Synergy: A Lightweight Block Cipher with Variable Bit Rotation Feistel Network

Anders Lindman

Independent Researcher anders_lindman@yahoo.com

Abstract. Synergy is a lightweight block cipher designed for resource-constrained environments such as IoT devices, embedded systems, and mobile applications. Built around a 16-round Feistel network, 8 independent pseudorandom number generators (PRNGs) ensure strong diffusion and confusion through the generation of per-block unique round keys. With a 1024-bit key and a 64-bit block size, Synergy mitigates vulnerabilities to ML-based cryptanalysis by using a large key size in combination with key- and data-dependent bit rotations, which reduce statistical biases and increase unpredictability. By utilizing 32-bit arithmetic for efficient processing, Synergy achieves high throughput, low latency, and low power consumption, providing performance and security for applications where both are critical.

1 Introduction

Symmetric encryption is a cryptographic method that uses the same key for both encryption and decryption, securing data across various applications. Lightweight versions have the potential to protect resource-constrained devices such as IoT, edge computing, 5G networks, mobile devices, and RFID tags with minimal computational overhead.

This paper introduces Synergy, a new cipher designed to efficiently secure data in lightweight applications.

1.1 The Synergy Cipher

The growing demand for secure cryptographic solutions in resource-constrained environments necessitates lightweight block ciphers that balance security and performance. Such environments often have limited computational power, memory, and energy resources, making efficiency a critical design factor. Simultaneously, advancements in quantum computing and AI-driven cryptanalysis present increasing threats to traditional key sizes, potentially requiring larger keys for long-term security. Current lightweight ciphers often struggle to meet these dual requirements of strong security and high efficiency, offering either insufficient security or suboptimal performance. Synergy addresses these challenges by combining a 1024-bit key with computational efficiency and a low memory footprint.

The name *Synergy* reflects the cryptographic synergy achieved by integrating high-speed yet cryptographically weak pseudorandom number generators

(PRNGs) with a lightweight Feistel network to construct a robust block cipher. Rather than relying on the inherent strength of the PRNGs, Synergy leverages structure, dynamic round transformations, and high-entropy key material to collectively obscure internal states and mitigate statistical leakage, even under emerging threats such as ML-based cryptanalysis. This design enables the use of a 1024-bit key while maintaining high throughput and low latency, making it well-suited for resource-constrained environments with high demands for both performance and security. The strength of this approach lies in the round function of the Feistel network, which incorporates variable bitwise rotations to obscure the internal states of eight independently seeded PRNGs. By altering the transformation of these states dynamically on a per-block basis, the cipher mitigates statistical biases and pattern predictability in the resulting ciphertexts.

2 Background and Related Work

2.1 Block Ciphers Overview

A **block cipher** is a symmetric-key cryptographic algorithm that encrypts data in fixed-size blocks (e.g., 64-bit, 128-bit). It applies a series of mathematical transformations to produce ciphertext using a secret key. Block ciphers are widely used in secure communication, file encryption, and authentication [1].

How Block Ciphers Work - Encryption: Plaintext is transformed into ciphertext using the secret key and the cipher's algorithm. - **Decryption**: Ciphertext is transformed back into plaintext using the same secret key and the inverse of the cipher's algorithm. - **Modes of Operation**: To handle data larger than a single block, block ciphers use modes of operation (e.g., CBC, CTR, GCM).

2.2 Related Work

Lightweight cryptographic algorithms have gained prominence due to their suitability for resource-constrained environments. Notable examples include:

– **Ascon**: Selected by NIST for its Lightweight Cryptography standardization project, Ascon combines encryption and message authentication in a single operation [2].
– **Speck**: Developed by the NSA, Speck supports various block and key sizes, offering flexibility for constrained environments [3].
– **PRESENT**: Designed in 2007, PRESENT operates on 64-bit blocks and supports 80- or 128-bit keys, providing compact design and resistance to cryptanalysis [4].

3 Novelty and Contributions

Synergy introduces the following novelties:

- A 1024-bit key for a lightweight block cipher.
- Data- and round-key-dependent bit rotations in a Feistel network.
- The use of fast pseudorandom number generators (PRNGs) combined to provide security.
- Per-round and per-block unique round keys generated with PRNGs.
- A lightweight key schedule for high-performance encryption with large keys.

The use of data-dependent bit rotations in cryptographic design has been explored before, as demonstrated by IBM in their design of MARS [5], an AES finalist. In Synergy, the novel aspect is how variable bit rotations depend on both the data and the pseudorandom round keys to achieve a strong avalanche effect that is unique for each block.

4 Design Principles of Synergy

4.1 General Structure

Synergy utilizes a 16-round Feistel network with a 64-bit block size and a 1024-bit key. Eight independent Xorshift128 [6] PRNGs generate unique round keys for each round and block. While these PRNGs are vulnerable in isolation, their combination and obfuscation within the Feistel network ensure strong diffusion and confusion, effectively securing both the plaintext and the round keys.

4.2 Xorshift PRNGs

To mitigate the known limitations of xorshift generators, particularly the issue of short periods highlighted by Panneton and L'Ecuyer [10], Synergy combines eight independent xorshift generators. Each generator has a period of $2^{128} - 1$, significantly extending the shorter periods mentioned and reducing the risk of repeating key sequences. This aligns with recommendations to use "a larger number of xorshifts together with a long period" to improve the statistical properties of the generated sequences. However, the long period alone is not sufficient for cryptographic security. The crucial non-linear mixing within the dynamic key schedule, combined with the diffusion provided by the Feistel network through variable bit rotations, mitigates the inherent linearity of the xorshift generators and enhances their statistical properties for cryptographic applications. The design with independently seeded PRNG instances provides a large key space, in preparation for the rapid advancements in cryptanalysis [7].

4.3 Round Function

The round function F for encryption in Synergy is defined as:

$$F(R_{i+1}, K_i) = K_i \oplus \mathrm{ROL}(R_i, (R_i\ 0x1F)) \tag{1}$$

Here, $\mathrm{ROL}(R_i, (R_i \& 0x1F))$ represents a left rotation of R_i by a number of bits determined by its lower five bits. This operation ensures entropy preservation across rounds, improving diffusion and obscuring relationships between plaintext and ciphertext. The bitwise rotation in the round function depends on the value of the left half block, making the rotation variable and hard to predict.

4.4 Number of Rounds

Our analysis suggests that 8 rounds provide moderate diffusion, which may resist some cryptanalytic techniques but could be vulnerable to advanced attacks. At 16 rounds, diffusion approaches an ideal level, where small changes in input lead to nearly 50% of output bits flipping on average. Doubling the number of rounds from 8 to 16 significantly increases the probability of single bit flips in plaintext data affecting the variable bit rotations, making it substantially more difficult for future attackers to analyze and exploit potential weaknesses.

In addition to enhancing diffusion, increasing the number of rounds plays a crucial role in confusion. Each round of the Feistel network mixes the round key with the data. With more rounds, the data undergoes this mixing process multiple times with different round keys, making it progressively more difficult for an attacker to work backward from the ciphertext to recover the round keys or the master key. The increased complexity effectively "hides" the round keys, making cryptanalytic attacks that rely on key recovery significantly more challenging.

Table 1 shows the average Hamming distance for one million ciphertext pairs, where one bit at a pseudorandom position in the plaintext differs.

Table 1. Average Hamming Distance for Different Rounds (1,000,000 runs)

Number of Rounds	Average Hamming Distance	Standard Deviation
8	30.889494	5.512236
16	32.001687	3.992993
24	32.002514	3.995475
32	32.003161	4.006110

4.5 Block Size

Synergy's block size of 64 bits enables efficient processing of half-blocks as single 32-bit integers in the Feistel network. This design enhances performance on 32-bit architectures while minimizing memory usage, making it suitable for resource-constrained devices.

The security of the small block size is strengthened in Synergy by the generation of pseudorandom unique round keys for each block with a period of $2^{128} - 1$ (block period of $2^{127} - 1$ since two pseudorandom numbers per PRNG are used per block), thus mitigating the weaknesses generally associated with small block sizes when fixed key schedules are used.

4.6 Key Derivation

The use of a 1024-bit key significantly increases security by mitigating risks associated with smaller key sizes. The key size aligns with the number of Xorshift128 instances in Synergy, where each instance contains four 32-bit integers as internal state, totaling $8 \times 4 \times 32 = 1024$ bits. This key size ensures that all PRNG instances can be directly seeded from the key and IV without entropy loss.

The 1024-bit IV length matches the key size, providing sufficient entropy for key derivation and enabling robust mixing of the key and IV. This enhances the security of the derived keys and prevents attacks such as IV reuse, where repeated IVs can compromise confidentiality.

The key derivation function takes two inputs:

– `IV[8]`: An 8-element array of 64-bit integers representing an initialization vector.
– `key[8]`: An 8-element array of 64-bit integers representing a cryptographic key.

The derived key is $key \oplus IV$, ensuring low-latency key derivation. However, reusing keys across multiple encryptions can increase vulnerability to cryptanalysis, making ephemeral keys essential for security.

Secure key management strategies, such as a two-tiered key system using ephemeral keys—e.g., derived from a smaller key—can be adopted based on implementation needs.

5 Security Considerations

While Synergy incorporates several design features to enhance security, a comprehensive cryptanalytic evaluation is beyond the scope of this paper. A full cryptanalytic evaluation is necessary to establish the security of Synergy for real-world applications, including resistance against machine learning models (ML models) [9].

However, some aspects of Synergy's security have been considered. The cipher utilizes eight independent Xorshift128 PRNGs, which are known to have weaknesses [10]. The Feistel structure, with its data- and key-dependent rotations, is expected to provide significant diffusion and confusion, potentially mitigating these weaknesses. The key-dependent rotation, where the rotation amount depends on the lower 5 bits of the right half of the block (representing all possible bit positions in the 32-bit block half), introduces nonlinearity and makes it more difficult to predict the internal states of the PRNGs. The hypothesis is that this dynamic rotation in combination with 16 rounds in the Feistel network significantly complicates attacks that rely on predicting the PRNG outputs.

ML models are expected to become increasingly capable of predicting PRNGs [11], and even with data- and key-dependent variable bit rotations in the Feistel network, patterns may be leaked in the ciphertexts that ML models can use for prediction. In anticipation of possibilities like these, Synergy uses eight independently seeded PRNGs for generating unique round keys for each round and for each block.

The 1024-bit key in Synergy provides entropy for seeding all eight Xorshift128 generators. Synergy's design prioritizes lightweight operations, including key derivation, to achieve minimal latency and suitability for resource-constrained environments. This focus on lightweight key derivation, however, necessitates careful key management practices. Specifically, we recommend the use of ephemeral keys, where a unique key is generated for each encryption session.

6 Performance Evaluation

All calculations in the Synergy algorithm use 32-bit arithmetic with only bitwise operations. Therefore, Synergy is expected to have high performance and low power consumption in both software implementations running on low-power 32-bit CPUs and hardware implementations.

7 Conclusion

With a novel approach to symmetric encryption, Synergy, a lightweight block cipher has been presented that enables high throughput, low latency and low power consumption while at the same time allowing for a large key size. Synergy achieves robust security and efficiency through the use of fast PRNGs in combination with variable bit rotations in a Feistel network, making it well-suited for lightweight cryptographic applications.

Future work may focus on evaluating the security of the cipher and how it can be implemented on different platforms.

Acknowledgments

We thank the contributors and reviewers for their valuable feedback.

References

1. Douglas R. Stinson. *Cryptography: Theory and Practice*. Chapman and Hall/CRC, 3rd edition, 2006.
2. Christoph Dobraunig, Maria Eichlseder, Felix Mendel, and Martin Schläffer. *Ascon v1.2*. Submission to the NIST Lightweight Cryptography Standardization Process, 2019.
3. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, pp. 1–6, June 2015.
4. A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pp. 450–466. Springer, 2007.
5. R. G. Jutla, S. M. Matyas Jr, L. O. Peyravian, and D. S. Zunic. MARS–a candidate cipher for AES.
6. George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8:1–6, 2003.
7. D. Gowda, J. Garg, S. Garg, K. D. V. Prasad, and S. Suneetha. Future Outlook: Synergies Between Advanced AI and Cryptographic Research. In *Innovative Machine Learning Applications for Cryptography*, pp. 27–46. IGI Global, 2024.
8. Melissa E. O'Neill. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. 2014.
9. T. R. Lee, J. S. Teh, N. Jamil, J. L. S. Yan, and J. Chen. Lightweight block cipher security evaluation based on machine learning classifiers and active S-boxes. *IEEE Access*, 9:134052–134064, 2021.
10. François Panneton and Pierre L'Ecuyer. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 15(4):346–361, 2005.
11. A. Singh, K. B. Sivangi, and A. N. Tentu. Machine learning and cryptanalysis: An in-depth exploration of current practices and future potential. *Journal of Computing Theories and Applications*, 1(3):257–272, 2024.
12. C. Bouillaguet, F. Martinez, and J. Sauvage. Practical seed-recovery for the PCG pseudo-random number generator. *IACR Transactions on Symmetric Cryptology*, 2020.

A Appendix: Pseudocode for Synergy

Below is the pseudocode for Synergy, providing a high-level description of its algorithms:

Algorithm 1 Global Constants and Arrays

```
1: Define $NUM\_ROUNDS = 16$
2: Define $NUM\_PRNGS = 8$
3: Define $PRNGS\_MODULO = NUM\_PRNGS - 1$
4: Global Arrays: $x[NUM\_PRNGS]$, $y[NUM\_PRNGS]$,
   $z[NUM\_PRNGS]$, $w[NUM\_PRNGS]$, $rk[NUM\_ROUNDS]$
```

Algorithm 2 Initialize Seeds

```
Require: $key[]$ and $iv[]$ are arrays of size at least $NUM\_PRNGS \times 4$
Ensure: Global arrays $x[]$, $y[]$, $z[]$, $w[]$ are initialized based on $key[]$ and $iv[]$
1: function INITIALIZESEEDS($key[]$, $iv[]$)
2:     for $i = 0$ to $NUM\_PRNGS - 1$ do
3:         $x[i] \leftarrow key[i] \oplus iv[i]$
4:         $y[i] \leftarrow key[i + NUM\_PRNGS] \oplus iv[i + NUM\_PRNGS]$
5:         $z[i] \leftarrow key[i + NUM\_PRNGS \times 2] \oplus iv[i + NUM\_PRNGS \times 2]$
6:         $w[i] \leftarrow key[i + NUM\_PRNGS \times 3] \oplus iv[i + NUM\_PRNGS \times 3]$
7:     end for
8: end function
```

Algorithm 3 XOR128 Function

```
Require: $i$ is an integer such that $0 \leq i < NUM\_PRNGS$
Ensure: A pseudo-random number is generated and returned
1: function XOR128($i$)
2:     $t \leftarrow x[i] \oplus (x[i] \ll 11)$
3:     $x[i] \leftarrow y[i]$
4:     $y[i] \leftarrow z[i]$
5:     $z[i] \leftarrow w[i]$
6:     $w[i] \leftarrow w[i] \oplus (w[i] \gg 19) \oplus t \oplus (t \gg 8)$
7:     return $w[i]$
8: end function
```

Algorithm 4 Feistel Encryption

```
Require: $plaintext[]$ contains valid data, $blockIndex$ is within bounds, global arrays are initialized
Ensure: Encrypted ciphertext is stored in $ciphertext[blockIndex]$ and $ciphertext[blockIndex + 1]$
 1: function feistelEncrypt($plaintext[]$, $ciphertext[]$, $blockIndex$)
 2:     $left \leftarrow plaintext[blockIndex]$
 3:     $right \leftarrow plaintext[blockIndex + 1]$
 4:     for $i = 0$ to $NUM\_ROUNDS - 1$ do
 5:         $rk[i] \leftarrow xor128(i \& PRNGS\_MODULO)$
 6:     end for
 7:     for $i = 0$ to $NUM\_ROUNDS - 1$ do
 8:         $temp \leftarrow right$
 9:         $n \leftarrow right \wedge 0x1f$
10:         $right \leftarrow left \oplus rk[i] \oplus ((right \ll n) \vee (right \gg (32 - n)))$
11:         $left \leftarrow temp$
12:     end for
13:     $ciphertext[blockIndex] \leftarrow left$
14:     $ciphertext[blockIndex + 1] \leftarrow right$
15: end function
```

Algorithm 5 Feistel Decryption

```
Require: $ciphertext[]$ contains valid data, $blockIndex$ is within bounds, global arrays are initialized
Ensure: Decrypted plaintext is stored in $plaintext[blockIndex]$ and $plaintext[blockIndex + 1]$
 1: function feistelDecrypt($ciphertext[]$, $plaintext[]$, $blockIndex$)
 2:     $left \leftarrow ciphertext[blockIndex]$
 3:     $right \leftarrow ciphertext[blockIndex + 1]$
 4:     for $i = 0$ to $NUM\_ROUNDS - 1$ do
 5:         $rk[i] \leftarrow xor128(i \& PRNGS\_MODULO)$
 6:     end for
 7:     for $i = NUM\_ROUNDS - 1$ downto 0 do
 8:         $temp \leftarrow left$
 9:         $n \leftarrow left \wedge 0x1f$
10:         $left \leftarrow right \oplus rk[i] \oplus ((left \ll n) \vee (left \gg (32 - n)))$
11:         $right \leftarrow temp$
12:     end for
13:     $plaintext[blockIndex] \leftarrow left$
14:     $plaintext[blockIndex + 1] \leftarrow right$
15: end function
```

Algorithm 6 Encrypt Function

```
Require: plaintext[] PKCS#7 padded, key[], and iv[] are valid arrays, len is even and
    within bounds
Ensure: Encrypted ciphertext is stored in ciphertext[]
1: function ENCRYPT(plaintext[], key[], iv[], ciphertext[], len)
2:     initializeSeeds(key[], iv[])
3:     for $i = 0$ to $len - 1$ step 2 do
4:         feistelEncrypt(plaintext[], ciphertext[], i)
5:     end for
6: end function
```

Algorithm 7 Decrypt Function

```
Require: ciphertext[], key[], and iv[] are valid arrays, len is even and within bounds
Ensure: Decrypted plaintext is stored in plaintext[]
1: function DECRYPT(ciphertext[], key[], iv[], plaintext[], len)
2:     initializeSeeds(key[], iv[])
3:     for $i = 0$ to $len - 1$ step 2 do
4:         feistelDecrypt(ciphertext[], plaintext[], i)
5:     end for
6: end function
```

B Appendix: C Code Implementation

An example C code implementation of Synergy:

```c
#include <stdint.h>
#include <string.h>

#define NUM_ROUNDS 16
#define NUM_PRNGS 8
#define PRNGS_MODULO (NUM_PRNGS - 1)

uint32_t x[NUM_PRNGS];
uint32_t y[NUM_PRNGS];
uint32_t z[NUM_PRNGS];
uint32_t w[NUM_PRNGS];
uint32_t rk[NUM_ROUNDS];

void initializeSeeds(const uint32_t key[], const uint32_t iv[]) {
    for (int i = 0; i < NUM_PRNGS; i++) {
        x[i] = key[i] ^ iv[i];
        y[i] = key[i + NUM_PRNGS] ^ iv[i + NUM_PRNGS];
        z[i] = key[i + NUM_PRNGS * 2] ^ iv[i + NUM_PRNGS * 2];
        w[i] = key[i + NUM_PRNGS * 3] ^ iv[i + NUM_PRNGS * 3];
    }
}

static inline uint32_t xor128(int i) {
    uint32_t t = x[i] ^ (x[i] << 11);
    x[i] = y[i];
    y[i] = z[i];
    z[i] = w[i];
    w[i] = w[i] ^ (w[i] >> 19) ^ t ^ (t >> 8);
    return w[i];
}

void feistelEncrypt(const uint32_t plaintext[], uint32_t ciphertext[], int
     blockIndex) {
    uint32_t left = plaintext[blockIndex];
    uint32_t right = plaintext[blockIndex + 1];

    for (int i = 0; i < NUM_ROUNDS; i++) {
        rk[i] = xor128(i & PRNGS_MODULO);
    }

    for (int i = 0; i < NUM_ROUNDS; i++) {
        uint32_t temp = right;
        uint32_t n = right & 0x1f;
        right = left ^ rk[i] ^ ((right << n) | (right >> (32 - n)));
        left = temp;
    }

    ciphertext[blockIndex] = left;
    ciphertext[blockIndex + 1] = right;
}

void feistelDecrypt(const uint32_t ciphertext[], uint32_t plaintext[], int
     blockIndex) {
    uint32_t left = ciphertext[blockIndex];
    uint32_t right = ciphertext[blockIndex + 1];

    for (int i = 0; i < NUM_ROUNDS; i++) {
        rk[i] = xor128(i & PRNGS_MODULO);
    }

    for (int i = NUM_ROUNDS - 1; i >= 0; i--) {
        uint32_t temp = left;
        uint32_t n = left & 0x1f;
        left = right ^ rk[i] ^ ((left << n) | (left >> (32 - n)));
        right = temp;
    }

    plaintext[blockIndex] = left;
    plaintext[blockIndex + 1] = right;
}

void encrypt(const uint32_t plaintext[], const uint32_t key[], const uint32_t iv[], uint32_t ciphertext[], int len) {
    initializeSeeds(key, iv);

    for (int i = 0; i < len; i += 2) {
        feistelEncrypt(plaintext, ciphertext, i);
    }
}

void decrypt(const uint32_t ciphertext[], const uint32_t key[], const uint32_t iv[], uint32_t plaintext[], int len) {
    initializeSeeds(key, iv);

    for (int i = 0; i < len; i += 2) {
        feistelDecrypt(ciphertext, plaintext, i);
    }
}

int main() {
    // Initialize key material (replace with secure randomness)
    uint8_t keyMaterial[NUM_PRNGS * 16];
    for (int i = 0; i < sizeof(keyMaterial); i++) {
        keyMaterial[i] = i + 1;
    }

    // Convert key material to uint32_t array
    uint32_t key[NUM_PRNGS * 4];
    memcpy(key, keyMaterial, sizeof(keyMaterial));

    // Initialize IV (zero-filled for demonstration)
    uint32_t iv[NUM_PRNGS * 4] = {0};

    // Define message
    uint32_t message[] = {101, 102, 103, 104, 105, 0x04040404}; // PKCS#7 padded
    int len = sizeof(message) / sizeof(message[0]);

    // Encrypt
    uint32_t ciphertext[len];
    encrypt(message, key, iv, ciphertext, len);
    printf("Ciphertext: ");
    for (int i = 0; i < len; i++) {
        printf("%08x ", ciphertext[i]);
    }
    printf("\n");

    // Decrypt
    uint32_t plaintext[len];
    decrypt(ciphertext, key, iv, plaintext, len);
    printf("Plaintext: ");
    for (int i = 0; i < len - 1; i++) {
        printf("%u ", plaintext[i]);
    }
    printf("\n");

    return 0;
}
```

Listing 1.1. C Code Example