

**AN ANALYSIS AND A COMPARATIVE STUDY OF CRYPTOGRAPHIC
ALGORITHMS USED ON THE INTERNET OF THINGS (IoT) BASED ON
AVALANCHE EFFECT.**

By

KHUMBELO DIFFERENCE MUTHAVHINE

Student number: 62037773

Submitted in accordance with the requirements

For the degree of

Magister Technologiae

In the

Department of Electrical Engineering

At the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: DR. M. SUMBWANYAMBE

July 2018

Declaration

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

Name: Mr. Khumbelo Difference Muthavhine.

Signature: 

Date: 24 January 2019

Papers published from this dissertation

K. D. Muthavhine and M. Sumbwanyambe (2018) “*An Analysis and a Comparative Study of Cryptographic Algorithms Used on the Internet of Things (IoT) Based on Avalanche Effect*”, in proceedings of ICOIACT conference, Indonesia, 6-8 March 2018. It is edited, evaluated and published by Institute of Electrical and Electronics Engineers (IEEE) on 30 April 2018.

Available: <https://ieeexplore.ieee.org/document/8350759>.

Abstract

Ubiquitous computing is already weaving itself around us and it is connecting everything to the network of networks. This interconnection of objects to the internet is new computing paradigm called the Internet of Things (IoT) networks. Many capacity and non-capacity constrained devices, such as sensors are connecting to the Internet. These devices interact with each other through the network and provide a new experience to its users. In order to make full use of this ubiquitous paradigm, security on IoT is important. There are problems with privacy concerns regarding certain algorithms that are on IoT, particularly in the area that relates to their avalanche effect. In simple terms, avalanche effect means that a small change in the plaintext or key should create a significant change in the ciphertext. The higher the significant change, the higher the security of that algorithm. If the avalanche effect of an algorithm is less than 50% then that algorithm is weak and can create security undesirability in any network. In this, case IoT.

In this study, we propose to do the following: (1) Search and select existing block cryptographic algorithms (maximum of ten) used for authentication and encryption from different devices used on IoT. (2) Analyse the avalanche effect of select cryptographic algorithms and determine if they give efficient authentication on IoT. (3) Improve their avalanche effect by designing a mathematical model that improves their robustness against attacks. This is done through the usage of the initial vector XORed with plaintext and final vector XORed with cipher text. (4) Test the new mathematical model for any enhancement on the avalanche effect of each algorithm as stated in the preceding sentences. (5) Propose future work on how to enhance security on IoT.

Results show that, when using the proposed method with variation of key, the avalanche effect significantly improved for seven out of ten algorithms. This means that we have managed to improve 70% of algorithms tested. Therefore indicating a substantial success rate for the proposed method as far as the avalanche effect is concerned. We propose that the seven algorithms be replaced by our improved versions in each of their implementation on IoT (especially in cases where the key is varied). On the other hand, when using the proposed method with variation of plaintexts, only four out of ten algorithms reflected an increase of avalanche effect. Again, the proposed method yields positive results though only 40% in this case. We recommend that these four algorithms be replaced with the proposed algorithms in IoT whenever the plaintext is varied.

Acknowledgements

I would first like to send my gratitude to my supervisor, Dr. Sumbwanyambe Mbuyu of the Department of Electrical and Mining Engineering at the University of South Africa (UNISA). The doctor's office was always open for consultations whenever I had problems or questions regarding my experiments, results or research. Finally, I must submit my gratitude to my wife, Mrs. Tinswalo Audry Mabasa Muthavhine and to my children (Frankie Muthavhine, Lulama Muthavhine, Mukoni Muthavhine and Khuliso Muthavhine) for giving me the necessary support and constant motivation throughout my research and studies. This achievement would not have been successful without them. Thank you very much.

Mr. Khumbelo Difference Muthavhine

Table of Contents:

Acknowledgements	v
List of Figures	ix
List of Tables	xii
Abbreviations and Glossary	xiii
Basic Mathematical Operators and Symbols	xv
CHAPTER 1. INTRODUCTION	1
1.1 Background of Study	1
1.1.1 <i>Definition of Internet of Things (IoT)</i>	1
1.1.2 <i>Benefits of Internet of Things (IoT)</i>	1
1.1.3 <i>The Avalanche Effect of Cryptographic Algorithms</i>	2
1.1.4 <i>Security of the Internet of Things (IoT)</i>	3
1.2 Problem Statement	6
1.3 Hypothesis	6
1.4 Research Questions	6
1.5 Research Objectives	7
1.6 Research Methodology	8
1.7 Significance of the Study	10
1.8 The Research Roadmap	10
1.9 Chapter Summary	11
CHAPTER 2: LITERATURE REVIEW	12
2.1. Introduction	12
2.2. Overview of Internet of Things	12
2.2.2. <i>IoT Architecture</i>	14
2.2.3. <i>Device-to-Device Communication model</i>	15
2.2.4. <i>Device-to-Cloud Communication</i>	16
2.2.5. <i>Device-to-Gateway Communication</i>	16
2.2.6. <i>Back-End Data Sharing communication</i>	17
2.2.7. <i>Security and the Internet of Things</i>	18
2.3 Encryption Methods and Techniques	20
2.3.1 <i>Symmetric Encryption</i>	20
2.3.2 <i>Asymmetric Encryption</i>	21
2.3.3 <i>Steganography</i>	21
2.3.4 <i>Vigenère Encryption</i>	21
2.3.5 <i>Hashing (Authenticated) Encryption</i>	21

2.4 Algorithms Used In the Security of IoT	22
2.4.1 The Advanced Encryption Standard (AES) Algorithm.....	22
2.4.2 Blowfish Algorithm	23
2.4.3 Camellia Algorithm.....	23
2.4.4 CAST-128 Algorithm.....	23
2.4.5 Clefia Algorithm.....	23
2.4.6 Data Encryption Standard (DES) Algorithm	24
2.4.7 Modular Multiplication based Block Cipher (MMB) Algorithm	24
2.4.8 Rivest Cipher 5 (RC-5)-32/32/16 Algorithm	24
2.4.9 Serpent Algorithm	25
2.4.10 Skipjack Algorithm.....	25
2.5 Avalanche Effect and Security in the Internet of Things	25
2.6 Types of Attacks on Internet of Things	26
2.6.1 Denial of Service (DoS) Attack	26
2.6.2 Man in the Middle (MITM) Attack.....	27
2.6.3 Eavesdropping Attack	27
2.6.4 Honeypot Attack.....	27
2.6.5 Collision (Bit Error) Attack	28
2.6.6 Differential Cryptanalysis Attack.....	28
2.6.7 Differential Fault Attack	28
2.7 Related Work	28
2.8 Chapter Summary	33
CHAPTER 3: METHODOLOGY	34
3.1. Introduction	34
3.2 Source Of Initial And Final Vector	36
3.3. PI Methodology: The Overview	36
3.4 Methodology of Study Based On the Avalanche Effect	37
3.4.1 Need to Calculate the Avalanche Effect.....	37
3.4.2. Method to Calculate the Avalanche Effect.....	37
3.5. Research Design	39
3.6. Experimental Procedure	40
3.6.1. Simulation 1: Testing of Avalanche Effect on AES	42
3.6.2. Simulation 2: Testing of Avalanche Effect on Blowfish	45
3.6.3. Simulation 3: Testing of Avalanche Effect on Camellia	48
3.6.4. Simulation 4: Testing of Avalanche Effect on Cast-128	51

3.6.5. Simulation 5: Testing of Avalanche Effect on Clefia	54
3.6.6. Simulation 6: Testing of Avalanche Effect on DES.....	57
3.6.7. Simulation 7: Testing of Avalanche Effect on MMB.....	60
3.6.8. Simulation 8: Testing of Avalanche Effect on RC5	63
3.6.9. Simulation 9: Testing of Avalanche Effect on Serpent.....	66
3.6.10. Simulation 10: Testing of Avalanche Effect on Skipjack	69
3.4. Chapter Summary	72
CHAPTER 4: RESULTS DISCUSSION AND ANALYSIS	73
4.1 Introduction.....	73
4.2.1. Results 1: The Avalanche Effect on AES.....	74
4.2.2. Results 2: The Avalanche Effect on Blowfish.....	78
4.2.3 Results 3: The Avalanche Effect on Camellia	82
4.2.4. Results 4: The Avalanche Effect on Cast-128.	86
4.2.5. Results 5 5: The Avalanche Effect on Clefia.....	90
4.2.6. Results 6: The Avalanche Effect on DES	94
4.2.7. Results 7: The Avalanche Effect on MMB.....	98
4.2.8. Results 8: The Avalanche Effect on RC5.....	102
4.2.9. Results 9: The Avalanche Effect on Serpent	106
4.2.10. Results 10: The Avalanche Effect on Skipjack	110
4.2.11. Results 11: The Speed and Avalanche Effect on All Ten Algorithms.....	114
4.3. Chapter Summary	124
CHAPTER 5: CONCLUSION AND FUTURE WORK	125
References.....	129
APPENDIX 1: Calculation of AES avalanche effect.	147
APPENDIX 2: The value of PI in hexadecimal after the first digit [122].	168

List of Figures

Figure 1.1: The model of well-known algorithms.	5
Figure 1.2: Model plan of proposed work, initial and final vectors are implemented.....	5
Figure 1.3: Detailed flow diagram of the proposed study with XOR block inserted.	8
Figure 2.1: IoT architecture and network layers [29]	15
Figure 2.2: Device-to-device communication model [30].....	15
Figure 2.3: Device-to-cloud communication model [31].	16
Figure 2.4: Device-to-gateway communication model diagram [30].....	17
Figure 2.5: Back-end data sharing model diagram [30]	18
Figure 2.6: IoT Security Challenges [27].	19
Figure 2.7: IoT reference model: Security [41].	20
Figure 2.8: Proposed work diagram explaining how initial and final vectors are derived.	33
Figure 3.1: Standard model with an explanation of block sizes.	35
Figure 3.2: Proposed model of algorithms with an explanation of XOR operation.	35
Figure 3.3: Flowchart diagram on how to calculate avalanche effect of algorithm like AES.	38
Figure 3.4: Proposed model with the explanations of block sizes of each building block.	39
Figure 3.5: Example of reading simulation and table	41
Figure 3.6: Simulation of avalanche effect on standard AES when plaintext is varied.....	43
Figure 3.7: Simulation of avalanche effect on proposed AES when plaintext is varied.	43
Figure 3.8: Simulation of avalanche effect on standard AES when key is varied.....	44
Figure 3.9 Simulation of avalanche effect on proposed AES when key is varied.....	44
Figure 3.10: Simulation of avalanche effect on standard Blowfish when plaintext is varied.	46
Figure 3.11: Simulation of avalanche effect on proposed Blowfish when plaintext is varied.....	46
Figure 3.12: Simulation of avalanche effect on standard Blowfish when key is varied.....	47
Figure 3.13: Simulation of avalanche effect on proposed Blowfish when key is varied.....	47
Figure 3.14: Simulation of avalanche effect on standard Camellia when plaintext is varied.....	49
Figure 3.15: Simulation of avalanche effect on proposed Camellia when plaintext is varied.....	49
Figure 3.16: Simulation of avalanche effect on standard Camellia when key is varied.....	50
Figure 3.17: Simulation of avalanche effect on proposed Camellia when key is varied.....	50
Figure 3.18: Simulation of avalanche effect on standard Cast-128 when plaintext is varied.....	52
Figure 3.19: Simulation of avalanche effect on proposed Cast-128 when plaintext is varied.....	52
Figure 3.20: Simulation of avalanche effect on standard Cast-128 when key is varied.....	53
Figure 3.21: Simulation of avalanche effect on proposed Cast-128 when key is varied.....	53
Figure 3.22: Simulation of avalanche effect on standard Clefia when plaintext is varied.	55
Figure 3.23: Simulation of avalanche effect on proposed Clefia when plaintext is varied.	55
Figure 3.24: Simulation of avalanche effect on standard Clefia when key is varied.....	56
Figure 3.25: Simulation of avalanche effect on proposed Clefia when key is varied.	56
Figure 3.26: Simulation of avalanche effect on standard DES when plaintext is varied.....	58
Figure 3.27: Simulation of avalanche effect on proposed DES when plaintext is varied.	58
Figure 3.28: Simulation of avalanche effect on standard DES when key is varied.....	59
Figure 3.29: Simulation of avalanche effect on proposed DES when key is varied.....	59
Figure 3.30: Simulation of avalanche effect on standard MMB when plaintext is varied.	61

Figure 3.31: Simulation of avalanche effect on proposed MMB when plaintext is varied.	61
Figure 3.32: Simulation of avalanche effect on standard MMB when key is varied.....	62
Figure 3.33: Simulation of avalanche effect on proposed MMB when key is varied.	62
Figure 3.34: Simulation of avalanche effect on standard RC5 when plaintext is varied.....	64
Figure 3.35: Simulation of avalanche effect on proposed RC5 when plaintext is varied.....	64
Figure 3.36: Simulation of avalanche effect on standard RC5 when key is varied.	65
Figure 3.37: Simulation of avalanche effect on proposed RC5 when key is varied.....	65
Figure 3.38: Simulation of avalanche effect on standard Serpent when plaintext is varied. ...	67
Figure 3.39: Simulation of avalanche effect on proposed Serpent when plaintext is varied...	67
Figure 3.40: Simulation of avalanche effect on standard Serpent when key is varied.	68
Figure 3.41: Simulation of avalanche effect on proposed Serpent when key is varied.	68
Figure 3.42: Simulation of avalanche effect on standard Skipjack when plaintext is varied. .	70
Figure 3.43: Simulation of avalanche effect on proposed Skipjack when plaintext is varied.	70
Figure 3.44: Simulation of avalanche effect on standard Skipjack when key is varied.	71
Figure 3.45: Simulation of avalanche effect on proposed Skipjack when key is varied.	71
Figure 4.1: Results of avalanche effect on AES when plaintext was varied.	75
Figure 4.2: Results of speed taken on AES when plaintext was varied.....	76
Figure 4.3: Results of avalanche effect on AES when key was varied.....	77
Figure 4.4: Results of speed taken on AES when key was varied.	78
Figure 4.5: Results of avalanche effect on Blowfish when plaintext was varied.	79
Figure 4.6: Results of speed taken on Blowfish when plaintext was varied.....	80
Figure 4.7: Results of avalanche effect on Blowfish when key was varied.	81
Figure 4.8: Results of speed taken on Blowfish when key was varied.....	82
Figure 4.9: Results of avalanche effect on Camellia when plaintext was varied.	83
Figure 4.10: Results of speed taken on Camellia when plaintext was varied.....	84
Figure 4.11: Results of avalanche effect on Camellia when key was varied.....	85
Figure 4.12: Results of speed taken on Camellia when key was varied.	86
Figure 4.13: Results of avalanche effect on Cast-128 when plaintext was varied.	87
Figure 4.14: Results of speed taken on Cast-128 when plaintext was varied.....	88
Figure 4.15: Results of avalanche effect on Cast-128 when key was varied.....	89
Figure 4.16: Results of speed taken on Cast-128 when key was varied.	90
Figure 4.17: Results of avalanche effect on Clefia when plaintext was varied.	91
Figure 4.18: Results of speed taken on Clefia when plaintext was varied	92
Figure 4.19: Results of avalanche effect on Clefia when key was varied.	93
Figure 4.20: Results of speed taken on Clefia when key was varied.....	94
Figure 4.21: Results of avalanche effect on DES when plaintext was varied.	95
Figure 4.22: Results of speed taken on DES when plaintext was varied.....	96
Figure 4.23: Results of avalanche effect on DES when key was varied.....	97
Figure 4.24: Results of speed taken on DES when key was varied.	98
Figure 4.25: Results of avalanche effect on MMB when plaintext was varied.	99
Figure 4.26: Results of speed taken on MMB when plaintext was varied.	100
Figure 4.27: Results of avalanche effect on MMB when key was varied.	101
Figure 4.28: Results of speed taken on MMB when key was varied.....	102
Figure 4.29: Results of avalanche effect on RC5 when plaintext was varied.....	103

Figure 4.30: Results of speed taken on RC5 when plaintext was varied.	104
Figure 4.31: Results of avalanche effect on RC5 when key was varied.	105
Figure 4.32: Results of speed taken on RC5 when key was varied.	106
Figure 4.33: Results of avalanche effect on Serpent when plaintext was varied.	107
Figure 4.34: Results of speed taken on Serpent when plaintext was varied.	108
Figure 4.35: Results of avalanche effect on Serpent when key was varied.	109
Figure 4.36: Results of speed taken on Serpent when key was varied.	110
Figure 4.37: Results of avalanche effect on Skipjack when plaintext was varied.	111
Figure 4.38: Results of speed taken on Skipjack when plaintext was varied.	112
Figure 4.39: Results of avalanche effect on Skipjack when key was varied.	113
Figure 4.40: Results of speed taken on Skipjack when key was varied.	114
Figure 4.41: Results of avalanche effect of all algorithms tested when plaintext is varied..	116
Figure 4.42: Results of Avalanche effect when plaintext was varied.	117
Figure 4.43: Result of the speeds of all algorithms tested when plaintext was varied.	119
Figure 4.44: Results of avalanche effect of all algorithms tested when key was varied.	121
Figure 4.45: Results of Avalanche effect when key was varied.	122
Figure 4.46: Results of the speed of all algorithms test when key was varied.	123

List of Tables.




Table 2.1: Summary of the protocols used on IoT [26].	14
Table 3.1: Algorithms and their usage within IoT.	40
Table 4.1: Results of standard and proposed AES when plaintext was varied.	74
Table 4.2: Results of standard and proposed AES when key was varied.	77
Table 4.3: Results of standard and proposed Blowfish when plaintext was varied.	79
Table 4.4: Results of standard and proposed Blowfish when key was varied.	81
Table 4.5: Results of standard and proposed Camellia when plaintext was varied.	83
Table 4.6: Results of standard and proposed Camellia when key was varied.	85
Table 4.7: Results of standard and proposed Cast-128 when plaintext was varied.	87
Table 4.8: Results of standard and proposed Cast-128 when key was varied.	89
Table 4.9: Results of standard and proposed Clefia when plaintext was varied.	91
Table 4.10: Results of standard and proposed Clefia when key was varied.	93
Table 4.11: Results of standard and proposed DES when plaintext was varied.	95
Table 4.12: Results of standard and proposed DES when key was varied.	97
Table 4.13: Results of standard and proposed MMB when plaintext was varied.	99
Table 4.14: Results of standard and proposed MMB when key was varied.	101
Table 4.15: Results of standard and proposed RC5 when plaintext was varied.	103
Table 4.16: Results of standard and proposed RC5 when key was varied.	105
Table 4.17: Results of standard and proposed Serpent when plaintext was varied.	106
Table 4.18: Results of standard and proposed Serpent when key was varied.	109
Table 4.19: Results of standard and proposed Skipjack when plaintext was varied.	111
Table 4.20: Results of standard and proposed Skipjack when key was varied.	112
Table 4.21: Results of avalanche effect of all algorithms tested when plaintext is varied.	115
Table 4.22: Result of the speeds of all algorithms tested when plaintext was varied.	118
Table 4.23: Results of avalanche effect of all algorithms tested when key was varied.	120
Table 4.24: Results of the speed of all algorithms test when key was varied.	123

Abbreviations and Glossary

3G:	Third generation
AES	Advanced Encryption Standard (AES) algorithm
AES:	AES is used on IoT to secure sensors and contactless smart cards.
Blowfish:	Blowfish is used to secure application and network layer of IoT.
BYD	Bring your device to work.
Camellia:	Camellia has been used on a prototype (encryption) for IoT.
Cast-128:	Cast has been used as one of the prototype of encryption for IoT.
Clelia:	Clelia has been used on IoT to secure health-care devices.
DES	Data Encryption standard
DES:	DES is mostly used algorithm on IoT to secure the prototype of encryption for IoT.
DoS	Denial of Service
e	Euler's number
FIPS	Federal Information Processing Standards
G	G-Permutation
GF (q)	Finite field with q elements
GF (q) ⁿ	n-dimensional vector space of GF (q)
IERC	International Energy Research Centre
IoT	Internet of Things
IP	Initial Permutation
IP ⁻¹	Inverse of initial permutation
Kn	Subkey K of n (specific) number of round
Lsb	Least significant bit
M	Plaintext was represented by initial permutation (IP)
MITM	Man in the Middle attack
MMB	Modular Multiplication based Block Cipher ()
MMB:	MMB is imbedded on the software's application of IoT.
Msb	Most significant bit
NIST	National Institute of Standards and Technology
P(S)	P-permutation of the values S

<i>r</i>	Number of rounds user want to run
RC-5	Rivest Cipher 5 algorithm
RC5:	RC5 algorithm is implemented on Mica2 hardware (base station of IoT)
<i>S</i>	A substitution operation or S-box
Serpent:	Serpent is used to secure sensors of IoT.
Skipjack:	Skipjack algorithm is implemented on Mica2 hardware (base station of IoT).
<i>w</i>	Word bit chosen from three numbers: 16, 32 or 64 on RC algorithm.

Basic Mathematical Operators and Symbols

\oplus	The exclusive-or operation
$g \circ f$	The composition of the functions f and g ($(g \circ f)(x) = g(f(x))$)
$=$ or $:=$	Equals to
XOR	bitwise addition
\cdot	Multiplication in the range $GF(2^8)$
AND	Bitwise AND operation and
OR	Bitwise OR operation.
	Bitwise AND operation
	Bitwise OR operation
\ll	Left shifting of bits
\gg	Right shifting of bits
$\{0, 1\}^n$	Bit-string of length n ,
b	The number of bytes in an original key size selected
$E(X)$:	E-permutation of the value X
$X[U-V]$	Bit string of X cut from the U -th bit to the V -th bit, U and V are positions
σ	delta constant value used on MMB
Υ	gamma constant value used on MMB
ϕ or φ	fractional part of the golden ratio calculated as follows
	The exclusive-or operation or addition modulo 2

CHAPTER 1. INTRODUCTION

1.1 Background of Study

In this section, we give an overview of the Internet of Things (IoT); its definition, components, benefits, trending and security. We also define the avalanche effect and cryptographic algorithm. We also discuss how the avalanche effect of cryptographic algorithms affects the security of IoT.

1.1.1 Definition of Internet of Things (IoT)

The Internet has always been a network of networks, connecting computers together to share information. What has changed over the past two decades is the ability to connect remote and mobile things, objects, utilities or assets to the Internet and the cloud using wireless communications and low-cost sensors with fast computing and big storage [1]. Johnson explains in [2], when all these things are interconnected it is called Internet of Things (IoT). The International Energy Research Centre (IERC) defines IoT as follows:

"A dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual “things” have identities, physical attributes and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network." [3].

Although there are several definitions of IoT, they all mean the same thing, which is, smart objects or devices connected to each other and connected to the internet and sometimes to a cloud system.

1.1.2 Benefits of Internet of Things (IoT)

The growth of IoT is driven by the vast advancement in technology seen in fast and smart devices equipped with computing and storage capabilities, as well as the decreasing cost of manufacturing electronic gadgets. Holdowsky et al. indicated in [4] that over the last two decades, microprocessors' computational power has improved, doubling every three years,

making the processing power of gadgets more suitable for IoT. Kambies et al. reported in their research [4] that the price of sensors has consistently been reduced over the past several years, and the price reductions are expected to continue well into the future. In support of Kambies et al., Johnson indicated in [2] that the average cost of an accelerometer is now 40 cents, compared to two USD in 2006. Generally, sensors vary widely in price, but many are now affordable enough to support IoT and its applications in ubiquitous computing. On both theoretical and practical fronts of sensor development, researchers and developers have concentrated on improving the accuracy of sensors. Holdowsky et al. reported in [4] that sensors of IoT are now able to report close to the real measured value. Accuracy such as that of GPS devices is one of the things that will drive the growth of IoT. Now given the fact that IoT connects different devices together, has the ability to collect huge amounts of data, and even transports data on high-speed networks than traditional internet or computers, it is clear that IoT has huge storage capacity.

IoT with its many benefits such as low cost components, speed, user-friendliness, huge storage, computational power and accuracy, makes it to be recommended and used by almost everyone anywhere in the world. In fact, Kouns in [1] mentions Gartner's prediction that by 2017, 50 percent of employers might ask their employees to "bring your own device" (BYOD) to work, thus adding to the growth of IoT. Furthermore, he extended the prediction by indicating that by 2020 there would be over 26 billion connected devices [1].

1.1.3 The Avalanche Effect of Cryptographic Algorithms

The main concern with such an enormous network, with various kinds of devices connected to it, is security. In particular, personal privacy is at risk, as these devices may expose sensitive information and potentially pose security risks. This is where cryptographic solutions in the form of ciphers are used, for integrity of information, authentication of users, and secrecy. Although there are several types of cryptographic solutions for the security of IoT, we focus our research on ciphers. We herein refer to ciphers as cryptographic algorithms.

A cryptographic algorithm is a mechanism that is used to encrypt information using plaintext (clear and readable information in static or transit) and key (like password) as inputs and ciphertext (scrabbled information in static or transit) as output on platforms like IoT. The basic usage of algorithms is to encrypt and decrypt information.

In terms of security, a cryptographic algorithm can be evaluated in several ways, and using various cryptanalytic techniques. In order to declare an algorithm cryptographically secure, its security must be tested against known cryptographic attacks. One of the techniques to avoid the success of these attacks, the tested crypto algorithm must have a strong avalanche effect [5], [6], [7]. The avalanche effect is a measure of how a small change in an input affects the outputs bits. In the context of the symmetric ciphers, this small change in the plaintext or key should cause a huge change (that is more than 50%) in the ciphertext [8], [9].

Clearly, in order to secure information stored or data in transit, the IoT needs cryptographic algorithms that have good avalanche effect [10], [11].

1.1.4 Security of the Internet of Things (IoT)

There are many attacks on IoT, some of which are due to implementation and configuration flaws, such as devices that use cryptographic algorithms with poor avalanche effect [4]. We now give a few examples of various attacks as reported in the literature.

Firstly, the Denial of Service (DoS) attacks. Holdowsky gives the DoS attack as an example of the many attacks that can be launched against machines connected to IoT [4]. DoS is the process where an intruder manipulates functionality of service on network infrastructure [6=12]. These types of attacks are a concern for IoT because they increase in proportion to the number of IoT connected devices that are under risk. These include remote IoT devices such as sensors, which are less likely to be properly secured [13]. DoS attacks get worse when more devices without strong cryptographic algorithm with high avalanche effect are interconnected [6], [8].

Secondly, is eavesdropping, that is interception of communication. Alsaadi et al. [13] discovered that eavesdropping poses a security challenge in IoT. It was found that passive attackers could intercept communication channels such as the internet, local wired networks, IoT and wireless networks, in order to access data from a stream of information [4]. Although it is not easy to prevent eavesdropping, it is easier to first encrypt data or a stream of information before transmission, by using algorithms with high avalanche effect [40], [6].

Thirdly, we look at attacks that use ‘weak’ devices to send malicious information to other connected devices [12]. In some cases intruders may exploit security vulnerabilities to create risks that may range from software risk to physical risk in some instances [14].

To support the statement above, Walters [15] reported on how he was able to remotely attack two different insulin pumps that were connected to IoT and succeeded to change their settings so that they stopped delivering medicine. Such an attack is dangerous, even fatal for patients who are dependent on those insulin pumps for their regular insulin dosage.

Klenk et al. [16] explained a situation where an attacker obtained access to a car’s internal computer network without touching the car. He described how the attacker was able to hack into a car’s built-in telematics unit and control the vehicle’s engine and braking systems.

In such situations control systems, vehicles, and even the human body can be accessed and manipulated, causing injury or worse, due to an unauthorized access to control systems. (i.e. Vehicles, body planted medical devices, SCADA, computer systems and manufacturing plants) [17]. Lequetica reports in [18] of how automotive manufacturers and car rental companies are proactively trying to address the issues of car security in the broader sense of IoT. These examples further emphasise the need for implementation of cryptographic algorithms that have high avalanche effect.

An algorithm that has poor avalanche effect does not consider the bit error (collision) characteristics that may occur when data is encrypted [19]. Collision is the process when one algorithm gives multiple numbers of same output (ciphertext) even though the inputs (plaintext or key) are totally different. Vijayrangan et al. [20] showed that there are algorithms on platforms like IoT that can pose bit error (collision) attack to the intruders because of poor

avalanche effect. Patidar et al. [8] indicated that, if an error occurs in the encrypted data over IoT, which is more likely to happen on medium platforms such as wireless medium, the decryption procedure at the receiver might cause half of the original bits to be in error due to the weak avalanche effect. Therefore, there is a need for new enhanced encryption algorithm with high avalanche effect that will take into consideration or handle the bit error or collision characteristics when data travels or sent over IoT based on avalanche effect [19], [9], [7], [8]. IoT needs a crypto algorithm with high avalanche effect. In most of the algorithms like Advanced Encryption Standard (AES) and Data Encryption Standard (DES), there is usually no initial and final vectors implemented as in Figure 1.1. In the proposed work, the initial and final vectors are implemented to the selected algorithms to enhance their strength and their avalanche effect (see Figure 1.2).

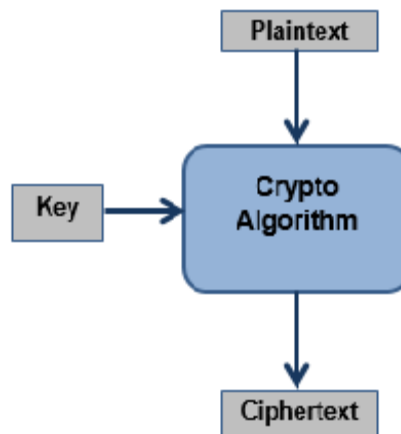


Figure 1.1: The model of well-known algorithms.

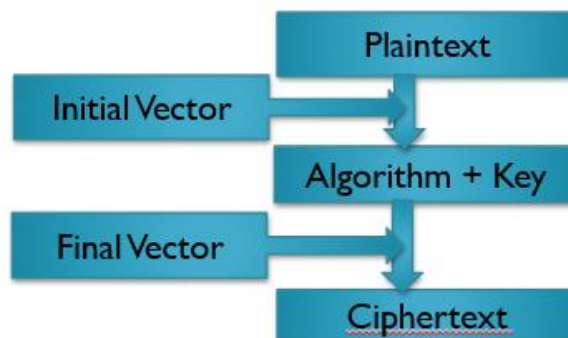


Figure 1.2: Model plan of proposed work, initial and final vectors are implemented.

1.2 Problem Statement

Protecting communication on IoT is very hard, not only in protecting application data, but also on routing and in other metadata. IoT has many vulnerabilities from design to cross-site. There are problems or concerns of privacy, such as lack of encryption, insecure software and hardware, insufficient authentication and authorization. Several methods have been proposed in order to combat this. Perhaps the most popular methods are the analysis of the speed of algorithms, analysis on the power consumption of the algorithm, analysis on the time the algorithm takes to encrypt, and analysis on the memory needed to install algorithms. The main aim of this study is to enhance the avalanche effect of algorithms used on IoT by implementing initial and final vectors. Implementation of initial and final vectors have never been used before as the purpose of enhancement of the avalanche effect. An algorithm that has poor avalanche effect compromises the security on IoT. Essentially an intruder can easily attack a cryptographic algorithm that has a weak avalanche effect on IoT by using attacks mentioned in 1.1.

1.3 Hypothesis

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

H1 There is a relationship between the avalanche effect of cryptographic algorithms used on IoT and their level of security.

H2 Improving the avalanche effect of cryptographic algorithms to increase the security of IoT.

1.4 Research Questions

The primary research question forming the crux of this research study reads; what is the relationship between the avalanche effect of algorithms used on IoT and their level of security.

The secondary research questions reads as follows:

- i. What literature is available on security concerns on IoT?
- ii. What are the types of algorithms used on IoT?

- iii. Do these algorithms have acceptable significant levels of avalanche effect (that is more than 50%)?
- iv. How can we possibly increase the avalanche effect of these algorithms if they don't have acceptable levels of avalanche effect (of more than 50%)?
- v. What are the benefit of having an algorithm with high avalanche effect on IoT?
- vi. What is the relationship between crypto algorithms used on IoT and avalanche effect?
- vii. How can we possibly give proposed future work on how to enhance the avalanche effect of these algorithms?

1.5 Research Objectives

In this study, we propose a model where the initial and final vector will be used to enhance avalanche effect. The specific objectives are:

- i. To carry out literature review on security concerns of IoT.
- ii. To determine the type of algorithms that are used to create avalanche effect on IoT.
- iii. To analyze avalanche effect by implementing the source code of avalanche effect using C++.
- iv. To compare avalanche effect levels of crypto algorithms used in IoT using the avalanche effect
- v. To investigate the possibility to increase the avalanche effect of these algorithms if they don't have acceptable significance of avalanche effect (more than 50%) using C++.
- vi. To invitigate the benefit of having an algorithm with high avalanche effect on IoT.
- vii. To investigate the relationship between crypto algorithms used on IoT and avalanche effect.
- viii. To design mathematical model that gives more confusion and diffusion to intruder using C++.
- ix. To compare the results of existing algorithms and the new designed mathematical model of algorithm.
- x. To publish our results in accredited journals and conference proceedings like IEEE.

1.6 Research Methodology

In this study, the research methodology to be used will be the experimental research method that will be conducted as follows: (1) Searching and selecting existing cryptographic algorithms used for authentication and encryption in the context of IoT. (2) Analysing the avalanche effect of selected existing cryptographic algorithms from step one. (3) Improving their avalanche effect by designing a mathematical model (where an initial vector will be XORed with the plaintext and a final vector XORed with the ciphertext) that gives more confusion and diffusion to intruder. (4) Testing the new mathematical model if it really enhances the avalanche effect of each algorithm that is used on IoT. (5) To propose the future work on how to enhance security on IoT and finally give the conclusion. (6) Showing that the high avalanche effect of an algorithm means high security of an algorithm. (7) Showing the relationship between crypto algorithms used on IoT and avalanche effect. (8) Giving the proposed future work on how to enhance the avalanche effect of these algorithms. Figure 1.3 shows the proposed model of study. Whereas Table 1.1 shows the causal relationship between research objectives and research questions.

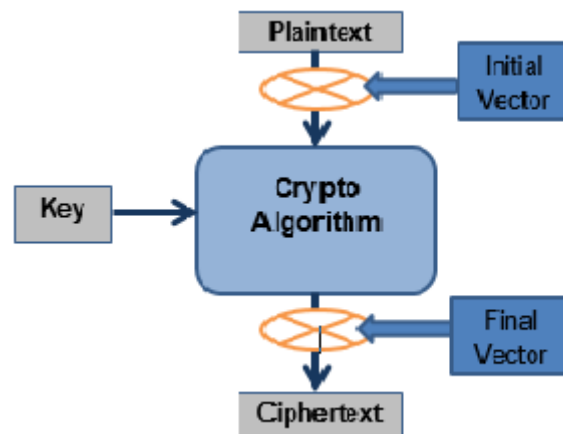


Figure 1.3: Detailed flow diagram of the proposed study with XOR block inserted.

Table 1.1: Causal relationship between research objectives and research questions.

Research Questions	Research Objectives	Procedure or methodology taken
What literature is available on security concerns in IoT?	To carry out literature review on security concerns in IoT	Journals, articles, papers and books were used to compile the literature review of concerning IoT security.
What types of algorithms are mostly used to create Avalanche effect on the IoT?	To determine the type of algorithms that are used to create avalanche effect on IoT.	From literature we found ten algorithms mostly used on IoT. Namely: DES, AES, Serpent, Blowfish, Camellia, MMB, Cast, RC5, Clefia and Skipjack.
Do these algorithms have acceptable significance of avalanche effect?	To measure and compare avalanche effect of crypto algorithms used in IoT.	Most of algorithms found from literature review had no significance of avalanche effect when avalanche effect was calculated using C/C++ code.
How can we possibly increase the avalanche effect of these algorithms if they don't have acceptable significance of avalanche effect?	To investigate various methods of improving the avalanche effect of algorithms.	We increased avalanche effect by using initial and final vectors derived from PI value, in most algorithms using C++.
What are the benefit of having an algorithms with high avalanche effect on IoT?	To improve their avalanche effect by designing mathematical model that gives more confusion and diffusion to intruder using C++.	It is found that if algorithm has high avalanche effect, it gives confusion and diffusion to the intruders and hackers. Meaning it is cumbersome to crack that algorithm. Refer to chapter 4.
What is the comparison between avalanche effects levels of existing crypto algorithms used on IoT and proposed ones?	To compare the results of existing algorithms and proposed one based on avalanche effect.	It is found that it is possible to increase avalanched effect on certain algorithms used on IoT using the proposed method.
How can we possibly give proposed future work on how to enhance the avalanche effect of these algorithms?	To analyze if it is possible to increase the avalanche effect of these algorithms if they don't have acceptable significance of avalanche effect using C++.	The proposed work showed that it is possible to increase avalanche effect using initial and final vector from Pi value.

1.7 Significance of the Study

After the study, the increase of avalanche effect on each algorithm has the potential to securing communication, data and sensitive information transported and stored on IoT. It is our belief that results, underpinned by these studies and work, will give other researchers the knowledge on how to improve and implement the security on IoT. We believe that as we explore the issues of security surrounding IoT, we are going to be adding yet another dimension of security to it. That is enhancing the security of the algorithms used on IoT. The study will give cryptographers the necessary background on how to improve avalanche effect when designing algorithms.

1.8 The Research Roadmap

Chapter 1 discussed an introduction of the study. This includes background of study, problem statement, hypothesis, research questions, research methodology, research objectives, significance of study, research roadmap and chapter summary.

Chapter 2 will review the literature related to our study. This includes: introduction, overview of IoT, encryption methodology, algorithms used in the security of IoT, avalanche effect, security in the IoT, types of attack on IoT, related work and chapter summary.

Chapter 3 will discuss the research methodology related to our study. This includes introduction, source of initial and final vectors, PI methodology that is the overview of PI, methodology of study based on the avalanche effect, research design, experimental procedure and chapter summary.

Chapter 4 will give the results, discussion and analysis related to our study. This includes graphical comparison of avalanche effect, the mathematical discussion and analysis.

Chapter 5 is the conclusion and future work. This includes conclusion of the study, its results analysis and proposed future work to be done.

1.9 Chapter Summary

In this chapter, we defined IoT, its origin, how is it growing on daily bases, its affordability and accessibility, its fastness, its user-friendliness and its security. We also discussed cryptographic algorithms and their avalanche effect as the background. The problem statement, research methodology, hypothesis, research questions, research objectives, significance of the study and the research roadmap were discussed in this chapter to give understanding of introduction and background of research.

CHAPTER 2: LITERATURE REVIEW

2.1. Introduction

In this chapter, we present literature review of Internet of Things: its protocols, architecture, communication models (that are device-to-device, device-to-cloud, device-to-gateway, and back-end data) and security.

We also present literature review of encryption methodologies and techniques used on IoT: its symmetric cryptography, asymmetric cryptography, steganography, vigenère and hash (authenticated) function. We present types of algorithms used on IoT: The Advance Encryption Standards (AES), Blowfish, Camellia, CAST-128, Clefia, Data Encryption Standard (DES), Rivest Cipher 5 (RC5), Modular Multiplications based Block cipher (MMB), Serpent and Skipjack algorithm. These are the algorithms selected for the study because they are mostly used on different devices of IoT. Refer to section 2.4 , section 2.7 and Table 3.1. We present their origins, avalanche effects and security.

We also present literature review of the types of attacks used on Internet of Things like Denial of Service (DoS), Man in the Middle (MITM), eavesdropping, honeypot, differential cryptanalysis and differential fault attack. These attacks are used to attack cryptographic algorithms mentioned above due to their avalanche effects. Intruder can crack the machine, information in transit, strength of algorithm to get secret key and information.

Lastly we present the related work done by other researchers concerning the related studies. Little has been done in this regard. In our related work we present literature of where these cryptographic algorithms are installed on IoT. We present attacks used to crack these algorithms which are installed on IoT due to their avalanche effect. We also present other studies done by other researchers to enhance security of these algorithms using avalanche effect and the results found.

2.2. Overview of Internet of Things

IoT uses protocols, network layers, wireless connectivity, communication models (that are device-to-device communication, device-gateway communication, back-end data sharing

communication, device-to-cloud communication) and small devices connected to each other, which operate on high speed and have huge storage, flexible to cloud computing and other advantages compared to standard internet.

2.2.1 IoT Protocols

IoT has many protocols developed from the International Organization for Standardization (ISO) stack for IoT devices operation [21]. Security of these protocols is determined by strong cryptographic algorithms [15], [22] – [23]. Several protocols exist within IoT stack, such as the Constrained Application Protocol (CoAP) which is messaging protocol, Infrastructure protocol for networking and the Identification protocol used to identify the user. Other protocols such as the Message Queuing Telemetry Transport protocol (MQTT) is used for messaging and is maintained by the Advanced Message Queue Protocol (AMQP), Discovery protocol which is used to discover web and nodes. Others are the Data Protocols or the Representational state transfer (REST) protocol. The REST protocol is used to handle data like the web socket. Apart from the protocols mentioned above several others protocols exist which are the Device Management protocol that provides ways on how to manage devices, the semantic protocol which provides web services, the stomp protocol that handles text oriented messaging. All of these protocols are designed to save energy, with slow to computing time and less memory because they have space limitation and limited power supply as the use batteries [24]. Due to these limitations, it follows that the security problem is one of the issue when dealing with IoT [25], [24]. The summary of the protocols used on IoT are defined in Table 2.1.

Table 2.1: Summary of the protocols used on IoT [26].

Protocol	CoAp	XMPP	RESRful HTTP	MQTT
Transport	UDP	TCP	TCP	TCP
Messaging	Request/Response	Publish/Subscribe/Request/Response	Request/Response	Publish/Subscribe/Request/Response
2G,3G,4G Suitability (1000s nodes)	Excellent	Excellent	Excellent	Excellent
LLN Suitability (1000s nodes)	Excellent	Fair	Fair	Fair
Compute Resources	10 ks RAM/Flash	10 ks RAM/Flash	10 ks RAM/Flash	10 ks RAM/Flash
Success Stories	Utility Fields Area Networks	Remote management of consumer white goods	Smart Energy Profile 2 (premise energy management/home services)	Extending enterprise messaging into IoT application

2.2.2. IoT Architecture

There are several existing stack layers within IoT which are used in IoT architecture. However, Security of these layers is determined by strong cryptographic algorithms [21]. Within such domains researchers are busy increasing network layers of IoT architecture by breaking down some of the main layers into sublayers [21]. Recently, IoT is considered academically and practically by several researchers, that its architecture is basically composed of three layers: the perception layer, the network layer and the application layer [27]. The application layer handles all applications of IoT while the network layer deals with connection to the network such as the wireless or the wired network [27]. The perception layer is used to request, acquire, collect and process the data from IoT communications [28]. Figure 2.1 shows IoT architecture and its network layers.

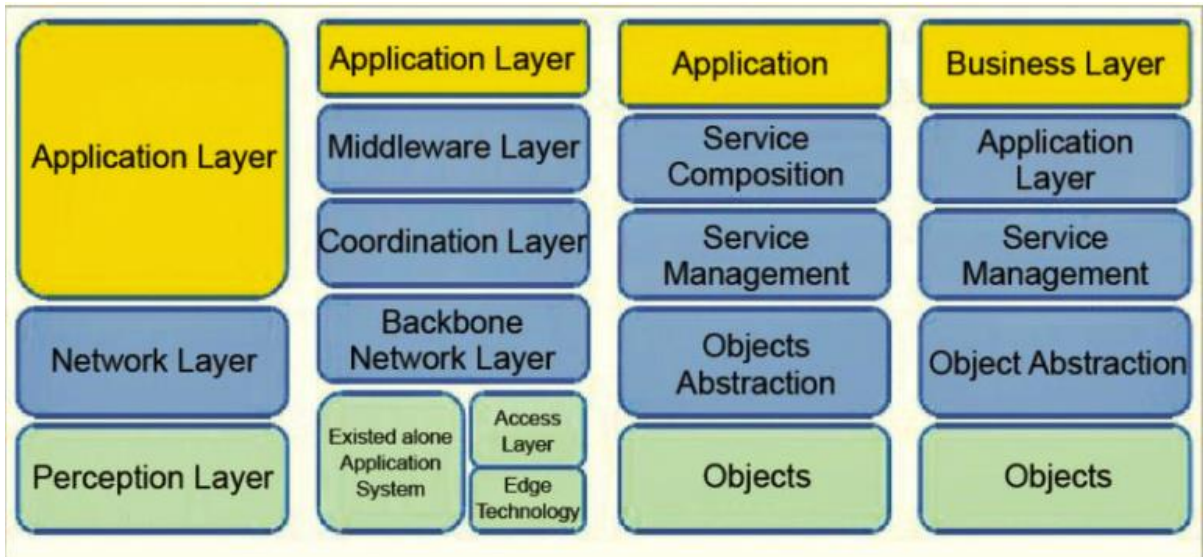


Figure 2.1: IoT architecture and network layers [29]

2.2.3. Device-to-Device Communication model

The device-to-device communication model is the mechanism when two or more devices are directly connected to establish communication amongst one another, without using gateway, cloud computing and servers [27]. These devices communicate over various kind of applications like SHAREit, Bluetooth, ZigBee or Z-Wave [30]. Its security and trust rely on direct connection between the devices by pairing them [31]. No internet protocol is used on device-device communication model [27]. Santosh et al. [30], described device-to-device communication model as shown in Figure 2.2.

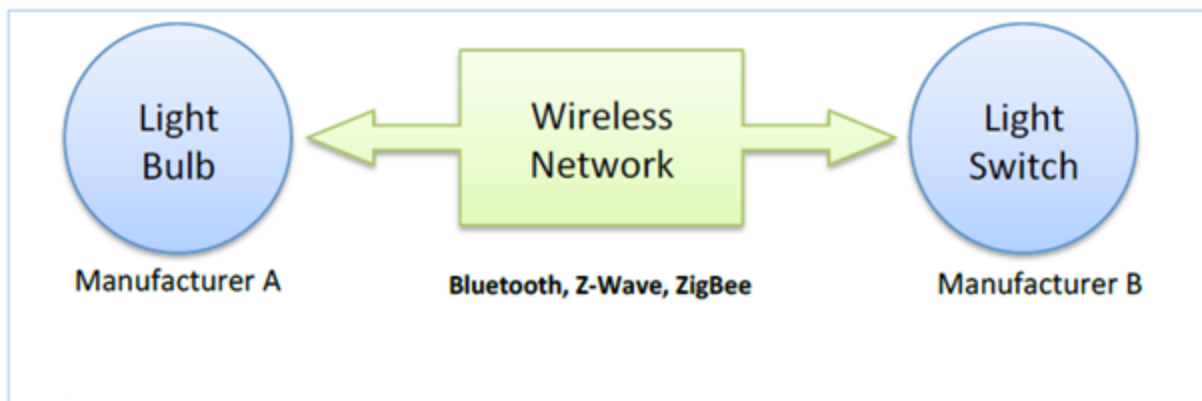


Figure 2.2: Device-to-device communication model [30].

2.2.4. Device-to-Cloud Communication

In a device-to-cloud communication, IoT devices connect directly to an internet cloud service like service providers to establish communication amongst each other [27]. This approach normally takes advantage of existing channel communications mechanisms like Ethernet, mobile phone service providers or Wi-Fi connections to establish a connection between the device and the internet network, which finally connects to the cloud computing and services [30]. Security and privacy rely on service provider [31]. Communication might become dysfunctional if service provider disappears or changes hosting provider [31]. Santosh et al. [30] described device-to-cloud communication model as shown in Figure 2.3.

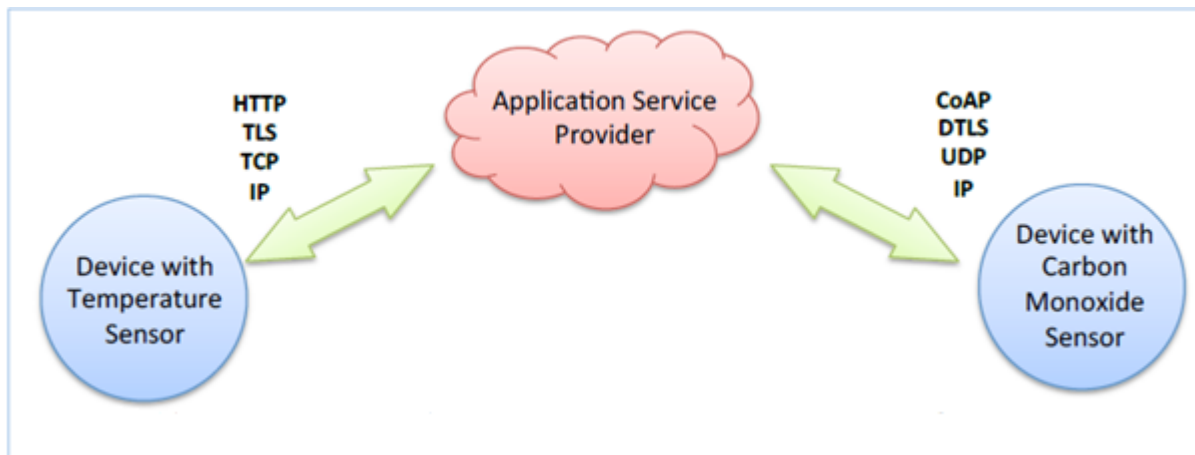


Figure 2.3: Device-to-cloud communication model [31].

2.2.5. Device-to-Gateway Communication

The device-to-gateway communication is the process where the devices connected to IoT gateways as a channel to reach cloud services. IoT gateway's function is to fill the communication gap amongst IoT devices, systems, sensors, equipment and the cloud. IoT gateway also provides internal processing and storage solutions. Security and data confidentiality rely on website visited and browsed during communication [31]. Websites might become dangerous if intruder wants to attack, he/she might hack or phish information from the user using fake website [31]. Santosh et al. [30] described device-to-gateway communication model as shown in Figure 2.4.

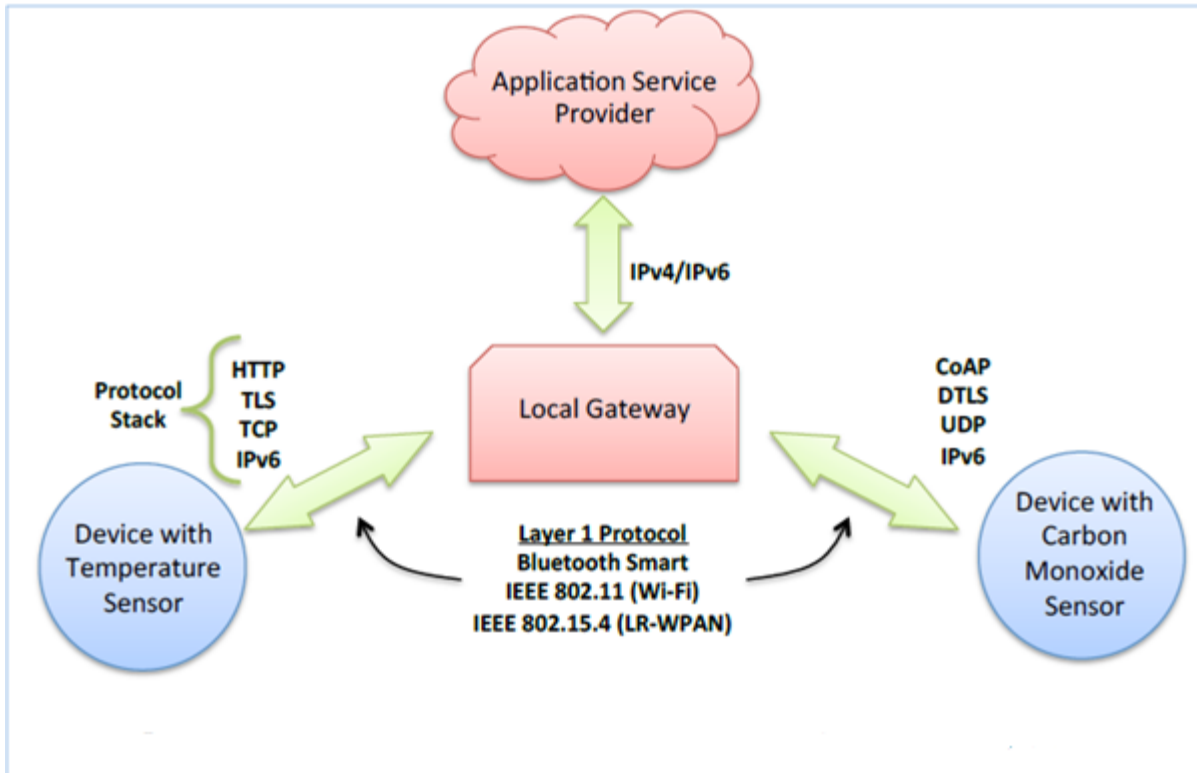


Figure 2.4: Device-to-gateway communication model diagram [30].

2.2.6. Back-End Data Sharing communication

The back-end data-sharing communication alludes to a communication architecture that allows users to transfer and analyze data from a cloud service in combination with data from other sources [27]. The need of internet protocol is not necessary in the communication [30]. Security and trust rely on the application service providers [31]. If the application service provider stops the services or put software on application that has the open-back-door, all user's information and data might be exploited, read or sent to the wrong recipients [31]. Santosh et al. [30] graphically described back-end data sharing communication model as shown in Figure 2.5.

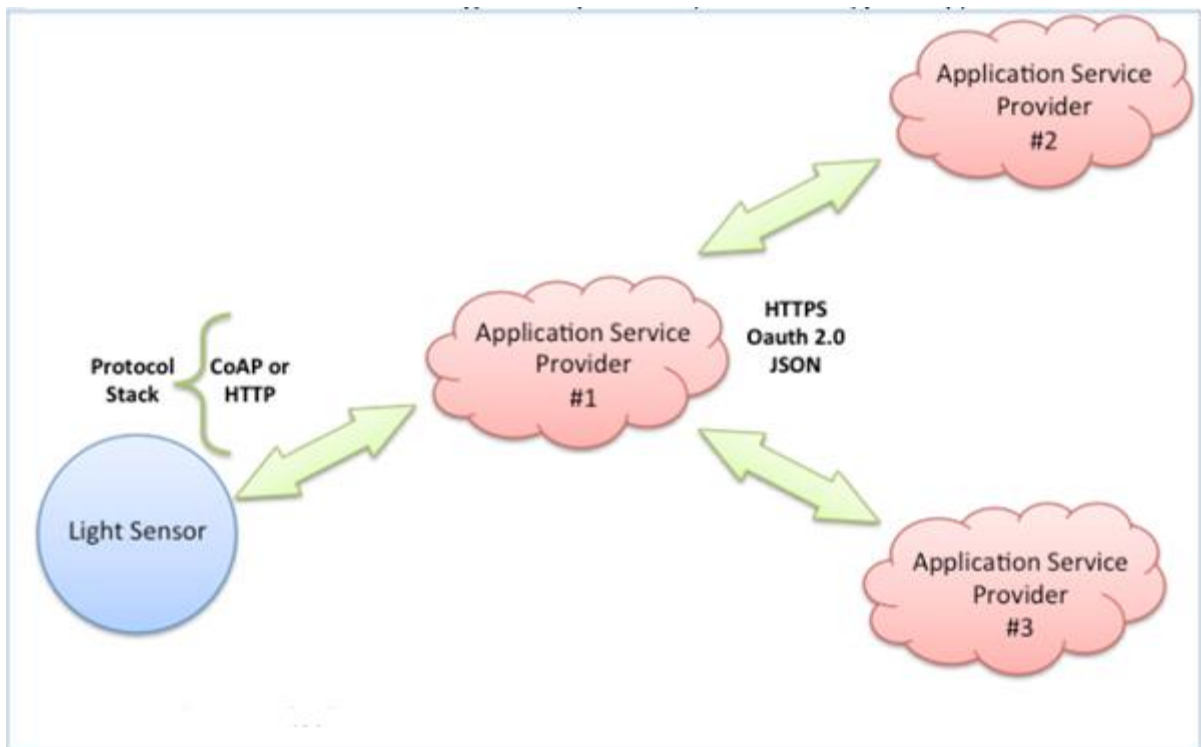


Figure 2.5: Back-end data sharing model diagram [30]

2.2.7. Security and the Internet of Things

Usually, devices connected to IoT expose sensitive information and may become a potential security risk such as: (1) Privacy defined as a mechanism of blocking all unauthorized accesses. If there is a lack of securing information, that leads to enable unauthorized access and misuse of personal information, then there is a lack of privacy; (2) Data confidentiality defined as a mechanism to keep secrecy of data and information. If there is a lack of keeping secret of data and information, that leads to facilitate attacks on the storage devices of IoT, then there is a lack of data confidentiality; and (3) Trust is a mechanism of identifying and verifying the sender, data or device used on IoT. If there is a lack of verifying the legitimacy of devices, persons, service providers or cloud services when users are exchanging information over IoT, this leads to create safety risks [32], [33]. The main three IoT security risks are represented by Bilal [27] as shown in Figure 2.6.

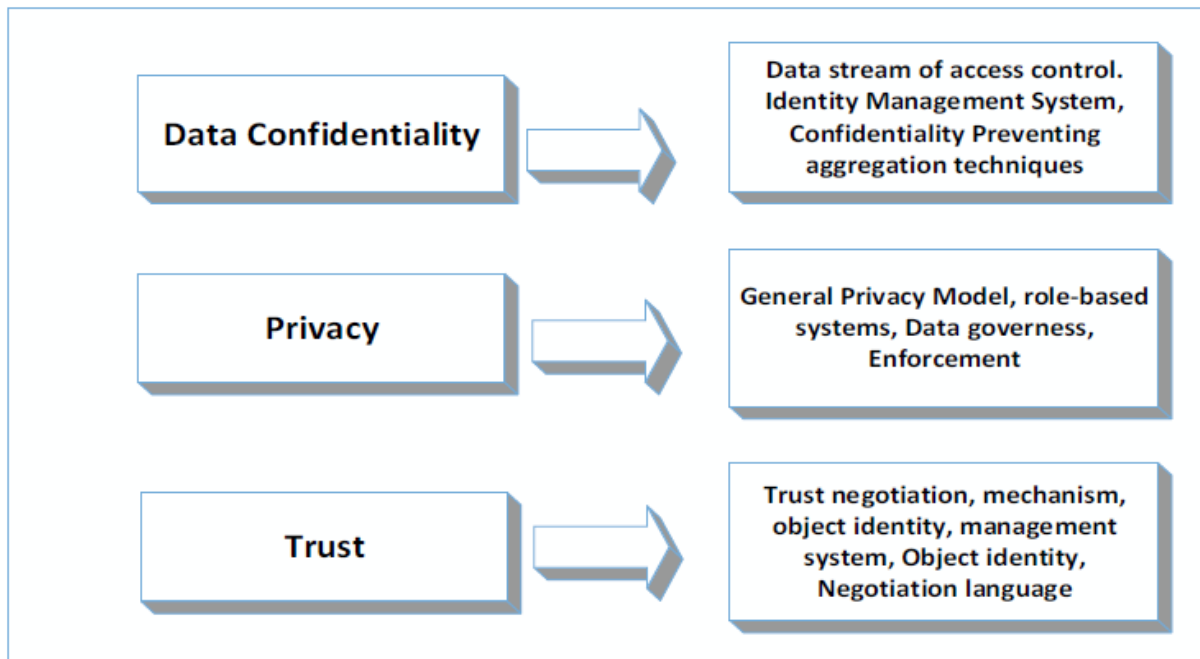


Figure 2.6: IoT Security Challenges [27].

Intruders could exploit user identity passwords, credit card numbers, leading to them being vulnerable to theft or fraud [34]. Thus, the more devices are connected to IoT in their homes or workplaces, the more vulnerabilities which an unauthorized person could use to access sensitive information [34]. Most of devices have no cryptographic algorithms, and others have algorithms with less avalanche effect [9], [7], [35]. This exposes variability on devices [8].

Security vulnerabilities on connected device could lead to potential attacks at the end-user's network, or facilitate attacks on other systems [36], [37]. An intruder could exploit security vulnerabilities to create risks that could affect software or hardware in some cases [41, 38]. For example, attacks like: Data confidentiality, data integrity, data authentication, data freshness, availability, time synchronization and many more explained by Borgohain [36], Walters [38] and by Klenk [39]. Therefore there is a need to develop algorithms with high avalanche effect to secure all levels of IoT [40], [8]. Therefore, all levels of IoT from physical devices, controllers, connectivity, servers (edges), data accumulation, data abstraction, application, collaboration up to the top processes should be protected [41], in other words everything that is used on IoT should be secured before use to achieve identity management, authentication, secure storage, secure communication, secure network access and secure content [42]. This can be shown in Figure 2.7 on the right hand side of it [41].

Internet of Things Reference Model: Security

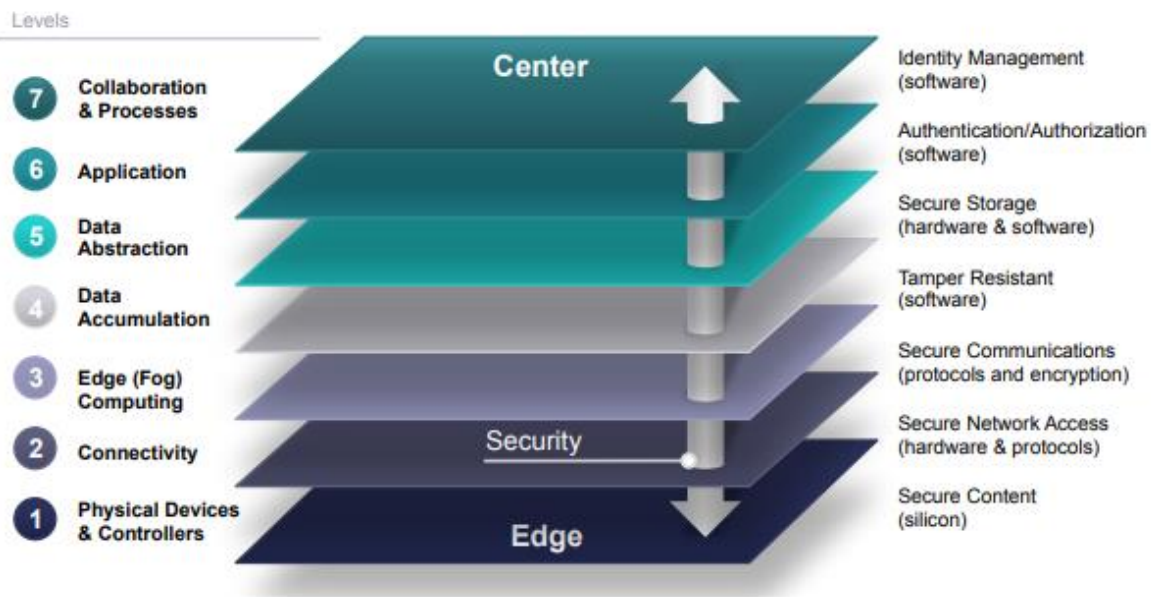


Figure 2.7: IoT reference model: Security [41].

2.3 Encryption Methods and Techniques

Cryptography is the art of encrypting and decrypting information, data and messages. During ancient ages, encryption was done using the pen-and-paper methods based on the letter substitutions and shifting such as Vigenère and Steganography encryption. Today networks like IoT focus on digital cryptographic systems such as symmetric, asymmetric and hash function encryption that can encrypt and decrypt information, data and messages using computers.

2.3.1 Symmetric Encryption

This is the kind of encryption that uses a same key to secure data from sender to receiver for secure communication [43]. Kaur et al. [44] explained the process of symmetric encryption as uniform or symmetric because there is only one key used for encryption and decryption process. There are several different types of symmetric key algorithms that can be used, such as AES, DES, Blowfish, Clefia and Serpent [45].

2.3.2 Asymmetric Encryption

This is the kind of encryption method, where the key used to decrypt data or information is totally different, compared, to the key used to encrypt same data or information [44]. Asymmetric encryption is also known as public-key encryption [46]. There are many asymmetric encryption key algorithms such as Elliptic Curve Cryptographic algorithm (ECC), Rivest Shimar Aglemen (RSA), Diffie-Hellman and Digital Signature Algorithm (DSA) [47].

2.3.3 Steganography

This is the kind of encryption which puts data or information onto other mediums in an unnoticeable way [48]. These mediums are objects that are usually viewed by human beings. These objects can be picture, audio, and video files [49]. A very simple example of steganography is the invisible ink that is used to write invisible text on a paper that has visible text, the receiver will ignore the visible text and read the invisible one written by invisible ink using a candle [50]. Another example is when the video is played but pictures on video are sending a message which totally different video to unauthorized person. The grass and trees on the videos can be used as messages, and are read as morsecode (the ring tones of an ancient phone were recognized by long-short tones, on-off tones).

2.3.4 Vigenère Encryption

In a vigenère encryption, each letter of the alphabet is substituted or shifted to some number of places [51]. For example, in a vigenère encryption of shift 3, A letter A would become E, B would become F, Y would become C and so on [52]. The vigenère encryption is composed of several shifts of encryption in sequence with different shift values [53]. An example of vigenère encryption is Caesar algorithm [51].

2.3.5 Hashing (Authenticated) Encryption

In hashing, a unique fixed-length signature is created for a specific data or information set [54]. Each and every “hash” is different to a specific data or information, so little changes to that data or information would be easy to notice. When data or information is hashed, it cannot be

reversed nor deciphered [55]. It is simple to tell if the data or information received has been tempered with or not. Hashing is used to check if intruder has tempered with communication [54].

2.4 Algorithms Used In the Security of IoT

Several algorithms are used to secure IoT. In this study we selected ten algorithms that are used on IoT. The reason is that, they are mostly implemented on different devices of IoT. These algorithms are:

- i. AES: AES is used on IoT to secure sensors and contactless smart cards.
- ii. Blowfish: Blowfish is used to secure application and network layer of IoT.
- iii. Camellia: Camellia is used on a prototype (encryption) for IoT.
- iv. Cast-128: Cast is used as one of the prototype of encryption for IoT.
- v. Clefia: Clefia is used on IoT to secure health-care devices.
- vi. DES: DES is mostly used algorithm on IoT to secure the prototype of encryption for IoT.
- vii. MMB: MMB is imbedded on the software applications of IoT.
- viii. RC5: RC5 algorithm is implemented on Mica2 hardware (base station of IoT).
- ix. SWSDSSerpent: Serpent is used to secure sensors of IoT.
- x. Skipjack: Skipjack algorithm is implemented on Mica2 hardware (base station of IoT).

2.4.1 The Advanced Encryption Standard (AES) Algorithm

AES is defined as a cryptographic algorithm that was designed by Rijndael and it was submitted to the National Institute of Standards and Technology (NIST) in order to secure electronic data [56], [57]. The specification of AES was explained in Federal Information Processing Standards (FIPS) Publication in 1997 and it was accepted by NIST [58], [59]. The AES was analyzed as a block cipher used to encrypt/decrypt blocks of 128 bits and had capacity of using keys of 128, 196 or 256 bits [60]. AES is mostly used in hardware and software of IoT [56].

2.4.2 Blowfish Algorithm

Blowfish algorithm is as an algorithm created by Bruce Schneier in 1993, it uses a 64-block size and a key length of 33 up to 448 bits [61]. Blowfish is used to secure applications and network layer of IoT [62]. In this study, we focus on Blowfish which uses a key of 128 bits called Blowfish-128 encryption algorithm. Blowfish is described as a block cipher using 16 rounds [61], [62].

2.4.3 Camellia Algorithm

Camellia is defined as a block cipher algorithm designed by three companies called Telephone Corporation, Nippon Telegraph and Mitsubishi Electric Corporation in 2000 [63]. It was submitted to ISO/IEC JTC 1/SC 27 as a consideration proposal for an international encryption standard in 2000 [59]. Camellia has been accepted for use by the ISO/IEC [63]. These companies (Telephone Corporation, Nippon Telegraph and Mitsubishi Electric Corporation) combined expertise from their companies to develop Camellia [65]. Camellia is defined to use the 128-bit block size and 128, 192 and 256-bit key sizes [66]. Encryption and decryption of Camellia is defined as the same procedure but the order of the sub-keys is reversed in decryption process [67].

2.4.4 CAST-128 Algorithm

CAST-128 is defined as symmetric block cipher algorithm developed in 1996 by S. Tavares and C. Adams [67], [68]. CAST-128 is explained as a cipher that uses 64-bit plaintext blocks under a key size 128 bits [69], [68]. The algorithm is defined to operate on Feistel network structure same as DES [67] with 12 or 16 rounds.

2.4.5 Clefia Algorithm

Clefi algorithm is an algorithm designed by Nagoya University and Sony [70], [43]. It was kept secret by Sony as proprietary algorithm until the weakness was found on its S-Boxes configuration, after that it was published to the public domain [71]. It is a 128 bits block cipher and uses different key lengths: 128, 192 and 256 bits [72]. Clefia is explained to use two Feistel

functions, a 4-branch and an 8-branch [43]. The number of rounds are determined by the length of key [70]. For 128 bits key, 18 rounds are used. 192 bits key 22 rounds are used and 256 bits key 26 round are used [43].

2.4.6 Data Encryption Standard (DES) Algorithm

DES is defined as a block cipher algorithm widely used on IoT in the world [45], [73]. Kammer et al. [74] indicated that man called Horst Feistel developed a function called feistel in 1970. This function was used as a building block to strengthen DES. After development, the algorithm was submitted to the National Bureau of Standards (NBS) after the invitation to propose an algorithm for the protection of data and information of the United State of America (USA) government [75]. In 1976, NBS slightly modified DES after picking up some weaknesses on the size of key against brute-force attacks, which was made known to the public by Federal Information Processing Standard (FIPS) of the USA in 1977. DES was developed to encrypt and decrypt blocks of plaintext and ciphertext of 64-bits long respectively and uses 56 bits of key [73]. It is a 16 rounds block cipher [74], [75]. Decryption was applied by using the same sub-keys used in encryption, but in the reverse order [76], [73].

2.4.7 Modular Multiplication based Block Cipher (MMB) Algorithm

Modular Multiplication based Block Cipher (MMB) is defined as a block cipher that was developed by Govaerts, Daemen and Vandewalle in 1993 [77], [78]. Their goal was mainly to replace IDEA block cipher [79]. MMB is an algorithm which uses 6 rounds for decryption and encryption [62]. The plaintext block and key size are 128 bits each [79], [78].

2.4.8 Rivest Cipher 5 (RC-5)-32/32/16 Algorithm

Rivest Cipher (RC5) is a symmetric block cipher that was developed by Ronald Rivest in 1994 [80], [81]. His aim was to develop a fast algorithm to secure data. It is said to be a fast symmetric block cipher because of its elementary computational operations of encryption on full words of data takes a short time to encrypt [81]. Unlike other algorithms, RC5 was designed to use parameters set by the user before encryption takes place [80] and depending on the environment or the platform where it is going to be implemented. These parameters are defined

as w , r , and b [81]. Where w is word byte chosen from the three numbers 16, 32 or 64 [80]. The w is a number of word byte allowed to be chosen for RC5 [81]. The second parameter r is the number of rounds user want to run [80]. The third parameter b is the number of bytes in an original key size selected [80], [81]. The key size is defined to range from 0 to 255 bits [80], [81].

2.4.9 Serpent Algorithm

Serpent is a symmetric block cipher designed by Lars Knudsen, Ross Anderson and Eli Biham in 1998 [82], [83]. Their goal was to submit the algorithm to be considered as a candidate of the Advanced Encryption Standard [82]. Serpent is adopted from DES algorithm, its S-Boxes are extracted from DES and has a new structure that gives confusion to the intruder [83]. It uses a 128-bits block size and key size of 128, 192 or 256 bits [84].

2.4.10 Skipjack Algorithm

Skipjack is an algorithm that was designed by National Security Agency (NSA) in 1993 for the tools like hardware crypto-processors used in the military agencies for privacy, confidentiality and integrity services [85], [86]. NSA is an intelligent department of USA, one of its core functions is to develop and implement cryptographic standards for department of defense in the USA. This algorithm uses a plaintext and ciphertext block size of 64 bits, a key block size of 80-bits, and is defined to have 32 rounds [85], [86]. In this study, an 80-bits key will be used because of the limitation of key block specification of the algorithm.

2.5 Avalanche Effect and Security in the Internet of Things

Zibideh [19] defined the avalanche effect as a desirable property of traditional algorithms like Advanced Encryption Standard (AES), Data Encryption Standard (DES) and other well-known algorithm used on IoT. The avalanche effect is satisfied when one input bit is changed, each of the output bits should change with a probability of more than 50% [88]. In context of symmetric ciphers a small change of the plaintext should cause a huge change in the ciphertext [89]. Vijayrangan et al. [20] showed that there are algorithms that can expose attacks like bit error (collision) attack to the intruders because of their poor avalanche effect. The avalanche effect's

definition is based on the bits of an input and the output bits of the algorithm. In [90], it was found that these traditional algorithms could be exposed to a bit error during decryption process. Sobti et al. [91] showed that there is a need to develop algorithms which are resistant to attacks for prevention of the intruders to access and attack communications on IoT. Even if algorithm was used to strengthen security of IoT, if it does not have sufficient avalanche effect [7], [35], therefore it does not prevent the bit error characteristics when used on IoT [92]. If an error occurred in the encrypted data over IoT, which was found to be more likely to happen on channels such as the wireless channel of IoT, the decryption procedure at the receiver side could cause half of the original bits to be in error due to the poor avalanche effect [92]. Within this theoretical paradigm, it is clear that there is a need of new secure encryption algorithms or modification of traditional algorithms that would take into consideration or handle the bit error characteristics on IoT by enhancing or considering avalanche effect [89], [35], [8]. In the following section we will discuss several attacks used on IoT due to lack of sufficient avalanche effect.

2.6 Types of Attacks on Internet of Things

There are many attacks that have been discovered since the establishment of IoT. In this section we discuss an overview of attacks that are mostly used on IoT: Attacks like Denial of Service (DoS), Man in the Middle (MITM), Eavesdropping, Honeypot Attack, Collision (Bit Error), Differential Cryptanalysis and Differential Fault Attack.

2.6.1 Denial of Service (DoS) Attack

DoS attack is defined as when an intruder manipulates functionality of services on network infrastructure [33]. Studies on DoS attacks such as those done by Alsaadi et al. [93] discovered many implementation mistakes and configuration flaws on IoT's deployments and developments. For instance, attacks such as DoS could occur on machines connected to IoT. DoS was found to be a concern due to the fact that a number of IoT devices were found to be under the risk of being attacked, including remote IoT devices such as sensors where cryptographic algorithms are implemented. These sensors are unlikely to be properly secured, which would make them easy to be exploited [93]. In summary, DoS have been found to be more problematic, because if more devices are interconnected, the more intruders could have

access to them [94]. As the more and more devices become connected to IoT without proper algorithms or algorithms with weak avalanche effects, vulnerabilities could increase allowing intruders to connect to fake devices that could also be used in such attacks [37], [95].

2.6.2 Man in the Middle (MITM) Attack

Man-in-the-Middle attack (MITM) attack is when the intruder secretly transfers and possibly manipulates the communication between sender and receiver who believe they are secretly communicating with each other [10]. MITM attacks usually destroys data confidentiality. Data confidentiality is a big problem on IoT devices and services if a weak algorithm is used [96]. On IoT functionality, not only would the user have access to information but also have access to other interceptive objects [10]. Within IoT domain, there is a need to address the following important issues: access control, authorization, need for strong cryptographic algorithms and identity management [96]. IoT devices have been seen as not being able to verify the entity (person or any other device) for authorization to gain access to a service [10]. To do verification of entity, a strong cryptographic algorithm must be implemented to identify and encrypt entity.

2.6.3 Eavesdropping Attack

An eavesdropping attack is where the intruder secretly collect or steal information that is transferred over network by either computers, wireless media or devices [37]. Alsaadi et al. [93] indicated that there are some of the scholars who were able to discover eavesdropping (interception of communication) on IoT. It was found that passive attackers could intercept communication channels such as the internet, local wired networks and wireless networks, to access data from the stream of information [37]. If communication (data and information) is encrypted by a strong algorithm, the intruder could not read nor hear the communication even if he can steal or access it [93], [37].

2.6.4 Honeypot Attack

According to Yusuff [97] honeypot is an instrument used to collect data and/or information stored during and after communication. This means that when an algorithm is designed to secure communication for IoT, it must be borne in mind that it has to be strong enough to

prevent data or information from being collected, analyzed, intruded, or attacked by an unauthorized human being or hacker using honeypot attack [98].

2.6.5 Collision (Bit Error) Attack

A collision attack is an attack when an intruder tries to find two different input bit strings of crypto algorithm that produce the same output bit strings of result [99]. In simple terms when two different inputs give the same output, it is called a collision. If an attacker attacks crypto algorithm using a collision, this attack is called a collision attack [100].

2.6.6 Differential Cryptanalysis Attack

Differential cryptanalysis is when the intruder attacks cryptographic algorithm by studying the differences in an input and the result differences in an output [101]. This attack was first published after attacking the full 16-round DES in less than 2^{55} complexity [70]. It is different to collision attack, because it first XOR the inputs to get the difference, then the difference is used as the input to recover the secret key [102].

2.6.7 Differential Fault Attack

Differential fault Attack is a technical attack that powerful cryptanalytic technique that disorganizes and manoeuvres any type of cryptographic machine so that it can yield erroneous results to discover secret keys [103]. Wei et al. [104] used this attack to crack Serpent algorithm. Rivain [103] used to crack DES algorithm.

2.7 Related Work

IoT uses AES algorithm to encrypt its sensors [22]. Paul et al. [105] carried out some work on how to enhance the avalanche effect on AES algorithm. Their main aim was to enhance avalanche effect of AES to prevent attacks [105]. So many attacks used to crack AES have been successful on its first few rounds but unsuccessful on full rounds [106]. An attack was used on AES by Dunkelman et al. [106] in 2015. This attack is called differential cryptanalysis, it is used to recover the related key used on AES. Dunkelman et al. [106] manage to reduce

AES's complexity to recover AES's secret key bits. To prevent these attacks, Paul et al. [105] used matrix based transposition method. Matrix based transposition is when a key is transposed by matrix before is mixed with an algorithm. They transpose the encryption key using a size 16x256 matrix. They indicated that the use of insecure cryptographic algorithm needs encoding of information based on the avalanche effect to deal with security attacks [105]. In their research, it was shown that by applying a based transposition matrix procedure method, avalanche effect was highly improved on the first round (from 48% to 56%). When the method was applied on more than eight rounds, the avalanche effect almost remained the same as that of standard AES. From their results, when eight rounds were used, the avalanche effect changed from 56% to 57%.

IoT uses Camellia algorithm to secure its medical data systems [107]. Santoso et al. [108] carried out some work on how to improve the avalanche effect on Camellia algorithm. The main aim to improve avalanche effect was to avoid attacks on medical data systems. Walters [15] attacked medical insulin pumps and temper with settings remotely. To avoid this kind of attack, Santoso et al. [108] compared all types of Camellia algorithms based on avalanche effect. From the results, they got avalanche effect from minimum of 45% to a maximum of 60% [108]. They recommended that a Camellia with 60% of avalanche effect, should be used to secure medical data on IoT systems.

IoT uses Blowfish algorithm to encrypt its network layer [22]. More recently, Mahindrakar [10] carried some work on analysis of Blowfish's avalanche effect without modifying it. From his results, he indicated that the avalanche effect of Blowfish algorithm is not strong enough and it is insecure. He indicated that Blowfish gave less than 50% of the avalanche effect when one bit was flipped in each round. He discovered that the avalanche effect of Blowfish was almost strong only when the plaintext was changed, than when the key was changed. Mahindrakar indicated that Blowfish is vulnerable to be attacked if its avalanche effect is not increased [10].

IoT uses DES algorithm to secure its prototypes [22]. Paul et al [105] indicated that DES is no longer secure due to its smaller size of the key (56 bits). DES is also vulnerable to brute force attack [105]. It is also vulnerable to differential cryptanalysis attack [70]. Ramanujam et al. [109] used ancient cryptographic algorithms (Playfair, Ceaser and Vigenere algorithms) to scramble input bits with modern cryptographic algorithms blocks of DES and Blowfish to make new algorithm that will secure better IoT's prototypes than DES. They combined four algorithms to make one algorithm [109]. They used ciphertext derived from three ancient

algorithms as the plaintext of modern algorithm [109]. They found that the average avalanche effect of standard Blowfish algorithm being 28.7100% but that of standard DES being 54.6800%. After that they mixed ancient and modern algorithms to make new algorithm, they managed to get more than 70% of avalanche effect, this new algorithm had an excellent avalanche effect but it was expensive to implement due to a limited memory of the device [109].

IoT uses Cast-128 algorithm to secure its prototypes [15]. Wang et al. [110] attacked Cast-128 using improved differential cryptanalysis attack. They managed to attack Cast-128 successfully, and more easier and simple to attack when the key was weak. Krishnamurthy et al. [111] tried to enhance the security of Cast-128 algorithm by modifying its feistel function to improve the avalanche effect. They managed to get 66.6600% of avalanche effect [111]. Which is an improvement compared to 50%.

IoT uses Clefia algorithm to secure its Radio-Frequency Identification (RFID) [112]. Boura et al. [113] successfully attacked Clefia algorithm using differential cryptanalysis attack. Mostly, differential cryptanalysis attack works better when the avalanche effect of the algorithm is low [70]. Differential cryptanalysis attack is the same attack used to attack DES [70].

IoT uses Secure Force algorithm to encrypt its sensors [114]. Extant literature such as in Shujaat et al. [11] compared Secure Force (SF), DES and AES algorithms without modifying them [11]. SF algorithm is non-complexity algorithm for IoT. It is needed when space of installation is limited in some certain devices like sensors [11]. Shujaat et al. [11] did not give a new modification method to enhance the avalanche effect of these three standard algorithms; they just analysed them without modification or proposed method. SF 64, 128 and 192 gave the avalanche effect of 58.2%, 51.5500% and 45.7000% respectively [11]. Whereas the avalanche results for AES-128 was 44.9200% [11] and of DES-64 was 65.6300% [11].

IoT uses DES to encrypt its prototype [22]. It is already indicated that DES is vulnerable to brute force attack due to its smaller size of key (56 bits) [105]. Ibrahim et al. [115] compared DES Feistel Network (FN) and three types of DES Extended Feistel Network (EFN) (Type one, two and three, EFN is the process of multiplying FN several time in one algorithm). Ibrahim et al. [115] chose to run this experiment of EFN on DES instead of using standard DES. From their results analysis, they indicated that the more EFN used on DES, the more first eight rounds executed better avalanche effect [115]. When the number of rounds increase, the avalanche effect of DES with EFN became ineffective [115]. They also indicated that, using EFN method was the more expensive operation than using normal FN of DES.

IoT uses RC5 algorithm to secure its Mica2 hardware [116]. Kaliski-Yin in 1995 attacked RC5 algorithm using single half-round characteristics attack [117]. This attack is the same as differential cryptanalysis attack but instead of attacking all rounds of algorithms, half of algorithm's rounds are attack depending on the characteristics (weakness and strength) of algorithm [117]. Ali [118] used avalanche effect to improve the security of RC5 algorithm. He managed to reduce vulnerability of matching characteristics attack by increasing block size complexity from 2^{32} to 2^{256} and analysed their avalanche effects. His proposed method work successfully compared to the standard RC5 algorithm [118].

IoT uses Tiny Encryption Algorithm (TEA) to encrypt its Mica2 hardware [116]. Abdelhalim et al. [94] modified Tiny Encryption Algorithm (TEA) and named it Modified Tiny Encryption Algorithm (MTEA). They left shifted the TEA's register three times to modify TEA. Their main aim of using the Linear Feedback Shift Register (LFSR) was to overcome the security weakness of the TEA algorithm against attacks and to improve its avalanche effect [94]. Abdelhalim et al. [94] compared MTEA and TEA under the analysis of avalanche effect. The results showed that there was no big improvement of avalanche effect when MTEA and TEA were compared because they yielded almost the same results (that was almost 33% in average) of avalanche effect [94], and both MTEA and TEA had poor avalanche effect (that was less than 50%).

Dewangan et al. [40] modified AES by changing the form of plaintext and encryption key. They added a key stream generator to AES to improve the encryption performance. In their proposal, they mapped plaintext and encryption key in different binary codes before using them as the inputs of the AES algorithm. These binary codes were extracted from weighted and unweighted code. From their results, they showed that they managed to enhance avalanche effect of AES from 73% to 76% when one bit of key changes. On the side of plaintext changes, they showed an improvement from 76% to 80% [40]. Mandal et al [9] also used that method of using binary codes, but DES was chosen instead of AES. They got an average of avalanche effect from 44% to 64%, when plaintext changes in 5421 binary code. The key was mapped with gray code [9].

IoT uses Serpent algorithm to encrypt its information and data [104]. Wei et al. [104] indicated that there are strong attacks abilities used on Serpent algorithm. He even supported his statement by attacking Serpent algorithm using differential fault attack [104]. Aghajanzadeh et al. [137] tried to combine Serpent and RC4 to be one algorithm called RC4-Serpent to prevent

these strong attacks abilities mentioned above by Wei et al. [104]. He managed to enhance the avalanched effect of Serpent based on RC4-Serpent algorithm from 58% to 64%.

Maita et al. [119] performed a work to enhance security of algorithms where Pseudo Random Number Generator (PRNG) was used to increase complexity of the key generation of DES and AES. From their experimental results, when both DES and AES algorithms mixed with the PRNG, depicted the avalanche effect improvement of 36.3% average [119].

IoT uses Skipjack algorithm to secure its Mica2 hardware [116]. Biham et al. [120] attacked Skipjack algorithm using differential cryptanalysis attack. They managed to attack some few rounds of Skipjack algorithm. Their attack failed when they tried to attack a full 32 rounds of Skipjack [120]. Maram et al. [121] proposed a new modified Skipjack algorithm to enhance the avalanche effect compared to standard Skipjack algorithm. The main aim was to prevent attacks like differential cryptanalysis attack. They used dynamic key-dependent S-box instead of fixed S-box of standard Skipjack. The proposed algorithm produced better results of the avalanche effect that is from 76% to 98% [121].

From the above literature review, little has been done to test the avalanche effect using initial vector XORed with plaintext and final vector XORed with ciphertext. In this proposal we will use initial vector XORed with plaintext and final vector XORed with cipher text and test avalanche effect of all ten algorithms described in section 2.4. The vectors will be extracted from irrational digits of PI after digit 3. Proposed work is given by Figure 2.8.

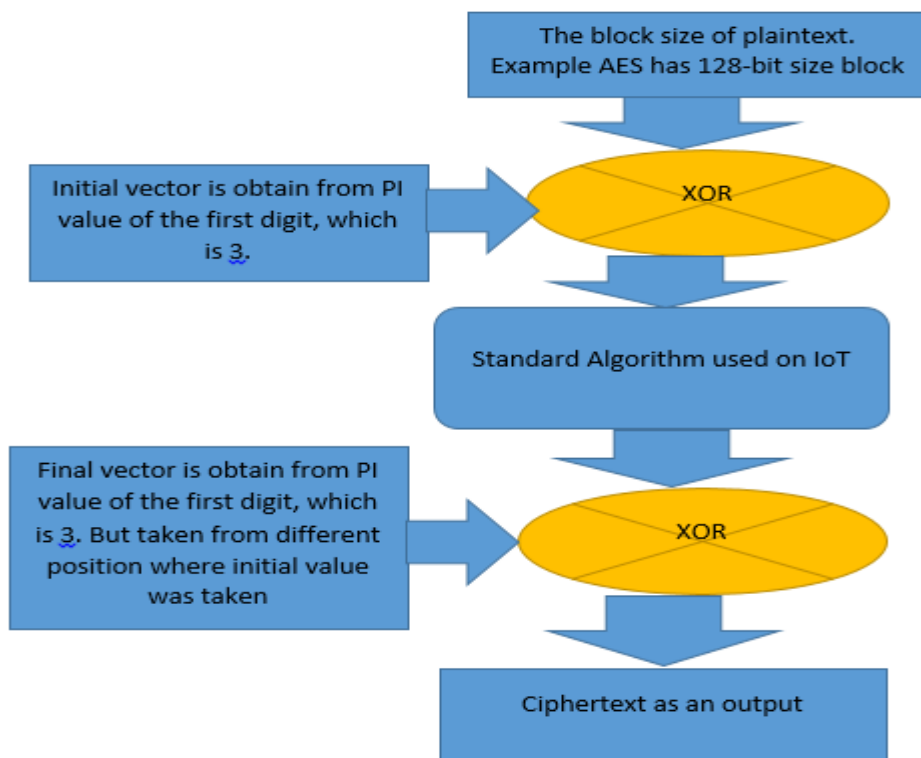


Figure 2.8: Proposed work diagram explaining how initial and final vectors are derived.

2.8 Chapter Summary

This chapter provided a literature review of IoT: its protocols, network layers, security, model of communications, privacy, trust, data confidentiality, some algorithms used on IoT, avalanche effect of cryptographic algorithm used on IoT, types of attacks on algorithms, encryption methods and techniques. Different types algorithms that are used over IoT were analysed. In the next chapter will discuss the research methodology related to our study. This includes introduction, source of initial and final vectors, PI methodology that is the overview of PI, methodology of study based on the avalanche effect, research design, experimental procedure and chapter summary.

CHAPTER 3: METHODOLOGY

3.1. Introduction

In this section, we present the methods used for this study. In this study, we used the simulation research based on the comparison and analysis of different cryptographic algorithms used on IoT. We selected ten cryptographic algorithms (defined in section 2.4) used on IoT and simulated them in C++. In theory, any programming language could be used to develop algorithm, but C++ is an authorised “official” language of cryptography used by Federal Information Processing Standards (FIPS), National Institute of Standards and Technology (NIST), and other cryptographers. The reason being that it is difficult to learn than other programming languages. Therefore, it is difficult to crack an algorithm written in C++ compared to others. The type of computer used is Hewlett-Packard Compaq Elite 8300 Convertible Minitower. The main reason for choosing these algorithms was to analyse their avalanche effects and compare them with the proposed method. The comparative method used to measure the avalanche effect was simulated using C++ programming language. The proposed method was to XOR the plaintext with initial vector and XOR ciphertext with the final vector in each algorithm mentioned in chapter 2. After that, we did an analysis of the avalanche effect on the proposed algorithms and compared them with the ones used on IoT. The design of the selected algorithms was based on plaintext key algorithms and ciphertext only. In the proposed model, we added two blocks: an initial and the final vector as compared to standard model. From Figure 3.1 and Figure 3.2, one can see that the blocks are not the same. The difference is in an initial vector and the final vector imposed on Figure 3.2 which is the proposed model of algorithms. Of paramount importance is the question of how and where an initial and the final vectors were taken or generated from? Also, of paramount importance, is why were they taken or generated from there? The explanation and answers are elaborated in section 3.2 and 3.3.

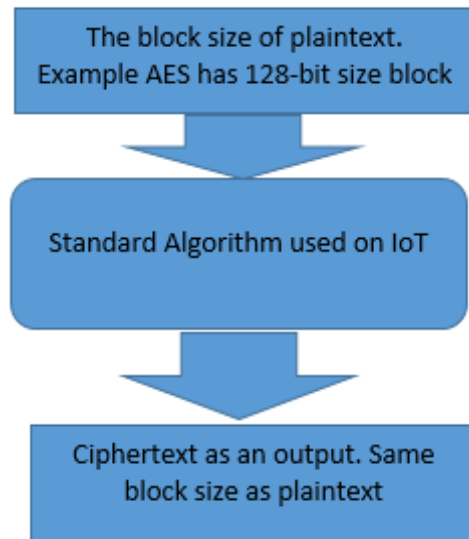


Figure 3.1: Standard model with an explanation of block sizes.

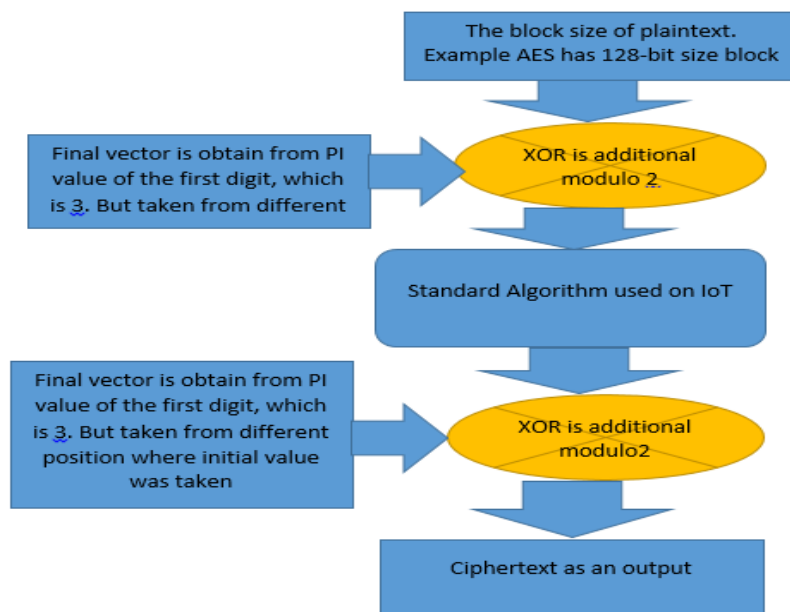


Figure 3.2: Proposed model of algorithms with an explanation of XOR operation.

3.2 Source Of Initial And Final Vector

To explain the questions posed in section 3.1, the values of an initial and the final vector were taken from the value of PI after the digit 3. Yang [122] generated 8366 hexadecimal (hex) digits string of the value PI after the digit 3. We used Yang's string to create an initial and the final vectors by extracting their values from two different positions of Yang's string . We must state here that the values of an initial and the final vector are not the same because they are extracted from different position of 8366 hex digits string. Yang's string has 8366 positions. Refer to appendix 2 for an overview of Yang's string. In appendix 2 we decided to indicate the selected value of an initial vector highlighted in red and the final vector highlighted in green for flexibility of study. Refer to appendix 2.

3.3. PI Methodology: The Overview

The value of PI after the first digit which is 3 was used due to its characteristics. The value of PI in hexadecimal notation is calculated by using Yang's string [122] and it has 8366 positions. If we check the value of PI, from the digits after the first digit 3, we can see that this value is an irrational number. Refer to appendix 2. An irrational number is the number calculated from ratios (or fractions) of integers [123]. When the fraction or ratio of distances of two line segments is an irrational number, then line segments are also irrational, meaning that they have no measure in common, that is, there is no distance, no matter how short, that could not be used to determine the distances between the two given line segments as integer that can multiply itself [122]. The other characteristic is that even if one tries to do a sequence and series calculation on the value of PI, the PI value will still not yield the pattern [123]. Therefore it would be hard for the intruder to attack or crack the algorithm which is mixed with PI values. It is deduced that it will be difficult to intrude or attack a cryptographic algorithm using PI values or any irrational number because of the characteristics of irrational numbers and way the algorithms is coded [122]. We took the advantage of PI characteristics from the above explanation as an advantage and that is where an initial and the final vector values are derived or extracted from.

3.4 Methodology of Study Based On the Avalanche Effect

In this section, we discuss an overview of the methodology of avalanche effect: how to calculate the avalanche effect percentage, the flowchart of avalanche effect, formula to calculate avalanche effect and method to calculate avalanche effect.

3.4.1 Need to Calculate the Avalanche Effect

The avalanche effect is most desirable property for most of the cryptographic algorithms. If an input is changed slightly (for example flipping a single input bit) the output must change excessively (more than 50% the output bits should flip) [7]. One main reason for the avalanche effect is that by flipping only one bit of the input, if there is large change in the output, then it is harder to perform an attack (intrusion or hacking) on the cryptographic algorithm [35]. Present literature has shown that an algorithm with high avalanche effect is a strong algorithm [7].

3.4.2. Method to Calculate the Avalanche Effect

The formula to calculate the avalanche effect is defined [7], [35] as follows:

$$\text{Avalanche effect} = \frac{\text{Average number of flipped bits in ciphertext}}{\text{Number of the bit in ciphertext}} * 100\% \quad (3.1)$$

In this study we fixed the key and flipped the plaintext bit from left to right and then we checked the number or the amount of bits that changed in ciphertext and compared it to the un-flipped ones. In the second experiment we fixed the plaintext and flipped one bit of key from left to right and then checked the number of bits that would have changed in ciphertext. The bits were flipped according to the size of input. Refer to appendix 1 for mathematical procedure, or refer to Figure 3.3 for flowchart on how to calculate avalanche effect of algorithm like AES. AES is used as an example.

In accordance with the above, we will give an example of the experiment by using the AES algorithm. This is basically to show how the avalanche effect is calculated using the flowchart

given in Figure 3.3. AES algorithm is 128 block cipher, meaning that the size of ciphertext and plaintext are both 128 bits. Refer to appendix 1 for mathematical procedure, or refer to Figure 3.3 for flowchart on how to calculate avalanche effect of algorithm like AES. AES is used as an example.

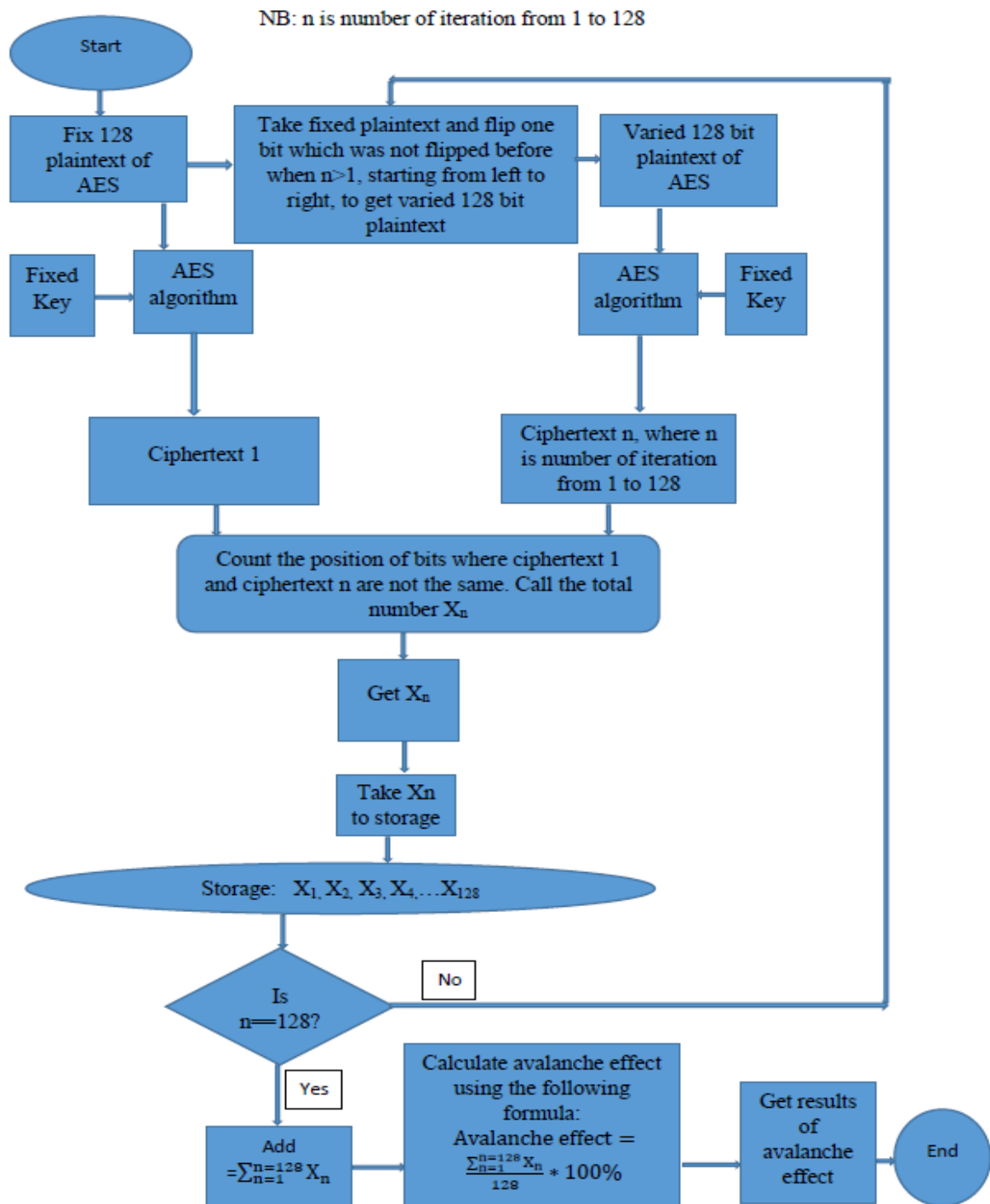


Figure 3.3: Flowchart diagram on how to calculate avalanche effect of algorithm like AES.

For more detailed explanation on how to calculate avalanche effect step by step mathematically. Refer to appendix 1. Every aspect discussed in this section is mathematically demonstrated on appendix 1.

3.5. Research Design

In this study, the research methodology to be used will be quantitative. It will be more of experimental research method that will be conducted as follows: (1) Searching and selecting existing cryptographic algorithms for authentication and encryption in the context of IoT; (2) Analyzing the avalanche effect of selected cryptographic algorithms from step one if they really give efficient security on IoT based on avalanche effect; (3) Improving their avalanche effect by designing mathematical model (where initial vector will be XORed with the plaintext and final vector XORed with ciphertext) that gives more confusion and diffusion to intruder as indicated in Figure 3.4; (4) Testing the proposed model if it really enhances the avalanche effect on each and every algorithm selected in step one; (5) Giving the proposed future work on how to enhance security on IoT and give the conclusion. (6) And finally publish at least one paper from this study by IEEE.

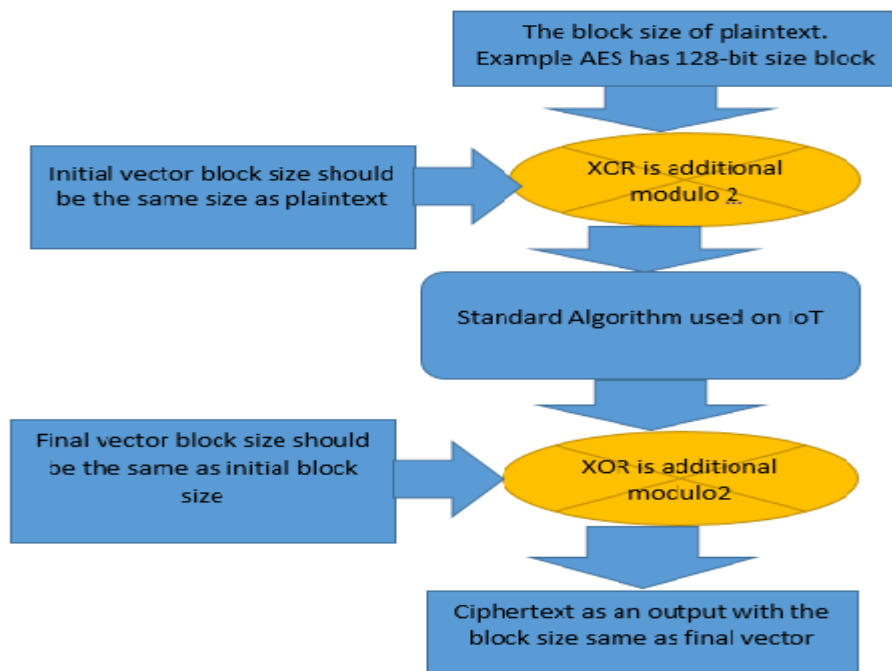


Figure 3.4: Proposed model with the explanations of block sizes of each building block.

3.6. Experimental Procedure

The experimental procedure was to determine how secure the different algorithms used on IoT and compare with each other. In this research, ten different algorithms were selected, because they are widely used on IoT. See Table 3, which is comparing the performance of each other based on different parameters such as the avalanche effect, time and speed.

Table 3.1: Algorithms and their usage within IoT.

Name of Algorithm	How algorithm is used within the internet of things
AES	IoT uses AES to secure its sensors and contactless smart cards [22], [15].
Blowfish	IoT uses Blowfish to secure its application and network layer of IoT [22], [124].
Camellia	IoT uses Camellia to secure its prototypes [15].
Cast-128	IoT uses Cast to secure its prototypes [23].
Clefia	IoT uses Clefia to secure its health-care devices [15].
DES	IoT uses DES to secure most of its devices. It is mostly used on IoT [22].
MMB	IoT uses MMB to secure its software's application [125].
RC5	IoT uses RC5 algorithm to secure its Mica2 hardware (base station of IoT) [126].
Serpent	IoT uses Serpent to secure its sensors [102].
Skipjack	IoT uses Skipjack algorithm to secure Mica2 hardware (base station of IoT) [126].

In this simulation procedure, an algorithm is classified as standard if it is not modified by the proposed methodology. That is if it is taken and analysed as it is from the original developers.

An algorithm is classified as proposed algorithm if it is modified by introducing our methodology of the initial and final vectors on it.

The screenshot below, which is Figure 3.5, depicts the output of C++ code simulation example when avalanche effect and time were executed by our C++ code.

```

68.000000
0123456789abcdeffedcba9876543200
One changed
55.000000
0123456789abcdeffedcba9876543218
One changed
65.000000
0123456789abcdeffedcba9876543214
One changed
68.000000
0123456789abcdeffedcba9876543212
One changed
64.000000
0123456789abcdeffedcba9876543211
One changed
63.000000

Average of bits changed out of 128 = 62.804688

Avalanche Effect in Percentage= ((Average of bits changed out of 128 )/128) x 100
= 49.066162

-----
Process exited after 0.1158 seconds with return value 0
Press any key to continue
  
```

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to encrypt	Speed taken to encrypt
Standard AES	NO	NO	49.066162 5%	0.1158 sec	1105.35 bit/sec

Figure 3.5: Example of reading simulation and table

The other two main characteristics that differentiates one encryption algorithm from another is its ability to encrypt data when its time and speed are also measured [127]. We calculated the time taken to perform avalanche effect on each and every algorithm. The speed of algorithm was calculated as follows:

We flipped one bit from left to right until to the end, one bit at the time. For example if 128 bits algorithm is tested, it means that the encryption process was conducted 128 times. It therefore suffices to say that $128 \times 128 = 16384$ bits were encrypted during the avalanche effect. We can then calculate the speed as follows:

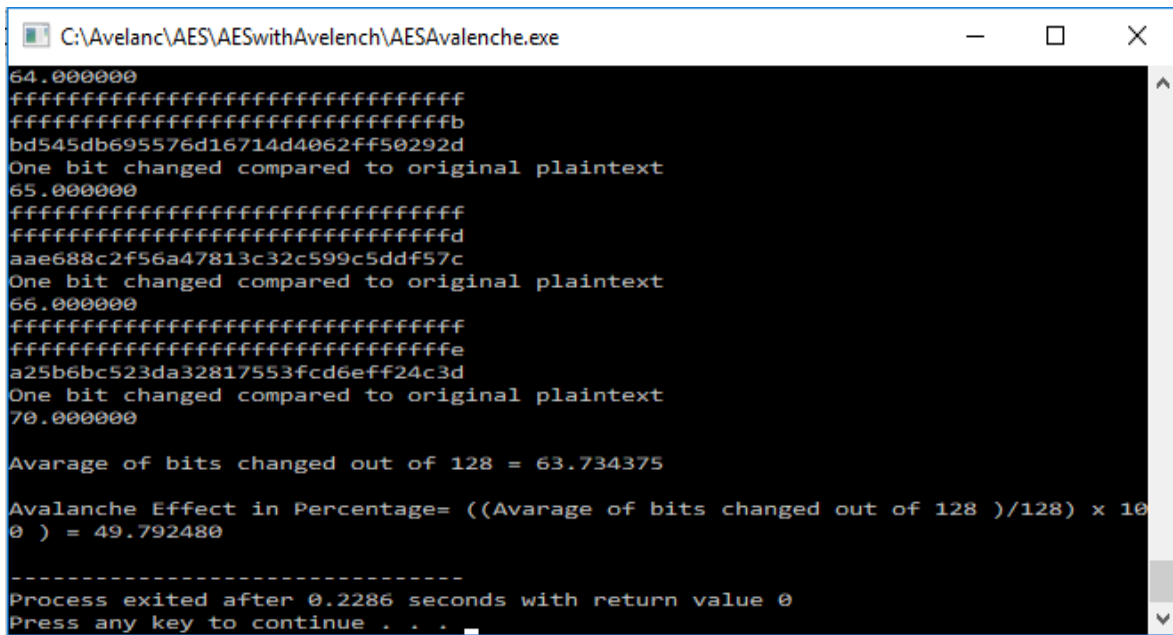
$$Speed = \frac{\text{Number (size) of bits encrypted during avalanche effect}}{\text{Time taken to encrypt size of bits during avalanche effect}} \quad (3.2)$$

3.6.1. Simulation 1: Testing of Avalanche Effect on AES

When a cryptographic algorithm is published to the public domain by their developers, it is published with its test vectors. Test vectors are the sets of inputs and outputs provided to user of the system (in this study the system is cryptographic algorithm) in order to test that algorithm. In cryptography, test vectors are used for algorithm testing, verification and validation.

As we mentioned earlier that IoT uses AES algorithm to secure its smart cards [22]. We studied AES algorithm from [15], [40], [119] and analysed how it works from [56], [64], [58], [60], [6]. We programed AES algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our AES algorithm is encrypting and decrypting according to the specification of its origin (developers), we used test vectors found in [15, p. 35]. Then we called it a standard AES algorithm because it gave us the same test vector defined in [15, p. 35]. After that, we modified it using an initial and the final vectors as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it the proposed AES algorithm. The proposed AES algorithm's test vectors became totally different to the test vector found in [15, p. 35]. Therefore, it is a different algorithm compared to standard AES, with its own different test vectors. From these two (standard and proposed AES) algorithms, we calculated their avalanche effects when key was fixed and plaintext was varied, and vice versa. AES has two inputs (plaintext and key). AES algorithm uses plaintext of 128 bits long as a first input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of the avalanche effect's procedure. Again, AES algorithm uses key of 128 bits long as a second input. We varied each key bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of avalanche effect. We even calculated their speed when key was fixed and plaintext was varied, and vice versa. We finally had four codes of AES algorithms: (1) Standard AES when key varies, (2) Standard AES when plaintext varies, (3) Proposed AES when key varies and (4) Proposed AES when plaintext varies. Below, in Figure 3.6 to Figure 3.9, we present the executable simulation screenshots of

four different AES algorithms mentioned above. Figure 3.6 presents simulation of avalanche effect on standard AES when plaintext is varied with the value of 49.79248 percent.



```
C:\Avelanc\AES\AESwithAvelench\AESAvalanche.exe
64.000000
ffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffb
bd545db695576d16714d4062ff50292d
One bit changed compared to original plaintext
65.000000
ffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffd
aae688c2f56a47813c32c599c5ddf57c
One bit changed compared to original plaintext
66.000000
ffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffe
a25b6bc523da32817553fcd6eff24c3d
One bit changed compared to original plaintext
70.000000

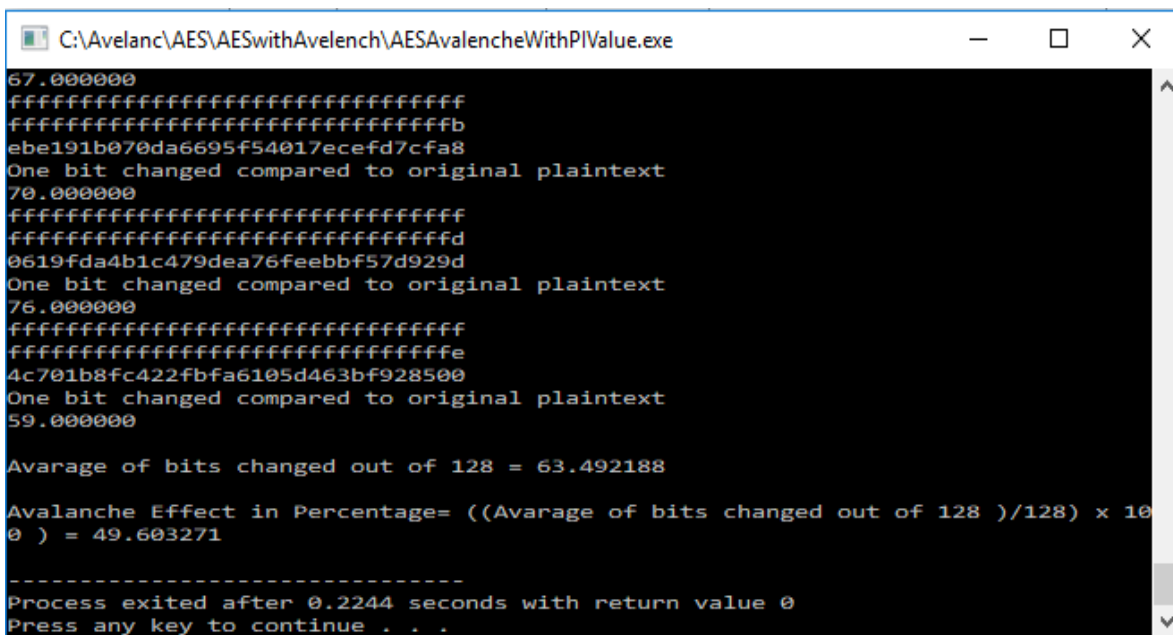
Avarage of bits changed out of 128 = 63.734375

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 49.792480

-----
Process exited after 0.2286 seconds with return value 0
Press any key to continue . . .
```

Figure 3.6: Simulation of avalanche effect on standard AES when plaintext is varied.

Simulation of avalanche effect on proposed AES was conducted when plaintext was varied. Figure 3.7 depicts the results of 49.6033% of avalanche effect when plaintext of proposed AES was varied.



```
C:\Avelanc\AES\AESwithAvelench\AESAvalancheWithPIValue.exe
67.000000
ffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffb
ebe191b070da6695f54017ecef7cfa8
One bit changed compared to original plaintext
70.000000
ffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffd
0619fda4b1c479dea76feebbf57d929d
One bit changed compared to original plaintext
76.000000
ffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffe
4c701b8fc422fbfa6105d463bf928500
One bit changed compared to original plaintext
59.000000

Avarage of bits changed out of 128 = 63.492188

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 49.603271

-----
Process exited after 0.2244 seconds with return value 0
Press any key to continue . . .
```

Figure 3.7: Simulation of avalanche effect on proposed AES when plaintext is varied.

Simulation of avalanche effect on standard AES was conducted when key was varied. Figure 3.8 depicts the results of 49.0662% of avalanche effect when key of standard AES was varied.

```

C:\Avelanc\AES\AESwithKEy\AESAvalenche.exe
68.000000
0123456789abcdeffedcba9876543200
One changed
55.000000
0123456789abcdeffedcba9876543218
One changed
65.000000
0123456789abcdeffedcba9876543214
One changed
68.000000
0123456789abcdeffedcba9876543212
One changed
64.000000
0123456789abcdeffedcba9876543211
One changed
63.000000

Avarage of bits changed out of 128 = 62.804688

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 100
) = 49.066162

-----
Process exited after 0.1158 seconds with return value 0
Press any key to continue . . .

```

Figure 3.8: Simulation of avalanche effect on standard AES when key is varied.

Simulation of avalanche effect on proposed AES was conducted when key was varied. Figure 3.9 depicts the results of 49.9390 % of avalanche effect when key of proposed AES was varied.

```

C:\Avelanc\AES\AESwithKEy\AESAvalencheWithPIValue.exe
62.000000
0123456789abcdeffedcba9876543200
One changed
57.000000
0123456789abcdeffedcba9876543218
One changed
61.000000
0123456789abcdeffedcba9876543214
One changed
65.000000
0123456789abcdeffedcba9876543212
One changed
66.000000
0123456789abcdeffedcba9876543211
One changed
71.000000

Avarage of bits changed out of 128 = 63.921875

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 100
) = 49.938965

-----
Process exited after 0.118 seconds with return value 0
Press any key to continue . . .

```

Figure 3.9 Simulation of avalanche effect on proposed AES when key is varied.

3.6.2. Simulation 2: Testing of Avalanche Effect on Blowfish

As we mentioned earlier that IoT uses Blowfish algorithm to secure its applications and network layer [124], [22]. We studied Blowfish algorithm from [10], [128], [61] and analysed how it works from [128], [61], [129], [124]. We programmed Blowfish algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our Blowfish algorithm is encrypting and decrypting according to the specification of its origin, we used test vectors from Blowfish's developers found in [130]. Then we called it a standard Blowfish algorithm because it gave us the same test vector defined in [130]. After that, we modified it using an initial and the final vectors as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it the proposed Blowfish algorithm. From these two (standard and proposed Blowfish) algorithms, we calculated their avalanche effects when key was fixed and plaintext was varied, and vice versa. Blowfish has two inputs (plaintext and key). Blowfish algorithm uses plaintext of 128 bits long as a first input like AES. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of the avalanche effect's calculation. Again, Blowfish algorithm uses a key of 128 bits long as a second input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of avalanche effect. We even calculated their speeds when key was fixed and plaintext was varied, and vice versa. We finally had four codes of Blowfish: (1) Standard Blowfish when key varies, (2) Standard Blowfish when plaintext varies, (3) Proposed Blowfish when key varies and (4) Proposed Blowfish when plaintext varies. Below (in Figure 3.10 to Figure 3.13), we present the executable simulation screenshots of four different Blowfish algorithms mentioned above. Simulation of avalanche effect on standard Blow was conducted when plaintext was varied. Figure 3.10 depicts the results of 50.5615% of avalanche effect when plaintext of standard Blowfish was varied.

```

C:\Avelanc\Blowfish\BlowFishWithAvalench\BlowfishWithOutVectors.exe
38.000000
ffffffffffffffffffef
One changed
32.000000
ffffffffffffffffff7
One changed
29.000000
ffffffffffffffffffb
One changed
30.000000
ffffffffffffffffffd
One changed
35.000000
ffffffffffffffffffe
One changed
29.000000

Avarage of bits changed out of 64 = 32.359375

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 50.561523

-----
Process exited after 0.06141 seconds with return value 0
Press any key to continue . . .

```

Figure 3.10: Simulation of avalanche effect on standard Blowfish when plaintext is varied.

Simulation of avalanche effect on proposed Blow was conducted when plaintext was varied. Figure 3.11 depicts the results of 48.3398% of avalanche effect when plaintext of proposed Blowfish was varied.

```

C:\Avelanc\Blowfish\BlowFishWithAvalench\BlowfishWithVectors.exe
29.000000
ffffffffffffffffffef
One changed
24.000000
ffffffffffffffffff7
One changed
25.000000
ffffffffffffffffffb
One changed
34.000000
ffffffffffffffffffd
One changed
32.000000
ffffffffffffffffffe
One changed
34.000000

Avarage of bits changed out of 64 = 30.937500

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 48.339844

-----
Process exited after 0.0611 seconds with return value 0
Press any key to continue . . .

```

Figure 3.11: Simulation of avalanche effect on proposed Blowfish when plaintext is varied.

Simulation of avalanche effect on standard Blow was conducted when key was varied. Figure 3.12 depicts the results of 50.4517% of avalanche effect when key of standard Blowfish was varied.

```
C:\Avelanc\Blowfish\BlowFishWithAvalencKey\BlowfishWithOutVectors.exe
39.000000
0123456789abcdeffedcba9876543200
One changed
39.000000
0123456789abcdeffedcba9876543218
One changed
27.000000
0123456789abcdeffedcba9876543214
One changed
31.000000
0123456789abcdeffedcba9876543212
One changed
34.000000
0123456789abcdeffedcba9876543211
One changed
33.000000

Avarage of bits changed out of 64 = 32.289063

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 50.451660

-----
Process exited after 0.1233 seconds with return value 0
Press any key to continue . . .
```

Figure 3.12: Simulation of avalanche effect on standard Blowfish when key is varied.

Simulation of avalanche effect on proposed Blow was conducted when key was varied. Figure 3.13 depicts the results of 49.9878% of avalanche effect when key of proposed Blowfish was varied.

```
C:\Avelanc\Blowfish\BlowFishWithAvalencKey\BlowfishWithVectors.exe
31.000000
0123456789abcdeffedcba9876543200
One changed
34.000000
0123456789abcdeffedcba9876543218
One changed
36.000000
0123456789abcdeffedcba9876543214
One changed
35.000000
0123456789abcdeffedcba9876543212
One changed
37.000000
0123456789abcdeffedcba9876543211
One changed
31.000000

Avarage of bits changed out of 64 = 31.992188

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 49.987793

-----
Process exited after 0.1223 seconds with return value 0
Press any key to continue . . .
```

Figure 3.13: Simulation of avalanche effect on proposed Blowfish when key is varied.

3.6.3. Simulation 3: Testing of Avalanche Effect on Camellia

As we mentioned earlier that IoT uses Camellia algorithm to secure its medical data systems [107]. We studied Camellia algorithm from [65], [64] and analysed how it works from [63], [79], [113]. We programmed Camellia algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our Camellia algorithm is encrypting and decrypting according to the specification of its origin, we used test vectors from [79, p. 22]. Then we called it a standard Camellia algorithm because it gave us the same test vector defined in [79, p. 22]. After that, we modified it using an initial and the final vector as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it the proposed Camellia algorithm. From these two (standard and proposed Camellia) algorithms, we calculated their avalanche effects when key was fixed and plaintext was varied, and vice versa. Camellia has two inputs (plaintext and key). Camellia algorithm uses plaintext of 128 bits long as a first input like AES. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of the avalanche effect's procedure. Again, Camellia algorithm uses key of 192 bits long as a second input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of avalanche effect. We even calculated their speeds when key was fixed and plaintext was varied, and vice versa. We finally had four codes of Camellia: (1) Standard Camellia when key varies, (2) Standard Camellia when plaintext varies, (3) Proposed Camellia when key varies and (4) Proposed Camellia when plaintext varies. Below (in Figure 3.14 to Figure 3.17) we present the executable simulation screenshots of four different Camellia algorithms mentioned above. Simulation of avalanche effect on standard Camellia was conducted when plaintext was varied. Figure 3.14 depicts the results of 49.4690% of avalanche effect when plaintext of standard Camellia was varied.

```

C:\Avelanc\Camellia\CamelliaWithVector\CamelliaWithOutVectors.exe
74.000000
fffffffffffffffffffffffffffffffffb
One changed
66.000000

fffffffffffffffffffffffffffffd
One changed
61.000000

fffffffffffffffffffffffffffffe
One changed
66.000000

Avarage of bits changed out of 128 = 63.320313

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 49.468994

-----
Process exited after 0.1576 seconds with return value 0
Press any key to continue . . .

```

Figure 3.14: Simulation of avalanche effect on standard Camellia when plaintext is varied.

Simulation of avalanche effect on proposed Camellia was conducted when plaintext was varied. Figure 3.15 depicts the results of 50.0977% of avalanche effect when plaintext of proposed Camellia was varied.

```

C:\Avelanc\Camellia\CamelliaWithVector\CamelliaWithVectors.exe
71.000000
fffffffffffffffffffffffffffffffffb
One changed
61.000000

fffffffffffffffffffffffffffffd
One changed
77.000000

fffffffffffffffffffffffffffffe
One changed
63.000000

Avarage of bits changed out of 128 = 64.125000

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 50.097656

-----
Process exited after 0.1217 seconds with return value 0
Press any key to continue . . .

```

Figure 3.15: Simulation of avalanche effect on proposed Camellia when plaintext is varied.

Simulation of avalanche effect on standard Camellia was conducted when key was varied. Figure 3.16 depicts the results of 49.6094% of avalanche effect when key of standard Camellia was varied.


```
C:\Avelanc\Camellia\CamelliaWithKey\CamelliaWithOutVectors.exe
61.000000
0011223344556677fedcba98765432100011223344556673

One changed
68.000000
0011223344556673fedcba98765432100011223344556675

One changed
62.000000
0011223344556675fedcba98765432100011223344556676

One changed
67.000000

Avarage of bits changed out of 192 = 63.500000

Avalanche Effect in Percentage= ((Avarage of bits changed out of 192 )/128) x 100
) = 49.609375

-----
Process exited after 0.1829 seconds with return value 0
Press any key to continue . . .
```

Figure 3.16: Simulation of avalanche effect on standard Camellia when key is varied.

Simulation of avalanche effect on proposed Camellia was conducted when key was varied. Figure 3.17 depicts the results of 49.8983% of avalanche effect when key of proposed Camellia was varied.

```
C:\Avelanc\Camellia\CamelliaWithKey\CamelliaWithVectors.exe
67.000000
0011223344556677fedcba98765432100011223344556673

One changed
66.000000
0011223344556673fedcba98765432100011223344556675

One changed
68.000000
0011223344556675fedcba98765432100011223344556676

One changed
66.000000

Avarage of bits changed out of 192 = 63.869793

Avalanche Effect in Percentage= ((Avarage of bits changed out of 192 )/128) x 100
) = 49.898277

-----
Process exited after 0.1859 seconds with return value 0
Press any key to continue . . .
```

Figure 3.17: Simulation of avalanche effect on proposed Camellia when key is varied.

3.6.4. Simulation 4: Testing of Avalanche Effect on Cast-128

As we mentioned earlier that IoT uses Cast-128 algorithm to secure its prototypes [15], [23]. We studied Cast-128 algorithm from [69], [64], [68] and analysed how it works from [68], [110], [111]. We programed Cast-128 algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our Cast-128 algorithm is encrypting and decrypting according to the specification of its origin, we used test vectors found in [68, p. 15]. Then we called it a standard Cast-128 algorithm because it gave us the same test vector defined in [68, p. 15]. After that, we modified it using an initial and the final vector as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it the proposed Cast-128 algorithm. From these two (standard and proposed Cast-128) algorithms, we calculated their avalanche effect when key was fixed and plaintext was varied, and vice versa. Cast-128 has two inputs (plaintext and key). Camellia algorithm uses plaintext of 128 bits long as a first input like AES. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of avalanche effect. Again, Cast-128 algorithm uses key of 128 bits long as a second input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of avalanche effect's calculations. We even calculated their speeds when key was fixed and plaintext was varied, and vice versa. We finally had four codes of Cast-128: (1) Standard Cast-128 when key varies, (2) Standard Cast-128 when plaintext varies, (3) Proposed Cast-128 when key varies and (4) Proposed Cast-128 when plaintext varies. Below (in Figure 3.18 to Figure 3.21) we present executable simulation screenshots of four different Cast-128 algorithms mentioned above. Simulation of avalanche effect on standard Cast-128 was conducted when plaintext was varied. Figure 3.18 depicts the results of 48.8281% of avalanche effect when plaintext of standard Cast-128 was varied.

```

C:\Avelanc\CAST-128\CastWithVectors\Cast.exe
38.000000
ffffffffffffffffffef
One changed
32.000000
ffffffffffffffffff7
One changed
30.000000
ffffffffffffffffffb
One changed
31.000000
ffffffffffffffffffd
One changed
29.000000
ffffffffffffffffffe
One changed
34.000000

Avarage of bits changed out of 64 = 31.250000

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 48.828125

-----
Process exited after 0.06299 seconds with return value 0
Press any key to continue . . .

```

Figure 3.18: Simulation of avalanche effect on standard Cast-128 when plaintext is varied.

Simulation of avalanche effect on proposed Cast-128 was conducted when plaintext was varied. Figure 3.19 depicts the results of 48.3164% of avalanche effect when plaintext of standard Cast-128 was varied.

```

C:\Avelanc\CAST-128\CastWithVectors\CastWithVectors.exe
29.000000
FFFFFFFFFFFFFFFFffffef
One changed
43.000000
FFFFFFFFFFFFFFFFffff7
One changed
37.000000
FFFFFFFFFFFFFFFFffffb
One changed
23.000000
FFFFFFFFFFFFFFFFffffd
One changed
33.000000
FFFFFFFFFFFFFFFFffffe
One changed
29.000000

Avarage of bits changed out of 64 = 31.562500

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 49.316406

-----
Process exited after 0.096 seconds with return value 0
Press any key to continue . . .

```

Figure 3.19: Simulation of avalanche effect on proposed Cast-128 when plaintext is varied.

Simulation of avalanche effect on standard Cast-128 was conducted when key was varied. Figure 3.20 depicts the results of 50.1221% of avalanche effect when key of standard Cast-128 was varied.

```
C:\Avelanc\CAST-128\CastWithVectorsKey\Cast.exe
31.000000
0123456712345678234567893456788a
One changed
38.000000
01234567123456782345678934567892
One changed
31.000000
0123456712345678234567893456789e
One changed
36.000000
01234567123456782345678934567898
One changed
39.000000
0123456712345678234567893456789b
One changed
31.000000

Avarage of bits changed out of 128 = 32.078125

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/64) x 100
) = 50.122070

-----
Process exited after 0.1239 seconds with return value 0
Press any key to continue . . .
```

Figure 3.20: Simulation of avalanche effect on standard Cast-128 when key is varied.

Simulation of avalanche effect on proposed Cast-128 was conducted when key was varied. Figure 3.21 depicts the results of 50.1709% of avalanche effect when key of proposed Cast-128 was varied.

```
C:\Avelanc\CAST-128\CastWithVectorsKey\CastWithVectors.exe
34.000000
0123456712345678234567893456788a
One changed
28.000000
01234567123456782345678934567892
One changed
27.000000
0123456712345678234567893456789e
One changed
36.000000
01234567123456782345678934567898
One changed
35.000000
0123456712345678234567893456789b
One changed
34.000000

Avarage of bits changed out of 128 = 32.109375

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/64) x 100
) = 50.170898

-----
Process exited after 0.1251 seconds with return value 0
Press any key to continue . . .
```

Figure 3.21: Simulation of avalanche effect on proposed Cast-128 when key is varied.

3.6.5. Simulation 5: Testing of Avalanche Effect on Clefia

As we mentioned earlier that IoT uses Clefia algorithm to secure its Radio-Frequency Identification (RFID) [112]. We studied Clefia algorithm from [70], [72], [43] and analysed how it works from [51], [113], [131]. We programmed Clefia algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our Clefia algorithm is encrypting and decrypting according to the specification of its origin, we used test vectors found in [131, p. 29]. Then we called it a standard Clefia algorithm because it gave us the same test vector defined in [131, p. 29]. After that, we modified it using an initial and the final vector as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it the proposed Clefia algorithm. From these two (standard and proposed Clefia) algorithms, we calculated their avalanche effects when key was fixed and plaintext was varied, and vice versa. Clefia has two inputs (plaintext and key). Clefia algorithm uses plaintext of 128 bits long as a first input like AES. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of avalanche effect's procedure. Again, Clefia algorithm uses key of 128 bits long as a second input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of the avalanche effect's calculation. We even calculated their speeds when key was fixed and plaintext was varied, and vice versa. We finally had four codes of Clefia: (1) Standard Clefia when key varies, (2) Standard Clefia when plaintext varies, (3) Proposed Clefia when key varies and (4) Proposed Clefia when plaintext varies. Below (in Figure 3.22 to Figure 3.25) we present the executable simulation screenshots of four different Clefia algorithms mentioned above. Simulation of avalanche effect on standard Clefia was conducted when plaintext was varied. Figure 3.22 depicts the results of 50.2807% of avalanche effect when plaintext of standard Clefia was varied.

```

C:\Avelanc\CLEFIA\CLEFIAWithAvelanch\CLEFIA.exe
63.000000
ffffffffffffffffffffffffffffef
One changed
67.000000
ffffffffffffffffffffffffffff7
One changed
70.000000
ffffffffffffffffffffffffffffb
One changed
50.000000
ffffffffffffffffffffffffffffd
One changed
64.000000
ffffffffffffffffffffffffffffe
One changed
69.000000

Avarage of bits changed out of 128 = 64.359375

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 50.280762

-----
Process exited after 0.1177 seconds with return value 0
Press any key to continue . . .

```

Figure 3.22: Simulation of avalanche effect on standard Clefia when plaintext is varied.

Simulation of avalanche effect on proposed Clefia was conducted when plaintext was varied. Figure 3.23 depicts the results of 49.8230% of avalanche effect when plaintext of proposed Clefia was varied.

```

C:\Avelanc\CLEFIA\CLEFIAWithAvelanch\CLEFIAWithVectors.exe
68.000000
ffffffffffffffffffffffffffffef
One changed
45.000000
ffffffffffffffffffffffffffff7
One changed
67.000000
ffffffffffffffffffffffffffffb
One changed
67.000000
ffffffffffffffffffffffffffffd
One changed
65.000000
ffffffffffffffffffffffffffffe
One changed
61.000000

Avarage of bits changed out of 128 = 63.773438

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 49.822998

-----
Process exited after 0.1218 seconds with return value 0
Press any key to continue . . .

```

Figure 3.23: Simulation of avalanche effect on proposed Clefia when plaintext is varied.

Simulation of avalanche effect on standard Clefia was conducted when key was varied. Figure 3.24 depicts the results of 49.9023% of avalanche effect when key of standard Clefia was varied.

```

C:\Avelanc\CLEFIA\CLEFIWithAvelanchKey\CLEFIA.exe
67.000000
0123456789abcdeffedcba9876543200
One changed
63.000000
0123456789abcdeffedcba9876543218
One changed
63.000000
0123456789abcdeffedcba9876543214
One changed
60.000000
0123456789abcdeffedcba9876543212
One changed
54.000000
0123456789abcdeffedcba9876543211
One changed
63.000000

Avarage of bits changed out of 128 = 63.875000

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 100
= 49.902344

-----
Process exited after 0.1455 seconds with return value 0
Press any key to continue . . .

```

Figure 3.24: Simulation of avalanche effect on standard Clefia when key is varied.

Simulation of avalanche effect on proposed Clefia was conducted when key was varied. Figure 3.25 depicts the results of 50.1587% of avalanche effect when key of proposed Clefia was varied.

```

C:\Avelanc\CLEFIA\CLEFIWithAvelanchKey\CLEFIWithVectors.exe
66.000000
0123456789abcdeffedcba9876543200
One changed
57.000000
0123456789abcdeffedcba9876543218
One changed
66.000000
0123456789abcdeffedcba9876543214
One changed
53.000000
0123456789abcdeffedcba9876543212
One changed
63.000000
0123456789abcdeffedcba9876543211
One changed
72.000000

Avarage of bits changed out of 128 = 64.203125

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 100
= 50.158691

-----
Process exited after 0.1219 seconds with return value 0
Press any key to continue . . .

```

Figure 3.25: Simulation of avalanche effect on proposed Clefia when key is varied.

3.6.6. Simulation 6: Testing of Avalanche Effect on DES

As we mentioned earlier that IoT uses DES algorithm to secure its prototypes [22]. We studied DES algorithm from [9], [119], [64] and analysed how it works from [74], [76], [99], [103]. We programed DES algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our DES algorithm is encrypting and decrypting according to the specification of its origin, we used test vectors found in [132]. Then we called it a standard DES algorithm because it gave us the same test vector defined in [132]. After that, we modified it using an initial and the final vector as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it the proposed DES algorithm. From these two (standard and proposed DES) algorithms, we calculated their avalanche effects when key was fixed and plaintext was varied, and vice versa. DES has two inputs (plaintext and key). DES algorithm uses plaintext of 64 bits long as a first input unlike AES. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of avalanche effect's procedure. Again, DES algorithm uses key of 56 bits long as a second input unlike AES. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of avalanche effect. We even calculated their speeds when key was fixed and plaintext was varied, and vice versa. We finally had four codes of DES: (1) Standard DES when key varies, (2) Standard DES when plaintext varies, (3) Proposed DES when key varies and (4) Proposed DES when plaintext varies. Below (in Figure 3.26 to Figure 3.29) we present the executable simulation screenshots of four different DES algorithms mentioned above. Simulation of avalanche effect on standard DES was conducted when plaintext was varied. Figure 3.26 depicts the results of 62.8662% of avalanche effect when plaintext of standard DES was varied.


```

C:\Avelanc\DES\DESWithAvalanche\des.exe
40.000000
ffffffffffffffef
One changed
44.000000
ffffffffffffff7
One changed
45.000000
ffffffffffffffb
One changed
46.000000
ffffffffffffffd
One changed
42.000000
ffffffffffffffe
One changed
43.000000

Avarage of bits changed out of 64 = 40.234375

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 62.866211

-----
Process exited after 0.0585 seconds with return value 0
Press any key to continue . . .

```

Figure 3.26: Simulation of avalanche effect on standard DES when plaintext is varied.

Simulation of avalanche effect on proposed DES was conducted when plaintext was varied. Figure 3.27 depicts the results of 58.8379% of avalanche effect when plaintext of proposed DES was varied.

```

C:\Avelanc\DES\DESWithAvalanche\DESWithAvalanche.exe
32.000000
ffffffffffffffef
One changed
38.000000
ffffffffffffff7
One changed
32.000000
ffffffffffffffb
One changed
32.000000
ffffffffffffffd
One changed
34.000000
ffffffffffffffe
One changed
45.000000

Avarage of bits changed out of 64 = 37.656250

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 58.837891

-----
Process exited after 0.05773 seconds with return value 0
Press any key to continue . . .

```

Figure 3.27: Simulation of avalanche effect on proposed DES when plaintext is varied.

Simulation of avalanche effect on standard DES was conducted when key was varied. Figure 3.28 depicts the results of 43.8721% of avalanche effect when key of standard DES was varied.

```
C:\Avelanc\DES\DESWithAvalancheKey\des.exe
30.000000
0123456789abcdff
One changed
34.000000
0123456789abcde7
One changed
32.000000
0123456789abcdeb
One changed
29.000000
0123456789abcded
One changed
36.000000
0123456789abcdee
One changed
0.000000

Avarage of bits changed out of 64 = 28.078125

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 43.872070

-----
Process exited after 0.06162 seconds with return value 0
Press any key to continue . . .
```

Figure 3.28: Simulation of avalanche effect on standard DES when key is varied.

Simulation of avalanche effect on proposed DES was conducted when key was varied. Figure 3.29 depicts the results of 44.2139% of avalanche effect when key of proposed DES was varied.

```
C:\Avelanc\DES\DESWithAvalancheKey\DESWithAvalanche.exe
33.000000
0123456789abcdff
One changed
35.000000
0123456789abcde7
One changed
36.000000
0123456789abcdeb
One changed
35.000000
0123456789abcded
One changed
27.000000
0123456789abcdee
One changed
0.000000

Avarage of bits changed out of 64 = 28.296875

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 44.213867

-----
Process exited after 0.0635 seconds with return value 0
Press any key to continue . . .
```

Figure 3.29: Simulation of avalanche effect on proposed DES when key is varied.

3.6.7. Simulation 7: Testing of Avalanche Effect on MMB

As we mentioned earlier that IoT uses MMB algorithm to secure its software's application [125]. We studied MMB algorithm from [77], [133], [134] and analysed how it works from [77], [133], [134]. We programmed MMB algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our MMB algorithm is encrypting and decrypting according to the specification of its origin, we used test vectors found in [77]. Then we called it a standard MMB algorithm because it gave us the same test vector defined in [77]. After that, we modified it using an initial and the final vector as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it the proposed MMB algorithm. From these two (standard and proposed MMB) algorithms, we calculated their avalanche effects when key was fixed and plaintext was varied, and vice versa. MMB has two inputs (plaintext and key). MMB algorithm uses plaintext of 128 bits long as a first input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of avalanche effect's calculation. Again, MMB algorithm uses key of 128 bits long as a second input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of avalanche effect. We even calculated their speed when key was fixed and plaintext was varied, and vice versa. We finally had four codes of MMB: (1) Standard MMB when key varies, (2) Standard MMB when plaintext varies, (3) Proposed MMB when key varies and (4) Proposed MMB when plaintext varies. Below (in Figure 3.30 to Figure 3.33) we present the executable simulation screenshots of four different MMB algorithms mentioned above. Simulation of avalanche effect on standard MMB was conducted when plaintext was varied. Figure 3.30 depicts the results of 49.7742% of avalanche effect when plaintext of standard MMB was varied.

```

C:\Avelanc\mmb\mmbwithavalench\MMBWithOutAvalanche.exe
64.000000
fffffffffffffffffffffffffffffffffb
3554cc9c13a6b3fbbcdf74117d08bbdb
0800000000000000dc344000000000
One changed
59.000000
fffffffffffffffffffffffffffffd
3554cc9c13a6b3fbbcdf74117d08bbdb
0800000000000000dc344000000000
One changed
69.000000
fffffffffffffffffffffffffffe
3554cc9c13a6b3fbbcdf74117d08bbdb
0800000000000000dc344000000000
One changed
64.000000

Avarage of bits changed out of 128 = 63.710938

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 49.774170

-----
Process exited after 0.2415 seconds with return value 0
Press any key to continue . . .

```

Figure 3.30: Simulation of avalanche effect on standard MMB when plaintext is varied.

Simulation of avalanche effect on proposed MMB was conducted when plaintext was varied. Figure 3.31 depicts the results of 49.7498% of avalanche effect when plaintext of proposed MMB was varied.

```

C:\Avelanc\mmb\mmbwithavalench\MMBWithAvalanche.exe
76.000000
3d6483683af22c07b2a4a4ab8f6e86da
d28d13977d64a8ad5467108691995588
08000000000000004c354000000000
One changed
61.000000
3d6483683af22c07b2a4a4ab8f6e86dc
d28d13977d64a8ad5467108691995588
08000000000000004c354000000000
One changed
75.000000
3d6483683af22c07b2a4a4ab8f6e86df
d28d13977d64a8ad5467108691995588
08000000000000004c354000000000
One changed
72.000000

Avarage of bits changed out of 128 = 63.679688

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 49.749756

-----
Process exited after 0.232 seconds with return value 0
Press any key to continue . . .

```

Figure 3.31: Simulation of avalanche effect on proposed MMB when plaintext is varied.

Simulation of avalanche effect on standard MMB was conducted when key was varied. Figure 3.32 depicts the results of 49.6765% of avalanche effect when key of standard MMB was varied.

```

C:\Avelanc\mmb\mmbwithavalenchKey\MMBWithOutAvalanche.exe
61.000000
0123456789abcdeffedcba9876543200
One changed
67.000000
0123456789abcdeffedcba9876543218
One changed
61.000000
0123456789abcdeffedcba9876543214
One changed
71.000000
0123456789abcdeffedcba9876543212
One changed
61.000000
0123456789abcdeffedcba9876543211
One changed
67.000000

Avarage of bits changed out of 128 = 63.585938

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 100
) = 49.676514

-----
Process exited after 0.1216 seconds with return value 0
Press any key to continue . . .

```

Figure 3.32: Simulation of avalanche effect on standard MMB when key is varied.

Simulation of avalanche effect on proposed MMB was conducted when key was varied. Figure 3.33 depicts the results of 49.6399% of avalanche effect when key of proposed MMB was varied.

```

C:\Avelanc\mmb\mmbwithavalenchKey\MMBWithAvalanche.exe
80.000000
0123456789abcdeffedcba9876543200
One changed
59.000000
0123456789abcdeffedcba9876543218
One changed
63.000000
0123456789abcdeffedcba9876543214
One changed
72.000000
0123456789abcdeffedcba9876543212
One changed
67.000000
0123456789abcdeffedcba9876543211
One changed
66.000000

Avarage of bits changed out of 128 = 63.539063

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 100
) = 49.639893

-----
Process exited after 0.1216 seconds with return value 0
Press any key to continue . . .

```

Figure 3.33: Simulation of avalanche effect on proposed MMB when key is varied.

3.6.8. Simulation 8: Testing of Avalanche Effect on RC5

As we mentioned earlier that IoT uses RC5 algorithm to secure its Mica2 hardware [116]. We studied RC5 algorithm from [135], [81], [118] and analysed how it works from [117], [118], [136]. We programmed RC5 algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our RC5 algorithm is encrypting and decrypting according to the specification of its origin, we used test vectors from [184, p 270]. Then we called it a standard RC5 algorithm because it gave us the same test vector defined in [184, p 270]. After that, we modified it using an initial and the final vector as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it the proposed RC5 algorithm. From these two (standard and proposed RC5) algorithms, we calculated their avalanche effect when key was fixed and plaintext was varied, and vice versa. RC5 has two inputs (plaintext and key). RC5 algorithm uses plaintext of 128 bits long as a first input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of avalanche effect's procedure. Again, RC5 algorithm uses key of 256 bits long as a second input, unlike DES. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of avalanche effect's calculation. We even calculated their speed when key was fixed and plaintext was varied, and vice versa. We finally had four codes of RC5: (1) Standard RC5 when key varies, (2) Standard RC5 when plaintext varies, (3) Proposed RC5 when key varies and (4) Proposed RC5 when plaintext varies. Below (in Figure 3.34 to Figure 3.37) we present the executable simulation screenshots of four different RC5 algorithms mentioned above. Simulation of avalanche effect on standard RC5 was conducted when plaintext was varied. Figure 3.34 depicts the results of 76.1719% of avalanche effect when plaintext of standard RC5 was varied.

```

C:\Avelanc\RC5\RC5WithAvelanchEffect\RC5WithOutPlvECTOR.exe
51.000000
ffffffffffffffffffef
One changed
58.000000
ffffffffffffffffff7
One changed
47.000000
ffffffffffffffffffb
One changed
45.000000
ffffffffffffffffffd
One changed
52.000000
ffffffffffffffffffe
One changed
55.000000

Avarage of bits changed out of 64 = 48.750000

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 76.171875

-----
Process exited after 0.0606 seconds with return value 0
Press any key to continue . . .

```

Figure 3.34: Simulation of avalanche effect on standard RC5 when plaintext is varied.

Simulation of avalanche effect on proposed RC5 was conducted when plaintext was varied. Figure 3.35 depicts the results of 76.9043% of avalanche effect when plaintext of proposed RC5 was varied.

```

C:\Avelanc\RC5\RC5WithAvelanchEffect\RC5WithPlvECTOR.exe
47.000000
ffffffffffffffffffef
One changed
48.000000
ffffffffffffffffff7
One changed
55.000000
ffffffffffffffffffb
One changed
47.000000
ffffffffffffffffffd
One changed
48.000000
ffffffffffffffffffe
One changed
49.000000

Avarage of bits changed out of 64 = 49.218750

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 76.904297

-----
Process exited after 0.05999 seconds with return value 0
Press any key to continue . . .

```

Figure 3.35: Simulation of avalanche effect on proposed RC5 when plaintext is varied.

Simulation of avalanche effect on standard RC5 was conducted when key was varied. Figure 3.36 depicts the results of 49.1821% of avalanche effect when key of standard RC5 was varied.

```
C:\Avelanc\RC5\RC5WithAvalanchEffectKey\RC5WithOutPlvECTOR.exe
33.000000
0123456789abcdeffedcba9876543200
One changed
29.000000
0123456789abcdeffedcba9876543218
One changed
28.000000
0123456789abcdeffedcba9876543214
One changed
30.000000
0123456789abcdeffedcba9876543212
One changed
26.000000
0123456789abcdeffedcba9876543211
One changed
41.000000

Avarage of bits changed out of 128 = 31.476563

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/64) x 100
) = 49.182129

-----
Process exited after 0.1165 seconds with return value 0
Press any key to continue . . .
```

Figure 3.36: Simulation of avalanche effect on standard RC5 when key is varied.

Simulation of avalanche effect on proposed RC5 was conducted when key was varied. Figure 3.37 depicts the results of 49.7925% of avalanche effect when key of proposed RC5 was varied.

```
C:\Avelanc\RC5\RC5WithAvalanchEffectKey\RC5WithPlvECTOR.exe
29.000000
0123456789abcdeffedcba9876543200
One changed
35.000000
0123456789abcdeffedcba9876543218
One changed
26.000000
0123456789abcdeffedcba9876543214
One changed
28.000000
0123456789abcdeffedcba9876543212
One changed
34.000000
0123456789abcdeffedcba9876543211
One changed
31.000000

Avarage of bits changed out of 128 = 31.867188

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/64) x 100
) = 49.792480

-----
Process exited after 0.1325 seconds with return value 0
Press any key to continue . . .
```

Figure 3.37: Simulation of avalanche effect on proposed RC5 when key is varied.

3.6.9. Simulation 9: Testing of Avalanche Effect on Serpent

As we mentioned earlier that IoT uses Serpent algorithm to secure its sensors, information and data [104], [102]. We studied Serpent algorithm from [101], [82], [83], [118] and analysed how it works from [104], [137], [138]. We programed Serpent algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our Serpent algorithm is encrypting and decrypting according to the specification of its origin, we used test vectors found in [138]. Then we call it standard Serpent algorithm because it gave us the same test vector defined in [138]. After that, we modified it using an initial and the final vector as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it proposed Serpent algorithm. From these two (standard and proposed Serpent) algorithms, we calculated their avalanche effect when key was fixed and plaintext was varied, and vice versa. Serpent has two inputs (plaintext and key). Serpent algorithm uses plaintext of 128 bits long as a first input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of avalanche effect. Again, Serpent algorithm uses key of 256 bits long as a second input unlike DES. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of avalanche effect. We even calculated their speeds when key was fixed and plaintext was varied, and vice versa. We finally had four codes of Serpent: (1) Standard Serpent when key varies, (2) Standard Serpent when plaintext varies, (3) Proposed Serpent when key varies and (4) Proposed Serpent when plaintext varies. Below (in Figure 3.8 to Figure 3.41) we present the executable simulation screenshots of four different Serpent algorithms mentioned above. Simulation of avalanche effect on standard Serpent was conducted when plaintext was varied. Figure 3.38 depicts the results of 50.3845% of avalanche effect when plaintext of standard Serpent was varied.

```

C:\Avelanc\Serpent\SerpentWithAvalanche\SerpentWithOutPiVectors.exe
67.000000
ffffffffffffffffffffffffffffffffffef
One changed
64.000000
ffffffffffffffffffffffffffffffffff7
One changed
76.000000
ffffffffffffffffffffffffffffffffffb
One changed
64.000000
ffffffffffffffffffffffffffffffffffd
One changed
61.000000
ffffffffffffffffffffffffffffffffffe
One changed
68.000000

Avarage of bits changed out of 128 = 64.492188

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 50.384521

-----
Process exited after 0.1215 seconds with return value 0
Press any key to continue . . .

```

Figure 3.38: Simulation of avalanche effect on standard Serpent when plaintext is varied.

Simulation of avalanche effect on proposed Serpent was conducted when plaintext was varied. Figure 3.39 depicts the results of 49.7986% of avalanche effect when plaintext of proposed Serpent was varied.

```

C:\Avelanc\Serpent\SerpentWithAvalanche\SerpentWithPiVectors.exe
72.000000
ffffffffffffffffffffffffffffffffffef
One changed
61.000000
ffffffffffffffffffffffffffffffffff7
One changed
58.000000
ffffffffffffffffffffffffffffffffffb
One changed
61.000000
ffffffffffffffffffffffffffffffffffd
One changed
60.000000
ffffffffffffffffffffffffffffffffffe
One changed
53.000000

Avarage of bits changed out of 128 = 63.742188

Avalanche Effect in Percentage= ((Avarage of bits changed out of 128 )/128) x 10
0 ) = 49.798584

-----
Process exited after 0.125 seconds with return value 0
Press any key to continue . . .

```

Figure 3.39: Simulation of avalanche effect on proposed Serpent when plaintext is varied.

Simulation of avalanche effect on standard Serpent was conducted when key was varied. Figure 3.40 depicts the results of 49.8657% of avalanche effect when key of standard Serpent was varied.

```

C:\Avelanc\Serpent\SerpentWithAvalancheKEY\SerpentWithOutPiVectors.exe
68.000000
0123456789abcdeffedcba98765432300123456789abcdeffedcba9876543200
One changed
74.000000
0123456789abcdeffedcba98765432000123456789abcdeffedcba9876543218
One changed
65.000000
0123456789abcdeffedcba98765432180123456789abcdeffedcba9876543214
One changed
72.000000
0123456789abcdeffedcba98765432140123456789abcdeffedcba9876543212
One changed
63.000000
0123456789abcdeffedcba98765432120123456789abcdeffedcba9876543211
One changed
67.000000

Avarage of bits changed out of 256 = 63.828125

Avalanche Effect in Percentage= ((Avarage of bits changed out of 256 )/128) x 10
0 ) = 49.865723

-----
Process exited after 0.2714 seconds with return value 0
Press any key to continue . . .

```

Figure 3.40: Simulation of avalanche effect on standard Serpent when key is varied.

Simulation of avalanche effect on proposed Serpent was conducted when key was varied. Figure 3.41 depicts the results of 50.5341% of avalanche effect when key of proposed Serpent was varied.

```

C:\Avelanc\Serpent\SerpentWithAvalancheKEY\SerpentWithPiVectors.exe
62.000000
0123456789abcdeffedcba98765432300123456789abcdeffedcba9876543200
One changed
61.000000
0123456789abcdeffedcba98765432000123456789abcdeffedcba9876543218
One changed
71.000000
0123456789abcdeffedcba98765432180123456789abcdeffedcba9876543214
One changed
64.000000
0123456789abcdeffedcba98765432140123456789abcdeffedcba9876543212
One changed
57.000000
0123456789abcdeffedcba98765432120123456789abcdeffedcba9876543211
One changed
67.000000

Avarage of bits changed out of 256 = 64.683594

Avalanche Effect in Percentage= ((Avarage of bits changed out of 256 )/128) x 10
0 ) = 50.534058

-----
Process exited after 0.2744 seconds with return value 0
Press any key to continue . . .

```

Figure 3.41: Simulation of avalanche effect on proposed Serpent when key is varied.

3.6.10. Simulation 10: Testing of Avalanche Effect on Skipjack

As we mentioned earlier that I IoT uses Skipjack algorithm to secure its Mica2 hardware [116]. We studied Skipjack algorithm from [85], [139], [120] and analysed how it works from [140], [141], [86], [142]. We programmed Skipjack algorithm according to the analysis mentioned above using C++ code. We optimise the code to get maximum efficiency. To verify if our Skipjack algorithm is encrypting and decrypting according to the specification of its origin, we used test vectors found in [142]. Then we called it a standard Skipjack algorithm because it gave us the same test vector defined in [142]. After that, we modified it using an initial and the final vector as we proposed in Figure 1.2 and Figure 2.8 using C++ program. Then we called it the proposed Skipjack algorithm. From these two (standard and proposed Skipjack) algorithms, we calculated their avalanche effects when key was fixed and plaintext was varied, and vice versa. Skipjack has two inputs (plaintext and key). Skipjack algorithm uses plaintext of 64 bits long as a first input. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary plaintext according to the definition of avalanche effect. Again, Skipjack algorithm uses key of 80 bits long as a second input, unlike any algorithm defined above. We varied each bit from the first to last bit, one at a time to get better results of the avalanche effect. That is how we vary the key according to the definition of avalanche effect. We even calculated their speeds when key was fixed and plaintext was varied, and vice versa. We finally had four codes of Skipjack: (1) Standard Skipjack when key varies, (2) Standard Skipjack when plaintext varies, (3) Proposed Skipjack when key varies and (4) Proposed Skipjack when plaintext varies. Below (in Figure 3.42 to Figure 3.45) we present the executable simulation screenshots of four different Skipjack algorithms mentioned above. Simulation of avalanche effect on standard Skipjack was conducted when plaintext was varied. Figure 3.42 depicts the results of 48.7793% of avalanche effect when plaintext of standard Skipjack was varied.

```

C:\Avelanc\Skipjack\SkipjackWithAvalanchEffect\SkipjackWithOutPiVALUE.exe
22.000000
fffffffffffffffffef
One changed
33.000000
fffffffffffffffff7
One changed
31.000000
fffffffffffffffffb
One changed
32.000000
fffffffffffffffffd
One changed
34.000000
fffffffffffffffffe
One changed
38.000000

Avarage of bits changed out of 64 = 31.218750

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 48.779297

-----
Process exited after 0.06052 seconds with return value 0
Press any key to continue . . .

```

Figure 3.42: Simulation of avalanche effect on standard Skipjack when plaintext is varied.

Simulation of avalanche effect on proposed Skipjack was conducted when plaintext was varied. Figure 3.43 depicts the results of 49.2188% of avalanche effect when plaintext of proposed Skipjack was varied.

```

C:\Avelanc\Skipjack\SkipjackWithAvalanchEffect\SkipjackWithPiVALUE.exe
34.000000
3d6483683af22c17
One changed
35.000000
3d6483683af22c0f
One changed
35.000000
3d6483683af22c03
One changed
34.000000
3d6483683af22c05
One changed
24.000000
3d6483683af22c06
One changed
38.000000

Avarage of bits changed out of 64 = 31.500000

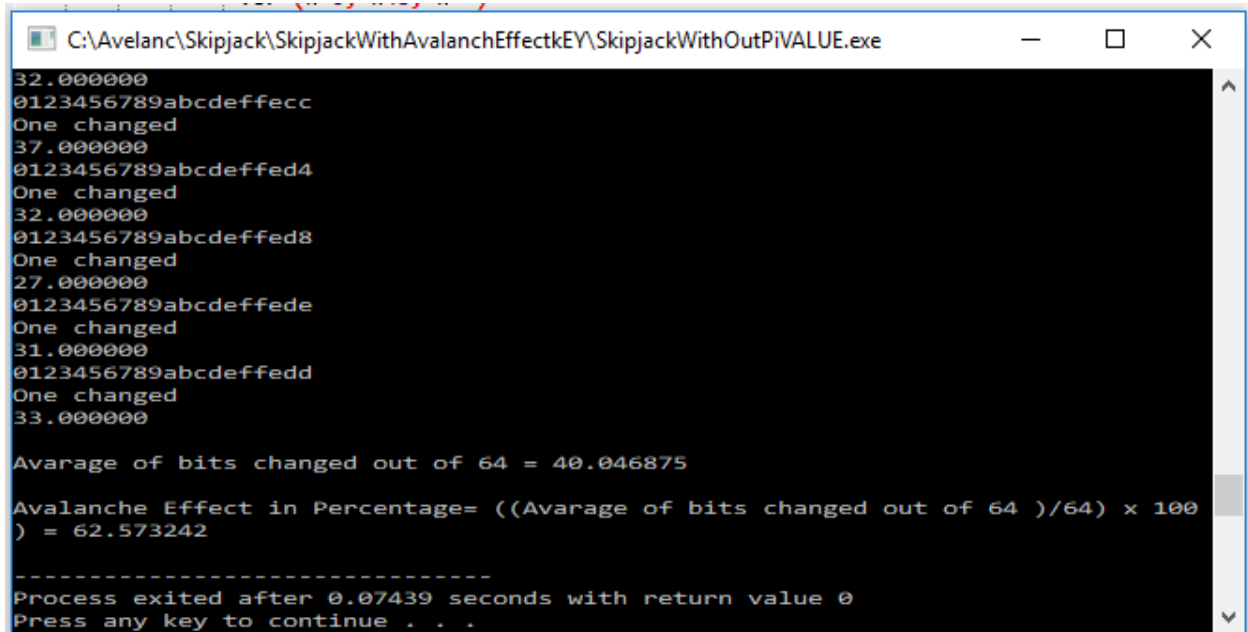
Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 49.218750

-----
Process exited after 0.06474 seconds with return value 0
Press any key to continue . . .

```

Figure 3.43: Simulation of avalanche effect on proposed Skipjack when plaintext is varied.

Simulation of avalanche effect on standard Skipjack was conducted when key was varied. Figure 3.44 depicts the results of 62.5732% of avalanche effect when key of standard Skipjack was varied.



```
C:\Avelanc\Skipjack\SkipjackWithAvalanchEffectkEY\SkipjackWithOutPiVALUE.exe
32.000000
0123456789abcdeffecc
One changed
37.000000
0123456789abcdeffed4
One changed
32.000000
0123456789abcdeffed8
One changed
27.000000
0123456789abcdeffede
One changed
31.000000
0123456789abcdeffedd
One changed
33.000000

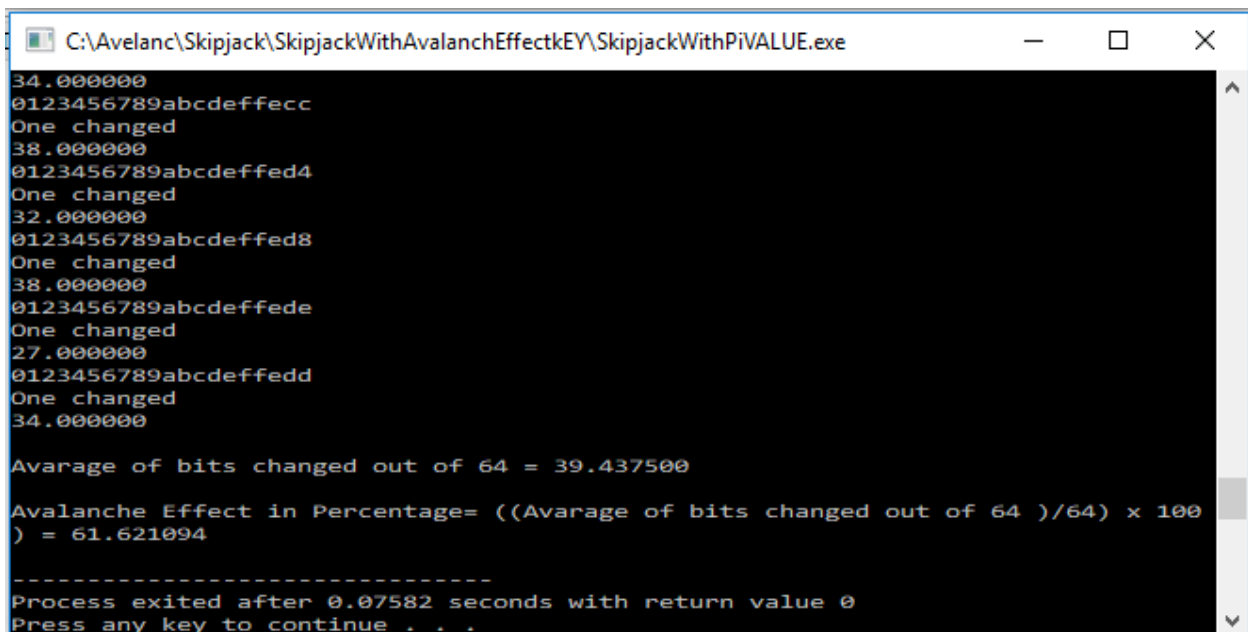
Avarage of bits changed out of 64 = 40.046875

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 62.573242

-----
Process exited after 0.07439 seconds with return value 0
Press any key to continue . . .
```

Figure 3.44: Simulation of avalanche effect on standard Skipjack when key is varied.

Simulation of avalanche effect on proposed Skipjack was conducted when key was varied. Figure 3.45 depicts the results of 61.6211% of avalanche effect when key of proposed Skipjack was varied.



```
C:\Avelanc\Skipjack\SkipjackWithAvalanchEffectkEY\SkipjackWithPiVALUE.exe
34.000000
0123456789abcdeffecc
One changed
38.000000
0123456789abcdeffed4
One changed
32.000000
0123456789abcdeffed8
One changed
38.000000
0123456789abcdeffede
One changed
27.000000
0123456789abcdeffedd
One changed
34.000000

Avarage of bits changed out of 64 = 39.437500

Avalanche Effect in Percentage= ((Avarage of bits changed out of 64 )/64) x 100
) = 61.621094

-----
Process exited after 0.07582 seconds with return value 0
Press any key to continue . . .
```

Figure 3.45: Simulation of avalanche effect on proposed Skipjack when key is varied.

3.4. Chapter Summary

In this chapter we presented the different methods that were used for the study, like source of initial and final vectors, PI methodology, methodology of study based on avalanche effect, flow chart of avalanche effect, research design, experimental procedures and finally simulation screenshots of all ten different algorithms when avalanche affect was tested using C++ programming. In the next chapter we will analyze and discuss the results.

CHAPTER 4: RESULTS DISCUSSION AND ANALYSIS

4.1 Introduction

As we mentioned earlier the avalanche effect is a desirable property of cryptographic algorithms, if input is changed slightly the output must change excessively [143], [6]. One main reason why the avalanche effect is necessary is that by flipping only one bit of input, if there is large change in the output, and then this shows that it is harder to perform an attack (intrusion or hacking) on the cryptographic algorithm [6]. Oppenheim et al [144] indicated that an algorithm with high avalanche effect is a strong algorithm.

The other two main desirable properties (except the avalanche effect) that differentiates one encryption algorithm from another is its time and speed to encrypt data [145]. We also measured time and speed during the avalanche process. We calculated the time taken to perform avalanche effect on each and every algorithm. The speed of algorithm was calculated as follows: To start with, if the avalanche effect is calculated when the plaintext is varied, and given that the size of the plaintext is 128 bits (for example), then flipping one bit from left to right until to the end of 128 bits means that the encryption process is conducted 128 times. This simply means that $128 \times 128 = 16384$ bits were encrypted during the avalanche effect. In order to calculate the speed we used the equation 3.2.

As a preamble, and essential to this study, an algorithm is standard if it is not modified anywhere by us in this study. It is taken as it is and analysed as it is from their designers. When we say proposed algorithm, we mean we modified the standard algorithm using the proposed method of introducing the initial and final vectors on the standard algorithm. See Figure 3.1 and Figure 3.2 for more explanation.

4.2 Results and analysis

4.2.1. Results 1: The Avalanche Effect on AES

From Table 4.1 and Figure 4.1 when the plaintext was varied the proposed AES showed low avalanche effect as compared to the standard AES. Then standard AES performed better in terms of the avalanche effect when the plaintext was varied. Given our hypothesis (**H1**), one can easily conclude that the AES algorithm performed as per our expectation concerning the needed avalanche effect. From Table 4.1 the standard AES algorithm has an avalanche effect, which is improved than the proposed AES algorithm. Standard AES algorithm has 49.7925 % whereas proposed AES algorithm has 49.6033%. From Table 4.1 standard AES algorithm is slow compared to proposed AES algorithm. Standard AES takes 0.2286 seconds to encrypt whereas proposed AES takes 0.2244 seconds to encrypt.

Table 4.1: Results of standard and proposed AES when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD AES	NO	NO	49.7925 %	0.2286 sec	71671,0411 bit/sec
PROPOSED AES	YES	YES	49.6033%	0.2244 sec	73012,4777 bit/sec

From Figure 4.1 the standard AES algorithm has an avalanche effect, which is improved than the proposed AES algorithm. Standard AES algorithm has 49.7925% whereas proposed AES algorithm has 49.6033%. When plaintext was varied.

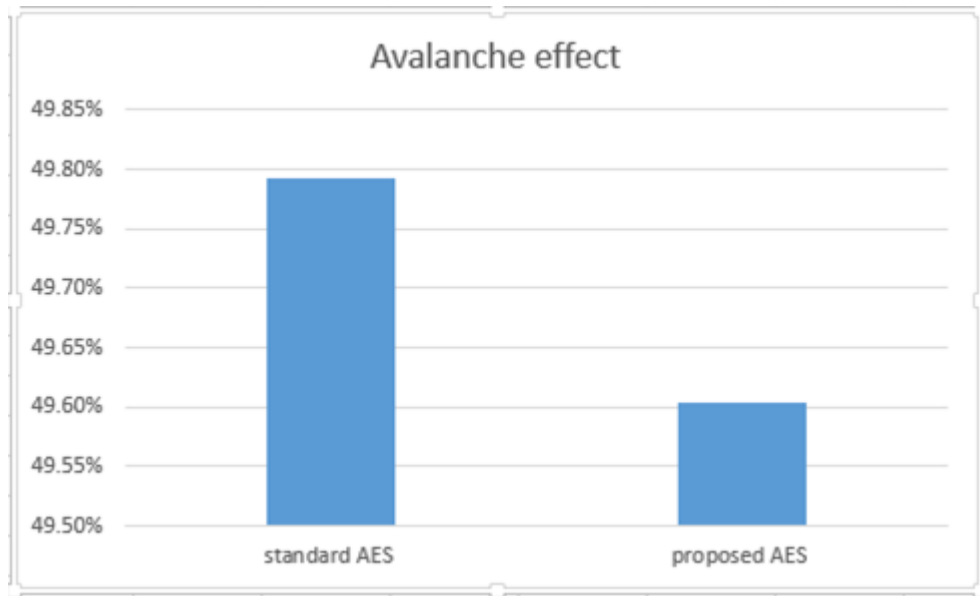


Figure 4.1: Results of avalanche effect on AES when plaintext was varied.

From Table 4.1 and Figure 4.2, the proposed AES was faster than the standard AES. IoT uses standard AES to secure its sensors and contactless smart cards. Within these domains, the proposed algorithm of AES performed better and it could be suitable for applications such as sensors and contactless smart cards of IoT when speed is considered. From 4.2, speed of standard AES is 71671, 0411 bit/sec, whereas of the proposed AES algorithm is 73012, 4777 bit/sec.

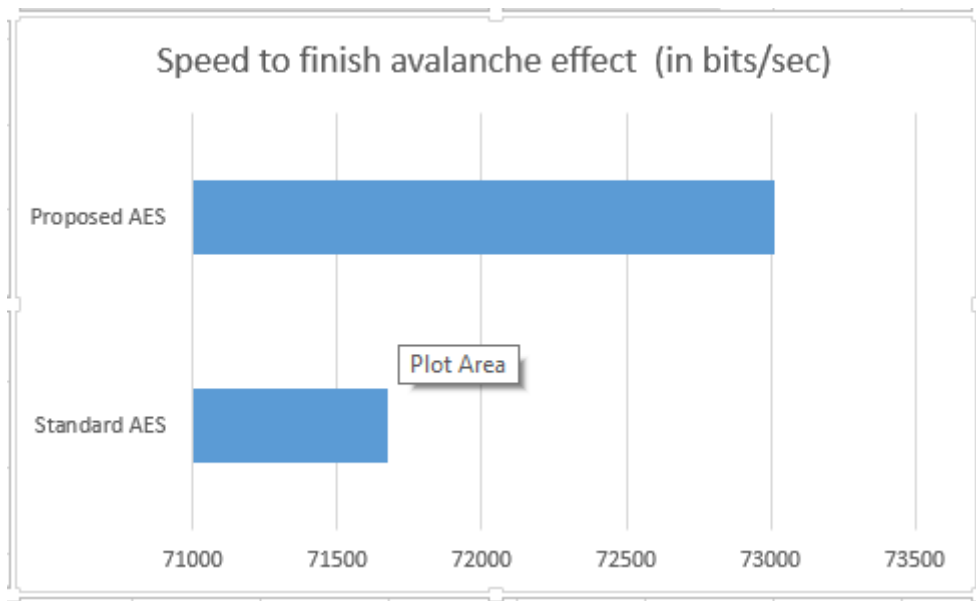


Figure 4.2: Results of speed taken on AES when plaintext was varied.

From Table 4.2 and Figure 4.3, the proposed algorithm managed to increase the avalanche effect of standard AES algorithm key was varied, that is from 49.0600% up to 49.9300 % by using the proposed method. This means that the modified (proposed) AES can replace the standard AES because of its improved avalanche effect if the user wants to vary the key. From Table 4.2 the standard AES algorithm has an avalanche effect which is low than the proposed AES algorithm. Standard AES algorithm has 49.0662% whereas proposed AES algorithm has 49.9390%. From Table 4.2 standard AES algorithm is fast compared to proposed AES algorithm. Standard AES takes 0.1158 seconds to encrypt whereas proposed AES takes 0.1180 seconds to encrypt.

Table 4.2: Results of standard and proposed AES when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD AES	NO	NO	49.0662%	0.1158 sec	141485.3195 bit/sec
PROPOSED AES	YES	YES	49.9390%	0.118 sec	138847,4576 bits/sec

From Figure 4.3 the standard AES algorithm has an avalanche effect which is low than the proposed AES algorithm. Standard AES algorithm has 49.0662% whereas proposed AES algorithm has 49.9390%. When key was varied.

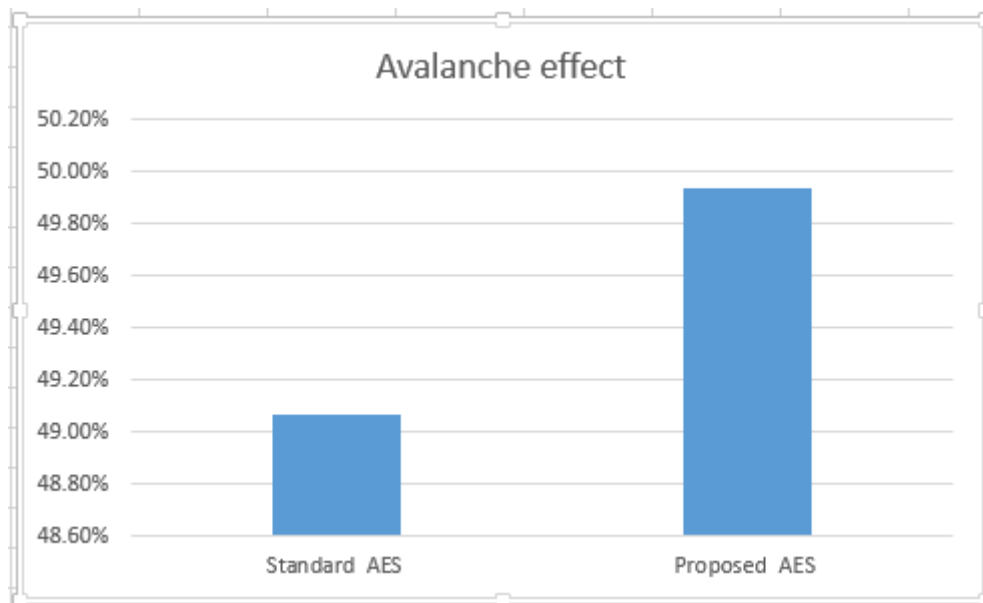


Figure 4.3: Results of avalanche effect on AES when key was varied.

From Table 4.2 and Figure 4.4 when the key was varied the proposed AES was slower in terms of speed. Then standard AES was still the best in terms of speed when the key was varied. In terms of speed, the standard AES worked better on IoT when the user varies the key. From

Figure 4.4, the speed of standard AES is 141485.3195 bit/sec, whereas of proposed AES is 138847,4576 bits/sec.

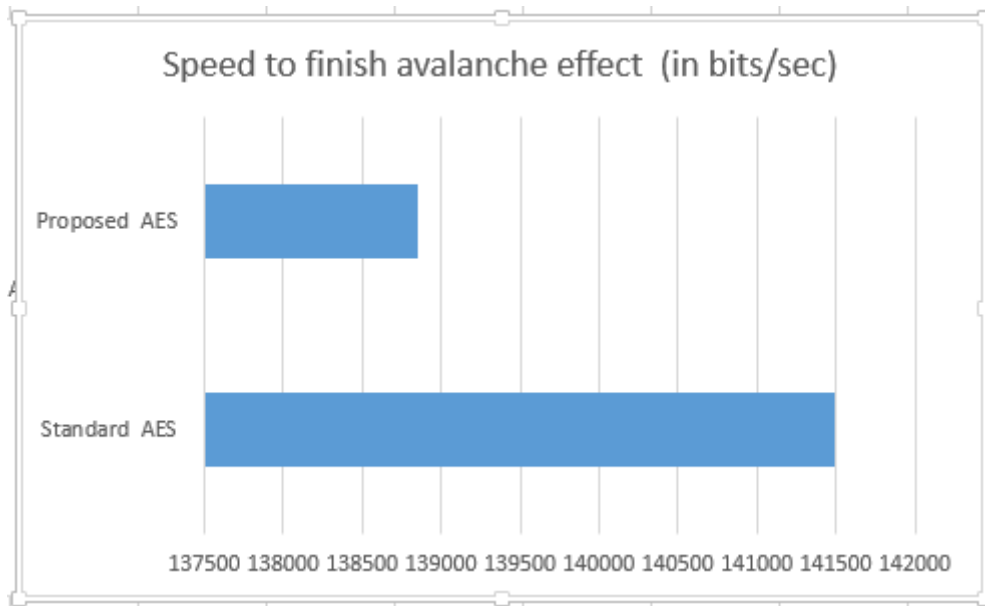


Figure 4.4: Results of speed taken on AES when key was varied.

4.2.2. Results 2: The Avalanche Effect on Blowfish

From Table 4.3 and Figure 4.5 it shows that when the plaintext was varied the proposed Blowfish algorithm showed a low avalanche effect compared to the standard Blowfish. This showed that the standard Blowfish was still the best in terms of the avalanche effect when the plaintext was varied. Given the above scenario, it would not be necessary to replace the standard Blowfish algorithm with the proposed one, especially if the user wants an algorithm, which has an improved avalanche effect when plaintext is varied. From Table 4.3 the standard Blowfish algorithm has an avalanche effect, which is improved than the proposed Blowfish algorithm. Standard Blowfish algorithm has 50.5615% whereas proposed Blowfish algorithm has 48.3398%. From Table 4.3 standard Blowfish algorithm is slow compared to proposed Blowfish algorithm. Standard Blowfish takes 0.0614 seconds to encrypt whereas proposed Blowfish takes 0.0611 seconds to encrypt.

Table 4.3: Results of standard and proposed Blowfish when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD BLOWFISH	NO	NO	50.5615%	0.0614 sec	66699,2347 bit/sec
PROPOSED BLOWFISH	YES	YES	48.3398%	0.0611sec	67037,6432 bit/sec

From Figure 4.5 the standard Blowfish algorithm has an avalanche effect, which is improved than the proposed Blowfish algorithm. Standard Blowfish algorithm has 50.5615% whereas proposed Blowfish algorithm has 48.3398%. When plaintext was varied.

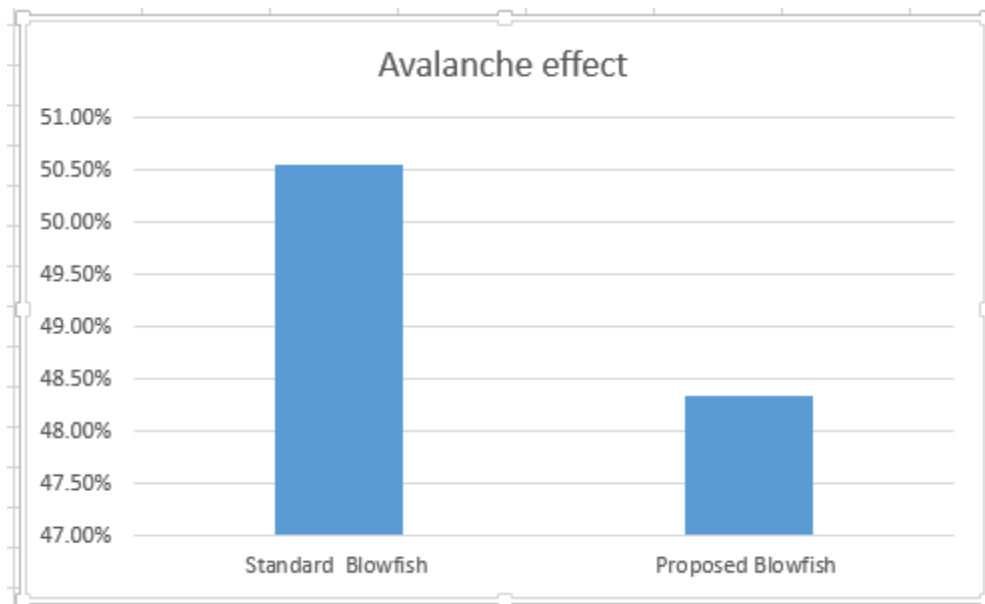


Figure 4.5: Results of avalanche effect on Blowfish when plaintext was varied.

From Table 4.3 and Figure 4.6, the proposed Blowfish was faster than the standard Blowfish. IoT uses the standard Blowfish to secure its applications and network layers. This means that if any user wants to vary plaintext and secure IoT's applications and network layers, and wants to use a fast algorithm like Blowfish, the proposed Blowfish algorithm is an appropriate one.

From Figure 4.6, speed of standard Blowfish is 66699,2347 bit/sec, whereas of proposed Blowfish is 67037,6432 bit/sec.

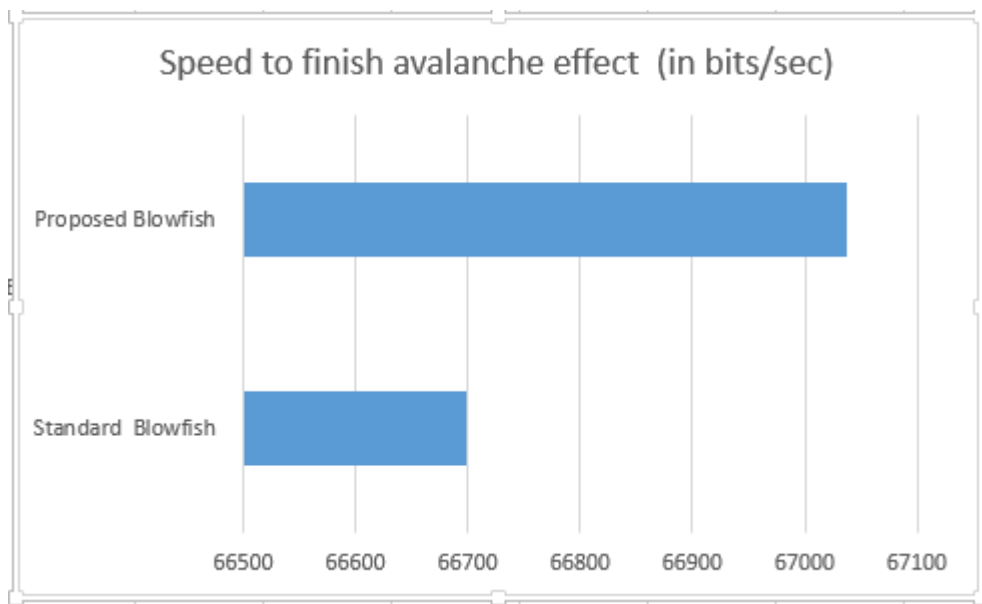


Figure 4.6: Results of speed taken on Blowfish when plaintext was varied.

From Table 4.4 and Figure 4.7, when the key was varied the proposed Blowfish had a low avalanche effect as compared to the standard Blowfish. Then standard Blowfish is still the best in terms of the avalanche effect if plaintext is varied. In this regard, there was no need to replace standard Blowfish algorithms from IoT if the user wants an algorithm that has an improved avalanche effect when plaintext is varied. From Table 4.4 the standard Blowfish algorithm has an avalanche effect, which is improved than the proposed Blowfish algorithm. Standard Blowfish algorithm has 50.4517% whereas proposed Blowfish algorithm has 49.9878%. From Table 4.4 standard Blowfish algorithm took exact same time as the proposed Blowfish algorithm took during encryption. They both took 0.1233 seconds to encrypt.

Table 4.4: Results of standard and proposed Blowfish when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD BLOWFISH	NO	NO	50.4517%	0.1233 sec	132879,1565 bits/sec
PROPOSED BLOWFISH	YES	YES	49.9878%	0.1233 sec	132879,1565 bits/sec

From Figure 4.7 the standard Blowfish algorithm has an avalanche effect, which is improved than the proposed Blowfish algorithm. Standard Blowfish algorithm has 50.4517% whereas proposed Blowfish algorithm has 49.9878%. When key was varied.

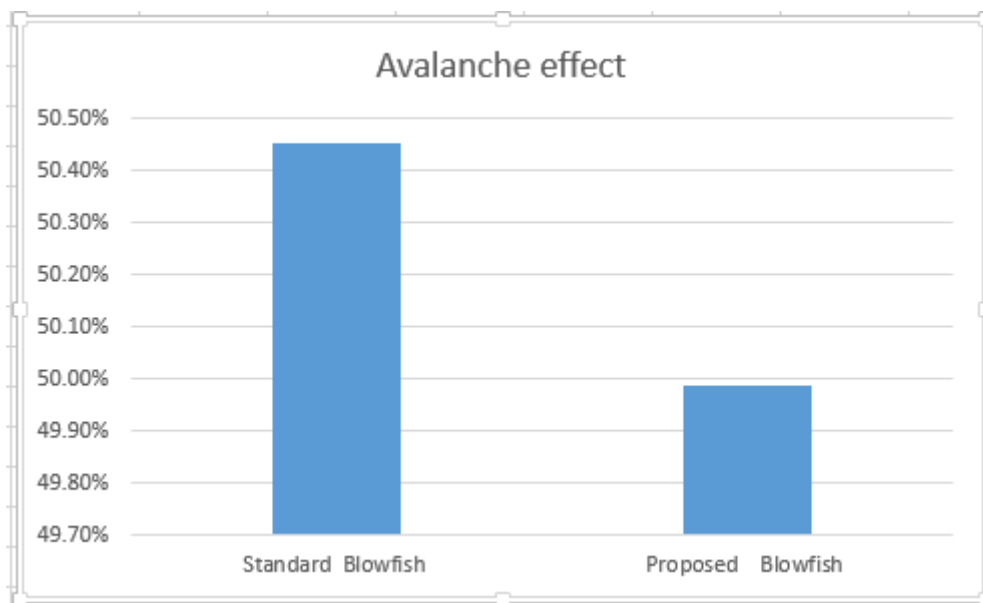


Figure 4.7: Results of avalanche effect on Blowfish when key was varied.

From Table 4.4 and Figure 4.8 and from our experiment the speed and time of proposed and standard Blowfish algorithm were found to be equal when key was varied. It does not matter to choose which one between the two if the user wants algorithm like Blowfish with an

improved speed when key varies. Both standard and proposed Blowfish algorithms has the same speed of 132879,1565 bits/sec.

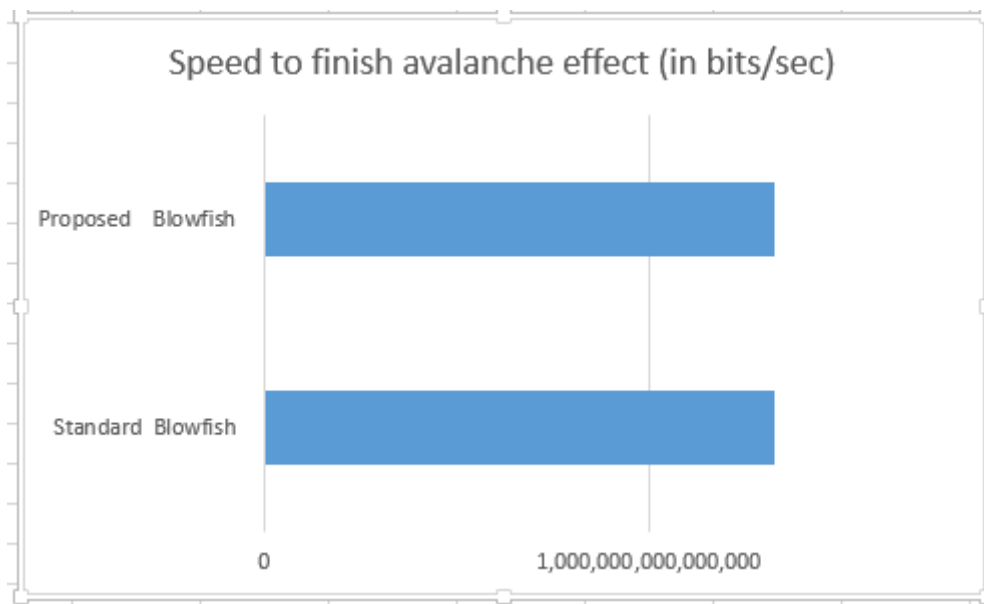


Figure 4.8: Results of speed taken on Blowfish when key was varied.

4.2.3 Results 3: The Avalanche Effect on Camellia

From Table 4.5 and Figure 4.9, the proposed Camellia algorithm managed to increase the avalanche effect of standard Camellia algorithm when plaintext was varied. That is from 49.5% up to 50.1000% by using the proposed method. This means that our modified (proposed) Camellia algorithm can replace the standard Camellia algorithm when one wants to choose between the two algorithms because it has an improved avalanche effect when plaintext varies. We suggested that instead of using standard Camellia algorithm to secure IoT's protocols, the proposed Camellia algorithm could replace the standard Camellia algorithm, to get high security on IoT's protocols if plaintext varies. From Table 4.5 the standard Camellia algorithm has an avalanche effect which is low than the proposed Camellia algorithm. Standard Camellia algorithm has 49.4690% whereas proposed Camellia algorithm has 50.0977%. From Table 4.5 standard Camellia algorithm is slow compared to proposed Camellia algorithm. Standard Camellia takes 0.1576 seconds to encrypt whereas proposed Camellia takes 0.1217 seconds to encrypt.

Table 4.5: Results of standard and proposed Camellia when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD CAMELLIA	NO	NO	49.4690%	0.1576 sec	103959,3909 bits/sec
PROPOSED CAMELLIA	YES	YES	50.0977%	0.1217sec	134626,1289 bits/sec

From Figure 4.9 the standard Camellia algorithm has an avalanche effect which is low than the proposed Camellia algorithm. Standard Camellia algorithm has 49.4690% whereas proposed Camellia algorithm has 50.0977%. When plaintext was varied.

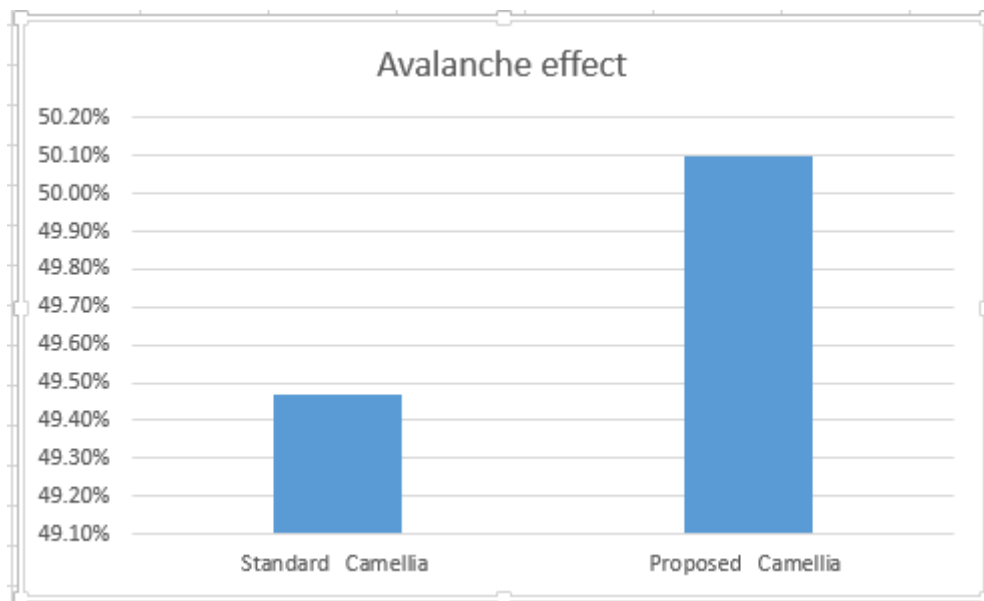


Figure 4.9: Results of avalanche effect on Camellia when plaintext was varied.

From Table 4.5 and Figure 4.10, the proposed Camellia algorithm was faster than the standard Camellia algorithm. IoT uses standard Camellia algorithm to secure its data and information. In this situation, users who want to vary plaintext and encrypt data and information of IoT in high speed then he/she could use the proposed algorithm as it gives an improved encryption

speed. From Figure 4.10, speed of standard Camellia is 103959,3909 bits/sec, whereas of proposed Camellia is 134626, 1289 bits/sec.

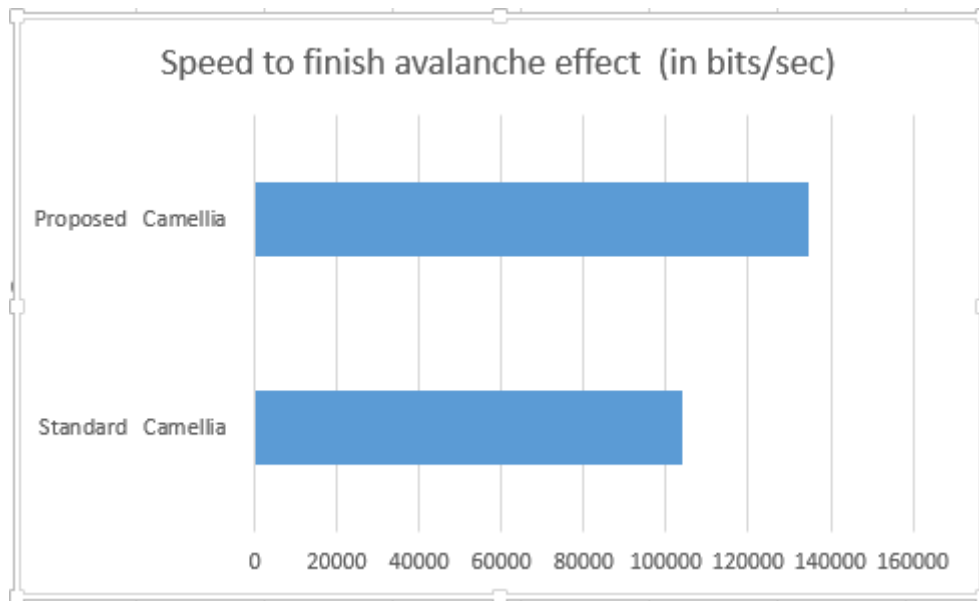


Figure 4.10: Results of speed taken on Camellia when plaintext was varied.

From Table 4.6 and Figure 4.11, the proposed Camellia algorithm showed an improved avalanche effect compare to standard one. The avalanche effect was increased from 49.6094% up to 49.8983% when the proposed method was used. This means that the proposed Camellia algorithm could be used in place of standard Camellia algorithm if there is a need to have an algorithm with an improved avalanche effect in line with the suggestions of [143], [146]. We suggested that instead of using standard Camellia algorithm to secure IoT's protocols when key varies, then the proposed Camellia algorithm could be used to secure IoT's protocols. From Table 4.6 the standard Camellia algorithm has an avalanche effect which is low than the proposed Camellia algorithm. Standard Camellia algorithm has 49.6094% whereas proposed Camellia algorithm has 49.8983%. From Table 4.6 standard Camellia algorithm is fast compared to proposed Camellia algorithm. Standard Camellia takes 0.1829 seconds to encrypt whereas proposed Camellia takes 0.1859 seconds to encrypt.

Table 4.6: Results of standard and proposed Camellia when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD CAMELLIA	NO	NO	49.6094%	0.1829 sec	201552,7612 bits/sec
PROPOSED CAMELLIA	YES	YES	49.8983%	0.1859 sec	198300,1614 bits/sec

From Figure 4.11 the standard Camellia algorithm has an avalanche effect which is low than the proposed Camellia algorithm. Standard Camellia algorithm has 49.6094% whereas proposed Camellia algorithm has 49.8983%. When key was varied.

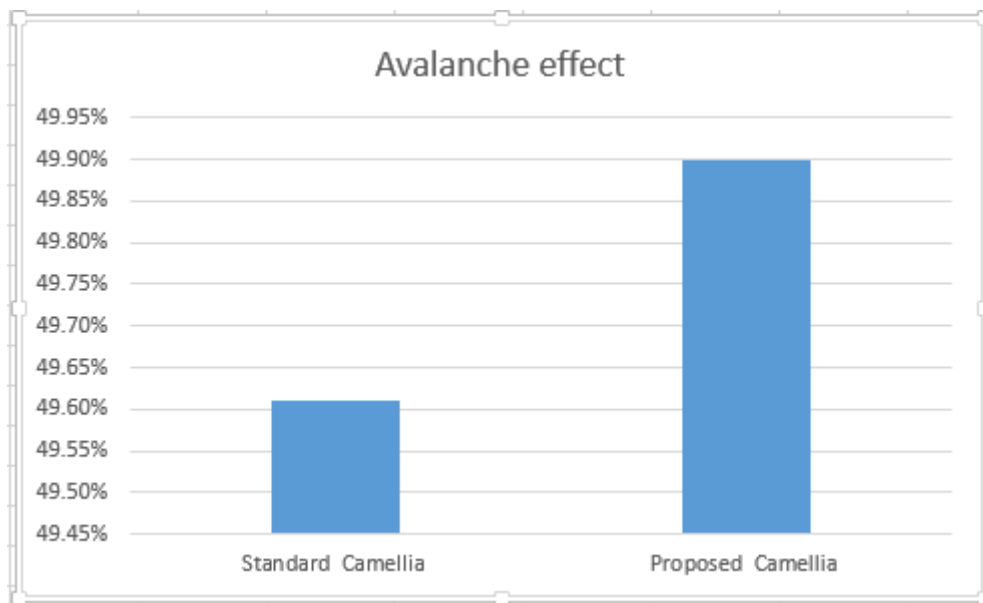


Figure 4.11: Results of avalanche effect on Camellia when key was varied.

From Table 4.6 and Figure 4.12 showed that the proposed algorithm showed slowness when the key was varied. In this case, it was observed that the standard Camellia algorithm was the best in terms of speed when the key was varied. Given this situation, it is therefore not necessary to use the proposed Camellia algorithm on IoT if one wants a fast algorithm when

key is varied. From Figure 4.12, speed of standard Camellia is 201552,7612 bits/sec, whereas of proposed Camellia is 198300,1614 bits/sec.

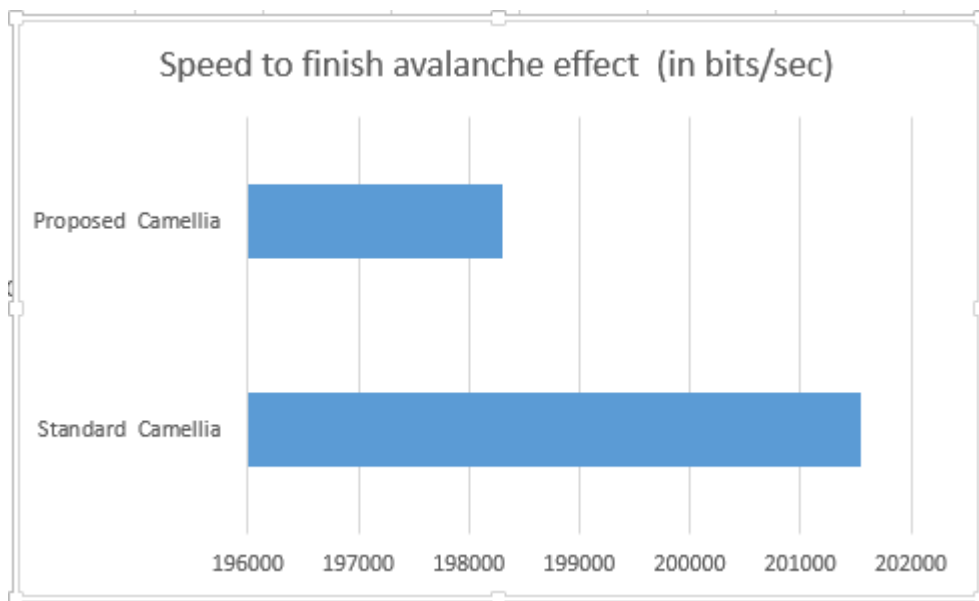


Figure 4.12: Results of speed taken on Camellia when key was varied.

4.2.4. Results 4: The Avalanche Effect on Cast-128.

From Table 4.7 and Figure 4.13, the proposed Cast-128 algorithm yielded an improved avalanche compared standard Cast-128 algorithm when plaintext varied. That is the avalanche effect was increased from 48.8281 % up to 49.3164% by using the proposed method. This means that our modified (proposed) Cast-128 algorithm can replace the standard Cast-128 algorithm when one wants to choose between the two algorithms because it has an improved avalanche effect. This is according to [143], [6]. IoT uses standard Cast-128 algorithm to secure its prototypes, then we recommend that it should be replace by the proposed Cast-128 algorithm to give high security of IoT's prototypes. From Table 4.7 the standard Cast-128 algorithm has an avalanche effect which is low than the proposed Cast-128 algorithm. Standard Cast-128 algorithm has 48.8281% whereas proposed Cast-128 algorithm has 49.3164%. From Table 4.7 standard Cast-128 algorithm is fast compared to proposed Cast-128 algorithm. Standard Cast-128 took 0.0630 seconds to encrypt whereas proposed Cast-128 takes 0.0960 seconds to encrypt.

Table 4.7: Results of standard and proposed Cast-128 when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD CAST-128	NO	NO	48.8281%	0.06299 sec	260104,7785 bits/sec
PROPOSED CAST-128	YES	YES	49.3164%	0.0960 sec	170666,6667 bits/sec

From Figure 4.13 the standard Cast-128 algorithm has an avalanche effect which is low than the proposed Cast-128 algorithm. Standard Cast-128 algorithm has 48.8281% whereas proposed Cast-128 algorithm has 49.3164%. When plaintext was varied.

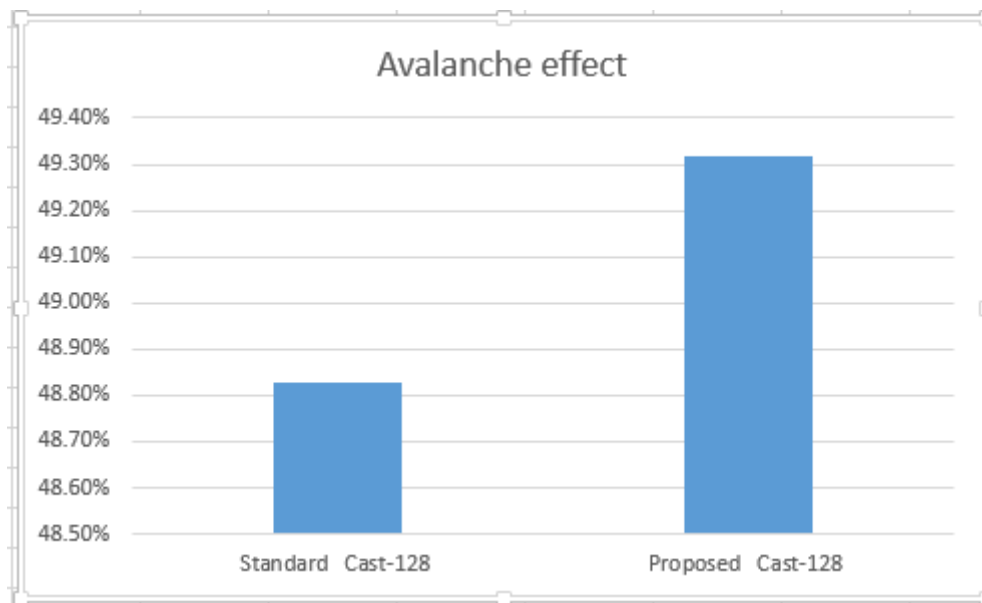


Figure 4.13: Results of avalanche effect on Cast-128 when plaintext was varied.

From Table 4.7 and Figure 4.14 when the plaintext is varied the proposed Cast-128 is slow. Then standard Cast-128 is still best in speed if the plaintext is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm, which is fast when plaintext varied.

From Figure 4.14, speed of standard Cast-128 is 260104,7785 bits/sec, whereas of proposed Cast-128 is 170666,6667 bits/sec.

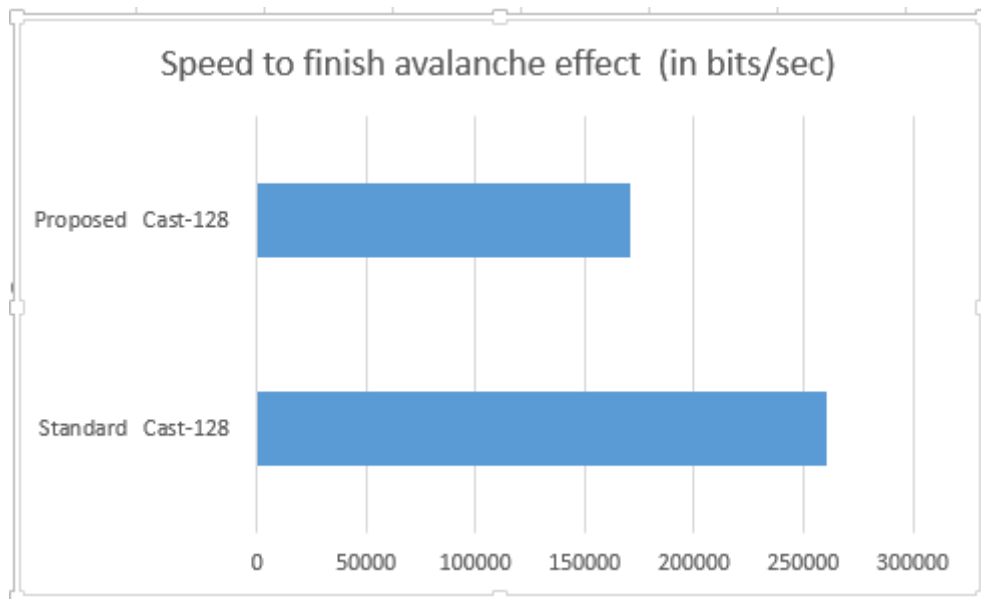


Figure 4.14: Results of speed taken on Cast-128 when plaintext was varied.

From Table 4.8 and Figure 4.15, the proposed Cast-128 yielded an improved avalanche effect compared to standard Cast-128 algorithm when key varied. The avalanche effect was increased from 50.1221% up to 50.1781% by using the proposed method. We recommend that the proposed Cast-128 algorithm can replace the standard Cast-128 algorithm when one wants to choose between the two, because it has an improved avalanche effect according to [143]. IoT uses standard Cast-128 algorithm to secure its prototypes. We recommend that the proposed Cast-128 algorithm can be used to secure IoT's prototypes when key varies. From Table 4.8 the standard Cast-128 algorithm has an avalanche effect which is low than the proposed Cast-128 algorithm. Standard Cast-128 algorithm has 50.1221% whereas proposed Cast-128 algorithm has 50.1781%. From Table 4.8 standard Cast-128 algorithm is fast compared to proposed Cast-128 algorithm. Standard Cast-128 takes 0.1239 seconds to encrypt whereas proposed Cast-128 takes 0.1251 seconds to encrypt.

Table 4.8: Results of standard and proposed Cast-128 when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD CAST-128	NO	NO	50.1221%	0.1239 sec	132235,6740 bits/sec
PROPOSED CAST-128	YES	YES	50.1781%	0.1251 sec	130967,2262 bits/sec

From Figure 4.15 the standard Cast-128 algorithm has an avalanche effect which is low than the proposed Cast-128 algorithm. Standard Cast-128 algorithm has 50.1221% whereas proposed Cast-128 algorithm has 50.1781%. When key was varied.

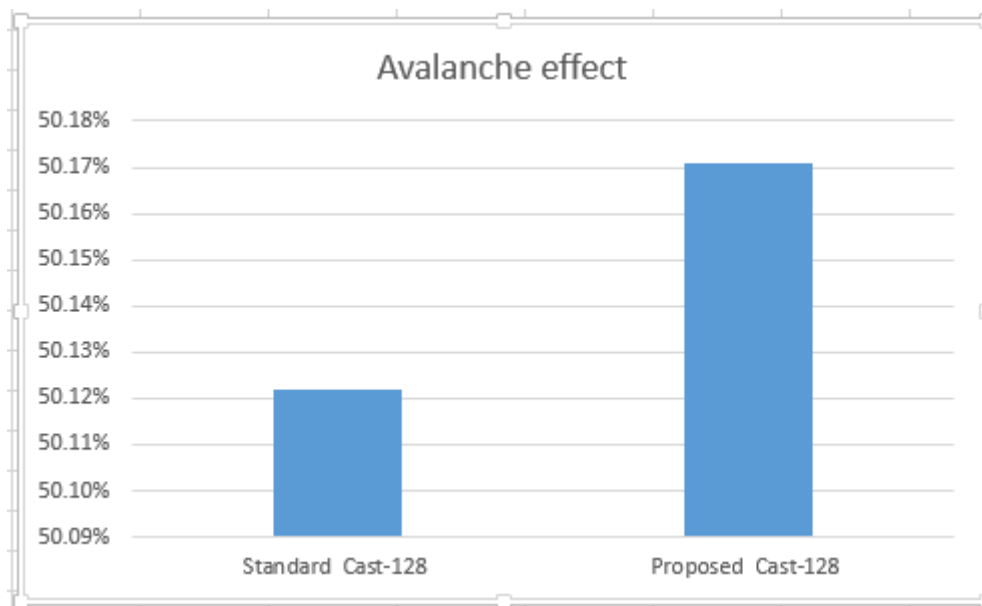


Figure 4.15: Results of avalanche effect on Cast-128 when key was varied.

From Table 4.8 and Figure 4.16 when the key is varied the proposed Cast-128 is slow. Then standard Cast-128 is still best in speed if the key is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that is fast when key varies. From Figure 4.16, speed of standard Cast-128 is 132235,6740 bits/sec, whereas of proposed Cast-128 is 130967,2262 bits/sec.

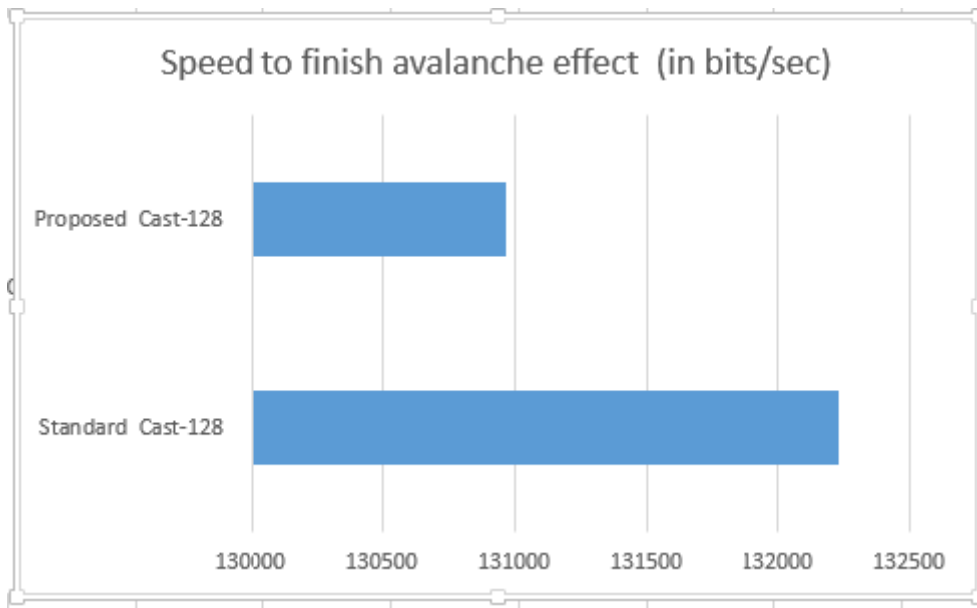


Figure 4.16: Results of speed taken on Cast-128 when key was varied.

4.2.5. Results 5 5: The Avalanche Effect on Clefia

From Table 4.9 and Figure 4.17 when the plaintext is varied the proposed Clefia yields a low avalanche effect compared to standard Clefia. Then standard Clefia is still best in avalanche effect if the plaintext is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that has an improved avalanche effect when plaintext is varied. From Table 4.9 the standard Clefia algorithm has an avalanche effect, which is improved than the proposed Clefia algorithm. Standard Clefia algorithm has 50.2808% whereas proposed Clefia algorithm has 49.8230%. From Table 4.9 standard Clefia algorithm is fast compared to proposed Clefia algorithm. Standard Clefia takes 0.1177 seconds to encrypt whereas proposed Clefia takes 0.1218 seconds to encrypt.

Table 4.9: Results of standard and proposed Clefia when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD CLEFIA	NO	NO	50.2808%	0.1177 sec	139201,3594 bits/sec
PROPOSED CLEFIA	YES	YES	49.8230%	0.1218sec	134515,5994 bits/sec

From Figure 4.17 the standard Clefia algorithm has an avalanche effect, which is improved than the proposed Clefia algorithm. Standard Clefia algorithm has 50.2808% whereas proposed Clefia algorithm has 49.8230%. When plaintext was varied.

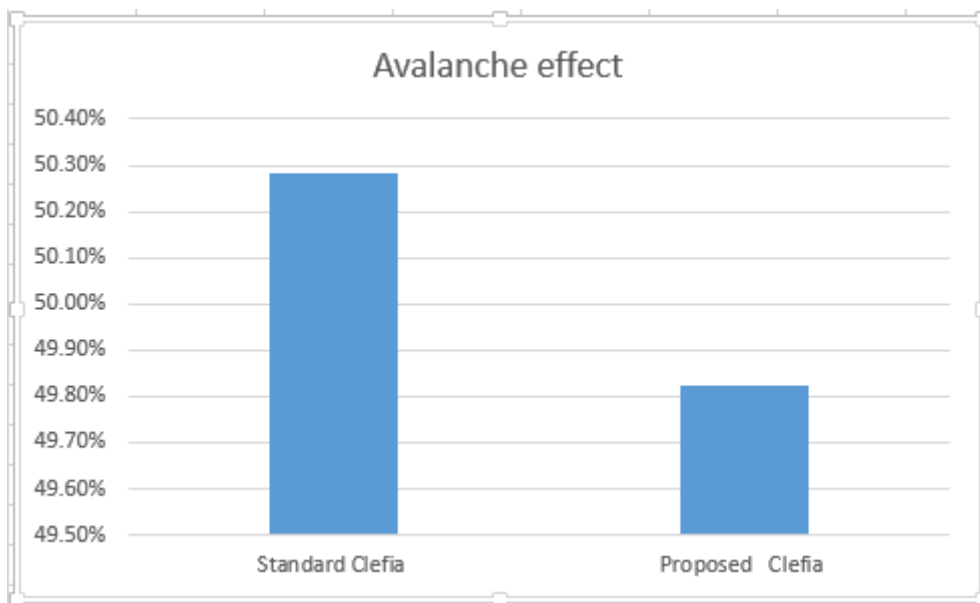


Figure 4.17: Results of avalanche effect on Clefia when plaintext was varied.

From Table 4.9 and Figure 4.18, when the plaintext is varied the proposed Clefia is slow. Then standard Clefia is still best in speed if the plaintext is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that is fast when plaintext varied. From Figure 4.18, speed of standard Clefia is 139201,3594 bits/sec, whereas of proposed Clefia is 134515,5994 bits/sec.

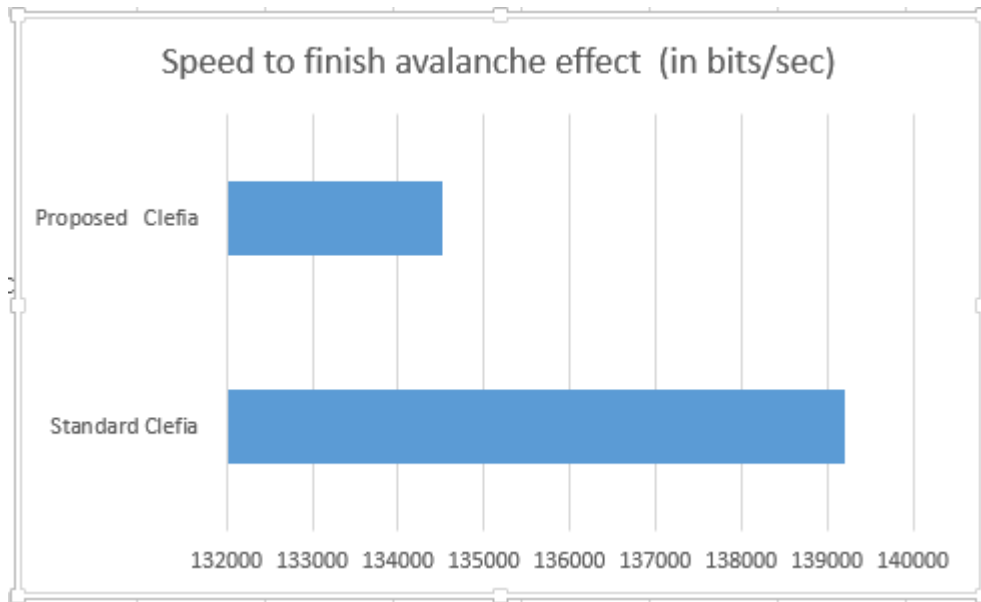


Figure 4.18: Results of speed taken on Clefia when plaintext was varied

From Table 4.10 and Figure 4.19, the proposed Clefia algorithm yielded an improved avalanche effect compared to standard Clefia algorithm when key varied. The avalanche effect increased from 49.9500% up to 50.2000 % by using the proposed method. We recommend that the proposed Clefia algorithm can replace the standard Clefia algorithm when one wants to choose between the two, because it has an improved avalanche effect. This means the proposed Clefia algorithm is more secure than standard Clefia algorithm [143], [6]. IoT uses Clefia to secure its health-care devices. Then it is suggested that proposed Clefia algorithm should be used to secure IoT's health-care devices. From Table 4.10 the standard Clefia algorithm has an avalanche effect which is low than the proposed Clefia algorithm. Standard Clefia algorithm has 49.9023% whereas proposed Clefia algorithm has 50.1587%. From Table 4.10 standard Clefia algorithm is slow compared to proposed Clefia algorithm. Standard Clefia takes 0.1455 seconds to encrypt whereas proposed Clefia takes 0.1219 seconds to encrypt.

Table 4.10: Results of standard and proposed Clefia when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD CLEFIA	NO	NO	49.9023%	0.1455 sec	112604,8110 bits/sec
PROPOSED CLEFIA	YES	YES	50.1587%	0.1219 sec	134405,2502 bits/sec

From Figure 4.19 the standard Clefia algorithm has an avalanche effect which is low than the proposed Clefia algorithm. Standard Clefia algorithm has 49.9023% whereas proposed Clefia algorithm has 50.1586%. When key was varied.

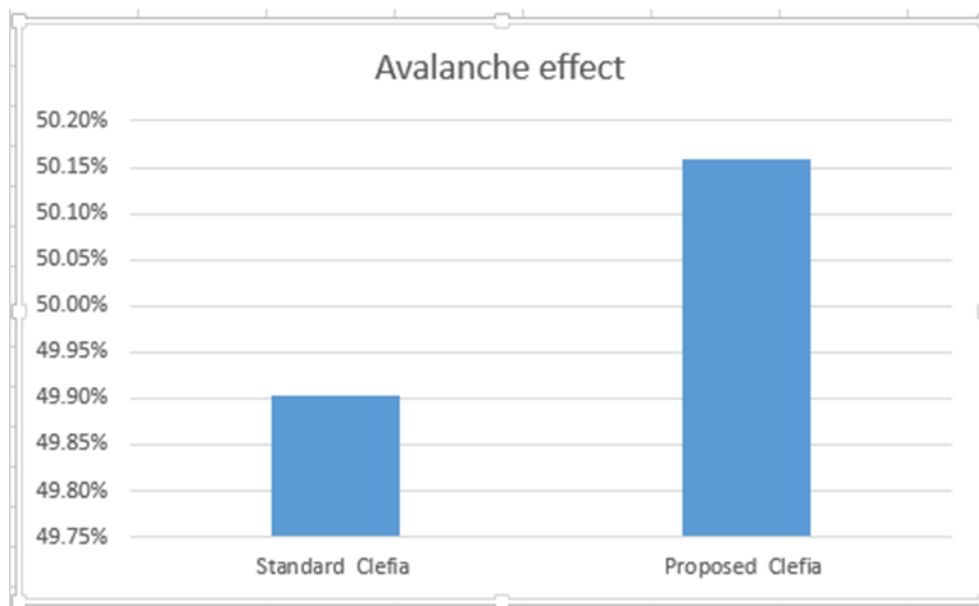


Figure 4.19: Results of avalanche effect on Clefia when key was varied.

From Table 4.10 and Figure 4.20, the proposed Clefia algorithm is faster than the standard Clefia algorithm. IoT uses standard Clefia secure its health-care devices. Then if a user want to vary key and encrypt or secure health-care devices on IoT, and wants to use fast algorithm like Clefia algorithm, then the proposed Clefia algorithm is a better option than standard one.

From Figure 4.20, speed of standard Clefia is 112604,8110 bits/sec, whereas of proposed Clefia is 134405,2502 bits/sec.

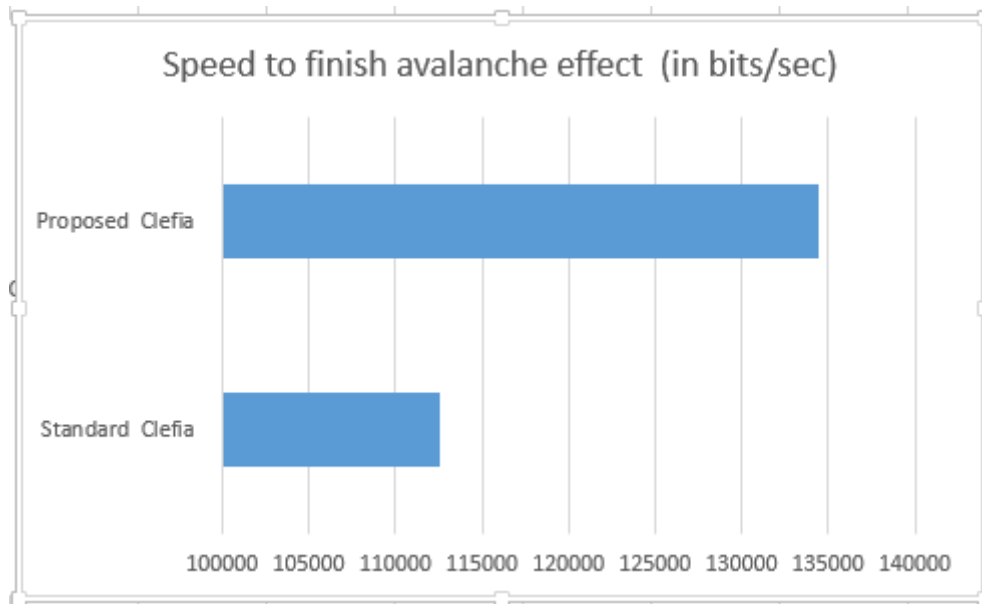


Figure 4.20: Results of speed taken on Clefia when key was varied.

4.2.6. Results 6: The Avalanche Effect on DES

From Table 4.11 and Figure 4.21, the proposed DES algorithm yielded low avalanche effect compared to standard DES algorithm when plaintext varied. Then standard DES algorithm is still best in avalanche effect if the plaintext is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that has an improved avalanche effect when plaintext is varied. From Table 4.11 the standard DES algorithm has an avalanche effect, which is improved than the proposed DES algorithm. Standard DES algorithm has 62.8660% whereas proposed DES algorithm has 58.8379%. From Table 4.11 standard DES algorithm is slow compared to proposed DES algorithm. Standard DES takes 0.0585 seconds to encrypt whereas proposed DES takes 0.05773 seconds to encrypt.

Table 4.11: Results of standard and proposed DES when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD DES	NO	NO	62.8662%	0.0585 sec	70017,0940 bits/sec
PROPOSED DES	YES	YES	58.8379%	0.05773 sec	70950,9787 bits/sec

From Table 4.21 the standard DES algorithm has an avalanche effect, which is improved than the proposed DES algorithm. Standard DES algorithm has 62.8662% whereas proposed DES algorithm has 58.8379%. When plaintext was varied.

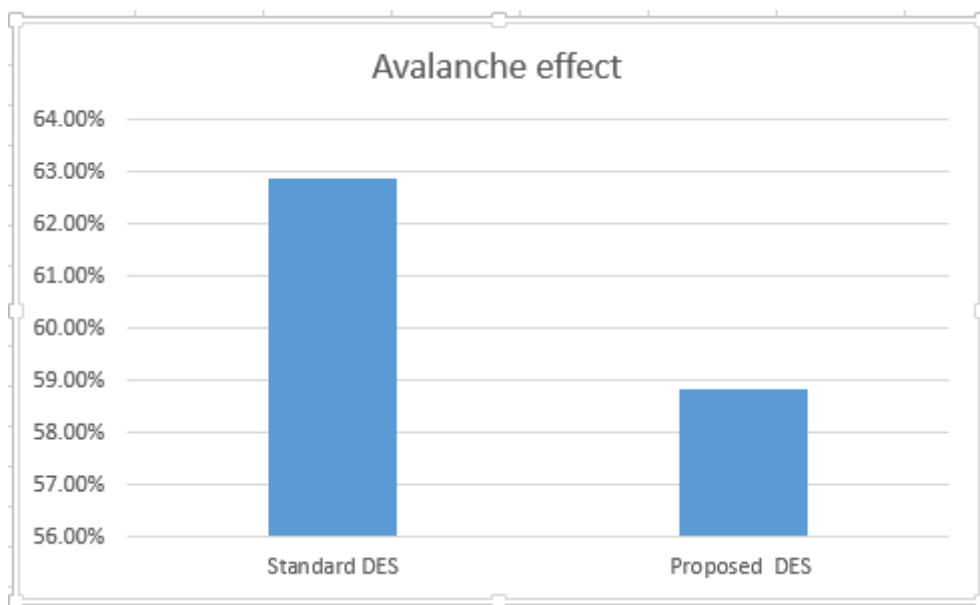


Figure 4.21: Results of avalanche effect on DES when plaintext was varied.

From Table 4.11 and Figure 4.22, the proposed DES algorithm is faster than the standard DES algorithm. IoT uses standard DES algorithm to secure its prototypes and many devices. Then if a user want to vary plaintext and encrypt IoT’s prototypes and devices, and wants to use fast algorithm like DES, then the proposed DES is a better option than standard one. From Figure 4.22, speed of standard DES is 70017,0940 bits/sec, whereas of proposed DES is 70950,9787 bits/sec.

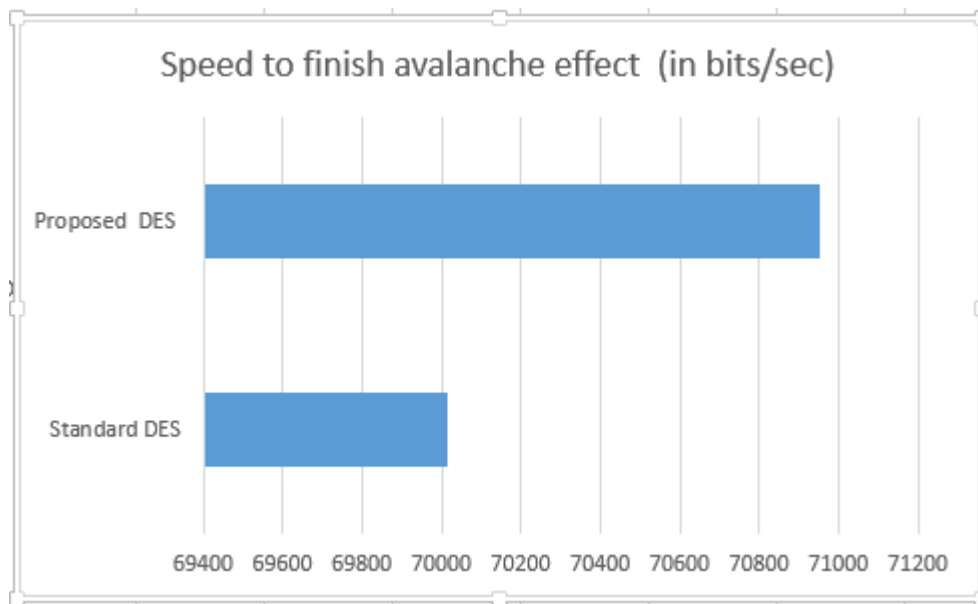


Figure 4.22: Results of speed taken on DES when plaintext was varied.

From Table 4.12 and Figure 4.23, the proposed DES algorithm yielded an improved avalanche effect compared to standard DES algorithm when key varied. The avalanche effect was increased from 43.8721% up to 44.2139% by using the proposed method. We recommend that the proposed DES algorithm replace the standard DES algorithm when one wants to choose between the two, because it has an improved avalanche effect according to [143], [6]. Standard DES algorithm is mostly used on IoT. This standard DES algorithm should be replaced from IoT because it has low avalanche effect compared to the proposed DES algorithm when the key varies. From Table 4.12 the standard DES algorithm has an avalanche effect which is low than the proposed DES algorithm. Standard DES algorithm has 43.8721% whereas proposed DES algorithm has 44.2139%. From Table 4.12 standard DES algorithm is fast compared to proposed DES algorithm. Standard DES takes 0.0616 seconds to encrypt whereas proposed DES takes 0.0635 seconds to encrypt.

Table 4.12: Results of standard and proposed DES when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD DES	NO	NO	43.8721%	0.06162 sec	66471,9247 bits/sec
PROPOSED DES	YES	YES	44.2139%	0.0635 sec	64503,9370 bits/sec

From Table 4.23 the standard DES algorithm has an avalanche effect which is low than the proposed DES algorithm. Standard DES algorithm has 43.8721% whereas proposed DES algorithm has 44.2139%. When key was varied.

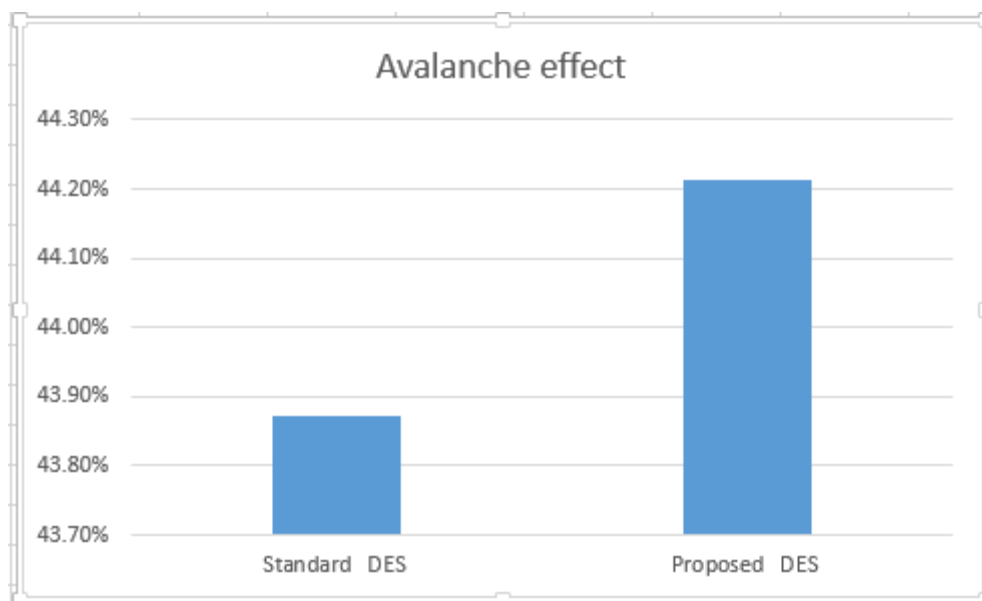


Figure 4.23: Results of avalanche effect on DES when key was varied.

From Table 4.12 and Figure 4.24, when the key is varied the proposed DES is slow. Then standard DES is still best in speed if the key is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm, which is faster when key varies. From Figure 4.24, speed of standard DES is 66471,9247 bits/sec, whereas of proposed DES is 64503,9370 bits/sec.

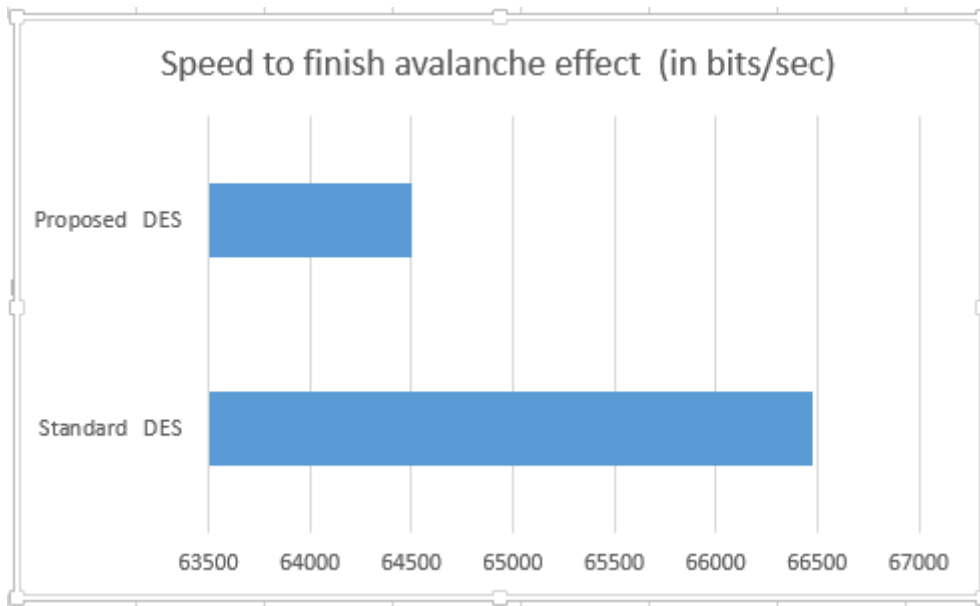


Figure 4.24: Results of speed taken on DES when key was varied.

4.2.7. Results 7: The Avalanche Effect on MMB

From Table 4.13 and Figure 4.25, when the plaintext is varied the proposed MMB algorithm has low avalanche effect compared to standard MMB algorithm. Then standard MMB is still best in avalanche effect if the plaintext is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that has an improved avalanche effect when plaintext is varied. From Table 4.13 the standard MMB algorithm has an avalanche effect, which is improved than the proposed MMB algorithm. Standard MMB algorithm has 49.7742% whereas proposed MMB algorithm has 49.7498%. From Table 4.13 standard MMB algorithm is slow compared to proposed MMB algorithm. Standard MMB takes 0.2415 seconds to encrypt whereas proposed MMB takes 0.2320 seconds to encrypt.

Table 4.13: Results of standard and proposed MMB when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD MMB	NO	NO	49.7742%	0.2415 sec	67842,6501 bits/sec
PROPOSED MMB	YES	YES	49.7498%	0.232 sec	70620,6897 bits/sec

From Table 4.25 the standard MMB algorithm has an avalanche effect, which is improved than the proposed MMB algorithm. Standard MMB algorithm has 49.7742% whereas proposed MMB algorithm has 49.7498%. When plaintext was varied.

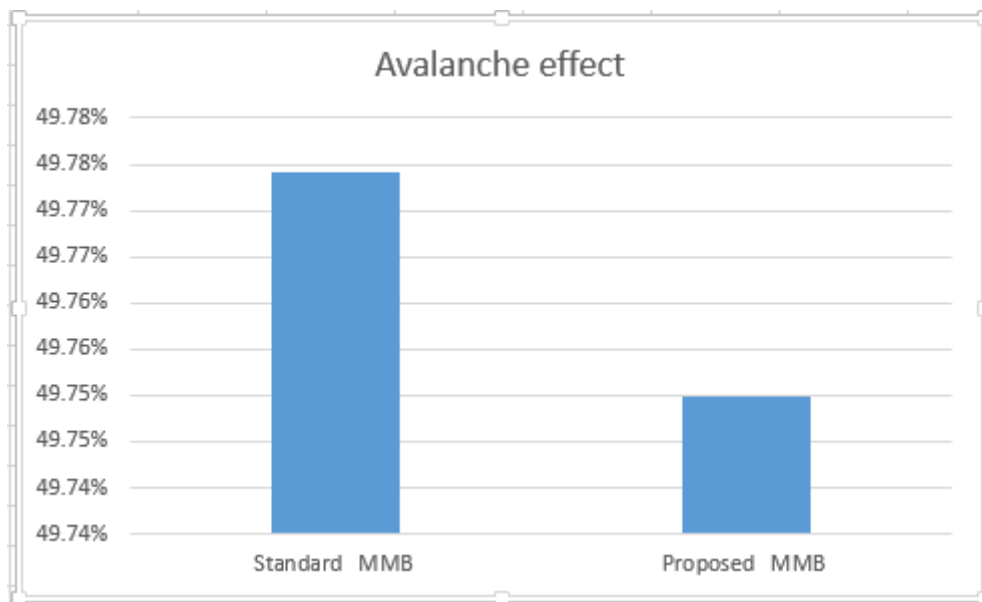


Figure 4.25: Results of avalanche effect on MMB when plaintext was varied.

From Table 4.13 and Figure 4.26, the proposed MMB algorithm is faster than the standard MMB algorithm. IoT uses standard MMB algorithm to secure its software applications. Then if a user want to vary plaintext and encrypt IoT’s software applications and wants to use fast algorithm like MMB, then the proposed MMB algorithm is the best to choose. From Figure 4.26, speed of standard MMB is 67842,6501 bits/sec, whereas of proposed MMB is 70620,6897 bits/sec.

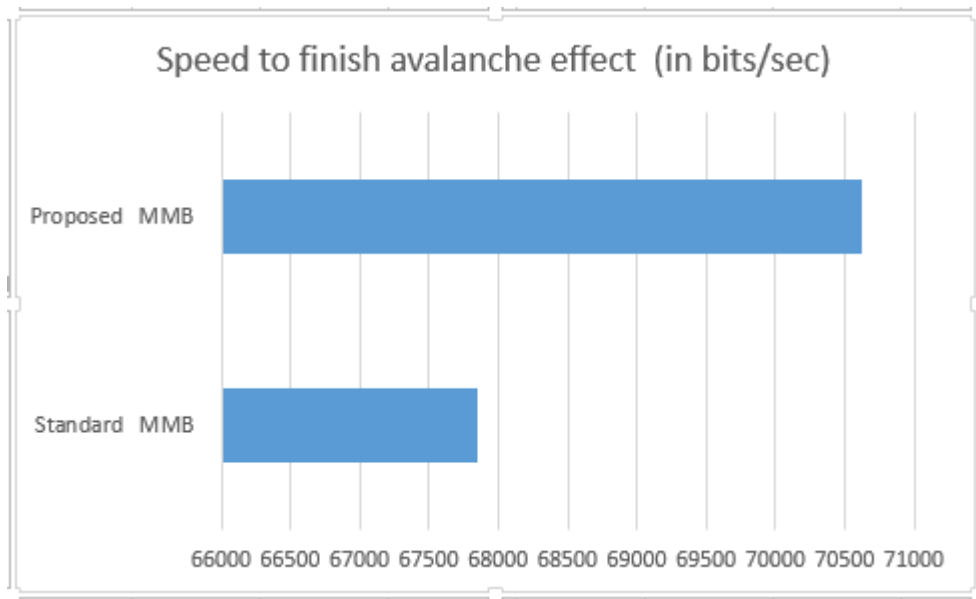


Figure 4.26: Results of speed taken on MMB when plaintext was varied.

From Table 4.14 and Figure 4.27 when the key is varied the proposed MMB algorithm has low avalanche effect compared to standard MMB algorithm. Then standard MMB algorithm is still best in avalanche effect if the plaintext is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that has an improved avalanche effect when plaintext is varied. From Table 4.14 the standard MMB algorithm has an avalanche effect, which is improved than the proposed MMB algorithm. Standard MMB algorithm has 49.6765% whereas proposed MMB algorithm has 49.6399%. From Table 4.14 both standard and proposed MMB algorithm took exact same time to encrypt. Both standard and proposed MMB took 0.1216 second to encrypt.

Table 4.14: Results of standard and proposed MMB when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD MMB	NO	NO	49.6765%	0.1216 sec	134736,8421 bits/sec
PROPOSED MMB	YES	YES	49.6399%	0.1216 sec	134736,8421 bits/sec

From Table 4.27 the standard MMB algorithm has an avalanche effect which is improved than the proposed MMB algorithm. Standard MMB algorithm has 49.6765% whereas proposed MMB algorithm has 49.6399%. When key was varied.

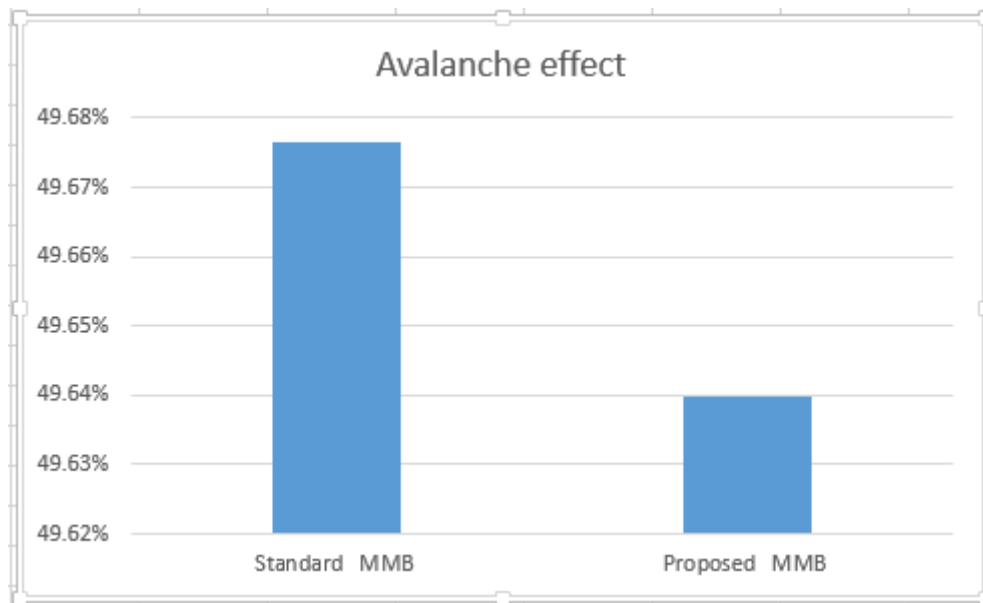


Figure 4.27: Results of avalanche effect on MMB when key was varied.

From Table 4.14 and Figure 4.28, both proposed and standard MMB algorithms yielded equal speed and time when key varied. Therefore, user can pick one between the two it does not matter. Only if a user focus on speed and when the key varies. Standard and proposed MMB has the same of 134736,8421 bits/sec.

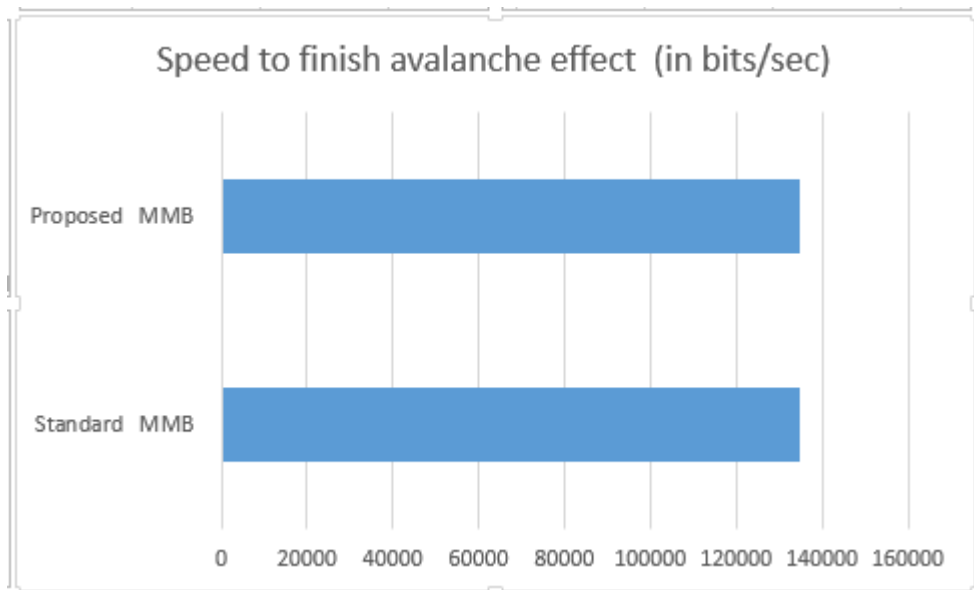


Figure 4.28: Results of speed taken on MMB when key was varied.

4.2.8. Results 8: The Avalanche Effect on RC5

From Table 4.15 and Figure 4.29, the proposed RC5 algorithm yielded an improved avalanche effect compared to standard RC5 algorithm when plaintext varied. The avalanche effect increased from 76.2% up to 76.9 % after the proposed method was implemented. This means that the proposed RC5 algorithm can replace the standard RC5 algorithm when one wants to choose between the two because it has an improved avalanche effect. Algorithm that has an improved avalanche effect has high security [143], [6]. IoT uses standard RC5 algorithm to secure its Mica2 hardware (base station of IoT). It is recommended to use the proposed RC5 algorithm from IoT's Mica2 hardware, only when plaintext varies. We suggest that the proposed one have to be used on Mica2 hardware because it has an improved avalanche effect. From Table 4.15 the standard RC5 algorithm has an avalanche effect which is low than the proposed RC5 algorithm. Standard RC5 algorithm has 76.1719% whereas proposed RC5 algorithm has 76.9043%. From Table 4.15 standard RC5 algorithm is slow compared to proposed RC5 algorithm. Standard RC5 takes 0.0606 seconds to encrypt whereas proposed RC5 takes 0.0599 seconds to encrypt.

Table 4.15: Results of standard and proposed RC5 when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD RC5	NO	NO	76.1719%	0.0606 sec	68266,6667 bits/sec
PROPOSED RC5	YES	YES	76.9043%	0.0599 sec	68380,6344 bits/sec

From Figure 4.29 the standard RC5 algorithm has an avalanche effect which is low than the proposed RC5 algorithm. Standard RC5 algorithm has 76.1719% whereas proposed RC5 algorithm has 76.9043%. When plaintext was varied.

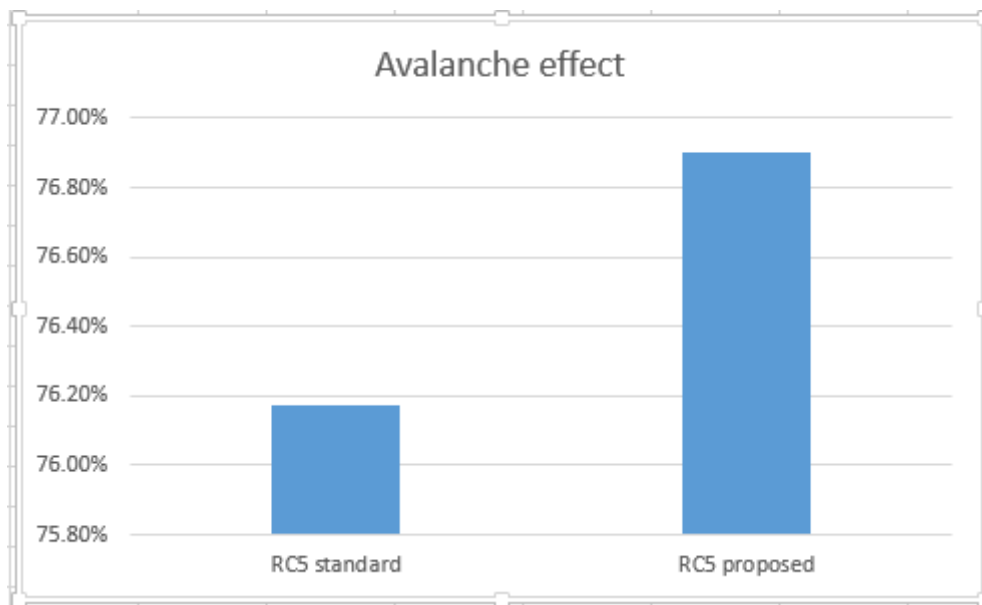


Figure 4.29: Results of avalanche effect on RC5 when plaintext was varied.

From Table 4.15 and Figure 4.30, the proposed RC5 algorithm is faster than the standard RC5 algorithm. IoT uses standard RC5 algorithm to encrypt its Mica2 hardware. Then if a user want to vary plaintext and encrypt the IoT's Mica2 hardware and wants to use fast algorithm like

RC5, then the proposed RC5 is the best to choose. From Figure 4.30, speed of standard RC5 is 68266,6667 bits/sec, whereas of proposed RC5 is 68380,6344 bits/sec.

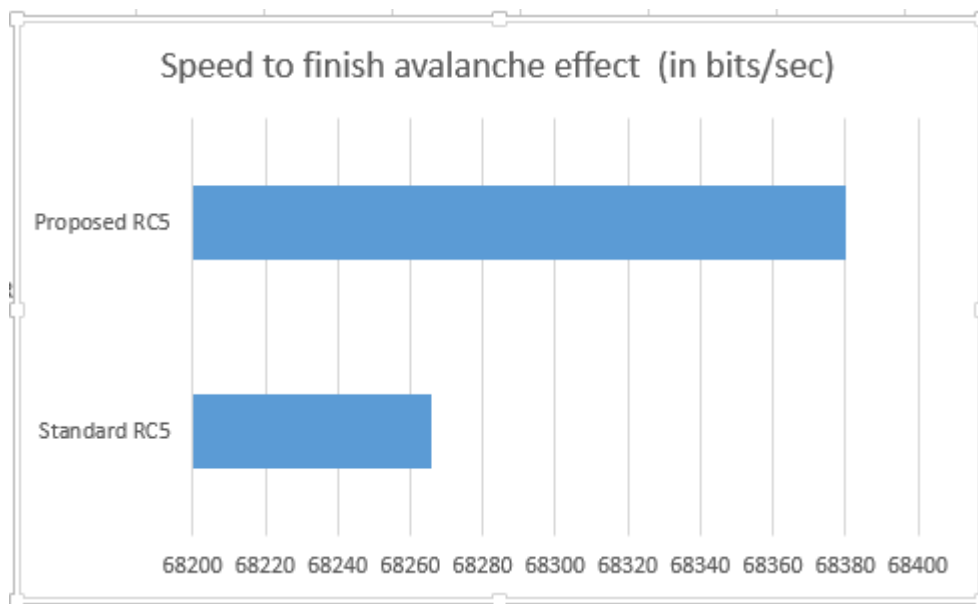


Figure 4.30: Results of speed taken on RC5 when plaintext was varied.

From Table 4.16 and Figure 4.31, the proposed RC5 algorithm yielded an improved avalanche effect compared to standard RC5 algorithm when key varied. The avalanche effect was increased from 49.1821% up to 49.7923 % by using the proposed method. We recommend that the proposed RC5 algorithm can replace the standard RC5 algorithms when one wants to choose between the two, because it has an improved avalanche effect, only when the key varies. IoT should use the proposed RC5 algorithm to secure its Mica2 hardware only when the key is varied. Then we suggested that the standard RC5 algorithm should be removed from Mica2 hardware and the proposed RC5 be used to provide enhanced security as compared to the standard one on Mica2 hardware. From Table 4.16 the standard RC5 algorithm has an avalanche effect which is low than the proposed RC5 algorithm. Standard RC5 algorithm has 49.1821% whereas proposed RC5 algorithm has 49.7923%. From Table 4.16 standard RC5 algorithm is fast compared to proposed RC5 algorithm. Standard RC5 takes 0.1165 seconds to encrypt whereas proposed RC5 takes 0.1325 seconds to encrypt.

Table 4.16: Results of standard and proposed RC5 when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD RC5	NO	NO	49.1821%	0.1165 sec	140635,1931 bits/sec
PROPOSED RC5	YES	YES	49.7923%	0.1325 sec	123652,8302 bits/sec

From Figure 4.31 the standard RC5 algorithm has an avalanche effect which is low than the proposed RC5 algorithm. Standard RC5 algorithm has 49.1821% whereas proposed RC5 algorithm has 49.7923%. When key was varied.

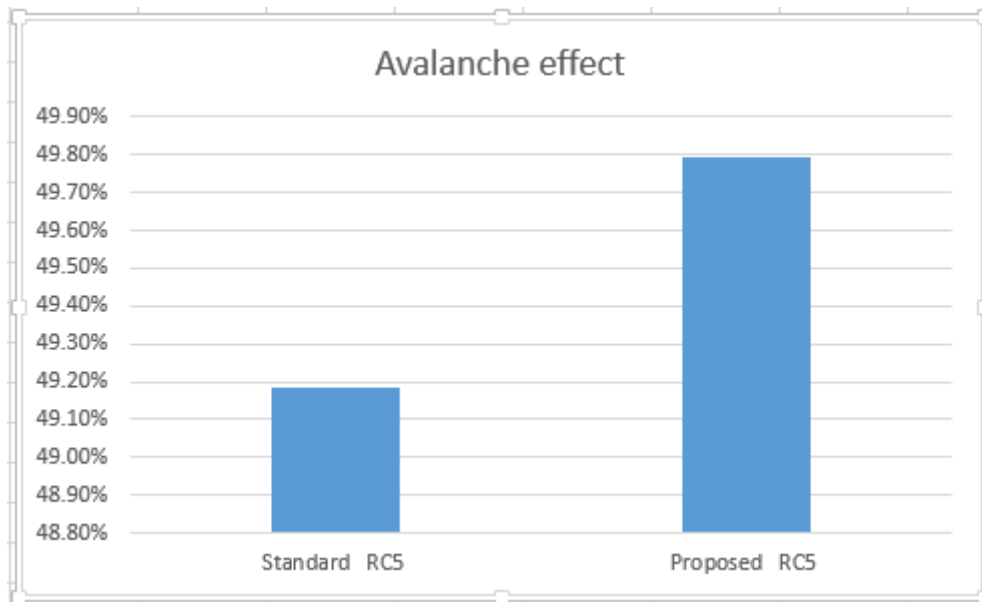


Figure 4.31: Results of avalanche effect on RC5 when key was varied.

From Table 4.16 and Figure 4.32, when the key is varied the proposed RC5 is slow. Then standard RC5 is still best in speed if the key is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that is fast when key varies. From Figure 4.32, speed of standard RC5 is 140635,1931 bits/sec, whereas of proposed RC5 is 123652,8302 bits/sec.

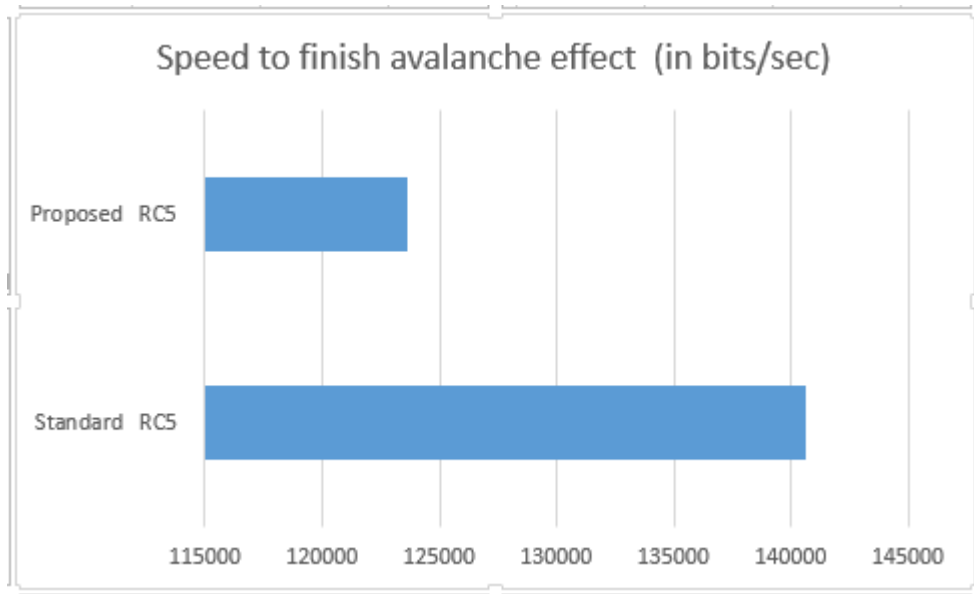


Figure 4.32: Results of speed taken on RC5 when key was varied.

4.2.9. Results 9: The Avalanche Effect on Serpent

From Table 4.17 and Figure 4.33, when the plaintext is varied the proposed Serpent algorithm has low avalanche effect compared to standard Serpent algorithm. Then standard Serpent algorithm is still best in avalanche effect if the plaintext is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that has an improved avalanche effect when plaintext is varied. From Table 4.17 the standard Serpent algorithm has an avalanche effect which is improved than the proposed Serpent algorithm. Standard Serpent algorithm has 50.3845% whereas proposed Serpent algorithm has 49.7986%. From Table 4.17 standard Serpent algorithm is fast compared to proposed Serpent algorithm. Standard Serpent takes 0.1215 seconds to encrypt whereas proposed Serpent takes 0.1250 seconds to encrypt.

Table 4.17: Results of standard and proposed Serpent when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD SERPENT	NO	NO	50.3845%	0.1215 sec	134847,7367 bits/sec
PROPOSED SERPENT	YES	YES	49.7986%	0.125 sec	131072,0000 bits/sec

From Figure 4.33 the standard Serpent algorithm has an avalanche effect, which is improved than the proposed Serpent algorithm. Standard Serpent algorithm has 50.3845% whereas proposed Serpent algorithm has 49.7986%. When plaintext was varied.

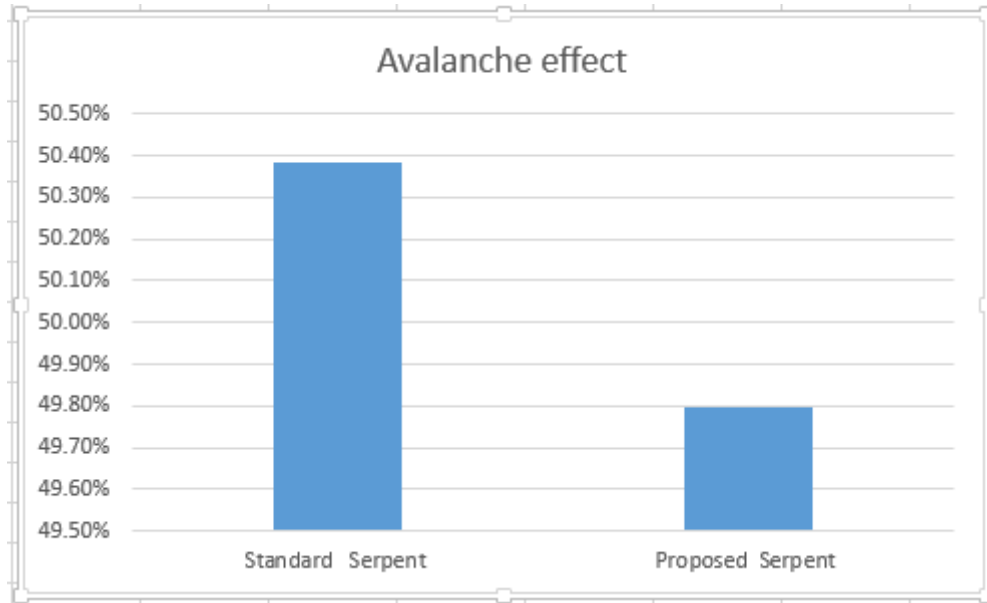


Figure 4.33: Results of avalanche effect on Serpent when plaintext was varied.

From Table 4.17 and Figure 4.34, when the plaintext is varied the proposed Serpent is slow. Then standard Serpent is still best in speed if the plaintext is varied. Therefore, there is no need to replace it from internet IoT if the user wants an algorithm, which is faster when plaintext varied. From Figure 4.34, speed of standard Serpent is 134847,7366 bits/sec, whereas of proposed Serpent is 131072.0000 bits/sec.

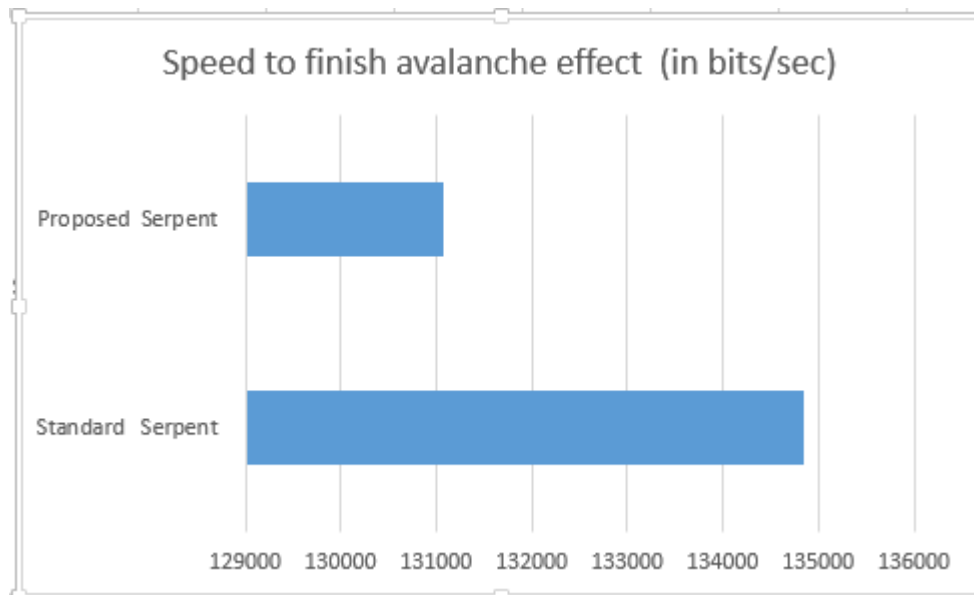


Figure 4.34: Results of speed taken on Serpent when plaintext was varied.

From Table 4.18 and Figure 4.35, the proposed Serpent algorithm yielded an improved avalanche effect compare to standard Serpent algorithm when key varied. The avalanche effect was increased from 49.8656% up to 50.5341% by using the proposed method. We recommend that the proposed Serpent algorithm can replace the standard Serpent algorithm when one wants to choose between the two. Algorithm that has an improved avalanche effect has high security [143], [6]. IoT uses standard Serpent algorithm to secure its sensors. Therefore, the proposed Serpent algorithm should be used to secure IoT's sensors, only when the key varies. From Table 4.18 the standard Serpent algorithm has an avalanche effect which is low than the proposed Serpent algorithm. Standard Serpent algorithm has 49.8656% whereas proposed Serpent algorithm has 50.5341%. From Table 4.18 standard Serpent algorithm is fast compared to proposed Serpent algorithm. Standard Serpent takes 0.2714 seconds to encrypt whereas proposed Serpent takes 0.2744 seconds to encrypt.

Table 4.18: Results of standard and proposed Serpent when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD SERPENT	NO	NO	49.8657%	0.2714 sec	241473,8394 bits/sec
PROPOSED SERPENT	YES	YES	50.5341%	0.2744 sec	238833,8192 bits/sec

From Figure 4.35 the standard Serpent algorithm has an avalanche effect which is low than the proposed Serpent algorithm. Standard Serpent algorithm has 49.8657% whereas proposed Serpent algorithm has 50.5341%. When key was varied.

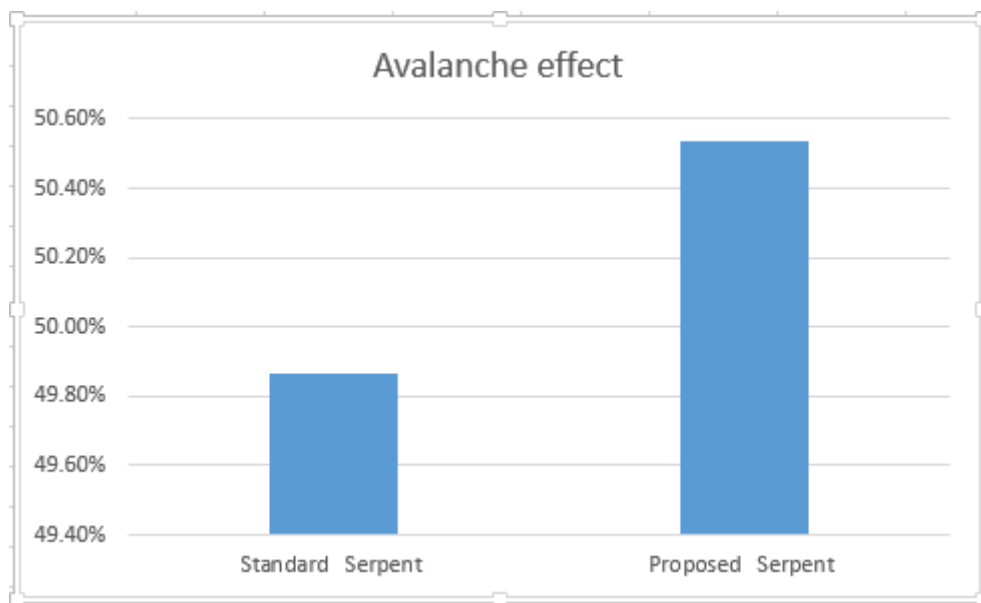


Figure 4.35: Results of avalanche effect on Serpent when key was varied.

From Table 4.18 and Figure 4.36, when the key is varied the proposed Serpent algorithm is slow. Then standard Serpent algorithm is still best in speed if the key is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that is fast when key varies. From Figure 4.36, speed of standard Serpent is 241473,8395 bits/sec, whereas of proposed Serpent is 238833,8192 bits/sec.

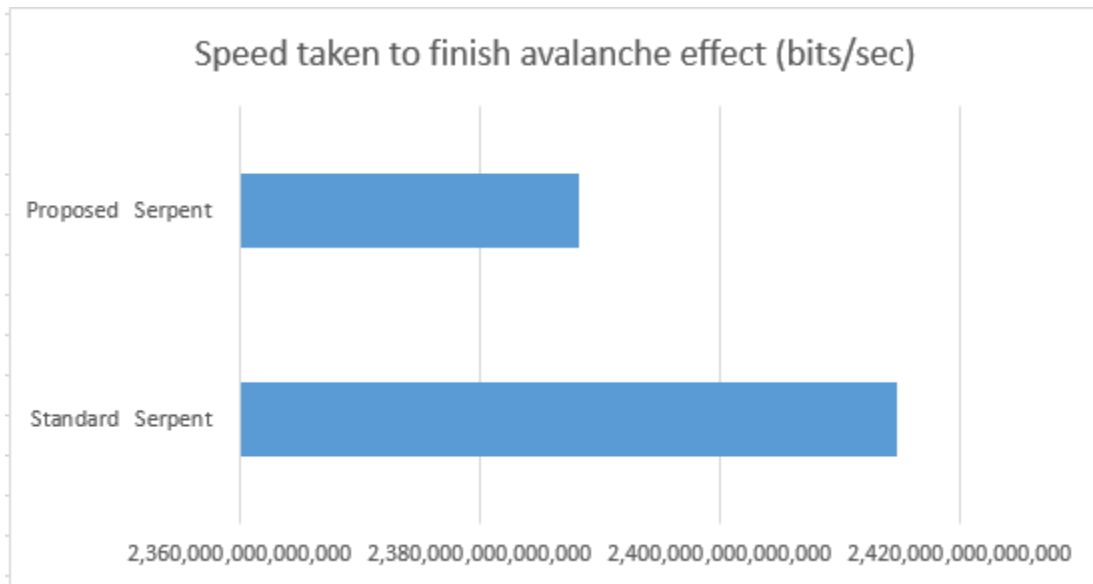


Figure 4.36: Results of speed taken on Serpent when key was varied.

4.2.10. Results 10: The Avalanche Effect on Skipjack

From Table 4.19 and Figure 4.37, the proposed Skipjack algorithm yielded an improved avalanche effect compared to standard Skipjack algorithm when plaintext varied. The avalanche effect was increased from 48.7793% up to 49.2188% by using the proposed method. We recommend that the proposed Skipjack algorithm replace the standard Skipjack algorithm when one wants to choose between the two, because it has an improved avalanche effect. Algorithm that has an improved avalanche effect has high security [143], [6]. IoT uses standard Skipjack algorithm to secure its Mica2 hardware. Then standard Skipjack algorithm should be replaced by the proposed Skipjack algorithm from substation IoT's Mica2 hardware the proposed Skipjack has an improved avalanche effect. From Table 4.19 the standard Skipjack algorithm has an avalanche effect which is low than the proposed Skipjack algorithm. Standard Skipjack algorithm has 48.7793% whereas proposed Skipjack algorithm has 49.2188%. From Table 4.19 standard Skipjack algorithm is fast compared to proposed Skipjack algorithm. Standard Skipjack takes 0.0605 seconds to encrypt whereas proposed Skipjack takes 0.0647 seconds to encrypt.

Table 4.19: Results of standard and proposed Skipjack when plaintext was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD SKIPJACK	NO	NO	48.7793%	0.0605 sec	67680,1058 bits/sec
PROPOSED SKIPJACK	YES	YES	49.2186%	0.0647 sec	63268,45845 bit/sec

From Figure 4.37 the standard Skipjack algorithm has an avalanche effect which is low than the proposed Skipjack algorithm. Standard Skipjack algorithm has 48.7793% whereas proposed Skipjack algorithm has 49.2188%. When plaintext was varied.

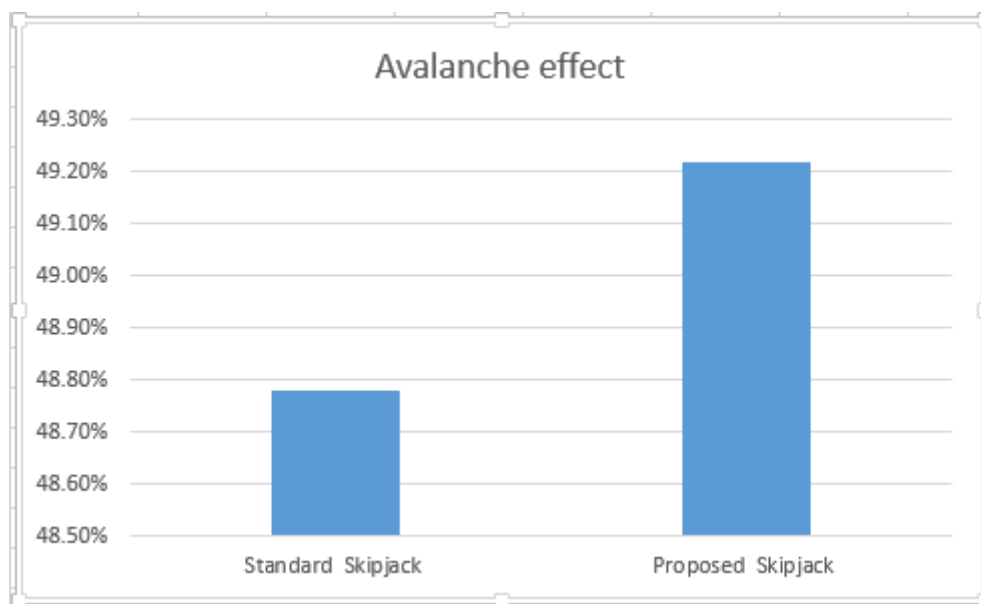


Figure 4.37: Results of avalanche effect on Skipjack when plaintext was varied.

From Table 4.19 and Figure 4.38, when the plaintext is varied the proposed Skipjack algorithm is slow compared to standard Skipjack algorithm. Then standard Skipjack algorithm is still best in speed if the plaintext is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that is fast when plaintext varied. From Figure 4.38, speed of standard Skipjack is 67680,1058 bits/sec whereas of proposed Skipjack is 63268,4585 bit/sec.

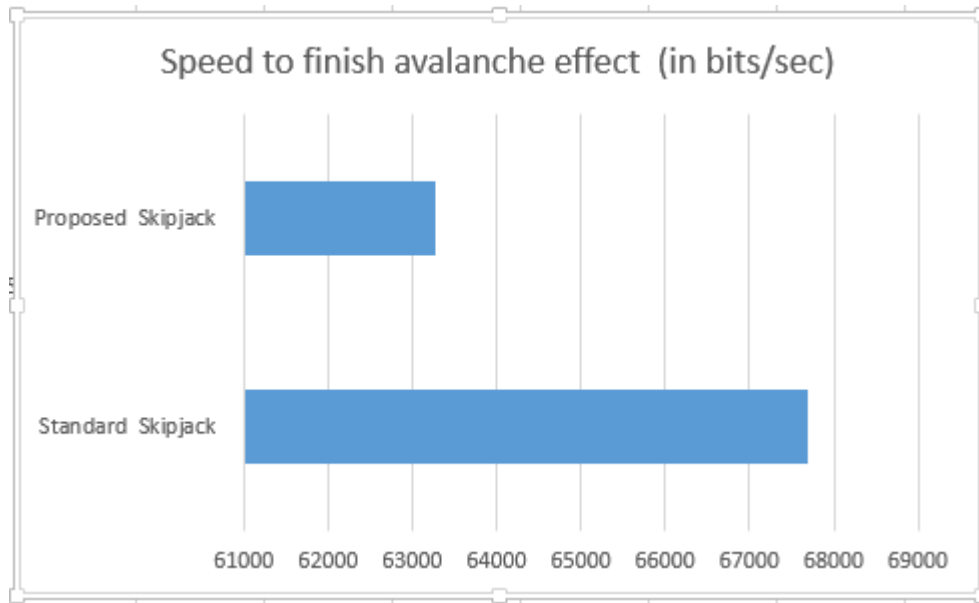


Figure 4.38: Results of speed taken on Skipjack when plaintext was varied.

From Table 4.20 and Figure 4.39, when the key is varied the proposed Skipjack algorithm has low avalanche effect compared to standard Skipjack algorithm. Then standard Skipjack algorithm is still the best in avalanche effect if the plaintext is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that has an improved avalanche effect when plaintext is varied. From Table 4.20 the standard Skipjack algorithm has an avalanche effect, which is improved than the proposed Skipjack algorithm. Standard Skipjack algorithm has 62.5732% whereas proposed Skipjack algorithm has 61.6211%. From Table 4.20 standard Skipjack algorithm is fast compared to proposed Skipjack algorithm. Standard Skipjack takes 0.0744 seconds to encrypt whereas proposed Skipjack takes 0.0758 seconds to encrypt.

Table 4.20: Results of standard and proposed Skipjack when key was varied.

Algorithm	Initial vector	Final vector	Avalanche effect	Time taken to finish avalanche effect	Speed taken to finish avalanche effect
STANDARD SKIPJACK	NO	NO	62.5732%	0.0744 sec	67680,1058 bits/sec
PROPOSED SKIPJACK	YES	YES	61.6211%	0.0758 sec	63268,4584 bit/sec

From Figure 4.39 the standard Skipjack algorithm has an avalanche effect, which is improved than the proposed Skipjack algorithm. Standard Skipjack algorithm has 62.5732% whereas proposed Skipjack algorithm has 61.6211%. When key was varied.

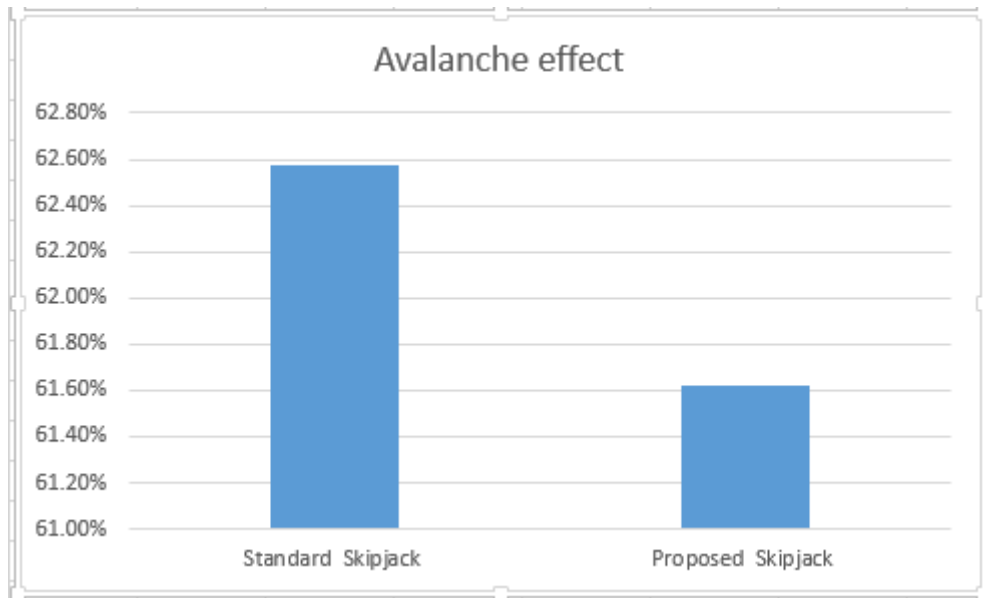


Figure 4.39: Results of avalanche effect on Skipjack when key was varied.

From Table 4.20 and Figure 4.40, when the key is varied the proposed Skipjack algorithm is slow compared to standard Skipjack algorithm. Then standard Skipjack algorithm is still best in speed if the key is varied. Therefore, there is no need to replace it for IoT if the user wants an algorithm that is fast when key varies. From Figure 4.40, speed of standard Skipjack is 67680,1058 bits/sec, whereas of proposed Skipjack is 63268,4584 bit/sec.

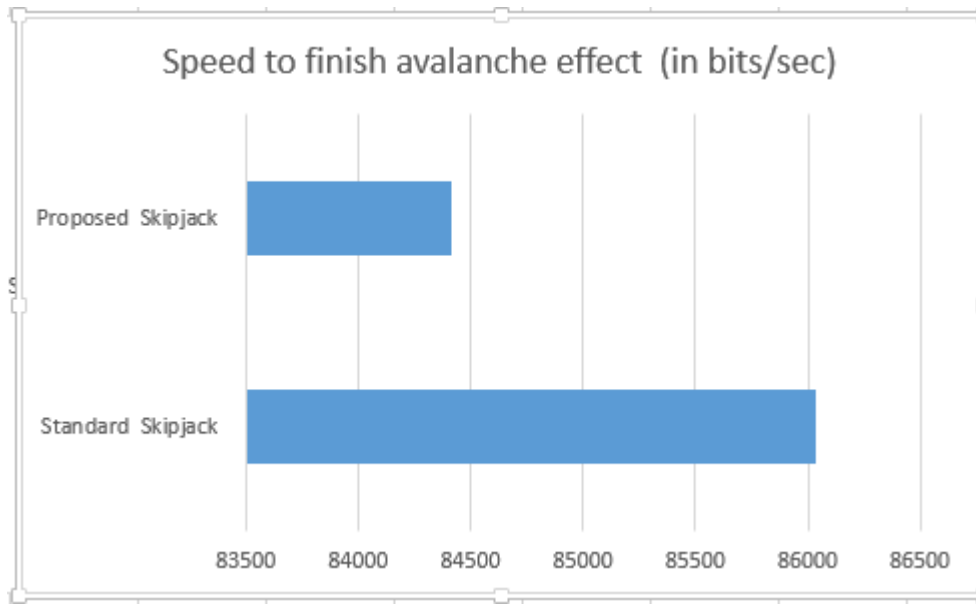


Figure 4.40: Results of speed taken on Skipjack when key was varied.

4.2.11. Results 11: The Speed and Avalanche Effect on All Ten Algorithms

From the Figure 4.41 and Table 4.21, it is clear that the proposed method worked very sufficiently when we compare all algorithms put on test. We found that with the proposed RC5, Skipjack, Cast-128 and Camellia algorithms have improved avalanche effect compared their standard algorithms when plaintext was varied. The proposed RC5 has a value of avalanche effect of 76.9043%, it is heightened in red on Table 4.21. The proposed Skipjack has a value of avalanche effect of 49.2188%, also heightened in red on Table 4.21. The proposed Cast-128 has a value of avalanche effect of 49.3164%, it is heightened in red on Table 4.21. The proposed Camellia has a value of avalanche effect of 50.0977%, also heightened in red on Table 4.21. Therefore the proposed RC5, Skipjack, Cast-128 and Camellia algorithms must be selected if one wants the algorithm with an improved avalanche effect when plaintext is varied compared to their standard algorithms implemented on IoT. Algorithm that has an improved avalanche effect has high security [143], [6]. Therefore, the proposed method worked positively according to our results from Figure 4.41.

Table 4.21: Results of avalanche effect of all algorithms tested when plaintext is varied.

Algorithm Tested	Avalanche effect of standard algorithm	Avalanche effect of proposed algorithm
AES	49.7925 %	49.60327 %
BLOWFISH	50.5615%	48.33984%
CAMELLIA	49.4690%	50.0977%
CAST-128	48.8281%	49.3164%
CLEFIA	50.28076%	49.8230%
DES	62.8662%	58.8379%
MMB	49.7742%	49.7498%
RC5	76.1719%	76.9043%
SERPENT	50.3845%	49.7988%
SKIPJACK	48.77930%	49.21875%

The proposed RC5 has a value of avalanche effect of 76.9043%, is presented graphically on Figure 4.41. The proposed Skipjack has a value of avalanche effect of 49.2188%, also heightened in red on Table 4.21. The proposed Cast-128 has a value of avalanche effect of 49.3164%, is presented graphically on Figure 4.41. The proposed Camellia has a value of avalanche effect of 50.0977%, also heightened in red on Table 4.21.

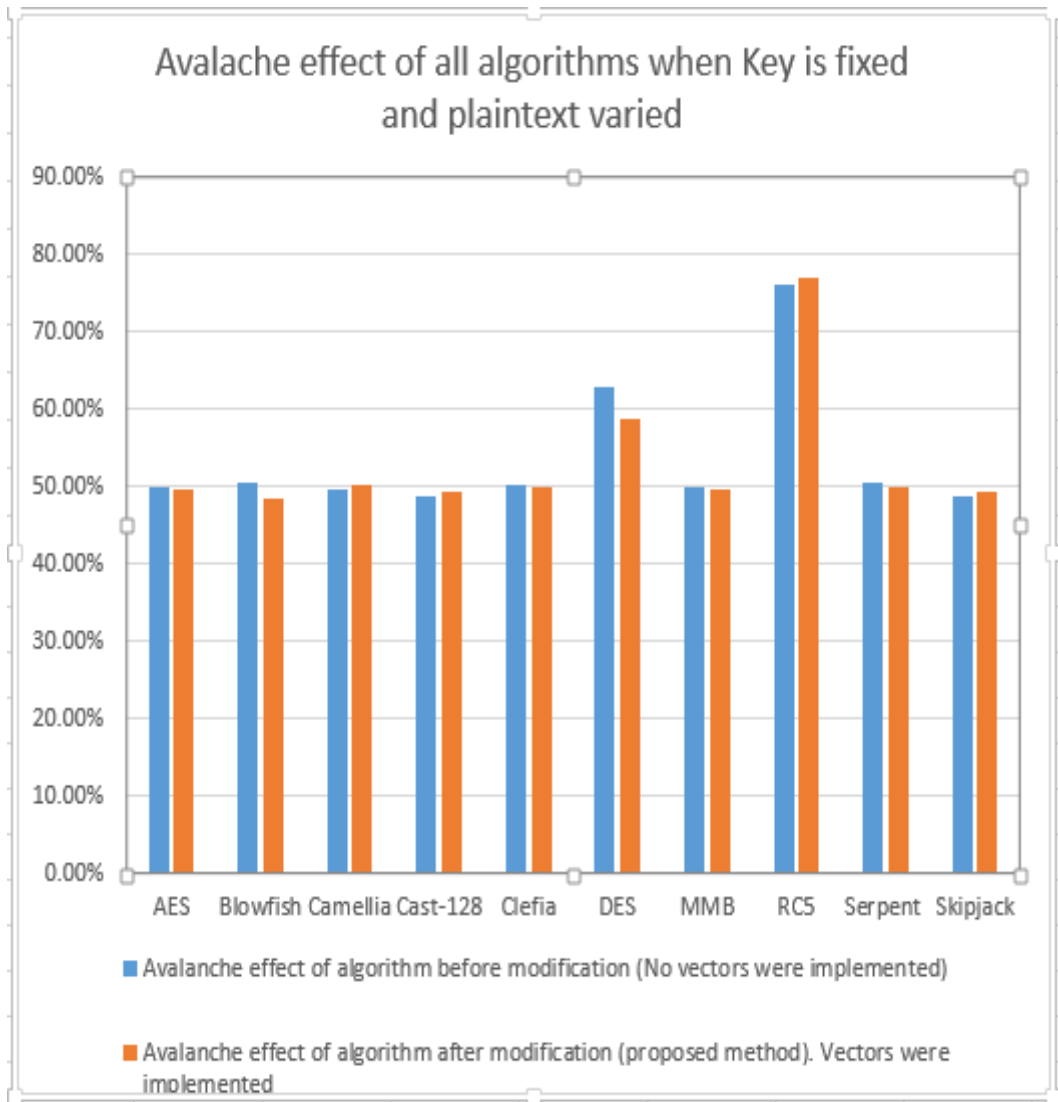


Figure 4.41: Results of avalanche effect of all algorithms tested when plaintext is varied.

From Figure 4.42. The avalanche effect was increased on four algorithms out of ten algorithms by using the proposed method. This means an increase of 40% of algorithms security is achieved. Therefore, the proposed method gave us positive results (40%) according to Figure 4.42. Wherever these four standard algorithms are implemented on IoT, therefore the proposed algorithms should replace them to enhance IoT security.

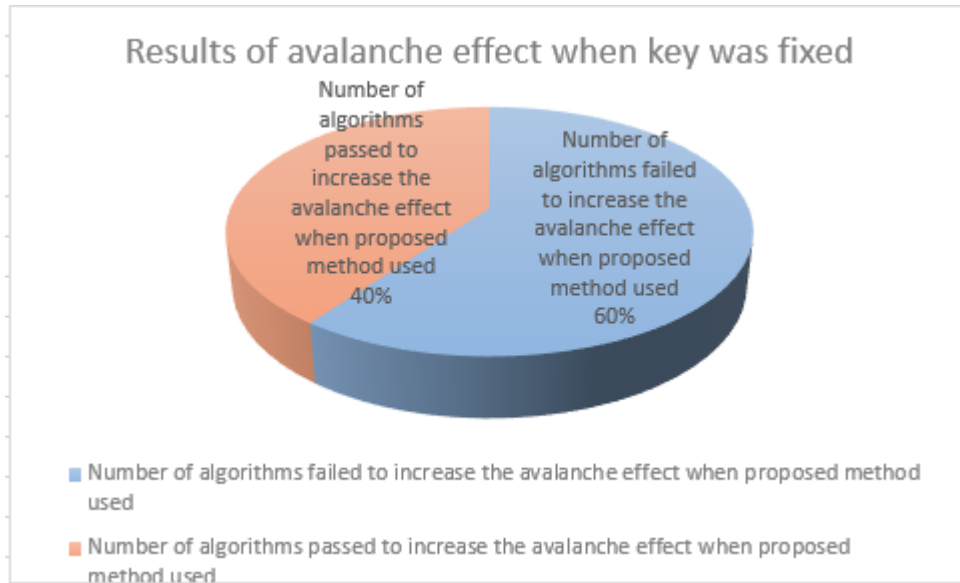


Figure 4.42: Results of Avalanche effect when plaintext was varied.

From the Figure 4.43, it is clear that the proposed method worked very sufficiently when we compare all algorithms put on test. We found that with the proposed MMB, DES, RC5, Camellia and AES algorithms were fast algorithms compared to the standard MMB, DES, RC5 Camellia and AES algorithms. Therefore, it is recommended that the proposed MMB, DES, Camellia and AES algorithms will be used as compared to standard MMB, DES, Camellia or AES algorithms if one wants an algorithm that has an improved speed when plaintext is varied [143], [6]. The proposed MMB has a speed of 70620.6897 bits/sec it is heighted in red on Table 4.22. The proposed DES has a speed of 70950.9787 bits/sec also heighted in red on Table 4.22. The proposed Camellia has a speed of 134626.1298 bits/sec also heighted in red on Table 4.22. The proposed RC5 has a speed of 68380.6344 bits/sec also heighted in red on Table 4.22. The proposed AES has a speed of 73012.4777 bits/sec also heighted in red on Table 4.22. Therefore, the proposed method worked positively according to our results from Figure 4.43. IoT needs algorithm that is fast to encrypt [145], therefore the proposed MMB, DES, RC5, Camellia and AES are faster than standard MMB, DES, Camellia and AES when plaintext is varied. The faster the algorithm, the higher the security [53].

Table 4.22: Result of the speeds of all algorithms tested when plaintext was varied.

Algorithm Tested	Speed calculated to finish avalanche effect of standard algorithm (in bits/sec)	Speed calculated to finish avalanche effect of proposed algorithm (in bits/sec)
AES	71671.0411	73012.4777
BLOWFISH	66699.2347	67037.6432
CAMELLIA	103959.3909	134626.1298
CAST-128	260104.7785	170666.6667
CLEFIA	139201.3594	134515.5993
DES	70017.0940	70950.9787
MMB	67842.6501	70620.6897
RC5	68266.66667	68380.6344
SERPENT	134847.7366	131072.0000
SKIPJACK	67680.1058	63268.4584

The proposed MMB has a speed of 70620.6897 bits/sec is presented graphically on Figure 4.43. The proposed DES has a speed of 70950.9787 bits/sec is presented graphically on Figure 4.43. The proposed Camellia has a speed of 134626.12983 bits/sec is presented graphically on Figure 4.43. The proposed RC5 has a speed of 68380.6344 bits/sec is presented graphically on Figure 4.43. The proposed AES has a speed of 73012.4777 bits/sec is presented graphically on Figure 4.43.

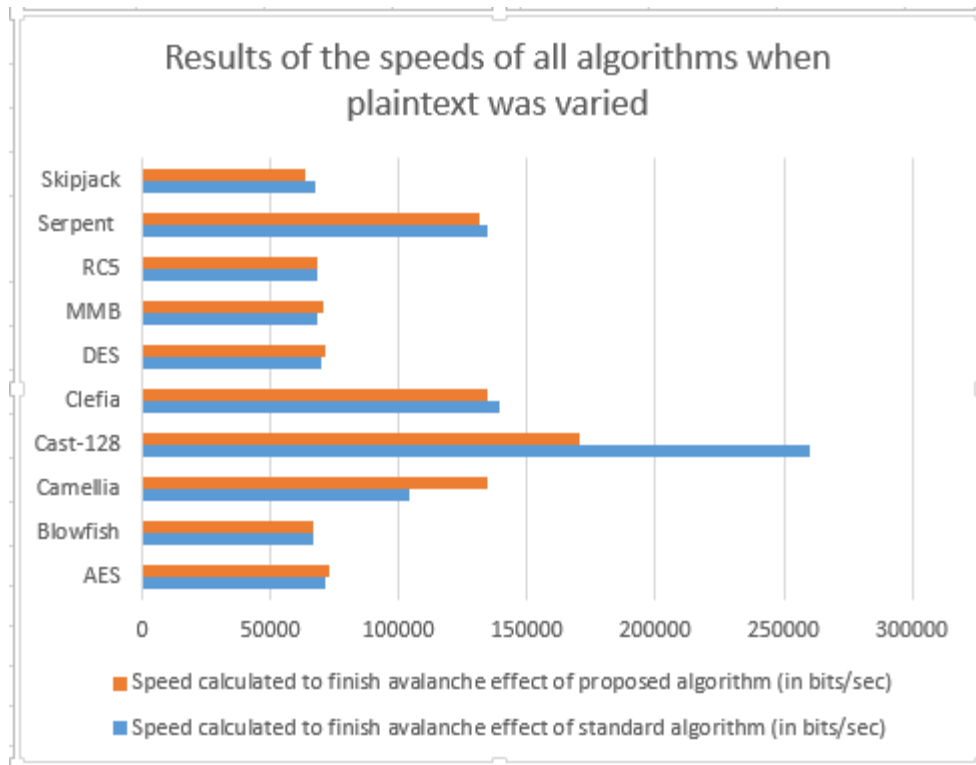


Figure 4.43: Result of the speeds of all algorithms tested when plaintext was varied.

From the Figure 4.44, it is clear that the proposed method worked very sufficiently when we compare all algorithms put on test. We found that with the proposed AES, Camellia, Cast-128, Clefia, DES, RC5 and Serpent algorithms yield an improved avalanche effect algorithm compared to their standard algorithms. The proposed AES has a value of avalanche effect of 49.9390%, it is heightened in red on Table 4.23. The proposed Camellia has a value of avalanche effect of 49.9893%, it is heightened in red on Table 4.23. The proposed Cast-128 has a value of avalanche effect of 50.1709%, also heightened in red on Table 4.23. The proposed Clefia has a value of avalanche effect of 50.1587%, also heightened in red on Table 4.23. The proposed DES has a value of avalanche effect of 44.2139%, also heightened in red on Table 4.23. The proposed RC5 has a value of avalanche effect of 49.7925, also heightened in red on Table 4.23. The proposed Serpent has a value of avalanche effect of 50.5341%, also heightened in red on Table 4.23. Therefore, it is recommended that the proposed AES, Camellia, Cast-128, Clefia, DES, RC5 and Serpent algorithms will be used as compared to their standard algorithms if one wants the algorithms with an improved avalanche effect when key is varied [143], [6]. We managed to increase the security of seven algorithms out of ten. That is 70%. Therefore, the proposed method worked positively according to our results from Figure 4.44.

Table 4.23: Results of avalanche effect of all algorithms tested when key was varied.

Algorithm Tested	Avalanche effect of algorithm before modification (No vectors were implemented)	Avalanche effect of algorithm after modification (proposed method). Vectors were implemented
AES	49.0662%	49.9390%
BLOWFISH	50.4517%	49.9878%
CAMELLIA	49.6094%	49.9893%
CAST-128	50.1221%	50.1709%
CLEFIA	49.9023%	50.1587%
DES	43.8721%	44.21387%
MMB	49.6765%	49.639893%
RC5	49.1821%	49.7925%
SERPENT	49.8657%	50.5341%
SKIPJACK	62.5732%	61.6211%

The proposed AES has a value of avalanche effect of 49.9390%, is indicated on Figure 4.44. The proposed Camellia has a value of avalanche effect of 49.9893%, is presented graphically on Figure 4.44. The proposed Cast-128 has a value of avalanche effect of 50.1709%, is presented graphically on Figure 4.44. The proposed Clefia has a value of avalanche effect of 50.1587%, is presented graphically on Figure 4.44. The proposed DES has a value of avalanche effect of 44.2139%, is presented graphically on Figure 4.44. The proposed RC5 has a value of avalanche effect of 49.7925, is presented graphically on Figure 4.44.

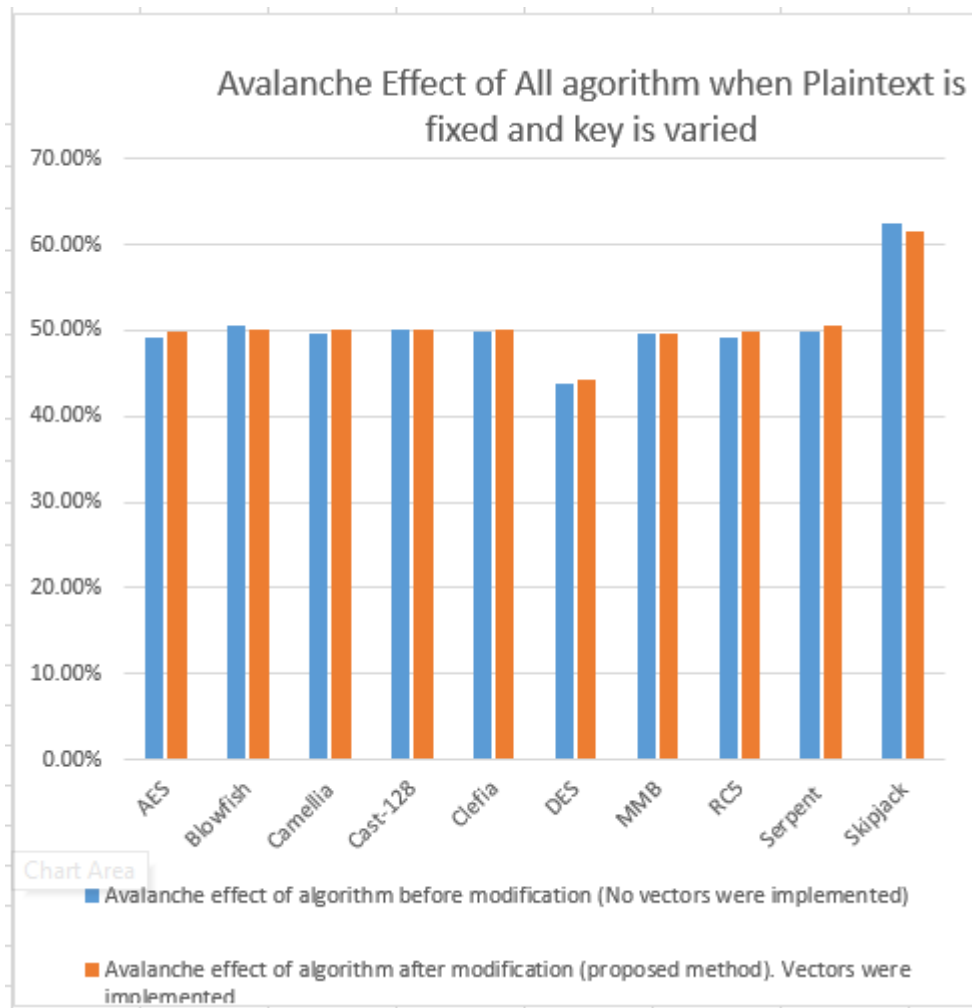


Figure 4.44: Results of avalanche effect of all algorithms tested when key was varied.

From Figure 4.45. The avalanche effect was increased. Seven out of ten algorithms enhance their avalanche effects when the proposed method used. That is when the key was varied. This means that we have managed to increase 70% of algorithms tested when key was varied. Therefore, the proposed method gave us positive results of 70% according to Figure 4.5. Wherever these seven algorithms are implemented on IoT, the proposed algorithms should replace them in order to increase IoT security.

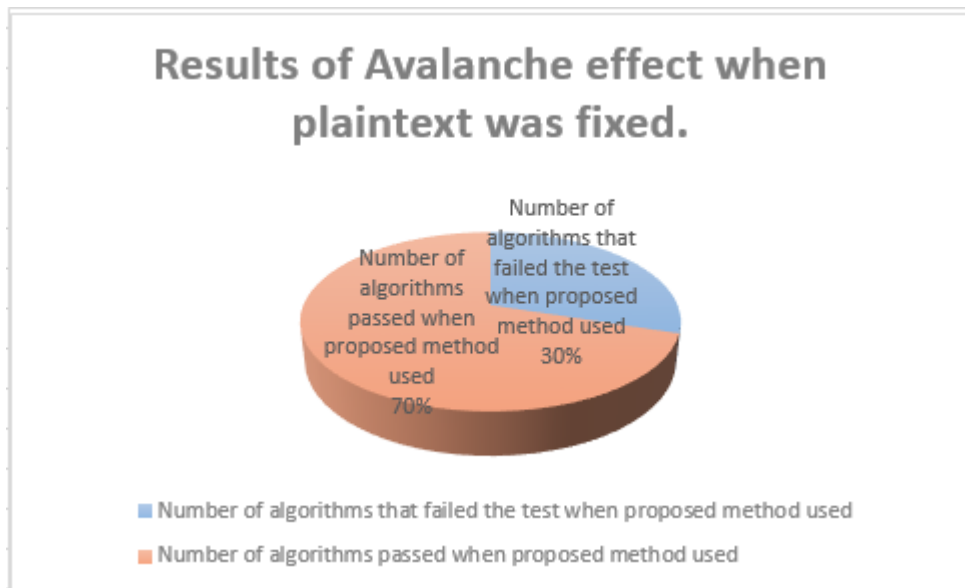


Figure 4.45: Results of Avalanche effect when key was varied.

From the Figure 4.46, it is clear that the proposed method worked very sufficiently when we compare all ten algorithms put on test. We find that the modified (proposed) AES and Clefia are faster algorithms compared to standard (AES and Clefia) of algorithms. The proposed AES has a speed of 138847.4576 bits/sec also heightened in red on Table 4.24. The proposed Clefia has a speed of 134405.2502 bits/sec also heightened in red on Table 4.24. Therefore, it is recommended that the proposed AES and Clefia will be used as compared to standard ones if one wants the algorithms like AES and Clefia with an improved speed when key is varied [143], [6]. Therefore, the proposed method worked positively according to our results from Figure 4.36. The internet of things needs the algorithm with is fast to encrypt [145], therefore the proposed AES and Clefia are faster than standard AES and Clefia when algorithms are tested.

Table 4.24: Results of the speed of all algorithms test when key was varied.

Algorithm Tested	Speed calculated to finish avalanche effect of standard algorithm (in bits/sec)	Speed calculated to finish avalanche effect of proposed algorithm (in bits/sec)
AES	138261.6034	138847.4576
BLOWFISH	132879.1565	132879.1565
CAMELLIA	201552.7611	198300.1614
CAST-128	132235.6740	130967.2262
CLEFIA	112604.8110	134405.25021
DES	66471.9247	64503.9370
MMB	134736.8421	134736.8421
RC5	140635.1931	123652.8302
SERPENT	241473.8394	238833.8192
SKIPJACK	86033.0690	84410.4458

The proposed AES has a speed of 138847.4576 bits/sec as indicated on Figure 4.46. The proposed Clefia has a speed of 134405.2502 bits/sec as indicated on Figure 4.46.

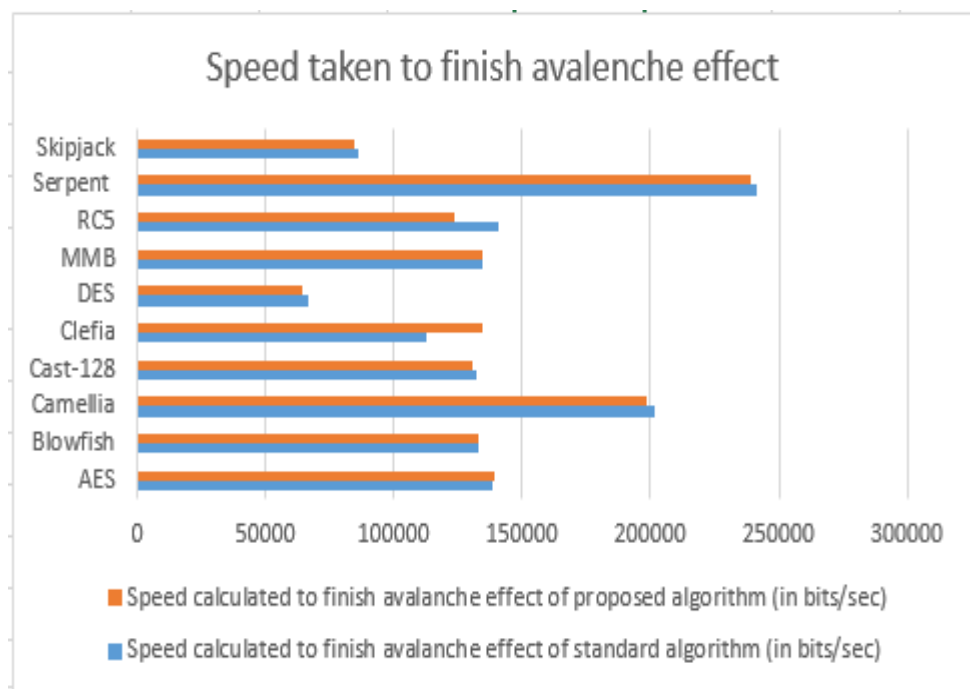


Figure 4.46: Results of the speed of all algorithms test when key was varied.

4.3. Chapter Summary

This chapter explained the results and analysis of all different methodologies discussed in chapter 3. It produced the results and analysis of avalanche effect of different types of cryptographic algorithms used on IoT. It gave results and analysis of the speed yielded by different algorithms. Additionally, it also gave the time taken to encrypt form different types algorithms.

CHAPTER 5: CONCLUSION AND FUTURE WORK

In this section, we conclude by summarising the work done in this dissertation from chapter 1 up to chapter 4: its introduction, literature review, methodology, results discussion, results analysis and future work.

In chapter 1, we set out to define the background of the study, definition of IoT, benefits of IoT, security, set out the problem statement, hypothesis, research questions, research objectives, significance and the research roadmap of this study. All these aspects were outlined in chapter 1.

Chapter 2 reviewed the literature that is related to our study. In the literature review, we explained the basic background of internet and dug deeper into IoT. We explained the security of IoT and the avalanche effect of crypto algorithms used to secure IoT. In addition, we explained different types of attacks (Denial of Service, Man-in-the-middle etc.) used by intruders to attack IoT. Additionally, a list of algorithms used on IoT were presented. Furthermore, we explained the origin of algorithms, who developed them and why they were developed. Finally, related work done by others researchers were also presented.

Chapter 3 discussed the research methodology that was used in our study. We explained the comparison method was used to measure the avalanche effect. In addition, we explained how we generated an initial and the final vector on the proposed work. We describe an overview and the strength of PI mathematically. We explained the need of avalanche effect on the security of IoT and the methods used to calculate them. In addition, chapter 3 explained the experimental procedures conducted when measuring avalanche effect, time and speed of all algorithms related to our study.

Chapter 4 gave the results, discussion and analysis that are related to our study. In the results, the programs written in C++ language executed the output. We discussed the results of avalanche effect of specific algorithms when their keys were fixed and plaintexts were varied and vice versa (or when plaintexts were fixed and keys were varied). We discussed the results from the output on how long (time) does an algorithm take to finish encryption (using avalanche effect to get more sample) when key was fixed and plaintext was varied and in reverse (or when plaintext was fixed and key was varied). We discuss the results from the

output on how fast (speed) an algorithm take to complete encryption (using the size of block the algorithm required and avalanche effect to get more sample) when key was fixed and plaintext was varied (or when plaintext was fixed and key was varied).

From the results, it was clear that the proposed method yielded better results by improving speed of certain standard algorithms when plaintext was varied. The minimum speed increase was 113.967 bits/sec for RC5 and maximum increase was 30667.2523 bits/sec for Camellia. Therefore, if the speed of standard algorithm is improved by proposed one, then the proposed algorithm is recommended to be used on IoT. The speed of standard AES has been improved from 71671.0411 bits/second to 73012.4777 bits/second by the proposed AES, giving the difference of 1341.4359 bits/second. The speed of standard Camellia has been improved from 103959.3909 bits/second to 134626.1298 bits/second by the proposed Camellia, giving the difference of 30667.2523 bits/second. The speed of standard DES has been improved from 70017.0940 bits/second to 70950.9787 bits/second by the proposed DES, giving the difference of 933.8847 bits/second. The speed of standard MMB has been improved from 67842.6501 bits/second to 70620.6897 bits/second by the proposed MMB, giving the difference of 2778.0397 bits/second. The speed of standard RC5 has improved from 68266.6667 bits/second to 68380.6344 bits/second by the proposed RC5, giving the difference of 113.9677 bits/second. That is when plaintext is varied.

From the results, it was clear that the proposed method yielded better results by improving speed of certain standard algorithms when key was varied. The minimum speed increase was 585.8542 bits/sec for AES and maximum increase was 21800.43921 bits/sec for Clefia. Therefore, if the speed of standard algorithm is improved by proposed one, then the proposed algorithm is recommended to be used on IoT. The speed of standard AES has been improved from 138261.6034 bits/second to 138847.4576 bits/second by the proposed AES, giving the difference of 585.8542 bits/second. The speed of standard Clefia has been improved from 112604.8110 bits/second to 134405.25021 bits/second by the proposed Clefia, giving the difference of 21800.43921 bits/second. That is when key is varied.

Out of ten algorithms that are used on IoT we slightly manged to improve avalanche effects of four algorithms, when the proposed algorithms plaintext was varied. The minimum percentage

increase of avalanche effect was 0.4395% for Skipjack and maximum increase was 0.7324% for RC5. This results are not better as expected but they are good results because the avalanche effect of proposed Camellia, Cast-128, RC5 and Skipjack are approaching (or more than) 50%. The main goal of this study is achieve avalanche effect, which is equal, or greater than 50%. Therefore, if the avalanche of standard algorithm is improved by proposed one, then the proposed algorithm is recommended to be used on IoT. The avalanche effect of standard Camellia has been slightly improved from 49.4690% to 50.0977% by the proposed Camellia, giving the difference of 0.6287%. The avalanche effect of standard Cast-128 has been slightly improved from 48.8281% to 49.3164% by the proposed Cast-128, giving the difference of 0.4883%. The avalanche effect of standard RC5 has been slightly improved from 76.1719% to 76.9043% by the proposed RC5, giving the difference of 0.7324%. The avalanche effect of standard Skipjack has been slightly improved from 48.7793% to 49.21875% by the proposed Skipjack, giving the difference of 0.4395%. This is when the plaintext was varied.

Out of ten algorithms that are used on IoT we slightly managed to improve avalanche effects of seven algorithms, when the proposed algorithms key was varied. The minimum percentage increase of avalanche effect was 0.0488% for Cast-128 and maximum increase was 0.8728% for AES. This results are not better as expected but they are good results because the avalanche effect of proposed Cast-128 and AES are approaching (or more than) 50%. The main goal of this study is achieve avalanche effect, which is equal, or greater than 50%. Therefore, if the avalanche of standard algorithm is improved by proposed one, then the proposed algorithm is recommended to be used on IoT. The avalanche effect of standard AES has been slightly improved from 49.0662% to 49.9390% by the proposed AES, giving the difference of 0.8728%. The avalanche effect of standard Camellia has been slightly improved from 49.6094% to 49.9893% by the proposed Camellia, giving the difference of 0.3799%. The avalanche effect of standard Cast-128 has been slightly improved from 50.1221% to 50.1709% by the proposed Cast-128, giving the difference of 0.0488%. The avalanche effect of standard Clefia has been slightly improved from 49.9023% to 50.1587% by the proposed Clefia, giving the difference of 0.2564%. The avalanche effect of standard DES has been slightly improved from 43.8721% to 44.21387% by the proposed DES, giving the difference of 0.3417%. The avalanche effect of standard RC5 has been slightly improved from 49.1821% to 49.7925% by the proposed RC5, giving the difference of 0.6104%. The avalanche effect of standard Serpent

has been slightly improved from 49.8657% to 50.5341% by the proposed Serpent, giving the difference of 0.6684%. This is when the key was varied.

Future work is to improve the results of avalanche effect from good to better results that is to improve avalanche effect by more than 2% difference instead of 0.8%. The other future work is to study the quality of encryption and decryption used by algorithms, by using image processing, that is comparing the original image and decrypted image using correlations and coefficients.

The hypothesis is right due to the following reasons: (1) It was found that there is a relationship between the avalanche effect of algorithms used on IoT and their security, the relation is high avalanche effect algorithm gives high security. (2) Certain algorithms were improved by proposed algorithms.

Research questions were answered: (1) The literature review on security of IoT is available in libraries, on internet, published papers, journals, conferences etc. (2) Types of algorithms to secure IoT are cryptographic algorithm. (3) Certain algorithms used on IoT has less than 50% of avalanche effect. (4) Certain algorithms were managed to be improved by proposed algorithms. (5) The benefit of high avalanche effect on IoT is to improve security.

References

[1] J. Kouns, “Bring Your Own Internet of Things BYO-IoT” 2015 RSA Conference, pp 4-5. [Online]. Available: <https://docplayer.net/16470302-Bring-your-own-internet-of-things-byo-iot.html>. [Accessed December 21, 2017].

[2] B Johnson, “The Internet of Things is nothing new for Johnson Controls”, 2017. [Online]. Available: <http://www.johnsoncontrols.com/insights/2017/thought-leadership/johnson-controls-continues-to-lead-in-iot-space>. [Accessed May 31, 2018].

[3] European Research Cluster on the Internet of Things, “Internet of Things,” *European Research Cluster on the Internet of Things*, [Online]. Available: http://www.internet-of-things-research.eu/about_iot.htm. [Accessed May 3, 2018].

[4] J. Holdowsky, M. Mahto, M. E. Raynor and M. Cotteleer, “Inside the Internet of Things (IoT), “2015, A Primer on the Technologies Building the IoT. [Online]. Available: <https://www2.deloitte.com/insights/us/en/focus/internet-of-things/iot-primer-iot-technologies-applications.html>. [Accessed May 31, 2018].

[5] T. Kambies, M.E. Raynor, D.M. Pankratz and G. Wadekar, “Closing the digital divide: IoT in retail’s transformative potential: The Internet of Things in the retail industry”, 2016. [Online]. Available: <https://www2.deloitte.com/insights/us/en/focus/internet-of-things/iot-retail-strategies.html>. [Accessed May 31, 2018].

[6] A. Kumar, N.amita Tiwari, “Effective Implementation and Avalanche Effect of AES.” 2013 International Journal of Security, Privacy and Trust Management (IJSPTM), Vol. 1, 2012, pp 31-35.

[7] A. Kumar, N. Tiwari, “Effective Implementation and Avalanche Effect of AES”, 2012 International Journal of Security, Privacy and Trust Management (IJSPTM), Vol. 1, 2012, pp 31-35.

[8] G. Patidar, N. Agrawal and S. Tarmakar, “A block based encryption model to improve Avalanche Effect for data security”, International Journal of Scientific and Research Publications, Vol. 3, (2013), pp.1-4. [Online]. Available: <https://www.ijsr.net>. [Accessed April 31, 2018].

[9] A. K. Mandal, and A. Tiwari, “Analysis of Avalanche Effect in Plaintext of DES using Binary Codes”, Chhatrapati Shivaji Institute of Technology, 2012, pp 166-0171.

[10] M.S. Mahindrakar, “Evaluation of Blowfish Algorithm based on Avalanche Effect,” 2014 International Journal of Innovations in Engineering and Technology (IJJET), Vol. 4, 2014, pp 99-103. [Online]. Available: <http://ijjet.com/wp-content/uploads/2014/06/15.pdf>. [Accessed December 21, 2017].

[11] K. Shujaat, I. M. Sohail, K. K. Ahmed and E. Mansoor, “Security Analysis of Secure Force Algorithm for Wireless Sensor Networks performance evaluation of 64, 128 and 192-bit secure force algorithm architecture”, 2014 Asian Journal of Engineering, Sciences & Technology, Vol. 4 No. 2, 2014, pp 46-52.

[12] D. Drushti and U. Hardik, “Security and Privacy Consideration for Internet of Things in Smart Home Environments”, 2014 International Journal of Engineering Research and Development, Vol. 10, pp 73-83.

[13] E. Alsaadi and A. Tubaishat, “Internet of Things: Features, Challenges, and Vulnerabilities International Journal of Advanced Computer Science and Information Technology.” 2015 IJACSIT, Vol. 4, No. 1, 2015, Pp: 1-13.

[14] T. Borgohain, U. Kumar and S. Sanyal, “Survey of Security and Privacy Issues of Internet of Things”, 2016 IEEE Internet of Things Journal, 2015, pp 1-7. [Online]. Available: https://www.researchgate.net/publication/270763270_Survey_of_Security_and_Privacy_Issues_of_Internet_of_Things. [Accessed December 27, 2017].

- [15] P. Walters, “The Risks of Using Portable Devices”, 2016 The United States Computer Emergency Readiness Team (US-CERT), 2016, pp 1- 3.
- [16] H. Klenk, H. B. Keller, E. Plödereder, P. Dencker (Hrsg.), “Automotive – Safety & Security”, 2015 Sicherheit und Zuverlässigkeit für automobile Informationstechnik , pp 95-96.
- [17] F. Telefonica, “Trend Report Insecurity in the Internet of Things Trend Report Insecurity in the Internet of Things”, 06/10/2015. [Online]. Available: https://www.elevenpaths.com/wp-content/uploads/2015/10/TDS_Insecurity_in_the_IoT.pdf. [Accessed November 21, 2017].
- [18] I. Lequetica, “Automotive IoT is disrupting the car rental industry”, 2017 Geotab management by measurement, 2016. [Online]. Available: <https://www.geotab.com/blog/automotive-iot/>. [Accessed June 31, 2018].
- [19] W.Y. Zibideh and M. M. Matalgah, “Alleviating the Effect of the Strict Avalanche Criterion (SAC) of Symmetric-Key Encryption in Wireless Communication Channels” 2011 International Conference on Communications and Information Technology (ICCII), pp 1, 2011.
- [20] N. Vijayrangan, “Method for preventing and detecting hash collisions of data during the data transmission” Peer to Patent, 2013. [Online] Available: <https://www.peertopatent.org/method-for-preventing-and-detecting-hash-collisions-of-data-during-the-data-transmission/>. [Accessed May 31, 2018].
- [21] S. Al-Sarawi, M. Anbar, K. Alieyan and M. Alzubaidi, “Internet of Things (IoT) Communication Protocols: Review”, IEEE 8th International Conference on Information Technology, 2017, pp 1-10.
- [22] O. Gunnsteinsson, “A Search for a Convenient Data Encryption Algorithm for an Internet of Things Device.” 2016, Chalmers University of Technology, Gothenburg, Sweden 2016.

- [23] N. Saxena and A. Reza Sadeghi. "Radio Frequency Identification: Security and Privacy Issues," 2014 10th International Workshop, RFIDSec, Oxford, UK, 2014, pp 61.
- [24] M. Morrow, "Securing the Internet of Things: A Proposed Framework", 2015, pp 1-7. [Online]. Available: <https://blogs.cisco.com/sp/securing-the-internet-of-things-a-proposed-framework>. [Accessed Jul 31, 2017].
- [25] M. Mohammedi and M. Aledhari, "Internet of Things: A Survey on Enabling Technologies, Protocols and Applications", IEEE Communications Surveys & Tutorials, Vol. 17, 2015.
- [26] P. Duffy, "Beyond MQTT: A Cisco View on IoT Protocols", Digital Transformation, 2013.
- [27] M. Bilal, "A Review of Internet of Things Architecture, Technologies and Analysis Smartphone-based Attacks Against 3D printers", Zhejiang University Hangzhou, China.
- [28] S. N. Swamy, D. Jadhav and N. Kulkarni, "Security threats in the application layer in IOT applications", International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2017.
- [29] D. Anand, A. Rathore, S. Kumar and S. Durgapal, "Awesome & Useful IoT Protocols", Electronics For You (EFY) magazine, 2017.
- [30] S. Kulkarni and S. Kulkarni, "Communication Models in Internet of Things: A Survey" International Journal of Science Technology & Engineering (IJSTE), Vol: 3, May 2017, pp 87-91. [Online]. Available: <http://www.ijste.org/articles/IJSTEV3I11049.pdf>. [Accessed January 31, 2018].
- [31] D. Thaler, H. Tschofenig and T.M. Barnes, "Architectural Considerations in Smart Object Networking", IAB RFC 7452, IETF 92 Technical Plenary.

- [32] M. Abomhara and G. M. Kjøien, “Cyber Security and the Internet of Things: Vulnerabilities, Threats, Intruders and Attacks” 2015 Journal of Cyber Security, University of Agder, Norway, Vol. 4, 2015, pp 66-69.
- [33] A. Nordrum, “Popular internet of things Forecast of 50 Billion Devices by 2020 is Outdated,” 2016 IEEE Spectrum: Technology, Engineering, and Science News, 2016, pp 5-7.
- [34] P. Paganini, “How Hackers Violate Privacy and Security of Smart Home”, 2015 InfoSec Institute, 2016, pp 3-6.
- [35] K. Kishore, S. Satapathy, K. Udgata and N. Bisval, “Randomised Approach for Block Cipher Encryption and Calculation of the avalanche effect coefficient”, 2016 Frontiers of Intelligent Computing: Theory and Practical.
- [36] T. Borgohain, U. Kumar and S. Sanyal, “Survey of Security and Privacy Issues of Internet of Things”, 2015. Int. J. Advanced Networking and Applications Vol. 6, No. 4, Assam Engineering College, 2015, pp 2373-2378. [Online]. Available: <http://www.ijana.in/papers/V6I4-3.pdf>. [Accessed: Aug. 17, 2017].
- [37] I. van der Elzen and J. van Heugten, “Techniques for detecting compromised IoT devices”, 2017 University of Amsterdam, 2017, pp 1-17.
- [38] P. Walters, “The Risks of Using Portable Devices”, 2012 Carnegie Mellon University. Produced for US-CERT, a government organization, 2016 The United States Computer Emergency Readiness Team (US-CERT), 2012, pp 1- 3.
- [39] H. Klenk, H. B. Keller, E. Plödereder and P. Dencker (Hrsg.), “Automotive – Safety & Security”, 2015 Sicherheit und Zuverlässigkeit für automobile Informationstechnik , 2015, pp 95-96.

[40] C. P. Dewangan and S. Agrawal, "A Novel Approach to Improve Avalanche Effect of AES Algorithm," 2012 International Journal of Advanced Research in Computer Engineering & Technology Vol. 1, 2012, pp 248-252.

[41] Source: Cisco, "The Internet of Things Reference Model", Controlled Distribution. [Online]. Available: http://cdn.iotwf.com/resources/71/IoT_Reference_Model_White_Paper_June_4_2014.pdf. [Accessed December 27, 2017].

[42] H. M. Aldosari, "A Proposed Security Layer for the Internet of Things Communication Reference Model", International Conference on Communication, Management and Information Technology (ICCMIT), 2015, pp 421-428. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-642-14478-3_42?LI=true. [Accessed May 31, 2018].

[43] M. Katagi and S. Moriai, "The 128-Bit Block Cipher CLEFIA," 2011 Sony Corporation, Vol. 6114, 2011, pp 1-19.

[44] J. Kaur and E. ManpreetKaur, "Data Encryption Using Different Techniques: A Review", International Journal of Advanced Research in Computer Science, Vol. 8, No. 4, May 2017, pp 252-255.

[45] C. Collberg, "Cryptography — Symmetric Key" 2012, University of Arizona CSc 466/566 Computer Security 6, Version: 2012/02/22 16:14:32, 2012, pp 1-56.

[46] M. Fischlin, "Public-Key Encryption (Asymmetric Encryption)", Summer School, Romania 2014.

[47] P.K. Arya, M.S. Aswal and V. Kumar, "Comparative Study of Asymmetric Key Cryptographic Algorithms", International Journal of Computer Science and Communication Networks, Vol. 5, pp 17-21.

- [48] S. Channalli and A. Jadhav, “Steganography an Art of Hiding Data”, International Journal on Computer Science and Engineering, Vol. 1, 2009, pp 137-14.
- [49] N.G. McDonald, “Past, Present, and Future Methods of Cryptography and Data Encryption”, University of Utah, 2015.
- [50] A. Toumazis, “Steganography”, 2009. [Online]. Available: [Accessed Jan 31, 2018].
- [51] 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.
- [52] K. M. Martin, “Everyday Cryptography”, Oxford University Press, 2012, pp 142-149.
- [53] L. D. Smith, "Substitution Ciphers". Cryptography the Science of Secret Writing: The Science of Secret Writing. Dover Publications, 1943, pp 81-90.
- [54] M. Behrens, “Understanding the 3 main types of encryption”, 2014. [Online]. Available: <https://spin.atomicobject.com/2014/11/20/encryption-symmetric-asymmetric-hashing/>. [Accessed May 31, 2018].
- [55] R. Smith, “Understanding encryption and cryptography basics”. 2015. [Online]. Available: <https://searchsecurity.techtarget.com/Understanding-encryption-cryptography>. [Accessed Jun 31, 2018].
- [56] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, J. F. Dray, “Advanced Encryption Standard (AES)”, Federal Inf. Process. Stds. (NIST FIPS), 2001.
- [57] D. L. Evans, P. J. Bond and A. L. Bement “Security Requirements for Cryptographic Modules”, FIPS Publication 140-2, 1994.

- [58] R. D. Bajaj and U.M. Gokhale, “AES Algorithm for Encryption”, 2016 International Journal of Latest Research in Engineering and Technology (IJLRET), Vol. 02, 2016, pp 63-68.
- [59] S. Bhargav, L. Chen, A. Majumdar and S. Ramudith, “128-bit AES decryption”, 2008 CSEE 4840 Embedded System Design Spring, Columbia University, Vol. 4840, 2008, pp 8347-8350.
- [60] A. Jagadev and V. Senapati, “Advanced Encryption Standard (AES) Implementation.” 2009, Rourkela May 2009. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.606.1732&rep=rep1&type=pdf>. [Accessed May 31, 2017].
- [61] P. Singh and K. Singh, “Cryptography in Image Using Blowfish Algorithm”, 2015 International Journal of Science and Research (IJSR), Vol. 4, 2015, pp 150-154.
- [62] K. Bawale, Y. S. Patil, and P. R. Pawale, “ Image Encryption Technique using Blowfish Algorithm “ 2015 International Journal of Advance Foundation And Research In Science & Engineering (IJAFRSE), Vol. 1, 2015, pp 1-6.
- [63] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima and T. Tokita, “Specification of Camellia — a 128-bit Block Cipher”, 2001 Nippon Telegraph and Telephone Corporation and Mitsubishi Electric Corporation, 2001.
- [64] A. Satoh and S. Morioka, “Hardware-Focused Performance Comparison for the Standard Block Ciphers AES, Camellia, and Triple-DES”, 2001 Tokyo Research Laboratory IBM Japan Ltd, 2001.
- [65] Y. L. Yin, “A Note on the Block Cipher Camellia,” 2000 NTT Multimedia Communication Laboratories 250 Cambridge Avenue, Palo Alto, CA 94306 6, Vol. 1.11.

- [66] M. Matsui, S. Moriai and J. Nakajima, “A Description of the Camellia Encryption Algorithm,” 2004 Network Working Group Mitsubishi Electric Corporation and Sony Computer Entertainment Inc. pp 1-15.
- [67] C.M. Adams, “Constructing Symmetric Ciphers using the CAST Design Procedure, Designs, Codes, and Cryptography”, Vol. 12, pp 283–316, 1997.
- [68] C. Adams, “The CAST-128 Encryption Algorithm”, 1997 Network Working Group and Entrust Technologies Category, 1997, pp 1-15.
- [69] J. Nakahara Jr and M. Rasmussen, “Linear Analysis of reduced-round CAST-128 and CAST-256”, 2016 LSI-TEC, Brazil, 2016, pp 1-5.
- [70] W. Wang and X.Y. Wang, “Impossible differential cryptanalysis of CLEFIA-128/192/256” 2009 Journal of Software by Institute of Software, the Chinese Academy of Sciences, Vol. 20, No. 9, 2009, pp. 2587–2596.
- [71] A. Biryukov and I. Nikolić, “Security Analysis of the Block Cipher”, Clefia version 1.1 final report. [Online]. Available: <https://www.cryptrec.go.jp/exreport/cryptrec-ex-2202-2012p2.pdf>. [Accessed December 01, 2018].
- [72] S.S. Ali and D. Mukhopadhyay, “Protecting Last Four Rounds of CLEFIA is Not Enough against Differential Fault Analysis,” Indian Institute of Technology Kharagpur, 2016.
- [73] J. Chouinard, “Design of secure system. Notes on Data Encryption Standard,” The University of Ottawa, Canada, 2002.
- [74] R.G. Kammer and W. Meheron, “Data Encryption Standard (DES)”, 1999 U.S. National, Institution of Standards and Technology, FIPS Publication 46-3,1999, pp 1-27.

- [75] F. El-Zoghdy, Y. A. Nada, and A. A. Abdo, "How Good Is The DES Algorithm In Image Ciphering?" 2011 Int. J. Advanced Networking and Applications, 796 Vol. 02, 2011, pp 796-803.
- [76] J. Thakur and N. Kumar "DES, AES and Blowfish: Symmetric Key Cryptography Algorithms Simulation Based Performance Analysis", 2011. [Online]. Available: www.ijetae.com. [Accessed: Aug. 17, 2017].
- [77] K. Jia, J. Chen, M. Wang and W. Xiaoyun, "Practical-time Attack on the Full MMB Block," 2010 Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Jinan 250100, China, 2010, pp 1-13.
- [78] J. Daemen, R. Govaerts and J. Vandewalle, "Block Ciphers Based on Modular Arithmetic", Katholieke Universiteit Leuven, Laboratorium ESAT Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium.
- [79] K. Aoki, T. Ichikawa, et al, "Specification of Camellia", Nippon Telegraphy and Telephone Corporation and Mitsubishi Electric Corporation, Vol. 2.0, 2001.
- [80] R. Rymon, "Conventional Cryptography", Efi Arazi School of Computer Science IDC, Herzliya, 2008.
- [81] R. L. Rivest, "The RC5 Encryption Algorithm", 1997 MIT 545 Laboratory for Computer Science Technology Square Cambridge Mass Revised, 1997.
- [82] R. Anderson, E. Biham and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", pp 1-23. [Online]. Available: <https://www.cl.cam.ac.uk/~rja14/Papers/serpent>. [Accessed June 27, 2017].
- [83] S. Mister, "Properties of the Building Blocks of Serpent," 2000 Entrust Technologies, pp 1-9, 2000.

- [84] G. Kuznetsov, R. Karri and M. Gossel, "Error Detection by Parity Modification for the 128-bit Serpent Encryption Algorithm", Polytechnical University Brooklyn, USA University of Potsdam, Germany, 2016, pp 1-24.
- [85] A. Muhammad, "Implementation and verification of SKIPJACK Algorithm using Verilog," IEEE, Vol. 2, 1998, pp 1-9.
- [86] P. Bantarha, L. Knudsen and D. Wagner, "On the structure of Skipjack", 2000, University of Bergen, N-5020 Bergen, Norway.
- [87] D. D. Moskovich. "An Overview of the State of the Art for Practical Quantum Key Distribution", Vol. 4, 2015, pp 1-26.
- [88] C. Bourke, "CSCE 477/877", University of Nebraska, Lincoln, 2015, pp 5-138.
- [89] Y. Shaikh and A. Jain, "Investigation of Symmetric Block Cipher Algorithms" 2011 International Institute of Professional Studies, Devi Ahilya University, 2011.
- [90] M. M. Matalgah and W. Y. Zibideh "Alleviating the Effect of the Strict Avalanche Criterion (SAC) of Symmetric-Key Encryption in Wireless Communication Channels" 2011 International Conference on Communications and Information Technology (ICCII), Aqaba 2011, pp 1-9.
- [91] R. Sobti and G. Geetha, "Cryptographic Hash Functions: A Review", IJCSI International Journal of Computer Science Issues, Vol. 9, No. 2, No 2, March 2012, pp 461-479. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.402.7241&rep=rep1&type=pdf>. [Accessed May 31, 2018].
- [92] C. Paar and J. Pelzl, "Understanding Cryptography", 2010 Springer Heidelberg Dordrecht London, New York, ACM Computing Classification, 2010.

[93] E. Alsaadi and A. Tubaishat, "Internet of Things: Features, Challenges, and Vulnerabilities International Journal of Advanced Computer Science and Information Technology" 2015 IJACSIT Switzerland, Vol. 4, No. 1, 2015, pp: 1-13.

[94] M. B. Abdelhalim, M. El-Mahallawy, M. Ayyad and A. Elhennawy, "Design and Implementation of an Encryption Algorithm for use in RFID, " 2013 System International Journal of RFID Security and Cryptography (IJRFIDSC), Vol. 2, No. 1, 2013, pp 51-57.

[95] J. Pawlick and Q. Zhu, "Internet of Things: Privacy & Security in a Connected World" Transcript of Workshop at 182. Transcript of Workshop, 2015, pp5-55.

[96] R. Leidos, C. Garlati, D. Lingenfelter and B. Russell, "Security Guidance for Early Adopters of the Internet of Things (IoT)", 2015 Cloud Security alliance and Mobile Working Group Peer Reviewed Document, 2015, pp 1-54.

[97] M.N. Yusuff, "Honeypot Revealed", 2005. [Online]. Available: <http://files.spogel.com/abstracts/p-1528--Honeypots.pdf>. [Accessed May 31, 2018].

[98] L. Spitzner, "Honeypots Definition and Value of Honeypots", 2002. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1255091. [Accessed June 01, 2017].

[99] K. Schramm, T. Wollinger, and C. Paar, "A New Class of Collision Attacks and its Application to DES", Communication Security Group (COSY), Ruhr-Universität Bochum, Germany Universitätsstrasse 150 44780 Bochum, Germany.

[100] I. Dinur, O. Dunkelman and A. Shamir¹, "Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials", The Weizmann Institute, Rehovot, Israel.

[101] L. Wei, T. Zhi, G. Dawu, S. Li, Q. Bo, L. Zhiqiang and L. Ya, "An effective differential fault analysis on the Serpent cryptosystem in the Internet of Things" 2014 China Communications, Vol.11, 2014, pp 129-139.

[102] L. Wei, “An effective differential fault analysis on the Serpent cryptosystem in the Internet of Things” 2014 Sch. of Computer. Sci. & Technol., Donghua Univ., Shanghai, China, Vol: 11, 2014.

[103] M. Rivain, “Differential Fault Analysis on DES Middle Rounds”, Oberthur Technologies and University of Luxembourg.

[104] L. Wei, T Zhi and G. Dau, “An effective differential fault analysis on the Serpent cryptosystem in the Internet of Things”, China Communications Vol. 11, No. 6, June 2014.

[105] A.J. Paul, A. Saju and R. Lekshimi, “Data based Transposition to Enhance Data Avalanche and Differential Data Propagation in Advanced Encryption Standard,” International Journal of Computer Applications, Vol. 67– No.12, 2013, pp 6-9.

[106] O. Dunkelman, N. Keller , A. Biryukov and D. Khovratovich , A. Shamir, “Improved Single-Key Attacks on 8-Round AES-192 and AES-256”, Journal of Cryptology, Vol. 28, No. 3, 2015, pp1-29. [Online]. Available: <https://eprint.iacr.org/2010/322.pdf>. [Accessed May 31, 2018].

[107] N. Alassaf, B. Alkazemi and A. Gutub, Applicable Light-Weight Cryptography to Secure Medical Data in IoT Systems”, Journal of Research in Engineering and Applied Sciences, 2017, pp 50-58. [Online]. Available: https://www.researchgate.net/publication/316147802_Applicable_LightWeight_Cryptography_to_Secure_Medical_Data_in_IoT_Systems. [Accessed May 31, 2018].

[108] L.W. Santoso, G. S. Budhi and L. Sutanto, “ Perbandigan Apliasi Menggunakan Metode Camelia 128 Bit Key Dan 256 Key”, Jurnal Informatika, Vol. 12, No. 2, November 2014, pp 109-115. [Online]. Available: <http://jurnalinformatika.petra.ac.id/index.php/inf/article/view/19147>. [Accessed May 31, 2018].

[109] S. Ramanujam and M. Karuppiah, “Designing an algorithm with high Avalanche Effect,” 2011 IJCSNS International Journal of Computer Science and Network Security, Vol. 11, No.1, 2011, pp 106-111. [Online]. Available: http://paper.ijcsns.org/07_book/201101/20110116.pdf. [Accessed February 21, 2017].

[110] S. Wang, T. Cui and M. Wang, “Improved Differential Cryptanalysis of CAST-128 and CAST-256”, Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Jinan 250100, China, 2017, pp 18-32. https://link.springer.com/chapter/10.1007/978-3-319-54705-3_2citeas. [Accessed June 31, 2018].

[111] G.N. Krishnamurthy, V. Ramaswamy, G.H. Leela G.H and M.E Ashalatha, “Performance enhancement of Blowfish and CAST-128 algorithms and Security analysis of improved Blowfish algorithm using Avalanche effect”, IJCSNS 244 International Journal of Computer Science and Network Security, Vol. 8 No. 3. 2008, pp. 244-250 [Online]. Available: http://paper.ijcsns.org/07_book/200803/20080336.pdf. [Accessed May 31, 2018].

[112] D. Sehrawat and N. Singh Gill, “Lightweight Block Ciphers for IoT based applications: A Review”, International Journal of Applied Engineering Research ISSN 0973-4562 Vol. 13, pp. 2258-2270, 2018.

[113] C. Boura, M. Naya-Plasencia and V. Suder, “Scrutinizing and Improving Impossible Differential Attacks: Applications to Clefia, Camellia, LBlock and Simon (Full Version)” Versailles Saint-Quentin-en-Yvelines University, France.

[114] S. Khan, M. S. Ibrahim, K. A. Khan and M. Ebrahim, “Security Analysis of Secure Force Algorithm for Wireless Sensor Networks”, Asian Journal of Engineering Science and Technology, 2015, [Online]. Available: <https://arxiv.org/abs/1509.00981>. [Accessed May 31, 2018].

- [115] S. Ibrahim, B. Mohd, A. Maarof and N. B. Idris, "Avalanche Analysis of Extended Feistel Network," Universiti Teknologi, Malaysia, Proceedings of the Postgraduate Annual Research Seminar, 2005, pp 265-269.
- [116] M. Talbi and M. Salim Bouhlef, "Application of a Lightweight Encryption Algorithm to a Quantized Speech Image for Secure IoT", Sciences Electroniques, Technologie de l'Information et Télécommunications (SETIT), 2018.
- [117] A. Biryukov and V. Velichkov, "On Improving Data Complexity of Attacks on RC5 Laboratory of Algorithmic, Cryptology and Security (LACS)", University of Luxembourg, Early Symmetric Crypto, 2015.
- [118] Y.H. Ali, "Proposed 256 bits RC5 Encryption Algorithm Using Type-3 Feistel Network", Eng. & Tech. Journal, Vol. 28, No.12, 2010, pp 2337-2345. [Online]. Available: https://www.researchgate.net/publication/312029656_Proposed_256_bits_RC5_Encryption_Algorithm_Using_Type-3_Feistel_Network. [Accessed Jan 31, 2018].
- [119] A. A. Maaita and H. A. Alsewadi, "A Multi-Threaded Symmetric Block Encryption Scheme Implementing PRNG for DES and AES Systems," 2017 International Journal of Advanced Computer Science and Applications, Vol. 8, No. 2, 2017 pp 76-82.
- [120] E. Biham, A. Biryukov and A. Shamir, "Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials", Institute of Technology, Technion, Israel.
- [121] B. Maram and J. M. Gnanasekar, "A Block Cipher Algorithm to Enhance the Avalanche Effect Using Dynamic Key-Dependent S-Box and Genetic Operations", Research and Development Centre, Bharathiar University, Coimbatore.
- [122] H. Yang "Blowfish - 8-Byte Block Cipher", 2017 Cryptography Tutorials - Herong's Tutorial Examples - Version 5.31, First 8366 Hex Digit of PI, 2017.

- [123] P. Trüb, "π trillion digits of π", 2016 Project, Dectris Ltd. 5400 Baden Switzerland, 2016. [Online]. Available: <https://www.dectris.com/company/news/newsroom/success-story-details/pi-computed-to-22-4-trillion-digits>. [Accessed December 27, 2017].
- [124] M. Suresh, M.B Neema, "Hardware implementation of blowfish algorithm for the secure data transmission in Internet of Things" 2016 Global Colloquium in Recent Advancement and Effectual Researches in Engineering, Science and Technology (RAEREST), 2016.
- [125] J. Daemen, R. Govaerts, J. Vandewalle, "Block Ciphers Based on Modular Arithmetic", 1993 Proc. of the third Symposium on the State and Progress of Research in Cryptography, 1993.
- [126] M. Usman, I. Ahmed, M. I. Aslam, S. Khan and U. A. Shah, "SIT: A Lightweight Encryption Algorithm for Secure Internet of Things", 2017 International Journal of Advanced Computer Science and Applications, Vol. 8, 2017.
- [127] A. L. Abdel-Karim, "Tamimi Performance Analysis of Data Encryption Algorithms". 2014. [Online]. Available: https://www.researchgate.net/publication/228775009_Performance_Analysis_of_Data_Encryption_Algorithms. [Accessed Jun 31, 2018].
- [128] K. N. Prasetyo, Y. Purwanto and D. Darlis, "An implementation of data encryption for Internet of Things using blowfish algorithm on FPGA" 2014 Information and Communication Technology (ICoICT), 2014 2nd International Conference, 2014.
- [129] G.N. Krishnamurthy, V. Ramaswamy, G.H. Leela and M.E. Ashalatha, "Performance enhancement of Blowfish and CAST-128 algorithms and Security analysis of improved Blowfish algorithm using Avalanche effect" 2008 IJCSNS International Journal of Computer Science and Network Security, Vol. 8 No.3, 2008, pp 244-250.
- [130] E. Young, "HomePage-EricYoung," *schneier.com*, para 3, DES Validation test. [Online]. Available: <https://www.schneier.com/code/vectors.txt>. [Accessed Nov. 14, 2017].

[131] Sony Corporation, “The 128-bit Blockcipher CLEFIA,” Specification, Version 1.0, 2010. [Online]. Available: http://www.cryptrec.go.jp/english/cryptrec_13_spec_cypherlist_files/PDF/22_00espec.pdf. [Accessed May 31, 2018].

[132] J.O. Grabbe, “The DES Algorithm Illustrated”, 2011. [Online]. Available: <http://page.math.tu-berlin.de/~kant/teaching/hess/krypto-ws2006/des.htm>. [Accessed: Aug. 17, 2017].

[133] K Jia, J. Meiqin, and X. Wang, “Practical Attack on the Full MMB Block Cipher”, Institute for Advanced Study, Tsinghua University, Beijing, China.

[134] T. Ashur and O. Dunkelman, “A Practical-Time Related-Key Boomerang Attack on MMB,” International Conference on Cryptology and Network Security CANS 2013: Cryptology and Network Security, pp 271-290, 2013.

[135] K. Agarwal, P. Rao and A. Kaminsky, “Implementation of the RC5 block cipher algorithm and implementing a variation of meet-in-the-middle attack on it”, 2013 Cryptography Spring 2013 - Team Project Report, 2013, pp1-21.

[136] A. J. Menezes, J. Katz and P. C. van Oorschot, “*Handbook of Applied Cryptography*”, CRC Press, 1996.

[137] N. Aghajanzadeh, F. Aghajanzadeh and H. R. Kargar, “Developing a new Hybrid Cipher using AES, RC4 and SERPENT for Encryption and Decryption”, International Journal of Computer Applications, Vol. 69, May 2013, pp 53-62. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.404.1822&rep=rep1&type=pdf>. [Accessed December 31, 2017].

[138] Project NESSIE, “New European Schemes for Signature, Integrity, and Encryption,” [Online]. Available: <http://www.cs.technion.ac.il/~biham/Reports/Serpent/Serpent-256-28.verified.test-vectors>. [Accessed December. 3, 2017].

[139] J. Pawling “Use of the KEA and SKIPJACK Algorithms in CMS”. 2000. [Online]. Available: <https://tools.ietf.org/html/rfc2876>. [Accessed May 31, 2018].

[140] E. Biham, A. Biryukov, O. Dunkelman and E. Richardson, “Initial Observations on Skipjack: Cryptanalysis of Skipjack-3XOR”, 2002, pp 1-14. [Online]. Available: <https://dl.acm.org/citation.cfm?id=694578>. [Accessed May 31, 2018].

[141] L. Knudsen and D. Wagner, “On the structure of Skipjack” , 2001 Discrete Applied Mathematics 111, University of Bergen, N-5020 Bergen, Norway University of California Berkeley, Soda Hall, Berkeley, 2000, pp 103-116.

[142] Ziel Project, “New European Schemes for Signature, Integrity, and Encryption,” [Online]. Available: <https://cordis.europa.eu/project/rcn/54113/factsheet/de>. [Accessed October 2017]. [Accessed February 3, 2018].

[143] S. Ramanujam and M. Karuppiah, “An algorithm with high Avalanche Effect”, 2011 International Journal of Computer Science and Network Security, Vol. 11, 2011, pp 106-111.

[144] L. Oppenheim and S. Tal, “The Internet of TR069 Things: One Exploit to Rule Them All” 2015 RSA Conference, 2015.

[145] A. A. Tamimi, “Performance Analysis of Data Encryption Algorithms”. 2015. [Online]. Available: www.cs.wustl.edu/~jain/cse567-06/ftp/encryption_perf/. [Accessed May 31, 2018].

[146] M. Abomhara and G. M. Kjøien, “Cyber Security and the Internet of Things: Vulnerabilities, Threats, Intruders and Attacks”, University of Agder, Norway, pp 66-69.

APPENDIX 1: Calculation of AES avalanche effect.

We will give example by AES algorithm on how avalanche effect is calculated manually. AES algorithm is 128 block cipher, this means that the size of ciphertext and plaintext are both 128 bits.

- i. First we fixed the key, in this study the key is:

{0123456789abcdeffedcba9876543210} in hexadecimal,

- ii. Then we generated plaintext, in this study the plaintext is:

{ffffffffffffffffffffffffffffffff}.

This is 128 bit in hexadecimal number. Note that f in hexadecimal number is 15 in decimal and is 1111 in binary. Flipping one bit of generated plaintext from left to right we get 7 in hexadecimal number which is 0111 in binary, b = 1011, d = 1101, e = 1110 etc. In this example we will indicate the position of where the bit is changed (flipped) and its ciphertext by red color to simplify the explanation.

- iii. ffffffffffffffffffffffffffffffffff (Generated plaintext)

7ffffffffffffffffffffffffffffffff (One bit flipped from generated plaintext, from (f=1111))

then we get (7= 0111) because the first bit is flipped from 1 to 0;

We encrypted the two plaintext separately (generated and flipped one) and we got the following two ciphertext:

592373540ae1b202615e6d210d868a8c (Ciphertext of generated plaintext)

500ebff928c4892891726dcd29bd5469 (Ciphertext of one bit flipped from original plaintext)

The above red cipher indicates that, by simply flipping one bit from left (f to **7**) the whole ciphertext changed, but not all bits in ciphertext change, then we calculated the position where the bits are not the same between generated cipher and the flipped cipher. In this example we found there is 58 bits difference out of 128.

Bits different from original ciphertext:

58 bits different from original cipher. The value of 58 bit difference is calculated as follows:

- iv. From above we have ciphertext from original plaintext which is 592373540ae1b202615e6d210d868a8c is equivalent to 010110010010001101110011010101000000101011100001101100100000001001100

00101011110011011010010000100001101100001101000101010001100 in binary and ciphertext generated when one bit is flipped which is

500ebff928c4892891726dcd29bd5469

Is equivalent to

010100000000111010111111111100100101000110001001000100100101000100100101110010011011011100110100101001101111010101010001101001 in binary.

- v. Observing these two ciphertext. One can count how many bit positions that are not the same (equal) to each other at specific position when two ciphertext are compared. One can see that they are 58 bits positions that make two string not the same. That where 58 bits different value come from. We continued with the flipping the bits until the last bit therefore we had to repeat step i. to v. 128 to get better results. Then we calculated the average number of bit difference as follows:

Average number of flipped bits in ciphertext =

$$\frac{\text{sum of bits different from original cipher}}{128 \text{ (Since we did flipping 128 times)}}$$

Then, the final step is to calculate the avalanche effect

- vi. Avalanche effect (for this example AES) = $\frac{\text{Average number of flipped bits in ciphertext}}{128 \text{ (Size of CipherText of AES is 128)}} * 100\%$

100%

- vii. We continued flipping the bit from left to right one at time as follows:

ffffffffffffffffffffffffffffffff (original plaintext)

bffffffffffffffffffffffffffffffff (2nd bit of original plaintext flipped (f=1111) and b= 1011)

592373540ae1b202615e6d210d868a8c (original ciphertext)

2f2931ae1db3c3135d33123a3844bdea (2nd bit flipped ciphertext output)

Bits different from original ciphertext:

64 bits different from original cipher

- viii. ffffffffffffffffffffffffffffffffff (original plaintext)

dfffffffffffffffffffffffffffffffff (3rd bit of original plaintext flipped (f=1111) and d=1101)

592373540ae1b202615e6d210d868a8c (original ciphertext)

6f63de38aa9bf55506608bb74c6ef130 (Ciphertext of flipped bit of original plaintext)

66 bits different from original cipher,

- ix. ffffffffffffffffffffffffffffffffff (original plaintext)
 efffffffffffffffffffffffffffffffff (4th bit of original plaintext flipped (f=1111) and e=1110)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 c863152dc28ccc91cb58f27ce6a2d0f8 (4th bit flipped ciphertext output)
 64 bits different from original cipher
- x. ffffffffffffffffffffffffffffffffff (original plaintext)
 f7fffffffffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 d93f2264665062cc977a19dd64976bcb (Ciphertext of flipped bit of original plaintext)
 57 bits different from original cipher
- xi. ffffffffffffffffffffffffffffffffff (original plaintext)
 fbfffffffffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 f81f2b649cc19391e0736673db64ed11 (Ciphertext of flipped bit of original plaintext)
 54 bits different from original cipher
- xii. ffffffffffffffffffffffffffffffffff (original plaintext)
 fdfffffffffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 f3f4af2dcca6ba6fbef35c613b213bae (Ciphertext of flipped bit of original plaintext)
 65 bits different from original cipher
- xiii. ffffffffffffffffffffffffffffffffff (original plaintext)
 fefffffffffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 f3477c356117d03d89c3d76f219573d7 (Ciphertext of flipped bit of original plaintext)
 69 bits different from original cipher
- xiv. ffffffffffffffffffffffffffffffffff (original plaintext)
 ff7fffffffffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 856fef909f5738dc681d5fa254a600f7 (Ciphertext of flipped bit of original plaintext)
 58 bits different from original cipher
- xv. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffbfffffffffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 ce70cd7d75029d4520bb750432d4dc8f (Ciphertext of flipped bit of original plaintext)

- 66 bits different from original cipher
- xvi. ffffffffffffffffffffffffffffffffff (original plaintext)
 fdfxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 e58946d8a9817449af3768a45bdb0464 (Ciphertext of flipped bit of original plaintext)
- 61 bits different from original cipher
- xvii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffeffffffffxxxxxxxxxxxxxxxxxxxxx (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 04baa1f38785c3572b434da01ebc1c46 (Ciphertext of flipped bit of original plaintext)
- 58 bits different from original cipher
- xviii. ffffffffffffffffffffffffffffffffff (original plaintext)
 fff7ffffffffxxxxxxxxxxxxxxxxxxxx (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 56a1cf7ff0d271bc7ee2a86fe085dd11 (Ciphertext of flipped bit of original plaintext)
- 71 bits different from original cipher
- xix. ffffffffffffffffffffffffffffffffff (original plaintext)
 fffbxxxxxxxxxxxxxxxxxxxxxxxxxxxx (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 6b3c495acb0f472bf6ebd44f0d125625 (Ciphertext of flipped bit of original plaintext)
- 65 bits different from original cipher
- xx. ffffffffffffffffffffffffffffffffff (original plaintext)
 fffdxxxxxxxxxxxxxxxxxxxxxxxxxxxx (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 be7ce2a3cb38cd020e4eb8ad8b7f69f3 (Ciphertext of flipped bit of original plaintext)
- 73 bits different from original cipher
- xxi. ffffffffffffffffffffffffffffffffff (original plaintext)
 fffexxxxxxxxxxxxxxxxxxxxxxxxxxxx (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 e9303e1844784a127dabdc71db55ddb (Ciphertext of flipped bit of original plaintext)
- 61 bits different from original cipher
- xxii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffff7xxxxxxxxxxxxxxxxxxxxxxxxxxx (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)

- 26a4ad7e59cae2b48c11d86113a4888f (Ciphertext of flipped bit of original plaintext)
61 bits different from original cipher
- xxiii. ffffffff (original plaintext)
ffffb (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
9e60b950a5645b53bec408c8d02fd9e (Ciphertext of flipped bit of original plaintext)
61 bits different from original cipher
- xxiv. ffffffff (original plaintext)
ffffd (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
3c8e82b81bd97a22e0ce995b2a88957a (Ciphertext of flipped bit of original plaintext)
60 bits different from original cipher
- xxv. ffffffff (original plaintext)
ffffe (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
b960e21b42fed7474cfdc2355028a4e3 (Ciphertext of flipped bit of original plaintext)
64 bits different from original cipher
- xxvi. ffffffff (original plaintext)
fffff7 (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
f140793b9eea8356d2d2d7e27b96da50 (Ciphertext of flipped bit of original plaintext)
57 bits different from original cipher
- xxvii. ffffffff (original plaintext)
fffffb (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
7612e4e3e277716d97e92b8a276af515 (Ciphertext of flipped bit of original plaintext)
76 bits different from original cipher
- xxviii. ffffffff (original plaintext)
fffffd (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
89490174a4140da2dda5b044f6556555 (Ciphertext of flipped bit of original plaintext)
78 bits different from original cipher
- xxix. ffffffff (original plaintext)
fffffe (One bit flipped from original plaintext)

592373540ae1b202615e6d210d868a8c (original ciphertext)
9b6b3fd8c0a3fcacfa39a31add40f3d7 (Ciphertext of flipped bit of original plaintext)
63 bits different from original cipher

xxx. ffffffff (original plaintext)
fffff7fffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
efbae24054097bd2f9ee6eab511fac5a (Ciphertext of flipped bit of original plaintext)
57 bits different from original cipher

xxxii. ffffffff (original plaintext)
fffffbfffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
230f778028cca06309caf8f7919e88fc (Ciphertext of flipped bit of original plaintext)
49 bits different from original cipher

xxxiii. ffffffff (original plaintext)
fffffdfffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
b5931327b552231245f2597eb158e246 (Ciphertext of flipped bit of original plaintext)
64 bits different from original cipher

xxxiiii. ffffffff (original plaintext)
fffffefffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
ecb7a8d57857344b9d833bebe7f4bc58 (Ciphertext of flipped bit of original plaintext)
68 bits different from original cipher

xxxv. ffffffff (original plaintext)
fffff7fffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
71405f4b70abebbd6322202c4fae8ae (Ciphertext of flipped bit of original plaintext)
53 bits different from original cipher

xxxvi. ffffffff (original plaintext)
fffffbbfffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
e4962471b22828629e4938b167dc74aa (Ciphertext of flipped bit of original plaintext)
69 bits different from original cipher

xxxvii. ffffffff (original plaintext)

ffffffffdfffffffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
a4a4a4b3a2810708031f377ad6dd5490 (Ciphertext of flipped bit of original plaintext)
69 bits different from original cipher

xxxvii. ffffffffffffffffffffffffffffffff (original plaintext)
ffffffffefffffffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
7caabf59aca85007d61c5b4000ec09eb (Ciphertext of flipped bit of original plaintext)
56 bits different from original cipher

xxxviii. ffffffffffffffffffffffffffffffff (original plaintext)
fffffff7fffffffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
744e492605199b52e7c50bbb280aba96 (Ciphertext of flipped bit of original plaintext)
58 bits different from original cipher

xxxix. ffffffffffffffffffffffffffffffff (original plaintext)
fffffffbffffffffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
44ce4cd592a81e30ef0fd6cbbfc18f5b (Ciphertext of flipped bit of original plaintext)
65 bits different from original cipher

xl. ffffffffffffffffffffffffffffffff (original plaintext)
ffffffffdfffffffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
523d9d4b0835703bd1f1fc0959db8f77 (Ciphertext of flipped bit of original plaintext)
61 bits different from original cipher

xli. ffffffffffffffffffffffffffffffff (original plaintext)
fffffffeffffffffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
dc0877e53f347b270d6931a8b64e5ef2 (Ciphertext of flipped bit of original plaintext)
63 bits different from original cipher

xlii. ffffffffffffffffffffffffffffffff (original plaintext)
fffffff7fffffffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
9e1b485e3d52312be04eab08c490f422 (Ciphertext of flipped bit of original plaintext)
59 bits different from original cipher

- xliii. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffbfffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 f94009050a302a5e2e7621ca66b07ce7 (Ciphertext of flipped bit of original plaintext)
 61 bits different from original cipher
- xliv. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffdfffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 3ce2294d6daf9b06d0fa7d076a9a1cad (Ciphertext of flipped bit of original plaintext)
 52 bits different from original cipher
- xlv. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffeffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 b639f7d1793f47fa8d9bca98f0e13ba8 (Ciphertext of flipped bit of original plaintext)
 74 bits different from original cipher
- xlvi. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffff7fffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 c560efeb8a248924e5af60dcb6abba41 (Ciphertext of flipped bit of original plaintext)
 65 bits different from original cipher
- xlvii. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffbfffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 9418f75e97ad6d1ef2a9d70c40ee9175 (Ciphertext of flipped bit of original plaintext)
 69 bits different from original cipher
- xlviii. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffdfffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 e894666c49b161c66fe05629f4e13d2f (Ciphertext of flipped bit of original plaintext)
 65 bits different from original cipher
- xliv. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffeffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 d4a95149c0dd95a973f662eea4ff8047 (Ciphertext of flipped bit of original plaintext)

- 61 bits different from original cipher
- i. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffff7fffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 458ad2de5342eea7e094ff2a19bfa853 (Ciphertext of flipped bit of original plaintext)
- 56 bits different from original cipher
- ii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffbfffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 baf5719e039e0baa26fb6f41f3acdf22 (Ciphertext of flipped bit of original plaintext)
- 62 bits different from original cipher
- iii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffdfffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 411f5b2578a62656d5c244c9fdde8f12 (Ciphertext of flipped bit of original plaintext)
- 55 bits different from original cipher
- liii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffeffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 16ee9d358cef19f9c4ac4c707eec4bd3 (Ciphertext of flipped bit of original plaintext)
- 69 bits different from original cipher
- liv. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffff7fffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 6217d8ef40dea047d763dc1a42282043 (Ciphertext of flipped bit of original plaintext)
- 72 bits different from original cipher
- lv. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffbfffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 99a9b7138f946d3c8407079229e5c218 (Ciphertext of flipped bit of original plaintext)
- 61 bits different from original cipher
- lvi. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffdfffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)

- da0cd45f803a92e814539afd52e920e9 (Ciphertext of flipped bit of original plaintext)
71 bits different from original cipher
- lvii. ffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffefffffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
2d8399d13b166995acb53cf8136a1d96 (Ciphertext of flipped bit of original plaintext)
71 bits different from original cipher
- lviii. ffffffffffffffffffffffffffffffff(original plaintext)
fffffffffffff7fffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
b6c8de76732504745e52478d8bf63c1f (Ciphertext of flipped bit of original plaintext)
68 bits different from original cipher
- lix. ffffffffffffffffffffffffffffffff (original plaintext)
fffffffffffffbfffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
f42c0e82e138e57fbacbcfb5c42bc9a2 (Ciphertext of flipped bit of original plaintext)
74 bits different from original cipher
- lx. ffffffffffffffffffffffffffffffff (original plaintext)
fffffffffffffdfffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
9ca356d8eecea665e5967de034fae1e4 (Ciphertext of flipped bit of original plaintext)
53 bits different from original cipher
- lxi. ffffffffffffffffffffffffffffffff (original plaintext)
fffffffffffffefffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
17b0c3cde5b98e1a7faf6863260a77ab (Ciphertext of flipped bit of original plaintext)
62 bits different from original cipher
- lxii. ffffffffffffffffffffffffffffffff (original plaintext)
fffffffffffff7fffffffffffffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
401d3b362f83a378af808e6a77720eb6 (Ciphertext of flipped bit of original plaintext)
62 bits different from original cipher
- lxiii. ffffffffffffffffffffffffffffffff (original plaintext)
fffffffffffffbfffffffffffffffffff (One bit flipped from original plaintext)

- 592373540ae1b202615e6d210d868a8c (original ciphertext)
7ffb2b2c0d3aeb2d810dad9ae48fdb21 (Ciphertext of flipped bit of original plaintext)
62 bits different from original cipher
- lxiv. ffffffff (original plaintext)
fffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
011276d249e33e27fc640ee9a9221786 (Ciphertext of flipped bit of original plaintext)
50 bits different from original cipher
- lxv. ffffffff (original plaintext)
fffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
49c8614390ba9b9234a9a4aec4970f11 (Ciphertext of flipped bit of original plaintext)
61 bits different from original cipher
- lxvi. ffffffff (original plaintext)
fffffff7 (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
efeedc022f929740a2e61e91a192da9d (Ciphertext of flipped bit of original plaintext)
59 bits different from original cipher
- lxvii. ffffffff (original plaintext)
fffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
f58188269a60cec4d05342dedd49604d (Ciphertext of flipped bit of original plaintext)
68 bits different from original cipher
- lxviii. ffffffff (original plaintext)
fffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
cd9b49eabc543191b491d5a8c72fa97e (Ciphertext of flipped bit of original plaintext)
68 bits different from original cipher
- lxix. ffffffff (original plaintext)
fffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
e92902412c2574eb4af3e00ea79627fb (Ciphertext of flipped bit of original plaintext)
61 bits different from original cipher
- lxx. ffffffff (original plaintext)

- ffffffffffffffff7fffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 7e65e820163d8675b656aac208fb21c8 (Ciphertext of flipped bit of original plaintext)
 65 bits different from original cipher
- lxxi.
 - ffffffffffffffffffffffffffffffff (original plaintext)
 - ffffffffffffffffbfffffffffffffff (One bit flipped from original plaintext)
 - 592373540ae1b202615e6d210d868a8c (original ciphertext)
 - cc591ef3f644c21056eaf635838b4e8e (Ciphertext of flipped bit of original plaintext)
 - 61 bits different from original cipher
- lxxii.
 - ffffffffffffffffffffffffffffffff (original plaintext)
 - ffffffffffffffffdfffffffffffffff (One bit flipped from original plaintext)
 - 592373540ae1b202615e6d210d868a8c (original ciphertext)
 - bcc7c3ccc95a4011833e76ecf37c89a (Ciphertext of flipped bit of original plaintext)
 - 58 bits different from original cipher
- lxxiii.
 - ffffffffffffffffffffffffffffffff (original plaintext)
 - fffffffffffffffffefffffffffffffff (One bit flipped from original plaintext)
 - 592373540ae1b202615e6d210d868a8c (original ciphertext)
 - f0127df46118504a57f10434a15be4c3 (Ciphertext of flipped bit of original plaintext)
 - 66 bits different from original cipher
- lxxiv.
 - ffffffffffffffffffffffffffffffff (original plaintext)
 - fffffffffffffffff7fffffffffffffff (One bit flipped from original plaintext)
 - 592373540ae1b202615e6d210d868a8c (original ciphertext)
 - 7e533318c924ad29d13329cfda90f8cd (Ciphertext of flipped bit of original plaintext)
 - 59 bits different from original cipher
- lxxv.
 - ffffffffffffffffffffffffffffffff (original plaintext)
 - fffffffffffffffffbfffffffffffffff (One bit flipped from original plaintext)
 - 592373540ae1b202615e6d210d868a8c (original ciphertext)
 - 325297bb143918088bf5ad4a51dc1fed (Ciphertext of flipped bit of original plaintext)
 - 66 bits different from original cipher
- lxxvi.
 - ffffffffffffffffffffffffffffffff (original plaintext)
 - fffffffffffffffffdfffffffffffffff (One bit flipped from original plaintext)
 - 592373540ae1b202615e6d210d868a8c (original ciphertext)
 - 5f53c31d84021839665f5fe2fb2349a5 (Ciphertext of flipped bit of original plaintext)
 - 57 bits different from original cipher

- lxxvii. ffffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffffffffffffffe (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 e6f3492ca8cab8dc87988224edc016cd (Ciphertext of flipped bit of original plaintext)
 63 bits different from original cipher
- lxxviii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffff7 (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 13342168a3ea88c03b8e7c7d87494033 (Ciphertext of flipped bit of original plaintext)
 61 bits different from original cipher
- lxxix. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffbf (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 fa90fb641b1a3a8951dfeceddf7bfa52 (Ciphertext of flipped bit of original plaintext)
 58 bits different from original cipher
- lxxx. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffdf (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 34f72f82b73c9b9350bf435fd3bae7fc (Ciphertext of flipped bit of original plaintext)
 71 bits different from original cipher
- lxxxi. ffffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffffffffffffffe (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 4f6fdc67a014d0107090abfeb654570f (Ciphertext of flipped bit of original plaintext)
 68 bits different from original cipher
- lxxxii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffff7 (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 7eff481e6b8e53b3809f113e3335db61 (Ciphertext of flipped bit of original plaintext)
 70 bits different from original cipher
- lxxxiii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffbf (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 61f9da511bd2c644b5a4290e47b1ff5c (Ciphertext of flipped bit of original plaintext)

- 60 bits different from original cipher
- lxxxiv. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffffffffffffffdffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 9e08327f1430314b5166023cb4575060 (Ciphertext of flipped bit of original plaintext)
- 63 bits different from original cipher
- lxxxv. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffffffffffffffefffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 9ca0a0ea4783a6b4ff111b3a9932305f (Ciphertext of flipped bit of original plaintext)
- 68 bits different from original cipher
- lxxxvi. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffffffffffffff7ffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 66df2e07094e68917066fd3bce22f217 (Ciphertext of flipped bit of original plaintext)
- 64 bits different from original cipher
- lxxxvii. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffffffffffffffbffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 c9fc3c67a4cfd4eb343a19ff982c3c3a (Ciphertext of flipped bit of original plaintext)
- 71 bits different from original cipher
- lxxxviii. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffffffffffffffdffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 62b489194d166bd07bbd8f3c81a63c2f (Ciphertext of flipped bit of original plaintext)
- 69 bits different from original cipher
- lxxxix. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffffffffffffffefffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 d345d6bd56067254350871890d7f0f3f (Ciphertext of flipped bit of original plaintext)
- 59 bits different from original cipher
- xc. ffffffffffffffffffffffffffffffff (original plaintext)
 fffffffffffffffffffff7ffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)

- 1d4d86679786f47c9eb6b84e5807a2a0 (Ciphertext of flipped bit of original plaintext)
70 bits different from original cipher
- xci. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffbfffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
ea7ab8d885231476f449ffaee24d956 (Ciphertext of flipped bit of original plaintext)
65 bits different from original cipher
- xcii. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffdfffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
f88776e71b7b7dbdaf662f2517666cd9 (Ciphertext of flipped bit of original plaintext)
58 bits different from original cipher
- xciii. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffefffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
0a19887309846557ff99877514ccd5d5 (Ciphertext of flipped bit of original plaintext)
69 bits different from original cipher
- xciv. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffff7fffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
d242bb5b6c21cc5776f914469d12e497 (Ciphertext of flipped bit of original plaintext)
63 bits different from original cipher
- xcv. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffbfffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
66a7284e9e97e95380aeb94b2b327023 (Ciphertext of flipped bit of original plaintext)
67 bits different from original cipher
- xcvi. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffdfffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
da6f98470d2598dbd02e19059dc4336c (Ciphertext of flipped bit of original plaintext)
54 bits different from original cipher
- xcvii. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffefffffffff (One bit flipped from original plaintext)

592373540ae1b202615e6d210d868a8c (original ciphertext)
ee9ad0aa8f516843e886b0e94bcf5fce (Ciphertext of flipped bit of original plaintext)
64 bits different from original cipher

xcviii. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffff7fffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
a71fe038d313905e03deb1258173b35d
62 bits different from original cipher

xcix. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffffbfffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
0b5add380cbd5cdf42abd641a0a51d09 (Ciphertext of flipped bit of original plaintext)
68 bits different from original cipher

c. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffffdfffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
249f6a0a162fcae391e2e43c5a06f497 (Ciphertext of flipped bit of original plaintext)
67 bits different from original cipher

ci. ffffffffffffffffffffffffffffffffff (original plaintext)
fffffffffffffffffffffefffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
120a70fba96e811c885162325cbdea76 (Ciphertext of flipped bit of original plaintext)
64 bits different from original cipher

cii. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffff7fffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
9b17e0e5dd9e8f88ace9707a2c1569e8
70 bits different from original cipher

ciii. ffffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffffbfffffffff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
a7f04582f70d9bada744044249a12d80 (Ciphertext of flipped bit of original plaintext)
70 bits different from original cipher

civ. ffffffffffffffffffffffffffffffffff (original plaintext)

- ffffffffffffffffffffffdffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 9294db16aa5b5639c53b77f7ae5521ab (Ciphertext of flipped bit of original plaintext)
 65 bits different from original cipher
- cv. ffffffffffffffffffffffdffffff (original plaintext)
 ffffffffffffffffffffffeffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 70f1a4e3e5679e4c410705eb3bf6b5a5
 64 bits different from original cipher
- cvi. ffffffffffffffffffffffdffffff (original plaintext)
 ffffffffffffffffffffff7ffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 91ac421ee67c29b1043f8b087bf1d5f8 (Ciphertext of flipped bit of original plaintext)
 70 bits different from original cipher
- cvii. ffffffffffffffffffffffdffffff (original plaintext)
 ffffffffffffffffffffffbffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 4582211b2838320a906e95de1cc37944 (Ciphertext of flipped bit of original plaintext)
 57 bits different from original cipher
- cviii. ffffffffffffffffffffffdffffff (original plaintext)
 ffffffffffffffffffffffdffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 ee4e9e5bea6640526ce786f122c37898 (Ciphertext of flipped bit of original plaintext)
 67 bits different from original cipher
- cix. ffffffffffffffffffffffdffffff (original plaintext)
 ffffffffffffffffffffffeffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 1abd53ee411527ec3b0b9a9ba11c56c3 (Ciphertext of flipped bit of original plaintext)
 71 bits different from original cipher
- cx. ffffffffffffffffffffffdffffff (original plaintext)
 ffffffffffffffffffffff7ffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 a5f9c694626cafa6ff357ad21b16506e (Ciphertext of flipped bit of original plaintext)
 66 bits different from original cipher

- cxviii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffbffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 4dea41e5fef61b5bca1d607d08d17973 (Ciphertext of flipped bit of original plaintext)
 66 bits different from original cipher
- cxix. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffdffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 e05b00f9d998fe609067357de91de79c (Ciphertext of flipped bit of original plaintext)
 66 bits different from original cipher
- cx. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffeffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 8d75d5919c9a918e8bf64a34f3f83e73 (Ciphertext of flipped bit of original plaintext)
 72 bits different from original cipher
- cxvi. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffff7ffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 7212325a88e044fc58172f372fc823ba (Ciphertext of flipped bit of original plaintext)
 54 bits different from original cipher
- cxvii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffbffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 c0599d215a45b5c20651ba88228eaa5a (Ciphertext of flipped bit of original plaintext)
 61 bits different from original cipher
- cxviii. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffdffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 a4ac3a9b8d2a84b069b659ce3415b190 (Ciphertext of flipped bit of original plaintext)
 69 bits different from original cipher
- cxix. ffffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffeffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 5aa1df70da5713137e35ffb5b24ee5ca (Ciphertext of flipped bit of original plaintext)

- 58 bits different from original cipher
- cxviii. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffff7fff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 fbca7c43152fb1ac83ab480757541708 (Ciphertext of flipped bit of original plaintext)
- 64 bits different from original cipher
- cxix. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffffbfff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 d02119cb35f5dab4905b1d0e6744734d (Ciphertext of flipped bit of original plaintext)
- 61 bits different from original cipher
- cxx. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffffdf (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 6c0ea469962f063a824ababb914713b9 (Ciphertext of flipped bit of original plaintext)
- 67 bits different from original cipher
- cxxi. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffffef (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 cd4ef41ef215060dce6b3330bd09f53f (Ciphertext of flipped bit of original plaintext)
- 70 bits different from original cipher
- cxxii. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffff7ff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 5db5182775f468b608a55d1c58b07e3c (Ciphertext of flipped bit of original plaintext)
- 68 bits different from original cipher
- cxxiii. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffffbff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 2ba6bc9eb9beb9205076461c02b4bf91 (Ciphertext of flipped bit of original plaintext)
- 62 bits different from original cipher
- cxxiv. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffffdf (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)

- 965ebf5d678c3cdb350efbbfd701efc (Ciphertext of flipped bit of original plaintext)
66 bits different from original cipher
- cxxv. ffffffffffffffffffffffffffffffff (original plaintext)
fffffffffffffffffffffffffffffeff (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
86ed39d2248fa0c2bfb5160b0a639e22 (Ciphertext of flipped bit of original plaintext)
67 bits different from original cipher
- cxxvi. ffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffffffff7f (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
b458525a0cd23e0dec5dd74d09c050e5 (Ciphertext of flipped bit of original plaintext)
58 bits different from original cipher
- cxxvii. ffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffffffffbf (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
6359b9cf7cc2b390af7a544f9a66c61a (Ciphertext of flipped bit of original plaintext)
61 bits different from original cipher
- cxxviii. ffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffffffffdf (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
9539e6f563b6ca3a2e387a6bd1ad125c (Ciphertext of flipped bit of original plaintext)
61 bits different from original cipher
- cxxix. ffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffffffffef (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
3142d7ce79327988575aa5c66f0a3303 (Ciphertext of flipped bit of original plaintext)
61 bits different from original cipher
- cxxx. ffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffffffff7 (One bit flipped from original plaintext)
592373540ae1b202615e6d210d868a8c (original ciphertext)
23c8d1a7f913b68454421daa1f6e29a3 (Ciphertext of flipped bit of original plaintext)
64 bits different from original cipher
- cxxxi. ffffffffffffffffffffffffffffffff (original plaintext)
ffffffffffffffffffffffffffffb (One bit flipped from original plaintext)

592373540ae1b202615e6d210d868a8c (original ciphertext)
 e4772ee29fb6df1410132d43f2d6a3a1 (Ciphertext of flipped bit of original plaintext)
 65 bits different from original cipher

cxxxii. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 f3c5fb96ff8bf5835d6ca8b8c85b7ff0 (Ciphertext of flipped bit of original plaintext)
 66 bits different from original cipher

cxxxiii. ffffffffffffffffffffffffffffffff (original plaintext)
 ffffffffffffffffffffffffffffffff (One bit flipped from original plaintext)
 592373540ae1b202615e6d210d868a8c (original ciphertext)
 fb781891293b8083140d91f7e274c6b1 (Ciphertext of flipped bit of original plaintext)
 70 bits different from original cipher

cxxxiv. Average number of flipped bits in ciphertext =

$$\frac{\text{sum of bits different from original cipher}}{128 \text{ (Since we did flipping 128 times)}} = 63.492188$$

cxxxv. Then, the final step is to calculate the avalanche effect

cxxxvi. Avalance effect (for this example AES) = $\frac{63.492188}{128 \text{ (Size of CipherText of AES is 128)}} * 100\% = 49.603271875\%$

cxxxvii. There are algorithm that are less than 128 bit size, like DES. DES is 64 bit.

cxxxviii. Then instead of using ffffffffffffffffffffffffffffffff (128 bits) used on AES we used ffffffffffffffff (64 bits) and instead of using IntialVector[16]= {0xC2,0x9B,0x7C,0x97,0xC5,0x0D,0xD3,0xF8,0x4D,0x5B,0x5B,0x54,0x70,0x91,0x79,0x21} ; and the final vector is array defined as unsigned char FinalVector[16]= {0xBA,0x69,0x8D,0xFB,0x5A,0xC2,0xFF,0xD7,0x2D,0xBD,0x01,0xAD,0xFB,0x7B,0x8E,0x1A} ;

cxxxix. We used IntialVector[8]= {0xC2,0x9B,0x7C,0x97,0xC5,0x0D,0xD3,0xF8}, and the finalvector is array defined as unsigned char FinalVector[8]= {0xBA,0x69,0x8D,0xFB,0x5A,0xC2,0xFF,0xD7};

APPENDIX 2: The value of PI in hexadecimal after the first digit [122].

243F6A8885A308D313198A2E03707344A4093822299F31D0082EFA98EC4E6C89452821
E638D01377BE5466CF34E90C6CC0AC29B7C97C50DD3F84D5B5B54709179216D5D98
979FB1BD1310BA698DFB5AC2FFD72DBD01ADFB7B8E1AFED6A267E96BA7C9045F
12C7F9924A19947B3916CF70801F2.....[122].

