FAULT ANALYSIS OF AUTHENTICATED ENCRYPTION SYSTEMS

A THESIS

Submitted in partial fulfillment of the
requirements for the award of the degree
of
Master of Technology

by

Lt Col O P Mishra

CENTER FOR ELECTRIC VEHICLE AND INTELLIGENT TRANSPORT SYSTEMS,
INDIAN INSTITUTE OF TECHNOLOGY, INDORE

MAY 2023

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **FAULT ANALYSIS OF AUTHENTICATED ENCRYPTION SYSTEMS** in the partial fulfillment of the requirements for the award of the degree of **MASTER OF TECHNOLOGY** and submitted in the **Center for Electric Vehicle and Intelligent Transport Systems, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from Jun 2022 to May 2023 under the supervision of **Dr Bodhisatwa Mazumdar, Associate Professor, PhD Coordinator, Department of Computer Science and Engineering, Indian Institute of Technology Indore.**

The matter presented in this thesis has not been submitted by me for the award of any other degree to this or any other institute.

05 JUN 23

(Lt Col O P Mishra)

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

Bodhisatwa Mazumdar
06-06-2023

(Dr. Bodhisatwa Mazumdar)

Lt Col O P Mishra has successfully given his M.Tech. Oral Examination held on **May 18, 2023.**

Bodhisatwa Mazumdar

Signature(s) of Supervisor(s) of M.Tech. thesis
Date: 06-06-2023

Convener, DPGC
Date:

Abhijit Ghosh

Signature of PSPC Member #1
(**Prof. Abhijeet Ghosh**)
Date: 08/06/23

Signature of PSPC Member #2
(**Prof. Neminath Hubbali**)
Date: 09-06-2023

ACKNOWLEDGEMENT

My deep gratitude goes firstly to my supervisor, **Dr. Bodhisatwa Mazumdar**, who expertly guided me throughout my two years of Master of Technology. I was fortunate to have an advisor who offered me the constant motivation and productive support that preceded this work to attain this form. I would also like to thank my PSPC members, **Prof. Abhijeet Ghosh** and **Prof. Neminath Hubbali** for their continuous inputs for the advancement of this project.

A very special thanks to my batchmates Mr. Maitreya Jain (M Tech. scholar), Mr. Vivek Paliwal (M Tech. scholar), and Mr. Prashant Mall (M Tech. scholar) for their moral support and supportive environment for the completion of the project. I am also thankful to **Prof. Srivathsan Vasudevan**, IIT Indore.

I want to extend my deepest appreciation and admiration to my beloved parents **Shri Yashodhar Mishra** and **Mrs Kunti Mishra** for their invaluable blessings and unwavering support throughout my M Tech project. I am truly grateful to my Wife, **Shashi Mishra** whose constant encouragement has boosted my spirits and kept me motivated. I would also like to express my heartfelt thanks to all those who have provided assistance, whether directly or indirectly, during this project.

With Regards,

Lt Col O P Mishra

Dedicated

to

my beloved

Mom & Dad

Wife & Kids

Abstract

This project report presents the implementation and analysis of the GIFT cipher, a lightweight symmetric encryption algorithm, using the Python programming language. The GIFT cipher has gained significant attention in the field of cryptography due to its impressive security features and efficient performance.

The report begins with an overview of the GIFT cipher's structure and operation, highlighting its key components such as the Substitution-Permutation Network (SPN) and the Feistel-like structure. The Python implementation of the GIFT cipher is discussed in detail, covering the encryption and decryption processes along with the generation of round keys.

Furthermore, a comparative analysis is conducted to showcase the advantages of the GIFT cipher over its competitors. The key benefits of the GIFT cipher include:

- **Lightweight Design**: The GIFT cipher is specifically designed to be highly efficient on resource-constrained devices such as low-power microcontrollers and Internet of Things (IoT) devices. It offers a balance between security and computational requirements, making it ideal for lightweight applications.
- **Strong Security**: Despite its lightweight nature, the GIFT cipher provides a high level of security. It offers resistance against various cryptographic attacks, including differential and linear attacks. The careful design choices in GIFT's round function and key schedule contribute to its robustness.
- **Fast Execution**: The GIFT cipher demonstrates remarkable speed in both software and hardware implementations. Its compact design and efficient algorithms enable quick encryption and decryption operations, making it suitable for real-time applications and systems with limited processing power, as in Smart and Connected Vehicles.
- **Minimal Memory Footprint**: GIFT requires relatively low memory resources, making it well-suited for devices with limited storage capacity. This characteristic makes it a viable choice for embedded systems and lightweight cryptographic applications.

The report concludes with a discussion on potential future research and improvements for the GIFT cipher, as well as its applications in various domains requiring lightweight and secure encryption. Overall, the GIFT cipher proves to be a promising solution for achieving a balance between security and efficiency in lightweight cryptographic systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ALGORITHMS

Chapter 1: Ciphers- A Detailed Overview

1.1. Introduction

Ciphers play a crucial role in the field of cryptography, which is the practice of secure communication in the presence of adversaries. A cipher is an algorithm or method used to encrypt and decrypt messages, transforming plaintext (readable form) into ciphertext (unreadable form) and vice versa. Throughout history, ciphers have been used to protect sensitive information, maintain privacy, and ensure secure communication between individuals or organizations. This document explores the origin, initial developments, need for ciphers, use case scenarios and the latest trends in the field.

1.2. Origin of Ciphers

The use of ciphers dates back thousands of years. Ancient civilizations, such as the Egyptians and Greeks, employed various encryption techniques. One of the earliest known ciphers is the Caesar cipher, named after Julius Caesar, who used it to communicate with his generals. The Caesar cipher involves shifting each letter in the plaintext by a fixed number of positions in the alphabet. For instance, with a shift of 3, "HELLO" would become "KHOOR."

1.3. Initial Developments

Over time, more sophisticated ciphers were developed to enhance security. One notable development was the invention of the polyalphabetic cipher by Leon Battista Alberti in the 15th century. This cipher used multiple alphabets and variable substitution, making it more resistant to frequency analysis attacks.

The 19th century witnessed the emergence of mechanical cipher machines like the Enigma machine, which was famously used by the German military during World War II. These machines utilized complex rotors and electrical circuits to perform encryption and decryption operations.

The development of computers in the 20th century revolutionized cryptography. Symmetric-key ciphers like the Data Encryption Standard (DES) and Advanced Encryption Standard (AES) were introduced, along with asymmetric-key ciphers like the RSA algorithm. These algorithms formed the basis for modern cryptographic systems.

1.4. Need for Ciphers: Ciphers serve several critical purposes in today's digital world:

1.4.1 Confidentiality

Ciphers ensure the confidentiality of information by encrypting it. Only authorized individuals possessing the correct decryption key can transform the ciphertext back into its original plaintext form. This is crucial for protecting sensitive data such as personal information, financial transactions, and classified government communications.

1.4.2 **Integrity**

Ciphers can also provide integrity by enabling the detection of any unauthorized modifications to the encrypted data. Cryptographic hash functions, which generate fixed-size unique outputs (hashes) for variable-size inputs, are often used in conjunction with ciphers to ensure the integrity of messages.

1.4.3 **Authentication**

Ciphers play a role in authentication mechanisms. Digital signatures, which are created using asymmetric-key ciphers, allow the recipient to verify the sender's identity and ensure the message's integrity. This is particularly important in electronic transactions and secure communication between parties.

1.4.4 **Non-Repudiation**

Non-repudiation refers to the ability to prove that a message was sent or received by a particular party, preventing them from denying their involvement. Ciphers contribute to non-repudiation by providing digital signatures and ensuring that messages cannot be tampered with or forged.

1.5. **Use Case Scenarios** Ciphers find applications in various domains:

1.5.1 **Secure Communication**

Ciphers are extensively used for secure communication over the internet. Secure protocols like Transport Layer Security (TLS) and Secure Shell (SSH) utilize ciphers to encrypt data transmitted between clients and servers. This ensures that sensitive information, such as passwords, credit card details, and personal messages, remains confidential.

1.5.2 **Data Storage**

Ciphers are crucial in protecting stored data, such as databases, file systems, and cloud storage. Encryption algorithms like AES are employed to encrypt data at rest, safeguarding it from unauthorized access even if physical storage devices are compromised.

1.5.3 **Military and Government Applications**

Ciphers have always been integral to military and government operations. They are used for secure communication between military personnel, intelligence agencies, and diplomats. Ciphers ensure the secrecy and integrity of classified information critical for national security.

1.5.4 **Financial Transactions**

Ciphers are extensively employed in financial systems to secure online banking, electronic funds transfer, and digital currency transactions. The use of encryption guarantees the confidentiality and integrity of financial data, preventing unauthorized access and fraud.

1.6. **Latest Trends**

Cryptography and ciphers continue to evolve, driven by advancements in computing power and emerging threats. Some notable trends include:

1.6.1 **Quantum Cryptography**

The advent of quantum computing poses a potential threat to traditional encryption algorithms. Quantum cryptography, which relies on the principles of quantum mechanics, aims to develop cryptographic systems that are resistant to quantum attacks. Techniques such as quantum key distribution (QKD) offer the promise of secure key exchange over insecure channels.

1.6.2 **Homomorphic Encryption**

Homomorphic encryption allows computations to be performed directly on encrypted data without requiring decryption. This emerging field of cryptography has significant implications for secure data processing, enabling confidential computations on sensitive data while maintaining privacy.

1.6.3 **Post-Quantum Cryptography**

Post-quantum cryptography focuses on developing cryptographic algorithms that are resistant to attacks by both classical and quantum computers. As quantum computers become more powerful, post-quantum cryptographic algorithms aim to ensure the long-term security of encrypted data.

1.7. **Conclusion**

Ciphers have a rich history and continue to play a crucial role in ensuring the confidentiality, integrity, authentication, and non-repudiation of digital information. They find applications in secure communication, data storage, military operations, and financial transactions. As technology advances, new challenges and trends such as quantum cryptography, homomorphic encryption, and post-quantum cryptography drive the development of more secure and robust ciphers, enabling us to protect sensitive information in an ever-evolving digital landscape.

Chapter 2: Light Weight Ciphers: A Detailed Overview

2.1 Introduction

Lightweight ciphers are cryptographic algorithms specifically designed to provide secure and efficient encryption for resource-constrained devices. These devices often have limited processing power, memory, and energy capabilities. Lightweight ciphers aim to strike a balance between security and efficiency, making them suitable for low-power devices such as IoT devices, embedded systems, and wireless sensors. This chapter provides a comprehensive overview of lightweight ciphers, including their origin, initial developments, need, use case scenarios, and latest trends.

2.2 Origin of Lightweight Ciphers

The need for lightweight ciphers emerged with the proliferation of resource-constrained devices that require cryptographic protection. The field of lightweight cryptography gained prominence in the early 2000s when researchers recognized the limitations of traditional cryptographic algorithms for such devices.

2.3. Initial Developments

The initial developments in lightweight ciphers focused on designing algorithms that provided strong security while minimizing computational overhead and memory requirements. Notable early lightweight ciphers include:

2.3.1 Skipjack

Skipjack is a symmetric-key block cipher developed by the National Security Agency (NSA) in the 1990s. It was intended for use in the Clipper chip, a controversial government encryption initiative. Skipjack was designed to be efficient on 16-bit microcontrollers and had a 64-bit block size.

2.3.2 RC5

RC5 is a block cipher developed by Ronald Rivest in the mid-1990s. It is known for its flexibility and simplicity, making it suitable for resource-constrained devices. RC5 supports variable block sizes, key sizes, and number of rounds, allowing customization based on specific requirements.

2.3.3 Simon and Speck

Simon and Speck are a family of lightweight block ciphers developed by the National Security Agency (NSA) in 2013. They are designed to provide high performance and security with a small code footprint. Simon and Speck offer various block and key sizes, allowing customization for different applications.

2.4. **Need for Lightweight Ciphers**

The need for lightweight ciphers arises from the limitations of resource-constrained devices. These devices often have restricted computational power, memory, and energy resources, making traditional cryptographic algorithms impractical. The specific needs for lightweight ciphers include

2.4.1 **Security**

Resource-constrained devices still require secure communication and data protection. Lightweight ciphers provide a solution by offering cryptographic algorithms that are specifically designed to balance security and efficiency in constrained environments.

2.4.2 **Efficiency**

Lightweight ciphers aim to optimize performance on low-power devices. They prioritize efficient encryption and decryption operations to minimize computational overhead, memory usage, and energy consumption.

2.4.3 **Scalability**

Many lightweight ciphers offer customizable parameters such as block size, key size, and number of rounds. This scalability enables their adaptation to different resource constraints, allowing for flexibility in deployment across a wide range of devices.

2.5. **Use Case Scenarios**

Lightweight ciphers find applications in various domains where resource-constrained devices require secure communication and data protection. Some common use case scenarios include:

2.5.1 **Internet of Things (IoT) Devices**

Lightweight ciphers are essential for securing communication between IoT devices, which often operate with limited resources. They enable secure data exchange and ensure the privacy and integrity of sensitive information transmitted within IoT ecosystems.

2.5.2 **Embedded Systems**

Embedded systems, such as microcontrollers and smart cards, often have stringent constraints on processing power and memory. Lightweight ciphers provide secure encryption for these devices, allowing them to protect sensitive data in applications such as smart grids, automotive systems, and medical devices.

2.5.3 **Wireless Sensor Networks**

Wireless sensor networks rely on lightweight ciphers to secure communication between sensor nodes. These networks are commonly deployed in environments where resources are scarce, such as environmental monitoring, industrial automation, Smart and Connected Vehicles and surveillance systems.

2.6. **Latest Trends**

The field of lightweight cryptography continues to evolve to address emerging challenges and advancements in technology. Some notable trends in lightweight ciphers include:

2.6.1 **Hardware Acceleration**

To further enhance the efficiency of lightweight ciphers, there is a growing trend of utilizing hardware acceleration techniques. Hardware accelerators, such as dedicated cryptographic co-processors or Field-Programmable Gate Arrays (FPGAs), can offload cryptographic operations, improving performance and reducing energy consumption.

2.6.2 **Post-Quantum Lightweight Cryptography**

As the field of post-quantum cryptography advances, efforts are being made to develop lightweight cryptographic algorithms that are resistant to attacks by both classical and quantum computers. These algorithms aim to ensure the long-term security of lightweight devices in the post-quantum era.

2.6.3 **Standardization Efforts**

Standardization bodies, such as the National Institute of Standards and Technology (NIST), are actively involved in the evaluation and selection of lightweight cryptographic algorithms. Ongoing standardization efforts aim to provide industry-wide guidelines and recommendations for the implementation of lightweight ciphers.

2.7. **Conclusion**

Lightweight ciphers have emerged as a crucial component of cryptographic solutions for resource-constrained devices. Their focus on security, efficiency, and scalability makes them ideal for applications in IoT, embedded systems, and wireless sensor networks. As technology advances, trends like hardware acceleration, post-quantum lightweight cryptography, and standardization efforts continue to shape the field, ensuring the availability of secure and efficient encryption solutions for lightweight devices in the evolving digital landscape.

Chapter 3: Light Weight Ciphers in Electric Vehicles and Intelligent Transportation Systems

3.1 Introduction:

The implementation of lightweight ciphers in the domain of electric vehicles (EVs) and intelligent transportation systems (ITS) plays a crucial role in ensuring the security and privacy of data transmitted and stored within these systems.

3.2 Employment of Lightweight Ciphers:

Here is an elaboration on how lightweight ciphers are employed in these domains:

3.2.1 Secure Communication:

Lightweight ciphers are utilized to secure communication channels in EVs and ITS. EVs often rely on wireless communication protocols for vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication. Lightweight ciphers are employed to encrypt data transmitted over these channels, protecting it from unauthorized access and ensuring confidentiality. Lightweight ciphers are used to encrypt data transmitted over wireless communication protocols in EVs and ITS, such as Dedicated Short-Range Communications (DSRC) or Cellular Vehicle-to-Everything (C-V2X). These ciphers ensure that the data exchanged between vehicles, infrastructure components, and control centers remains confidential and protected from eavesdropping or unauthorized access.

Lightweight ciphers provide secure encryption and decryption operations with optimized computational efficiency, allowing real-time communication between vehicles and infrastructure.

3.2.2 Authentication and Access Control:

Lightweight ciphers are used to establish secure authentication and access control mechanisms in EVs and ITS. They help verify the identity of vehicles, users, and infrastructure components, preventing unauthorized access to critical systems and data. Lightweight ciphers play a vital role in securing processes such as key exchange, digital signatures, and secure bootstrapping. Lightweight ciphers play a vital role in establishing secure authentication and access control mechanisms in EVs and ITS.

They are used for verifying the identity of vehicles, users, and infrastructure components during the establishment of secure communication channels. Lightweight ciphers ensure that only authorized entities can access and interact with the systems, mitigating the risk of unauthorized vehicle entry or malicious control.

3.2.3 **Secure Firmware Updates**:

Lightweight ciphers are employed to ensure the integrity and authenticity of firmware updates in EVs and ITS. These systems require regular software updates to enhance performance, fix vulnerabilities, and introduce new features. Lightweight ciphers provide encryption and digital signatures to verify the integrity and authenticity of firmware updates, preventing malicious modifications. EVs and ITS components often require firmware updates to enhance functionality, fix vulnerabilities, or introduce new features. Lightweight ciphers are used to secure these firmware updates by providing encryption and digital signatures. Encryption ensures that the firmware is transmitted securely, protecting it from interception and tampering. Digital signatures verify the integrity and authenticity of the firmware, ensuring that it has not been modified by unauthorized entities.

3.2.4 **Data Privacy and Confidentiality**:

Lightweight ciphers are essential for maintaining data privacy and confidentiality in EVs and ITS. These systems generate and process vast amounts of sensitive data, including location information, driver behavior, and vehicle diagnostics. Lightweight ciphers encrypt this data, preventing unauthorized access and preserving privacy. EVs and ITS generate and process a wide range of sensitive data, including location information, vehicle diagnostics, driver behavior, and personal identifiable information (PII). Lightweight ciphers are utilized to encrypt this data, preserving privacy and ensuring confidentiality. By encrypting the data at its source, sensitive information remains protected even if it is intercepted or accessed by unauthorized parties.

3.2.5 **Resource Efficiency**:

Resource-constrained devices, such as EVs and ITS components, often have limited processing power and energy resources. Lightweight ciphers are designed to be computationally efficient, minimizing the computational overhead and energy consumption while providing adequate security. This ensures that the implementation of encryption does not significantly impact the overall system performance. EVs and ITS components often have limited processing power, memory, and energy resources. Lightweight ciphers are designed to be computationally efficient, minimizing the impact on system performance while still providing adequate security.

These ciphers employ lightweight cryptographic algorithms and optimizations, such as reduced key sizes, streamlined operations, and optimized implementations, to ensure that encryption and decryption operations can be performed efficiently within the resource constraints of the systems.

3.2.6 **Standardization Efforts**: Standardization bodies, such as the Institute of Electrical and Electronics Engineers (IEEE), International Organization for Standardization (ISO), and the National Highway Traffic Safety Administration (NHTSA), actively participate in defining cryptographic standards and guidelines for secure implementations in EVs and ITS.

These standards often consider lightweight cryptographic algorithms to meet the specific resource constraints of these systems.

3.2.6.1 Standardization bodies, such as the Institute of Electrical and Electronics Engineers (IEEE), International Organization for Standardization (ISO), and the National Highway Traffic Safety Administration (NHTSA), actively contribute to the development of cryptographic standards and guidelines for secure implementations in EVs and ITS.

3.2.6.2 These standards often consider the specific resource constraints of EVs and ITS components, including lightweight cryptographic algorithms as viable options.

3.2.6.3 The standardization efforts ensure interoperability, compatibility, and the adoption of secure cryptographic practices in the domain, promoting trust and reliability in EVs and ITS systems.

3.3 **Conclusion:** By implementing lightweight ciphers in EVs and ITS, the domain benefits from enhanced security, privacy, and integrity of the systems and data. It enables secure communication, protects against unauthorized access and data tampering, and ensures efficient utilization of system resources.

Chapter 4: GIFT CIPHER and Optimization.

4.1 Introduction.

The design strategy and improvements made to the PRESENT cipher, resulted in an enhanced version called GIFT. GIFT is a simplified and efficient design that outperforms ciphers like SIMON and SKINNY, making it one of the most energy-efficient ciphers available. The design of GIFT addresses the weaknesses of parent PRESENT related to linear hulls.

The implementation of lightweight ciphers in electric vehicles (EVs) and intelligent transportation systems (ITS) is crucial due to the growing demand for secure communication and authentication in these domains. Previous works in this field have presented the need for lightweight cryptography in constrained devices and the limitations of existing cryptography standards like AES and SHA-2. Lightweight algorithms such as PRESENT, PHOTON, and SPONGENT have been included in ISO standards.

Comparing lightweight primitives is a complex task, considering the diverse range of use cases, from passive RFID tags to battery-powered devices and low-latency applications. The importance of area minimization and the throughput/area ratio as key criteria for lightweight encryption. The range of platforms to consider for research and development is wide, from tiny RFID tags to powerful ARM processors and high-end servers.

The analysis of the PRESENT cipher, showcases its developer's inspiration from SERPENT and specifically designed for lightweight hardware implementations. Noticeable weaknesses of PRESENT are related to the clustering of linear trails and the creation of powerful linear hulls. Since the publication of PRESENT, there has been significant advances in security analysis and primitive design including the NSA's proposed ciphers SIMON and SPECK, and the introduction of the tweakable block cipher SKINNY.

Although SIMON and SKINNY have advantages in terms of efficiency, PRESENT is still recognized as an elegant design. However, it has not benefited from recent advancements in the research community. There was a need to revisit the design of PRESENT, leveraging the advances in construction and cryptanalysis to push it to its limits, doing so has resulted in the improved GIFT cipher. The enhanced GIFT cipher due to its focused approach to handle challenges in implementation in EVs and ITS has been thus chosen as the Cipher to work upon further.

4.2 Improvements in GIFT Cipher.

PRESENT Cipher has a major drawback - the linear hulls. The security of PRESENT primarily relies on its Sbox, as the diffusion layer consists only of a bit permutation. While the PRESENT Sbox exhibits excellent cryptographic properties, it is relatively expensive. To address this, GIFT is designed so that the bit permutation in conjunction with the Difference Distribution Table (DDT)/Linear Approximation Table (LAT) of the Sbox, allows to remove the constraint of a branching number of 3. This innovative approach, which intricately combines the linear layer and the Sbox in a Substitution-Permutation-Network (SPN) cipher, is new in GIFT Cipher.

By eliminating the constraint, a more cost-effective Sbox is created for GIFT, which significantly reduces both the size and speed of the cipher compared to PRESENT. Table 1 and 2 illustrates the performance of GIFT vis-à-vis others.

	Area (GE)	Delay (ns)	Cycles	TP_{MAX} (MBit/s)	Power (μW) (@10MHz)	Energy (pJ)
GIFT-64-128	1345	1.83	29	1249.0	74.8	216.9
SKINNY-64-128	1477	1.84	37	966.2	80.3	297.0
PRESENT 64/128	1560	1.63	33	1227.0	71.1	234.6
SIMON 64/128	1458	1.83	45	794.8	72.7	327.3
GIFT-128-128	1997	1.85	41	1729.7	116.6	478.1
SKINNY-128-128	2104	1.85	41	1729.7	132.5	543.3
SIMON 128/128	2064	1.87	69	1006.6	105.6	728.6

Table 1: Performance of GIFT vis-à-vis others

Cipher	nb. of rds	gate cost (per bit per round) int. cipher	key sch.	total	nb. of op. w/o key sch.	nb. of op. w/ key sch.	round-based impl. area
GIFT -64-128	28	1 N 2 X		1 N 2 X	3×28 $= 84$	3×28 $=$ **84**	$1 + 2.67 \times 2$ $=$ **6.34**
SKINNY -64-128	36	1 N 2.25 X	0.625 X	1 N 2.875 X	3.25×36 $= 117$	3.875×36 $=$ **139.5**	$1 + 2.67 \times 2.875$ $=$ **8.68**
SIMON -64/128	44	0.5 A 1.5 X	1.5 X	0.5 A 3.0 X	2×44 $= 88$	3.5×44 $=$ **154**	$0.67 + 2.67 \times 3$ $=$ **8.68**
PRESENT -128	31	1 A 3.75 X	0.125 A 0.344 X	1.125 A 4.094 X	4.75×31 $= 147.2$	5.22×31 $=$ **161.8**	$1.5 + 2.67 \times 4.094$ $=$ **12.43**
GIFT -128-128	40	1 N 2 X		1 N 2 X	3.0×40 $= 120$	3.0×40 $=$ **120**	$1 + 2.67 \times 2$ $=$ **6.34**
SKINNY -128-128	40	1 N 2.25 X		1 N 2.25 X	3.25×40 $= 130$	3.25×40 $=$ **130**	$1 + 2.67 \times 2.25$ $=$ **7.01**
SIMON -128/128	68	0.5 A 1.5 X	1 X	0.5 A 2.5 X	2×68 $= 136$	3×68 $=$ **204**	$0.67 + 2.67 \times 2.5$ $=$ **7.34**
AES -128	10	4.25 A 16 X	1.06 A 3.5 X	5.31 A 19.5 X	20.25×10 $= 202.5$	24.81×10 $=$ **248.1**	$7.06 + 2.67 \times 19.5$ $=$ **59.12**

Table 2: Performance of GIFT vis-à-vis others

In terms of security, GIFT provides a robust bound against simple differential and linear attacks. A comprehensive analysis using state-of-the-art cryptanalysis techniques has proven GIFT's improvements.

GIFT is available in two versions:

- GIFT-64 - 128-bit key with a 64-bit block size
- GIFT-128 - 128-bit key with a 128-bit block size.

The sole distinction between these versions lies in the bit permutation, which is modified in GIFT-128 to accommodate twice as many state bits compared to GIFT-64.

4.3 **Specifications**

GIFT-64-128 is a 28-round SPN cipher whereas GIFT-128-128 is a 40-round SPN cipher. GIFT can be represented in three different formats. However, conventional 1D representation is adopted, similar to that of PRESENT, where the bits are arranged in a row.

4.4 **Round function**: The round function of GIFT consists of three stages: SubCells, PermBits, and AddRoundKey, which can be metaphorically compared to the process of wrapping a ***gift***. Firstly, the SubCells step places the content inside a box. Then, the PermBits step represents wrapping a ribbon around the box. Lastly, the AddRoundKey step ties a knot to secure the content. The details have been described for GIFT-64, and the implementation has been completed for GIFT-128.

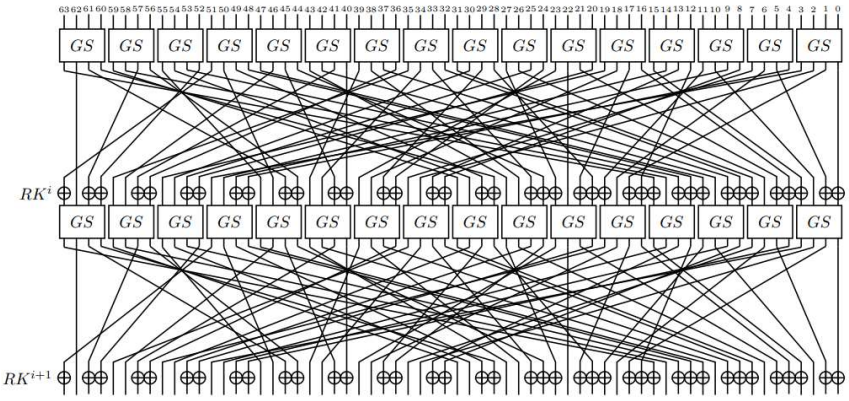

Fig 1: Illustrates 2 Rounds of GIFT -64

Initialization. The cipher takes in a n-bit plaintext $b_{n-1}\, b_{n-2}... b_0$ as the cipher state S, where n = 64, 128 and b_0 being the least significant bit. The cipher state can also be expressed as **s** many 4-bit nibbles.

$S = w_{s-1}||w_{s-2}||...||w_0$, where s = 16, 32. The cipher also receives a 128-bit key $K = k_7||k_6||...||k_0$ as the key state, where k_i is a 16-bit word.

SubCells. Both versions of GIFT use the same invertible 4-bit Sbox, GS. The Sbox is applied to every nibble of the cipher state.

$w_i \leftarrow GS(w_i), \forall i \in \{0, ..., s-1\}$.

The action of this Sbox in hexadecimal notation is given in Table 3

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$GS(x)$	1	a	4	c	6	f	3	9	2	d	b	7	5	0	8	e

Table 3: Specifications of GIFT Sbox GS

PermBits. The bit permutation used in GIFT-64 and GIFT-128 are given in Table 4 and 5 respectively. It maps bits from bit position *i* of the cipher state to bit position P(*i*).

$$b_{P(i)} \leftarrow b_i, \ \forall i \in \{0, ..., n-1\}.$$

The permutations can also be expressed as:

$$P_{64}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 16\left(\left(3\left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4)\right) \bmod 4\right) + (i \bmod 4),$$

$$P_{128}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 32\left(\left(3\left\lfloor \frac{i \bmod 16}{4} \right\rfloor + (i \bmod 4)\right) \bmod 4\right) + (i \bmod 4).$$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P_{64}(i)$	0	17	34	51	48	1	18	35	32	49	2	19	16	33	50	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P_{64}(i)$	4	21	38	55	52	5	22	39	36	53	6	23	20	37	54	7
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P_{64}(i)$	8	25	42	59	56	9	26	43	40	57	10	27	24	41	58	11
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P_{64}(i)$	12	29	46	63	60	13	30	47	44	61	14	31	28	45	62	15

Table 4: Bit permutation used in GIFT-64

The action of this Sbox in hexadecimal notation is given in Table

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P_{128}(i)$	0	33	66	99	96	1	34	67	64	97	2	35	32	65	98	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P_{128}(i)$	4	37	70	103	100	5	38	71	68	101	6	39	36	69	102	7
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P_{128}(i)$	8	41	74	107	104	9	42	75	72	105	10	43	40	73	106	11
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P_{128}(i)$	12	45	78	111	108	13	46	79	76	109	14	47	44	77	110	15
i	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
$P_{128}(i)$	16	49	82	115	112	17	50	83	80	113	18	51	48	81	114	19
i	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
$P_{128}(i)$	20	53	86	119	116	21	54	87	84	117	22	55	52	85	118	23
i	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
$P_{128}(i)$	24	57	90	123	120	25	58	91	88	121	26	59	56	89	122	27
i	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
$P_{128}(i)$	28	61	94	127	124	29	62	95	92	125	30	63	60	93	126	31

Table 5: Bit permutation used in GIFT-128

AddRoundKey. This step consists of adding the round key and round constants. An n/2-bit round key RK is extracted from the key state, it is further partitioned into 2 s-bit words RK = U||V = $u_{s-1}...u_0||v_{s-1}...v_0$, where s = 16, 32 for GIFT-64 and GIFT-128 respectively.

For GIFT-64, U and V are XORed to $\{b_{4i+1}\}$ and $\{b_{4i}\}$ of the cipher state respectively.

$$b_{4i+1} \leftarrow b_{4i+1} \oplus u_i \, , \, b_{4i} \leftarrow b_{4i} \oplus v_i \, , \, \forall i \in \{0, ..., 15\}.$$

For GIFT-128, U and V are XORed to $\{b_{4i+2}\}$ and $\{b_{4i+1}\}$ of the cipher state respectively.

$$b_{4i+2} \leftarrow b_{4i+2} \oplus u_i \, , \, b_{4i+1} \leftarrow b_{4i+1} \oplus v_i \, , \, \forall i \in \{0, ..., 31\}.$$

For both versions of GIFT, a single bit "1" and a 6-bit round constant $C = c_5c_4c_3c_2c_1c_0$ are XORed into the cipher state at bit position n − 1, 23, 19, 15, 11, 7 and 3 respectively

$$b_{n-1} \leftarrow b_{n-1} \oplus 1,$$

$$b_{23} \leftarrow b_{23} \oplus c_5, b_{19} \leftarrow b_{19} \oplus c_4, b_{15} \leftarrow b_{15} \oplus c_3,$$

$$b_{11} \leftarrow b_{11} \oplus c_2, b_7 \leftarrow b_7 \oplus c_1, b_3 \leftarrow b_3 \oplus c_0.$$

Key schedule and round constants. The key schedule and round constants are the same for both versions of GIFT, the only difference is the round key extraction. A round key is first extracted from the key state before the key state update.

For GIFT-64, two 16-bit words of the key state are extracted as the round key RK = U||V.

$$U \leftarrow k_1, V \leftarrow k_0.$$

For GIFT-128, four 16-bit words of the key state are extracted as the round key RK = U||V

$$U \leftarrow k_5||k_4, V \leftarrow k_1||k_0.$$

The key state is then updated as follows,

$$k_7||k_6||...||k_1||k_0 \leftarrow k_1 \gg 2||k_0 \gg 12||...||k_3||k_2,$$

where $\gg i$ is an i bits right rotation within a 16-bit word.

The round constants are generated using the same 6-bit affine LFSR as SKINNY, whose state is denoted as $(c_5, c_4, c_3, c_2, c_1, c_0)$. Its update function is defined as:

$$(c_5, c_4, c_3, c_2, c_1, c_0) \leftarrow (c_4, c_3, c_2, c_1, c_0, c_5 \oplus c_4 \oplus 1).$$

The six bits are initialized to zero, and updated before being used in a given round. The values of the constants for each round are given in the table below, encoded to byte values for each round, with c_0 being the least significant bit.

Rounds	Constants
1 - 16	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E
17 - 32	1D,3A,35,2B,16,2C,18,30,21,02,05,0B,17,2E,1C,38
33 - 48	31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04

4.5 MODIFICATION OF GIFT CIPHER

4.5.1 **Introduction**: Cryptographic algorithms play a vital role in ensuring the security of transmitted and received data between IoT devices. However, many IoT devices possess limited computing power, memory, and resources, making it challenging to implement cryptographic algorithms on such devices. Lightweight cryptography algorithms are designed to efficiently utilize available resources, enabling effective operation on devices with restricted capabilities. Design of quantum gates for GIFT symmetric key cryptography have been devised and implemented. One of the most important factors when evaluating quantum circuits is optimizing the number of qubits. When designing quantum circuits, new qubits are allocated to temporal storage or new values.

However, by using an on-the-fly approach to recycle the initially allocated qubits until the encryption is finished an efficient Sbox quantum is implementation.
The GIFT implementation have been evaluated using the IBM ProjectQ framework, a quantum computer emulator (https://github.com/ProjectQFramework/ProjectQ, accessed on 10 May 2021). IBM ProjectQ uses a variety of quantum compilers that allow developers to simulate quantum computers or draw quantum circuits. Among them, the resource counter compiler, which is a quantum resource estimator, measures quantum resources by analyzing qubits, quantum gates, and circuit depth. Compared with quantum implementation results of other block ciphers GIFT quantum circuits' results were commendable.

4.5.2 **GIFT Block Cipher**

The GIFT block cipher is a symmetric key cryptography using the Substitution Permutation Network (SPN) method. In the GIFT block cipher, each round performs four steps: Sbox, Permutation, AddRoundKey and Constant XOR. The encryption operation of GIFT block cipher is described in Figure 2.

4.5.2.1 **Sbox of GIFT Block Cipher**

The n-bit block (n = 64, 128) is split into 4 bits and becomes the input value of the 4-bit Sbox.

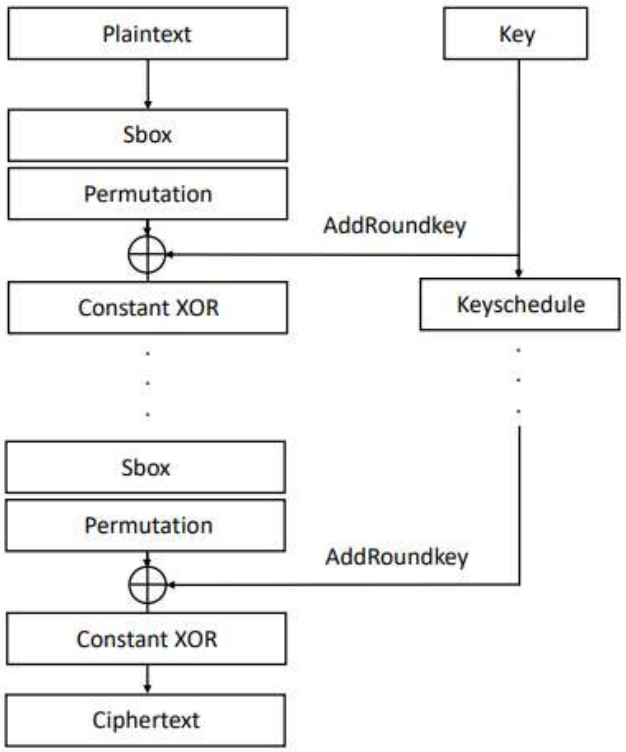

Fig 2: Encryption process of GIFT block cipher

4.5.2.2 **Permutation of GIFT Block Cipher**

In the permutation, GIFT-64/128 replaces the P64(i)-th bit of block B with the i-th bit of block B. Details on the permutation of GIFT-64/128 are shown in Table 6. In this paper, detailed Table on permutation of GIFT-128/128 is omitted. Permutation Table of GIFT-128/128 is at https://link.springer.com/chapter/10.1007/978-3-319-66787-4_16/tables/5

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P_{64}(i)$	0	17	34	51	48	1	18	35	23	49	2	19	16	33	50	3
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P_{64}(i)$	4	21	38	55	52	5	22	39	36	53	6	23	20	37	54	7
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P_{64}(i)$	8	25	42	59	56	9	26	43	40	57	10	27	24	41	58	11
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P_{64}(i)$	12	29	46	63	60	13	30	47	44	61	14	31	28	45	62	15

Table 6: Permutation of GIFT-64 bit

4.5.2.3 **AddRoundkey of GIFT Block Cipher**

In the GIFT-64/128 block cipher, k_0 and k_1 (32-bit total) are selected from the key ($K = k_7, ..., k_0$). k_0 and k_1 are used as U and V of the round key as follows, $RK = U||V = u_{15}...u_0||v_{15}...v_0$ ($U = k_1$, $V = k_0$). The round key is exclusive-ORed with the block B, where U is XORed to b_{4i+1} and V is XORed to b_{4i}.

$$b_{4i+1} \leftarrow b_{4i+1} \oplus u_i\,,\ b_{4i} \leftarrow b_{4i} \oplus v_i\,,\ i = 0, ..., 15$$

In the GIFT-128/128 block cipher, k_0, k_1, k_4, and k_5 (64-bit in a total) are selected from the key K. k_0, k_1, k_4 and k_5 are used as U and V of the round key as follows, $RK = U||V = u_{31}...u_0||v_{31}...v_0$ ($U = k_5||k_4$, $V = k_1||k_0$). The round key is XORed to the block B, where U is XORed to b_{4i+2} and V is XORed to b_{4i+1}

$$.\ b_{4i+2} \leftarrow b_{4i+2} \oplus u_i\,,\ b_{4i+1} \leftarrow b_{4i+1} \oplus v_i\,,\ i = 0, ..., 31$$

4.5.2.4 **Constant XOR of GIFT Block Cipher**

Round constants C given in Table 7 are used in GIFT-64/128 and GIFT-128/128 block ciphers. Single bit and round constants ($C = c5c4c3c2c1c0$) are XORed to block B as in Equations below.

$b_n{-}1 \leftarrow - b_n{-}1 \oplus 1$,

$b_{23} \leftarrow - b_{23} \oplus c_5$, $b_{19} \leftarrow - b_{19} \oplus c_4$, $b_{15} \leftarrow - b_{15} \oplus c_3$,

$b_{11} \leftarrow - b_{11} \oplus c_2$, $b_7 \leftarrow - b_7 \oplus c_1$, $b_3 \leftarrow - b_3 \oplus c_0$.

Rounds	Constants C															
1 to 16	01	03	07	0F	1F	3E	3D	3B	37	2F	1E	3C	39	33	27	0E
17 to 32	1D	3A	35	2B	16	2C	18	30	21	02	05	0B	17	2E	1C	38
33 to 48	31	23	06	0D	1B	36	2D	1A	34	29	12	24	08	11	22	04

Table 7: Round Constants

4.5.2.5. **Keyschedule of GIFT Block Cipher**

In GIFT-64/128 and GIFT-128/128 block ciphers, the Keyschedule updates key ($K = k_7, ..., k_0$) and extracts the round key from the updated key K. The Keyschedule is shown in Equation above. The notation ($\gg$ i) denotes a right rotation operation (i-bit).

4.5.3. **Quantum Gates and Algorithm**

4.5.3.1 **Quantum Gates**

To perform the work of classical gates, quantum gates should be implemented. CNOT gate and Toffoli gate are the most commonly used in quantum circuits. The CNOT gate receives two qubits and XOR the first qubit to the second qubit (i.e., CNOT (a, b) $\longrightarrow$ a = a, b = a $\oplus$ b). This gate stores the XOR result of the two input qubits in the second qubit. The quantum circuit of CNOT gate is shown in Figure 3 (left). The Toffoli gate is more expensive and complex than the CNOT gate. Three qubits are input to the Toffoli gate, and the AND result of the first and second qubits is XORed to the third qubit (i.e., Toffoli (a, b, c) $\longrightarrow$ a = a, b = b, c = c $\oplus$ (a $\cdot$ b). The quantum circuit of Toffoli gate is shown in Figure 3 (right). The notation ($\cdot$) indicates AND operation

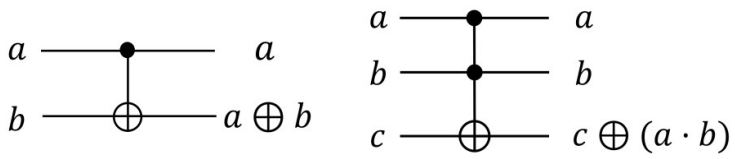

Fig 3: CNOT gate (left) and Toffoli gate (right)

The logical-OR quantum gate is composed of a combination of of Toffoli gate and X gate, as shown in Figure 4. The X gate operates on a single qubit and performs a NOT operation. In the logical OR quantum gate, a and b (input qubits) are changed (0 to 1, 1 to 0) by the X gate (i.e., Quantum OR (a, b, c) $\longrightarrow$ a = $\sim$ a, b = $\sim$ b, c = c $\oplus$ (a $\vee$ b). To return to the original a or b, reversible gate must be performed by executing the X gate once more. The OR quantum gate (reversible) is shown in Figure 4. The notation $\vee$ represents logical-OR operation

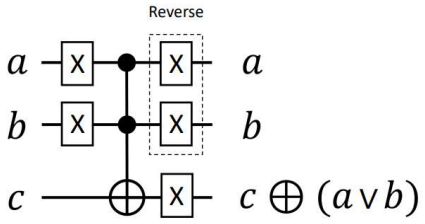

Fig 4: Logical-OR quantum gate

4.5.4. **Quantum Circuit for GIFT Block Cipher**

In the presented GIFT−n/128 quantum circuit, only (n+128)-qubits are allocated respectively to assign plaintext (n-bit) and key (128-bit). Therefore, it is optimized without additional qubits. All operations, including AddRoundkey, Sbox, Permutation, Constant XOR, and Keyschedule, were optimized in terms of quantum resources

4.5.4.1 **Sbox of GIFT Block Cipher**

The creator of the GIFT block cipher introduced two variations of the Sbox. One version is specifically designed for efficient software implementation, while the other version is tailored for optimal hardware implementation. Detailed processes are given in Algorithms 1 and 2 for software-oriented and hardware-oriented, respectively. Comparing these two Sboxes, the hardware-friendly Sbox offers more advantages over the software-friendly Sbox when it comes to quantum circuits.

Algorithm 1: Software-oriented implementation of GIFT Sbox

Input: 4-bit input $x(x_3, x_2, x_1, x_0)$ (before entering Sbox).

Output: 4-bit output $x(x_3, x_2, x_1, x_0)$ (after performing Sbox)

1: $x_1 \leftarrow x_1$ XOR (x_0 AND x_2)

2: $t \leftarrow x_0$ XOR (x_1 AND x_3)

3: $x_2 \leftarrow x_2$ XOR (t OR x_1)

4: $x_0 \leftarrow x_3$ XOR x_2

5: $x_1 \leftarrow x_1$ XOR x_3

6: $x_0 \leftarrow$ NOT x_0

7: $x_2 \leftarrow x_2$ XOR (t AND x1)

8: $x_3 \leftarrow$ t

9: return $x(x_3, x_2, x_1, x_0)$

In the software-friendly Sbox operation (see Algorithm 1), the input and output of operations are different (e.g., $x_0 = x_3$ XOR x2). Qubits in quantum computers must be initialized to zero to overwrite the new value. In order to initialize a qubit to zero, the same value must exist in another qubit. The Algorithm 1 is designed as a quantum circuit. Since the new value cannot be overwritten (e.g., $x_0 \leftarrow x_3$ XOR x_2, we have to allocate a new qubit and also allocate an additional qubit for temporary storage. On the other hand, we can see that in the hardware-oriented Sbox design of Algorithm 2, the input and output of the operation are always the same.

Algorithm 2: Hardware-oriented implementation of GIFT Sbox

Input: 4-bit input $x(x_3, x_2, x_1, x_0)$ (before entering Sbox).

Output: 4-bit output $x(x_3, x_2, x_1, x_0)$ (after performing Sbox).

1: $x_1 \leftarrow x_1$ XNOR (x_0 NAND x_2)

2: $x_0 \leftarrow x_0$ XNOR (x_1 NAND x_3)

3: $x_2 \leftarrow x_2$ XNOR (x_0 NOR x_1)

4: $x_3 \leftarrow x_3$ XNOR x_2

5: $x_1 \leftarrow x_1$ XNOR x_3

6: $x_2 \leftarrow x_2$ XNOR (x_0 NAND x_1)

7: return $x(x_0, x_2, x_1, x_3)$

4.5.4.2 **Modification to existing SBox**:

Therefore, we were able to optimize the quantum circuit by choosing a hardware friendly Sbox. The resulting value can be stored in qubits that are entered into the operation. For example, the quantum circuit for line 4 of Algorithm 2 corresponds to lines 9 and 10 of Algorithm 1. The operation continues on x_3 without allocating additional qubits. Therefore, no additional qubits are used and an optimized 4-qubit Sbox quantum circuit can be implemented. The implementation of GIFT Sbox quantum circuit is described in Algorithm 3.

The NOT operation is performed twice on lines 1, 2, 3 and 6 of the Algorithm 2. Two NOT operations cancel each other. The arrangement of input and output qubits is altered in Algorithm 3. It can be performed with one Swap gate on x_0, x_3. As mentioned earlier, Swap gates are not considered quantum resources. Quantum circuit for Sbox of GIFT block cipher is described in Figure 5.

Algorithm 3: Quantum circuits for Sbox of GIFT block cipher

Input: 4-qubit input $x(x_3, x_2, x_1, x_0)$ (before entering Sbox).

Output: 4-qubit output $x(x_3, x_2, x_1, x_0)$ (after performing Sbox).

1: $x_1 \longleftarrow$ Toffoli (x_0, x_2, x_1)

2: $x_0 \longleftarrow$ Toffoli (x_1, x_3, x_0)

3: $x_0 \longleftarrow$ X (x_0)

4: $x_1 \longleftarrow$ X (x_1)

5: $x_2 \longleftarrow$ Toffoli (x_0, x_1, x_2)

6: $x_2 \longleftarrow$ X (x_2)

7: $x_0 \longleftarrow$ X (x_0) (reverse)

8: $x_1 \longleftarrow$ X (x_1) (reverse)

9: $x_3 \longleftarrow$ CNOT (x_2, x_3)

10: $x_3 \longleftarrow$ X (x_3)

11: $x_1 \longleftarrow$ CNOT (x_3, x_1)

12: $x_1 \leftarrow - X(x_1)$
13: $x_2 \leftarrow -$ Toffoli (x_0, x_1, x_2)
14: return $x(x_0, x_2, x_1, x_3$

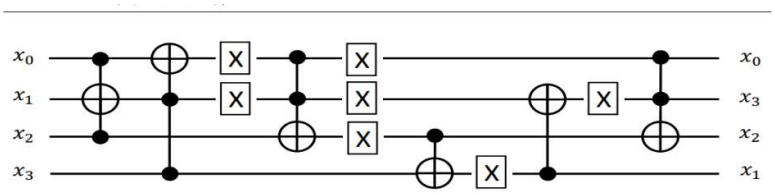

Figure 5. Quantum circuit for Sbox of GIFT block cipher.

4.5.4.3 **Permutation of GIFT Block Cipher**

After the Sbox operation, the permutation of Table 4 is performed. Similar to the permutation of PRESENT block cipher, bit position changes can be made using Swap gates and are not counted as quantum resources. Therefore, by relabeling the qubits, the permutation of GIFT block cipher can be done without quantum resources.

4.5.4.3 **AddRoundkey of GIFT Block Cipher**

In the AddRoundkey operation, the round key RK (n/2-bit) is XORed to the block B (n/2-bit). The XOR operation can be done with the CNOT gate. At this time, the result of qubit should be B. The AddRoundkey of GIFT-64/128 and GIFT-128/128 block ciphers are similar, only the number of bits is different. In the GIFT-128/128 block cipher, double the CNOT gates are used compared to the GIFT-64/128 block cipher. Quantum circuits for Addroundkey of GIFT-64/128 and GIFT-128/128 block ciphers are shown in Algorithm 4 and Algorithm 5, respectively.

Algorithm 4: Quantum circuit for AddRoundkey of GIFT-64/128 block cipher

Input: 64-qubit block $B(b_{63}, ..., b_0)$, 32-qubit round key $RK(rk_{31}, ..., rk_0)$.

Output: 64-qubit block $B(b_{63}, ..., b_0)$ after AddRoundKey.

1: for i = 0 to 15 do

2: $b_{4i} \leftarrow -$ CNOT (rk_i , b_{4i})

3: $b_{4i+1} \leftarrow -$ CNOT (r_{ki+16}, b_{4i+1})

4: end for

5: return $B(b_{63}, ..., b_0)$

Algorithm 5: Quantum circuits for AddRoundkey of GIFT-128/128 block cipher

Input: 128-qubit block B(b_{127}, ..., b_0), 64-qubit round key RK(rk_{63}, ...,rk_0).

Output: 128-qubit block B(b_{127}, ..., b_0) after AddRoundKey. 1: for i = 0 to 31 do

2: $b_{4i+1} \leftarrow -$ CNOT (r_{ki} , b_{4i+1})

3: $b_{4i+2} \leftarrow -$ CNOT (r_{ki+32}, b_{4i+2})

4: end for

5: return B(b_{127}, ..., b_0)

4.5.4.4 **Constant XOR of GIFT Block Cipher**

The round constant C in Table 5 and the single bit are XORed to block B. Since the constant C for each round is already set, we performed X gates on b_{23}, b_{19}, b_{15}, b_{11}, b_7, and b_3 only for positions where bit of C is one. When the round constant C is 3 in round 2, c_0 and c_1 are $_1$. Therefore, the X gate (b_3) and X gate (b_7) are performed. For a single bit, an X gate is always performed on b_{n-1}. In this way, no qubits are used for Constant XOR. Moreover, CNOT gates are not used; only X gates are used. The implementation of the GIFT-n/128 constant XOR quantum circuit is described in Algorithm 9

Algorithm 6: Quantum circuits for constant XOR of GIFT-n/128 block cipher

Input: b_{n-1}, b_{23}, b_{19}, b_{15}, b_{11}, b_7, b_3 of n-bit block B.

Output: b_{n-1}, b_{23}, b_{19}, b_{15}, b_{11}, b_7, b_3 of n-bit block B after Constant XOR.

1: b_{23}, b_{19}, b_{15}, b_{11}, b_7, $b_3 \leftarrow -$X (b_{23}, b_{19}, b_{15}, b_{11}, b_7, b_3) according to round constant C (c_5, c_4, c_3, c_2, c_1, c_0)

2: $b_{n-1} \leftarrow -$ X (b_{n-1})

3: return b_{n-1}, b_{23}, b_{19}, b_{15}, b_{11}, b_7, b_3

4.5.4.5 **Keyschedule of GIFT Block Cipher**

In GIFT, the state of the key is updated after the round key is used, which is shown in Equation. The Keyschedule of PRESENT block cipher uses Sbox and round i, while the GIFT only changes the bit positions of key K. It can be done using only Swap gates, and using the method of relabeling qubits does not require quantum resources.

Chapter 5 : Implementation of GIFT CIPHER in Python

5.1 Encryption

The File upon running asks for a Message to be encrypted.

Ask For a message to be encrypted in string format and convert it in Binary format. Calculate number of blocks require to encrypt complete massage.

```
#GIFT 128 bit Encryption and Decryption:

comp_in_text = input('Enter text message = ')
c_num  = 0
for ele in comp_in_text:
    c_num = c_num << 8
    v = ord(ele)
    c_num = c_num + v


##Input_text  = int(input("Enter Input text in hexadecimal : "), 16)
Input_text = c_num;
In_length = len(hex(Input_text)) - 2;
print(In_length)
lp_req = int(In_length / 32) + 1
print(lp_req)
```

Defining function for removing extra symbols like 0X

```
import binascii
def hex_to_ascii(hex_str):
    hex_str = hex_str. replace(' ', ''). replace('0x', ''). replace('\t', ''). replace('\n', '')
    ascii_str = binascii. unhexlify(hex_str)
    return ascii_str
```

Define default 128 bit encryption Key. Option to set new key given.

```
Enc_Key     = 45897555121545541 #int(input("Enter Encrytion key in hexadecimal : "), 16)       # key 32*4 = 128 bit
```

Array for permutation in Encryption and Decryption

```
GIFT_P = [0, 33, 66, 99, 96,  1, 34, 67, 64, 97,  2, 35, 32, 65, 98,  3,
          4, 37, 70,103,100,  5, 38, 71, 68,101,  6, 39, 36, 69,102,  7,
          8, 41, 74,107,104,  9, 42, 75, 72,105, 10, 43, 40, 73,106, 11,
         12, 45, 78,111,108, 13, 46, 79, 76,109, 14, 47, 44, 77,110, 15,
         16, 49, 82,115,112, 17, 50, 83, 80,113, 18, 51, 48, 81,114, 19,
         20, 53, 86,119,116, 21, 54, 87, 84,117, 22, 55, 52, 85,118, 23,
         24, 57, 90,123,120, 25, 58, 91, 88,121, 26, 59, 56, 89,122, 27,
         28, 61, 94,127,124, 29, 62, 95, 92,125, 30, 63, 60, 93,126, 31 ]


GIFT_P_inv = [   0,  5, 10, 15, 16, 21, 26, 31, 32, 37, 42, 47, 48, 53, 58, 63,
 64, 69, 74, 79, 80, 85, 90, 95, 96,101,106,111,112,117,122,127,
 12,  1,  6, 11, 28, 17, 22, 27, 44, 33, 38, 43, 60, 49, 54, 59,
 76, 65, 70, 75, 92, 81, 86, 91,108, 97,102,107,124,113,118,123,
  8, 13,  2,  7, 24, 29, 18, 23, 40, 45, 34, 39, 56, 61, 50, 55,
 72, 77, 66, 71, 88, 93, 82, 87,104,109, 98,103,120,125,114,119,
  4,  9, 14,  3, 20, 25, 30, 19, 36, 41, 46, 35, 52, 57, 62, 51,
 68, 73, 78, 67, 84, 89, 94, 83,100,105,110, 99,116,121,126,115 ]
```

Array for Round Constant

```
GIFT_RC = [ 0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F,
            0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B,
            0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E,
            0x1C, 0x38, 0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A,
            0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04, 0x09, 0x13,
            0x26, 0x0c, 0x19, 0x32, 0x25, 0x0a, 0x15, 0x2a, 0x14, 0x28,
            0x10, 0x20 ]     # 62 items
```

Encryption function definition takes input text, Key and No of rounds as input and returns cipher text as output.

```
## --------------------- Encryption function 128 bit -------

def GIFT_enc(text,masterkey,n_round):
##    print('In Encryption function')
##    print('Input Text = ',hex(text))
##    print('masterkey = ',hex(masterkey))
    rounds = n_round;   #  40
    key    = masterkey
```

SBox and Permutation Implementation.

```
# sub cell
    for r in range(rounds):
        inpu = 0;
        for i in range(31,-1,-1):
            bit = (temp_text >> 4*i) & 0XF;
            inp = GIFT_S[bit];
            inpu = inpu * 16 + inp;
##        print('After S_box = ',hex(inpu))

    # Permute the bits
        perm_val = 0;
        for i in range(128):
            bit = (inpu >> i) & 0X1;
            multi = GIFT_P[i]
            perm_val = perm_val + bit*2**multi
```

Adding of the Round Key**.**

```
##        print('perm val =' ,hex(perm_val))

       bits = perm_val;
   # Add Round Key
       kbc = 0   #  (kbc -> key bit counter)
       r_val = 0;
       for i in range(32):
           k_v1 = (key >> kbc) & 0X1;
           #print(k_v1)
           k_v2 = (key >> (kbc + 64)) & 0X1;
           #print(k_v2)
           if(k_v1==1):
               b1 = (bits >> 4*i + 1) & 0X1
               #bits = bits - b1*2**(4*i+1) + (1-b1)*2**(4*i+1)
               bits = bits + (1-2*b1)*2**(4*i+1)
           if(k_v2 ==1):
               b2 = (bits >> 4*i + 2) & 0X1
               #bits = bits - b1*2**(4*i+1) + (1-b1)*2**(4*i+1)
               bits = bits + (1-2*b2)*2**(4*i+2)
           kbc = kbc + 1
```

Adding the Round Constant.

```
  # add constant
      k3 = GIFT_RC[r]&0x1;
      if(k3==1):
          b3 = (bits >> 3) & 0x1
          bits = bits + (1-2*b3)*2**(3)

      k7 = (GIFT_RC[r]>>1)&0x1;
      if(k7==1):
          b7 = (bits >> 7) & 0x1
          bits = bits + (1-2*b7)*2**(7)
```

```
        k11 = (GIFT_RC[r]>>2)&0x1;
        if(k11==1):
            b11 = (bits >> 11) & 0x1
            bits = bits + (1-2*b11)*2**(11)

        k15 = (GIFT_RC[r]>>3)&0x1;
        if(k15==1):
            b15 = (bits >> 15) & 0x1
            bits = bits + (1-2*b15)*2**(15)

        k19 = (GIFT_RC[r]>>4)&0x1;
        if(k19==1):
            b19 = (bits >> 19) & 0x1
            bits = bits + (1-2*b19)*2**(19)

        k23 = (GIFT_RC[r]>>5)&0x1;
        if(k23==1):
            b23 = (bits >> 23) & 0x1

    b127 = (bits >> 127) & 0x1
    bits = bits + (1-2*b127)*2**(127)

    temp_text = bits
    # print('Add round key = ', hex(bits))
```

Updating the Key.

```
    # Key update

        last_key_32 = key & 0xFFFFFFFF
        temp_key1 = key >> 32;
        temp_key2 = last_key_32 << 96
        temp_key = temp_key2 + temp_key1;
     #  print('temp kay = ',hex(temp_key))

        key= ((key >> 96) << 96) + (temp_key & 0xffffffffffffffffffffffff);   # 96 bit update

        temp_24 = (temp_key >> 96) & 0xF;
        temp_25 = (temp_key >> 100) & 0xF;
        temp_26 = (temp_key >> 104) & 0xF;
        temp_27 = (temp_key >> 108) & 0xF;

        temp_28 = (temp_key >> 112) & 0xF;
        temp_29 = (temp_key >> 116) & 0xF;
        temp_30 = (temp_key >> 120) & 0xF;
        temp_31 = (temp_key >> 124) & 0xF;

        key_27_24 = (temp_26 << 12) + (temp_25 << 8) + (temp_24 << 4) + temp_27

        k_28 = ((temp_28 &0xc)>>2) ^ ((temp_29 &0x3)<<2);
        k_29 = ((temp_29 &0xc)>>2) ^ ((temp_30 &0x3)<<2);
        k_30 = ((temp_30&0xc)>>2) ^ ((temp_31  &0x3)<<2);
        k_31 = ((temp_31 &0xc)>>2) ^ ((temp_28 &0x3)<<2);

        key_31_28 = (k_31 << 12) +  (k_30 << 8) +(k_29 << 4) + k_28;

        key =(key_31_28 << 112) +(key_27_24 <<96) +(key & 0xffffffffffffffffffffffff)

     #   print('updated Key after', r , ' = ', hex(key))
     #   print('---------------------------------- \n ')
    return(temp_text)
```

5.2 Decryption

Defining Decryption function which takes cipher text, Encryption key and Number of Rounds as input and gives recovered text massage as output.

First Generate key for all rounds.

```
# ----------------- Decryption function 128 Bit -------------------------

def GIFT_Decryp(C_text,masterkey,n_round):
    rounds = n_round;   #  40
    key    = masterkey
   # temp_text = C_text

##    round_key_state[rounds][n_bit]
##    temp_key[128]

    round_key_state = list();
    round_key_state.insert(0,key)
 # Key Schedules
    for r in range(rounds):
        last_key_32 = key & 0xFFFFFFFF
        temp_key1 = key >> 32;
        temp_key2 = last_key_32 << 96
        temp_key = temp_key2 + temp_key1;
        #print('temp kay = ',hex(temp_key))

        key= ((key >> 96) << 96) + (temp_key & 0xffffffffffffffffffffffff);

        temp_24 = (temp_key >> 96) & 0xF;
        temp_25 = (temp_key >> 100) & 0xF;
        temp_26 = (temp_key >> 104) & 0xF;
        temp_27 = (temp_key >> 108) & 0xF;

  temp_29 = (temp_key >> 116) & 0xF;
  temp_30 = (temp_key >> 120) & 0xF;
  temp_31 = (temp_key >> 124) & 0xF;

  key_27_24 = (temp_26 << 12) + (temp_25 << 8) + (temp_24 << 4) + temp_27

  k_28 = ((temp_28 &0xc)>>2) ^ ((temp_29 &0x3)<<2);
  k_29 = ((temp_29 &0xc)>>2) ^ ((temp_30 &0x3)<<2);
  k_30 = ((temp_30&0xc)>>2) ^ ((temp_31  &0x3)<<2);
  k_31 = ((temp_31 &0xc)>>2) ^ ((temp_28 &0x3)<<2);

  key_31_28 = (k_31 << 12) +  (k_30 << 8) +(k_29 << 4) + k_28;

  key =(key_31_28 << 112) +(key_27_24 <<96) +(key & 0xffffffffffffffffffffffff)

#  print('updated Key after', r , ' = ', hex(key))
  round_key_state.insert(r+1,key)
```

Use Same or Different Key for Decryption.

```
  inpu = C_text
  for r in range(rounds-1,-1,-1):
      #print('-------- Decryption Round ', r , 'Starting ----------')
      bits = inpu
     # print('bits = ',hex(bits))
      key = round_key_state[r]
    #  print('Key used = ',hex(key))
```

```
# Add Round Key
    kbc = 0   #  (kbc -> key bit counter)
    r_val = 0;
    for i in range(32):
        k_v1 = (key >> kbc) & 0X1;
        #print(k_v1)
        k_v2 = (key >> (kbc + 64)) & 0X1;
        #print(k_v2)
        if(k_v1==1):
            b1 = (bits >> 4*i + 1) & 0X1
            #bits = bits - b1*2**(4*i+1) + (1-b1)*2**(4*i+1)
            bits = bits + (1-2*b1)*2**(4*i+1)
        if(k_v2 ==1):
            b2 = (bits >> 4*i + 2) & 0X1
            #bits = bits - b1*2**(4*i+1) + (1-b1)*2**(4*i+1)
            bits = bits + (1-2*b2)*2**(4*i+2)
        kbc = kbc + 1
   # print('Bits after Xoring =', hex(bits))

# add constant
    k3 = GIFT_RC[r]&0x1;
    if(k3==1):
        b3 = (bits >> 3) & 0x1
        bits = bits + (1-2*b3)*2**(3)

    k7 = (GIFT_RC[r]>>1)&0x1;
    if(k7==1):
        b7 = (bits >> 7) & 0x1
        bits = bits + (1-2*b7)*2**(7)

    k11 = (GIFT_RC[r]>>2)&0x1;
    if(k11==1):
        b11 = (bits >> 11) & 0x1
        bits = bits + (1-2*b11)*2**(11)

    k15 = (GIFT_RC[r]>>3)&0x1;
    if(k15==1):
        b15 = (bits >> 15) & 0x1
        bits = bits + (1-2*b15)*2**(15)

    k19 = (GIFT_RC[r]>>4)&0x1;
    if(k19==1):
        b19 = (bits >> 19) & 0x1
        bits = bits + (1-2*b19)*2**(19)


   k23 = (GIFT_RC[r]>>5)&0x1;
   if(k23==1):
       b23 = (bits >> 23) & 0x1
       bits = bits + (1-2*b23)*2**(23)

   b127 = (bits >> 127) & 0x1
   bits = bits + (1-2*b127)*2**(127)

   inpu = bits
 #  print('Inv Add round key = ', hex(bits))
```

Inverse permutation and Sbox implementation.

```
    # Permute the bits
        perm_val = 0;
        for i in range(128):
            bit = (inpu >> i) & 0X1;
            multi = GIFT_P_inv[i]
            perm_val = perm_val + bit*2**multi
     #   print('perm val =' ,hex(perm_val))

        inpu = 0;
        temp_text = perm_val;


# sub cell

        for i in range(31,-1,-1):
            bit = (temp_text >> 4*i) & 0XF;
            inp = GIFT_S_inv[bit];
            inpu = inpu * 16 + inp;
     #   print('After S_box = ',hex(inpu))

    return(inpu)
```

5.3 **Control Function**

```
rec_text =''
cy_comp = 0;
init = 0
for tak in range(lp_req-1,-1,-1):
    r_inp = (Input_text >> tak*128) & 0Xffffffffffffffffffffffffffffffff;
    a_cypher  = GIFT_enc(r_inp,Enc_Key,3)
    print('128 bit cypher = ',hex(a_cypher))
    #print('enc tak = ',tak)
    cy_comp = (cy_comp << 128) + a_cypher

print(' \n Comp cypher = ',cy_comp)

print('\n -------------------- Decryption part  ---------------------')

for tak in range(lp_req-1,-1,-1):
    cypher = (cy_comp >> tak*128) & 0Xffffffffffffffffffffffffffffffff;
   # print('in recovering part')
    Rec_text = GIFT_Decryp(cypher,Enc_Key,3)
    Rec_text

    Rec_text1 = hex(Rec_text)

    des = hex_to_ascii(Rec_text1)
   # print(des)
    m = str(des)
    l = len(m) - 1
    add = m[2:l]
    print(add)
    rec_text = rec_text + add

print('\n Recovered Text = ',rec_text)
```

5.4 **Running and Output**

Enter Text Message upon Prompted with options. Programme runs, shows all steps and gives Output.

```
Enter text message = Final Testing of Encyption/Decryption of GIFT Cipher Implementation 22 Apr 2023
158
5
128 bit cypher =  0x5186334a830aa0d74a46be360e3c7e17
128 bit cypher =  0x9e26985d6ba38067e48731599d56a212
128 bit cypher =  0x26196059902921a894cc8114a8361c2
128 bit cypher =  0xeda74d30a787bd78f9365329fbf0441b
128 bit cypher =  0xc105b00e1a001705507da2dc78f9cce7

 Comp cypher =  145292739564223311389792105255515355301025830580059384300399751492202724758710952829750056383433966134018097

 -------------------- Decryption part  --------------------
Final Testing o
f Encyption/Decr
yption of GIFT C
ipher Implementa
tion 22 Apr 2023

 Recovered Text =  Final Testing of Encyption/Decryption of GIFT Cipher Implementation 22 Apr 2023
```

5.5 **Stage by Stage Output is as under**

```
After S_box =  0x11633d383a3541f63f9c963d3839413e
perm val = 0x935bb9d136df4257996919236324f32
Add round key =  0x8b35999f136ff62579b6919034324d38
updated Key after 0  =  0x13451452000000000000000000a30f95
-----------------------------------

After S_box =  0x27cfdddeac3ee34f9d73dad1c6c460c2
perm val = 0xad9e1764fd6718ca6fa6fd465ce9f9c0
Add round key =  0x2d9e1764dd4718e86fa6df647cebfb4a
updated Key after 1  =  0xc028f950134514520000000000000000
-----------------------------------

After S_box =  0x40d8a9360069a2823eb30e369587e76b
perm val = 0xd182330e870aa4d34a42ba360a3876df
Add round key =  0x5186334a830aa0d74a46be360e3c7e17
updated Key after 2  =  0xc028f9501345145200000000
-----------------------------------

After S_box =  0xfa23cc6b2c1bb1096b6378c318c598a9
perm val = 0x6c98640820233ac8bea1efd9bf59311b
Add round key =  0xa898640820637ac8fae5afddbb5db993
updated Key after 3  =  0xc028f95013451452
-----------------------------------

128 bit cypher =  0xa898640820637ac8fae5afddbb5db993
After S_box =  0x33416f383c9d91963d3e384e663f3c94
perm val = 0x2f9198f133ed766e5893bf2916586074
Add round key =  0xad91baf333ef746e58b3bf2b1458627e
updated Key after 0  =  0x13451452000000000000000000a30f95
-----------------------------------

After S_box =  0xb0da7beccc8e9638f27c7e47a6f23498
perm val = 0x9ecbf639afc066aa7da15f6182cea5e4
Add round key =  0x1ecbf6398fe066887da17d43a2cca76e
updated Key after 1  =  0xc028f950134514520000000000000000
-----------------------------------

After S_box =  0xa857e3cd2e81332290ba906cb455b938
perm val = 0x1e2298196fa38463e48335599952aada
Add round key =  0x9e26985d6ba38067e48731599d56a212
updated Key after 2  =  0xc028f9501345145200000000
-----------------------------------

After S_box =  0xd843d2f037bc21398629cafdd0f3b4a4
perm val = 0x47b9af5008f30c8ef5100c719a63fbbe
Add round key =  0x83b9af5008b34c8eb1544c759e677336
updated Key after 3  =  0xc028f95013451452
-----------------------------------

128 bit cypher =  0x83b9af5008b34c8eb1544c759e677336
After S_box =  0x9d91963d3e38413e3341696d63f6416c
perm val = 0x1bbd2c7c942537f591925c64df621322
Add round key =  0x99bd0e7e942735f591b25c66dd621128
updated Key after 0  =  0x13451452000000000000000000a30f95
-----------------------------------
```

```
After S_box =  0xdd701898d649cfefda74f5330034aa42
perm val = 0x59ee7512180d4342d95ed30ee0dfaf28
Add round key =  0xd9ee7512382d4360d95ef12cc0ddada2
updated Key after 1  =  0xc028f95013451452000000000000000000
-----------------------------------

After S_box =  0xd889fa4c2406c310df8ea455100b0b4
perm val = 0x826596419d02961e8948cc114e87690a
Add round key =  0x26196059902921a894cc8114a8361c2
updated Key after 2  =  0xc028f9501345145200000000
-----------------------------------

After S_box =  0x143ad31fdd14d4ab2d6552aa6b2c3a54
perm val = 0x9f5c0ee3055878763393c38d69cf7220
Add round key =  0x5b5c0ee30518387677d783896dcbfaa8
updated Key after 3  =  0xc028f95013451452
-----------------------------------

128 bit cypher =  0x5b5c0ee30518387677d783896dcbfaa8
After S_box =  0x3d91323f94416d3091353f303f38963a
perm val = 0x1b0513bbb60353309358199353d6b66e
Add round key =  0x990531b9b60151309378199151d6b464
updated Key after 0  =  0x13451452000000000000000000a30f95
-----------------------------------

After S_box =  0xdd1fca7d731afac1dc92addafa037636
perm val = 0xdff65967543a8b26bc3dbeb3db09cc96
Add round key =  0x5ff65967741a8b04bc3d9c91fb0bce1c
updated Key after 1  =  0xc028f95013451452000000000000000000
-----------------------------------

After S_box =  0xfee3fd3996ab271675c0d5dae71758a5
perm val = 0x6da34d74a387b97cf9325729fff44cd3
Add round key =  0xeda74d30a787bd78f9365329fbf0441b
updated Key after 2  =  0xc028f9501345145200000000
-----------------------------------

After S_box =  0x80b960c1b9297092edc3fc4de7e166a7
perm val = 0x94854c668a3ab6be0493ed42b1b0ddf7
Add round key =  0x50854c668a7af6be40d7ad46b5b4557f
updated Key after 3  =  0xc028f95013451452
-----------------------------------

128 bit cypher =  0x50854c668a7af6be40d7ad46b5b4557f
After S_box =  0x963d3e3841c4c4416a919441c4c1c4cc
perm val = 0xbb44704c42d0a08c19448544f60d1ddc
Add round key =  0x3944524e42d2a28c19648546f40d1fd6
updated Key after 0  =  0x13451452000000000000000000a30f95
-----------------------------------

After S_box =  0xcd66f4686404b425ad362f63e610ae03
perm val = 0x4c40127252667322e501ae0aee4fe7cd
Add round key =  0xcc40127272467300e5018c28ce4de547
updated Key after 1  =  0xc028f95013451452000000000000000000
-----------------------------------

After S_box =  0x5561a49494639c118f1a254258608f69
perm val = 0x4101b04a1e0013015079a6dc7cfdc42f
```

```
Add round key =  0xc105b00e1a001705507da2dc78f9cce7
updated Key after 2  =  0xc028f9501345145200000000
-----------------------------------

After S_box =  0x5a1f7118ab11a91ff190b40592ed5589
perm val = 0xfd3950ac4337b6c9b18a115110998db5
Add round key =  0x393950ac4377f6c9f5ce5155149d053d
updated Key after 3  =  0xc028f95013451452
-----------------------------------

128 bit cypher =  0x393950ac4377f6c9f5ce5155149d053d

 Comp cypher =  300471069069956353695490499041205826416974345604895952024012271469957862376138276724880245720402026974104570479361524920555687454257634210511519925056527483898602263064258600064492355871290297 3

 ------------------- Decryption part  --------------------
updated Key after 0  =  0x13451452000000000000000000a30f95
updated Key after 1  =  0xc028f95013451452000000000000000000
updated Key after 2  =  0xc028f9501345145200000000
updated Key after 3  =  0xc028f95013451452
-------- Decryption Round  3 Starting ----------
Key used =  0xc028f9501345145200000000
Bits after Xoring = 0xec98640820233ac8bea1efd9bf59b993
Inv Add round key =  0x6c98640820233ac8bea1efd9bf59311b
perm val = 0xfa23cc6b2c1bb1096b6378c318c598a9
After S_box =  0x5186334a830aa0d74a46be360e3c7e17
-------- Decryption Round  2 Starting ----------
Key used =  0xc028f95013451452000000000000000000
Bits after Xoring = 0x5182330e870aa4d34a42ba360a387e57
Inv Add round key =  0xd182330e870aa4d34a42ba360a3876df
perm val = 0x40d8a9360069a2823eb30e369587e76b
After S_box =  0x2d9e1764dd4718e86fa6df647cebfb4a
-------- Decryption Round  1 Starting ----------
Key used =  0x13451452000000000000000000a30f95
Bits after Xoring = 0x2d9e1764fd6718ca6fa6fd465ce9f948
Inv Add round key =  0xad9e1764fd6718ca6fa6fd465ce9f9c0
perm val = 0x27cfdddeac3ee34f9d73dad1c6c460c2
After S_box =  0x8b35999f136ff62579b6919034324d38
-------- Decryption Round  0 Starting ----------
Key used =  0xa30f954d142145
Bits after Xoring = 0x8935bb9d136df4257996919236324f3a
Inv Add round key =  0x935bb9d136df4257996919236324f32
perm val = 0x11633d383a3541f63f9c963d3839413e
After S_box =  0x46696e616c2054657374696e67206f
Final Testing o
updated Key after 0  =  0x13451452000000000000000000a30f95
updated Key after 1  =  0xc028f95013451452000000000000000000
updated Key after 2  =  0xc028f9501345145200000000
updated Key after 3  =  0xc028f95013451452
-------- Decryption Round  3 Starting ----------
Key used =  0xc028f9501345145200000000
Bits after Xoring = 0xc7b9af5008f30c8ef5100c719a637336
Inv Add round key =  0x47b9af5008f30c8ef5100c719a63fbbe
perm val = 0xd843d2f037bc21398629cafdd0f3b4a4
After S_box =  0x9e26985d6ba38067e48731599d56a212
-------- Decryption Round  2 Starting ----------
Key used =  0xc028f95013451452000000000000000000
```

```
Bits after Xoring = 0x9e2298196fa38463e48335599952a252
Inv Add round key =  0x1e2298196fa38463e48335599952aada
perm val = 0xa857e3cd2e81332290ba906cb455b938
After S_box =  0x1ecbf6398fe066887da17d43a2cca76e
-------- Decryption Round  1 Starting ----------
Key used =  0x13451452000000000000000000a30f95
Bits after Xoring = 0x1ecbf639afc066aa7da15f6182cea56c
Inv Add round key =  0x9ecbf639afc066aa7da15f6182cea5e4
perm val = 0xb0da7beccc8e9638f27c7e47a6f23498
After S_box =  0xad91baf333ef746e58b3bf2b1458627e
-------- Decryption Round  0 Starting ----------
Key used =  0xa30f954d142145
Bits after Xoring = 0xaf9198f133ed766e5893bf291658607c
Inv Add round key =  0x2f9198f133ed766e5893bf2916586074
perm val = 0x33416f383c9d91963d3e384e663f3c94
After S_box =  0x6620456e63797074696f6e2f44656372
f Encyption/Decr
updated Key after 0  =  0x13451452000000000000000000a30f95
updated Key after 1  =  0xc028f9501345145200000000000000000
updated Key after 2  =  0xc028f9501345145200000000
updated Key after 3  =  0xc028f95013451452
-------- Decryption Round  3 Starting ----------
Key used =  0xc028f9501345145200000000
Bits after Xoring = 0x1f5c0ee3055878763393c38d69cffaa8
Inv Add round key =  0x9f5c0ee3055878763393c38d69cf7220
perm val = 0x143ad31fdd14d4ab2d6552aa6b2c3a54
After S_box =  0x26196059902921a894cc8114a8361c2
-------- Decryption Round  2 Starting ----------
Key used =  0xc028f9501345145200000000000000000
Bits after Xoring = 0x26596419d02961e8948cc114e876182
Inv Add round key =  0x826596419d02961e8948cc114e87690a
perm val = 0xd889fa4c2406c310df8ea455100b0b4
After S_box =  0xd9ee7512382d4360d95ef12cc0ddada2
-------- Decryption Round  1 Starting ----------
Key used =  0x13451452000000000000000000a30f95
Bits after Xoring = 0xd9ee7512180d4342d95ed30ee0dfafa0
Inv Add round key =  0x59ee7512180d4342d95ed30ee0dfaf28
perm val = 0xdd701898d649cfefda74f5330034aa42
After S_box =  0x99bd0e7e942735f591b25c66dd621128
-------- Decryption Round  0 Starting ----------
Key used =  0xa30f954d142145
Bits after Xoring = 0x9bbd2c7c942537f591925c64df62132a
Inv Add round key =  0x1bbd2c7c942537f591925c64df621322
perm val = 0x9d91963d3e38413e3341696d63f6416c
After S_box =  0x797074696f6e206f6620474946542043
yption of GIFT C
updated Key after 0  =  0x13451452000000000000000000a30f95
updated Key after 1  =  0xc028f9501345145200000000000000000
updated Key after 2  =  0xc028f9501345145200000000
updated Key after 3  =  0xc028f95013451452
-------- Decryption Round  3 Starting ----------
Key used =  0xc028f9501345145200000000
Bits after Xoring = 0x14854c668a3ab6be0493ed42b1b0557f
Inv Add round key =  0x94854c668a3ab6be0493ed42b1b0ddf7
perm val = 0x80b960c1b9297092edc3fc4de7e166a7
After S_box =  0xeda74d30a787bd78f9365329fbf0441b
-------- Decryption Round  2 Starting ----------
Key used =  0xc028f9501345145200000000000000000
```

```
Bits after Xoring = 0xeda34d74a387b97cf9325729fff4445b
Inv Add round key =  0x6da34d74a387b97cf9325729fff44cd3
perm val = 0xfee3fd3996ab271675c0d5dae71758a5
After S_box =  0x5ff65967741a8b04bc3d9c91fb0bce1c
-------- Decryption Round  1 Starting ----------
Key used =  0x13451452000000000000000000a30f95
Bits after Xoring = 0x5ff65967543a8b26bc3dbeb3db09cc1e
Inv Add round key =  0xdff65967543a8b26bc3dbeb3db09cc96
perm val = 0xdd1fca7d731afac1dc92addafa037636
After S_box =  0x990531b9b60151309378199151d6b464
-------- Decryption Round  0 Starting ----------
Key used =  0xa30f954d142145
Bits after Xoring = 0x9b0513bbb60353309358199353d6b666
Inv Add round key =  0x1b0513bbb60353309358199353d6b66e
perm val = 0x3d91323f94416d3091353f303f38963a
After S_box =  0x697068657220496d706c656d656e7461
ipher Implementa
updated Key after 0  =  0x13451452000000000000000000a30f95
updated Key after 1  =  0xc028f9501345145200000000000000000
updated Key after 2  =  0xc028f9501345145200000000
updated Key after 3  =  0xc028f95013451452
-------- Decryption Round  3 Starting ----------
Key used =  0xc028f9501345145200000000
Bits after Xoring = 0x7d3950ac4337b6c9b18a11511099053d
Inv Add round key =  0xfd3950ac4337b6c9b18a115110998db5
perm val = 0x5a1f7118ab11a91ff190b40592ed5589
After S_box =  0xc105b00e1a001705507da2dc78f9cce7
-------- Decryption Round  2 Starting ----------
Key used =  0xc028f95013451452000000000000000000
Bits after Xoring = 0xc101b04a1e0013015079a6dc7cfdcca7
Inv Add round key =  0x4101b04a1e0013015079a6dc7cfdc42f
perm val = 0x5561a49494639c118f1a254258608f69
After S_box =  0xcc40127272467300e5018c28ce4de547
-------- Decryption Round  1 Starting ----------
Key used =  0x13451452000000000000000000a30f95
Bits after Xoring = 0xcc40127252667322e501ae0aee4fe745
Inv Add round key =  0x4c40127252667322e501ae0aee4fe7cd
perm val = 0xcd66f4686404b425ad362f63e610ae03
After S_box =  0x3944524e42d2a28c19648546f40d1fd6
-------- Decryption Round  0 Starting ----------
Key used =  0xa30f954d142145
Bits after Xoring = 0x3b44704c42d0a08c19448544f60d1dd4
Inv Add round key =  0xbb44704c42d0a08c19448544f60d1ddc
perm val = 0x963d3e3841c4c4416a919441c4c1c4cc
After S_box =  0x74696f6e203232204170722032303233
tion 22 Apr 2023

 Recovered Text =  Final Testing of Encyption/Decryption of GIFT Cipher Imp
lementation 22 Apr 2023
```

Chapter 6: Future Scope of Work

The GIFT cipher has already demonstrated its potential as a lightweight encryption algorithm with strong security properties. However, there are several areas that offer promising avenues for future research and development. The potential future scope of work on the GIFT cipher includes:

- Cryptanalysis: Despite its security features, further analysis of the GIFT cipher's resistance against advanced cryptanalysis techniques would be beneficial. Researchers can explore different attack scenarios and assess the cipher's robustness. This can help in identifying potential vulnerabilities and refining the cipher's design.
- Hardware Implementations: While the GIFT cipher has shown efficient performance in software implementations, exploring its hardware implementations can lead to even faster and more energy-efficient encryption and decryption processes. Designing dedicated hardware architectures optimized for GIFT can unlock its full potential in resource-constrained devices.
- Side-Channel Analysis: Conducting side-channel analysis on the GIFT cipher can evaluate its resistance against side-channel attacks, such as power analysis and timing attacks. This research can contribute to developing countermeasures and ensuring the cipher's security in practical scenarios.
- Standardization: GIFT cipher standardization can help establish it as a recognized encryption algorithm in the industry. Collaborating with cryptographic communities and organizations to evaluate and potentially standardize the GIFT cipher can increase its adoption and facilitate interoperability.
- Lightweight Protocol Integration: Integrating the GIFT cipher into lightweight cryptographic protocols, such as those used in IoT and wireless sensor networks, can further enhance the security and efficiency of these systems. Research can focus on exploring the compatibility and performance of GIFT within different lightweight protocol frameworks.
- Post-Quantum Security: With the advent of quantum computing, there is a growing need for post-quantum secure algorithms. Investigating the post-quantum security properties of the GIFT cipher and exploring its potential as a lightweight post-quantum encryption solution can be an interesting direction for future research.

- Cryptographic Implementations: Evaluating the GIFT cipher's implementation on different platforms and programming languages can provide insights into its portability, performance, and ease of integration. This includes exploring implementations on embedded systems, mobile devices, and specialized hardware.

By focusing on these areas of research, the future work on the GIFT cipher can further strengthen its security, optimize its performance, and expand its applicability in various domains requiring lightweight and secure encryption.

Bibliography

1. Author(s): Jang, Jae-Hyun; Hong, Seokhie; Lim, Jongin

Title: Quantum Resource Estimation for Lightweight Block Ciphers

Book Title: Cryptology and Network Security

Editor(s): Nguyen, Phong Q.; Oswald, Elisabeth; Rangasamy, Jayavardhana

Publisher: Springer International Publishing

Year: 2017

Page range: 315-333

URL: https://link.springer.com/chapter/10.1007/978-3-319-66787-4_16

2. Singh, Simon -. "'The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography.'" Simon Singh, May , simonsingh.net/cryptography/cipher-challenge/.

3. Joyner, David. "Historical Cipher Machines and Much More." Crypto Museum, "Ciphers" - Crypto Museum, Feb. 2004, cryptomuseum.com/.

4. Computer Security Division, Information Technology Laboratory, et al. "Lightweight Cryptography: CSRC." CSRC, D. A. Kucuk and S. Zeadally, May 2017, csrc.nist.gov/Projects/Lightweight-Cryptography.

5. BUEREN, GERALDINE VAN. Childhood Abused: Protecting Children against Torture, Cruel, Inhuman and Degrading Treatment... and Punishment. ed., ROUTLEDGE, 2020.

6. Banik, Subhadeep, et al. "GIFT: A Small Present." Lecture Notes in Computer Science, Springer Science+Business Media, 2017, pp. 321–45. https://doi.org/10.1007/978-3-319-66787-4_16.

7. Jang, Kyungbae, et al. "Efficient Implementation of PRESENT and GIFT on Quantum Computers." Applied Sciences, vol. 11, no. 11, MDPI, May 2021, p. 4776. https://doi.org/10.3390/app11114776.