change the value of $state32hold$ by using it as an illustration of the value of:

$state32hold = state32hold \times (\frac{state32hold}{M}) + M \times (\frac{M}{state32hold})$

Where $state32hold$ is a 32-bit S-Box Blocker function input.

If the value of $state32hold$ is greater than M, the first loop of the if statement is opened.

Assign:

$Q = 4559351687$,

$P = 4302746963$ and

$iSecret$ is a random number in the range $(P \oplus Q)$. This number will be combined with the magic numbers throughout the Blocker function. $iSecret$ is an unpredictable and irreversible random number. Close the first looping if statement.

If the value of $state32hold$ is less than or equal to M, the second loop of if or $else$ statements is opened. Assign : $iSecret$ be a random number with the range from up to $(Q \oplus M)$. Assign $M = state32hold$ and $Q = state32hold <<< 2$. Where $<<<$ represents left round shifting of the number of bits, for example, 5 in decimal notation = 0101 in binary notation. If you round 0101 to the left (by one), you get 1010 in binary notation, which equals 10 in decimal notation or A in hexadecimal notation. As a result, in decimal notation, $5 <<< 1 = 10$.

$P = state32hold <<< 4$

$Q = M \oplus P$

$P = M \oplus Q$

$M = Q \oplus P$

Change the values of M, P, and Q to a random number between 0 and $iSecret$; this can be expressed mathematically as:

$M = rand(M)\ modulo\ iSecret$

$P = rand(P)\ modulo\ iSecret$

$Q = rand(Q)\ modulo\ iSecret$

The *modulo* operative, is the arithmetical operator that passes the remainder of a division random number x, designated by ($rand(x)$ and $iSecret$. x in this research x could be M, P, or Q. Close the second if or $else$ statement loop.

The declared values from the first and second if statements should be collected. If the recollected values of Q and P are more significant than zero, a variable called $TempState$ is created. do: Assign

$TempState = NOT(state32hold)ANDQ$.

In mathematics, NOT and AND are bitwise operators.

If the input is a positive integer, the NOT operator returns a negative number multiplied by one. $NOT(10) = -11$, $NOT(5) = -6$, $NOT(2) = -3$, and so on.

do: assign

$state32hold = |(state32hold \oplus Q|$, where $|x|$ denotes an absolute operator. An absolute operator converts every negative variable to a positive variable. For example, $|-y| = |y| = y$. do: assign

$statehold = (\frac{state32hold}{P}) \oplus 187$

$iSecret = rand(iSecret)\ modulo\ (P \oplus M)$ do: assign

$Q = TempState <<< 1$.

It should be observed that the issuance of $Q = TempState <<< 1$ decreases the value of Q indefinitely until Q is less than zero. A Blocker Function checks to see if Q is greater than zero. If Q is more significant than zero, either repeat the third for loop until Q is less than zero or change the values of Q, P, and M to be random numbers ranging from 0 to the value of $statehold$. $Q = rand(Q)\ modulo\ (statehold)$

$M = rand(M)\ modulo\ (statehold)$

$P = rand(P)\ modulo\ (statehold)$

Convert or substitute the new *statehold* value used by various Serpent functions or algorithm building blocks.

Put the third for loop to rest. Deactivate a Blocker function.

A Blocker Function accepts a 32-bit S-Box output value as input and returns a new *statehold* value as output. A Blocker Function also makes P, M, and Q unfactorizable polynomials. Random numbers and XOR operators add complexity and confusion to prevent attackers from reverse engineering. To change the values of variables within a Blocker Function, use the XOR operator and $rand()$. When invaders reverse a Blocker Function to calculate the exact information used in that situation, the random numbers and XOR operators also provide a problematic input range.

The values of M, P, and Q are continuously maintained as unfactorizable polynomial variables that are non-linear and difficult to reverse to construct DLCT using any machine or computer. Intruders can create hidden, unseen, and unchangeable variables using random numbers and XOR operators. A Blocker function creates a unique 32-Bit S-Box suitable for the new Magic Serpent Algorithm. A Blocker Function distracts the attacker by containing many mathematical random numbers and XOR operators.

Furthermore, most XOR operators in mathematics and random numbers are irreversible. More mathematical characteristics of a Blocker function and C++ explanations can be found in Figure 4.4.

```
uint8_t Blocker(uint64_t state32hold, uint8_t statehold)
 //The variable state32hold is an
//output of a newly 32-output-bit generated S-Box,
 //At this stage, state32hold is an input of Blocker.
 // statehold is an output of Blocker.
{   uint64_t Q=0, P=0, TempState =0, M=4294967296;
    uint64_t iSecret;
    state32hold = state32hold * (bool)(state32hold/ M)
     + M * (bool)(M / state32hold);
    if (state32hold > M)
    {   Q = 4559351687;
        P= 4302746963 % Q;
//M-value, P-value and Q value are unfacorizable polynomials.
        iSecret = rand() % (P^Q);
//Intializing a random number to be used in Blocker.
    }
    else
    {//Unfactorizable polynomial numbers are
     //non-linear and cumbersome to construct LAT tables using PC.
        iSecret = rand()% (Q^M);
        M = state32hold;
        Q = state32hold <<2;
        P = state32hold <<4;
        Q = M ^ P;
        P = M ^ Q;
        M = Q ^ P;
//Changing M, P and Q to be random of a range iSecret
        M = rand()% iSecret;
        P = rand()% iSecret;
        Q = rand()% iSecret;
//Note that iSecret is also random from iSecret = rand()% (Q^M);
//At this stage M, P and Q are unknown, invisible and irreversible to
//an intruder, since they are random numbers.
    }while (Q % M)   //Second modular operator (%) to make Q-value
        {   //unknown invisible and irreversible to an intruder.
        TempState = (~state32hold) & Q;
        state32hold = _abs64(state32hold^ Q);
        statehold= ((state32hold)/P)^0273;
        iSecret = rand()% (P^M);
        Q = TempState << 1;
        }
//Changing M, P and Q to be random of a range statehold
        Q = rand()% statehold;
        M = rand()% statehold;
        P = rand()% statehold;
        return statehold;
//At this stage M, P and Q are unknown, invisible and irreversible
//to an intruder, since they random.
}
```

Figure 4.4: New Generated Function called Blocker

4.2.2 Results of DL attack on Serpent

This study experimentally verified and analyzed the DL attack performed in [178] on a 12-round Serpent. In round 0, the attack was based on the fundamental 11-round DL attack, which used a plaintext pair and a pair that provides the input differentials of 28 participating S-boxes. As a result, changing the Serpent algorithm and launching a 12-round attack against Serpent with 256-bit keys were viable options.

Dunkelman *et al.* [178] attempted all possible input differences to round 1, yielding the difference $LT^{-1}(\Xi P) = 20000000000001A00E00400000000000_x$. S-boxes 2, 3, 19, and 23, for example, had no effect on the difference because they did not change the participating bits of $LT^{-1}(\Xi P)$. As a result, Dunkelman *et al.* [178] built plaintext structures that took this into account and obtained a 12-round attack on Serpent:

i. Dunkelman *et al.* [178] picked $N = 2^{123.5}$ plaintexts, each consisting of $2^{11.5}$ structures, and each was chosen by selecting: (a) A plaintext abitaray $\wp_0$. (b)The plaintexts $\wp_1, ..., \wp_{2^{112}-1}$ that differed from $\wp_0$ by all $2^{112} - 1$ possibilities of non-empty subgroups of the bits used for inputs of all S-Boxes but apart from 2, 3, 19, and 23 in round 0 [178].

ii. The ciphertexts of those plaintext structures encrypted using the private unknown key K were requested by Dunkelman *et al.* [178].

iii. Using those 28 S-boxes, partially encrypt all plaintexts in the first round for each input 112-bit K_0 value and use the original 11-round DL attack on Serpent [178].

iv. Dunkelman *et al.* [178] for each experimental key revealed. Subkeys of $112 + 20 + 28 = 160$ bits. 112-bit round 0, 20-bit round 1, and 28-bit

round 11 are tested simultaneously with an accuracy test [178]. With the appearance of more than 84 percent completion rate, the accurate estimation of the 160-bit was expected to be the typical frequently expected value [178].

v. The remaining key bits were obtained using supplementary techniques [178].

The study experimentally confirmed that the data attack complexity was $2^{123.5}$ for the plaintexts used. For the partial encryption in Step (iii), the time attack complexity is $2^{123.5} \times (\frac{28}{384}) \times 2^{112} = 2^{231.7}$, and $2^{137.4} \times 2^{112} = 2^{249.4}$ for the repeated trials of the 11-round DL attack [178].

The study also experimentally confirmed that on a 10-round DL attack of Serpent using 128-bit keys, the data complexity was $2^{101.2}$ for plaintexts and $2^{115.2}$ for time encryption.

4.2.3 Procedure of DL attack on a Mag_Serpent

Mag_Serpent employed a new 32-bit S-box, which refused to execute the C++ DLCT from various computers and machines due to memory constraints on those multiple computers and appliances. No computers or devices could compute the DLCT of a 2^4 X 2^{32} = 16 x 4294967296 matrix, which was assumed to contain 68719476736 entities. Without DLCT, a DL attack on a newly generated 4 x 32 S-Boxes of Mag_Serpent algorithm was impractical. No rounds out of 32 were attacked using the DL attack due to the new 32-bit output S-Boxes, which hampered the development of the DLCT due to memory constraints.

Theoretical development of DLCT was examined and experimentally programmed in C++ code for validation, testing, confirmation, and verification. The DL attack was possible on the Serpent, according to the results. The size of the S-Boxes was the primary building block that performed all possibilities of the DL attack. The Serpent's S-Boxes were 4 x 4, indicating that

the input and output were 4 bits. The experiment determined that building the DLCT with the 4 x 4 Serpent S-Boxes was simple. Refer to Table 4.9 and Figure 4.5.

The DLCT of 4 x 4 and 4 x 32 S-Boxes were generated using a C++ program. The code validation was tested using 4 x 4 Serpent S-Boxes and a newly developed 4 x 32 Mag_Serpent S-Box. The goal of validating the code was to ensure that the written C++ experimental output DLCT corresponded to the theoretical outputs. It is worth noting that the DLCT of a 4 x 4 S-Box is a matrix of 2^4 X 2^4 = 16 x 16 matrix with 256 entities. Refer to Table 4.9 and Figure 4.5.

The experiment was carried on using a newly developed 4 x 32 S-Box of the Mag_Serpent algorithm. The program failed five hours before the DLCT was to be executed. The DLCT of a 2^4 X 2^{32} = 16 x 4294967296 matrix, expected to contain 68719476736 entities, could not be computed by a computer or machine. Without the DLCT, a DL attack on a newly developed 4 x 32 S-Box of the Mag_Serpent algorithm was impractical. Refer to Table 4.9 and Figure 4.5.

The first integer in the DLCT of a 4 x 4 S-Box was 16, which is (2^4) when S-Box required four bits output as the most specific parameter. In binary notation, 16 is a byte donated as 00010000. If each 4 x 4 S-Box DLCT is treated as a byte, the memory needed to build a 4 x 4 S-Box DLCT was 8 bits x 256 = 256 bytes. A 4 x 4 S-Box DLCT can display 256 items at once. A computer can efficiently handle 4096 bytes. Refer to Table 4.9 and Figure 4.5.

Based on the above calculations, S-Box required 32 bits as the first parameter. The study assumed that the first number item in the DLCT of a 4 x 32 S-Box would be 4294967296, which is (2^{32}). 4294967296 is a triple-word made up of five bytes donated as follows:

00000000100000000000000000000000000000000 in binary notation.

If each 4 x 32 S-Box DLCT element were treated as a triple-word, the

memory required to construct a 4 x 32 S-Box DLCT would be 40 $bits \times 2^4 \times 2^{32} = 343597383680$ $bytes$. The expected number of entities displayed on a 4 x 32 S-Box DLCT was 43597383680. A computer could not easily handle each item's computation memory of 343597383680 bytes. As a result, the C++ DLCT of the 4 x 32 S-Box program failed before execution. Refer to Table 4.9 and Figure 4.5.

The Serpent S-Box DLCT was a table 2^4 rows x 2^4 columns with a high probability of comprehending a key. The C++ program was used in the experiment to generate the DLCT of a 4 x 4 Serpent S-Box. The examination results confirmed that it was possible to attack the Serpent algorithm using the DLCT. The research used newly created 32 output bit S-Boxes on Serpent found on IoT devices to prevent a DL attack. Refer to Table 4.9 and Figure 4.5.

Table 4.9 represented the assumed DLCT, and Figure 4.5 represented the experimentally analyzed DLCT performed by running a C++ DLCT code. A C++ DLCT code was written to demonstrate and confirm that the study of building a DLCT was carried out using all methods of DL attack on a Serpent.

The code was also applied to Serpent and Mag_Serpent to see if a DL attack was possible. All of the results were presented and explained throughout the development of the DLCT, and S-Boxes were implemented using a novel approach of utilizing 32-bits. A Blocker function was used in the study to generate a new 32-Bit S-Box suitable for the new Mag_Serpent algorithm and distract the intruder.

When attackers reverse back a Blocker Function to determine the accurate information in that situation, the random numbers and XOR operators also provide a problematical input range. To construct DLCT using any machine or computer, the values of M, P, and Q are continuously maintained as unfactorizable polynomial variables that are non-linear and difficult to reverse.

Intruders can use random numbers and XOR operators to create hidden, unseen, and unchangeable variables. A Blocker function generates a one-of-a-kind 32-Bit S-Box suitable for the new Magic Serpent Algorithm. Because it contains many mathematical random numbers and XOR operators, a Blocker Function distracts the attacker. Furthermore, the majority of mathematical XOR operators and random numbers are irreversible. Figure 4.4 contains additional mathematical characteristics of a Blocker function as well as C++ explanations.

Mag_Serpent was resistant to a DL attack in this study and created the new 4 x 32 S-Boxes. The new 32-bit S-Boxes suitable for the new Mag_Serpent algorithm were inserted into the survey using a Blocker function. The study used the Blocker function to confuse the attacker because it contains many mathematical random numbers and XOR operators.

Furthermore, the majority of mathematical XOR operators and random numbers are irreversible. After incorporating a Blocker function and the new 4 x 32 S-Boxes, the new Mag_Serpent successfully decrypted and encrypted. The code for the newly Mag_Serpent is available upon request. The C++ code confirmed that a DL attack was permissible on several rounds, including round 12, before applying a Blocker function and the new 4 x 32 S-Boxes to a standard Serpent. Nonetheless, after employing a Blocker function and the novel 4 x 32 S-Boxes, C++ code confirmed that the study successfully blocked the DL attack on Mag_Serpent. Furthermore, due to a computer's memory constraint, creating a DLCT of 2^{32} rows and columns matrix was challenging.

4.3 Results, Discussions, and Analysis of Securing Serpent against DL Attack

The DL attack was possible on the Serpent, according to the results. The size of the S-Boxes was the main building block that performed all DL attack possibilities. S-Boxes on the Serpent were 4 x 4, indicating 4-bit input and

4-bit output. The experiment determined that building the DLCT with the 4 x 4 Serpent S-Boxes was simple. Refer to Table 4.9 and Figure 4.5.

The DLCT of 4 x 4 and 4 x 32 S-Boxes were generated using a C++ program. Code validation was investigated using 4 x 4 Serpent S-Boxes and a newly developed 4 x 32 Mag_Serpent S-Box. The goal of validating the code was to ensure that the written C++ experimental output DLCT corresponded to the theoretical results. It is worth noting that the DLCT of a 4 x 4 S-Box is a matrix of 2^4 X 2^4 = 16 x 16 matrix with 256 entities. Refer to Table 4.9 and Figure 4.5.

The experiment used a newly developed 4 x 32 S-Box of the Mag Serpent algorithm. The program failed five hours before the DLCT was to be executed. The DLCT of a 2^4 X 2^{32} = 16 x 4294967296 matrix, expected to contain 68719476736 entities, could not be computed by a computer or machine. Without the DLCT, a DL attack on a newly developed 4 x 32 S-Box of the Mag_Serpent algorithm was impractical.

The DL attack was possible on the Serpent, according to the results. The size of the S-Boxes was the main building block that performed all DL attack possibilities. S-Boxes on the Serpent were 4 x 4, indicating 4-bit input and 4-bit output. The experiment determined that building the DLCT with the 4 x 4 Serpent S-Boxes was simple. Refer to Table 4.9 and Figure 4.5.

The first integer in the DLCT of a 4 x 4 S-Box was 16, which is (2^4) when S-Box required four bits output as the most specific parameter. In binary notation, 16 is a byte donated as 00010000. If each 4 x 4 S-Box DLCT is treated as a byte, the memory needed to build a 4 x 4 S-Box DLCT was 8 bits x 256 = 256 bytes. It is worth noting that 256 items are displayed on a 4 x 4 S-Box DLCT. A computer can handle 4096 bytes efficiently. Refer to Table 4.9 and Figure 4.5.

Based on the above calculations, S-Box required 32 bits as the first parameter. The study assumed that the first number item in the DLCT of a 4 x 32 S-Box would be 4294967296, which is (2^{32}). 4294967296 is a triple-word

made up of five bytes donated as

0000000010000000000000000000000000000000 in binary notation.

If each 4 x 32 S-Box DLCT element were treated as a triple-word, the memory required to construct a 4 x 32 S-Box DLCT would be 40 $bits \times 2^4 \times 2^{32} = 343597383680$ $bytes$. The expected number of entities displayed on a 4 x 32 S-Box DLCT was 343597383680. A computer could not easily handle each item's computation memory of 343597383680 bytes. As a result, the C++ DLCT of the 4 x 32 S-Box program failed before execution. Refer to Table 4.9 and Figure 4.5.

The Serpent S-Box DLCT was a table 2^4 rows x 2^4 columns with a high probability of comprehending a key. The C++ program was used in the experiment to generate the DLCT of a 4 x 4 Serpent S-Box. The examination results confirmed that it was possible to attack the Serpent algorithm using the DLCT. The research used newly created 32 output bit S-Boxes on Serpent found on IoT devices to prevent a DL attack. Refer to Table 4.9 and Figure 4.5.

Since the new output bits were increased from 4-Bits to 32-Bits, the new 32-Bits-output S-Boxes prevented the construction of DLCT, which was assumed to be a 2^4 x 2^{32} matrix. That is, DLCT requires a 2^4 x $2^{32} = 256$ x 4294967296 matrix, which requires a large amount of computer memory to compute and display. The experiment demonstrated that constructing the DLCT of a new 32-Bits-output S-Box using Equation 4.1 was impractical if the Blocker function was embedded on Serpent's structure because the maximum size limitation was limited memory required had been exceeded.

Due to a computer's memory limitation, the C++ program for constructing the DLCT of the new S-Boxes clashed before DLCT was finally built. A standard computer cannot create a matrix of 256 columns x 4294967296 rows. The experiment also confirmed that constructing a 256 x 4294967296 matrix was impractical because a computer requires a maximum memory of 2^{64}, which is impossible. Without DLCT, there is no DL attack. In this study,

using the Blocker function and 32-Bits-output S-Boxes to protect Serpent from DL attacks worked.

When the Blocker function was added to Serpent's infrastructure, all positions of 32-Bits S-Boxes became unchangeable. A new 32-Bits-output $SB_0(X)$, for example, cannot be changed or substituted with any arbitrary new 32-Bits-output S-Box such as $SB_2(X)$,..., or $SB_7(X)$. Even if the sizes are equal, the newly generated S-Boxes cannot be replaced by any 32-Bits-output S-Box taken from other known algorithms.

This study investigated and carried out all procedures used to attack the original Serpent with DL attacks. The C++ programs were created to test whether an original Serpent could be attacked with DLCT and DL attacks. The C++ programs validated and carried out the same results as shown in Table 4.9. Table 4.9 contained the theoretical results discovered by Bar-On *et al.* [97] when the original Serpent was attacked with the 4-Bits-output S-Box defined in Table 4.1. All of the procedures used to attack an original Serpent by Bar-On *et al.* [97] were performed with the assistance of C++ programs.

The C++ programs in this study confirmed and validated the theoretical results defined in [97]. Figure 4.5 depicts the experimental results obtained in this study. The elements in Figure 4.5. Table 4.9 and Figure 4.5 were the same. The theoretical DLCT results were shown in Table 4.9, as explained by ar-On *et al.* [97], and the experimental DLCT results were shown in Figure 4.5. On rounds 10 and 11, the Serpent was attacked, but Mag_Serpent resisted the DL attack.Refer to Table 4.20 and Figure 4.6.

Before implementing the new S-Boxes and Blocker approach, the C++ experiment demonstrated that a DL attack on an original Serpent was possible. Nonetheless, the DL attack was blocked on a new modified Serpent called Mag_Serpent after the novelty of using the new 32-Bits-output S-Boxes and Blocker function. Refer to Table 4.18 and Table 5.15.

The Avalanche Effect is a desirable feature of algorithms in cryptography

[169]. If one of the input bits is inverted (flipped), the output bits must significantly improve. In robust algorithms, such a slight change in either the plaintext or the key should result in excessive variation in the ciphertext [169]. The Avalanche Effect is used to develop a method known as the Strict Avalanche Criterion (SAC) for evaluating the encryption robustness of algorithm [170].

The SAC is achieved if a specific input bit, either plaintext or key, returns the transformation of ciphertext output bits with a probability of 50 percent [170]. The experiment used the Avalanche Effect on Serpent and Mag_Serpent to obtain SAC. According to the results, the Serpent and a newly generated Mag_Serpent algorithm had a better SAC characteristic. When compared to SAC characteristics, the Avalanche Effect of Mag_Serpent and Serpent on both key and plaintext was approximately 50%. Refer to Table 4.21 and Figure 4.7 to Figure 4.12.

One of the essential parameters in cryptography is the amount of memory required before installing an algorithm. Regardless of encryption strength, it is ignored if an algorithm requires more memory than the platform or environment in which it is installed. The memory of both Serpent and Mag_Serpen was measured in the study using the C++ program. Serpent and Mag_Serpen had memory sizes of 11181 bytes and 13206 bytes, respectively. Refer to Table 4.22 and Figure 4.13 to Figure 4.15.

Serpent and Mag_Serpent encryption and decryption were tested to see if both algorithms worked flawlessly for encryption and decryption. Using an image, the C++ program was used to test the encryption and decryption of Serpent and Mag_Serpent. The results showed that both Serpent and Mag_Serpent encryption and decryption worked as expected. Refer to Figure 4.16.

```
8  8  8  8  8  8  8  8  8  8  8  8  8  8  8  8
8  0  -4  0  -4  -4  0  4  0  -4  0  0  0  4  0  0
8  0  0  0  -4  0  0  -4  -8  0  0  0  4  0  0  4
8  -4  0  0  4  -4  0  -4  0  0  -4  0  0  4  0  0
8  0  0  -8  0  0  0  0  -8  0  0  8  0  0  0  0
8  4  0  0  0  0  -4  0  0  0  4  0  -4  0  -4  -4
8  -4  -4  0  0  0  0  0  8  -4  -4  0  0  0  0  0
8  0  4  0  0  0  -4  0  0  4  0  0  -4  0  -4  -4
8  -4  0  0  -4  0  -4  4  0  0  -4  0  0  0  4  0
8  0  0  -8  0  0  0  0  0  0  0  0  0  0  0  0
8  0  -4  0  4  0  -4  -4  0  -4  0  0  0  0  4  0
8  0  0  0  -4  0  0  -4  0  0  0  -8  4  0  0  4
8  0  4  0  0  -4  0  0  0  4  0  0  -4  -4  0  -4
8  -4  -4  8  0  4  4  0  0  -4  -4  0  0  -4  -4  0
8  4  0  0  0  -4  0  0  0  0  4  0  -4  -4  0  -4
8  0  0  0  0  4  4  0  0  0  0  -8  0  -4  -4  0
```

Figure 4.5: C++ Experimental Results of DLCT

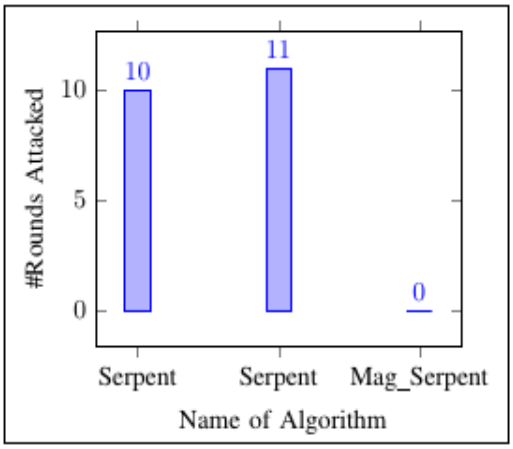


Figure 4.6: DL Attack Outcomes

Table 4.18: Results of feasibility of constructing DLCT before and after 32-Bits-output S-Boxes and Blocker were Applied

Name of Algorithms	**Before 32-Bits-output S-Boxes and Blocker were Applied**	**After 32-Bits-output S-Boxes and Blocker were Applied**
Serpent	The construction of the DLCT was feasible.	The requirement of memory made DLCT construction impossible.

Table 4.19: Results of key discovery before and after 32-Bits-output S-Boxes and Blocker were Applied

Name of Algorithms	**Before 32-Bits-output S-Boxes and Blocker were Applied**	**After 32-Bits-output S-Boxes and Blocker were Applied**
Serpent	In each round, the key was revealed.	There was no key discovery because there was no DLCT or DL attack.

Table 4.20: DL Attack Outcomes

Name of Algorithm	**Time Complexity**	**Data Complexity**	**Rounds Attacked**
Serpent	$2^{115.5}$	$2^{101.2}$	10
Serpent	$2^{231.7}$	$2^{249.4}$	11
Mag_Serpent	∞	∞	0

Table 4.21: When one bit of the key and plaintext was flipped, the avalanche effect occurred

Name of Algorithm	**Key Avalanche Effect in Percentage**	**Plaintext Avalanche Effect in Percentage**
Serpent	49.8657	50.3842
Mag_Serpent	50.5340	49.7985

Table 4.22: Memory Needed for Installation of Algorithms

Name of Algorithm	**Memory Required in Bytes**
Serpent	11181
Mag_Serpent	13206

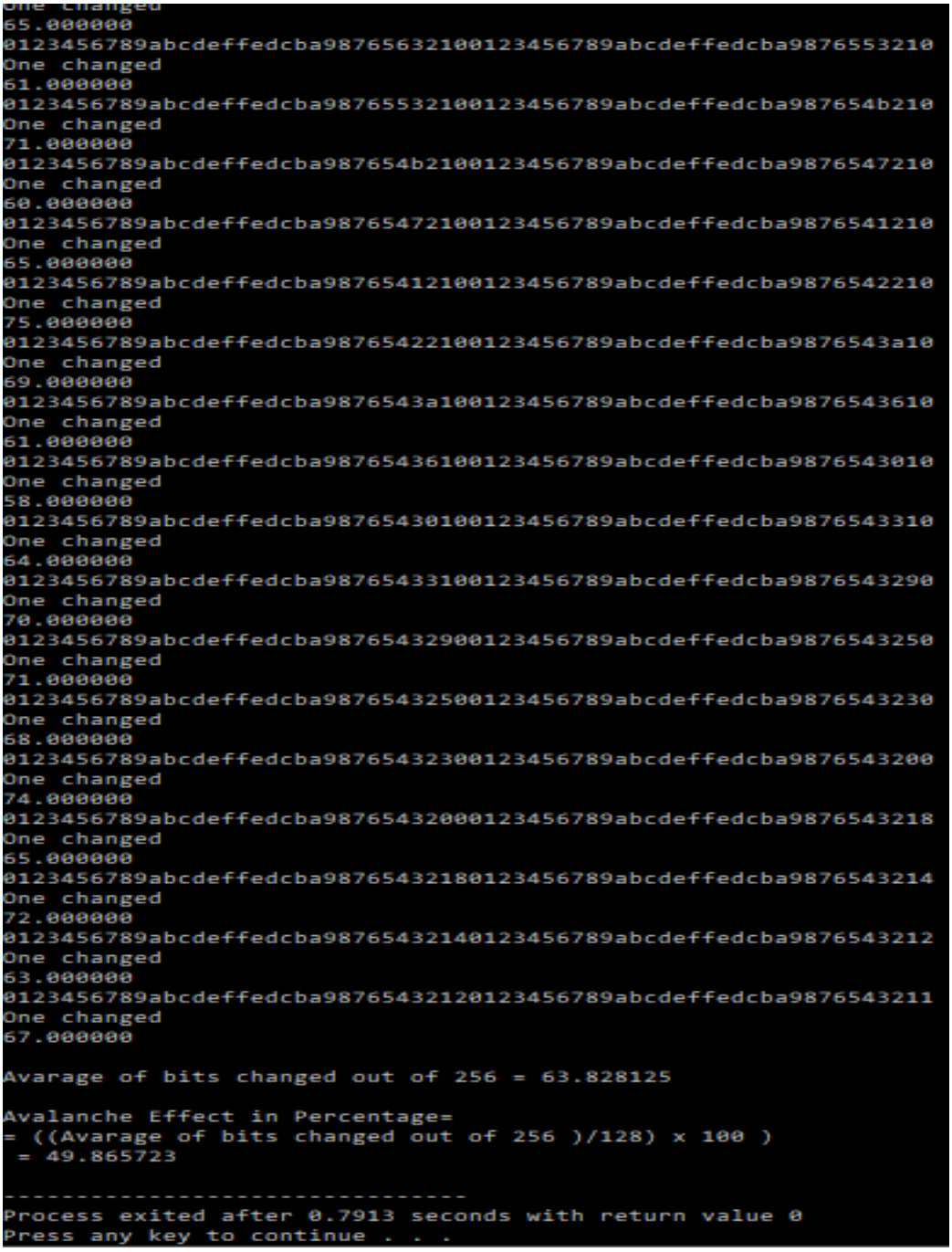


Figure 4.7: Avalanche Experiment Serpent's Effect Whenever One Bit of a Key Was Started Flipping

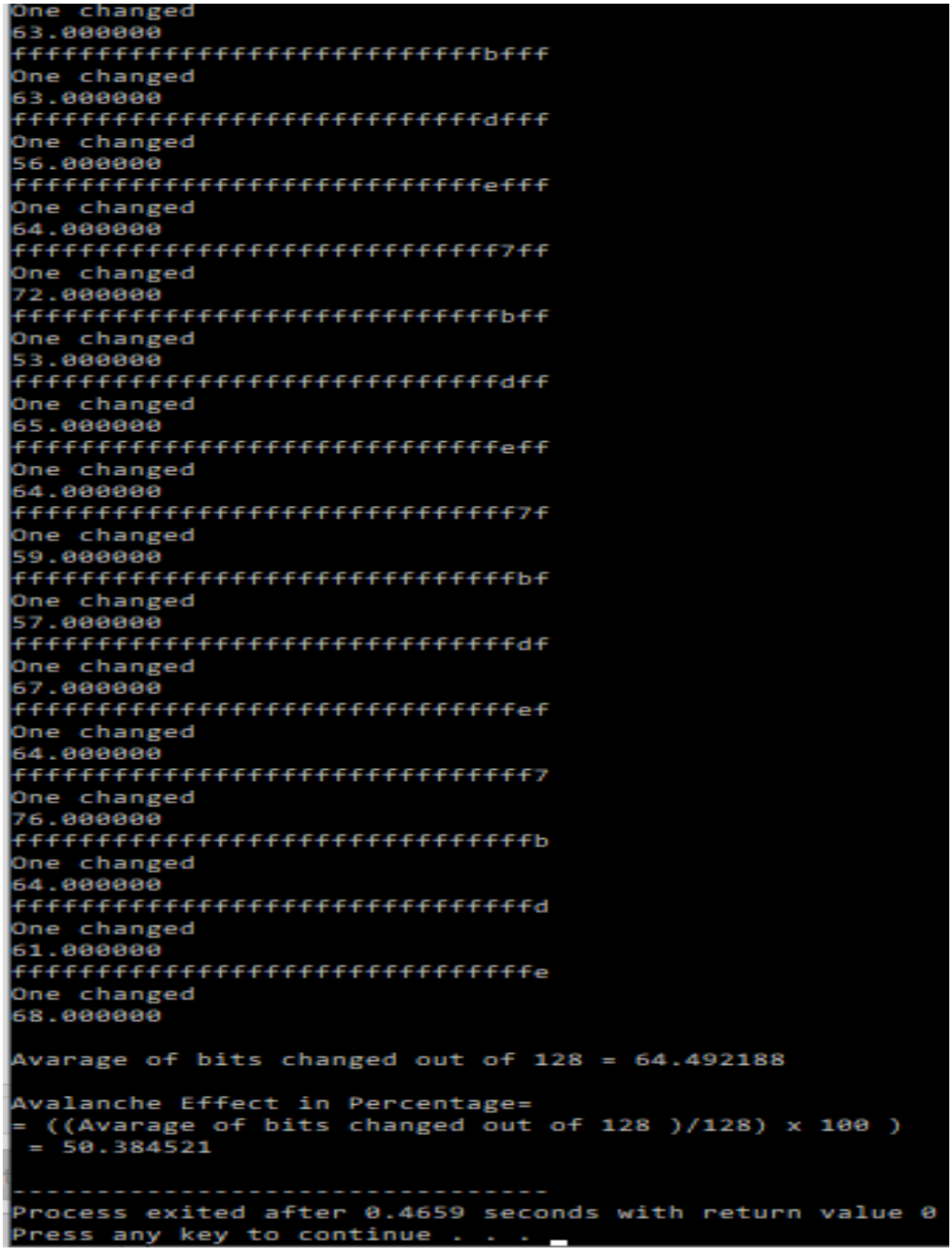


Figure 4.8: Avalanche Experiment Serpent's Effect When One Bit of Plaintext was Flipped

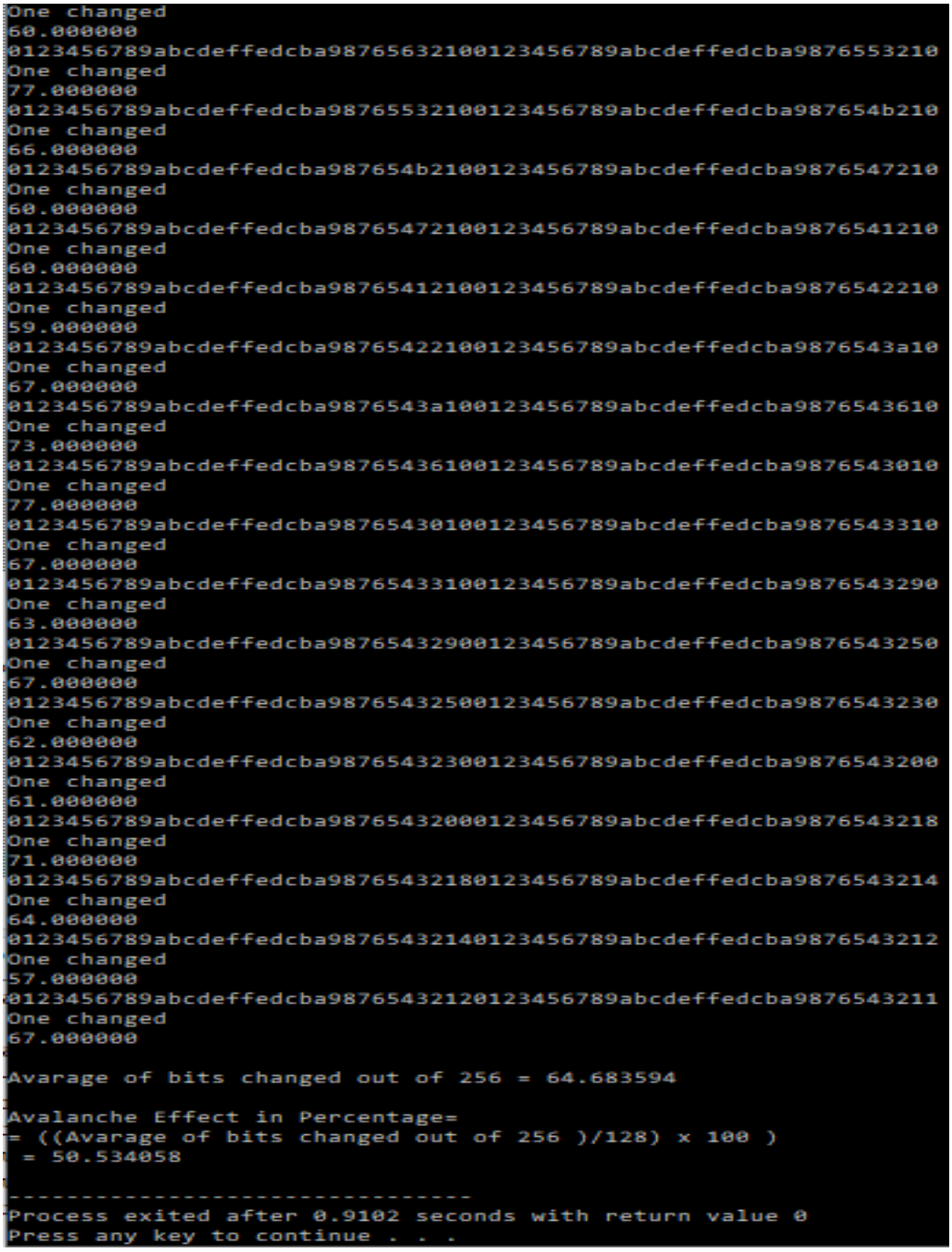


Figure 4.9: The Avalanche Effect of Mag_Serpent When One Bit of a Key Was Flipped

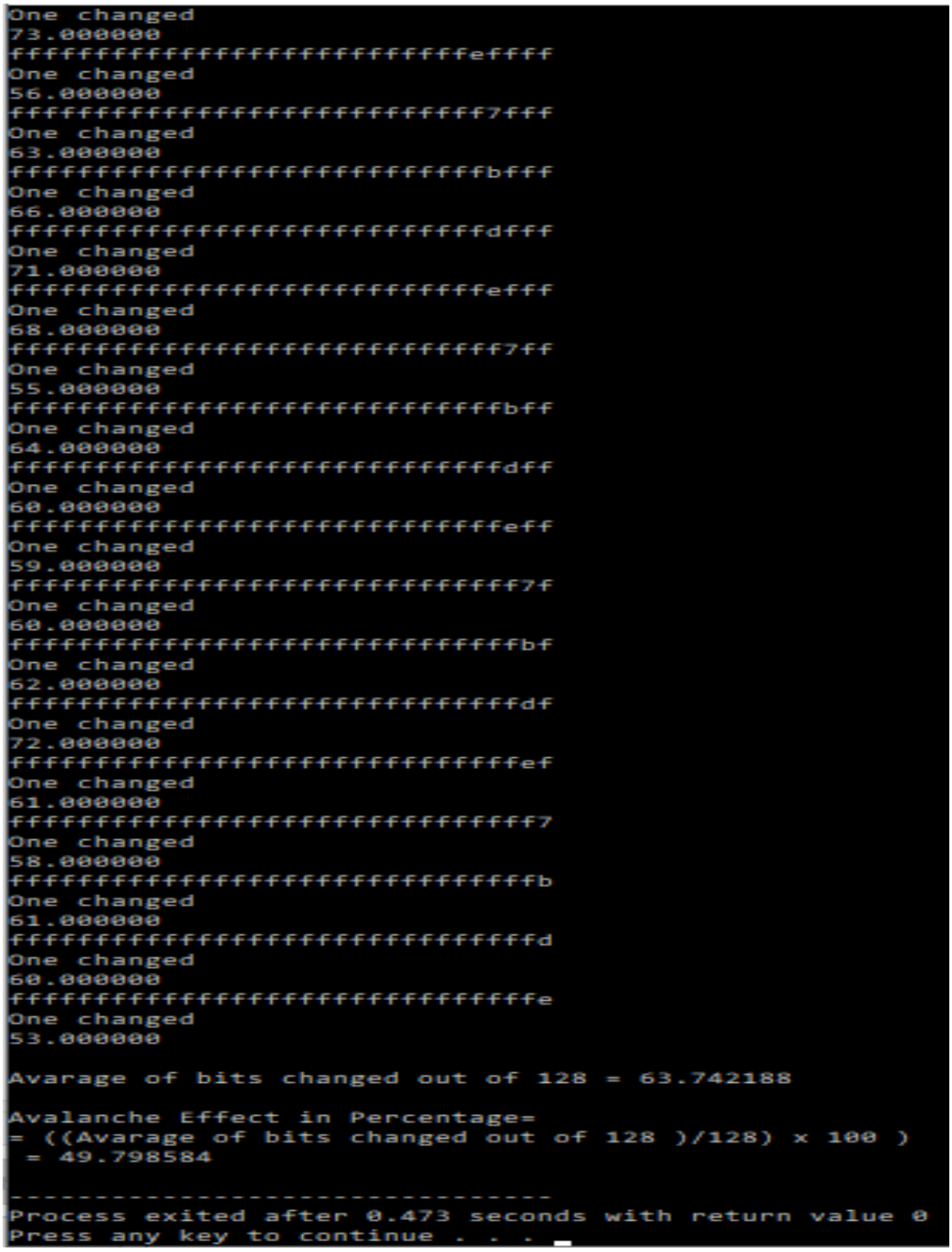


Figure 4.10: The Avalanche Effect of Mag_Serpent when One Bit of Plaintext was Flipped

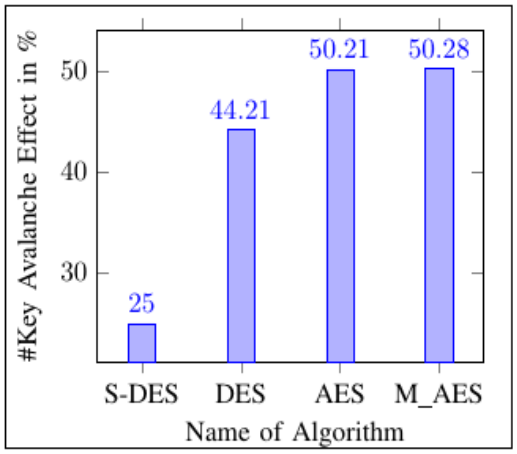


Figure 4.11: Key Avalanche Effect Experiment in Percentage

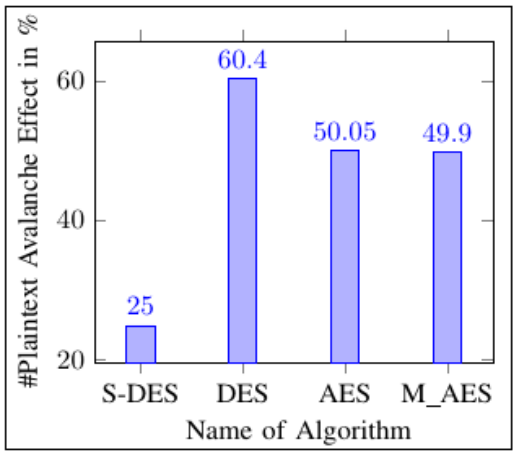


Figure 4.12: Plaintext Avalanche Effect in Percentage Experimental Results

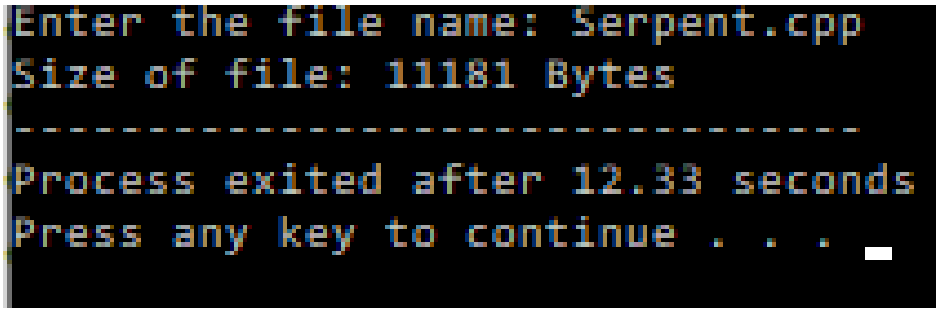


Figure 4.13: Experimental Results Memory for Serpent Installation

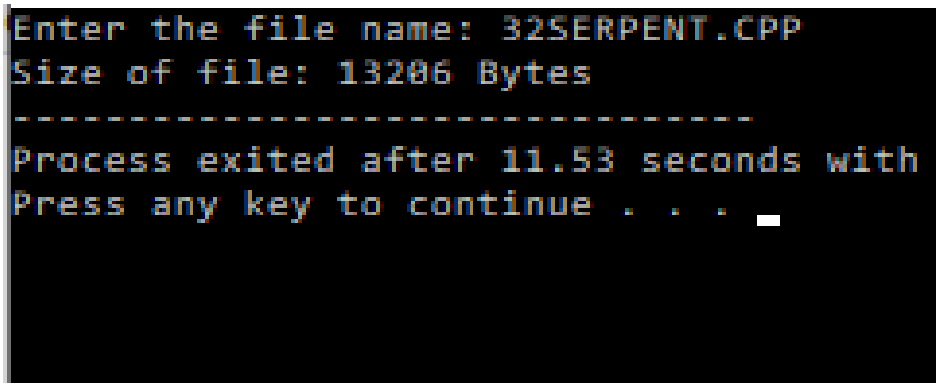


Figure 4.14: Experimental Results Memory for Mag_Serpent Installation

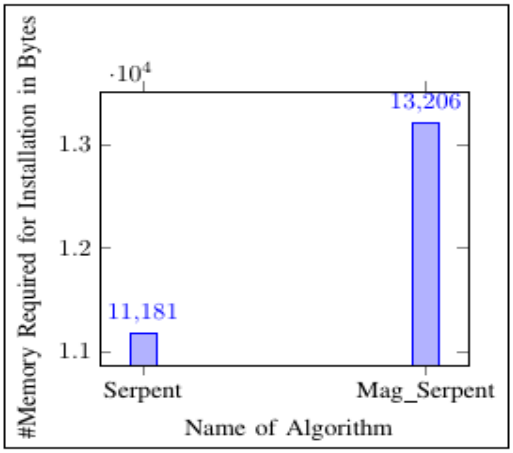


Figure 4.15: Byte Memory Required for Installation

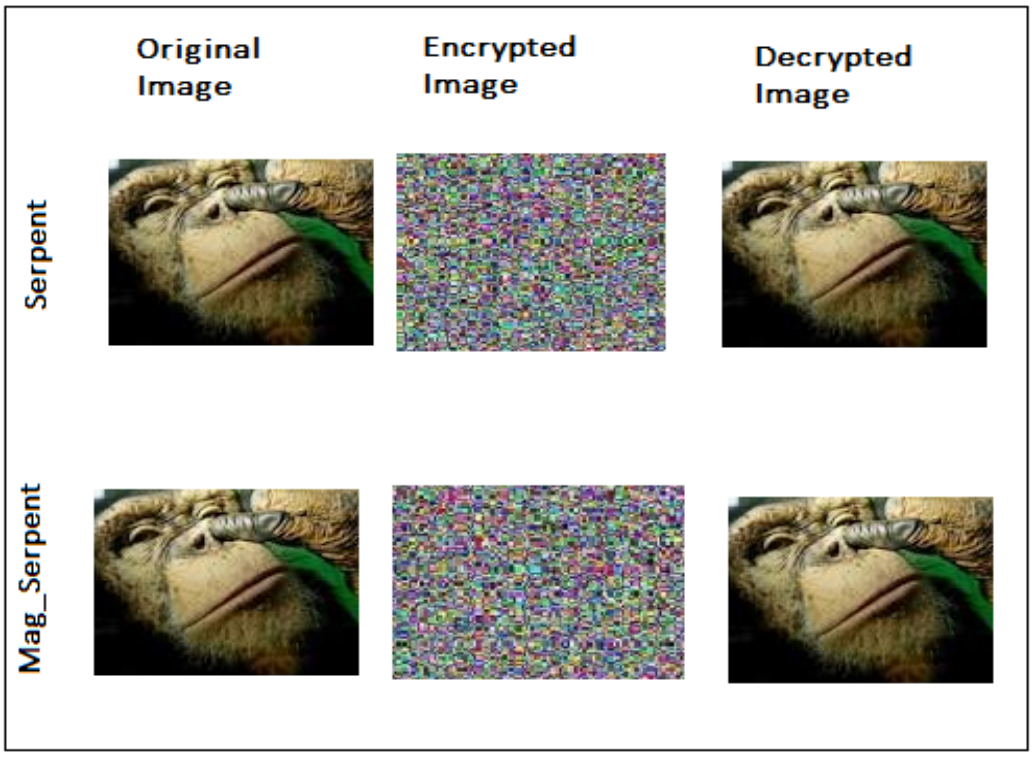


Figure 4.16: Encryption and Decryption Images of Serpent and Mag_Serpent

4.4 Summary of Using Blocker Function to Prevent DL Attack on Serpent

The study demonstrated that using magic numbers and the Blocker function, the Serpent algorithm to secure data stored on IoT devices were protected against DL attacks. It has been confirmed that drawing a DLCT with an output S-Box of $32 - bits$ is impossible. Furthermore, it has been demonstrated that without DLCT, there is no DL attack. In this study, a new modified Serpent named Mag_Serpent was created. Future work will compare the power consumption of Mag_Serpent to that of a standard Serpent.

Chapter 5

The Design of Khumbelo Function on the Camellia Algorithm to Prevent Attacks

The following chapter is based on a two journals submitted by:

i. K. D. Muthavhine and M. Sumbwanyambe, "Applying the Khumbelo Function on the Camellia Algorithm to Proactively Prevent a Wide Range of Attacks on IoT Devices"

   Status: Submitted to IEEE Access.

ii. K. D. Muthavhine and M. Sumbwanyambe, "Conversion of Clefia Algorithm to Decrease Memory Restrictions Encountered on IoT by Applying CMA Method," IEEE, pp. 1-7, 2022.

   Website: *https://ieeexplore.ieee.org/document/9856353*

   Status: Published.

   Publisher: IEEE.

Abstract: Camellia is one of the cryptographic algorithms implemented on many Internet of Things (IoT) devices [99]. However, an intruder uses the

Substitution Box (S-Box) distinguisher to attack her Camellia cipher [100]. A distinguisher is a table that provides the probability of guessing the algorithm's secret key [99]. Distinguisher's features are used in most attacks. The most well-known characteristic features are the Linear Approximation Table (LAT), Difference Distribution Table (DDT), and Differential Linear Connections Table (DLCT) [100]. This work focuses on preventing these attacks by deflecting the construction of an S-Box distinguisher with a new function called the Khumbelo. The Khumbelo function prevented distinguisher construction by lowering the construction probability. The Khumbelo function successfully reduced the attack probability of LAT (54.6875 percent to 0 percent), DDT (1.5625 percent to 0 percent), and DLCT (50.0000 percent to 0 percent). The Khumbelo function is generated using a 4-Byte output S-Box instead of Camellia's original 1-Byte output S-box. Also, the Khumbelo function consists of many modulo operators. New 4-Byte output S-boxes and modulo operators confuse and block intruders to build distinguishers. After successfully embedding the Khumbelo function in the traditional camellia, the newly modified camellia was coined K_Camellia.

5.1 Introduction

The Internet of Things (IoT) is a modern critical system of connecting devices, things, people, objects, digital machines, computers, and objects to transmit information [179], [99]. IoT is emerging as an effective technology with devices in key areas for creating needed networks. Depending on the current task, IoT customers use IoT networks [99], [180]. The company uses IoT devices and networks to exchange classified information [99], [181]. Some of this information is transmitted unencrypted or encrypted by algorithms vulnerable to attacks [99], [181].

Encryption algorithms are primarily used to ensure security and privacy [181]. One of the cryptographic algorithms used in IoT devices is Camellia. Section 1A shows that various types of IoT devices use Camel-

lia. Some cryptographic algorithms used by IoT devices have been compromised, such as Camellia. Encryption algorithms are unavoidable but should be pre-designed to withstand attacks [99], [181]. Many organizations ask researchers to develop algorithms resistant to attacks [182]. For example, Sleem [183] actively pointed out that improving encryption algorithms is an ongoing process, as intruders are cracking known algorithms on IoT devices. The Federal Information Security Agency has pointed out that the fact that the algorithm has recently proven to be robust does not mean that intruders will not attack it in the future [184].

Current research shows that intruders use various attack methods to attack the Camellia algorithm. For example:

i. LAT is used in LC attacks. See Table 4, page 11 of [100].

ii. DDT is used for DC attacks. See [100], page 21, Table 7.

iii. Attack DDT with a boomerang. refer [185], page 4.

iv. The truncated derivative is a variant of the DC attack [186]. Moreover, the shortened differential succeeds with DDT. The previous statement is also on page VII of [186].

v. The Man-in-the-Middle attack is a variant of the DC attack [187]. Additionally, as pointed out in [188] on page 136, Man-in-the-Middle attacks persist from DDT.

vi. The zero-correlation linear attack is a variant of the LC attack that uses LAT as the LC attack. Page 119 of [187]. [189] also supports the previous statement.

vii. DL attack is based on DLCT [97].

This work focuses on preventing these attacks by diverting the construction of S-Box distinguishers using a new function called the Khumbelo function.

The Khumbelo function hindered the construction of distinguishers by reducing the probability of guessing the private key. The Khumbelo function successfully reduced the attack probability of LAT (54.6875 percent to 0 percent), DDT (1.5625 percent to 0 percent), and DLCT (50.0000 percent to 0 percent). Table 5.1 shows various attacks on these properties. The Khumbelo function is generated using a 4-Byte output S-Box instead of the original 1-Byte output S-box in standard Camellia. The Khumbelo function also consists of several modulo operators. New 4-Byte output S-Boxes and modulo operators confuse and block intruders to build distinguishers. After successfully applying the Khumbelo function to the traditional Camellia, a new modified Camellia was coined K_Camellia. In this study, conventional (standard) Camellia is called Camellia. K_Camellia is a new algorithm first created and defined in this work (novelty) after embedding the Khumbelo function into Camellia.

Table 5.1: Attacks and Distinguishers

Attacks	**Distinguishers**
The Linear Cryptanalysis (LC) Attack	LAT
The Differential Cryptanalysis (DC) Attack	DDT
The Boomerang Attack	DDT
The Truncated Differential Attack	DDT
Meet-in-the-middle Attack	DDT
Zero-Correlation Linear Distinguisher Attacks	LAT
The Differential Linear (DL) Attacks	DLCT

5.1.1 Relationship between IoT devices and Camellia Algorithm

Camellia algorithm is used for privacy and security in many IoT devices, including smart homes, e-health, e-commerce, smart hospitals, and smart cities [190]. For instance, the IoT devices embed a Camellia algorithm in 180 nm Complementary metal-oxide-semiconductor (CMOS) technology for different key lengths [190].

IoT devices like Hardware (MICAz mote) and Riverbed (OPNET) Modeler use Camellia. IoT platform uses hardware (MICAz mote) to enable low-power and wireless sensor networks [191]. Riverbed (OPNET) Modeler is a modeling and analysis environment for communication networks and distributed systems [191]. For security and privacy, hardware (MICAz mote) and Riverbed (OPNET) embed a Camellia algorithm in Random Access Memory (RAM) and Read-Only Memory (ROM) [191].

Some IoT devices which use Camellia are the Blockchain-Based Secure Centralized Big Data (BOBS CRABID) model. The BOBS CRABID model is made up of three parts. Reduced for Data Clustering, Data Collection and Processing from IoT Devices, and Multi-Factor Authentication [192]. A Camellia algorithm is used in Multi-Factor authentication to authenticate all devices using Internet Protocol (IP) addresses, Identification (ID), Media Access Control (MAC) addresses, and Physical Unclonable Functions (PUF) [192].

Actuators and sensors are two other IoT-based devices that use Camellia. In an Edge Enabled Smart Grid Network, semi-quantum key distribution (SQKD) and an intrusion detection system (IDS) use actuators and sensors to communicate [193]. During the communication process, SQKD distributes the encryption key generated by the Camellia algorithm [193].

According to the preceding statements, there is a connection between IoT devices and the Camellia algorithm. The relationship is based on confidentiality, security during communication, and data transmission [190]- [193].

5.1.2 Cryptographic Attacks

In this section, different cryptographic attacks are explained in detail as follows:

5.1.2.1 The Linear Cryptanalysis (LC) Attack

The LC attack exploits the most likely linearity conditions affecting plaintext, ciphertext, and subkey bits [100], following linear analysis of the S-Box [100]. The intruder creates a probability table known as LAT. The intruder employs LAT to guess the secret keys easily. No LC attack will succeed without LAT [194]. When the intruder is carrying out the LC attack, LAT is a critical distinguisher needed [100]. Most attacks rely on the distinguishers. For instance, the distinguisher of the LC attack is LAT. These distinguishers are probability tables to assist intruders in guessing secret keys of cryptographic algorithms like Camellia and DES. Table 5.1 shows different attacks related to their distinguishers.

The study summarized LC attacks using Simplified Data Encryption Standard's S-Box (S-DES). There is a W-Bit key block for every V-Bit of input (plaintext) and output (ciphertext) blocks, so plaintext is $\varrho = \varrho_1, \varrho_2, ..., \varrho_{V-1}, \varrho_V$, the ciphertext is $\subset = \subset_1, \subset_2, ..., \subset_{V-1}, \subset_V$, and the key is $K = K_1, K_2, ..., K_{W-1}, K_W$. Then specify

$$\varrho[\gamma 1, \gamma 2, ..., \gamma a] \oplus \subset [\eta 1, \eta 2, ..., \eta b] = K[\psi 1, \psi 2, ..., \psi c] \tag{5.1}$$

Where x =1 or 0, $1 \leqslant \gamma$, $b \leqslant V$, $1 \leqslant c \leqslant W$, and γ, η and ψ represent bit locations with a probability of $\varrho \neq 0.5$. Furthermore, ϱ is calculated from LAT, with values ranging from 0.5 to 1. The correlation is then calculated; if it is 0 or 1, assume that $K[\psi 1, \psi 2, ..., \psi c] = 0$ or $K[\psi 1, \psi 2, ..., \psi c] = 1$, respectively. To obtain key bits, solve $K[\psi 1, \psi 2, ..., \psi c] = 0$ and $K[\psi 1, \psi 2, ..., \psi c] = 1$ simultaneously using chosen plaintext/ciphertext.

A zero-correlation linear distinguisher is also a variant of the LC attack [195]. Liu, Sun, Wang, Varici, and Gu [195] recovered the secret key after 14-round of Camellia-256 and 13 rounds of Camellia-192. An S-Box of Cipher is

considered substantial (or attack-resistant) if the output bits vary with (1/2) probability whenever one bit is flipped [194].

5.1.2.2 The Differential Cryptanalysis (DC) Attack

The DC attack is carried out as follows: Intruders consider an S-Box of cipher to be vital if the output bits show a considerable variation with (1/2) probability whenever any input bit is complemented [180], [196]. In light of S- Box's flaws, intruders can efficiently compute the highest difference probabilities of $(\Delta\varrho_j, \Delta \subset_j)$ of $(1/(2^m))$ pairwise comparisons, where m is the output and j is the j^{th} bit of ϱ_j and $\subset_j$, respectively [180] [196]. The highest difference probabilities of each S-Box pair $(\Delta\varrho_j, \Delta \subset_j)$ of $(1/(2^m))$ is incorporated and applied in a different number of rounds until the intruders recover the secret key [180]. Note that the boomerang, truncated differentials, Meet-in-the-Middle, and impossible differential attacks are all variants of DC attacks because they all use the DDT [196]. The intruder employs DDT to quickly guess the secret keys. No DC attack will succeed without DDT [180], [196].

Page 21 of [100] gave the theory of constructing the first DDT of DES's first S-Box. This study analyzed the theory of constructing the DDT using the experimental C++ code of DDT to compare and verify the code and the results on page 21 of [100]. The study conducted the DDT's experimental C++ code. The results corresponded to the theoretical results given DDT on page 21 of [100]. The DC attack on DES was carried out in the study using DDT and Equation 5.2. When $K \oplus \varrho_i$ and $K \oplus \varrho_j$ are 4 and 8, respectively, and ϱ_i and ϱ_j are 2 and 4, respectively, the probability of guessing the correct key using DDT is 2. That is eight 2 or 4 digits. The study tested 2 and 4 to find the correct key and discovered that the valid key is 4. According to Khurana and Kumari [187], boomerang and truncated differential attacks are variants of DC attacks. The study used boomerang and truncated differential attacks on Camellia, which were influenced by Khurana and Kumari [187].

$$\begin{aligned} &\Delta SBox(K \oplus \varrho_i) \oplus SBox(K \oplus \varrho_j) = \varrho_i \oplus \varrho_j \\ &where\ \varrho_i = \varrho_i,\ \varrho_j = \varrho_j\ and\ K = Key\ to\ be\ predicated \end{aligned} \tag{5.2}$$

Liu, Sun, Wang, Varici, and Gu [195] recovered the secret key after 14-round of Camellia-256 and 13 rounds of Camellia-192.

5.1.2.3 The Boomerang Attack

The boomerang attack is carried out as follows: The encryption scheme is split into two shorter cryptosystems, E_0 and E_1 [197] - [200]. For intruders to apply the boomerang attack, the differential probability of S-Box must be high, greater than 50% [199] [200]. Assume the intruder divided a 10-round Camelia S-Box into E_0 and E_1 with r and $10 - r$ rounds, respectively [197], [198]. The differential probability of each sub-characteristic S-Box will be bounded by $Prob_0 \leq (2^{-30})^{r/4}$ and $Prob_1 \leq (2^{-30})^{(10-r)/4}$. Where $Prob_0$ and $Prob_1$ are S-Box probabilities derived from E_0 and E_1, respectively [197], [198]. If there are high probabilities, intruders can easily apply the boomerang attack [199], [200].

According to Lu, Wei, Pasalic, and Fouque [199], the Camellia is attackable for up to 9 rounds using the Boomerang attack. The Boomerang attack uses the probability of distinguishers from the algorithm's poor S-Boxes [199]. The boomerang attack is a variation (derived from) of the DC attack [187]. Furthermore, the boomerang attack thrives on DDT, as stated on page 4 of [185].

5.1.2.4 The Truncated Differential Attack

The concept of truncated differential attack is an extension of the concept of DC attack [201]. The DC attack looks for the fundamental difference between two plaintexts, while the truncated variant looks for only determined differences [202]. The truncated differential attack generates predictions from

only a subset of the block rather than the entire block. The truncated differential attack is a variation (derivation) of the DC attack [187]. The truncated differential attack is carried out as follows [201]:

i. The study calculated the differential probability of a useful active S-Box.

ii. XOR probability of useful active S-Box outputs.

iii. Create a table of DDT.

iv. Create the additional (mathematical or statistical) functions required to complete the attack.

When intruders use the truncated differential attack, they frequently target the S-Box. Li, Jia, Wang, and Dong [202] used the truncated differential attack on Camellia and were successful in attacking 11-round and 12-round Camellia with $2^{121.3}$ and $2^{185.3}$ plaintexts, respectively. According to [202]'s work, the truncated differential attack is the same as the impossible attack. The truncated differential is a variation of the DC attack [186]. Furthermore, truncated differential thrives on DDT; the preceding statement is also given on page vii of [186].

5.1.2.5 Meet-in-the-Middle Attack

The Meet-in-the-Middle attack divides the input of the cipher's building block X into two distinct segments, such that $X = X_1 \bullet X_0$. Other attacks, such as the boomerang and truncated differential attacks, are then used to complete the process [202]. The Meet-in-the-Middle attack is a variant of the DC attack [187]. Furthermore, the Meet-in-the-Middle attack thrives on DDT, as stated on page 136 of [188].

Li, Jia, Wang, and Dong [202] used the truncated differential attack on Camellia and were successful in attacking 11-round and 12-round Camellia with $2^{121.3}$ and $2^{185.3}$ plaintexts, respectively.

5.1.2.6 Zero-Correlation Linear Distinguisher Attacks

The zero-correlation distinguisher employs a linear hull with correlation zero [203]. In contrast to the standard LC attack, which employs linear characteristics with high correlations, this attack is known as linear parameterization with zero correlation [195]. The LC attack extension of impossible differential cryptanalysis is a zero-correlation linear attack [203]. S-Box is involved in the attack process because it is the LC attack extension of impossible differential cryptanalysis.

The intruder can also employ similar Meet-in-the-Middle technology. Furthermore, the intruder can use impossible differential cryptanalysis to create a zero-correlation linear distinguisher [195], [203]. Liu, Sun, Wang, Varici, and Gu [195] recovered the secret key after 14-round of Camellia-256 and 13 rounds of Camellia-192. A zero-correlation linear attack is a variation of the LC attack [187]. The preceding statement is also supported by [189] on page 119.

5.2 Objective of the Study

Using the Khumbelo function in Camellia, this study aims to distract the construction of distinguishers (attack engines) from preventing the spread of attacks. To accomplish the following goals:

i. To prevent attacks by using a novel function called the Khumbelo function to distract the construction of S-Box distinguishers.

ii. To distract the distinguishers' construction by reducing the probability of construction using the Khumbelo function.

iii. To successfully reduce the attack probability of LAT (from 54.6875 percent to 0 percent), DDT (from 1.5625 percent to 0 percent), and DLCT (from 50.0000 percent to 0 percent) using the Khumbelo function.

iv. To generate a 4-Byte output S-Box for a new algorithm using the Khumbelo function instead of an original 1-Byte output S-Box in Camellia, which is vulnerable to many attacks.

v. To build the Khumbelo function using many irreversible modulo operators.

vi. To confuse and block intruders from constructing distinguishers and attacks using the new 4-Byte output S-Box and modulo operators from the Khumbelo function.

vii. To successfully embed the Khumbelo function in the traditional Camellia, get a newly modified Camellia.

viii. To coin and define the newly modified Camellia as K_Camellia, first and solely inverted in this study.

5.3 Review of the Camellia Algorithm

Camellia is a symmetric algorithm developed by Telephone (NTT) Corp, Nippon Telegraph, and Mitsubishi Electric Corporation (MEC) [204], [207]. Camellia uses key lengths of 128 bits, 192 bits, and 256 bits to improve software and hardware performance on standard 32-Bit and 8-Bit processors (for example, smart cards, embedded systems, OpenPGP, and IPSec) [204], [206], [207].

Camellia is a Feistel-based cipher with either 18 or 24 rounds depending on the key used [205]. Every six rounds, the self-styled FL function (a rational conversion layer) or its inverse is applied [205], [206]. The cipher includes four 8 x 8-Bit S-Boxes, including output, input affine transformations, and mathematical functions [205]. This cipher also uses input and output keys for the whitening process [205], [206].

Camellia is a Feistel-structured cryptographic algorithm, and the round function has an SPN structure [204] [206], [207]. Figure 5.1 depicts how

FL and $(FL)^{-1}$- functions are incorporated between each 6-round of a cryptosystem [205] [206].

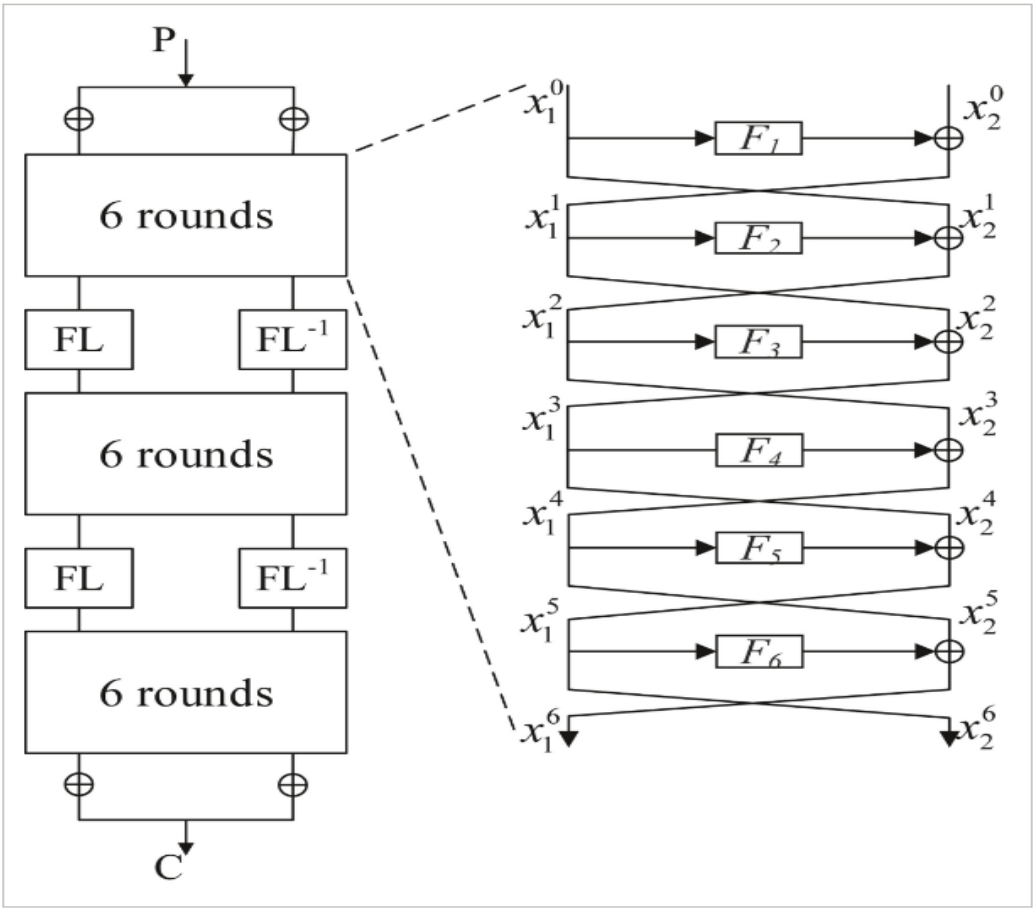


Figure 5.1: Camellia Structure [206]

The linear FL and $(FL)^{-1}$ - functions are not included in the specific Feistel structure cited in [206], [207]. Figure 5.2 depicts the architecture of the encryption process as a Camellia round function. The S transformation component contains four types of 8 x 8 S-Boxes (1-Byte output S-Box), which are described below: $SBox_1(XY)$, $SBox_2(XY)$, $SBox_3(XY)$, and $SBox_4(XY)$ given in Table 5.2, where input X and Y intersect the 1-Byte

output is generated.

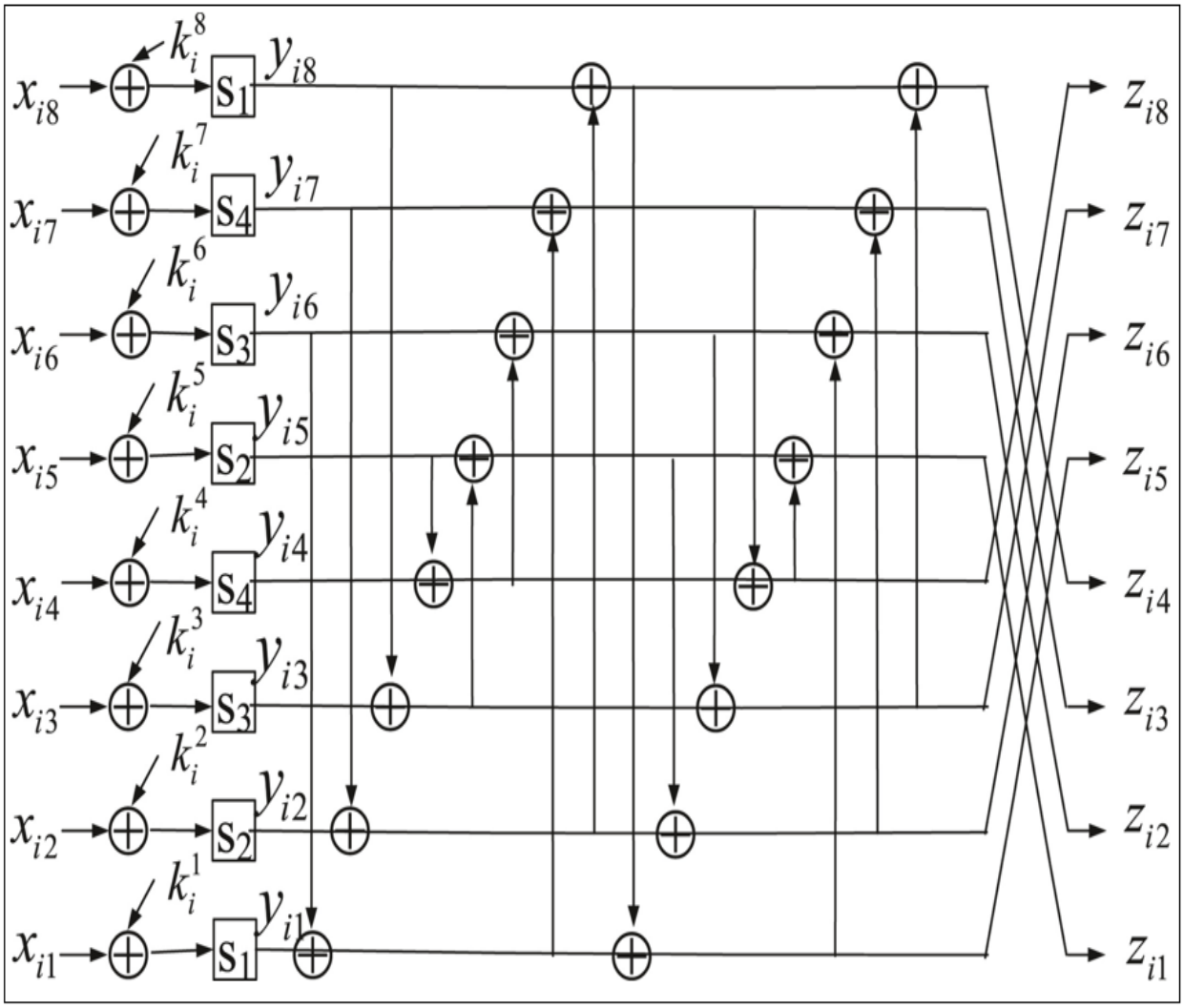


Figure 5.2: Camellia Round Function [206]

For instance, if $X = 0$ in hexadecimal and $Y = 3$ in hexadecimal, then input $XY = 03$ in hexadecimal. To generate the 1-Byte ouput of $XY = 03$, 0 and 3 intersect at 236 in the $SBox_1(XY)$, therefore, $SBox_1(XY) = SBox_1(03) = 236$ in binary which is 1-Byte, as shown in Table 5.2. $SBox_2(XY)$,

Table 5.2: First S-Box ($SBox_1$) of Camellia

Input X	**Input** Y															
	0.	**1.**	**2.**	**3.**	**4.**	**5.**	**6.**	**7.**	**8.**	**9.**	**A.**	**B.**	**C.**	**D.**	**E.**	**F.**
0.	112.	130.	44.	236.	179.	39.	192.	229.	228.	133.	87.	53.	234.	12.	174.	65.
1.	35.	239.	107.	147.	69.	25.	165.	33.	257.	14.	79.	78.	29.	101.	146.	189.
2.	134.	184.	175.	143.	124.	235.	31.	206.	62.	48.	220.	95.	94.	197.	11.	26.
3.	166.	225.	57.	202.	213.	71.	93.	61.	217.	1.	90.	214.	81.	86.	108.	77.
4.	147.	13.	154.	102.	251.	204.	176.	45.	116.	18.	43.	32.	240.	177.	132.	153.
5.	223.	76.	203.	194.	52.	126.	118.	5.	109.	183.	169.	49.	209.	23.	4.	215.
6.	20.	88.	58.	97.	222.	27.	17.	28.	50.	15.	156.	22.	83.	24.	242.	34.
7.	254.	68.	207.	178.	195.	181.	122.	145.	36.	8.	232.	168.	96.	252.	105.	80.
8.	170.	208.	160.	125.	161.	137.	98.	151.	84.	91.	30.	149.	224.	225.	100.	210.
9.	16.	196.	0.	72.	163.	247.	117.	219.	138.	3.	230.	218.	9.	63.	221.	148.
A.	135.	92.	131.	2.	205.	74.	144.	51.	115.	103.	246.	243.	157.	127.	191.	226.
B.	82.	155.	216.	38.	200.	55.	198.	59.	129.	150.	111.	75.	19.	190.	99.	46.
C.	233.	121.	169.	140.	159.	110.	188.	142.	41.	245.	249.	182.	47.	253.	180.	99.
D.	120.	152.	6.	106.	231.	70.	113.	186.	212.	37.	171.	66.	136.	162.	141.	250.
E.	114.	7.	185.	85.	248.	238.	172.	10.	54.	73.	42.	104.	60.	56.	241.	164.
F.	64.	40.	211.	123.	187.	201.	67.	193.	21.	227	173.	244.	119.	199.	128.	158.

$SBox_3(XY)$, and $SBox_4(XY)$ are derived as follows from $SBox_1(XY)$:

$SBox_2(XY) = SBox_1(XY) <<< 1$;

$SBox_3(XY) = SBox_1(XY) >>> 1$;

$SBox_4(XY) = SBox_1(XY <<< 1)$; Where $<<<$ and $>>>$ represent right and left rotational shifts, respectively.

The S transformation component is made up of $z_1, z_2, ..., z_8$ and $y_1, y_2, ..., y_8$, which are mathematically defined as follows:

$z_1 = y_7 \oplus y_8 \oplus y_4 \oplus y_6 \oplus y_3 \oplus y_1$;

$z_2 = y_7 \oplus y_8 \oplus y_1 \oplus y_4 \oplus y_3 \oplus y_6$;

$z_3 = y_6 \oplus y_8 \oplus y_1 \oplus y_3 \oplus y_2 \oplus y_5$;

$z_4 = y_6 \oplus y_7 \oplus y_2 \oplus y_4 \oplus y_3 \oplus y_5$;

$z_5 = y_8 \oplus y_7 \oplus y_6 \oplus y_2 \oplus y_1$;

$z_6 = y_7 \oplus y_8 \oplus y_2 \oplus y_3 \oplus y_5$;

$z_7 = y_5 \oplus y_6 \oplus y_8 \oplus y_4 \oplus y_3$;

$z_8 = y_5 \oplus y_6 \oplus y_7 \oplus y_4 \oplus y_1$;

$y_1 = z_6 \oplus z_7 \oplus z_8 \oplus z_4 \oplus z_3 \oplus z_1$;

$y_2 = z_5 \oplus z_7 \oplus z_8 \oplus z_4 \oplus z_3 \oplus z_1$;

$y_3 = z_5 \oplus z_6 \oplus z_8 \oplus z_4 \oplus z_2 \oplus z_1$;

$y_4 = z_5 \oplus z_6 \oplus z_7 \oplus z_3 \oplus z_2 \oplus z_1$;

$y_5 = z_5 \oplus z_7 \oplus z_8 \oplus z_2 \oplus z_1$;

$y_6 = z_5 \oplus z_6 \oplus z_8 \oplus z_3 \oplus z_2$;

$y_6 = z_5 \oplus z_6 \oplus z_8 \oplus z_3 \oplus z_2$;

$y_8 = z_6 \oplus z_7 \oplus z_8 \oplus z_4 \oplus z_1$;

The P function is depicted in Figure 5.1. The P function is an 8 x 8 matrix with an inverse P^{-1} as shown in Figure 5.3. The key generation discussion is not included in this study because the process is the same as shown in Figure 5.1. The reader should consult other literature reviews for

more information on the key generation process. The P and P^{-1} matrices are used during the encryption and decryption processes. Since Camellia is a block cipher, in the decryption process, the round keys are applied in reverse order compared to the encryption process.

$$P = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \quad P^{-1} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Figure 5.3: P Function and its Inverse P^{-1} [206]

5.4 Contribution of the study

Using the Khumbelo function in Camellia, the study's contribution aims to distract the construction of distinguishers (attack engines) from the spread of attacks and to secure IoT devices using K_Camellia. To accomplish the following goals:

i. To give awareness of preventing attacks by using a novel function called the Khumbelo function to distract the construction of S-Box distinguishers.

ii. To give awareness of the procedure of distracting the distinguishers' construction by reducing the probability of construction using the Khumbelo function.

iii. To give awareness of the procedure of successfully reducing the attack probability of LAT (from 54.6875 percent to 0 percent), DDT (from 1.5625 percent to 0 percent), and DLCT (from 50.0000 percent to 0 percent) using the Khumbelo function.

iv. To give awareness of generating a 4-Byte output S-Box for a new algorithm using the Khumbelo function instead of an original 1-Byte output S-Box in Camellia, which is vulnerable to many attacks.

v. To give awareness of building the Khumbelo function using many irreversible modulo operators.

vi. To confuse and block intruders from constructing distinguishers and attacks using the new 4-Byte output S-Box and modulo operators from the Khumbelo function.

vii. To give awareness of the procedure of successfully embedding the Khumbelo function in the traditional Camellia, get a newly modified Camellia.

viii. To coin and define the newly modified Camellia as K_Camellia, which is resistant to many attacks. K_Camellia is firstly and solely inverted in this study.

5.5 The Overview of the Khumbelo Function

The Khumbelo function employs four 8 x 32 S-Boxes (4-Byte output S-Boxes) rather than the four 8 x 8 S-Boxes (1-Byte output S-Boxes) defined in the original Camellia, which are vulnerable to many attacks. All S-Boxes are converted to four 8 x 32 S-Boxes using the degree 32 polynomial with mod

2^{32} during the application of the Khumbelo function. The procedure is as follows:

i. Using Equation 5.3 to create a polynomial of 32 degrees.

$$\begin{aligned} & 0 + 1 + x + x^2 + x^{22} + x^{32} \\ & = 80200007_{hexidecimal} \\ & = 2149580807_{decimal} \end{aligned} \tag{5.3}$$

ii. Create entities of the first 8 x 32 S-Box using Equation 5.3 as follows:

First entity = $2149580807_{decimal} * Random(2149580807_{decimal} + 1) mod 2^{32}$

Second entity = $2149580807_{decimal} * Random(2149580807_{decimal} + 2) mod 2^{32}$

Third entity = $2149580807_{decimal} * Random(2149580807_{decimal} + 3) mod 2^{32}$, until to the last entity.

Last entity = $2149580807_{decimal} * Random(2149580807_{decimal} + 256) mod 2^{32}$.

iii. Then, after creating the first new 8 x 32 S-Box, say $SBox_1(XY)$, the following three S-Boxes are derived from the newly created S-Box, that is, $SBox_1(XY)$:

$SBox_2(XY) = SBox_1(XY) >>> 2$;

$SBox_3(XY) = SBox_1(XY) >>> 2$;

$SBox_4(XY) = SBox_1(XY >>> 2)$;

Whereas the $>>> 2$ operator is a (twice right rotational shift of bits), the original Camelia used $<<< 1$ (once left rotational shift of bits). Considering that they are derived from the created 8 x 32 $SBox_1(XY)$, then, $SBox_2(XY)$, $SBox_3(XY)$, and $SBox_4(XY)$ are now 8 x 32 S-Boxes.

iv. Create a for loop that runs from 0 to 7, as shown below: $for(i)$ where $i = 0, 1, 2, ..., 7$ $FirstInput = X[i] \oplus k[7 - i]$ where $k[i]$ are Y subkeys

and X[i] is any input that is required to be used as $SBox_i(XY)$ input Y. This is equal to $X[i] \oplus k[7-i] = XY$.

v. Replace $SBox_i(XY)$ with $SBox_i(FirstInput)$. Take note that $SBox_i(FirstInput)$ is now 32-Bit, and generate four 8-Bit from 32-Bit as $v[i]$ where $i = 0, 1, 2, 3, 4$. Keep $v[i]$ for the Khumbelo function to return the value. The $v[i]$ are generated by mod 32 and dividing 32-Bit segments. Since the modulo operator is irreversible, Modulo 32 confuses intruders from constructing the distinguishers.

vi. $SBox_i(XY)$ should be changed to $SBox_i(SecondInput)$, where $SecondInput = z_1 \oplus z_2 \oplus z_3 \oplus z_4$. Keep in mind that $SBox_i(SecondInput)$ is now 32-Bit; generate four 8-Bit from 32-Bit as $W[i]$, where $i = 0, 1, 2, 3, 4$. The $W[i]$ are generated by dividing 32-Bit into segments and using mod 32. Since the modulo operator is irreversible, Modulo 32 confuses intruders from conducting the attacks. Save the value of $W[i]$ to be used in the Khumbelo function.

vii. The Khumbelo function is defined above as a function that returns $v[i]$ and $W[i]$ using 8 x 32 S-Boxes. Fiestel function will call the $v[i]$ and $W[i]$ variables.

viii. Modify the original Camellia's Feistel function to use the $v[i]$ and $W[i]$ from the Khumbelo function as follows:

$z_1 = y_8 \oplus y_7 \oplus y_6 \oplus v[4] \oplus v[3] \oplus v[1]$

$z_2 = y_8 \oplus y_7 \oplus y_6 \oplus v[4] \oplus v[3] \oplus v[1]$

$z_3 = y_8 \oplus y_6 \oplus y_5 \oplus v[3] \oplus v[2] \oplus v[1]$

$z_4 = y_7 \oplus y_6 \oplus y_5 \oplus v[4] \oplus v[3] \oplus v[2]$

$z_5 = y_8 \oplus y_7 \oplus y_6 \oplus W[2] \oplus W[1]$

$z_6 = y_8 \oplus y_7 \oplus y_5 \oplus W[3] \oplus W[2]$

$$z_7 = y_8 \oplus y_6 \oplus y_5 \oplus W[4] \oplus W[3]$$

$$z_8 = y_7 \oplus y_6 \oplus y_5 \oplus W[4] \oplus W[1]$$

The study expanded the preceding explanation using C++ code.

Table 5.2 is the original 8 x 8 first S-Box (1-Byte output S-Box) of Camellia, which is vulnerable to many attacks before applying the Khumbelo function. Appendix C, Figure C.1 is the C++ code of Table 5.2. Appendix C, Figure C.1 becomes 8 x 32 (4-Byte output S-Box) after the Khumbelo function is applied, as shown in C++ code in Appendix C, Figure C.2. The C++ code of the Khumbelo function is embedded in the new K_Camellia as shown in Figure 5.4.

Figure 5.5 depicts the original Feistel function of Camellia before the Khumbelo function is embedded. Figure 5.5 is changed to Figure 5.6 after the implementation of the Khumbelo function in the newly K_Camellia, which shows how $v[i]$ and $W[i]$ are returned from the Khumbelo function.

5.5.1 Mathematical Explanation of the Khumbelo Function

Assume an intruder knows the values of the $v[i]$ and $W[i]$. To reverse engineer the Khumbelo function input using $v[i]$ and $W[i]$, the intruder must first know the exact value of $(Output[0]\%32)$ and $(Output[1]\%32)$ as shown in Figure 5.4. In C++ programming, %32 is modulo 32. Also, $(Output[0])$ and $(Output[1])$ are 32-Bit variables from the newly generated 8 x 32 S-Box. An 8-Bit number is any number modulo 32. As a result, regardless of whether $G\%32 = D_{8-Bit}$ will always be 8-Bit because G is driven by %32. If an intruder knows the value of D, it does not follow that it is simple to solve and find the "exact" value of G using $G\%32 = D_{8-Bit}$.

For example, suppose the intruder knows the value of $D_{8-Bit} = 21 = 0x15_{hexadecimal}$. It is worth noting that the prefix $0x$ represents hexadecimal notation. To calculate an exact value of G that is 32-Bit using $G\%32 = 0x15_{hexadecimal}$, the value G yields a long list of non-constant numbers that

confuse the attackers, such as G being equal to one of the following numbers: $0x100336b5$, $0x1045b1f5$, $0x20024ff5$, $0x20130b95$, $0x30016755$, $40016ff5$, and others not mentioned since the roots of G yields long list to satisfy the condition of $G\%32 = 0x15_{hexadecimal}$.

The intruder will have to learn which root was used as an exact value in the list to reverse engineer the Khumbelo function input. As a result, the Khumbelo function is irreversible because it contains more %32 values. The preceding mathematical explanation explained the confusion and diffusion properties of the Khumbelo function, as well as the benefits of the properties.

Assume the intruder is aware of the values of the 8-Bit variables $v[i]$ and $W[i]$. Since the Khumbelo function is a hash function, it is also difficult to guess the values of $SubstitutionBox$. According to the explanation, it takes 32-Bit input and produces 8-Bit output. Refer to Figure 5.4 for more information. The hash function is difficult to crack in cryptography. Consult hash literature reviews.

The Khumbelo function employs an 8 x 32 S-Box resistant to the construction of many distinguishers, such as LAT, DDT, and DLCT. Since the Khumbelo function is a hash function, it is resistant to many known attacks [208].

5.6 Materials and Methods Used

The materials used in this study were C++ codes to attack, construct the distinguishers, measure the probability of guessing the key, and measure the encryption strength. The codes were applied on AES, DES, Camellia, and K_Camellia.

```
void KhumbeloFunction(Word *SubstitutionBox, Byte *v, Byte *W)
{
    Word Output[2];
    Byte FirstInput,SecondInput;
    Byte x[8]= {0};
    Byte k[8]= {0};
    Byte y[8]= {0};
    Byte t[8]= {0};
    for (int i=0; i<8; i++)
    FirstInput = x[i]^k[7-i];
    Output[0] = SubstitutionBox[FirstInput];
    for (int i=0; i<1; i++)
    {
        for( i=0; i<4; i++ )
        {
        v[(i<<2)+0] = (Byte)((Output[0]%32)>>24&0xff);
        v[(i<<2)+1] = (Byte)((Output[0]%32)>>16&0xff);
        v[(i<<2)+2] = (Byte)((Output[0]%32)>> 8&0xff);
        v[(i<<2)+3] = (Byte)((Output[0]%32)>> 0&0xff);
        }
    }
    SecondInput = t[0]^t[1]^t[2]^t[3];
    Output[1] = SubstitutionBox[SecondInput];
    for (int i=0; i<2; i++)
    {
        for( i=0; i<4; i++ )
        {
        W[(i<<2)+0] = (Byte)((Output[1]%32)>>24&0xff);
        W[(i<<2)+1] = (Byte)((Output[1]%32)>>16&0xff);
        W[(i<<2)+2] = (Byte)((Output[1]%32)>> 8&0xff);
        W[(i<<2)+3] = (Byte)((Output[1]%32)>> 0&0xff);
        }
    }
}
```

Figure 5.4: C++ Khumbelo Function

```
void FeistelFuction( const Byte *x, const Byte *k, Byte *y )
{
    Byte t[8];

    t[0] = SubstitutionBox1(x[0]^k[0]);
    t[1] = SubstitutionBox2(x[1]^k[1]);
    t[2] = SubstitutionBox3(x[2]^k[2]);
    t[3] = SubstitutionBox4(x[3]^k[3]);
    t[4] = SubstitutionBox2(x[4]^k[4]);
    t[5] = SubstitutionBox3(x[5]^k[5]);
    t[6] = SubstitutionBox4(x[6]^k[6]);
    t[7] = SubstitutionBox1(x[7]^k[7]);

    y[0] ^= t[0]^t[2]^t[3]^t[4]^t[5];
    y[1] ^= t[0]^t[1]^t[3]^t[5]^t[6];
    y[2] ^= t[0]^t[1]^t[2]^t[6]^t[7];
    y[3] ^= t[1]^t[2]^t[3]^t[4]^t[7];
    y[4] ^= t[1]^t[2]^t[3]^t[5]^t[6]^t[7];
    y[5] ^= t[0]^t[2]^t[3]^t[4]^t[6]^t[7];
    y[6] ^= t[0]^t[1]^t[3]^t[4]^t[5]^t[7];
    y[7] ^= t[0]^t[1]^t[2]^t[4]^t[5]^t[6];
}
```

Figure 5.5: C++ standard Fiestel Function before Khumbelo Fuction was Applied

```
void FeistelFuction( const Byte *x, const Byte *k, Byte *y )
{
    Byte W[8];
    Byte v[8];
    Byte t[8];

    KhumbeloFunction(SubstitutionBox, v, W);

    t[0] = v[0]^x[0]^k[0];
    t[1] = v[1]^x[1]^k[1];
    t[2] = v[2]^x[2]^k[2];
    t[3] = v[3]^x[3]^k[3];

    t[4] = W[0]^x[4]^k[4];
    t[5] = W[1]^x[5]^k[5];
    t[6] = W[2]^x[6]^k[6];
    t[7] = W[3]^x[7]^k[7];

    y[0] ^= t[0]^t[2]^t[3]^t[4]^t[5];
    y[1] ^= t[0]^t[1]^t[3]^t[5]^t[6];
    y[2] ^= t[0]^t[1]^t[2]^t[6]^t[7];
    y[3] ^= t[1]^t[2]^t[3]^t[4]^t[7];
    y[4] ^= t[1]^t[2]^t[3]^t[5]^t[6]^t[7];
    y[5] ^= t[0]^t[2]^t[3]^t[4]^t[6]^t[7];
    y[6] ^= t[0]^t[1]^t[3]^t[4]^t[5]^t[7];
    y[7] ^= t[0]^t[1]^t[2]^t[4]^t[5]^t[6];
}
```

Figure 5.6: C++ Modified Fiestel Function after Khumbelo Fuction was Applied

5.6.1 Why DES?

In cryptography, most attacks are tested on DES before being applied to complicated algorithms like Camellia. DES is no longer used for securing confidential information but for testing and verifying the new upcoming methods to be applied to complex algorithms like Camellia. DES is no longer secure since it can be easily broken by spooks and the Electronic Frontier Foundation (EFF) [209]. DES is also used to teach cryptography to new students in cryptology classes. Before analyzing all theories on Camellia, this study tested all theoretical attacks, codes, and methods on DES to ensure that the Khumbelo function worked as expected on Camellia. Hence, the DES is included in this study even though it is no longer used in the practical world. The focus was to compare Camellia (before applying the Khumbelo function) with the new K_Camellia (after using the Khumbelo function).

The cryptographic attacks rely on the construction of S-Box distinguishers. The differentiators are probability tables that aid attackers in locating secret keys. If S-Box outputs have a few bits, the distinguisher has a high chance of guessing the key. LAT, DDT, and DLCT are the most commonly used distinguishers. According to Camellia's literature review, all attacks are related to LAT, DDT, and DLCT construction. The materials applied in this study are C++ programs designed to validate the Camellia attack literature review theory. These C++ programs are as follows:

i. LAT's C++ code. Intruders use LAT to attack complex algorithms with LC-related variant attacks (or attacks derived from LC), such as zero-correlation linear distinguishers attacks [210].

ii. DDT's C++ source code. DDT attacks Camellia using DC-related variant attacks (or DC-derived attacks) like a boomerang and truncated differential attacks [187].

iii. DLCT's C++ code launches DL-related variant attacks (or attacks

derived from DL) against Camellia. Such as LC and DC attacks.

iv. The study used Avalanche Effect C++ code for calculating encryption strength. The Avalanche Effect investigates the significance of ciphertext change when one bit of plaintext or key is flipped, regardless of the algorithm components [211]. When the Avalanche Effect is considered, an algorithm may have significant encryption strength but still be vulnerable to attack if intruders identify weak components. Strict Avalanche Criterion (SAC) is also calculated using the Avalanche Effect [212]. SAC is passed if the encryption strength from the Avalanche Effect is between 45% and 55%; otherwise, the algorithm fails the SAC criterion [212].

Page 11 of [100] gave the theory construction of the first LAT of DES's first S-Box. The study analyzed the theory of constructing LAT using the experimental C++ code of LAT to compare outputs and verify the code. The S-Box used is 4 x 4, as shown in Table 5.3. Equation 5.4 is used to calculate LAT [213]. The study conducted the experimental C++ code of LAT. The results correspond to the LAT's theoretical results, which are given on page 11 of [100].

Table 5.3: Simplified DES's S-Box

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
SBox(X)	E	4	D	1	2	F	B	8	3	A	6	C	5	9	0	7

$$LAT(X_i, Y_i) = \bigoplus_{i=1}^{4}(\gamma X_i) \oplus \bigoplus_{i=1}^{4}(\aleph Y_i) \tag{5.4}$$

Where $\gamma \epsilon [1, 0]$ and $\aleph \epsilon [1, 0]$. The results $LAT(X_i, Y_i)$ can be tabulated as LAT.

Page 21 of [100] gave the theory construction of the first DDT of DES's first S-Box. The study analyzed the theory using the experimental C++ code of DDT to compare and verify the code. Equation 5.5 is used to calculate the DDT. The study conducted the DDT's experimental C++ code. The experimental results correspond to the theoretical results given DDT on page 21 of [100].

$$DDT(X_i, Y_i) = \sum_{i=1}^{4} (X_i \oplus Y_i)(SBox(X_i) \oplus SBox(Y_i)) \tag{5.5}$$

On page 157 of [211], the theoretical calculation of the Avalanche Effect of DES when one bit of plaintext is flipped was examined. The bit change difference from page 157 of [211] is 39. The researchers created a C++ program to calculate the Avalanche Effect of DES when one bit of plaintext is flipped in round 10. The study validated the C++ Avalanche Effect code against the theory presented on page 157 of [211]. The study conducted the experimental C++ code for the DES Avalanche Effect with a ciphertext bit change difference of 39.0000 when the plaintext was changed from 00000000 to 00000001. The study performed the C++ bit flipping from left to right, changing one bit in each event until 00000001 was reached. When the plaintext string was 00000001, the Avalanche Effect of DES was the same as the theoretical Avalanche Effect given on page 157 of [211]. The study conducted the experimental key Avalanche Effect for an encryption key. The results represent the key Avalanche Effect in percentage, which was 43.627030 percent. The result of 43.627030 percent was roughly equal to the theoretical development of 43.8721 percent given on page 97 of [215]. The Serpent's S-Boxes are DES S-Boxes [216]. The theory construction of the first DLCT of Serpent's first S-Box is given on page 9 of [216]. This study analyzed the experimental C++ code of DLCT to compare and verify the code. The study conducted the experimental C++ code of DLCT. The results corresponded to the theoretical DLCT given on page 9 of [216]. The S-Box used was 4 x 4 size, as

shown in Table 5.4. The DL attack is a mathematical approach for cracking ciphers that use S-Boxes to construct DLCT and guess the secret keys.

Table 5.4: First S-Box of Serpent defined as $SBox_0(X)$

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SBox_0(X)$	3	8	F	1	A	6	5	B	E	D	4	2	7	0	9	C
$InvSBox_0(X)$	D	3	B	0	A	6	5	C	1	4	4	7	F	9	8	2

An intruder selects input (plaintext) pairs of an S-Box (ϱ_1, ϱ_2) and analyzes the output (ciphertext) pairs $(\S_1, \S_2)$ to construct DLCT using Equation 5.6. There is Δ, which is calculated as $\varrho_1 \oplus \varrho_2$ and $\varkappa$, which is calculated as $\S_1 \oplus \S_2$ from Equation 5.6. The dot operator is used for multiplication, which means that bits are multiplied rather than entire bytes.

$$DLCT(\Delta, \varkappa) = \sum_{SBox_i(x)\epsilon[1,0]} (-1)^{\varkappa.(SBox_i(x) \oplus SBox_i(x \oplus \Delta))} \tag{5.6}$$

After verifying all the developed C++ code for this study using DES for various attacks, the study deployed all of the above codes on Camellia, knowing that the materials (C++ codes) are tested and working correctly on DES.

5.6.2 Why AES?

AES was also used to compare Camellia and K_Camellia. Given that Camellia is a block cipher with the same level of security as AES and is used in OpenSSL and TLS, it was reasonable to assume that the vulnerability could have widespread consequences. The research described and compared AES susceptibility. Instead of using DES S-Box alone given in Table 5.4, the study used AES S-Box given in Appendix C, Figure C.3 to calculate LAT, DDT, and DLCT of AES. Additionally, the study calculated the AES Avalanche Effect. For the plaintext Avalanche Effect, AES yielded the Avalanche Effect of 50.5371%. AES yielded the ciphertext Avalanche Effect of 49.5788%.

5.7 Related Work of Cryptographic Attacks on Camellia

. According to the findings, intruders attack the Camellia algorithm with various attacks such as the LC, DC, boomerang, truncated differentials, Meet-in-the-Middle, impossible differential, and zero-correlation linear attacks using different distinguishers.

According to Wagner [200], the boomerang attack was based on a DC attack. Biryukov [217] confirmed Wagner [200]'s statement and added that the boomerang attack could combine the truncated differentials and Meet-in-the-Middle approach [217]. Dunkelman, Keller, Ronen, and Shamir [218] also confirmed that the boomerang attacks were derivatives of differential attacks that allowed the incorporation of two irrelevant differential properties, q, and p.

According to the above statements, this study also looked into Camellia's DC, truncated differentials, and Meet-in-the-Middle attacks, as these were the foundations of the boomerang attack. Even if intruders attacked Camellia, the study by Burak and BŁaszyŃski [219] discovered that users of IoT devices still implemented Camellia on the OpenMP Application Program Interface (API).

According to Yap, Khoo, and A. Poschmann [197], the Camellia algorithm was attacked using the boomerang. Yap, Khoo, and A. Poschmann [197] converted the traditional Feistel structure into a two-cell GF-NLFSR network while preserving all other building blocks, such as S-Boxes and linear distribution mapping. Compared to the Camellia algorithm, the p-Camellia was more resistant to boomerang attacks [197].

According to Aoki *et al.* [198], the best boomerang option for Camellia was to reduce Camellia to 8-round, which was bounded at 2^{-66} and acquired by $p\delta = 2^{-12}$ of 3-round and $p\delta = 2^{-54}$ of 5-round. Aoki *et al.* [198] attacked Camellia and indicated that the 3-round and 5-round attacks were much

faster than the full-round Camellia algorithm.

Shirai [220] used the boomerang attack to attack Camellia up to 9 rounds on the reduced-version of Camellia. By analyzing and comparing the new Meet-in-the-Middle attack results, Lu, *et al.* [199] confirmed Shirai [220]'s work that intruders attacked Camellia. Matsui and Nakajima [221] recommended that Camellia be used for Japanese e-Government platforms without consideration of attacks.

According to Biryukov and Nikoli [222], the best procedure for discovering the secret key using the boomerang attack was when the highest characteristic was on the third round and the lowest on the fourth round, to give a total of seven rounds of attack. Biryukov and Nikoli [222] discovered that the boomerang attack's probability of all active S-Boxes was more significant than 2^{-128}. Biryukov and Nikoli [222] attacked the reduced version of Camellia up to 7 rounds because the probability was more significant than 2^{-128}, which was easy to attack. Biryukov and Nikoli [222] attacked Camellia for more than seven rounds because the probability was less than 2^{-128}.

Lee *et al.* [223] presented a truncated differential cryptanalysis attack of altered Camellia that was reduced to 7 and 8 rounds. Lee *et al.* [223] discovered the 8-Bit key on 7-round and the 16-Bit key on 8-round, with 3*281 and 3*282 plaintext, respectively. The other component of the boomerang attack is the truncated differential cryptanalysis attack [217], [218]. Even if intruders attacked Camellia, Moriai and Kato [224] advocated for Camellia's use in Cryptographic Message Syntax (CMS) without consideration of attacks.

Bai and Li [225] successfully attacked the 11 rounds of Camellia-128, 11 rounds of Camellia-192, 12 rounds of Camellia-192, and 14 rounds of Camellia-256 using impossible differential cryptanalysis attacks and time complexity of $2^{123.6}$, $2^{121.7}$, $2^{171.4}$ and $2^{238.2}$, respectively.

Wu, Zhang, and Feng [226] discovered several nontransmissible 8-round Camellia attacks using impossible differentials. Wu, Zhang, and Feng [226] indicated that the intruders were aware of the maximum number of 7-round

attacks, which was feasible using impossible differentials. According to Wu, Zhang, and Feng [226], the impossible differentials attack attacked the 12 rounds of Camellia with the chosen plaintext of 2^{120} and 2^{181}. Even if intruders attacked Camellia, Poetro [227] recommended that Camellia be used in cryptographic images (CIs).

Wu, Zhang, and Feng [228] used the relationship between subkeys and the number of Camellia rounds, combined with several approaches in the secret key retrieval technique, to improve the impossible differential attack up to 12 rounds of Camellia-128 and 16 rounds of Camellia-256. Wu, Zhang, and Feng [228] successfully attacked 12-round and 16-round plaintexts of 2^{65} and 2^{89}, respectively. The illustrated results were more successful than any previously publicized Camellia attack results by Wu, Zhang, and Feng [226].

Lu *et al.* [199] described the infrequent 5-round and 6-round effects of Camellia and eventually used the infrequent to attack 10, 11, and 10 rounds of Camellia using Meet-in-the-Middle attacks to discover the 128-Bit key, 192-Bit key, and 256-Bit key, respectively. Even if intruders attacked Camellia, Aoki *et al.* [198] indicated that Camellia was used in various platforms such as software and hardware systems without considering attacks.

Mala, Dakhilalian, and Shakiba [229] introduced a hash table extension technique. Mala, Dakhilalian, and Shakiba [229] used the hash table extension technique to attack the 16-round Camellia-256 hash table. Mala, Dakhilalian, and Shakiba [229] continued to use the impossible differential attack; the attack's scope of the target subkey expanse was enormous, the initial phases were conducted slowly, and the hash table extension technique was very successful. According to Mala, Dakhilalian, and Shakiba [229], the impossible differential cryptanalysis on Camellia-256 requires $2^{124.1}$ known plaintexts with time complexity of $2^{249.3}$.

Liu *et al.* [230] exploited some fascinating key-dependent layer effects. Liu *et al.* [230] improved previously published results on impossible differential cryptanalysis attacks on reduced-version Camellia and demonstrated

new attack scrutiny. Liu *et al.* [230] introduced some new seven rounds for weak keys on Camellia using impossible differential attack. According to Liu *et al.*, [230], the weak keys that work on the impossible differential attack took up to 75% of the keyspace. As a result, Liu *et al.* [230] decided to eliminate the weak-key speculation and used the multiplier approach to reduce attacks on the reduced version of Camellia using the original keys. Second, Liu *et al.* [230] constructed a slew of differentials containing at least one of the eight rounds of Camellia using the impossible differential. Following that new result, Liu *et al.* [230] demonstrated that the key-dependent transformations inserted in Camellia could not effectively resist impossible differential cryptanalysis.

Liu *et al.* [230] presented a new attack method to scale impossible differential attacks on the reduced version of Camellia based on the set of differentials. Despite being attacked by intruders, the Camellia algorithm was still used in IoT protocols [215].

Liu *et al.* [195] used the zero-correlation linear cryptanalysis attack and investigated Camellia's security. Following the discovery of specific weak keys, Liu *et al.* [195] proposed some unique properties of the Camellia's $FL/(FL)^{-1}$ functions. Accordingly, Liu *et al.* [195] built the first known eight rounds of zero-correlation linear distinguisher of Camellia with $FL/(FL)^{-1}$ layers for the described weak keys.

The distinguisher covered the same number of rounds as the best-known zero-correlation as the linear distinguisher for Camellia without $FL/(FL)^{-1}$ layers. Liu *et al.* [195] indicated that $FL/(FL)^{-1}$ layers could not prevent zero-correlation linear cryptanalysis attack virtually for specific weak keys. Then, Liu *et al.* [195] used that new distinguisher to demonstrate key retrieval attacks on the 13 and 14 of Camellia-192, respectively. Even if the intruders attacked the Camellia, Čiča [231] indicated that consumers still use Camellia in modern communication networks.

The research discovered that most cryptographic attacks rely on distin-

guishers like LAT, DDT, and DLCT. Although the Camellia had been attacked, the study focused on preventing these attacks by employing a novel mathematical function known as the Khumbelo function before the entire IoT platforms, devices, users, and confidential information on IoT are compromised.

The Khumbelo function distracted the distinguishers' construction to prevent these attacks. The Khumbelo function reduced the probability of LAT, DDT, and DLCT attacks. If the S-Box has N-Bit input and M-Bit output, that S-Box is defined as N x M S-Box or (M-Bit output S-Box if only output bits/byte are considered). The Khumbelo function also converted a 1-Byte output S-Box (8 x 8) of an original Camellia to a 4-Byte output S-Box (8 x 32). The 8 x 8 S-Box constructs distinguishers of 2^8 x 2^8 = 256 x 256 matrices. The 256 x 256 matrices have a total of 65536 elements. Any ordinary computer can compute the 256 x 256 matrices. The 8 x 32 S-Box constructs distinguishers of 2^8 x 2^{32} = 256 x 4294967296 matrices with an expected element count of 1099511627776. Due to computational space, an ordinary computer could not compute 256 x 4294967296 matrices. The study used the Khumbelo function in Camellia to distract the construction of distinguishers (attack engines) from preventing the spread of attacks.

The Khumbelo function generated a 4-Byte output S-Box instead of an original 1-Byte output S-Box in Camellia. Additionally, the Khumbelo function is composed of many modulo operators. The new 4-Byte output S-Box and modulo operators confuse and block intruders to construct distinguishers.

5.8 Methods of Applying the Khumbelo Function on Camellia

The entire research methodology was summarized based on the novel approach of using the Khumbelo function to prevent multiple attacks. Refer to Figure 5.7.

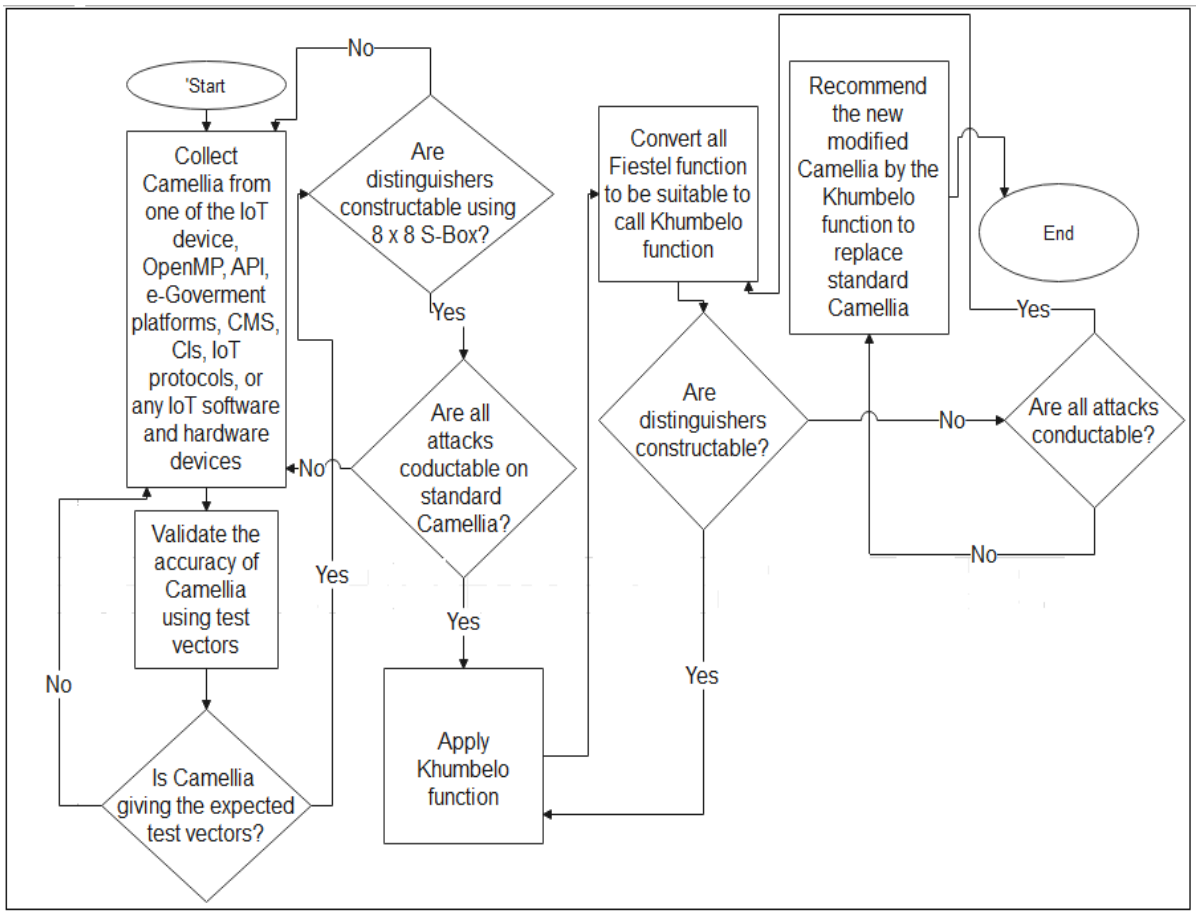


Figure 5.7: Research Methodology based on the Khumbelo

The methodology used the Khumbelo function to prevent the set number of rounds against the attacks before the Camellia was attacked. The following was the research methodology:

i. Select the Camellia algorithm from a list of IoT objects such as OpenMP, API, e-Government platforms, CMS, CIs, IoT protocols, or IoT software and hardware devices.

ii. Verify the accuracy of the Camellia using Camellia developers' test vectors.

iii. If Camellia does not produce Camellia developers' test vectors, then the algorithm collected from the IoT device is not Camellia. Repeat step (i) with a different device. Assume that the OpenMP test vectors do not correspond to the Camellia developers' test vectors and that OpenMP failed to produce the expected test vectors.

iv. Check to see if distinguishers like LAT, DDT, and DLCT can be built with the original 8 x 8 S-Box (1-Byte output S-Box). If so, proceed to step (v). Otherwise, repeat steps (i) - (iv) again.

v. On a Camellia, see if the boomerang, DC, trucked differential, Meet-in-the-Middle, zero-correlation linear distinguisher, and LC attacks are operable. If yes, proceed to step (vi). Otherwise, repeat steps (i) - (iv) again.

vi. Make use of the Khumbelo function. Use the newly generated 8 x 32 S-Box (4-Byte output S-Box) as shown in Figure 5.4. Refer to Appendix C, Figure C.2.

vii. Replace the old Feistel function with the 8 x 32 S-Box generated using the Khumbelo function. That is, replace Figure 5.5 with Figure 5.6.

viii. Check to see if distinguishers like LAT, DDT, and DLCT can be built with the newly generated 8 x 32 S-Box generated using the Khumbelo function. Refer to Appendix C, Figure C.2. If so, repeat steps (vi) - (viii) . Otherwise, proceed to step (ix).

ix. Check whether the boomerang, DC, trucked differential, Meet-in-the-Middle, zero-correlation linear distinguisher, and LC attacks can be carried out on a K_Camellia, as stated in a literature review. If so, repeat steps (vi) through (ix). Otherwise, proceed to step (x).

x. Accept a new K_Camellia protected by the Khumbelo function as the recommended cipher against boomerang, DC, trucked differential, Meet-in-the-Middle, zero-correlation linear distinguisher, DC, and LC attacks when compared to the Camellia cipher.

5.9 Theoretical Analysis and Discussion

This study's theoretical analysis was based on C++ program experiments and mathematical analysis to validate the theoretical results of attacks presented in literature reviews. Before applying K_Camellia, the theoretical analysis was performed on DES, AES, and Camellia. The reason was that there was a lot of literature review on DES, AES, and Camellia before applying the code to K_Camellia. There will be no literature review of K_Camellia because it is a new algorithm developed in this study. To validate theoretical analysis using codes, S-Boxes of DES were used.

It should be noted that DES was used in cryptology for learning and testing purposes, according to [209]. Before attempting to attack any complex algorithm such as Camellia, cryptologists first test the attack codes on DES [209]. The study compared and verified the theoretical results using code by analyzing the theory construction of the first LAT of DES's first S-Box, on page 11 of [100]. The results were consistent with the theoretical results presented by LAT on page 11 of [100]. In the study, the LC attack

on DES was carried out using LAT. The findings confirmed that DES was susceptible to LC attack.

The C++ experimental results of LAT construction were shown in Figure 5.8 to validate theoretical results given on page 11 of [232]. During the LC attack on DES, the theoretical LAT on [100] was used to calculate the probability of guessing the key, the number of rounds to attack, and the complexity. By using C++ experimental LAT in Figure 5.8, this study confirmed the theory presented in [232].

Figure 5.9 depicted the C++ experimental results of DDT construction to validate the theoretical results presented on page 4 of [233]. During the DC attack on DES, the theoretical DDT on [233] was used to calculate the probability of guessing the key, the number of rounds to attack, and the complexity. This study validated the theory presented on [233] by employing C++ experimental LAT in Figure 5.9.

The C++ experimental results of DLCT construction were shown in Figure 5.10 to validate the theoretical results given on page 9 of [216]. During the DL attack on DES, the theoretical DLCT on [216] was used to calculate the probability of guessing the key, the number of rounds to attack, and the complexity. Using C++ experimental LAT in Figure 5.10, this study confirmed the theory given on [216].

This research only provided the experimental LAT, DDT, and DLCT of DES. Other algorithms' experimental LAT, DDT, and DLCT in C++ required more than four pages to display. As a result, the distinguishers were not displayed in this study. For example, the C++ experimental LAT, DDT, and DLCT of an 8 x 8 S-Box of Camellia was a 2^8 x 2^8 matrix, which was 256 x 256 matrix that required five pages to display each. Imagining the display of the C++ experimental LAT, DDT, and DLCT an 8 x 32 S-Box of K_Camellia, which would be the matrix of 2^8 x 2^{32}, which was theoretically calculated as 256 x 4294967296 matrix each, more 500000 pages would be required to display these matrices. As a result, the distinguishers of

other algorithms were not included in this study, but the study did conduct and analyze the C++ experimental results of the omitted distinguishers. Section 5.10 contains more information on the theoretical and experimental attack. The same method was used to analyze all algorithms. Refer to Table 5.5 up to Table 5.10.

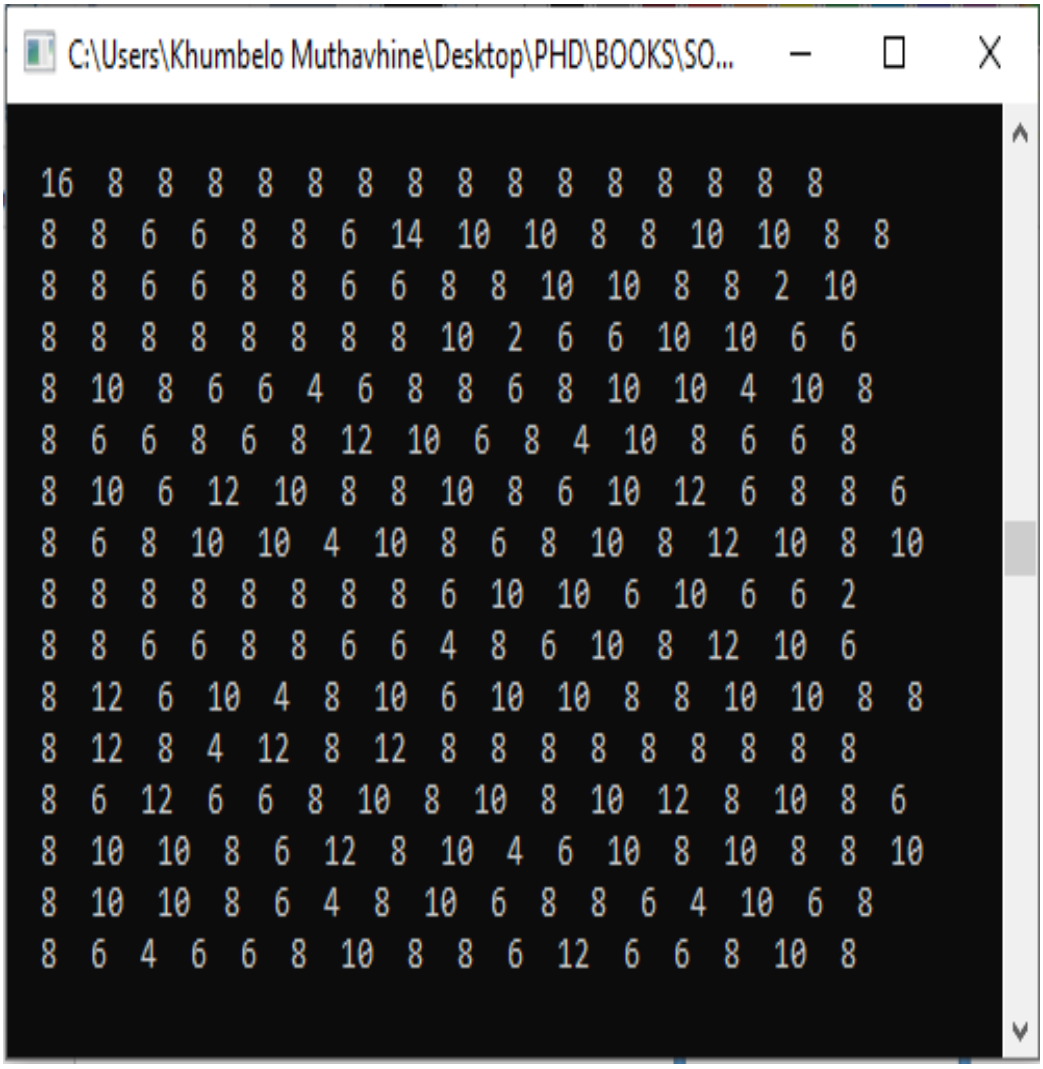


Figure 5.8: C++ Results of LAT

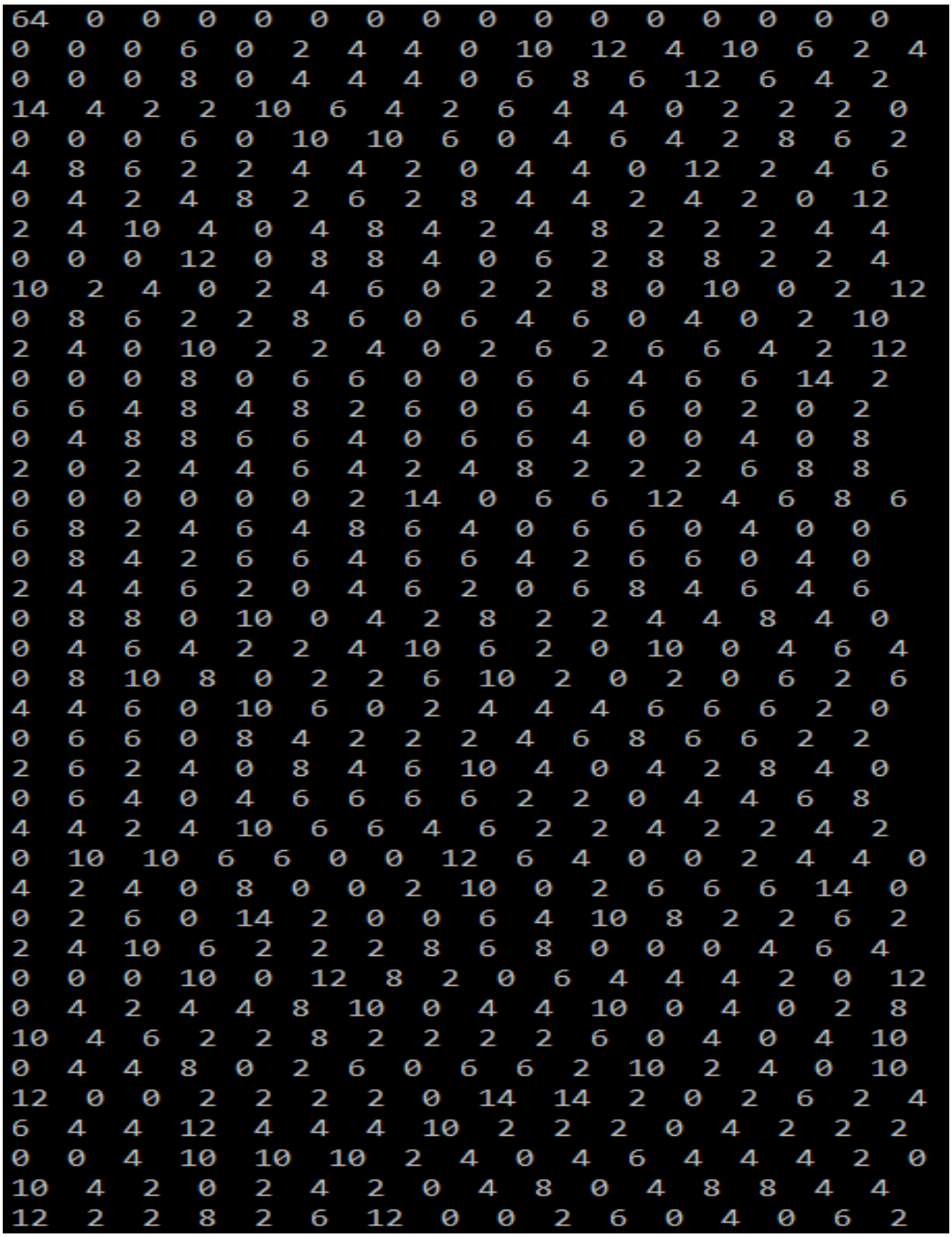


Figure 5.9: C++ Results of DDT

Figure 5.10: C++ Results of DLCT

The study examined the memory required to install all algorithms using C++ code. For example, to obtain the memory required to install the DES algorithm, the DES file was accessed via C++, and the code calculated the file size. Refer to Figure 5.11 for more information. After calling the DES.CPP file, the size of the DES. CPP file was calculated to be 15019 Bytes. On page 79 of [214], 15019 Bytes confirmed theoretical results of 15 Megabytes of DES. The same method was used to analyze all algorithms. Refer to Table 5.20 and Figure 5.19 for more information.

The theoretical results of the Avalanche Effect of AES on [215], indicating that AES passed SAC, were analyzed using C++ coding in the study. Figure 5.12 depicted the C++ experimental results of AES's plaintext Avalanche Effect. Other algorithms' Avalance Effects were also tested using the code. Only the plaintext of the Avalanche Effect of AES was displayed in this study to demonstrate an executable file and how the Avalanche Effect was calculated and confirmed. The same method was used to analyze all algorithms.

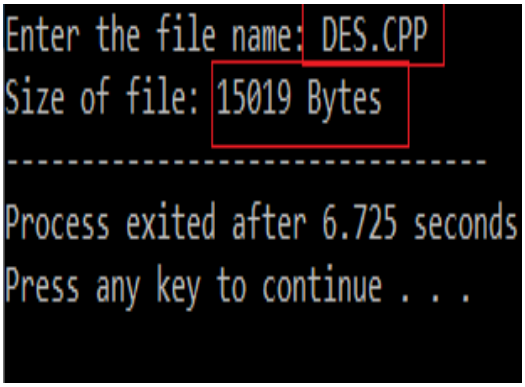


Figure 5.11: Memory Needed for DES in C++

For more information, refer to Table 5.12 and 5.11.

Furthermore, the zero-correlation linear distinguisher attack is a variant of the LC attack, as cited in [210], [187]. When the plaintext/ciphertext pair N was chosen, Equation 5.7 predicted that $p = |\frac{N}{16}| > 0$ from LAT. In round 8, the study guessed the position of key bits by using a 2^{34} chosen-plaintext/ciphertext pair. Because zero-correlation linear distinguisher attacks are a variant of the LC attack, the results confirmed that DES was vulnerable to them [210], [187]. If the LAT contained a -6 occurrence, $p > |\frac{-6}{16}|$ was used.

$$\begin{aligned} &\Delta \subset \oplus \Delta \varrho \oplus \ SBox(K, \varrho) = \Delta K \\ &where \ \subset = Ciphertext, \ \varrho = Plaintext \ and \ K = Key \end{aligned} \tag{5.7}$$

Assume that the probability is $p = |\frac{1}{2}| = 0$, Equation 5.8 holds, but obtaining key bits is difficult because S-Box is not included in the equation, and the LAT probability of guessing the key is low, less than $\frac{1}{2}$. Then Equation 5.8 is insignificant.

$$\begin{aligned} &\Delta \subset \oplus \Delta \varrho = \Delta K \\ &where \ \subset = Ciphertext, \ \varrho = Plaintext \ and \ K = Key \end{aligned} \tag{5.8}$$

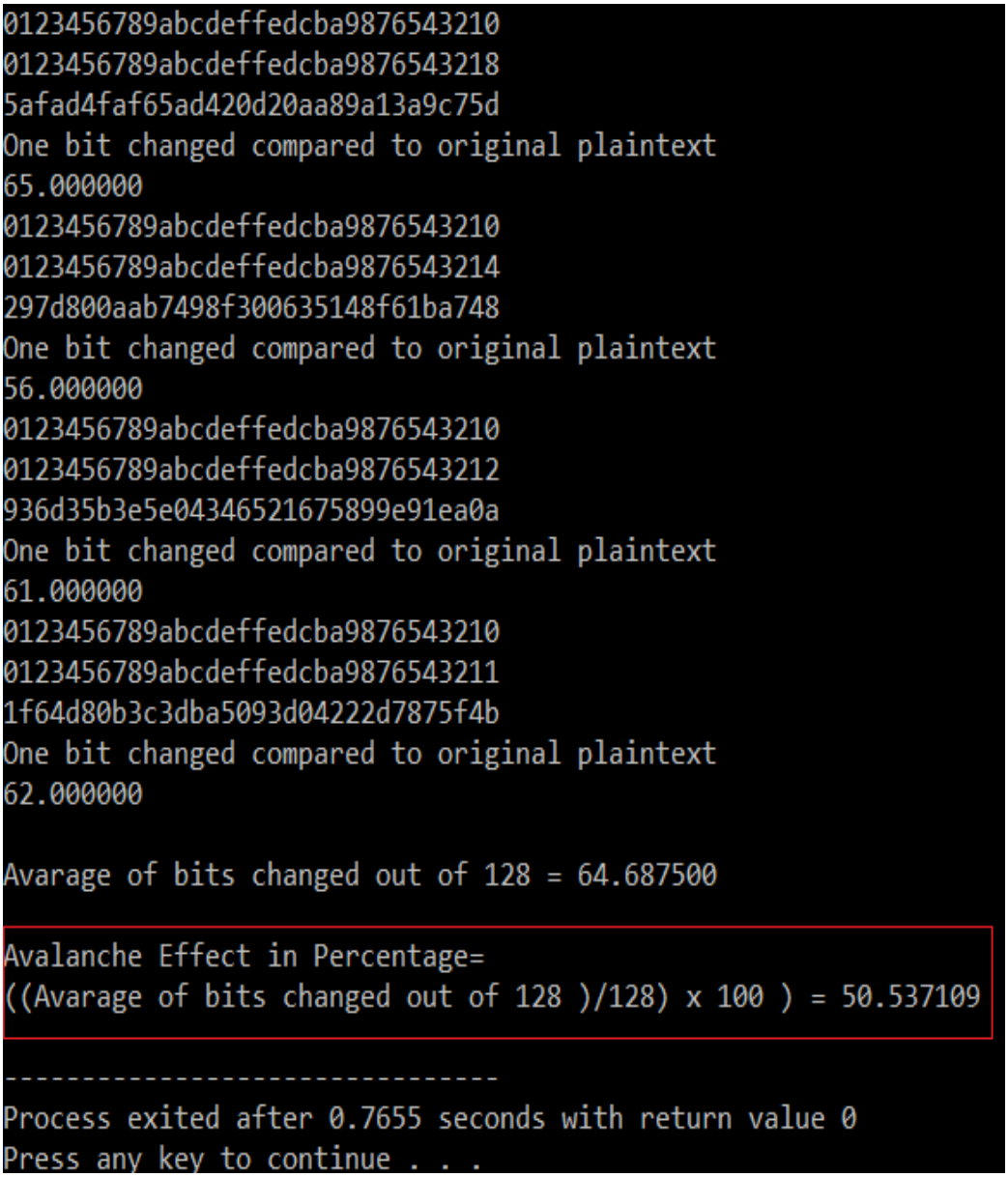


Figure 5.12: AES's Plaintext Avalanche Effect in C++

LAT has been used in LC-related variant attacks (attacks derived from LC), such as zero-correlation linear distinguishers attacks, according to [210]. The study used LC and zero-correlation linear distinguishers attacks on Camellia, which were influenced by Bogdanov and Vejre's theoretical analysis results [210]. The experimental results corroborated the theoretical findings in [210]. Camellia was found to be vulnerable to LC attacks and a variant of LC attacks (zero-correlation linear distinguisher attack).

The study compared and verified the experimental C++ code of DDT with the theory construction of the first DDT of DES's first S-Box on page 21 of [100]. The DDT's experimental C++ code was used in the study. The results were consistent with the theoretical results presented by DDT on page 21 of [100]. In the study, the DC attack on DES was carried out using DDT and Equation 5.2. The experimental results matched the theoretical findings on page 21 of [100]. The boomerang, truncated differentials, Meet-in-the-Middle, and impossible differential attacks are variants (derivations) of the DC attack, according to [100]. The findings confirmed that DES was vulnerable to DC attacks and variants (boomerang, truncated differentials, Meet-in-the-Middle, and impossible differential attacks).

When $K \oplus \varrho_i$ and $K \oplus \varrho_j$ are 4 and 8, respectively, and ϱ_i and ϱ_j are 2 and 4, respectively, the probability of guessing the correct key using DDT is 2. That is 8, 2, or 4 digits. The study tested 2 and 4 to find the valid key and discovered that the correct key is 4.

According to [187], the boomerang and truncated differential attacks are variants of DC attacks. Camellia was attacked with the boomerang and truncated differential attacks in this study. The study examined the theoretical calculation of the Avalanche Effect of DES when one bit of plaintext was flipped in round 10 on page 157 of [211]. The bit difference between page 157 of [211] is 39. When one bit of plaintext was flipped in round 10, the researchers wrote a C++ program to calculate the Avalanche Effect of DES. The reason was to compare the C++ Avalanche Effect code to the theory on

page 157 of [211].

After changing the plaintext from 00000000 to 00000001, the study ran the experimental C++ code for the DES Avalanche Effect with a ciphertext bit change difference of 39.0000. The experiment flipped bits from left to right, changing one bit in each event until 00000001. The Avalanche Effect of DES was the same as the theoretical Avalanche Effect given on page 157 of [211] when the plaintext string was 00000001.

According to [216], the Serpent's S-Boxes are DES S-Boxes. This study analyzed the theory construction of the first DLCT of Serpent's first S-Box, which was given on page 9 of [216], and tested the experimental C++ code of DLCT to compare and verify the code. The DLCT experimental C++ code was used in the study. The experimental results were consistent with the theoretical results presented by DLCT on page 9 of [216].

The research was based on experimental procedures using C++ codes for distinguisher constructibility. LAT, DDT, and DLCT were the distinguishers. The study discovered that these distinguishers were the most valuable and necessary tools for administering the majority of cryptographic attacks. Many cryptographic attacks would only be possible to carry out with these distinguishers. For example:

i. The DC attack used DDT to succeed [180], [196].

ii. The LC attack relied on LAT to succeed [100].

iii. DDT was used to help the boomerang attack succeed. Refer to page 4 of [185].

iv. Truncated differential thrived on DDT. Refer to page vii of [186].

v. A zero-correlation linear attack was a variation (derivation of) the LC attack [187]. Also, refer to page 119 of [189].

vi. DDT was used to succeed in a Meet-in-the-Middle attack. Refer to page 136 of [188].

vii. The DL attack used DLCT to succeed. Refer to page 9 of [216].

These distinguishers provide intruders with a probability table for guessing any algorithm's secret keys. The study discovered that because all attacks were based on distinguishers, therefore no construction of distinguishers, no attacks [199], [203]. As a result, this study aimed to use the Khumbelo function to prevent cryptographic attacks on Camellia.

Imagining the display of the C++ experimental LAT, DDT, or DLCT an 8 x 32 S-Box of K_Camellia, which would be the matrix of 2^8 x 2^{32}, which was theoretically calculated as 256 x 4294967296 matrix each. No ordinary computer could compute and execute the 256 x 4294967296 matrix using C++. The program crashed before execution. Hence the study found that executing the 256 x 4294967296 matrix was impractical. Theoretically, if there are no LAT, DDT, or DLCT, then there are no LC-related, DC-related, or LC-related attacks. Therefore, the Khumbelo function protected K_Camellia against cryptographic attacks.

The Khumbelo function hampered the development of these distinguishers. The Khumbelo function is generated using a 4-Byte output S-Box instead of an original 1-Byte output S-Box in Camellia. Additionally, the Khumbelo function is composed of many modulo operators. The new 4-Byte output S-Box and modulo operators confuse and block intruders to construct distinguishers.

5.10 Results of Cryptographic Attacks on Camellia

The Khumbelo function distracted the distinguishers' construction to prevent these attacks. The Khumbelo function decreased the probability of LAT, DDT, and DLCT attacks. If an S-Box has N-Bit input and M-Bit output, it is defined as N x M S-Box or (M-Bit output S-Box if only output bits/byte are considered). The Khumbelo function also converted an original Camel-

lia's 1-byte S-Box (8 x 8) to a 4-byte S-Box (8 x 32) output. The 8 x 8 S-Box constructs distinguishers for matrices of 2^8 x 2^8 = 256 x 256. There are 65536 elements in the 256 x 256 matrices. Any ordinary computer can compute the 256 x 256 matrices. The 8 x 32 S-Box constructs distinguishers of matrices of 2^8 x 2^{32} = 256 x 4294967296 with an expected element count of 1099511627776. An ordinary computer could not compute 256 x 4294967296 matrices due to computational space constraints. The Khumbelo function in Camellia was used in the study to distract the construction of distinguishers (attack engines) from preventing the spread of attacks. The results of the construction feasibility of LAT, DDT, and DLCT on different sizes of S-Boxes were shown in Table 5.5 5.6 and 5.7, respectively. The results showed that it was impossible to construct distinguishers after applying the Khumbelo function due to the memory required to construct a matrix of 256 x 4294967296 matrices with a total of 1099511627776 entities. Refer to Table 5.5, 5.6 and 5.7 for more information.

The code for generating distinguishers for a new 4-Byte output S-Box failed before construction due to the amount of memory required by a computer to run, display, and execute a 256 4294 967 296 matrix. A maximum of 2^{64} memory allocation was required when building any distinguishers on a laptop. Calculating 2^{32} x 256 necessitates more memory allocation than 2^{64}, which is impractical. Due to memory constraints in a computer, the research confirmed that it was impractical to create a table or any distinguishers of 2564 294 967 296. Storage limits on Macintosh (Apple) and Microsoft (HP) computers were 2^{64}, resulting in a complicated construction of distinguishers for the various attacks. Calculating the probabilities of guessing a secret key using a 32-Bit output S-Box was impractical. Tables and graphs were used in this study to analyze and discuss the findings. Table 5.8 and Figure 5.13 showed the LAT results of DES, AES, Camellia, and K_Camellia.

DES, AES, and Camellia had probabilities of 37.5 percent, 54.6875 percent, and 54.6875 percent, respectively. Any attack that relied on LAT's

probabilities could quickly attack DES, AES, and Camellia. The study confirmed the theoretical results found in [100], [101], and [104] because the experimental results were the same. For more information, refer to Table 5.8 and Figure 5.13. Furthermore, Table 5.8 and Figure 5.13 demonstrated that constructing LAT of K_Camellia using 8 x 32 S-Box newly generated using the Khumbelo function was impossible.

Table 5.9 and Figure 5.14 show the DDT results of DES, AES, Camellia, and K_Camellia. Figure 5.14 showed 0% DDT results because no DDT was constructed due to computer memory constraints.

The probability of guessing the DES, AES, and Camellia keys were 6.25 percent, 1.5625, and 1.5625 percent, respectively. These findings suggest that any DDT-based attack could use the probabilities discovered in DDT to attack DES, AES, and Camellia. The study confirmed the theoretical results found in [100], [234], and [104] because the experimental results were the same. For more information, refer to Table 5.9 and Figure 5.14. Furthermore, Table 5.9 and Figure 5.14 demonstrated that it was impossible to construct DDT of K_Camellia using 8 x 32 S-Box newly generated using the Khumbelo function.

The DLCT results of DES, AES, Camellia, and K_Camellia were shown in Table 5.10 and Figure 5.15. Table 5.10 and Figure 5.15 show the DLCT results of DES, AES, Camellia, and K_Camellia. The probabilities for DES, AES, and Camellia were 50%, 50%, and 50%, respectively. These findings suggest that any DLCT-based attack could use the probabilities discovered in DLCT to attack DES, AES, and Camellia. The study confirmed the theoretical results found in [100], [216], and [104] because the experimental results were the same. For more information, refer to Table 5.10 and Figure 5.15. In addition, Table 5.10 and Figure 5.15 demonstrated that the DLCT of K_Camellia could not be constructed using an 8 x 32 S-Box generated using the Khumbelo function. Figure 5.15 showed 0% DLCT results because no DLCT was constructed due to computer memory constraints.

The study discovered that because all attacks were based on distinguishers, there would be no attacks [199], [203]. As a result, the goal of this study was to use the Khumbelo function to prevent several Camellia attacks. The development of these distinguishers was blocked by the Khumbelo function. On a standard computer, it would be impossible to build all 8 x 32 S-Box distinguishers. In Camellia, the Khumbelo function was generated using a 4-Byte output S-Box rather than the original 1-Byte output S-Box. Furthermore, the Khumbelo function contained a large number of modulo operators. Intruders were confused and prevented from constructing distinguishers by the new 4-Byte output S-Box and modulo operators.

As shown in Table 5.11 and Figure 5.16, the key Avalanche Effect of DES was 43.6270%, which was outside the required SAC criterion range (between 45% and 55%). As a result, DES failed to pass the critical Avalanche Effect. The AES, Camellia, and K_Camellia met the SAC criterion because their key Avalance Effects were 50.6184, 49.5788, and 50.5666 percent, respectively. Table 5.11 and Figure 5.16 show that the AES, Camellia, and K_Camellia met the criterion because their SACs were between 45% and 55%. The study confirmed the theory results in [215], which stated that DES failed the SAC criterion by 43.8720%, close to the experimental results of 43.6270%. The study also confirmed the theoretical findings in [215], which stated that AES and Camellia met the SAC criterion by 49.0661% and 49.6093%, respectively. Camellia met the SAC criterion by 49.0661% and 49.6093%, respectively, according to the theoretical and experimental AES results. K_Camellia is a novel algorithm developed for this study. As a result, theoretical results are still unavailable. Refer to Table 5.11 and Figure 5.16.

As shown in Table 5.12 and Figure 5.17, the plaintext Avalanche Effect of DES was 62.1337%, which was outside the required SAC criterion range (between 45% and 55%). As a result, DES failed the plaintext Avalanche Effect. The AES, Camellia, and K_Camellia met the SAC criterion because their plaintext Avalance Effects were 50.5371, 50.2502, and 49.3774 percent,

respectively. Table 5.12 and Figure 5.17 show that the AES, Camellia, and K_Camellia met the criterion because SACs ranged between 45% and 55%. The study confirmed the theory results in [215], which stated that DES failed the SAC criterion by 62.8662%, close to the experimental results of 62.1337%. The study also validated the theoretical results in [215], which stated that AES and Camellia met the SAC criterion by 49.7924% and 49.4689%, respectively. According to the theoretical SAC criterion results, Camellia met the SAC criterion by 50.5371% and 50.2502%. K_Camellia is a novel algorithm developed for this study. As a result, theoretical results still need to be made available. For more information, refer to Table 5.12 and Figure 5.17.

The speed of DES, AES, Camellia, and K_Camellia using C++ codes in an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz 3.40 GHz of HP during encryption of a 256 Megabyte (MB) file was measured in the study. According to the results of the experiments, the speeds of DES, AES, Camellia, and K_Camellia were 41.08 MB/second, 55.44 MB/second, 55.43 MB/second, and 52.54 MB/second, respectively. As a result, DES was the slowest algorithm, with a speed of 41.08 MB/second, followed by K_Camellia, which had a speed of 52.54 MB/second. AES and Camellia had nearly identical speeds of 55.44 MB/second and 55.43 MB/second, respectively. For more information, refer to Table 5.13 and Figure 5.18.

The theoretical results showed that the speeds of DES, AES, and Camellia were 21.34 MB/second, 48.23 MB/second, and 77.34 MB/second, respectively. As a result, DES was the slowest algorithm, with a speed of 21.34 MB/second, followed by AES, which had a speed of 48.23 MB/second. Camellia, with a speed of 77,34 MB/second, was the fastest algorithm compared to DES and AES. K_Camellia is a novel algorithm developed for this study. As a result, theoretical results still need to be made available. All theoretical results were discovered using [235] and [198]. For more information, refer to Table 5.13 and Figure 5.18. Because this study used an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz instead of the machines used in [235]

and [198], the theoretical results were unrelated to the experimental results. Refer to [235] and [198] for more information on machines used in theoretical results.

The experimental and theoretical results of the number of rounds attacked, where the key was discovered on each algorithm during the LC, DC, and DL attacks, were shown in Table 5.14, 5.15 and 5.16, respectively. DES revealed the key in all rounds of the LC, DC, and DL attacks, resulting in a 100% attack because the key was revealed in all 16 rounds. The DES algorithm has 16 rounds.

AES revealed the key in 6 rounds during LC and DC attacks, that 42.8571% attack since AES is 14 round algorithm. AES revealed the key after four rounds during the DL attack, a 28.5714% attack. Refer to Table 5.14, 5.15 and 5.16.

Camellia revealed the key after six rounds of LC, DC, and DL attacks, resulting in a 42.8571% attack because Camellia is a 16-round algorithm. The 42.8571% attack occurred before Camellia was subjected to the Khumbelo function. The results of the Khumbelo function showed that no keys were revealed, and no round was attacked because there were no LAT, DDT, or DLCT constructions. As a result, LC, DC, and DL attacks on K_Camellia were impossible. Refer to Table 5.14, 5.15 and 5.16.

The experimental and theoretical results of data complexity during LC, DC, and DL attacks were shown in Table 5.17, 5.18, and 5.19, respectively. DES demonstrated a key after complexity of 2^{43}, $2^{42.5}$, and $2^{42.5}$ during the LC, DC, and DL attacks, respectively. AES demonstrated a key after complexity of 2^{128}, 2^{131}, and $2^{238.5}$ during the LC, DC, and DL attacks, respectively. In contrast, Camellia demonstrated a key after complexity of $2^{249.3}$, $2^{238.2}$, and 2^{89} during the LC, DC, and DL attacks, respectively. That was before the use of the Khumbelo function. After applying the Khumbelo function to the K_Camellia, there was no complexity because the Khumbelo function prevented the construction of all distinguishers used on attacks. Refer to

Table 5.17 to 5.19.

The memory allocation of various algorithms is shown in Table 5.20 and Figure 5.19. DES required more memory, 16.805 Megabytes, than AES, which required 13.388 Megabytes. As a result, regarding memory allocation, AES had a clear advantage over DES. Camellia and K_Camellia, on the other hand, required 8.924 and 8.985 Megabytes for memory allocation, respectively. If 8.924 and 8.985 Megabytes were rounded off, Camellia and K_Camellia had a total memory allocation of 9 Megabytes for installation. As a result, the Khumbelo function did not affect the memory of the original Camellia compared to the K_Camellia.

Table 5.5: Feasibility and Number of Entities needed to Construct LAT

Algorithm Name	**S-Box size**	**Feasibility**	**Number of Entities**
DES	4 x 4	Easy to construct	2^4 x 2^4=256
AES	8 x 8	Easy to construct	2^8 x 2^8 = 65536
Camellia	8 x 8	Easy to construct	2^8 x 2^8 = 65536
K_Camellia	8 x 32	Impossible to construct. No ordinary computer can compute each entity of 2^8 x 2^{32} = 1099511627776 without crashing due to memory constraints. Therefore Khumbelo function blocked the construction of LAT to prevent LC attack.	2^8 x 2^{32} = 1099511627776

All algorithms encrypted and decrypted the same image successfully. Re-

Table 5.6: Feasibility and Number of Entities needed to Construct DDT

Algorithm Name	**S-Box size**	**Feasibility**	**Number of Entities**
DES	4 x 4	Easy to construct	2^4 x $2^4 = 256$
AES	8 x 8	Easy to construct	2^8 x $2^8 = 65536$
Camellia	8 x 8	Easy to construct	2^8 x $2^8 = 65536$
K_Camellia	8 x 32	Impossible to construct. No ordinary computer can compute each entity of 2^8 x $2^{32} = 1099511627776$ without crashing due to memory constraints. Therefore Khumbelo function blocked the construction of DDT to prevent DC attack.	2^8 x $2^{32} = 1099511627776$

Table 5.7: Feasibility and Number of Entities needed to Construct DLCT

Algorithm Name	**S-Box size**	**Feasibility**	**Number of Entities**
DES	4 x 4	Easy to construct	2^4 x $2^4 = 256$
AES	8 x 8	Easy to construct	2^8 x $2^8 = 65536$
Camellia	8 x 8	Easy to construct	2^8 x $2^8 = 65536$
K_Camellia	8 x 32	Impossible to construct. No ordinary computer can compute each entity of 2^8 x $2^{32} = 1099511627776$ without crashing due to memory constraints. Therefore Khumbelo function blocked the construction of DLCT to prevent DL attack.	2^8 x $2^{32} = 1099511627776$

Table 5.8: Probability Results of LAT

Algorithm Name	**Experimental Highest Probability**	**Theoretical Highest Probability**	**Remarks**
DES	$\frac{6}{16} = 37.5000\%$	Same as theoretical results given by [100] that is 37.5000%	The results showed the feasibility of LAT construction on DES. Therefore DES is vulnerable to LC-related attacks
AES	$\frac{144}{256} = 56.2500\%$	Same as theoretical results given by [101] that is 56.2500%	The results showed the feasibility of LAT construction on AES. Therefore AES is vulnerable to LC-related attacks
Camellia	$\frac{144}{256} = 56.2500\%$	Camellia is the AES candidate and inspiration for S-Box [104]. Therefore LAT of Camellia is the same as the LAT Camellia, which is 56.2500%.	The results showed the feasibility of LAT construction on Camellia. Therefore, Camellia is vulnerable to LC-related attacks
K_Camellia	No probability since the code crashed due to memory needed for computation. The Khumbelo function blocked the construction of LAT.	K_Camellia is a new algorithm for this study. Therefore the would be no theoretical LAT results in the literature review.	The results showed NO feasibility of LAT construction on K_Camellia, therefore K_Camellia is NOT vulnerable to LC related attacks

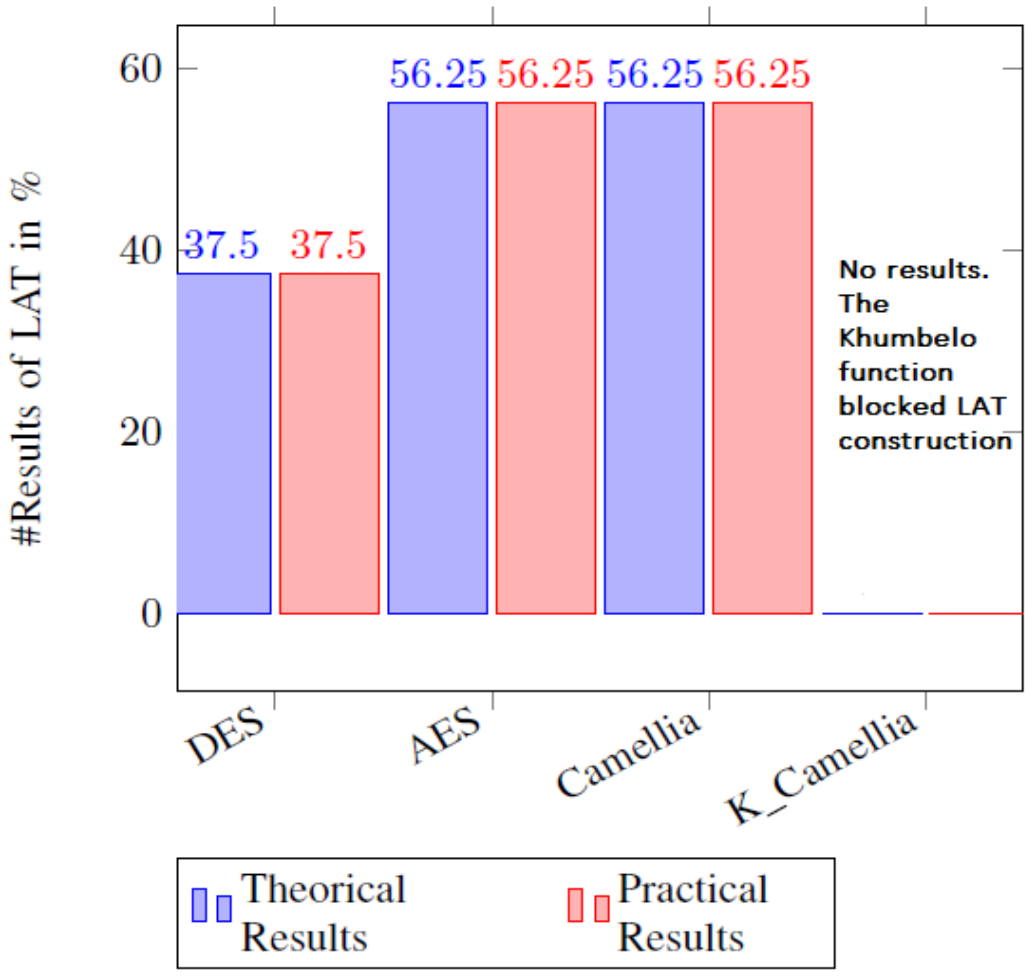


Figure 5.13: The Graph of LAT Results

Table 5.9: Probability Results of DDT

Algorithm Name	**Experimental Highest Probability**	**Theoretical Highest Probability**	**Remarks**
DES	$\frac{4}{16} = 6.25\%$	Same as theoretical results given by [100] that is 6.25%.	The results showed the feasibility of DDT construction on DES. Therefore DES is vulnerable to DC-related attacks
AES	$\frac{4}{256} = 1.5625\%$	1.5625%. Deduced from calculation [234]	The results showed the feasibility of DDT construction on AES. Therefore AES is vulnerable to DC-related attacks
Camellia	$\frac{4}{256} = 1.5625\%$	Camellia is the AES candidate and inspiration for S-Box [104]. Therefore DDT of Camellia is the same as the DDT Camellia, which is 1.5625%.	The results showed the feasibility of DDT construction on Camellia. Therefore Camellia is vulnerable to DC-related attacks
K_Camellia	No probability since the code crashed due to memory needed for computation. The Khumbelo function blocked the construction of DDT.	K_Camellia is a new algorithm for this study. Therefore the would be no theoretical DDT results in the literature review.	The results showed NO feasibility of DDT construction on K_Camellia, K_Camellia is NOT vulnerable to DC related attacks

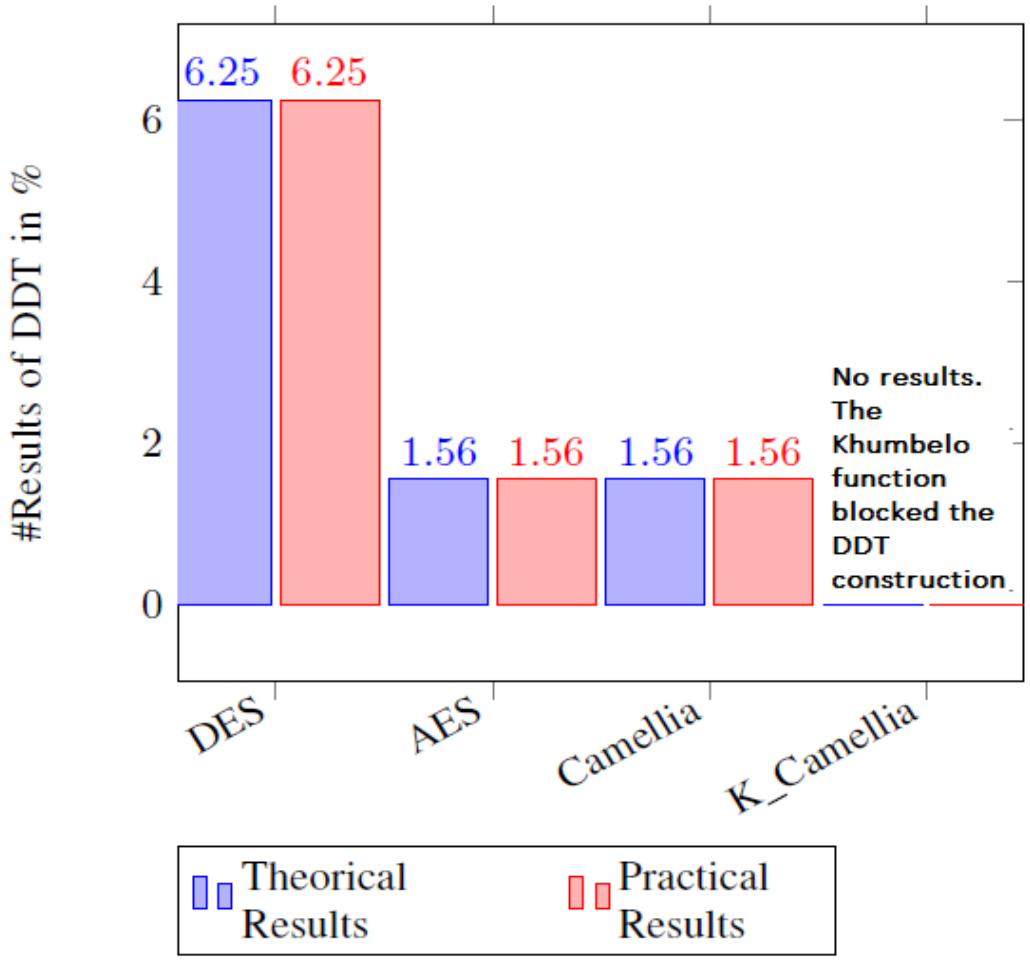


Figure 5.14: The Graph of DDT Results

Table 5.10: Probability Results of DLCT

Algorithm Name	**Experimental Highest Probability**	**Theoretical Highest Probability**	**Remarks**
DES	$\frac{8}{16} = 50.0\%$	Same as theoretical results given by [216] that is 50.0%	The results showed the feasibility of DLCT construction on DES. Therefore DES is vulnerable to DL-related attacks
AES	$\frac{128}{256} = 50.0\%$	Same as theoretical results given by [216] that is $\frac{1}{2} = 50.0\%$	The results showed the feasibility of DLCT construction on AES. Therefore AES is vulnerable to DL-related attacks
Camellia	$\frac{128}{256} = 50.0\%$	Camellia is the AES candidate and inspiration for S-Box [104]. Therefore DLCT of Camellia is the same as the DLCT of AES, 50.0%	The results showed the feasibility of DLCT construction on Camellia. Therefore Camellia is vulnerable to DL-related attacks
K_Camellia	No probability since the code crashed due to memory needed for computation. The Khumbelo function blocked the construction of DLCT.	K_Camellia is a new algorithm for this study. Therefore the would be no theoretical DLCT results in the literature review.	The results showed NO feasibility of DLCT construction on K_Camellia, therefore K_Camellia is NOT vulnerable to DL related attacks

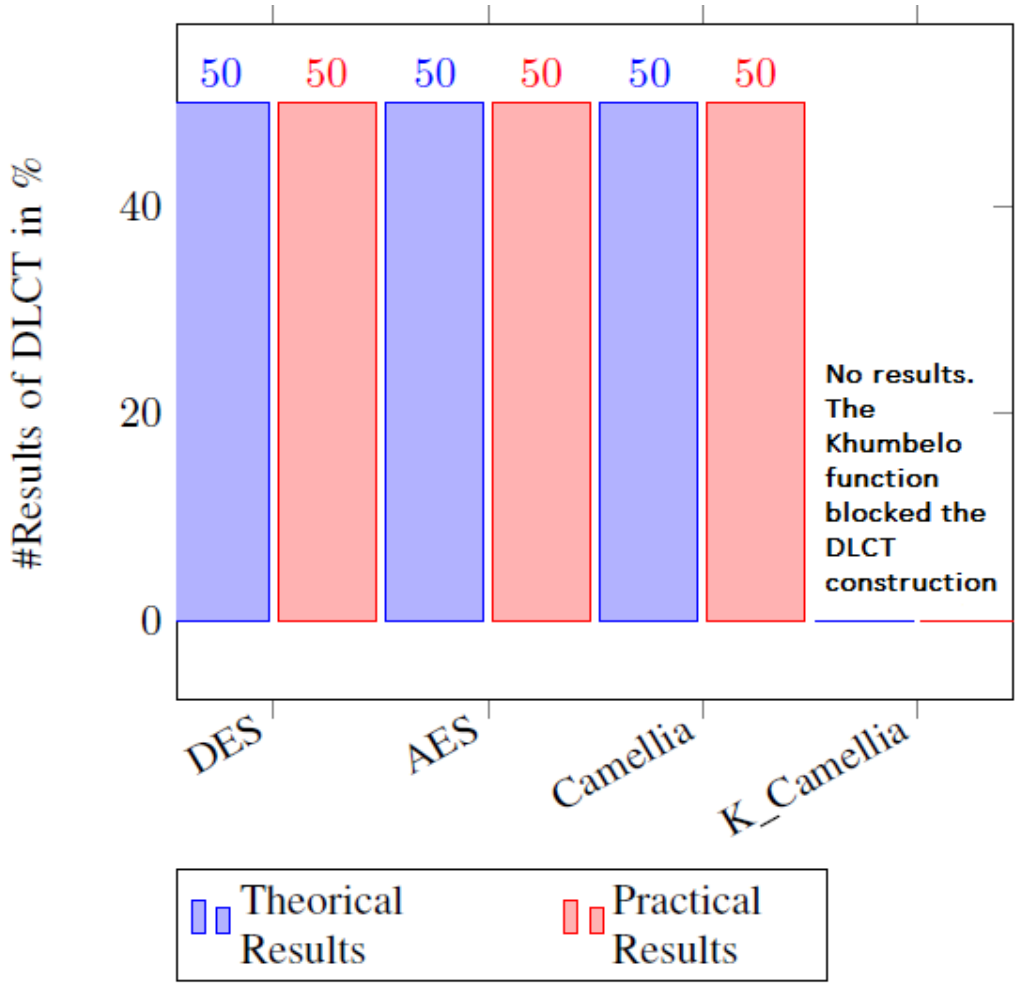


Figure 5.15: The Graph of DLCT Results

Table 5.11: Results of Key Avalanche Effect

Name of Algorithm	**Experimental Key Avalanche Effect in Percentage**	**Theoretical Key Avalanche Effect in Percentage**	**Remarks**
DES	43.6270%	43.8720% given by [215]	Failed SAC since out of range (from 45% to 55%)
AES	49.5788%	49.0661% given by [215]	Passed SAC within (45% to 55%)
Camellia	50.6184%	49.6093% given by [215]	Passed SAC within (45% to 55%)
K_Camellia	50.5655%	K_Camellia is a new algorithm for this study. Therefore, theoretical results are yet to be available.	Passed SAC within (45% to 55%). Therefore Khumbelo function maintained the Avalanche Effect of K_Camellia to between 45% to 55%, which is acceptable in cryptography.

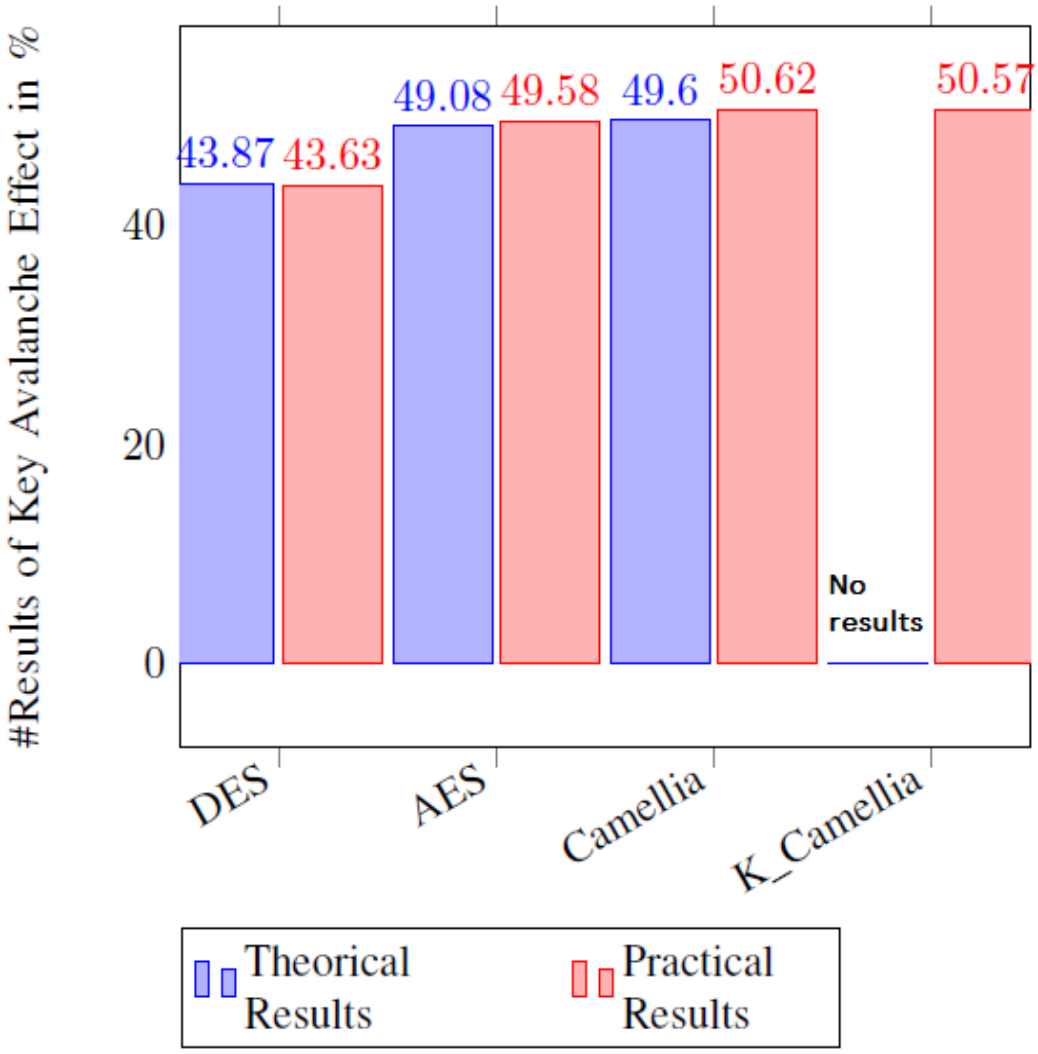


Figure 5.16: The Key Avalanche Effect Results

Table 5.12: Results of Plaintext Avalanche Effect

Name of Algorithm	**Experimental Plaintext Avalanche Effect in Percentage**	**Theoretical Plaintext Avalanche Effect in Percentage**	**Remarks**
DES	62.1337%	62.8662% given by [215]	Failed SAC since out of range (from 45% to 55%)
AES	50.5371%	49.7924% given by [215]	Passed SAC within (45% to 55%)
Camellia	50.2502%	49.4689% given by [215]	Passed SAC within (45% to 55%)
K_Camellia	49.3774%	K_Camellia is a new algorithm for this study. Therefore, theoretical results are yet to be available.	Passed SAC within (45% to 55%). Therefore Khumbelo function maintained the Avalanche Effect of K_Camellia to between 45% to 55%, which is acceptable in cryptography

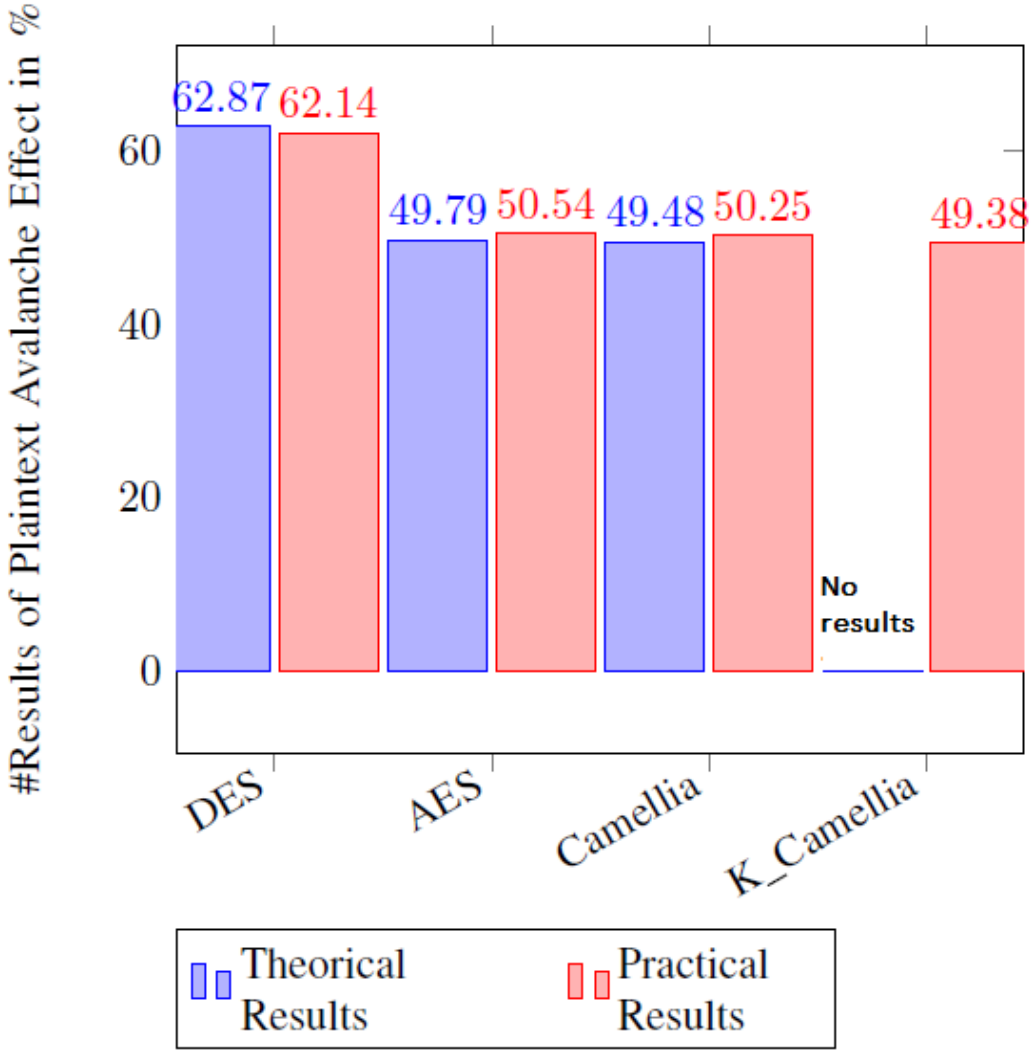


Figure 5.17: The Plaintext Avalanche Effect Results

Table 5.13: Time and Speed

Algorithm Name	**File Encrypted in Megabyte**	**Time in seconds**	**Experimental Speed in MB/seconds**	**Theoretical Speed in MB/seconds**
DES	256	6.2312	41.0835	21.340 [235]
AES	256	4.6180	55.4352	48.229 [235]
Camellia	256	4.6183	55.4316	77.34 [198]
K_Camellia	256	4.8723	52.5419	No theoretical results because K_Camellia is a new algorithm first coined and developed in this study.

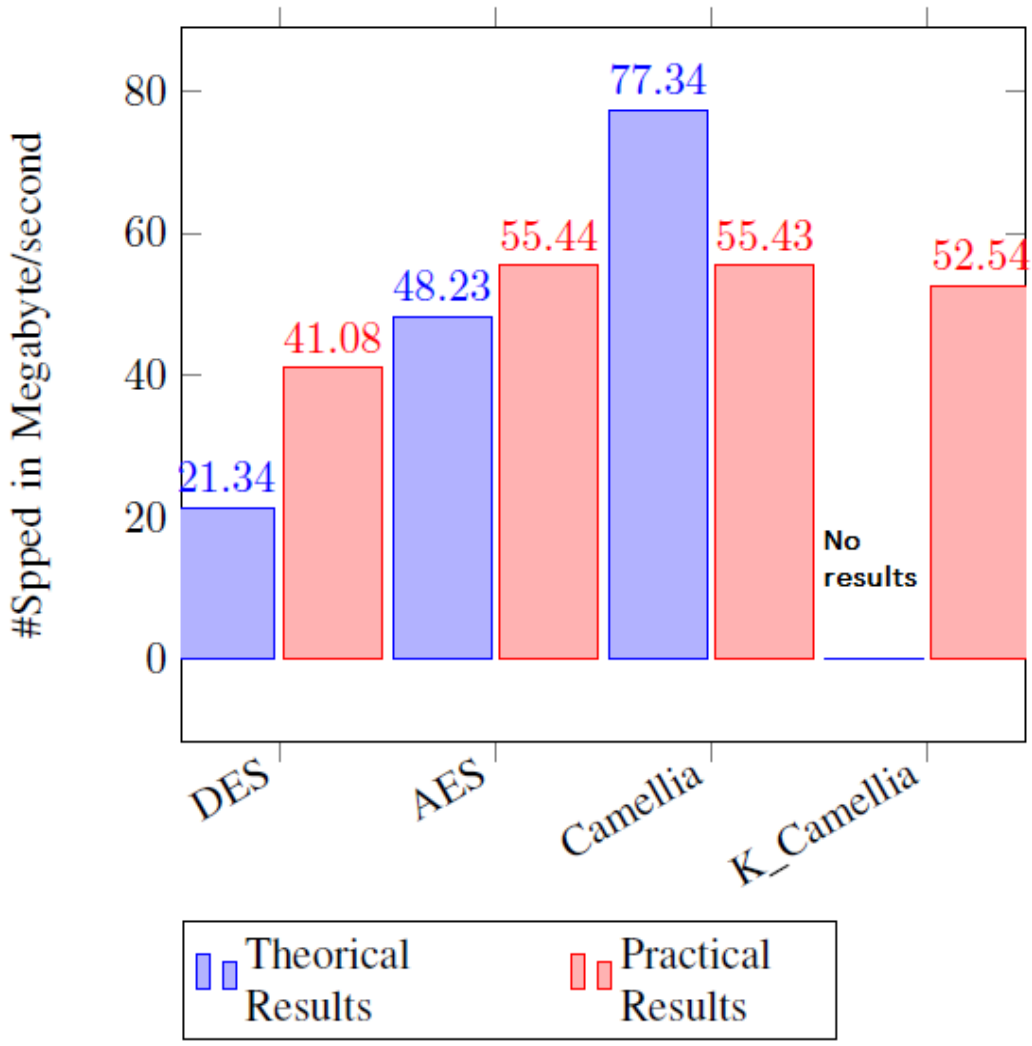


Figure 5.18: Results of Speed

Table 5.14: Discovery of Encryption Keys in Number of Rounds during LC Attack

Algorithm Name	**Theoretical Rounds Attacked**	**Experimental Rounds Attack**
DES	16 rounds, that is $\frac{16}{16}$ = 100%	16 rounds, that is $\frac{16}{16}$ = 100%
AES	6 rounds, that is $\frac{6}{14}$ = 42.8571% [101] - [102].	14 rounds, that is $\frac{6}{14}$ = 42.8571%
Camellia	6 rounds, that is $\frac{14}{16}$ = 87.5% [225], [195], [228], [229].	6 rounds, that is $\frac{14}{16}$ = 87.5%
K_Camellia	No round attacked. K_Camellia is a new algorithm coined in this study. Therefore no theoretical results	No keys and rounds were cracked since there was no LAT construction after Khumbelo function was applied. Therefore LC attack was impossible

Table 5.15: Discovery of Encryption Keys in Number of Rounds during DC Attack

Algorithm Name	**Theoretical Rounds Attacked**	**Experimental Rounds Attack**
DES	16 rounds, that is $\frac{16}{16}$ = 100%	16 rounds, that is $\frac{16}{16}$ = 100%
AES	6 rounds, that is $\frac{6}{14}$ = 42.8571% [101] - [102].	6 rounds, that is $\frac{6}{14}$ = 42.8571%
Camellia	14 rounds, that is $\frac{14}{16}$ = 87.5% [225], [195], [228], [229].	14 rounds, that is $\frac{14}{16}$ = 87.5%
K_Camellia	No round attacked. K_Camellia is a new algorithm coined in this study. Therefore no theoretical results	No keys and rounds were cracked since there was no DDT construction after Khumbelo function was applied. Therefore DC attack was impossible

Table 5.16: Discovery of Encryption Keys in Number of Rounds during DL Attack

Algorithm Name	**Theoretical Rounds Attacked**	**Experimental Rounds Attack**
DES	All rounds, that is 100%	All rounds, that is 100%
AES	4 rounds, that is $\frac{4}{14}$ = 28.5714% [101] - [102].	14 rounds, that is $\frac{14}{14}$ = 100%
Camellia	14 rounds, that is $\frac{14}{16}$ = 87.5% [225], [195], [228], [229].	14 rounds, that is $\frac{14}{16}$ = 87.5%
K_Camellia	No round attacked. K_Camellia is a new algorithm coined in this study. Therefore no theoretical results	No keys and rounds were cracked since there was no DLCT construction after Khumbelo function was applied. Therefore DL attack was impossible

Table 5.17: Data Complexity during LC Attack

Algorithm Name	**Theoretical Complexity**	**Experimental Complexity**
DES	2^{43} [103]	2^{43}
AES	2^{128} [101] - [102].	2^{131}
Camellia	$2^{249.3}$ [229].	$2^{249.3}$
K_Camellia	No complexity. K_Camellia is a new algorithm coined in this study. Therefore no theoretical results	No complexity after the Khumbelo function was applied since no LAT was feasible.

Table 5.18: Data Complexity during DC Attack

Algorithm Name	**Theoretical Complexity**	**Experimental Complexity**
DES	$2^{42.5}$ [103]	$2^{42.5}$
AES	2^{131} [101] - [102].	2^{131}
Camellia	$2^{238.2}$ [225].	$2^{238.2}$
K_Camellia	No complexity. K_Camellia is a new algorithm coined in this study. Therefore no theoretical results	No complexity after the Khumbelo function was applied since no DDT was feasible.

Table 5.19: Data Complexity during DL Attack

Algorithm Name	**Theoretical Complexity**	**Experimental Complexity**
DES	$2^{42.5}$ [103]	$2^{42.5}$
AES	2^{131} [101] - [102].	2^{131}
Camellia	2^{89} [228].	2^{89}
K_Camellia	No complexity. K_Camellia is a new algorithm coined in this study. Therefore no theoretical results	No complexity after the Khumbelo function was applied since no DLCT was feasible.

Table 5.20: Memory Needed to Install Algorithm

Algorithm Name	**Memory Needed in Megabyte**
DES	15.019
AES	13.388
Camellia	8.924
K_Camellia	8.983 Therefore, the Khumbelo function marginally increased the memory of K_Camellia, compared to Camellia's memory.

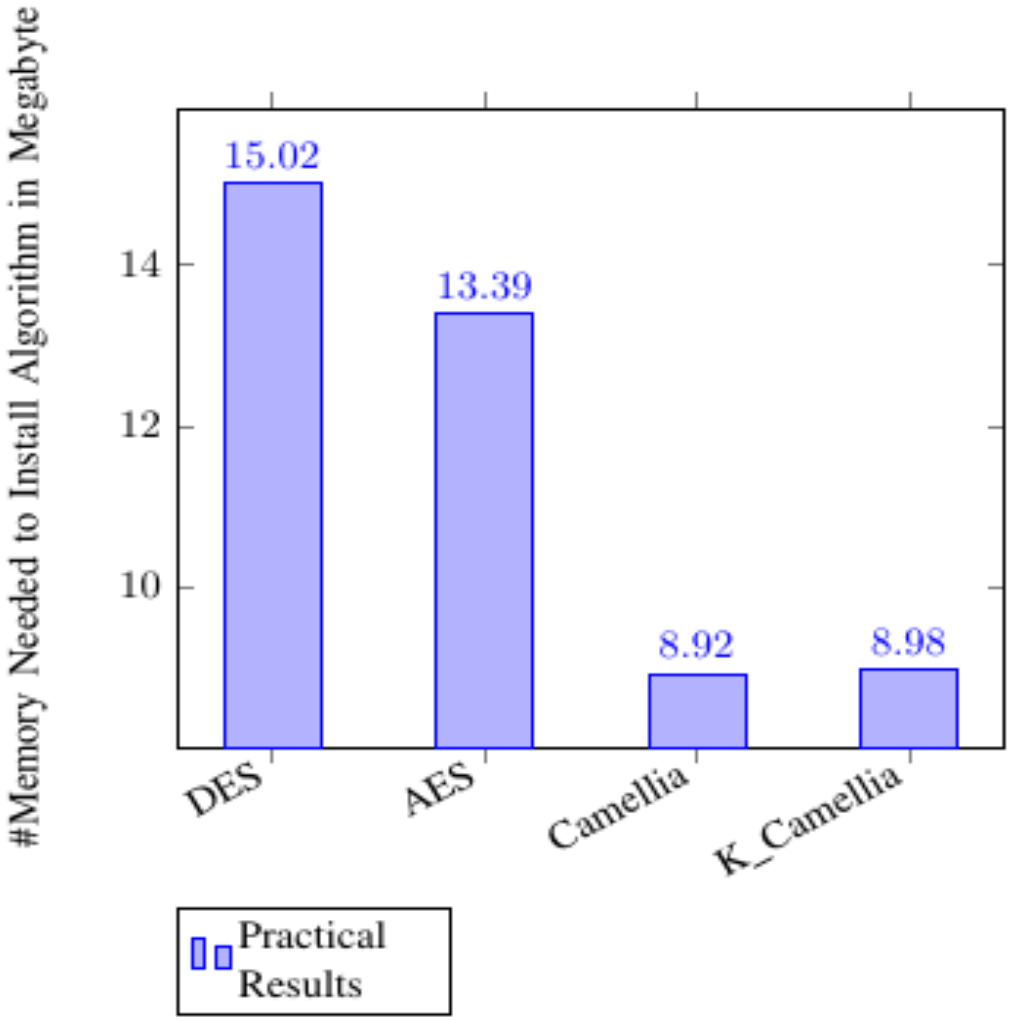


Figure 5.19: Memory Needed for Installation of Different Algorithms

fer to Figure 5.20, 5.21, and 5.22 for more information. Figure 5.20 showed the encryption and decryption process of the AES. Figure 5.21 showed the encryption and decryption process of the Camellia. Figure 5.22 showed the encryption and decryption process of the newly K_Camellia. The difference among Figure 5.20, 5.21, and 5.22 were encryption images. The original and decrypted images were the same in Figure 5.20, 5.21, and 5.22. The encrypted images in Figure 5.21 and Figure 5.22 were not the same after applying the Khumbelo function on Camellia. The new K_Camellia algorithm was the product of applying the Khumbelo function. The newly K_Camellia algorithm encrypted the image differently than the Camellia.

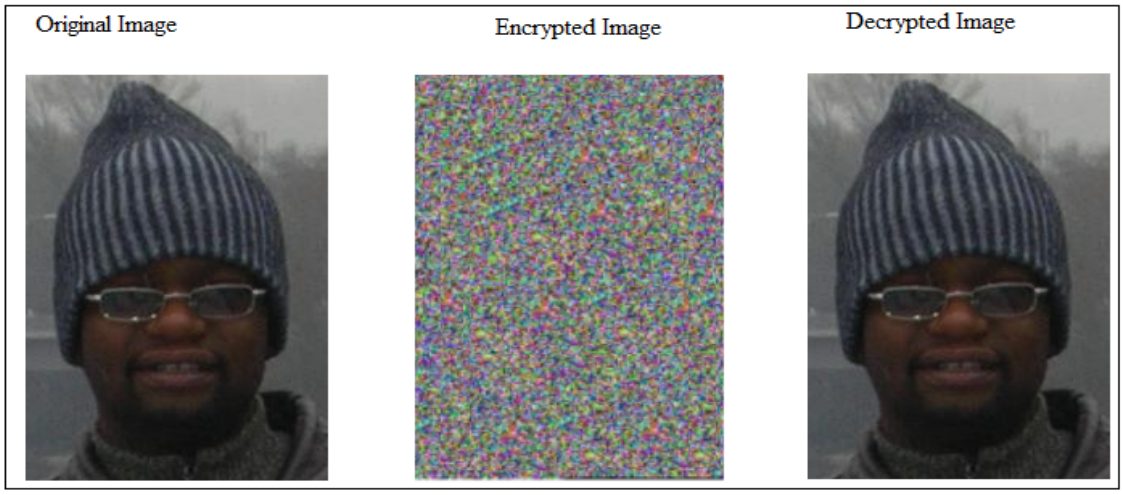


Figure 5.20: Image Encryption and Decryption Using C++ of the AES Algorithm

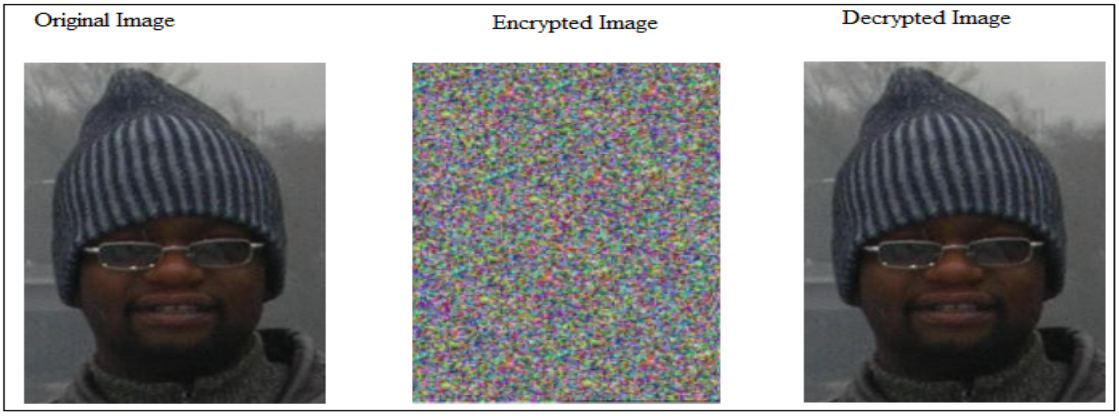


Figure 5.21: Image Encryption and Decryption Using C++ of the Traditional Camellia Algorithm

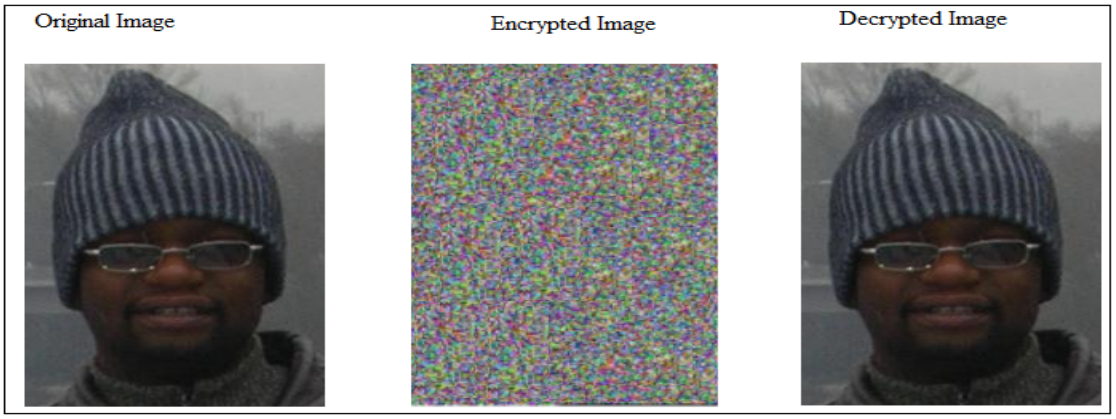


Figure 5.22: Image Encryption and Decryption Using C++ of the K_Camellia Algorithm

5.11 Summary of Using Khumbelo Function to Prevent DC Attacks on Camellia

Camellia was found to be one of the encryption algorithms implemented in many Internet of Things (IoT) devices. However, intruders attacked the Camellia cipher using S-Boxes' distinguishers. The distinguishers are tables that provide probabilities of guessing the algorithms' secret key. The distinguishers were found to be used in the majority of attacks. The most well-known distinguishers were found to be Linear Approximation Table (LAT), Difference-Distribution Table (DDT), and Differential-Linear Connectivity Table (DLCT). This research focused on preventing these attacks by using a novel function called the Khumbelo function to distract the construction of S-Box distinguishers. The Khumbelo function distracted the distinguishers' construction by reducing the probability of construction. The Khumbelo function successfully reduced the attack probability of LAT (from 54.6875 percent to 0 percent), DDT (from 1.5625 percent to 0 percent), and DLCT (from 50.0000 percent to 0 percent). The Khumbelo function was generated using a 4-Byte output S-Box instead of an original 1-Byte output S-Box in Camellia. Additionally, the Khumbelo function was composed of many modulo operators. The new 4-Byte output S-Box and modulo operators confused and blocked intruders from constructing distinguishers.

In the future, the study will apply the Khumbelo function to algorithms such as Blowfish, Magenta, and Skipjack. The attacks studied were all variants of LC and DC attacks. The Khumbelo function will be tested to prevent other upcoming attacks unrelated to LC and DC attacks.

Chapter 6

Conclusion and Objectives Evaluation

In summary, this chapter serves as the essential supporting structure, tying together the study results to represent the success of preventing cryptographic attacks on the Internet of Things (IoT). This chapter starts by evaluating the objectives of this research and then summarizes key research contributions made in this work. It then makes additional research recommendations before finalizing remarks.

6.1 Research Contribution and Objective Evaluation

The study aimed to prevent cryptographic attacks on the Internet of Things (IoT). The study produced remarkable results in preventing cryptographic attacks from IoT devices using the KDM function, Khumbelo function, and Blocker function. Each study objective is revisited to evaluate the research findings critically. Reviewing each study objective emphasizes the crucial areas where this research constructed an original contribution.

Objective (i.) To use the KDM function to prevent DC attack in the AES algorithm used on IoT devices.

The study invented a novel approach called the Khumbelo Difference Muthavhine (KDM) function to prevent DC attacks. The KDM function was tested on AES.

Contribution:

A. The KDM function protects against DC attacks.

B. Cryptography and IoT researchers are being introduced to the KDM function.

Objective (ii.) To use the Blocker function to prevent DL attack in the Serpent algorithm used on IoT devices.

The study invented a novel approach called the Blocker function to prevent DL attacks. The Blocker function was tested on Serpent.

Contribution:

A. The Blocker function protects against DC attacks.

B. Cryptography and IoT researchers are being introduced to the Blocker function.

Objective (iii.) To use the Khumbelo function to prevent LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks in the Camellia algorithm used on IoT devices.

The study invented a novel approach called the Khumbelo function to prevent LC and DC attacks. The Khumbelo function was tested on Camellia.

Contribution:

A. The Khumbelo function provides protection against LC, DC, DL, boomerang, truncated differential, meet-in-the-middle, and zero-correlation-linear-distinguisher attacks.

B. Cryptography and IoT researchers are being introduced to the Khumbelo function.

Objective (iv.) To publish papers with an internal and external organization such as MDPI and IEEE.

This study produced journal and conference papers.

Contribution:

A. Journal papers produced out of this study are published in MDPI.

B. Conference papers produced out of this study are published in IEEE.

6.2 Future Research Recommendations and Suggestions

Extensive research incorporating the following recommendations and future work will be required based on the findings of this thesis.

i. To recommend and suggest using the KDM function in the future to prevent other attacks on the various algorithms (like Blowfish, Twofish and Rivest-Shamir-Adleman (RSA)) that were not covered by this study [236].

ii. To recommend and suggest using the Blocker function in the future to prevent other attacks on the various algorithms that were not covered by this study.

iii. To recommend and suggest using the Khumbelo function in the future to prevent other attacks on the various algorithms that were not covered by this study.

iv. To publish journal papers with the organization like MDPI and IEEE.

6.3 Closing Statements

The results described in this study show that the KDM function, Blocker function, and Khumbelo function are promising methods for preventing vari-

ous attacks on the AES, Serpent, and Camellia. The KDM function, Khumbelo function, and Blocker function managed to prevent cryptographic attacks since all 8 x 8 S-Boxes are changed to 8 x 32 S-Box depending on the particular chapter. The 8 x 32 S-Box was expected to build distinguishers from 2^8 x 2^{32} = 256 x $4,294,967,296$ matrices with $1,099,511,627,776$ elements. Due to memory constraints, an ordinary computer could not compute 256 x $4,294,967,296$ matrices. This study was initially thought to be impossible until this study demonstrated the feasibility of using the KDM function, Blocker function, and Khumbelo function to prevent various cryptographic attacks on IoT devices. Researchers who read journal and conference papers from this thesis were inspired to use the KDM, Blocker, Muthavhine, and Khumbelo functions.

Bibliography

[1] S. Millar, "IoT Security Challenges and Mitigations: An Introduction," Rapid7 LLC, pp. 1-5, 2022.

[2] V. O. Nyangaresi, A. J. Rodrigues, and S. O. Abeka, "Secure Algorithm for IoT Devices Authentication," pp. 1-22, 2023.

[3] I. Kuzminykh, M. Yevdokymenko, and V. Sokolov, "Encryption Algorithms in IoT: Security vs Lifetime," Researchgate, pp. 1-21, 2021.

[4] A. Čolaković and M. Hadzialic, "Internet of Things (IoT): A Review of Enabling Technologies, Challenges, and Open Research Issues", Computer Networks, pp. 11-39, 2018.

[5] H. Suo, J. Wan, C. Zou and J. Liu, "Security in the Internet of Things: A Review" 2012 International Conference on Computer Science and Electronics Engineering, pp. 648-651, 2012.

[6] Z. H. Hu, "The Research of Several Key Question of Internet of Things," International Conference on Intelligence Science and Information Engineering, pp. 362-365, 2011.

[7] K. D. Muthavhine and M. Sumbwanyambe, "An Analysis and a Comparative Study of Cryptographic Algorithms used on the Internet of Things (IoT) Based on Avalanche Effect", International Conference on Information and Communications Technology (ICOIACT), 2018.

[8] L. Atzori, A. Iera, and G. Morabito, "Understanding the Internet of Things: Definition, Potentials, and Societal Role of a Fast Evolving Paradigm" Ad Hoc Networks, Elsevier, pp. 1-22, 2016.

[9] K. Ashton, "How to Fly a Horse: The Secret History of Creation, Invention, and Discovery", 2015.

[10] P. P. Ray, "A Survey on Internet of Things Architectures," Journal of King Saud University - Computer and Information Sciences, Vol 30, No. 3, pp. 291-319, 2018.

[11] F. Pacheco-Torgal, E. Rasmussen *et al.*, "Start-Up Creation: The Smart Eco-efficient Built Environment", One-IT Smart, pp. 1-3, 2018.

[12] K. K. Patel and S. M. Patel, "Internet of Things-IOT: Definition, Characteristics, Architecture, Enabling Technologies, Application and Future Challenges", International Journal of Engineering Science and Computing, Vol. 6, No. 5, 2019.

[13] P. Singh, "Internet of Things (IoT): A Literature Review", International Research Journal of Engineering and Technology (IRJET), Vol. 3, No. 12, 2016.

[14] TechTarget IoT Agenda, "Using Digital Twin Tech to Solve IoT Issues".

[15] IEEE, "IEEE P1451.6 Terms and Definitions," 2022.

[16] J. Kouns, "Bring Your Own Internet of Things BYO-IoT" 2015 RSA Conference, pp. 4-5, 2015.

[17] J. Nendick, "Internet of Things Human-Machine Interactions that Unlock possibilities," Media and Entertainment. 2011 Sony Corporation, Vol. 6114, pp. 1-19, 2011, 2022.

[18] A. Zahedi, “Intension to Adopt Smart Cards” Lulea University of Technology, pp. 1-23, 2022.

[19] Security Technology Alliance, “Smart Card Primer”, pp. 1-23, 2022.

[20] A. Mahajan, A. Verma and D. Pahuja, “Smart Card: Turning Point of Technology”, International Journal of Computer Science and Mobile Computing, IJCSMC, Vol. 3, No. 10, pp. 982 – 987, 2014.

[21] M. Hitchcock, “The End of Money: Bible Prophecy and the Coming Economic Collapse”, 2013.

[22] FirstData, “EMV: A to Z (Terms and Definitions)”, pp. 1-23, 2022.

[23] D. Kearns, ”Where to Learn about Smart Cards”. Network World, 2007.

[24] CardPlus, “Cards Plus Terminology” pp. 1-23, 2022.

[25] A. A. Eteng, S. K. A. Rahim and C. Y. Leow, “RFID in the Internet of Things”, Wiley Online Library, 2018.

[26] A. Ullah, “IoT: Applications of RFID and Issues”, International Journal of Internet of Things and Web Services, Vol. 3 pp. 1- 5, 2018.

[27] S. Maharjan, “RFID and IOT: An Overview”, Simula Research Laboratory University of Oslo, pp. 1-25, 2010.

[28] M. Kaur, M. Sandhu, N. Mohan and P. S. Sandhu, “RFID Technology Principles, Advantages, Limitations and Its Applications”, International Journal of Computer and Electrical Engineering, Vol. 3, No.1, pp. 151 - 157, 2011.

[29] IoT Agenda, “RFID (Radio Frequency Identification)”, pp. 1-23, 2022.

[30] C. Jechlitschek, “A Survey Paper on Radio Frequency Identity (RIFD) Trends” Inside the Internet of Things (IoT)”, Radio Frequency Identity RIFD, pp. 1- 13, 2023.

[31] D. Christin, A. Reinhardt, P. S. Mogre and R. Steinmetz, "Wireless Sensor Networks and the Internet of Things: Selected Challenges", pp. 32- 34, 2009.

[32] M. Nkomo, G.P. Hancke, A. M. Abu-Mahfouz, S. Sinha and A. J. Onumanyi, "Overlay Virtualized Wireless Sensor Networks for Application in Industrial Internet of Things: A Review", Vol. 18, No: 10, 2018.

[33] A. A. Halim, N. M. Hassan, A. Zakaria, L. M. Kamarudin and A. H. A. Bakar, "Internet of Things Technology for Greenhouse Monitoring and Management Systems Based on Wireless Sensor Network", ARPN Journal of Engineering and Applied Sciences, Vol. 11, No. 22, pp. 13169-13175, 2016.

[34] H. Zhou, "The Internet of Things in the Cloud", CRC Press Taylor and Francis Group, pp. 146-321, 2013.

[35] F. Lewis, "Introduction to Crossbow Mica2 Sensors", Automation and Robotics Research Institute University of Texas at Arlington, 2017.

[36] Mica2, "Wireless Measurement System",Mica, pp. 1-23, 2022.

[37] M. G. C. Torres, "Energy Consumption in Wireless Sensor Networks Using GSP", Electronics Engineer, Universidad Pontificia Bolivariana, Medellín, Colombia, 2006.

[38] O. Gunnsteinsson, "A Search for a Convenient Data Encryption Algorithm for an Internet of Things Device", Chalmers University of Technology, 2016.

[39] P. Walters, "The Risks of Using Portable Devices", 2016 The United States Computer Emergency Readiness Team (US-CERT), pp. 1-3, 2016.

[40] L. Chang, R. Steinfied, C. Jakob, A. Peel, T. Dinh, and J. Newson, "Surprising Developments in Artificial Intelligence Cryptography", Scram Software Securing Data in the Cloud, 2017.

[41] K. M. Alallayah, W. F. A. El-Wahed, M. Amin and A. H. Alhamam, "Attack of Against Simplified Data Encryption Standard Cipher System Using Neural Networks", Journal of Computer Science, Vol. 6, No. 1, pp. 29-35, 2010.

[42] S. Prajapat, A. Thakur, K. Maheshwari, and R. S. Thakur, "Cryptic Mining in Light of Artificial Intelligence", (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 6, No. 8, 2015.

[43] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms", International Workshop on Selected Areas in Cryptography, Springer, Vol. 2012, pp. 39-56, 2012.

[44] B. Sreenivasa, M. Kumar, M. N. Shaikh and S. Sathyanarayana, "A Study on Encryption Decryption Algorithm for Big Data Analytics in Cloud", International Journal of Latest Trends in Engineering and Technology Special Issue SACAIM, pp. 323-329, 2016.

[45] F. J. Rodríguez-Lera, V. Matellán-Olivera, J. Balsa-Comerón, Á. M. Guerrero-Higueras and C. Fernández-Llamas, "Message Encryption in Robot Operating System: Collateral Effects of Hardening Mobile Robots", Frontiers In ICT, pp. 1-3, 2018.

[46] L. Wei and *et al.*, "An Effective Differential Fault Analysis on the Serpent Cryptosystem in the Internet of Things", School of Computer Science and Technology, Donghua Univ., Shanghai, Vol. 11, 2014.

[47] J. Daemen and V. Rijmen, "AES Proposal: Rijndael", Joan Daemen and Vincent Rijmen, pp. 1-45, 2023.

[48] FIPS Publication 197, "Announcing the Advanced Encryption Standard (AES)", Federal Information Processing Standards Publication, pp. 1-25, 2001.

[49] J. Nechvatal *et al.*, "Report on the Development of the Advanced Encryption Standard (AES)", Journal of Research of the National Institute of Standards and Technology, Vol. 106, No. 3, pp. 511–577, 2001.

[50] B. Rothke, "A Look at the Advanced Encryption Standard (AES)", Information Security Management Handbook, 2007.

[51] K. Laxmi and N. A. Dawande, "Encryption Algorithms Used for Secured Communication", International Journal of Science and Research (IJSR), Vol. 2 No. 12, pp. 2319-7064, 2013.

[52] B. S. W. Poetro, "Implementation of 128 Bits Camellia Algorithm for Cryptography in Digital Image", IAES International Conference on Electrical Engineering, Computer Science and Informatics IOP Publishing, Vol. 7, No. 5A, pp. 1-23, 2011.

[53] A. M. Alabaichi, "Analysis of Some Security Criteria for S-boxes in Blowfish Algorithm", International Journal of Digital Content Technology and its Applications, Vol. 190, 2017.

[54] K. Aoki and *et al.*, "Specification of Camellia - a 128-bit Block Cipher", Nippon Telegraph and Telephone Corporation and Mitsubishi Electric Corporation, pp. 1- 33, 2001.

[55] M. Matsui, J. Nakajima and S. Moriai, "A Description of the Camellia Encryption Algorithm", Mitsubishi Electric Corporation and Sony Computer Entertainment Inc., pp. 1-35, 2014.

[56] S. Moriai and A. Kato, "Use of the Camellia Encryption Algorithm in Cryptographic Message Syntax (CMS)", Sony Computer Entertainment Inc. and NTT Software Corporation, 2014.

[57] Z. Čiča, "Pipelined Implementation of Camellia Encryption Algorithm", 24th Telecommunications Forum (TELFOR), Vol. 1449, pp. 339-348, 2016.

[58] R. G. Kammer and W. M. Daley, "Data Encryption Standards (DES)", The Federal Information Processing Standards Publication Series of the National Institute of Standards and Technology (NIST), pp. 1-33, 1999.

[59] K. Rabah, "Theory and Implementation of Data Encryption Standard: A Review", Information Technology Journal, Vol. 4, No. 4, pp. 307-325, 2005.

[60] P. T. Kenekayoro, "The Data Encryption Standard Thirty Four Years Later: An Overview", African Journal of Mathematics and Computer Science Research Vol. 3, No.10, pp. 267-269, 2010.

[61] C. Ding, "The Data Encryption Standard in Detail", Department of Computer Science Hong Kong University of Science and Technology, pp. 1-50,

[62] R. Anderson and E. Biham and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", Cambridge University, pp. 1-45, 2023.

[63] A. M. Nazlee, F. A. Hussin and F. A. Hussin, "Serpent encryption algorithm implementation on Compute Unified Device Architecture (CUDA)", Proceedings of 2009 Student Conference on Research and Development (SCOReD), 2009.

[64] T. Kohno, J. Kelsey and B. Schneier, "Preliminary Cryptanalysis of Reduced-Round Serpent", Reliable Software Technologies and Counterpane Internet Security, Inc., pp. 1-19, 2023.

[65] M. H. Taher, A. E. T. El Deen and M. E. Abo-Elsoud, "Hardware Implementation of the Serpent Block Cipher using FPGA technology", International Journal of Electronics and Communication Engineering and Technology (IJECET), Vol. 5, No. 10, pp. 34-44, 2014.

[66] NIST, "Skipjack and KEA Specifications", NIST, Vol. 2, pp. 1-23, 1998.

[67] H. Poston and K. Dhandhania, "Cryptographic Attacks: Types of Attacks with Examples, and How to Defend Against Them Common Lounge", pp. 1-3, 2022.

[68] H. Poston and K. Dhandhania, "Cryptographic Attacks: Types of Attacks with Examples, and How to Defend Against Them Common Lounge", pp. 1-3, 2022.

[69] O. Lo, W. J. Buchanan and D. Carson, "Power Analysis Attacks on the AES-128 S-box using Differential Power Analysis (DPA) and Correlation Power Analysis (CPA)", Journal of Cyber Security Technology, pp. 1-22, 2016.

[70] P. Kocher, J. Jaffe and B. Jun, "Introduction of Differential Power Analysis and Related Attacks", Cryptographic Research, pp. 1-9, 2022.

[71] Lecture Notes on Computer Systems, "Data Encryption Standard (DES)", Tribhuvan University Kathmandu, 2016.

[72] D. S. A. Elminaam, H. M. A. Kader and M. M. Hadhoud, "Tradeoffs between Energy Consumption and Security of Symmetric Encryption Algorithms", International Journal of Computer Theory and Engineering, Vol. 1, No. 3, pp. 1793-8201, 2009.

[73] B. J. Mohd and T. Hayajneh, "Lightweight Block Ciphers for IoT: Energy Optimization and Survivability Techniques", IEEE, Vol. 6, pp. 35966-36789, 2018.

[74] T. S. Messerges, E. A. Dabbish and R. H. Sloan, "Power Analysis Attacks of Modular Exponentiation in Smartcards", Springer-Verlag Berlin Heidelberg and CHES, Vol. 1717, pp. 144-157, 1999.

[75] E. Biham, "On Matsui's linear cryptanalysis", Workshop on the Theory and Application of Cryptographic Techniques, Advances in Cryptology — EUROCRYPT', Vol. 950, pp. 341–355, 1994.

[76] P. Junod, "Linear Cryptanalysis of DES", ETH Zurich University, pp. 1-75, 2022.

[77] H. M. Heys, "A Tutorial on Linear and Differential Cryptanalysis", Electrical and Computer Engineering Faculty of Engineering and Applied Science Memorial University of Newfoundland, pp. 1-33,

[78] A. D. Dwivedi, P. Morawiecki and S. Wojtowicz, "Differential-linear and Impossible Differential Cryptanalysis of Round-reduced Scream", Site Press Science and Technology Publications, pp. 501-506,

[79] B. Sullivan, "Preventing a Brute Force or Dictionary Attack: How to Keep the Brutes Away from Your Loot", SPI Dynamics, pp. 1-5,

[80] J. Pawlick and Q. Zhu, "Internet of Things: Privacy and Security in a Connected World", Transcript of Workshop at 182. Transcript of Workshop, pp. 5-55, 2015.

[81] R. Hosseinkhani and H. H. S. Javadi, "Using Cipher Key to Generate Dynamic S-Box in AES Cipher System", International Journal of Computer Science and Security (IJCSS), Vol. 6, No. 1, pp. 19-28, 2012.

[82] Y. Javed, A. Shahid Khan, A. Qahar and J. Abdullah, "Preventing DoS Attacks on IoT Using AES," Researchgate, pp. 55-60, 2018.

[83] J. Rokan, G. H Majeed and A. Farhan, "Internet of Things Security using New Chaotic System and Lightweight AES," Journal of Al-Qadisiyah for Computer Science and Mathematics, pp. 45-52, 2019.

[84] P. S. Munoz, N. Tran, B. Craig, B. Dezfouli and Y. Liu, "Analyzing the Resource Utilization of AES Encryption on IoT Devices," Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), pp. 1-8, 2018.

[85] B. M. Alshammari, R. Guesmi, T. Guesmi, H. Alsaif and A. Alzamil, "Implementing a Symmetric Lightweight Cryptosystem in Highly Constrained IoT Devices by Using a Chaotic S-Box," Symmetry, pp. 1-20, 2021.

[86] D. A. F. Saraiva, V. R. Q. Leithardt, D. de Paula, A. S. Mendes, G. V. González and P. Crocker, "PRISEC: Comparison of Symmetric Key Algorithms for IoT Devices," Sensors and MDPI, pp. 1-23, 2019.

[87] S. Shanthi Rekha and P. Saravanan, "Low-Cost AES-128 Implementation for Edge Devices on IoT Applications," Journal of Circuits, Systems and Computers, pp. 1-16, 2019.

[88] Lorawan Security, *Full End–to–End Encryption for IoT Application Providers*, Lora-Alliance, pp. 1-4, 2020.

[89] U. Farooq, N. U. Hasan, I. Baig and N. Shehzad, "Efficient adaptive framework for securing the Internet of Things devices," EURASIP Journal on Wireless Communications and Networking, pp. 1-16, 2019.

[90] V. Nandan and R. G. S. Rao, "An Efficient AES Algorithm for IoT-based Applications," International Journal of Engineering and Advanced Technology (IJEAT), pp. 1939-1944, 2019.

[91] K. D. Muthavhine and M. Sumbwanyambe, "An Analysis and a Comparative Study of Cryptographic Algorithms used on the Internet of Things (IoT) Based on Avalanche Effect," Unisa Institutional Repository, 2018.

[92] K. O. A. Alimi, K. Ouahada, A. M. Abu-Mahfouz and S. Rimer, "A Survey on the Security of Low Power Wide Area Networks: Threats, Challenges, and Potential Solutions," sensors and MDP1, pp. 1-9, 2020.

[93] B.Sophia, L. Jeril, M. K. Harnesh and V. L. Kumar, "A Secure Remote Clinical Sensor Network Approach for Privacy Enhancement," IOP Conference Series: Materials Science and Engineering, pp. 1-8, 2021.

[94] VMware SD-WAN, *VMware SD-WAN Edge platform specifications*, Velocloud, pp. 1-14, 2020.

[95] J. Ahamed, M.D. Zahid and K. Ahmad, "AES and MQTT based security system in the internet of Things," Journal of Discrete Mathematical Sciences and Cryptography, pp. 1589-1598, 2020.

[96] M. Khurana and M. Kumar, "Variants of Differential and Linear Cryptanalysis," International Journal of Computer Applications, pp. 20-29, 2015.

[97] A. Bar-On, O. Dunkelman, N. Keller, and A. Weizman, "DLCT: A New Tool for Differential-Linear Cryptanalysis," Lecture Notes in Computer Science, pp. 313–342, 2019.

[98] A. Canteaut, L. Kölsch and F. Wiemer, "Observations on the DLCT and Absolute Indicators," ICAR, pp. 1-18, 2019.

[99] M. A. Guptha, "Internet of Things and its Applications," Malla Reddy College of Engineering and Technology, pp. 1-146, 2021.

[100] H. M. Heys, "A Tutorial on Linear and Differential Cryptanalysis," IOActive, pp. 1-33, 2015.

[101] A. Kak, "Lecture 8: AES: The Advanced Encryption Standard," Purdue University, pp. 1-94, 2022.

[102] A. Biryukov and D. Khovratovich, "Related-key Cryptanalysis of the Full AES-192 and AES-256," ICAR, pp. 1-19, 2022.

[103] P. Junod, "Linear Cryptanalysis of DES," ETH, pp. 1-35, 2022.

[104] C. Blondeau, "Impossible Differential Attack on 13-round Camellia-192," Information Processing Letters, vol. 115, no.9, pp. 660-666, 2015.

[105] M. Tunstall, "Practical complexity Differential Cryptanalysis and fault analysis of AES," Journal of Cryptographic Engineering, pp. 219 – 230, 2011.

[106] H. Tran-Dang, "Towards the Internet of Things for Physical Internet: Perspectives and Challenges," IEEE, pp.1-26, 2020.

[107] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia and M. Dohler, "Standardized Protocol Stack for the Internet of (Important) Things" IEEE Communications Surveys and Tutorials, Vol. 15, No. 3, pp. 1-18, 2013.

[108] B. A. R. Azamuddin, "Rotation Project Title: Survey on IoT Security" Washington University in St. Louis, Vol. 4, No. 4, pp. 1 - 26, 2023.

[109] A. A. Ali, "Constrained Application Protocol (CoAP) for the IoT", IoT Seminor, High Integrity System, Frankfurt University of Applied Sciences, pp. 1-30, 2018.

[110] P. R. Egl, "MQTT - Message Queueing Telemetry Transport Introduction to MQTT, a protocol for M2M and IoT applications", Peteregli.net, pp. 1-33, 2017.

[111] M. Laine, “RESTful Web Services for the Internet of Things”, Department of Media Technology Aalto University School of Science, pp. 1-23, 2022.

[112] D. Jeevani and M.Balajee, “Effective Device Management for Internet of Things”, International Journal of Engineering and Information Systems (IJEAIS), Vol. 1 No. 8, pp. 172-181, 2017.

[113] P. Duffy, “Beyond MQTT: A Cisco View on IoT Protocols”, Cisco Blogs, 2013.

[114] M. R. Abdmeziem, D. Tandjaoui and I. Romdhani, “Architecting the Internet of Things: State of the Art”, pp. 1-33, 2015.

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

[116] M. Burhan, R. A. Rehman, B. Khan and B. Kim, “IoT Elements, Layered Architectures and Security Issues: A Comprehensive Survey”, Department of Computer Science, National University of Computer and Emerging Sciences, pp. 1-19, 2018.

[117] V. Ram, ”The OSI Reference Model,” Tutorialspoint, pp. 1-5, 2020.

[118] O. Bello and S. Zeadally, “Intelligent Device-to-Device Communication in the Internet of Things” IEEE Systems Journal, Vol. 10, No. 3, pp. 1-11, 2014.

[119] O. Horyachyy, “Comparison of Wireless Communication Technologies used in a Smart Home: Analysis of Wireless Sensor Node Based on Arduino in Home Automation Scenario”, Faculty of Computing Blekinge Institute of Technology, pp. 1-67, 2017.

[120] A. Noureen, U. Shoaib and M. S. Sarfraz, "Secure Device Pairing Methods: An Overview", (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 8, No. 9, 2017.

[121] D. Paraskevopoulos, "5 Things to Know about IoT Protocols," IoT Analytics, pp. 1-6, 2022.

[122] Cloud Standards Customer Council, "Cloud Customer Architecture for IoT" Journal of Systems Integrations, pp. 1-23, 2022.

[123] J. Fernandez, I. Vidal and F. Valera, "Enabling the Orchestration of IoT Slices through Edge and Cloud Micro service Platforms", Sensors (Basel), Vol. 19, No. 13, 2019.

[124] A. Kaushik, "IOT-An Overview", Tylor and Francis Group, An Informer Business, pp. 1-23, 2022.

[125] J. Treadway, "Using an IoT Gateway to Connect the 'Things' to the Cloud", IoT Agenda, 2018.

[126] R. Smith, "Understanding Encryption and Cryptography Basics", 2015.

[127] M. Katagi and S. Moriai, "The 128-Bit Block Cipher Clefia", Sony Corporation, Vol. 6114, pp. 1-19, 2011.

[128] J. Kaur and E. M. Kaur, "Data Encryption Using Different Techniques: A Review", International Journal of Advanced Research in Computer Science, Vol. 8, No. 4, pp. 252-255, 2017.

[129] M. Fischlin, "Public-Key Encryption (Asymmetric Encryption)", Summer School, Romania 2014.

[130] S. Channalli and A. Jadhav, "Steganography an Art of Hiding Data", International Journal on Computer Science and Engineering, Vol. 1, pp. 137-14, 2023.

[131] N. G. McDonald, “Past, Present, and Future Methods of Cryptography and Data Encryption”, Department of Electrical and Computer Engineering University of Utah, pp. 1-21, 2015.

[132] A. Toumazis, “Steganography”, 2009.

[133] A. A. Bruen and M. A. Forcinito, “Cryptography, Information Theory and Error-Correction: A Handbook for the 21st Century”, John Wiley and Sons, 2011, pp. 21-27, 2023.

[134] K. M. Martin, “Everyday Cryptography”, Oxford University Press, 2012, pp. 142-149, 2023.

[135] L. D. Smith, ”Substitution Ciphers. Cryptography the Science of Secret Writing: The Science of Secret Writing”, Dover Publications, 1943, pp. 81-90.

[136] M. Behrens, “Understanding the 3 Main Types of Encryption”, 2014.

[137] E. Biham and N. Keller. ”Cryptanalysis of Reduced Variants of Rijndael,” The 3rd AES Candidate Conference. 2000.

[138] J.H. Cheon, , M. Kim, K. Kim, J. Y. Lee and S. Kang, ”Improved Impossible Differential Cryptanalysis of Rijndael and Crypton”, Information Security and Cryptology - ICISC 2001: 4th International Conference, pp. 39-49, 2002.

[139] C. Raphael and W. Phan, ”Impossible Differential Cryptanalysis of 7-round Advanced Encryption Standard (AES),” Sciencedirect, pp. 33-38, 2004.

[140] M. R. Z’aba and M. A. Maarof, ”A Survey on the Cryptanalysis of the Advanced Encryption Standard,” Core, pp. 97-102, 2006.

[141] L. Lacko-Barto˘sova, "Linear and Differential Cryptanalysis of Reduced-Round AES," Slovenska Akademia Vied, pp. 51-61, 2011.

[142] G. Jakimoski and Y. Desmedt, "Related-Key Differential Cryptanalysis of 192-bit Key AES Variants," Research Gate, pp. 208-221, 2003.

[143] Z. Hu and Z. He, "A New Method for Impossible Differential Cryptanalysis of 7-Round AES-192," 2011 2nd International Symposium on Intelligence Information Processing and Trusted Computing, pp. 1-12, 2011.

[144] L. Grassi, "Mixture Differential Cryptanalysis and Structural Truncated Differential Attacks on round-reduced AES," Graz University of Technology, pp. 1-66, 2017.

[145] S. Simmons, "Algebraic Cryptanalysis of Simplified AES," Citeseerx, pp, 1-9, 2019.

[146] A. D. A. Gemellia, "Differential Attack on Mini-AES" AIP Conference Proceedings, pp. 1-10, 2012.

[147] R. Ankele, S. Banik, A. Chakraborti, E. List, "Related-Key Impossible-Differential Attack on Reduced-Round Skinny," Applied Cryptography and Network Security, pp. 1-11, 2017.

[148] K. Amrita, N. Gupta and R. Mishra, "An Overview of Cryptanalysis on AES," International Journal of Advance Research in Science and Engineering (IJARSE), pp. 368-649, 2018.

[149] V. Rijmen "10 years of Rijndael," Research Group Cosic and Ku Leuven, pp. 1-70, 2021.

[150] K. B. Jithendra and T. K. Shahana, "New Results in Related Key Impossible Differential Cryptanalysis on Reduced Round AES-192," The Institute of Electrical and Electronics Engineers (IEEE), pp. 1-28, 2018.

[151] L Rouquette and C. Solnon, "Abstract XOR: A Global Constraint Dedicated to Differential Cryptanalysis," Archive HAL, pp. 566–584, 2020.

[152] R. Anderson, E. Biham and L. Knudsen, "Serpent and Smartcards," Cambridge University, pp. 1-8, 2021.

[153] K. J. Compton, B. Timm and J. Van Laven, "A Simple Power Analysis Attack on the Serpent Key Schedule," IACR, pp. 1-10, 2009.

[154] R. Anderson, E. Biham and L. Knudsen, "The Case for Serpent," Case Study, pp. 1-5, 2012.

[155] E. Biham, O. Dunkelman and N. Keller, "Linear Cryptanalysis of Reduced Round Serpent," Fast Software Encryption, pp. 1-12, 2001.

[156] H. Yap, K. Khoo and A. Poschmann, "Parallelizing the Camellia and SMS4 Block Ciphers- Extended Version," International Journal of Applied Cryptography, vol. 3, no. 1, pp. 1-20, 2013.

[157] S. Lee, S. Hong, S. Lee, J. Lim, and S. Yoon, "Truncated Differential Cryptanalysis of Camellia," Information Security and Cryptology, pp. 1-15, 2001.

[158] A. Biryukov, "Boomerang Attack," Encyclopedia of Cryptography and Security, pp. 1-59, 2011.

[159] O. Dunkelman, N. Keller, E. Ronen and A. Shamir, "The Retracing Boomerang

[160] D. Bai and L. Li, "New Impossible Differential Attacks on Camellia," ICAR, PP. 1-15, 2011.

[161] W. Wu, W. Zhang and D. Feng, "Improved Impossible Differential Cryptanalysis of Reduced-Round Camellia," International Workshop on Selected Areas in Cryptography, pp. 442–456, 2008.

[162] J. Lu, Y. Wei, E. Pasalic and P. A. Fouque, "Meet-in-the-Middle Attack on Reduced Versions of the Camellia Block Cipher," DIENS, pp. 1-18, 2011.

[163] Z. Liu, B. Sun, Q. Wang, K. Varici, D. Gu, "Improved Zero-Correlation Linear Cryptanalysis of Reduced-Round Camellia under Weak Keys," IET Information Security, pp. 1-9, 2015.

[164] A. Kak, "AES: The Advanced Encryption Standard," Engineering Purdue, pp. 1-92, 2021.

[165] O. I. Abiodun, E. O. Abiodun, M. Alawida, R. S. Alkhawaldeh and H. Arshad , "A Review on the Security of the Internet of Things: Challenges and Solutions," Springer, pp. 2603–2637, 2021.

[166] R. Verma and A. K. Sharma, "Cryptography: Avalanche effect of AES and RSA," International Journal of Scientific and Research Publications, pp. 1-7, 2020.

[167] N. A. M. Ariffin and A. Y. A. Ashawesh, "Enhanced AES Algorithm Based on 14 Rounds in Securing Data and Minimizing Processing Time," The Electrochemical Society, pp. 1-9, 2021.

[168] E. Biham and A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems," Journal of Cryptology, pp. 3-72, 1991.

[169] A. Subandi, M. S. Lydia1 and R. W. Sembiring, "Analysis of RC6-Lite Implementation for Data Encryption," Scitepress, pp. 42-47,2021.

[170] S. D. Sanap and V. More, "Performance Analysis of Encryption Techniques Based on Avalanche effect and Strict Avalanche Criterion," 2021 3rd International Conference on Signal Processing and Communication (ICPSC), pp. 676-679, 2021.

[171] F. Wiemer, "Security Arguments and Tool-based Design of Block Ciphers," Faculty of Mathematics at Ruhr-Universität Bochum, ICAR, pp. 1-188, 2019.

[172] C. R. Martin, "Chapter 17: Smells and Heuristics - G25 Replace Magic Numbers with Named Constants," pp. 300, 2020.

[173] J. Maguire, "Bjarne Stroustrup on Educating Software Developers". Datamation, 2018.

[174] Blog, IoT, Technology, "Why is the Internet of Things Important to our Everyday Lives?," pp 1-8, 2019.

[175] OECD Digital Economy Policy Papers, The Internet of Things Seizing the Benefits and Addressing the Challanges," pp. 1-57, 2016

[176] J. H. Ziegeldorf1, O. G. Morchon and K. Wehrle, "Privacy in the Internet of Things: Threats and Challenges," Communication and Distributed Systems, pp 1-14, 2021.

[177] B. Najafi, B. Sadeghian, M. S. Zamani, A. Valizadeh, "High Speed Implementation of Serpent Algorithm," Academia, pp. 718-721.

[178] O. Dunkelman, S. Indesteege and N. Keller, "A Differential-Linear Attack on 12-Round Serpent," International Conference on Cryptology in India, pp. 308-321, 2008.

[179] A. A. Laghari, K. Wu, R. A. Laghari, M. Ali and A. A. Khan, "A Review and State of Art of Internet of Things (IoT)," Archives of Computational Methods in Engineering, pp. 1-20, 2021.

[180] M. Hamza, "A Beginner's Guide to The Internet of Things (IoT)," 2022, Disruptive Technologies, pp. 1-15, 2022.

[181] M. Schöffel , F. Lauer , C. C. Rheinländer and N. Wehn, "Secure IoT in the Era of Quantum Computers—Where Are the Bottlenecks?," MDPI Sensors, pp. 1-21, 2022.

[182] D. Moody, "The Beginning of the End: The Fist NIST PQC Standard," Post-Quantum Cryptography Team, pp. 1-31, 2022.

[183] L. Sleem, "Design and Implementation of Lightweight and Secure Cryptographic Algorithms for Embedded Devices," HAL Archives-Ouvertes, pp. 1-111, 2021.

[184] Technical Guideline, "Cryptographic Mechanisms: Recommendations and Key Lengths," Federal Office for Information Security, pp. 1-90, 2022.

[185] I. Whutahaean, A. A. Lestari, and B. H. Susanti, "A Tutorial of Boomerang Attack on Small Present," Journal of Physics: Conference Series, vol. 1836, no. 012029, pp. 1-8, 2021.

[186] R. H. Makarim "Relating Undisturbed Bits to Other Properties of Substitution Boxes," Middle East Technical University, pp. 1-37, 2014.

[187] M. Khurana and M. Kumari, "Variants of Differential and Linear Cryptanalysis," IACR, pp. 1-10, 2015.

[188] R. AlTawy and A. M. Youssef, "Differential Sieving for 2-Step Matching Meet-in-the-Middle Attack with Application to Lblock," Concordia University, LNCS 8898, pp. 126–139, 2015.

[189] G. Bansod, "A New Ultra Lightweight Encryption Design for Security at Node Level," International Journal of Security and Its Applications, vol. 10, no. 12, pp.111-128, 2016.

[190] B. Rashidi, "Flexible and High-Throughput Structures of Camellia Block cipher for Security of the Internet of Things," IET Computers and Digital Techniques, pp. 1-14, 2020.

[191] U. Panahi and C. Bayılmış, "Enabling Secure Data Transmission for Wireless Sensor Networks based IoT Applications," Ain Shams Engineering Journal, pp. 1-4, 2022.

[192] B. S. Sawiris, S. A. El-Gaber and M. A. Abdel-Fattah, "Centralization of Big Data Using Distributed Computing Approach in IoT," International Journal of Intelligent Engineering and Systems, vol. 14, no.4, pp. 393-409, 2021.

[193] A. M. Alkhiari, S. Mishra, and M. AlShehri, "Blockchain-Based SQKD and IDS in Edge Enabled Smart Grid Network," Computers, Materials and Continua, vol. 70, no.2. pp. 2150-2169, 2022.

[194] L. C. N Chew and E. S. Ismail, "S-Box Construction Based on Linear Fractional Transformation and Permutation Function," Symmetry vol. 12, no.5, 2020.

[195] Z. Liu, B. Sun, Q. Wang, K. Varici, D. Gu, "Improved Zero-Correlation Linear Cryptanalysis of Reduced-Round Camellia under Weak Keys," IET Information Security, pp. 1-9, 2015.

[196] R. Bhatnagar, "Internet of Things (IoT) The Rise of the Connected World," Deloitte, pp. 1-34, 2022.

[197] H. Yap, K. Khoo and A. Poschmann, "Parallelizing the Camellia and SMS4 Block Ciphers- Extended Version," International Journal of Applied Cryptography, vol. 3, no. 1, pp. 1-20, 2013.

[198] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms– Design and Analysis," LNCS, pp. 39–56, 2012

[199] J. Lu, Y. Wei, E. Pasalic and P. A. Fouque, "Meet-in-the-Middle Attack on Reduced Versions of the Camellia Block Cipher," DIENS, pp. 1-18, 2011.

[200] D. Wagner, "The Boomerang Attack," International Workshop on Fast Software Encryption, pp 156–170, 1999.

[201] M. Kanda and T. Matsumoto "Security of Camellia against Truncated Differential Cryptanalysis," FSE 2001, LNCS 2355, pp. 286–299, 2002.

[202] L. Li, K. Jia, X. Wang and X Dong, "Meet-in-the-Middle Technique for Truncated Differential and Its Applications to Clefia and Camellia," IACR, FSE, pp. 1-19, 2015.

[203] K. Zhang and X. Lai, "Another Perspective on Automatic Construction of Integral Distinguishers for ARX Ciphers," MDPI Symmetry, vol. 14, no. 3, pp. 1-45, 2022.

[204] G. C. Kessler, "An Overview of Cryptography," Garykessler.net, pp. 1-1895, 2022.

[205] C. W. Ci, S. Z. M. Naziri, R. C. Ismail, R. Hussin, M. N. M. Isa and M. S. S. M. Basir, "Crypto-Core Design using Camellia Cipher," 5th International Conference on Electronic Design (ICED) 2020, pp. 1-13, 2020.

[206] Y. Li, H. Lin, M. Liang and Y. Sun, "A New Quantum Cryptanalysis Method on Block Cipher Camellia," IET Information Security, pp. 1-25, 2021.

[207] N. Bagheri, S. Sadeghi, P. Ravi, S. Bhasin and H. Soleimany, "SIPFA: Statistical Ineffective Persistent Faults Analysis on Feistel Ciphers," IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 0, no. 0, pp. 1–24, 2022.

[208] A. K. Sharma, S.K. Mittal and S. Mittal, "Attacks on Cryptographic Hash Functions and Advances," International Journal of Information and Computing Science," pp. 89-96, 2018.

[209] "DES is not Secure," Found in https://www.freeswan.org/freeswan_trees/freeswan-1.5/doc/DES.html

[210] A. Bogdanov and P. S. Vejre, "Linear Cryptanalysis of DES with Asymmetries," Asiacrypt, pp. 187–216, 2017.

[211] Lecture Notes of Cryptography and Network Security from Cleveland State University, "Chapter 6: Data Encryption Standard (DES)," YUC, pp. 143-174, 2019.

[212] L. Li, J. Liu, Y. Guo, B. Liu, "A New S-Box Construction Method Meeting Strict Avalanche Criterion," Journal of Information Security and Applications, vol. 66, pp. 103-135, 2022.

[213] D. Mukhopadhyay, "Linear Cryptanalysis," Indian Institute of Technology Kharagpur, pp. 1-16, 2020.

[214] J. B. Awotunde1, A. O. Ameen, I. D. Oladipo and A. R. Tomori, M. Abdulraheem"Evaluation of Four Encryption Algorithms for Viability, Reliability and Performance Estimation," Nigerian Journal of Technological Developmement, Vol. 13, No. 2, 2016.

[215] K. D. Muthavhine and M. Sumbwanyambe, "An analysis and a Comparative Study of Cryptographic Algorithms used on the Internet of Things (IoT) based on Avalanche Effect," Uir.Unisa, pp. 1-184, 2018.

[216] A. Bar-On, O. Dunkelman, N. Keller and A. Weizman, "DLCT: A New Tool for Differential-Linear Cryptanalysis," IACR, pp. 1-30, 2019.

[217] A. Biryukov, "Boomerang Attack," Encyclopedia of Cryptography and Security, pp. 1-59, 2011.

[218] O. Dunkelman, N. Keller, E. Ronen and A. Shamir, "The Retracing Boomerang Attack," ICAR, pp. 1-45, 2019.

[219] D. Burak, P. BŁaszyŃski, "Parallelization of the Camellia Encryption Algorithm," PAK, vol. 55, no.10, 2009.

[220] T. Shirai, "Differential, Linear, Boomerang and Rectangle Cryptanalysis of Reduced Round Camellia," In Proceedings of the Third NESSIE Workshop, 2002.

[221] M. Matsui and J. Nakajima, "A Description of the Camellia Encryption Algorithm," Network Working Group, pp. 1-11, 2004.

[222] A. Biryukov and I. Nikoli, "Security Analysis of the Block Cipher Camellia," Cryptrec, pp. 1-25, 2012.

[223] S. Lee, S. Hong, S. Lee, J. Lim, and S. Yoon, "Truncated Differential Cryptanalysis of Camellia," Information Security and Cryptology, pp. 1-15, 2001.

[224] S. Moriai and A. Kato, "Use of the Camellia Encryption Algorithm in Cryptographic Message Syntax," Network Working Group, pp. 1-13, 2004.

[225] D. Bai and L. Li, "New Impossible Differential Attacks on Camellia," ICAR, pp. 1-15, 2011.

[226] W. Wu, W. Zhang and D. Feng, "Impossible Differential Cryptanalysis of ARIA and Camellia," ICAR, pp. 1-15, 2006.

[227] B. S. W. Poetro, "Implementation of 128 bits Camellia Algorithm for Cryptography in Digital Image," IOP Conference Series Materials Science and Engineering, pp. 1-5, 2017.

[228] W. Wu, W. Zhang and D. Feng, "Improved Impossible Differential Cryptanalysis of Reduced-Round Camellia," International Workshop on Selected Areas in Cryptography, pp. 442–456, 2008.

[229] H. Mala, M. Dakhilalian, and M. Shakiba, "Impossible Differential Cryptanalysis of Reduced–round Camellia–256," IET Information Security, vol. 5, no. 3, pp. 129–134, 2011.

[230] Y. Liu, L. Li, D. Gu, X. Wang, Z. Liu, J. Chen and W. Li, "New Observations on Impossible Differential Cryptanalysis of Reduced-Round Camellia," HAL Open Science, pp 90-109, 2012.

[231] Z. Čiča, "Pipelined implementation of Camellia encryption algorithm," 2016 24th Telecommunications Forum, pp. 1-4, 2016.

[232] D. Mukhopadhyay, "Linear Cryptanalysis," IIT Kharagpur, pp. 1-17, 2023.

[233] M. Nasiri, "Cryptanalytic Attacks on DES Block Cipher," 3rd National Industrial Mathematics Conference, pp. 1-10, 2016.

[234] M. E. K. Al-Shammary and S. S. M. Al-Dabbagh, "Differential Distribution Table Implementation DDT: Survey," Technium vol. 4, no.10, pp.15-30, 2022.

[235] A. A. Tamimi, "Performance Analysis of Data Encryption Algorithms," Wustl.edu, pp. 1-14, 2013.

[236] S. Gautam, S. S. Gaur and, H. S. Kalsi, "A Comparative Study and Analysis of Cryptographic Algorithms: RSA, Des, Aes, Blowfish, 3-DES, and Twofish," IJRECE , No. 1, Vol. 7, pp. 996-999, 2019.

Appendices

Appendix A

```
word32 FindSubtitutionBoxValue(byte tab)
{word32 fTable[256] = {
0x5b57a360, 0x7374783c, 0x13ee8f72, 0x4c80f900, 0x82c1d7ef, 0x31252edf, 0x81d46d49,
0x80e702a3, 0x62c2f890, 0x1f0f8f3a, 0x197f0f56, 0x47672e6f, 0x56b48e22, 0x61d58dea,
0x60e82344, 0x83388d42, 0x7adfcd6c, 0x24a00f1e, 0x3376b97e, 0x4fbfee45, 0x6f481851,
0x3300042b, 0x6d6d4305, 0x2516c471, 0x35c8441d, 0x54d9b8d6, 0x7461e2e2, 0x1dab6f41,
0x33ed6ed1, 0x29b9d9af, 0x2f4a5993, 0x224e847f, 0x146544c5, 0x772a22d4, 0x10af9a2d,
0x59f38367, 0x1a6c79fc, 0x7f82e2aa, 0x3c462ea7, 0x3ffbd93f, 0x181aef5d, 0x649dcddc,
0x48549915, 0x6a2e4dc0, 0x7bcd3812, 0x83af4295, 0x2c0b644e, 0x363ef970, 0x55506e29,
0x4cf7ae53, 0x415ff938, 0x6601edd5, 0x7ff997fd, 0x3daa4ea0, 0x572b4375, 0x1b59e4a2,
0x494203bb, 0x60716df1, 0x79f262c6, 0x815db7f6, 0x824b229c, 0x19f5c4a9, 0x40e943e5,
0x5ae0ee0d, 0x2a308f02, 0x64271889, 0x48cb4e68, 0x16b6cf64, 0x6940e31a, 0x53ec4e30,
0x4b938e5a, 0x1c474f48, 0x7e1ec2b1, 0x6339ade3, 0x1bd099f5, 0x7c43ed65, 0x703582f7,
0x52fee38a, 0x258d79c4, 0x6514832f, 0x372c6416, 0x658b3882, 0x74d89835, 0x4d6e63a6,
0x2cf8cef4, 0x2de6399a, 0x68537874, 0x77a0d827, 0x397deeb5, 0x3890840f, 0x4b1cd907,
0x3bcf7954, 0x3f0e6e99, 0x14dbfa18, 0x73eb2d8f, 0x52882e37, 0x2855b9b6, 0x72870d96,
0x68ca2dc7, 0x5d3278ac, 0x27684f10, 0x128a6f79, 0x3e97b946, 0x40728e92, 0x2fc10ee6,
0xfc22f87, 0x54630383, 0x37a31969, 0x424d63de, 0x563dd8cf, 0x57a1f8c8, 0x4679c3c9,
0x55c7237c, 0x849cad3b, 0x624c433d, 0x69b7986d, 0x615ed897, 0x858a17e1, 0x754f4d88,
0x1e98d9e7, 0x78178d7a, 0x7cbaa2b8, 0x72fdc2e9, 0x797bad73, 0x763cb82e, 0x30ae798c,
0x32129985, 0x1d34b9ee, 0x2e5ceeed, 0x7da80d5e, 0x590618c1, 0x42c41931, 0x319be432,
0x2b94aefb, 0x6ed162fe, 0x2ed3a440, 0x21d7cf2c, 0x7122ed9d, 0x130124cc, 0x72105843,
0x76b36d81, 0x6c09230c, 0x3f8523ec, 0x1ae32f4f, 0x4f4938f2, 0x6de3f858, 0x26042f17,
0x1e222494, 0x51240e3e, 0x6b926db9, 0x521178e4, 0x5e9698a5, 0x6c7fd85f, 0x458c5923,
0x75c602db, 0x6cf68db2, 0x3ae20eae, 0x3a6b595b, 0x267ae46a, 0x5bce58b3, 0x43b183d7,
0x4ed2839f, 0x3cbce3fa, 0x242959cb, 0x597cce14, 0x15c964be, 0x23b2a478, 0x4e5bce4c,
0x34dad977, 0x4c0a43ad, 0x36b5aec3, 0x3e2103f3, 0x216119d9, 0x49b8b90e, 0x11264f80,
0x2d6f8447, 0x27df0463, 0x32894ed8, 0x1cbe049b, 0x4515a3d0, 0x70ac384a, 0x6aa50313,
0x8513628e, 0x2b1df9a8, 0x6e5aadab, 0x8425f7e8, 0x4aa623b4, 0x5f0d4df8, 0x7a691819,
0x2c8219a1, 0x39f4a408, 0x5c450e06, 0x5ffab89e, 0x119d04d3, 0x1ffcf9e0, 0x5818ae1b,
0x19085a03, 0x35518eca, 0x1f86448d, 0x7f0c2d57, 0x22c539d2, 0x47dde3c2, 0x4428392a,
0x4a2f6e61, 0x8600cd34, 0x50ad58eb, 0x16401a11, 0x788e42cd, 0x1552af6b, 0x7e957804,
0x66ef587b, 0x449eee7d, 0x46030e76, 0x433ace84, 0x2943245c, 0x5a6a38ba, 0x7d31580b,
0x7199a2f0, 0x519ac391, 0x3d33994d, 0x5036a398, 0x4de518f9, 0x172d84b7, 0x7b5682bf,
0x1377da1f, 0x46f0791c, 0x17a43a0a, 0x67dcc321, 0x537598dd, 0x34642424, 0x2073af33,
0x6b1bb866, 0x26f199bd, 0x28cc6f09, 0x588f636e, 0x6fbecda4, 0x80704d50, 0x5cbbc359,
0x6678a328, 0x5f84034b, 0x3b58c401, 0x41d6ae8b, 0x5e1fe352, 0x1891a4b0, 0x3819cebc,
0x20ea6486, 0x63b06336, 0x7904f820, 0x67660dce, 0x5da92dff, 0x2aa74455, 0x1038e4da,
0x1213ba26, 0x39073962, 0x233bef25, 0x3037c439
};
return fTable[tab];
}
```

Figure A.1: AES S-Box with 32-Bit Output

```
    uint64_t S_Box[256] =   {

0x643daa4ea0, 0x7d494203bb, 0x7846f0791c, 0x7c48cb4e68, 0xf37ff997fd,
0x6c415ff938, 0x70433ace84,0xc66b1bb866, 0x3126042f17, 0x21038e4da,
0x683f8523ec, 0x2c23b2a478, 0xff858a17e1, 0xd87374783c, 0xac5f0d4df8,
0x774679c3c9, 0xcb6d6d4305, 0x834c0a43ad, 0xca6cf68db2, 0x7e49b8b90e,
0xfb83af4295, 0x5a39073962, 0x4830ae798c, 0xf17f0c2d57, 0xae5ffab89e,
0xd572105843, 0xa35ae0ee0d, 0xb060e82344, 0x9d5818ae1b, 0xa55bce58b3,
0x73449eee7d, 0xc168ca2dc7, 0xb8649dcddc, 0xfe8513628e, 0x9453ec4e30,
0x27216119d9, 0x3728cc6f09, 0x402cf8cef4, 0xf8824b229c, 0xcd6e5aadab,
0x3527df0463, 0xa65c450e06, 0xe679f262c6, 0xf27f82e2aa, 0x724428392a,
0xd973eb2d8f, 0x32267ae46a, 0x16197f0f56, 0x5119d04d3, 0xc86c09230c,
0x241ffcf9e0, 0xc46a2e4dc0, 0x191ae32f4f, 0x9755506e29, 0x61213ba26,
0x9b572b4375, 0x8130124cc, 0x13181aef5d, 0x814b1cd907, 0xe3788e42cd,
0xec7cbaa2b8, 0x2821d7cf2c, 0xb3624c433d, 0x7646030e76, 0xa13ee8f72,
0x844c80f900, 0x2d242959cb, 0x1b1bd099f5, 0x1c1c474f48, 0x6f42c41931,
0x5b397deeb5, 0xa159f38367, 0x5335c8441d, 0x3c2b1df9a8, 0xd772fdc2e9,
0xb462c2f890, 0x2a22c539d2, 0xe47904f820, 0x30258d79c4, 0x854cf7ae53,
0x54363ef970, 0xd270ac384a, 0x10fc22f87, 0xee7da80d5e, 0x211e98d9e7,
0xfd849cad3b, 0xb261d58dea, 0x5c39f4a408, 0x6b40e943e5, 0xcc6de3f858,
0xbf67dcc321, 0x3a2a308f02, 0x4b32129985, 0x4d3300042b, 0x593890840f,
0xd06fbecda4, 0xd1703582f7, 0xf07e957804, 0xab5e9698a5, 0xfc8425f7e8,
0x442ed3a440, 0x4e3376b97e, 0x3427684f10, 0x864d6e63a6, 0x462fc10ee6,
0xfa83388d42, 0x310af9a2d, 0x804aa623b4, 0x5134dad977, 0x3d2b94aefb,
0xa0597cce14, 0xa95da92dff, 0x5235518eca, 0xa45b57a360, 0x412d6f8447,
0x90521178e4, 0x93537598dd, 0x9e588f636e, 0x3929b9d9af, 0xf6815db7f6,
0xbd66ef587b, 0xb764271889, 0xdb74d89835, 0x221f0f8f3a, 0x11172d84b7,
0x1008600cd34,0xf480704d50, 0xd37122ed9d, 0xce6ed162fe, 0xd1552af6b,
0x141891a4b0, 0xed7d31580b, 0x603bcf7954, 0x9855c7237c, 0x452f4a5993,
0x181a6c79fc, 0xc56aa50313, 0xa85d3278ac, 0x7f4a2f6e61, 0x3e2c0b644e,
0x653e2103f3, 0x5e3ae20eae, 0x1a1b59e4a2, 0x744515a3d0, 0x613c462ea7,
0x824b938e5a, 0x5034642424, 0xdd75c602db, 0x231f86448d, 0x2b233bef25,
0x9152882e37, 0x894ed2839f, 0x473037c439, 0xef7e1ec2b1, 0xb96514832f,
0x1519085a03, 0xdf76b36d81, 0x5f3b58c401, 0xc14dbfa18, 0xdc754f4d88,
0xe177a0d827, 0x3326f199bd, 0x3b2aa74455, 0xb146544c5, 0x4a319be432,
0x7128a6f79, 0x252073af33, 0x5d3a6b595b, 0xc369b7986d, 0xd47199a2f0,
0xad5f84034b, 0x633d33994d, 0x9252fee38a, 0x9654d9b8d6, 0xe5797bad73,
0x7a47dde3c2, 0xe87adfcd6c, 0xc96c7fd85f, 0x382943245c, 0x6e424d63de,
0x8e51240e3e, 0xd672870d96,0x4f33ed6ed1, 0xaa5e1fe352, 0x6d41d6ae8b,
0x5737a31969, 0xf580e702a3, 0xeb7c43ed65, 0x663e97b946, 0x7b48549915,
0xaf60716df1, 0x91377da1f, 0xbb6601edd5, 0x7947672e6f, 0x2620ea6486,
0x2f2516c471, 0x1d1cbe049b, 0xa75cbbc359, 0xb56339ade3, 0xc76b926db9,
0xe97b5682bf, 0xde763cb82e, 0x75458c5923, 0x201e222494, 0x4c32894ed8,
0xbe67660dce, 0x8c5036a398, 0x8b4fbfee45, 0x7143b183d7, 0x3f2c8219a1,
0xb663b06336, 0x673f0e6e99, 0x4931252edf, 0x411264f80, 0xf781d46d49,
0xf16401a11, 0x623cbce3fa, 0x362855b9b6, 0x583819cebc, 0xba658b3882,
0x874de518f9, 0xc26940e31a, 0x1e1d34b9ee, 0x9f590618c1,0xe278178d7a,
0xf982c1d7ef, 0x99563dd8cf, 0x1217a43a0a, 0x6a40728e92, 0xda7461e2e2,
0x8f519ac391, 0x9554630383, 0x9c57a1f8c8, 0x1f1dab6f41, 0x884e5bce4c,
0xea7bcd3812, 0xcf6f481851, 0x56372c6416, 0x29224e847f, 0xe0772a22d4,
0x8d50ad58eb, 0xa25a6a38ba, 0x8a4f4938f2, 0xe15c964be, 0xc068537874,
0xe77a691819, 0x432e5ceeed, 0x693ffbd93f, 0x422de6399a, 0x9a56b48e22,
0x2e24a00f1e, 0x1016b6cf64. 0xb1615ed897. 0x5536b5aec3. 0xbc6678a328.
0x1719f5c4a9 };
```

Figure A.2: New Inverse AES S-Box with 32-Bit Output

Appendix B

```
//New 4x32 bit Serpent S-boxes//
static const uint64_t S[8][16] = {
{0x411264f80, 0x91377da1f, 0x1016b6cf64,
 0x21038e4da, 0xb146544c5, 0x7128a6f79,
 0x61213ba26, 0xc14dbfa18, 0xf16401a11,
 0xe15c964be, 0x5119d04d3, 0x310af9a2d,
 0x8130124cc, 0x10fc22f87, 0xa13ee8f72,
 0xd1552af6b},
{0x1016b6cf64,
 0xd1552af6b, 0x310af9a2d, 0x8130124cc,
 0xa13ee8f72, 0x10fc22f87, 0x61213ba26,
 0xb146544c5, 0x21038e4da, 0xc14dbfa18,
 0xf16401a11, 0x91377da1f, 0x7128a6f79,
 0xe15c964be, 0x411264f80, 0x5119d04d3},
{0x91377da1f, 0x7128a6f79, 0x8130124cc,
0xa13ee8f72, 0x411264f80, 0xd1552af6b,
0xb146544c5, 0x1016b6cf64, 0xe15c964be,
0x21038e4da, 0xf16401a11, 0x5119d04d3,
0x10fc22f87, 0xc14dbfa18, 0x61213ba26,
0x310af9a2d},
{0x10fc22f87, 0x1016b6cf64, 0xc14dbfa18,
 0x91377da1f, 0xd1552af6b, 0xa13ee8f72,
 0x7128a6f79, 0x411264f80, 0xe15c964be,
 0x21038e4da, 0x310af9a2d, 0x5119d04d3,
 0xb146544c5, 0x8130124cc, 0x61213ba26,
 0xf16401a11},
{0x21038e4da, 0x1016b6cf64, 0x91377da1f,
0x411264f80, 0xd1552af6b, 0x10fc22f87,
0xc14dbfa18, 0x7128a6f79, 0x310af9a2d,
0x61213ba26, 0x5119d04d3, 0xb146544c5,
0xa13ee8f72, 0xf16401a11, 0x8130124cc,
0xe15c964be},
{0x1016b6cf64, 0x61213ba26, 0x310af9a2d,
 0xc14dbfa18, 0x5119d04d3, 0xb146544c5,
 0xa13ee8f72, 0xd1552af6b, 0x10fc22f87,
 0x411264f80, 0xf16401a11, 0x91377da1f,
 0xe15c964be, 0x7128a6f79, 0x8130124cc,
 0x21038e4da},
{0x8130124cc, 0x310af9a2d, 0xd1552af6b,
 0x61213ba26, 0x91377da1f, 0x5119d04d3,
 0x7128a6f79, 0xc14dbfa18, 0xf16401a11,
 0xa13ee8f72, 0x21038e4da, 0x1016b6cf64,
 0xe15c964be, 0x411264f80, 0xb146544c5,
 0x10fc22f87},
{0x21038e4da, 0xe15c964be, 0x1016b6cf64,
 0x10fc22f87, 0xf16401a11, 0x91377da1f,
 0x310af9a2d, 0xc14dbfa18, 0x8130124cc,
 0x5119d04d3, 0xd1552af6b, 0xb146544c5,
 0xa13ee8f72, 0x411264f80, 0x61213ba26,
 0x7128a6f79}};
```

Figure B.1: New 32-bit S-Boxes of Serpent written in C++

```
//New x32 bit Serpent inverse S-boxes//
static const uint64_t IS[8][16] = {
{0x411264f80, 0x10fc22f87, 0x7128a6f79,
 0xe15c964be, 0xa13ee8f72, 0xf16401a11,
 0x1016b6cf64, 0x91377da1f, 0x61213ba26,
 0xd1552af6b, 0xc14dbfa18, 0x8130124cc,
 0xb146544c5, 0x21038e4da, 0x5119d04d3,
 0x310af9a2d},
{0x1016b6cf64, 0xb146544c5, 0x21038e4da,
0xe15c964be, 0x61213ba26, 0x411264f80,
0x7128a6f79, 0x10fc22f87, 0x5119d04d3,
0xa13ee8f72, 0xf16401a11, 0x8130124cc,
0x310af9a2d, 0xd1552af6b, 0x91377da1f,
0xc14dbfa18},
{0x91377da1f, 0x1016b6cf64, 0x310af9a2d,
0xa13ee8f72, 0x5119d04d3, 0x21038e4da,
0xe15c964be, 0xf16401a11, 0xc14dbfa18,
0x7128a6f79, 0x61213ba26, 0x411264f80,
0x8130124cc, 0xd1552af6b, 0xb146544c5,
0x10fc22f87},
{0x61213ba26, 0x10fc22f87, 0x91377da1f,
0x411264f80, 0xb146544c5, 0xa13ee8f72,
0x8130124cc, 0xf16401a11, 0x310af9a2d,
0xd1552af6b, 0xc14dbfa18, 0x7128a6f79,
0x5119d04d3, 0x1016b6cf64, 0xe15c964be,
0x21038e4da},
{0x10fc22f87, 0xa13ee8f72, 0xb146544c5,
0x8130124cc, 0xc14dbfa18, 0xf16401a11,
0x7128a6f79, 0xe15c964be, 0x411264f80,
0x61213ba26, 0xd1552af6b, 0x310af9a2d,
0x5119d04d3, 0x91377da1f, 0x1016b6cf64,
0x21038e4da},
{0xd1552af6b, 0xa13ee8f72, 0x1016b6cf64,
0x5119d04d3, 0xc14dbfa18, 0xf16401a11,
0x21038e4da, 0x310af9a2d, 0x10fc22f87,
0x411264f80, 0x7128a6f79, 0xe15c964be,
0x61213ba26, 0x91377da1f, 0xb146544c5,
0x8130124cc},
{0x61213ba26, 0x91377da1f, 0x310af9a2d,
0xf16401a11, 0x1016b6cf64, 0x7128a6f79,
0xd1552af6b, 0x411264f80, 0xc14dbfa18,
0x5119d04d3, 0x8130124cc, 0xa13ee8f72,
0x21038e4da, 0xe15c964be, 0xb146544c5,
0x10fc22f87},
{0xe15c964be, 0x411264f80, 0xc14dbfa18,
0x10fc22f87, 0xb146544c5, 0x7128a6f79,
0x61213ba26, 0xd1552af6b, 0x21038e4da,
0xf16401a11, 0x5119d04d3, 0x8130124cc,
0x1016b6cf64, 0xa13ee8f72, 0x91377da1f,
0x310af9a2d}};
```

Figure B.2: New Inverse of 32-bit S-Boxes of Serpent written in C++

Appendix C

```
const Byte SubstitutionBox[256] = {
112,130, 44,236,179, 39,192,229,228,133, 87, 53,234, 12,174, 65,
 35,239,107,147, 69, 25,165, 33,237, 14, 79, 78, 29,101,146,189,
134,184,175,143,124,235, 31,206, 62, 48,220, 95, 94,197, 11, 26,
166,225, 57,202,213, 71, 93, 61,217,  1, 90,214, 81, 86,108, 77,
139, 13,154,102,251,204,176, 45,116, 18, 43, 32,240,177,132,153,
223, 76,203,194, 52,126,118,  5,109,183,169, 49,209, 23,  4,215,
 20, 88, 58, 97,222, 27, 17, 28, 50, 15,156, 22, 83, 24,242, 34,
254, 68,207,178,195,181,122,145, 36,  8,232,168, 96,252,105, 80,
170,208,160,125,161,137, 98,151, 84, 91, 30,149,224,255,100,210,
 16,196,  0, 72,163,247,117,219,138,  3,230,218,  9, 63,221,148,
135, 92,131,  2,205, 74,144, 51,115,103,246,243,157,127,191,226,
 82,155,216, 38,200, 55,198, 59,129,150,111, 75, 19,190, 99, 46,
233,121,167,140,159,110,188,142, 41,245,249,182, 47,253,180, 89,
120,152,  6,106,231, 70,113,186,212, 37,171, 66,136,162,141,250,
114,  7,185, 85,248,238,172, 10, 54, 73, 42,104, 60, 56,241,164,
 64, 40,211,123,187,201, 67,193, 21,227,173,244,119,199,128,158};
```

Figure C.1: Standard C++ Camellia 8 x 8 S-Box

```
Word SubstitutionBox[256] = {
0x43b183d7, 0x4c0a43ad, 0x242959cb, 0x7d31580b, 0x62c2f890, 0x21d7cf2c, 0x68ca2dc7, 0x79f262c6,
0x797bad73, 0x4d6e63a6, 0x3819cebc, 0x2855b9b6, 0x7c43ed65, 0x1552af6b, 0x60716df1, 0x2de6399a,
0x1ffcf9e0, 0x7e957804, 0x415ff938, 0x53ec4e30, 0x2fc10ee6, 0x1b59e4a2, 0x5c450e06, 0x1f0f8f3a,
0x7da80d5e, 0x16401a11, 0x34642424, 0x33ed6ed1, 0x1d34b9ee, 0x3e97b946, 0x537598dd, 0x67660dce,
0x4de518f9, 0x6514832f, 0x60e82344, 0x521178e4, 0x494203bb, 0x7cbaa2b8, 0x1e222494, 0x6f481851,
0x2c8219a1, 0x26042f17, 0x75c602db, 0x3bcf7954, 0x3b58c401, 0x6b1bb866, 0x14dbfa18, 0x1bd099f5,
0x5cbbc359, 0x78178d7a, 0x2a308f02, 0x6d6d4305, 0x72870d96, 0x30ae798c, 0x3ae20eae, 0x2c0b644e,
0x7461e2e2, 0x1038e4da, 0x397deeb5, 0x72fdc2e9, 0x35518eca, 0x37a31969, 0x41d6ae8b, 0x3376b97e,
0x5036a398, 0x15c964be, 0x572b4375, 0x3f0e6e99, 0x8425f7e8, 0x6e5aadab, 0x615ed897, 0x24a00f1e,
0x458c5923, 0x181aef5d, 0x23b2a478, 0x1e98d9e7, 0x7f0c2d57, 0x61d58dea, 0x4cf7ae53, 0x56b48e22,
0x772a22d4, 0x3300042b, 0x6de3f858, 0x69b7986d, 0x27df0463, 0x4a2f6e61, 0x4679c3c9, 0x1213ba26,
0x424d63de, 0x649dcddc, 0x5e1fe352, 0x267ae46a, 0x70ac384a, 0x1a6c79fc, 0x1119d04d3, 0x7374783c,
0x19085a03, 0x3890840f, 0x2aa74455, 0x3cbce3fa, 0x76b36d81, 0x1c474f48, 0x17a43a0a, 0x1cbe049b,
0x26f199bd, 0x16b6cf64, 0x5818ae1b, 0x19f5c4a9, 0x363ef970, 0x1ae32f4f, 0x7ff997fd, 0x1f86448d,
0x858a17e1, 0x2f4a5993, 0x6fbecda4, 0x624c433d, 0x6a2e4dc0, 0x63b06336, 0x48549915, 0x52fee38a,
0x2073af33, 0x1377da1f, 0x7b5682bf, 0x5da92dff, 0x3c462ea7, 0x849cad3b, 0x40728e92, 0x34dad977,
0x5e9698a5, 0x703582f7, 0x59f38367, 0x49b8b90e, 0x5a6a38ba, 0x4f4938f2, 0x3d33994d, 0x55c7237c,
0x36b5aec3, 0x39f4a408, 0x1dab6f41, 0x54d9b8d6, 0x77a0d827, 0x8600cd34, 0x3e2103f3, 0x7122ed9d,
0x172d84b7, 0x6aa50313, 0x0fc22f87, 0x31252edf, 0x5b57a360, 0x824b229c, 0x46030e76, 0x754f4d88,
0x4fbfee45, 0x11264f80, 0x7a691819, 0x74d89835, 0x13ee8f72, 0x2cf8cef4, 0x763cb82e, 0x54630383,
0x4e5bce4c, 0x3a6b595b, 0x4c80f900, 0x10af9a2d, 0x6ed162fe, 0x32129985, 0x52882e37, 0x27684f10,
0x4515a3d0, 0x3f8523ec, 0x81d46d49, 0x80704d50, 0x588f636e, 0x4aa623b4, 0x68537874, 0x788e42cd,
0x35c8441d, 0x57a1f8c8, 0x73eb2d8f, 0x216119d9, 0x6c7fd85f, 0x2943245c, 0x6b926db9, 0x2b1df9a8,
0x4b938e5a, 0x55506e29, 0x433ace84, 0x32894ed8, 0x1891a4b0, 0x67dcc321, 0x3daa4ea0, 0x2516c471,
0x7bcd3812, 0x47dde3c2, 0x5d3278ac, 0x50ad58eb, 0x597cce14, 0x42c41931, 0x66ef587b, 0x519ac391,
0x22c539d2, 0x815db7f6, 0x83388d42, 0x64271889, 0x258d79c4, 0x8513628e, 0x6339ade3, 0x39073962,
0x47672e6f, 0x563dd8cf, 0x128a6f79, 0x40e943e5, 0x7adfcd6c, 0x3037c439, 0x4428392a, 0x6601edd5,
0x72105843, 0x20ea6486, 0x5f0d4df8, 0x2e5ceeed, 0x4ed2839f, 0x5ae0ee0d, 0x51240e3e, 0x83af4295,
0x449eee7d, 0x130124cc, 0x658b3882, 0x372c6416, 0x82c1d7ef, 0x7e1ec2b1, 0x5f84034b, 0x146544c5,
0x28cc6f09, 0x319be432, 0x233bef25, 0x3ffbd93f, 0x2b94aefb, 0x29b9d9af, 0x7f82e2aa, 0x5bce58b3,
0x2d6f8447, 0x224e847f, 0x7199a2f0, 0x48cb4e68, 0x6678a328, 0x6cf68db2, 0x2ed3a440, 0x6940e31a,
0x197f0f56, 0x7904f820, 0x5ffab89e, 0x80e702a3, 0x46f0791c, 0x6c09230c, 0x4b1cd907, 0x590618c1};
```

Figure C.2: New Camellia 8 x 32 S-Box

```
int FindSubtitutionBoxValue(int Number)
{
    int S_Box[256] =   {
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
    0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
    0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
    0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
    0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
    0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
    0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
    0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
    0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
    0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
    0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
    0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
    0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
    0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
    0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
    0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
    0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };
    return S_Box[Number];
}
```

Figure C.3: Standard C++ AES 8 x 8 S-Box