

Internet Research Task Force (IRTF)
Request for Comments: 8452
Category: Informational
ISSN: 2070-1721

S. Gueron
University of Haifa and Amazon
A. Langley
Google LLC
Y. Lindell
Bar-Ilan University and Unbound Tech
April 2019

AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption

Abstract

This memo specifies two authenticated encryption algorithms that are nonce misuse resistant -- that is, they do not fail catastrophically if a nonce is repeated.

This document is the product of the Crypto Forum Research Group.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Research Task Force (IRTF). The IRTF publishes the results of Internet-related research and development activities. These results might not be suitable for deployment. This RFC represents the consensus of the Crypto Forum Research Group of the Internet Research Task Force (IRTF). Documents approved for publication by the IRSG are not candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <https://www.rfc-editor.org/info/rfc8452>.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction	2
2. Requirements Language	3
3. POLYVAL	3
4. Encryption	4
5. Decryption	7
6. AEADs	10
7. Field Operation Examples	10
8. Worked Example	10
9. Security Considerations	11
10. IANA Considerations	14
11. References	14
11.1. Normative References	14
11.2. Informative References	15
Appendix A. The Relationship between POLYVAL and GHASH	17
Appendix B. Additional Comparisons with AES-GCM	19
Appendix C. Test Vectors	20
C.1. AEAD_AES_128_GCM_SIV	20
C.2. AEAD_AES_256_GCM_SIV	30
C.3. Counter Wrap Tests	41
Acknowledgements	42
Authors' Addresses	42

1. Introduction

The concept of Authenticated Encryption with Additional Data (AEAD) [RFC5116] couples confidentiality and integrity in a single operation, avoiding the risks of the previously common practice of using ad hoc constructions of block-cipher and hash primitives. The most popular AEAD, AES-GCM [GCM], is seeing widespread use due to its attractive performance.

However, some AEADs (including AES-GCM) suffer catastrophic failures of confidentiality and/or integrity when two distinct messages are encrypted with the same key and nonce. While the requirements for AEADs specify that the pair of (key, nonce) shall only ever be used once, and thus prohibit this, this is a worry in practice.

Nonce misuse-resistant AEADs do not suffer from this problem. For this class of AEADs, encrypting two messages with the same nonce only discloses whether the messages were equal or not. This is the minimum amount of information that a deterministic algorithm can leak in this situation.

This memo specifies two nonce misuse-resistant AEADs: AEAD_AES_128_GCM_SIV and AEAD_AES_256_GCM_SIV. These AEADs are designed to be able to take advantage of existing hardware support

for AES-GCM and can decrypt within 5% of the speed of AES-GCM (for multikilobyte messages). Encryption is, perforce, slower than AES-GCM, because two passes are required in order to achieve that nonce misuse-resistance property. However, measurements suggest that it can still run at two-thirds of the speed of AES-GCM.

We suggest that these AEADs be considered in any situation where nonce uniqueness cannot be guaranteed. This includes situations where there is no stateful counter or where such state cannot be guaranteed, as when multiple encryptors use the same key. As discussed in Section 9, it is RECOMMENDED to use this scheme with randomly chosen nonces.

This document represents the consensus of the Crypto Forum Research Group (CFRG).

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. POLYVAL

The GCM-SIV construction is similar to GCM: the block cipher is used in counter mode to encrypt the plaintext, and a polynomial authenticator is used to provide integrity. The authenticator in GCM-SIV is called POLYVAL.

POLYVAL, like GHASH (the authenticator in AES-GCM; see [GCM], Section 6.4), operates in a binary field of size 2^{128} . The field is defined by the irreducible polynomial $x^{128} + x^{127} + x^{126} + x^{121} + 1$. The sum of any two elements in the field is the result of XORing them. The product of any two elements is calculated using standard (binary) polynomial multiplication followed by reduction modulo the irreducible polynomial.

We define another binary operation on elements of the field: $\text{dot}(a, b)$, where $\text{dot}(a, b) = a * b * x^{-128}$. The value of the field element x^{-128} is equal to $x^{127} + x^{124} + x^{121} + x^{114} + 1$. The result of this multiplication, $\text{dot}(a, b)$, is another field element.

Polynomials in this field are converted to and from 128-bit strings by taking the least significant bit of the first byte to be the coefficient of x^0 , the most significant bit of the first byte to be the coefficient of x^7 , and so on, until the most significant bit of the last byte is the coefficient of x^{127} .

POLYVAL takes a field element, H , and a series of field elements X_1, \dots, X_s . Its result is S_s , where S is defined by the iteration $S_0 = 0$; $S_j = \text{dot}(S_{j-1} + X_j, H)$, for $j = 1..s$.

We note that $\text{POLYVAL}(H, X_1, X_2, \dots)$ is equal to $\text{ByteReverse}(\text{GHASH}(\text{ByteReverse}(H) * x, \text{ByteReverse}(X_1), \text{ByteReverse}(X_2), \dots))$, where ByteReverse is a function that reverses the order of 16 bytes. See Appendix A for a more detailed explanation.

4. Encryption

AES-GCM-SIV encryption takes a 16- or 32-byte key-generating key, a 96-bit nonce, and plaintext and additional data byte strings of variable length. It outputs an authenticated ciphertext that will be 16 bytes longer than the plaintext. Both encryption and decryption are only defined on inputs that are a whole number of bytes.

If the key-generating key is 16 bytes long, then AES-128 is used throughout. Otherwise, AES-256 is used throughout.

The first step of encryption is to generate per-nonce, message-authentication and message-encryption keys. The message-authentication key is 128 bit, and the message-encryption key is either 128 (for AES-128) or 256 bit (for AES-256).

These keys are generated by encrypting a series of plaintext blocks that contain a 32-bit, little-endian counter followed by the nonce, and then discarding the second half of the resulting ciphertext. In the AES-128 case, $128 + 128 = 256$ bits of key material need to be generated, and, since encrypting each block yields 64 bits after discarding half, four blocks need to be encrypted. The counter values for these blocks are 0, 1, 2, and 3. For AES-256, six blocks are needed in total, with counter values 0 through 5 (inclusive).

In pseudocode form, where "++" indicates concatenation and "x[:8]" indicates taking only the first eight bytes from x:

```
func derive_keys(key_generating_key, nonce) {
    message_authentication_key =
        AES(key = key_generating_key,
            block = little_endian_uint32(0) ++ nonce)[:8] ++
        AES(key = key_generating_key,
            block = little_endian_uint32(1) ++ nonce)[:8]
    message_encryption_key =
        AES(key = key_generating_key,
            block = little_endian_uint32(2) ++ nonce)[:8] ++
        AES(key = key_generating_key,
            block = little_endian_uint32(3) ++ nonce)[:8]

    if bytelen(key_generating_key) == 32 {
        message_encryption_key +=
            AES(key = key_generating_key,
                block = little_endian_uint32(4) ++ nonce)[:8] ++
            AES(key = key_generating_key,
                block = little_endian_uint32(5) ++ nonce)[:8]
    }

    return message_authentication_key, message_encryption_key
}
```

Define the "length block" as a 16-byte value that is the concatenation of the 64-bit, little-endian encodings of $\text{bytelen}(\text{additional_data}) * 8$ and $\text{bytelen}(\text{plaintext}) * 8$. Pad the plaintext and additional data with zeros until they are each a multiple of 16 bytes, the AES block size. Then X_1, X_2, \dots (the series of field elements that are inputs to POLYVAL) are the concatenation of the padded additional data, the padded plaintext, and the length block.

Calculate $S_s = \text{POLYVAL}(\text{message-authentication-key}, X_1, X_2, \dots)$. XOR the first twelve bytes of S_s with the nonce and clear the most significant bit of the last byte. Encrypt the result with AES using the message-encryption key to produce the tag.

(It's worth highlighting a contrast with AES-GCM here: AES-GCM authenticates the encoded additional data and ciphertext, while AES-GCM-SIV authenticates the encoded additional data and plaintext.)

The encrypted plaintext is produced by using AES, with the message-encryption key, in counter mode (see [SP800-38A], Section 6.5) on the unpadded plaintext. The initial counter block is the tag with the most significant bit of the last byte set to one. The counter

advances by incrementing the first 32 bits interpreted as an unsigned, little-endian integer, wrapping at 2^{32} . The result of the encryption is the encrypted plaintext (truncated to the length of the plaintext), followed by the tag.

In pseudocode form, the encryption process can be expressed as:

```
func right_pad_to_multiple_of_16_bytes(input) {
    while (bytelen(input) % 16 != 0) {
        input = input ++ "\x00"
    }
    return input
}

func AES_CTR(key, initial_counter_block, in) {
    block = initial_counter_block

    output = ""
    while bytelen(in) > 0 {
        keystream_block = AES(key = key, block = block)
        block[0:4] = little_endian_uint32(
            read_little_endian_uint32(block[0:4]) + 1)

        todo = min(bytelen(in), bytelen(keystream_block))
        for j = 0; j < todo; j++ {
            output = output ++ (keystream_block[j] ^ in[j])
        }

        in = in[todo:]
    }

    return output
}

func encrypt(key_generating_key,
             nonce,
             plaintext,
             additional_data) {
    if bytelen(plaintext) > 2^36 {
        fail()
    }
    if bytelen(additional_data) > 2^36 {
        fail()
    }

    message_encryption_key, message_authentication_key =
        derive_keys(key_generating_key, nonce)
```

```

length_block =
    little_endian_uint64(bytelen(additional_data) * 8) ++
    little_endian_uint64(bytelen(plaintext) * 8)
padded_plaintext = right_pad_to_multiple_of_16_bytes(plaintext)
padded_ad = right_pad_to_multiple_of_16_bytes(additional_data)
S_s = POLYVAL(key = message_authentication_key,
              input = padded_ad ++ padded_plaintext ++
                    length_block)
for i = 0; i < 12; i++ {
    S_s[i] ^= nonce[i]
}
S_s[15] &= 0x7f
tag = AES(key = message_encryption_key, block = S_s)

counter_block = tag
counter_block[15] |= 0x80
return AES_CTR(key = message_encryption_key,
               initial_counter_block = counter_block,
               in = plaintext) ++
    tag
}

```

5. Decryption

Decryption takes a 16- or 32-byte key-generating key, a 96-bit nonce, and ciphertext and additional data byte strings of variable length. It either fails or outputs a plaintext that is 16 bytes shorter than the ciphertext.

To decrypt an AES-GCM-SIV ciphertext, first derive the message-encryption and message-authentication keys in the same manner as when encrypting.

If the ciphertext is less than 16 bytes or more than $2^{36} + 16$ bytes, then fail. Otherwise, split the input into the encrypted plaintext and a 16-byte tag. Decrypt the encrypted plaintext with the message-encryption key in counter mode, where the initial counter block is the tag with the most significant bit of the last byte set to one. Advance the counter for each block in the same way as when encrypting. At this point, the plaintext is unauthenticated and **MUST NOT** be output until the following tag confirmation is complete:

Pad the additional data and plaintext with zeros until they are each a multiple of 16 bytes, the AES block size. Calculate the length block and X_1 , X_2 , ... as above and compute $S_s = \text{POLYVAL}(\text{message-authentication-key}, X_1, X_2, \dots)$

Compute the expected tag by XORing S_s and the nonce, clearing the most significant bit of the last byte and encrypting with the message-encryption key. Compare the provided and expected tag values in constant time. Fail the decryption if they do not match (and do not release the plaintext); otherwise, return the plaintext.

In pseudocode form, the decryption process can be expressed as:


```

func decrypt(key_generating_key,
             nonce,
             ciphertext,
             additional_data) {
    if bytelen(ciphertext) < 16 || bytelen(ciphertext) > 2^36 + 16 {
        fail()
    }
    if bytelen(additional_data) > 2^36 {
        fail()
    }

    message_encryption_key, message_authentication_key =
        derive_keys(key_generating_key, nonce)

    tag = ciphertext[bytelen(ciphertext)-16:]

    counter_block = tag
    counter_block[15] |= 0x80
    plaintext = AES_CTR(key = message_encryption_key,
                       initial_counter_block = counter_block,
                       in = ciphertext[:bytelen(ciphertext)-16])

    length_block =
        little_endian_uint64(bytelen(additional_data) * 8) ++
        little_endian_uint64(bytelen(plaintext) * 8)
    padded_plaintext = right_pad_to_multiple_of_16_bytes(plaintext)
    padded_ad = right_pad_to_multiple_of_16_bytes(additional_data)
    S_s = POLYVAL(key = message_authentication_key,
                  input = padded_ad ++ padded_plaintext ++
                        length_block)

    for i = 0; i < 12; i++ {
        S_s[i] ^= nonce[i]
    }
    S_s[15] &= 0x7f
    expected_tag = AES(key = message_encryption_key, block = S_s)

    xor_sum = 0
    for i := 0; i < bytelen(expected_tag); i++ {
        xor_sum |= expected_tag[i] ^ tag[i]
    }

    if xor_sum != 0 {
        fail()
    }

    return plaintext
}

```

6. AEADs

We define two AEADs, in the format of RFC 5116, that use AES-GCM-SIV: `AEAD_AES_128_GCM_SIV` and `AEAD_AES_256_GCM_SIV`. They differ only in the size of the AES key used.

The key input to these AEADs becomes the key-generating key. Thus, `AEAD_AES_128_GCM_SIV` takes a 16-byte key and `AEAD_AES_256_GCM_SIV` takes a 32-byte key.

The parameters for `AEAD_AES_128_GCM_SIV` are then as follows: `K_LEN` is 16, `P_MAX` is 2^{36} , `A_MAX` is 2^{36} , `N_MIN` and `N_MAX` are 12, and `C_MAX` is $2^{36} + 16$.

The parameters for `AEAD_AES_256_GCM_SIV` differ only in the key size: `K_LEN` is 32, `P_MAX` is 2^{36} , `A_MAX` is 2^{36} , `N_MIN` and `N_MAX` are 12, and `C_MAX` is $2^{36} + 16$.

7. Field Operation Examples

Polynomials in this document will be written as 16-byte values. For example, the sixteen bytes `010000000000000000000000000000492` would represent the polynomial $x^{127} + x^{124} + x^{121} + x^{114} + 1$, which is also the value of x^{-128} in this field.

If `a = 66e94bd4ef8a2c3b884cfa59ca342b2e` and
`b = ff000000000000000000000000000000`,
 then `a + b = 99e94bd4ef8a2c3b884cfa59ca342b2e`,
`a * b = 37856175e9dc9df26ebc6d6171aa0ae9`, and
`dot(a, b) = ebe563401e7e91ea3ad6426b8140c394`.

8. Worked Example

Consider the encryption of the plaintext "Hello world" with the additional data "example" under key `ee8e1ed9ff2540ae8f2ba9f50bc2f27c` using `AEAD_AES_128_GCM_SIV`. The random nonce that we'll use for this example is `752abad3e0afb5f434dc4310`.

In order to generate the message-authentication and message-encryption keys, a counter is combined with the nonce to form four blocks. These blocks are encrypted with the key given above:

Counter	Nonce	Ciphertext
00000000	752abad3e0afb5f434dc4310	-> 310728d9911f1f38c40e952ca83d093e
01000000	752abad3e0afb5f434dc4310	-> 37b24316c3fab9a046ae90952daa0450
02000000	752abad3e0afb5f434dc4310	-> a4c5ae624996327947920b2d2412474b
03000000	752abad3e0afb5f434dc4310	-> c100be4d7e2c6edd1efef004305ab1e7

The latter halves of the ciphertext blocks are discarded and the remaining bytes are concatenated to form the per-message keys. Thus, the message-authentication key is 310728d9911f1f3837b24316c3fab9a0, and the message-encryption key is a4c5ae6249963279c100be4d7e2c6edd.

The length block contains the encoding of the bit lengths of the additional data and plaintext, respectively. The string "example" is seven characters, thus 56 bits (or 0x38 in hex). The string "Hello world" is 11 characters, or 88 = 0x58 bits. Thus, the length block is 380000000000000000005800000000000000.

The input to POLYVAL is the padded additional data, padded plaintext, and then the length block. This is 6578616d706c650000000000000000000048656c6c6f20776f726c64000000000038000000000000005800000000000000, based on the ASCII encoding of "example" (6578616d706c65) and "Hello world" (48656c6c6f20776f726c64).

Calling POLYVAL with the message-authentication key and the input above results in S_s = ad7fcf0b5169851662672f3c5f95138f.

Before encrypting, the nonce is XORed in and the most significant bit of the last byte is cleared. This gives d85575d8b1c630e256bb6c2c5f95130f, because that bit happened to be one previously. Encrypting with the message-encryption key (using AES-128) gives the tag, which is 4fbcdeb7e4793f4a1d7e4faa70100af1.

In order to form the initial counter block, the most significant bit of the last byte of the tag is set to one. That doesn't result in a change in this example. Encrypting this with the message key (using AES-128) gives the first block of the keystream: 1551f2c1787e81deac9a99f139540ab5.

The final ciphertext is the result of XORing the plaintext with the keystream and appending the tag. That gives 5d349ead175ef6b1def6fd4fbcdeb7e4793f4a1d7e4faa70100af1.

9. Security Considerations

AES-GCM-SIV decryption involves first producing an unauthenticated plaintext. This plaintext is vulnerable to manipulation by an attacker; thus, if an implementation released some or all of the plaintext before authenticating it, other parts of a system may process malicious data as if it were authentic. AES-GCM might be less likely to lead implementations to do this because there the ciphertext is generally authenticated before, or concurrently with, the plaintext calculation. Therefore, this text requires that implementations MUST NOT release unauthenticated plaintext. Thus, system designers should consider memory limitations when picking the

size of AES-GCM-SIV plaintexts: large plaintexts may not fit in the available memory of some machines, tempting implementations to release unverified plaintext.

A detailed cryptographic analysis of AES-GCM-SIV appears in [AES-GCM-SIV], and the remainder of this section is a summary of that paper.

The AEADs defined in this document calculate fresh AES keys for each nonce. This allows a larger number of plaintexts to be encrypted under a given key. Without this step, AES-GCM-SIV encryption would be limited by the birthday bound like other standard modes (e.g., AES-GCM, AES-CCM [RFC3610], and AES-SIV [RFC5297]). This means that when 2^{64} blocks have been encrypted overall, a distinguishing adversary who is trying to break the confidentiality of the scheme has an advantage of $1/2$. Thus, in order to limit the adversary's advantage to 2^{-32} , at most 2^{48} blocks can be encrypted overall. In contrast, by deriving fresh keys from each nonce, it is possible to encrypt a far larger number of messages and blocks with AES-GCM-SIV.

We stress that nonce misuse-resistant schemes guarantee that if a nonce repeats, then the only security loss is that identical plaintexts will produce identical ciphertexts. Since this can also be a concern (as the fact that the same plaintext has been encrypted twice is revealed), we do not recommend using a fixed nonce as a policy. In addition, as we show below, better-than-birthday bounds are achieved by AES-GCM-SIV when the nonce repetition rate is low. Finally, as shown in [BHT18], there is a great security benefit in the multiuser/multikey setting when each particular nonce is reused by a small number of users only. We stress that the nonce misuse-resistance property is not intended to be coupled with intentional nonce reuse; rather, such schemes provide the best possible security in the event of nonce reuse. Due to all of the above, it is **RECOMMENDED** that AES-GCM-SIV nonces be randomly generated.

Some example usage bounds for AES-GCM-SIV are given below. The adversary's advantage is the "AdvEnc" from [key-derive] and is colloquially the ability of an attacker to distinguish ciphertexts from random bit strings. The bounds below limit this advantage to 2^{-32} . For up to 256 uses of the same nonce and key (i.e., where one can assume that nonce misuse is no more than this bound), the following message limits should be respected (this assumes a short additional authenticated data (AAD), i.e., less than 64 bytes):

2^{29} messages, where each plaintext is at most 1 GiB

2^{35} messages, where each plaintext is at most 128 MiB

2^{49} messages, where each plaintext is at most 1 MiB

2^{61} messages, where each plaintext is at most 16 KiB

Suzuki et al. [multi-birthday] show that even if nonces are selected uniformly at random, the probability that one or more values would be repeated 256 or more times is negligible until the number of nonces reaches 2^{102} . (Specifically, the probability is $1/((2^{96})^{(255)}) * \text{Binomial}(q, 256)$, where q is the number of nonces.) Since 2^{102} is vastly greater than the limit on the number of plaintexts per key given above, we don't feel that this limit on the number of repeated nonces will be a problem. This also means that selecting nonces at random is a safe practice with AES-GCM-SIV. The bounds obtained for random nonces are as follows (as above, for these bounds, the adversary's advantage is at most 2^{-32}):

2^{32} messages, where each plaintext is at most 8 GiB

2^{48} messages, where each plaintext is at most 32 MiB

2^{64} messages, where each plaintext is at most 128 KiB

For situations where, for some reason, an even higher number of nonce repeats is possible (e.g., in devices with very poor randomness), the message limits need to be reconsidered. Theorem 7 in [AES-GCM-SIV] contains more details, but for up to 1,024 repeats of each nonce, the limits would be (again assuming a short AAD, i.e., less than 64 bytes):

2^{25} messages, where each plaintext is at most 1 GiB

2^{31} messages, where each plaintext is at most 128 MiB

2^{45} messages, where each plaintext is at most 1 MiB

2^{57} messages, where each plaintext is at most 16 KiB

In addition to calculating fresh AES keys for each nonce, these AEADs also calculate fresh POLYVAL keys. Previous versions of GCM-SIV did not do this and instead used part of the AEAD's key as the POLYVAL key. Bleichenbacher pointed out [Bleichenbacher16] that this allowed an attacker who controlled the AEAD key to force the POLYVAL key to be zero. If a user of this AEAD authenticated messages with a secret additional-data value, then this would be insecure as the attacker could calculate a valid authenticator without knowing the input. This does not violate the standard properties of an AEAD as the

additional data is not assumed to be confidential. However, we want these AEADs to be robust against plausible misuse and also to be drop-in replacements for AES-GCM and so derive nonce-specific POLYVAL keys to avoid this issue.

We also wish to note that the probability of successful forgery increases with the number of attempts that an attacker is permitted. The advantage defined in [key-derive] and used above is specified in terms of the ability of an attacker to distinguish ciphertexts from random bit strings. It thus covers both confidentiality and integrity, and Theorem 6.2 in [key-derive] shows that the advantage increases with the number of decryption attempts, although much more slowly than with the number of encryptions; the dependence on the number of decryption queries for forgery is actually only linear, not quadratic. The latter is an artifact of the bound in the paper not being tight. If an attacker is permitted extremely large numbers of attempts, then the tiny probability that any given attempt succeeds may sum to a non-trivial chance.

A security analysis of a similar scheme without nonce-based key derivation appears in [GCM-SIV], and a full analysis of the bounds when applying nonce-based key derivation appears in [key-derive]. A larger table of bounds and other information appears at [aes-gcm-siv-homepage].

The multiuser/multikey security of AES-GCM-SIV was studied by [BHT18], which showed that security is almost the same as in the single-user setting, as long as nonces do not repeat many times across many users. This is the case when nonces are chosen randomly.

10. IANA Considerations

IANA has added two entries to the "AEAD Algorithms" registry: AEAD_AES_128_GCM_SIV (Numeric ID 30) and AEAD_AES_256_GCM_SIV (Numeric ID 31), both referencing this document as their specification.

11. References

11.1. Normative References

- [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>>.

[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/info/rfc8174>>.

[SP800-38A]

Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST SP 800-38A, DOI 10.6028/NIST.SP.800-38A, December 2001, <<https://csrc.nist.gov/publications/detail/sp/800-38a/final>>.

11.2. Informative References

[AES-GCM-SIV]

Gueron, S., Langley, A., and Y. Lindell, "AES-GCM-SIV: Specification and Analysis", July 2017, <<https://eprint.iacr.org/2017/168>>.

[aes-gcm-siv-homepage]

Gueron, S., Langley, A., and Y. Lindell, "Webpage for the AES-GCM-SIV Mode of Operation", <<https://cyber.biu.ac.il/aes-gcm-siv/>>.

[BHT18]

Bose, P., Hoang, V., and S. Tessaro, "Revisiting AES-GCM-SIV: Multi-user Security, Faster Key Derivation, and Better Bounds", Advances in Cryptology - EUROCRYPT 2018, DOI 10.1007/978-3-319-78381-9_18, May 2018, <<https://eprint.iacr.org/2018/136.pdf>>.

[Bleichenbacher16]

Bleichenbacher, D., "Subject: AES-GCM-SIV security of the additional data", message to the cfrg mailing list, 24 June 2016, <<https://mailarchive.ietf.org/arch/msg/cfrg/qgh-Yxmj7CC7cq2YZLpmfGA3x-o>>.

[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>>.

[GCM-SIV]

Gueron, S. and Y. Lindell, "GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One Cycle Per Byte", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/2810103.2813613, October 2015, <<http://doi.acm.org/10.1145/2810103.2813613>>.

[key-derive]

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, DOI 10.1145/3133956.3133992, 2017, <<https://doi.org/10.1145/3133956.3133992>>.

[multi-birthday]

Suzuki, K., Tonien, D., Kurosawa, K., and K. Toyota, "Birthday Paradox for Multi-collisions", Information Security and Cryptology - ICISC 2006, Lecture Notes in Computer Science, Volume 4296, DOI 10.1007/11927587_5, 2006, <http://dx.doi.org/10.1007/11927587_5>.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September 2003, <<https://www.rfc-editor.org/info/rfc3610>>.

[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>>.

[RFC5297] Harkins, D., "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)", RFC 5297, DOI 10.17487/RFC5297, October 2008, <<https://www.rfc-editor.org/info/rfc5297>>.

Appendix A. The Relationship between POLYVAL and GHASH

GHASH and POLYVAL both operate in $GF(2^{128})$, although with different irreducible polynomials: POLYVAL works modulo $x^{128} + x^{127} + x^{126} + x^{121} + 1$ and GHASH works modulo $x^{128} + x^7 + x^2 + x + 1$. Note that these irreducible polynomials are the "reverse" of each other.

GHASH also has a different mapping between 128-bit strings and field elements. Whereas POLYVAL takes the least significant to most significant bits of the first byte to be the coefficients of x^0 to x^7 , GHASH takes them to be the coefficients of x^7 to x^0 . This continues until, for the last byte, POLYVAL takes the least significant to most significant bits to be the coefficients of x^{120} to x^{127} , while GHASH takes them to be the coefficients of x^{127} to x^{120} .

The combination of these facts means that it's possible to "convert" values between the two by reversing the order of the bytes in a 16-byte string. The differing interpretations of bit order takes care of reversing the bits within each byte, and then reversing the bytes does the rest. This may have a practical benefit for implementations that wish to implement both GHASH and POLYVAL.

In order to be clear which field a given operation is performed in, let `mulX_GHASH` be a function that takes a 16-byte string, converts it to an element of GHASH's field using GHASH's convention, multiplies it by x , and converts it back to a string. Likewise, let `mulX_POLYVAL` be a function that converts a 16-byte string to an element of POLYVAL's field using POLYVAL's convention, multiplies it by x , and converts it back.

Given the 16-byte string `01000000000000000000000000000000`, `mulX_GHASH` of that string is `00800000000000000000000000000000` and `mulX_POLYVAL` of that string is `02000000000000000000000000000000`. As a more general example, given `9c98c04df9387ded828175a92ba652d8`, `mulX_GHASH` of that string is `4e4c6026fc9c3ef6c140bad495d3296c` and `mulX_POLYVAL` of it is `3931819bf271fada0503eb52574ca5f2`.

Lastly, let `ByteReverse` be the function that takes a 16-byte string and returns a copy where the order of the bytes has been reversed.

Now GHASH and POLYVAL can be defined in terms of one another:

```
POLYVAL(H, X_1, ..., X_n) =
ByteReverse(GHASH(mulX_GHASH(ByteReverse(H)), ByteReverse(X_1), ...,
ByteReverse(X_n)))
```

```
GHASH(H, X_1, ..., X_n) =
ByteReverse(POLYVAL(mulX_POLYVAL(ByteReverse(H)), ByteReverse(X_1),
..., ByteReverse(X_n)))
```

As a worked example:

```
let H = 25629347589242761d31f826ba4b757b,
    X_1 = 4f4f95668c83dfb6401762bb2d01a262, and
    X_2 = d1a24ddd2721d006bbe45f20d3c9f362.
POLYVAL(H, X_1, X_2) = f7a3b47b846119fae5b7866cf5e5b77e.
```

If we wished to calculate this given only an implementation of GHASH, then the key for GHASH would be

```
mulX_GHASH(ByteReverse(H)) = dcbaa5dd137c188ebb21492c23c9b112.
```

```
Then ByteReverse(GHASH(dcba..., ByteReverse(X_1), ByteReverse(X_2)))
= f7a3b47b846119fae5b7866cf5e5b77e, as required.
```

In the other direction, GHASH(H, X_1, X_2) = bd9b3997046731fb96251b91f9c99d7a. If we wished to calculate this given only an implementation of POLYVAL, then we would first calculate the key for POLYVAL:

```
mulX_POLYVAL(ByteReverse(H)) = f6ea96744df0633aec8424b18e26c54a.
```

```
Then ByteReverse(POLYVAL(f6ea..., ByteReverse(X_1), ByteReverse(X_2)))
= bd9b3997046731fb96251b91f9c99d7a.
```

Appendix B. Additional Comparisons with AES-GCM

Some functional properties that differ between AES-GCM and AES-GCM-SIV that are also worth noting:

AES-GCM allows plaintexts to be encrypted in a streaming fashion -- i.e., the beginning of the plaintext can be encrypted and transmitted before the entire message has been processed. AES-GCM-SIV requires two passes for encryption and so cannot do this.

AES-GCM allows a constant additional-data input to be precomputed in order to save per-message computation. AES-GCM-SIV varies the authenticator key based on the nonce and so does not permit this.

The performance for AES-GCM versus AES-GCM-SIV on small machines can be roughly characterized by the number of AES operations and the number of $GF(2^{128})$ multiplications needed to process a message.

Let $a = (\text{bytelen}(\text{additional-data}) + 15) / 16$ and
 $p = (\text{bytelen}(\text{plaintext}) + 15) / 16$.

Then AES-GCM requires $p + 1$ AES operations and $p + a + 1$ field multiplications.

Defined similarly, AES-GCM-SIV with AES-128 requires $p + 5$ AES operations and $p + a + 1$ field multiplications. With AES-256, that becomes $p + 7$ AES operations.

With large machines, the available parallelism becomes far more important, and such simple performance analysis is no longer representative. For such machines, we find that decryption of AES-GCM-SIV is only about 5% slower than AES-GCM, as long as the message is at least a couple of kilobytes. Encryption tends to run about two-thirds the speed because of the additional pass required.

Appendix C. Test Vectors

C.1. AEAD_AES_128_GCM_SIV

```

Plaintext (0 bytes) =
AAD (0 bytes) =
Key = 01000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 00000000000000000000000000000000
POLYVAL result = 00000000000000000000000000000000
POLYVAL result XOR nonce = 03000000000000000000000000000000
... and masked = 03000000000000000000000000000000
Tag = dc20e2d83f25705bb49e439eca56de25
Initial counter = dc20e2d83f25705bb49e439eca56dea5
Result (16 bytes) = dc20e2d83f25705bb49e439eca56de25

```

```

Plaintext (8 bytes) = 0100000000000000
AAD (0 bytes) =
Key = 01000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 01000000000000000000000000000000
0000000000000000004000000000000000
POLYVAL result = eb93b7740962c5e49d2a90a7dc5cec74
POLYVAL result XOR nonce = e893b7740962c5e49d2a90a7dc5cec74
... and masked = e893b7740962c5e49d2a90a7dc5cec74
Tag = 578782fff6013b815b287c22493a364c
Initial counter = 578782fff6013b815b287c22493a36cc
Result (24 bytes) = b5d839330ac7b786578782fff6013b81
5b287c22493a364c

```

```

Plaintext (12 bytes) = 01000000000000000000000000000000
AAD (0 bytes) =
Key = 01000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 01000000000000000000000000000000
0000000000000000006000000000000000
POLYVAL result = 48eb6c6c5a2dbe4a1dde508fee06361b
POLYVAL result XOR nonce = 4beb6c6c5a2dbe4a1dde508fee06361b
... and masked = 4beb6c6c5a2dbe4a1dde508fee06361b
Tag = a4978db357391a0bc4fdec8b0d106639

```

Initial counter = a4978db357391a0bc4fdec8b0d1066b9
 Result (28 bytes) = 7323ea61d05932260047d942a4978db3
 57391a0bc4fdec8b0d106639

Plaintext (16 bytes) = 01000000000000000000000000000000
 AAD (0 bytes) =
 Key = 01000000000000000000000000000000
 Nonce = 03000000000000000000000000000000
 Record authentication key = d9b360279694941ac5dbc6987ada7377
 Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
 POLYVAL input = 01000000000000000000000000000000
 00000000000000000000000000000000
 POLYVAL result = 20806c26e3c1de019e111255708031d6
 POLYVAL result XOR nonce = 23806c26e3c1de019e111255708031d6
 ... and masked = 23806c26e3c1de019e11125570803156
 Tag = 303aaf90f6fe21199c6068577437a0c4
 Initial counter = 303aaf90f6fe21199c6068577437a0c4
 Result (32 bytes) = 743f7c8077ab25f8624e2e948579cf77
 303aaf90f6fe21199c6068577437a0c4

Plaintext (32 bytes) = 01000000000000000000000000000000
 02000000000000000000000000000000
 AAD (0 bytes) =
 Key = 01000000000000000000000000000000
 Nonce = 03000000000000000000000000000000
 Record authentication key = d9b360279694941ac5dbc6987ada7377
 Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
 POLYVAL input = 01000000000000000000000000000000
 02000000000000000000000000000000
 00000000000000000000000000000000
 POLYVAL result = ce6edc9a50b36d9a98986bbf6a261c3b
 POLYVAL result XOR nonce = cd6edc9a50b36d9a98986bbf6a261c3b
 ... and masked = cd6edc9a50b36d9a98986bbf6a261c3b
 Tag = 1a8e45dcd4578c667cd86847bf6155ff
 Initial counter = 1a8e45dcd4578c667cd86847bf6155ff
 Result (48 bytes) = 84e07e62ba83a6585417245d7ec413a9
 fe427d6315c09b57ce45f2e3936a9445
 1a8e45dcd4578c667cd86847bf6155ff

Plaintext (48 bytes) = 01000000000000000000000000000000
 02000000000000000000000000000000
 03000000000000000000000000000000
 AAD (0 bytes) =
 Key = 01000000000000000000000000000000
 Nonce = 03000000000000000000000000000000

```

Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 0100000000000000000000000000000000000000000000000000
0200000000000000000000000000000000000000000000000000
0300000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
POLYVAL result = 81388746bc22d26b2abc3dcb15754222
POLYVAL result XOR nonce = 82388746bc22d26b2abc3dcb15754222
... and masked = 82388746bc22d26b2abc3dcb15754222
Tag = 5e6e311dbf395d35b0fe39c2714388f8
Initial counter = 5e6e311dbf395d35b0fe39c2714388f8
Result (64 bytes) = 3fd24ce1f5a67b75bf2351f181a475c7
b800a5b4d3dcf70106b1eea82fa1d64d
f42bf7226122fa92e17a40eeaac1201b
5e6e311dbf395d35b0fe39c2714388f8

```

```

Plaintext (64 bytes) = 0100000000000000000000000000000000000000000000000000
0200000000000000000000000000000000000000000000000000
0300000000000000000000000000000000000000000000000000
0400000000000000000000000000000000000000000000000000

```

```

AAD (0 bytes) =
Key = 0100000000000000000000000000000000000000000000000000
Nonce = 0300000000000000000000000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 0100000000000000000000000000000000000000000000000000
0200000000000000000000000000000000000000000000000000
0300000000000000000000000000000000000000000000000000
0400000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
POLYVAL result = 1e39b6d3344d348f6044f89935d1cf78
POLYVAL result XOR nonce = 1d39b6d3344d348f6044f89935d1cf78
... and masked = 1d39b6d3344d348f6044f89935d1cf78
Tag = 8a263dd317aa88d56bdf3936dba75bb8
Initial counter = 8a263dd317aa88d56bdf3936dba75bb8
Result (80 bytes) = 2433668f1058190f6d43e360f4f35cd8
e475127cfca7028ea8ab5c20f7ab2af0
2516a2bdcbc08d521be37ff28c152bba
36697f25b4cd169c6590d1dd39566d3f
8a263dd317aa88d56bdf3936dba75bb8

```

```

Plaintext (8 bytes) = 0200000000000000000000000000000000000000000000000000
AAD (1 bytes) = 01
Key = 0100000000000000000000000000000000000000000000000000
Nonce = 0300000000000000000000000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377

```

Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 01000000000000000000000000000000
02000000000000000000000000000000
0800000000000000004000000000000000
POLYVAL result = b26781e7e2c1376f96bec195f3709b2a
POLYVAL result XOR nonce = b16781e7e2c1376f96bec195f3709b2a
... and masked = b16781e7e2c1376f96bec195f3709b2a
Tag = 3b0a1a2560969cdf790d99759abd1508
Initial counter = 3b0a1a2560969cdf790d99759abd1588
Result (24 bytes) = 1e6daba35669f4273b0a1a2560969cdf
790d99759abd1508

Plaintext (12 bytes) = 02000000000000000000000000000000
AAD (1 bytes) = 01
Key = 01000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 01000000000000000000000000000000
02000000000000000000000000000000
0800000000000000006000000000000000
POLYVAL result = 111f5affb18e4cc1164a01bdc12a4145
POLYVAL result XOR nonce = 121f5affb18e4cc1164a01bdc12a4145
... and masked = 121f5affb18e4cc1164a01bdc12a4145
Tag = 08299c5102745aaa3a0c469fad9e075a
Initial counter = 08299c5102745aaa3a0c469fad9e07da
Result (28 bytes) = 296c7889fd99f41917f4462008299c51
02745aaa3a0c469fad9e075a

Plaintext (16 bytes) = 02000000000000000000000000000000
AAD (1 bytes) = 01
Key = 01000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 01000000000000000000000000000000
02000000000000000000000000000000
0800000000000000008000000000000000
POLYVAL result = 79745ab508622c8a958543675fac4688
POLYVAL result XOR nonce = 7a745ab508622c8a958543675fac4688
... and masked = 7a745ab508622c8a958543675fac4608
Tag = 8f8936ec039e4e4bb97ebd8c4457441f
Initial counter = 8f8936ec039e4e4bb97ebd8c4457449f
Result (32 bytes) = e2b0c5da79a901c1745f700525cb335b
8f8936ec039e4e4bb97ebd8c4457441f

```

Plaintext (32 bytes) = 0200000000000000000000000000000000
0300000000000000000000000000000000
AAD (1 bytes) = 01
Key = 0100000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 0100000000000000000000000000000000
0200000000000000000000000000000000
0300000000000000000000000000000000
0800000000000000000000000000000000
POLYVAL result = 2ce7daaf7c89490822051255b12eca6b
POLYVAL result XOR nonce = 2fe7daaf7c89490822051255b12eca6b
... and masked = 2fe7daaf7c89490822051255b12eca6b
Tag = e6af6a7f87287da059a71684ed3498e1
Initial counter = e6af6a7f87287da059a71684ed3498e1
Result (48 bytes) = 620048ef3c1e73e57e02bb8562c416a3
19e73e4caac8e96a1ecb2933145a1d71
e6af6a7f87287da059a71684ed3498e1

```

```

Plaintext (48 bytes) = 0200000000000000000000000000000000
0300000000000000000000000000000000
0400000000000000000000000000000000
AAD (1 bytes) = 01
Key = 0100000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 0100000000000000000000000000000000
0200000000000000000000000000000000
0300000000000000000000000000000000
0400000000000000000000000000000000
0800000000000000000000000000000000
POLYVAL result = 9ca987715d69c1786711dfcd22f830fc
POLYVAL result XOR nonce = 9fa987715d69c1786711dfcd22f830fc
... and masked = 9fa987715d69c1786711dfcd22f8307c
Tag = 6a8cc3865f76897c2e4b245cf31c51f2
Initial counter = 6a8cc3865f76897c2e4b245cf31c51f2
Result (64 bytes) = 50c8303ea93925d64090d07bd109dfd9
515a5a33431019c17d93465999a8b005
3201d723120a8562b838cdff25bf9d1e
6a8cc3865f76897c2e4b245cf31c51f2

```

```

Plaintext (64 bytes) = 0200000000000000000000000000000000
0300000000000000000000000000000000
0400000000000000000000000000000000

```



```

Plaintext (20 bytes) = 0300000000000000000000000000000000
AAD (18 bytes) = 04000000
Key = 0100000000000000000000000000000000
Nonce = 0200
0300000000000000000000000000000000

```

```

Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 0100000000000000000000000000000000000000000000000000
0200000000000000000000000000000000000000000000000000
0300000000000000000000000000000000000000000000000000
0400000000000000000000000000000000000000000000000000
9000000000000000000000000000000000000000000000000000
POLYVAL result = 4781d492cb8f926c504caa36f61008fe
POLYVAL result XOR nonce = 4481d492cb8f926c504caa36f61008fe
... and masked = 4481d492cb8f926c504caa36f610087e
Tag = 24afc9805e976f451e6d87f6fe106514
Initial counter = 24afc9805e976f451e6d87f6fe106594
Result (36 bytes) = 6bb0fecf5ded9b77f902c7d5da236a43
91dd029724afc9805e976f451e6d87f6
fe106514

```

```

Plaintext (18 bytes) = 0300000000000000000000000000000000000000000000000000
0400
AAD (20 bytes) = 0100000000000000000000000000000000000000000000000000
02000000
Key = 0100000000000000000000000000000000000000000000000000
Nonce = 0300000000000000000000000000000000000000000000000000
Record authentication key = d9b360279694941ac5dbc6987ada7377
Record encryption key = 4004a0dcd862f2a57360219d2d44ef6c
POLYVAL input = 0100000000000000000000000000000000000000000000000000
0200000000000000000000000000000000000000000000000000
0300000000000000000000000000000000000000000000000000
0400000000000000000000000000000000000000000000000000
a000000000000000000000000000000000000000000000000000
POLYVAL result = 75cbc23a1a10e348aeb8e384b5cc79fd
POLYVAL result XOR nonce = 76cbc23a1a10e348aeb8e384b5cc79fd
... and masked = 76cbc23a1a10e348aeb8e384b5cc797d
Tag = bff9b2ef00fb47920cc72a0c0f13b9fd
Initial counter = bff9b2ef00fb47920cc72a0c0f13b9fd
Result (34 bytes) = 44d0aaf6fb2f1f34add5e8064e83e12a
2adabff9b2ef00fb47920cc72a0c0f13
b9fd

```

```

Plaintext (0 bytes) =
AAD (0 bytes) =
Key = e66021d5eb8e4f4066d4adb9c33560e4
Nonce = f46e44bb3da0015c94f70887
Record authentication key = 036ee1fe2d7926af68898095e54e7b3c
Record encryption key = 5e46482396008223b5c1d25173d87539
POLYVAL input = 0000000000000000000000000000000000000000000000000000
POLYVAL result = 0000000000000000000000000000000000000000000000000000
POLYVAL result XOR nonce = f46e44bb3da0015c94f70887000000000

```

```

... and masked =      f46e44bb3da0015c94f7088700000000
Tag =                  a4194b79071b01a87d65f706e3949578
Initial counter =      a4194b79071b01a87d65f706e39495f8
Result (16 bytes) =    a4194b79071b01a87d65f706e3949578

```

```

Plaintext (3 bytes) =  7a806c
AAD (5 bytes) =        46bb91c3c5
Key =                  36864200e0eaf5284d884a0e77d31646
Nonce =                bae8e37fc83441b16034566b
Record authentication key = 3e28de1120b2981a0155795ca2812af6
Record encryption key =   6d4b78b31a4c9c03d8db0f42f7507fae
POLYVAL input =        46bb91c3c5000000000000000000000000
                          7a806c0000000000000000000000000000
                          2800000000000000001800000000000000
POLYVAL result =       43d9a745511dcfa21b96dd606f1d5720
POLYVAL result XOR nonce = f931443a99298e137ba28b0b6f1d5720
... and masked =       f931443a99298e137ba28b0b6f1d5720
Tag =                  711bd85bc1e4d3e0a462e074eea428a8
Initial counter =      711bd85bc1e4d3e0a462e074eea428a8
Result (19 bytes) =    af60eb711bd85bc1e4d3e0a462e074ee
                        a428a8

```

```

Plaintext (6 bytes) =  bdc66f146545
AAD (10 bytes) =       fc880c94a95198874296
Key =                  aedb64a6c590bc84d1a5e269e4b47801
Nonce =                afc0577e34699b9e671fdd4f
Record authentication key = 43b8de9cea62330d15cccfc84a33e8c8
Record encryption key =   8e54631607e431e095b54852868e3a27
POLYVAL input =        fc880c94a9519887429600000000000000
                          bdc66f14654500000000000000000000
                          5000000000000000003000000000000000
POLYVAL result =       26498e0d2b1ef004e808c458e8f2f515
POLYVAL result XOR nonce = 8989d9731f776b9a8f171917e8f2f515
... and masked =       8989d9731f776b9a8f171917e8f2f515
Tag =                  d6a9c45545cfc11f03ad743dba20f966
Initial counter =      d6a9c45545cfc11f03ad743dba20f9e6
Result (22 bytes) =    bb93a3e34d3cd6a9c45545cfc11f03ad
                        743dba20f966

```

```

Plaintext (9 bytes) =  1177441f195495860f
AAD (15 bytes) =       046787f3ea22c127aaf195d1894728
Key =                  d5cc1fd161320b6920ce07787f86743b
Nonce =                275d1ab32f6d1f0434d8848c
Record authentication key = 8a51df64d93eaf667c2c09bd454ce5c5
Record encryption key =   43ab276c2b4a473918ca73f2dd85109c

```

POLYVAL input = 046787f3ea22c127aaf195d189472800
 1177441f195495860f0000000000000000
 7800000000000000000000480000000000000000
 POLYVAL result = 63a3451c0b23345ad02bba59956517cf
 POLYVAL result XOR nonce = 44fe5faf244e2b5ee4f33ed5956517cf
 ... and masked = 44fe5faf244e2b5ee4f33ed59565174f
 Tag = 1d02fd0cd174c84fc5dae2f60f52fd2b
 Initial counter = 1d02fd0cd174c84fc5dae2f60f52fdab
 Result (25 bytes) = 4f37281f7ad12949d01d02fd0cd174c8
 4fc5dae2f60f52fd2b

Plaintext (12 bytes) = 9f572c614b4745914474e7c7
 AAD (20 bytes) = c9882e5386fd9f92ec489c8fde2be2cf
 97e74e93
 Key = b3fed1473c528b8426a582995929a149
 Nonce = 9e9ad8780c8d63d0ab4149c0
 Record authentication key = 22f50707a95dd416df069d670cb775e8
 Record encryption key = f674a5584ee21fe97b4cebc468ab61e4
 POLYVAL input = c9882e5386fd9f92ec489c8fde2be2cf
 97e74e93000000000000000000000000000000
 9f572c614b4745914474e7c70000000000
 a000000000000000000000600000000000000000
 POLYVAL result = 0cca0423fba9d77fe7e2e6963b08cdd0
 POLYVAL result XOR nonce = 9250dc5bf724b4af4ca3af563b08cdd0
 ... and masked = 9250dc5bf724b4af4ca3af563b08cd50
 Tag = c1dc2f871fb7561da1286e655e24b7b0
 Initial counter = c1dc2f871fb7561da1286e655e24b7b0
 Result (28 bytes) = f54673c5ddf710c745641c8bc1dc2f87
 1fb7561da1286e655e24b7b0

Plaintext (15 bytes) = 0d8c8451178082355c9e940fea2f58
 AAD (25 bytes) = 2950a70d5a1db2316fd568378da107b5
 2b0da55210cc1c1b0a
 Key = 2d4ed87da44102952ef94b02b805249b
 Nonce = ac80e6f61455bfac8308a2d4
 Record authentication key = 0b00a29a83e7e95b92e3a0783b29f140
 Record encryption key = a430c27f285aed913005975c42eed5f3
 POLYVAL input = 2950a70d5a1db2316fd568378da107b5
 2b0da55210cc1c1b0a00000000000000000000
 0d8c8451178082355c9e940fea2f5800
 c800000000000000000000780000000000000000
 POLYVAL result = 1086ef25247aa41009bbc40871d9b350
 POLYVAL result XOR nonce = bc0609d3302f1bbc8ab366dc71d9b350
 ... and masked = bc0609d3302f1bbc8ab366dc71d9b350
 Tag = 83b3449b9f39552de99dc214a1190b0b
 Initial counter = 83b3449b9f39552de99dc214a1190b8b

Result (31 bytes) = c9ff545e07b88a015f05b274540aa183
b3449b9f39552de99dc214a1190b0b

Plaintext (18 bytes) = 6b3db4da3d57aa94842b9803a96e07fb
6de7

AAD (30 bytes) = 1860f762ebfbd08284e421702de0de18
baa9c9596291b08466f37de21c7f

Key = bde3b2f204d1e9f8b06bc47f9745b3d1
Nonce = ae06556fb6aa7890bebc18fe

Record authentication key = 21c874a8bad3603d1c3e8784df5b3f9f
Record encryption key = d1c16d72651c3df504eae27129d818e8

POLYVAL input = 1860f762ebfbd08284e421702de0de18
baa9c9596291b08466f37de21c7f0000
6b3db4da3d57aa94842b9803a96e07fb
6de70000000000000000000000000000
f0000000000000000000000000000000

POLYVAL result = 55462a5afa0da8d646481e049ef9c764

POLYVAL result XOR nonce = fb407f354ca7d046f8f406fa9ef9c764

... and masked = fb407f354ca7d046f8f406fa9ef9c764

Tag = 3e377094f04709f64d7b985310a4db84

Initial counter = 3e377094f04709f64d7b985310a4db84

Result (34 bytes) = 6298b296e24e8cc35dce0bed484b7f30
d5803e377094f04709f64d7b985310a4
db84

Plaintext (21 bytes) = e42a3c02c25b64869e146d7b233987bd
dfc240871d

AAD (35 bytes) = 7576f7028ec6eb5ea7e298342a94d4b2
02b370ef9768ec6561c4fe6b7e7296fa
859c21

Key = f901cfe8a69615a93fdf7a98cad48179
Nonce = 6245709fb18853f68d833640

Record authentication key = 3724f55f1d22ac0ab830da0b6a995d74
Record encryption key = 75ac87b70c05db287de779006105a344

POLYVAL input = 7576f7028ec6eb5ea7e298342a94d4b2
02b370ef9768ec6561c4fe6b7e7296fa
859c2100000000000000000000000000

e42a3c02c25b64869e146d7b233987bd

dfc240871d0000000000000000000000

18010000000000000000a800000000000000

POLYVAL result = 4cbba090f03f7d1188ea55749fa6c7bd

POLYVAL result XOR nonce = 2efed00f41b72ee7056963349fa6c7bd

... and masked = 2efed00f41b72ee7056963349fa6c73d

Tag = 2d15506c84a9edd65e13e9d24a2a6e70

Initial counter = 2d15506c84a9edd65e13e9d24a2a6ef0

Result (37 bytes) = 391cc328d484a4f46406181bcd62efd9

C.2. AEAD_AES_256_GCM_SIV

```
Plaintext (0 bytes) =  
AAD (0 bytes) =  
Key = 01000000000000000000000000000000  
      00000000000000000000000000000000  
Nonce = 03000000000000000000000000000000  
Record authentication key = b5d3c529dfafac43136d2d11be284d7f  
Record encryption key = b914f4742be9e1d7a2f84adbbf96dec3  
      456e3c6c05ecc157cdbf0700fedad222  
POLYVAL input = 00000000000000000000000000000000  
POLYVAL result = 00000000000000000000000000000000  
POLYVAL result XOR nonce = 03000000000000000000000000000000  
... and masked = 03000000000000000000000000000000  
Tag = 07f5f4169bbf55a8400cd47ea6fd400f  
Initial counter = 07f5f4169bbf55a8400cd47ea6fd408f  
Result (16 bytes) = 07f5f4169bbf55a8400cd47ea6fd400f
```

```
Plaintext (8 bytes) =          010000000000000000
AAD (0 bytes) =
Key =                          0100000000000000000000000000000000
                                00000000000000000000000000000000
Nonces =                       03000000000000000000000000
Record authentication key =    b5d3c529dfafac43136d2d11be284d7f
Record encryption key =       b914f4742be9e1d7a2f84adbbf96dec3
                                456e3c6c05ecc157cdbf0700fedad222
POLYVAL input =               0100000000000000000000000000000000
                                0000000000000000004000000000000000
POLYVAL result =              05230f62f0eac8aa14fe4d646b59cd41
POLYVAL result XOR nonce =    06230f62f0eac8aa14fe4d646b59cd41
... and masked =              06230f62f0eac8aa14fe4d646b59cd41
Tag =                         843122130f7364b761e0b97427e3df28
Initial counter =             843122130f7364b761e0b97427e3dfa8
Result (24 bytes) =           c2ef328e5c71c83b843122130f7364b7
                                61e0b97427e3df28
```

```

Plaintext (12 bytes) = 01000000000000000000000000
AAD (0 bytes) =
Key = 0100000000000000000000000000000000
      0000000000000000000000000000000000
Nonce = 03000000000000000000000000
Record authentication key = b5d3c529dfafac43136d2d11be284d7f
Record encryption key = b914f4742be9e1d7a2f84adbbf96dec3

```

```

Plaintext (32 bytes) = 0100000000000000000000000000000000
0200000000000000000000000000000000

AAD (0 bytes) =
Key = 0100000000000000000000000000000000
0000000000000000000000000000000000
03000000000000000000000000000000

Nonce =
Record authentication key = b5d3c529dfafac43136d2d11be284d7f
Record encryption key = b914f4742be9e1d7a2f84adbbf96dec3
456e3c6c05ecc157cdbf0700fedad222

POLYVAL input = 0100000000000000000000000000000000
0200000000000000000000000000000000
0000000000000000000000000100000000000000

POLYVAL result = 899b6381b3d46f0def7aa0517ba188f5
POLYVAL result XOR nonce = 8a9b6381b3d46f0def7aa0517ba188f5
... and masked = 8a9b6381b3d46f0def7aa0517ba18875
Tag = e819e63abcd020b006a976397632eb5d
Initial counter = e819e63abcd020b006a976397632ebdd

```

Result (48 bytes) = 4a6a9db4c8c6549201b9edb53006cba8
21ec9cf850948a7c86c68ac7539d027f
e819e63abcd020b006a976397632eb5d

Plaintext (48 bytes) = 01000000000000000000000000000000
02000000000000000000000000000000
03000000000000000000000000000000

AAD (0 bytes) =

Key = 01000000000000000000000000000000
00000000000000000000000000000000

Nonce = 03000000000000000000000000000000

Record authentication key = b5d3c529dfafac43136d2d11be284d7f

Record encryption key = b914f4742be9e1d7a2f84addbf96dec3

456e3c6c05ecc157cdbf0700fedad222

POLYVAL input = 01000000000000000000000000000000
02000000000000000000000000000000
03000000000000000000000000000000

00000000000000000000000000000000

00000000000000000000000000000000

POLYVAL result = c1f8593d8fc29b0c290cae1992f71f51

POLYVAL result XOR nonce = c2f8593d8fc29b0c290cae1992f71f51

... and masked = c2f8593d8fc29b0c290cae1992f71f51

Tag = 790bc96880a99ba804bd12c0e6a22cc4

Initial counter = 790bc96880a99ba804bd12c0e6a22cc4

Result (64 bytes) = c00d121893a9fa603f48ccc1ca3c57ce

7499245ea0046db16c53c7c66fe717e3

9cf6c748837b61f6ee3adcee17534ed5

790bc96880a99ba804bd12c0e6a22cc4

Plaintext (64 bytes) = 01000000000000000000000000000000
02000000000000000000000000000000
03000000000000000000000000000000
04000000000000000000000000000000

AAD (0 bytes) =

Key = 01000000000000000000000000000000
00000000000000000000000000000000

Nonce = 03000000000000000000000000000000

Record authentication key = b5d3c529dfafac43136d2d11be284d7f

Record encryption key = b914f4742be9e1d7a2f84addbf96dec3

456e3c6c05ecc157cdbf0700fedad222

POLYVAL input = 01000000000000000000000000000000
02000000000000000000000000000000
03000000000000000000000000000000

04000000000000000000000000000000

00000000000000000000000000000000

00000000000000000000000000000000

POLYVAL result = 6ef38b06046c7c0e225efaef8e2ec4c4

POLYVAL result XOR nonce = 6df38b06046c7c0e225efaef8e2ec4c4


```

... and masked = 6df38b06046c7c0e225efaef8e2ec444
Tag = 112864c269fc0d9d88c61fa47e39aa08
Initial counter = 112864c269fc0d9d88c61fa47e39aa88
Result (80 bytes) = c2d5160a1f8683834910acdafc41fbb1
632d4a353e8b905ec9a5499ac34f96c7
e1049eb080883891a4db8caaa1f99dd0
04d80487540735234e3744512c6f90ce
112864c269fc0d9d88c61fa47e39aa08

```

```

Plaintext (8 bytes) = 020000000000000000
AAD (1 bytes) = 01
Key = 0100000000000000000000000000000000
0000000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = b5d3c529dfafac43136d2d11be284d7f
Record encryption key = b914f4742be9e1d7a2f84addbf96dec3
456e3c6c05ecc157cdbf0700fedad222
POLYVAL input = 0100000000000000000000000000000000
0200000000000000000000000000000000
0800000000000000000000000000000000
POLYVAL result = 34e57bafe011b9b36fc6821b7ffb3354
POLYVAL result XOR nonce = 37e57bafe011b9b36fc6821b7ffb3354
... and masked = 37e57bafe011b9b36fc6821b7ffb3354
Tag = 91213f267e3b452f02d01ae33e4ec854
Initial counter = 91213f267e3b452f02d01ae33e4ec8d4
Result (24 bytes) = 1de22967237a813291213f267e3b452f
02d01ae33e4ec854

```

```

Plaintext (12 bytes) = 02000000000000000000000000000000
AAD (1 bytes) = 01
Key = 0100000000000000000000000000000000
0000000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = b5d3c529dfafac43136d2d11be284d7f
Record encryption key = b914f4742be9e1d7a2f84addbf96dec3
456e3c6c05ecc157cdbf0700fedad222
POLYVAL input = 0100000000000000000000000000000000
0200000000000000000000000000000000
0800000000000000000000000000000000
POLYVAL result = 5c47d68a22061c1ad5623a3b66a8e206
POLYVAL result XOR nonce = 5f47d68a22061c1ad5623a3b66a8e206
... and masked = 5f47d68a22061c1ad5623a3b66a8e206
Tag = c1a4a19ae800941ccdc57cc8413c277f
Initial counter = c1a4a19ae800941ccdc57cc8413c27ff
Result (28 bytes) = 163d6f9cc1b346cd453a2e4cc1a4a19a
e800941ccdc57cc8413c277f

```

```

Plaintext (16 bytes) = 0200000000000000000000000000000000
AAD (1 bytes) = 01
Key = 0100000000000000000000000000000000
      0000000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = b5d3c529dfafac43136d2d11be284d7f
Record encryption key = b914f4742be9e1d7a2f84addbf96dec3
                        456e3c6c05ecc157cdbf0700fedad222
POLYVAL input = 0100000000000000000000000000000000
                  0200000000000000000000000000000000
                  0800000000000000000000000000000000
POLYVAL result = 452896726c616746f01d11d82911d478
POLYVAL result XOR nonce = 462896726c616746f01d11d82911d478
... and masked = 462896726c616746f01d11d82911d478
Tag = b292d28ff61189e8e49f3875ef91aff7
Initial counter = b292d28ff61189e8e49f3875ef91aff7
Result (32 bytes) = c91545823cc24f17dbb0e9e807d5ec17
                  b292d28ff61189e8e49f3875ef91aff7

```

```

Plaintext (32 bytes) = 0200000000000000000000000000000000
                        0300000000000000000000000000000000
AAD (1 bytes) = 01
Key = 0100000000000000000000000000000000
      0000000000000000000000000000000000
Nonce = 03000000000000000000000000000000
Record authentication key = b5d3c529dfafac43136d2d11be284d7f
Record encryption key = b914f4742be9e1d7a2f84addbf96dec3
                        456e3c6c05ecc157cdbf0700fedad222
POLYVAL input = 0100000000000000000000000000000000
                  0200000000000000000000000000000000
                  0300000000000000000000000000000000
                  0800000000000000000000000000000000
POLYVAL result = 4e58c1e341c9bb0ae34eda9509dfc90c
POLYVAL result XOR nonce = 4d58c1e341c9bb0ae34eda9509dfc90c
... and masked = 4d58c1e341c9bb0ae34eda9509dfc90c
Tag = aea1bad12702e1965604374aab96dbbc
Initial counter = aea1bad12702e1965604374aab96dbbc
Result (48 bytes) = 07dad364bfc2b9da89116d7bef6daaaf
                  6f255510aa654f920ac81b94e8bad365
                  aea1bad12702e1965604374aab96dbbc

```

```

Plaintext (48 bytes) = 0200000000000000000000000000000000
                        0300000000000000000000000000000000
                        0400000000000000000000000000000000
AAD (1 bytes) = 01
Key = 0100000000000000000000000000000000

```

```

Nonce = 00000000000000000000000000000000
Record authentication key = 03000000000000000000000000000000
Record encryption key = b5d3c529dfafac43136d2d11be284d7f
POLYVAL input = b914f4742be9e1d7a2f84adbbf96dec3
POLYVAL result = 456e3c6c05ecc157cdfbf0700fedad222
POLYVAL result XOR nonce = 01000000000000000000000000000000
... and masked = 02000000000000000000000000000000
Tag = 03000000000000000000000000000000
Initial counter = 04000000000000000000000000000000
Result (64 bytes) = 0800000000000000080010000000000000
                    2566a4aff9a525df9772c16d4eaf8d2a
                    2666a4aff9a525df9772c16d4eaf8d2a
                    2666a4aff9a525df9772c16d4eaf8d2a
                    03332742b228c647173616cfd44c54eb
                    03332742b228c647173616cfd44c54eb
                    c67a1f0f567a5198aa1fcc8e3f213143
                    36f7f51ca8b1af61feac35a86416fa47
                    fbca3b5f749cdf564527f2314f42fe25
                    03332742b228c647173616cfd44c54eb

```

Plaintext (64 bytes) =	02000000000000000000000000000000 03000000000000000000000000000000 04000000000000000000000000000000 05000000000000000000000000000000
AAD (1 bytes) =	01
Key =	01000000000000000000000000000000 00000000000000000000000000000000
Nonce =	03000000000000000000000000000000
Record authentication key =	b5d3c529dfafac43136d2d11be284d7f
Record encryption key =	b914f4742be9e1d7a2f84adbbf96dec3 456e3c6c05ecc157cdbf0700fedad222
POLYVAL input =	01000000000000000000000000000000 02000000000000000000000000000000 03000000000000000000000000000000 04000000000000000000000000000000 05000000000000000000000000000000 0800000000000000000002000000000000
POLYVAL result =	da58d2f61b0a9d343b2f37fb0c519733
POLYVAL result XOR nonce =	d958d2f61b0a9d343b2f37fb0c519733
... and masked =	d958d2f61b0a9d343b2f37fb0c519733
Tag =	5bde0285037c5de81e5b570a049b62a0
Initial counter =	5bde0285037c5de81e5b570a049b62a0
Result (80 bytes) =	67fd45e126bf b9a79930c43aad2d3696 7d3f0e4d217c1e551f59727870beef c9 8cb933a8fce9de887b1e40799988db1f c3f91880ed405b2dd298318858467c89 5bde0285037c5de81e5b570a049b62a0

```

Plaintext (4 bytes) = 02000000
AAD (12 bytes) = 01000000000000000000000000000000
Key = 0100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
Nonce = 0300000000000000000000000000000000
Record authentication key = b5d3c529dfafac43136d2d11be284d7f
Record encryption key = b914f4742be9e1d7a2f84addbf96dec3
456e3c6c05ecc157cdbf0700fedad222
POLYVAL input = 0100000000000000000000000000000000000000000000000000000000000000
0200000000000000000000000000000000000000000000000000000000000000
6000000000000000000000000000000000000000000000000000000000000000
POLYVAL result = 6dc76ae84b88916e073a303aafde05cf
POLYVAL result XOR nonce = 6ec76ae84b88916e073a303aafde05cf
... and masked = 6ec76ae84b88916e073a303aafde054f
Tag = 1835e517741dfddccfa07fa4661b74cf
Initial counter = 1835e517741dfddccfa07fa4661b74cf
Result (20 bytes) = 22b3f4cd1835e517741dfddccfa07fa4
661b74cf

```

```

Plaintext (20 bytes) = 0300000000000000000000000000000000000000000000000000000000000000
04000000
AAD (18 bytes) = 0100000000000000000000000000000000000000000000000000000000000000
0200
Key = 0100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
Nonce = 0300000000000000000000000000000000
Record authentication key = b5d3c529dfafac43136d2d11be284d7f
Record encryption key = b914f4742be9e1d7a2f84addbf96dec3
456e3c6c05ecc157cdbf0700fedad222
POLYVAL input = 0100000000000000000000000000000000000000000000000000000000000000
0200000000000000000000000000000000000000000000000000000000000000
0300000000000000000000000000000000000000000000000000000000000000
0400000000000000000000000000000000000000000000000000000000000000
9000000000000000000000000000000000000000000000000000000000000000
POLYVAL result = 973ef4fd04bd31d193816ab26f8655ca
POLYVAL result XOR nonce = 943ef4fd04bd31d193816ab26f8655ca
... and masked = 943ef4fd04bd31d193816ab26f86554a
Tag = b879ad976d8242acc188ab59cabfe307
Initial counter = b879ad976d8242acc188ab59cabfe387
Result (36 bytes) = 43dd0163cdb48f9fe3212bf61b201976
067f342bb879ad976d8242acc188ab59
cabfe307

```

```

Plaintext (18 bytes) = 0300000000000000000000000000000000000000000000000000000000000000
0400
AAD (20 bytes) = 0100000000000000000000000000000000000000000000000000000000000000

```

[illegible]

```

POLYVAL result = 2800000000000000001800000000000000
POLYVAL result XOR nonce = b91f91f96b159a7c611c05035b839e92
... and masked = 5dabe9f8c4d5cd0255759e9d5b839e92
Tag = 5dabe9f8c4d5cd0255759e9d5b839e12
Initial counter = 93da9bb81333aee0c785b240d319719d
Result (19 bytes) = 93da9bb81333aee0c785b240d319719d
0eaccb93da9bb81333aee0c785b240d319719d

```

```

Plaintext (6 bytes) = 195495860f04
AAD (10 bytes) = 6787f3ea22c127aaf195
Key = 6545fc880c94a95198874296d5cc1fd1
61320b6920ce07787f86743b275d1ab3
2f6d1f0434d8848c1177441f
Nonce = e156e1f9b0b07b780cbe30f259e3c8da
Record authentication key = 6fc1c494519f944aae52fcd8b14e5b17
Record encryption key = 1b5a9429d3b76e430d49940c0021d612
POLYVAL input = 6787f3ea22c127aaf19500000000000000
195495860f040000000000000000000000
500000000000000000030000000000000000
POLYVAL result = 2c480ed9d236b1df24c6eec109bd40c1
POLYVAL result XOR nonce = 032511dde6ee355335b1aade09bd40c1
... and masked = 032511dde6ee355335b1aade09bd4041
Tag = 6b62b84dc40c84636a5ec12020ec8c2c
Initial counter = 6b62b84dc40c84636a5ec12020ec8cac
Result (22 bytes) = a254dad4f3f96b62b84dc40c84636a5ec12020ec8c2c
c12020ec8c2c

```

```

Plaintext (9 bytes) = c9882e5386fd9f92ec
AAD (15 bytes) = 489c8fde2be2cf97e74e932d4ed87d
Key = d1894728b3fed1473c528b8426a58299
5929a1499e9ad8780c8d63d0ab4149c0
9f572c614b4745914474e7c7
Nonce = 0533fd71f4119257361a3ff1469dd4e5
Record authentication key = 4feba89799be8ac3684fa2bb30ade0ea
Record encryption key = 51390e6d87dcf3627d2ee44493853abe
POLYVAL input = 489c8fde2be2cf97e74e932d4ed87d00
c9882e5386fd9f92ec0000000000000000
780000000000000000048000000000000000
POLYVAL result = bf160bc9ded8c63057d2c38aae552fb4
POLYVAL result XOR nonce = 204127a8959f83a113a6244dae552fb4
... and masked = 204127a8959f83a113a6244dae552f34
Tag = c0fd3dc6628dfe55ebb0b9fb2295c8c2
Initial counter = c0fd3dc6628dfe55ebb0b9fb2295c8c2
Result (25 bytes) = 0df9e308678244c44bc0fd3dc6628dfe55ebb0b9fb2295c8c2
55ebb0b9fb2295c8c2

```

```

Plaintext (12 bytes) = 1db2316fd568378da107b52b
AAD (20 bytes) = 0da55210cc1c1b0abde3b2f204d1e9f8
                    b06bc47f
Key = a44102952ef94b02b805249bac80e6f6
      1455bfac8308a2d40d8c845117808235
Nonce = 5c9e940fea2f582950a70d5a
Record authentication key = 64779ab10ee8a280272f14cc8851b727
Record encryption key = 25f40fc63f49d3b9016a8eeeb75846e0
                        d72ca36ddb312b6f5ef38ad14bd2651
POLYVAL input = 0da55210cc1c1b0abde3b2f204d1e9f8
                b06bc47f00000000000000000000000000000000
                1db2316fd568378da107b52b0000000000
                a000000000000000000000000000000000
POLYVAL result = cc86ee22c861e1fd474c84676b42739c
POLYVAL result XOR nonce = 90187a2d224eb9d417eb893d6b42739c
... and masked = 90187a2d224eb9d417eb893d6b42731c
Tag = 404099c2587f64979f21826706d497d5
Initial counter = 404099c2587f64979f21826706d497d5
Result (28 bytes) = 8dbeb9f7255bf5769dd56692404099c2
                    587f64979f21826706d497d5

```

```

Plaintext (15 bytes) = 21702de0de18baa9c9596291b08466
AAD (25 bytes) = f37de21c7ff901cfe8a69615a93fdf7a
                  98cad481796245709f
Key = 9745b3d1ae06556fb6aa7890bebc18fe
      6b3db4da3d57aa94842b9803a96e07fb
      6de71860f762ebfbd08284e4
Nonce = 27c2959ed4daea3b1f52e849478de376
Record authentication key = 307a38a5a6cf231c0a9af3b527f23a62
Record encryption key = e9a6ff09aff8ae669f760153e864fc93
POLYVAL input = f37de21c7ff901cfe8a69615a93fdf7a
                98cad481796245709f0000000000000000
                21702de0de18baa9c9596291b0846600
                c800000000000000007800000000000000
POLYVAL result = c4fa5e5b713853703bcf8e6424505fa5
POLYVAL result XOR nonce = a91d463b865ab88beb4d0a8024505fa5
... and masked = a91d463b865ab88beb4d0a8024505f25
Tag = b3080d28f6ebb5d3648ce97bd5ba67fd
Initial counter = b3080d28f6ebb5d3648ce97bd5ba67fd
Result (31 bytes) = 793576dfa5c0f88729a7ed3c2f1bfff3
                    080d28f6ebb5d3648ce97bd5ba67fd

```

```

Plaintext (18 bytes) = b202b370ef9768ec6561c4fe6b7e7296
AAD (30 bytes) = fa85
                  9c2159058b1f0fe91433a5bdc20e214e
                  ab7fecef4454a10ef0657df21ac7

```

```

Key = b18853f68d833640e42a3c02c25b6486
      9e146d7b233987bddfc240871d7576f7
Nonce = 028ec6eb5ea7e298342a94d4
Record authentication key = 670b98154076ddb59b7a9137d0dcc0f0
Record encryption key = 78116d78507fbe69d4a820c350f55c7c
      b36c3c9287df0e9614b142b76a587c3f
POLYVAL input = 9c2159058b1f0fe91433a5bdc20e214e
      ab7fecef4454a10ef0657df21ac70000
      b202b370ef9768ec6561c4fe6b7e7296
      fa85000000000000000000000000000000
      f000000000000000009000000000000000
POLYVAL result = 4e4108f09f41d797dc9256f8da8d58c7
POLYVAL result XOR nonce = 4ccfce1bc1e6350fe8b8c22cda8d58c7
... and masked = 4ccfce1bc1e6350fe8b8c22cda8d5847
Tag = 454fc2a154fea91f8363a39fec7d0a49
Initial counter = 454fc2a154fea91f8363a39fec7d0ac9
Result (34 bytes) = 857e16a64915a787637687db4a951963
      5cdd454fc2a154fea91f8363a39fec7d
      0a49

Plaintext (21 bytes) = ced532ce4159b035277d4dfbb7db6296
      8b13cd4eec
AAD (35 bytes) = 734320ccc9d9bbbb19cb81b2af4ecbc3
      e72834321f7aa0f70b7282b4f33df23f
      167541
Key = 3c535de192eaed3822a2fbbe2ca9dfc8
      8255e14a661b8aa82cc54236093bbc23
Nonce = 688089e55540db1872504e1c
Record authentication key = cb8c3aa3f8dbaeb4b28a3e86ff6625f8
Record encryption key = 02426ce1aa3ab31313b0848469a1b5fc
      6c9af9602600b195b04ad407026bc06d
POLYVAL input = 734320ccc9d9bbbb19cb81b2af4ecbc3
      e72834321f7aa0f70b7282b4f33df23f
      16754100000000000000000000000000
      ced532ce4159b035277d4dfbb7db6296
      8b13cd4eec0000000000000000000000
      180100000000000000a800000000000000
POLYVAL result = ffd503c7dd712eb3791b7114b17bb0cf
POLYVAL result XOR nonce = 97558a228831f5ab0b4b3f08b17bb0cf
... and masked = 97558a228831f5ab0b4b3f08b17bb04f
Tag = 9d6c7029675b89eaf4ba1ded1a286594
Initial counter = 9d6c7029675b89eaf4ba1ded1a286594
Result (37 bytes) = 626660c26ea6612fb17ad91e8e767639
      edd6c9faee9d6c7029675b89eaf4ba1d
      ed1a286594

```


C.3. Counter Wrap Tests

The tests in this section use AEAD_AES_256_GCM_SIV and are crafted to test correct wrapping of the block counter.

```

Plaintext (32 bytes) = 00000000000000000000000000000000
                        4db923dc793ee6497c76dcc03a98e108
AAD (0 bytes) =
Key = 00000000000000000000000000000000
      00000000000000000000000000000000
      00000000000000000000000000000000
Nonce = 00000000000000000000000000000000
Record authentication key = dc95c078a24089895275f3d86b4fb868
Record encryption key = 779b38d15bffb63d39d6e9ae76a9b2f3
                        75d11b0e3a68c422845c7d4690fa594f
POLYVAL input = 00000000000000000000000000000000
                  4db923dc793ee6497c76dcc03a98e108
                  00000000000000000000000000000000
POLYVAL result = 7367cdb411b730128dd56e8edc0eff56
POLYVAL result XOR nonce = 7367cdb411b730128dd56e8edc0eff56
... and masked = 7367cdb411b730128dd56e8edc0eff56
Tag = ffffffff00000000000000000000000000000000
Initial counter = ffffffff00000000000000000000000080
Result (48 bytes) = f3f80f2cf0cb2dd9c5984fcd9a908456c
                    c537703b5ba70324a6793a7bf218d3ea
                    ffffffff00000000000000000000000000

```

```

Plaintext (24 bytes) = eb3640277c7ffd1303c7a542d02d3e4c
                        00000000000000000000
AAD (0 bytes) =
Key = 00000000000000000000000000000000
      00000000000000000000000000000000
      00000000000000000000000000000000
Nonce = 00000000000000000000000000000000
Record authentication key = dc95c078a24089895275f3d86b4fb868
Record encryption key = 779b38d15bffb63d39d6e9ae76a9b2f3
                        75d11b0e3a68c422845c7d4690fa594f
POLYVAL input = eb3640277c7ffd1303c7a542d02d3e4c
                  00000000000000000000000000000000
                  00000000000000000000000000000000
POLYVAL result = 7367cdb411b730128dd56e8edc0eff56
POLYVAL result XOR nonce = 7367cdb411b730128dd56e8edc0eff56
... and masked = 7367cdb411b730128dd56e8edc0eff56
Tag = ffffffff00000000000000000000000000000000
Initial counter = ffffffff00000000000000000000000080
Result (40 bytes) = 18ce4f0b8cb4d0cac65fea8f79257b20
                    888e53e72299e56dffffffff00000000
                    00000000000000000000000000000000

```

Acknowledgements

The authors would like to thank Daniel Bleichenbacher, Uri Blumenthal, Deb Cooley's team at NSA Information Assurance, Scott Fluhrer, Tetsu Iwata, Tibor Jager, John Mattsson, Ondrej Mosnacek, Kenny Paterson, Bart Preneel, Yannick Seurin, and Bjoern Tackmann for their helpful suggestions and review.

Authors' Addresses

Shay Gueron
University of Haifa and Amazon
Abba Khoushy Ave 199
Haifa 3498838
Israel

Email: shay@math.haifa.ac.il

Adam Langley
Google LLC
345 Spear St
San Francisco, CA 94105
United States of America

Email: agl@google.com

Yehuda Lindell
Bar-Ilan University and Unbound Tech
Ramat Gan
5290002
Israel

Email: Yehuda.Lindell@biu.ac.il