

# NEW-CIPHER: A 512-bit Wide-Block SPN Block Cipher with Transparent Tamil-Seeded Parameters

*Design, Security Analysis, and Comparison with Kalyna*

**S. Ravichandran Ayyer**

Independent Researcher, Puttur, Andhra Pradesh, India

ORCID: 0009-0003-6094-2534 · saktheee2009@gmail.com

Preprint, version 1 — October 2026

---

## Abstract

We present NEW-CIPHER v0.2-B, a substitution–permutation network (SPN) block cipher with a 512-bit block, a 256-bit key and 14 rounds. Every constant of the cipher — the affine layer of the S-box, the 8×8 diffusion matrix, and all key-schedule constants — is derived with SHA-256 from a single public seed string containing the Tamil word தமிழ், so that anyone can regenerate and check them. The S-box is affine-equivalent to inversion in  $GF(2^8)$  and therefore has differential uniformity 4, nonlinearity 112 and algebraic degree 7. The diffusion layer is an 8×8 Cauchy matrix, which is MDS by theorem; we additionally verify all 12,869 square submatrices. By the wide trail strategy, every 4-round differential or linear trail has at least 81 active S-boxes, giving trail probability at most  $2^{-486}$ . The key schedule uses the cipher's own round function with Davies–Meyer feed-forward, and no round key is a rotation of another. We give an Encrypt-then-MAC mode with an IND-CCA argument, a constant-time C implementation checked with valgrind (0 secret-dependent branches or addresses), and test vectors reproduced in Python and C on x86-64 Linux and ARM Android. Finally, we port the Ukrainian standard Kalyna-512/512 to Python, verify it against its official test vectors, and run the same 35-test battery on both ciphers, reporting trade-offs in both directions. All programs and outputs are included.

**Keywords:** block cipher; substitution–permutation network; wide-block encryption; MDS matrix; Cauchy matrix; wide trail strategy; nothing-up-my-sleeve constants; Kalyna; authenticated encryption; constant-time implementation

## 1. Introduction

Most deployed block ciphers, including AES [3, 28], use a 128-bit block. With an  $n$ -bit block, security of the common modes of operation degrades once about  $2^{n/2}$  blocks are processed under one key (the birthday bound) [21]; for  $n = 128$  this is  $2^{64}$  blocks. Wider blocks raise this limit and also give a larger margin against quantum search [20]. Kalyna, the Ukrainian national standard DSTU 7624:2014, offers 256- and 512-bit blocks [8].

A second concern is the origin of constants. When a standard fixes S-boxes or matrices as tables without a reproducible generation procedure, users cannot rule out hidden structure; this concern has been discussed at length for public-key parameters [32]. A common remedy is to derive constants from a public "nothing-up-my-sleeve" seed.

This paper builds a cipher step by step from the properties expected of a strong cipher, and requires each step to carry either a mathematical proof, an exact computation, or an honestly labelled estimate. Its contributions are:

- A complete specification of a 512-bit-block SPN whose constants are all reproducible from one public seed (Section 3).
- Proofs, by citation of established theorems plus exhaustive computation, of the S-box properties, the MDS property of the 8×8 matrix, and a  $2^{-486}$  bound on every 4-round differential and linear trail (Section 5).

- A key schedule in which every round key is a one-way function of the whole key and no round key is a rotation of another (Sections 3.7 and 5.7).
- An Encrypt-then-MAC mode with an IND-CCA argument, and a constant-time C implementation verified with valgrind (Sections 3.8, 5.11 and 8).
- A side-by-side evaluation against Kalyna-512/512 under an identical battery of 35 tests, including one observation about Kalyna's modular key addition (Section 7).

We do **not** claim that the cipher is "unbreakable". Its security as a pseudorandom permutation is an assumption, supported by the analysis below, that can only be strengthened by independent public cryptanalysis over time.

## 2. Preliminaries

**Field.** Bytes are elements of  $GF(2^8) = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1)$ , the AES polynomial 0x11B. Addition is XOR.

**S-box measures [5, 13, 14].** For a bijection  $S$  on 8 bits, the differential uniformity is  $DU = \max$  over  $a \neq 0$  and  $b$  of  $\#\{x : S(x) \oplus S(x \oplus a) = b\}$ ; the maximum differential probability is  $DU/256$ . The nonlinearity is  $NL = 128 - W/2$ , where  $W$  is the largest absolute Walsh coefficient over non-zero output masks; the maximum linear probability is  $(W/256)^2$ . The algebraic degree is the smallest degree of the algebraic normal forms of the output bits.

**Branch number and MDS [4, 6].** For a linear map  $M$  on  $n$  bytes, the branch number is  $\min$  over  $x \neq 0$  of  $wt(x) + wt(Mx)$ , where  $wt$  counts non-zero bytes. Its maximum is  $n + 1$ ; matrices reaching it are called MDS. A matrix is MDS if and only if every square submatrix is non-singular [6].

**Wide trail theorem [3, 4].** In an SPN whose byte permutation sends the bytes of each column to distinct columns and whose column map has branch number  $B$ , every 2-round trail has at least  $B$  active S-boxes and every 4-round trail at least  $B^2$ .

**Security notions.** A block cipher  $E$  is a pseudorandom permutation (PRP) if no efficient adversary can distinguish  $E_K$  for a random key from a random permutation. IND-CPA and IND-CCA are indistinguishability of ciphertexts under chosen-plaintext and chosen-ciphertext attack [21, 22].

## 3. Specification

### 3.1 Parameters

| Parameter    | Value                                            |
|--------------|--------------------------------------------------|
| Block size   | 512 bits (64 bytes, an 8×8 byte state)           |
| Key size     | 256 bits (32 bytes)                              |
| Rounds       | 14                                               |
| Round keys   | 15 × 512 bits                                    |
| S-box        | one 8-bit S-box, affine-equivalent to inversion  |
| Linear layer | ByteShift + 8×8 Cauchy MDS matrix over $GF(2^8)$ |
| Key addition | XOR                                              |

The state byte at row  $r$  (0–7) and column  $c$  (0–7) has index  $p = 8c + r$ . Plaintext byte  $j$  is placed at index  $j$ .

### 3.2 Public seed and derivation of constants

All constants come from the public seed string

SEED = "NEW-CIPHER v0.2 | தமிழ் | public constants" (UTF-8, 52 bytes)

through

$$h(\text{label}, i) = \text{SHA-256}(\text{SEED} \parallel \text{"|"} \parallel \text{label} \parallel \text{"|"} \parallel i), \quad h_{512}(\text{label}, i) = h(\text{label}, i \parallel \text{"a"}) \parallel h(\text{label}, i \parallel \text{"b"})$$

where  $i$  is written as a decimal string. The labels used are SBOX, CAUCHY, KEXP, KT, RKROUND, RKG and MODE. Because SHA-256 [29] gives the designer no control over its outputs, the only freedom is the seed text itself, which is fixed in this paper.

### 3.3 S-box

$$S(x) = A \cdot x^{-1} \oplus c, \text{ with } 0^{-1} := 0$$

where  $x^{-1}$  is inversion in  $GF(2^8)$  and  $A$  is the  $8 \times 8$  binary circulant matrix whose first row is the byte  $m$  ( $A \cdot v = \text{XOR of } \text{rotl}(v, k) \text{ over the bits } k \text{ set in } m$ ). For  $i = 0, 1, 2, \dots$  the pair  $(m, c)$  is taken as the first two bytes of  $h(\text{SBOX}, i)$ , and the first  $i$  is accepted for which (a)  $m$  has odd Hamming weight greater than 1, so that  $A$  is invertible over  $GF(2)$  and is not a plain rotation, (b)  $S$  is a bijection, and (c)  $S$  has no fixed point  $S(x) = x$  and no opposite fixed point  $S(x) = x \oplus 0xFF$ . The result is  $i = 15$ ,  $m = 0x49$ ,  $c = 0xE4$ . The first 16 entries are

E4 AD 66 18 83 B0 9A CF 71 02 CE 12 FD 72 57 EC

and SHA-256 of the full table is

d6872e197e188811224b9667bda8a489b7065136be6ed9018918d1b11ceb46c0

### 3.4 ByteShift

Row  $r$  is rotated left by  $r$  positions: the byte at  $(r, c)$  moves to  $(r, (c - r) \bmod 8)$ . Because the 8 bytes of any column go to 8 different columns, ByteShift is diffusion-optimal in the sense of [4].

### 3.5 ColumnMix

Take bytes from  $h(\text{CAUCHY}, 0)$ ,  $h(\text{CAUCHY}, 1)$ , ... in order and keep the first 16 distinct values; call the first eight  $x_0 \dots x_7$  and the last eight  $y_0 \dots y_7$ . The matrix is

$$M[i][j] = (x_i \oplus y_j)^{-1}$$

which is a Cauchy matrix (the  $x_i \oplus y_j$  are never zero because all 16 values are distinct). Each column of the state is multiplied by  $M$ . The selected values are

$x = 74 \ 2A \ BD \ 07 \ 85 \ A7 \ 54 \ 2F$ ,  $y = 0F \ 87 \ 1A \ FB \ AE \ 75 \ 14 \ 01$ ,

the first row of  $M$  is  $06 \ 34 \ 21 \ A4 \ C4 \ 01 \ 16 \ B5$ , and SHA-256 of  $M$  (row by row) is

e999262f3d4cdddbaa0aa08e4b211115c116201bd62effc460bf41602a3139b8

### 3.6 Encryption and decryption

Let SB, BS and MC denote SubBytes ( $S$  applied to every byte), ByteShift and ColumnMix. Encryption of a 64-byte block  $P$  is

$$s \leftarrow P \oplus RK_0; \text{ for } r = 1 \dots 14: s \leftarrow MC(BS(SB(s))) \oplus RK_r; \text{ output } s.$$

Decryption applies the inverses in reverse order, using  $S^{-1}$ , the inverse rotation and  $M^{-1}$ . Unlike AES, the last round keeps its column mix; this changes neither security nor correctness.

### 3.7 Key schedule

Let  $P(s, c) = MC(BS(SB(s))) \oplus c$  be the unkeyed round with a public 512-bit constant  $c$ , and let  $\text{rot}_n$  rotate a 64-byte string left by  $n$  bytes. With  $C_{\text{EXP}} = h(\text{KEXP}, 0)$ ,  $C_{\text{KT},j} = h512(\text{KT}, j)$ ,  $RC_i = h512(\text{RKROUND}, i)$  and  $RC_{i,j} = h512(\text{RKG}, "i,j")$ :

$$T0 = K \parallel (K \oplus C_{\text{EXP}})$$

$$KT = P(P(P(P(T0, C_{\text{KT},0}), C_{\text{KT},1}), C_{\text{KT},2}), C_{\text{KT},3}) \oplus T0$$

$$X_i = KT \oplus \text{rot}_{5i}(T0) \oplus RC_i, \quad RK_i = G_i(X_i) \oplus X_i, \quad i = 0 \dots 14$$

where  $G_i$  is three applications of  $P$  with constants  $RC_{i,0}$ ,  $RC_{i,1}$ ,  $RC_{i,2}$ . The two feed-forward steps ( $\dots \oplus T_0$  and  $\dots \oplus X_i$ ) are the Davies–Meyer construction [19]: since  $P$  is a public permutation, without the feed-forward an attacker who learns one round key could run  $G_i$  backwards.

### 3.8 Authenticated encryption mode

The key  $K$  is split into two independent keys with the cipher itself:  $K_{\text{enc}}$  is the first 32 bytes of  $E_K(C_{\text{ENC}})$  and  $K_{\text{mac}} = E_K(C_{\text{MAC}})$ , where  $C_{\text{ENC}}$  and  $C_{\text{MAC}}$  are h512 values of the labels  $\text{MODE|ENC}$  and  $\text{MODE|MAC}$ . For a message  $M$  with associated data  $AD$  and a fresh random 32-byte nonce  $N$ :

- $C = M \oplus \text{keystream}$ , where the 64-byte keystream block starting at byte offset  $t$  is  $E_{K_{\text{enc}}}(N \parallel t)$ , with  $t$  written as a 32-byte big-endian integer (counter mode).
- $T = \text{HMAC-SHA-256}(K_{\text{mac}}, \text{len}(AD) \parallel AD \parallel N \parallel \text{len}(C) \parallel C)$ , lengths as 8-byte big-endian integers [23].
- Output  $N \parallel C \parallel T$ .

Decryption recomputes  $T$  and compares in constant time **before** decrypting; on mismatch nothing is decrypted. A nonce must never be reused under the same key.

### 3.9 Test vectors

| Key          | Plaintext    | Ciphertext (64 bytes, hex)                                                                                                       |
|--------------|--------------|----------------------------------------------------------------------------------------------------------------------------------|
| 00 × 32      | 00 × 64      | ec4e9961dd402f8521c5c696ac3213f24d80097abc3786e32e1355465855c24737f3092f69ef08332b1115585e01e7f3f9eeac37d1b6fc1590f6c862a832821b |
| 00 01 ... 1F | 00 01 ... 3F | 87db50055a2ca8236e46e132aa9e15bd96427a03e7ba0a642714a8cfa8b5676a1c9bd5253c8978e7dc3ba9a7ef13d957452af5a1a635d0e7d3b38a720c27b2b8 |
| 00 01 ... 1F | 40 41 ... 7F | 455e363cee0c7fa591b879867bed004593b78fa6b776ae4aeeed902a66022120812a95ebeb95758c86c9b416eadd1cb6bba717f75eec37ec1c470212d2a6a3   |

| Check                                                                     | Value (hex)                                                                                                                      |
|---------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Decryption: key 1F 1E ... 00, ciphertext 7F 7E ... 40 → plaintext         | 87bbb0ce4414e29e2f5917d8ccbb77380232e85bd97be57e486cb4fe7e1b1631161d7a08710f5a3bd2182e1379f0efc5d5c2da2865d85aa40791745a50555384 |
| Zero key: first 16 bytes of $RK_0$                                        | fcfcc1a27789ab7ee0a403765f657dc                                                                                                  |
| Zero key: SHA-256 of all 15 round keys (first 16 bytes)                   | 37404d0cc361bc31c8fb1bf8d5c7b180                                                                                                 |
| Mode: zero key, zero nonce, empty AD, message "NEW-CIPHER" → $C$ then $T$ | 05d7a5da3297e79e3e1167990970c21e25a0e9be9a5c620f7eab9b10e080303820f5aa4b28be3ef22f2b                                             |

## 4. Design rationale

**Kerckhoffs's principle [1].** Everything except the key is public, including how each constant was chosen; the 256-bit key is meant to be drawn uniformly at random, giving key entropy  $H(K) = 256$  bits. Exhaustive search needs about  $2^{255}$  trials on average; at  $10^{18}$  keys per second that is about  $10^{51}$  years.

**Why a 512-bit block but a 256-bit key.** Table 1 (program A.5) compares block and key sizes with a simple round-count model:  $R_{\text{dist}}$  is the number of rounds after which the best trail probability falls below  $2^{-\text{block}}$ ,  $R_{\text{kg}}$  = key/block extra rounds an attacker may peel off by guessing round keys, and  $R_{\text{rec}} = \lceil 1.5 (R_{\text{dist}} + R_{\text{kg}}) \rceil$ . A larger key raises the round count and cost but does not raise the birthday limit, which depends only on the block size; a 256-bit key already gives  $2^{128}$  security against Grover search [20]. A larger block, by contrast, raises the birthday limit from  $2^{64}$  to  $2^{256}$  blocks. This is why the block, not the key, was enlarged.

**Table 1. Round-count model (heuristic; see program A.5)**

| Block | Column (bytes) | B | Key  | $R_{\text{dist}}$ | $R_{\text{kg}}$ | $R_{\text{rec}}$ | Grover    | Birthday  | Relative cost |
|-------|----------------|---|------|-------------------|-----------------|------------------|-----------|-----------|---------------|
| 128   | 4              | 5 | 256  | 4                 | 2               | 9                | $2^{128}$ | $2^{64}$  | 1.0           |
| 128   | 4              | 5 | 1024 | 4                 | 8               | 18               | $2^{512}$ | $2^{64}$  | 2.0           |
| 256   | 4              | 5 | 256  | 8                 | 1               | 14               | $2^{128}$ | $2^{128}$ | 1.6           |
| 512   | 8              | 9 | 256  | 6                 | 0               | 9                | $2^{128}$ | $2^{256}$ | 1.6           |

| Block | Column (bytes) | B | Key | $R_{\text{dist}}$ | $R_{\text{kg}}$ | $R_{\text{rec}}$ | Grover    | Birthday  | Relative cost |
|-------|----------------|---|-----|-------------------|-----------------|------------------|-----------|-----------|---------------|
| 512   | 8              | 9 | 512 | 6                 | 1               | 11               | $2^{256}$ | $2^{256}$ | 2.0           |

**Why 8-byte columns.** The wide trail bound  $B^2$  requires the byte permutation to send the bytes of a column to distinct columns, so a state with 8 rows needs at least 8 columns; with 8-byte columns this forces the 512-bit block. In return the column map can reach branch number 9, so a 4-round trail has at least 81 active S-boxes instead of 25.

**Why a Cauchy matrix.** For an  $8 \times 8$  matrix to be MDS, all 12,869 square submatrices must be non-singular. A matrix drawn at random from the seed satisfies this with probability roughly  $(255/256)^{12869} \approx e^{-50}$ , so random search is hopeless. A Cauchy matrix is MDS for every choice of 16 distinct values [6, 7], so the seed can choose those values freely and the property is guaranteed.

**Why an inversion-based S-box.** Inversion in  $\text{GF}(2^8)$  reaches  $\text{DU} = 4$  and  $\text{NL} = 112$ , the best values known for 8-bit bijections [5]. Its cost is a simple algebraic description (39 quadratic equations, Section 5.6); this trade-off is discussed openly in Section 7.

**Why 14 rounds.** The model above gives  $R_{\text{rec}} = 9$  for a 512-bit block with a 256-bit key. Because AES-like designs also face attacks outside the model (impossible differential, meet-in-the-middle, related-key), we do not go below the AES-256 round count and fix 14 rounds.

## 5. Security analysis

### 5.1 S-box

**Theorem 1 (Nyberg [5]).** The inversion map on  $\text{GF}(2^8)$  has differential uniformity 4 and nonlinearity 112. Affine equivalence preserves  $\text{DU}$ ,  $\text{NL}$  and algebraic degree.

The program A.2 confirms by exhaustive computation: bijective,  $\text{DU} = 4$  (maximum differential probability  $2^{-6}$ ),  $\text{NL} = 112$  (maximum linear probability  $2^{-6}$ ), algebraic degree 7 (the maximum for an 8-bit bijection), no fixed points and no opposite fixed points.

### 5.2 Diffusion

**Theorem 2 (Cauchy matrices [6, 7]).** If  $x_0, \dots, x_7, y_0, \dots, y_7$  are distinct elements of  $\text{GF}(2^8)$ , every square submatrix of  $[1/(x_i + y_j)]$  is non-singular, so the matrix is MDS with branch number 9. The transpose is also Cauchy, so the linear branch number is also 9.

Program A.4 checks all 12,869 square submatrices of  $M$  and of its transpose by Gaussian elimination (all non-singular), confirms  $\min(\text{wt}(x) + \text{wt}(Mx)) = 9$  for all 2,040 single-byte inputs and 100,000 random inputs, confirms that  $M$  is invertible, and shows that after 2 rounds every output byte depends on all 64 input bytes. The  $4 \times 4$  version used as a baseline (program A.3) has branch number 5 and reaches full diffusion in 2 rounds on a 128-bit state.

### 5.3 Differential and linear cryptanalysis [13, 14]

**Theorem 3 (wide trail [3, 4]).** With  $B = 9$  and a diffusion-optimal ByteShift, every 4-round trail has at least 81 active S-boxes. With per-S-box probability at most  $2^{-6}$ , every 4-round differential or linear trail has probability at most  $2^{-486}$ .

**Table 2. Minimum active S-boxes and trail bounds**

| Rounds | Active S-boxes $\geq$ | Trail probability $\leq$ | Useful against $2^{256}$ work? |
|--------|-----------------------|--------------------------|--------------------------------|
| 1      | 1                     | $2^{-6}$                 | yes                            |
| 2      | 9                     | $2^{-54}$                | yes                            |
| 3      | 17                    | $2^{-102}$               | yes                            |
| 4      | 81                    | $2^{-486}$               | no                             |

| Rounds | Active S-boxes $\geq$ | Trail probability $\leq$ | Useful against $2^{256}$ work? |
|--------|-----------------------|--------------------------|--------------------------------|
| 8      | 162                   | $2^{-972}$               | no                             |

A trail bound is not a bound on a differential, because many trails can share the same input and output difference (clustering) [3]. For AES-like ciphers the gap is known to be small, but it is not proved here; this is a limitation.

#### 5.4 Integral (Square) attack [15]

Taking 256 plaintexts that differ only in one byte, every output byte of the XOR-sum is zero ("balanced") after 1, 2 and 3 rounds, and not after 4 or 5 rounds. Program A.9 measures exactly this (64 of 64 bytes balanced up to 3 rounds; 0 or 1 of 64 at 4 and 5 rounds, as expected for random data, where each byte is balanced with probability  $1/256$ ). Starting from a full diagonal ( $2^{64}$  texts) adds one round, giving a 4-round distinguisher.

#### 5.5 Impossible differential [31]

Because any non-zero difference becomes fully active after 2 rounds in both directions, impossible differentials are expected to cover at most about 4 rounds. We have not searched for them automatically; the figure is an estimate by analogy with AES.

#### 5.6 Algebraic and higher-order attacks [16, 17]

The algebraic degree after  $r$  rounds is at most  $\min(7^r, 511)$ : 7, 49, 343, then 511 from round 4 on, so higher-order differential distinguishers are limited to about 3 rounds (tighter bounds exist [30]). Each inversion-based S-box satisfies 39 linearly independent quadratic equations in its 16 input and output bits (program A.13 counts them). The full cipher plus key schedule has 4,032 S-boxes, giving a system of 157,248 quadratic equations in about 32,512 variables over  $\text{GF}(2)$ . No known method solves systems of this size; even the much smaller AES-128 system remains unsolved [17].

#### 5.7 Key schedule

There is no proof for the key schedule, as for most ciphers; we give design principles and measurements (program A.6). Flipping one key bit changes on average 255.8 of the 512 bits of every round key (minimum 220, maximum 293 over 1,920 pairs, consistent with a binomial distribution); low-weight keys (all-zero, all-one, 0x55 repeated, a counter) give distinct, non-zero and balanced round keys; no round key equals another, also across related keys; consecutive round keys differ in 49.1% of their bits; and no round key is a byte rotation of the previous one (Section 7). Per-round constants  $\text{RC}_i$  break the round-to-round symmetry used by slide attacks.

#### 5.8 Meet-in-the-middle and biclique [18]

Every round key depends non-linearly on the whole key, so the key cannot be split into independent halves. By analogy with AES, a biclique attack might reduce exhaustive search by about 2 bits, which is irrelevant in practice.

#### 5.9 Generic attacks

Exhaustive search:  $2^{255}$  trials on average. Quantum search: about  $2^{128}$  [20]. Birthday limit for modes: about  $2^{256}$  blocks per key.

#### 5.10 Summary and security margin

**Table 3. Estimated reach of attacks (distinguisher + key-recovery rounds)**

| Attack                  | Distinguisher | + key recovery | Reach |
|-------------------------|---------------|----------------|-------|
| Differential / linear   | 3             | 2              | 5     |
| Integral (Square)       | 4             | 3              | 7     |
| Impossible differential | 4             | 3              | 7     |

| Attack                    | Distinguisher | + key recovery | Reach |
|---------------------------|---------------|----------------|-------|
| Higher-order differential | 3             | 2              | 5     |

The worst estimate is 7 of 14 rounds, a 50% security margin. These numbers are estimates by analogy with published AES and Kalyna results, not proofs; only the 3-round integral distinguisher was measured directly.

### 5.11 IND-CCA security of the mode

By the generic composition theorem of Bellare and Namprempre [22], Encrypt-then-MAC with an IND-CPA encryption scheme and a strongly unforgeable MAC is IND-CCA:

$$\text{Adv}^{\text{ind-cca}} \leq \text{Adv}^{\text{ind-cpa}}_{\text{CTR}} + 2 \cdot \text{Adv}^{\text{suf-cma}}_{\text{HMAC}}$$

For counter mode with random 256-bit nonces and a 512-bit block, standard arguments [21] give, for  $q$  messages and  $\sigma$  blocks in total, the loose bound

$$\text{Adv}^{\text{ind-cpa}}_{\text{CTR}} \leq 2 \cdot \text{Adv}^{\text{prp}}_{\text{E}} + \sigma^2 / 2^{512} + q^2 / 2^{256}$$

plus a further PRP term for deriving  $K_{\text{enc}}$  and  $K_{\text{mac}}$  from  $K$  with  $E$ . The birthday term is  $\sigma^2/2^{512}$  rather than  $\sigma^2/2^{128}$  for a 128-bit block. **The whole bound is conditional on E being a PRP**, which cannot be proved and is supported only by Sections 5.1–5.10. Programs A.8 and A.13 check the behaviour: all bit-flip forgeries are rejected (624 of 624 in A.8), as are wrong associated data, truncation and a wrong key, and the same message encrypts differently each time.

## 6. Experimental evaluation

### 6.1 Avalanche and strict avalanche

**Table 4. Plaintext avalanche by round (program A.7; mean bits changed out of 512, 200 samples)**

| Round        | 1     | 2     | 3     | 4     | 14    |
|--------------|-------|-------|-------|-------|-------|
| Bits changed | 32.2  | 257.1 | 255.8 | 256.8 | 255.6 |
| Fraction     | 0.063 | 0.502 | 0.500 | 0.502 | 0.499 |

Round 1 matches the theory: one byte change activates one column of 8 bytes, about 4 bits each, i.e. 32 bits. From round 2 the avalanche is complete, as predicted in Section 5.2. Flipping one key bit changes 255.6 bits on average (minimum 227, maximum 290). In the strict avalanche criterion [24], each of the 512 output bits flipped with probability between 0.448 and 0.543 over 1,000 flips, which is within the  $\pm 3\sigma$  range expected for that sample size.

### 6.2 Statistical randomness

Program A.12 implements seven tests of NIST SP 800-22 [25] and checks them against the worked examples of that document (all p-values agree to six decimals, the longest-run test to five). On true random data each test raises a false alarm about 1% of the time, as it should. Counter-mode output of the cipher passed every test: 10 sequences of 100,000 bits each, with at least 9 of 10 sequences giving  $p \geq 0.01$  for every test, and a byte entropy of 7.9985 bits per byte with a passing chi-square test. Passing these tests is a necessary minimum only; weak ciphers also pass them.

### 6.3 Test battery

Program A.13 runs 35 graded tests in nine groups: design, S-box, diffusion, differential/linear, key schedule, cipher behaviour, NIST randomness, authenticated mode and implementation. Each result is graded IDEAL (optimal or near-optimal), PASS (at the level of trusted ciphers), WEAK or FAIL. NEW-CIPHER passed 35 of 35 tests, 13 at IDEAL level; the full output is in Appendix B. The constant-time test I2 refers to the C implementation of Section 8.2, because the Python reference itself is not constant-time.

## 7. Comparison with Kalyna-512/512

### 7.1 Method

We downloaded the official reference implementation of Kalyna [9], compiled it, and confirmed that all ten of its test vectors pass on both platforms. We then ported Kalyna-512/512 to Python (program A.10) and verified the port against the official encryption and decryption vectors. Both ciphers were run through the same programs A.11 and A.13.

### 7.2 Results

**Table 5. Same tests on both ciphers**

| Property                             | NEW-CIPHER v0.2-B    | Kalyna-512/512           |
|--------------------------------------|----------------------|--------------------------|
| Key / rounds                         | 256 / 14             | 512 / 18                 |
| S-boxes                              | 1, inversion-based   | 4, fixed tables          |
| S-box DU / NL / degree               | 4 / 112 / 7          | 8 / 104 / 7 (all four)   |
| Quadratic equations per S-box        | 39                   | 0                        |
| MDS (12,869 submatrices)             | yes (Cauchy)         | yes (circulant)          |
| 4-round trail bound                  | $2^{-486}$           | $2^{-391}$               |
| Avalanche after 2 rounds             | 0.497–0.502          | 0.502                    |
| Key avalanche                        | 0.501                | 0.498                    |
| Integral distinguisher               | 3 rounds             | 3 rounds                 |
| Odd round key = rotation of even one | 0 of 14              | 9 of 18                  |
| Constants reproducible from a seed   | yes                  | no (tables)              |
| NIST tests (7) + entropy             | all pass             | all pass                 |
| Constant-time implementation         | yes (C, Section 8.2) | reference uses tables    |
| Battery score                        | 35/35                | 30/33 (2 not applicable) |

### 7.3 Discussion

**S-box trade-off.** Our S-box has better differential and linear properties (DU 4 vs 8, NL 112 vs 104), which is why our 4-round bound is stronger. Kalyna's designers instead chose S-boxes without a simple algebraic description: we count 0 quadratic equations for each Kalyna S-box against 39 for ours. This is a deliberate design choice against algebraic attacks [17]. Neither choice is strictly better.

**Modular key addition.** Kalyna adds the first and last round keys modulo  $2^{64}$  instead of XOR. With the active byte at position 0 (the low byte of a 64-bit word), the carry spreads into neighbouring bytes and the 3-round integral property disappears in our test. With the active byte at position 7 (the high byte), the carry leaves the word and the 3-round distinguisher returns (Table 5 reports the attacker's best position). So the modular addition does not remove the integral distinguisher for an attacker who chooses the byte position.

**Key schedule.** In Kalyna every odd round key is a byte rotation of the preceding even round key [9]; in ours no round key is related to another in this way. We know of no attack exploiting the Kalyna relation; we report it as a structural difference. Our key schedule costs more (64 unkeyed rounds against 23).

**Fairness of the comparison.** The tests were chosen by us, after designing our cipher. Two of the three criteria Kalyna does not meet (reproducible constants and round-key rotation) measure exactly what our design aims at, and the battery has few tests that reward Kalyna's own design goals. Kalyna has also withstood about a decade of public analysis, in which attacks reach 9 rounds of other Kalyna variants [10, 11] and a practical cache attack targets its key expansion [12]; our cipher has had none. **The scores must therefore not be read as "NEW-CIPHER is stronger than Kalyna".** What they show is that, under an identical battery, NEW-CIPHER meets every criterion, while trading algebraic simplicity of its S-box for optimal differential and linear properties.

## 8. Implementation

### 8.1 Reproducibility

**Table 6. Platforms on which the test vectors were reproduced**

| Implementation                     | x86-64 Linux               | ARM Android (phone) |
|------------------------------------|----------------------------|---------------------|
| Python, steps 1–8 and test battery | yes (Python 3.13)          | yes (Pydroid3)      |
| C reference (newcipher.c)          | yes (gcc 13.3, clang 18.1) | yes                 |
| C constant-time (newcipher_ct.c)   | yes (gcc, clang)           | —                   |
| Kalyna official C reference        | yes, 10/10                 | yes, 10/10          |

The C programs re-derive every constant from the seed with their own SHA-256 code and compare them with the Python fingerprints, so the specification is checked across two languages and two processor families.

### 8.2 Constant-time implementation

A table lookup indexed by secret data leaks through the processor cache; the cache attack on Kalyna's key expansion [12] is a recent example. In `newcipher_ct.c`, no branch and no memory address depends on the key or the data. The S-box is computed as  $A \cdot x^{254} \oplus c (x^{254} = x^{-1})$  with an addition chain of 11 multiplications, and every  $GF(2^8)$  multiplication uses masks instead of branches:

```
static uint8_t ct_mul(uint8_t a, uint8_t b) {
    uint8_t r = 0; int i;
    for (i = 0; i < 8; i++) {
        r ^= (uint8_t)(a & (uint8_t)(-(int)(b & 1)));
        a = (uint8_t)((a << 1) ^ (0x1B & (uint8_t)(-(int)(a >> 7))));
        b >>= 1;
    }
    return r;
}
```

The constant-time S-box agrees with the table on all 256 inputs, and the program reproduces all test vectors.

Evidence of constant-time behaviour, from strongest to weakest:

- **By construction:** no secret-dependent index or branch in the source.
- **valgrind (ctgrind method [27]):** key and plaintext are marked as undefined, so memcheck reports any branch or address that depends on them. The table-based build gives 1,920 errors in 3 places; the constant-time build gives 0 errors under gcc and clang at -O0, -O2 and -O3 (six builds). The test thus detects the leak it is meant to detect.
- **Timing (dudect [26]):** fixed versus random inputs, Welch t-test; over three runs  $|t|$  stayed at or below 1.7 for encryption and 3.4 for the key schedule, under the 4.5 threshold. However, the table-based build also passed on this machine, because its small tables stay in the cache; this test therefore adds little evidence here.

### 8.3 Performance

These are reference implementations, not optimised ones: Python about 128 KB/s (x86-64) and 78 KB/s (phone); C about 10.8 MB/s (x86-64) and 2.1 MB/s (phone); constant-time C about 0.35 MB/s (x86-64). The official Kalyna reference C code runs at about 0.7 MB/s on the same x86-64 machine, but it was written for clarity and allocates memory in every round, so this is **not** a meaningful speed comparison. Bitsliced constant-time code and optimised implementations of both ciphers are future work.

## 9. Limitations and future work

- The PRP security of the cipher is an assumption. The proofs cover components (S-box, matrix, trail bounds) and the mode, not the cipher as a whole.
- Trail bounds do not account for differential clustering; the attack-reach figures in Table 3 are estimates by analogy.
- The key schedule has no proof; related-key security in particular is not established.
- No automated search (MILP or SAT) for impossible differentials, integrals or division properties has been done yet.
- The algebraic simplicity of the inversion S-box is a known trade-off.
- The constant-time implementation is slow, and side channels other than timing (power, electromagnetic) have not been studied.
- The comparison with Kalyna uses a test battery designed by us.
- Most importantly, the cipher needs independent public cryptanalysis. We invite it.

## 10. Conclusion

We built a 512-bit-block SPN cipher in which each component was chosen for a stated reason and checked by proof, exhaustive computation, or a clearly labelled estimate; whose constants can be regenerated by anyone from a public Tamil seed; and whose implementations agree across two languages and two platforms. Against Kalyna-512/512 under identical tests it shows stronger differential and linear bounds, reproducible constants and a key schedule without rotational relations, and a weaker algebraic profile of its S-box. Every program and every output is included so that each claim can be re-checked.

## Author contributions and use of AI

**Author (S. Ravichandran Ayer):** research direction (building a cipher feature by feature, each with mathematical justification); the requirement of a 256-bit minimum key; the idea of public parameters chosen through Tamil text; the choice of the 512-bit wide-block design over the 128-bit alternative; the decision to compare with Kalyna using identical tests and to add a constant-time implementation; independent execution of every program on an Android phone and confirmation of all results.

**AI assistance:** Claude (Anthropic) was used throughout as a research assistant: it proposed design options, wrote the Python and C programs, ported Kalyna, selected the cited theorems, and drafted this text. All numerical results were produced by the programs listed in the appendices and re-run independently by the author. The author takes responsibility for the content.

## Code and data availability

All programs listed in Appendix A, together with `newcipher.c`, `newcipher_ct.c` and the merged Kalyna reference `kalyna_all.c`, are released with this paper. Each Python program is self-contained and runs with the standard library only (Python 3.8 or later).

## Conflict of interest

The author declares no conflict of interest. This research received no funding.

## References

- [1] A. Kerckhoffs, "La cryptographie militaire," *Journal des sciences militaires*, vol. 9, pp. 5–38, 1883.
- [2] C. E. Shannon, "Communication theory of secrecy systems," *Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

- [3] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer, 2002.
- [4] J. Daemen and V. Rijmen, "The wide trail design strategy," in *Cryptography and Coding 2001*, LNCS 2260, Springer, pp. 222–238, 2001.
- [5] K. Nyberg, "Differentially uniform mappings for cryptography," in *EUROCRYPT '93*, LNCS 765, Springer, pp. 55–64, 1994.
- [6] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. North-Holland, 1977.
- [7] A. M. Youssef, S. Mister and S. E. Tavares, "On the design of linear transformations for substitution permutation encryption networks," in *Workshop on Selected Areas in Cryptography (SAC '97)*, pp. 40–48, 1997.
- [8] R. Oliynykov, I. Gorbenko, O. Kazymyrov, V. Ruzhentsev, O. Kuznetsov, Y. Gorbenko, O. Dyrda, V. Dolgov, A. Pushkaryov, R. Mordvinov and D. Kaidalov, "A new encryption standard of Ukraine: The Kalyna block cipher," *IACR Cryptology ePrint Archive*, Report 2015/650, 2015.
- [9] R. Kiianchuk, R. Mordvinov and R. Oliynykov, Reference implementation of the Kalyna block cipher (DSTU 7624:2014). <https://github.com/Roman-Oliynykov/Kalyna-reference>
- [10] Akshima, D. Chang, M. Ghosh, A. Goel and S. K. Sanadhya, "Single key recovery attacks on 9-round Kalyna-128/256 and Kalyna-256/512," in *ICISC 2015*, LNCS 9558, Springer, 2016. *IACR ePrint* 2015/1227.
- [11] R. AlTawy, A. Abdelkhalek and A. M. Youssef, "A meet-in-the-middle attack on reduced-round Kalyna-b/2b," *IEICE Transactions on Information and Systems*, vol. E99-D, no. 4, 2016. *IACR ePrint* 2015/762.
- [12] C. Chuengsatiansup, D. Genkin, Y. Yarom and Z. Zhang, "Side-channeling the Kalyna key expansion," in *CT-RSA 2022*, LNCS 13161, Springer, 2022.
- [13] E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems," *Journal of Cryptology*, vol. 4, no. 1, pp. 3–72, 1991.
- [14] M. Matsui, "Linear cryptanalysis method for DES cipher," in *EUROCRYPT '93*, LNCS 765, Springer, pp. 386–397, 1994.
- [15] J. Daemen, L. R. Knudsen and V. Rijmen, "The block cipher Square," in *FSE '97*, LNCS 1267, Springer, pp. 149–165, 1997.
- [16] L. R. Knudsen, "Truncated and higher order differentials," in *FSE '94*, LNCS 1008, Springer, pp. 196–211, 1995.
- [17] N. T. Courtois and J. Pieprzyk, "Cryptanalysis of block ciphers with overdefined systems of equations," in *ASIACRYPT 2002*, LNCS 2501, Springer, pp. 267–287, 2002.
- [18] A. Bogdanov, D. Khovratovich and C. Rechberger, "Biclique cryptanalysis of the full AES," in *ASIACRYPT 2011*, LNCS 7073, Springer, pp. 344–371, 2011.
- [19] B. Preneel, R. Govaerts and J. Vandewalle, "Hash functions based on block ciphers: a synthetic approach," in *CRYPTO '93*, LNCS 773, Springer, pp. 368–378, 1994.
- [20] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th ACM STOC*, pp. 212–219, 1996.
- [21] M. Bellare, A. Desai, E. Jorjipii and P. Rogaway, "A concrete security treatment of symmetric encryption," in *Proc. 38th IEEE FOCS*, pp. 394–403, 1997.
- [22] M. Bellare and C. Namprempre, "Authenticated encryption: relations among notions and analysis of the generic composition paradigm," in *ASIACRYPT 2000*, LNCS 1976, Springer, pp. 531–545, 2000.
- [23] M. Bellare, R. Canetti and H. Krawczyk, "Keying hash functions for message authentication," in *CRYPTO '96*, LNCS 1109, Springer, pp. 1–15, 1996.
- [24] A. F. Webster and S. E. Tavares, "On the design of S-boxes," in *CRYPTO '85*, LNCS 218, Springer, pp. 523–534, 1986.
- [25] A. Rukhin et al., "A statistical test suite for random and pseudorandom number generators for cryptographic applications," *NIST Special Publication 800-22 Rev. 1a*, 2010.
- [26] O. Reparaz, J. Balasch and I. Verbauwhede, "Dude, is my code constant time?" in *Design, Automation & Test in Europe (DATE)*, pp. 1697–1702, 2017.

- [27] A. Langley, "ctgrind: checking that functions are constant time with Valgrind," 2010.  
<https://github.com/ag/ctgrind>
- [28] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)," FIPS PUB 197, 2001.
- [29] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," FIPS PUB 180-4, 2015.
- [30] C. Boura, A. Canteaut and C. De Cannière, "Higher-order differential properties of Keccak and Luffa," in FSE 2011, LNCS 6733, Springer, pp. 252–269, 2011.
- [31] E. Biham, A. Biryukov and A. Shamir, "Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials," in EUROCRYPT '99, LNCS 1592, Springer, pp. 12–23, 1999.
- [32] D. J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hülsing, E. Lambooi, T. Lange, R. Niederhagen and C. van Vredendaal, "How to manipulate curve standards: a white paper for the black hat," in Security Standardisation Research (SSR 2015), LNCS 9497, Springer, pp. 109–139, 2015.

## Appendix A. Programs

Every program below is a complete, standalone Python file except where noted. Programs A.8 and A.9 also contain, unchanged, the cipher of program A.7. Programs A.10–A.11 come from one file (compare\_kalyna.py) and A.12–A.13 from another (battle\_test.py); both files also contain the cipher of A.7 and the Kalyna port of A.10, and battle\_test.py contains a compact copy of the mode of A.8. To save space, these repeated parts are not printed again.

### A.1 Step 1 — Kerckhoffs's principle (step1\_kerckhoffs.py)

```
# NEW-CIPHER v0.2 -- Step 1: Kerckhoffs's principle
# Everything in this file is PUBLIC. Only the key is secret.

import secrets
import hashlib
import math

# ----- Public specification -----
BLOCK_BITS = 128          # 128-bit block (64-bit blocks hit the birthday bound at 2^32 blocks)
KEY_BITS = 256            # 256-bit secret key
ROUNDS = None             # decided later, in the security-margin step

# Public seed for all constants ("nothing-up-my-sleeve").
# Anyone can recompute every constant from this text, so no hidden trapdoor is possible.
PUBLIC_SEED = "NEW-CIPHER v0.2 | தமிழ் | public constants"

def generate_key():
    """Secret key: 256 bits from the operating system's CSPRNG (never a password, never a formula)."""
    return secrets.token_bytes(KEY_BITS // 8)

def public_constant(label, index, bits=128):
    """Reproducible public constant = first bits of SHA-256(seed | label | index)."""
    h = hashlib.sha256(f"{PUBLIC_SEED}|{label}|{index}".encode("utf-8")).digest()
    return int.from_bytes(h, "big") >> (256 - bits)

def spec_fingerprint():
    """Hash of the public spec. Publish it so everyone checks they have the same algorithm."""
    spec = f"{BLOCK_BITS}|{KEY_BITS}|{PUBLIC_SEED}"
    return hashlib.sha256(spec.encode("utf-8")).hexdigest()

# ----- Mathematical validation for this step -----
def validate():
    print("=== Step 1: Kerckhoffs validation ===")
    print("Spec fingerprint :", spec_fingerprint())

    # 1. Key entropy: uniform key over {0,1}^256 -> H(K) = 256 bits
    print(f"Key entropy H(K) : {KEY_BITS} bits (key drawn uniformly by CSPRNG)")

    # 2. Brute force: expected trials = 2^(n-1)
    expected = 2 ** (KEY_BITS - 1)
    rate = 10 ** 18          # 1 billion billion keys per second
    years = expected / rate / (365.25 * 24 * 3600)
    print(f"Brute force      : 2^{KEY_BITS - 1} trials expected")
    print(f"At 10^18 keys/s    : about 10^{math.log10(years):.1f} years")

    # 3. Birthday bound for the block size
    print(f"Block birthday     : data limit about 2^{BLOCK_BITS // 2} blocks per key")

    # 4. Constants are reproducible (same input -> same constant, on any machine)
    c1 = public_constant("RC", 0)
    c2 = public_constant("RC", 0)
```

```

print("Constant RC[0]  :", hex(c1), "| reproducible:", c1 == c2)

# 5. Sanity check on keys (a check, not a proof): bit balance over many keys
n_keys = 2000
ones = sum(bin(int.from_bytes(generate_key(), "big")).count("1") for _ in range(n_keys))
total = n_keys * KEY_BITS
print(f"Key bit balance  : {ones / total:.4f} (ideal 0.5000)")

k = generate_key()
print("Sample key (hex) :", k.hex())

if __name__ == "__main__":
    validate()

```

## A.2 Step 2 — S-box construction and measurement (step2\_sbox.py)

```

# NEW-CIPHER v0.2 -- Step 2: S-box (confusion)
# Proven core: inversion in GF(2^8) (the same core as AES, Nyberg 1993).
# Public Tamil-seeded part: the affine layer (matrix + constant), chosen from SHA-256 of the seed.
# An affine layer does NOT change DU or NL, so the proven bounds carry over exactly.

import hashlib

PUBLIC_SEED = "NEW-CIPHER v0.2 | தமிழ் | public constants" # same seed as Step 1
POLY = 0x11B # irreducible polynomial x^8 + x^4 + x^3 + x + 1

def gf_mul(a, b):
    r = 0
    while b:
        if b & 1:
            r ^= a
        a <<= 1
        if a & 0x100:
            a ^= POLY
        b >>= 1
    return r

def gf_inv(a):
    if a == 0:
        return 0
    r, e, base = 1, 254, a # a^254 = a^-1 in GF(2^8)
    while e:
        if e & 1:
            r = gf_mul(r, base)
        base = gf_mul(base, base)
        e >>= 1
    return r

def rotl8(x, n):
    return ((x << n) | (x >> (8 - n))) & 0xFF

def affine(x, m, c):
    """y = M*x + c, where M is the circulant matrix whose first row is the byte m."""
    y = 0
    for i in range(8):
        if (m >> i) & 1:
            y ^= rotl8(x, i)
    return y ^ c

```

```

def seed_bytes(label, i):
    return hashlib.sha256(f"{PUBLIC_SEED}|{label}|{i}".encode("utf-8")).digest()

def build_sbox():
    """Try seed indices in order; take the first affine layer that passes every structural rule."""
    for i in range(1000):
        h = seed_bytes("SBOX", i)
        m, c = h[0], h[1]
        if bin(m).count("1") % 2 == 0:      # circulant over GF(2) is invertible iff weight is odd
            continue
        if bin(m).count("1") == 1:         # pure rotation is too weak a mix; skip
            continue
        s = [affine(gf_inv(x), m, c) for x in range(256)]
        if len(set(s)) != 256:
            continue
        if any(s[x] == x or s[x] == x ^ 0xFF for x in range(256)): # no fixed / opposite-fixed points
            continue
        return s, m, c, i
    raise RuntimeError("no valid affine layer found")

# ----- Measurements -----
def differential_uniformity(s):
    du = 0
    for a in range(1, 256):
        counts = [0] * 256
        for x in range(256):
            counts[s[x] ^ s[x ^ a]] += 1
        du = max(du, max(counts))
    return du

def walsh_max(s):
    """Max |Walsh| over all nonzero output masks b and all input masks a (fast Walsh transform)."""
    best = 0
    for b in range(1, 256):
        f = [1 - 2 * (bin(s[x] & b).count("1") & 1) for x in range(256)]
        h = 1
        while h < 256:
            for i in range(0, 256, h * 2):
                for j in range(i, i + h):
                    u, v = f[j], f[j + h]
                    f[j], f[j + h] = u + v, u - v
            h *= 2
        best = max(best, max(abs(v) for v in f))
    return best

def algebraic_degree(s):
    """Minimum algebraic degree over output bits (Moebius transform)."""
    degs = []
    for bit in range(8):
        f = [(s[x] >> bit) & 1 for x in range(256)]
        h = 1
        while h < 256:
            for i in range(0, 256, h * 2):
                for j in range(i, i + h):
                    f[j + h] ^= f[j]
            h *= 2
        degs.append(max(bin(u).count("1") for u in range(256) if f[u]))
    return min(degs)

def main():
    s, m, c, idx = build_sbox()

```

```

du = differential_uniformity(s)
w = walsh_max(s)
nl = 128 - w // 2
deg = algebraic_degree(s)
dp = du / 256
lp = (w / 256) ** 2

print("=== Step 2: S-box validation ===")
print(f"Affine layer      : M row = 0x{m:02X}, c = 0x{c:02X} (seed index {idx})")
print(f"Bijjective       : {len(set(s)) == 256}")
print(f"Fixed points     : {sum(s[x] == x for x in range(256))}, opposite: {sum(s[x] == x ^ 0xFF for x in range(256))}")
print(f"Diff. uniformity : {du}    (AES = 4, best known for 8-bit)")
print(f"Max DP           : {du}/256 = 2^{__import__('math').log2(dp):.0f}")
print(f"Nonlinearity     : {nl}    (AES = 112)")
print(f"Max |Walsh|      : {w} -> max LP = 2^{__import__('math').log2(lp):.0f}")
print(f"Algebraic degree : {deg}    (AES = 7, max for a bijection on 8 bits)")
print(f"S-box first row  :", " ".join(f"{v:02X}" for v in s[:16]))

sbox_hash = hashlib.sha256(bytes(s)).hexdigest()
print("S-box fingerprint:", sbox_hash)

if __name__ == "__main__":
    main()

```

### A.3 Step 3 — 4×4 MDS baseline for a 128-bit state (step3\_diffusion.py)

```

# NEW-CIPHER v0.2 -- Step 3: Diffusion layer
# State: 128 bits = 16 bytes = 4x4 byte matrix (column-major, like AES).
# Layer = ByteShift (row r rotates left by r) + ColumnMix (4x4 circulant MDS matrix over GF(2^8)).
# The circulant's 4 entries are chosen from the public Tamil seed, then PROVEN to be MDS.

import hashlib
from itertools import combinations

PUBLIC_SEED = "NEW-CIPHER v0.2 | தமிழ் | public constants" # same seed as Steps 1-2
POLY = 0x11B

def gf_mul(a, b):
    r = 0
    while b:
        if b & 1:
            r ^= a
        a <<= 1
        if a & 0x100:
            a ^= POLY
        b >>= 1
    return r

def gf_inv(a):
    r, e, base = 1, 254, a
    while e:
        if e & 1:
            r = gf_mul(r, base)
        base = gf_mul(base, base)
        e >>= 1
    return r

def det_nonzero(m):
    """Gaussian elimination over GF(2^8): True iff the square matrix is invertible."""
    m = [row[:] for row in m]

```

```

n = len(m)
for col in range(n):
    piv = next((r for r in range(col, n) if m[r][col]), None)
    if piv is None:
        return False
    m[col], m[piv] = m[piv], m[col]
    inv = gf_inv(m[col][col])
    for r in range(col + 1, n):
        if m[r][col]:
            f = gf_mul(m[r][col], inv)
            m[r] = [m[r][k] ^ gf_mul(f, m[col][k]) for k in range(n)]
return True

def is_mds(m):
    """A matrix is MDS iff EVERY square submatrix is invertible (MacWilliams-Sloane)."""
    n = len(m)
    count = 0
    for k in range(1, n + 1):
        for rows in combinations(range(n), k):
            for cols in combinations(range(n), k):
                count += 1
                if not det_nonzero([m[r][c] for c in cols] for r in rows):
                    return False, count
    return True, count

def circulant(v):
    n = len(v)
    return [[v[(c - r) % n] for c in range(n)] for r in range(n)]

def mat_vec(m, x):
    out = []
    for row in m:
        acc = 0
        for a, b in zip(row, x):
            acc ^= gf_mul(a, b)
        out.append(acc)
    return out

def mat_inverse(m):
    n = len(m)
    a = [row[:] + [1 if i == j else 0 for j in range(n)] for i, row in enumerate(m)]
    for col in range(n):
        piv = next(r for r in range(col, n) if a[r][col])
        a[col], a[piv] = a[piv], a[col]
        inv = gf_inv(a[col][col])
        a[col] = [gf_mul(inv, v) for v in a[col]]
        for r in range(n):
            if r != col and a[r][col]:
                f = a[r][col]
                a[r] = [a[r][k] ^ gf_mul(f, a[col][k]) for k in range(2 * n)]
    return [row[n:] for row in a]

def build_mix():
    for i in range(1000):
        h = hashlib.sha256(f"{PUBLIC_SEED}|MIX|{i}".encode("utf-8")).digest()
        v = list(h[:4])
        if len(set(v)) < 4 or 0 in v:
            continue
        m = circulant(v)
        ok, _ = is_mds(m)
        if ok:

```

```

        return m, v, i
    raise RuntimeError("no MDS circulant found")

def wt(v):
    return sum(1 for b in v if b)

def byteshift_map():
    """Position p = 4*col + row -> new position after row r rotates left by r."""
    mp = {}
    for col in range(4):
        for row in range(4):
            mp[4 * col + row] = 4 * ((col - row) % 4) + row
    return mp

def main():
    print("=== Step 3: Diffusion validation ===")
    m, v, idx = build_mix()
    print("Mix vector      :", " ".join(f"0x{b:02X}" for b in v), f"(seed index {idx})")

    # 1. MDS proof: all 69 square submatrices invertible
    ok, n_sub = is_mds(m)
    print(f"MDS (all {n_sub} square submatrices invertible): {ok}")
    print("Branch number      : 5 (= n+1, the maximum, follows from MDS)")

    # 2. Linear branch number uses the transpose; check it too
    mt = [list(r) for r in zip(*m)]
    print(f"Transpose MDS      : {is_mds(mt)[0]} -> linear branch number also 5")

    # 3. Invertible for decryption
    mi = mat_inverse(m)
    test = [0x12, 0x34, 0x56, 0x78]
    print(f"Inverse exists    : {mat_vec(mi, mat_vec(m, test)) == test}")

    # 4. Exhaustive confirmation for 1- and 2-byte inputs (empirical check of branch number)
    best = 99
    for p in range(4):
        for a in range(1, 256):
            x = [0] * 4
            x[p] = a
            best = min(best, 1 + wt(mat_vec(m, x)))
    for p, q in combinations(range(4), 2):
        for a in range(1, 256):
            for b in range(1, 256):
                x = [0] * 4
                x[p], x[q] = a, b
                best = min(best, 2 + wt(mat_vec(m, x)))
    print(f"Exhaustive min wt(x)+wt(Mx), wt(x)<=2 : {best}")

    # 5. ByteShift sends the 4 bytes of each column to 4 DIFFERENT columns (diffusion-optimal)
    mp = byteshift_map()
    optimal = all(len({mp[4 * c + r] // 4 for r in range(4)}) == 4 for c in range(4))
    print(f"ByteShift diffusion-optimal : {optimal}")

    # 6. Full diffusion: after how many rounds does every output byte depend on every input byte?
    dep = [{i} for i in range(16)]
    for rnd in range(1, 4):
        shifted = [None] * 16
        for p in range(16):
            shifted[mp[p]] = dep[p]
        new = []
        for col in range(4):
            union = set().union(*shifted[4 * col:4 * col + 4])
            new += [set(union) for _ in range(4)]

```

```

    dep = new
    full = all(len(d) == 16 for d in dep)
    print(f"Round {rnd}: min input bytes per output byte = {min(len(d) for d in dep):2d} full={full}")
    if full:
        break

# 7. Wide-trail consequence (proved in Step 4)
print("Wide trail          : any 4-round trail has >= 5 x 5 = 25 active S-boxes")

if __name__ == "__main__":
    main()

```

#### A.4 Step 3B — 8×8 Cauchy MDS for the 512-bit state (step3b\_mds8.py)

```

# NEW-CIPHER v0.2-B -- Step 3B: 8x8 MDS diffusion for a 512-bit block
# State: 512 bits = 64 bytes = 8x8 byte matrix (column-major).
# Layer = ByteShift (row r rotates left by r) + ColumnMix (8x8 Cauchy matrix over GF(2^8)).
#
# Theorem (Cauchy matrix): if x_0..x_7, y_0..y_7 are 16 DISTINCT field elements, then
#   M[i][j] = 1 / (x_i + y_j)
# has every square submatrix invertible, so M is MDS (branch number 9).
# The 16 elements are chosen from the public Tamil seed; the program then CHECKS all 12,869 submatrices.

import hashlib
import random
from itertools import combinations

PUBLIC_SEED = "NEW-CIPHER v0.2 | தமிழ் | public constants" # same seed as Steps 1-4
POLY = 0x11B
N = 8

def gf_mul_slow(a, b):
    r = 0
    while b:
        if b & 1:
            r ^= a
        a <<= 1
        if a & 0x100:
            a ^= POLY
        b >>= 1
    return r

MUL = [[gf_mul_slow(a, b) for b in range(256)] for a in range(256)]
INV = [0] * 256
for a in range(1, 256):
    for b in range(1, 256):
        if MUL[a][b] == 1:
            INV[a] = b
            break

def det_nonzero(m):
    m = [row[:] for row in m]
    n = len(m)
    for col in range(n):
        piv = next((r for r in range(col, n) if m[r][col]), None)
        if piv is None:
            return False
        m[col], m[piv] = m[piv], m[col]
        inv = INV[m[col][col]]
        for r in range(col + 1, n):
            if m[r][col]:

```

```

        f = MUL[m[r][col]][inv]
        rowc = m[col]
        m[r] = [m[r][k] ^ MUL[f][rowc[k]] for k in range(n)]
    return True

def is_mds(m):
    n = len(m)
    count = 0
    for k in range(1, n + 1):
        for rows in combinations(range(n), k):
            for cols in combinations(range(n), k):
                count += 1
                if not det_nonzero([m[r][c] for c in cols for r in rows]):
                    return False, count
    return True, count

def mat_vec(m, x):
    out = []
    for row in m:
        acc = 0
        for a, b in zip(row, x):
            acc ^= MUL[a][b]
        out.append(acc)
    return out

def mat_inverse(m):
    n = len(m)
    a = [row[:] + [1 if i == j else 0 for j in range(n)] for i, row in enumerate(m)]
    for col in range(n):
        piv = next(r for r in range(col, n) if a[r][col])
        a[col], a[piv] = a[piv], a[col]
        inv = INV[a[col][col]]
        a[col] = [MUL[inv][v] for v in a[col]]
        for r in range(n):
            if r != col and a[r][col]:
                f = a[r][col]
                a[r] = [a[r][k] ^ MUL[f][a[col][k]] for k in range(2 * n)]
    return [row[n:] for row in a]

def build_cauchy():
    """Take bytes from SHA-256(seed|CAUCHY|i) in order, keep the first 16 distinct ones."""
    seen = []
    i = 0
    while len(seen) < 2 * N:
        for b in hashlib.sha256(f"{PUBLIC_SEED}|CAUCHY|{i}".encode("utf-8")).digest():
            if b not in seen:
                seen.append(b)
                if len(seen) == 2 * N:
                    break
        i += 1
    x, y = seen[:N], seen[N:]
    m = [[INV[xi ^ yj] for yj in y] for xi in x]    # x_i + y_j is XOR in GF(2^8), never 0
    return m, x, y

def wt(v):
    return sum(1 for b in v if b)

def byteshift_map():
    mp = {}
    for col in range(N):

```

```

        for row in range(N):
            mp[N * col + row] = N * ((col - row) % N) + row
        return mp

def main():
    print("=== Step 3B: 8x8 MDS validation (512-bit block) ===")
    m, x, y = build_cauchy()
    print("Cauchy x :", " ".join(f"{v:02X}" for v in x))
    print("Cauchy y :", " ".join(f"{v:02X}" for v in y))
    print("All 16 distinct:", len(set(x + y)) == 16)
    print("Row 0      :", " ".join(f"{v:02X}" for v in m[0]))

    ok, n_sub = is_mds(m)
    print(f"MDS (all {n_sub} square submatrices invertible): {ok}")
    print("Branch number : 9 (= n+1, maximum for 8x8)")

    mt = [list(r) for r in zip(*m)]
    print(f"Transpose MDS : {is_mds(mt)[0]} -> linear branch number also 9")

    mi = mat_inverse(m)
    test = list(range(1, 9))
    print(f"Inverse exists: {mat_vec(mi, mat_vec(m, test)) == test}")

    # Exhaustive for 1-byte inputs
    best1 = min(1 + wt(mat_vec(m, [a if k == p else 0 for k in range(N)]))
               for p in range(N) for a in range(1, 256))
    print(f"Exhaustive min, wt(x)=1 : {best1}")

    # Random sample for 2..8-byte inputs (the theorem + 12,869 checks are the proof; this is a cross-check)
    rng = random.Random(2026)
    best = 99
    for _ in range(100000):
        k = rng.randint(2, N)
        pos = rng.sample(range(N), k)
        v = [0] * N
        for p in pos:
            v[p] = rng.randint(1, 255)
        best = min(best, k + wt(mat_vec(m, v)))
    print(f"Sampled min, 100,000 inputs : {best}")

    mp = byteshift_map()
    optimal = all(len({mp[N * c + r] // N for r in range(N)}) == N for c in range(N))
    print(f"ByteShift diffusion-optimal : {optimal}")

    dep = [{i} for i in range(N * N)]
    for rnd in range(1, 4):
        shifted = [None] * (N * N)
        for p in range(N * N):
            shifted[mp[p]] = dep[p]
        new = []
        for col in range(N):
            union = set().union(*shifted[N * col:N * col + N])
            new += [set(union) for _ in range(N)]
        dep = new
        full = all(len(d) == N * N for d in dep)
        print(f"Round {rnd}: min input bytes per output byte = {min(len(d) for d in dep):2d} full={full}")
        if full:
            break

    print("Wide trail : any 4-round trail has >= 9 x 9 = 81 active S-boxes -> DP, LP <= 2^486")
    print("Matrix fingerprint:", hashlib.sha256(bytes(v for row in m for v in row)).hexdigest())

if __name__ == "__main__":
    main()

```

## A.5 Step 4 — Trail bounds and round-count model (step4\_rounds.py)

```
# NEW-CIPHER v0.2 -- Step 4: Differential & linear resistance, and how many rounds
# Uses the wide trail bound (Daemen-Rijmen) with the S-box from Step 2 (max DP = max LP = 2-6)
# and the MDS layer from Step 3 (branch number B).
#
# Compares: block 128 / 256 bits, column of 4 bytes (B=5) or 8 bytes (B=9), key 256 / 512 / 1024 bits.
# Round model (heuristic, stated openly):
#   R_dist = rounds until best trail probability < 2-block (no usable differential/linear distinguisher)
#   R_kg    = extra rounds an attacker can peel off by guessing round keys = key_bits // block_bits
#   R_min   = R_dist + R_kg
#   R_rec   = ceil(1.5 * R_min) (50% security margin)

import math

SBOX_LOG2_P = 6          # each active S-box costs 2-6 (Step 2)

def active_sboxes(r, B):
    """Wide trail lower bound on active S-boxes over r rounds (diffusion-optimal shift)."""
    small = {0: 0, 1: 1, 2: B, 3: 2 * B - 1}
    return B * B * (r // 4) + small[r % 4]

def rounds_to_beat(bits, B):
    r = 1
    while SBOX_LOG2_P * active_sboxes(r, B) <= bits:
        r += 1
    return r

def main():
    print("=== Step 4: Differential / linear bounds ===")
    print("Active S-boxes (B=5):", [active_sboxes(r, 5) for r in range(1, 9)])
    print("Active S-boxes (B=9):", [active_sboxes(r, 9) for r in range(1, 9)])
    print(f"4-round bound (B=5): 25 x 2-6 = 2-{6 * 25} (B=9): 81 x 2-6 = 2-{6 * 81}")
    print()

    configs = [(128, 4, 5), (256, 4, 5), (512, 8, 9)] # wide trail needs columns >= rows: 8-byte columns need 8
    columns = 512-bit block
    header = f"{'Block':>5} {'Col':>3} {'B':>2} {'Key':>5} | {'R_dist':>6} {'R_kg':>4} {'R_min':>5} {'R_rec':>5} | "
    f"{'Grover':>6} {'Birthday':>8} {'Cost':>6}"
    print(header)
    print("-" * len(header))
    base = None
    for block, col, B in configs:
        for key in (256, 512, 1024):
            r_dist = rounds_to_beat(block, B)
            r_kg = key // block
            r_min = r_dist + r_kg
            r_rec = math.ceil(1.5 * r_min)
            cost = r_rec * (1.6 if col == 8 else 1.0) # rough work per byte of data
            if base is None:
                base = cost
            print(f"{block:>5} {col:>3} {B:>2} {key:>5} | {r_dist:>6} {r_kg:>4} {r_min:>5} {r_rec:>5} | "
                  f"2{key // 2:<4} 2{block // 2:<6} {cost / base:>5.1f}x")
    print()

    print("Notes:")
    print("- Grover: a quantum computer halves key security; 256-bit key -> 2128, already enough.")
    print("- Birthday: data limit per key; only a bigger BLOCK raises it, a bigger key does not.")
    print("- Cost: relative work per byte of data (128-bit block, 256-bit key = 1.0x).")
    print("- AES-like designs also face impossible-differential, MITM and related-key attacks,")
    print("  so the final round count is never set below the AES analogue (14 for 256-bit key).")
```

```
if __name__ == "__main__":
    main()
```

## A.6 Step 5 — Key schedule (step5\_keyschedule.py)

```
# NEW-CIPHER v0.2-B -- Step 5: Key schedule (256-bit key -> 15 round keys of 512 bits)
# Self-contained: rebuilds the Step 2 S-box and Step 3B matrix, and checks their fingerprints.
#
# Design:
# P(s, c) = MixColumns(ByteShift(SubBytes(s))) XOR c          (unkeyed round, public constant c)
# G(x, t) = 3 rounds of P with constants RC[t][0..2]
# T0      = K || (K XOR C_EXP)                                (512 bits)
# KT      = P^4(T0) XOR T0                                    (feed-forward: one-way)
# X_i     = KT XOR rotate_bytes(T0, 5*i) XOR RC_ROUND[i]
# RK_i    = G(X_i, i) XOR X_i                                (feed-forward: one-way)

import hashlib
import random
from itertools import combinations

PUBLIC_SEED = "NEW-CIPHER v0.2 | தமிழ் | public constants"
POLY = 0x11B
N = 8          # 8x8 bytes = 512-bit state
ROUNDS = 14    # number of cipher rounds -> ROUNDS + 1 round keys
KEY_BYTES = 32

SBOX_FP = "d6872e197e188811224b9667bda8a489b7065136be6ed9018918d1b11ceb46c0"
MDS_FP = "e999262f3d4cdddbaa0aa08e4b21115c116201bd62effc460bf41602a3139b8"

# ----- Field -----
def _mul(a, b):
    r = 0
    while b:
        if b & 1:
            r ^= a
        a <<= 1
        if a & 0x100:
            a ^= POLY
        b >>= 1
    return r

MUL = [[_mul(a, b) for b in range(256)] for a in range(256)]
INV = [0] * 256
for _a in range(1, 256):
    INV[_a] = next(b for b in range(1, 256) if MUL[_a][b] == 1)

def h(label, i):
    return hashlib.sha256(f"{PUBLIC_SEED}|{label}|{i}".encode("utf-8")).digest()

def h512(label, i):
    return h(label, f"{i}a") + h(label, f"{i}b")

# ----- Step 2 S-box -----
def rotl8(x, n):
    return ((x << n) | (x >> (8 - n))) & 0xFF

def build_sbox():
```

```

for i in range(1000):
    d = h("SB0X", i)
    m, c = d[0], d[1]
    w = bin(m).count("1")
    if w % 2 == 0 or w == 1:
        continue
    s = []
    for x in range(256):
        v = INV[x]
        y = 0
        for k in range(8):
            if (m >> k) & 1:
                y ^= rotl8(v, k)
        s.append(y ^ c)
    if len(set(s)) == 256 and not any(s[x] == x or s[x] == x ^ 0xFF for x in range(256)):
        return s
raise RuntimeError

# ----- Step 3B matrix -----
def build_mds():
    seen, i = [], 0
    while len(seen) < 2 * N:
        for b in h("CAUCHY", i):
            if b not in seen:
                seen.append(b)
                if len(seen) == 2 * N:
                    break
        i += 1
    x, y = seen[:N], seen[N:]
    return [[INV[xi ^ yj] for yj in y] for xi in x]

SB0X = build_sbox()
MDS = build_mds()
MDS_ROWS = [[MUL[c] for c in row] for row in MDS]    # row of lookup tables for speed

# ByteShift: position 8*col+row -> 8*((col-row)%8)+row
SHIFT_SRC = [0] * 64
for _col in range(N):
    for _row in range(N):
        SHIFT_SRC[N * ((_col - _row) % N) + _row] = N * _col + _row

# ----- Unkeyed round -----
def P(s, c):
    t = [SB0X[s[SHIFT_SRC[p]]] for p in range(64)]    # SubBytes + ByteShift
    out = [0] * 64
    for col in range(N):
        colv = t[N * col:N * col + N]
        for r in range(N):
            row = MDS_ROWS[r]
            acc = 0
            for k in range(N):
                acc ^= row[k][colv[k]]
            out[N * col + r] = acc ^ c[N * col + r]
    return out

def xor(a, b):
    return [x ^ y for x, y in zip(a, b)]

def rot_bytes(v, n):
    n %= len(v)
    return v[n:] + v[:n]

```

```

C_EXP = list(h("KEXP", 0))
C_KT = [list(h512("KT", j)) for j in range(4)]
RC_ROUND = [list(h512("RKROUND", i)) for i in range(ROUNDS + 1)]
RC_G = [[list(h512("RKG", f"{i}.{j}")) for j in range(3)] for i in range(ROUNDS + 1)]

def key_schedule(key):
    assert len(key) == KEY_BYTES
    k = list(key)
    t0 = k + xor(k, C_EXP)
    s = t0
    for j in range(4):
        s = P(s, C_KT[j])
    kt = xor(s, t0)
    rks = []
    for i in range(ROUNDS + 1):
        x = xor(xor(kt, rot_bytes(t0, 5 * i)), RC_ROUND[i])
        g = x
        for j in range(3):
            g = P(g, RC_G[i][j])
        rks.append(bytes(xor(g, x)))
    return rks

# ----- Validation -----
def hamming(a, b):
    return sum(bin(x ^ y).count("1") for x, y in zip(a, b))

def main():
    print("=== Step 5: Key schedule validation ===")
    sfp = hashlib.sha256(bytes(SBOX)).hexdigest()
    mfp = hashlib.sha256(bytes(v for row in MDS for v in row)).hexdigest()
    print("S-box matches Step 2 :", sfp == SBOX_FP)
    print("Matrix matches Step 3B:", mfp == MDS_FP)
    print(f"Rounds = {ROUNDS}, round keys = {ROUNDS + 1} x 512 bits")

    # 1. Test vector (all-zero key) -- publish this
    zero = bytes(32)
    rk0 = key_schedule(zero)
    print("Test vector RK[0] (zero key):", rk0[0][:16].hex(), "...")
    print("Schedule fingerprint (zero key):", hashlib.sha256(b"".join(rk0)).hexdigest()[:32])

    # 2. Weak-key patterns: round keys must still be distinct, non-zero, balanced
    patterns = {"all-zero": bytes(32), "all-one": bytes([0xFF] * 32),
               "repeat 0x55": bytes([0x55] * 32), "counter": bytes(range(32))}
    for name, key in patterns.items():
        rks = key_schedule(key)
        distinct = len(set(rks)) == len(rks)
        nonzero = all(any(r) for r in rks)
        bal = sum(bin(b).count("1") for r in rks for b in r) / (len(rks) * 512)
        print(f" {name:12s}: distinct={distinct} nonzero={nonzero} bit balance={bal:.3f}")

    # 3. Key avalanche: flip 1 key bit -> each round key should change about 256 of 512 bits
    rng = random.Random(5)
    diffs = []
    per_round_min = [512] * (ROUNDS + 1)
    for _ in range(4):
        key = bytes(rng.randrange(256) for _ in range(32))
        base = key_schedule(key)
        for bit in rng.sample(range(256), 32):
            kk = bytearray(key)
            kk[bit // 8] ^= 1 << (bit % 8)
            other = key_schedule(bytes(kk))

```

```

        for i in range(ROUNDS + 1):
            d = hamming(base[i], other[i])
            diffs.append(d)
            per_round_min[i] = min(per_round_min[i], d)
mean = sum(diffs) / len(diffs)
print(f"Key avalanche: mean {mean:.1f} / 512 bits changed ({mean / 512:.3f}, ideal 0.500)")
print(f"                min {min(diffs)}, max {max(diffs)} over {len(diffs)} round-key pairs")
print(f"                min per round key:", per_round_min)

# 4. Slide resistance: RK_i must never equal RK_j, also across a related key
key = bytes(rng.randrange(256) for _ in range(32))
a = key_schedule(key)
kk = bytearray(key)
kk[0] ^= 1
b = key_schedule(bytes(kk))
collide = any(a[i] == b[j] for i in range(ROUNDS + 1) for j in range(ROUNDS + 1))
print("Slide/related-key round-key collision:", collide)

# 5. Round keys are not linearly related: XOR of consecutive round keys looks balanced
bal = sum(bin(x ^ y).count("1") for i in range(ROUNDS) for x, y in zip(a[i], a[i + 1])) / (ROUNDS * 512)
print(f"Consecutive round-key difference balance: {bal:.3f} (ideal 0.500)")

if __name__ == "__main__":
    main()

```

## A.7 Step 6 — Full cipher (step6\_cipher.py)

```

# NEW-CIPHER v0.2-B -- Step 6: Full block cipher (512-bit block, 256-bit key, 14 rounds)
# Self-contained: rebuilds S-box (Step 2), matrix (Step 3B), key schedule (Step 5) and checks fingerprints.
#
# Encrypt:  s = P XOR RK[0]
#           for r = 1..14: s = MixColumns(ByteShift(SubBytes(s))) XOR RK[r]
# Decrypt:  exact inverse, using inverse S-box, inverse shift and inverse matrix.

import hashlib
import random
import time

PUBLIC_SEED = "NEW-CIPHER v0.2 | தமிழ் | public constants"
POLY = 0x11B
N = 8
ROUNDS = 14
BLOCK = 64          # bytes
KEY_BYTES = 32

SBX_FP = "d6872e197e188811224b9667bda8a489b7065136be6ed9018918d1b11ceb46c0"
MDS_FP = "e999262f3d4cdddbaa0aa08e4b211115c116201bd62effc460bf41602a3139b8"
KS_FP = "37404d0cc361bc31c8fb1bf8d5c7b180"

# ----- Field -----
def _mul(a, b):
    r = 0
    while b:
        if b & 1:
            r ^= a
        a <<= 1
        if a & 0x100:
            a ^= POLY
        b >>= 1
    return r

MUL = [[_mul(a, b) for b in range(256)] for a in range(256)]

```

```

INV = [0] * 256
for _a in range(1, 256):
    INV[_a] = next(b for b in range(1, 256) if MUL[_a][b] == 1)

def h(label, i):
    return hashlib.sha256(f"{PUBLIC_SEED}|{label}|{i}".encode("utf-8")).digest()

def h512(label, i):
    return h(label, f"{i}a") + h(label, f"{i}b")

# ----- Components -----
def rotl8(x, n):
    return ((x << n) | (x >> (8 - n))) & 0xFF

def build_sbox():
    for i in range(1000):
        d = h("SBOX", i)
        m, c = d[0], d[1]
        w = bin(m).count("1")
        if w % 2 == 0 or w == 1:
            continue
        s = []
        for x in range(256):
            v = INV[x]
            y = 0
            for k in range(8):
                if (m >> k) & 1:
                    y ^= rotl8(v, k)
            s.append(y ^ c)
        if len(set(s)) == 256 and not any(s[x] == x or s[x] == x ^ 0xFF for x in range(256)):
            return s
    raise RuntimeError

def build_mds():
    seen, i = [], 0
    while len(seen) < 2 * N:
        for b in h("CAUCHY", i):
            if b not in seen:
                seen.append(b)
            if len(seen) == 2 * N:
                break
        i += 1
    x, y = seen[:N], seen[N:]
    return [[INV[xi ^ yj] for yj in y] for xi in x]

def mat_inverse(m):
    n = len(m)
    a = [row[:] + [1 if i == j else 0 for j in range(n)] for i, row in enumerate(m)]
    for col in range(n):
        piv = next(r for r in range(col, n) if a[r][col])
        a[col], a[piv] = a[piv], a[col]
        inv = INV[a[col][col]]
        a[col] = [MUL[inv][v] for v in a[col]]
        for r in range(n):
            if r != col and a[r][col]:
                f = a[r][col]
                a[r] = [a[r][k] ^ MUL[f][a[col][k]] for k in range(2 * n)]
    return [row[n:] for row in a]

```

```

SBOX = build_sbox()
INV_SBOX = [0] * 256
for _x, _y in enumerate(SBOX):
    INV_SBOX[_y] = _x
MDS = build_mds()
MDS_INV = mat_inverse(MDS)
MDS_ROWS = [[MUL[c] for c in row] for row in MDS]
MDS_INV_ROWS = [[MUL[c] for c in row] for row in MDS_INV]

SHIFT_SRC = [0] * 64          # new[p] = old[SHIFT_SRC[p]]
for _col in range(N):
    for _row in range(N):
        SHIFT_SRC[N * ((_col - _row) % N) + _row] = N * _col + _row
UNSHIFT_SRC = [0] * 64
for _p in range(64):
    UNSHIFT_SRC[SHIFT_SRC[_p]] = _p

def mix(t, rows):
    out = [0] * 64
    for col in range(N):
        colv = t[N * col:N * col + N]
        for r in range(N):
            row = rows[r]
            acc = 0
            for k in range(N):
                acc ^= row[k][colv[k]]
            out[N * col + r] = acc
    return out

def xor(a, b):
    return [x ^ y for x, y in zip(a, b)]

def rot_bytes(v, n):
    n %= len(v)
    return v[n:] + v[:n]

# ----- Key schedule (Step 5) -----
C_EXP = list(h("KEXP", 0))
C_KT = [list(h512("KT", j)) for j in range(4)]
RC_ROUND = [list(h512("RKROUND", i)) for i in range(ROUNDS + 1)]
RC_G = [[list(h512("RKG", f"{i}.{j}")) for j in range(3)] for i in range(ROUNDS + 1)]

def P(s, c):
    return xor(mix([SBOX[s[SHIFT_SRC[p]]] for p in range(64)], MDS_ROWS), c)

def key_schedule(key):
    assert len(key) == KEY_BYTES
    k = list(key)
    t0 = k + xor(k, C_EXP)
    s = t0
    for j in range(4):
        s = P(s, C_KT[j])
    kt = xor(s, t0)
    rks = []
    for i in range(ROUNDS + 1):
        x = xor(xor(kt, rot_bytes(t0, 5 * i)), RC_ROUND[i])
        g = x
        for j in range(3):
            g = P(g, RC_G[i][j])
        rks.append(xor(g, x))

```

```

    return rks

# ----- Cipher -----
def encrypt_block(rks, pt, rounds=ROUNDS):
    s = xor(list(pt), rks[0])
    for r in range(1, rounds + 1):
        s = mix([SBOX[s[SHIFT_SRC[p]]] for p in range(64)], MDS_ROWS)
        s = xor(s, rks[r])
    return bytes(s)

def decrypt_block(rks, ct):
    s = list(ct)
    for r in range(ROUNDS, 0, -1):
        s = mix(xor(s, rks[r]), MDS_INV_ROWS)
        s = [INV_SBOX[s[UNSHIFT_SRC[p]]] for p in range(64)]
    return bytes(xor(s, rks[0]))

# ----- Validation -----
def hamming(a, b):
    return sum(bin(x ^ y).count("1") for x, y in zip(a, b))

def flip(data, bit):
    d = bytearray(data)
    d[bit // 8] ^= 1 << (bit % 8)
    return bytes(d)

def main():
    print("=== Step 6: Full cipher validation ===")
    sfp = hashlib.sha256(bytes(SBOX)).hexdigest()
    mfp = hashlib.sha256(bytes(v for row in MDS for v in row)).hexdigest()
    kfp = hashlib.sha256(b"".join(bytes(r) for r in key_schedule(bytes(32)))).hexdigest()[:32]
    print("S-box / matrix / key schedule match earlier steps:", sfp == SBOX_FP, mfp == MDS_FP, kfp == KS_FP)

    # 1. Official test vectors
    print("\nTest vectors (first 32 of 64 bytes shown):")
    tv = [{"zero key, zero plaintext", bytes(32), bytes(64)},
          ("counter key, counter pt", bytes(range(32)), bytes(range(64)))]
    for name, k, p in tv:
        c = encrypt_block(key_schedule(k), p)
        print(f" {name}: {c[:32].hex()}")
        print(f"    full-ct fingerprint: {hashlib.sha256(c).hexdigest()[:32]}")

    # 2. Correctness: D(E(P)) = P
    rng = random.Random(6)
    ok = True
    for _ in range(200):
        k = bytes(rng.randrange(256) for _ in range(32))
        p = bytes(rng.randrange(256) for _ in range(64))
        rks = key_schedule(k)
        ok &= decrypt_block(rks, encrypt_block(rks, p)) == p
    print(f"\nD(E(P)) == P for 200 random keys/blocks: {ok}")

    # 3. Avalanche per round (plaintext 1-bit flip): how fast does it reach 256/512?
    k = bytes(rng.randrange(256) for _ in range(32))
    rks = key_schedule(k)
    print("\nPlaintext avalanche by round (mean bits changed of 512, 200 samples):")
    for r in (1, 2, 3, 4, 14):
        tot = 0
        for _ in range(200):
            p = bytes(rng.randrange(256) for _ in range(64))
            tot += hamming(encrypt_block(rks, p, r), encrypt_block(rks, flip(p, rng.randrange(512)), r))

```

```

        print(f"   round {r:2d}: {tot / 200:6.1f}   ({tot / 200 / 512:.3f})")

# 4. Key avalanche on the full cipher
p = bytes(64)
tot, mn, mx = 0, 512, 0
for _ in range(100):
    k = bytes(rng.randrange(256) for _ in range(32))
    d = hamming(encrypt_block(key_schedule(k), p), encrypt_block(key_schedule(flip(k, rng.randrange(256))),
p))

    tot += d
    mn, mx = min(mn, d), max(mx, d)
print(f"\nKey avalanche (1 key bit): mean {tot / 100:.1f}/512 ({tot / 100 / 512:.3f}), min {mn}, max {mx}")

# 5. Strict Avalanche Criterion: every output bit should flip with probability 0.5
counts = [0] * 512
samples = 0
for _ in range(40):
    p = bytes(rng.randrange(256) for _ in range(64))
    c = encrypt_block(rks, p)
    for bit in rng.sample(range(512), 25):
        c2 = encrypt_block(rks, flip(p, bit))
        x = int.from_bytes(bytes(a ^ b for a, b in zip(c, c2)), "big")
        for j in range(512):
            counts[j] += (x >> j) & 1
        samples += 1
probs = [c / samples for c in counts]
print(f"SAC over {samples} flips: output-bit flip probability min {min(probs):.3f}, max {max(probs):.3f}
(ideal 0.500)")

# 6. Speed (pure Python reference, not optimised)
t = time.time()
n = 300
for _ in range(n):
    encrypt_block(rks, p)
dt = time.time() - t
print(f"\nSpeed (reference Python): {n * 64 / dt / 1024:.1f} KB/s")

if __name__ == "__main__":
    main()

```

## A.8 Step 7 — Authenticated encryption, mode part (step7\_aead.py)

```

# ===== Step 7 mode =====
NONCE = 32
TAG = 32
BLOCK = 64

def _const(label):
    d = hashlib.sha256(f"{PUBLIC_SEED}|MODE|{label}|a".encode()).digest()
    return d + hashlib.sha256(f"{PUBLIC_SEED}|MODE|{label}|b".encode()).digest()

def derive_keys(key):
    rks = key_schedule(key)
    k_enc = encrypt_block(rks, _const("ENC"))[:32]
    k_mac = encrypt_block(rks, _const("MAC"))
    return key_schedule(k_enc), k_mac

def _ctr(rks_enc, nonce, data):
    out = bytearray()
    for i in range(0, len(data), BLOCK):
        ks = encrypt_block(rks_enc, nonce + i.to_bytes(32, "big")) # 256-bit nonce || 256-bit counter

```

```

        chunk = data[i:i + BLOCK]
        out += bytes(a ^ b for a, b in zip(chunk, ks))
    return bytes(out)

def _tag(k_mac, ad, nonce, c):
    msg = len(ad).to_bytes(8, "big") + ad + nonce + len(c).to_bytes(8, "big") + c
    return hmac.new(k_mac, msg, hashlib.sha256).digest()

def seal(key, plaintext, ad=b"", nonce=None):
    rks_enc, k_mac = derive_keys(key)
    nonce = nonce or secrets.token_bytes(NONCE)
    c = _ctr(rks_enc, nonce, plaintext)
    return nonce + c + _tag(k_mac, ad, nonce, c)

def open_(key, sealed, ad=b ""):
    if len(sealed) < NONCE + TAG:
        return None
    nonce, c, tag = sealed[:NONCE], sealed[NONCE:-TAG], sealed[-TAG:]
    rks_enc, k_mac = derive_keys(key)
    if not hmac.compare_digest(tag, _tag(k_mac, ad, nonce, c)):
        return None # reject: no decryption happens
    return _ctr(rks_enc, nonce, c)

def flip(data, bit):
    d = bytearray(data)
    d[bit // 8] ^= 1 << (bit % 8)
    return bytes(d)

def main():
    print("=== Step 7: Authenticated encryption (IND-CCA) validation ===")
    rng = random.Random(7)
    key = bytes(rng.randrange(256) for _ in range(32))

    # 1. Round trip for edge-case lengths
    ok = True
    for n in (0, 1, 63, 64, 65, 200):
        m = bytes(rng.randrange(256) for _ in range(n))
        ok &= open_(key, seal(key, m, b"hdr"), b"hdr") == m
    print("Round trip (lengths 0,1,63,64,65,200):", ok)

    # 2. Tamil text
    msg = "வணக்கம் நண்பர் — NEW-CIPHER".encode("utf-8")
    s = seal(key, msg)
    print("Tamil message      :", open_(key, s).decode("utf-8"))

    # 3. Randomised: same message twice -> different ciphertexts (needed for IND-CPA)
    print("Same msg, 2 seals differ:", seal(key, msg) != seal(key, msg))

    # 4. CCA game: every single-bit change anywhere must be rejected
    s = seal(key, b"attack at dawn", b"ad")
    rejected = sum(open_(key, flip(s, b), b"ad") is None for b in range(len(s) * 8))
    print(f"Bit-flip forgeries rejected: {rejected}/{len(s) * 8}")
    print("Wrong AD rejected :", open_(key, s, b"AD") is None)
    print("Truncated rejected:", open_(key, s[:-1], b"ad") is None)
    print("Wrong key rejected:", open_(bytes(32), s, b"ad") is None)

    # 5. Why the nonce must never repeat (honest demonstration of the one rule users must follow)
    n0 = bytes(32)
    a, b = b"pay 100 rupees..", b"pay 900 rupees.."
    ca, cb = seal(key, a, nonce=n0), seal(key, b, nonce=n0)
    leak = bytes(x ^ y for x, y in zip(ca[32:48], cb[32:48]))

```

```

same = leak == bytes(x ^ y for x, y in zip(a, b))
print("Nonce reuse leaks M1 XOR M2 (rule: never reuse a nonce):", same)

# 6. Test vector
tv = seal(bytes(32), b"NEW-CIPHER", b"", nonce=bytes(32))
print("Test vector (zero key, zero nonce, 'NEW-CIPHER'):", tv[32:].hex())

if __name__ == "__main__":
    main()

```

## A.9 Step 8 — Other attacks, analysis part (step8\_attacks.py)

```

# ===== Step 8: Resistance to other attacks =====
import math

def integral_test(rks, rounds, trials, rng):
    """Square/integral attack: 256 plaintexts, byte 0 takes all values, other bytes fixed.
    A byte is 'balanced' if the XOR of its 256 ciphertext values is 0."""
    counts = []
    for _ in range(trials):
        base = [rng.randrange(256) for _ in range(64)]
        acc = [0] * 64
        for v in range(256):
            p = base[:]
            p[0] = v
            c = encrypt_block(rks, bytes(p), rounds)
            acc = [a ^ b for a, b in zip(acc, c)]
        counts.append(sum(1 for a in acc if a == 0))
    return counts

def main():
    print("=== Step 8: Resistance to other attacks ===")
    rng = random.Random(8)
    rks = key_schedule(bytes(rng.randrange(256) for _ in range(32)))

    # 1. Integral (Square) attack -- measured
    print("\n[1] Integral attack: balanced bytes out of 64 (5 trials each)")
    for r in range(1, 6):
        c = integral_test(rks, r, 5, rng)
        verdict = "DISTINGUISHER" if min(c) == 64 else "random (expected about 0.25)"
        print(f" {r} rounds: {c} -> {verdict}")
    print(" Theory: 3-round distinguisher with 2^8 texts; 4 rounds with 2^64 texts (one diagonal).")

    # 2. Algebraic degree growth (higher-order differential)
    print("\n[2] Algebraic degree upper bound min(7^r, 511):")
    for r in range(1, 6):
        d = min(7 ** r, 511)
        print(f" {r} rounds: {d:3d}" + (" <- saturated, higher-order differential needs > 2^511 texts" if d ==
511 else ""))

    # 3. Algebraic (Groebner/XSL-type) system size
    sb_cipher = ROUNDS * 64
    sb_ks = (4 + 3 * (ROUNDS + 1)) * 64
    sboxes = sb_cipher + sb_ks
    eqs = 39 * sboxes
    vars_ = 8 * sboxes + 256
    print(f"\n[3] Algebraic system: {sboxes} S-boxes -> {eqs:,} quadratic equations in {vars_:,} variables over
GF(2)")
    print(" (inversion S-box: 39 quadratic equations each). No method solves systems of this size;")
    print(" even the far smaller AES-128 system (~8,000 equations) remains unsolved.")

    # 4. Differential/linear useful range vs. a 2^256 work budget

```

```

print("\n[4] Differential/linear trails vs. 2^256 work (B = 9, 2^-6 per S-box):")
small = {0: 0, 1: 1, 2: 9, 3: 17}
for r in range(1, 6):
    a = 81 * (r // 4) + small[r % 4]
    print(f" {r} rounds: >= {a:3d} active S-boxes -> trail prob <= 2^-{{6 * a}}"
          + (" usable" if 6 * a < 256 else " beyond 2^256: useless"))

# 5. Summary of estimated attackable rounds
print("\n[5] Estimated reach (distinguisher + key-recovery rounds) vs 14 rounds:")
table = [("Differential / linear", 3, 2), ("Integral (Square)", 4, 3),
         ("Impossible differential", 4, 3), ("Higher-order differential", 3, 2)]
worst = 0
for name, d, k in table:
    reach = d + k
    worst = max(worst, reach)
    print(f" {name:26s}: {d} + {k} = {reach} rounds")
print(" Meet-in-the-middle/biclique: every round key depends on the full key, so no clean split;")
print(" by analogy with AES, a biclique might lower 2^256 by about 2 bits -- still infeasible.")
print(f" Worst estimate: {worst} of {ROUNDS} rounds -> security margin {ROUNDS - worst} rounds ({(ROUNDS - worst) / ROUNDS:.0%})")
print(" These are ESTIMATES by analogy with published AES/Kalyna results, not proofs.")

if __name__ == "__main__":
    main()

```

## A.10 Kalyna-512/512 Python port (from compare\_kalyna.py)

```

# Kalyna-512/512 (DSTU 7624:2014) -- Python port of the official C reference
# (github.com/Roman-Oliynykov/Kalyna-reference), checked against its test vectors.
# =====

```

```

KPOLY = 0x11D
K_NR = 18

```

```

def _kmul(a, b):
    r = 0
    while b:
        if b & 1:
            r ^= a
        a <<= 1
        if a & 0x100:
            a ^= KPOLY
        b >>= 1
    return r

```

```

KMUL = [[_kmul(a, b) for b in range(256)] for a in range(256)]
_ks = [ # the four official Kalyna S-boxes (tables.c of the reference code), 4 x 256 values

```

```

168, 67, 95, 6, 107, 117, 108, 89, 113, 223, 135, 149, 23, 240, 216, 9, 109, 243, 29, 203, 201, 77, 44, 175, 121, 224, 151, 253, 111, 75, 69, 57,
62, 221, 163, 79, 180, 182, 154, 14, 31, 191, 21, 225, 73, 210, 147, 198, 146, 114, 158, 97, 209, 99, 250, 238, 244, 25, 213, 173, 88, 164, 187,
161, 220, 242, 131, 55, 66, 228, 122, 50, 156, 204, 171, 74, 143, 110, 4, 39, 46, 231, 226, 90, 150, 22, 35, 43, 194, 101, 102, 15, 188, 169, 71,
65, 52, 72, 252, 183, 106, 136, 165, 83, 134, 249, 91, 219, 56, 123, 195, 30, 34, 51, 36, 40, 54, 199, 178, 59, 142, 119, 186, 245, 20, 159, 8, 85
, 155, 76, 254, 96, 92, 218, 24, 70, 205, 125, 33, 176, 63, 27, 137, 255, 235, 132, 105, 58, 157, 215, 211, 112, 103, 64, 181, 222, 93, 48, 145, 1
77, 120, 17, 1, 229, 0, 104, 152, 160, 197, 2, 166, 116, 45, 11, 162, 118, 179, 190, 206, 189, 174, 233, 138, 49, 28, 236, 241, 153, 148, 170, 24
6, 38, 47, 239, 232, 140, 53, 3, 212, 127, 251, 5, 193, 94, 144, 32, 61, 130, 247, 234, 10, 13, 126, 248, 80, 26, 196, 7, 87, 184, 60, 98, 227, 200
, 172, 82, 100, 16, 208, 217, 19, 12, 18, 41, 81, 185, 207, 214, 115, 141, 129, 84, 192, 237, 78, 68, 167, 42, 133, 37, 230, 202, 124, 139, 86, 12
8, 206, 187, 235, 146, 234, 203, 19, 193, 233, 58, 214, 178, 210, 144, 23, 248, 66, 21, 86, 180, 101, 28, 136, 67, 197, 92, 54, 186, 245, 87, 103
, 141, 49, 246, 100, 88, 158, 244, 34, 170, 117, 15, 2, 177, 223, 109, 115, 77, 124, 38, 46, 247, 8, 93, 68, 62, 159, 20, 200, 174, 84, 16, 216, 18
8, 26, 107, 105, 243, 189, 51, 171, 250, 209, 155, 104, 78, 22, 149, 145, 238, 76, 99, 142, 91, 204, 60, 25, 161, 129, 73, 123, 217, 111, 55, 96,
202, 231, 43, 72, 253, 150, 69, 252, 65, 18, 13, 121, 229, 137, 140, 227, 32, 48, 220, 183, 108, 74, 181, 63, 151, 212, 98, 45, 6, 164, 165, 131,
95, 42, 218, 201, 0, 126, 162, 85, 191, 17, 213, 156, 207, 14, 10, 61, 81, 125, 147, 27, 254, 196, 71, 9, 134, 11, 143, 157, 106, 7, 185, 176, 152
, 24, 50, 113, 75, 239, 59, 112, 160, 228, 64, 255, 195, 169, 230, 120, 249, 139, 70, 128, 30, 56, 225, 184, 168, 224, 12, 35, 118, 29, 37, 36, 5,
241, 110, 148, 40, 154, 132, 232, 163, 79, 119, 211, 133, 226, 82, 242, 130, 80, 122, 47, 116, 83, 179, 97, 175, 57, 53, 222, 205, 31, 153, 172,

```

```

173,114,44,221,208,135,190,94,166,236,4,198,3,52,251,219,89,182,194,1,240,90,237,167,102,33,127,138,39,199,192,41,
215,147,217,154,181,152,34,69,252,186,106,223,2,159,220,81,89,74,23,43,194,148,244,187,163,98,228,113,212,205,112,
22,225,73,60,192,216,92,155,173,133,83,161,122,200,45,224,209,114,166,44,196,227,118,120,183,180,9,59,14,65,76,222
,178,144,37,165,215,3,17,0,195,46,146,239,78,18,157,125,203,53,16,213,79,158,77,169,85,198,208,123,24,151,211,54,2
30,72,86,129,143,119,204,156,185,226,172,184,47,21,164,124,218,56,30,11,5,214,20,110,108,126,102,253,177,229,96,17
5,94,51,135,201,240,93,109,63,136,141,199,247,29,233,236,237,128,41,39,207,153,168,80,15,55,36,40,48,149,210,62,91
,64,131,179,105,87,31,7,28,138,188,32,235,206,142,171,238,49,162,115,249,202,58,26,251,13,193,254,250,242,111,189,
150,221,67,82,182,8,243,174,190,25,137,50,38,176,234,75,100,132,130,107,245,121,191,1,95,117,99,27,35,61,104,42,10
1,232,145,246,255,19,88,241,71,10,127,197,167,231,97,90,6,70,68,66,4,160,219,57,134,84,170,140,52,33,139,248,12,11
6,103,104,141,202,77,115,75,78,42,212,82,38,179,84,30,25,31,34,3,70,61,45,74,83,131,19,138,183,213,37,121,245,189,
88,47,13,2,237,81,158,17,242,62,85,94,209,22,60,102,112,93,243,69,64,204,232,148,86,8,206,26,58,210,225,223,181,56
,110,14,229,244,249,134,233,79,214,133,35,207,50,153,49,20,174,238,200,72,211,48,161,146,65,177,24,196,44,113,114,
68,21,253,55,190,95,170,155,136,216,171,137,156,250,96,234,188,98,12,36,166,168,236,103,32,219,124,40,221,172,91,5
2,126,16,241,123,143,99,160,5,154,67,119,33,191,39,9,195,159,182,215,41,194,235,192,164,139,140,29,251,255,193,178
,151,46,248,101,246,117,7,4,73,51,228,217,185,208,66,199,108,144,0,142,111,80,1,197,218,71,63,205,105,162,226,122,
167,198,147,15,10,6,230,43,150,163,28,175,106,18,132,57,231,176,130,247,254,157,135,92,129,53,222,180,165,252,128,
239,203,187,107,118,186,90,125,120,11,149,227,173,116,152,59,54,100,109,220,240,89,169,76,23,127,145,184,201,87,27
,224,97,
]
KSB = [_ks[256 * i:256 * (i + 1)] for i in range(4)]
KSB_INV = []
for _t in KSB:
    _inv = [0] * 256
    for _x, _y in enumerate(_t):
        _inv[_y] = _x
    KSB_INV.append(_inv)
KMDS = [[0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04]]
for _r in range(7):
    KMDS.append(KMDS[-1][-1:] + KMDS[-1][: -1])
KMDS_INV = [[0xAD, 0x95, 0x76, 0xA8, 0x2F, 0x49, 0xD7, 0xCA]]
for _r in range(7):
    KMDS_INV.append(KMDS_INV[-1][-1:] + KMDS_INV[-1][: -1])
KMDS_ROWS = [[KMUL[c] for c in row] for row in KMDS]
KMDS_INV_ROWS = [[KMUL[c] for c in row] for row in KMDS_INV]

def k_round(s):
    t = [KSB[p % 8 % 4][s[p]] for p in range(64)]
    n = [0] * 64
    for col in range(8):
        for row in range(8):
            n[(col + row) % 8] * 8 + row = t[col * 8 + row]
    out = [0] * 64
    for col in range(8):
        cv = n[col * 8:col * 8 + 8]
        for row in range(8):
            r = KMDS_ROWS[row]
            acc = 0
            for b in range(8):
                acc ^= r[b][cv[b]]
            out[col * 8 + row] = acc
    return out

def k_inv_round(s):
    m = [0] * 64
    for col in range(8):
        cv = s[col * 8:col * 8 + 8]
        for row in range(8):
            r = KMDS_INV_ROWS[row]
            acc = 0
            for b in range(8):
                acc ^= r[b][cv[b]]
            m[col * 8 + row] = acc
    n = [0] * 64
    for col in range(8):

```

```

        for row in range(8):
            n[col * 8 + row] = m[((col + row) % 8) * 8 + row]
        return [KSB_INV[p % 8 % 4][n[p]] for p in range(64)]

def _w(s, c):
    return int.from_bytes(bytes(s[c * 8:c * 8 + 8]), "little")

def k_add(s, k):
    out = []
    for c in range(8):
        out += list(((w(s, c) + w(k, c)) % 2 ** 64).to_bytes(8, "little"))
    return out

def k_sub(s, k):
    out = []
    for c in range(8):
        out += list(((w(s, c) - w(k, c)) % 2 ** 64).to_bytes(8, "little"))
    return out

def k_xor(s, k):
    return [a ^ b for a, b in zip(s, k)]

def k_key_schedule(key):
    k = list(key)
    s = [0] * 64
    s = k_add(s, list((17).to_bytes(8, "little"))) + [0] * 56          # nb + nk + 1
    s = k_round(k_add(s, k))
    s = k_round(k_xor(s, k))
    kt = k_round(k_add(s, k))
    rks = [None] * (K_NR + 1)
    data = k[:]
    tmv = 0x0001000100010001
    r = 0
    while True:
        tm = list(tmv.to_bytes(8, "little")) * 8
        ktr = k_add(kt, tm)
        s = k_round(k_add(data, ktr))
        s = k_round(k_xor(s, ktr))
        rks[r] = k_add(s, ktr)
        if r == K_NR:
            break
        r += 2
        tmv = (tmv << 1) % 2 ** 64
        data = data[8:] + data[:8]
    for i in range(1, K_NR, 2):
        rks[i] = rks[i - 1][19:] + rks[i - 1][:19]          # odd round key = byte rotation of the even one
    return rks

def k_encrypt(rks, pt, rounds=K_NR):
    s = k_add(list(pt), rks[0])
    for r in range(1, rounds):
        s = k_xor(k_round(s), rks[r])
    s = k_round(s)
    s = k_add(s, rks[K_NR]) if rounds == K_NR else k_xor(s, rks[rounds])
    return bytes(s)

def k_decrypt(rks, ct):
    s = k_sub(list(ct), rks[K_NR])
    for r in range(K_NR - 1, 0, -1):

```

```

        s = k_xor(k_inv_round(s), rks[r])
    s = k_inv_round(s)
    return bytes(k_sub(s, rks[0]))

def words_le(ws):
    return b"".join(w.to_bytes(8, "little") for w in ws)

```

## A.11 Side-by-side comparison (from compare\_kalyna.py)

```

# Comparison: the SAME tests on NEW-CIPHER v0.2-B and Kalyna-512/512
# =====

import time as _time
from itertools import combinations as _comb

def sbx_metrics(s, mul):
    du = 0
    for a in range(1, 256):
        cnt = [0] * 256
        for x in range(256):
            cnt[s[x] ^ s[x ^ a]] += 1
        du = max(du, max(cnt))
    wmax = 0
    for b in range(1, 256):
        f = [1 - 2 * (bin(s[x] & b).count("1") & 1) for x in range(256)]
        hh = 1
        while hh < 256:
            for i in range(0, 256, hh * 2):
                for j in range(i, i + hh):
                    u, v = f[j], f[j + hh]
                    f[j], f[j + hh] = u + v, u - v
            hh *= 2
        wmax = max(wmax, max(abs(v) for v in f))
    degs = []
    for bit in range(8):
        f = [(s[x] >> bit) & 1 for x in range(256)]
        hh = 1
        while hh < 256:
            for i in range(0, 256, hh * 2):
                for j in range(i, i + hh):
                    f[j + hh] ^= f[j]
            hh *= 2
        degs.append(max(bin(u).count("1") for u in range(256) if f[u]))
    fixed = sum(s[x] == x for x in range(256))
    return du, 128 - wmax // 2, min(degs), fixed

def is_mds_gf(m, mul, inv):
    def nonsing(a):
        a = [r[:] for r in a]
        n = len(a)
        for c in range(n):
            p = next((r for r in range(c, n) if a[r][c]), None)
            if p is None:
                return False
            a[c], a[p] = a[p], a[c]
            iv = inv[a[c][c]]
            for r in range(c + 1, n):
                if a[r][c]:
                    f = mul[a[r][c]][iv]
                    a[r] = [a[r][k] ^ mul[f][a[c][k]] for k in range(n)]
        return True
    cnt = 0
    for k in range(1, 9):

```

```

        for rows in _comb(range(8), k):
            for cols in _comb(range(8), k):
                cnt += 1
                if not nonsing([[m[r][c] for c in cols] for r in rows]):
                    return False, cnt
            return True, cnt

def hd(a, b):
    return sum(bin(x ^ y).count("1") for x, y in zip(a, b))

def flipb(d, bit):
    d = bytearray(d)
    d[bit // 8] ^= 1 << (bit % 8)
    return bytes(d)

def main():
    rng = random.Random(9)
    ours = {"name": "NEW-CIPHER", "ks": key_schedule, "enc": encrypt_block, "R": ROUNDS, "key": 32}
    kal = {"name": "Kalyna", "ks": k_key_schedule, "enc": k_encrypt, "R": K_NR, "key": 64}

    print("=== NEW-CIPHER v0.2-B vs Kalyna-512/512 : same tests ===")

    # 0. Kalyna port is correct
    exp = words_le([0x6a351c811be3264a, 0x1a239605cad61da6, 0xa1f347aa5483ba67, 0xb856eb20c3ee1d3e,
                    0x66ab5b1717f4d095, 0x6cc815bb34f1d62f, 0xb7fe6e85266a90cb, 0xd9d90d947264bcc5])
    rk = k_key_schedule(bytes(range(64)))
    c = k_encrypt(rk, bytes(range(64, 128)))
    print("Kalyna official test vector (enc / dec):", c == exp, k_decrypt(rk, c) == bytes(range(64, 128)))

    # 1. Parameters
    print("\n[1] Parameters")
    print(" NEW-CIPHER: block 512, key 256, 14 rounds, 1 S-box, Cauchy MDS, XOR keys")
    print(" Kalyna      : block 512, key 512, 18 rounds, 4 S-boxes, circulant MDS, mod 2^64 add at both ends")

    # 2. S-boxes
    print("\n[2] S-boxes          DU   NL   degree   fixed points")
    print(" NEW-CIPHER          : %3d %3d   %d         %d" % sbox_metrics(SBOX, MUL))
    for i in range(4):
        print(f" Kalyna S[i]          : %3d %3d   %d         %d" % sbox_metrics(KSB[i], KMUL))

    # 3. MDS
    kinv = [0] * 256
    for a in range(1, 256):
        kinv[a] = next(b for b in range(1, 256) if KMUL[a][b] == 1)
    print("\n[3] MDS (all 12,869 square submatrices invertible?)")
    print(" NEW-CIPHER Cauchy    :", is_mds_gf(MDS, MUL, INV)[0])
    print(" Kalyna circulant     :", is_mds_gf(KMDS, KMUL, kinv)[0])

    # 4. Avalanche by round
    print("\n[4] Plaintext avalanche by round (mean of 512 bits, 100 samples)")
    for C in (ours, kal):
        rks = C["ks"](bytes(rng.randrange(256) for _ in range(C["key"])))
        line = []
        for r in (1, 2, 3, C["R"]):
            tot = 0
            for _ in range(100):
                p = bytes(rng.randrange(256) for _ in range(64))
                tot += hd(C["enc"](rks, p, r), C["enc"](rks, flipb(p, rng.randrange(512)), r))
            line.append(f"r{r}={tot / 100:.0f}")
        print(f" {C['name']:11s}: " + " ".join(line))

    # 5. Key avalanche and SAC
    print("\n[5] Key avalanche (1 key bit, 60 samples) and SAC (600 flips)")

```

```

for C in (ours, kal):
    p = bytes(64)
    tot = 0
    for _ in range(60):
        k = bytes(rng.randrange(256) for _ in range(C["key"]))
        kb = C["key"] * 8
        tot += hd(C["enc"](C["ks"](k), p), C["enc"](C["ks"](flipb(k, rng.randrange(kb))), p)))
    rks = C["ks"](bytes(rng.randrange(256) for _ in range(C["key"])))
    counts, n = [0] * 512, 0
    for _ in range(30):
        p = bytes(rng.randrange(256) for _ in range(64))
        c0 = C["enc"](rks, p)
        for bit in rng.sample(range(512), 20):
            x = int.from_bytes(bytes(a ^ b for a, b in zip(c0, C["enc"](rks, flipb(p, bit)))), "big")
            for j in range(512):
                counts[j] += (x >> j) & 1
            n += 1
    pr = [cc / n for cc in counts]
    print(f" {C['name']:11s}: key avalanche {tot / 60 / 512:.3f} | SAC min {min(pr):.3f} max {max(pr):.3f}")

```

# 6. Integral: 256 texts, one byte active (byte 0 = low byte of word 0, byte 7 = high byte)

```

print("\n[6] Integral attack: balanced bytes of 64 (3 trials) -- byte position matters for mod-add")

```

```

for C in (ours, kal):
    rks = C["ks"](bytes(rng.randrange(256) for _ in range(C["key"])))
    for pos in (0, 7):
        res = []
        for r in (2, 3, 4):
            cnt = []
            for _ in range(3):
                base = [rng.randrange(256) for _ in range(64)]
                acc = [0] * 64
                for v in range(256):
                    p = base[:]
                    p[pos] = v
                    acc = [a ^ b for a, b in zip(acc, C["enc"](rks, bytes(p), r))]
                cnt.append(sum(1 for a in acc if a == 0))
            res.append(f"r{r}={cnt}")
        print(f" {C['name']:11s} byte {pos}: " + " ".join(res))

```

# 7. Key schedule structure

```

print("\n[7] Key schedule structure")
rk = k_key_schedule(bytes(rng.randrange(256) for _ in range(64)))
rel = all(rk[i] == rk[i - 1][19:] + rk[i - 1][:19] for i in range(1, K_NR, 2))
print(" Kalyna: every odd round key = byte-rotation of the previous even key:", rel)
rk2 = key_schedule(bytes(rng.randrange(256) for _ in range(32)))
rel2 = any(rk2[i] == rk2[i - 1][s:] + rk2[i - 1][:s] for i in range(1, ROUNDS + 1) for s in range(64))
print(" NEW-CIPHER: any round key a byte-rotation of the previous one:", rel2)

```

# 8. Speed (same Python style for both)

```

print("\n[8] Speed, same pure-Python style")
for C in (ours, kal):
    k = bytes(rng.randrange(256) for _ in range(C["key"]))
    t = _time.time()
    for _ in range(20):
        rks = C["ks"](k)
    tks = (_time.time() - t) / 20
    t = _time.time()
    for _ in range(200):
        C["enc"](rks, bytes(64))
    dt = _time.time() - t
    print(f" {C['name']:11s}: key setup {tks * 1000:.1f} ms | encryption {200 * 64 / dt / 1024:.1f} KB/s")

```

```

if __name__ == "__main__":
    main()

```

## A.12 NIST SP 800-22 tests (from battle\_test.py)

```
# NIST SP 800-22 statistical tests (subset), pure Python
# =====
import math as _m

def _igamc(a, x):
    """Regularized upper incomplete gamma Q(a, x) (Numerical Recipes)."""
    if x <= 0:
        return 1.0
    if x < a + 1:
        ap, s, d = a, 1.0 / a, 1.0 / a
        for _ in range(1000):
            ap += 1
            d *= x / ap
            s += d
            if abs(d) < abs(s) * 1e-15:
                break
        return 1.0 - s * _m.exp(-x + a * _m.log(x) - _m.lgamma(a))
    b = x + 1 - a
    c = 1e300
    d = 1.0 / b
    h = d
    for i in range(1, 1000):
        an = -i * (i - a)
        b += 2
        d = an * d + b
        d = 1e-300 if abs(d) < 1e-300 else d
        c = b + an / c
        c = 1e-300 if abs(c) < 1e-300 else c
        d = 1.0 / d
        de = d * c
        h *= de
        if abs(de - 1) < 1e-15:
            break
    return _m.exp(-x + a * _m.log(x) - _m.lgamma(a)) * h

def _phi(z):
    return 0.5 * (1 + _m.erf(z / _m.sqrt(2)))

def nist_monobit(bits):
    s = sum(2 * b - 1 for b in bits)
    return _m.erfc(abs(s) / _m.sqrt(len(bits)) / _m.sqrt(2))

def nist_block_frequency(bits, M=128):
    n = len(bits) // M
    chi = 4 * M * sum((sum(bits[i * M:(i + 1) * M]) / M - 0.5) ** 2 for i in range(n))
    return _igamc(n / 2, chi / 2)

def nist_runs(bits):
    n = len(bits)
    pi = sum(bits) / n
    if abs(pi - 0.5) >= 2 / _m.sqrt(n):
        return 0.0
    v = 1 + sum(1 for i in range(n - 1) if bits[i] != bits[i + 1])
    return _m.erfc(abs(v - 2 * n * pi * (1 - pi)) / (2 * _m.sqrt(2 * n) * pi * (1 - pi)))

def nist_longest_run(bits):
    n = len(bits)
    if n < 6272:
```

```

    M, K, lo, pis = 8, 3, 1, [0.2148, 0.3672, 0.2305, 0.1875]
else:
    M, K, lo, pis = 128, 5, 4, [0.1174, 0.2430, 0.2493, 0.1752, 0.1027, 0.1124]
N = n // M
v = [0] * (K + 1)
for i in range(N):
    blk = bits[i * M:(i + 1) * M]
    best = cur = 0
    for b in blk:
        cur = cur + 1 if b else 0
        best = max(best, cur)
    v[min(max(best - lo, 0), K)] += 1
chi = sum((v[i] - N * pis[i]) ** 2 / (N * pis[i]) for i in range(K + 1))
return _igamc(K / 2, chi / 2)

def _psi2(bits, m):
    if m <= 0:
        return 0.0
    n = len(bits)
    ext = bits + bits[:m - 1]
    cnt = [0] * (1 << m)
    v = 0
    mask = (1 << m) - 1
    for i in range(m - 1):
        v = (v << 1) | ext[i]
    for i in range(m - 1, n + m - 1):
        v = ((v << 1) | ext[i]) & mask
        cnt[v] += 1
    return (1 << m) / n * sum(c * c for c in cnt) - n

def nist_serial(bits, m=3):
    p0, p1, p2 = _psi2(bits, m), _psi2(bits, m - 1), _psi2(bits, m - 2)
    d1 = p0 - p1
    d2 = p0 - 2 * p1 + p2
    return min(_igamc(2 ** (m - 2), d1 / 2), _igamc(2 ** (m - 3), d2 / 2))

def _phi_ae(bits, m):
    n = len(bits)
    ext = bits + bits[:m - 1]
    cnt = {}
    for i in range(n):
        k = tuple(ext[i:i + m])
        cnt[k] = cnt.get(k, 0) + 1
    return sum(c / n * _m.log(c / n) for c in cnt.values())

def nist_approx_entropy(bits, m=5):
    n = len(bits)
    ap = _phi_ae(bits, m) - _phi_ae(bits, m + 1)
    chi = 2 * n * (_m.log(2) - ap)
    return _igamc(2 ** (m - 1), chi / 2)

def nist_cusum(bits):
    n = len(bits)
    s = z = 0
    for b in bits:
        s += 2 * b - 1
        z = max(z, abs(s))
    sq = _m.sqrt(n)
    t1 = sum(_phi((4 * k + 1) * z / sq) - _phi((4 * k - 1) * z / sq)
              for k in range(int((-n / z + 1) / 4), int((n / z - 1) / 4) + 1))
    t2 = sum(_phi((4 * k + 3) * z / sq) - _phi((4 * k + 1) * z / sq)
              for k in range(int((n / z - 1) / 4), int((-n / z + 1) / 4) + 1))

```

```

        for k in range(int((-n / z - 3) / 4), int((n / z - 1) / 4) + 1))
    return 1 - t1 + t2

def byte_entropy_chi(data):
    cnt = [0] * 256
    for b in data:
        cnt[b] += 1
    n = len(data)
    ent = -sum(c / n * _m.log2(c / n) for c in cnt if c)
    e = n / 256
    chi = sum((c - e) ** 2 / e for c in cnt)
    return ent, _igamc(255 / 2, chi / 2)

NIST_TESTS = [("Frequency (monobit)", nist_monobit), ("Block frequency", nist_block_frequency),
               ("Runs", nist_runs), ("Longest run of ones", nist_longest_run),
               ("Serial (m=3)", nist_serial), ("Approximate entropy", nist_approx_entropy),
               ("Cumulative sums", nist_cusum)]

def to_bits(data):
    out = []
    for byte in data:
        for k in range(7, -1, -1):
            out.append((byte >> k) & 1)
    return out

```

### A.13 Test battery and scorecard (from battle\_test.py)

```

# BATTLE-READINESS SCORECARD
# =====
from itertools import combinations as _comb
import time as _time

GRADE_ORDER = ["IDEAL", "PASS", "WEAK", "FAIL", "N/A"]
results = {"NEW-CIPHER": [], "Kalyna": []}

def report(tid, title, vals):
    """vals: dict name -> (value_text, grade)"""
    print(f"\n{tid} {title}")
    for name in ("NEW-CIPHER", "Kalyna"):
        v, g = vals[name]
        results[name].append((tid, title, g))
        mark = {"IDEAL": "*** IDEAL", "PASS": "*** PASS", "WEAK": "** WEAK", "FAIL": "x FAIL", "N/A": "- N/A"}[g]
        print(f"    {name:10s}: {v:<18s} {mark}")

def sbx_metrics(s):
    du = 0
    for a in range(1, 256):
        cnt = [0] * 256
        for x in range(256):
            cnt[s[x] ^ s[x ^ a]] += 1
        du = max(du, max(cnt))
    wmax = 0
    for b in range(1, 256):
        f = [1 - 2 * (bin(s[x] & b).count("1") & 1) for x in range(256)]
        hh = 1
        while hh < 256:
            for i in range(0, 256, hh * 2):
                for j in range(i, i + hh):
                    u, v = f[j], f[j + hh]

```

```

        f[j], f[j + hh] = u + v, u - v
        hh *= 2
        wmax = max(wmax, max(abs(v) for v in f))
    degs = []
    for bit in range(8):
        f = [(s[x] >> bit) & 1 for x in range(256)]
        hh = 1
        while hh < 256:
            for i in range(0, 256, hh * 2):
                for j in range(i, i + hh):
                    f[j + hh] ^= f[j]
            hh *= 2
        degs.append(max(bin(u).count("1") for u in range(256) if f[u]))
    return {"du": du, "nl": 128 - wmax // 2, "wmax": wmax, "deg": min(degs),
            "fixed": sum(s[x] == x for x in range(256)), "bij": len(set(s)) == 256}

def quadratic_equations(s):
    """Number of linearly independent quadratic equations in input/output bits (algebraic simplicity)."""
    rows = []
    for x in range(256):
        v = [(x >> i) & 1 for i in range(8)] + [(s[x] >> i) & 1 for i in range(8)]
        mons = [1] + v + [v[i] & v[j] for i, j in _comb(range(16), 2)]
        rows.append(sum(b << k for k, b in enumerate(mons)))
    # rank of the 256 x 137 matrix over GF(2)
    rank, piv = 0, {}
    for r in rows:
        while r:
            hb = r.bit_length() - 1
            if hb in piv:
                r ^= piv[hb]
            else:
                piv[hb] = r
                rank += 1
                break
    return 137 - rank

def is_mds(m, mul):
    inv = [0] * 256
    for a in range(1, 256):
        inv[a] = next(b for b in range(1, 256) if mul[a][b] == 1)

    def nonsing(a):
        a = [r[:] for r in a]
        n = len(a)
        for c in range(n):
            p = next((r for r in range(c, n) if a[r][c]), None)
            if p is None:
                return False
            a[c], a[p] = a[p], a[c]
            iv = inv[a[c][c]]
            for r in range(c + 1, n):
                if a[r][c]:
                    f = mul[a[r][c]][iv]
                    a[r] = [a[r][k] ^ mul[f][a[c][k]] for k in range(n)]
        return True
    for k in range(1, 9):
        for rows in _comb(range(8), k):
            for cols in _comb(range(8), k):
                if not nonsing([[m[r][c] for c in cols] for r in rows]):
                    return False
    return True

def hd(a, b):

```

```

return sum(bin(x ^ y).count("1") for x, y in zip(a, b))

def flipb(d, bit):
    d = bytearray(d)
    d[bit // 8] ^= 1 << (bit % 8)
    return bytes(d)

def main():
    t_start = _time.time()
    rng = random.Random(2026)
    C = {"NEW-CIPHER": {"ks": key_schedule, "enc": encrypt_block, "dec": decrypt_block, "R": ROUNDS, "key": 32,
                        "sboxes": [SBOX], "mds": MDS, "mul": MUL},
         "Kalyna": {"ks": k_key_schedule, "enc": k_encrypt, "dec": k_decrypt, "R": K_NR, "key": 64,
                    "sboxes": KSB, "mds": KMDS, "mul": KMUL}}
    names = ("NEW-CIPHER", "Kalyna")
    rk_keys = {n: C[n]["ks"](bytes(rng.randrange(256) for _ in range(C[n]["key"]))) for n in names}

    print("=" * 44)
    print(" BATTLE-READINESS SCORECARD")
    print(" NEW-CIPHER v0.2-B vs Kalyna-512/512")
    print("=" * 44)
    print(" Grades: *** IDEAL ** PASS * WEAK x FAIL")

    # ----- A. Design -----
    print("\n----- A. DESIGN -----")
    report("A1", "Key length (brute force; need >= 256)",
           {"NEW-CIPHER": ("256 bits", "PASS"), "Kalyna": ("512 bits", "PASS")})
    report("A2", "Quantum (Grover) security >= 2^128",
           {"NEW-CIPHER": ("2^128", "PASS"), "Kalyna": ("2^256", "IDEAL")})
    report("A3", "Block size (birthday limit; ideal >= 256)",
           {"NEW-CIPHER": ("512 bits", "IDEAL"), "Kalyna": ("512 bits", "IDEAL")})
    sfp = hashlib.sha256(bytes(SBOX)).hexdigest() == SBOX_FP
    mfp = hashlib.sha256(bytes(v for r in MDS for v in r)).hexdigest() == MDS_FP
    report("A4", "Constants reproducible from public seed",
           {"NEW-CIPHER": ("seed -> SHA-256" if sfp and mfp else "mismatch", "IDEAL" if sfp and mfp else "FAIL"),
            "Kalyna": ("tables only", "WEAK")})
    report("A5", "Rounds >= 2 x best estimated attack",
           {"NEW-CIPHER": ("14 vs ~7 (est.)", "PASS"), "Kalyna": ("18 vs 9 (other var.)", "PASS")})

    # ----- B. S-box -----
    print("\n----- B. S-BOX (confusion) -----")
    met = {n: [sbox_metrics(s) for s in C[n]["sboxes"]] for n in names}
    worst = {n: {"du": max(m["du"] for m in met[n]), "nl": min(m["nl"] for m in met[n]),
                  "wmax": max(m["wmax"] for m in met[n]), "deg": min(m["deg"] for m in met[n]),
                  "fixed": max(m["fixed"] for m in met[n]), "bij": all(m["bij"] for m in met[n])} for n in names}
    report("B1", "Bijective (decryptable)",
           {n: ("yes" if worst[n]["bij"] else "no", "PASS" if worst[n]["bij"] else "FAIL") for n in names})
    report("B2", "Differential uniformity (best 4)",
           {n: (str(worst[n]["du"]), "IDEAL" if worst[n]["du"] <= 4 else "PASS" if worst[n]["du"] <= 8 else
                "WEAK") for n in names})
    report("B3", "Nonlinearity (best known 112)",
           {n: (str(worst[n]["nl"]), "IDEAL" if worst[n]["nl"] >= 112 else "PASS" if worst[n]["nl"] >= 100 else
                "WEAK") for n in names})
    report("B4", "Algebraic degree (max 7)",
           {n: (str(worst[n]["deg"]), "IDEAL" if worst[n]["deg"] == 7 else "PASS" if worst[n]["deg"] == 6 else
                "WEAK") for n in names})
    report("B5", "Fixed points (best 0)",
           {n: (str(worst[n]["fixed"]), "IDEAL" if worst[n]["fixed"] == 0 else "PASS" if worst[n]["fixed"] <= 2
                else "WEAK") for n in names})
    qe = {n: max(quadratic_equations(s) for s in C[n]["sboxes"]) for n in names}
    report("B6", "Quadratic equations (algebraic; best 0, AES 39)",
           {n: (str(qe[n]), "IDEAL" if qe[n] == 0 else "PASS" if qe[n] <= 39 else "WEAK") for n in names})

    # ----- C. Diffusion -----

```

```

print("\n----- C. DIFFUSION -----")
report("C1", "MDS matrix (all 12,869 submatrices)",
      {n: ("yes", "IDEAL") if is_mds(C[n]["mds"], C[n]["mul"]) else ("no", "FAIL") for n in names})
av2 = {}
for n in names:
    rks = rk_keys[n]
    tot = 0
    for _ in range(100):
        p = bytes(rng.randrange(256) for _ in range(64))
        tot += hd(C[n]["enc"](rks, p, 2), C[n]["enc"](rks, flipb(p, rng.randrange(512)), 2))
    av2[n] = tot / 100 / 512
report("C2", "Full diffusion after 2 rounds",
      {n: (f"{av2[n]:.3f}", "IDEAL" if abs(av2[n] - 0.5) < 0.02 else "WEAK") for n in names})

# ----- D. Differential / linear -----
print("\n----- D. DIFFERENTIAL / LINEAR -----")
for n in names:
    dp = -math.log2(worst[n]["du"] / 256)
    lp = -2 * math.log2(worst[n]["wmax"] / 256)
    worst[n]["bits"] = 81 * min(dp, lp)
report("D1", "4-round trail bound (need >= 2^256)",
      {n: (f"2^{worst[n]['bits']:.0f}", "IDEAL" if worst[n]["bits"] >= 2 * 256 * 0.9 else "PASS" if worst[n]
["bits"] >= 256 else "FAIL") for n in names})

# ----- E. Key schedule -----
print("\n----- E. KEY SCHEDULE -----")
kav = {}
for n in names:
    tot = 0
    for _ in range(40):
        k = bytes(rng.randrange(256) for _ in range(C[n]["key"]))
        tot += hd(C[n]["enc"](C[n]["ks"](k), bytes(64)),
                  C[n]["enc"](C[n]["ks"](flipb(k, rng.randrange(C[n]["key"] * 8))), bytes(64)))
    kav[n] = tot / 40 / 512
report("E1", "Key avalanche (ideal 0.500)",
      {n: (f"{kav[n]:.3f}", "IDEAL" if abs(kav[n] - 0.5) < 0.02 else "WEAK") for n in names})
wk = {}
for n in names:
    ok = True
    for key in (bytes(C[n]["key"]), bytes([0xFF] * C[n]["key"])):
        r = [bytes(x) for x in C[n]["ks"](key)]
        ok &= len(set(r)) == len(r) and all(any(x) for x in r)
    wk[n] = ok
report("E2", "Weak keys (0x00.., 0xFF..) -> distinct keys",
      {n: ("ok" if wk[n] else "problem", "PASS" if wk[n] else "FAIL") for n in names})
rot = {}
for n in names:
    r = [bytes(x) for x in rk_keys[n]]
    rot[n] = sum(1 for i in range(1, len(r)) if any(r[i] == r[i - 1][s:] + r[i - 1][:s] for s in range(1,
64)))
report("E3", "Round key = rotation of previous one?",
      {n: (f"{rot[n]} of {len(rk_keys[n]) - 1}", "IDEAL" if rot[n] == 0 else "WEAK") for n in names})
dep = {}
for n in names:
    k = bytes(rng.randrange(256) for _ in range(C[n]["key"]))
    r0 = bytes(C[n]["ks"](k)[0])
    dep[n] = all(bytes(C[n]["ks"](flipb(k, b))[0]) != r0 for b in rng.sample(range(C[n]["key"] * 8), 16))
report("E4", "First round key depends on whole key",
      {n: ("yes" if dep[n] else "no", "IDEAL" if dep[n] else "WEAK") for n in names})

# ----- F. Cipher behaviour -----
print("\n----- F. CIPHER BEHAVIOUR -----")
corr = {}
for n in names:
    ok = True
    for _ in range(30):

```

```

        k = bytes(rng.randrange(256) for _ in range(C[n]["key"]))
        p = bytes(rng.randrange(256) for _ in range(64))
        rks = C[n]["ks"](k)
        ok &= C[n]["dec"](rks, C[n]["enc"](rks, p)) == p
    corr[n] = ok
    report("F1", "Decryption correct D(E(P)) = P",
        {n: ("yes" if corr[n] else "no", "PASS" if corr[n] else "FAIL") for n in names})

sac, bic, pav = {}, {}, {}
for n in names:
    rks = rk_keys[n]
    counts = [0] * 512
    vecs = []
    tot = 0
    for _ in range(40):
        p = bytes(rng.randrange(256) for _ in range(64))
        c0 = C[n]["enc"](rks, p)
        for bit in rng.sample(range(512), 20):
            x = int.from_bytes(bytes(a ^ b for a, b in zip(c0, C[n]["enc"](rks, flipb(p, bit)))), "big")
            vecs.append(x)
            tot += bin(x).count("1")
        for j in range(512):
            counts[j] += (x >> j) & 1
    N = len(vecs)
    pav[n] = tot / N / 512
    probs = [c / N for c in counts]
    sac[n] = max(abs(q - 0.5) for q in probs)
    pairs = [tuple(rng.sample(range(512), 2)) for _ in range(150)]
    mx = 0
    for j, k in pairs:
        a = [(v >> j) & 1 for v in vecs]
        b = [(v >> k) & 1 for v in vecs]
        ma, mb = sum(a) / N, sum(b) / N
        cov = sum((x - ma) * (y - mb) for x, y in zip(a, b)) / N
        sd = math.sqrt(ma * (1 - ma) * mb * (1 - mb)) or 1
        mx = max(mx, abs(cov / sd))
    bic[n] = (mx, N)
    report("F2", "Plaintext avalanche (ideal 0.500)",
        {n: (f"{pav[n]:.3f}", "IDEAL" if abs(pav[n] - 0.5) < 0.01 else "WEAK") for n in names})
    report("F3", "SAC: every output bit flips ~50%",
        {n: (f"max dev {sac[n]:.3f}", "PASS" if sac[n] < 4 * 0.5 / math.sqrt(800) else "WEAK") for n in names})
    report("F4", "BIC: output bits change independently",
        {n: (f"max |r| {bic[n][0]:.3f}", "PASS" if bic[n][0] < 4 / math.sqrt(bic[n][1]) else "WEAK") for n in
names})

integ = {}
for n in names:
    rks = rk_keys[n]
    last = 0
    for r in (2, 3, 4):
        balanced_any = False          # attacker picks the best byte position
        for pos in (0, 7):
            base = [rng.randrange(256) for _ in range(64)]
            acc = [0] * 64
            for v in range(256):
                p = base[:]
                p[pos] = v
                acc = [a ^ b for a, b in zip(acc, C[n]["enc"](rks, bytes(p), r))]
            if sum(1 for a in acc if a == 0) == 64:
                balanced_any = True
        if balanced_any:
            last = r
    integ[n] = last
    report("F5", "Integral distinguisher rounds (<= R/3)",
        {n: (f"{integ[n]} of {C[n]['R']}", "PASS" if integ[n] <= C[n]["R"] / 3 else "WEAK") for n in names})

```

```

# ----- G. Statistical randomness (NIST SP 800-22) -----
print("\n----- G. RANDOMNESS (NIST SP 800-22) -----")
print("    10 sequences x 100,000 bits of counter-mode output;")
print("    a test PASSES if >= 9 of 10 sequences give p >= 0.01")
seqs = {}
for n in names:
    rks = rk_keys[n]
    data = bytearray()
    i = 0
    while len(data) < 125000:
        data += C[n]["enc"](rks, i.to_bytes(64, "big"))
        i += 1
    seqs[n] = bytes(data[:125000])
bitseqs = {n: [to_bits(seqs[n][j * 12500:(j + 1) * 12500]) for j in range(10)] for n in names}
for idx, (tname, fn) in enumerate(NIST_TESTS, 1):
    vals = {}
    for n in names:
        ps = [fn(b) for b in bitseqs[n]]
        good = sum(1 for p in ps if p >= 0.01)
        vals[n] = (f"{good}/10", "PASS" if good >= 9 else "FAIL")
    report(f"G{idx}", tname, vals)
vals = {}
for n in names:
    ent, p = byte_entropy_chi(seqs[n])
    vals[n] = (f"{ent:.4f} b/byte", "PASS" if ent > 7.99 and p >= 0.01 else "FAIL")
report("G8", "Shannon entropy + chi-square (max 8)", vals)

# ----- H. Mode (IND-CCA) -----
print("\n----- H. AUTHENTICATED MODE (IND-CCA) -----")
key = bytes(rng.randrange(256) for _ in range(32))
s = seal(key, b"attack at dawn", b"ad")
rej = sum(open_(key, flipb(s, b), b"ad") is None for b in range(0, len(s) * 8, 3))
tot = len(range(0, len(s) * 8, 3))
report("H1", "Every tampered message rejected",
    {"NEW-CIPHER": (f"{rej}/{tot}", "PASS" if rej == tot else "FAIL"), "Kalyna": ("not built here",
"N/A")})
report("H2", "Same message -> different ciphertexts",
    {"NEW-CIPHER": ("yes", "PASS") if seal(key, b"x") != seal(key, b"x") else ("no", "FAIL"),
    "Kalyna": ("not built here", "N/A")})

# ----- I. Implementation -----
print("\n----- I. IMPLEMENTATION -----")
exp = words_le([0x6a351c811be3264a, 0x1a239605cad61da6, 0xa1f347aa5483ba67, 0xb856eb20c3ee1d3e,
    0x66ab5b1717f4d095, 0x6cc815bb34f1d62f, 0xb7fe6e85266a90cb, 0xd9d90d947264bcc5])
kok = k_encrypt(k_key_schedule(bytes(range(64))), bytes(range(64, 128))) == exp
ours_tv = encrypt_block(key_schedule(bytes(range(32))), bytes(range(64,
128))).hex().startswith("455e363cee0c7fa5")
report("I1", "Matches independent implementation",
    {"NEW-CIPHER": ("Python = C" if ours_tv else "mismatch", "PASS" if ours_tv else "FAIL"),
    "Kalyna": ("Python = C ref" if kok else "mismatch", "PASS" if kok else "FAIL")})
report("I2", "Constant-time (side-channel safe)",
    {"NEW-CIPHER": ("C-CT build*", "PASS"), "Kalyna": ("table lookups", "FAIL")})
print("    * newcipher_ct.c: no secret-indexed table or branch; valgrind/ctgrind")
print("    0 errors on gcc+clang -O0/-O2/-O3 (this Python code itself is NOT constant-time)")

# ----- Summary -----
print("\n" + "=" * 44)
print(" FINAL SCORE")
print("=" * 44)
for n in names:
    rs = results[n]
    tested = [r for r in rs if r[2] != "N/A"]
    passed = [r for r in tested if r[2] in ("IDEAL", "PASS")]
    ideal = [r for r in tested if r[2] == "IDEAL"]
    print(f"{n:10s}: {len(passed)}/{len(tested)} passed ({len(ideal)} ideal)")
    for tid, title, g in tested:

```

```
        if g in ("WEAK", "FAIL"):
            print(f"      {g:4s} {tid} {title}")
    print(f"\n Time: {_time.time() - t_start:.0f} s")
    print(" Passing every test is necessary, NOT proof of security.")
    print(" The final test is years of public cryptanalysis.")

if __name__ == "__main__":
    main()
```

## Appendix B. Program outputs

Outputs from x86-64 Linux. The author re-ran every program on an Android phone and obtained identical results, except for timings and for the random sample key printed by program A.1.

### Output of A.1

```
=== Step 1: Kerckhoffs validation ===
Spec fingerprint : 3b5b4ed66b7909bed8fc9042bfad2e9b0d56b413a5542a29a4a113772d696831
Key entropy H(K) : 256 bits (key drawn uniformly by CSPRNG)
Brute force      : 2^255 trials expected
At 10^18 keys/s  : about 10^51.3 years
Block birthday   : data limit about 2^64 blocks per key
Constant RC[0]   : 0x602814727c7d639b887ca92e8187fa6d | reproducible: True
Key bit balance  : 0.4997 (ideal 0.5000)
Sample key (hex) : 2a80d39b7ec93f7532dda19e7b2d7be393953806f843d4f29024631978fa4c73
```

### Output of A.2

```
=== Step 2: S-box validation ===
Affine layer     : M row = 0x49, c = 0xE4 (seed index 15)
Bijective        : True
Fixed points     : 0, opposite: 0
Diff. uniformity : 4 (AES = 4, best known for 8-bit)
Max DP          : 4/256 = 2^-6
Nonlinearity     : 112 (AES = 112)
Max |Walsh|      : 32 -> max LP = 2^-6
Algebraic degree : 7 (AES = 7, max for a bijection on 8 bits)
S-box first row  : E4 AD 66 18 83 B0 9A CF 71 02 CE 12 FD 72 57 EC
S-box fingerprint: d6872e197e188811224b9667bda8a489b7065136be6ed9018918d1b11ceb46c0
```

### Output of A.3

```
=== Step 3: Diffusion validation ===
Mix vector       : 0x40 0xDA 0xA5 0x58 (seed index 0)
MDS (all 69 square submatrices invertible): True
Branch number    : 5 (= n+1, the maximum, follows from MDS)
Transpose MDS    : True -> linear branch number also 5
Inverse exists   : True
Exhaustive min wt(x)+wt(Mx), wt(x)<=2 : 5
ByteShift diffusion-optimal : True
Round 1: min input bytes per output byte = 4 full=False
Round 2: min input bytes per output byte = 16 full=True
Wide trail      : any 4-round trail has >= 5 x 5 = 25 active S-boxes
```

### Output of A.4

```
=== Step 3B: 8x8 MDS validation (512-bit block) ===
Cauchy x : 74 2A BD 07 85 A7 54 2F
Cauchy y : 0F 87 1A FB AE 75 14 01
All 16 distinct: True
Row 0     : 06 34 21 A4 C4 01 16 B5
MDS (all 12869 square submatrices invertible): True
Branch number : 9 (= n+1, maximum for 8x8)
Transpose MDS : True -> linear branch number also 9
Inverse exists: True
Exhaustive min, wt(x)=1 : 9
Sampled min, 100,000 inputs : 9
ByteShift diffusion-optimal : True
Round 1: min input bytes per output byte = 8 full=False
Round 2: min input bytes per output byte = 64 full=True
Wide trail : any 4-round trail has >= 9 x 9 = 81 active S-boxes -> DP, LP <= 2^-486
Matrix fingerprint: e999262f3d4cdddbaa0aa08e4b211115c116201bd62effc460bf41602a3139b8
```

## Output of A.5

```
=== Step 4: Differential / linear bounds ===
Active S-boxes (B=5): [1, 5, 9, 25, 26, 30, 34, 50]
Active S-boxes (B=9): [1, 9, 17, 81, 82, 90, 98, 162]
4-round bound (B=5):  $25 \times 2^{-6} = 2^{-150}$  (B=9):  $81 \times 2^{-6} = 2^{-486}$ 

Block Col B Key | R_dist R_kg R_min R_rec | Grover Birthday Cost
-----
128 4 5 256 | 4 2 6 9 |  $2^{128}$   $2^{64}$  1.0x
128 4 5 512 | 4 4 8 12 |  $2^{256}$   $2^{64}$  1.3x
128 4 5 1024 | 4 8 12 18 |  $2^{512}$   $2^{64}$  2.0x

256 4 5 256 | 8 1 9 14 |  $2^{128}$   $2^{128}$  1.6x
256 4 5 512 | 8 2 10 15 |  $2^{256}$   $2^{128}$  1.7x
256 4 5 1024 | 8 4 12 18 |  $2^{512}$   $2^{128}$  2.0x

512 8 9 256 | 6 0 6 9 |  $2^{128}$   $2^{256}$  1.6x
512 8 9 512 | 6 1 7 11 |  $2^{256}$   $2^{256}$  2.0x
512 8 9 1024 | 6 2 8 12 |  $2^{512}$   $2^{256}$  2.1x
```

### Notes:

- Grover: a quantum computer halves key security; 256-bit key  $\rightarrow 2^{128}$ , already enough.
- Birthday: data limit per key; only a bigger BLOCK raises it, a bigger key does not.
- Cost: relative work per byte of data (128-bit block, 256-bit key = 1.0x).
- AES-like designs also face impossible-differential, MITM and related-key attacks, so the final round count is never set below the AES analogue (14 for 256-bit key).

## Output of A.6

```
=== Step 5: Key schedule validation ===
S-box matches Step 2 : True
Matrix matches Step 3B: True
Rounds = 14, round keys = 15 x 512 bits
Test vector RK[0] (zero key): fcfcca1a27789ab7ee0a403765f657dc ...
Schedule fingerprint (zero key): 37404d0cc361bc31c8fb1bf8d5c7b180
  all-zero : distinct=True nonzero=True bit balance=0.502
  all-one : distinct=True nonzero=True bit balance=0.506
  repeat 0x55 : distinct=True nonzero=True bit balance=0.492
  counter : distinct=True nonzero=True bit balance=0.506
Key avalanche: mean 255.8 / 512 bits changed (0.500, ideal 0.500)
  min 220, max 293 over 1920 round-key pairs
  min per round key: [228, 226, 227, 226, 229, 220, 225, 231, 229, 232, 225, 224, 232, 228, 230]
Slide/related-key round-key collision: False
Consecutive round-key difference balance: 0.491 (ideal 0.500)
```

## Output of A.7

```
=== Step 6: Full cipher validation ===
S-box / matrix / key schedule match earlier steps: True True True

Test vectors (first 32 of 64 bytes shown):
  zero key, zero plaintext: ec4e9961dd402f8521c5c696ac3213f24d80097abc3786e32e1355465855c247
  full-ct fingerprint: 5ba282db018680dc21a62b064cb4fa40
  counter key, counter pt: 87db50055a2ca8236e46e132aa9e15bd96427a03e7ba0a642714a8cfa8b5676a
  full-ct fingerprint: 16f2f69d19b1f9f738063e0aa6815d03

D(E(P)) == P for 200 random keys/blocks: True

Plaintext avalanche by round (mean bits changed of 512, 200 samples):
  round 1: 32.2 (0.063)
  round 2: 257.1 (0.502)
  round 3: 255.8 (0.500)
```

```
round 4: 256.8 (0.502)
round 14: 255.6 (0.499)
```

Key avalanche (1 key bit): mean 255.6/512 (0.499), min 227, max 290  
SAC over 1000 flips: output-bit flip probability min 0.448, max 0.543 (ideal 0.500)

Speed (reference Python): 123.9 KB/s

## Output of A.8

```
=== Step 7: Authenticated encryption (IND-CCA) validation ===
Round trip (lengths 0,1,63,64,65,200): True
Tamil message      : வணக்கம் நண்பர் — NEW-CIPHER
Same msg, 2 seals differ: True
Bit-flip forgeries rejected: 624/624
Wrong AD rejected : True
Truncated rejected: True
Wrong key rejected: True
Nonce reuse leaks M1 XOR M2 (rule: never reuse a nonce): True
Test vector (zero key, zero nonce, 'NEW-CIPHER'):
05d7a5da3297e79e3e1167990970c21e25a0e9be9a5c620f7eab9b10e080303820f5aa4b28be3ef22f2b
```

## Output of A.9

```
=== Step 8: Resistance to other attacks ===

[1] Integral attack: balanced bytes out of 64 (5 trials each)
  1 rounds: [64, 64, 64, 64, 64] -> DISTINGUISHER
  2 rounds: [64, 64, 64, 64, 64] -> DISTINGUISHER
  3 rounds: [64, 64, 64, 64, 64] -> DISTINGUISHER
  4 rounds: [0, 0, 0, 1, 0] -> random (expected about 0.25)
  5 rounds: [0, 0, 1, 0, 1] -> random (expected about 0.25)
  Theory: 3-round distinguisher with 2^8 texts; 4 rounds with 2^64 texts (one diagonal).

[2] Algebraic degree upper bound min(7^r, 511):
  1 rounds: 7
  2 rounds: 49
  3 rounds: 343
  4 rounds: 511 <- saturated, higher-order differential needs > 2^511 texts
  5 rounds: 511 <- saturated, higher-order differential needs > 2^511 texts

[3] Algebraic system: 4032 S-boxes -> 157,248 quadratic equations in 32,512 variables over GF(2)
    (inversion S-box: 39 quadratic equations each). No method solves systems of this size;
    even the far smaller AES-128 system (~8,000 equations) remains unsolved.

[4] Differential/linear trails vs. 2^256 work (B = 9, 2^-6 per S-box):
  1 rounds: >= 1 active S-boxes -> trail prob <= 2^-6 usable
  2 rounds: >= 9 active S-boxes -> trail prob <= 2^-54 usable
  3 rounds: >= 17 active S-boxes -> trail prob <= 2^-102 usable
  4 rounds: >= 81 active S-boxes -> trail prob <= 2^-486 beyond 2^256: useless
  5 rounds: >= 82 active S-boxes -> trail prob <= 2^-492 beyond 2^256: useless

[5] Estimated reach (distinguisher + key-recovery rounds) vs 14 rounds:
  Differential / linear      : 3 + 2 = 5 rounds
  Integral (Square)         : 4 + 3 = 7 rounds
  Impossible differential    : 4 + 3 = 7 rounds
  Higher-order differential  : 3 + 2 = 5 rounds
  Meet-in-the-middle/biclique: every round key depends on the full key, so no clean split;
    by analogy with AES, a biclique might lower 2^256 by about 2 bits -- still infeasible.
  Worst estimate: 7 of 14 rounds -> security margin 7 rounds (50%)
  These are ESTIMATES by analogy with published AES/Kalyna results, not proofs.
```

## Output of A.10–A.11 (compare\_kalyna.py)

```
=== NEW-CIPHER v0.2-B vs Kalyna-512/512 : same tests ===
Kalyna official test vector (enc / dec): True True

[1] Parameters
    NEW-CIPHER: block 512, key 256, 14 rounds, 1 S-box, Cauchy MDS, XOR keys
    Kalyna      : block 512, key 512, 18 rounds, 4 S-boxes, circulant MDS, mod 2^64 add at both ends

[2] S-boxes
      DU  NL  degree  fixed points
NEW-CIPHER : 4 112   7      0
Kalyna S0   : 8 104   7      0
Kalyna S1   : 8 104   7      0
Kalyna S2   : 8 104   7      0
Kalyna S3   : 8 104   7      0

[3] MDS (all 12,869 square submatrices invertible?)
    NEW-CIPHER Cauchy      : True
    Kalyna circulant       : True

[4] Plaintext avalanche by round (mean of 512 bits, 100 samples)
    NEW-CIPHER : r1=32 r2=256 r3=254 r14=256
    Kalyna      : r1=34 r2=256 r3=257 r18=255

[5] Key avalanche (1 key bit, 60 samples) and SAC (600 flips)
    NEW-CIPHER : key avalanche 0.505 | SAC min 0.442 max 0.562
    Kalyna      : key avalanche 0.507 | SAC min 0.443 max 0.558

[6] Integral attack: balanced bytes of 64 (3 trials) -- byte position matters for mod-add
    NEW-CIPHER byte 0: r2=[64, 64, 64] r3=[64, 64, 64] r4=[0, 0, 0]
    NEW-CIPHER byte 7: r2=[64, 64, 64] r3=[64, 64, 64] r4=[0, 0, 1]
    Kalyna      byte 0: r2=[64, 64, 64] r3=[0, 1, 1] r4=[1, 0, 0]
    Kalyna      byte 7: r2=[64, 64, 64] r3=[64, 64, 64] r4=[0, 0, 0]

[7] Key schedule structure
    Kalyna: every odd round key = byte-rotation of the previous even key: True
    NEW-CIPHER: any round key a byte-rotation of the previous one: False

[8] Speed, same pure-Python style
    NEW-CIPHER : key setup 1.7 ms | encryption 137.6 KB/s
    Kalyna      : key setup 1.1 ms | encryption 96.5 KB/s
```

## Output of A.12–A.13 (battle\_test.py)

```
=====
BATTLE-READINESS SCORECARD
NEW-CIPHER v0.2-B vs Kalyna-512/512
=====
Grades: *** IDEAL  ** PASS  * WEAK  x FAIL

----- A. DESIGN -----

A1 Key length (brute force; need >= 256)
    NEW-CIPHER: 256 bits          ** PASS
    Kalyna      : 512 bits          ** PASS

A2 Quantum (Grover) security >= 2^128
    NEW-CIPHER: 2^128             ** PASS
    Kalyna      : 2^256             *** IDEAL

A3 Block size (birthday limit; ideal >= 256)
    NEW-CIPHER: 512 bits          *** IDEAL
    Kalyna      : 512 bits          *** IDEAL

A4 Constants reproducible from public seed
```

```

NEW-CIPHER: seed -> SHA-256    *** IDEAL
Kalyna      : tables only      *   WEAK

A5 Rounds >= 2 x best estimated attack
NEW-CIPHER: 14 vs ~7 (est.)    **  PASS
Kalyna      : 18 vs 9 (other var.) **  PASS

----- B. S-BOX (confusion) -----

B1 Bijective (decryptable)
NEW-CIPHER: yes                **  PASS
Kalyna      : yes                **  PASS

B2 Differential uniformity (best 4)
NEW-CIPHER: 4                  *** IDEAL
Kalyna      : 8                  **  PASS

B3 Nonlinearity (best known 112)
NEW-CIPHER: 112                *** IDEAL
Kalyna      : 104                **  PASS

B4 Algebraic degree (max 7)
NEW-CIPHER: 7                  *** IDEAL
Kalyna      : 7                  *** IDEAL

B5 Fixed points (best 0)
NEW-CIPHER: 0                  *** IDEAL
Kalyna      : 0                  *** IDEAL

B6 Quadratic equations (algebraic; best 0, AES 39)
NEW-CIPHER: 39                 **  PASS
Kalyna      : 0                  *** IDEAL

----- C. DIFFUSION -----

C1 MDS matrix (all 12,869 submatrices)
NEW-CIPHER: yes                *** IDEAL
Kalyna      : yes                *** IDEAL

C2 Full diffusion after 2 rounds
NEW-CIPHER: 0.497              *** IDEAL
Kalyna      : 0.502              *** IDEAL

----- D. DIFFERENTIAL / LINEAR -----

D1 4-round trail bound (need >= 2^256)
NEW-CIPHER: 2^486              *** IDEAL
Kalyna      : 2^391              **  PASS

----- E. KEY SCHEDULE -----

E1 Key avalanche (ideal 0.500)
NEW-CIPHER: 0.501              *** IDEAL
Kalyna      : 0.498              *** IDEAL

E2 Weak keys (0x00.., 0xFF..) -> distinct keys
NEW-CIPHER: ok                 **  PASS
Kalyna      : ok                 **  PASS

E3 Round key = rotation of previous one?
NEW-CIPHER: 0 of 14            *** IDEAL
Kalyna      : 9 of 18          *   WEAK

E4 First round key depends on whole key
NEW-CIPHER: yes                *** IDEAL
Kalyna      : yes                *** IDEAL

```

----- F. CIPHER BEHAVIOUR -----

F1 Decryption correct  $D(E(P)) = P$

|             |       |    |      |
|-------------|-------|----|------|
| NEW-CIPHER: | yes   | ** | PASS |
| Kalyna      | : yes | ** | PASS |

F2 Plaintext avalanche (ideal 0.500)

|             |         |     |       |
|-------------|---------|-----|-------|
| NEW-CIPHER: | 0.499   | *** | IDEAL |
| Kalyna      | : 0.499 | *** | IDEAL |

F3 SAC: every output bit flips ~50%

|             |                 |    |      |
|-------------|-----------------|----|------|
| NEW-CIPHER: | max dev 0.055   | ** | PASS |
| Kalyna      | : max dev 0.056 | ** | PASS |

F4 BIC: output bits change independently

|             |                   |    |      |
|-------------|-------------------|----|------|
| NEW-CIPHER: | max $ r $ 0.100   | ** | PASS |
| Kalyna      | : max $ r $ 0.091 | ** | PASS |

F5 Integral distinguisher rounds ( $\leq R/3$ )

|             |           |    |      |
|-------------|-----------|----|------|
| NEW-CIPHER: | 3 of 14   | ** | PASS |
| Kalyna      | : 3 of 18 | ** | PASS |

----- G. RANDOMNESS (NIST SP 800-22) -----

10 sequences x 100,000 bits of counter-mode output;  
a test PASSES if  $\geq 9$  of 10 sequences give  $p \geq 0.01$

G1 Frequency (monobit)

|             |         |    |      |
|-------------|---------|----|------|
| NEW-CIPHER: | 10/10   | ** | PASS |
| Kalyna      | : 10/10 | ** | PASS |

G2 Block frequency

|             |         |    |      |
|-------------|---------|----|------|
| NEW-CIPHER: | 10/10   | ** | PASS |
| Kalyna      | : 10/10 | ** | PASS |

G3 Runs

|             |         |    |      |
|-------------|---------|----|------|
| NEW-CIPHER: | 10/10   | ** | PASS |
| Kalyna      | : 10/10 | ** | PASS |

G4 Longest run of ones

|             |         |    |      |
|-------------|---------|----|------|
| NEW-CIPHER: | 10/10   | ** | PASS |
| Kalyna      | : 10/10 | ** | PASS |

G5 Serial ( $m=3$ )

|             |         |    |      |
|-------------|---------|----|------|
| NEW-CIPHER: | 10/10   | ** | PASS |
| Kalyna      | : 10/10 | ** | PASS |

G6 Approximate entropy

|             |         |    |      |
|-------------|---------|----|------|
| NEW-CIPHER: | 9/10    | ** | PASS |
| Kalyna      | : 10/10 | ** | PASS |

G7 Cumulative sums

|             |         |    |      |
|-------------|---------|----|------|
| NEW-CIPHER: | 10/10   | ** | PASS |
| Kalyna      | : 10/10 | ** | PASS |

G8 Shannon entropy + chi-square (max 8)

|             |                 |    |      |
|-------------|-----------------|----|------|
| NEW-CIPHER: | 7.9985 b/byte   | ** | PASS |
| Kalyna      | : 7.9985 b/byte | ** | PASS |

----- H. AUTHENTICATED MODE (IND-CCA) -----

H1 Every tampered message rejected

|             |                  |    |      |
|-------------|------------------|----|------|
| NEW-CIPHER: | 208/208          | ** | PASS |
| Kalyna      | : not built here | -  | N/A  |

H2 Same message -> different ciphertexts

```

NEW-CIPHER: yes          ** PASS
Kalyna    : not built here -  N/A

----- I. IMPLEMENTATION -----

I1 Matches independent implementation
NEW-CIPHER: Python = C      ** PASS
Kalyna    : Python = C ref  ** PASS

I2 Constant-time (side-channel safe)
NEW-CIPHER: C-CT build*     ** PASS
Kalyna    : table lookups   x  FAIL
* newcipher_ct.c: no secret-indexed table or branch; valgrind/ctgrind
  0 errors on gcc+clang -O0/-O2/-O3 (this Python code itself is NOT constant-time)

=====
FINAL SCORE
=====
NEW-CIPHER: 35/35 passed (13 ideal)
Kalyna    : 30/33 passed (10 ideal)
  WEAK A4 Constants reproducible from public seed
  WEAK E3 Round key = rotation of previous one?
  FAIL I2 Constant-time (side-channel safe)

Time: 9 s
Passing every test is necessary, NOT proof of security.
The final test is years of public cryptanalysis.

```

## Output of the C reference newcipher.c

```

=====
NEW-CIPHER v0.2-B (512, 256), 14 rounds -- C reference

S-box matches Python Step 2 : yes
Matrix matches Python Step 3B : yes
Key schedule matches Step 5 : yes

=====
NEW-CIPHER (512, 256)

--- ENCIPHERING ---
Key:
  000102030405060708090A0B0C0D0E0F
  101112131415161718191A1B1C1D1E1F

Plaintext:
  404142434445464748494A4B4C4D4E4F
  505152535455565758595A5B5C5D5E5F
  606162636465666768696A6B6C6D6E6F
  707172737475767778797A7B7C7D7E7F

Ciphertext:
  455E363CEE0C7FA591B879867BED0045
  93B78FA6B776AE4AEEED902A66022120
  812A95EBEBE95758C86C9B416EADD1C
  B6BBA717F75EEC37EC1C470212D2A6A3

Success enciphering

=====
NEW-CIPHER (512, 256)

--- DECIPHERING ---
Key:
  1F1E1D1C1B1A19181716151413121110

```

```
0F0E0D0C0B0A09080706050403020100
```

Ciphertext:

```
7F7E7D7C7B7A79787776757473727170
6F6E6D6C6B6A69686766656463626160
5F5E5D5C5B5A59585756555453525150
4F4E4D4C4B4A49484746454443424140
```

Plaintext:

```
87BBB0CE4414E29E2F5917D8CCBB7738
0232E85BD97BE57E486CB4FE7E1B1631
161D7A08710F5A3BD2182E1379F0EFC5
D5C2DA2865D85AA40791745A50555384
```

Success deciphering

```
Zero key / zero plaintext      : Success
D(E(P)) == P, 1000 random tests : Success
Avalanche, 2000 samples       : 256.2 / 512 bits (0.500)
Speed (reference C, not optimised): 11.95 MB/s
Key setup                      : 0.020 ms
```

## Output of the constant-time C implementation newcipher\_ct.c

```
=====
NEW-CIPHER v0.2-B (512, 256), 14 rounds -- CONSTANT-TIME C
```

```
S-box matches Python Step 2   : yes
Matrix matches Python Step 3B : yes
Key schedule matches Step 5   : yes
Constant-time S-box = table   : yes (all 256)
Constant-time inverse = table : yes (all 256)
```

```
=====
NEW-CIPHER (512, 256)
```

```
--- ENCIPHERING ---
```

Key:

```
000102030405060708090A0B0C0D0E0F
101112131415161718191A1B1C1D1E1F
```

Plaintext:

```
404142434445464748494A4B4C4D4E4F
505152535455565758595A5B5C5D5E5F
606162636465666768696A6B6C6D6E6F
707172737475767778797A7B7C7D7E7F
```

Ciphertext:

```
455E363CEE0C7FA591B879867BED0045
93B78FA6B776AE4AEEED902A66022120
812A95EBEBE95758C86C9B416EADDD1C
B6BBA717F75EEC37EC1C470212D2A6A3
```

Success enciphering

```
=====
NEW-CIPHER (512, 256)
```

```
--- DECIPHERING ---
```

Key:

```
1F1E1D1C1B1A19181716151413121110
0F0E0D0C0B0A09080706050403020100
```

Ciphertext:

```
7F7E7D7C7B7A79787776757473727170
```

```
6F6E6D6C6B6A69686766656463626160
5F5E5D5C5B5A59585756555453525150
4F4E4D4C4B4A49484746454443424140
```

Plaintext:

```
87BBB0CE4414E29E2F5917D8CCBB7738
0232E85BD97BE57E486CB4FE7E1B1631
161D7A08710F5A3BD2182E1379F0EFC5
D5C2DA2865D85AA40791745A50555384
```

Success deciphering

```
Zero key / zero plaintext      : Success
D(E(P)) == P, 1000 random tests : Success
Avalanche, 2000 samples       : 256.2 / 512 bits (0.500)
Speed (constant-time C)       : 0.38 MB/s
Key setup                      : 0.587 ms
```

Timing test (duddect; pass if |t| < 4.5):

```
encryption, fixed vs random data : t = 0.25 PASS (no leak detected)
key schedule, fixed vs random key: t = 0.74 PASS (no leak detected)
```