Galois Extended Mode

Scott Arciszewski

Trail of Bits

Abstract
AES [FIPS-197] is a 128-bit block cipher that underpins most secure communications in the modern era. GCM (NIST SP800-38D) is an authenticated block cipher mode usually used with AES. In this paper, we analyze some limitations of AES-GCM and propose a new block cipher mode, GEM, that extends GCM to provide better security bounds without introducing additional security assumptions. We specify two variants, AES-128-GEM and AES-256-GEM.

Introduction

AES-GCM, which combines the AES block cipher with Galois/Counter Mode, is an authenticated cipher mode that provides strong guarantees of confidentiality and integrity when it is used correctly.

Limitations of AES-GCM

There have been several reported weaknesses in GCM's authentication, including the so-called Forbidden Attack [Joux] if an initialization vector (IV, also called a nonce) is reused and additional weaknesses with truncated tags [Fer05] due to the linear nature of GHASH. Additionally, AES-GCM does not provide protection against so-called Invisible Salamanders [Dod19].

To prevent IV reuse (and therefore the Forbidden Attack), users are expected to encrypt no more than 2^{32} messages under a single key (with randomly selected IVs). This threshold is not arbitrary: It is the point at which, having encrypted 2^n messages, they incur a 2^{-n} probability of a random IV colliding with each additional encryption. This threshold is often called the Birthday Bound, due to its relation to the Birthday paradox.

The designers of AES-GCM specified 96-bit IVs and 32-bit internal counters. The birthday bound of AES-GCM is a result of a 96-bit random nonce. The 32-bit internal counter further limits the length of each encrypted message to $2^{32} - 2$ AES blocks (or $2^{36} - 32$ bytes). This restricts the utility of AES-GCM for encrypting large files in one message, such as virtual machine snapshots or large software deliverables.

Our Contribution

We specify AES-GEM (Galois Extended Mode), a tweak of AES-GCM that addresses the weaknesses described in this introduction with minimal performance overhead or changes to the usual AES-GCM scheme.

AES with Galois Extended Mode (AES-GEM)

AES-GEM can be constructed from the encryption operation of the AES block cipher, the GHASH universal hash function, and the XOR bitwise operation. The AES decrypt function is not necessary.

At a high level, AES-GEM uses AES-CBC-MAC as a fast key derivation function. An extension to the nonce is used with this key derivation function to produce a subkey, which is then used to encrypt messages with a variant of AES-GCM that uses 64-bit IVs and 64-bit counters. Additionally, the output of GHASH is encrypted with AES in ECB mode to add non-linearity to the structure of the authentication tags. This adjustment addresses a weakness identified in [Fer05].

We specify two versions of AES-GCM: One with 128-bit keys, and one with 256-bit keys. AES-128-GEM accepts a 196-bit IV or Nonce, while AES-256-GEM accepts a 256-bit IV or Nonce. We recommend AES-256-GEM for most workloads.

AES-256-GEM

AES-256-GEM uses 256-bit keys with 256-bit nonces. The first 192 bits of nonce are used to produce a 256-bit subkey. The remaining 64 bits of nonce are used for data encryption in Counter Mode.

DeriveSubKey (256-bit mode)

Inputs:

1. Key (K), 256 bits
2. Nonce (N), 192 bits

Algorithm:

1. Set `b0 = AES-CBC-MAC(K, N || 0x4145532D_323536)`
2. Set `b1 = AES-CBC-MAC(K, N || 0x4145532D_47454D)`
3. Return `(b0 || b1) xor K`

Output:

A 256-bit subkey for use with the rest of the algorithm.

Comments:

The constants here, `0x4145532D_323536` and `0x4145532D_47454D` are the ASCII values for "AES-256" and "AES-GEM", respectively.

The full nonce is used for both halves, and is larger than one AES block. The constants fill 15 bytes (120 bits) of the second block, allowing exactly 1 byte of padding for AES-CBC-MAC.

AES-CBC-MAC uses an all-zero initialization vector for simplicity.

The output of the CBC-MAC calls are never revealed directly, so the additional CBC-MAC tweaks (length prefix, encrypting the last block, etc.) are not necessary for our purposes.

The output of each CBC-MAC call has about 128 bits of entropy, except that each half will never collide due to AES being a PRP rather than a PRF.

To alleviate this concern, we borrow a trick from Salsa20's design: XOR the original encryption key with the output of this function to ensure a 256-bit uniformly random distribution of bits to any attacker that doesn't already know the input key.

Encryption (256-bit mode)

Inputs:

1. Key (K), 256 bits
2. Nonce (N), 256 bits
3. Plaintext (P), 0 to $2^{64} - 1$ bits
4. Additional authenticated data (A), 0 to $2^{64} - 1$ bits
5. Authentication tag length (t), 32 to 128 bits

Algorithm:

1. Let `subkey = DeriveSubKey(K, N[0:24])`
2. Let `H = AES-256-ECB(subkey, 0xFFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFF)`
3. Let `j0 = N[24:32] || 0xFFFFFFFF_FFFFFFFE`
4. Let `C = AES-256-CTR(subkey, N[24:32] || 0x00000000_00000000, P)`
5. Let `u = 128 * ceil(len(C)/128) - len(C)` and `v = 128 * ceil(len(A)/128) - len(A)`

6. Let `S = GHASH(H, A || repeat(0, v) || C || repeat(0, u) || len(A) || len(C))` where `repeat(b, x)` is a repeating sequence of length `x` bits with value `b`, as with GCM
7. Let `S2 = AES-256-ECB(K, S)`
8. Let `T = MSB_t(AES-256-CTR(subkey, j0) xor S2)`

Outputs:

1. Ciphertext, C, equal in length to the plaintext P.
2. Authentication tag, T.

Comments:

Where GCM uses a 32-bit internal counter, we specify 64 bits instead. The most significant 64 bits of the counter nonce are the remaining bits from the 256-bit nonce that were not used to derive a subkey.

Applications **MAY** cache the subkey for multiple encryptions if the first 192 bits of the nonce are the same, to save on subkey derivation overhead, but **MUST NOT** reuse the same 256-bit nonce twice for a given key, K.

Although a 64-bit counter would theoretically permit longer plaintexts than 2^{64} bits, the encoding of lengths in the GHASH step is restricted to 2^{64} bits. This is congruent to the maximum length of AAD in AES-GCM. These internal counter values are inaccessible due to GHASH length encoding.

2^{64} bits is equal to 2^{61} bytes (where the size of a byte is 8 bits), which is 2^{57} AES blocks. The range of block counters that AES-CTR can reach can be described by the interval [`0x00000000_00000000`, `0x01ffffff_ffffffff`].

Therefore, we reserve internal counter values `0x02000000_00000000` through `0xffffffff_ffffffff`.

The counter values `0xffffffff_fffffffe` and `0xffffffff_ffffffff` are reserved for j0 (which is used to encrypt the authentication tag) and H (which is the authentication key for GHASH), respectively.

The counter values `0xffffffff_fffffffc` and `0xffffffff_fffffffd` are reserved for calculating an optional key commitment value.

For authentication, the output of GHASH is not used directly, as with AES-GCM. Instead, the output is encrypted in ECB mode using the original key, K, rather than the subkey. This encryption of the GHASH output addresses a weakness with AES-GCM tag truncation [Fer05].

We use a keyed cipher for encrypting the GHASH output, rather than an all-zero key (which would be sufficient for non-linearity), because if an attacker does not know the AES key, they cannot decrypt this value to obtain the raw GHASH, even under a nonce reuse condition. If the universally held assumption that AES is a secure permutation holds true, this encryption of the GHASH state should also reduce the impact of a nonce reuse.

We use K here, rather than the derived subkey, for two reasons:

First, the subkey will be used for encrypting a lot of blocks. It's feasible that some (nonce || counter) block will collide with a GHASH output. Using the subkey for this purpose would require extra considerations. Conversely, K is only otherwise used for key derivation in a way that is never directly revealed to attackers.

Second, this choice binds the permutation of the GHASH output to the original key. If two (key, nonce) pairs produce a subkey collision, unless K is also the same, it remains computationally infeasible to forge authentication tags for all ciphertexts.

Decryption (256-bit mode)

Inputs:

1. Key (K), 256 bits
2. Nonce (N), 256 bits
3. Ciphertext (C), 0 to $2^{64} - 1$ bits
4. Authentication Tag (T), 32 to 128 bits
5. Additional authenticated data (A), 0 to $2^{64} - 1$ bits

Algorithm:

1. Set `subkey = DeriveSubKey(K, N[0:24])`
2. Let `H = AES-256-ECB(subkey, 0xFFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFF)`
3. Let `j0 = N[24:32] || 0xFFFFFFFF_FFFFFFFE`
4. Let `u = 128 * ceil(len(C)/128) - len(C)` and `v = 128 * ceil(len(A)/128) - len(A)`
5. Let `S = GHASH(H, A || repeat(0, v) || C || repeat(0, u) || len(A) || len(C))` where `repeat(b, x)` is a repeating sequence of length `x` bits with value `b`, as with GCM
6. Let `S2 = AES-256-ECB(K, S)`

7. Let `T2 = MSB_t(AES-CTR(subkey, j0) xor S2)`
8. Compare `T` with `T2` in constant-time. If they do not match, abort.
9. Let `P = AES-256-CTR(subkey, N[24:32] || 0x00000000_00000000, C)`

Outputs:

1. Plaintext (P), or error.

Key Commitment

Inputs:

1. Key (K), 256 bits
2. Nonce (N), 192 bits

Algorithm:

1. Set `subkey = DeriveSubKey(K, N[0:24])`
2. Set `P = repeat(0, 32)`
3. Set `Q = AES-256-CTR(subkey, 0xffffffff_fffffffc, P)`

Output:

A 256-bit value that can only be produced by a given input key.

Verifying Key Commitment

Inputs:

1. Key (K), 256 bits
2. Nonce (N), 192 bits
3. Commitment (Q), 256 bits

Algorithm:

1. Set `Q2 = KeyCommit(K, N)`
2. Compare `Q` with `Q2` in constant-time.

Output:

Boolean (Q == Q2)

AES-128-GEM

AES-128-GEM uses 128-bit keys with 192-bit nonces. The first 128 bits of nonce are used to produce a 128-bit subkey. The remaining 64 bits of nonce are used with AES-128-CTR and GHASH as expected.

DeriveSubKey (128-bit mode)

Inputs:

1. Key (K), 128 bits
2. Nonce (N), 128 bits

Algorithm:

1. Set `b = AES-CBC-MAC(K, N[0:24] || 0x47454D2D_313238)`
3. Return `b xor K`

Output:

A 128-bit subkey for use with the rest of the algorithm.

Comments:

The constant `0x47454D2D_313238` is the ASCII string `GEM-128`.

Encryption (128-bit mode)

Inputs:

1. Key (K), 128 bits
2. Nonce (N), 192 bits
3. Plaintext (P), 0 to $2^{64} - 1$ bits
4. Additional authenticated data (A), 0 to $2^{64} - 1$ bits
5. Authentication tag length (t), 32 to 128 bits

Algorithm:

1. Let `subkey = DeriveSubKey(K, N[0:16])`
2. Let `H = AES-ECB(subkey, 0xFFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFF)`
3. Let `j0 = N[16:24] || 0xFFFFFFFF_FFFFFFFE`

4. Let `C = AES-128-CTR(subkey, N[16:24] || 0x00000000_00000000, P)`
5. Let `u = 128 * ceil(len(C)/128) - len(C)` and `v = 128 * ceil(len(A)/128) - len(A)`
6. Let `S = GHASH(H, A || repeat(0, v) || C || repeat(0, u) || len(A) || len(C))` where `repeat(b, x)` is a repeating sequence of length `x` bits with value `b`, as with GCM
7. Let `S2 = AES-128-ECB(K, S)` - GCM does not do this
8. Let `T = MSB_t(AES-128-CTR(subkey, j0) xor S2)`

Outputs:

1. Ciphertext, C, equal in length to the plaintext P.
2. Authentication tag, T.

Comments:

The construction is similar to AES-256-GEM.

Decryption (128-bit mode)

Inputs:

1. Key (K), 128 bits
2. Nonce (N), 192 bits
3. Ciphertext (C), 0 to $2^{64} - 1$ bits
4. Authentication Tag (T), 32 to 128 bits
5. Additional authenticated data (A), 0 to $2^{64} - 1$ bits

Algorithm:

1. Set `subkey = DeriveSubKey(K, N[0:16])`
2. Let `H = AES-128-ECB(subkey, 0xFFFFFFFF_FFFFFFFF_FFFFFFFF_FFFFFFFF)`
3. Let `j0 = N[16:24] || 0xFFFFFFFF_FFFFFFFE`
4. Let `u = 128 * ceil(len(C)/128) - len(C)` and `v = 128 * ceil(len(A)/128) - len(A)`
5. Let `S = GHASH(H, A || repeat(0, v) || C || repeat(0, u) || len(A) || len(C))` where `repeat(b, x)` is a repeating sequence of length `x` bits with value `b`, as with GCM
6. Let `S2 = AES-128-ECB(K, S)`
7. Let `T2 = MSB_t(AES-128-CTR(subkey, j0) xor S2)`
8. Compare `T` with `T2` in constant-time. If they do not match, abort.

9. Let `P = AES-128-CTR(subkey, N[16:24] || 0x00000000_00000000, C)`

Outputs:

1. Plaintext (P), or error.

Key Commitment

Inputs:

1. Key (K), 128 bits
2. Nonce (N), 128 bits

Algorithm:

1. Set `subkey = DeriveSubKey(K, N[0:16])`
2. Set `P = repeat(0, 32)`
3. Set `Q = AES-CTR(subkey, 0xffffffff_fffffffc, P)`

Output:

A 256-bit value that can only be produced by a given input key.

Verifying Key Commitment

Inputs:

1. Key (K), 128 bits
2. Nonce (N), 128 bits
3. Commitment (Q), 256 bits

Algorithm:

1. Set `Q2 = KeyCommit(K, N)`
2. Compare `Q` with `Q2` in constant-time.

Output:

Boolean (Q == Q2)

Security Analysis

The prior work for analyzing the security of AES-GCM can be applied to AES-GCM, except that our design aims to alleviate some of the limitations and weaknesses discovered in GCM.

Beyond Birthday Bound Security

AES-128-GEM combines 128 bits of additional random nonce with a 128-bit encryption key to derive a separate 128-bit AES subkey. The encryption that uses the subkey has a further 64 bits of random nonce. This allows a single 128-bit AES-GEM input key to be used for 2^{80} messages with random nonces before the 2^{-32} collision risk is realized.

AES-256-GEM combines 192 bits of additional random nonce with a 256-bit encryption key to derive a separate 256-bit AES subkey. The encryption that uses the subkey has a further 64 bits of random nonce. This allows a single 256-bit AES-GEM input key to be used for 2^{112}messages with random nonces before the 2^{-32} collision risk is realized.

Secure Encryption for Longer Messages

Both AES-GEM variants support a maximum message length of 2^{61} bytes, or about two exabytes. This is also the maximum length of additional authenticated data for AES-GCM, and is due to the GHASH length encoding being defined over bits rather than bytes.

This new maximum length is far higher than what applications need today.

Secure Short Tags

Encrypting the final GHASH output S with AES eliminates the linear relationship between the ciphertext C and S. This final permutation of the bits in S is performed in addition to the final XOR with the keystream block (`E(j0)`) that AES-GCM has always performed.

The key used to encrypt S to yield S_2 is the same input key that is used to derive subkeys, while the keystream block is derived from the subkey. The relationship between C and S is nonlinear in such a way that knowledge of the input key is necessary to recover the GHASH authentication key, H.

Key Commitment

The optional Key Commitment and Key Commit Verification algorithms defined above utilize the observation that two successive blocks of AES is not expected to collide for two different keys, even when there are 2^{128} AES-256 keys that will convert a given AES plaintext block into a given AES ciphertext block by the pigeonhole principle.

Performance

AES-256-GEM: Subkey derivation requires 4 additional calls to `AES.encrypt()`, compared to AES-GCM, plus 4 XORs of 128-bit blocks.

AES-128-GEM: Subkey derivation requires 2 additional calls to `AES.encrypt()`, compared to AES-GEM, plus 2 XORs of 128-bit blocks.

Both Modes: Calculating the authentication tag requires one additional `AES.encrypt()` call to encrypt the GHASH output, S, to yield S2. Additionally, subkey derivation will incur the cost of setting up an additional AES key schedule. Applications can amortize the key schedule cost by reusing subkeys for many messages.

Key Commitment for both modes costs two additional AES block encryptions and 32 bytes of bandwidth.

AEAD Construction

Should NIST decide to adopt GEM as the basis for a future standard, we define AEAD interfaces [RFC5116] as follows.

AEAD_AES_256_GEM

K_LEN is 32 octets.

P_MAX is 2^61 - 1 octets.

A_MAX is 2^61 - 1 octets.

N_MIN and N_MAX are both 32 octets.

C_MAX is 2^51 + 47 octets.

An AEAD_AES_256_GEM ciphertext is exactly 48 bytes longer than its corresponding plaintext.

AEAD_AES_128_GEM

K_LEN is 16 octets.

P_MAX is 2^61 - 1 octets.

A_MAX is 2^61 - 1 octets.

N_MIN and N_MAX are both 24 octets.

C_MAX is 2^51 + 47 octets.

An AEAD_AES_128_GEM ciphertext is exactly 48 bytes longer than its corresponding plaintext.

Discussion

The authentication tag is 48 octets in both cases. The first 16 bytes is the authentication tag calculated over the message. The remaining 32 bytes come from the Key Commitment algorithm defined for each AES-GEM variant above.

Tag = T || Q

When verifying an AEAD message authentication tag, both components (T, Q) must be verified for the message to be accepted. This verification must be performed in constant-time.

Applications that need shorter authentication tags **MAY** define their own AEAD interface that omits Q and truncates T to an acceptable length.

Acknowledgments

This paper is a draft for submission to the NIST Workshop on the Requirements for an Accordion Cipher Mode 2024. Thanks to Tjaden Hess and Opal Wright for their extensive review of the earlier drafts for this design.

References

- [Dod19] Dodis, et al. “Fast Message Franking: From Invisible Salamanders to Encryptment” https://eprint.iacr.org/2019/016
- [Fer05] Ferguson, Niels. “Authentication Weaknesses in GCM.” https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/cwc-gcm/ferguson2.pdf
- [Joux] Joux, Antoine. “Authentication Failures in NIST Version of GCM.” https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/800-38-series-drafts/gcm/joux_comments.pdf
- [FIPS-197] Federal Information Processing Standards 197, National Institute of Standards and Technology. https://csrc.nist.gov/pubs/fips/197/final
- [NIST-SP800-38D] NIST Special Publication 800-38D. “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.” https://csrc.nist.gov/pubs/sp/800/38/d/final

- [RFC5116] Internet Engineering Task Force RFC 5116. “An Interface And Algorithm for Authenticated Encryption.” https://datatracker.ietf.org/doc/html/rfc5116