

Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 21 August 2025

F. Denis
Fastly Inc.
S. Lucas
Individual Contributor
17 February 2025

The AEGIS Family of Authenticated Encryption Algorithms
draft-irtf-cfrg-aegis-aead-16

Abstract

This document describes the AEGIS-128L, AEGIS-256, AEGIS-128X, and AEGIS-256X AES-based authenticated encryption algorithms designed for high-performance applications.

The document is a product of the Crypto Forum Research Group (CFRG). It is not an IETF product and is not a standard.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at <https://github.com/cfrg/draft-irtf-cfrg-aegis-aead>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 21 August 2025.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction	4
2. Conventions and Definitions	6
3. The AEGIS-128L Algorithm	8
3.1. Authenticated Encryption	9
3.2. Authenticated Decryption	10
3.3. The Update Function	11
3.4. The Init Function	12
3.5. The Absorb Function	13
3.6. The Enc Function	13
3.7. The Dec Function	14
3.8. The DecPartial Function	14
3.9. The Finalize Function	15
4. The AEGIS-256 Algorithm	16
4.1. Authenticated Encryption	16
4.2. Authenticated Decryption	17
4.3. The Update Function	19
4.4. The Init Function	20
4.5. The Absorb Function	21
4.6. The Enc Function	21
4.7. The Dec Function	22
4.8. The DecPartial Function	22
4.9. The Finalize Function	23
5. Parallel Modes	24
5.1. Additional Conventions and Definitions	24
5.2. Authenticated Encryption	25
5.3. Authenticated Decryption	25
5.4. AEGIS-128X	26
5.4.1. The Update Function	26
5.4.2. The Init Function	27
5.4.3. The Absorb Function	28
5.4.4. The Enc Function	28
5.4.5. The Dec Function	29
5.4.6. The DecPartial Function	29
5.4.7. The Finalize Function	30
5.5. AEGIS-256X	31
5.5.1. The Update Function	31
5.5.2. The Init Function	32
5.5.3. The Absorb Function	34
5.5.4. The Enc Function	34
5.5.5. The Dec Function	34

5.5.6.	The DecPartial Function	35
5.5.7.	The Finalize Function	35
5.6.	Implementation Considerations	36
5.7.	Operational Considerations	36
6.	Encoding (ct, tag) Tuples	37
7.	AEGIS as a Stream Cipher	37
8.	AEGIS as a Message Authentication Code	38
8.1.	AEGISMAC-128L	39
8.2.	AEGISMAC-256	39
8.3.	AEGISMAC-128X	40
8.3.1.	The Mac Function	40
8.3.2.	The FinalizeMac Function	40
8.4.	AEGISMAC-256X	41
8.4.1.	The Mac Function	41
8.4.2.	The FinalizeMac Function	42
9.	Implementation Status	43
10.	Security Considerations	44
10.1.	Usage Guidelines	44
10.1.1.	Key and Nonce Selection	44
10.1.2.	Committing Security	44
10.1.3.	Multi-User Security	45
10.2.	Implementation Security	45
10.3.	Security Guarantees	45
11.	IANA Considerations	46
12.	References	47
12.1.	Normative References	47
12.2.	Informative References	48
Appendix A.	Test Vectors	50
A.1.	AESRound Test Vector	50
A.2.	AEGIS-128L Test Vectors	50
A.2.1.	Update Test Vector	50
A.2.2.	Test Vector 1	51
A.2.3.	Test Vector 2	51
A.2.4.	Test Vector 3	52
A.2.5.	Test Vector 4	52
A.2.6.	Test Vector 5	53
A.2.7.	Test Vector 6	53
A.2.8.	Test Vector 7	54
A.2.9.	Test Vector 8	54
A.2.10.	Test Vector 9	55
A.3.	AEGIS-256 Test Vectors	55
A.3.1.	Update Test Vector	55
A.3.2.	Test Vector 1	55
A.3.3.	Test Vector 2	56
A.3.4.	Test Vector 3	56
A.3.5.	Test Vector 4	57
A.3.6.	Test Vector 5	57
A.3.7.	Test Vector 6	58

A.3.8. Test Vector 7	58
A.3.9. Test Vector 8	59
A.3.10. Test Vector 9	59
A.4. AEGIS-128X2 Test Vectors	60
A.4.1. Initial State	60
A.4.2. Test Vector 1	61
A.4.3. Test Vector 2	61
A.5. AEGIS-128X4 Test Vectors	62
A.5.1. Initial State	62
A.5.2. Test Vector 1	63
A.5.3. Test Vector 2	64
A.6. AEGIS-256X2 Test Vectors	64
A.6.1. Initial State	65
A.6.2. Test Vector 1	65
A.6.3. Test Vector 2	66
A.7. AEGIS-256X4 Test Vectors	67
A.7.1. Initial State	67
A.7.2. Test Vector 1	68
A.7.3. Test Vector 2	69
A.8. AEGISMAC Test Vectors	69
A.8.1. AEGISMAC-128L Test Vector	69
A.8.2. AEGISMAC-128X2 Test Vector	70
A.8.3. AEGISMAC-128X4 Test Vector	70
A.8.4. AEGISMAC-256 Test Vector	71
A.8.5. AEGISMAC-256X2 Test Vector	71
A.8.6. AEGISMAC-256X4 Test Vector	72
Acknowledgments	73
Authors' Addresses	73

1. Introduction

This document describes the AEGIS family of Authenticated Encryption with Associated Data (AEAD) algorithms [AEGIS], which were chosen for high-performance applications in the CAESAR (Competition for Authenticated Encryption: Security, Applicability, and Robustness) competition.

Among the finalists, AEGIS-128 was chosen as the winner for this category. However, AEGIS-128L, another finalist, offers enhanced performance and a stronger security margin [ENP20] [JLD22] [LIMS21] [STSI23]. Additionally, AEGIS-256, which also reached the final round, provides 256-bit security and supports higher usage limits.

Therefore, this document specifies the following variants:

- * AEGIS-128L, which has a 128-bit key, a 128-bit nonce, a 1024-bit state, a 128- or 256-bit authentication tag, and processes 256-bit input blocks.

- * AEGIS-256, which has a 256-bit key, a 256-bit nonce, a 768-bit state, a 128- or 256-bit authentication tag, and processes 128-bit input blocks.
- * AEGIS-128X, which is a mode based on AEGIS-128L, specialized for CPUs with large vector registers and vector AES instructions.
- * AEGIS-256X, which is a mode based on AEGIS-256, specialized for CPUs with large vector registers and vector AES instructions.

All variants are inverse-free and constructed from the AES encryption round function [FIPS-AES].

The AEGIS cipher family offers performance that significantly exceeds AES-GCM on CPUs with AES instructions. Similarly, software implementations not using AES instructions can also be faster, although to a lesser extent.

Unlike with AES-GCM, nonces can be safely chosen at random with no practical limit when using AEGIS-256 and AEGIS-256X. AEGIS-128L and AEGIS-128X also allow for more messages to be safely encrypted when using random nonces.

With some existing AEAD schemes, such as AES-GCM, an attacker can generate a ciphertext that successfully decrypts under multiple different keys (a partitioning oracle attack) [LGR21]. This ability to craft a (ciphertext, authentication tag) pair that verifies under multiple keys significantly reduces the number of required interactions with the oracle to perform an exhaustive search, making it practical if the key space is small. For example, with password-based encryption, an attacker can guess a large number of passwords at a time by recursively submitting such a ciphertext to an oracle, which speeds up a password search by reducing it to a binary search.

With AEGIS, finding distinct (key, nonce) pairs that successfully decrypt a given (associated data, ciphertext, authentication tag) tuple is believed to have a complexity that depends on the tag size. A 128-bit tag provides 64-bit committing security, which is generally acceptable for interactive protocols. With a 256-bit tag, finding a collision becomes impractical.

Unlike most other AES-based AEAD constructions, leaking a state does not leak the key or previous states.

Finally, an AEGIS key is not required after the initialization function, and there is no key schedule. Thus, ephemeral keys can be erased from memory before any data has been encrypted or decrypted, mitigating cold boot attacks.

Note that an earlier version of Hongjun Wu and Bart Preneel's paper introducing AEGIS specified AEGIS-128L and AEGIS-256 with a different Finalize function. We follow the specification of [AEGIS], which can be found in the References section of this document.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Throughout this document, "byte" is used interchangeably with "octet" and refers to an 8-bit sequence.

Primitives:

- * {}: an empty bit array.
- * $|x|$: the length of x in bits.
- * $a \wedge b$: the bitwise exclusive OR operation between a and b .
- * $a \& b$: the bitwise AND operation between a and b .
- * $a || b$: the concatenation of a and b .
- * $a \bmod b$: the remainder of the Euclidean division between a as the dividend and b as the divisor.
- * $\text{LE64}(x)$: returns the little-endian encoding of unsigned 64-bit integer x .
- * $\text{ZeroPad}(x, n)$: returns x after appending zeros until its length is a multiple of n bits. No padding is added if the length of x is already a multiple of n , including when x is empty.
- * $\text{Truncate}(x, n)$: returns the first n bits of x .
- * $\text{Split}(x, n)$: returns x split into n -bit blocks, ignoring partial blocks.
- * $\text{Tail}(x, n)$: returns the last n bits of x .

- * AESRound(in, rk): a single round of the AES encryption round function, which is the composition of the SubBytes, ShiftRows, MixColumns, and AddRoundKey transformations, as defined in Section 5 of [FIPS-AES]. Here, in is the 128-bit AES input state, and rk is the 128-bit round key.
- * Repeat(n, F): n sequential evaluations of the function F.
- * CtEq(a, b): compares a and b in constant-time, returning True for an exact match and False otherwise.

AEGIS internal functions:

- * Update(M0, M1) or Update(M): the state update function.
- * Init(key, nonce): the initialization function.
- * Absorb(ai): the input block absorption function.
- * Enc(xi): the input block encryption function.
- * Dec(ci): the input block decryption function.
- * DecPartial(cn): the input block decryption function for the last ciphertext bits when they do not fill an entire block.
- * Finalize(ad_len_bits, msg_len_bits): the authentication tag generation function.

Input blocks are 256 bits for AEGIS-128L and 128 bits for AEGIS-256.

AES blocks:

- * Si: the i-th AES block of the current state.
- * S'i: the i-th AES block of the next state.
- * {Si, ...Sj}: the vector of the i-th AES block of the current state to the j-th block of the current state.
- * C0: an AES block built from the following bytes in hexadecimal format: { 0x00, 0x01, 0x01, 0x02, 0x03, 0x05, 0x08, 0x0d, 0x15, 0x22, 0x37, 0x59, 0x90, 0xe9, 0x79, 0x62 }.
- * C1: an AES block built from the following bytes in hexadecimal format: { 0xdb, 0x3d, 0x18, 0x55, 0x6d, 0xc2, 0x2f, 0xf1, 0x20, 0x11, 0x31, 0x42, 0x73, 0xb5, 0x28, 0xdd }.

AES blocks are always 128 bits in length.

Input and output values:

- * key: the encryption key (128 bits for AEGIS-128L, 256 bits for AEGIS-256).
- * nonce: the public nonce (128 bits for AEGIS-128L, 256 bits for AEGIS-256).
- * ad: the associated data.
- * msg: the plaintext.
- * ct: the ciphertext.
- * tag: the authentication tag (128 or 256 bits).

3. The AEGIS-128L Algorithm

AEGIS-128L has a 1024-bit state, made of eight 128-bit blocks $\{S_0, \dots, S_7\}$.

The parameters for this algorithm, whose meaning is defined in [RFC5116], Section 4, are:

- * K_LEN (key length) is 16 bytes (128 bits).
- * P_MAX (maximum length of the plaintext) is $2^{61} - 1$ bytes ($2^{64} - 8$ bits).
- * A_MAX (maximum length of the associated data) is $2^{61} - 1$ bytes ($2^{64} - 8$ bits).
- * N_MIN (minimum nonce length) = N_MAX (maximum nonce length) = 16 bytes (128 bits).
- * C_MAX (maximum ciphertext length) = P_MAX + tag length = $(2^{61} - 1) + 16$ or 32 bytes (in bits: $(2^{64} - 8) + 128$ or 256 bits).

Distinct associated data inputs, as described in [RFC5116], Section 3, MUST be unambiguously encoded as a single input. It is up to the application to create a structure in the associated data input if needed.

3.1. Authenticated Encryption

Encrypt(msg, ad, key, nonce)

The Encrypt function encrypts a message and returns the ciphertext along with an authentication tag that verifies the authenticity of the message and associated data, if provided.

Security:

- * For a given key, the nonce MUST NOT be reused under any circumstances; doing so allows an attacker to recover the internal state.
- * The key MUST be randomly chosen from a uniform distribution.

Inputs:

- * msg: the message to be encrypted (length MUST be less than or equal to P_MAX).
- * ad: the associated data to authenticate (length MUST be less than or equal to A_MAX).
- * key: the encryption key.
- * nonce: the public nonce.

Outputs:

- * ct: the ciphertext.
- * tag: the authentication tag.

Steps:

```
Init(key, nonce)

ct = {}

ad_blocks = Split(ZeroPad(ad, 256), 256)
for ai in ad_blocks:
    Absorb(ai)

msg_blocks = Split(ZeroPad(msg, 256), 256)
for xi in msg_blocks:
    ct = ct || Enc(xi)

tag = Finalize(|ad|, |msg|)
ct = Truncate(ct, |msg|)

return ct and tag
```

3.2. Authenticated Decryption

```
Decrypt(ct, tag, ad, key, nonce)
```

The Decrypt function decrypts a ciphertext, verifies that the authentication tag is correct, and returns the message on success or an error if tag verification failed.

Security:

- * If tag verification fails, the decrypted message and wrong authentication tag MUST NOT be given as output. The decrypted message MUST be overwritten with zeros before the function returns.
- * The comparison of the input tag with the expected_tag MUST be done in constant time.

Inputs:

- * ct: the ciphertext to decrypt (length MUST be less than or equal to C_MAX).
- * tag: the authentication tag.
- * ad: the associated data to authenticate (length MUST be less than or equal to A_MAX).
- * key: the encryption key.
- * nonce: the public nonce.

Outputs:

- * Either the decrypted message msg or an error indicating that the authentication tag is invalid for the given inputs.

Steps:

Init(key, nonce)

msg = {}

```
ad_blocks = Split(ZeroPad(ad, 256), 256)
for ai in ad_blocks:
    Absorb(ai)
```

```
ct_blocks = Split(ct, 256)
cn = Tail(ct, |ct| mod 256)
```

```
for ci in ct_blocks:
    msg = msg || Dec(ci)
```

```
if cn is not empty:
    msg = msg || DecPartial(cn)
```

```
expected_tag = Finalize(|ad|, |msg|)
```

```
if CtEq(tag, expected_tag) is False:
    erase msg
    erase expected_tag
    return "verification failed" error
else:
    return msg
```

3.3. The Update Function

Update(M0, M1)

The Update function is the core of the AEGIS-128L algorithm. It updates the state {S0, ...S7} using two 128-bit values.

Inputs:

- * M0: the first 128-bit block to be absorbed.
- * M1: the second 128-bit block to be absorbed.

Modifies:

* $\{S_0, \dots S_7\}$: the state.

Steps:

```
S'0 = AESRound(S7, S0 ^ M0)
S'1 = AESRound(S0, S1)
S'2 = AESRound(S1, S2)
S'3 = AESRound(S2, S3)
S'4 = AESRound(S3, S4 ^ M1)
S'5 = AESRound(S4, S5)
S'6 = AESRound(S5, S6)
S'7 = AESRound(S6, S7)
```

```
S0 = S'0
S1 = S'1
S2 = S'2
S3 = S'3
S4 = S'4
S5 = S'5
S6 = S'6
S7 = S'7
```

3.4. The Init Function

Init(key, nonce)

The Init function constructs the initial state $\{S_0, \dots S_7\}$ using the given key and nonce.

Inputs:

* key: the encryption key.
* nonce: the public nonce.

Defines:

* $\{S_0, \dots S_7\}$: the initial state.

Steps:

```
S0 = key ^ nonce
S1 = C1
S2 = C0
S3 = C1
S4 = key ^ nonce
S5 = key ^ C0
S6 = key ^ C1
S7 = key ^ C0
```

```
Repeat(10, Update(nonce, key))
```

3.5. The Absorb Function

```
Absorb(ai)
```

The Absorb function absorbs a 256-bit input block *ai* into the state $\{S0, \dots S7\}$.

Inputs:

- * *ai*: the 256-bit input block.

Steps:

```
t0, t1 = Split(ai, 128)
Update(t0, t1)
```

3.6. The Enc Function

```
Enc(xi)
```

The Enc function encrypts a 256-bit input block *xi* using the state $\{S0, \dots S7\}$.

Inputs:

- * *xi*: the 256-bit input block.

Outputs:

- * *ci*: the 256-bit encrypted block.

Steps:

```

z0 = S1 ^ S6 ^ (S2 & S3)
z1 = S2 ^ S5 ^ (S6 & S7)

t0, t1 = Split(xi, 128)
out0 = t0 ^ z0
out1 = t1 ^ z1

Update(t0, t1)
ci = out0 || out1

return ci

```

3.7. The Dec Function

Dec(ci)

The Dec function decrypts a 256-bit input block ci using the state {S0, ...S7}.

Inputs:

- * ci: the 256-bit encrypted block.

Outputs:

- * xi: the 256-bit decrypted block.

Steps:

```

z0 = S1 ^ S6 ^ (S2 & S3)
z1 = S2 ^ S5 ^ (S6 & S7)

t0, t1 = Split(ci, 128)
out0 = t0 ^ z0
out1 = t1 ^ z1

Update(out0, out1)
xi = out0 || out1

return xi

```

3.8. The DecPartial Function

DecPartial(cn)

The DecPartial function decrypts the last ciphertext bits cn using the state {S0, ...S7} when they do not fill an entire block.

Inputs:

- * `cn`: the encrypted input.

Outputs:

- * `xn`: the decryption of `cn`.

Steps:

```
z0 = S1 ^ S6 ^ (S2 & S3)
```

```
z1 = S2 ^ S5 ^ (S6 & S7)
```

```
t0, t1 = Split(ZeroPad(cn, 256), 128)
```

```
out0 = t0 ^ z0
```

```
out1 = t1 ^ z1
```

```
xn = Truncate(out0 || out1, |cn|)
```

```
v0, v1 = Split(ZeroPad(xn, 256), 128)
```

```
Update(v0, v1)
```

```
return xn
```

3.9. The Finalize Function

```
Finalize(ad_len_bits, msg_len_bits)
```

The `Finalize` function computes a 128- or 256-bit tag that authenticates the message and associated data.

Inputs:

- * `ad_len_bits`: the length of the associated data in bits.

- * `msg_len_bits`: the length of the message in bits.

Outputs:

- * `tag`: the authentication tag.

Steps:

```

t = S2 ^ (LE64(ad_len_bits) || LE64(msg_len_bits))

Repeat(7, Update(t, t))

if tag_len_bits == 128:
    tag = S0 ^ S1 ^ S2 ^ S3 ^ S4 ^ S5 ^ S6
else:
    # 256 bits
    tag = (S0 ^ S1 ^ S2 ^ S3) || (S4 ^ S5 ^ S6 ^ S7)

return tag

```

4. The AEGIS-256 Algorithm

AEGIS-256 has a 768-bit state, made of six 128-bit blocks {S0, ...S5}.

The parameters for this algorithm, whose meaning is defined in [RFC5116], Section 4, are:

- * K_LEN (key length) is 32 bytes (256 bits).
- * P_MAX (maximum length of the plaintext) is $2^{61} - 1$ bytes ($2^{64} - 8$ bits).
- * A_MAX (maximum length of the associated data) is $2^{61} - 1$ bytes ($2^{64} - 8$ bits).
- * N_MIN (minimum nonce length) = N_MAX (maximum nonce length) = 32 bytes (256 bits).
- * C_MAX (maximum ciphertext length) = P_MAX + tag length = $(2^{61} - 1) + 16$ or 32 bytes (in bits: $(2^{64} - 8) + 128$ or 256 bits).

Distinct associated data inputs, as described in [RFC5116], Section 3, MUST be unambiguously encoded as a single input. It is up to the application to create a structure in the associated data input if needed.

4.1. Authenticated Encryption

```
Encrypt(msg, ad, key, nonce)
```

The Encrypt function encrypts a message and returns the ciphertext along with an authentication tag that verifies the authenticity of the message and associated data, if provided.

Security:

- * For a given key, the nonce MUST NOT be reused under any circumstances; doing so allows an attacker to recover the internal state.
- * The key MUST be randomly chosen from a uniform distribution.

Inputs:

- * msg: the message to be encrypted (length MUST be less than or equal to P_MAX).
- * ad: the associated data to authenticate (length MUST be less than or equal to A_MAX).
- * key: the encryption key.
- * nonce: the public nonce.

Outputs:

- * ct: the ciphertext.
- * tag: the authentication tag.

Steps:

Init(key, nonce)

ct = {}

```
ad_blocks = Split(ZeroPad(ad, 128), 128)
for ai in ad_blocks:
    Absorb(ai)
```

```
msg_blocks = Split(ZeroPad(msg, 128), 128)
for xi in msg_blocks:
    ct = ct || Enc(xi)
```

```
tag = Finalize(|ad|, |msg|)
ct = Truncate(ct, |msg|)
```

return ct and tag

4.2. Authenticated Decryption

Decrypt(ct, tag, ad, key, nonce)

The Decrypt function decrypts a ciphertext, verifies that the authentication tag is correct, and returns the message on success or an error if tag verification failed.

Security:

- * If tag verification fails, the decrypted message and wrong authentication tag MUST NOT be given as output. The decrypted message MUST be overwritten with zeros before the function returns.
- * The comparison of the input tag with the expected_tag MUST be done in constant time.

Inputs:

- * ct: the ciphertext to decrypt (length MUST be less than or equal to C_MAX).
- * tag: the authentication tag.
- * ad: the associated data to authenticate (length MUST be less than or equal to A_MAX).
- * key: the encryption key.
- * nonce: the public nonce.

Outputs:

- * Either the decrypted message msg or an error indicating that the authentication tag is invalid for the given inputs.

Steps:

```

Init(key, nonce)

msg = {}

ad_blocks = Split(ZeroPad(ad, 128), 128)
for ai in ad_blocks:
    Absorb(ai)

ct_blocks = Split(ZeroPad(ct, 128), 128)
cn = Tail(ct, |ct| mod 128)

for ci in ct_blocks:
    msg = msg || Dec(ci)

if cn is not empty:
    msg = msg || DecPartial(cn)

expected_tag = Finalize(|ad|, |msg|)

if CtEq(tag, expected_tag) is False:
    erase msg
    erase expected_tag
    return "verification failed" error
else:
    return msg

```

4.3. The Update Function

Update(M)

The Update function is the core of the AEGIS-256 algorithm. It updates the state $\{S_0, \dots S_5\}$ using a 128-bit value.

Inputs:

- * msg: the 128-bit block to be absorbed.

Modifies:

- * $\{S_0, \dots S_5\}$: the state.

Steps:

```
S'0 = AESRound(S5, S0 ^ M)
S'1 = AESRound(S0, S1)
S'2 = AESRound(S1, S2)
S'3 = AESRound(S2, S3)
S'4 = AESRound(S3, S4)
S'5 = AESRound(S4, S5)
```

```
S0 = S'0
S1 = S'1
S2 = S'2
S3 = S'3
S4 = S'4
S5 = S'5
```

4.4. The Init Function

Init(key, nonce)

The Init function constructs the initial state $\{S_0, \dots, S_5\}$ using the given key and nonce.

Inputs:

- * key: the encryption key.
- * nonce: the public nonce.

Defines:

- * $\{S_0, \dots, S_5\}$: the initial state.

Steps:

```

k0, k1 = Split(key, 128)
n0, n1 = Split(nonce, 128)

```

```

S0 = k0 ^ n0
S1 = k1 ^ n1
S2 = C1
S3 = C0
S4 = k0 ^ C0
S5 = k1 ^ C1

```

```

Repeat(4,
  Update(k0)
  Update(k1)
  Update(k0 ^ n0)
  Update(k1 ^ n1)
)

```

4.5. The Absorb Function

```
Absorb(ai)
```

The Absorb function absorbs a 128-bit input block *ai* into the state {*S0*, ...*S5*}.

Inputs:

- * *ai*: the 128-bit input block.

Steps:

```
Update(ai)
```

4.6. The Enc Function

```
Enc(xi)
```

The Enc function encrypts a 128-bit input block *xi* using the state {*S0*, ...*S5*}.

Inputs:

- * *xi*: the 128-bit input block.

Outputs:

- * *ci*: the 128-bit encrypted block.

Steps:

```
z = S1 ^ S4 ^ S5 ^ (S2 & S3)
```

```
Update(xi)
```

```
ci = xi ^ z
```

```
return ci
```

4.7. The Dec Function

```
Dec(ci)
```

The Dec function decrypts a 128-bit input block *ci* using the state {*S0*, ...*S5*}.

Inputs:

- * *ci*: the 128-bit encrypted block.

Outputs:

- * *xi*: the 128-bit decrypted block.

Steps:

```
z = S1 ^ S4 ^ S5 ^ (S2 & S3)
```

```
xi = ci ^ z
```

```
Update(xi)
```

```
return xi
```

4.8. The DecPartial Function

```
DecPartial(cn)
```

The DecPartial function decrypts the last ciphertext bits *cn* using the state {*S0*, ...*S5*} when they do not fill an entire block.

Inputs:

- * *cn*: the encrypted input.

Outputs:

- * *xn*: the decryption of *cn*.

Steps:

```
z = S1 ^ S4 ^ S5 ^ (S2 & S3)
```

```
t = ZeroPad(cn, 128)
out = t ^ z
```

```
xn = Truncate(out, |cn|)
```

```
v = ZeroPad(xn, 128)
Update(v)
```

```
return xn
```

4.9. The Finalize Function

```
Finalize(ad_len_bits, msg_len_bits)
```

The Finalize function computes a 128- or 256-bit tag that authenticates the message and associated data.

Inputs:

- * ad_len_bits: the length of the associated data in bits.
- * msg_len_bits: the length of the message in bits.

Outputs:

- * tag: the authentication tag.

Steps:

```
t = S3 ^ (LE64(ad_len_bits) || LE64(msg_len_bits))
```

```
Repeat(7, Update(t))
```

```
if tag_len_bits == 128:
    tag = S0 ^ S1 ^ S2 ^ S3 ^ S4 ^ S5
else:
    # 256 bits
    tag = (S0 ^ S1 ^ S2) || (S3 ^ S4 ^ S5)
```

```
return tag
```

5. Parallel Modes

Some CPUs, such as Intel and Intel-compatible CPUs with the VAES extensions, include instructions to efficiently apply the AES round function to a vector of AES blocks.

AEGIS-128X and AEGIS-256X are optional, specialized modes designed to take advantage of these instructions. They share the same properties as the ciphers they are based on but can be significantly faster on these platforms, even for short messages.

AEGIS-128X and AEGIS-256X are parallel evaluations of multiple AEGIS-128L and AEGIS-256 instances, respectively, with distinct initial states. On CPUs with wide vector registers, different states can be stored in different 128-bit lanes of the same vector register, allowing parallel updates using vector instructions.

The modes are parameterized by the parallelism degree. With 256-bit registers, 2 parallel operations can be applied to 128-bit AES blocks. With 512-bit registers, the number of instances can be raised to 4.

The state of a parallel mode is represented as a vector of AEGIS-128L or AEGIS-256 states.

5.1. Additional Conventions and Definitions

- * D : the degree of parallelism.
- * R : the absorption and output rate of the mode. With AEGIS-128X, the rate is $256 * D$ bits. With AEGIS-256X, the rate is $128 * D$ bits.
- * $V[j,i]$: the j -th AES block of the i -th state. i is in the $[0..D)$ range. For AEGIS-128X, j is in the $[0..8)$ range, while for AEGIS-256X, j is in the $[0..6)$ range.
- * $V'[j,i]$: the j -th AES block of the next i -th state.
- * $ctx[i]$: the i -th context separator. This is a 128-bit mask made of a byte representing the state index, followed by a byte representing the highest index and 112 all-zero bits.
- * $Byte(x)$: the value x encoded as 8 bits.

5.2. Authenticated Encryption

```
Encrypt(msg, ad, key, nonce)
```

The Encrypt function of AEGIS-128X resembles that of AEGIS-128L, and similarly, the Encrypt function of AEGIS-256X mirrors that of AEGIS-256, but processes R-bit input blocks per update.

Steps:

```
Init(key, nonce)
```

```
ct = {}
```

```
ad_blocks = Split(ZeroPad(ad, R), R)
for ai in ad_blocks:
    Absorb(ai)
```

```
msg_blocks = Split(ZeroPad(msg, R), R)
for xi in msg_blocks:
    ct = ct || Enc(xi)
```

```
tag = Finalize(|ad|, |msg|)
ct = Truncate(ct, |msg|)
```

```
return ct and tag
```

5.3. Authenticated Decryption

```
Decrypt(ct, tag, ad, key, nonce)
```

The Decrypt function of AEGIS-128X resembles that of AEGIS-128L, and similarly, the Decrypt function of AEGIS-256X mirrors that of AEGIS-256, but processes R-bit input blocks per update.

Steps:

```

Init(key, nonce)

msg = {}

ad_blocks = Split(ZeroPad(ad, R), R)
for ai in ad_blocks:
    Absorb(ai)

ct_blocks = Split(ct, R)
cn = Tail(ct, |ct| mod R)

for ci in ct_blocks:
    msg = msg || Dec(ci)

if cn is not empty:
    msg = msg || DecPartial(cn)

expected_tag = Finalize(|ad|, |msg|)

if CtEq(tag, expected_tag) is False:
    erase msg
    erase expected_tag
    return "verification failed" error
else:
    return msg

```

5.4. AEGIS-128X

5.4.1. The Update Function

```
Update(M0, M1)
```

The AEGIS-128X Update function is similar to the AEGIS-128L Update function but absorbs $R (= 256 * D)$ bits at once. $M0$ and $M1$ are $128 * D$ bits instead of 128 bits but are split into 128-bit blocks, each of them updating a different AEGIS-128L state.

Steps:

```

m0 = Split(M0, 128)
m1 = Split(M1, 128)

for i in 0..D:
    V'[0,i] = AESRound(V[7,i], V[0,i] ^ m0[i])
    V'[1,i] = AESRound(V[0,i], V[1,i])
    V'[2,i] = AESRound(V[1,i], V[2,i])
    V'[3,i] = AESRound(V[2,i], V[3,i])
    V'[4,i] = AESRound(V[3,i], V[4,i] ^ m1[i])
    V'[5,i] = AESRound(V[4,i], V[5,i])
    V'[6,i] = AESRound(V[5,i], V[6,i])
    V'[7,i] = AESRound(V[6,i], V[7,i])

    V[0,i] = V'[0,i]
    V[1,i] = V'[1,i]
    V[2,i] = V'[2,i]
    V[3,i] = V'[3,i]
    V[4,i] = V'[4,i]
    V[5,i] = V'[5,i]
    V[6,i] = V'[6,i]
    V[7,i] = V'[7,i]

```

5.4.2. The Init Function

```
Init(key, nonce)
```

The Init function initializes a vector of D AEGIS-128L states with the same key and nonce but a different context $ctx[i]$. The context is added to the state before every update.

Steps:

```

for i in 0..D:
    V[0,i] = key ^ nonce
    V[1,i] = C1
    V[2,i] = C0
    V[3,i] = C1
    V[4,i] = key ^ nonce
    V[5,i] = key ^ C0
    V[6,i] = key ^ C1
    V[7,i] = key ^ C0

nonce_v = {}
key_v = {}
for i in 0..D:
    nonce_v = nonce_v || nonce
    key_v = key_v || key

for i in 0..D:
    ctx[i] = ZeroPad(Byte(i) || Byte(D - 1), 128)

Repeat(10,
    for i in 0..D:
        V[3,i] = V[3,i] ^ ctx[i]
        V[7,i] = V[7,i] ^ ctx[i]

    Update(nonce_v, key_v)
)

```

5.4.3. The Absorb Function

Absorb(ai)

The Absorb function is similar to the AEGIS-128L Absorb function but absorbs R bits instead of 256 bits.

Steps:

```

t0, t1 = Split(ai, R)
Update(t0, t1)

```

5.4.4. The Enc Function

Enc(xi)

The Enc function is similar to the AEGIS-128L Enc function but encrypts R bits instead of 256 bits.

Steps:

```

z0 = {}
z1 = {}
for i in 0..D:
    z0 = z0 || (V[6,i] ^ V[1,i] ^ (V[2,i] & V[3,i]))
    z1 = z1 || (V[2,i] ^ V[5,i] ^ (V[6,i] & V[7,i]))

t0, t1 = Split(xi, R)
out0 = t0 ^ z0
out1 = t1 ^ z1

Update(t0, t1)
ci = out0 || out1

return ci

```

5.4.5. The Dec Function

Dec(ci)

The Dec function is similar to the AEGIS-128L Dec function but decrypts R bits instead of 256 bits.

Steps:

```

z0 = {}
z1 = {}
for i in 0..D:
    z0 = z0 || (V[6,i] ^ V[1,i] ^ (V[2,i] & V[3,i]))
    z1 = z1 || (V[2,i] ^ V[5,i] ^ (V[6,i] & V[7,i]))

t0, t1 = Split(ci, R)
out0 = t0 ^ z0
out1 = t1 ^ z1

Update(out0, out1)
xi = out0 || out1

return xi

```

5.4.6. The DecPartial Function

DecPartial(cn)

The DecPartial function is similar to the AEGIS-128L DecPartial function but decrypts up to R bits instead of 256 bits.

Steps:

```

z0 = {}
z1 = {}
for i in 0..D:
    z0 = z0 || (V[6,i] ^ V[1,i] ^ (V[2,i] & V[3,i]))
    z1 = z1 || (V[2,i] ^ V[5,i] ^ (V[6,i] & V[7,i]))

t0, t1 = Split(ZeroPad(cn, R), 128 * D)
out0 = t0 ^ z0
out1 = t1 ^ z1

xn = Truncate(out0 || out1, |cn|)

v0, v1 = Split(ZeroPad(xn, R), 128 * D)
Update(v0, v1)

return xn

```

5.4.7. The Finalize Function

```
Finalize(ad_len_bits, msg_len_bits)
```

The Finalize function finalizes every AEGIS-128L instance and combines the resulting authentication tags using the bitwise exclusive OR operation.

Steps:

```

t = {}
u = LE64(ad_len_bits) || LE64(msg_len_bits)
for i in 0..D:
    t = t || (V[2,i] ^ u)

Repeat(7, Update(t, t))

if tag_len_bits == 128:
    tag = ZeroPad({}, 128)
    for i in 0..D:
        ti = V[0,i] ^ V[1,i] ^ V[2,i] ^ V[3,i] ^ V[4,i] ^ V[5,i] ^ V[6,i]
        tag = tag ^ ti
else:
    # 256 bits
    ti0 = ZeroPad({}, 128)
    ti1 = ZeroPad({}, 128)
    for i in 0..D:
        ti0 = ti0 ^ V[0,i] ^ V[1,i] ^ V[2,i] ^ V[3,i]
        ti1 = ti1 ^ V[4,i] ^ V[5,i] ^ V[6,i] ^ V[7,i]
    tag = ti0 || ti1

return tag

```

5.5. AEGIS-256X

5.5.1. The Update Function

Update(M)

The AEGIS-256X Update function is similar to the AEGIS-256 Update function but absorbs R ($128 * D$) bits at once. M is $128 * D$ bits instead of 128 bits and is split into 128-bit blocks, each of them updating a different AEGIS-256 state.

Steps:

```

m = Split(M, 128)

for i in 0..D:
    V'[0,i] = AESRound(V[5,i], V[0,i] ^ m[i])
    V'[1,i] = AESRound(V[0,i], V[1,i])
    V'[2,i] = AESRound(V[1,i], V[2,i])
    V'[3,i] = AESRound(V[2,i], V[3,i])
    V'[4,i] = AESRound(V[3,i], V[4,i])
    V'[5,i] = AESRound(V[4,i], V[5,i])

    V[0,i] = V'[0,i]
    V[1,i] = V'[1,i]
    V[2,i] = V'[2,i]
    V[3,i] = V'[3,i]
    V[4,i] = V'[4,i]
    V[5,i] = V'[5,i]

```

5.5.2. The Init Function

```
Init(key, nonce)
```

The Init function initializes a vector of D AEGIS-256 states with the same key and nonce but a different context `ctx[i]`. The context is added to the state before every update.

Steps:

```

k0, k1 = Split(key, 128)
n0, n1 = Split(nonce, 128)

for i in 0..D:
    V[0,i] = k0 ^ n0
    V[1,i] = k1 ^ n1
    V[2,i] = C1
    V[3,i] = C0
    V[4,i] = k0 ^ C0
    V[5,i] = k1 ^ C1

k0_v, k1_v = {}, {}
k0n0_v, k1n1_v = {}, {}
for i in 0..D:
    k0_v = k0_v || k0
    k1_v = k1_v || k1
    k0n0_v = k0n0_v || (k0 ^ n0)
    k1n1_v = k1n1_v || (k1 ^ n1)

for i in 0..D:
    ctx[i] = ZeroPad(Byte(i) || Byte(D - 1), 128)

Repeat(4,
    for i in 0..D:
        V[3,i] = V[3,i] ^ ctx[i]
        V[5,i] = V[5,i] ^ ctx[i]

    Update(k0_v)
    for i in 0..D:
        V[3,i] = V[3,i] ^ ctx[i]
        V[5,i] = V[5,i] ^ ctx[i]

    Update(k1_v)
    for i in 0..D:
        V[3,i] = V[3,i] ^ ctx[i]
        V[5,i] = V[5,i] ^ ctx[i]

    Update(k0n0_v)
    for i in 0..D:
        V[3,i] = V[3,i] ^ ctx[i]
        V[5,i] = V[5,i] ^ ctx[i]

    Update(k1n1_v)
)

```

5.5.3. The Absorb Function

Absorb(ai)

The Absorb function is similar to the AEGIS-256 Absorb function but absorbs R bits instead of 128 bits.

Steps:

Update(ai)

5.5.4. The Enc Function

Enc(xi)

The Enc function is similar to the AEGIS-256 Enc function but encrypts R bits instead of 128 bits.

Steps:

```
z = {}  
for i in 0..D:  
    z = z || (V[1,i] ^ V[4,i] ^ V[5,i] ^ (V[2,i] & V[3,i]))
```

Update(xi)

ci = xi ^ z

return ci

5.5.5. The Dec Function

Dec(ci)

The Dec function is similar to the AEGIS-256 Dec function but decrypts R bits instead of 128 bits.

Steps:

```
z = {}  
for i in 0..D:  
    z = z || (V[1,i] ^ V[4,i] ^ V[5,i] ^ (V[2,i] & V[3,i]))
```

xi = ci ^ z

Update(xi)

return xi

5.5.6. The DecPartial Function

```
DecPartial(cn)
```

The DecPartial function is similar to the AEGIS-256 DecPartial function but decrypts up to R bits instead of 128 bits.

Steps:

```
z = {}
for i in 0..D:
    z = z || (V[1,i] ^ V[4,i] ^ V[5,i] ^ (V[2,i] & V[3,i]))

t = ZeroPad(cn, R)
out = t ^ z

xn = Truncate(out, |cn|)

v = ZeroPad(xn, 128 * D)
Update(v)

return xn
```

5.5.7. The Finalize Function

```
Finalize(ad_len_bits, msg_len_bits)
```

The Finalize function finalizes every AEGIS-256 instance and combines the resulting authentication tags using the bitwise exclusive OR operation.

Steps:

```

t = {}
u = LE64(ad_len_bits) || LE64(msg_len_bits)
for i in 0..D:
    t = t || (V[3,i] ^ u)

Repeat(7, Update(t))

if tag_len_bits == 128:
    tag = ZeroPad({}, 128)
    for i in 0..D:
        ti = V[0,i] ^ V[1,i] ^ V[2,i] ^ V[3,i] ^ V[4,i] ^ V[5,i]
        tag = tag ^ ti

else:
    # 256 bits
    ti0 = ZeroPad({}, 128)
    ti1 = ZeroPad({}, 128)
    for i in 0..D:
        ti0 = ti0 ^ V[0,i] ^ V[1,i] ^ V[2,i]
        ti1 = ti1 ^ V[3,i] ^ V[4,i] ^ V[5,i]
    tag = ti0 || ti1

return tag

```

5.6. Implementation Considerations

AEGIS-128X and AEGIS-256X with a degree of 1 are identical to AEGIS-128L and AEGIS-256, respectively. This property can be used to reduce the size of a generic implementation.

In AEGIS-128X, V can be represented as eight 256-bit registers (when $D = 2$) or eight 512-bit registers (when $D = 4$). In AEGIS-256X, V can be represented as six 256-bit registers (when $D = 2$) or six 512-bit registers (when $D = 4$). With this representation, loops over $0..D$ in the above pseudocode can be replaced by vector instructions.

5.7. Operational Considerations

The AEGIS parallel modes are specialized and can only improve performance on specific CPUs.

The degrees of parallelism implementations are encouraged to support are 2 (for CPUs with 256-bit registers) and 4 (for CPUs with 512-bit registers). The resulting algorithms are called AEGIS-128X2, AEGIS-128X4, AEGIS-256X2, and AEGIS-256X4.

The following table summarizes how many bits are processed in parallel (rate), the memory requirements (state size), and the minimum vector register size a CPU should support for optimal performance.

Algorithm	Rate (bits)	Optimal Register Size	State Size (bits)
AEGIS-128L	256	128 bits	1024
AEGIS-128X2	512	256 bits	2048
AEGIS-128X4	1024	512 bits	4096
AEGIS-256	128	128 bits	768
AEGIS-256X2	256	256 bits	1536
AEGIS-256X4	512	512 bits	3072

Table 1

Note that architectures with smaller vector registers but with many registers and large pipelines may still benefit from the parallel modes.

Protocols SHOULD opt for a parallel mode only when all the involved parties agree on a specific variant. AEGIS-128L and AEGIS-256 SHOULD remain the default choices.

Implementations MAY choose not to include the parallel AEGIS modes.

6. Encoding (ct, tag) Tuples

Applications MAY keep the ciphertext and the authentication tag in distinct structures or encode both as a single string.

In the latter case, the tag MUST immediately follow the ciphertext:

```
combined_ct = ct || tag
```

7. AEGIS as a Stream Cipher

All AEGIS variants can also be used as stream ciphers.

```
Stream(len, key, nonce)
```

The Stream function expands a key and an optional nonce into a variable-length keystream.

Inputs:

- * len: the length of the keystream to generate in bits.
- * key: the AEGIS key.
- * nonce: the AEGIS nonce. If unspecified, it is set to N_MAX zero bytes.

Outputs:

- * stream: the keystream.

Steps:

```
if len == 0:
    return {}
else:
    stream, tag = Encrypt(ZeroPad({ 0 }, len), {}, key, nonce)
    return stream
```

This is equivalent to encrypting a len all-zero bits message without associated data and discarding the authentication tag.

Instead of relying on the generic Encrypt function, implementations can omit the Finalize function.

After initialization, the Update function is called with constant parameters, allowing further optimizations.

8. AEGIS as a Message Authentication Code

All AEGIS variants can be used to construct a Message Authentication Code (MAC).

For all the variants, the Mac function takes a key, a nonce, and data as input and produces a 128- or 256-bit tag as output.

Mac(data, key, nonce)

Security:

- * This is the only function that allows the reuse of (key, nonce) pairs with different inputs.

- * AEGIS-based MAC functions MUST NOT be used as hash functions: if the key is known, inputs causing state collisions can easily be crafted.
- * Unlike hash-based MACs, tags MUST NOT be used for key derivation as there is no guarantee that they are uniformly random.

Inputs:

- * data: the input data to authenticate (length MUST be less than or equal to A_MAX).
- * key: the secret key.
- * nonce: the public nonce.

Outputs:

- * tag: the authentication tag.

8.1. AEGISMAC-128L

AEGISMAC-128L refers to the Mac function based on the building blocks of AEGIS-128L.

Steps:

```
Init(key, nonce)
data_blocks = Split(ZeroPad(data, 256), 256)
for di in data_blocks:
    Absorb(di)
tag = Finalize(|data|, tag_len_bits)
return tag
```

8.2. AEGISMAC-256

AEGISMAC-256 refers to the Mac function based on the building blocks of AEGIS-256.

Steps:

```
Init(key, nonce)
data_blocks = Split(ZeroPad(data, 128), 128)
for di in data_blocks:
    Absorb(di)
tag = Finalize(|data|, tag_len_bits)
return tag
```

8.3. AEGISMAC-128X

AEGISMAC-128X is based on the building blocks of AEGIS-128X but replaces the Finalize function with a dedicated FinalizeMac function.

8.3.1. The Mac Function

Steps:

```
Init(key, nonce)
data_blocks = Split(ZeroPad(data, R), R)
for di in data_blocks:
    Absorb(di)
tag = FinalizeMac(|data|)
return tag
```

8.3.2. The FinalizeMac Function

FinalizeMac(data_len_bits)

The FinalizeMac function computes a 128- or 256-bit tag that authenticates the input data.

It finalizes all the instances, absorbs the resulting tags into the first state, and computes the final tag using that single state, as done in AEGIS-128L.

Steps:

```

t = {}
u = LE64(data_len_bits) || LE64(tag_len_bits)
for i in 0..D:
    t = t || (V[2,i] ^ u)

Repeat(7, Update(t, t))

tags = {}
if tag_len_bits == 128:
    for i in 0..D: # tag from state 0 is included
        ti = V[0,i] ^ V[1,i] ^ V[2,i] ^ V[3,i] ^ V[4,i] ^ V[5,i] ^ V[6,i]
        tags = tags || ti
else:
    # 256 bits
    for i in 1..D: # tag from state 0 is skipped
        ti0 = V[0,i] ^ V[1,i] ^ V[2,i] ^ V[3,i]
        ti1 = V[4,i] ^ V[5,i] ^ V[6,i] ^ V[7,i]
        tags = tags || (ti0 || ti1)

if D > 1:
    # Absorb tags into state 0; other states are not used anymore
    for v in Split(tags, 256):
        x0, x1 = Split(v, 128)
        Absorb(ZeroPad(x0, R / 2) || ZeroPad(x1, R / 2))

    u = LE64(D) || LE64(tag_len_bits)
    t = ZeroPad(V[2,0] ^ u, R)
    Repeat(7, Update(t, t))

if tag_len_bits == 128:
    tag = V[0,0] ^ V[1,0] ^ V[2,0] ^ V[3,0] ^ V[4,0] ^ V[5,0] ^ V[6,0]
else:
    # 256 bits
    t0 = V[0,0] ^ V[1,0] ^ V[2,0] ^ V[3,0]
    t1 = V[4,0] ^ V[5,0] ^ V[6,0] ^ V[7,0]
    tag = t0 || t1

return tag

```

8.4. AEGISMAC-256X

AEGISMAC-256X is based on the building blocks of AEGIS-256X but replaces the Finalize function with a dedicated FinalizeMac function.

8.4.1. The Mac Function

Steps:

```
Init(key, nonce)
data_blocks = Split(ZeroPad(data, R), R)
for di in data_blocks:
    Absorb(di)
tag = FinalizeMac(|data|)
return tag
```

8.4.2. The FinalizeMac Function

```
FinalizeMac(data_len_bits)
```

The FinalizeMac function computes a 128- or 256-bit tag that authenticates the input data.

It finalizes all the instances, absorbs the resulting tags into the first state, and computes the final tag using that single state, as done in AEGIS-256.

```

t = {}
u = LE64(data_len_bits) || LE64(tag_len_bits)
for i in 0..D:
    t = t || (V[3,i] ^ u)

Repeat(7, Update(t))

tags = {}
if tag_len_bits == 128:
    for i in 1..D: # tag from state 0 is skipped
        ti = V[0,i] ^ V[1,i] ^ V[2,i] ^ V[3,i] ^ V[4,i] ^ V[5,i]
        tags = tags || ti
else:
    # 256 bits
    for i in 1..D: # tag from state 0 is skipped
        ti0 = V[0,i] ^ V[1,i] ^ V[2,i]
        ti1 = V[3,i] ^ V[4,i] ^ V[5,i]
        tags = tags || (ti0 || ti1)

if D > 1:
    # Absorb tags into state 0; other states are not used anymore
    for v in Split(tags, 128):
        Absorb(ZeroPad(v, R))

    u = LE64(D) || LE64(tag_len_bits)
    t = ZeroPad(V[3,0] ^ u, R)
    Repeat(7, Update(t))

if tag_len_bits == 128:
    tag = V[0,0] ^ V[1,0] ^ V[2,0] ^ V[3,0] ^ V[4,0] ^ V[5,0]
else:
    # 256 bits
    t0 = V[0,0] ^ V[1,0] ^ V[2,0]
    t1 = V[3,0] ^ V[4,0] ^ V[5,0]
    tag = t0 || t1

return tag

```

9. Implementation Status

This note is to be removed before publishing as an RFC.

Multiple implementations of the schemes described in this document have been developed and verified for interoperability.

A comprehensive list of known implementations and integrations can be found at <https://github.com/cfrg/draft-irtf-cfrg-aegis-aead>, which includes reference implementations closely aligned with the pseudocode provided in this document.

10. Security Considerations

10.1. Usage Guidelines

10.1.1. Key and Nonce Selection

All AEGIS variants MUST be used in a nonce-respecting setting: for a given key, a nonce MUST only be used once, even with different tag lengths. Failure to do so would immediately reveal the bitwise difference between two messages.

Every key MUST be randomly chosen from a uniform distribution.

The nonce MAY be public or predictable. It can be a counter, the output of a permutation, or a generator with a long period.

With AEGIS-128L and AEGIS-128X, random nonces can safely encrypt up to 2^{48} messages using the same key with negligible ($\sim 2^{-33}$, to align with NIST guidelines) collision probability.

With AEGIS-256 and AEGIS-256X, random nonces can be used with no practical limits.

10.1.2. Committing Security

An authentication tag may verify under multiple keys, nonces, or associated data, but AEGIS is assumed to be key committing in the receiver-binding game. This mitigates common attacks when used with low-entropy keys such as passwords. Finding distinct keys and/or nonces that successfully verify the same (ad, ct, tag) tuple is expected to require $\sim 2^{64}$ attempts with a 128-bit authentication tag and $\sim 2^{128}$ attempts with a 256-bit tag.

AEGIS is fully committing in the restricted setting where an adversary cannot control the associated data. As shown in [IR23], with the ability to alter the associated data, it is possible to efficiently find multiple keys that will verify the same authenticated ciphertext.

Protocols mandating a fully committing scheme without that restriction can provide the associated data as input to a cryptographic hash function and use the output as the ad parameter of the Encrypt and Decrypt functions. The selected hash function must ensure a minimum of 128-bit collision and preimage resistance. An instance of such a function is SHA-256 [RFC6234].

Alternatively, the associated data can be fed into a collision-resistant KDF, such as HKDF [RFC5869], via the info input to derive the key parameter. The ad parameter can then be left empty. Note that the salt input MUST NOT be used since large salts get hashed, which affects commitment. Furthermore, this requires values concatenated to form the info input to be unambiguously encoded, like by appending their lengths.

10.1.3. Multi-User Security

AEGIS nonces match the size of the key. AEGIS-128L and AEGIS-128X feature 128-bit nonces, offering an extra 32 bits compared to the commonly used AEADs in IETF protocols at the time of writing. The AEGIS-256 and AEGIS-256X variants provide even larger nonces. With 192 random bits, 64 bits remain available to optionally encode additional information.

In all these variants, unused nonce bits can encode a key identifier, enhancing multi-user security. If every key has a unique identifier, multi-target attacks do not provide any advantage over single-target attacks.

10.2. Implementation Security

If tag verification fails, the unverified plaintext and computed authentication tag MUST NOT be released. As shown in [VV18], even a partial leak of the plaintext without verification facilitates chosen ciphertext attacks.

The security of AEGIS against timing and physical attacks is limited by the implementation of the underlying AESRound function. Failure to implement AESRound in a fashion safe against timing and physical attacks, such as differential power analysis, timing analysis, or fault injection attacks, may lead to leakage of secret key material or state information. The exact mitigations required for timing and physical attacks depend on the threat model in question.

Regardless of the variant, the key and nonce are only required by the Init function; other functions only depend on the resulting state. Therefore, implementations can overwrite ephemeral keys with zeros right after the last Update call of the initialization function.

10.3. Security Guarantees

AEGIS-256 offers 256-bit security against plaintext and state recovery, whereas AEGIS-128L offers 128-bit security.

Under the assumption that the secret key is unknown to the attacker, all AEGIS variants offer at least 128-bit security against forgery attacks.

Encrypting the same message with the same key and nonce but different associated data generates distinct ciphertexts that do not reveal any additional information about the message. However, (key, nonce) pairs MUST NOT be reused, even if the associated data differs.

AEGIS has been shown to have reforgeability resilience in [FLLW17]. Without the ability to set the associated data, a successful forgery does not increase the probability of subsequent forgeries.

AEGIS-128X and AEGIS-256X share the same security properties and requirements as AEGIS-128L and AEGIS-256, respectively. In particular, the security level and usage limits remain the same [D23].

AEGIS is considered secure against guess-and-determine attacks aimed at recovering the state from observed ciphertexts.

This resilience extends to quantum adversaries operating within the Q1 model, where the attacker has access to a quantum computer but is restricted to classical (non-quantum) communications with the systems under attack. In this model, quantum attacks offer no practical advantage in decrypting previously recorded ciphertexts or in recovering the encryption key.

This document extends the original specification by introducing optional support for 256-bit authentication tags, which are constructed similarly to the 128-bit tags. As shown in [SSI24], with 256-bit tags, all AEGIS variants achieve more than 128-bit security against forgery by differential attacks.

Security analyses of AEGIS can be found in [AEGIS], [M14], [FLLW17], [ENP20], [LIMS21], [JLD22], [STSI23], [IR23], [BS23], [AIKRS24], and [SSI24].

11. IANA Considerations

IANA has assigned the following identifiers in the AEAD Algorithms Registry:

Algorithm Name	ID
AEAD_AEGIS128L	32
AEAD_AEGIS256	33

Table 2: AEGIS entries
in the AEAD Algorithms
Registry

IANA is requested to update the references of these entries to refer to the final version of this document.

IANA is also requested to register the following entries in the AEAD Algorithms Registry:

Algorithm Name	ID
AEAD_AEGIS128X2	
AEAD_AEGIS128X4	
AEAD_AEGIS256X2	
AEAD_AEGIS256X4	

Table 3: Missing AEGIS
entries in the AEAD
Algorithms Registry

The identifier is left to be assigned by the IANA expert.

12. References

12.1. Normative References

- [FIPS-AES] NIST, "Advanced encryption standard (AES)", NIST Federal Information Processing Standards Publications 197, DOI 10.6028/NIST.FIPS.197, November 2001, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>>.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<https://www.rfc-editor.org/rfc/rfc5116>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/rfc/rfc5869>>.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/rfc/rfc6234>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.

12.2. Informative References

- [AEGIS] Wu, H. and B. Preneel, "AEGIS: A Fast Authenticated Encryption Algorithm (v1.1)", 2016, <<https://competitions.cr.yp.to/round3/aegisv11.pdf>>.
- [AIKRS24] Anand, R., Isobe, T., Kundu, A. K., Rahman, M., and S. Suryawanshi, "Differential fault attack on AES-based encryption schemes: application to B5G/6G ciphers—Rocca, Rocca-S and AEGIS", Journal of Cryptographic Engineering, 2024, DOI 10.1007/s13389-024-00360-6, 2024, <<https://doi.org/10.1007/s13389-024-00360-6>>.
- [BS23] Bonnetain, X. and A. Schrottenloher, "Single-query Quantum Hidden Shift Attacks", Cryptology ePrint Archive, Paper 2023/1306, 2023, <<https://eprint.iacr.org/2023/1306>>.
- [D23] Denis, F., "Adding more parallelism to the AEGIS authenticated encryption algorithms", Cryptology ePrint Archive, Paper 2023/523, 2023, <<https://eprint.iacr.org/2023/523>>.

- [ENP20] Eichlseder, M., Nageler, M., and R. Primas, "Analyzing the Linear Keystream Biases in AEGIS", IACR Transactions on Symmetric Cryptology, 2019(4), pp. 348368, DOI 10.13154/tosc.v2019.i4.348-368, 2020, <<https://doi.org/10.13154/tosc.v2019.i4.348-368>>.
- [FLLW17] Forler, C., List, E., Lucks, S., and J. Wenzel, "Reforgeability of Authenticated Encryption Schemes", Cryptology ePrint Archive, Paper 2017/332, 2017, <<https://eprint.iacr.org/2017/332>>.
- [IR23] Isobe, T. and M. Rahman, "Key Committing Security Analysis of AEGIS", Cryptology ePrint Archive, Paper 2023/1495, 2023, <<https://eprint.iacr.org/2023/1495>>.
- [JLD22] Jiao, L., Li, Y., and S. Du, "Guess-and-Determine Attacks on AEGIS", The Computer Journal, vol 65, 2022(8), pp. 22212230, DOI 10.1093/comjnl/bxab059, 2022, <<https://doi.org/10.1093/comjnl/bxab059>>.
- [LGR21] Len, J., Grubbs, P., and T. Ristenpart, "Partitioning Oracle Attacks", 30th USENIX Security Symposium (USENIX Security 21), pp. 195212, 2021, <<https://www.usenix.org/conference/usenixsecurity21/presentation/len>>.
- [LIMS21] Liu, F., Isobe, T., Meier, W., and K. Sakamoto, "Weak Keys in Reduced AEGIS and Tiaoxin", IACR Transactions on Symmetric Cryptology, 2021(2), pp. 104139, DOI 10.46586/tosc.v2021.i2.104-139, 2021, <<https://doi.org/10.46586/tosc.v2021.i2.104-139>>.
- [M14] Minaud, B., "Linear Biases in AEGIS Keystream", Selected Areas in Cryptography. SAC 2014. Lecture Notes in Computer Science, vol 8781, pp. 290305, DOI 10.1007/978-3-319-13051-4_18, 2014, <https://doi.org/10.1007/978-3-319-13051-4_18>.
- [SSI24] Shiraya, T., Sakamoto, K., and T. Isobe, "Bit-Wise Analysis for Forgery Attacks on AES-Based AEAD Schemes", Advances in Information and Computer Security. IWSEC 2024. Lecture Notes in Computer Science, vol 14977, DOI 10.1007/978-981-97-7737-2_1, 2024, <https://doi.org/10.1007/978-981-97-7737-2_1>.

- [STSI23] Shiraya, T., Takeuchi, N., Sakamoto, K., and T. Isobe, "MILP-based security evaluation for AEGIS/Tiaoxin-346/Rocca", IET Information Security, vol 17, 2023(3), pp. 458-467, DOI 10.1049/ise2.12109, 2023, <<https://doi.org/10.1049/ise2.12109>>.
- [TEST-VECTORS] "AEGIS Test Vectors", commit 8e289c40, 2025, <<https://github.com/cfrg/draft-irtf-cfrg-aegis-aead/tree/8e289c40/test-vectors>>.
- [VV18] Vaudenay, S. and D. Vizir, "Can Caesar Beat Galois?", Applied Cryptography and Network Security. ACNS 2018. Lecture Notes in Computer Science, vol 10892, pp. 476-494, DOI 10.1007/978-3-319-93387-0_25, 2018, <https://doi.org/10.1007/978-3-319-93387-0_25>.

Appendix A. Test Vectors

The following test vectors are also available in JSON format at [TEST-VECTORS]. In this format, byte strings are represented as JSON strings containing their hexadecimal encoding.

A.1. AESRound Test Vector

in : 000102030405060708090a0b0c0d0e0f

rk : 101112131415161718191a1b1c1d1e1f

out : 7a7b4e5638782546a8c0477a3b813f43

A.2. AEGIS-128L Test Vectors

A.2.1. Update Test Vector

S0 : 9b7e60b24cc873ea894ecc07911049a3
S1 : 330be08f35300faa2ebf9a7b0d274658
S2 : 7bbd5bd2b049f7b9b515cf26fbe7756c
S3 : c35a00f55ea86c3886ec5e928f87db18
S4 : 9ebccafce87cab446396c4334592c91f
S5 : 58d83e31f256371e60fc6bb257114601
S6 : 1639b56ea322c88568a176585bc915de
S7 : 640818ffb57dc0fbc2e72ae93457e39a

M0 : 033e6975b94816879e42917650955aa0
M1 : fcc1968a46b7e97861bd6e89af6aa55f

After Update:

S0 : 596ab773e4433ca0127c73f60536769d
S1 : 790394041a3d26ab697bde865014652d
S2 : 38cf49e4b65248acd533041b64dd0611
S3 : 16d8e58748f437bfff1797f780337cee
S4 : 9689ecdf08228c74d7e3360cca53d0a5
S5 : a21746bb193a569e331e1aa985d0d729
S6 : 09d714e6fcf9177a8ed1cde7e3d259a6
S7 : 61279ba73167f0ab76f0a11bf203bdf

A.2.2. Test Vector 1

key : 10010000000000000000000000000000
nonce : 10000200000000000000000000000000
ad :
msg : 00000000000000000000000000000000
ct : c1c0e58bd913006feba00f4b3cc3594e
tag128: abe0ece80c24868a226a35d16bdae37a
tag256: 25835bfbb21632176cf03840687cb968
 cace4617af1bd0f7d064c639a5c79ee4

A.2.3. Test Vector 2

A.2.10. Test Vector 9

This test MUST return a “verification failed” error.

```
key   : 1001000000000000000000000000000000
nonce : 1000020000000000000000000000000000
ad    : 0001020304050607
ct    : 79d94593d8c2119d7e8fd9b8fc77
tag128: 6c04b3dba849b2701effbe32c7f0fab8
tag256: 86f1b80bfb463aba711d15405d094baf
       4a55a15dbfec81a76f35ed0b9c8b04ad
```

A.3. AEGIS-256 Test Vectors

A.3.1. Update Test Vector

```
S0   : 1fa1207ed76c86f2c4bb40e8b395b43e
S1   : b44c375e6c1e1978db64bcd12e9e332f
S2   : 0dab84bfa9f0226432ff630f233d4e5b
S3   : d7ef65c9b93e8ee60c75161407b066e7
S4   : a760bb3da073fbd92bdc24734b1f56fb
S5   : a828a18d6a964497ac6e7e53c5f55c73

M    : b165617ed04ab738afb2612c6d18a1ec
```

After Update:

```
S0   : e6bc643bae82dfa3d991b1b323839dcd
S1   : 648578232ba0f2f0a3677f617dc052c3
S2   : ea788e0e572044a46059212dd007a789
S3   : 2f1498ae19b80da13fba698f088a8590
S4   : a54c2ee95e8c2a2c3dae2ec743ae6b86
S5   : a3240fceb68e32d5d114df1b5363ab67
```

A.3.2. Test Vector 1

key : 10010000000000000000000000000000
00000000000000000000000000000000

nonce : 10000200000000000000000000000000
00000000000000000000000000000000

ad :

msg : 00000000000000000000000000000000

ct : 754fc3d8c973246dcc6d741412a4b236

tag128: 3fe91994768b332ed7f570a19ec5896e

tag256: 1181ald18091082bf0266f66297d167d
2e68b845f61a3b0527d31fc7b7b89f13

A.3.3. Test Vector 2

key : 10010000000000000000000000000000
00000000000000000000000000000000

nonce : 10000200000000000000000000000000
00000000000000000000000000000000

ad :

msg :

ct :

tag128: e3def978a0f054afd1e761d7553afba3

tag256: 6a348c930adbd654896e1666aad67de9
89ea75ebaa2b82fb588977b1ffec864a

A.3.4. Test Vector 3

key : 10010000000000000000000000000000
00000000000000000000000000000000

nonce : 10000200000000000000000000000000
00000000000000000000000000000000

ad : 0001020304050607

msg : 000102030405060708090a0b0c0d0e0f
101112131415161718191a1b1c1d1e1f

ct : f373079ed84b2709faee373584585d60
accd191db310ef5d8b11833df9dec711

tag128: 8d86f91ee606e9ff26a01b64ccbdd91d

tag256: b7d28d0c3c0ebd409fd22b4416050307
3a547412da0854bfb9723020dab8da1a

A.3.5. Test Vector 4

key : 10010000000000000000000000000000
00000000000000000000000000000000

nonce : 10000200000000000000000000000000
00000000000000000000000000000000

ad : 0001020304050607

msg : 000102030405060708090a0b0c0d

ct : f373079ed84b2709faee37358458

tag128: c60b9c2d33ceb058f96e6dd03c215652

tag256: 8c1cc703c81281bee3f6d9966e14948b
4a175b2efbdc31e61a98b4465235c2d9

A.3.6. Test Vector 5

key : 10010000000000000000000000000000
00000000000000000000000000000000

nonce : 10000200000000000000000000000000
00000000000000000000000000000000

ad : 000102030405060708090a0b0c0d0e0f
101112131415161718191a1b1c1d1e1f
20212223242526272829

msg : 101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f
3031323334353637

ct : 57754a7d09963e7c787583a2e7b859bb
24fa1e04d49fd550b2511a358e3bca25
2a9b1b8b30cc4a67

tag128: ab8a7d53fd0e98d727accca94925e128

tag256: a3aca270c006094d71c20e6910b5161c
0826df233d08919a566ec2c05990f734

A.3.7. Test Vector 6

This test MUST return a “verification failed” error.

key : 10000200000000000000000000000000
00000000000000000000000000000000

nonce : 10010000000000000000000000000000
00000000000000000000000000000000

ad : 0001020304050607

ct : f373079ed84b2709faee37358458

tag128: c60b9c2d33ceb058f96e6dd03c215652

tag256: 8c1cc703c81281bee3f6d9966e14948b
4a175b2efbdc31e61a98b4465235c2d9

A.3.8. Test Vector 7

This test MUST return a “verification failed” error.

key : 10010000000000000000000000000000
00000000000000000000000000000000

nonce : 10000200000000000000000000000000
00000000000000000000000000000000

ad : 0001020304050607

ct : f373079ed84b2709faee37358459

tag128: c60b9c2d33ceb058f96e6dd03c215652

tag256: 8c1cc703c81281bee3f6d9966e14948b
4a175b2efbdc31e61a98b4465235c2d9

A.3.9. Test Vector 8

This test MUST return a “verification failed” error.

key : 10010000000000000000000000000000
00000000000000000000000000000000

nonce : 10000200000000000000000000000000
00000000000000000000000000000000

ad : 0001020304050608

ct : f373079ed84b2709faee37358458

tag128: c60b9c2d33ceb058f96e6dd03c215652

tag256: 8c1cc703c81281bee3f6d9966e14948b
4a175b2efbdc31e61a98b4465235c2d9

A.3.10. Test Vector 9

This test MUST return a “verification failed” error.

key : 10010000000000000000000000000000
00000000000000000000000000000000

nonce : 10000200000000000000000000000000
00000000000000000000000000000000

ad : 0001020304050607

ct : f373079ed84b2709faee37358458

tag128: c60b9c2d33ceb058f96e6dd03c215653

tag256: 8c1cc703c81281bee3f6d9966e14948b
4a175b2efbdc31e61a98b4465235c2da

A.4. AEGIS-128X2 Test Vectors

A.4.1. Initial State

key : 000102030405060708090a0b0c0d0e0f

nonce : 101112131415161718191a1b1c1d1e1f

ctx[0]: 00010000000000000000000000000000
ctx[1]: 01010000000000000000000000000000

After initialization:

V[0,0]: a4fc1ad9a72942fb88bd2cabbba6509a
V[0,1]: 80a40e392fc71084209b6c3319bdc6cc

V[1,0]: 380f435cf801763b1f0c2a2f7212052d
V[1,1]: 73796607b59b1b650ee91c152af1f18a

V[2,0]: 6ee1de433ea877fa33bc0782abff2dcb
V[2,1]: b9fab2ab496e16d1facaffd5453cbf14

V[3,0]: 85f94b0d4263bfa86fdf45a603d8b6ac
V[3,1]: 90356c8cadbaa2c969001da02e3fecaf0

V[4,0]: 09bd69ad3730174bcd2ce9a27cd1357e
V[4,1]: e610b45125796a4fcf1708cef5c4f718

V[5,0]: fcdeb0cf0a87bf442fc82383ddb0f6d6
V[5,1]: 61ad32a4694d6f3cca313a2d3f4687aa

V[6,0]: 571c207988659e2cdfbdaae77f4f37e3
V[6,1]: 32e6094e217573bf91fb28c145a3efa8

V[7,0]: ca549badf8faa58222412478598651cf
V[7,1]: 3407279a54ce76d2e2e8a90ec5d108eb

A.4.2. Test Vector 1

key : 000102030405060708090a0b0c0d0e0f

nonce : 101112131415161718191a1b1c1d1e1f

ad :

msg :

ct :

tag128: 63117dc57756e402819a82e13eca8379

tag256: b92c71fdbd358b8a4de70b27631ace90
cffd9b9cfba82028412bac41b4f53759

A.4.3. Test Vector 2

```
key   : 000102030405060708090a0b0c0d0e0f

nonce : 101112131415161718191a1b1c1d1e1f

ad    : 0102030401020304

msg   : 04050607040506070405060704050607
        04050607040506070405060704050607
        04050607040506070405060704050607
        04050607040506070405060704050607
        04050607040506070405060704050607
        04050607040506070405060704050607
        04050607040506070405060704050607
        0405060704050607

ct     : 5795544301997f93621b278809d6331b
        3bfa6f18e90db12c4aa35965b5e98c5f
        c6fb4e54bcb6111842c20637252eff74
        7cb3a8f85b37de80919a589fe0f24872
        bc926360696739e05520647e390989e1
        eb5fd42f99678a0276a498f8c454761c
        9d6aacb647ad56be62b29c22cd4b5761
        b38f43d5a5ee062f

tag128: 1aebc200804f405cab637f2adebb6d77

tag256: c471876f9b4978c44f2ae1ce770cdb11
        a094ee3feca64e7afcd48bfe52c60eca
```

A.5. AEGIS-128X4 Test Vectors

A.5.1. Initial State

```
key   : 000102030405060708090a0b0c0d0e0f

nonce : 101112131415161718191a1b1c1d1e1f

ctx[0]: 00030000000000000000000000000000
ctx[1]: 01030000000000000000000000000000
ctx[2]: 02030000000000000000000000000000
ctx[3]: 03030000000000000000000000000000
```

After initialization:

V[0,0]: 924eb07635003a37e6c6575ba8ce1929
V[0,1]: c8b6a5d91475445e936d48e794be0ce2
V[0,2]: fcd37d050e24084befe3bbb219d64760
V[0,3]: 2e9f58cfb893a8800220242c373a8b18

V[1,0]: 1a1f60c4fab64e5471dc72edfcf6fe6b
V[1,1]: c1e525ebea2d6375a9edd045dce96381
V[1,2]: 97a3e25abd228a44d4a14a6d3fe9185c
V[1,3]: c2d4cf7f4287a98744645674265d4ca8

V[2,0]: 7bb50c534f6ec4780530ff1cce8a16e8
V[2,1]: 7b08d57557da0b5ef7b5f7d98b0ba189
V[2,2]: 6bfcac34ddb68404821a4d665303cb0f
V[2,3]: d95626f6dfad1aed7467622c38529932

V[3,0]: af339fd2d50ee45fc47665c647cf6586
V[3,1]: d0669b39d140f0e118a4a511efe2f95a
V[3,2]: 7a94330f35c194fadda2a87e42cdeccc
V[3,3]: 233b640d1f4d56e2757e72c1a9d8ecb1

V[4,0]: 9f93737d699ba05c11e94f2b201bef5e
V[4,1]: 61caf387cf7cfd3f8300ac7680ccfd76
V[4,2]: 5825a671ecef03b7a9c98a601ae32115
V[4,3]: 87a1fe4d558161a8f4c38731f3223032

V[5,0]: 7a5aca78d636c05bbc702b2980196ab6
V[5,1]: 915d868408495d07eb527789f282c575
V[5,2]: d0947bfbcd1d3309cdffc9be1503aea62
V[5,3]: 8834ea57a15b9fbdc0245464a4b8cbef

V[6,0]: e46f4cf71a95ac45b6f0823e3abala86
V[6,1]: 8c4ecef682fc44a8eba911b3fc7d99f9
V[6,2]: a4fb61e2c928a2ca760b8772f2ea5f2e
V[6,3]: 3d34ea89da73caa3016c280500a155a3

V[7,0]: 85075f0080e9d618e7eb40f57c32d9f7
V[7,1]: d2ab2b320c6e93b155a3787cb83e5281
V[7,2]: 0b3af0250ae36831a1b072e499929bcb
V[7,3]: 5cce4d00329d69f1aae36aa541347512

A.5.2. Test Vector 1

key : 000102030405060708090a0b0c0d0e0f
nonce : 101112131415161718191a1b1c1d1e1f
ad :
msg :
ct :
tag128: 5bef762d0947c00455b97bb3af30dfa3
tag256: a4b25437f4be93cfa856a2f27e4416b4
 2cac79fd4698f2cdbe6af25673e10a68

A.5.3. Test Vector 2

key : 000102030405060708090a0b0c0d0e0f
nonce : 101112131415161718191a1b1c1d1e1f
ad : 0102030401020304
msg : 04050607040506070405060704050607
 04050607040506070405060704050607
 04050607040506070405060704050607
 04050607040506070405060704050607
 04050607040506070405060704050607
 04050607040506070405060704050607
 04050607040506070405060704050607
 0405060704050607
ct : e836118562f4479c9d35c17356a83311
 4c21f9aa39e4dda5e5c87f4152a00fce
 9a7c38f832eafe8b1c12f8a7cf12a81a
 1ad8a9c24ba9dedfbdaa586ffea67ddc
 801ea97d9ab4a872f42d0e352e2713da
 cd609f9442c17517c5a29daf3e2a3fac
 4ff6b1380c4e46df7b086af6ce6bc1ed
 594b8dd64aed2a7e
tag128: 0e56ab94e2e85db80f9d54010caabfb4
tag256: 69abf0f64a137dd6e122478d777e98bc
 422823006cf57f5ee822dd78397230b2

A.6. AEGIS-256X2 Test Vectors

A.6.1. Initial State

key : 000102030405060708090a0b0c0d0e0f
 101112131415161718191a1b1c1d1e1f

nonce : 101112131415161718191a1b1c1d1e1f
 202122232425262728292a2b2c2d2e2f

ctx[0]: 00010000000000000000000000000000
ctx[1]: 01010000000000000000000000000000

After initialization:

V[0,0]: eca2bf4538442e8712d4972595744039
V[0,1]: 201405efa9264f07911db58101903087

V[1,0]: 3e536a998799408a97f3479a6f779d48
V[1,1]: 0d79a7d822a5d215f78c3bf2feb33ae1

V[2,0]: cf8c63d6f2b4563cdd9231107c85950e
V[2,1]: 78d17ed7d8d563ff11bd202c76864839

V[3,0]: d7e0707e6bfbbad913bc94b6993a9fa0
V[3,1]: 097e4b1bfff40d4c19cb29dfd125d62f2

V[4,0]: a373cf6d537dd66bc0ef0f2f9285359f
V[4,1]: c0d0ae0c48f9df3faaf0e7be7768c326

V[5,0]: 9f76560dcae1efacabdcce446ae283bc
V[5,1]: bd52a6b9c8f976a26ec1409df19e8bfe

A.6.2. Test Vector 1

key : 000102030405060708090a0b0c0d0e0f
 101112131415161718191a1b1c1d1e1f

nonce : 101112131415161718191a1b1c1d1e1f
 202122232425262728292a2b2c2d2e2f

ad :

msg :

ct :

tag128: 62cdbab084c83dacdb945bb446f049c8

tag256: 25d7e799b49a80354c3f881ac2f1027f
 471a5d293052bd9997abd3ae84014bb7

A.6.3. Test Vector 2

```
key   : 000102030405060708090a0b0c0d0e0f
       101112131415161718191a1b1c1d1e1f

nonce : 101112131415161718191a1b1c1d1e1f
       202122232425262728292a2b2c2d2e2f

ad    : 0102030401020304

msg   : 0405060704050607040506070405060704050607
       0405060704050607040506070405060704050607
       0405060704050607040506070405060704050607
       0405060704050607040506070405060704050607
       0405060704050607040506070405060704050607
       0405060704050607040506070405060704050607
       0405060704050607040506070405060704050607
       0405060704050607

ct    : 73110d21a920608fd77b580f1e442808
       7a7365cb153b4eeca6b62e1a70f7f9a8
       d1f31f17da4c3acfacb2517f2f5e1575
       8c35532e33751a964d18d29a599d2dc0
       7f9378339b9d8c9fa03d30a4d7837cc8
       eb8b99bcbba2d11cd1a0f994af2b8f94
       7ef18473bd519e5283736758480abc99
       0e79d4ccab93dde9

tag128: 94a3bd44ad3381e36335014620ee638e

tag256: 0392c62b17ddb00c172a010b5a327d0f
       97317b6fbaee31ef741f004d7adc1e81
```

A.7. AEGIS-256X4 Test Vectors

A.7.1. Initial State

```
key   : 000102030405060708090a0b0c0d0e0f
       101112131415161718191a1b1c1d1e1f

nonce : 101112131415161718191a1b1c1d1e1f
       202122232425262728292a2b2c2d2e2f

ctx[0]: 0003000000000000000000000000000000000000
ctx[1]: 0103000000000000000000000000000000000000
ctx[2]: 0203000000000000000000000000000000000000
ctx[3]: 0303000000000000000000000000000000000000

After initialization:
```

```

V[0,0]: 482a86e8436cd2361063a4b2702769b9
V[0,1]: d95a2be81c9245b22996f68eea0122f9
V[0,2]: 0c2a3b348b1a5e256c6751377318c41e
V[0,3]: f64436a21653fe7cf2e0829a177db383

V[1,0]: e705e8866267717d96092e58e78b574c
V[1,1]: d1dd412142df9806cc267af2fe1d830e
V[1,2]: 30e7dfd3c9941b8394e95bdf5bac99d9
V[1,3]: 9f27186f8a4fab86820689822c3c74d2

V[2,0]: e1aa6af5d9e31dde8d94a48a0810fa89
V[2,1]: 63555cdf0d98f18fb75b029ad80786c0
V[2,2]: a3ee0e4a3429a9539e4fcec385475608
V[2,3]: 28ea527d31ef61df498dc107fe02df99

V[3,0]: 37f06808410c8f3954525ae44584d3be
V[3,1]: 8fcc23bca2fe2209f93d34e2da35b33d
V[3,2]: 33156347df89eaa69ab11096362daccf
V[3,3]: bbe58d9dbe8c5b0469be5a87086db5d4

V[4,0]: d1c9eb37fecbc5ada7b351fa4f501f32
V[4,1]: 0b9b803283c1538628b507c8f6432434
V[4,2]: bfb8b6d4f87cce28825c7e92f54b8728
V[4,3]: 8917bb5b09c32f900c6a5ald63c46264

V[5,0]: 4f6110c2ef0c3c687e90c1e5532ddf8e
V[5,1]: 031bd85d99f64684d23728a0453c72a1
V[5,2]: 10bc7ec34d4119b5bdeb6c7dfc458247
V[5,3]: 591ece530aeaa5c9867220156f5c25e3

```

A.7.2. Test Vector 1

```

key      : 000102030405060708090a0b0c0d0e0f
           101112131415161718191a1b1c1d1e1f

nonce    : 101112131415161718191a1b1c1d1e1f
           202122232425262728292a2b2c2d2e2f

ad       :

msg      :

ct       :

tag128: 3b7fee6cee7bf17888ad11ed2397beb4

tag256: 6093a1a8aab20ec635dc1ca71745b01b
        5bec4fc444c9ffbebd710d4a34d20eaf

```

A.7.3. Test Vector 2

```
key      : 000102030405060708090a0b0c0d0e0f
          101112131415161718191a1b1c1d1e1f

nonce    : 101112131415161718191a1b1c1d1e1f
          202122232425262728292a2b2c2d2e2f

ad       : 0102030401020304

msg      : 04050607040506070405060704050607
          04050607040506070405060704050607
          04050607040506070405060704050607
          04050607040506070405060704050607
          04050607040506070405060704050607
          04050607040506070405060704050607
          04050607040506070405060704050607
          0405060704050607

ct       : bec109547f8316d598b3b7d947ad4c0e
          f5b98e217cffa0d858ad49ae34109a95
          abc5b5fada820c4d6ae2fca0f5e2444e
          52a04a1edb7bec71408de3e199500521
          94506be3ba6a4de51a15a577ea0e4c14
          f7539a13e751a555f48d0f49fecffb22
          0525e60d381e2efa803b09b7164ba59f
          dc66656affd51e06

tag128:  ec44b512d713f745547be345bcc66b6c

tag256:  ba3168ecd7f7120c5e204a7e0d616e39
          5675ddfe00e4e5490a5ba93bbl1a70555
```

A.8. AEGISMAC Test Vectors

A.8.1. AEGISMAC-128L Test Vector

Acknowledgments

The AEGIS family of authenticated encryption algorithms was invented by Hongjun Wu and Bart Preneel.

The state update function leverages the AES permutation invented by Joan Daemen and Vincent Rijmen. They also authored the Pelican MAC, which partly motivated the design of the AEGIS MAC.

We would like to thank the following individuals for their contributions:

- * Eric Lagergren and Daniel Bleichenbacher for catching a broken test vector, and Daniel Bleichenbacher for many helpful suggestions.
- * Soatok Dreamseeker for his early review of the draft and for suggesting the addition of negative test vectors.
- * John Preu Mattsson for his review of the draft and for suggesting how AEGIS should be used in the context of DTLS and QUIC.
- * Bart Mennink and Charlotte Lefevre, as well as Takanori Isobe and Mostafizar Rahman, for investigating the committing security of the schemes specified in this document.
- * Scott Fluhrer for his review of the draft as a member of the CFRG Crypto Review Panel.
- * Yawning Angel, Chris Barber, and Neil Madden for their review of the draft.
- * Gilles Van Assche for reviewing the draft and providing insightful comments on the implications of nonce reuse in AEGIS-128X and AEGIS-256X.

Authors' Addresses

Frank Denis
Fastly Inc.
Email: fde@00f.net

Samuel Lucas
Individual Contributor
Email: samuel-lucas6@pm.me