

Network Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: 22 January 2026

F. Denis  
Fastly Inc.  
P. Pham  
L. Prabel  
S. Sun  
Huawei  
21 July 2025

The HiAE Authenticated Encryption Algorithm  
draft-pham-cfrg-hiae-02

## Abstract

This document describes HiAE, a high-throughput authenticated encryption algorithm designed for next-generation wireless systems (6G) and high-speed data transmission 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/hiae-aead/draft-pham-hiae>.

## 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 22 January 2026.

## 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. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

1. Introduction . . . . .	3
2. Conventions and Definitions . . . . .	4
3. The HiAE Algorithm . . . . .	6
3.1. Algorithm Parameters . . . . .	7
3.2. Authenticated Encryption . . . . .	7
3.3. Authenticated Decryption . . . . .	8
3.4. Core Functions . . . . .	10
3.4.1. The State Rotation Function . . . . .	10
3.4.2. The State Update Functions . . . . .	11
3.5. Initialization and Processing Functions . . . . .	13
3.5.1. The Init Function . . . . .	14
3.5.2. The Absorb Function . . . . .	14
3.5.3. The Enc Function . . . . .	15
3.5.4. The Dec Function . . . . .	15
3.5.5. The DecPartial Function . . . . .	16
3.5.6. The Finalize Function . . . . .	16
4. Encoding (ct, tag) Tuples . . . . .	17
5. Alternative Operating Modes . . . . .	17
5.1. HiAE as a Stream Cipher . . . . .	17
5.2. HiAE as a Message Authentication Code . . . . .	18
6. Security Considerations . . . . .	19
6.1. Classic Setting . . . . .	19
6.2. Quantum Setting . . . . .	20
6.3. Attack Considerations . . . . .	20
7. Implementation Considerations . . . . .	21
7.1. State Rotation Optimization . . . . .	21
7.1.1. Cycling Index Approach . . . . .	21
7.1.2. State Access Pattern . . . . .	21
7.1.3. Batch Processing Optimization . . . . .	22
7.2. Platform-Specific Optimizations . . . . .	22
7.2.1. ARM NEON Optimizations . . . . .	22
7.2.2. Intel AES-NI Optimizations . . . . .	24
7.3. Decryption Performance . . . . .	27
7.4. Security Considerations for Implementations . . . . .	27
7.5. Validation . . . . .	27
8. IANA Considerations . . . . .	27
9. References . . . . .	28

9.1. Normative References . . . . .	28
9.2. Informative References . . . . .	28
Appendix A. Test Vectors . . . . .	29
A.1. Test Vector 1 - Empty plaintext, no AD . . . . .	29
A.2. Test Vector 2 - Single block plaintext, no AD . . . . .	29
A.3. Test Vector 3 - Empty plaintext with AD . . . . .	30
A.4. Test Vector 4 - Rate-aligned plaintext (256 bytes) . . . . .	30
A.5. Test Vector 5 - Rate + 1 byte plaintext . . . . .	31
A.6. Test Vector 6 - Rate - 1 byte plaintext . . . . .	32
A.7. Test Vector 7 - Medium plaintext with AD . . . . .	33
A.8. Test Vector 8 - Single byte plaintext . . . . .	34
A.9. Test Vector 9 - Two blocks plaintext . . . . .	35
A.10. Test Vector 10 - All zeros plaintext . . . . .	35
Appendix B. Function-by-Function Example . . . . .	36
B.1. Initial Values for Example . . . . .	36
B.2. AESL Function Example . . . . .	36
B.3. Update Function Example . . . . .	37
B.3.1. Example: Update . . . . .	37
B.4. Initialize Function Example . . . . .	37
B.5. Enc Function Example . . . . .	38
B.6. Finalize Function Example . . . . .	39
B.7. Complete Encryption Example . . . . .	39
Authors' Addresses . . . . .	39

## 1. Introduction

The evolution of wireless networks toward 6G, alongside the growing demands of cloud service providers and CDN operators, requires cryptographic algorithms capable of delivering unprecedented throughput while maintaining strong security guarantees. Current high-performance authenticated encryption schemes achieve impressive speeds by leveraging platform-specific SIMD instructions, particularly AES-NI on x86 architectures [AES-NI]. Notable examples include AEGIS [I-D.irtf-cfrg-aegis-aead], SNOW-V [SNOW-V], and Rocca-S [ROCCA-S].

While these platform-specific optimizations deliver high performance on their target architectures, they create a significant performance disparity across different hardware platforms. These algorithms excel on x86 processors equipped with AES-NI but exhibit substantially degraded performance on ARM architectures that implement SIMD functionality through NEON instructions. This inconsistency poses a critical challenge for modern network deployments where ARM processors dominate mobile devices, edge computing nodes, and increasingly, data center environments.

The architectural differences between x86 and ARM extend beyond instruction set variations. They encompass fundamental distinctions in how AES round functions are implemented in hardware, pipeline structures, and memory subsystems. These differences mean that algorithms optimized for one architecture may inadvertently create bottlenecks on another, resulting in unpredictable performance characteristics across heterogeneous deployments.

The transition to 6G networks amplifies these challenges. Next-generation wireless systems will rely heavily on software-defined networking (SDN) and cloud radio access networks (Cloud RAN), requiring cryptographic algorithms that perform consistently across diverse hardware platforms. The stringent latency requirements and massive data rates anticipated for 6G, potentially exceeding 1 Tbps, demand encryption schemes that can leverage the full capabilities of both x86 and ARM architectures without compromise.

This document presents HiAE (High-throughput Authenticated Encryption), an authenticated encryption algorithm explicitly designed to address these cross-platform performance challenges. Through careful algorithmic design, HiAE delivers high performance on both x86 and ARM architectures by efficiently utilizing the capabilities of each platform without being overly dependent on architecture-specific features.

The remainder of this document is organized as follows: Section 2 establishes notation and conventions. Section 3 provides the complete specification of the HiAE algorithm, including its three operational modes. Sections 4-6 detail the specific use cases as an AEAD cipher, stream cipher, and MAC. Section 7 analyzes security considerations, while Section 8 discusses implementation aspects. The appendix provides comprehensive test vectors for validation.

## 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.

Basic operations:

- \* {}: 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 concatenation of  $a$  and  $b$ .
- \*  $a \bmod b$ : the remainder of the Euclidean division between  $a$  as the dividend and  $b$  as the divisor.

#### Data manipulation:

- \*  $LE64(x)$ : returns the little-endian encoding of unsigned 64-bit integer  $x$ .
- \*  $ZeroPad(x, n)$ : 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.
- \*  $Truncate(x, n)$ : returns the first  $n$  bits of  $x$ .
- \*  $Tail(x, n)$ : returns the last  $n$  bits of  $x$ .
- \*  $Split(x, n)$ : returns  $x$  split into  $n$ -bit blocks, ignoring partial blocks.

#### Cryptographic operations:

- \*  $AESL(x)$ : A single AES round function without key addition. Given a 128-bit AES state  $x$ , this function applies the following AES transformations in sequence:
  1.  $SubBytes$ : Apply the AES S-box to each byte
  2.  $ShiftRows$ : Cyclically shift the rows of the state
  3.  $MixColumns$ : Mix the columns of the state

Formally:  $AESL(x) = MixColumns(ShiftRows(SubBytes(x)))$

These transformations are as specified in Section 5 of [FIPS-AES]. This is NOT the full AES encryption algorithm. It is a single round without the  $AddRoundKey$  operation (equivalent to using a zero round key). A test vector for this function is provided in Appendix B.

#### Control flow and comparison:

- \*  $Repeat(n, F)$ :  $n$  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.

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: `{ 0x32, 0x43, 0xf6, 0xa8, 0x88, 0x5a, 0x30, 0x8d, 0x31, 0x31, 0x98, 0xa2, 0xe0, 0x37, 0x07, 0x34 }`.
- \* `C1`: an AES block built from the following bytes in hexadecimal format: `{ 0x4a, 0x40, 0x93, 0x82, 0x22, 0x99, 0xf3, 0x1d, 0x00, 0x82, 0xef, 0xa9, 0x8e, 0xc4, 0xe6, 0xc8 }`.

The constants `C0` and `C1` are domain separation constants derived from the fractional parts of  $\pi$  and  $e$ , respectively.

Input and output values:

- \* `key`: the encryption key (256 bits).
- \* `nonce`: the public nonce (128 bits).
- \* `ad`: the associated data.
- \* `msg`: the plaintext.
- \* `ct`: the ciphertext.
- \* `tag`: the authentication tag (128 bits).

### 3. The HiAE Algorithm

This section provides the complete specification of HiAE. The algorithm operates on a 2048-bit internal state organized as sixteen 128-bit blocks, combining AES round functions with an efficient update mechanism to achieve both high security and cross-platform performance.

### 3.1. Algorithm Parameters

HiAE maintains a 2048-bit state organized as sixteen 128-bit blocks denoted  $\{S_0, S_1, S_2, \dots, S_{15}\}$ . Each block  $S_i$  represents a 128-bit AES state that can be processed independently by AES round functions. This large state size provides security margins while enabling efficient parallel processing on modern architectures.

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) = 16 bytes (128 bits).
- \*  $C\_MAX$  (maximum ciphertext length) =  $P\_MAX$  + tag length =  $(2^{61} - 1) + 16$  or 32 bytes (in bits:  $(2^{64} - 8) + 128$  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.

### 3.2. 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

### 3.3. 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 fails.

Security:



- \* If tag verification fails, the decrypted message and incorrect 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(ct, 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
```

### 3.4. Core Functions

The following sections describe the fundamental operations that form the building blocks of HiAE. These functions manipulate the 2048-bit state to provide confusion, diffusion, and the absorption of input data.

#### 3.4.1. The State Rotation Function

Rol()

The Rol function provides diffusion by rotating the sixteen 128-bit blocks of the state one position to the left. This ensures that local changes propagate throughout the entire state over multiple rounds.

Modifies:

\* {S0, ...S15}: the state.

Steps:

```
t = S0
S0 = S1
S1 = S2
S2 = S3
S3 = S4
S4 = S5
S5 = S6
S6 = S7
S7 = S8
S8 = S9
S9 = S10
S10 = S11
S11 = S12
S12 = S13
S13 = S14
S14 = S15
S15 = t
```

### 3.4.2. The State Update Functions

The state update functions form the cryptographic core of HiAE. They combine the AESL transformation with XOR operations and state rotation to achieve both security and efficiency.

#### 3.4.2.1. The Update Function

Update(xi)

The Update function is the core of the HiAE algorithm. It updates the state {S0, ...S15} using a 128-bit value.

Inputs:

- \* xi: the 128-bit block to be absorbed.

Modifies:

- \* {S0, ...S15}: the state.

Steps:

```
t = AESL(S0 ^ S1) ^ xi
S0 = AESL(S13) ^ t
S3 = S3 ^ xi
S13 = S13 ^ xi
```

Rot()

#### 3.4.2.2. The UpdateEnc Function

UpdateEnc(mi)

The UpdateEnc function extends the basic Update function to provide encryption. It absorbs a plaintext block while simultaneously generating the corresponding ciphertext block through an additional XOR with state block S9.

Inputs:

\* mi: a 128-bit block to be encrypted.

Outputs:

\* ci: the encrypted 128-bit block.

Modifies:

\* {S0, ...S15}: the state.

Steps:

```
t = AESL(S0 ^ S1) ^ mi
ci = t ^ S9
S0 = AESL(S13) ^ t
S3 = S3 ^ mi
S13 = S13 ^ mi
```

Rol()

return ci

#### 3.4.2.3. The UpdateDec Function

UpdateDec(ci)

The UpdateDec function provides the inverse operation of UpdateEnc. It processes a ciphertext block to recover the plaintext while maintaining the same state update pattern, ensuring that encryption and decryption produce identical internal states.

Inputs:

\* ci: a 128-bit block to be decrypted.

Outputs:

\* `mi`: the decrypted 128-bit block.

Modifies:

\* `{S0, ...S15}`: the state.

Steps:

```
t = ci ^ S9
mi = AESL(S0 ^ S1) ^ t
S0 = AESL(S13) ^ t
S3 = S3 ^ mi
S13 = S13 ^ mi
```

`Rol()`

`return mi`

#### 3.4.2.4. The Diffuse Function

`Diffuse(x)`

The Diffuse function ensures full state mixing by performing 32 consecutive update operations. This function is critical for security during initialization and finalization phases, guaranteeing that every bit of the key and nonce influences the entire state, and that the authentication tag depends on all state bits.

Inputs:

\* `x`: a 128-bit input value.

Modifies:

\* `{S0, ...S15}`: the state.

Steps:

```
Repeat(32,
  Update(x)
)
```

#### 3.5. Initialization and Processing Functions

The following functions implement the high-level operations of HiAE: initialization, data absorption, encryption/decryption, and finalization.

### 3.5.1. The Init Function

Init(key, nonce)

The Init function constructs the initial state {S0, ...S15} from the encryption key and nonce. The initialization process carefully distributes key material across the state and applies the Diffuse function to ensure all state bits are cryptographically mixed before processing begins.

Inputs:

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

Defines:

- \* {S0, ...S15}: the initial state.

Steps:

k0, k1 = Split(key, 128)

```
S0 = C0
S1 = k1
S2 = nonce
S3 = C0
S4 = ZeroPad({ 0 }, 128)
S5 = nonce ^ k0
S6 = ZeroPad({ 0 }, 128)
S7 = C1
S8 = nonce ^ k1
S9 = ZeroPad({ 0 }, 128)
S10 = k1
S11 = C0
S12 = C1
S13 = k1
S14 = ZeroPad({ 0 }, 128)
S15 = C0 ^ C1
```

Diffuse(C0)

```
S9 = S9 ^ k0
S13 = S13 ^ k1
```

### 3.5.2. The Absorb Function

Absorb(ai)

The Absorb function processes associated data by incorporating 128-bit blocks into the internal state. This function is used exclusively for authenticated data that should influence the authentication tag but not produce ciphertext output.

Inputs:

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

Steps:

Update(ai)

### 3.5.3. The Enc Function

Enc(mi)

The Enc function encrypts a single 128-bit plaintext block. It serves as a simple wrapper around UpdateEnc, providing a clean interface for the block-by-block encryption process.

Inputs:

- \* mi: the 128-bit input block.

Outputs:

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

Steps:

```
ci = UpdateEnc(mi)
return ci
```

### 3.5.4. The Dec Function

Dec(ci)

The Dec function decrypts a single 128-bit ciphertext block. Like Enc, it provides a clean interface by wrapping the UpdateDec function.

Inputs:

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

Outputs:

- \* `mi`: the 128-bit decrypted block.

Steps:

```
mi = UpdateDec(ci)
return mi
```

### 3.5.5. The DecPartial Function

`DecPartial(cn)`

The `DecPartial` function handles the special case of decrypting a partial block at the end of a ciphertext. This function carefully reconstructs the keystream to decrypt blocks smaller than 128 bits while maintaining the same state evolution as encryption.

Inputs:

- \* `cn`: the encrypted input.

Outputs:

- \* `mn`: the decryption of `cn`.

Steps:

```
# Step 1: Recover the keystream that would encrypt a full zero block
ks = AESL(S0 ^ S1) ^ ZeroPad(cn) ^ S9

# Step 2: Construct a full 128-bit ciphertext block
# by appending the appropriate keystream bits
ci = cn || Tail(ks, 128 - |cn|)

# Step 3: Decrypt the full block using standard UpdateDec
mi = UpdateDec(ci)

# Step 4: Extract only the decrypted bytes corresponding to the partial input
mn = Truncate(mi, |cn|)

return mn
```

### 3.5.6. The Finalize Function

`Finalize(ad_len_bits, msg_len_bits)`



The Finalize function completes the authentication process by generating a 128-bit tag. It incorporates the lengths of both the associated data and message (each encoded as 8 bytes in little-endian format), applies the Diffuse function for final mixing, and combines all state blocks to produce the authentication tag.

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 = (LE64(ad_len_bits) || LE64(msg_len_bits))
Diffuse(t)

tag = S0 ^ S1 ^ S2 ^ S3 ^ S4 ^ S5 ^ S6 ^ S7 ^
      S8 ^ S9 ^ S10 ^ S11 ^ S12 ^ S13 ^ S14 ^ S15

return tag
```

#### 4. 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
```

#### 5. Alternative Operating Modes

While HiAE is primarily designed as an authenticated encryption algorithm, its flexible structure allows it to operate in two additional modes: as a stream cipher for keystream generation and as a message authentication code (MAC) for data authentication without encryption.

##### 5.1. HiAE as a Stream Cipher

The stream cipher mode of HiAE generates a keystream by encrypting an all-zero message.

```
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 HiAE key.
- \* nonce: the HiAE 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 message of len zero bits 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.

## 5.2. HiAE as a Message Authentication Code

In MAC mode, HiAE processes input data without generating ciphertext, producing only an authentication tag. This mode is useful when data authenticity is required without confidentiality.

```
Mac(data, key, nonce)
```

Security:

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

- \* HiAE-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.

#### Steps:

```
Init(key, nonce)
data_blocks = Split(ZeroPad(data, 128), 128)
for di in data_blocks:
    Absorb(di)
tag = Finalize(|data|, 0)
return tag
```

## 6. Security Considerations

### 6.1. Classic Setting

HiAE provides 256-bit security against key recovery and state recovery attacks, along with 128-bit security for integrity against forgery attempts.

It is important to note that the encryption security assumes the attacker cannot successfully forge messages through repeated trials [HiAE-Clarification].

Regarding keystream bias attacks, analysis shows that at least 150-bit security is guaranteed by HiAE.

Finally, HiAE is assumed to be secure against key-committing attacks, but it has not been proven to be secure in the everything-committing setting.

## 6.2. Quantum Setting

HiAE targets a security strength of 128 bits against key recovery attacks and forgery attacks in the quantum setting. Security is not claimed against online superposition queries to cryptographic oracle attacks, as such attacks are highly impractical in real-world applications.

## 6.3. Attack Considerations

HiAE is assumed to be secure against the following attacks:

1. Key-Recovery Attack: 256-bit security against key recovery attacks.
2. Differential Attack: 256-bit security against differential attacks in the initialization phase.
3. Forgery Attack: 128-bit security against forgery attacks.
4. Integral Attack: Secure against integral attacks.
5. State-Recovery Attack:
  - \* Guess-and-Determine Attack: The time complexity of the guess-and-determine attack cannot be lower than  $2^{256}$ .
  - \* Algebraic Attack: The system of equations to recover HiAE states cannot be solved with time complexity lower than  $2^{256}$ .
6. Linear Bias: At least 150-bit security against statistical attacks.
7. Key-Committing Attacks: Secure in the FROB, CMT1, and CMT2 models.
8. Everything-Committing Attacks: Security is not claimed in the CMT3 model.

The details of the cryptanalysis can be found in the paper [HiAE].

## 7. Implementation Considerations

HiAE is designed to balance the performance of XOR and AES instructions across both ARM and x86 architectures while being optimized to push performance to its limits. The algorithm's XAXX structure enables platform-specific optimizations by exploiting the fundamental differences in how ARM and Intel processors implement AES round functions.

### 7.1. State Rotation Optimization

Instead of performing physical rotations with the `Rol()` function, implementations can use a cycling index (offset) approach to avoid copying the entire 2048-bit state on every rotation. This optimization provides significant performance improvements across all platforms.

#### 7.1.1. Cycling Index Approach

The standard `Rol()` function requires copying all sixteen 128-bit blocks:

```
t = S0
S0 = S1
S1 = S2
...
S15 = t
```

This approach copies 2048 bits of data on every rotation. An optimized implementation can instead:

1. Keep the state blocks in a fixed array position
2. Maintain an offset variable tracking the logical position of S0
3. Map logical state block  $S_i$  to physical position  $(i + \text{offset}) \bmod 16$
4. Replace the entire `Rol()` operation with:  $\text{offset} = (\text{offset} + 1) \bmod 16$

#### 7.1.2. State Access Pattern

With this optimization, the logical-to-physical state block mapping becomes:

\* Logical S0 maps to physical position  $\text{offset} \bmod 16$

- \* Logical S3 maps to physical position  $(3 + \text{offset}) \bmod 16$
- \* Logical S9 maps to physical position  $(9 + \text{offset}) \bmod 16$
- \* Logical S13 maps to physical position  $(13 + \text{offset}) \bmod 16$

This approach is mathematically equivalent to the specification but eliminates the expensive memory operations associated with state rotation. Since `Rol()` is called in every `Update()`, `UpdateEnc()`, and `UpdateDec()` operation, this optimization provides substantial performance benefits during encryption and decryption operations.

#### 7.1.3. Batch Processing Optimization

Since the offset cycles back to zero every 16 operations ( $\text{offset} \bmod 16$ ), implementations may benefit from processing data in batches of 16 blocks. After processing 16 consecutive input blocks, the logical state mapping returns to its original configuration, which can simplify implementation and potentially enable further optimizations such as loop unrolling or vectorization of the batch processing logic.

When the offset is aligned to zero at the start of a batch, implementations can hardcode the specific offset values for each operation within the unrolled batch processing function, eliminating the need for modular arithmetic during the inner loop and providing additional performance benefits.

#### 7.2. Platform-Specific Optimizations

The key to HiAE's cross-platform efficiency lies in understanding how different architectures implement AES operations.

The following optimizations leverage architectural differences between ARM and Intel processors to maximize HiAE's performance while maintaining cryptographic correctness.

##### 7.2.1. ARM NEON Optimizations

ARM processors with NEON SIMD extensions can efficiently compute  $\text{AESL}(x^y)$  and (with SHA3 extensions) three-way XOR operations. For convenience, the following additional primitives can be defined:

- \* `XAESL(x,y)`: Computes  $\text{AESL}(x^y)$  in a single fused operation (assembly instruction `AESE AESMC`, or equivalently C intrinsic `vaesmcq_u8(vaeseq_u8(x,y))`)

\* XOR3(x,y,z): Computes  $x \oplus y \oplus z$  in a single three-way XOR instruction (assembly instruction EOR3, or equivalently C intrinsic `veor3q_u8(x,y,z)`)

#### 7.2.1.1. ARM-Optimized Update Function

Original implementation:

```
Update(xi)
    t = AESL(S0 ^ S1) ^ xi
    S0 = AESL(S13) ^ t
    S3 = S3 ^ xi
    S13 = S13 ^ xi
    Rol()
```

ARM-optimized implementation:

```
Update_ARM(xi)
    t = XAESL(S0, S1) ^ xi
    S0 = AESL(S13) ^ t
    S3 = S3 ^ xi
    S13 = S13 ^ xi
    Rol()
```

#### 7.2.1.2. ARM-Optimized UpdateEnc Function

Original implementation:

```
UpdateEnc(mi)
    t = AESL(S0 ^ S1) ^ mi
    ci = t ^ S9
    S0 = AESL(S13) ^ t
    S3 = S3 ^ mi
    S13 = S13 ^ mi
    Rol()
    return ci
```

ARM-optimized implementation:

```
UpdateEnc_ARM(mi)
    t = XAESL(S0, S1) ^ mi
    ci = t ^ S9
    S0 = AESL(S13) ^ t
    S3 = S3 ^ mi
    S13 = S13 ^ mi
    Rol()
    return ci
```

#### 7.2.1.3. ARM-Optimized DecPartial Function

Original implementation:

```
DecPartial(cn)
ks = AESL(S0 ^ S1) ^ ZeroPad(cn) ^ S9
ci = cn || Tail(ks, 128 - |cn|)
mi = UpdateDec(ci)
mn = Truncate(mi, |cn|)
return mn
```

ARM-optimized implementation:

```
DecPartial_ARM(cn)
ks = XOR3(XAESL(S0, S1), ZeroPad(cn), S9)
ci = cn || Tail(ks, 128 - |cn|)
mi = UpdateDec_ARM(ci)
mn = Truncate(mi, |cn|)
return mn
```

#### 7.2.1.4. ARM-Optimized UpdateDec Function

Original implementation:

```
UpdateDec(ci)
t = ci ^ S9
mi = AESL(S0 ^ S1) ^ t
S0 = AESL(S13) ^ t
S3 = S3 ^ mi
S13 = S13 ^ mi
Rol()
return mi
```

ARM-optimized implementation:

```
UpdateDec_ARM(ci)
t = ci ^ S9
mi = XAESL(S0, S1) ^ t
S0 = AESL(S13) ^ t
S3 = S3 ^ mi
S13 = S13 ^ mi
Rol()
return mi
```

#### 7.2.2. Intel AES-NI Optimizations

Intel processors with AES-NI can efficiently compute  $\text{AESL}(y)^z$  patterns. We can define the following additional function:



```
* AESLX(y,z): Computes AESL(y) ^ z using a single instruction
  (assembly instruction AESENC, or equivalently C intrinsic
   _mm_aesenc_si128(y, z))
```

#### 7.2.2.1. Intel-Optimized Update Function

Original implementation:

```
Update(xi)
  t = AESL(S0 ^ S1) ^ xi
  S0 = AESL(S13) ^ t
  S3 = S3 ^ xi
  S13 = S13 ^ xi
  Rol()
```

Intel-optimized implementation:

```
Update_Intel(xi)
  t = AESL(S0 ^ S1) ^ xi
  S0 = AESLX(S13, t)
  S3 = S3 ^ xi
  S13 = S13 ^ xi
  Rol()
```

#### 7.2.2.2. Intel-Optimized UpdateEnc Function

Original implementation:

```
UpdateEnc(mi)
  t = AESL(S0 ^ S1) ^ mi
  ci = t ^ S9
  S0 = AESL(S13) ^ t
  S3 = S3 ^ mi
  S13 = S13 ^ mi
  Rol()
  return ci
```

Intel-optimized implementation:

```
UpdateEnc_Intel(mi)
  t = AESL(S0 ^ S1) ^ mi
  ci = t ^ S9
  S0 = AESLX(S13, t)
  S3 = S3 ^ mi
  S13 = S13 ^ mi
  Rol()
  return ci
```

### 7.2.2.3. Intel-Optimized DecPartial Function

Original implementation:

```
DecPartial(cn)
ks = AESL(S0 ^ S1) ^ ZeroPad(cn) ^ S9
ci = cn || Tail(ks, 128 - |cn|)
mi = UpdateDec(ci)
mn = Truncate(mi, |cn|)
return mn
```

Intel-optimized implementation:

```
DecPartial_Intel(cn)
ks = AESL(S0 ^ S1) ^ ZeroPad(cn) ^ S9
ci = cn || Tail(ks, 128 - |cn|)
mi = UpdateDec_Intel(ci)
mn = Truncate(mi, |cn|)
return mn
```

### 7.2.2.4. Intel-Optimized UpdateDec Function

Original implementation:

```
UpdateDec(ci)
t = ci ^ S9
mi = AESL(S0 ^ S1) ^ t
S0 = AESL(S13) ^ t
S3 = S3 ^ mi
S13 = S13 ^ mi
Rol()
return mi
```

Intel-optimized implementation:

```
UpdateDec_Intel(ci)
t = ci ^ S9
mi = AESL(S0 ^ S1) ^ t
S0 = AESLX(S13, t)
S3 = S3 ^ mi
S13 = S13 ^ mi
Rol()
return mi
```

### 7.3. Decryption Performance

It is expected that HiAE decryption will be slower than encryption due to inherent data dependencies in the algorithm. While encryption can process keystream generation and state updates in parallel, decryption must first recover the plaintext before performing any state updates. This sequential dependency chain is a consequence of HiAE's design, which incorporates plaintext into the internal state to provide strong authentication properties.

### 7.4. Security Considerations for Implementations

The security of HiAE against timing and physical attacks is limited by the implementation of the underlying AESL function. Failure to implement AESL 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.

When implementing the platform-specific optimizations described above, care must be taken to ensure that:

- \* All operations complete in constant time
- \* No secret-dependent memory accesses occur
- \* The optimization does not introduce timing variations based on input data

### 7.5. Validation

A complete list of known implementations and integrations is available at <https://github.com/hiae-aead/draft-pham-hiae>, including reference implementations. A comprehensive comparison of HiAE's performance with other high-throughput authenticated encryption schemes on ARM and x86 architectures is also provided, demonstrating the effectiveness of these platform-specific optimizations.

## 8. IANA Considerations

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

Algorithm Name	ID
AEAD_HIAE	

Table 1

## 9. References

### 9.1. Normative References

- [FIPS-AES] National Institute of Standards and Technology (NIST), "Advanced encryption standard (AES)", Federal Information Processing Standards Publication 197, November 2001, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>>.
- [I-D.irtf-cfrg-aegis-aead] Denis, F. and S. Lucas, "The AEGIS Family of Authenticated Encryption Algorithms", Work in Progress, Internet-Draft, draft-irtf-cfrg-aegis-aead-16, 17 February 2025, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-aegis-aead-16>>.
- [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>>.
- [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>>.

### 9.2. Informative References

- [AES-NI] Gueron, S., "Intel Advanced Encryption Standard (AES) New Instructions Set", 2010, <<https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf>>.

- [HiAE] Chen, H., Huang, T., Pham, P., and S. Wu, "HiAE: A High-Throughput Authenticated Encryption Algorithm for Cross-Platform Efficiency", Cryptology ePrint Archive, Paper 2025/377, 2025, <<https://eprint.iacr.org/2025/377>>.
- [HiAE-Clarification] Pham, P., Prabel, L., and S. Wu, "HiAE Remains Secure in Its Intended Model: A Clarification of Claimed Attacks", Cryptology ePrint Archive, Paper 2025/1235, 2025, <<https://eprint.iacr.org/2025/1235>>.
- [NIST-LWC] National Institute of Standards and Technology (NIST), "NIST Lightweight Cryptography (LWC)", 2025, <<https://csrc.nist.gov/projects/lightweight-cryptography>>.
- [ROCCA-S] Anand, R., Banik, S., Caforio, A., Fukushima, K., Isobe, T., Kiyomoto, S., Liu, F., Nakano, Y., Sakamoto, K., and N. Takeuchi, "An Ultra-High Throughput AES-Based Authenticated Encryption Scheme for 6G: Design and Implementation", Cryptology and Network Security (CANS 2023), DOI 10.1007/978-3-031-50594-2\_12, 2024, <[https://doi.org/10.1007/978-3-031-50594-2\\_12](https://doi.org/10.1007/978-3-031-50594-2_12)>.
- [SNOW-V] Ekdahl, P., Johansson, T., Maximov, A., and J. Yang, "SNOW-V: an AES-based Stream Cipher for 5G and 6G Communication Systems", IACR Transactions on Symmetric Cryptology, 2019(3), DOI 10.13154/tosc.v2019.i3.1-42, 2019, <<https://doi.org/10.13154/tosc.v2019.i3.1-42>>.

## Appendix A. Test Vectors

### A.1. Test Vector 1 - Empty plaintext, no AD

key : 4b7a9c3ef8d2165a0b3e5f8c9d4a7b1e  
2c5f8a9d3b6e4c7f0a1d2e5b8c9f4a7d

nonce : a5b8c2d9e3f4a7b1c8d5e9f2a3b6c7d8

ad :

msg :

ct :

tag : e3b7c5993e804d7e1f95905fe8fa1d74

### A.2. Test Vector 2 - Single block plaintext, no AD

key : 2f8e4d7c3b9a5e1f8d2c6b4a9f3e7d5c  
1b8a6f4e3d2c9b5a8f7e6d4c3b2a1f9e

nonce : 7c3e9f5a1d8b4c6f2e9a5d7b3f8c1e4a

ad :

msg : 55f00fcc339669aa55f00fcc339669aa

ct : 66fc201d96ace3ca550326964c2fa950

tag : 2e4d9b3bf320283de63ea5547454878d

#### A.3. Test Vector 3 - Empty plaintext with AD

key : 9f3e7d5c4b8a2f1e9d8c7b6a5f4e3d2c  
1b0a9f8e7d6c5b4a3f2e1d0c9b8a7f6e

nonce : 3d8c7f2a5b9e4c1f8a6d3b7e5c2f9a4d

ad : 394a5b6c7d8e9fb0c1d2e3f405162738  
495a6b7c8d9eafc0d1e2f30415263748

msg :

ct :

tag : 531a4d1ed47bda55d01cc510512099e4

#### A.4. Test Vector 4 - Rate-aligned plaintext (256 bytes)

```
tag      : f330ae219d6739aba556fe94776b486b
```

key : 3e9d6c5b4a8f7e2d1c9b8a7f6e5d4c3b  
2a1f0e9d8c7b6a5f4e3d2c1b0a9f8e7d

nonce : 6f2e8a5c9b3d7f1e4a8c5b9d3f7e2a6c

ad : 6778899aabbccddeef00112233445566

msg : cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc339669aa55f00fcc339669aa55f00f  
cc  
ct : 5d2d2c7f1ff780687c65ed69c08805c2  
69652b55f5d1ef005f25300d1f644b57  
e500d5b0d75f9b025fee04cfd422c6c  
3c472e6967ac60f69ff730d4d308faed  
beac375ae88da8ab78d26e496a5226b5  
ffd7834a2f76ecc495a444ffa3db60d8  
ec3fb75c0fcaa74966elcaec294c8eb7  
a4895aa2ble3976eb6bed2f975ff218d  
c98f86f7c95996f03842cee71c6c1bc5  
f7b64374e101b32927ed95432e88f8e3  
8835f1981325dbcec412a4254e964c22  
cf82688ee5e471c23a3537de7e51c288  
92e32c565aa86ab708c70cf01f0d0ee9  
781251759893d55e60e0d70014cb3afb  
45e0821ba6e82e0f490ff2efef2f62c5  
7332c68c11e6ed71ef730b62c3e05edf  
f6  
tag : 1122dc5bedc7cad4e196f7227b7102f3

#### A.6. Test Vector 6 - Rate - 1 byte plaintext



key : 8a7f6e5d4c3b2a1f0e9d8c7b6a5f4e3d  
2c1b0a9f8e7d6c5b4a3f2e1d0c9b8a7f

nonce : 4d8b2f6a9c3e7f5d1b8a4c6e9f3d5b7a

ad :

msg : 00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000  
00000000000000000000000000000000

ct : 322970ad70b2af87676d57dd0b27866d  
8c4f0e251b5162b93672delab7aaf20c  
d91e7751a31e19762aeaa4f3811657a3  
06787ff4ebc06957c1f45b7fd284ef87  
f3a902922999895ff26fddbd5986eac5  
ef856f6ae270136315c698ec7fe5a618  
8aa1847c00a3a870044e8d37e22b1bca  
b3e493d8ae984c7646f2536032a40910  
b6c0f317b916d5789189268c00ef4493  
bcb5fb0135974fa9bec299d473fdbf76  
f44107ec56b5941404fd4b3352576c31  
3169662f1664bd5bccf210a710aa6665  
fb3ec4fa3b4c648411fd09d4cada31b8  
947fdd486de45a4e4a33c151364e23be  
6b3fc14f0855b0518e733d5ea9051165  
25286bb2d6a46ac8ef73144e2046f9

tag : 7eb4461a035fe51eaf4a1829605e6227

#### A.7. Test Vector 7 - Medium plaintext with AD

key : 5d9c3b7a8f2e6d4c1b9a8f7e6d5c4b3a  
2f1e0d9c8b7a6f5e4d3c2b1a0f9e8d7c

nonce : 8c5a7d3f9b1e6c4a2f8d5b9e3c7a1f6d

ad : 95a6b7c8d9eafb0c1d2e3f5061728394  
a5b6c7d8e9fa0b1c2d3e4f60718293a4  
b5c6d7e8f90a1b2c3d4e5f708192a3b4  
c5d6e7f8091a2b3c4d5e6f8091a2b3c4

msg : 32e14453e7a776781d4c4e2c3b23bca2  
441ee4213bc3df25021b5106c22c98e8  
a7b310142252c8dcff70a91d55cdc910  
3c1eccd9b5309ef21793a664e0d4b63c  
83530dcd1a6ad0feda6ff19153e9ee62  
0325c1cb979d7b32e54f41da3af1c169  
a24c47c1f6673e115f0cb73e8c507f15  
eedf155261962f2d175c9ba3832f4933  
fb330d28ad6aae787f12788706f45c92  
e72aea146959d2d4fa01869f7d072a7b  
f43b2e75265e1a000dde451b64658919  
e93143d2781955fb4ca2a38076ac9eb4  
9adc2b92b05f0ec7

ct : ca3b18f0ffb25e4e1a6108abedcfc931  
841804c22a132a701d2f0b5eb845a380  
8028e9e1e0978795776c57a0415971cf  
e87abc72171a24fd11f3c331d1efe306  
e4ca1d8ede6e79cbd531020502d38026  
20d9453ffdd5633fe98ff1d12b057edd  
bd4d99ee6cabf4c8d2c9b4c7ee0d219b  
3b4145e3c63acde6c45f6d65e08dd06e  
f9dd2dde090f1f7579a5657720f348ae  
5761a8df321f20ad711a2c703b1c3f20  
0e4004da409daaa138f3c20f8f77c89c  
b6f46df671f25c75a6a7838a5d792d18  
a59c202fab564f0f

tag : 74ba4c28296f09101db59c37c4759bcf

#### A.8. Test Vector 8 - Single byte plaintext

key : 7b6a5f4e3d2c1b0a9f8e7d6c5b4a3f2e  
1d0c9b8a7f6e5d4c3b2a1f0e9d8c7b6a

nonce : 2e7c9f5d3b8a4c6f1e9b5d7a3f8c2e4a

ad :

msg : ff

ct : 51

tag : 588535eb70c53ba5cce0d215194cb1c9

#### A.9. Test Vector 9 - Two blocks plaintext

key : 4c8b7a9f3e5d2c6b1a8f9e7d6c5b4a3f  
2e1d0c9b8a7f6e5d4c3b2a1f0e9d8c7b

nonce : 7e3c9a5f1d8b4e6c2a9f5d7b3e8c1a4f

ad : c3d4e5f60718293a4b5c6d7e8fa0b1c2  
d3e4f5061728394a5b6c7d8e9fb0c1d2  
e3f405162738495a6b7c8d9eafc0d1e2

msg : aa55f00fcc339669aa55f00fcc339669  
aa55f00fcc339669aa55f00fcc339669

ct : 03694107097ff7ea0b1eac408fabb60a  
cd89df4d0288fa9063309e5e323bf78f

tag : 2a3144f369a893c3d756f262067e5e59

#### A.10. Test Vector 10 - All zeros plaintext

```
key   : 9e8d7c6b5a4f3e2d1c0b9a8f7e6d5c4b
       3a2f1e0d9c8b7a6f5e4d3c2b1a0f9e8d

nonce : 5f9d3b7e2c8a4f6d1b9e5c7a3d8f2b6e

ad    : daebfc0d1e2f405162738495a6b7c8d9

msg   : 00000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000
       00000000000000000000000000000000

ct    : eef78d00c4de4c557d5c769e499af7b9
       8e5ad36cdaf1ff775a8629d82751e97e
       8f98caa0773fe81ee40266f0d52ddbbe
       f621504863bf39552682b29748f8c244
       5c176cd63865732141edc59073cff90e
       5996a23a763f8dd058a6a91ada1d8f83
       2f5e600b39f799a698228b68d20cd189
       e5e423b253a44c78060435050698ccae

tag   : 59970b0b35a7822f3b88b63396c2da98
```

## Appendix B. Function-by-Function Example

This appendix provides step-by-step examples of HiAE internal functions for implementers. All values are in hexadecimal.

### B.1. Initial Values for Example

```
Key:    0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
Nonce:  00112233445566778899aabbccddeeff
AD:     48656c6c6f (5 bytes: "Hello")
Msg:    576f726c64 (5 bytes: "World")
```

### B.2. AESL Function Example

The AESL function performs a single AES encryption round with a zero round key.

Input Block: 00112233445566778899aabbccddeeff

Output Block: 6379e6d9f467fb76ad063cf4d2eb8aa3

### B.3. Update Function Example

The Update function modifies the internal state with an input block.

#### B.3.1. Example: Update

Initial state: (16 AES blocks after initialization)

```
S0: 7cc0a8cc3b5f3fbce67c59a0c8e64f23
S1: 0123456789abcdef0123456789abcdef
S2: 00112233445566778899aabbccddeeff
S3: 7cc0a8cc3b5f3fbce67c59a0c8e64f23
S4: 00000000000000000000000000000000
S5: 01224466ccfeaa88899abcfe01224466
S6: 00000000000000000000000000000000
S7: d3d0e4c0f95c1d6b3e3dc8c7a6f90001
S8: 00112233ccddee00112233ccddee00
S9: 00000000000000000000000000000000
S10: 0123456789abcdef0123456789abcdef
S11: 7cc0a8cc3b5f3fbce67c59a0c8e64f23
S12: d3d0e4c0f95c1d6b3e3dc8c7a6f90001
S13: 0123456789abcdef0123456789abcdef
S14: 00000000000000000000000000000000
S15: af104c0cc2f3228758410ff26f1f4e22
```

Input block: 48656c6c000000000000000000000000

After applying the Update function:

```
S0: 8a5b7f2c4d9e1a3f6b8c2d5e9f3a7b4c
S3: 344582a03b5f3fbce67c59a0c8e64f23
S13: 494608236b9a1a30123456789abcdef
(other blocks unchanged)
```

### B.4. Initialize Function Example

The Initialize function sets up the initial state from key and nonce.

Key: 0123456789abcdef0123456789abcdef  
0123456789abcdef0123456789abcdef  
Nonce: 00112233445566778899aabbccddeeff

Initial State (before diffusion rounds):

S0: 7cc0a8cc3b5f3fbce67c59a0c8e64f23  
S1: 0123456789abcdef0123456789abcdef  
S2: 00112233445566778899aabbccddeeff  
S3: 7cc0a8cc3b5f3fbce67c59a0c8e64f23  
S4: 00000000000000000000000000000000  
S5: 01224466ccfeaa88899abcfe01224466  
S6: 00000000000000000000000000000000  
S7: d3d0e4c0f95c1d6b3e3dc8c7a6f90001  
S8: 00112233ccddeeff00112233ccddeeff  
S9: 00000000000000000000000000000000  
S10: 0123456789abcdef0123456789abcdef  
S11: 7cc0a8cc3b5f3fbce67c59a0c8e64f23  
S12: d3d0e4c0f95c1d6b3e3dc8c7a6f90001  
S13: 0123456789abcdef0123456789abcdef  
S14: 00000000000000000000000000000000  
S15: af104c0cc2f3228758410ff26f1f4e22

After diffusion and final XORs:

S0: 3f8a2b5c9d4e7a1b6c2d9e5f3a8b4c7d  
S1: e2c8d5f6a3b7914e7d8c2b6a5f9e3d4c  
S2: 7a4b6e9d2c5f8b3ald4e7c9b6a5f3e2d  
S3: d5f8c2b6a9e3b7d14c5a8f2e6d9b3c7a  
S4: 1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e  
S5: a8b7c6d5e4f3029184736251a0b9c8d7  
S6: 5e6d7c8b9a0f1e2d3c4b5a6978879695  
S7: c2d3e4f506172839a4b5c6d7e8f90102  
S8: 9a8b7c6d5e4f30214132243546576879  
S9: 0123456789abcdef0123456789abcdef  
S10: 7b8c9d0e1f2a3b4c5d6e7f8091a2b3c4  
S11: e5f607182930a4b5c6d7e8f901234567  
S12: 3c4d5e6f708192a3b4c5d6e7f8091a2b  
S13: ccddeeff00112233445566778899aabb  
S14: a7b8c9d0e1f20314253647589a6b7c8d  
S15: 2a3b4c5d6e7f809102143526a7b8c9d0

#### B.5. Enc Function Example

The Enc function encrypts a single message block.

```
State: (after processing AD "Hello")
Message Block: 576f726c640000000000000000000000

Ciphertext Block: 8b3a5f2c9d4e7a1b6c2d9e5f3a8b4c7d

Updated State:
  S0:  modified based on updateEnc
  S3:  XORed with message block
  S13: XORed with message block
```

## B.6. Finalize Function Example

The `Finalize` function produces the authentication tag.

```

State: (after processing all AD and message)
AD length:  5 bytes
Msg length: 5 bytes

Length encoding block: 280000000000000000 280000000000000000
                        (40 bits)           (40 bits)

```

After diffusion rounds with length block:

$$\text{Tag} = S_0 \wedge S_1 \wedge \dots \wedge S_{15} = \text{c4d8f3a2b5e9617d4c8a2f5b3e9d7a16}$$

### B.7. Complete Encryption Example

```
Key:      0123456789abcdef0123456789abcdef
          0123456789abcdef0123456789abcdef
Nonce:    00112233445566778899aabbccddeeff
AD:       48656c6c6f ("Hello")
Plaintext: 576f726c64 ("World")

Ciphertext: 8b3a5f2c9d
Tag:       c4d8f3a2b5e9617d4c8a2f5b3e9d7a16
```

### Authors' Addresses

Frank Denis  
Fastly Inc.  
Email: [fd@00f.net](mailto:fd@00f.net)

Phuong Pham  
Huawei  
Email: [pham.phuong@huawei.com](mailto:pham.phuong@huawei.com)

Lucas Prabel  
Huawei  
Email: [lucas.prabel@huawei.com](mailto:lucas.prabel@huawei.com)

Shuzhou Sun  
Huawei  
Email: [sunshuzhou@huawei.com](mailto:sunshuzhou@huawei.com)