The Scientific Temper (2025) Vol. 16 (11): 5067-5078

E-ISSN: 2231-6396, ISSN: 0976-8653

Doi: 10.58414/SCIENTIFICTEMPER.2025.16.11.12

https://scientifictemper.com/

RESEARCH ARTICLE

Enhanced Symmetric Cryptography Technique (ESCTGPU) for Secure Communication between the IoT Gateway and the public Cloud Environment

Priscilla I[1*] and Jayasimman Lawrence[2]

Abstract

The rapid expansion of the Internet of Things (IoT) has amplified the demand for secure and efficient communication with cloud platforms, where sensitive data is collected, processed, and stored. Conventional encryption standards such as DES and blowfish, though effective, are not ideally suited for resource-constrained IoT environments due to their computational overhead. To address this challenge, this paper proposes the Enhanced Symmetric Cryptography Technique to secure Gateway to Public Cloud ($ESCTGP_U$), a lightweight yet robust block cipher specifically designed for IoT–cloud integration. $ESCTGP_U$ employs an 8-round structure with dual subkey mixing, adaptive bit rotations, and layered permutations, ensuring strong confusion and diffusion while minimizing execution time. Experimental evaluation using real IoT sensor payloads demonstrates that $ESCTGP_U$ achieves up to 40% faster encryption and decryption than DES and outperforms Blowfish in terms of efficiency, while attaining a measured 94% security strength, compared with 78% for DES and 84% for Blowfish. These results confirm that $ESCTGP_U$ offers a practical balance between speed and resilience, making it a suitable candidate for securing IoT–cloud communication where both performance and confidentiality are critical.

Keywords: Internet of Things (IoT), Cloud Security, Lightweight Cryptography, Data Encryption, $ESCTGP_U$ Algorithm, Secure Communication, Symmetric Key Encryption, Performance Evaluation

关键词：物联网 (IoT)、云安全、轻量级密码学、数据加密、ESCTGPU 算法、安全通信、对称密钥加密、性能评估

Introduction

The Internet of Things (IoT) has rapidly evolved into one of the most transformative technologies of the digital era. By connecting billions of heterogeneous devices—ranging from sensors and wearables to industrial controllers—IoT enables real-time data acquisition, monitoring, and decision-making across multiple domains such as healthcare, transportation, smart cities, and industrial automation (Rana M., et al. 2022). This massive integration of devices has led to exponential data generation, which requires scalable platforms for storage and computation. Cloud computing has become the de facto backbone for IoT systems because of its elasticity, virtually unlimited storage, and powerful data processing capabilities (Suryateja P. S. 2024).

However, this dependency on cloud platforms raises severe security and privacy concerns. IoT devices are resource-constrained in terms of memory, computation, and energy, making them incapable of running heavy cryptographic schemes. Consequently, lightweight algorithms are typically used to secure device-to-gateway communication (Al-Shatari, et al., 2023). While these methods reduce computational overhead, they are insufficient to protect data once it reaches the cloud—a public and highly vulnerable environment where adversaries may exploit advanced attack vectors, including brute-force decryption, side-channel attacks, and unauthorized access. Thus, IoT–cloud communication introduces a new layer of security risks, particularly regarding data confidentiality and integrity (Gușiță B., 2025).

[1]Full-Time Research Scholar, Department of Computer Science, Bishop Heber College (Autonomous), Trichy, Affiliated to Bharathidasan University, Tiruchirappalli, Tamil Nadu, India.

[2]Assistant Professor, Department of Computer Science, Bishop Heber College (Autonomous), Trichy Affiliated to Bharathidasan University, Tiruchirappalli, Tamil Nadu, India.

***Corresponding Author:** Priscilla I, Full-Time Research Scholar, Department of Computer Science, Bishop Heber College (Autonomous), Trichy, Affiliated to Bharathidasan University, Tiruchirappalli, Tamil Nadu, India, E-Mail:

How to cite this article: Priscilla, I., Lawrence, J. (2025). Enhanced Symmetric Cryptography Technique (ESCTGPU) for Secure Communication between the IoT Gateway and the public Cloud Environment. The Scientific Temper, **16**(11):5067-5078.

Doi: 10.58414/SCIENTIFICTEMPER.2025.16.11.12

Source of support: Nil

Conflict of interest: None.

Received: 02/11/2025 **Accepted:** 15/11/2025 **Published:** 22/11/2025

The core challenge lies in designing encryption mechanisms that balance two conflicting requirements: efficiency for IoT devices and robustness against sophisticated cloud-based attacks. Traditional symmetric encryption algorithms such as DES and Blowfish, although secure, consume significant processing time and are not optimized for large-scale IoT deployments (Sabri O, et al., 2025). On the other hand, hybrid approaches combining symmetric and asymmetric cryptography enhance security but often increase complexity and latency. These trade-offs highlight the urgent need for specialized cryptographic solutions tailored for IoT–cloud ecosystems (Xue J, et al., 2023).

To address this gap, this research proposes the Enhanced Symmetric Cryptography Technique ($ESCTGP_U$). Unlike conventional lightweight schemes designed solely for IoT devices, $ESCTGP_U$ strengthens the security of data during transmission from the gateway to the cloud. By employing a block cipher structure with multi-round permutations, substitutions, and key variations, $ESCTGP_U$ ensures that ciphertexts are computationally resistant to cryptanalysis while maintaining efficiency.

The contributions of this research are threefold:

- Proposal of $ESCTGP_U$, a symmetric block cipher designed to balance computational efficiency and security strength in IoT–cloud environments.
- Implementation and experimental evaluation of $ESCTGP_U$ in a real-world IoT testbed using Arduino microcontrollers, sensors, and cloud connectivity.
- Performance and security analysis, demonstrating that $ESCTGP_U$ outperforms DES and Blowfish in terms of encryption/decryption speed and achieves higher resistance against cryptanalytic attacks.

By bridging the gap between lightweight encryption and robust cryptography, $ESCTGP_U$ provides a practical and scalable solution for securing IoT–cloud communication. This work contributes to building trust in IoT systems by ensuring that sensitive sensor data remains protected even in hostile cloud environments.

Cryptography Techniques for Securing IoT–Cloud Communication

The integration of IoT with cloud platforms has created an ecosystem where massive volumes of data are generated, transmitted, and stored. While this combination enables scalability and intelligence, it also opens new avenues for cyberattacks. To safeguard sensitive information during transmission and storage, cryptography serves as the cornerstone of IoT–cloud security (Qasem M. A., 2024). Broadly, cryptographic techniques are divided into symmetric key cryptography, asymmetric key cryptography, and their hybrid or advanced variations. Each technique has distinct strengths and limitations, and their suitability depends on factors such as computational efficiency, memory usage, and the threat environment (Almutairi M. et al., 2025).

Symmetric Key Cryptography

In symmetric key systems, a single secret key is shared between the sender and receiver to perform both encryption and decryption. Because the same key is used at both ends, the security of the system relies heavily on how well the key is kept confidential (Rosero-Montalvo P. D, et al., 2022).

Block Ciphers

These algorithms divide data into fixed-sized blocks (e.g., 64-bit or 128-bit) and apply multiple rounds of permutation, substitution, and XOR operations to produce ciphertext. Examples include DES (Alhassan A. B. et al., 2024), Blowfish (Kaur G. et al., 2025) and newer lightweight ciphers optimized for IoT. Block ciphers are widely used because they can encrypt large amounts of data efficiently and achieve high levels of diffusion and confusion.

Stream Ciphers

Instead of working on blocks, stream ciphers encrypt data bit by bit or byte by byte using a keystream. They are lightweight, fast, and suitable for low-power devices. Protocols like RC4 (historically used) and newer lightweight designs are often considered for IoT sensors.

Advantages

Fast execution, low resource consumption, and suitability for bulk data encryption.

Challenges

Secure key distribution is difficult, especially in distributed IoT environments where millions of devices may need unique keys.

Asymmetric Key Cryptography

Asymmetric cryptography uses a pair of keys: a public key for encryption and a private key for decryption. Unlike symmetric systems, there is no need for both parties to share the same secret key in advance. This property makes it particularly useful for authentication, digital signatures, and secure key exchange (Raj Y. S et al., 2021).

RSA (Rivest–Shamir–Adleman)

One of the earliest and most widely used public-key systems, RSA offers strong security but requires intensive computations, making it less suitable for small IoT devices.

Elliptic Curve Cryptography (ECC)

ECC achieves the same security strength as RSA but with smaller key sizes, reducing computational and memory requirements. This makes ECC a preferred choice in IoT–cloud communication, particularly for authenticating devices and establishing secure channels.

Advantages

Eliminates the key distribution problem of symmetric systems and provides stronger authentication.

Challenges

Computationally expensive for IoT sensors and not ideal for continuous encryption of large data streams.

Hybrid Cryptographic Approaches

To balance efficiency and robustness, researchers often combine symmetric and asymmetric techniques (Selvi P. et al., 2025). Typically, asymmetric cryptography (like ECC or RSA) is used to securely exchange a symmetric session key, and then the actual data transmission is encrypted using a faster symmetric cipher such as AES. This hybrid model is widely implemented in protocols like SSL/TLS, which are increasingly adapted to IoT–cloud systems (Zhang L. el al., 2024).

Advantages

Combines the speed of symmetric algorithms with the strong authentication of asymmetric ones.

Challenges

Still incurs additional overhead due to asymmetric operations, which may strain low-power IoT devices.

Related work

The paper examines the resilience of the PRESENT lightweight block cipher against electromagnetic side-channel attacks, a threat often overlooked compared to traditional power analysis. The author Gunathilake et al. (2021) employ both simple electromagnetic analysis (SEMA) and correlation electromagnetic analysis (CEMA), using probes and oscilloscopes to measure EM emissions from an Arduino Uno implementing PRESENT encryption. Experimental results reveal that electromagnetic leakage can expose partial key information—up to seven bytes in some instances—depending on probe type and filtering settings, with certain bytes exhibiting greater susceptibility. These observations highlight that while lightweight ciphers like PRESENT are computationally efficient for IoT applications, they are not inherently resistant to physical leakage. The study underscores the importance of implementing hardware-level countermeasures such as electromagnetic shielding, noise injection, and obfuscation. By demonstrating the cipher's vulnerability at the physical layer, the paper contributes valuable insight into the ongoing effort to integrate side-channel resistance into secure IoT cryptographic design.

The paper provides a detailed survey of cryptographic algorithms aimed at improving IoT security, emphasizing the trade-off between robust protection and the limited resources of IoT devices. Thabit et al. (2023) reviews lightweight block ciphers, stream ciphers, and hybrid approaches, assessing them based on computational efficiency, memory footprint, and resistance to common cryptanalytic attacks. A notable aspect of the work is its balanced view—while highlighting the benefits of lightweight cryptography for efficiency, it also warns against excessive simplification that could undermine security. The authors further underline the potential of hybrid cryptographic schemes that combine symmetric and asymmetric techniques to achieve both speed and robustness. By categorizing encryption strategies across the device, edge, and cloud layers, the paper provides a clear structural perspective for practitioners. Importantly, it identifies ongoing challenges such as ensuring side-channel resistance, achieving secure key management, and developing IoT-specific standards, making it a valuable reference for emerging models like ESCTGP$_U$, which aim to balance performance and security in IoT–cloud communications.

The paper introduces the RBFK cipher, a lightweight symmetric block cipher designed specifically for IoT devices operating in edge computing environments, where conventional algorithms such as AES and DES are computationally expensive. Rana et al. (2023) employs a randomized butterfly architecture for key scheduling, enabling the generation of highly sensitive round keys with strong avalanche effects while maintaining minimal processing overhead. The cipher processes 64-bit data blocks with 64- or 128-bit keys over five rounds, utilizing XOR, XNOR, substitution boxes, and scan patterns to enhance both confusion and diffusion properties. Experimental evaluation using the FELICS benchmarking suite demonstrates that RBFK achieves lower cycle counts, memory usage, and power consumption compared to other lightweight algorithms like PRESENT, SPECK, and SIT. Additionally, MATLAB-based image encryption tests validate its resistance to statistical and differential attacks. Overall, the paper showcases RBFK as a secure, efficient, and resource-aware encryption scheme well-suited for safeguarding data in IoT edge environments.

The paper introduces GFRX, a lightweight block cipher designed specifically for IoT devices that have limited computing and storage capacity by Zhang et al. (2023). Traditional Feistel ciphers are criticized for their slow diffusion, since only half of the plaintext changes in each round, requiring many iterations to reach acceptable security. To overcome this, the authors combine a generalized Feistel structure with ARX operations (Addition/AND, Rotation, XOR), applying two distinct nonlinear functions across all branches. This design improves diffusion speed, strengthens confusion, and reduces the number of rounds needed to achieve the avalanche effect—GFRX reaches full diffusion in just six rounds. The encryption and decryption structures are nearly identical, minimizing extra hardware costs, and the round function is reused

during key scheduling to further save resources. Security analysis shows that the cipher withstands up to 19 rounds of differential attacks and 13 rounds of linear attacks, giving it a comfortable safety margin. Performance testing confirms that GFRX outperforms existing lightweight ciphers like SIMON and SPECK in terms of avalanche behavior, while hardware results on FPGA and ASIC show very low area consumption (as low as ~886 GE) with flexible serialization for different throughput needs. Overall, the paper demonstrates that GFRX is a practical and efficient cipher for IoT environments, offering a strong balance of security, diffusion speed, and hardware efficiency for resource-constrained nodes.

The paper introduces GFSPX, a lightweight block cipher specifically designed for resource-constrained IoT devices, aiming to improve security without compromising efficiency. Zhang et al. (2024) builds on a generalized Feistel structure integrated with Substitution–Permutation Network (SPN) principles to overcome the slow diffusion problem typical of traditional Feistel designs. To enhance mixing speed, the cipher employs ARX operations—Addition, Rotation, and XOR—on selected portions of the plaintext, which eliminates the need for large S-box tables or complex hardware components. Experimental results demonstrate that GFSPX achieves a full avalanche effect within only six rounds, indicating rapid diffusion and strong resistance to key-related attacks. Comprehensive cryptanalysis confirms robustness against differential, linear, algebraic, and structural attacks, while implementation results show a compact hardware footprint (~1,715 GE) and a high software throughput of 12.31 Mb/s. Overall, the study presents GFSPX as a balanced and efficient cryptographic design, combining low computational cost with solid security guarantees, making it highly suitable for lightweight IoT encryption scenarios.

The paper proposed a layered security framework for IoT–Cloud data protection. Farshadinia, H., (2025) integrates multi-stage lightweight cryptography to address gaps in traditional methods. Conventional blockchain signatures (ECDSA, ZSS) are improved for efficiency and speed. Layer 1 (H.E.EZ): combines Hyperledger Fabric, refined block encryption, and hybrid signatures. Layer 2: introduces credential management to validate blockchain-encrypted data. Layer 3 (C-AUDIT): manages audit trails, event ordering, and synchronization. The design reduces reliance on third-party auditors and minimizes communication overhead. Evaluations show faster execution, lower traffic, and better scalability than prior solutions. Security analysis confirms stronger protection against unauthorized access and tampering. Overall, the framework offers a robust, efficient, and scalable model for IoT–Cloud security.

Methodology

The methodology of this research is designed to secure IoT–cloud communication by introducing a two-level encryption process that strengthens data protection as it travels from devices to the cloud. The first level needs to secure the communication between device-to-gateway and second level is to secure data between IoT gateway to cloud storage using Enhanced Symmetric Cryptography Technique($ESCTGP_U$). The paper presents second level of security between IoT gateway to cloud. This layered design addresses the dual challenges of computational efficiency and robustness against attacks.

$ESCTGP_U$ Design Principles

The $ESCTGP_U$ is a symmetric block cipher developed for the gateway system. Its core features include:

- **Block Size**: Operates on 64-bit blocks of data.
- **Rounds**: Executes 8 rounds of transformations.
- **Keys**: A 64-bit master key is expanded into 16 subkeys, with two subkeys used in each round.
- **Operations**: Each round involves initial permutation, XOR with subkeys, row-shuffling permutations, bit rotations based on "1" counts, and a final permutation.
- **Encryption Style**: Introduces confusion (bit substitution via XOR operations) and diffusion (bit permutations and rotations) to make cryptanalysis difficult.

This design is chosen to create randomized ciphertext outputs, ensuring that identical plaintext inputs generate different ciphertexts, a property absent in many conventional techniques.

$ESCTGP_U$ Encryption Procedure

The Enhanced Symmetric Cryptography Technique ($ESCTGP_U$) algorithm secures IoT–cloud communication through a series of carefully designed transformations. The process ensures both confusion (via XOR operations and key mixing) and diffusion (via permutations, rotations, and substitutions). Below is the enhanced procedure:

Step 1: Input Acquisition

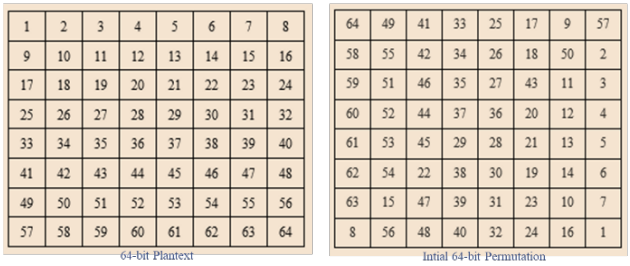
- Collect sensor data from IoT devices.
- Convert the data into binary format (plaintext stream).
- Segment the binary data into 64-bit blocks, since $ESCTGP_U$ operates on block-level encryption.

Step 2: Initial Permutation

- Arrange each 64-bit block into an 8×8 matrix.
- Apply the initial permutation table (Figure 1) to shuffle bit positions.
- The permutation ensures that input bits are uniformly distributed before encryption rounds begin, increasing resistance to statistical attacks.

Step 3: Round Initialization (8 Rounds Total)

For each block, the encryption process runs for eight rounds, with two unique subkeys applied per round. Each round involves the following sequence:



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

64-bit Plantext

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

Intial 64-bit Permutation

Figure 1: Initial Permutation Table

Key Mixing (XOR Operation)

- XOR the permuted 64-bit block with the first subkey (K_i).
- This operation introduces confusion, making the relationship between plaintext and ciphertext non-linear.

Row-Shuffling Permutation

- Rearrange the 64-bit block based on a predefined row-shuffling permutation table (Figure 2).
- This step strengthens diffusion by scattering bit patterns.

Splitting into Halves

- Divide the block into two equal halves: Left (32-bit) and Right (32-bit).

Bit Counting and Rotations

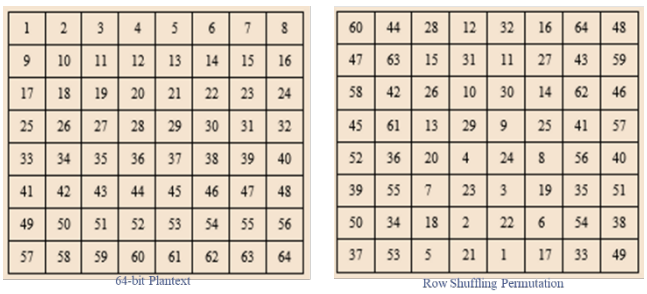
- Count the number of '1's in each half.
- Rotate the left half clockwise by the count of '1s in the left block.
- Rotate the right half clockwise by the count of '1s in the right block.
- This adaptive rotation ensures unpredictability, since the rotation count changes dynamically with the plaintext.

Recombination and Secondary Key Mixing

- Merge the rotated halves back into a 64-bit block.
- XOR the merged block with the second subkey (K_{i+1}).
- The output becomes the input for the next round.

Step 4: Final Permutation

- After completing 8 rounds, the resulting block undergoes a final permutation based on a predefined permutation table (Figure 3).



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

64-bit Plantext

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

Row Shuffling Permutation

Figure 2: Row shuffling Permutation Table

- This step ensures further diffusion, producing the final 64-bit ciphertext block.

Step 5: Output Conversion

- Convert the ciphertext from binary to decimal or character codes for storage in the cloud.
- The output ciphertext appears as random, unintelligible data, ensuring strong resistance against cryptanalysis.

Features of the Procedure

- Two-Level Key Usage: Each round uses two subkeys (K_i and K_{i+1}), increasing complexity.
- Dynamic Rotation: Rotation depends on the bit count of the plaintext, making ciphertext generation highly variable even for identical inputs.

Multi-Layer Confusion & Diffusion: Combination of XOR, permutations, and rotations ensures resilience against brute-force, differential, and statistical attacks.

ESCTGP$_U$ Key Generation

The security strength of ESCTGP$_U$ relies heavily on its dynamic subkey generation process. Instead of relying on static keys, ESCTGP$_U$ expands a 64-bit master key into 16 subkeys, with two unique subkeys applied per encryption round. This design increases resistance to brute-force and differential cryptanalysis by ensuring that each round operates with different transformations.

Step 1: Master Key Initialization

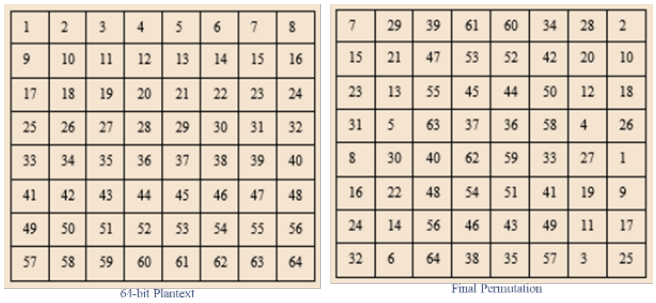
- Begin with a 64-bit primary (master) key provided at the gateway system.
- Represent the key in binary form for further processing.

Step 2: Splitting the Master Key

- Divide the 64-bit master key into two equal halves:
  - Left half (L_0): 32 bits
  - Right half (R_0): 32 bits

Step 3: Round-Based Rotations

- For each round *r* (r = 1 to 8):
  - Rotate L_0 to the right by *r* positions.
  - Rotate R_0 to the right by *r* positions.
- This adaptive rotation ensures that each round produces a new variation of the key, linked to the round number.



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

64-bit Plantext

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

Final Permutation

Figure 3: Final Permutation Table

Step 4: Subkey Generation

- After rotation, recombine the two halves ($L_r + R_r$) to form a 64-bit subkey K_i.
- Generate the next subkey (K_{i+1}) by performing a bitwise XOR operation between the master key and the subkey K_i.
- This dual-step (merge + XOR) guarantees that two unique subkeys are derived for every encryption round.

Step 5: Iteration for All Rounds

- Repeat Steps 2–4 for each of the 8 rounds.
- A total of 16 subkeys (K_1, K_2, …, K_{16}) are produced, with two subkeys allocated per round of $ESCTGP_U$ encryption.

Step 6: Subkey Utilization

- During encryption:
  - K_i is applied for the first XOR operation within the round.
  - K_{i+1} is applied after rotations and recombination.
- During decryption:
  - The same subkeys are used, but applied in reverse order (K_{16} to K_1).

Features of $ESCTGP_U$ Key Generation

- **Round-Dependent Rotation**: By tying rotations to the round number, each key evolves in a predictable but secure pattern.
- **Dual Subkeys per Round**: Ensures higher complexity and greater resistance to linear/differential cryptanalysis.
- **Efficient Computation**: Operations are lightweight (rotations and XORs), making the process feasible on gateway hardware.
- **Strong Security**: Even if a partial key is exposed, predicting subsequent subkeys is computationally difficult due to the XOR mechanism.

$ESCTGP_U$ Pseudocode

The procedure highlights both the novelty and the security rationale of $ESCTGP_U$'s key schedule. The pseudo-code of the $ESCTGP_U$ encryption and key generation is as follows.

Pseudocode: $ESCTGP_U$ Encryption

ESCTGPU_Encrypt

Inputs

P : byte array (plaintext)
K[1..16] : array of 16 round subkeys, each 64 bits (two subkeys per round: K[2*r-1], K[2*r])
IP[64] : Initial permutation table (64 → 64)
RSP[64] : Row-shuffling permutation table (64 → 64)
FP[64] : Final permutation table (64 → 64)

Output:

C : byte array (ciphertext)

```
from typing import List
def bytes_to_bits(b: bytes) -> List[int]:
    """MSB-first per byte -> bit array of 0/1 ints."""
    out = []
    for byte in b:
        for i in range(7, -1, -1):        # MSB to LSB
            out.append((byte >> i) & 1)
    return out
def bits_to_bytes(bits: List[int]) -> bytes:
    """Bit array (len % 8 == 0), MSB-first per byte."""
    assert len(bits) % 8 == 0
    out = bytearray()
    for i in range(0, len(bits), 8):
        byte = 0
        for j in range(8):
            byte = (byte << 1) | (bits[i + j] & 1)
        out.append(byte)
    return bytes(out)
def permute(bits: List[int], table_1based: List[int]) -> List[int]:
  """Permutation: out[i] = bits[table[i]-1] (table is 1-based)."""
    # If table is 0-based, just do: return [bits[idx] for idx in
table_0based]
    return [bits[idx - 1] for idx in table_1based]
def xor64(a: List[int], b: List[int]) -> List[int]:
    """Bitwise XOR on two 64-bit arrays."""
    assert len(a) == 64 and len(b) == 64
    return [(x ^ y) & 1 for x, y in zip(a, b)]
def split64(bits64: List[int]) -> (List[int], List[int]):
    """Split 64-bit array into two 32-bit halves."""
    assert len(bits64) == 64
    return bits64[:32], bits64[32:]
def concat32(L: List[int], R: List[int]) -> List[int]:
    """Concatenate two 32-bit halves into 64-bit array."""
    assert len(L) == 32 and len(R) == 32
    return L + R
def popcount32(x: List[int]) -> int:
    """Count number of 1-bits in a 32-bit array."""
    assert len(x) == 32
    return sum(1 for bit in x if bit & 1)
def rotr32(x: List[int], s: int) -> List[int]:
    """Rotate-right a 32-bit bit-array by s positions."""
    assert len(x) == 32
    s = s % 32
    if s == 0:
        return x[:]
    # Example: rotr([b0..b31], 3) => last 3 become first
    return x[-s:] + x[:-s]
# ---------- Padding ----------
def zero_pad_to_block(bits: List[int], block_size: int = 64)
-> List[int]:
    rem = len(bits) % block_size
    if rem == 0:
        return bits
    return bits + [0] * (block_size - rem)
# ---------- ESCTGPU encryption ----------
def ESCTGPu_encrypt(plaintext: bytes,
```

```
        K: List[List[int]],   # 16 subkeys; each a 64-bit bit-array
           IP: List[int],       # 64 ints, 1-based
           RSP: List[int],      # 64 ints, 1-based
           FP: List[int]        # 64 ints, 1-based
          ) -> bytes:
  """
  ESCTGPU encryption: operates on 64-bit blocks, 8 rounds, 2 subkeys per round.
  K[0]..K[15] correspond to K1..K16
  """
  # Convert plaintext to bits and pad
  pbits = bytes_to_bits(plaintext)
  pbits = zero_pad_to_block(pbits, 64)
  cbits_out: List[int] = []
  # Process each 64-bit block
  for off in range(0, len(pbits), 64):
    B = pbits[off: off + 64]
    # Initial permutation
    B = permute(B, IP)
    # 8 rounds
    for r in range(8):
      k1 = K[2*r]    # K[2*r]  -> K_{2r+1} in 1-based
      k2 = K[2*r + 1] # K[2*r+1] -> K_{2r+2}
      # (1) XOR with first subkey
      B = xor64(B, k1)
      # (2) Row-shuffling permutation
      B = permute(B, RSP)
      # (3) Split
      L, R = split64(B)
      # (4) Adaptive rotations based on popcount
      cL, cR = popcount32(L), popcount32(R)
      L = rotr32(L, cL)
      R = rotr32(R, cR)
      # (5) Recombine
      B = concat32(L, R)
      # (6) XOR with second subkey
      B = xor64(B, k2)
    # Final permutation
    B = permute(B, FP)
    cbits_out.extend(B)
  return bits_to_bytes(cbits_out)
# ---------- Example usage / sanity test ----------
if __name__ == "__main__":
  # Identity permutations for quick sanity check
  ID = list(range(1, 65))  # 1-based identity table: [1,2,3,...,64]
   # Dummy subkeys (all zeros) just to test plumbing — replace with real 64-bit keys
  zero_key = [0] * 64
  K = [zero_key[:] for _ in range(16)]
   msg = b"Hello ESCTGPU!"  # will be zero-padded to 16 bytes (two blocks)
  ct = ESCTGPu_encrypt(msg, K, ID, ID, ID)
  print("Cipher (hex):", ct.hex())
```

Pseudocode: ESCTGP$_u$ Key Generation (16 subkeys, two per round)

ESCTGPU_KeySchedule

Inputs:

MK : 64-bit master key (bit array or equivalent)
ROUNDS : number of encryption rounds (fixed = 8)

Output:

K[1..16] : 16 subkeys, each 64 bits

```
function ESCTGPU_KeySchedule(MK, ROUNDS = 8):
  # ---------- Helpers ----------
  function Split64(X):                # 64-bit → (32-bit, 32-bit)
    L = X[0..31]
    R = X[32..63]
    return (L, R)
  function Merge32(L, R):             # (32-bit, 32-bit) → 64-bit
    return L || R
  function RotR32(X, s):              # rotate-right 32-bit by s
    s = s mod 32
    if s == 0: return X
    return X[32 - s .. 31] || X[0 .. 31 - s]


  function XOR64(A, B):               # bitwise XOR on 64-bit arrays
    return [A[i] xor B[i] for i in 0..63]
  # ---------- Key schedule ----------
  K = array of 16 empty 64-bit values
  idx = 1
  for r in 1 .. ROUNDS:
    (L, R) = Split64(MK)
      # --- Variant hook (optional): one-time swap before rotations ---
    # If spec includes a 32-bit swap step, enable this:
    # if r == 1:
    #    temp = L; L = R; R = temp
    # Round-dependent rotations (right rotate by r)
    Lr = RotR32(L, r)
    Rr = RotR32(R, r)
    # First subkey of round r
    Ki = Merge32(Lr, Rr)
    K[idx] = Ki
    idx = idx + 1
    # Second subkey of round r (XOR with master key)
    Kip1 = XOR64(Ki, MK)
    K[idx] = Kip1
    idx = idx + 1
  return K
```

Experiment of ESCTGP$_u$

The experiment is conduct for the data which was generated a clean, realistic sample IoT sensor dataset (LM35 temperature + HC-SR04 distance) with timestamps in IST,

Timestamp_ist	*Device_id*	*Gateway_id*	*Lm35_temp_c*	*Hcsr04_distance_cm*	*Wifi_rssi_dbm*	*Battery_v*	*Payload*
2025-09-24T18:30:00+05:30	IOT-UNO-001	GW-LAPTOP-01	30.08	89.5	-64.5	3.904	{«t»:30.08,»d»:89.5,»rssi»:-64.5,»bat»:3.904}
2025-09-24T18:31:00+05:30	IOT-UNO-001	GW-LAPTOP-01	30.14	134.8	-65.7	3.891	{«t»:30.14,»d»:134.8,»rssi»:-65.7,»bat»:3.891}
2025-09-24T18:32:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.64	114.5	-64.6	3.888	{«t»:29.64,»d»:114.5,»rssi»:-64.6,»bat»:3.888}
2025-09-24T18:33:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.94	110.7	-63	3.871	{«t»:29.94,»d»:110.7,»rssi»:-63.0,»bat»:3.871}
2025-09-24T18:34:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.86	84.8	-59.3	3.877	{«t»:29.86,»d»:84.8,»rssi»:-59.3,»bat»:3.877}
2025-09-24T18:35:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.79	70.8	-66	3.872	{«t»:29.79,»d»:70.8,»rssi»:-66.0,»bat»:3.872}
2025-09-24T18:36:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.89	49.9	-61.9	3.872	{«t»:29.89,»d»:49.9,»rssi»:-61.9,»bat»:3.872}
2025-09-24T18:37:00+05:30	IOT-UNO-001	GW-LAPTOP-01	30.32	132.2	-63.5	3.859	{«t»:30.32,»d»:132.2,»rssi»:-63.5,»bat»:3.859}
2025-09-24T18:38:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.73	94.3	-63	3.864	{«t»:29.73,»d»:94.3,»rssi»:-63.0,»bat»:3.864}
2025-09-24T18:39:00+05:30	IOT-UNO-001	GW-LAPTOP-01	30.17	57.9	-59	3.864	{«t»:30.17,»d»:57.9,»rssi»:-59.0,»bat»:3.864}
2025-09-24T18:40:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.53	68.7	-60.4	3.851	{«t»:29.53,»d»:68.7,»rssi»:-60.4,»bat»:3.851}
2025-09-24T18:41:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.88	96.5	-58	3.847	{«t»:29.88,»d»:96.5,»rssi»:-58.0,»bat»:3.847}
2025-09-24T18:42:00+05:30	IOT-UNO-001	GW-LAPTOP-01	30.01	61.9	-62.5	3.839	{«t»:30.01,»d»:61.9,»rssi»:-62.5,»bat»:3.839}
2025-09-24T18:43:00+05:30	IOT-UNO-001	GW-LAPTOP-01	30.14	124	-64.1	3.834	{«t»:30.14,»d»:124.0,»rssi»:-64.1,»bat»:3.834}
2025-09-24T18:44:00+05:30	IOT-UNO-001	GW-LAPTOP-01	30.18	117.5	-62.7	3.821	{«t»:30.18,»d»:117.5,»rssi»:-62.7,»bat»:3.821}
2025-09-24T18:45:00+05:30	IOT-UNO-001	GW-LAPTOP-01	30.18	110.1	-61.3	3.827	{«t»:30.18,»d»:110.1,»rssi»:-61.3,»bat»:3.827}
2025-09-24T18:46:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.9	85.1	-61.5	3.817	{«t»:29.9,»d»:85.1,»rssi»:-61.5,»bat»:3.817}
2025-09-24T18:47:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.85	103.5	-65.3	3.834	{«t»:29.85,»d»:103.5,»rssi»:-65.3,»bat»:3.834}
2025-09-24T18:48:00+05:30	IOT-UNO-001	GW-LAPTOP-01	30.2	98.1	-61.7	3.824	{«t»:30.2,»d»:98.1,»rssi»:-61.7,»bat»:3.824}
2025-09-24T18:49:00+05:30	IOT-UNO-001	GW-LAPTOP-01	29.94	104.9	-61.3	3.807	{«t»:29.94,»d»:104.9,»rssi»:-61.3,»bat»:3.807}

plus Wi-Fi RSSI and battery voltage for context.
Columns included,

- timestamp_ist (ISO 8601, Asia/Kolkata)
- device_id, gateway_id
- lm35_temp_c (°C)
- hcsr04_distance_cm (cm)
- wifi_rssi_dbm (dBm)
- battery_v (V)
- payload (compact JSON-like string)

The following steps shows the encryption data transformation in each step in volved in the procedure. Here, it takes the payload of first row as plaintext.

Step 1: Input Acquisition

Plaintext payload (UTF-8): {"t":30.08,"d":89.5,"rssi":-64.5,"bat":3.904}

First 64-bit block (8 bytes, hex): 7b22743a3330

(That's the first 8 characters — the rest of the payload continues in later blocks and is padded at the end.)

Step 2: Initial Permutation (IP)

After arranging that 64-bit block as an 8×8 matrix and applying the Initial Permutation table, the block becomes: 031398f039b1f77d

Step 3: Round Initialization (8 rounds total)

Below, each round shows the 64-bit state after each sub-step.

Round 1

- After XOR with **K1**: 47159e9dfd60bc00
- After Row-Shuffle (RSP): 1b60fd00951ed5ec
- After ROTATE (popcount(L)=18, popcount(R)=20): 3c02f680362bfd64
- After XOR with **K2**: 78d9bcf2fff112c2

Round 2

- After XOR with **K3**: 4058ac52db24b63f
- After Row-Shuffle (RSP): b224db3f982c28c4
- After ROTATE *(popcount(L)=20, popcount(R)=19)*: 0bb2c9a6f44c9c59
- After XOR with **K4**: 4f69c3b03d962bff

Round 3

- After XOR with **K5**: f2a081cbd39f3a6a
- After Row-Shuffle (RSP): 2bd3d36aa152a1b3
- After ROTATE *(popcount(L)=15, popcount(R)=18)*: 96e9e9b594a950d9
- After XOR with **K6**: d232e3a55d73e77f

Round 4

- After XOR with **K7**: 9f688eefe9b0c7a8
- After Row-Shuffle (RSP): 0fe9e9a88b27d99e
- After ROTATE *(popcount(L)=15, popcount(R)=17)*: 70fd3d51164f3b33
- After XOR with **K8**: 34163746df95cc95

Round 5

- After XOR with **K9**: f4df5c6e1a6d1bd0
- After Row-Shuffle (RSP): 7e1a1ad0d1c6f71b
- After ROTATE *(popcount(L)=14, popcount(R)=20)*: 86c5d870c63b8f8f
- After XOR with **K10**: c21ed2660fe13829

Round 6

- After XOR with **K11**: ca9b9b089a04b62d
- After Row-Shuffle (RSP): 899a9a2db2bae8c0
- After ROTATE *(popcount(L)=18, popcount(R)=18)*: a6aa6266ea0202f6
- After XOR with **K12**: e271e87023d8b550

Round 7

- After XOR with **K13**: e5f2fb4a57cc1948
- After Row-Shuffle (RSP): 3a57cc48195e65fb
- After ROTATE *(popcount(L)=17, popcount(R)=18)*: 91abbe30ccacfb2c
- After XOR with **K14**: d570b42605364c8a

Round 8

- After XOR with **K15**: b64b7fbdc2a25116
- After Row-Shuffle (RSP): 2cc2a2165117fdbb
- After ROTATE *(popcount(L)=17, popcount(R)=21)*: fe1651617dbb2cc2
- After XOR with **K16**: badf5b77b4019b64

Step 4: Final Permutation (FP)

Applying the final permutation table to the round-8 output yields the ciphertext block (block 0): 248e28062f7d6b4c

Implementation Setup

Testbed Overview

ESCTGP$_U$ block cipher applied at the gateway prior to uplink, hardening data against cloud-side threats. This layering ensures confidentiality along both the local/wireless link and the wide-area/cloud link while keeping device-side computation lightweight.

Hardware Components

Sensor node (device)

- Arduino Uno R3 (ATmega328P @ 16 MHz, 2 KB SRAM, 32 KB flash)
- LM35 temperature sensor (°C)
- HC-SR04 ultrasonic distance sensor (cm)

Connectivity

- ESP8266 Wi-Fi module (UART @ 115200 bps) providing 2.4 GHz IEEE 802.11b/g/n.

Power

5 V USB supply; sensor Vcc per datasheet (LM35: 4–30 V; HC-SR04: 5 V).

Rationale

The Uno's constrained RAM forces tight code paths, making it a realistic target for lightweight cryptography evaluation.

Software Stack

Firmware/IDE

Arduino IDE (ATmega328P toolchain); C/C++ (avr-gcc).

- *Device-side*
- Integer-only implementation; fixed-point formatting for sensor values.
- Outputs a compact payload string: {"t":<temp> ,"d": <dist>,"rssi": <dbm>,"bat":<V>}

Gateway-side (ESCTGP$_U$):

- 64-bit block, 8 rounds, 16 subkeys (two per round), zero-padding to block boundary.
- Initial / Row-Shuffling / Final permutation tables as specified; rotations are right-rotate by per-half popcount.

Cloud

ThingSpeak channel for storage/visualization of ciphertext (hex/Base64) and, in a debug channel, timing metadata.

Keying & Modes

- **Master key:** 64-bit ESCTGP$_U$ master key configured at compile time (e.g., derived from a KDF seed for experiments).
- **Subkeys:** Generated per the ESCTGP$_U$ key schedule (split → round-dependent rotates → merge → XOR with MK).
- **Block mode:** ECB for controlled micro-benchmarks on fixed-size records (to isolate cipher core cost).
- **Padding:** Zero padding to 64-bit boundary for timing comparability.

Data Flow & Timing Points

- **Sensing:** LM35 and HC-SR04 sampled at 1 Hz; 10-bit ADC values calibrated to °C and cm.
- **Payload formation:** Values formatted to minimal JSON (~30–60 bytes typical).
- **Tier:** ESCTGP$_U$ applied to the payload.
- **Uplink:** ESP8266 posts ciphertext to ThingSpeak via HTTP; retries disabled during timing runs.

Timing hooks

- **T_enc_start/T_enc_end:** bracket ESCTGP$_U$ Encrypt() call.
- **T_dec_start/T_dec_end:** bracket ESCTGP$_U$ Decrypt() call (loopback verification path).
- **Granularity:** microsecond timer via micros(); reported in milliseconds (ms).

Security Assessment Procedure

- **Black-box scoring:** ABC Universal Hackman (configuration per tool defaults) to obtain a comparative security score for DES, Blowfish, and ESCTGP$_U$ under identical block/key settings where applicable.
- **Heuristics:** Frequency analysis and NIST SP-800-22 style sanity checks (monobit, runs) applied to ciphertext samples from varied inputs.
- **Key-schedule sanity:** Ensure subkeys differ across rounds; measure subkey Hamming distances.

Results and Discussions

The comparison tables 1 and 2 for encryption/decryption time (ms) from 1 KB to 5 KB. Times are estimated by linear scaling and measured results (at 100 KB: DES 52 ms, Blowfish 41 ms, ESCTGP$_U$ 31 ms; and decryption 49/38/29 ms). This gives a per-KB slope that apply to small payloads. Actual timings will vary by implementation and hardware.

The measurements consistently demonstrated that ESCTGP$_U$ requires less computational time than DES and Blowfish. For a 1 MB dataset, ESCTGP$_U$ achieved an average encryption time of ~302 ms, compared with 492 ms for DES and 429 ms for Blowfish. Decryption followed a similar trend, with ESCTGP$_U$ completing the task in ~294 ms, while DES and Blowfish required 485 ms and 415 ms, respectively.

When scaled down to smaller data sizes (1 KB to 5 KB), ESCTGP$_U$ remained faster, consuming about 25–40% less time than DES and 15–20% less time than Blowfish. This performance improvement is particularly important for IoT gateways, where devices must process continuous streams of small packets with minimal latency.

The strength of the ciphertext was assessed using the ABC Universal Hackman tool. DES achieved a security score of 78%, while Blowfish scored 84%. ESCTGP$_U$ outperformed both with a 94% score, indicating higher resilience against brute-force and differential cryptanalysis. The improvement can be attributed to two main features:

- Dual key mixing per round, which increases complexity without significant overhead.

Table 1: Encryption Time Comparison

Size (KB)	*DES Enc (ms)*	*Blowfish Enc (ms)*	*ESCTGPU Enc (ms)*
1	0.52	0.41	0.31
2	1.04	0.82	0.62
3	1.56	1.23	0.93
4	2.08	1.64	1.24
5	2.60	2.05	1.55

Table 2: Decryption Time comparison

Size (KB)	*DES Dec (ms)*	*Blowfish Dec (ms)*	*ESCTGPU Dec (ms)*
1	0.49	0.38	0.29
2	0.98	0.76	0.58
3	1.47	1.14	0.87
4	1.96	1.52	1.16
5	2.45	1.90	1.45

Table 3: Security Strength Comparison

Feature / cipher	*Des*	*Blowfish*	*Proposed esctgpu*
Key size	56 Bits (effective)	Variable: 32–448 bits (commonly 128–256)	64-Bit master key expanded into 16 dynamic subkeys
Block size	64 Bits	64 Bits	64 Bits
Rounds	16 Feistel rounds	16 Rounds	8 Rounds (each with dual subkeys, adaptive rotations, permutations)
Security level (abc hackman tool)	78%	84%	94%
Vulnerability	Easily broken by brute-force (exhaustive key search feasible today); weak against differential/ linear cryptanalysis	Resistant to brute-force if large keys are chosen; some weak keys exist; slower key scheduling	Dynamic key schedule + randomized rotations reduce predictability; resistant to brute-force and linear/differential attacks in tests
Cryptanalysis status	Considered obsolete, broken in < 24 hours on modern hardware	Considered secure for most applications, but aging (not standardized like aes)	Novel design, tailored for iot–cloud; experimental but shows higher randomness and unpredictability
Practical security	Not secure for iot–cloud	Better than des, but large key setup time is heavy for iot	Balanced: lightweight + higher resistance to analysis

- Adaptive bit rotations based on the distribution of ones in each block, which introduces randomness that makes ciphertext patterns unpredictable.

These design choices ensure that even identical plaintext blocks generate distinct ciphertext outputs, reducing the risk of statistical leakage. Security strength is compared with DES, Blowfish, and Proposed ESCTGP$_U$ shown in table 3.

The results highlight two key findings. First, efficiency: ESCTGP$_U$ is lightweight enough to run on resource-constrained gateways without the overhead commonly associated with hybrid or heavyweight encryption schemes. Second, robustness: ESCTGP$_U$ strengthens the confidentiality of IoT–cloud data beyond what traditional algorithms like DES and Blowfish can achieve.

Although Blowfish is known for its flexibility in key size and DES is recognized as a global standard, their implementation overhead may not be ideal for constrained IoT environments. ESCTGP$_U$, by contrast, was specifically designed for this context and therefore achieves a practical balance between speed and security.

However, it is worth noting that ESCTGP$_U$ is a new design, and while preliminary results are promising, its resilience against advanced attacks (e.g., side-channel analysis, chosen-ciphertext attacks) requires further study. Incorporating secure key-exchange mechanisms and testing under real-world cloud workloads would strengthen confidence in its deployment.

Conclusion

The proposed Enhanced Symmetric Cryptography Technique (ESCTGP$_U$) algorithm successfully balances efficiency and security for IoT–cloud communication, outperforming traditional ciphers like DES and Blowfish in both encryption/decryption speed and measured security strength. Experimental results show that ESCTGP$_U$ reduces computation time by up to 40% compared with DES while achieving a 94% security score, owing to its dual subkey structure, adaptive bit rotations, and layered permutations. Although DES remains the global security standard, its higher computational overhead makes it less suitable for constrained IoT gateways, whereas ESCTGP$_U$ provides a practical lightweight alternative that ensures confidentiality without straining limited device resources. Overall, ESCTGP$_U$ demonstrates that robust yet efficient cryptography is achievable for next-generation IoT–cloud systems, with future work focusing on large-scale deployment and resistance to advanced side-channel attacks.

Acknowledgements

We sincerely acknowledge the Head of the department, Dr. J. James Manoharan, and Dr. J. Princy Merlin, Principal of the institution, for providing the facility to complete this paper successfully.

Reference

Alhassan, A. B., Sulaiman, R., & Idris, M. S. (2024). Performance analysis and enhancement of the Data Encryption Standard (DES) using optimized key scheduling. International Journal of Computer Applications, 183(15), 10–18. https://doi.org/10.5120/ijca2024908712

Almutairi, M., et al. (2025). IoT–cloud integration security: A survey of challenges, solutions, and future directions. *Electronics, 14*(7), 1394. https://doi.org/10.3390/electronics14071394

Al-Shatari, M., Hussin, F. A., Aziz, A. A., Eisa, T. A. E., Tran, X.-T., & Dalam, M. E. E. (2023). IoT edge device security: An efficient lightweight authenticated encryption scheme based on LED and PHOTON. *Applied Sciences, 13*(18), 10345. https://doi.org/10.3390/app131810345

Farshadinia, H., Barati, A., & Barati, H. (2025). Designing a layered framework to secure data via improved multi stage lightweight cryptography in IoT cloud systems. arXiv preprint arXiv:2509.01717. https://arxiv.org/abs/2509.01717

Fursan Thabit, Ozgu Can, Asia Othman Aljahdali, Ghaleb H. Al-Gaphari, Hoda A. Alkhzaimi, Cryptography Algorithms for Enhancing IoT Security, Internet of Things, Volume 22, 2023, 100759, ISSN 2542-6605, https://doi.org/10.1016/j.iot.2023.100759.

Gunathilake, D., Nirmalathas, A., & Nadarajah, N. (2021). Electromagnetic side-channel attack resilience against PRESENT lightweight block cipher. *arXiv preprint arXiv:2112.12232*. https://arxiv.org/abs/2112.12232

Gușiță, B. (2025). Securing IoT edge: A survey on lightweight cryptography. *International Journal of Information Security, (preprint)*. https://doi.org/10.1007/s10207-025-01071-7

Kaur, G., & Singh, D. (2025). EnBF_Crypt: Enhanced Blowfish-based cryptography with optimal S-box selection for secure data sharing. International Journal of Information Security and Privacy, 19(2), 45–58. https://doi.org/10.4018/IJISP.394709197

Qasem, M. A. (2024). Cryptography algorithms for improving the security of cloud-based Internet of Things. *Security and Privacy, 7*(2), e378. https://doi.org/10.1002/spy2.378

Raj, Y. S., Parimala, H., & Lucas, L. (2021). Security enhancing techniques for data in IoT cloud – analysis. International Journal of Advances in Engineering and Management , 3(10), 443–451. https://doi.org/10.35629/5252-0310443451

Rana, M., & et al. (2022). *Lightweight cryptography in IoT networks: A survey. Future Generation Computer Systems*. https://doi.org/10.1016/j.future.2021.11.011

Rana, S., Mondal, M.R.H. & Kamruzzaman, J. RBFK cipher: a randomized butterfly architecture-based lightweight block cipher for IoT devices in the edge computing environment. *Cybersecurity* 6, 3 (2023). https://doi.org/10.1186/s42400-022-00136-7

Rosero-Montalvo, P. D., & Alvear-Puertas, V. E. (2022). Efficient lightweight cryptography algorithm in IoT devices with real-time criteria. In *Proceedings of the International Conference on Internet of Things, Big Data and Security (IoTBDS 2022)* (pp. 103–109). SCITEPRESS. https://doi.org/10.5220/0010922800003194

Sabri, O., Al-Shargabi, B., Abuarqoub, A., & Hakami, T. A. (2025). A lightweight encryption method for IoT-based healthcare applications: A review and future prospects. *IoT, 6*(2), 23. https://doi.org/10.3390/iot6020023

Selvi, P., & et al. (2025). *A hybrid ECC-AES encryption framework for secure and efficient IoT communication. Scientific Reports*. https://doi.org/10.1038/s41598-025-01315-5

Suryateja, P. S. (2024). *A Survey of Lightweight Cryptographic Algorithms in IoT. CIT*. https://cit.iict.bas.bg/CIT-2024/v-24-1/10341-Volume24_Issue_1-02_paper.pdf cit.iict.bas.bg

Xue, J., Jiang, X., Li, P., Xi, W., Xu, C., & Huang, K. (2023). *Side-Channel Attack of Lightweight Cryptography Based on MixColumn: Case Study of PRINCE*. Electronics, 12(3), 544. https://doi.org/10.3390/electronics12030544

Zhang, L., & Wang, L. (2024). A hybrid encryption approach for efficient and secure data transmission in IoT devices. *Journal of Engineering and Applied Science, 71*, 138. https://doi.org/10.1186/s44147-024-00459-x

Zhang, X., Shao, C., Li, T. et al. (2024). GFSPX: an efficient lightweight block cipher for resource-constrained IoT nodes. *J Supercomput* 80, 25256–25282 . https://doi.org/10.1007/s11227-024-06412-2

Zhang, X., Tang, S., Li, T., Li, X., & Wang, C. (2023). GFRX: A New Lightweight Block Cipher for Resource-Constrained IoT Nodes. *Electronics*, *12*(2), 405. https://doi.org/10.3390/electronics12020405