Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 17 May 2026

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

13 November 2025

The HiAE Authenticated Encryption Algorithm
draft-pham-cfrg-hiae-05

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 17 May 2026.

Copyright Notice

Table of Contents

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.

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.

* { 0 }: a single zero byte (8 zero bits).

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

* a ^ b: the bitwise exclusive OR 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.

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 the Test Vectors section.

  While Intel AES-NI and ARM NEON provide instructions with similar parameters and descriptions (such as _mm_aesenc_si128 on Intel and vaesmcq_u8(vaeseq_u8(...)) on ARM), these instructions are not functionally equivalent. The architectural differences in how AES round functions are implemented require platform-specific optimization strategies, as detailed in the Implementation Considerations section.

Control flow and comparison:

* Repeat(n, F): n sequential evaluations of 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 }.

* ZERO: an AES block of all zeros (128 zero bits).

The constants C0 and C1 are domain separation constants derived from the fractional parts of π 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 {S0, S1, S2, ..., S15}. Each block Si 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 bytes ((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 mi in msg_blocks:
    ct = ct || Enc(mi)

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

return (ct, 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
```

Rol()

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(x0, x1)

The Diffuse function ensures full state mixing by performing 32 consecutive update operations, alternating between two input values. 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:

* x0: a 128-bit input value for even-numbered updates (updates 0, 2, 4, ..., 30).

* x1: a 128-bit input value for odd-numbered updates (updates 1, 3, 5, ..., 31).

Modifies:

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

Steps:

```
Repeat(16,
  Update(x0)
  Update(x1)
)
```

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 = k0
 S2 = C0
 S3 = nonce
 S4 = ZERO
 S5 = k0
 S6 = ZERO
 S7 = C1
 S8 = k1
 S9 = ZERO
S10 = nonce ^ k1
S11 = C0
S12 = C1
S13 = k1
S14 = ZERO
S15 = C0 ^ C1

Diffuse(k0, 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, 128) ^ 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, 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.

Security:

* When the nonce is fixed (including when using the default all-zeros nonce), a unique key MUST be used for each invocation to maintain security.

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.

Note: Implementations of the Encrypt and Decrypt functions are not required to support MAC-only mode. This is an optional feature that can be implemented separately.

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.

Usage constraints:

* Tag truncation is not allowed. Implementations MUST use the full 128-bit authentication tag.

* A single key MUST NOT be used to protect more than 2^64 messages.

* For a given key, the nonce MUST NOT be reused under any circumstances (as specified in the Authenticated Encryption section).

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 at the birthday bound security level (64 bits), but it is not secure in the context-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 oracles, 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, CMT-1, and CMT-2 models at the birthday bound security level.

8. Context-Committing Attacks: Security is not claimed in the CMT-3 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 Si to physical position (i + offset) mod 16

4. Replace the entire Rol() operation with: offset = (offset + 1) mod 16

7.1.2. State Access Pattern

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

* Logical S0 maps to physical position offset mod 16

* Logical S3 maps to physical position (3 + offset) mod 16

* Logical S9 maps to physical position (9 + offset) mod 16

* Logical S13 maps to physical position (13 + offset) mod 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 (offset mod 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 AESL(x^y) and (with SHA3 extensions) three-way XOR operations. For convenience, the following additional primitives can be defined:

* XAESL(x, y): Computes 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^y^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 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
```

7.2.1.4. ARM-Optimized DecPartial Function

Original implementation:

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

7.2.2. Intel AES-NI Optimizations

Intel processors with AES-NI can efficiently compute 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 = AESLX(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)
 ci = AESLX(S0 ^ S1, mi ^ S9)
  t = ci ^ S9
 S0 = AESLX(S13, t)
 S3 =  S3 ^ mi
S13 = S13 ^ mi
Rol()
return ci
```

7.2.2.3. 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 = AESLX(S0 ^ S1, t)
 S0 = AESLX(S13, t)
 S3 =  S3 ^ mi
S13 = S13 ^ mi
Rol()
return mi
```

7.2.2.4. Intel-Optimized DecPartial Function

Original implementation:

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

Intel-optimized implementation:

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

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, Update 1, DOI 10.6028/NIST.FIPS.197-upd1, May 2023, <https://doi.org/10.6028/NIST.FIPS.197-upd1>.

[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-18, 5 October 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-aegis-aead-18>.

[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] Chen, H., Huang, T., Pham, P., 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>.

[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", Computer Security ESORICS 2023, DOI 10.1007/978-3-031-50594-2_12, 2024, <https://doi.org/10.1007/978-3-031-50594-2_12>.

[SNOW-V] Ekdahl, P., Johansson, T., Maximov, A., and J. Yang, "A new SNOW stream cipher called SNOW-V", 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   : a25049aa37deea054de461d10ce7840b
```

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

```
   key   : 2f8e4d7c3b9a5e1f8d2c6b4a9f3e7d5c
           1b8a6f4e3d2c9b5a8f7e6d4c3b2a1f9e

   nonce : 7c3e9f5a1d8b4c6f2e9a5d7b3f8c1e4a

   ad    :

   msg   : 55f00fcc339669aa55f00fcc339669aa

   ct    : af9bd1865daa6fc351652589abf70bff

   tag   : ed9e2edc8241c3184fc08972bd8e9952
```

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

```
   key   : 9f3e7d5c4b8a2f1e9d8c7b6a5f4e3d2c
           1b0a9f8e7d6c5b4a3f2e1d0c9b8a7f6e

   nonce : 3d8c7f2a5b9e4c1f8a6d3b7e5c2f9a4d

   ad    : 394a5b6c7d8e9fb0c1d2e3f405162738
           495a6b7c8d9eafc0d1e2f30415263748

   msg   :

   ct    :

   tag   : 7e19c04f68f5af633bf67529cfb5e5f4
```

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

```
   key   : 6c8f2d5a9e3b7f4c1d8a5e9f3c7b2d6a
           4f8e1c9b5d3a7e2f4c8b6d9a1e5f3c7d

   nonce : 9a5c7e3f1b8d4a6c2e9f5b7d3a8c1e6f

   ad    :

   msg   : ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff
           ffffffffffffffffffffffffffffffff

   ct    : cf9f118ccc3ae98998ddaae1a5d1f9a1
           69e4ca3e732baf7178cdd9a353057166
           8fe403e77111eac3da34bf2f25719cea
           09445cc58197b1c6ac490626724e7372
           707cfb60cdba8262f0e33a1ef8adda1f
           2e390a80c58e5c055d9be9bbccdc06ad
           af74f1dcaa372204bf42e5e0e0ac5943
           7a353978298837023f79fac6daa1fe8f
           6bcaaaf060ae2e37ed7b7da0577a7643
           5f0403b8e277b6bc2ea99682f2d0d577
           77fec6d901e0d8fc7cf46bb97336812a
           2d8cfd39053993288cce2c077fce0c6c
           00e99cf919281b261acf86b058164f10
           1d9c24e8f40b4fa0ed60955eeeb4e33f
           f1087519c13db8e287199a7df7e94b0d
           368da9ccf3d2ecebfa46f860348f8e3c

   tag   : 4f42c3042cba3973153673156309dd69
```

A.5. Test Vector 5 - Rate + 1 byte plaintext

```
   key   : 3e9d6c5b4a8f7e2d1c9b8a7f6e5d4c3b
           2a1f0e9d8c7b6a5f4e3d2c1b0a9f8e7d

   nonce : 6f2e8a5c9b3d7f1e4a8c5b9d3f7e2a6c

   ad    : 6778899aabbccddeef00112233445566

   msg   : cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc339669aa55f00fcc339669aa55f00f
           cc

   ct    : 522e4cd9b0881809d80e149bb4ed8b8a
           dd70b7257afca6c2bc38e4da11e290cf
           cabd9dd1d4ed8c514482f444f903e42e
           c21a7a605ee37f95a504ec667fabec40
           66eb4521cdaf9c4eb7b62d659ab0a936
           3b145f1120c1b2e589ab9cb893d01be0
           d22182fc7de4932f1e8652b50e4a0d48
           c49a8a1232b201e2e535cd95c15cf0ee
           389b75e372653579c72c4dd1906fd81c
           2b9fc2483fab8b4df5a09d59753b5bd4
           1334be2e5085e349b6e5aac0c555a0a8
           3e94eab974052131f8d451c9d85389a3
           6126f93464e6f93119c6b1bf15b4c0a9
           e6c9beb52e82c846c472f87c15ac49e9
           9d59248ba7e6b97ca04327769d6b8c1f
           751d95dba709fb335183c21476836ea1
           ab

   tag   : 61bac11505dd8bbf55e7fbb7489de7b0
```

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
           000000000000000000000000000000

   ct    : 2ba49be54eb675efe446fd597721d4cd
           ca6e01f1a51728a859d8f206d13cdb08
           ba4f0fe78fbbd6885964ed54e9beceed
           1ff306642c4761e67efa7a2620e57128
           15b5e9f066b42e879cd62e7adc2821e5
           08311b88a6ee14bedcbac7ce339994c0
           09bbbadf9444748e4ab9a91acbbc7301
           742dab74aa1be6847ad8e9f08c170359
           b87e0ccd480812aaaf847aff03c2e858
           1c55848c2b50f6c6608540fe82627a2c
           0f5ee37fbe9cdeab5f6c9799702bd303
           2bf733e2108d03247cd20edaa2c322e5
           bf086bfecc4ac97b61096f016c57d5d0
           1c24d398cefd5ae8131c1f51f172ce9c
           6d3b8395d396dcbd70b4af790018796b
           31f0b0ad6198f86e5e1f26e9258492

   tag   : 221dd1b69afb4e0c149e0a058e471a4a
```

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    : 1d8d56867870574d1c4ac114620c6a2a
           bb44680fe321dd116601e2c92540f85a
           11c41dcac9814397b8f37b812cd52c93
           2db6ecbaa247c3e14f228bd792334570
           2fc43ad1eb1b8086e2c3c57bb602971c
           29772a35dfb1c45c66f81633e67fdc8d
           8005457ddbe4179312abab981049eb0a
           0a555b9fa01378878d7349111e2446fd
           e89ce64022d032cbf0cf2672e00d7999
           ed8b631c1b9bee547cbe464673464a4b
           80e8f72ad2b91a40fdcee5357980c090
           b34ab5e732e2a7df7613131ee42e42ec
           6ae9b05ac5683ebe

   tag   : e93686b266c481196d44536eb51b5f2d
```

A.8. Test Vector 8 - Single byte plaintext

```
   key   : 7b6a5f4e3d2c1b0a9f8e7d6c5b4a3f2e
           1d0c9b8a7f6e5d4c3b2a1f0e9d8c7b6a

   nonce : 2e7c9f5d3b8a4c6f1e9b5d7a3f8c2e4a

   ad    :

   msg   : ff

   ct    : 21

   tag   : 3cf9020bd1cc59cc5f2f6ce19f7cbf68
```

A.9. Test Vector 9 - Two blocks plaintext

```
   key   : 4c8b7a9f3e5d2c6b1a8f9e7d6c5b4a3f
           2e1d0c9b8a7f6e5d4c3b2a1f0e9d8c7b

   nonce : 7e3c9a5f1d8b4e6c2a9f5d7b3e8c1a4f

   ad    : c3d4e5f60718293a4b5c6d7e8fa0b1c2
           d3e4f5061728394a5b6c7d8e9fb0c1d2
           e3f405162738495a6b7c8d9eafc0d1e2

   msg   : aa55f00fcc339669aa55f00fcc339669
           aa55f00fcc339669aa55f00fcc339669

   ct    : c2e199ac8c23ce6e3778e7fd0b4f8f75
           2badd4b67be0cdc3f6c98ae5f6fb0d25

   tag   : 7aea3fbce699ceb1d0737e0483217745
```

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    : fc7f1142f681399099c5008980e73420
           65b4e62a9b9cb301bdf441d3282b6aa9
           3bd7cd735ef77755b4109f86b7c09083
           8e7b05f08ef4947946155a03ff483095
           152ef3dec8bdddae3990d00d41d5ee6c
           90dcf65dbed4b7ebbe9bb4ef096e1238
           d388bf15faacdb7a68be19dddc8a5b74
           216f4442bfa32d1dfccdc9c4020baec9

   tag   : ad0b841c3d145a6ee86dc7b67338f113
```

A.11. Test Vector 11 - Partial-block AD (padding demonstration)

This test vector specifically demonstrates the padding behavior when associated data length is not a multiple of the block size (128 bits). The AD is 13 bytes (104 bits), which requires 3 bytes (24 bits) of zero padding to reach the next block boundary.

```
   key   : 1122334455667788112233445566778811
           2233445566778811223344556677881 1

   nonce : aabbccddeeff0011aabbccddeeff0011

   ad    : 0102030405060708090a0b0c0d
           (13 bytes - padded to 16 bytes with zeros)

   msg   : 48656c6c6f576f726c64
           (10 bytes)

   ct    : 1fb0e0348c6a3a917133

   tag   : 7d292173b55ba02dae56ac1224b7e775
```

Appendix B. Function-by-Function Example

This appendix provides step-by-step examples of HiAE internal functions for implementers. All values are in hexadecimal. The examples use the following test data:

* Key: 0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef

* Nonce: 00112233445566778899aabbccddeeff

* AD: 48656c6c6f (5 bytes: “Hello”)

* Msg: 576f726c64 (5 bytes: “World”)

B.1. AESL Function Example

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

Input Block: 00112233445566778899aabbccddeeff

Output Block: 6379e6d9f467fb76ad063cf4d2eb8aa3

B.2. Initialize Function Example

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

```
   Key:   0123456789abcdef0123456789abcdef
          0123456789abcdef0123456789abcdef
   Nonce: 00112233445566778899aabbccddeeff

   Initial state (before diffusion rounds):
     S0:  3243f6a8885a308d313198a2e0370734
     S1:  0123456789abcdef0123456789abcdef
     S2:  3243f6a8885a308d313198a2e0370734
     S3:  00112233445566778899aabbccddeeff
     S4:  00000000000000000000000000000000
     S5:  0123456789abcdef0123456789abcdef
     S6:  00000000000000000000000000000000
     S7:  4a4093822299f31d0082efa98ec4e6c8
     S8:  0123456789abcdef0123456789abcdef
     S9:  00000000000000000000000000000000
     S10: 01326754cdfeab9889baefdc45762310
     S11: 3243f6a8885a308d313198a2e0370734
     S12: 4a4093822299f31d0082efa98ec4e6c8
     S13: 0123456789abcdef0123456789abcdef
     S14: 00000000000000000000000000000000
     S15: 7803652aaac3c39031b3770b6ef3e1fc

   State after Init (after diffusion):
     S0:  2a622bda4d229c9fc4b7d1a25399e321
     S1:  a04980e6407654f2760e59be74c05f9c
     S2:  308859e5787ab6c1705abbaecebfc316
     S3:  04782e5069799e9f3e325836a2ff3bcc
     S4:  007a12163596057152dc58a0f78cef2c
     S5:  7ce304cf04de20f8116cef51eea19e24
     S6:  c554da7a91164fc30e38c76ec038e66a
     S7:  3edcba33d2a5d02a4b598c5ffa003513
     S8:  50db6b4f18b7e282b8918311685abc18
     S9:  4253eebe1b72fec70ac3ec478cf9f2f2
     S10: 6bc09743dc45ba191c18a0d275ef9a8f
     S11: 251524aba97200f5b31eeeecb0f0a1c1
     S12: e1a99d3dd105e14085d6a0200f1d0c35
     S13: a4730f9d0f36ad7c67880342deed5310
     S14: 7d6ca34814e6e065c8cb4fa11ba2f8c4
     S15: 2b545cefe484f2e0ba5fb6359faffeca
```

B.3. Update Function Example

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

```
   Initial state (after initialization):
     S0:  2a622bda4d229c9fc4b7d1a25399e321
     S1:  a04980e6407654f2760e59be74c05f9c
     S2:  308859e5787ab6c1705abbaecebfc316
     S3:  04782e5069799e9f3e325836a2ff3bcc
     S4:  007a12163596057152dc58a0f78cef2c
     S5:  7ce304cf04de20f8116cef51eea19e24
     S6:  c554da7a91164fc30e38c76ec038e66a
     S7:  3edcba33d2a5d02a4b598c5ffa003513
     S8:  50db6b4f18b7e282b8918311685abc18
     S9:  4253eebe1b72fec70ac3ec478cf9f2f2
     S10: 6bc09743dc45ba191c18a0d275ef9a8f
     S11: 251524aba97200f5b31eeeecb0f0a1c1
     S12: e1a99d3dd105e14085d6a0200f1d0c35
     S13: a4730f9d0f36ad7c67880342deed5310
     S14: 7d6ca34814e6e065c8cb4fa11ba2f8c4
     S15: 2b545cefe484f2e0ba5fb6359faffeca

   Input block: 48656c6c6f000000000000000000000000

   After applying the Update function:
     S0:  a04980e6407654f2760e59be74c05f9c
     S1:  308859e5787ab6c1705abbaecebfc316
     S2:  4c1d423c06799e9f3e325836a2ff3bcc
     S3:  007a12163596057152dc58a0f78cef2c
     S4:  7ce304cf04de20f8116cef51eea19e24
     S5:  c554da7a91164fc30e38c76ec038e66a
     S6:  3edcba33d2a5d02a4b598c5ffa003513
     S7:  50db6b4f18b7e282b8918311685abc18
     S8:  4253eebe1b72fec70ac3ec478cf9f2f2
     S9:  6bc09743dc45ba191c18a0d275ef9a8f
     S10: 251524aba97200f5b31eeeecb0f0a1c1
     S11: e1a99d3dd105e14085d6a0200f1d0c35
     S12: ec1663f16036ad7c67880342deed5310
     S13: 7d6ca34814e6e065c8cb4fa11ba2f8c4
     S14: 2b545cefe484f2e0ba5fb6359faffeca
     S15: 4672d0d4a6a8fc93fe85701ff61a9e10
```

B.4. Enc Function Example

The Enc function encrypts a single message block.

State (after processing AD "Hello"):
S0: a04980e6407654f2760e59be74c05f9c
S1: 308859e5787ab6c1705abbaecebfc316
S2: 4c1d423c06799e9f3e325836a2ff3bcc
S3: 007a12163596057152dc58a0f78cef2c
S4: 7ce304cf04de20f8116cef51eea19e24
S5: c554da7a91164fc30e38c76ec038e66a
S6: 3edcba33d2a5d02a4b598c5ffa003513
S7: 50db6b4f18b7e282b8918311685abc18
S8: 4253eebe1b72fec70ac3ec478cf9f2f2
S9: 6bc09743dc45ba191c18a0d275ef9a8f
S10: 251524aba97200f5b31eeeecb0f0a1c1
S11: e1a99d3dd105e14085d6a0200f1d0c35
S12: ec1663f16036ad7c67880342deed5310
S13: 7d6ca34814e6e065c8cb4fa11ba2f8c4
S14: 2b545cefe484f2e0ba5fb6359faffeca
S15: 4672d0d4a6a8fc93fe85701ff61a9e10

Message Block: 576f726c640000000000000000000000

Ciphertext Block: 03e5d2157300b718595429195c9278e7

Updated State after Enc:
S0: 308859e5787ab6c1705abbaecebfc316
S1: 4c1d423c06799e9f3e325836a2ff3bcc
S2: 5715607a5196057152dc58a0f78cef2c
S3: 7ce304cf04de20f8116cef51eea19e24
S4: c554da7a91164fc30e38c76ec038e66a
S5: 3edcba33d2a5d02a4b598c5ffa003513
S6: 50db6b4f18b7e282b8918311685abc18
S7: 4253eebe1b72fec70ac3ec478cf9f2f2
S8: 6bc09743dc45ba191c18a0d275ef9a8f
S9: 251524aba97200f5b31eeeecb0f0a1c1
S10: e1a99d3dd105e14085d6a0200f1d0c35
S11: ec1663f16036ad7c67880342deed5310
S12: 2a03d12470e6e065c8cb4fa11ba2f8c4
S13: 2b545cefe484f2e0ba5fb6359faffeca
S14: 4672d0d4a6a8fc93fe85701ff61a9e10
S15: 9c56037e72109cee878398424f789257

B.5. Finalize Function Example

The Finalize function produces the authentication tag.

```
   State (after processing the AD and message):
     S0:  308859e5787ab6c1705abbaecebfc316
     S1:  4c1d423c06799e9f3e325836a2ff3bcc
     S2:  5715607a5196057152dc58a0f78cef2c
     S3:  7ce304cf04de20f8116cef51eea19e24
     S4:  c554da7a91164fc30e38c76ec038e66a
     S5:  3edcba33d2a5d02a4b598c5ffa003513
     S6:  50db6b4f18b7e282b8918311685abc18
     S7:  4253eebe1b72fec70ac3ec478cf9f2f2
     S8:  6bc09743dc45ba191c18a0d275ef9a8f
     S9:  251524aba97200f5b31eeeecb0f0a1c1
     S10: e1a99d3dd105e14085d6a0200f1d0c35
     S11: ec1663f16036ad7c67880342deed5310
     S12: 2a03d12470e6e065c8cb4fa11ba2f8c4
     S13: 2b545cefe484f2e0ba5fb6359faffeca
     S14: 4672d0d4a6a8fc93fe85701ff61a9e10
     S15: 9c56037e72109cee878398424f789257

   AD length:  5 bytes (40 bits)
   Msg length: 5 bytes (40 bits)

   Length encoding block: 2800000000000000 2800000000000000
                          (40 bits)        (40 bits)

   Tag = S0 ^ S1 ^ ... ^ S15 = 45178cd06ef0a8bed8e9082fe49ec818
```

B.6. Complete Encryption Example

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

   Ciphertext: 03e5d21573
   Tag:        45178cd06ef0a8bed8e9082fe49ec818
```

Acknowledgments

The HiAE algorithm leverages the AES permutation invented by Joan Daemen and Vincent Rijmen.

We would like to thank Samuel Lucas for his review of the draft.

Authors' Addresses

Frank Denis
Fastly Inc.

Email: fd@00f.net

Phuong Pham
Huawei
Email: pham.phuong@huawei.com

Lucas Prabel
Huawei
Email: lucas.prabel@huawei.com

Shuzhou Sun
Huawei
Email: sunshuzhou@huawei.com