Crypto Forum
Internet Draft
Intended status: Informational
Expires: March 16, 2026

Shay Gueron
University of Haifa and Meta

September 16, 2025

Double Nonce Derive Key AES-GCM (DNDK-GCM)

draft-gueron-cfrg-dndkgcm-03

Abstract

This document specifies an authenticated encryption algorithm called Double Nonce Derive Key AES-GCM (DNDK-GCM). It operates with a 32-byte root key and is designed to encrypt with a 24-byte random nonce and optionally to provide for key commitment.

Encryption takes the root key and a 15-byte portion of the random nonce, and derives a fresh 32-byte encryption key and (optionally) a key commitment value. Then, it invokes AES-GCM with the derived key and the remaining bytes of the nonce, and outputs the ciphertext, authentication tag and the key commitment value.

Although this is not the primary use case, it is also possible to use DNDK-GCM with a non-repeating but non-random nonce (i.e., a "counter-based nonce").

The low collision probability in a collection of 24-byte random nonces and the per-nonce derivation of an encryption key extend the lifetime of the root key, and the scheme can support processing up to 2^64 bytes under a given root key.

DNDK-GCM introduces a relatively small overhead compared to using AES-GCM directly, and its security relies only on the standard assumption that AES acts as a pseudorandom permutation.

Status of this Memo

Copyright Notice

in the Revised BSD License.

Table of Contents

1. Introduction

Authenticated Encryption with Additional Data (AEAD) [RFC5116] is a fundamental cryptographic tool providing both confidentiality and integrity in a single scheme. AES-GCM [GCM] is one of the most widely used AEAD schemes, owing to its high performance on modern platforms with native instructions to accelerate AES computations and polynomial multiplications used for authentication.

However, AES-GCM has notable limitations, including nonce length requirements and the absence of key commitment, which have led to the development of alternative approaches.

1.1. Limitation 1: Short Nonces Enforce Frequent Key Rotation

AES-GCM is a nonce-respecting AEAD, meaning that a unique nonce must be used for each message encrypted under the same key. Reusing a nonce for different messages can lead to a severe compromise of both confidentiality and integrity.

Applications can enforce nonce uniqueness by constructing nonces with a counter. However, in many real-world scenarios, maintaining state to track counters is costly or vulnerable to errors. In practice, systems often generate a random nonce for each message under the assumption that the probability of nonce collisions during the key's lifetime is sufficiently small. In this context, NIST's Special Publication 800-38D [GCM] states (Section 8) that:

> The probability that the authenticated encryption function will ever be invoked with the same IV and the key on two (or more) distinct sets of input data shall be no greater than 2^-32.

For AES-GCM in a 12-byte random setting, nonce collision probability in Q calls is at most Q^2/2^97, and this bound is also essentially tight. Upper bounding this probability by 2^-32 limits the lifetime of a key to at most 2^32.5 encryptions. In many scenarios, e.g., for processing data at the cloud scale, this imposes a too-frequent key rotation rate. Note that AES-GCM can consume nonces of arbitrary lengths, but the key rotation limitation is not fully alleviated by using longer than 12 bytes nonces even with a stateful counter, and independently, the birthday bound limits the amount of data that can be encrypted under a single key.

1.2. Limitation 2: Lack of Key Commitment

AES-GCM has the following property: it is possible to find two (or more) keys K1, K2, and two (or more) messages M1, M2, such that encrypting M1 under K1 and encrypting M2 under K2 give the same ciphertext-tag pair. As a result, a receiver decrypting a ciphertext with an unverifiable key might mistakenly accept a plaintext as valid, even if it was crafted by a malicious actor. This lack of key commitment property is not unique to AES-GCM. It applies to other AEADs where authentication uses universal hashing rather than e.g., (less efficient) collision resistant hashing. In some scenarios, lack of key commitment can be a problem [DGRW18], [LGR21], [ADG+22] and adding some mitigation would be useful. Several approaches for adding key commitment are detailed in [ADG+22], and some methods have already been deployed in cloud systems (e.g., AWS Encryption SDK [Gue20]).

1.3. DNDK-GCM Design Goals

DNDK-GCM is designed to address the limitations of AES-GCM by:

a) Minimizing performance overhead compared to AES-GCM.

b) Reusing vetted and optimized AES-GCM implementations to leverage existing hardware instructions (e.g., AES-NI [Gue10] and PCLMULQDQ [GK08]).

c) Relying on widely accepted cryptographic assumptions for security. Namely, that the output of AES (with a uniform random key) is indistinguishable from the output of a uniform random permutation. This is already the assumption that establishes the security of AES-GCM.

2. Requirements Language

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.

3. Preliminaries and Notation

A byte is an integer between 0 and 255, commonly represented by two hexadecimal digits. For example, the decimal value 5 is represented as 0x05 in hexadecimal, and 135 is represented as 0x87.

Let U be an array of bytes of length u. The bytes of U are indexed from 0 to u-1 and denoted as U [j], where j=0, 1, ..., u-1.

Let T = U [a: b] denote the sub-array of U containing bytes from index a to b (inclusive), where 0 <= a <= b <= (u-1). The length of T is (b-a+1). For example, if U = [0x01, 0x02, 0x03, 0x04], then U [1: 3] = [0x02, 0x03, 0x04]. For a = b, T = U [a: a] is U [a] and T = U [0: u-1] implies T = U. If a = b+1, the sub-array U [a: b] is defined as empty.

Note: In this document, the two boundary indices in T [a: b] are inclusive, which differs from conventions in some (not all) programming languages, such as Python, where the upper boundary is exclusive.

Let U0 be an array of u0 bytes and U1 an array of u1 bytes. Suppose that 0 <= a1 <= b1 <=(u1-1), 0 <= a0 <= b0 <= (u0-1), and that b0 - a0 = b1 - a1 = s. Then, the notation U1 [a1: b1] = U0 [a0: b0] is an assignment that sets U1 [a1 + j] = U0 [a0 + j], j = 0, ..., s.

Concatenation of arrays is denoted by "||". If U1 is an array of u1 bytes and U0 is an array of u0 bytes then U0 || U1 is the array W of u0+u1 bytes whose bytes are W [j] = U0 [j], j = 0, ... u0-1 and W [j] = U1 [j-u0], j = u0, ..., u0+u1-1. For example, if U0 = [0x01, 0x02] and U1 = [0x03, 0x04], then U0 || U1 = [0x01, 0x02, 0x03, 0x04].

If U0 and U1 are arrays of u bytes, then the XOR of U0 and U1, denoted by U0 XOR U1, is an array U of u bytes that is the result of the bitwise exclusive OR operation of their corresponding bytes of U0 and U1. For example, if U0 = [0x3a, 0xff, 0x00] and U1 = [0xc3, 0x0f, 0xaa], then 0x3a XOR 0xc3 = 0xf9, 0xff XOR 0x0f = 0xf0, 0x00 XOR 0xaa = 0xaa and U0 XOR U1 is U = [0xf9, 0xf0, 0xaa] .

For an integer s, (0x00)^s denotes an array of s zero bytes. If s = 0, (0x00)^0 is defined as the empty string. An array of 16 bytes is also called a "block".

Note that the bytewise specification of an array is agnostic to the way that the array is listed as a sequence (of bytes), i.e., whether byte 0 is in the rightmost or the leftmost position. The byte positions in the sequence need to be specified unambiguously.

Example. Let U0 be an array of length u0 = 16 (block), listed as follows:

```
    byte index (position)      000102030405060708090101112131415
    U0 =                       3dfcc48e6cf4e1d25a8ff0de58c97a69
```

(i.e., U0 [00] = 0x3d, U0 [07] = 0xd2, U0 [15] = 0x69). Then,

```
    byte index (position)      000102030405060708090101112131415
    U0 [00: 15] =              3dfcc48e6cf4e1d25a8ff0de58c97a69
    U0 [00: 07] =              3dfcc48e6cf4e1d2
    U0 [08: 15] =              5a8ff0de58c97a69
    U0 [12: 14] =              58c97a
    U0 [13: 13] =              c9
    U0 [01: 00] =
```

Let U1 be an array of 16 bytes. The assignment U1 [4: 6] = U0 [12: 14] determines 3 bytes of U1 as follows:

```
    byte index (position)      000102030405060708090101112131415
    U1 =                       ********58c97a******************
```

(the unspecified bytes are denoted by "**")

The symbol FAIL is used here to indicate a failed integrity check. The symbol != is the "not-equal" operator.

Here, AES refers to the Advanced Encryption Standard [AES], specifically, to the version with a 256-bit key (equivalently 32-byte key) AES256. If K (key) is an array of 32 bytes and Z is an array of 16 bytes, then C = AES (K, Z) is the 16-byte array that results from encrypting Z with AES under the key K.

Here, AES-GCM refers to the Authenticated Encryption with Additional Data (AEAD) scheme standardized in [GCM]. AES-GCM encryption and

decryption are denoted AES-GCM.Enc and AES-GCM.Dec, respectively. They take the tuples (N, A, P) and (N, A, C, T), respectively, where N is a nonce, A is the Additional Authenticated Data (AAD), P is the plaintext message, C is the ciphertext, and T is the authentication tag. Here, the inputs and outputs are assumed to be arrays of bytes, K is an array of 32 bytes, N is an array of 12 bytes, T is an array of 16 bytes, A is an array of at most 2^61 - 1 bytes and P (and C) is an array of at most 2^36 - 32 bytes. AES-GCM.Enc outputs the pair (C, T), which explicitly identifies the ciphertext and the tag. Equivalently, it is possible to view the concatenation of C, T (i.e., (C || T) as a single array that is called "ciphertext-blob" (and parsed accordingly). From this perspective, AES-GCM.Enc produces a single output (ciphertext-blob), while AES-GCM.Dec takes three inputs (N, A, ciphertext-blob).

Note (serialization): This document does not mandate a specific wire format for {N, C, T, KC}. Decryption takes inputs (K, N, A, ciphertext-blob), where ciphertext-blob is C || T (and, when KC is present, C || T || KC). Protocols MAY carry N separately from the ciphertext-blob and MAY arrange the fields in any order, provided the receiver can parse them unambiguously (e.g., some protocols may place KC first to enable early commitment checks while C and T are in transit).

4. DNDK-GCM Definition

4.1. Input and Output Ranges

DNDK-GCM is defined with the length and range parameters K_LEN, N_LEN, A_MAX, P_MAX, C_MAX, T_LEN, and KC_LEN.

- K_LEN: Length of the root key (in bytes).

- N_LEN: Length of the nonce (in bytes).

- A_MAX: Maximum length of Additional Authenticated Data (AAD).

- P_MAX: Maximum length of plaintext.

- C_MAX: Maximum length of ciphertext.

- T_LEN: Length of the authentication tag (in bytes).

- KC_LEN: Length of the key commitment value (in bytes).

Encryption is denoted by DNDK-GCM.Enc and decryption is denoted by DNDK-GCM.Dec. These procedures are defined for inputs that are arrays of bytes.

DNDK-GCM.Enc takes as input a tuple (K, N, A, P) where K is the secret (root) key, N is the (randomly selected) nonce, A is the AAD, and P is the plaintext message. It outputs ciphertext C, an authentication tag T and a commitment value KC (that may optionally be an empty array). In this notation, C, T, KC are explicitly identified and named, and the length of C is the same as the length of P.

Equivalently, one may view the concatenation C || T (and, when KC is present, C || T || KC) as a single array, called the "ciphertext-blob" (to be parsed accordingly with the matching offsets for C, T, and possibly KC). Under this view, DNDK-GCM.Enc has a single output (namely ciphertext-blob) and the byte-length of ciphertext-blob equals to the length of P plus 48 when KC is not empty, and equals to the length of P plus 16 when KC is empty.

DNDK-GCM.Dec takes a tuple (K, N, A, C, T, KC), where K is the root key, N is the nonce, A is the AAD, C is the ciphertext, T is the authentication tag and KC is the commitment value. It outputs a

plaintext message P with the same byte-length as C, or a failure indication (FAIL).

Equivalently, if the concatenation of C, T, KC is viewed as a single array "ciphertext-blob", DNDK-GCM.Dec has four inputs (namely K, N, A, ciphertext-blob).

The root key (K) MUST consist of K_LEN bytes. It MUST be generated in a uniformly random (or pseudorandom) way and known only to the parties (sender and recipient) using the scheme.

The nonce (N) MUST consist of N_LEN bytes preferably selected uniformly at random for every encryption. The AAD (A) MUST consist of at most A_MAX bytes, the message (P) MUST consist of at most P_MAX bytes, the ciphertext (C) MUST consist of at most C_MAX bytes, the authentication tag (T) MUST consist of T_LEN bytes, and the key commitment value (KC) MUST consist of KC_LEN bytes. The arrays A, P, C may be empty.

Inputs to DNDK-GCM.Enc and to DNDK-GCM.Dec are assumed to be valid and within the defined range of lengths (i.e., DNDK-GCM.Enc and DNDK-GCM.Dec check the inputs and abort if they are invalid).

Tag truncation: The AES-GCM specification [GCM] (Section 5.2.1.2) permits 16-byte tags to be truncated to 12 bytes or even shorter tags of 8 or 4 bytes, where these lengths are accepted subject to the usage constraints outlined in Appendix C of [GCM]. For simplicity and brevity, this document defines DNDK-GCM only with a 16-byte tag and implicitly ignores a tag truncation option (even to the innocuous truncation to 12 bytes). However, DNDK-GCM usages that require (and settle with) a shorter tag of d < 16 bytes, MAY truncate it to d = 12, 8, or 4 bytes as specified in [GCM]. The formatting and agreement of such truncation is left for the communicating parties or protocol.

4.2. DNDK-GCM Context and Options

The DNDK-GCM context is KC_Choice (either 0 or 1) and LN (12 <= LN <= 27). These parameters determine the scheme’s configuration as follows:

The choice KC_Choice = 0 indicates that a commitment value (KC) is neither computed nor output by DNDK-GCM.Enc and is not required as input to DNDK-GCM.Dec. Equivalently, KC is an empty byte array.

The choice KC_Choice = 1 indicates that a KC value is computed, KC is part of the DNDK-GCM.Enc output and KC is a required input for DNDK-GCM.Dec.

LN is the byte-length of the nonce.

A DNDK-GCM root key MUST be used with only one configuration that is pre-agreed by the communicating parties that use the scheme.

4.3. The Preamble: Derive-L1-LN-L2-L3

The algorithm "Derive-L1-LN-L2-L3" that takes a root key (K) of L1 bytes and a nonce (N) of LN bytes (12 <= LN <= 27). It outputs a derived key (DerivedKey) of L2 bytes and (optionally, depending on KC_Choice) a key commitment value (KeyCommit) of L3 bytes. This document sets the parameter values to L1 = L2 = L3 = 32.

For brevity, Derive-32-24-32-32 is referred to as "Derive".

If LN < 27, Derive appends (27-LN) zero bytes to N to define a 27-byte array named NPadded. If LN = 27, NPadded = N.

The sub-array of the first 15 bytes of NPadded is called NHead, and the sub-array of the remaining 12 bytes is called NTail.

Derive uses the 15 bytes of NHead and "ConfigByte", a byte that encodes KC_Choice and LN. It applies the XORP pseudorandom function (defined in [Iwa06], section 3, under the name ’F’) with the root key K and the block NHead || ConfigByte and produces a 32-byte DerivedKey and (optionally) a 32-byte KeyCommit.

ConfigByte is defined by

    ConfigByte = 128 * KC_Choice + 8 * (LN - 12)

ConfigByte is divisible by 8 so its three least significant bits are zero. Thus, for all 0 <= w <= 7, the bytes ConfigByte and (ConfigByte + w) have the same five most significant bits.

For example, with KC_Choice = 1 and LN = 24, ConfigByte = 0xe0 (=224) and (ConfigByte + 4) = 0xe4 (=228). With KC_Choice = 0 and LN = 24, ConfigByte = 0x60 (=96) and (ConfigByte + 3) = 0x63 (=99).

Derive also produces NTail as the 12-byte value GCM_IV that is subsequently used for the AES-GCM call.

Algorithm 1 defines Derive.

```
    Algorithm 1. Derive (K, N)
    Input: K (root key), N (array of LN bytes)
    Context: KC_Choice, LN
    LN = len (N)
```

```
NPadded = N || (0x00)^(27-LN)
NHead = NPadded [0: 14]
NTail = NPadded [15: 26]
GCM_IV = NTail

ConfigByte = 128 * KC_Choice + 8 * (LN - 12)

B0 = NHead || (ConfigByte + 0)
B1 = NHead || (ConfigByte + 1)
B2 = NHead || (ConfigByte + 2)
if KC_Choice = 1:
    B3 = NHead || (ConfigByte + 3)
    B4 = NHead || (ConfigByte + 4)
X0 = AES (K, B0)
X1 = AES (K, B1)
X2 = AES (K, B2)
if KC_Choice = 1:
    X3 = AES (K, B3)
    X4 = AES (K, B4)
Y1 = X1 XOR X0
Y2 = X2 XOR X0
DerivedKey = Y1 || Y2

KeyCommit = (0x00)^0
if KC_Choice = 1:
    Y3 = X3 XOR X0
```

```
        Y4 = X4 XOR X0
        KeyCommit = Y3 || Y4
    Output: (GCM_IV, DerivedKey, KeyCommit)
+-------------------------------------------------------------------+
```

4.4. Encryption

DNDK-GCM.Enc receives the tuple (K, N, A, P) as input. It first runs the preamble Derive over K and N to compute GCM_IV, a derived key (DK) and if KC_Choice = 1, a 32-byte key commitment value (KC). It then invokes AES-GCM.Enc using DK as the encryption key, the 12-byte nonce GCM_IV, A, and P. This produces the ciphertext C with the same length as P and the authentication tag T. The output is C, T, KC (or, equivalently, C || T || KC) if viewed as a single array called ciphertext-blob. Recall that if KC_Choice = 0, then KC is an empty array. This allows for writing the output of DNDK-GCM.Enc as C, T, KC (equivalently C || T || KC) for both values of KC_Choice.

DNDK-GCM.Enc is defined by Algorithm 2.

```
+-------------------------------------------------------------------+
    Algorithm 2. DNDK-GCM.Enc
    Input: K, N, A, P
    Context: KC_Choice, LN
    (GCM_IV, DK, KC) = Derive (K, N)
    (C, T) = AES-GCM.Enc (DK, GCM_IV, A, P)
    Output: C, T, KC
+-------------------------------------------------------------------+
```

4.5. Decryption

DNDK-GCM.Dec receives the tuple (K, N, A, C, T, KC) as input (or equivalently, C || T || KC as a ciphertext blob). It first runs the preamble "Derive" on K and N to compute GCM_IV, a derived key (DK), and, if KC_Choice = 1, a 32-byte key commitment value candidate (KC'). If KC != KC', decryption fails. In this case, DNDK-GCM.Dec terminates immediately without executing AES-GCM.Dec and outputs FAIL.

Alternatively, to avoid an early exit, DNDK-GCM.Dec can proceed with AES-GCM.Dec but force a failure indication FAIL, regardless of the decryption result. In both cases, DNDK-GCM.Dec outputs FAIL if KC != KC'.

AES-GCM.Dec is then invoked using DK as the decryption key, GCM_IV (a 12-byte nonce), A, C, and T. This process either retrieves a message P (with the same length as C) or returns an authentication failure indication. The final output of DNDK-GCM.Dec is P (only if both KC = KC' and AES-GCM authentication succeeds) or FAIL.

Notably, AES-GCM.Dec does not release any part of P until full authentication is successful (FAIL not returned at any AES-GCM.Dec stage).

The early-exit variant of DNDK-GCM.Dec is defined in Algorithm 3.

```
+--------------------------------------------------------------+
    Algorithm 3. DNDK-GCM.Dec
    Input: K, N, A, C, T, KC
    Context: KC_Choice, LN
    (GCM_IV, DK, KC') = Derive (K, N)
    If KC' != KC output FAIL and abort
    P / FAIL = AES-GCM.Dec (DK, GCM_IV, A, C, T)
    Output: P or FAIL
```

+--+

Appendix A provides step-by-step worked examples with intermediate values. Appendix B provides a small Python script that reproduces the Appendix A vectors (it does not print values; the byte strings are listed in Appendix A). Appendix C provides a compact, self-contained Python reference for DNDK-GCM encryption and decryption, with input-range checks and a constant-time key-commitment check; it prints inputs and final outputs only and includes a small demo for Appendix A1.

Note about comparison and side channels: The comparison of the received key commitment value (KC) with the recomputed value (KC') MUST be performed in a manner that is free from side-channel leaks, especially timing attacks. For example, if an implementation compares KC and KC' byte by byte and exits at the first mismatch, an adversary could guess the correct value of KC one byte at a time. To prevent this, the comparison should be performed in constant time.

4.6. DNDK-GCM Parameters

DNDK-GCM is defined with the following parameter values: K_LEN = 32, A_MAX = 2^61 - 1, P_MAX = 2^36 - 32, C_MAX = 2^36 - 32, T_LEN = 16, and KC_LEN = 32.

Although Algorithms 1, 2, 3 are defined for any N_LEN = LN such that 12 <= LN <= 27, DNDK-GCM is constrained here only to the two options, N_LEN = LN = 24 and N_LEN = LN = 12.

4.7. Using a Random Nonce and a Counter Nonce

DNDK-GCM is a nonce respecting AEAD and therefore, for a given root

key, DNDK-GCM.Enc must use unique nonces.

The main goal of the DNDK-GCM design is to support the random nonce scenario where the LN-byte nonce N is chosen uniformly at random. The use of sufficiently long random nonces (e.g., LN = 24) makes the nonce collision probability below the maximum acceptable threshold (e.g., 2^-32) during the allowed lifetime of the root key. When LN = 12, the nonce collision probability is the same as the nonce collision probability of AES-GCM (with a 12-byte nonce).

DNDK-GCM can be used in the non-repeating but non-random nonce selection ("counter nonce") scenario. For such usage, it is RECOMMENDED to construct the nonce N such that the values of NHead = NPadded [0: 14] never repeat. Such a nonce selection is called "unique NHead nonce".

Since the usage of a counter nonce requires maintaining a state, there should be no difficulty with using unique NHead nonces. A possible example, for the case of LN = 24, is to set the nonce for the i-th invocation of DNDK-GCM.Enc, to

```
Ni = encode (i, 12) || (0x00)^(LN-12)
```

4.8. Usage Limits

This section describes the limits on the DNDK-GCM usage with a given root key.

In the random nonce settings: for LN = 24, a given root key can be used for encrypting as many as 2^64 messages while processing up to 2^64 plaintext blocks. For LN = 12, a given root key can be used for encrypting at most 2^32.5 messages.

In the unique NHead counter nonce settings: for LN = 24 and LN = 12, a given root key can be used for encrypting as many as 2^64 messages

while processing up to 2^64 plaintext blocks.

5. AEAD

We define four AEADs, in the format of [RFC5116], that use DNDK-GCM, namely AEAD_DNDK_GCM_LN_24_KC_1, AEAD_DNDK_GCM_LN_24_KC_0, AEAD_DNDK_GCM_LN_12_KC_1, and AEAD_DNDK_GCM_LN_12_KC_0.

The common parameters for these four AEADs are as follows: K_LEN = 32, A_MAX = 2^61 - 1, P_MAX = 2^36 - 32, C_MAX = 2^36 - 32, T_LEN = 16.

They differ in the parameters N_LEN and KC_LEN as follows:

For AEAD_DNDK_GCM_LN_24_KC_1, N_LEN = 24, and KC_LEN = 32.

For AEAD_DNDK_GCM_LN_24_KC_0, N_LEN = 24, and KC_LEN = 0.

For AEAD_DNDK_GCM_LN_12_KC_1, N_LEN = 12, and KC_LEN = 32.

For AEAD_DNDK_GCM_LN_12_KC_0, N_LEN = 12, and KC_LEN = 0.

6. Security Considerations

6.1. Performing The Checks During Decryption

DNDK-GCM decryption involves two checks:

Check 1. matching the input key commitment value with the computed key commitment value.

Check 2. matching the input authentication tag when AES-GCM.Dec is invoked.

Both checks must be performed, and no part of the plaintext should be released until they both pass. An implementation of AES-GCM.Dec MAY continue the computations even if the first check fails, to release the failure indication only after executing AES-GCM.

Note that the first check is bypassed if KC_Choice = 0 (i.e., the most significant bit of ConfigByte equals 0), indicating no key commitment is required.

6.2. The Key Commitment

To find two (adversarial) root keys K1, K2, and two nonce-AAD-plaintext triples N1, A1, P1, N2, A2, P2, such that DNDK-GCM.Enc (K1, N1, A1, P1) = DNDK-GCM.Enc (K2, N2, A2, P2), an adversary needs to match the 32-byte key commitment values as defined by Derive.

6.3. Security Bounds for DNDK-GCM

Suppose that DNDK-GCM is used for encrypting Q <= 2^64 messages (Nj, Aj, Mj) where Mj consists of Lj <= Lm blocks and Aj consists of at most La blocks, j = 1, ... , Q. Assume that the PRP advantage of AES (in this multikey setting) is negligible for such Q and Lm.

Derive is an instantiation of the XORP construction [Iwa06] with parameter w <= 4, which is run over

NHeadj = NPaddedj [0: 14] || ConfigByte

where

NPaddedj = Nj || (0x00)^(27-LN)

The distinguishing advantage of an adversary observing XORP samples over at most Q distinct NHeadj values is at most w^2 Q/2^(128), which, for w <= 4 is at most Q/2^(124) and is negligible for Q <= 2^64 (see [IMC16]).

Up to this advantage, the sequence of (at most Q) derived keys from (at most Q) distinct NHeadj values is indistinguishable from a sequence of (at most Q) uniform random 32-byte values. The probability to encounter a collision in a uniform random 32-byte sequence is at most Q^2/2^(257) <= 2^(-129).

In the unique NHead nonce setting, all the values of NHeadj are, by definition, distinct. Thus, up to the probability of 2^(-129) plus the XORP distinguishing advantage, the derived keys are distinct. Hence, in this scenario every derived key is used for encrypting exactly one message.

In the random nonce scenario with LN = 24, the probability of a nonce collision is at most Q^2/2^(193) <= 2^(-65). Here, the NHeadj values are not necessarily distinct when Q <= 2^(64) is large, but the probability to encounter a 3-way collision across the NHeadj values (i.e., 0 <= i < j < k <= Q such that NHeadi = NHeadj = NHeadk) is at most Q^3/(6 * 2^(240)) <= 2^(-50). Therefore, with a probability of at most 2^(-65) + 2^(-50) (in addition to the XORP distinguishing advantage), every derived key is used to encrypt at most two messages. In fact, for large Q, the vast majority of the derived keys are used for encrypting only one message.

It follows that (up to some small probability), DNDK-GCM can be viewed as AES-GCM in a multi-key scenario, where every key is used for encrypting at most two messages (of length <= Lm blocks).

Note that if Nj != Nk for some 1 <= j < k <= Q and NHeadj = NHeadk, then NTailj != NTailk. This implies that AES-GCM is invoked with the same derived key, but with distinct GCM_IV values.

Under these conditions, the distinguishing advantage of passively viewed ciphertext-tag(-Commitment) tuples from chosen inputs, and uniform random strings of the matching lengths, is dominated by

$$Q \quad (Lm + 2)^2 / 2^{128}$$

This upper bound can be further refined.

For integrity, an adversary that views $Q <= 2^{64}$ encryption results and tries a total of $Q' <= 2^{64}$ forgery attempts has a forgery success probability of at most

$$Q' * (Lm + La + 1) / 2^{127}$$

A detailed analysis is available in [GR25].

7. Design Rationale

The main goal of the DNDK-GCM design is to address the short nonce limitation of AES-GCM in the random nonce setting. To this end, DNDK-GCM supports a double-length nonce of 24 bytes, where the probability to encounter a nonce collision in Q uniform random samples is upper bounded by $Q^2/2^{193}$, which is negligible for $Q <= 2^{64}$.

The DNDK-GCM design extends the Derive-Key approach of [GL17], where the Derive function and the underlying AEAD share the same nonce, as seen in AES-GCM-SIV [RFC8452], which utilizes a 12-byte nonce. To enable using a 16-byte permutation (AES) as a primitive, Derive operates over a sub-array of NPadded, namely NHead, that is strictly shorter than a single block. The DNDK-GCM design chooses the longest possible such subarray, i.e. a 15-byte NHead. It populates a block with NHead and uses the remaining byte for encoding KC_Choice, LN, and while leaving 3 bits for a counter (counting from 0 to 2 or 4). The remaining 12 bytes of NPadded (NTail) are used as the AES-GCM IV; AES-GCM is invoked with the derived key (DK), and this 15|12 split is

fixed by definition and does not depend on LN.

In the random nonce setting, splitting NPadded into a 15-byte NHead and a 12-byte NTail minimizes the key collision and the 3-way key collision probabilities. This lowers the upper bounds on the distinguishing advantage of DNDK-GCM outputs from uniform random strings and allows for processing a very large number of messages and a very large amount of data under a given root key.

In the counter nonce setting, DNDK-GCM is a nonce respecting AEAD. Note that invocations of DNDK-GCM.Enc can choose to fix the 15-byte NHead and place a counter in NTail. This would fix the derived key and yield security bounds that are similar to those of AES-GCM (with a counter nonce). However, this is a non-optimal way to construct counter nonces and much better bounds are achieved in the unique NHead nonce settings where all the NHead values are, by definition, distinct.

With the choice LN = 12 and the counter nonce setting, DNDK-GCM reduces the upper bounds on the ciphertext indistinguishability compared to plain AES-GCM. This may justify the Derive overhead while using only 12-byte nonces. In the random nonce setting, both AEADs have the same limitations on the amounts of processed data and number of messages. In this case, DNDK-GCM is defined only for completeness.

DNDK-GCM employs the Beyond-Birthday-Bound (BBB) pseudorandom function XORP [Iwa06], which requires 5 AES calls (or 3 without key commitment) to output the required 64 bytes (or 32 bytes without key commitment). A non-BBB Derive based on counter mode might also suffice. Concretely, this counter-mode variant can be obtained by fixing X0 := (0x00)^16 in Algorithm 1 (and omitting its computation in implementations). This variant reduces the AES call count to 4 (or to 2 when KC_Choice = 0) but samples derived keys from a proper subset of the 32-byte key space (i.e., not uniform over all 2^256 values), and its security would require an additional non-standard

assumption about AES.

Incorporating KC_Choice and the key commitment into the configuration byte ConfigByte = 128 * KC_Choice + 8 * (LN - 12) creates a domain separation that leads to different derived keys per configuration.

Domain separation: The scheme is specified so that whether or not a key commitment (KC) is computed, is part of the agreed configuration between the sender and the recipient. Nevertheless, this choice is also encoded in the ConfigByte, and thus impacts the derived AES-GCM key. To argue why such a domain separation is desirable, we note that this feature provides a sort of cryptographic verification that the sender and recipient have actually agreed on the configuration. This may help detect mismatches, for example if encryption is computed with KC_Choice = 1 but KC is omitted from the blob that is submitted for decryption.

7.1. The DNDK-GCM overheads

This section describes the overheads of DNDK-GCM compared to AES-GCM (with a 32-byte key).

The function Derive requires 5 calls to AES256 with the root key if KC_Choice = 1, and 3 calls if KC_Choice = 0. These AES256 computations are executed over independent blocks and can be parallelized for better performance.

In addition, every encryption (and decryption) requires an AES256 key expansion with the fresh derived key (and other initialization steps for AES-GCM). This computational overhead is independent of the nonce length (LN). The total overhead is relatively small (see [GR25] for a full account).

8. IANA Considerations

IANA is requested to create a new registry entitled "DNDK-GCM AEAD Identifiers" in the "Internet Protocol Parameters" group.

Registration Policy: Expert Review (Section 4.5 of [RFC8126]).

Registry Contents:

Each registry entry consists of the following fields:

* Name: an ASCII, case-sensitive identifier that MUST match the pattern AEAD_DNDK_GCM_LN_<N>_KC_<b>, where <N> is one of {12,24} and <b> is one of {0,1}.
* LN: the nonce length in bytes; MUST be either 12 or 24.
* KC_Choice: 1 if key commitment is present; 0 otherwise.
* Reference: one or more stable references to a public specification.

Expert Review Guidance:

Designated Experts SHOULD verify that the Name conforms to the pattern above; that LN and KC_Choice are consistent with the Name; and that the Reference points to a stable, publicly accessible specification of the variant. Experts MUST reject registrations that collide with an existing Name or that are inconsistent with this document.

Initial Assignments:

The following identifiers are to be added at publication time:

Name	LN	KC_Choice	Reference
AEAD_DNDK_GCM_LN_24_KC_1	24	1	this document
AEAD_DNDK_GCM_LN_24_KC_0	24	0	this document
AEAD_DNDK_GCM_LN_12_KC_1	12	1	this document
AEAD_DNDK_GCM_LN_12_KC_0	12	0	this document

Future registrations MUST follow the Expert Review policy defined in [RFC8126].

9. References

9.1. Normative References

[AES] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS 197, November 2001.

The original FIPS 197 standard (2001) was superseded in May 2023; the updated DOI is https://doi.org/10.6028/NIST.FIPS.197-upd1.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST SP 800-38D, DOI 10.6028/NIST.SP.800-38D, November 2007, https://csrc.nist.gov/publications/detail/sp/800-38d/final.

Note: the GCM specification (NIST SP 800-38D) is planned to be revised in the near future.

https://csrc.nist.gov/News/2024/nist-to-revise-sp-80038d-gcm-and-gmac-modes

[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/info/rfc2119>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, June 2017.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://datatracker.ietf.org/doc/html/bcp14>.

9.2. Informative References

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008,

<https://www.rfc-editor.org/info/rfc5116>.

[ADG+22] Albertini, A., Duong, T., Gueron, S., Kolbl, S., Luykx, A., Schmieg, S., "How to Abuse and Fix Authenticated Encryption Without Key Commitment", in Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), pp. 3291-3308, 2022. [Online]. Available: https://www.usenix.org/system/files/usenixsecurity22-albertini.pdf

[DGRW18] Dodis, J., Grubbs, P., Ristenpart, T., Woodage, J., "Fast Message Franking: From Invisible Salamanders to Encryptment", CRYPTO 2018, Proceedings, Part I, Lecture Notes in Computer Science, vol. 10991, pp. 155-186, Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-96884-1_6

[Gue10] Gueron, S., "Intel Advanced Encryption Standard (AES) Instructions Set", Intel White Paper, Rev. 3, 1-94, 2010. [Online]. Available: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf

[Gue20] Gueron, S., "Key Committing AEADs", Cryptology ePrint Archive, Paper 2020/1153, 2020. [Online]. Available: https://eprint.iacr.org/2020/1153

[GK08] Gueron, S., Kounavis, M., "Carry-Less Multiplication and Its Usage for Computing the GCM Mode", Intel White Paper, Rev. 2.02, 1-84, 2014. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/clmul-wp-rev-2-02-2014-04-20.pdf

[GR25] Gueron, S., and Ristenpart, T., "DNDK: Combining Nonce and Key Derivation for Fast and Scalable AEAD", Cryptology

ePrint Archive, Paper 2025/785, 2025. [Online]. Available: https://eprint.iacr.org/2025/785

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

[GL17] Gueron, S. and Y. Lindell, "Better Bounds for Block Cipher Modes of Operation via Nonce-Based Key Derivation", Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017. [Online]. Available: https://doi.org/10.1145/3133956.3133992

[LGR21] J. Len, P. Grubbs, T. Ristenpart, "Partitioning Oracle Attacks," 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021, pp. 195-212. [Online]. Available: https://www.usenix.org/system/files/sec21summer_len.pdf

[Iwa06] Iwata, T., "New Blockcipher Modes of Operation with Beyond the Birthday Bound Security", Fast Software Encryption, FSE 2006, Revised Selected Papers, Lecture Notes in Computer Science, vol. 4047, pp. 310-327, Springer, 2006. [Online]. Available: https://doi.org/10.1007/11799313_20

[IMC16] Iwata, T., Mennink, B., and Vizar, D., "CENC is Optimally Secure," Cryptology ePrint Archive, Paper 2016/1087, 2016. [Online]. Available: https://eprint.iacr.org/2016/1087

Appendix A. DNDK-GCM Encryption Worked-Out Examples

This appendix provides DNDK-GCM encryption examples. Arrays of bytes are typed in increasing order of byte positions from left (byte 0) to right.

A1. A Worked-Out Example (LN = 24; With Key Commitment)

```
==========================================================================
                          DNDK-GCM example
==========================================================================
Nonce length (LN) = 24
KC_Choice =         1
==========================================================================
                                  ---------Bytes Position---------
                                  000102030405060708090101112131415
                                ...161718192021212324252627282930 31
                                  --------------------------------
                                  --------------------------------
Root key =                        010000000000000000000000000000000
                                  000000000000000000000000000000000
--------------------------------------------------------------------------
Input
--------------------------------------------------------------------------
Nonce =                           000102030405060708090a0b0c0d0e0f
                                  1011121314151617
aad (5 bytes) =                   0100000011
plaintext (4 bytes) =             11000001
--------------------------------------------------------------------------
Output
--------------------------------------------------------------------------
```

```
   cipher (4 bytes) =                8eee8a4b
   tag =                             8a1c8d0ceb7e07e3c834cafe75aa001f
   Key commitment value =            2baf00efd298de13055c9a6c39e05aee
                                     571583384357635e144fa21444239968
   ---------------------------------------------------------------------
   ciphertext + tag (20 bytes) =     8eee8a4b8a1c8d0ceb7e07e3c834cafe
                                     75aa001f
   ---------------------------------------------------------------------
   ciphertext + tag + commit
                      (52 bytes) =   8eee8a4b8a1c8d0ceb7e07e3c834cafe
                                     75aa001f2baf00efd298de13055c9a6c
                                     39e05aee571583384357635e144fa214
                                     44239968
   ---------------------------------------------------------------------
   Intermediate values
   ---------------------------------------------------------------------
   DerivedKey =                      3d1480ee39a968d581d16a578bdaf0e6
                                     719dcfff6e127b40bbdd844accea7e1c
   AES-GCM nonce  =                  0f1011121314151617000000
   Nonce PAD length (bytes) =        3
   Padded nonce (NPadded) =          000102030405060708090a0b0c0d0e0f
                                     1011121314151617000000
   ConfigByte
   128 * KC_Choice + 8 * (LN - 12)
   as int =                          224
   in hex =                          e0
   =====================================================================
   B0 =  000102030405060708090a0b0c0d0ee0
   X0 =  cd8a6170df972d556ae1eaf9ea9bd799
   ConfigBytePlus0 =  e0
```

```
   B1 =  000102030405060708090a0b0c0d0ee1
   X1 =  f09ee19ee63e4580eb3080ae6141277f
   ConfigBytePlus1 =  e1
   B2 =  000102030405060708090a0b0c0d0ee2
   X2 =  bc17ae8fb1855615d13c6eb32671a985
   ConfigBytePlus2 =  e2
   B3 =  000102030405060708090a0b0c0d0ee3
   X3 =  e625619f0d0ff3466fbd7095d37b8d77
   ConfigBytePlus3 =  e3
   B4 =  000102030405060708090a0b0c0d0ee4
   X4 =  9a9fe2489cc04e0b7eae48edaeb84ef1
   ConfigBytePlus4 =  e4
   Y1 =  3d1480ee39a968d581d16a578bdaf0e6
   Y2 =  719dcfff6e127b40bbdd844accea7e1c
   DerivedKey (= Y1 || Y2) =
         3d1480ee39a968d581d16a578bdaf0e6
         719dcfff6e127b40bbdd844accea7e1c
   Y3 =  2baf00efd298de13055c9a6c39e05aee
   Y4 =  571583384357635e144fa21444239968
   KeyCommit (= Y3 || Y4) =
         2baf00efd298de13055c9a6c39e05aee
         571583384357635e144fa21444239968
   ------------------------------------------------------------------
```

A2. A Worked-Out Example (LN = 24; No Key Commitment)

```
   ==================================================================
                             DNDK-GCM example
   ==================================================================
```

```
   Nonce length (LN) = 24
   KC_Choice =         0
   ========================================================================
                                       ---------Bytes Position---------
                                       000102030405060708090101112131415
                                    ...161718192021212324252627282930 31
                                       --------------------------------
                                       --------------------------------
   Root key =                          010000000000000000000000000000000
                                       000000000000000000000000000000000
   ------------------------------------------------------------------------
   Input
   ------------------------------------------------------------------------
   Nonce =                             000102030405060708090a0b0c0d0e0f
                                       1011121314151617
   aad (5 bytes) =                     0100000011
   plaintext (4 bytes) =               11000001
   ------------------------------------------------------------------------
   Output
   ------------------------------------------------------------------------
   cipher (4 bytes) =                  7f6e39cc
   tag =                               b61df0a502c167164e99fa23b7d12b9d
   Key commitment value =

   ------------------------------------------------------------------------
   ciphertext + tag (20 bytes) =       7f6e39ccb61df0a502c167164e99fa23
                                       b7d12b9d
   ------------------------------------------------------------------------
   ciphertext + tag + commit
                       (20 bytes) =    7f6e39ccb61df0a502c167164e99fa23
```

```
                                    b7d12b9d


   ---------------------------------------------------------------
   Intermediate values
   ---------------------------------------------------------------
   DerivedKey =                     d974a46fbbeb3dec953ce088ef6b6085
                                    73248947acf51606de5a1e5b72629197
   AES-GCM nonce  =                 0f1011121314151617000000
   Nonce PAD length (bytes) =       3
   Padded nonce (NPadded) =         000102030405060708090a0b0c0d0e0f
                                    1011121314151617000000
   ConfigByte
   128 * KC_Choice + 8 * (LN - 12)
   as int =                         96
   in hex =                         60
   ===============================================================
   B0 =  000102030405060708090a0b0c0d0e60
   X0 =  d3a8349b7cd72d37e061b49c3c5d2c12
   ConfigBytePlus0 =  60
   B1 =  000102030405060708090a0b0c0d0e61
   X1 =  0adc90f4c73c10db755d5414d3364c97
   ConfigBytePlus1 =  61
   B2 =  000102030405060708090a0b0c0d0e62
   X2 =  a08cbddcd0223b313e3baac74e3fbd85
   ConfigBytePlus2 =  62
   Y1 =  d974a46fbbeb3dec953ce088ef6b6085
   Y2 =  73248947acf51606de5a1e5b72629197
   DerivedKey (= Y1 || Y2) =
         d974a46fbbeb3dec953ce088ef6b6085
```

```
      73248947acf51606de5a1e5b72629197
   KeyCommit (= Y3 || Y4) =

   ------------------------------------------------------------------
```

A3. A Worked-Out Example (LN = 12; With Key Commitment)

```
   ==================================================================
                            DNDK-GCM example
   ==================================================================
   Nonce length (LN) = 12
   KC_Choice =         1
   ==================================================================
                                     ---------Bytes Position---------
                                     000102030405060708090101112131415
                                  ...161718192021212324252627282930 31
                                     --------------------------------
                                     --------------------------------
   Root key =                        010000000000000000000000000000000
                                     000000000000000000000000000000000
   ------------------------------------------------------------------
   Input
   ------------------------------------------------------------------
   Nonce =                           000102030405060708090a0b

   aad (5 bytes) =                   0100000011
   plaintext (4 bytes) =             11000001
   ------------------------------------------------------------------
   Output
```

```
   ---------------------------------------------------------------------
   cipher (4 bytes) =                1915d0bd
   tag =                             187b392eeb9b231a57a852db20e02201
   Key commitment value =            675fb3ec6d0e56002333c2504d1b70db
                                     47c3713775999c9600bedcfda76f8d8c
   ---------------------------------------------------------------------
   ciphertext + tag (20 bytes) =     1915d0bd187b392eeb9b231a57a852db
                                     20e02201
   ---------------------------------------------------------------------
   ciphertext + tag + commit
                      (52 bytes) =   1915d0bd187b392eeb9b231a57a852db
                                     20e02201675fb3ec6d0e56002333c250
                                     4d1b70db47c3713775999c9600bedcfd
                                     a76f8d8c
   ---------------------------------------------------------------------
   Intermediate values
   ---------------------------------------------------------------------
   DerivedKey =                      dfde3c721be6e0b0369770788941a293
                                     96c4e50dd81725d3832221fa47d564e1
   AES-GCM nonce  =                  000000000000000000000000
   Nonce PAD length (bytes) =        15
   Padded nonce (NPadded) =          000102030405060708090a0b00000000
                                     0000000000000000000000
   ConfigByte
   128 * KC_Choice + 8 * (LN - 12)
   as int =                          128
   in hex =                          80
   =====================================================================
   B0 =  000102030405060708090a0b00000080
   X0 =  78d5b87c1bf1f2ee7de63d4899309b82
```

```
   ConfigBytePlus0 =  80
   B1 =  000102030405060708090a0b00000081
   X1 =  a70b840e0017125e4b714d3010713911
   ConfigBytePlus1 =  81
   B2 =  000102030405060708090a0b00000082
   X2 =  ee115d71c3e6d73dfec41cb2dee5ff63
   ConfigBytePlus2 =  82
   B3 =  000102030405060708090a0b00000083
   X3 =  1f8a0b9076ffa4ee5ed5ff18d42beb59
   ConfigBytePlus3 =  83
   B4 =  000102030405060708090a0b00000084
   X4 =  3f16c94b6e686e787d58e1b53e5f160e
   ConfigBytePlus4 =  84
   Y1 =  dfde3c721be6e0b0369770788941a293
   Y2 =  96c4e50dd81725d3832221fa47d564e1
   DerivedKey (= Y1 || Y2) =
         dfde3c721be6e0b0369770788941a293
         96c4e50dd81725d3832221fa47d564e1
   Y3 =  675fb3ec6d0e56002333c2504d1b70db
   Y4 =  47c3713775999c9600bedcfda76f8d8c
   KeyCommit (= Y3 || Y4) =
         675fb3ec6d0e56002333c2504d1b70db
         47c3713775999c9600bedcfda76f8d8c
   ------------------------------------------------------------------
```

A4. A Worked-Out Example (LN = 12; No Key Commitment)

```
   ==================================================================
                            DNDK-GCM example
   ==================================================================
   Nonce length (LN) = 12
```

```
   KC_Choice =         0
   ========================================================================
                                      ---------Bytes Position---------
                                      000102030405060708090101112131415
                                   ...161718192021212324252627282930 31
                                      --------------------------------
                                      --------------------------------
   Root key =                         010000000000000000000000000000000
                                      000000000000000000000000000000000
   ------------------------------------------------------------------------
   Input
   ------------------------------------------------------------------------
   Nonce =                            000102030405060708090a0b

   aad (5 bytes) =                    0100000011
   plaintext (4 bytes) =              11000001
   ------------------------------------------------------------------------
   Output
   ------------------------------------------------------------------------
   cipher (4 bytes) =                 b95cf258
   tag =                              39e74511d997eaafd0f567d13758305b
   Key commitment value =

   ------------------------------------------------------------------------
   ciphertext + tag (20 bytes) =      b95cf25839e74511d997eaafd0f567d1
                                      3758305b
   ------------------------------------------------------------------------
   ciphertext + tag + commit
                       (20 bytes) =   b95cf25839e74511d997eaafd0f567d1
                                      3758305b
```

```
   ---------------------------------------------------------------
   Intermediate values
   ---------------------------------------------------------------
   DerivedKey =                        13c31bcaf1f11785e1dcb29d5d65541a
                                       4b371b1142bb60f39cea823f189e0a17
   AES-GCM nonce  =                    000000000000000000000000
   Nonce PAD length (bytes) =          15
   Padded nonce (NPadded) =            000102030405060708090a0b00000000
                                       0000000000000000000000
   ConfigByte
   128 * KC_Choice + 8 * (LN - 12)
   as int =                            0
   in hex =                            00
   ===============================================================
   B0 =  000102030405060708090a0b00000000
   X0 =  e794aa3c8558d394f822920ae2a9d6e5
   ConfigBytePlus0 =  00
   B1 =  000102030405060708090a0b00000001
   X1 =  f457b1f674a9c41119fe2097bfcc82ff
   ConfigBytePlus1 =  01
   B2 =  000102030405060708090a0b00000002
   X2 =  aca3b12dc7e3b36764c81035fa37dcf2
   ConfigBytePlus2 =  02
   Y1 =  13c31bcaf1f11785e1dcb29d5d65541a
   Y2 =  4b371b1142bb60f39cea823f189e0a17
   DerivedKey (= Y1 || Y2) =
         13c31bcaf1f11785e1dcb29d5d65541a
         4b371b1142bb60f39cea823f189e0a17
```

KeyCommit (= Y3 || Y4) =

Appendix B. Reproducing the Test Vectors (Python)

This appendix provides a small Python script that exactly reproduces the four test vectors in Appendix A. For brevity, the script does not print the intermediate values or outputs; those byte strings are shown in Appendix A (A1 to A4) and are the authoritative values for verification. The script computes the same quantities (including DerivedKey, KeyCommit when KC_Choice = 1, and GCM_IV) so that implementers can confirm the results by adding simple prints or assertions if desired. The normative specification is in Sections 3 to 5. The code uses AES-ECB and AES-GCM from PyCryptodome and is not (necessarily) side-channel hardened.

```
# DNDK-GCM encryption example
from Cryptodome.Cipher import AES
# Root key
RootKey = \
0x0100000000000000000000000000000000000000000000000000000000000000.\
to_bytes(32,'big')
# Nonce (24 bytes)
N =\
0x000102030405060708090a0b0c0d0e0f1011121314151617.\
to_bytes(24,'big')
'''
# The 12-byte case
###################
# Nonce (12 bytes)
```

```
N =\
0x000102030405060708090a0b.\
to_bytes(12,'big')
###################
'''

# aad and plaintext
aad = 0x0100000011.to_bytes(5,'big')
plaintext = 0x11000001.to_bytes(4,'big')
#
# Configuration
#
LN = len (N)
KC_Choice = 1 # 1/0 with/without key commitment
PADLEN = 27-LN
NPadded = N + (0x00).to_bytes (PADLEN, 'big')
NHead = NPadded [0: 15] # The first 15 bytes of the nonce
NTail = NPadded [15: 27]
###GCM_IV = NTail + (0x00).to_bytes (3, 'big')
GCM_IV = NTail
# Preparing blocks B0, B1, B2, B3, B4 for XORP
Configurenum = 128 * KC_Choice + 8*(LN-12)
ConfigBytePlus0 = (Configurenum + 0).to_bytes (1, 'big')
B0 = NHead + ConfigBytePlus0
ConfigBytePlus1 = (Configurenum + 1).to_bytes (1, 'big')
B1 = NHead + ConfigBytePlus1
ConfigBytePlus2 = (Configurenum + 2).to_bytes (1, 'big')
B2 = NHead + ConfigBytePlus2
if KC_Choice == 1:
    ConfigBytePlus3 = (Configurenum + 3).to_bytes (1, 'big')
```

```
        B3 = NHead + ConfigBytePlus3
        ConfigBytePlus4 = (Configurenum + 4).to_bytes (1, 'big')
        B4 = NHead + ConfigBytePlus4
    #
    # X values
    rootcipher = AES.new(key=RootKey, mode=AES.MODE_ECB)
    X0 = rootcipher.encrypt(B0)
    X1 = rootcipher.encrypt(B1)
    X2 = rootcipher.encrypt(B2)
    if KC_Choice == 1:
        X3 = rootcipher.encrypt(B3)
        X4 = rootcipher.encrypt(B4)
    # Y values
    Y1 = bytes(a ^ b for (a,b) in zip (X1, X0))
    Y2 = bytes(a ^ b for (a,b) in zip (X2, X0))
    DerivedKey = Y1 + Y2
    # Set KeyCommit to be empty
    # if KC_Choice = 0, KC, is empty
    KeyCommit = 0x00.to_bytes (0, 'big')
    # Otherwise, compute KeyCommit
    if KC_Choice == 1:
        Y3 = bytes(a ^ b for (a,b) in zip (X3, X0))
        Y4 = bytes(a ^ b for (a,b) in zip (X4, X0))
        KeyCommit = Y3 + Y4
    #
    # Encrypt (A, M) with AES-GCM under DerivedKey, key
    # and the 12-byte nonce GCM_IV
    aesgcmkey = DerivedKey
    cipher = AES.new(aesgcmkey, AES.MODE_GCM, nonce=GCM_IV)
    cipher.update(aad)
```

```
   ciphertext, tag = cipher.encrypt_and_digest(plaintext)
   # Combined cipher, tag, KeyCommit
   ciphertext_tag = ciphertext + tag
   ciphertext_tag_commit = ciphertext + tag + KeyCommit
   #
   # Printing the values can be done by e.g., the following lines:
   # print ("N = ", N.hex()[0: 32])
   # print (" ", N.hex()[32: 54])
   # print ("B0 = ", B0.hex())
   # print ("      ", KeyCommit.hex()[0: 32], sep='')
   # print ("      ", KeyCommit.hex()[32: 64], sep='')
```

Appendix C. Minimal Reference Implementation (Python)

This appendix provides a compact, self-contained Python sketch of DNDK-GCM encryption and decryption. It follows Section 4, performs basic input-range checks, and uses a constant-time byte-wise compare for the key-commitment check. It prints inputs and outputs only (no intermediate values). A small main demonstrates Appendix A1. This code is intended as an executable illustration and for harness tests. Production systems should use hardened libraries (for AES and AES-GCM).

Behavior summary (Enc/Dec checks):

Input ranges: RootKey = 32 bytes; N in {12,24}; kc_choice in {0,1}.

Decrypt: wrong KC length yields uniform 'FAIL' (kc_choice = 0 and KC != empty, or kc_choice = 1 and len(KC) != 32).

Decrypt: KC is compared to KC' in constant time; any mismatch causes an early 'FAIL' before invoking AES-GCM.

Decrypt: plaintext is returned only if the AES-GCM tag verifies; otherwise an exception is raised and no plaintext is released.

Encrypt: KC is returned only when kc_choice = 1; otherwise KC = empty.

10. Acknowledgments

The author would like to thank Jean Paul Degabriele, Gerald Doussot, Stan Drapkin, Isaac Elbaz, Sasha Frolov, Ed Knapp, Maximilian Lorlacks, Thomas Pornin, Eric Schorn and Falko Strenzke for their comments and suggestions.

11. Authors’ Addresses

Shay Gueron
University of Haifa
Abba Khoushy Ave 199
Haifa 3498838, Israel
Email: shay.gueron@gmail.com