Adding more parallelism to the AEGIS authenticated encryption algorithms

Frank Denis

Fastly Inc.
fde@00f.net

Abstract. While the round function of the AEGIS authenticated encryption algorithms is highly parallelizable, their mode of operation is not.

We introduce two new modes to overcome that limitation: AEGIS-128X and AEGIS-256X. These modes require minimal changes to existing implementations and retain the security properties of AEGIS-128L and AEGIS-256.

1 Introduction

AEGIS [WP14] is a family of three authenticated encryption algorithms, originally designed to use the AES-NI instruction set introduced by Intel in 2010 [ADF+10]. These instructions perform several compute-intensive parts of the AES algorithm, significantly improving the performance of software AES implementations while minimizing the risks of side-channel attacks.

However, concurrent AES round instructions are required to fully use the AES pipelines. The AEGIS round function was designed for this purpose and allows multiple AES blocks to be updated concurrently. Its design made it one of the fastest candidates of the CAESAR competition on Intel CPUs with hardware AES acceleration [ARAR16]. Comprehensive benchmarks by Ankele and Ankele [AA16] also placed AEGIS among the highest-performing CAESAR candidates on platforms with AES-NI.

The mode of operation, however, is similar to a duplex: after initialization, the state is updated recursively. This effectively limits the parallelism of the construction to that of the round function. Extending parallelism to the mode of operation itself could therefore improve performance further.

In [BLT15], Bogdanov, Lauridsen, and Tischhauser made a similar observation about several candidates of the CAESAR competition. They proposed a new "comb scheduler" to process multiple messages simultaneously.

Similarly, Keyak [BDP+16], another CAESAR competition candidate, supported parallelism in its "Motorist" mode through multiple independent lanes. It achieved domain separation by encoding the lane index into the state.

The modes presented here also encrypt multiple messages simultaneously using the same cipher. We assume that these are fragments of the same message and share the same key, initialization vector and length.

Given a parallelism degree ν, an input message is split into ν evenly distributed parts that can be encrypted concurrently. The resulting ciphertexts are then combined, along with their authentication tags.

The modes use the existing AEGIS algorithms, with a minor addition to their initialization functions.

The AEAD modes are parameterized by an integer parallelism degree $1 \leq \nu \leq 256$, with X1 denoting the parent algorithm. Our initialization analysis applies throughout this range.

2 Operations, variables and functions

The operations, variables and functions used in this document are defined below.

2.1 Operations

$\|x\|$	: Size of x in bits
$x \oplus y$	: Bitwise exclusive OR
$\mathbb{F}_{128L}(S, m_0, m_1)$	: AEGIS-128L state update function
$\mathbb{F}_{256}(S, m)$	: AEGIS-256 state update function
$x \| y$	: Concatenation of x and y
$Pad(x, \ell)$	: Add the fewest trailing 0 bits needed for a length divisible by ℓ
LE64(n)	: Encode n as a 64-bit little-endian integer
$Enc_{128L}(CTX, K, IV, \overline{A}_i, \overline{P}_i, \|A\|, \|P\|)$	: The AEGIS-128L encryption function with context separation
$Enc_{256}(CTX, K, IV, \overline{A}_i, \overline{P}_i, \|A\|, \|P\|)$	: The AEGIS-256 encryption function with context separation
$Enc_{128X}[\nu](K, IV, A, P)$	: The AEGIS-128X[ν] parallel encryption function
$Enc_{256X}[\nu](K, IV, A, P)$	: The AEGIS-256X[ν] parallel encryption function
$Trunc(x, \ell)$	: Truncate x to the first ℓ bits
`AESENC`(x, k)	: One round of AES encryption of block x with key k

2.2 Variables and constants

A	: Associated data or {} if unspecified
A_i	: 128-bit associated data block
$\hat{A}$	: A, zero-padded to a multiple of $r \cdot \nu$ bits
$\hat{A}_i$	: 128-bit associated data block
$\overline{A}_i$	: $\hat{A}$ fragment ($\frac{\|\hat{A}\|}{\nu}$ bits)
$\overline{A}_{i,j}$	: 128-bit $\hat{A}$ fragment block
C	: Ciphertext
$\hat{C}$	: Ciphertext before truncation to $\|P\|$ bits
$\hat{C}_i$	: A 128-bit ciphertext block
$\overline{C}_i$	: $\hat{C}$ fragment ($\frac{\|\hat{C}\|}{\nu}$ bits)
$\overline{C}_{i,j}$	: 128-bit C fragment block
$const_0$	: First half of the AEGIS constant (128 bits)
$const_1$	: Second half of the AEGIS constant (128 bits)
CTX	: Context separator
K_{128}	: 128-bit encryption key (AEGIS-128, -128L)
K_{256}	: 256-bit encryption key (AEGIS-256)
$K_{256,0}$	: First half of a 256-bit key
$K_{256,1}$	: Second half of a 256-bit key
P	: Plaintext
$\hat{P}$	: P, zero-padded to a multiple of $r \cdot \nu$ bits
$\hat{P}_i$	: 128-bit plaintext block
$\overline{P}_i$	: $\hat{P}$ fragment ($\frac{\|\hat{P}\|}{\nu}$ bits)
$\overline{P}_{i,j}$	: 128-bit $\hat{P}$ fragment block
IV_{128}	: 128-bit initialization vector (AEGIS-128, -128L)
IV_{256}	: 256-bit initialization vector (AEGIS-256)
$IV_{256,0}$	: First half of a 256-bit initialization vector
$IV_{256,1}$	: Second half of a 256-bit initialization vector
ν	: Parallelism degree ($1 \leq \nu \leq 256$)
r	: Absorption rate (128 or 256 bits)
S	: AEGIS state
S_i	: A 128-bit AEGIS state block
T	: Authentication tag for C
$\overline{T}_i$	: Authentication tag for $\overline{C}_i$

3 Context separation

From an application perspective, new AEGIS variants should ideally share the same interface as existing variants. That is, they should accept a single message, optional associated data, a 128- or 256-bit key, and a 128- or 256-bit initialization vector.

AEGIS requires nonce-respecting use. Vaudenay and Vizár [VV18] analyzed the robustness of CAESAR candidates, including AEGIS. They emphasized that nonce-respecting use is essential to the security guarantees of schemes such as AEGIS. Their work highlights that nonce reuse can lead to catastrophic security failures in many AEAD schemes. Reusing the same key and IV to encrypt different parts of a message would violate this requirement.

To avoid the universal forgery and decryption attacks identified by Vaudenay and Vizár, we extend the AEGIS initialization functions with a context to provide domain separation. This is essential for each parallel fragment to effectively operate as a separate cryptographic instance, avoiding the security issues of direct key and IV reuse. That is, for two different values $CTX_0 \neq CTX_1$, $Enc.[\cdot](CTX_0, \cdot)$ and $Enc.[\cdot](CTX_1, \cdot)$ act as two different functions of $\{K, IV, A, P\}$.

This approach is conceptually similar to the domain separation used in Keyak's "river" (parallel) mode [BDP+16]. There, the lane index is encoded into the state so that parallel instances process data independently despite sharing the same key. Keyak's design explicitly uses the encoded lane index to separate keystreams, much like our context byte.

A context consists of two bytes: one that separates parallel instances, and one that encodes the parallelism degree.

The context acts as a mask applied to specific words of the AEGIS states during initialization.

3.1 Augmenting AEGIS-128L for context separation

AEGIS-128L defines the initial state S as a vector of eight AES blocks $\{S_0, S_1, \ldots S_7\}$ set to:

Block	Initial value
S_0	$K_{128} \oplus IV_{128}$
S_1	$const_1$
S_2	$const_0$
S_3	$const_1$
S_4	$K_{128} \oplus IV_{128}$
S_5	$K_{128} \oplus const_0$
S_6	$K_{128} \oplus const_1$
S_7	$K_{128} \oplus const_0$

From this state, the original AEGIS-128L initialization function performs 10 updates as described in Algorithm 1.

Algorithm 1 Contextless AEGIS-128L initialization

```
function Initialize(K, IV)
    S ← {K_128 ⊕ IV_128, const_1, const_0, const_1, K_128 ⊕ IV_128, K_128 ⊕ const_0, K_128 ⊕ const_1, K_128 ⊕ const_0}
    i ← 0
    while i < 10 do
        S ← F_128L(S, IV_128, K_128)
        i ← i + 1
    end while
end function
```

We extend this function to accept a context parameter CTX. Before each update, the context is XORed into state blocks 3 and 7, as described in Algorithm 2.

When $CTX = 0^*$, the resulting state is exactly the same as in the original AEGIS-128L algorithm, without a context.

Algorithm 2 AEGIS-128L initialization with context

function INITIALIZE(CTX, K, IV)
  $S \leftarrow \{K_{128} \oplus IV_{128}, const_1, const_0, const_1, K_{128} \oplus IV_{128}, K_{128} \oplus const_0, K_{128} \oplus const_1, K_{128} \oplus const_0\}$
  $i \leftarrow 0$
  while $i < 10$ **do**
    $S_3 \leftarrow S_3 \oplus Pad(CTX, 128)$
    $S_7 \leftarrow S_7 \oplus Pad(CTX, 128)$
    $S \leftarrow \mathbb{F}_{128L}(S, IV_{128}, K_{128})$
    $i \leftarrow i + 1$
  end while
end function

3.2 Augmenting AEGIS-256 for context separation

AEGIS-256 accepts a 256-bit key K_{256} made of two AES blocks $\{K_{256,0}, K_{256,1}\}$. It also accepts a 256-bit initialization vector IV_{256} made of two AES blocks $\{IV_{256,0}, IV_{256,1}\}$.

The initial state S is a vector of six AES blocks $\{S_0, S_1, \dots S_5\}$ set to:

Block	Initial value
S_0	$K_{256,0} \oplus IV_{256,0}$
S_1	$K_{256,1} \oplus IV_{256,1}$
S_2	$const_1$
S_3	$const_0$
S_4	$K_{256,0} \oplus const_0$
S_5	$K_{256,1} \oplus const_1$

From this state, the original AEGIS-256 initialization function performs 16 updates as described in Algorithm 3.

Algorithm 3 Contextless AEGIS-256 initialization

function INITIALIZE(K, IV)
  $S \leftarrow \{K_{256,0} \oplus IV_{256,0}, K_{256,1} \oplus IV_{256,1}, const_1, const_0, K_{256,0} \oplus const_0, K_{256,1} \oplus const_1\}$
  $i \leftarrow 0$
  while $i < 4$ **do**
    $S \leftarrow \mathbb{F}_{256}(S, K_{256,0})$
    $S \leftarrow \mathbb{F}_{256}(S, K_{256,1})$
    $S \leftarrow \mathbb{F}_{256}(S, K_{256,0} \oplus IV_{256,0})$
    $S \leftarrow \mathbb{F}_{256}(S, K_{256,1} \oplus IV_{256,1})$
    $i \leftarrow i + 1$
  end while
end function

We extend this function to accept a context parameter CTX. Before each update, the context is XORed into state blocks 3 and 5, as described in Algorithm 4.

When $CTX = 0^*$, the resulting state is exactly the same as in the original AEGIS-256 algorithm, without a context.

3.3 Explicit lengths for finalization

The finalization functions of AEGIS-128L and AEGIS-256 absorb the lengths of the associated data and message before performing seven state updates (Algorithms 5 and 6).

The Enc_{128L} and Enc_{256} functions defined here take the total lengths $|A|$ and $|P|$ as inputs.

This is necessary because individual encryption functions process distinct fragments of A and P simultaneously. During finalization in AEGIS-128X[ν] and AEGIS-256X[ν], however, all instances use the same total lengths $|A|$ and $|P|$ instead of their individual fragment lengths.

Algorithm 4 AEGIS-256 initialization with context

function INITIALIZE(CTX, K, IV)
$S \leftarrow \{K_{256,0} \oplus IV_{256,0}, K_{256,1} \oplus IV_{256,1}, const_1, const_0, K_{256,0} \oplus const_0, K_{256,1} \oplus const_1\}$
$i \leftarrow 0$
while $i < 4$ **do**
$S_3 \leftarrow S_3 \oplus Pad(CTX, 128)$
$S_5 \leftarrow S_5 \oplus Pad(CTX, 128)$
$S \leftarrow \mathbb{F}_{256}(S, K_{256,0})$
$S_3 \leftarrow S_3 \oplus Pad(CTX, 128)$
$S_5 \leftarrow S_5 \oplus Pad(CTX, 128)$
$S \leftarrow \mathbb{F}_{256}(S, K_{256,1})$
$S_3 \leftarrow S_3 \oplus Pad(CTX, 128)$
$S_5 \leftarrow S_5 \oplus Pad(CTX, 128)$
$S \leftarrow \mathbb{F}_{256}(S, K_{256,0} \oplus IV_{256,0})$
$S_3 \leftarrow S_3 \oplus Pad(CTX, 128)$
$S_5 \leftarrow S_5 \oplus Pad(CTX, 128)$
$S \leftarrow \mathbb{F}_{256}(S, K_{256,1} \oplus IV_{256,1})$
$i \leftarrow i + 1$
end while
end function

Algorithm 5 The AEGIS-128L finalization function

function FINALIZE($|A|$, $|P|$, $|T|$)
$U \leftarrow S_2 \oplus (\text{LE64}(|A|) \| \text{LE64}(|P|))$
$i \leftarrow 0$
while $i < 7$ **do**
$S \leftarrow \mathbb{F}_{128L}(S, U, U)$
$i \leftarrow i + 1$
end while
if $|T| = 256$ **then**
return $(S_0 \oplus S_1 \oplus S_2 \oplus S_3) \| (S_4 \oplus S_5 \oplus S_6 \oplus S_7)$
else
return $S_0 \oplus S_1 \oplus S_2 \oplus S_3 \oplus S_4 \oplus S_5 \oplus S_6$
end if
end function

Algorithm 6 The AEGIS-256 finalization function

function FINALIZE($|A|$, $|P|$, $|T|$)
$U \leftarrow S_3 \oplus (\text{LE64}(|A|) \| \text{LE64}(|P|))$
$i \leftarrow 0$
while $i < 7$ **do**
$S \leftarrow \mathbb{F}_{256}(S, U)$
$i \leftarrow i + 1$
end while
if $|T| = 256$ **then**
return $(S_0 \oplus S_1 \oplus S_2) \| (S_3 \oplus S_4 \oplus S_5)$
else
return $S_0 \oplus S_1 \oplus S_2 \oplus S_3 \oplus S_4 \oplus S_5$
end if
end function

4 The AEGIS-128X[ν] and AEGIS-256X[ν] modes

AEGIS absorbs the associated data and message at a rate r, where $r = 256$ for AEGIS-128L and $r = 128$ for AEGIS-128 and AEGIS-256.

We define two new modes: AEGIS-128X[ν] and AEGIS-256X[ν]. They absorb $r \cdot \nu$ bits per state update across ν concurrent instances of the AEGIS-128L or AEGIS-256 encryption functions.

The associated data A and plaintext P are split into interleaved blocks with a stride of $r \cdot \nu$ bits as they arrive. The last blocks are padded as necessary.

We first pad A and P with trailing zero bits until their lengths are multiples of the stride length:

$$\hat{A} = Pad(A, r \cdot \nu)$$
$$\hat{P} = Pad(P, r \cdot \nu)$$

$\hat{A}$ is split into 128-bit blocks $\{\hat{A}_0, \hat{A}_1, \dots \hat{A}_{\frac{|\hat{A}|}{128}-1}\}$.

Interleaving these blocks produces ν independent messages of $\frac{|\hat{A}|}{\nu}$ bits each: $\{\overline{A}_0, \overline{A}_1, \dots \overline{A}_{\nu-1}\}$.

$$\overline{A}_0 = \hat{A}_0 \| \hat{A}_\nu \| \hat{A}_{2\nu} \| \hat{A}_{3\nu} \| \dots$$
$$\overline{A}_1 = \hat{A}_1 \| \hat{A}_{\nu+1} \| \hat{A}_{2\nu+1} \| \hat{A}_{3\nu+1} \| \dots$$
$$\overline{A}_2 = \hat{A}_2 \| \hat{A}_{\nu+2} \| \hat{A}_{2\nu+2} \| \hat{A}_{3\nu+2} \| \dots$$
$$\vdots$$
$$\overline{A}_{\nu-1} = \hat{A}_{\nu-1} \| \hat{A}_{\nu+(\nu-1)} \| \hat{A}_{2\nu+(\nu-1)} \| \hat{A}_{3\nu+(\nu-1)} \| \dots$$

Similarly, $\hat{P}$ is split into ν independent messages of $\frac{|\hat{P}|}{\nu}$ bits each: $\{\overline{P}_0, \overline{P}_1, \dots \overline{P}_{\nu-1}\}$.

4.1 AEGIS-128X

AEGIS-128X[ν] first encrypts and authenticates the plaintext and associated data fragments independently. This produces ν ciphertexts $\{\overline{C}_0, \overline{C}_1, \dots \overline{C}_{\nu-1}\}$ and authentication tags $\{\overline{T}_0, \overline{T}_1, \dots \overline{T}_{\nu-1}\}$.

$$\{\overline{C}_0, \overline{T}_0\} = Enc_{128L}(CTX \leftarrow 0 \| (\nu-1), K, IV, \overline{A}_0, \overline{P}_0, |A|, |P|)$$
$$\{\overline{C}_1, \overline{T}_1\} = Enc_{128L}(CTX \leftarrow 1 \| (\nu-1), K, IV, \overline{A}_1, \overline{P}_1, |A|, |P|)$$
$$\{\overline{C}_2, \overline{T}_2\} = Enc_{128L}(CTX \leftarrow 2 \| (\nu-1), K, IV, \overline{A}_2, \overline{P}_2, |A|, |P|)$$
$$\vdots$$
$$\{\overline{C}_{\nu-1}, \overline{T}_{\nu-1}\} = Enc_{128L}(CTX \leftarrow \nu-1 \| (\nu-1), K, IV, \overline{A}_{\nu-1}, \overline{P}_{\nu-1}, |A|, |P|)$$

The fragments $\{\overline{C}_0, \overline{C}_1, \dots \overline{C}_{\nu-1}\}$ are deinterleaved to produce the final ciphertext:

$$\hat{C} = \overline{C}_{0,0} \| \overline{C}_{1,0} \| \overline{C}_{2,0} \| \dots \| \overline{C}_{(\nu-1),0} \| \overline{C}_{0,1} \| \overline{C}_{1,1} \| \overline{C}_{2,1} \| \dots \| \overline{C}_{(\nu-1),1} \| \overline{C}_{0,2} \| \overline{C}_{1,2} \| \overline{C}_{2,2} \| \dots \| \overline{C}_{(\nu-1),2} \| \dots$$

$$C = Trunc(\hat{C}, |P|)$$

Finally, the AEGIS-128X[ν] authentication tag is the bitwise exclusive OR of the AEGIS-128L authentication tags:

$$T = \overline{T}_0 \oplus \overline{T}_1 \oplus \dots \oplus \overline{T}_{\nu-1}$$

When $\nu = 1$, the context has no bits set. AEGIS-128L and AEGIS-128X[1] are therefore equivalent.

4.2 AEGIS-256X

AEGIS-256X[ν] uses the same interleaving technique as AEGIS-128X[ν] to process $r \cdot \nu$ bits per state update.

The only difference is that fragments are encrypted and authenticated using the AEGIS-256 encryption function instead of the AEGIS-128L one.

$$\{\overline{C}_0, \overline{T}_0\} = Enc_{256}(CTX \leftarrow 0\|(\nu-1), K, IV, \overline{A}_0, \overline{P}_0, |A|, |P|)$$
$$\{\overline{C}_1, \overline{T}_1\} = Enc_{256}(CTX \leftarrow 1\|(\nu-1), K, IV, \overline{A}_1, \overline{P}_1, |A|, |P|)$$
$$\{\overline{C}_2, \overline{T}_2\} = Enc_{256}(CTX \leftarrow 2\|(\nu-1), K, IV, \overline{A}_2, \overline{P}_2, |A|, |P|)$$
$$\vdots$$
$$\{\overline{C}_{\nu-1}, \overline{T}_{\nu-1}\} = Enc_{256}(CTX \leftarrow \nu-1\|(\nu-1), K, IV, \overline{A}_{\nu-1}, \overline{P}_{\nu-1}, |A|, |P|)$$

The fragments $\{\overline{C}_0, \overline{C}_1, \ldots \overline{C}_{\nu-1}\}$ are deinterleaved to produce the AEGIS-256X[ν] ciphertext:

$$\hat{C} = \overline{C}_{0,0}\|\overline{C}_{1,0}\|\overline{C}_{2,0}\|\ldots\|\overline{C}_{(\nu-1),0}\| \overline{C}_{0,1}\|\overline{C}_{1,1}\|\overline{C}_{2,1}\|\ldots\|\overline{C}_{(\nu-1),1}\| \overline{C}_{0,2}\|\overline{C}_{1,2}\|\overline{C}_{2,2}\|\ldots\|\overline{C}_{(\nu-1),2}\|\ldots$$

$$C = Trunc(\hat{C}, |P|)$$

Finally, the AEGIS-256X[ν] authentication tag is the bitwise exclusive OR of the AEGIS-256 authentication tags:

$$T = \overline{T}_0 \oplus \overline{T}_1 \oplus \ldots \oplus \overline{T}_{\nu-1}$$

AEGIS-256 and AEGIS-256X[1] are equivalent.

5 Rationale

5.1 Security requirements

The AEGIS security claims have the following requirements:

- Each key should be generated uniformly at random.
- Each key and IV pair should protect at most one message and should not be used with two different tag sizes.
- The associated data and plaintext must each contain at most $2^{61} - 1$ bytes, and the tag must contain 128 or 256 bits, as specified in [DL25].
- If verification fails, the decrypted plaintext and the wrong authentication tag should not be returned.

AEGIS-128X[ν] and AEGIS-256X[ν] retain these requirements.

5.2 Design choices

Context separation. Each lane receives a unique context, distinguishing it from the other lanes that use the same key and IV. Sections 6 and 7 examine how these context differences spread during initialization and give lower bounds on S-box activity.

Deterministic processing order. Blocks follow a fixed interleaving order that matches common SIMD layouts. This avoids extra reordering or per-fragment identifiers.

Finalization strategy. Every lane uses the total input lengths, $|A|$ and $|P|$, in finalization. The padded fragments and these lengths uniquely represent the original inputs. A change to either length changes every lane's finalization length fields.

Fixed parallelism degree. The parallelism degree ν is fixed for each variant. This simplifies implementation and analysis and gives predictable processing costs. Applications can choose a variant suited to their hardware, including for short inputs.

5.3 Security equivalence

We claim the same security levels and usage limits as the parent algorithms, as specified in [DL25]. This claim assumes that the complete initialization provides computational domain separation between lanes.

A random tag guess succeeds with probability $2^{-|T|}$ per attempt.

5.4 Performance-security trade-offs

Each lane retains its parent's state update function and initialization length. The benchmarks in Section 11 show how the parallelism degree affects performance.

We did not include a parallel variant of AEGIS-128, which trades performance for a smaller state. Since performance is the main motivation for AEGIS-128X, AEGIS-128L seems a more natural choice.

6 Implications of the AEGIS-128X context separator

The context is a small, public tag XORed into words S_3 and S_7 before each initialization update. We write the AES round function as $\texttt{AESENC}(x, k) = R(x) \oplus k$, where R consists of SubBytes, ShiftRows, and MixColumns.

All bounds in this section and Section 7 apply to any two distinct lanes at the same degree $2 \leq \nu \leq 256$, for every key and IV. Their contexts differ in exactly one byte.

6.1 How the difference fills one word

SubBytes applies the AES S-box to each of the 16 bytes. For two AES computations with equal round keys, a single differing byte follows this pattern over two rounds:

$$1\,\text{byte} \longrightarrow 4\,\text{bytes} \longrightarrow 16\,\text{bytes}.$$

The first full state update computes $S_3' = R(S_2) \oplus S_3$ and $S_7' = R(S_6) \oplus S_7$ from the state after context injection. Since S_2 and S_6 are equal across lanes, the output differences are $\Delta S_3' = \Delta S_7' = \Delta CTX$.

The next context injection, before the second update, cancels these two differences. Differences in S_0 and S_4 remain. The activity bounds below account for this cancellation.

6.2 S-box activation

Table 1 gives conservative lower bounds on active S-boxes, including per-word byte counts. Activity is counted after context injection, just before SubBytes. Measured activity with random keys is much higher (Table 3).

Table 1. Conservative lower bounds on S-box activity for distinct lanes.

Updates	Byte lower bounds in $(S_0, \ldots, S_7)$	S-box lower bound
0	$(0, 0, 0, 1, 0, 0, 0, 1)$	2
1	$(4, 0, 0, 0, 4, 0, 0, 0)$	8
2	$(4, 16, 0, 1, 4, 16, 0, 1)$	42
3	$(0, 0, 4, 0, 0, 0, 4, 0)$	8
4–6	—	10 in total
7–9	—	9 in total
Total over 10 updates		79

Proposition 1 (S-box activity bound for AEGIS-128X). *AEGIS-128X initialization activates at least 79 S-boxes over ten updates.*

The bounds for updates 0–3, 4–6, and 7–9 sum to $60 + 10 + 9 = 79$. Appendix B gives the exact initialization model and the conservative constraints used to prove these bounds.

7 Implications of the AEGIS-256X context separator

AEGIS-256X uses six state words, $S_0, \ldots, S_5$. The context is XORed into S_3 and S_5 before each of the sixteen initialization updates.

7.1 S-box activation

Table 2 gives the corresponding conservative active-byte lower bounds. Each update executes 96 S-boxes (6 words × 16 per AES round).

Table 2. Conservative lower bounds on S-box activity for distinct lanes in AEGIS-256X.

Updates	Byte lower bounds in $(S_0, \ldots, S_5)$	S-box lower bound
0	$(0, 0, 0, 1, 0, 1)$	2
1	$(4, 0, 0, 0, 4, 0)$	8
2	$(4, 16, 0, 1, 4, 15)$	40
3	$(3, 0, 4, 0, 0, 0)$	7
4–6	—	10 in total
7–9	—	9 in total
10–12	—	9 in total
13–15	—	9 in total
Total over 16 updates		94

Proposition 2 (S-box activity bound for AEGIS-256X). *AEGIS-256X initialization activates at least 94 S-boxes over sixteen updates.*

With updates 0–3 grouped together, the bounds in Table 2 sum to $57 + 10 + 9 + 9 + 9 = 94$ active S-boxes. Appendix B gives the shared derivation.

8 Activity with uniformly random keys

AEGIS requires keys to be sampled uniformly [DL25]. Expected S-box activity is the sum of the probabilities that each input byte differs. This does not require independent byte differences.

If later byte differences under a uniformly sampled key are modeled as uniform over all 256 byte values, the expected activity per update is

$$128 \cdot \frac{255}{256} = 127.5 \quad \text{or} \quad 96 \cdot \frac{255}{256} = 95.625$$

for AEGIS-128X and AEGIS-256X, respectively. This model predicts nearly full activity after diffusion, although uniform key sampling alone does not establish uniform byte differences.

We tested each X2 and X4 variant with 10,000,000 uniformly random keys and a zero IV, then with another 10,000,000 uniformly random keys and random IVs. For each distinct lane pair, we counted active S-boxes throughout initialization. Each family contributed 140,000,000 comparisons, combining X2 and X4 (Table 3). Every final-state word differed in all sampled pairs.

Table 3. Universal lower bounds and measured activity with random keys.

Family	Universal lower bound	Observed mean	Observed range
AEGIS-128X	79	864.7	825–868
AEGIS-256X	94	1253.6	1220–1259

Tracking the active byte positions through the first five updates gives total activity upper bounds of

$$2 + 8 + 42 + 72 + 104 + 5 \cdot 128 = 868$$

for AEGIS-128X and

$$2 + 8 + 41 + 68 + 84 + 11 \cdot 96 = 1259$$

for AEGIS-256X. The observed totals are close to these upper bounds, consistent with the model's prediction of high activity.

The bounds of 79 and 94 are therefore very conservative.

9 Parallel AEGIS as a deterministic MAC

AEGIS-X uses the nonlinear FinalizeMac algorithm in MAC mode. This prevents the linear recombination attack described below.

9.1 An incorrect MAC construction

A naive but incorrect approach would reuse the AEAD mode's XOR tag combiner with a fixed nonce. This is not the MAC construction used by AEGIS-X. It would output:

$\tau = \bigoplus_{i=0}^{\nu-1} \tau_i$ where each τ_i is the ordinary AEGIS tag of lane i.

For this naive construction, three queries under the same nonce suffice to forge a tag, provided the substituted fragments preserve the original lengths: $|Z| = |X|$ and $|T| = |Y|$. All queries then use the same finalization length fields. Here, $\|$ denotes interleaving the lane inputs:

$\tau(Z \parallel T) = \tau(X \parallel Y) \oplus \tau(Z \parallel Y) \oplus \tau(X \parallel T)$

9.2 The FinalizeMac algorithm

For $t \in \{128, 256\}$ and the specified degrees $\nu \in \{2, 4\}$, [DL25] specifies the following tag computation:

Algorithm 7 FinalizeMac for Parallel AEGIS

1. Compute the per-lane tags $\tau_0, \ldots, \tau_{\nu-1}$ using the total data length in bits and t as the two finalization length fields.
2. Absorb $\tau_1 \| \ldots \| \tau_{\nu-1}$ into lane 0's state after step 1, at the parent algorithm's absorption rate. For AEGIS-128X with $t = 128$, prepend τ_0.
3. Apply the parent finalization to lane 0 with length fields ν and t, and output the resulting t-bit tag.

We use the MAC security claims and limits of [DL25].

Nonce reuse resilience. The MAC construction permits a fixed public nonce [DL25]. Nonce reuse remains prohibited for AEAD encryption.

10 Implementation notes

Implementing AEGIS-128X[ν] and AEGIS-256X[ν] requires only small changes to existing AEGIS-128L and AEGIS-256 implementations.

They apply the same operations as AEGIS-128L and AEGIS-256 to vectors of ν AES blocks instead of single blocks.

For example, with 256-bit registers, two AEGIS-128L states S and S' can be stored as:

$\{S_0, S'_0\}, \{S_1, S'_1\}, \{S_2, S'_2\}, \ldots \{S_7, S'_7\}$

This exactly matches the AEGIS-128X[2] interleaved representation. In addition to the forward AES permutation, AEGIS only requires the bitwise XOR and AND operations. Even with wide registers, such operations are very efficient on any CPU with vector instructions.

On CPUs that don't implement vectorized versions of the AES permutation, AEGIS-128X[ν] and AEGIS-256X[ν] can be implemented in different ways:

- Emulate AES vector instructions. This is the easiest option and keeps the code close to hardware-accelerated versions.
- Evaluate $\{\overline{A}_0, \overline{A}_1, \overline{A}_2, \dots \overline{A}_{\nu-1}\}$ and $\{\overline{C}_0, \overline{C}_1, \overline{C}_2, \dots \overline{C}_{\nu-1}\}$ sequentially, with periodic synchronization, for example after every memory page. This reduces cache locality, but also reduces register pressure.

11 Performance evaluation

Previous benchmarks [AA16] have shown AEGIS's strong performance on platforms with AES-NI hardware acceleration, while also highlighting opportunities for improvement. Although AEGIS has some internal round-level parallelism, it does not make full use of AES-NI. Parallelizing the mode of operation is therefore a promising way to substantially improve performance on SIMD architectures.

We implemented AEGIS-128X[ν] and AEGIS-256X[ν] in Zig. The code is available in the reference implementations of the IETF AEGIS specification [DL25].

On Intel, AMD and ARM CPUs with vector registers but without AES vector instructions, AEGIS-128L and AEGIS-256 perform as well as or better than their parallel counterparts.

On CPUs without vector registers, AEGIS-128L and AEGIS-256 are consistently faster than their parallel counterparts.

On CPUs with the VAES instruction set, however, AEGIS-128X[2] is almost twice as fast as AEGIS-128L with medium to large inputs. Examples include an Intel Raptor Lake processor (Table 4) and an AMD EPYC CPU (Table 5, Figure 1).

With short inputs, the overhead of the parallel versions is small (Figure 2). AEGIS-128X[2] is faster than AEGIS-128L for inputs of 256 bytes or more, and AEGIS-256X[2] is faster than AEGIS-256 for inputs of 128 bytes or more.

Algorithm	Throughput
AEGIS-128X[2]	318 Gbps
AEGIS-128L	190 Gbps
AES128-GCM	81 Gbps

Table 4. Intel Core i9-13900K benchmark results (16 KB messages, single core)

Algorithm	Throughput
AEGIS-128X[2]	239 Gbps
AEGIS-128L	118 Gbps
AES128-GCM	37 Gbps

Table 5. AMD EPYC 7543 benchmark results (16 KB messages, single core)

We benchmarked the AES128-GCM implementation from OpenSSL 3.1.0. For AEGIS, we used the specification's reference Zig code and our modified version supporting the 128X and 256X variants.

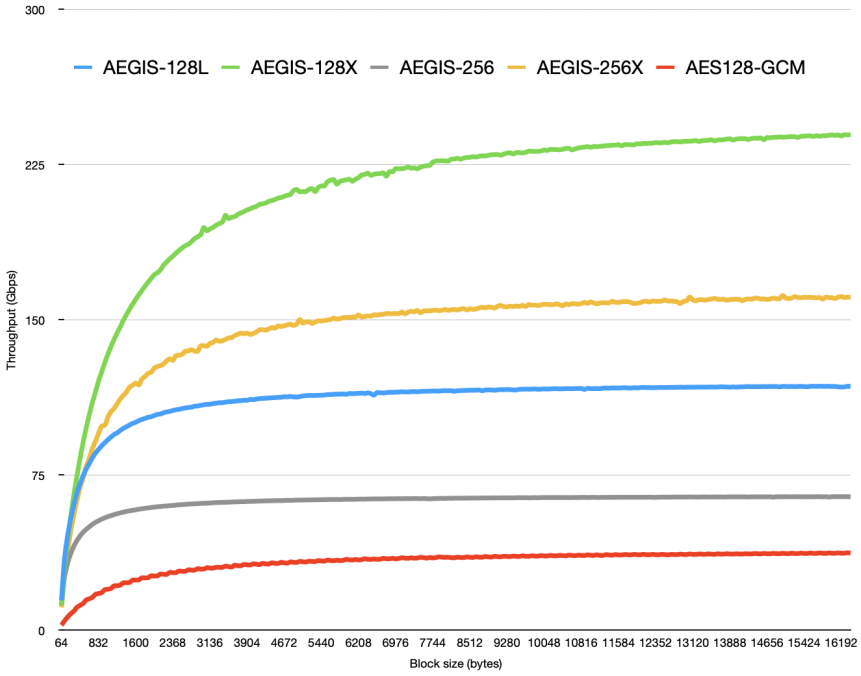

Fig. 1. AMD EPYC 7543 benchmark (single core)

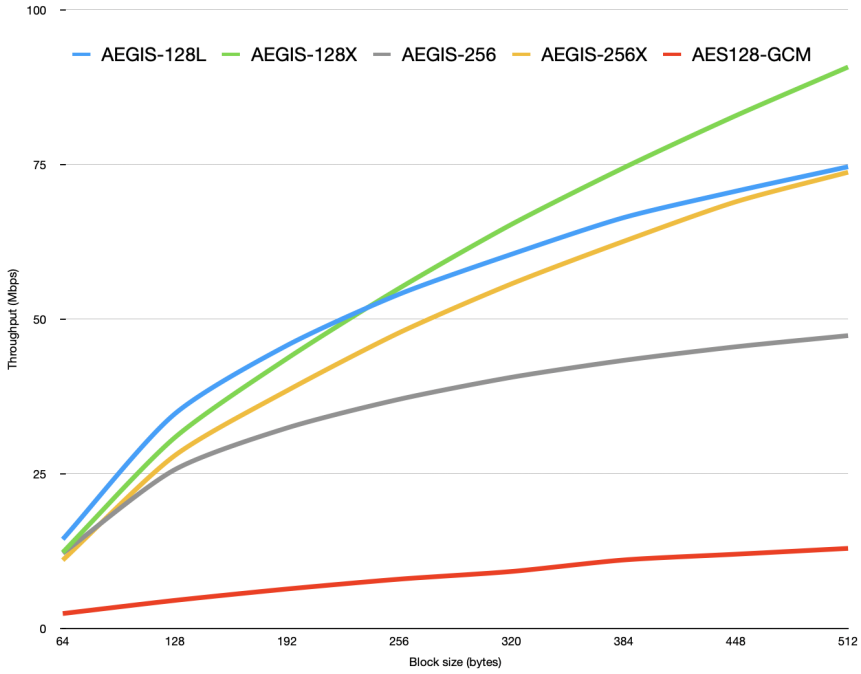

Fig. 2. AMD EPYC 7543 benchmark (single core, short inputs)

The AMD Zen4 family of CPUs supports the AVX-512 instruction set and 4-way AES instructions.

OpenSSL 3.1.1 includes an AES-GCM implementation specifically optimized for these CPUs, so we also benchmarked AEGIS on an AMD Ryzen 7 7700 CPU. Tables 6 and 7 summarize the results.

Algorithm	Throughput
AEGIS-128X[4]	348 Gbps
AEGIS-128X[2]	310 Gbps
AEGIS-128L	152 Gbps
AES128-GCM	94 Gbps

Table 6. AMD Ryzen 7 7700 benchmark results – 128-bit security (16 KB messages, single core)

Algorithm	Throughput
AEGIS-256X[4]	277 Gbps
AEGIS-256X[2]	184 Gbps
AEGIS-256	93 Gbps
AES256-GCM	81 Gbps

Table 7. AMD Ryzen 7 7700 benchmark results – 256-bit security (16 KB messages, single core)

Even generic, non-parallel variants of AEGIS outperform the fastest known implementations of AES-GCM on that hardware. The 2-way variants of AEGIS double the throughput.

The performance gains of the 4-way variants over the 2-way variants appear smaller. Other factors, such as memory bandwidth, may limit performance, especially with AEGIS-128X[4].

Independent measurements reported a larger performance gap between the 2-way and 4-way variants on Intel Xeon CPUs.

12 Conclusion

We propose two additions to the AEGIS family of authenticated encryption algorithms: AEGIS-128X[ν] and AEGIS-256X[ν].

They use new vector instructions to compute multiple instances of AEGIS-128L and AEGIS-256 in parallel. Inputs are split into ν pieces that can be processed independently. This directly matches how CPU instructions load, store and use vector registers.

The resulting modes are very similar to the non-parallel variants and share the same security guarantees. They significantly improve performance on CPUs that support these vector instructions.

References

AA16. R. Ankele and R. Ankele. Software benchmarking of the 2nd round CAESAR candidates. Cryptology ePrint Archive, Report 2016/740, 2016. `https://eprint.iacr.org/2016/740`.

ADF+10. K. D. Akdemir, M. G. Dixon, W. K. Feghali, P. G. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar. Breakthrough AES Performance with Intel ® AES New Instructions. 2010.

ADMA15. E. Andreeva, J. Daemen, B. Mennink, and G. V. Assche. Security of keyed sponge constructions using a modular proof approach. In *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, pages 364–384, 2015.

AJN14. J.-P. Aumasson, P. Jovanovic, and S. Neves. Norx: Parallel and scalable AEAD. In *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, pages 19–36, 2014.

ARAR16. Ankele, Ralph, Ankele, and Robin. Software Benchmarking of the 2nd round CAESAR Candidates. Cryptology ePrint Archive, Report 2016/740, 2016. `https://eprint.iacr.org/2016/740`.

BDP+16. G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. Caesar submission: Keyak v2. *IACR Trans. Symmetric Cryptol.*, 2016(1):29–46, 2016.

BLT15. A. Bogdanov, M. M. Lauridsen, and E. Tischhauser. Comb to Pipeline: Fast Software Encryption Revisited. In *FSE 2015*, *LNCS* 9054, pages 150–171. Springer, Berlin, Heidelberg, March 2015.

Den23. F. Denis. AEGIS-128X and AEGIS-256X implementations. GitHub, 2023. `https://github.com/jedisct1/aegis-128x`.

DL25. F. Denis and S. Lucas. The AEGIS family of authenticated encryption algorithms. Internet-Draft draft-irtf-cfrg-aegis-aead, Internet Engineering Task Force, 2025. Work in Progress.

STSI23. T. Shiraya, N. Takeuchi, K. Sakamoto, and T. Isobe. MILP-based security evaluation for AEGIS/Tiaoxin-346/Rocca. *IET Information Security*, 2023.

VV18. S. Vaudenay and D. Vizár. Can Caesar Beat Galois? - Robustness of CAESAR Candidates Against Nonce Reusing and High Data Complexity Attacks. In *ACNS 18International Conference on Applied Cryptography and Network Security*, *LNCS* 10892, pages 476–494, July 2018.

WP14. H. Wu and B. Preneel. AEGIS: A Fast Authenticated Encryption Algorithm. In *SAC 2013*, *LNCS* 8282, pages 185–201, August 2014.

A Test vectors

The following test vectors for AEGIS-128X[ν] and AEGIS-256X[ν] are in hexadecimal format.

A.1 AEGIS-128X[2]

```
Key:            000102030405060708090a0b0c0d0e0f
IV:             101112131415161718191a1b1c1d1e1f
AD:             (empty)
Plaintext:      (empty)
Ciphertext:     (empty)
128-bit tag:    63117dc57756e402819a82e13eca8379
```

```
Key:            000102030405060708090a0b0c0d0e0f
IV:             101112131415161718191a1b1c1d1e1f
AD:             0102030401020304
Plaintext:      0506070805060708050607 08
Ciphertext:     5696554c009a7e9c63182687
128-bit tag:    151892319d2ba51b59ab47301a03de3a
```

A.2 AEGIS-256X[2]

```
Key:            000102030405060708090a0b0c0d0e0f
                101112131415161718191a1b1c1d1e1f
IV:             101112131415161718191a1b1c1d1e1f
                202122232425262728292a2b2c2d2e2f
AD:             (empty)
Plaintext:      (empty)
Ciphertext:     (empty)
128-bit tag:    62cdbab084c83dacdb945bb446f049c8
```

```
Key:            000102030405060708090a0b0c0d0e0f
                101112131415161718191a1b1c1d1e1f
IV:             101112131415161718191a1b1c1d1e1f
                202122232425262728292a2b2c2d2e2f
AD:             0102030401020304
Plaintext:      0506070805060708050607 08
Ciphertext:     73110d21a920608fd77b580f
128-bit tag:    d2f65e8c45387fb2637d7f3fbbbf2a03
```

A.3 AEGIS-128X[4]

```
Key:            000102030405060708090a0b0c0d0e0f
IV:             101112131415161718191a1b1c1d1e1f
AD:             (empty)
Plaintext:      (empty)
Ciphertext:     (empty)
256-bit tag:    a4b25437f4be93cfa856a2f27e4416b4
                2cac79fd4698f2cdbe6af25673e10a68
```

```
Key:            000102030405060708090a0b0c0d0e0f
IV:             101112131415161718191a1b1c1d1e1f
AD:             0102030401020304
Plaintext:      0506070805060708050607 08
Ciphertext:     e935108a63f746939c36c07c
256-bit tag:    7281a7ca8ff6f6ba8bfb85608db3141f
                f13d3c408b154736bcaa65f436282b92
```

A.4 AEGIS-256X[4]

```
Key:            000102030405060708090a0b0c0d0e0f
                101112131415161718191a1b1c1d1e1f
IV:             101112131415161718191a1b1c1d1e1f
                202122232425262728292a2b2c2d2e2f
AD:             (empty)
Plaintext:      (empty)
Ciphertext:     (empty)
256-bit tag:    6093a1a8aab20ec635dc1ca71745b01b
                5bec4fc444c9ffbebd710d4a34d20eaf
```

```
Key:            000102030405060708090a0b0c0d0e0f
                101112131415161718191a1b1c1d1e1f
IV:             101112131415161718191a1b1c1d1e1f
                202122232425262728292a2b2c2d2e2f
AD:             0102030401020304
Plaintext:      0506070805060708050607 08
Ciphertext:     bec109547f8316d598b3b7d9
256-bit tag:    4adf0672fd2a5068296bde8d5f83049f
                2eed8c5731a64cd69102912ef092d7bf
```

B Derivation of the S-box activity bounds

We derive Propositions 1 and 2 from necessary conditions on two exact initializations with a shared key and IV. The bounds are conservative; their attainability is not established.

B.1 Exact initialization model

Both lanes start from the specified seeds in Sections 3.1 and 3.2, with shared key and IV variables and the actual constants.

Let $S_{r,i}^{(\ell)}$ be word i in lane ℓ before context injection at update r. Set $C_i^{(\ell)}$ to that lane's padded context at the injection positions, and to zero elsewhere. With indices modulo the number of state words, the exact recurrence is

$$X_{r,i}^{(\ell)} = S_{r,i}^{(\ell)} \oplus C_i^{(\ell)}, \qquad S_{r+1,i}^{(\ell)} = R(X_{r,i-1}^{(\ell)}) \oplus X_{r,i}^{(\ell)} \oplus M_{r,i}.$$

Here $M_{r,i}$ is the specified absorbed input, using the seed's key and IV variables, or zero at words with no absorption. The function R consists of the AES SubBytes, ShiftRows, and MixColumns operations. An S-box is active when the corresponding bytes of $X_{r,i}^{(a)}$ and $X_{r,i}^{(b)}$ differ.

B.2 Conservative activity constraints

For the universal lower bounds, it suffices to use the following relaxation of the exact model. It retains necessary conditions on byte differences but omits their actual nonzero values and some dependencies between AES calls.

Let w be the number of state words and J the set of context injection positions: $(w, J) = (8, \{3, 7\})$ for AEGIS-128X and $(6, \{3, 5\})$ for AEGIS-256X. The context difference is $c = (\gamma, 0, \ldots, 0)$ for a nonzero byte γ. Set $C_i = c$ for $i \in J$ and $C_i = 0$ otherwise. The seed and absorbed inputs are equal in both lanes, so the differences obey the recurrence below, with indices taken modulo w:

$$D_{0,i} = 0, \qquad X_{r,i} = D_{r,i} \oplus C_i, \qquad D_{r+1,i} = Y_{r,i-1} \oplus X_{r,i}, \tag{1}$$

where $X_{r,i}$ is the difference before update r and $Y_{r,i}$ is the difference between the two corresponding R outputs.

We represent byte differences by symbols and simplify $x \oplus 0 = x$ and $x \oplus x = 0$. Copied differences keep their symbols; other outputs get fresh symbols. Each symbol has an activity bit: zero when the corresponding bytes are equal and one otherwise. For $x \oplus y = z$, the activity bits a, b, d satisfy

$$a \le b + d, \qquad b \le a + d, \qquad d \le a + b.$$

The three activity bits cannot sum to one. Two nonzero inputs may cancel.

SubBytes preserves activity because the S-box is bijective. Number AES bytes by $4k+h$. ShiftRows then maps input activity to $s_{4k+h} = a_{4((k+h) \bmod 4)+h}$, for $k, h \in \{0, 1, 2, 3\}$. For each MixColumns column, let u and v be its input and output activity counts. Its branch number of five gives the constraints

$$u = 0 \iff v = 0, \qquad u > 0 \implies u + v \ge 5.$$

A column with zero difference has four zero outputs, which we propagate as exact zeros. All other byte values and dependencies between AES calls are unrestricted, so these constraints cover every actual execution.

For an update range I, let N_I count the active bytes in all $X_{r,i}$ with $r \in I$. We expand the recurrence from update zero through the end of each range. For AEGIS-128X, the constraints give

$$N_{0\text{–}3} \ge 60, \qquad N_{4\text{–}6} \ge 10, \qquad N_{7\text{–}9} \ge 9.$$

Since the ranges do not overlap, their lower bounds add to $60 + 10 + 9 = 79$.

For AEGIS-256X, set $(w, J) = (6, \{3, 5\})$ in recurrence (1). The corresponding bounds are $N_{0\text{–}3} \ge 57$, $N_{4\text{–}6} \ge 10$, $N_{7\text{–}9} \ge 9$, $N_{10\text{–}12} \ge 9$, and $N_{13\text{–}15} \ge 9$. Their sum is $57 + 10 + 9 + 9 + 9 = 94$.