The AEGIS family of authenticated encryption algorithms

Abstract

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

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/jedisct1/draft-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 6 February 2023.

Copyright Notice

Copyright (c) 2022 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.
Table of Contents

1. Introduction
2. Conventions and Definitions
3. The AEGIS-128L Algorithm
   3.1. Authenticated Encryption
   3.2. Authenticated Decryption
   3.3. The Init Function
   3.4. The Update Function
   3.5. The Enc Function
   3.6. The Dec Function
   3.7. The DecPartial Function
   3.8. The Finalize Function
4. The AEGIS-256 Algorithm
   4.1. Authenticated Encryption
   4.2. Authenticated Decryption
   4.3. The Init Function
   4.4. The Update Function
   4.5. The Enc Function
   4.6. The Dec Function
   4.7. The DecPartial Function
   4.8. The Finalize Function
5. Encoding (ct, tag) Tuples
6. Security Considerations
7. IANA Considerations
8. References
   8.1. Normative References
   8.2. Informative References
Appendix A. Test Vectors
   A.1. AESRound Test Vector
   A.2. AEGIS-128L Test Vectors
      A.2.1. Update Test Vector
      A.2.2. Test Vector 1
      A.2.3. Test Vector 2
      A.2.4. Test Vector 3
      A.2.5. Test Vector 4
      A.2.6. Test Vector 5
      A.2.7. Test Vector 6
      A.2.8. Test Vector 7
      A.2.9. Test Vector 8
      A.2.10. Test Vector 9
A.3. AEGIS-256 Test Vectors

A.3.1. Update Test Vector
A.3.2. Test Vector 1
A.3.3. Test Vector 2
A.3.4. Test Vector 3
A.3.5. Test Vector 4
A.3.6. Test Vector 5
A.3.7. Test Vector 6
A.3.8. Test Vector 7
A.3.9. Test Vector 8
A.3.10. Test Vector 9

Acknowledgments
Authors' Addresses

1. Introduction

This document describes the AEGIS-128L and AEGIS-256 authenticated encryption with associated data (AEAD) algorithms [AEGIS], which were chosen as additional finalists for high-performance applications in the Competition for Authenticated Encryption: Security, Applicability, and Robustness (CAESAR). Whilst AEGIS-128 was selected as a winner for this use case, AEGIS-128L has a better security margin alongside improved performance and AEGIS-256 uses a 256-bit key [LIMS21]. All variants of AEGIS are constructed from the AES encryption round function [FIPS-AES]. This document specifies:

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

The AEGIS cipher family offers performance that significantly exceeds that of AES-GCM with hardware support for parallelizable AES block encryption [AEGIS]. Similarly, software implementations 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. AEGIS-128L also allows 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 oracle) pair that verifies under multiple keys significantly reduces the number of required interactions with the oracle in order 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.

A key-committing AEAD scheme is more resistant against partitioning oracle attacks than non-committing AEAD schemes, making it significantly harder to find multiple keys that are valid for a given authentication tag. As of the time of writing, no research has been published claiming that AEGIS is not a key-committing AEAD scheme.

Finally, unlike most other AES-based AEAD constructions, such as Rocca and Tiaoxin, leaking the state does not leak the key.

Note that an earlier version of Hongjun Wu and Bart Preneel's paper introducing AEGIS specified AEGIS-128L and AEGIS-256 sporting differences with regards to the computation of the authentication tag and the number of rounds in Finalize() respectively. We follow the specification of [AEGIS] that is current at the time of writing, 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.

Primitives:

*|x|: the length of x in bits.

*a ^ 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 mod b: the remainder of the Euclidean division between a as the dividend and b as the divisor.

*LE64(x): the little-endian encoding of 64-bit integer x.

*ZeroPad(x, n): padding operation. Trailing zeros are concatenated to x until the total length is a multiple of n bits.
Truncate(x, n): truncation operation. The first n bits of x are kept.

Split(x, n): splitting operation. x is split into n-bit blocks, ignoring partial blocks.

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, False otherwise.

AEGIS internal functions:

Update(M0, M1): the state update function.

Init(key, nonce): the initialization 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, msg_len): 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: the constant 0x000101020305080d1522375990e97962 as an AES block.
3. The AEGIS-128L Algorithm

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

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

* K_LEN (key length) is 16 octets (128 bits).

* P_MAX (maximum length of the plaintext) is 2^{61} octets (2^{64} bits).

* A_MAX (maximum length of the associated data) is 2^{61} octets (2^{64} bits).

* N_MIN (minimum nonce length) = N_MAX (maximum nonce length) = 16 octets (128 bits).

* C_MAX (maximum ciphertext length) = P_MAX + tag length = 2^{61} + 16 octets (2^{64} + 128 bits).

Distinct associated data inputs, as described in [RFC5116], Section 3 shall 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 P_MAX).

*ad: the associated data to authenticate (length **MUST** be less than 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 xi in ad_blocks:
    Enc(xi)

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 message authentication tag MUST NOT be given as output. The decrypted message MUST be overwritten with zeros.

*The comparison of the input tag with the expected_tag MUST be done in constant time.

Inputs:

*ct: the ciphertext to be decrypted (length MUST be less than C_MAX).

*tag: the authentication tag.

*ad: the associated data to authenticate (length MUST be less than A_MAX).

*key: the encryption key.

*nnonce: the public nonce.

Outputs:

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

Steps:
3.3. The Init Function

The Init function constructs the initial state \{S0, ...S7\} using the given key and nonce.

Inputs:

*key: the encryption key.

*nonce: the nonce.

Defines:

*\{S0, ...S7\}: the initial state.

Steps:

```python
Init(key, nonce)

msg = {}

ad_blocks = Split(ZeroPad(ad, 256), 256)
for xi in ad_blocks:
    Enc(xi)

t_blocks = Split(ct, 256)
ctn = Tail(ctl, |ct| mod 256)
for ci in ct_blocks:
    msg = msg || Dec(ci)

if ctn is not empty:
    msg = msg || DecPartial(cnn)

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

if CtEq(tag, expected_tag) is False:
    erase msg
    return "verification failed" error
else:
    return msg
```
3.4. The Update Function

The Update function is the core of the AEGIS-128L algorithm. It updates the state \{S0, \ldots 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:

*\{S0, \ldots S7\}: the state.

Steps:

\begin{align*}
S'_0 &= \text{AESRound}(S7, S0 ^ M0) \\
S'_1 &= \text{AESRound}(S0, S1) \\
S'_2 &= \text{AESRound}(S1, S2) \\
S'_3 &= \text{AESRound}(S2, S3) \\
S'_4 &= \text{AESRound}(S3, S4 ^ M1) \\
S'_5 &= \text{AESRound}(S4, S5) \\
S'_6 &= \text{AESRound}(S5, S6) \\
S'_7 &= \text{AESRound}(S6, S7)
\end{align*}

S0 = S'_0 \\
S1 = S'_1 \\
S2 = S'_2 \\
S3 = S'_3 \\
S4 = S'_4 \\
S5 = S'_5 \\
S6 = S'_6 \\
S7 = S'_7
3.5. The Enc Function

Enc(xi)

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

Inputs:

*xi: the 256-bit input block.

Outputs:

*ci: the 256-bit encrypted block.

Steps:

\[ z_0 = S_6 \land S_1 \land (S_2 \land S_3) \]
\[ z_1 = S_2 \land S_5 \land (S_6 \land S_7) \]
\[ t_0, t_1 = \text{Split}(xi, 128) \]
\[ \text{out0} = t_0 \land z_0 \]
\[ \text{out1} = t_1 \land z_1 \]
\[ \text{Update}(t_0, t_1) \]
\[ ci = \text{out0} || \text{out1} \]

return ci

3.6. 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:

3.7. 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:

\[ z_0 = S_6 \oplus S_1 \oplus (S_2 \& S_3) \]
\[ z_1 = S_2 \oplus S_5 \oplus (S_6 \& S_7) \]

\[ t_0, t_1 = \text{Split}(ci, 128) \]
\[ \text{out0} = t_0 \oplus z_0 \]
\[ \text{out1} = t_1 \oplus z_1 \]

\[ \text{Update(out0, out1)} \]
\[ xi = \text{out0} \| \text{out1} \]

return \( xi \)

3.8. The Finalize Function

Finalize(ad_len, msg_len)

The Finalize function computes a 128-bit tag that authenticates the message and associated data.
Inputs:
* ad_len: the length of the associated data in bits.
* msg_len: the length of the message in bits.

Outputs:
* tag: the authentication tag.

Steps:
\[
t = S2 ^ (\text{LE64}(ad\_len) \ || \ \text{LE64}(msg\_len))
\]
Repeat(7, Update(t, t))
tag = S0 ^ S1 ^ S2 ^ S3 ^ S4 ^ S5 ^ S6
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 octets (256 bits).

* P_MAX (maximum length of the plaintext) is \(2^{61}\) octets (\(2^{64}\) bits).

* A_MAX (maximum length of the associated data) is \(2^{61}\) octets (\(2^{64}\) bits).

* N_MIN (minimum nonce length) = N_MAX (maximum nonce length) = 32 octets (256 bits).

* C_MAX (maximum ciphertext length) = P_MAX + tag length = \(2^{61} + 16\) octets (\(2^{64} + 128\) bits).

Distinct associated data inputs, as described in [RFC5116], Section 3 shall 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 P_MAX).

*ad: the associated data to authenticate (length **MUST** be less than A_MAX).

*key: the encryption key.

*nnonce: the public nonce.

Outputs:

*ct: the ciphertext.

*tag: the authentication tag.

Steps:

Init(key, nonce)

c = {}

ad_blocks = Split(ZeroPad(ad, 128), 128)
for xi in ad_blocks:
    Enc(xi)

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 message authentication tag MUST NOT be given as output. The decrypted message MUST be overwritten with zeros.

*The comparison of the input tag with the expected_tag MUST be done in constant time.

Inputs:

*ct: the ciphertext to be decrypted (length MUST be less than C_MAX).

*tag: the authentication tag.

*ad: the associated data to authenticate (length MUST be less than A_MAX).

*key: the encryption key.

*nnonce: the public nonce.

Outputs:

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

Steps:
The Init function constructs the initial state \{S_0, \ldots S_5\} using the given key and nonce.

**Inputs:**

*key*: the encryption key.

*nonce*: the nonce.

**Defines:**

*\{S_0, \ldots S_5\}: the initial state.*

**Steps:**

```python
Init(key, nonce)

msg = {}

ad_blocks = Split(ZeroPad(ad, 128), 128)
for xi in ad_blocks:
    Enc(xi)

cp_blocks = Split(ZeroPad(ct, 128), 128)
cn = Tail(ct, |ct| mod 128)
for ci in cp_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
    return "verification failed" error
else:
    return msg
```

4.3. The Init Function

The Init function constructs the initial state \{S_0, \ldots S_5\} using the given key and nonce.
4.4. The Update Function

Update(M)

The Update function is the core of the AEGIS-256 algorithm. It updates the state \{S0, ...S5\} using a 128-bit value.

Inputs:

*msg: the block to be absorbed.

Modifies:

*{S0, ...S5}: 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
4.5. The Enc Function

Enc(xi)

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

Inputs:

*xi: the input block.

Outputs:

*ci: the encrypted input block.

Steps:

\[ z = S1 \landdot{\lor} S4 \landdot{\lor} S5 \landdot{\lor} (S2 \landdot{\lor} S3) \]

Update(xi)

\[ ci = xi \landdot{\lor} z \]

return ci

4.6. The Dec Function

Dec(ci)

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

Inputs:

*ci: the encrypted input block.

Outputs:

*xi: the decrypted block.

Steps:

\[ z = S1 \landdot{\lor} S4 \landdot{\lor} S5 \landdot{\lor} (S2 \landdot{\lor} S3) \]

\[ xi = ci \landdot{\lor} z \]

Update(xi)

return xi

It returns the 128-bit block out.
4.7. 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:

\[ z = S1 \oplus S4 \oplus S5 \oplus (S2 \& S3) \]
\[ t = \text{ZeroPad}(cn, 128) \]
\[ \text{out} = t \oplus z \]
\[ xn = \text{Truncate}(\text{out}, |cn|) \]
\[ v = \text{ZeroPad}(xn, 128) \]
\[ \text{Update}(v) \]

return xn

4.8. The Finalize Function

Finalize(ad_len, msg_len)

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

Inputs:

*ad_len: the length of the associated data in bits.

*msg_len: the length of the message in bits.

Outputs:

*tag: the authentication tag.

Steps:
t = S3 ^ (LE64(ad_len) || LE64(msg_len))
Repeat(7, Update(t))
tag = S0 ^ S1 ^ S2 ^ S3 ^ S4 ^ S5
return tag

5. Encoding (ct, tag) Tuples

Applications MAY keep the ciphertext and the 128-bit 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

6. Security Considerations

AEGIS-256 offers 256-bit message security against plaintext and state recovery, whereas AEGIS-128L offers 128-bit security. Both have a 128-bit authentication tag, which implies that a given tag may verify under multiple keys. However, assuming AEGIS is key-committing, finding equivalent keys is expected to be significantly more difficult than for authentication schemes based on polynomial evaluation, such as GCM and Poly1305.

Under the assumption that the secret key is unknown to the attacker and the tag is not truncated, both AEGIS-128L and AEGIS-256 target 128-bit security against forgery attacks.

Both algorithms MUST be used in a nonce-respecting setting: for a given key, a nonce MUST only be used once. Failure to do so would immediately reveal the bitwise difference between two messages.

If tag verification fails, the decrypted message and wrong message authentication tag MUST NOT be given as output. As shown in the analysis of the (robustness of CAESAR candidates beyond their guarantees)[CRA18], even a partial leak of the plaintext without verification would facilitate chosen ciphertext attacks.

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, random nonces can safely encrypt up to $2^{48}$ messages using the same key with negligible collision probability.

With AEGIS-256, random nonces can be used with no practical limits.
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 also depend on the threat model in question.

Security analyses of AEGIS can be found in Chapter 4 of [AEGIS], in [Min14], in [ENP19], in [LIMS21], and in [JLD21].

7. IANA Considerations

IANA is requested to assign entries for AEAD_AEGIS128L and AEAD_AEGIS256 in the AEAD Registry with this document as reference.

8. References

8.1. Normative References


8.2. Informative References


Appendix A. Test Vectors

A.1. AESRound Test Vector

in : 000102030405060708090a0b0c0d0e0f
rk : 101112131415161718191a1b1c1d1e1f
out : 7a7b4e568782546a8c0477a3b813f43
A.2. AEGIS-128L Test Vectors

A.2.1. Update Test Vector

S0   : 9b7e60b24cc873ea894ecc07911049a3
S1   : 330be08f35300faa2ebf9a7b0d274658
S2   : 7bbd5bd2b049f7b9b51ccf26fbeb7756c
S3   : c35a00f55ec863886ec5e928f87db18
S4   : 9ebccafce87cab446396c4334592c91f
S5   : 58d83e31f256371e60fc6bb257114601
S6   : 16390b56a322c88568a176505bc915de
S7   : 640818f5b7dc0fbc2e72ae93457e39a

M0   : 033e6975b94816879e42917650955aa0
M1   : 033e6975b94816879e42917650955aa0

After Update:
S0   : 596ab773e4433ca0127c73f60536769d
S1   : 790394041a3d6ab697bde865014652d
S2   : 38cf49e4b65248acd533041b64dd611
S3   : 16d8e58748f437bfff1797f780337cee
S4   : 69761320f77d38b284cc9f335ac2f5a
S5   : a21746bb193a569e331e1aa985d0d729
S6   : 09d714e6f9177a8ed1cde7e3d259a6
S7   : 61279ba73167f0ab76f0a11bf203bdf

A.2.2. Test Vector 1

key   : 10010000000000000000000000000000
nonce: 10000200000000000000000000000000
ad    : 
msg   : 00000000000000000000000000000000
ct    : c1c0e58bd913006feba00f4b3cc3594e
tag   : abe0e0b8c24868a226a35d16bd00ae37a
A.2.3. Test Vector 2

key  : 10010000000000000000000000000000
nonce: 10000200000000000000000000000000
ad   :
msg  :
ct   :
tag  : c2b879a67def9d74e6c14f708bbcc9b4

A.2.4. Test Vector 3

key  : 10010000000000000000000000000000
nonce: 10000200000000000000000000000000
ad   : 0001020304050607
msg  : 000102030405060708090a0b0c0d0e0f
      101112131415161718191a1b1c1d1e1f
ct   : 79d94593d8c2119d7e8fd9b8fc77845c
      5c077a05b2528b6ac54b563aed8efe84
tag  : cc6f3372f6aa1bb82388d695c3962d9a

A.2.5. Test Vector 4

key  : 10010000000000000000000000000000
nonce: 10000200000000000000000000000000
ad   : 0001020304050607
msg  : 000102030405060708090a0b0c0d
ct   : 79d94593d8c2119d7e8fd9b8fc77
      5c04b3dba849b2701effbe32c7f0fab7
tag  : 5c04b3dba849b2701effbe32c7f0fab7
A.2.6. Test Vector 5

key : 10010000000000000000000000000000
nonce: 10000200000000000000000000000000
ad   : 000102030405060708090a0b0c0d0e0f
101112131415161718191a1b1c1d1e1f
20212223242526272829
msg : 101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f
3031323334353637
crct : 696b7f8a629f0df2657e30447a4538dc
7ede7f7a990f
tag : 7542a745733014f9474417b337399507

A.2.7. Test Vector 6

This test MUST return a "verification failed" error.

key : 10000200000000000000000000000000
nonce: 10010000000000000000000000000000
ad   : 0001020304050607
ctct : 79d94593d8c2119d7e8fd9b8fc77
tag : 5c04b3dba849b2701effbe32c7f0fab7

A.2.8. Test Vector 7

This test MUST return a "verification failed" error.

key : 10010000000000000000000000000000
nonce: 10000200000000000000000000000000
ad   : 0001020304050607
ctct : 79d94593d8c2119d7e8fd9b8fc78
tag : 5c04b3dba849b2701effbe32c7f0fab7

A.2.9. Test Vector 8

This test MUST return a "verification failed" error.
A.2.10. Test Vector 9

This test MUST return a "verification failed" error.

key : 100100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
nonce: 100002000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
ad    : 0001020304050608
ct     : 79d94593d8c2119d7e8fd9b8fc77
tag    : 5c04b3da849b2701effbe32c7f0fab7

A.3. AEGIS-256 Test Vectors

A.3.1. Update Test Vector

key : 100100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
nonce: 100002000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
ad    : 0001020304050607
ct     : 79d94593d8c2119d7e8fd9b8fc77
tag    : 6c04b3da849b2701effbe32c7f0fab8

S0 : 1fa1207ed76c86f2c4bb40e8b395b43e
S1 : b44c375e6c1e1978db64bcd12e9e332f
S2 : 0dab84bfa9f0226432ff630f233d4e5b
S3 : d7ef65c9b93e8ee60c75161407b066e7
S4 : a760bb3da073fbd92bdc24734b1f56fb
S5 : a828a186a964497ac6e753c5f55c73
M    : b165617ed04ab738af8b2612c6d18a1ec

After Update:
S0 : e6bc643bae82dfa3d991b1b323839dcd
S1 : 648578232ba0f2f0a3677f617dc052c3
S2 : ea788e0e572044a4659212dd007a789
S3 : 2f1498ae19b80da13fba698f088a8590
S4 : a54c2ee95e8c2a2c3da2ec743ae6b86
S5 : a3240fceb68e32d5d114df1b5363ab67
A.3.2. Test Vector 1

key : 10010000000000000000000000000000
      00000000000000000000000000000000
nonce: 10000200000000000000000000000000
       00000000000000000000000000000000
ad :
msg :
cryptograph

cryptograph
ct : 754fc3d8c973246dccc6d741412a4b236
tag : 3fe91994768b332ed7f570a19ec5896e

A.3.3. Test Vector 2

key : 10010000000000000000000000000000
      00000000000000000000000000000000
nonce: 10000200000000000000000000000000
       00000000000000000000000000000000
ad :
msg :
cryptograph

cryptograph
ct :
tag : e3def978a0f054afd1e761d7553afba3

A.3.4. Test Vector 3

key : 10010000000000000000000000000000
      00000000000000000000000000000000
nonce: 10000200000000000000000000000000
       00000000000000000000000000000000
ad : 0001020304050607
msg : 000102030405060708090a0b0c0d0e0f
      101112131415161718191a1b1c1d1e1f
cryptograph

cryptograph
cryptograph
ct : f373079ed84b2709faee373584585d60
      accd191db310ef5d8b11833df9dec711
tag : 8d86f91ee606e9ff26a01b64ccbdd91d
A.3.5. Test Vector 4

key : 10010000000000000000000000000000
00000000000000000000000000000000
nonce: 10000200000000000000000000000000
00000000000000000000000000000000
ad : 0001020304050607
msg : 000102030405060708090a0b0c0d
crypt : f373079ed84b2709faee37358458
tag : c60b9c2d33ceb058f96e6dd03c215652

A.3.6. Test Vector 5

key : 10010000000000000000000000000000
00000000000000000000000000000000
nonce: 10000200000000000000000000000000
00000000000000000000000000000000
ad : 000102030405060708090a0b0c0d0e0f
101112131415161718191a1b1c1d1e1f
20212223242526272829
msg : 101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f
3031323334353637
ct : 57754a7d09963e7c787583a2e7b859bb
24fa1e04d49fd550b2511a358e3bca25
2a9b1b8b30cc4a67
tag : ab8a7d53fd0e98d727accca94925e128

A.3.7. Test Vector 6

This test **MUST** return a "verification failed" error.
key : 100002000000000000000000000000000
     000000000000000000000000000000000
nonce: 10010000000000000000000000000000
     000000000000000000000000000000000
ad : 0001020304050607
ct : f373079ed84b2709faee37358458

tag : c60b9c2d33ceb058f96e6dd03c215652

A.3.8. Test Vector 7

This test MUST return a "verification failed" error.

key : 10010000000000000000000000000000
     000000000000000000000000000000000
nonce: 10000200000000000000000000000000
     000000000000000000000000000000000
ad : 0001020304050607
ct : f373079ed84b2709faee37358459

tag : c60b9c2d33ceb058f96e6dd03c215652

A.3.9. Test Vector 8

This test MUST return a "verification failed" error.

key : 10010000000000000000000000000000
     000000000000000000000000000000000
nonce: 10000200000000000000000000000000
     000000000000000000000000000000000
ad : 0001020304050607
ct : f373079ed84b2709faee37358459

tag : c60b9c2d33ceb058f96e6dd03c215652

A.3.10. Test Vector 9

This test MUST return a "verification failed" error.
The AEGIS authenticated encryption algorithm was invented by Hongjun Wu and Bart Preneel.

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

We would like to thank Eric Lagergren and Daniel Bleichenbacher for catching a broken test vector and Daniel Bleichenbacher for many helpful suggestions.

Authors' Addresses

Frank Denis
Fastly Inc.

Email: fde@00f.net

Fabio Enrico Renzo Scotoni
Individual Contributor

Email: fabio@esse.ch

Samuel Lucas
Individual Contributor

Email: samuel-lucas6@pm.me