Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 26 September 2026

N. Sullivan
Cryptography Consulting LLC

25 March 2026

Random-Access Authenticated Encryption
draft-sullivan-cfrg-raae-00

Abstract

This document defines Random-Access Authenticated Encryption (raAE), a mechanism for encrypting segmented content that supports decryption of individual segments without processing others, selective re-encryption of individual segments in place, and content-level integrity via an accumulator binding all segment authentication tags. raAE is parameterized over an Authenticated Encryption with Associated Data (AEAD) algorithm, a key derivation function (KDF), segment size, epoch length, and nonce generation mode.

raAE does not define wire formats, IANA registries, or key management; applications instantiate it with a chosen parameter set and protocol identifier. This document also defines "raAE-v1", a concrete profile with test vectors, supporting five AEAD algorithms: AES-256-GCM, ChaCha20-Poly1305, AES-256-GCM-SIV, AEGIS-256, and AEGIS-256X2, each with HKDF-SHA-256 and 65536-octet segments.

Discussion Venues

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

Discussion of this document takes place on the Crypto Forum Research Group mailing list (cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cfrg.

Source for this draft and an issue tracker can be found at https://github.com/grittygrease/draft-sullivan-cfrg-raae.

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/.

Copyright Notice

Table of Contents

1. Introduction

Consider a 1 tebibyte (TiB) encrypted backup stored as 2^24 segments of 64 KiB each. A user changes one segment. Today that service must either re-encrypt the entire file or treat each segment as an independent Authenticated Encryption with Associated Data (AEAD) message and give up cross-segment integrity. STREAM ([STREAM]) does not help here; it chains nonces across segments, so reading segment 1000 requires verifying segments 0 through 999 first.

Random-Access Authenticated Encryption (raAE) solves this. It encrypts each segment independently, supports in-place re-encryption, and binds all segment tags into a single accumulator for content-level integrity.

The mechanism works in layers. Start with a 32-octet random Content Encryption Key (CEK) and a per-content salt. From these, derive a commitment (lets the reader reject a wrong key before decrypting anything), a payload key, an accumulator key, and optionally a nonce base. For each segment, derive a per-segment key from the payload key (either directly or through an epoch-key layer that bounds nonce reuse), generate a nonce, and encrypt with an AEAD. Take the AEAD tag from each segment, feed it through a keyed pseudorandom function (PRF), and XOR all the outputs together into the accumulator.

The hierarchy is deterministic: CEK plus salt plus segment index is enough to derive any key.

This document defines "raAE-v1", a concrete profile supporting AES-256-GCM, ChaCha20-Poly1305, AES-256-GCM-SIV, AEGIS-256, and AEGIS-256X2, all with HKDF-SHA-256, and includes test vectors for each nonce generation mode. raAE does not define wire formats, IANA code points, or key management; consuming protocols handle those.

Section 2 fixes notation. Section 3 defines the mechanism. Section 4 states the domain separation requirements any concrete KDF framing must meet. Section 5 gives the security analysis. Section 6 defines raAE-v1.

1.1. Related Work

STREAM ([STREAM]) encrypts sequential data by chaining a nonce from one segment to the next: segment N's nonce depends on segment N-1's ciphertext, so decryption is inherently sequential. raAE replaces this chain with independent per-segment nonces (random, derived, or plaintext-bound), which is what makes random-access reads possible. Both constructions authenticate each segment individually; raAE adds a content-level accumulator that STREAM does not need because STREAM's chain already binds segment order.

FLOE ([FLOE]) formalized the raAE security notion as ra-ROR (random-access real-or-random), a simulation-based definition that captures confidentiality and authenticity when segments can be encrypted and decrypted in any order. ra-ROR is strictly stronger than nonce-based online authenticated encryption (nOAE2): a scheme can be nOAE2-secure (secure for sequential encryption with random-access decryption) but insecure when encryption order is also arbitrary.

FLOE proved that its construction (epoch key rotation, per-segment AEAD, and a hash-based integrity tag) satisfies ra-ROR, and defined ra-CMT (random-access context commitment) as a commitment notion independent of the underlying AEAD's commitment properties. raAE-v1 draws on FLOE's epoch key structure but uses a different integrity mechanism: a XOR-of-PRF accumulator that supports O(1) incremental update on segment rewrites. The informal security argument in Section 5 follows FLOE's reduction structure, adapted for the XOR-of-PRF accumulator.

1.2. Applicability

Use raAE when the content is too large for a single AEAD call and the application needs to read or write individual segments. The format must store per-segment nonces and tags.

For small messages, a single AEAD call is simpler and sufficient. For sequential streaming, STREAM ([STREAM]) gives stronger ordering guarantees with less machinery. For formats where per-segment metadata is prohibitively expensive, derived nonce mode removes the nonce overhead but restricts the AEAD to Misuse-Resistant Authenticated Encryption (MRAE, [RFC9771]) algorithms.

2. Conventions and Terminology

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.

A Segment is a fixed-size unit of plaintext. Each segment is individually encrypted and authenticated. The final segment of a message may be shorter than the nominal segment size.

The Content Encryption Key (CEK) is a 32-octet uniform random master key from which all other keys for a given message are derived.

The protocol_id is an application-chosen octet string that binds all KDF outputs to a specific protocol version and application context. Different applications MUST use different protocol_id values even when using the same underlying AEAD and KDF algorithms.

Other terms (epoch, epoch_key, payload_key, accumulator, commitment, and related values) are defined in Section 3 where they first appear.

2.1. Notation

I2OSP(n, w): The w-octet big-endian encoding of the non-negative integer n, as defined in [RFC8017].

uint64(n): Shorthand for I2OSP(n, 8).

uint8(n): Shorthand for I2OSP(n, 1).

||: Octet string concatenation.

lp16(x): The 2-octet big-endian length of x followed by x: I2OSP(len(x), 2) || x.

Encode(x1, ..., xn): Injective length-prefixed encoding: lp16(x1) || ... || lp16(xn). Each argument MUST be at most 65535 octets. The concrete framing in Section 6.2 specifies how Encode is used in KDF calls.

XOR: Bitwise exclusive-or of two octet strings of equal length.

x[a:b]: Octets a through b-1 inclusive of x (zero-based, half-open interval).

Nk: Key size in octets for the chosen AEAD algorithm.

Nn: Nonce size in octets for the chosen AEAD algorithm.

Nh: Hash or PRF output size in octets for the chosen KDF.

Nt: Authentication tag size in octets for the chosen AEAD algorithm.

3. The raAE Framework

The key hierarchy is shown below. A CEK and per-content salt feed the payload schedule, which derives four values. The payload key feeds per-segment keys (optionally through epoch keys), each segment produces an AEAD tag, and all tags feed the accumulator.

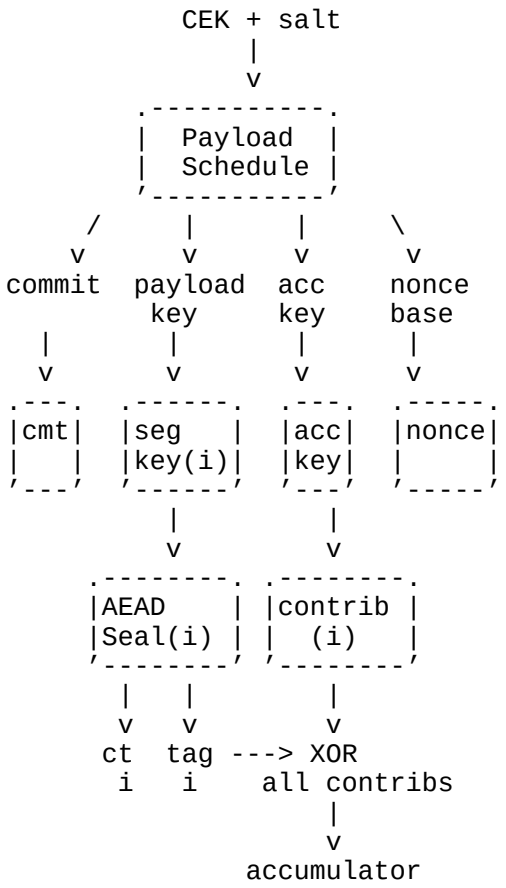

Figure 1: raAE Key Hierarchy

3.1. Parameters

A raAE parameter set is a tuple (AEAD, KDF, protocol_id, segment_size, epoch_length, nonce_mode, aad_label).

AEAD: An authenticated encryption with associated data algorithm satisfying the interface of [RFC5116], with Nk-octet keys, Nn-octet nonces, and Nt-octet authentication tags.

KDF: A key derivation function supporting the abstract interface defined in Section 3.3. Two-stage KDFs (HKDF) use Extract and Expand; single-stage KDFs (extendable-output function, XOF) use a direct derivation call.

protocol_id: An application-chosen octet string that binds all

derived values to a specific protocol version. Applications MUST NOT use the profile identifier "raAE-v1" as their protocol_id; the test vectors in this document use it for illustration only. A real application would use a value identifying itself (for example, "myapp-backup-v1").

segment_size: A positive integer that is a power of two and at least 4096 octets. Power-of-two sizes align to memory page boundaries and simplify offset arithmetic. All segments except possibly the last have exactly this many octets of plaintext. The segment size MUST remain within the per-invocation plaintext limit of the chosen AEAD (for example, 2^36 - 32 octets for AES-GCM per [NIST-SP-800-38D]). A zero-length plaintext produces a single segment of length 0 with is_final = 1.

epoch_length: A non-negative integer r, or absent (meaning the parameter is omitted from the tuple entirely, not set to a sentinel value). When present, each epoch covers exactly 2^r consecutive segments starting at a multiple of 2^r. When absent, all segments use the payload key directly and epoch derivation is skipped: segment_key(i) = payload_key for all i. When epoch_length = 0, each segment gets a unique derived key. The value r MUST be in the range 0 to 63 inclusive. Implementations MUST reject r >= 64.

nonce_mode: One of the strings "random", "derived", or "plaintext-bound". See Section 3.4. When nonce_mode is "derived", epoch_length MUST be absent. When nonce_mode is "random" or "plaintext-bound" and the AEAD uses 96-bit nonces, epoch_length MUST be present to bound per-key nonce usage.

aad_label: A protocol-chosen ASCII string used as the first argument to Encode in segment_aad construction. See Section 3.6.

Two sizes are determined by the chosen algorithms rather than by the parameter set directly.

Nh: The hash or PRF output size for the KDF, in octets. This is also the size of each accumulator contribution and of the acc_key.

commitment_length: A positive integer, defaulting to Nh, specifying the length of the key commitment prefix in octets.

3.2. Components

An encrypted message is built from three input values, four payload schedule outputs, per-segment values derived from those outputs, and a content-level integrity value.

3.2.1. Inputs

Three values are provided by the caller or constructed from parameters.

CEK: A 32-octet uniform random Content Encryption Key generated fresh for each message.

salt: A per-content random value of Nh octets, generated fresh for each message. The salt separates payload schedule outputs across messages even when the CEK is reused. An application that writes a new message MUST generate a fresh salt. Reusing a salt with the same CEK produces identical payload schedule outputs, causing nonce reuse in derived mode and correlated nonces in random mode. An application extending an existing message MUST reuse the existing salt.

payload_info: A framed encoding of the parameter set context. When epoch_length is absent:

```
payload_info = Encode(AEAD_id, segment_size_str,
                      KDF_id, salt)
```

When epoch_length is present:

```
payload_info = Encode(AEAD_id, segment_size_str,
                      KDF_id, epoch_length_str, salt)
```

The exact encoding is defined in Section 6.2.

3.2.2. Payload Schedule Outputs

Four values are derived from the CEK and payload_info. They are computed once per message and fixed for its lifetime.

commitment: A commitment_length-octet value derived from the CEK and payload_info. Stored at the start of the content to catch key or parameter mismatches before decryption begins.

payload_key: An Nk-octet key derived from the CEK and payload_info. Used directly as the segment key when epoch_length is absent, or as the root for epoch key derivation.

acc_key: An Nh-octet key derived from the CEK and payload_info. Used to compute per-segment accumulator contributions.

nonce_base: An Nn-octet value derived from the CEK and payload_info, used in derived nonce mode only.

3.2.3. Per-Segment Values

Each segment i has a set of derived values produced from the payload schedule outputs and the segment index.

epoch_key(i): An Nk-octet key for the epoch containing segment i, derived from the payload_key and the epoch index. Present only when epoch_length is specified.

segment_key(i): The AEAD key for segment i. Equal to epoch_key(i) when epoch_length is present; equal to payload_key otherwise.

nonce(i): The Nn-octet nonce for AEAD operations on segment i. Derived per the chosen nonce_mode.

segment_aad(i, is_final): Per-segment additional authenticated data that binds the segment to its index and whether it is the final segment.

tag(i): The Nt-octet AEAD authentication tag for segment i, produced by the AEAD Seal operation.

3.2.4. Content-Level Integrity

Two values provide integrity across the full set of segments.

contrib(i): An Nh-octet accumulator contribution for segment i, derived from acc_key, the segment index, and tag(i).

accumulator: An Nh-octet content-level integrity value equal to the aggregation of all contrib(i) values across all segments.

3.3. Key Derivation Structure

A message’s entire key hierarchy grows from two random values: the 32-octet CEK and the per-content salt. All other keys are derived deterministically from these two values, so there is no per-segment key state to manage or synchronize. The abstract interface below exists so that raAE can support both two-stage (HKDF) and single-stage (XOF) KDF families; Section 6 defines the only current concrete framing. All key material flows through this interface:

KDF(protocol_id, label, ikm, info, L)

* protocol_id: binds the output to a specific protocol version and application context.

* label: a unique ASCII string identifying the derivation role.

* ikm: input keying material, provided as a single octet string or an ordered list of octet strings.

* info: context information, provided as a single octet string or an ordered list of octet strings.

* L: the requested output length in octets.

The KDF MUST satisfy these requirements.

Outputs MUST be computationally indistinguishable from random given an unknown ikm, under the assumption that the underlying hash or PRF is a secure pseudorandom function.

The framing of (protocol_id, label, ikm, info, L) into the octet string input of the underlying primitive MUST be injective: each lp16 field is self-delimiting, so the concatenation can be uniquely parsed regardless of the number of arguments. Distinct input tuples MUST NOT produce the same primitive input. The concrete framing requirements are specified in Section 4.

The requested output length L MUST be committed in the primitive input. This prevents attacks where an adversary truncates a longer output to obtain a valid shorter one.

The protocol_id MUST appear in the primitive input. This prevents cross-protocol reuse of derived values.

Each label used within a protocol version MUST be distinct across all derivation roles.

The label appears in both the Extract and Expand phases of the two-stage KDF as a defensive redundancy measure: if either phase is weak in isolation, the label binding in the other phase preserves domain separation. Implementations MAY amortize the Extract computation internally when deriving multiple outputs from the same protocol_id, label, and ikm.

3.3.1. Payload Schedule Derivation

Four values are derived from the CEK and per-content salt. The commitment allows early rejection of a wrong key. The payload key encrypts segments (directly, or via epoch keys). The accumulator key authenticates the set of segment tags. The nonce base (derived-mode only) seeds deterministic per-segment nonces.

```
commitment  = KDF(protocol_id, "commit",
                  [CEK], [payload_info],
                  commitment_length)
payload_key = KDF(protocol_id, "payload_key",
                  [CEK], [payload_info], Nk)
acc_key     = KDF(protocol_id, "acc_key",
                  [CEK], [payload_info], Nh)
```

For derived nonce mode, a nonce base is also derived:

```
nonce_base  = KDF(protocol_id, "nonce_base",
                  [CEK], [payload_info], Nn)
```

3.3.2. Epoch Key Derivation

AES-256-GCM uses 96-bit nonces. The birthday bound for nonce collisions under a single key becomes a practical concern at around 2^32 encryptions. For rewritable content, this limit can be reached through repeated modifications to the same segments. Epoch keys partition the nonce space: each epoch key covers at most 2^r segment positions (initial writes plus rewrites), so the per-key invocation count is bounded regardless of the content’s total size.

When epoch_length = r is specified, the segment key for segment i is an epoch key derived from the payload key:

```
segment_key(i):
  epoch_index = i >> r
  return KDF(protocol_id, "epoch_key",
             [payload_key],
             [uint64(epoch_index)], Nk)
```

When epoch_length is absent, segment_key(i) = payload_key for all segment indices i. Epoch keys implement the parallel external rekeying pattern of [RFC8645], adapted for random-access patterns.

3.3.3. Key Schedule Design Rationale

The single-CEK design separates key agreement from content encryption. Different recipients can hold different wrappings of the same CEK; adding a recipient requires only a new key-wrapping operation, not re-encryption of the payload. The per-content salt makes the payload schedule unique even when a CEK is reused across messages, which matters for applications that derive CEKs from passwords or group keys. A separate salt lets the encryptor choose it at write time without coordinating with recipients; a counter in the CEK would require that coordination.

Epoch keys exist because 96-bit-nonce AEADs hit their birthday bound quickly: without epoch partitioning, large content with many rewrites exhausts the nonce space under a single key. Rekeying the entire content would defeat the random-access property, so epoch keys bound the per-key invocation count without requiring a full re-encryption.

Labels separate derivation roles: commitment and payload_key share the same inputs but different labels, making them independent under the PRF assumption. Once CEK and salt are chosen the hierarchy is fixed, with no mutable state to synchronize across writers.

3.4. Nonce Generation

Three nonce modes trade off AEAD flexibility, storage cost, and random number generator (RNG) trust. The chosen mode is part of the parameter set and MUST be consistent across all segments of a message.

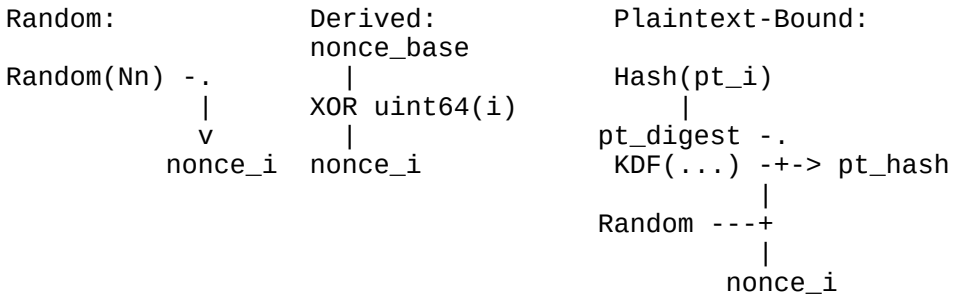

Figure 2: Nonce Generation Modes

3.4.1. Random Stored Nonces (nonce_mode = "random")

Each segment gets an independent random nonce:

nonce(i) = Random(Nn)

The nonce is generated fresh on initial write and on each rewrite of that segment. Each segment’s nonce is stored in per-segment metadata and MUST be accessible to any reader that decrypts individual segments.

This mode requires a functioning cryptographically secure pseudorandom number generator (CSPRNG). The nonce collision probability across independently generated per-segment nonces follows the standard birthday bound; see Section 5.4.

3.4.2. Derived Nonces (nonce_mode = "derived")

For MRAE AEADs, nonces are derived deterministically from the payload schedule. The AEAD nonce size Nn MUST be at least 8 octets. The XOR operation is applied to the last 8 octets of nonce_base, with the remaining octets unchanged:

```
nonce(i) = nonce_base[0:Nn-8]
           || (nonce_base[Nn-8:Nn] XOR uint64(i))
```

No per-segment nonce storage is required.

Nonce reuse on rewrite is tolerated by the MRAE AEAD. The degraded security is deterministic encryption (equality leakage), not plaintext recovery. Applications MUST use an MRAE AEAD such as AES-256-GCM-SIV ([RFC8452]) when using derived nonce mode.

3.4.3. Plaintext-Bound Nonces (nonce_mode = "plaintext-bound")

The plaintext-bound construction mixes plaintext content into nonce generation to defend against random number generator state duplication. It uses two passes.

Pass 1 compresses the plaintext to a fixed-size digest and then derives a collision-resistant hash bound to the encryption parameters. The compression step avoids passing the full segment (up to 65536 octets) through lp16, which is limited to 65535 octets. The salt is excluded because an application may need to compute pt_hash before choosing an output destination (for example, during content-addressed deduplication):

```
pt_digest(i) = Hash(plaintext_i)
encryption_params = Encode(AEAD_id, segment_size_str,
                           KDF_id)
pt_hash(i) = KDF(protocol_id, "pt-nonce",
                 [pt_digest(i)],
                 [encryption_params], Nh)
```

Hash is the hash function underlying the KDF (SHA-256 for HKDF-SHA-256). The pt_digest is always Nh octets, well within the lp16 limit.

Pass 2 mixes fresh random material with the payload key and the plaintext commitment:

```
nonce_ctx = Encode(protocol_id,
    uint64(i), pt_hash(i))
nonce(i) = KDF(protocol_id, "nonce",
    [Random(Nn), payload_key],
    [payload_info, nonce_ctx], Nn)
```

Pass 1 is a deterministic function of the plaintext and algorithm parameters; it targets collision resistance, not PRF security. Pass 2 mixes fresh random material with the payload key and the plaintext commitment. When an RNG produces duplicated state (for example, from a virtual machine snapshot), two calls with different plaintexts at the same segment index produce different nonces because pt_hash(i) differs. Two calls with the same plaintext at the same index produce the same (key, nonce, aad) triple under RNG duplication, resulting in deterministic encryption: the ciphertexts are identical, revealing that the plaintexts are equal. No additional information beyond this equality is leaked.

For plaintext-bound rewrites, the procedure is:

1. Recompute pt_hash(i) with the new plaintext.

2. Derive a fresh nonce(i) using the new pt_hash(i) and a new Random(Nn).

3. Seal the new plaintext under the new nonce.

4. Update the accumulator as in Section 3.8.

3.4.4. Nonce Mode Design Rationale

Random mode is simplest but trusts the CSPRNG completely. Derived mode removes that trust entirely at the cost of requiring an MRAE AEAD. Plaintext-bound mode splits the difference: it mixes plaintext into the nonce, so different content at the same index gets different nonces even under RNG duplication. The cost is one hash of the full segment (for pt_digest) plus two short KDF calls, comparable in time to the AEAD encryption itself.

3.5. Hedged Randomness (Optional)

When a long-term symmetric key sk of at least Nh octets is available to the encryptor, implementations SHOULD mix it into random generation using the hedging pattern of [RFC8937]. If only an asymmetric private key is available, it MUST first be processed through a KDF to produce a uniform symmetric key.

```
hedge_key = KDF(protocol_id, "hedge", sk, "", Nh)

HedgedRandom(n, label):
  return KDF(protocol_id, label,
      [hedge_key, Random(n)], "", n)
```

HedgedRandom output depends on both the CSPRNG and sk, so a weak CSPRNG alone cannot predict it. Hedging does not help when the CSPRNG state itself is duplicated (VM snapshots, fork without reseed); identical CSPRNG output still produces identical HedgedRandom output. The plaintext-bound nonce mode (Section 3.4) addresses that distinct threat and is orthogonal to hedging; applications with a long-term key SHOULD apply both.

3.6. Segment AAD

Each segment is encrypted with additional authenticated data (AAD) that binds it to its position in the content and to whether it is the final segment:

```
segment_aad(i, is_final):
  return Encode(aad_label,
      uint64(i), uint8(is_final))
```

The aad_label is the protocol-chosen label from the parameter set. The segment index i is encoded as a big-endian 8-octet integer, and is_final is 1 for the last segment and 0 for all others.

The salt is not included in the AAD because it is already bound into segment_key via the payload schedule; different messages produce different segment keys even at the same index. Transposing two segments within a message fails because the AAD at each position encodes the expected index. Truncation is detectable: a reader that sees is_final = 0 knows more data must follow. Extension is also caught: appending after a segment marked is_final = 1 either invalidates the original final segment on re-read or produces a ciphertext that cannot verify. Encoding both the index and the finality bit in the AAD, rather than relying on the accumulator alone, catches these attacks per-segment without requiring the reader to hold all tags before decrypting any one of them.

3.7. Commitment and Aggregation

Per-segment AEAD alone has two blind spots. Without a commitment, a reader with the wrong key attempts decryption of every segment before discovering the mismatch, leaking timing information in the process. Without an accumulator, an adversary who controls storage can replace segment 3's ciphertext with a different valid ciphertext for index 3 (from another message encrypted under the same key), and the per-segment AEAD check will pass. The commitment catches the first; the accumulator catches the second.

3.7.1. Key Commitment

The commitment is a commitment_length-octet value derived from the CEK and payload_info:

```
commitment = KDF(protocol_id, "commit",
                 [CEK], [payload_info],
                 commitment_length)
```

Before decrypting any segment, a reader derives the expected commitment from the CEK and payload_info and compares it octet-for-octet with the stored value. A mismatch indicates a wrong key, wrong parameter set, or corrupted header.

The commitment achieves CMT-1 security (as defined in [RFC9771]). Under the PRF assumption on the KDF, the probability that any other (CEK, payload_info) pair produces the same commitment is at most 2^(-8 * commitment_length), which is 2^(-256) for commitment_length = 32. Across q key queries the birthday bound gives a collision probability of at most q^2 / 2^(8 * commitment_length + 1).

Several AEADs including AES-GCM ([NIST-SP-800-38D]) and ChaCha20-Poly1305 ([RFC8439]) lack strong native key commitment. An external PRF-based commitment delivers uniform collision resistance regardless of the AEAD choice.

FLOE ([FLOE]) defines ra-CMT, a commitment notion for raAE schemes that is independent of the underlying AEAD's commitment properties. raAE achieves ra-CMT because the PRF-based commitment is checked before any AEAD operation: a wrong key is rejected at the commitment step, so the AEAD's lack of native key commitment cannot be exploited.

For reference, CMT-4 ([RFC9771]) depends on AEAD-specific properties beyond indistinguishability under chosen-ciphertext attack (IND-CCA2). AES-256-GCM and ChaCha20-Poly1305 do not achieve CMT-4; AES-256-GCM-SIV does. raAE's ra-CMT guarantee holds regardless, as long as the commitment check is not bypassed.

3.7.2. Aggregation

The accumulator binds all segment authentication tags into a single Nh-octet content-level integrity value.

The per-segment accumulator contribution is:

```
contrib(i) = KDF(protocol_id, "acc_contrib",
                 [acc_key],
                 [uint64(i), tag(i)], Nh)
```

The aggregation function used to combine contributions MUST be commutative, associative, and support O(1) incremental update when a single segment is rewritten. Formally, the aggregation function Agg over Nh-octet values MUST satisfy:

* Agg(a, b) = Agg(b, a) for all a, b.

* Agg(Agg(a, b), c) = Agg(a, Agg(b, c)) for all a, b, c.

* There exists an identity element e such that Agg(a, e) = a for all a.

* Given the current accumulator, the old contribution, and the new contribution, the updated accumulator is computable in O(1) time.

The accumulator for a message with N segments is:

```
accumulator = Agg(contrib(0), contrib(1),
                  ..., contrib(N-1))
```

When segment i is rewritten, the accumulator is updated as:

```
accumulator = Agg(Agg(accumulator,
                      inverse(old_contrib(i))),
                  new_contrib(i))
```

For XOR-based aggregation (the concrete instantiation defined in Section 6), this simplifies to:

```
accumulator = accumulator XOR old_contrib
              XOR new_contrib
```

A Merkle tree of tags would give O(log N) per-segment verification, but updates cost O(log N) hashes and require storing the tree. XOR-of-PRF trades per-segment verification (which requires all tags) for O(1) update and no auxiliary storage. For a rewrite-heavy workload, O(1) update matters more. To rewrite segment i, compute contrib(i) from the old tag (which is already stored alongside the ciphertext), XOR it out of the current accumulator, encrypt the new plaintext to get a new tag, compute the new contrib(i), and XOR it in. No other segment is read or decrypted; the accumulator update touches only the rewritten segment's metadata. A hash chain would force re-reading all prior tags on every rewrite, defeating the random-access property. The PRF-keyed contribution prevents an adversary without acc_key from predicting any contrib(i) value, so forging a valid accumulator for a modified set of segment tags requires breaking the KDF's PRF security. The accumulator does not bind the total segment count; see Section 7.5 for the implications and required mitigations.

3.8. Encryption and Decryption

Three operations use the framework above.

3.8.1. Encryption

Given a CEK, a fresh salt, a parameter set, and a plaintext P split into segments P_0 through P_{N-1}, encryption works in two phases.

First, derive commitment, payload_key, acc_key (and nonce_base for derived mode) from the payload schedule. Store the commitment as the first commitment_length octets of output.

Then, for each segment i from 0 to N-1:

1. Compute segment_key(i) per Section 3.3.

2. Compute nonce(i) per the chosen nonce_mode (Section 3.4).

3. Set is_final = 1 if i = N-1, else set is_final = 0.

4. Compute aad = segment_aad(i, is_final).

5. Compute C_i = AEAD.Seal(segment_key(i), nonce(i), aad, P_i). The ciphertext ct_i is the first len(P_i) octets of C_i and tag_i is the final Nt octets. This assumes C_i = ct_i || tag_i per [RFC5116] Section 2.1, which holds for all AEADs profiled in Section 6.1.

6. Store nonce(i) (random mode only), ct_i, and tag_i.

7. Compute contrib(i) = KDF(protocol_id, "acc_contrib", [acc_key], [uint64(i), tag_i], Nh).

After all segments are encrypted, compute the accumulator as the XOR of all contrib values:

```
accumulator = contrib(0) XOR contrib(1)
              XOR ... XOR contrib(N-1)
```

Store the accumulator alongside the commitment and segment data.

3.8.2. Decryption of Segment i

Given the CEK, salt, parameter set, segment index i, finality status, and the stored ciphertext for segment i:

1. Derive commitment, payload_key, acc_key from the CEK and salt via the payload schedule.

2. Verify that the stored commitment equals the derived commitment. If they differ, the implementation MUST NOT proceed with decryption, SHOULD zeroize derived key material, and MUST return an error to the caller.

3. Optionally, verify the stored accumulator against the derived contributions from all available segment tags before proceeding to segment decryption. Applications that have all segment tags available SHOULD perform this check. A reader decrypting a single segment without verifying the accumulator gets per-segment AEAD integrity only; cross-segment integrity (detecting substitution or removal) requires accumulator verification.

4. Compute segment_key(i) and nonce(i).

5. Compute aad = segment_aad(i, is_final).

6. Compute P_i = AEAD.Open(segment_key(i), nonce(i), aad, ct_i || tag_i) per [RFC5116] Section 2.2. Return a decryption error if verification fails.

Decryption can fail in three distinguishable ways: commitment mismatch (wrong key or parameters), AEAD verification failure (corrupted or tampered segment), and accumulator mismatch (segment set modified). Implementations SHOULD report these as distinct error conditions for local diagnosis, but SHOULD return a single opaque error to untrusted callers to avoid leaking an oracle (for example, distinguishing "wrong key" from "tampered segment" over a network).

3.8.3. Rewriting Segment i

The rewrite procedure assumes the caller has already verified the commitment or otherwise confirmed it holds the correct CEK. A rewrite under the wrong key produces unreadable ciphertext and corrupts the accumulator; the error is detectable only on a subsequent read.

Rewriting a segment does not require reading or decrypting any other segment. The old tag is already stored alongside the old ciphertext. For random nonce mode:

1. Read the old tag_i from storage.

2. Compute old_contrib = KDF(protocol_id, "acc_contrib", [acc_key], [uint64(i), old_tag_i], Nh).

3. Generate a fresh nonce(i) = Random(Nn).

4. Compute C_i = AEAD.Seal(segment_key(i), nonce(i), segment_aad(i, is_final), new_P_i). Extract new_ct_i and new_tag_i.

5. Compute new_contrib = KDF(protocol_id, "acc_contrib", [acc_key], [uint64(i), new_tag_i], Nh).

6. Update the accumulator: acc = acc XOR old_contrib XOR new_contrib.

7. Store the new nonce(i), new_ct_i, new_tag_i, and the updated accumulator.

For derived nonce mode, nonce(i) is computed per Section 3.4 and is the same on rewrite as on initial encryption. The MRAE AEAD tolerates this reuse. The accumulator update proceeds identically.

Note that the accumulator cannot be validated during an in-place rewrite. Validation requires computing contrib(j) for every segment j and checking that their XOR matches the stored accumulator, which means reading all tags. During a rewrite the application trusts the stored accumulator and applies the delta. Full validation is a separate operation that an application performs when it has access to all tags (for example, on a full read or an integrity audit).

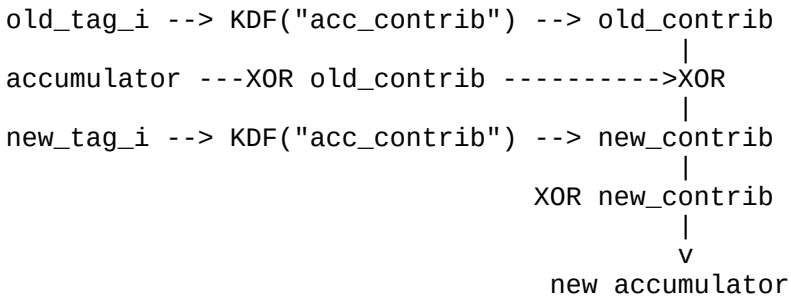

Figure 3: Accumulator Update on Segment Rewrite

4. Framing and Domain Separation Requirements

The security analysis in Section 5 assumes that distinct KDF input tuples produce distinct primitive inputs. This section states the requirements that any concrete framing must satisfy for that assumption to hold.

Injectivity: The mapping from the tuple (protocol_id, label, ikm, info, L) to the octet string input of the underlying hash or KDF primitive MUST be injective. Distinct input tuples MUST NOT produce the same primitive input.

Label uniqueness: Each derivation role, including commitment, payload key, accumulator key, nonce base, epoch key, accumulator contribution, segment nonce, and hedged nonce, MUST use a distinct label string within each protocol version.

Protocol binding: The protocol_id MUST appear as a distinct component of the KDF primitive input. Different application protocols using the same AEAD and KDF MUST use different protocol_id values to ensure that derived values from one protocol cannot be confused with those from another.

Output length commitment: The requested output length L MUST be part of the KDF primitive input. This prevents attacks in which an adversary attempts to use a truncated version of a longer derived output as a valid shorter one.

Cross-role isolation: A KDF output for one derivation role, such as payload_key, MUST be computationally independent of the output for any other role, such as acc_key, even when derived from the same CEK and payload_info. This property follows from label uniqueness and PRF security of the underlying primitive.

5. Security Properties and Assumptions

The adversary can observe all ciphertexts and content metadata, tamper with individual segments or their ordering, replace the accumulator, and attempt decryption with chosen keys. The adversary does not know the CEK, any derived key, or the per-content salt. When multiple messages share a CEK, the per-content salt separates payload schedule outputs across messages under the PRF assumption; the multi-message advantage grows linearly in the number of messages. raAE claims the properties below against this adversary.

The security argument assumes that the KDF is a PRF. For HKDF-SHA-256, HKDF-Extract with a public salt (protocol_id) acts as a randomness extractor on the IKM, which follows from the PRF property of the HMAC compression function per the HKDF analysis [RFC5869]. All raAE-v1 KDF outputs are at most Nh octets, so HKDF-Expand makes a single HMAC call per derivation and the PRF assumption applies directly.

5.1. Segment Confidentiality and Integrity

raAE targets ra-ROR security ([FLOE]): the adversary can adaptively interleave encryption and decryption of segments for different messages, in any order, across different keys. The security argument reduces ra-ROR to the multi-user real-or-random (mu-ROR) security of the underlying AEAD in three steps. First, PRF security of the KDF replaces payload_key with a uniformly random key (one hybrid step per message, since the salt makes each message's derivation independent). Second, each epoch_key is similarly replaced. Third, mu-ROR security of the AEAD bounds per-segment confidentiality and integrity loss.

The bound below uses the following variables. Let F be the number of messages encrypted under adversary-chosen keys, E the total number of epoch keys derived across all messages, and S the total number of segment encryption queries. For each per-epoch key k, let q_k be the number of segment encryptions under that key.

Conditioned on no nonce collisions among segments sharing a key (the birthday bound for each per-epoch key; see Section 5.4.1), the advantage is informally bounded by:

```
Adv <= F * 4 * Adv_PRF
       + E * Adv_PRF
       + S * Adv_muROR
       + Adv_acc
```

where Adv_PRF is the PRF advantage of the KDF, Adv_muROR is the per-query multi-user real-or-random advantage of the AEAD, and Adv_acc is the accumulator forgery advantage.

The total advantage is the sum of the nonce collision probability and the conditioned bound above. In derived nonce mode, nonces are distinct by construction and the collision term vanishes. In random nonce mode with epoch_length = 0 (one key per segment), q_k equals the rewrite count for a single segment position.

In practice the dominant term is Adv_muROR, which scales with the total number of segment encryption queries. The PRF terms (payload schedule and epoch keys) are typically negligible because F and E are small relative to S. The accumulator forgery advantage Adv_acc is bounded by Adv_PRF + q_forge * 2^(-8 * Nh), where q_forge is the number of forgery attempts. Under the PRF assumption each contrib value is uniformly random, so each attempt succeeds with probability at most 2^(-256) for Nh = 32.

A formal proof extending the ra-ROR analysis of FLOE [FLOE] to the XOR-of-PRF accumulator is in preparation; the bound above captures the reduction structure.

Segment ciphertexts are indistinguishable from random under the ra-ROR definition. Each segment is authenticated, including its binding to its index and finality status via the segment AAD. An adversary cannot modify, reorder, truncate, or extend the set of segments without causing an AEAD verification failure on at least one affected segment.

5.2. Key Commitment

raAE-v1 uses commitment_length = 32 (Nh octets) and Nt = 16. The birthday bound and CMT-1 analysis in Section 3.7.1 apply directly; the per-pair collision probability is at most 2^(-257), and the per-pair forgery probability is at most 2^(-128).

5.3. Accumulator Security

The accumulator is unforgeable without knowledge of the acc_key. An adversary who does not know acc_key cannot predict any individual contrib(i) value (by the PRF security of the KDF), and therefore cannot construct a valid accumulator for any set of segment tags that differs from the authenticated set.

The XOR-based aggregation is susceptible to cancellation: if an adversary can control two contributions simultaneously such that their XOR equals the XOR of two target contributions, a forgery is

possible. However, each contrib(i) is a KDF output keyed by acc_key and bound to the segment index and AEAD tag. Without knowledge of acc_key, the adversary cannot select segment tags to achieve a target contribution value.

An adversary who compromises acc_key can forge accumulator values without recovering any segment plaintext. Applications MUST treat acc_key exposure as equivalent to accumulator integrity loss and MUST rotate the CEK if acc_key compromise is suspected.

5.4. AEAD Usage Limits

Two bounds constrain the number of segment operations under a single key: a confidentiality bound driven by nonce collisions and a per-query integrity bound driven by forgery probability. Both are analyzed in [I-D.irtf-cfrg-aead-limits]. For raAE with segment_size = 65536 octets (L = 4096 AES blocks, or L' ~ 4099 Poly1305 blocks), the per-key limits are:

5.4.1. Confidentiality (Nonce Collision)

For random nonce mode with Nn-octet nonces and q segment encryptions under one key, the collision probability follows the birthday bound:

P(collision) <= q^2 / 2^(8*Nn + 1)

When epoch_length = r is specified, q counts encryptions per epoch key (initial writes plus rewrites within that epoch), not across the whole content. Each epoch key has an independent budget.

For derived nonce mode, nonces are deterministic and distinct across segment indices. Rewriting the same segment reuses the nonce, which is tolerated by the MRAE AEAD with degradation to equality leakage.

5.4.2. Integrity (Forgery)

Each AEAD decryption query gives the adversary a chance to forge a valid ciphertext. The forgery advantage per query depends on the AEAD and the segment size.

For AES-256-GCM ([I-D.irtf-cfrg-aead-limits] Section 5.1):

IA <= 2 * v * (L + 1) / 2^128

where v is the number of forgery attempts and L = 4096 blocks per segment.

For ChaCha20-Poly1305 ([I-D.irtf-cfrg-aead-limits] Section 5.2):

IA <= v * (L' + 1) / 2^103

The 2^103 denominator (not 2^128) reflects Poly1305's per-query forgery bound. At 65536-octet segments, the integrity limit for ChaCha20-Poly1305 is tighter than for AES-GCM.

5.4.3. Combined Budget

The binding constraint is whichever limit is reached first. For random nonce mode with 96-bit nonces, nonce collision (q < 2^32 at 2^(-32) target) is typically the binding constraint because integrity limits are much larger (v < 2^83 for AES-GCM, v < 2^59 for ChaCha20-Poly1305 at the same target).

Applications MUST track total segment encryptions per key and rotate the CEK before exceeding either limit.

5.5. Rewrite Budget

The rewrite budget is the per-key limit from Section 5.4 applied to segment rewrites. Each rewrite counts as one encryption query against the confidentiality bound and may trigger decryption queries (for verification) against the integrity bound.

With epoch_length = r and Nn = 12, each epoch key covers at most 2^r segment positions. The per-epoch nonce budget is independent of other epochs, so the content's total rewrite capacity scales with the number of epochs.

For derived nonce mode, nonce collisions cannot occur from rewrites because the AEAD is MRAE. The budget is limited by the AEAD's native integrity margin (approximately 2^48 messages for AES-256-GCM-SIV per [RFC8452] Section 6.2) and the application's key rotation policy.

6. raAE-v1 Profile

This section defines the "raAE-v1" profile. All values are as defined in this section unless otherwise specified. The identifier "raAE-v1" is reserved for the profile defined in this document. Applications MUST NOT reuse it for a different parameter set.

6.1. Algorithms

Algorithm	Identifier	Nk	Nn	Nt	MRAE
AES-256-GCM	aes-256-gcm	32	12	16	No
ChaCha20-Poly1305	chacha20-poly1305	32	12	16	No
AES-256-GCM-SIV	aes-256-gcm-siv	32	12	16	Yes
AEGIS-256	aegis-256	32	32	16	No
AEGIS-256X2	aegis-256x2	32	32	16	No

Table 1: Supported AEAD Algorithms

The MRAE column indicates Nonce Misuse-Resistant Authenticated Encryption as defined in [RFC9771]. All algorithms provide [RFC5116] semantics with 16-octet tags (Nt = 16). AES-256-GCM is specified in [NIST-SP-800-38D]. ChaCha20-Poly1305 is specified in [RFC8439]. AES-256-GCM-SIV is specified in [RFC8452]. AEGIS-256 and AEGIS-256X2 are specified in [I-D.irtf-cfrg-aegis-aead].

The epoch_length parameter interacts with the AEAD choice. For ChaCha20-Poly1305, epoch_length MUST be present because 96-bit nonces require epoch keys to bound the per-key AEAD invocation count. For AES-256-GCM, epoch_length MUST be present for the same reason. For AES-256-GCM-SIV, epoch_length MUST NOT be present because AES-256-GCM-SIV is an MRAE AEAD and uses derived nonce mode where epoch keys are not applicable. For AEGIS-256 and AEGIS-256X2, epoch_length SHOULD NOT be present because 256-bit nonces make collision probability negligible even without per-epoch key rotation.

The following table summarizes valid parameter combinations:

AEAD	nonce_mode	epoch_length
AES-256-GCM	random, pt-bound	MUST present
ChaCha20-Poly1305	random, pt-bound	MUST present
AES-256-GCM-SIV	derived	MUST NOT present
AEGIS-256	random, pt-bound	SHOULD NOT present
AEGIS-256X2	random, pt-bound	SHOULD NOT present

Table 2: Valid Parameter Combinations

The KDF is HKDF-SHA-256 ([RFC5869]), with Nh = 32. The framework supports single-stage KDFs; a future document may define a profile using one (for example, TurboSHAKE256).

The supported segment sizes are 16384 and 65536 octets. Both values are powers of two and at least 4096 octets. The 16384-octet size aligns to 16 KiB memory pages (for example, on Apple Silicon). The 65536-octet size aligns to 64 KiB (four 16 KiB pages).

The commitment_length is 32 octets. The aad_label is "raAE-DATA".

The per-AEAD usage limits for raAE-v1 follow the analysis in Section 5.4. The table below gives per-key limits for 65536-octet segments at a 2^(-32) advantage target.

AEAD	Conf. (q)	Integ. (v)	Binding
AES-256-GCM	2^32	2^83	conf.
ChaCha20-Poly1305	2^32	2^59	conf.
AES-256-GCM-SIV	(MRAE)	2^48	integ.
AEGIS-256	2^112	2^83	integ.
AEGIS-256X2	2^112	2^83	integ.

Table 3: Per-Key AEAD Usage Limits

Confidentiality values are rounded down to the nearest integer exponent. The "Binding" column indicates which limit is reached first in practice. For 96-bit-nonce AEADs with random nonces, confidentiality (nonce collision) binds. For AEGIS and GCM-SIV, integrity binds because nonce collisions are either negligible (256-bit nonces) or impossible (derived nonces). The AEGIS integrity values are conservative: with 256-bit tags AEGIS has a stronger per-query forgery bound than AES-GCM, but raAE-v1 uses 128-bit tags (Nt = 16) where the per-query bound is comparable. 2^83 (the AES-GCM value) is used as a floor pending a published per-query analysis in [I-D.irtf-cfrg-aegis-aead].

For AES-256-GCM with epoch_length = 0 (one segment per epoch): the per-epoch budget equals the rewrite count for a single segment position. At the default segment_size of 65536, a 1 TiB file contains approximately 2^24 segments. With epoch_length = 0, each segment can be rewritten approximately 2^32 times before the per-epoch birthday bound is reached at the 2^(-32) target. Epoch keys are fresh for each epoch, so rewrites in one epoch do not affect the collision bound for other epochs.

6.2. Concrete Framing

The concrete framing uses a length-prefixed encoding.

```
lp16(x) = I2OSP(len(x), 2) || x
Encode(x1, ..., xn) = lp16(x1) || ... || lp16(xn)
```

Each argument to Encode MUST be at most 65535 octets (the maximum representable by a 2-octet length prefix). Encode is injective: each lp16 field is self-delimiting, so the concatenation can be uniquely parsed regardless of the number of arguments. Distinct input tuples produce distinct output strings.

6.2.1. Two-Stage KDF (sha-256)

```
KDF(protocol_id, label, ikm, info, L):
  extract_input = Encode(protocol_id, label,
                         ...ikm)
  prk = HKDF-Extract(salt=protocol_id,
                     ikm=extract_input)
  expand_info = Encode(protocol_id, label,
                       ...info, I2OSP(L, 2))
  return HKDF-Expand(prk, expand_info, L)
```

The notation ...ikm means each element of the ikm list is a separate argument to Encode. When ikm is a single octet string rather than a list, it is treated as a one-element list. The protocol_id appears in both the HKDF salt and the extract_input as a defensive measure; binding in either position suffices for domain separation.

6.2.2. Payload Info Construction

When epoch_length is absent:

```
payload_info = Encode(AEAD_id, segment_size_str,
                      KDF_id, salt)
```

When epoch_length is present:

```
payload_info = Encode(AEAD_id, segment_size_str,
                      KDF_id, epoch_length_str, salt)
```

Each element is an ASCII string or raw octet string. AEAD_id and KDF_id are the identifier strings from Section 6.1 (for example, "aes-256-gcm" and "sha-256"). segment_size_str is the decimal ASCII representation of the segment size (for example, "65536"). epoch_length_str is the decimal ASCII representation of epoch_length (for example, "0" or "1").

6.2.3. Segment AAD

The following instantiates the abstract segment AAD construction in Section 3.6 for the raAE-v1 profile.

```
segment_aad(i, is_final):
  return Encode("raAE-DATA",
                I2OSP(i, 8),
                I2OSP(is_final, 1))
```

6.2.4. Labels

Derivation Role	Label
Commitment	"commit"
Payload key	"payload_key"
Accumulator key	"acc_key"
Nonce base	"nonce_base"

Epoch key	"epoch_key"
Accumulator contribution	"acc_contrib"
Plaintext-bound hash	"pt-nonce"
Plaintext-bound nonce	"nonce"
Hedged key	"hedge"

Table 4: KDF Labels by Role

Each label is an ASCII string and is distinct from all others.

6.3. Concrete Procedures

6.3.1. Concrete Payload Schedule

The following instantiates the abstract payload schedule derivation in Section 3.3 for the raAE-v1 profile.

```
payload_info = Encode(AEAD_id, segment_size_str,
                      KDF_id, salt)           ;; epoch absent
;; or:
payload_info = Encode(AEAD_id, segment_size_str,
                      KDF_id, epoch_length_str,
                      salt)                   ;; epoch present

commitment  = KDF("raAE-v1", "commit",
                  [CEK], [payload_info], 32)
payload_key = KDF("raAE-v1", "payload_key",
                  [CEK], [payload_info], Nk)
acc_key     = KDF("raAE-v1", "acc_key",
                  [CEK], [payload_info], Nh)

;; For derived nonce mode only:
nonce_base  = KDF("raAE-v1", "nonce_base",
                  [CEK], [payload_info], Nn)
```

6.3.2. Epoch Key

```
segment_key(i):
  if epoch_length absent:
    return payload_key
  epoch_index = i >> epoch_length
  return KDF("raAE-v1", "epoch_key",
             [payload_key],
             [I2OSP(epoch_index, 8)], Nk)
```

6.3.3. Segment Encryption

```
key_i     = segment_key(i)
nonce_i   = (per nonce_mode)
is_final  = 1 if i == N-1 else 0
aad       = Encode("raAE-DATA",
                   I2OSP(i, 8),
                   I2OSP(is_final, 1))
(ct_i, tag_i) = AEAD.Seal(key_i, nonce_i,
                          aad, plaintext_i)
```

6.3.4. Accumulator

```
contrib_i = KDF("raAE-v1", "acc_contrib",
                [acc_key],
                [I2OSP(i, 8), tag_i], Nh)

accumulator = contrib_0 XOR contrib_1
              XOR ... XOR contrib_{N-1}
```

6.3.5. Rewrite Update

```
old_contrib = KDF("raAE-v1", "acc_contrib",
                  [acc_key],
                  [I2OSP(i, 8), old_tag], Nh)
new_contrib = KDF("raAE-v1", "acc_contrib",
                  [acc_key],
                  [I2OSP(i, 8), new_tag], Nh)
accumulator = accumulator XOR old_contrib
              XOR new_contrib
```

Test vectors for the raAE-v1 profile are provided in Appendix B.

7. Security Considerations

raAE's ra-ROR security ([FLOE]) rests on three assumptions: the AEAD is mu-ROR secure, the KDF is a secure PRF, and nonces do not collide. Each subsection below describes a way one of these can fail and what breaks.

7.1. Nonce Misuse

Nonce reuse under a non-MRAE AEAD leaks plaintext: an adversary who observes two ciphertexts under the same key and nonce recovers the XOR of the plaintexts (for CTR-based AEADs) and can forge new ciphertexts.

Random nonce mode depends entirely on the CSPRNG. If the CSPRNG returns duplicated state, segments collide. Derived nonce mode is immune because nonces are deterministic; the MRAE AEAD degrades to equality leakage on rewrite, not plaintext recovery. Plaintext-bound mode partially defends against CSPRNG duplication: different plaintexts at the same index produce different nonces because pt_hash differs, but equal plaintexts still collide. Implementations that need full RNG-duplication defense MUST use derived nonce mode with an MRAE AEAD.

7.2. Parameter Set Mismatch

The nonce_mode and aad_label are not bound into the commitment or payload_info. If a reader uses a different nonce_mode or aad_label than the writer, the per-segment AEAD decryption will fail (wrong nonce or wrong AAD), but the commitment check will pass because the commitment depends only on the CEK and payload_info. Consuming protocols MUST authenticate the full parameter set, including nonce_mode and aad_label, before decryption. In particular, confusing derived for random mode with a non-MRAE AEAD causes nonce reuse on every rewrite, breaking confidentiality completely. Consuming protocols MUST integrity-protect the nonce_mode before attempting nonce recovery or AEAD decryption.

7.3. Framing and Label Errors

Two classes of implementation error break cross-role isolation. A non-injective framing function maps distinct KDF input tuples to the same primitive input, correlating outputs that should be independent; implementations MUST verify injectivity per Section 4. Reusing a label across roles (for example, "commit" for both commitment and payload key) has the same effect. Labels MUST be distinct within a protocol version, and a new version that changes any derivation MUST change the protocol_id.

7.4. Parameter Misuse

The nonce mode and AEAD choice are coupled. An MRAE AEAD in random nonce mode wastes its misuse resistance; a non-MRAE AEAD in derived nonce mode reuses nonces on every rewrite and breaks completely. Epoch keys compound the problem: they add a derivation layer that the MRAE nonce structure does not expect. The rule is simple. MRAE AEADs MUST use derived nonce mode and MUST NOT use epoch keys. Non-MRAE AEADs MUST use random or plaintext-bound nonce mode. See Section 6.1 for per-AEAD guidance.

7.5. Accumulator Limitations

The accumulator has four limitations worth highlighting.

First, it does not bind the segment count. An adversary who controls the storage layer can remove the final segment (the one marked is_final = 1) and present truncated content where every remaining segment individually verifies. Applications MUST store the expected segment count or content size independently. A reader decrypting a single segment without verifying the accumulator and without an independent segment count cannot detect truncation. Applications that support random-access single-segment reads MUST either store the total segment count alongside the commitment and accumulator, or verify the accumulator on every read.

Second, compromising acc_key lets an adversary forge accumulator values without recovering plaintext (see Section 5.3). Rotate the CEK if acc_key compromise is suspected.

Third, without accumulator verification a storage adversary can roll back individual segments to previously valid versions. The per-segment AEAD cannot detect this because the old ciphertext remains valid under the same key and index. Applications that skip accumulator verification on reads accept this rollback risk.

Fourth, a different aggregation function (not XOR) must still be commutative, associative, and support O(1) incremental update. The XOR-of-PRF instantiation in Section 6 satisfies all three; substitutes MUST be verified independently.

7.6. Salt Reuse

Reusing a salt with the same CEK across two files produces identical payload schedule outputs: the same payload_key, acc_key, and (in derived mode) the same nonces. An adversary can silently swap segments between the two files, and both per-segment AEAD checks and the accumulator will pass. This is a complete integrity break. Applications MUST ensure salt uniqueness per CEK; the MUST at Section 3.2 exists for this reason.

7.7. Rewrite Hazards

Two rewrite-related hazards deserve attention.

Applications MUST track total segment encryptions per key and rotate the CEK before exceeding the budget in Section 5.5. For AES-256-GCM with epoch_length = 0, that budget is roughly 2^32 rewrites per segment position; exceeding it risks nonce collisions and plaintext recovery.

When multiple writers rewrite different segments concurrently, each computes an independent accumulator delta (old_contrib XOR new_contrib). Applying these deltas is commutative, but the read-modify-write on the stored accumulator requires coordination. As a recovery mechanism, the accumulator can always be rebuilt from scratch by XOR-ing all contrib values.

raAE also does not guarantee atomic rewrites. A segment rewrite touches four values (nonce, ciphertext, tag, accumulator). A crash between any two of these leaves the content inconsistent. Applications MUST use write-ahead logging, copy-on-write, or an equivalent mechanism to make rewrites recoverable.

7.8. Password-Derived CEKs

The CEK is specified as a 32-octet uniform random value. If an application derives the CEK from a password, the commitment becomes a deterministic function of the password and payload_info (both known to the attacker), enabling offline dictionary attacks. Applications that derive CEKs from passwords MUST use a memory-hard KDF (such as Argon2id) to produce the CEK, and MUST NOT use the raw password output as the CEK without stretching.

7.9. Constant-Time Implementation

Several raAE operations handle secret data and MUST be implemented in constant time to prevent timing side-channels.

The KDF calls in the payload schedule and epoch key derivation take the CEK or payload_key as input keying material. Implementations MUST ensure that HKDF-Extract and HKDF-Expand execute in constant time with respect to their key inputs. In practice this is satisfied by HMAC implementations that do not branch on key octets.

The commitment comparison (Section 3.8) MUST use a constant-time octet comparison. A variable-time comparison leaks the position of the first differing octet, which reveals partial information about the derived commitment and thus about the CEK.

The XOR operations in accumulator update and derived nonce construction are inherently constant-time on standard hardware. No special care is needed for these.

AEAD Seal and Open operations inherit the constant-time requirements of the underlying AEAD. Implementations SHOULD use AEAD libraries that document constant-time guarantees.

7.10. Properties Not Provided

raAE protects segment content and binds segments together. It deliberately does not address four concerns that belong to the consuming protocol.

The CEK must remain available as long as any reader needs access, so there is no forward secrecy. Ciphertexts are not bound to any sender identity; a signing or MAC layer is needed for sender authentication. Key identifiers and structural features of the encrypted format are visible, so unlinkability requires application-layer measures. Finally, the accumulator verifies segments within a message but cannot detect replacement of the message itself; an adversary who swaps one encrypted message for another goes undetected unless the application binds message identity externally.

8. Privacy Considerations

raAE does not define wire formats, so the privacy properties below apply to the abstract mechanism; consuming protocols inherit responsibility for metadata protection.

The number of segments and the size of the final segment reveal the plaintext length to within one segment_size boundary. Applications that require length hiding SHOULD pad the final segment before encryption.

In random nonce mode, stored nonces change on each rewrite. An observer comparing two snapshots of the same encrypted content can identify which segments were rewritten by comparing per-segment nonces. The accumulator also changes on rewrite, revealing that at least one segment was modified. raAE does not provide write-pattern privacy; consuming protocols that require this property must re-encrypt all segments on each write.

The per-content salt is stored unencrypted and uniquely identifies a content object. Observers can use the salt to link different copies or versions of the same encrypted content across storage locations or backups. Applications requiring unlinkability must encrypt the salt under a separate mechanism.

In derived nonce mode, the accumulator is a deterministic function of the CEK, salt, and plaintext. Two encryptions of identical content under the same key material produce identical accumulators, revealing content equality to any observer of the stored accumulator.

For a systematic treatment of privacy threats in Internet protocols, see [RFC6973].

9. IANA Considerations

Algorithm identifiers in the raAE-v1 profile are scoped to that profile. Consuming protocols define their own identifier mappings. This document has no IANA actions.

10. References

10.1. Normative References

[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>.

[NIST-SP-800-38D]
Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007, <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/rfc/rfc5116>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[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>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/rfc/rfc8439>.

[RFC8452] Gueron, S., Langley, A., and Y. Lindell, "AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption", RFC 8452, DOI 10.17487/RFC8452, April 2019, <https://www.rfc-editor.org/rfc/rfc8452>.

[RFC9771] Bozhko, A., Ed., "Properties of Authenticated Encryption with Associated Data (AEAD) Algorithms", RFC 9771, DOI 10.17487/RFC9771, May 2025, <https://www.rfc-editor.org/rfc/rfc9771>.

10.2. Informative References

[FLOE] F卬brega, A., Len, J., Ristenpart, T., and G. Rubin, "Random-Access AEAD for Fast Lightweight Online Encryption", IACR ePrint 2025/2275, 2025, <https://eprint.iacr.org/2025/2275>.

[I-D.irtf-cfrg-aead-limits]
G端nther, F., Thomson, M., and C. A. Wood, "Usage Limits on AEAD Algorithms", Work in Progress, Internet-Draft, draft-irtf-cfrg-aead-limits-11, 4 December 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-aead-limits-11>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/rfc/rfc6973>.

[RFC8645] Smyshlyaev, S., Ed., "Re-keying Mechanisms for Symmetric Keys", RFC 8645, DOI 10.17487/RFC8645, August 2019, <https://www.rfc-editor.org/rfc/rfc8645>.

[RFC8937] Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020, <https://www.rfc-editor.org/rfc/rfc8937>.

[STREAM] Hoang, V. T., Reyhanitabar, R., Rogaway, P., and D. Viz叩r, "Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance", IACR ePrint 2015/189, 2015, <https://eprint.iacr.org/2015/189>.

Appendix A. Example File Layouts

This appendix is informative. It describes two example layouts a consuming protocol might use to store raAE output. raAE does not mandate either layout.

A.1. Linear Layout

In a linear layout the salt, commitment, accumulator, and segment data appear in sequence. The salt comes first because it is needed to derive all payload schedule values. The commitment follows so a reader can reject a wrong key before reading any segment data. The accumulator precedes the segment data so a streaming reader can verify content-level integrity before beginning decryption; this also removes any ambiguity about which segment is final. Segments then follow in index order.

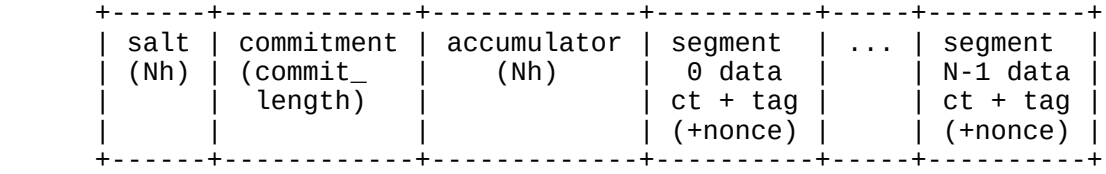

Figure 4: Linear Layout

In random nonce mode the per-segment nonce is stored alongside the ciphertext and tag for that segment. In derived nonce mode no nonce storage is required.

Linear layout supports streaming writes. A writer emits the salt, commitment, and a placeholder for the accumulator, then streams segments. After all segments are written the writer seeks back to the accumulator position and writes the final value in place.

A.2. Aligned Layout

In an aligned layout all per-segment metadata is collected into a header, followed by fixed-size ciphertext blocks. The header contains the salt, commitment, accumulator, and one (nonce, tag) pair per segment. Ciphertext blocks follow at fixed offsets.

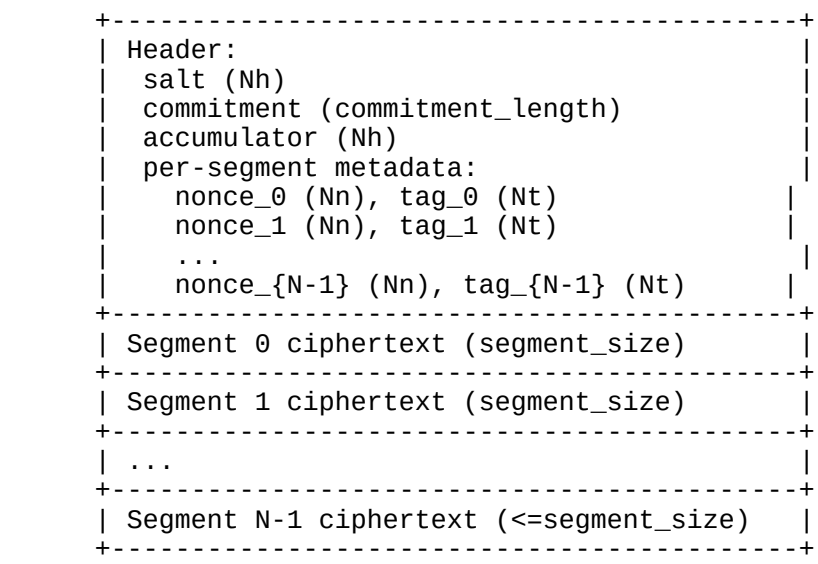

Figure 5: Aligned Layout

To read segment i, a reader seeks to header_size + i * segment_size. The header contains all nonces and tags, so the accumulator can be verified by reading only the header. This layout supports efficient random-access reads (one seek per segment) and enables accumulator verification without reading any segment ciphertext.

Appendix B. Test Vectors

This appendix is informative.

All test vectors use protocol_id = "raAE-v1" and KDF = sha-256. Unless stated otherwise, segment_size = 65536 octets. Vectors are provided for AES-256-GCM, ChaCha20-Poly1305, AES-256-GCM-SIV, AEGIS-256, and AEGIS-256X2. The initial vectors below use AES-256-GCM with nonce_mode = "random".

B.1. Single-Segment Vector

This vector encrypts the 12-octet string "Hello, raAE!" as a single segment. The payload_info is constructed by Encode()-ing the AEAD identifier "aes-256-gcm", the segment size "65536", the KDF identifier "sha-256", and the 32-octet salt. The commitment is then derived by calling KDF with protocol_id "raAE-v1", label "commit", the CEK as ikm, and payload_info as info. The KDF trace below shows the HKDF-Extract input (Encode of protocol_id, label, CEK) and the HKDF-Expand info (Encode of protocol_id, label, payload_info, and output length), followed by the 32-octet commitment. The payload_key and acc_key follow the same pattern with their respective labels.

```
Inputs:
  protocol_id:  "raAE-v1"
  CEK:          aaaaaaaa...aa (32 octets of 0xAA)
  salt:         04040404...04 (32 octets of 0x04)
  plaintext:    "Hello, raAE!" (12 octets)
  nonce_0:      030303030303030303030303
  AEAD:         aes-256-gcm
  segment_size: 65536
  KDF:          sha-256

payload_info (hex):
  000b6165732d3235362d67636d000536353533
  3600077368612d32353600200404040404040404
  0404040404040404040404040404040404040404040404

KDF Trace (commitment):
  extract_input:
    0007726141452d76310006636f6d6d6974
    0020aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    prk:
      cb10c1430a1c89fef79ddd6d99ef698b
      6a64161bbc349c2f6eb3e673e32508c6
    expand_info:
      0007726141452d76310006636f6d6d6974003f
      000b6165732d3235362d67636d000536353533
      3600077368612d32353600200404040404040404
      0404040404040404040404040404040404040404
      04040404040400020020
    commitment:
      454f1649919652acf3032d9331fbec23
      34c68fc7031f114fe15808d2029c91fa

  payload_key:
    170573c64e86782013e37149914db731
    d25968df650f85ea1062093f297aabe3
  acc_key:
    d4b04ab7b60d6d3fd4bc4f110f018279
    5c3bd3f5f9f4dcce2f82c2d7c2f284f0

  Segment 0 (is_final=1):
    nonce_0:      030303030303030303030303
    segment_aad:  Encode("raAE-DATA",
                         I2OSP(0, 8), I2OSP(1, 1))
      = 0009726141452d4441544100080000000000000000000101
    ct+tag:
      cb4139ff74b6e97c9e2e8adbb711ee1a
      212aa0d7054ecbd2d567fa49
    tag_0: b711ee1a212aa0d7054ecbd2d567fa49

  Accumulator (single segment):
    contrib_0:
      de0c0c543502add75f3ffdab8129bb0d
      d77d8a4a9da83184024cb153f58880a6
    accumulator:
      de0c0c543502add75f3ffdab8129bb0d
      d77d8a4a9da83184024cb153f58880a6
      (accumulator = contrib_0 for N=1)
```

B.2. Two-Segment Vector

```
  Inputs:
    protocol_id:  "raAE-v1"
    CEK:          aaaaaaaa...aa (32 octets of 0xAA)
    salt:         04040404...04 (32 octets of 0x04)
    segment_0:    "Block zero data!" (16 octets)
    segment_1:    "Final block." (12 octets, final)
```

```
    nonce_0:      030303030303030303030303
    nonce_1:      050505050505050505050505
    AEAD:         aes-256-gcm
    segment_size: 65536
    KDF:          sha-256

  Payload Schedule (same as single-segment):
    commitment:
      454f1649919652acf3032d9331fbec23
      34c68fc7031f114fe15808d2029c91fa
    payload_key:
      170573c64e86782013e37149914db731
      d25968df650f85ea1062093f297aabe3
    acc_key:
      d4b04ab7b60d6d3fd4bc4f110f018279
      5c3bd3f5f9f4dcce2f82c2d7c2f284f0

  Segment 0 (is_final=0):
    nonce_0:      030303030303030303030303
    segment_aad:  Encode("raAE-DATA",
                         I2OSP(0, 8), I2OSP(0, 1))
      = 0009726141452d4441544100080000000000000000000100
    ct+tag_0:
      c1483af070bab36b8d00ef9ed6fb1452
      36cf3e20e3de9375aaa2c2e2a873318e
    tag_0: 36cf3e20e3de9375aaa2c2e2a873318e

  Segment 1 (is_final=1):
    nonce_1:      050505050505050505050505
    segment_aad:  Encode("raAE-DATA",
                         I2OSP(1, 8), I2OSP(1, 1))
      = 0009726141452d4441544100080000000000000001000101
    ct+tag_1:
      a10003997560fbb42adc3a8de0b4131e
      e8e5d0154190bd588bf5e7a6
    tag_1: e0b4131ee8e5d0154190bd588bf5e7a6

  Accumulator (two segments):
    contrib_0:
      a61d5e6bcb37211246d6ac546f29262f
      9f39c690462bce8834a1292e0f55937a
    contrib_1:
      097c8a52de03b224dd43f471a9341282
      55f5c8b6d623ab87a46f5eb83cc706e3
    accumulator:
      af61d439153493369b955825c61d34ad
      cacc0e269008650f90ce779633929599
      (accumulator = contrib_0 XOR contrib_1)
```

B.3. Epoch Key Vectors

The following vectors use the same CEK and salt as above. They demonstrate the effect of epoch_length on payload_info and on segment key derivation.

```
--- epoch_length absent ---
payload_info (hex):
  000b6165732d3235362d67636d000536353533
  3600077368612d32353600200404040404040404
  0404040404040404040404040404040404040404040404
payload_key:
  170573c64e86782013e37149914db731
  d25968df650f85ea1062093f297aabe3
segment_key(0) = payload_key
segment_key(1) = payload_key

--- epoch_length = 0 ---
payload_info (hex):
  000b6165732d3235362d67636d000536353533
  3600077368612d32353600013000200404040404040404
  0404040404040404040404040404040404040404040404
  04040404
payload_key:
  223b82c12818dd4cb8da2b4ae5092075
  0a6bc404661c3dbb291a069aca0e3aa5
segment_key(0) = epoch_key(epoch=0):
  65cca11fda472b224be476566897c09c
  5006c856ec1698be47b27db8154e8a01
segment_key(1) = epoch_key(epoch=1):
  e9b26223a1ca32d620a2462170f56b24
  5f8d859519b7681a0fa229fc8a155e85

--- epoch_length = 1 ---
payload_info (hex):
  000b6165732d3235362d67636d000536353533
  3600077368612d32353600013100200404040404040404
  0404040404040404040404040404040404040404040404
  04040404
payload_key:
  23e9988c2cfd2db4f6e648fced969c81
  c7d676f31254def813a3f841fe733a5f
segment_key(0) = segment_key(1) = epoch_key(epoch=0):
  b0def46ad428a0c0395473c4129632b5
  127cb4c825d7db558551c0e27f5c7ebf
segment_key(2) = epoch_key(epoch=1):
  8af593d86913dfa1e3d193a4d9dc0378
  d51c1536b454986569e82420ff568eae
```

When epoch_length = 0, each segment receives a unique derived key (one segment per epoch). When epoch_length is absent, epoch derivation is skipped entirely and all segments use payload_key directly.

B.4. KDF Isolation Vectors

```
Common inputs:
  protocol_id: "raAE-v1"
  label:       "TEST-LABEL"
               (hex: 544553542d4c4142454c)
  ikm:         0a0b0c0d0e0f (6 octets)
  info:        "" (0 octets)

KDF("raAE-v1", "TEST-LABEL", ikm, "", 32):
  92e7e2777e02b90014ab3e66ffa55ad9
  2cdaba3aee1627c8dd51224ed6899e05

KDF("raAE-v1", "TEST-LABEL", ikm, "", 16):
  6a66aec2c022b339df1299b66a591fe2
```

The two outputs above share no prefix even though they differ only in the requested length L. This demonstrates that the output length commitment in the framing produces independent outputs for different L values.

B.5. Derived Nonce Mode Vector

This vector uses the same CEK, salt, and plaintext as the single-segment random vector, but with nonce_mode = "derived" and epoch_length absent. The nonce is computed deterministically from nonce_base. This vector uses AES-256-GCM with derived nonces for KDF testing convenience (same CEK and salt as the random-nonce vector above). In production, derived nonce mode MUST only be used with MRAE AEADs; see Section 3.4.

```
   Inputs:
     protocol_id:  "raAE-v1"
     CEK:          aaaaaaaa...aa (32 octets of 0xAA)
     salt:         04040404...04 (32 octets of 0x04)
     plaintext:    "Hello, raAE!" (12 octets)
     nonce_mode:   derived
     AEAD:         aes-256-gcm
     segment_size: 65536
     KDF:          sha-256

   Payload Schedule (same CEK and salt, epoch absent):
     commitment:   (same as single-segment vector)
     payload_key:  (same as single-segment vector)
     acc_key:      (same as single-segment vector)
     nonce_base:
       50328410634d38b5798e931e

   nonce_0 = nonce_base XOR uint64(0):
     50328410634d38b5798e931e

   Segment 0 (is_final=1):
     nonce_0:      50328410634d38b5798e931e
     segment_aad:
       0009726141452d44415441
       000800000000000000000000101
     ct+tag:
       bc72c63154666be5e8cc253a110ddc57
       7932263db32b2d861d5d6c61
     tag_0: 110ddc577932263db32b2d861d5d6c61

   Accumulator:
     contrib_0:
       84c0f459b51162bc69ad4f9e32ffc310
       ce8e47ea4d95372e246d9781ef63025b
     accumulator:
       84c0f459b51162bc69ad4f9e32ffc310
       ce8e47ea4d95372e246d9781ef63025b
```

B.6. Plaintext-Bound Nonce Mode Vector

This vector uses the same CEK, salt, and plaintext as above, but with nonce_mode = "plaintext-bound". The nonce depends on both the plaintext content and fresh random material. For reproducibility, Random(12) = 0x070707...07 (12 octets of 0x07).

```
   Inputs:
     protocol_id:  "raAE-v1"
     CEK:          aaaaaaaa...aa (32 octets of 0xAA)
     salt:         04040404...04 (32 octets of 0x04)
     plaintext:    "Hello, raAE!" (12 octets)
     Random(12):   070707070707070707070707
     nonce_mode:   plaintext-bound
     AEAD:         aes-256-gcm
     segment_size: 65536
     KDF:          sha-256

   Payload Schedule:
     (same as single-segment vector)

   nonce_0 (plaintext-bound):
     5e8def13adb2d65b5054fd15

   Segment 0 (is_final=1):
     nonce_0:      5e8def13adb2d65b5054fd15
     segment_aad:
       0009726141452d44415441
       000800000000000000000000101
     ct+tag:
       81df8f47c31648f379b0493bd746c293
       b815f4e94cbeb5530fde3f5b
     tag_0: d746c293b815f4e94cbeb5530fde3f5b

   Accumulator:
     contrib_0:
       c80f4a347e6e74a58d3ee8b25bb5a3f9
       982321de03593fbab7b471d99e2cf1bc
     accumulator:
       c80f4a347e6e74a58d3ee8b25bb5a3f9
       982321de03593fbab7b471d99e2cf1bc
```

B.7. ChaCha20-Poly1305 Vector

Single segment with random nonce mode and ChaCha20-Poly1305. Same CEK and salt as above; different AEAD_id changes the payload_info and all derived values.

```
   Inputs:
     AEAD:         chacha20-poly1305
     nonce_mode:   random
     (CEK, salt, plaintext, nonce as single-segment)

   commitment:
     1e30998c28c0224cca320e5ba27f8514
     d232b9e58f1df3dccffff903c5efedfd
   payload_key:
     12a66095dccb074137667f5f6fe9fc41
     0943dba7b9fdea052828609297ecb897
   acc_key:
     985ce823be86c332e410d30066cbd9f1
     1a9a840b8d691adda468ecbea988e2eb

   Segment 0 (is_final=1):
     nonce_0:      030303030303030303030303
     ct+tag:
       ff7ac17f504ffc08032b100aaa2ee764
       25e9128c8ff9d6ed8b66dc08
     tag_0: aa2ee76425e9128c8ff9d6ed8b66dc08

   Accumulator:
     accumulator:
       58babbc3e19ebdfc7e88bde91b8a9e3b
       42fc8f0090892783648761ad6cec65ed
```

B.8. AES-256-GCM-SIV Vector

Single segment with derived nonce mode and AES-256-GCM-SIV. epoch_length is absent (required for MRAE AEADs).

```
   Inputs:
     AEAD:         aes-256-gcm-siv
     nonce_mode:   derived
     (CEK, salt, plaintext as single-segment)

   commitment:
     5d6d5c00c15b2a6bf44f28cedd1b99f4
     35b0f51085470b2c5f5b9a4a2fe17cc9
   payload_key:
     ce2969d3b94dc1c4b173d3c1baf37de0
     b1a1a5fece2bcea662ba6fe284a8c0a8
   nonce_base:
     ef1630c621ebbe963a18ab66

   nonce_0 = nonce_base XOR uint64(0):
     ef1630c621ebbe963a18ab66

   Segment 0 (is_final=1):
     ct+tag:
       12c611b3a380d5474ea9af7686f2ca90
       63b34086d29e41bdfccb08f4
     tag_0: 86f2ca9063b34086d29e41bdfccb08f4

   Accumulator:
     accumulator:
       e131f4c66daf6b7c6300e190325a164a
       6058daf07d76670ebb1cfcdce937f97c
```

B.9. Rewrite Vector

This vector demonstrates accumulator update when segment 0 of the two-segment AES-256-GCM message is rewritten with new plaintext. The accumulator is updated by XOR-ing out the old contribution and XOR-ing in the new one.

```
   Original message:
     segment_0:    "Block zero data!" (16 octets)
     segment_1:    "Final block." (12 octets, final)
     accumulator:
       af61d439153493369b955825c61d34ad
       cacc0e269008650f90ce779633929599

   Rewrite segment 0:
     new_plaintext: "Updated data!!!!" (16 octets)
     new_nonce:     090909090909090909090909
     new_ct+tag:
       050fa5774cdfd95c94bec167dcf2a7d0
       daf41e183622c7fb6aeb355652f6c050
     new_tag:
       daf41e183622c7fb6aeb355652f6c050

   Accumulator update:
     old_contrib_0:
       a61d5e6bcb37211246d6ac546f29262f
       9f39c690462bce8834a1292e0f55937a
     new_contrib_0:
       83ef8c0d86c63f63ce507723ca44d46c
       d2755468d6923a5f5b0b8ae1860fddfa
     new_accumulator =
       old_acc XOR old_contrib_0 XOR new_contrib_0:
       8a93065f58c58d47131383526370c6ee
       87809cde00b191d8ff64d459bac8db19
```

B.10. Segment Size 16384 Vector

This vector uses segment_size = 16384 with AES-256-GCM. The same CEK, salt, plaintext, and nonce as the single-segment vector above are used; the different segment_size changes payload_info and all derived values.

```
   Inputs:
     AEAD:         aes-256-gcm
     segment_size: 16384
     nonce_mode:   random
     (CEK, salt, plaintext, nonce as single-segment)

   payload_info (hex):
     000b6165732d3235362d67636d000531
     3633383400077368612d323536002004
     0404040404040404040404040404040404
     040404040404040404040404040404

   commitment:
     3670f64513fa362f5ed8881ee41bba09
     e3e8c9d69f92f1018671c00995546022
   payload_key:
     30039f0450af5845f73eb170549cb81d
     30327157c72727ae6bfa4deca5bc11d4
   acc_key:
     d429ed408c97218e051141cd1a215086
     2cf799dda14eafd1e18920fbf06632fc

   Segment 0 (is_final=1):
     nonce_0:      030303030303030303030303
     segment_aad:
       0009726141452d44415441
       000800000000000000000000101
     ct+tag:
       7ecae9c12c31383e27f074c2cc735c19
       0d91f5fbb4b9b40f87608a97
     tag_0: cc735c190d91f5fbb4b9b40f87608a97

   Accumulator:
     contrib_0:
       66c8f92ec5341ae4fad08afdb3f509e1
       2e92cae583bd6b90a2f77fb75b4419fd
     accumulator:
       66c8f92ec5341ae4fad08afdb3f509e1
       2e92cae583bd6b90a2f77fb75b4419fd
```

B.11. Multi-Segment Full-Size Vector

This vector encrypts two full 65536-octet segments. It demonstrates the segment_aad is_final flag, per-segment AEAD, and accumulator XOR across full-size blocks. Segment 0 is 65536 octets of 0x00; segment 1 is 65536 octets of 0x01.

```
   Inputs:
     protocol_id:  "raAE-v1"
     CEK:          aaaaaaaa...aa (32 octets of 0xAA)
     salt:         04040404...04 (32 octets of 0x04)
     segment_0:    0x00 * 65536 (65536 octets)
     segment_1:    0x01 * 65536 (65536 octets, final)
     nonce_0:      030303030303030303030303
     nonce_1:      050505050505050505050505
     AEAD:         aes-256-gcm
     segment_size: 65536
     KDF:          sha-256

   Payload Schedule (same as single-segment):
     commitment:   (same as single-segment vector)
     payload_key:  (same as single-segment vector)
     acc_key:      (same as single-segment vector)

   Segment 0 (is_final=0):
     nonce_0:      030303030303030303030303
     segment_aad:
       0009726141452d44415441
       000800000000000000000000100
     ct (first 16): 832455931b9ac90eff6fcffab78f7573
     ct (last 16):  60aefecea60d483670e82d15030da101
     tag_0: 2ae0e657af52f40b5a97716e809727fb

   Segment 1 (is_final=1):
     nonce_1:      050505050505050505050505
     segment_aad:
       0009726141452d44415441
       000800000000000000001000101
     ct (first 16): e6686cf9184198d944be50a2cb6acef2
     ct (last 16):  cb929c96667c24ce1822d1c88d5613cb
     tag_1: 8a148be124e0f085638e81a4cc2c947a

   Accumulator (two segments):
     contrib_0:
       6670594c17d70d9ed935408cd3a07f93
       e599f389cef9d26003af30423b07c460
     contrib_1:
       b221f9b0b2ad7eb446842b22a7e80600
       b393e94f27a48e6e4e2e155dedf11b46
     accumulator:
       d451a0fca57a732a9fb16bae74487993
       560a1ac6e95d5c0e4d81251fd6f6df26
       (accumulator = contrib_0 XOR contrib_1)
```

B.12. AEGIS-256 Vector

Single segment with random nonce mode and AEGIS-256. Same CEK and salt as above. AEGIS-256 uses 32-octet nonces.

Inputs:
  AEAD: aegis-256
  nonce_mode: random
  (CEK, salt, plaintext as single-segment)

payload_info (hex):
  000961656769732d3235360005363535
  333600077368612d3235360020040404
  040404040404040404040404040404
  04040404040404040404040404

commitment:
  93cc15475b3383b353bb908f979cc493
  c271abb4409a1fb9a1588b508fb3ebd9
payload_key:
  041d039530a5c34fb19fee3f719fc4d3
  eefaa28da5df8a8ed24b7df78a2e990e
acc_key:
  37aa2cfb9b79fc8b0f97c327cc1c8bd9
  b9c5b70f3c3bd0ea24480b24a6b34d73

Segment 0 (is_final=1):
  nonce_0:
    030303030303030303030303030303
    030303030303030303030303030303
  ct+tag:
    219cc576e7c5662f8dda048000f22574
    b490f3af2c0b3a2842605dfa
  tag_0: 00f22574b490f3af2c0b3a2842605dfa

Accumulator:
  contrib_0:
    2561aa0b0317d3640be35a3b81edf5a5
    b1efebdeeaad67e253e77b12a8410a3e
  accumulator:
    2561aa0b0317d3640be35a3b81edf5a5
    b1efebdeeaad67e253e77b12a8410a3e

B.13. AEGIS-256X2 Vector

Single segment with random nonce mode and AEGIS-256X2. Same CEK and salt as above. AEGIS-256X2 uses 32-octet nonces.

```
   Inputs:
     AEAD:         aegis-256x2
     nonce_mode:   random
     (CEK, salt, plaintext as single-segment)

   payload_info (hex):
     000b61656769732d3235367832000536
     3535333600077368612d323536002004
     0404040404040404040404040404040404
     040404040404040404040404040404

   commitment:
     63f577c993f7ba7ed4acfca98366702e
     242c820055f6e67c143bcb2e6a15b87d
   payload_key:
     57e33ccba9081a1332632354af0cb00b
     54fb5a66742aa9e0079c77e49f25afec
   acc_key:
     96e4b420589f9fbd2103fb995372d91a
     8a5b5b6ae03425b5b6952b1ac792dea5

   Segment 0 (is_final=1):
     nonce_0:
       030303030303030303030303030303
       030303030303030303030303030303
     ct+tag:
       cba698d2ed783f03ef1c10839c7ce962
       94644faf13fdb98843f61457
     tag_0: 9c7ce96294644faf13fdb98843f61457

   Accumulator:
     contrib_0:
       768297bffc6313db1059ad3e714fdaad
       6386689b2dc4dc9aa87d250df622aad6
     accumulator:
       768297bffc6313db1059ad3e714fdaad
       6386689b2dc4dc9aa87d250df622aad6
```

Author’s Address

Nick Sullivan
Cryptography Consulting LLC
Email: nicholas.sullivan+ietf@gmail.com