Bitslicing the AEGIS

Frank Denis
Fastly, Inc.
fde@00f.net

Abstract. The AEGIS family of authenticated encryption algorithms is built on a large state whose update function is a cascade of dependent AES rounds. For bitslicing, this looks like the worst possible case.

Yet the parallelism is already inside a single update: for example, an AEGIS-128L update applies one AES round to all eight state blocks at once, filling an eight-block bitsliced data path by itself.

We describe a sliced representation that keeps the rest of the cipher cheap too and lets AEGIS-128L run 2.5 × fewer parallel AES rounds per byte than bitsliced AES-128-CTR.

On ARM, RISC-V, and WebAssembly, the resulting constant-time implementations are faster than side-channel-hardened AES-GCM on every platform measured, the X2 variants by factors of two to six. In pure JavaScript, where bitwise operators are exactly 32 bits wide, the same layout is faster than the widely used pure-JavaScript implementations of AES-GCM and ChaCha20-Poly1305.

1 Introduction

The AEGIS authenticated encryption algorithms [WP14] iterate the AES round function [Nat23] over a large state under continuous message injection.

For example, an AEGIS-128L update evaluates the round on eight 128-bit blocks and absorbs or encrypts 32 bytes in exchange.

Where each evaluation is a single hardware instruction, AEGIS is simple to implement securely and among the fastest authenticated ciphers in deployment. But without AES instructions, the picture changes, due to the traditional concerns about timing attacks on AES implementations.

And table-driven implementations leak through data-dependent memory accesses, which have repeatedly been turned into practical key recovery [Ber05, OST06].

The standard remedy is *bitslicing* [Bih97]: evaluate the cipher as a fixed sequence of bitwise word operations, whose timing cannot depend on the data. And bitsliced AES has powered constant-time AES-CTR and AES-GCM since the late 2000s [AP21, KS09, MN07, SS17].

But in order to achieve acceptable performance, bitslicing needs parallelism to fill the slices.

AES-CTR gets it from the mode, since counter blocks are independent. And at first, AEGIS looks like the worst case: every update depends on the full previous state, and nothing can be batched across the message.

This is actually quite the opposite: the parallelism bitslicing needs is already inside the primitive.

2 The AEGIS family, restated for bitslicing

2.1 Notation

Bytes are elements of $\{0,1\}^8$; bit k of a byte x is written $x[k]$, with bit 0 the least significant. We write $\oplus$ for XOR (extended bitwise to bytes, words and blocks), and $\wedge$ for bitwise AND.

Machine words have w bits: $x \gg n$ and $x \ll n$ denote zero-filling logical shifts, while $x \ggg n$ and $x \lll n$ denote rotation of a word towards the least and most significant positions respectively, so that bit j of $x \ggg n$ is bit $(j+n) \bmod w$ of x.

An *AES block* is a 16-byte value B, identified as usual with a 4×4 byte matrix $a_{r,c}$, $0 \le r, c \le 3$, where byte index $4c + r$ of B holds $a_{r,c}$ (column-major order, FIPS 197 [Nat23]).

2.2 The key-less AES round

The AES round function is the composition of SubBytes (the S-box s applied to each byte), ShiftRows (row r rotated left by r byte positions, $a_{r,c} \leftarrow a_{r,(c+r) \bmod 4}$), MixColumns (each column multiplied by the circulant matrix $\mathrm{circ}(2,3,1,1)$ over $\mathrm{GF}(2^8)$ with the AES reduction polynomial $x^8 + x^4 + x^3 + x + 1$), and a round-key addition.

AEGIS uses the AES round with the key addition exposed as an explicit operand: for blocks X and Y,

$$\text{AESRound}\ (X, Y) = \text{MixColumns}\ (\text{ShiftRows}\ (\text{SubBytes}\ (X))) \oplus Y.$$

We write $R = \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes}$ for the *key-less round*, so $\text{AESRound}\ (X, Y) = R(X) \oplus Y$, and $R^{\|}$ for the blockwise application of R to a tuple of blocks: $R^{\|}(X_0, ..., X_{n-1}) = (R(X_0), ..., R(X_{n-1}))$.

2.3 The AEGIS-128L, AEGIS-256, and AEGIS-X state updates

AEGIS-128L [DL25, WP14] maintains a state $S = (S_0, ..., S_7)$ of eight AES blocks. One update, parameterized by two message blocks M_0, M_1, is specified as

$$\begin{aligned} S'_0 &= \text{AESRound}\ (S_7, S_0 \oplus M_0), \quad S'_4 = \text{AESRound}\ (S_3, S_4 \oplus M_1), \\ S'_i &= \text{AESRound}\ (S_{i-1}, S_i), \qquad\qquad \text{for}\ \ i \in \{1, 2, 3, 5, 6, 7\}. \end{aligned}$$

AEGIS-256 maintains six blocks and absorbs one message block per update: $S'_0 = \text{AESRound}\ (S_5, S_0 \oplus M)$ and $S'_i = \text{AESRound}\ (S_{i-1}, S_i)$ for $i \in \{1, ..., 5\}$.

The parallel variants AEGIS-128XD and AEGIS-256XD [Den23, DL25] run D independent instances of the base cipher, separated by distinct constants folded into the state during initialization, over a message schedule that distributes consecutive rate blocks across instances; $D = 2$ is the relevant degree for the 64-bit targets considered here.

The AEGIS update function was designed so that all the AES rounds can be evaluated in parallel. For efficient bitslicing, we need to make that parallelism explicit and rearrange the update into a form that separates the three kinds of operations it performs: the AES round, the block rotation, and the message injection/keystream extraction.

Let ρ denote the cyclic rotation of block indices, $\rho(X_0, ..., X_{n-1}) = (X_{n-1}, X_0, ..., X_{n-2})$, i.e. $\rho(X)_i = X_{(i-1) \bmod n}$.

Since $\text{AESRound}\ (X, Y) = R(X) \oplus Y$ and the second operand enters linearly, each equation reads $S'_i = R(S_{i-1}) \oplus S_i \oplus m_i$, where m_i is the message block injected at position i (or zero).

Read all eight at once: the first operands are the blockwise round outputs with their indices shifted by one, i.e. the components of $\rho\big(R^{\|}(S)\big)$.

The whole update collapses to a single expression:

$$S' = \rho\big(R^{\|}(S)\big) \oplus S \oplus \widetilde{M},$$

with $\widetilde{M} = (M_0, 0, 0, 0, M_1, 0, 0, 0)$

and AEGIS-256 is the same expression over $n = 6$ blocks with $\widetilde{M} = (M, 0, ..., 0)$.

We call this the *parallel form* of the update; it splits the update into three layers with entirely different implementation characters:

1. $R^{\|}$: the key-less AES round applied to all n blocks *simultaneously*, precisely the computation a bitsliced AES implementation performs on n independent blocks, except that here the blocks belong to a single cipher instance processing a single message.
2. ρ: a relabeling of blocks. In a byte-oriented representation it is free (read indices off by one). What it costs in a sliced representation is the subject of Section 4.1: one masked intra-byte rotation per word.
3. $\oplus\, S \oplus \widetilde{M}$: XORs. Since every representation change we use is $\mathrm{GF}(2)$-linear, these commute with the representation and can be applied in whichever domain is cheapest.

The keystream is likewise blockwise and bit-local: AEGIS-128L outputs, per update,

$$Z_0 = S_1 \oplus S_6 \oplus (S_2 \wedge S_3), \quad Z_1 = S_2 \oplus S_5 \oplus (S_6 \wedge S_7),$$

and AEGIS-256 outputs $Z = S_1 \oplus S_4 \oplus S_5 \oplus (S_2 \wedge S_3)$.

Encryption XORs the keystream into the plaintext, and the *plaintext* blocks are absorbed by the next update.

Initialization seeds the state with constants and runs updates absorbing nonce and key material: 10 of them for AEGIS-128L, 16 for AEGIS-256; finalization runs 7 updates absorbing the data lengths, then XORs state blocks into the tag.

Every operation on the state is either an update in the parallel form or a bytewise boolean combination of state blocks; and the values entering and leaving the state touch a fixed, small subset of block positions.

AEGIS can therefore live its entire life inside any representation in which those two kinds of operations are cheap.

3 A sliced representation of parallel AES states

This section works with $n = 8$ blocks and 32-bit words; Section 5 and Section 7 lift both restrictions. A tuple $S = (S_0, ..., S_7)$ of AES blocks comprises $8 \times 128 = 1024$ bits; we re-arrange them into 32 words of 32 bits.

3.1 The layout

The *sliced representation* of $S = (S_0, ..., S_7)$, written $\mathrm{sl}(S)$, is a family of 32-bit words $W_{r,k}$, one per AES row $r \in \{0, ..., 3\}$ and bit index $k \in \{0, ..., 7\}$, filled as

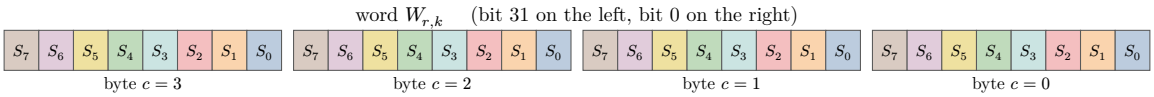
$$W_{r,k}[8c + b] = a^{(b)}_{r,c}[k], \quad c \in \{0, ..., 3\}, \ \ b \in \{0, ..., 7\},$$

where $a^{(b)}$ is the state matrix of block S_b.

The map sl is a fixed permutation of bit positions; in particular it is a bijection and GF(2)-linear, both used freely below.

Each word gathers, for one AES row r and one bit index k, the k-th bit of all 32 row-r bytes across the eight blocks. Inside a word, the byte position selects the AES *column* c, and the bit position within that byte (the *lane* b) selects the *block* S_b.

Figure 1 shows the anatomy of one word and the full 4×8 family.



	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
row $r=0$	$W_{0,0}$	$W_{0,1}$	$W_{0,2}$	$W_{0,3}$	$W_{0,4}$	$W_{0,5}$	$W_{0,6}$	$W_{0,7}$
row $r=1$	$W_{1,0}$	$W_{1,1}$	$W_{1,2}$	$W_{1,3}$	$W_{1,4}$	$W_{1,5}$	$W_{1,6}$	$W_{1,7}$
row $r=2$	$W_{2,0}$	$W_{2,1}$	$W_{2,2}$	$W_{2,3}$	$W_{2,4}$	$W_{2,5}$	$W_{2,6}$	$W_{2,7}$
row $r=3$	$W_{3,0}$	$W_{3,1}$	$W_{3,2}$	$W_{3,3}$	$W_{3,4}$	$W_{3,5}$	$W_{3,6}$	$W_{3,7}$

Fig. 1. The sliced representation of eight AES blocks. Top: one 32-bit word interleaves the four AES columns (bytes) and the eight blocks (lanes, color-coded). Bottom: the state is the family of $4 \times 8 = 32$ such words, one per (row, bit index) pair; each row of the table is one S-box circuit group, and the AES row r sets the uniform rotation amount of ShiftRows.

This layout is not new: it is the *barrel-shiftrows* representation of Adomnicai and Peyrin [AP21], introduced for batches of eight independent AES blocks on 32-bit microcontrollers.

Folding the four columns of a block into the word (sound because every AES layer treats columns uniformly) is what lets eight blocks fill a 32-bit word, and keeping the four rows in separate words is what turns ShiftRows into one uniform rotation per word.

All that changes here is what fills the eight lanes: the eight blocks of one AEGIS-128L state instead of eight counter-mode blocks. The layout is important for what follows because the AEGIS-specific operations also turn out cheap in it.

3.2 The AES layers in the sliced domain

SubBytes. As in any bitsliced AES, evaluating an 8-bit boolean circuit for the S-box with bitwise word operations on $(W_{r,0}, ..., W_{r,7})$ applies the S-box to all 32 row-r bytes at once; the four rows

take four independent, identical evaluations of one circuit. Circuit choice and the use of this four-fold symmetry are deferred to Section 6.

ShiftRows. One word rotation per word,

$$W'_{r,k} = W_{r,k} \ggg 8r,$$

since a byte moving from column $(c + r) \bmod 4$ to column c displaces its bits by $8r$ positions. Row 0 is untouched; rows 1, 2, 3 cost 24 rotations for the layer, at most three operations each without a rotate instruction, or one byte shuffle per vector (Section 6.2 exploits this).

MixColumns. Standard as well; we only fix notation that later sections reuse. With $3 \cdot y = \mathrm{xt}(y) \oplus y$, row r of the matrix $\mathrm{circ}(2, 3, 1, 1)$ reads

$$a'_r = \mathrm{xt}\big(a_r \oplus a_{r+1}\big) \oplus a_{r+1} \oplus a_{r+2} \oplus a_{r+3}$$

(indices modulo 4), and under slicing xt is pure wiring: it shifts each bit plane up by one and taps plane 7 back into planes 1, 3 and 4, per the reduction polynomial. Abbreviating

$$D_{r,k} = W_{r,k} \oplus W_{(r+1) \bmod 4,k}, \quad \Sigma_{r,k} = W_{(r+1) \bmod 4,k} \oplus W_{(r+2) \bmod 4,k} \oplus W_{(r+3) \bmod 4,k},$$

the layer becomes

$$W'_{r,0} = D_{r,7} \oplus \Sigma_{r,0}, \qquad W'_{r,k} = D_{r,k-1} \oplus \Sigma_{r,k} \quad (k \in \{2, 5, 6, 7\}),$$
$$W'_{r,k} = D_{r,k-1} \oplus D_{r,7} \oplus \Sigma_{r,k} \quad (k \in \{1, 3, 4\}),$$

for $32 + 32 + 32 + 12 = 108$ XORs for eight blocks, matching [AP21].

There is no AddRoundKey: AEGIS has no key schedule, and the feed-forward XOR that plays its role is handled in Section 4.

3.3 Conversion networks

Entering and leaving the sliced domain is a fixed permutation of 1024 bit positions, computed as usual with `SWAPMOVE` exchanges [AP21]: for words x, y, mask m and shift n,

$$t = (y \oplus (x \gg n)) \wedge m, \quad y \leftarrow y \oplus t, \quad x \leftarrow x \oplus (t \ll n),$$

six operations that swap the bits of y selected by m with the bits of x selected by $m \ll n$.

The permutation sl factors into two classical transposes: a *byte stage* transposing the 4×4 byte matrix of each block (4 swaps per block, 32 total), and a *bit stage* that, for each row, transposes block index against bit index, a three-level butterfly with strides 4, 2, 1 over the eight row words (12 swaps per row, 48 total).

The masks and shifts are listed in Algorithm 1.

The $32 + 48 = 80$ swaps are 480 word operations for eight blocks, or 3.75 per state byte. Every `SWAPMOVE` is an involution, so running the same list backwards computes sl^{-1} at the same cost.

These 480 operations are the tax paid every time the representation boundary is crossed, and they shape the implementation strategy: a bitsliced AES-CTR implementation crosses it twice per 10-round batch, while a naive sliced AEGIS would cross it *twice per round*.

But we can avoid paying full price.

4 AEGIS in the sliced domain

In its parallel form, one AEGIS-128L update is $S \leftarrow \rho\big(R^{\|}(S)\big) \oplus S \oplus \widetilde{M}$, and the keystream is a bytewise boolean combination of state blocks. $R^{\|}$ is exactly one sliced AES round (Section 3.2). This section gives sliced-domain algorithms for everything else: the rotation ρ (the diagonal wiring of Figure 2), the sparse injection $\oplus\, \widetilde{M}$, and the keystream extraction.

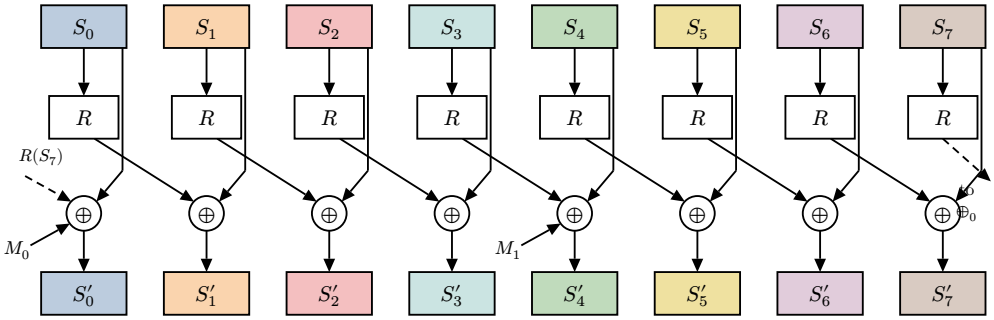


Fig. 2. The AEGIS-128L update in its parallel-round form (Section 2). All eight key-less rounds R form one sliced round evaluation $R^{\|}$; the diagonal wiring is the index rotation ρ; the vertical wiring is the feed-forward $\oplus\, S$; M_0, M_1 enter blocks 0 and 4. In the sliced domain, ρ is the intra-byte rotation of Section 4.1.

4.1 The block-index rotation

In a byte-oriented representation, ρ is free: it is absorbed into addressing.

Once the state is sliced, however, the block index is buried as the lane index *inside every byte of every word*, and a permutation of blocks becomes a genuine bit permutation. The layout was chosen so that this permutation is as regular as possible: ρ is realized by rotating *every byte of every word* left by one bit position, i.e. replacing each word $W_{r,k}$ by

$$\big((W_{r,k} \wedge \texttt{0x7f7f7f7f}) \ll 1\big) \vee \big((W_{r,k} \wedge \texttt{0x80808080}) \gg 7\big).$$

To see why, look at one byte of one word (Figure 3). Lane b of the rotated byte receives what lane $(b-1) \bmod 8$ held: bit k of byte $a_{r,c}$ of block $S_{(b-1) \bmod 8}$. That is exactly the sliced image of the rotated tuple $\rho(S)$, and no other coordinate (r, k, c) moves.

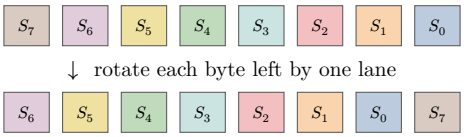


Fig. 3. One byte of one word under the intra-byte rotation (lane 7 on the left). Every byte of every one of the 32 words undergoes the same one-bit rotation: after it, lane b carries block $S_{(b-1) \bmod 8}$, realizing ρ.

The cost is five word operations per word (two ANDs, two shifts, one OR), or 160 for the state. That is one and a half times the MixColumns layer, and a sixth of the 960-operation unpack–repack round trip it makes unnecessary.

And this is why the state can stay resident in sliced form: the one inter-block operation of AEGIS happens to be a *uniform intra-byte* permutation under this layout.

Note also that ρ commutes with $R^{\|}$, so an implementation may rotate either the round input or the round output, whichever schedules better.

The feed-forward $\oplus\, S$ operation is trivial: sl is linear, so the XOR is performed directly on the 32 resident words.

4.2 Sparse message injection

The update XORs $\widetilde{M} = (M_0, 0, 0, 0, M_1, 0, 0, 0)$ into the state. Linearity again allows the injection to happen in the sliced domain,

$$W \leftarrow W \oplus \mathrm{sl}\big(\widetilde{M}\big),$$

at the cost of slicing $\widetilde{M}$, but $\widetilde{M}$ is supported on two of the eight blocks, and a conversion network specialized to this support is much smaller than the general one.

The reduction rests on one observation: a `SWAPMOVE` whose two selected bit groups are both zero does nothing and can be dropped.

Order the bit-stage butterfly of Section 3.3 by *decreasing* stride (block-distance 4, then 2, then 1) and trace where the data lives (Figure 4).

The byte stage acts within blocks, so only blocks 0 and 4 keep their four swaps each. In the bit stage, per row: at distance 4 only the pair $\{0, 4\}$ carries data (1 swap), and the data stays confined to lanes $\{0, 4\}$; at distance 2 the pairs $\{0, 2\}$ and $\{4, 6\}$ pick it up (2 swaps), spreading it to $\{0, 2, 4, 6\}$; at distance 1 all four pairs are live (4 swaps).

Of the 80 swaps, $8 + 4(1 + 2 + 4) = 36$ survive, everything else can be optimized out.

For an input supported on block 0 alone (the AEGIS-256 case) the same walk leaves $4 + 4(1 + 2 + 4) = 32$.

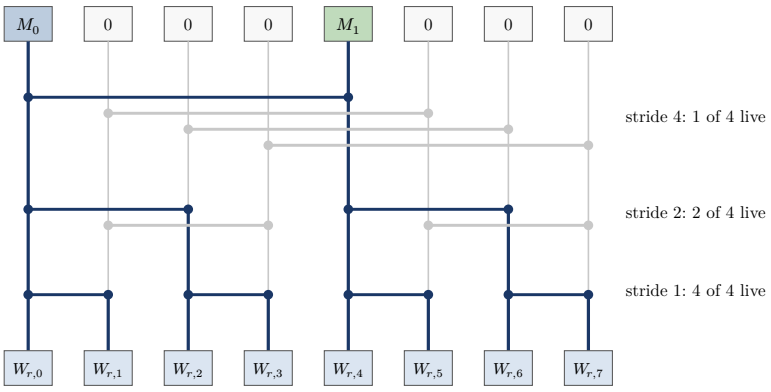


Fig. 4. The bit-stage butterfly of one row group r on an injection supported on blocks 0 and 4. Wires are the eight row-r words; thick segments carry data. Grayed swaps exchange zeros and are dropped: $1 + 2 + 4$ live swaps per row, plus the byte-stage swaps of the two live blocks, give 36 of 80. Read bottom-up, the same picture is the partial extraction of blocks 0 and 4.

The stride order is not cosmetic. Run the levels in increasing order instead and the first level tears the support $\{0, 4\}$ apart, raising the count from 36 to 48. Decreasing order puts $\{0, 4\}$ on a pair of the first level, which keeps the data confined longest. (For single-block support the count is 32 either way.)

To recover only blocks 0 and 4 from a fully populated sliced state, run the swaps of the same reduced network in reverse order.

Each swap is an involution, and the swaps that the reduced network omits only shuffle wires that end up discarded: they lie outside the cone of influence of the two wanted outputs. Partial extraction therefore costs the same 36 swaps (32 for one block).

Injecting a message thus costs $36 \times 6 = 216$ word operations for the sparse pack, 45% of a full conversion, plus 32 XORs to add it into the state; extracting two designated blocks costs 216.

4.3 Keystream extraction without unpacking

Each update of AEGIS-128L releases 32 bytes of keystream,

$$Z_0 = S_1 \oplus S_6 \oplus (S_2 \wedge S_3), \qquad Z_1 = S_2 \oplus S_5 \oplus (S_6 \wedge S_7).$$

These are *bytewise* boolean combinations of state blocks: bit k of byte (r, c) of Z_0 depends only on bit k of byte (r, c) of S_1, S_2, S_3, S_6. In the sliced domain all four source bits live in the *same word* $W_{r,k}$, at the four lane positions $1, 2, 3, 6$ of the same byte. Aligning lanes by shifting therefore computes the keystream with a handful of operations per word, with no representation change (Figure 5). For every word, compute

$$E(W) = ((W \gg 1) \oplus (W \gg 6) \oplus ((W \gg 2) \wedge (W \gg 3))) \wedge \texttt{0x01010101},$$
$$E'(W) = (((W \gg 2) \oplus (W \gg 5) \oplus ((W \gg 6) \wedge (W \gg 7))) \wedge \texttt{0x01010101}) \ll 4;$$

the words $E(W_{r,k}) \vee E'(W_{r,k})$ form $\mathrm{sl}(Z_0, 0, 0, 0, Z_1, 0, 0, 0)$.

All the shift amounts are below 8 and therefore stay within each byte.

In E, the shifts by $1, 6, 2, 3$ bring lanes $1, 6, 2, 3$ onto lane 0, where the XOR/AND combination evaluates $S_1 \oplus S_6 \oplus (S_2 \wedge S_3) = Z_0$ bit by bit; the mask clears every other lane. E' does the same for $Z_1 = S_2 \oplus S_5 \oplus (S_6 \wedge S_7)$, the trailing shift parks the result in lane 4, and the OR merges the two disjoint supports.

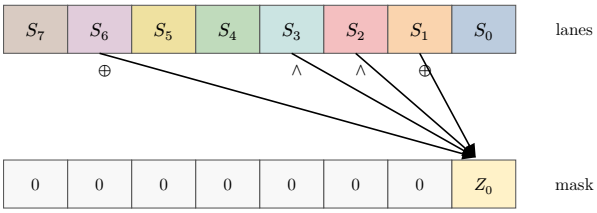


Fig. 5. Keystream extraction inside one byte of one word: lanes 1 and 6 are XORed, lanes 2 and 3 are ANDed, into lane 0; every other lane is masked to zero. The same shifts act on all four bytes of all 32 words simultaneously, yielding the Z_0 half of the sliced keystream $(Z_0, 0, ..., 0)$ without leaving the sliced domain; the Z_1 half is the analogous E', parked in lane 4.

Producing both keystream lanes costs at most 16 operations per word (six distinct shifts, two ANDs, four XORs, two masks, one shift, one OR), or 512 for the state. The two populated lanes are then recovered with the 36-swap partial extraction of Section 4.2, 216 operations.

The total is 728 word operations per 32 bytes of keystream. This replaces the full unpack–repack round trip (2×480 operations) a naive implementation would pay, and more importantly it leaves the resident state untouched in sliced form.

Decryption can be done exactly the same way: the ciphertext is XORed against the extracted keystream, and the resulting plaintext blocks are injected via the sparse network.

4.4 Persistent vs transient slicing

Two implementation strategies are possible, depending on whether the state is kept in sliced form for its entire life or converted back to byte form after each update.

Persistent slicing. The state is converted once at initialization and stays sliced for its entire life. Constant injections ((N, K) and the finalization lengths) are sliced *once* and reused.

Transient slicing. The state lives in byte form; each update packs a copy, applies the sliced round, and unpacks the result, paying 2×480 conversion operations per update. In exchange, ρ costs nothing (it is absorbed into the indexing of the feed-forward XOR), injection is 8 XORs, and keystream extraction is 24 operations.

Ideally, we should use persistent slicing in the phases that extract no keystream: initialization and associated data absorption.

Associated data costs 440 non-round operations per update against ≈ 1000, and the 17 constant-injection updates of AEGIS-128L's initialization and finalization (10 plus 7; AEGIS-256 initializes with 16) drop to ≈ 220, a $4\times$ reduction.

Conversely, transient slicing keeps the byte-form state available at all times, which simplifies short-message processing.

For optimal performance, the two can be mixed, e.g. persistent everywhere except the en/decryption phase.

Implementation-ready pseudo-code for the complete persistent-policy cipher is available in Appendix A.

5 AEGIS-256

The AEGIS-256 state has $n = 6$ blocks: too few to fill the eight lanes of the sliced layout, and not a power of two, so no butterfly fits them directly.

The remedy is to simply place blocks $S_0, ..., S_5$ in lanes 0–5 and leave lanes 6 and 7 identically zero. All machinery of Section 3 and Section 4 applies unchanged; the sliced round simply computes R of two all-zero blocks alongside the six real ones. The 25% occupancy loss mirrors the 6-versus-8 block economics of the cipher itself.

Only the rotation needs adjustment, since ρ must now cycle six lanes. A narrower window does it: replace every word by

$$((W \wedge \texttt{0x1f1f1f1f}) \ll 1) \vee ((W \wedge \texttt{0x20202020}) \gg 5).$$

Within each byte, the first term picks up lanes 0–4 and moves them to lanes 1–5; the second moves lane 5 back to lane 0 (Figure 6). That relabeling, $b \to (b + 1) \bmod 6$, is exactly ρ on six blocks, for the same reason as in Section 4.1. Better, neither term reads lanes 6 or 7, and neither shifted mask

(`0x1f` $\ll 1 =$ `0x3e`, `0x20` $\gg 5 =$ `0x01`) writes them, so the padding lanes come out zero *no matter what they contained*.

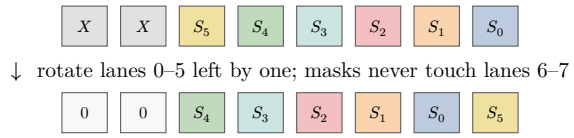


Fig. 6. One byte of one word under the AEGIS-256 windowed rotation (lane 7 on the left). Lanes 6–7 hold arbitrary garbage X (after a round, bits of $R(0) \neq 0$); the rotation cycles the six live lanes and writes zeros to the padding lanes, restoring the invariant at no extra cost.

Note that the padding lanes do *not* stay zero through a round due to the S-box, but they never affect output: the round is lane-independent, and only the six live lanes feed injection, keystream extraction, and the tag. The windowed rotation re-zeroes them before the feed-forward XOR (Algorithm 5 rotates the round output).

The cost is the same as in the eight-lane case: five operations per word. The sparse injection of the single message block M into lane 0 uses the 32-swap network of Section 4.2, and the keystream $Z = S_1 \oplus S_4 \oplus S_5 \oplus (S_2 \wedge S_3)$ is extracted into lane 0 the same way as for AEGIS-128L,

$$E_6(W) = ((W \gg 1) \oplus (W \gg 4) \oplus (W \gg 5) \oplus ((W \gg 2) \wedge (W \gg 3))) \wedge \texttt{0x01010101},$$

followed by the 32-swap partial unpack of one lane.

6 The AEGIS round function

The sliced round of Section 3 is dominated by its only nonlinear layer.

This section discusses the choice of S-box circuit, and shows how the four-fold symmetry of the layout turns the entire round into straight-line vector code without any data transposition.

6.1 S-box circuits in software

The sliced SubBytes layer is four evaluations of one boolean circuit for the AES S-box (Section 3.2), a problem with a rich literature.

Boyar and Peralta gave a 115-gate circuit with 32 AND gates and 83 XOR/XNOR gates of depth 28 [BP10], and later a depth-16 variant with 128 gates (34 of them AND) [BP12].

Maximov and Ekdahl, optimizing over a richer gate set (NAND, NOR, MUX), give a “fast” forward S-box of 130 gates with a critical path of 12 gates and a record-small variant of 102 gates at depth 24 [ME19], which turns out to be very efficient for software implementations.

Note that the software cost model is not the same as the hardware one.

Indeed, the gate count translates to instruction count only when the gate sets match: XNOR, NAND and NOR cost one instruction on some ISAs and two on others, and a MUX costs three logic operations. So a hardware-small circuit is not automatically software-small.

Depth matters only through instruction-level parallelism, and the four independent row groups (eight, with the X2 variants of Section 7) keep even narrow circuits dense in the pipeline.

Finally, register pressure is a major limiter, and the row-group decomposition is what makes 16-register ISAs workable: one group occupies eight words rather than the full 32.

The right choice depends on the target’s logic instructions and issue width more than on raw gate count.

Note that the affine constant of the S-box folds into XNORs of the circuit’s bottom linear layer at no extra cost.

6.2 Transpose-free vectorization

Packing several sliced words into one vector register is standard practice; the only decision is which words to group. Grouping the four rows of each bit index,

$$V_k = \big(W_{0,k}, W_{1,k}, W_{2,k}, W_{3,k}\big), \quad k \in \{0, ..., 7\},$$

works out particularly well because every layer treats the row index either uniformly or cyclically.

SubBytes is one evaluation of the S-box circuit on $V_0, ..., V_7$ with vector logic (the four row instances of Section 3.2, fused).

ShiftRows must rotate lane r by $8r$ bits, and four per-lane rotations by multiples of 8 are a single byte shuffle per vector.

MixColumns is the equations of Section 3.2 with the row-index cycling expressed as the lane rotation $\lambda(V)_r = V_{(r+1) \bmod 4}$: with $D_k = V_k \oplus \lambda(V_k)$ and $\Sigma_k = \lambda(V_k) \oplus \lambda^2(D_k)$, the same three formulas apply, for 16 lane shuffles and 27 vector XORs.

No data moves between different vectors V_k, so nothing is ever transposed between memory and vector form.

Storing the four lanes of each V_k contiguously serves scalar targets too: the S-box processes row groups one at a time anyway, and in the linear layers the four rows of a plane undergo identical operations that compilers merge into vector instructions on their own.

One caveat from practice: a measured WebAssembly engine executes the ShiftRows byte shuffle slower than the three-instruction scalar rotations it replaces, so the scalar path over the same memory layout remains the fastest option there.

The 4-lane vectors are 128 bits for AEGIS-128L and AEGIS-256, and 256 bits for the X2 variants of the next section, matching the two most common vector register sizes.

7 Wider words and the X-variants

AEGIS-128L's sliced state has a *natural word size*: eight blocks times four columns gives 32-bit words, and nothing in Section 3 knows what lies beyond bit 31. On a 64-bit data path, half of every register would idle.

The parallel variants of the AEGIS family [Den23, DL25] are exactly the cipher-level counterpart. AEGIS-128X2 runs two independent AEGIS-128L instances under a common message schedule: each 64-byte rate chunk is spread across the two instances' injection blocks, the instances are kept apart by distinct context constants folded into blocks 3 and 7 at every initialization update, and their tags are XORed at the end.

7.1 Two instances in one 64-bit word

Wider registers naturally hold more slices. Extend the layout of Section 3.1 to 64-bit words, with bytes 0–3 of each word slicing instance 0 and bytes 4–7 slicing instance 1: the low and high 32-bit halves of every word are the two instances' 32-bit slices, undisturbed.

Almost every formula acts within bytes or within bit planes and never moves data across bit positions, so it applies to 64-bit words verbatim with doubled masks: SubBytes, MixColumns, the keystream combinations and the feed-forward are purely bitwise; the intra-byte rotation of Section 4.1, with masks `0x7f7f7f7f7f7f7f7f` and `0x8080808080808080`, advances both instances' block rings at once, since every byte belongs entirely to one instance; and the conversion networks swap at distances 8, 16, 1, 2 and 4 bits, none of which crosses the 32-bit boundary, so the swap counts of Section 3.3 and Section 4.2 hold while each swap carries twice the data.

The one layer that notices the boundary is ShiftRows: a full 64-bit rotation would carry bytes across the instance boundary, so the two halves are rotated separately,

$$W' = ((W \gg 8r) \wedge M_r) \vee \left((W \ll (32 - 8r)) \wedge \overline{M_r}\right), \quad r \in \{1, 2, 3\},$$

$$M_1 = \texttt{0x00ffffff00ffffff}, \quad M_2 = \texttt{0x0000ffff0000ffff},$$

$$M_3 = \texttt{0x000000ff000000ff},$$

five operations per word in scalar code, or, on vector units, the same byte permutation as in Section 6.2, now on 256-bit vectors.

Per byte, everything except ShiftRows halves in cost.

Section 8 puts the overall gain at roughly 1.9 ×, and the measurements of Section 9 confirm 1.3 –1.7 × on 64-bit RISC-V and WebAssembly targets.

AEGIS-256X2 doubles the same way, with the six-lane embedding and windowed rotation of Section 5 in each half, and AEGIS-128X4 has a natural word size of 128 bits and slices onto SIMD registers by the same construction.

7.2 Two interleaved 32-bit states

On actual 32-bit platforms, the same X2 ciphers can instead be implemented as two *separate* 32-word sliced states whose instruction streams are interleaved.

A single S-box evaluation is a dependency chain of depth 12–28 (Section 6.1) over eight registers, which under-utilizes dual-issue pipelines; a second independent instance doubles the available instruction-level parallelism, at the price of a doubled working set.

Which side wins is target-specific.

The X2 measurements of Section 9 use the doubled 64-bit layout, since both platforms support 64-bit integers directly.

8 Cost analysis

All costs below are word operations (logic, shift, rotate, move) on a CPU with native rotates, counting `SWAPMOVE` as 6 operations and writing G for the software gate count of the chosen S-box circuit per row group ($G \approx 115$–130, Section 6.1); vector execution divides the logic counts by the lane count but does not change the structure.

One round evaluation inside an update (Section 3) costs:

$$\underbrace{4G}_{\text{SubBytes}} + \underbrace{24}_{\text{ShiftRows}} + \underbrace{108}_{\text{MixColumns}} + \underbrace{32}_{\text{copy}} = 4G + 164 \quad (\approx 624 \text{ ops for } G = 115)$$

for eight blocks, i.e. 32 bytes of AEGIS-128L message. The policy-dependent costs per update are collected in Table 1.

Table 1. Word-operation counts for one AEGIS-128L encryption update under the two residency policies of Section 4.4. For updates that extract no keystream (associated data), the persistent column drops to 440 and the transient column to 1000; for the 17 constant-injection updates of initialization and finalization, the persistent column drops to 224.

per 32-byte update	persistent	transient
conversions (pack/unpack of state)	—	$2 \times 480 = 960$
rotation ρ (Section 4.1)	160	— (relabeling)
feed-forward XOR	32	32
message injection (Section 4.2)	$216 + 32$	8
keystream extraction (Section 4.3)	$512 + 216$	24
total non-round	1168	1024
total with round ($G = 115$)	1792	1648

Against bitsliced AES-128-CTR. In the same representation, a CTR batch packs 8 counter blocks, runs 10 rounds (each also costing $4G + 164$, the state copy giving way to a 32-XOR round-key addition) and unpacks 128 bytes of keystream: $480 + 10 \times (4G + 164) + 480 + 32 \approx 7230$ operations, or 57 per byte at $G = 115$. (This counts MixColumns in all ten rounds; AES omits it from the last, which would save 108 operations, a 1.5% correction.) AEGIS-128L (transient) costs $1648/32 \approx 52$ per byte.

The round work alone differs by the factor 2.5 from the introduction; conversions and extraction dilute it to about 1.1 × at equal word width, since AEGIS-128L's conversion traffic per byte is four times CTR's.

The deeper difference is authentication: CTR is not an AEAD. A constant-time GHASH without carry-less multiplication costs a multiple of the CTR budget (in Käsper and Schwabe's measurements, constant-time AES-GCM runs 2.9 × slower than its AES-CTR component alone [KS09]) whereas AEGIS authenticates with seven finalization updates, a per-message constant.

Against *authenticated* baselines the advantage is therefore 2–3 × in operation counts, consistent with Section 9.

Doubling the width. AEGIS-128X2 under the doubled layout of Section 7 processes 64 bytes per update at almost the 32-byte operation counts: conversions, S-box, MixColumns, rotation, injection and extraction are width-oblivious, and only ShiftRows grows ($24 \to 120$ scalar operations).

The transient policy gives $(4G + 260) + 1024 \approx 1744$ operations per 64 bytes, just over 27 per byte: a 1.9 × improvement over AEGIS-128L wherever 64-bit logic throughput is double 32-bit. (A fair comparison notes that bitsliced CTR also scales with word width by batching 16 blocks; at $w = 64$ it costs about 32 per byte.

The distinction is that CTR buys its parallelism from the mode and can always fill any word, while AEGIS must buy it from the cipher degree, which is precisely why the X-variants exist.)

AEGIS-256. With six-of-eight lane occupancy and a 16-byte rate, the persistent policy costs the round plus $160 + 32 + (192 + 32) + (320 + 192) \approx 930$ non-round operations per update (E_6 extraction uses 10 operations per word and a 32-swap partial unpack), about 97 operations per byte at $G = 115$, a factor 1.7 over persistent AEGIS-128L at 56 per byte: the 2 × rate disadvantage, attenuated by the shared fixed costs.

Per-message overhead. AEGIS-128L's initialization and finalization together run 17 constant-injection updates, about $17 \times (624 + 224) \approx 14.4$ k operations persistent (twice that transient).

Counting update bodies only understates the fixed cost slightly: the one-off conversions at the boundaries (the initial pack (480), the sparse packs of the nonce–key and length pairs (2×216), the length-block partial unpack (192), and the final unpack for the tag (480)) add another ≈ 1.6 k, for a total near 16 k, the bulk-rate equivalent of some 300 bytes.

The techniques of this paper therefore shine brightest for messages upwards of a few hundred bytes; for very short messages the fixed cost dominates regardless of representation, as it does for every AEGIS implementation.

9 Measurements

The techniques of this paper were implemented[1] in portable C exposing both residency policies, the scalar and 4-lane vector rounds of Section 6.2, and the 64-bit doubled layout of Section 7.

Each measurement encrypts and authenticates 16-kilobyte messages in a loop on a single core, and reports plaintext bits over wall-clock time, tag computation included.

All protected baselines are bitsliced constant-time AES-GCM implementations from widely deployed libraries (OpenSSL, BoringSSL), built and run on the same machines (Table 2).

Table 2. Encryption throughput in Mb/s against side-channel-protected AES-128-GCM baselines (OpenSSL 3.3 bitsliced on Cortex-A53, BoringSSL bitsliced elsewhere). "Speedup" spans AEGIS-128L to AEGIS-128X2 where both were measured (AEGIS-128L only on Cortex-A53).

platform	AES-GCM (protected)	AEGIS-128L (this work)	AEGIS-128X2 (this work)	speedup
ARM Cortex-A53 (64-bit)	261	423	—	1.6 ×
SpacemiT X60 (RISC-V, no AES ext.)	137	193	333	1.4–2.4 ×
SiFive U74 (RISC-V, no AES ext.)	130	182	311	1.4–2.4 ×
WebAssembly (Apple M1 host)	480	2241	2912	4.7–6.1 ×

On an ARM Cortex-M4 microcontroller, bitsliced AEGIS-128L runs at 185 cycles per byte, against 246 for fixsliced AES-128-GCM with protected GHASH and 208 for *unprotected* table-based AES-128-GCM: the constant-time AEGIS implementation beats even the leaky AES-GCM one.

The same inversion appears on WebAssembly, where constant-time AEGIS-128L (2241 Mb/s) more than doubles unprotected table-based AES-128-GCM (1040 Mb/s).

JavaScript's bitwise operators work on 32-bit integers, which is exactly the natural word size of AEGIS-128L's sliced state. A pure TypeScript implementation[2] of the same layout (no native code, no SIMD) was benchmarked on an Apple M5 Max under the Bun runtime against the pure-JavaScript AES-GCM and ChaCha20-Poly1305 of the widely used noble-ciphers library (Table 3).

Bitsliced AEGIS-128L is the fastest of the measured pure-JavaScript AEADs at every message size, reaching 324 MB/s (megabytes per second, in contrast with the megabit figures of Table 2) on

[1] Our implementation is available at https://github.com/aegis-aead/aegis-bitsliced.

[2] The pure-TypeScript implementation is available at https://github.com/jedisct1/js-aegis-aead.

1 MB messages: 1.8 × the pure-JavaScript AES-128-GCM and 1.3 × ChaCha20-Poly1305, a cipher expressly designed for fast software implementation.

The parallel variants bring no speedup there: with no fast 64-bit integers and no vectors, their extra parallelism has nothing to map onto, exactly as the width analysis of Section 7 predicts.

Table 3. Encryption throughput in MB/s of a pure-TypeScript implementation of the sliced representation, Apple M5 Max under the Bun runtime; AES-GCM and ChaCha20-Poly1305 are the pure-JavaScript implementations of the noble-ciphers library.

algorithm (pure JS/TS)	64 B	1 KB	16 KB	1 MB
AEGIS-128L (this work)	50	224	310	324
ChaCha20-Poly1305	36	175	248	256
AEGIS-256 (this work)	36	146	198	204
AES-128-GCM	13	74	150	182
AES-256-GCM	12	66	122	143

On the Cortex-A53, swapping ten published circuits [Jeo+25, ME19] into the same AEGIS-128L implementation spans 377–423 Mb/s, a 12% spread, with the Maximov–Ekdahl "fast" circuit on top.

Gate count alone does not predict the ranking, which shifts with the target's logic instructions and issue width as discussed in Section 6.1.

Finally, for calibration against unprotected code: table-based AEGIS-128L runs at 782 Mb/s on the Cortex-A53 (1.85 × the bitsliced figure) and 507 Mb/s on the U74 (2.8 ×).

Protection thus costs a factor of 2–3 where table code runs at full speed, but it costs nothing, and indeed pays, relative to AES-GCM, which is the relevant alternative for an application that needs an AEAD.

10 Conclusion

AEGIS was designed for CPUs with AES instructions.

But we show that the design travels well to machines that have none: the state update *is* a parallel AES round plus three lane-local operations, so one AEGIS instance saturates a bitsliced data path on its own, and under the right layout every surrounding operation collapses to masked shifts and partial conversion networks.

The resulting constant-time implementations were faster than protected AES-GCM in every configuration we measured, and in some settings faster than unprotected AES-GCM.

The X2 variants convert spare word width into nearly proportional throughput, and in plain JavaScript, whose integer operations are exactly as wide as the sliced state's natural word, the same layout makes AEGIS-128L faster than the widely used pure-JavaScript implementations of both AES-GCM and ChaCha20-Poly1305.

A Pseudo-code reference

This appendix collects the algorithms of the paper as language-agnostic pseudo-code, in a form intended to be translated line by line into any language with unsigned 32-bit (or 64-bit) integers and bitwise operators.

Conventions: `^`, `&`, `|`, `~` are XOR, AND, OR, NOT; `<<`/`>>` are zero-filling shifts; `rotr32(x, n)` rotates towards the least significant bit; `<-` is assignment; all values are 32-bit words unless noted.

The sliced state is stored as a flat array `w[0..31]` with $W_{r,k} =$ `w[4k + r]`, the plane-major order recommended in Section 6.2; `le32(B, j)` reads bytes $4j..4j+3$ of a block B as a little-endian integer, and `LE64(n)` is the 8-byte little-endian encoding of n.

Every listing was transcribed from an executable model verified against the specification test vectors (Section 9).

For the AEGIS-128X2 forms, widen the words to 64 bits, double every mask, and replace `rotr32` by the per-half rotation, as described in Section 7.

Algorithm 1. `SwapMove` and the conversion network of Section 3.3. `Unpack` recovers byte form; each swap is an involution, so running the same list backwards inverts the permutation.

```
procedure SwapMove(ref a, ref b, mask, n):        // 6 word operations
    t <- (b ^ (a >> n)) & mask
    b <- b ^ t
    a <- a ^ (t << n)

procedure Pack(ref w[0..31]):
    // in:  w[4b + j] = le32(S_b, j), b in 0..7, j in 0..3
    // out: W[r,k] at w[4k + r]  (80 SwapMoves)
    for b in 0..7:                                // byte stage
        SwapMove(w[4b+0], w[4b+1], 0x00ff00ff, 8)
        SwapMove(w[4b+2], w[4b+3], 0x00ff00ff, 8)
        SwapMove(w[4b+0], w[4b+2], 0x0000ffff, 16)
        SwapMove(w[4b+1], w[4b+3], 0x0000ffff, 16)
    for r in 0..3:                                // bit stage, strides 4, 2, 1
        for j in (0,1,2,3): SwapMove(w[4j+r], w[4(j+4)+r], 0x0f0f0f0f, 4)
        for j in (0,1,4,5): SwapMove(w[4j+r], w[4(j+2)+r], 0x33333333, 2)
        for j in (0,2,4,6): SwapMove(w[4j+r], w[4(j+1)+r], 0x55555555, 1)

procedure Unpack(ref w[0..31]):
    apply the 80 SwapMoves of Pack in reverse order
```

Algorithm 2. Sparse conversion and partial extraction (Section 4.2). The 36-swap `PackSparse04` slices $(M_0, 0, 0, 0, M_1, 0, 0, 0)$; running its swap list backwards on a fully populated state recovers blocks 0 and 4 only. `PackSparse0`/`PartialUnpack0` (32 swaps) are the same with the $b = 4$ byte-stage group removed and $M_1 = 0$; the same 32-swap network recovers any single block b by targeting lane b.

```
function PackSparse04(M0, M1) -> w[0..31]:
    w[0..31] <- 0
    for j in 0..3: w[j] <- le32(M0, j); w[16+j] <- le32(M1, j)
    for b in (0, 4):                              // byte stage (live blocks)
        SwapMove(w[4b+0], w[4b+1], 0x00ff00ff, 8)
        SwapMove(w[4b+2], w[4b+3], 0x00ff00ff, 8)
        SwapMove(w[4b+0], w[4b+2], 0x0000ffff, 16)
        SwapMove(w[4b+1], w[4b+3], 0x0000ffff, 16)
    for r in 0..3:                                // surviving bit-stage swaps
        SwapMove(w[r],    w[16+r], 0x0f0f0f0f, 4)       // pair {0,4}
        for j in (0, 4):    SwapMove(w[4j+r], w[4(j+2)+r], 0x33333333, 2)
        for j in (0,2,4,6): SwapMove(w[4j+r], w[4(j+1)+r], 0x55555555, 1)
    return w

procedure PartialUnpack04(ref w[0..31]):
    // afterwards blocks 0 and 4 sit in w[0..3] and w[16..19] (le32 words);
    // the remaining wires hold unspecified values
    apply the 36 SwapMoves of PackSparse04 in reverse order
```

Algorithm 3. The sliced key-less AES round $R^{\|}$ (Section 3.2). `SboxCircuit` is any boolean AES S-box circuit (Section 6.1) evaluated with word operations.

```
procedure SlicedRound(ref W[0..3][0..7]):         // W[r][k] = w[4k+r]
    for r in 0..3:                                // SubBytes
        (W[r][0],...,W[r][7]) <- SboxCircuit(W[r][0],...,W[r][7])
    for r in 1..3:                                // ShiftRows
        for k in 0..7: W[r][k] <- rotr32(W[r][k], 8*r)
    for r in 0..3:                                // MixColumns
        for k in 0..7: D[r][k] <- W[r][k] ^ W[(r+1) mod 4][k]
    for r in 0..3:
        for k in 0..7: S[k] <- D[(r+1) mod 4][k] ^ W[(r+3) mod 4][k]
        out[r][0] <- D[r][7] ^ S[0]
        for k in (2,5,6,7): out[r][k] <- D[r][k-1] ^ S[k]
```

```
        for k in (1,3,4):   out[r][k] <- D[r][k-1] ^ D[r][7] ^ S[k]
    W <- out
```

Algorithm 4. AEGIS-128L update and keystream in the persistent policy (Section 4). `Minj` is a sliced injection produced by `PackSparse04`, computed once when the injected value is constant.

```
procedure Rotate(ref W):                          // the rotation rho
    for each word w in W:
        w <- ((w & 0x7f7f7f7f) << 1) | ((w & 0x80808080) >> 7)

procedure Update128L(ref W, Minj):
    T <- copy(W)
    SlicedRound(T)
    Rotate(T)
    for i in 0..31: W[i] <- W[i] ^ T[i] ^ Minj[i]

function Keystream128L(W) -> (Z0, Z1):            // 16-byte blocks
    for each word w in W, into the same position of Z:
        z  <- ((w >> 1) ^ (w >> 6) ^ ((w >> 2) & (w >> 3))) & 0x01010101
        z' <- ((w >> 2) ^ (w >> 5) ^ ((w >> 6) & (w >> 7))) & 0x01010101
        Z[...] <- z | (z' << 4)
    PartialUnpack04(Z)
    Z0 <- bytes of Z[0..3]; Z1 <- bytes of Z[16..19]
    return (Z0, Z1)
```

Algorithm 5. AEGIS-256 differences (Section 5): six-lane rotation, single-lane injection and extraction. Initialization and finalization follow the AEGIS-256 schedule with these primitives.

```
procedure Rotate6(ref W):                         // rho on 6 lanes
    for each word w in W:
        w <- ((w & 0x1f1f1f1f) << 1) | ((w & 0x20202020) >> 5)

procedure Update256(ref W, Minj):                 // Minj = PackSparse0(M)
    T <- copy(W); SlicedRound(T); Rotate6(T)
    for i in 0..31: W[i] <- W[i] ^ T[i] ^ Minj[i]

function Keystream256(W) -> Z:
    for each word w in W, into the same position of Z:
        Z[...] <- ((w>>1) ^ (w>>4) ^ (w>>5) ^ ((w>>2) & (w>>3))) & 0x01010101
    PartialUnpack0(Z); Z <- bytes of Z[0..3]
    return Z
```

Algorithm 6. AEGIS-128L authenticated encryption, persistent policy. C_0, C_1 are the specification constants; `pad32` zero-pads to 32 bytes. Decryption replaces the keystream XOR direction and injects the recovered plaintext.

```
function Encrypt128L(key K, nonce N, ad A, msg P) -> (ct, tag):
    S <- (K^N, C1, C0, C1, K^N, K^C0, K^C1, K^C0)  // byte-form blocks
    for b in 0..7, j in 0..3: W[4b+j] <- le32(S_b, j)
    Pack(W)
    Minit <- PackSparse04(N, K)
    repeat 10 times: Update128L(W, Minit)
    for each 32-byte chunk a of A:                 // last chunk zero-padded
        a <- pad32(a)
        Update128L(W, PackSparse04(a[0..15], a[16..31]))
    for each 32-byte chunk p of P:                 // last chunk zero-padded
        (Z0, Z1) <- Keystream128L(W)
        emit (p ^ (Z0 || Z1)) truncated to |p|
        q <- pad32(p)
        Update128L(W, PackSparse04(q[0..15], q[16..31]))
```

```
U <- copy(W); Unpack(U)  // or a 32-swap partial unpack of block 2 alone
t <- U.block[2] ^ (LE64(8*|A|) || LE64(8*|P|)) // block b = bytes of U[4b..4b+3]
Mfin <- PackSparse04(t, t)
repeat 7 times: Update128L(W, Mfin)
Unpack(W)                                      // the state is no longer needed
tag <- W.block[0] ^ ... ^ W.block[6]           // 128-bit tag
// 256-bit tag: (b0^b1^b2^b3) || (b4^b5^b6^b7)
return (ct, tag)
```

References

[AP21] Adomnicai, A., and Peyrin, T. Fixslicing AES-like Ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. IACR Transactions on Cryptographic Hardware and Embedded Systems 2021(1). pp. 402–425 (2021). Cryptology ePrint Archive, Paper 2020/1123.

[Ber05] Bernstein, D.J. Cache-timing attacks on AES. (2005). Preprint. https://cr.yp.to/papers.html#cachetiming

[Bih97] Biham, E. A Fast New DES Implementation in Software. In: Fast Software Encryption – FSE 1997. pp. 260–271 (1997).

[BP10] Boyar, J., and Peralta, R. A New Combinational Logic Minimization Technique with Applications to Cryptology. In: Experimental Algorithms – SEA 2010. pp. 178–189 (2010).

[BP12] Boyar, J., and Peralta, R. A Small Depth-16 Circuit for the AES S-Box. In: Information Security and Privacy Research – SEC 2012. pp. 287–298 (2012).

[Den23] Denis, F. Adding more parallelism to the AEGIS authenticated encryption algorithms. (2023). Cryptology ePrint Archive, Paper 2023/523.

[DL25] Denis, F., and Lucas, S. The AEGIS Family of Authenticated Encryption Algorithms. (2025). Internet-Draft draft-irtf-cfrg-aegis-aead-18, IRTF Crypto Forum Research Group. Approved for publication, in the RFC Editor queue.

[Jeo+25] Jeon, Y., Baek, S., Kim, G., and Kim, J. A Framework for Generating S-Box Circuits with Boyar–Peralta Algorithm-Based Heuristics, and Its Applications to AES, SNOW3G, and Saturnin. IACR Transactions on Cryptographic Hardware and Embedded Systems 2025(1). pp. 586–631 (2025). Cryptology ePrint Archive, Paper 2024/1996.

[KS09] Käsper, E., and Schwabe, P. Faster and Timing-Attack Resistant AES-GCM. In: Cryptographic Hardware and Embedded Systems – CHES 2009. pp. 1–17 (2009). Cryptology ePrint Archive, Paper 2009/129.

[ME19] Maximov, A., and Ekdahl, P. New Circuit Minimization Techniques for Smaller and Faster AES SBoxes. IACR Transactions on Cryptographic Hardware and Embedded Systems 2019(4). pp. 91–125 (2019). Cryptology ePrint Archive, Paper 2019/802.

[MN07] Matsui, M., and Nakajima, J. On the Power of Bitslice Implementation on Intel Core2 Processor. In: Cryptographic Hardware and Embedded Systems – CHES 2007. pp. 121–134 (2007).

[Nat23] National Institute of Standards and Technology. Advanced Encryption Standard (AES). (2023). FIPS Publication 197-upd1.

[OST06] Osvik, D.A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: The Case of AES. In: Topics in Cryptology – CT-RSA 2006. pp. 1–20 (2006).

[SS17] Schwabe, P., and Stoffelen, K. All the AES You Need on Cortex-M3 and M4. In: Selected Areas in Cryptography – SAC 2016. pp. 180–194 (2017). Cryptology ePrint Archive, Paper 2016/714.

[WP14] Wu, H., and Preneel, B. AEGIS: A Fast Authenticated Encryption Algorithm. In: Selected Areas in Cryptography – SAC 2013. pp. 185–201 (2014). AEGIS-128L is specified in the full version, Cryptology ePrint Archive, Paper 2013/695, and in the CAESAR submission v1.1.