Authenticated encryption

Authenticated encryption

- The result of applying in an integrated way encryption and MAC
  - simultaneously
  - for *confidentiality* + *integrity*
- Today considered crucial
  - Recommended in most modern standards (TLS 1.3, IPsec, messaging apps, NIST SP 800-series, ...)
- Universally denoted as AE

On AE

- often offered as single primitive in modern APIs
- makes chosen-ciphertext attacks less dangerous
  - attacker cannot choose a ciphertext and present it to the decryption in a proper way

AE organization

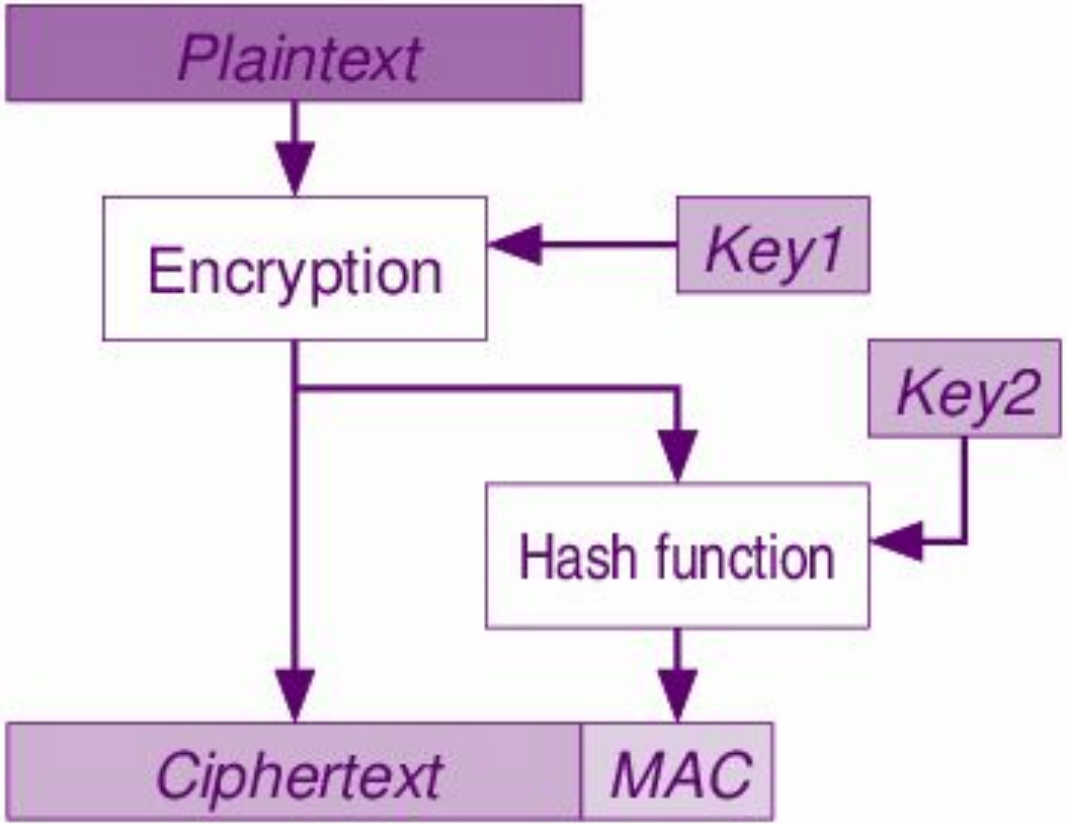
- AE typically combines
  - symmetric encryption algorithm
  - MAC
- Together, these produce
  - ciphertext
  - authentication tag
- Both are needed to decrypt and verify the original message
- Operations (simplified):
  - Input: message + secret key(s)
  - Output: encrypted message (ciphertext) + tag
  - To decrypt: verify the tag → only then reveal the plaintext

Approaches to AE

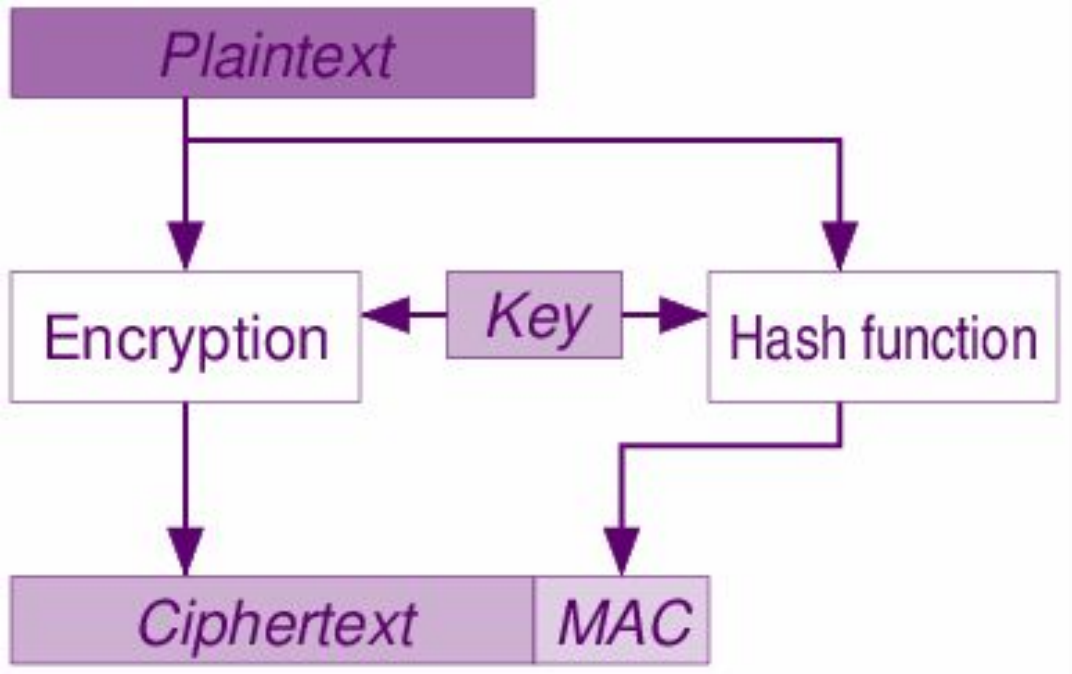
- **Encrypt-then-MAC (EtM)**
  - Widely accepted as the most secure and robust approach
  - Proven secure under strong cryptographic assumptions
- **Encrypt-and-MAC (E&M)**
  - Proving its security is an open and challenging problem
  - Generally considered insecure in practice
- **MAC-then-Encrypt (MtE)**
  - Proven secure in specific, limited settings (e.g., certain SSH implementations)
  - Its security in generic applications remains an open problem and is generally less robust than EtM

EtM

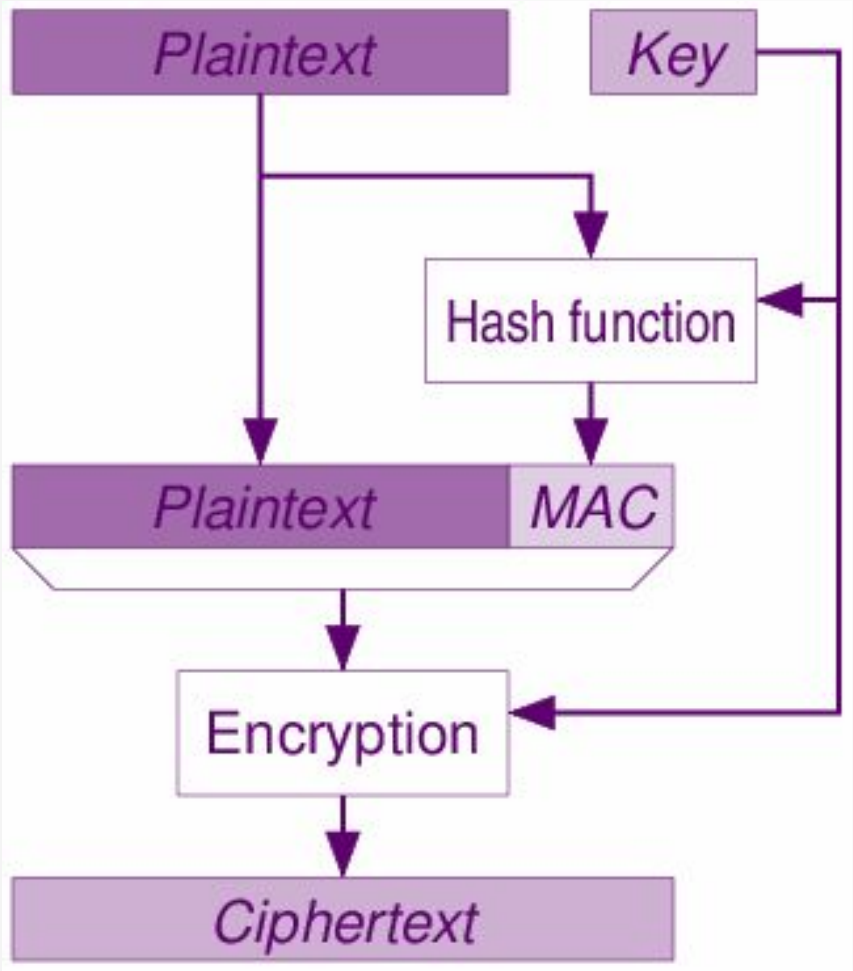
sometimes from a unique master key two subkeys are derived



E&M



MtE


Plaintext
Key
Hash function
Plaintext
MAC
Encryption
Ciphertext

Associated data

- Often, additional information is associated with the data we want to protect. This information may not require confidentiality, but it must be authenticated
    - for example: in a datagram, the payload must be both confidential and authentic, while the header only needs to be authentic
- In these cases, we better describe the scene by talking of Authenticated Encryption with Associated Data (AEAD)
    - associated data are often referred as Additional Authenticated Data (AAD)

Update

EtM

- Tag = MAC(Key2, AAD||Ciphertext||len(AAD)||len(Ciphertext))

E&M

- Tag = MAC(Key, AAD||M)
- vulnerable to CCA

MtE

1. Tag = MAC(Key2, AAD || Plaintext)
2. Input = Plaintext || tag
3. Ciphertext = ENC(Key1, input)
4. Send: Ciphertext + AAD (in clear)

AEAD: GCM

What is GCM?

- It's block mode of operation
    - It contains a MAC
- Galois/Counter Mode
- AEAD mode based on:
    - CTR mode for encryption
    - GHASH (the MAC, using polynomial hashing over $GF(2^{128})$)
    - Structure: EtM
- Designed for speed and parallelism
- In practice: when we use AES + GCM we have AEAD

Structure Overview

1. **Encryption**: Counter mode (CTR)
    - Block cipher (e.g., AES) in counter mode for speed

2. **Authentication**: GHASH function
    - Polynomial hash over GF(2^{128})

GCM Inputs and Outputs

Inputs:

- Key K (128 or 256 bits)
- IV (96 bits recommended)
- Plaintext P
- Associated Authenticated Data (AAD)

Outputs:

- Ciphertext C
- Authentication Tag T

Counter Mode Encryption

- $C_i = P_i \oplus E_K(ctr_i)$
- Counters derived from IV (nonce)
- Allows parallel block encryption
- Decryption identical: $P_{\_i} = C_{\_i} \oplus E_{\_K}(ctr_i)$

Note: Deterministic per IV; IV reuse breaks confidentiality

GHASH Function

Authentication Component

- Computes a polynomial hash over $GF(2^{128})$
- $H = E_K(0^{128})$
- $X_i = (X_{i-1} \oplus C_i) \cdot H$
- Includes AAD and lengths of AAD (in short: A) and C

Note: Multiplication in $GF(2^{128})$ is key to performance

Input to GHASH

Input: AAD || Ciphertext || length block

1. Take the AAD, pad with zeros to a multiple of 128 bits
2. Take the ciphertext, pad with zeros to a multiple of 128 bits
3. Append one final 128-bit length block, consisting of the bit-lengths of AAD and ciphertext (each encoded as a 64-bit integer, big-endian)

Insight on GHASH

- Inputs: AAD || Ciphertext || length block
- XOR and multiply in GF(2^{128}) with hash subkey H
- GHASH($X_1, X_2, \ldots, X_n$) = (((...((X_1 * H $\oplus$ X_2) * H $\oplus$ ...) * H) $\oplus$ X_n) mod P(x)
    - $\oplus$ = bitwise XOR (addition in GF(2^{128}))
    - * = carry-less multiplication in GF(2^{128})
    - $P(x) = x^{128} + x^7 + x^2 + x + 1$ is the fixed irreducible polynomial used for reduction
- Allows derivation of the 128-bit authentication tag

The Initial Counter Block J_0

Purpose

- Defines the starting counter value for AES in GCM
- Used both in encryption and authentication

Computation of J_0

- If IV = 96 bits (recommended): $J_0 = IV \ // \ 0^{31} \ // \ 1$
- If IV ≠ 96 bits: $J_0 = GHASH(H, IV)$

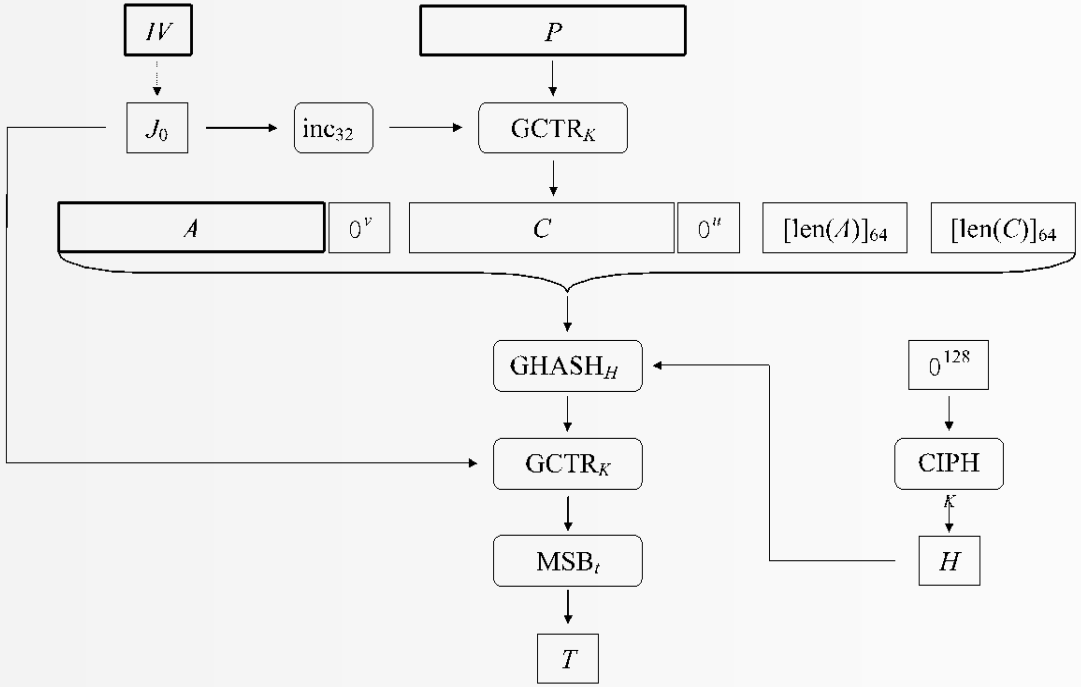
Usage

- Counter blocks: $J_i = inc_{32}(J_{i-1})$ (*increment the least significant 32 bits modulo 2^{32}, leaving the upper 96 bits unchanged*)
- Tag computation: $T = E_K(J_0) \oplus GHASH(A, C)$ (first 128 bits)

High-level summary of operations

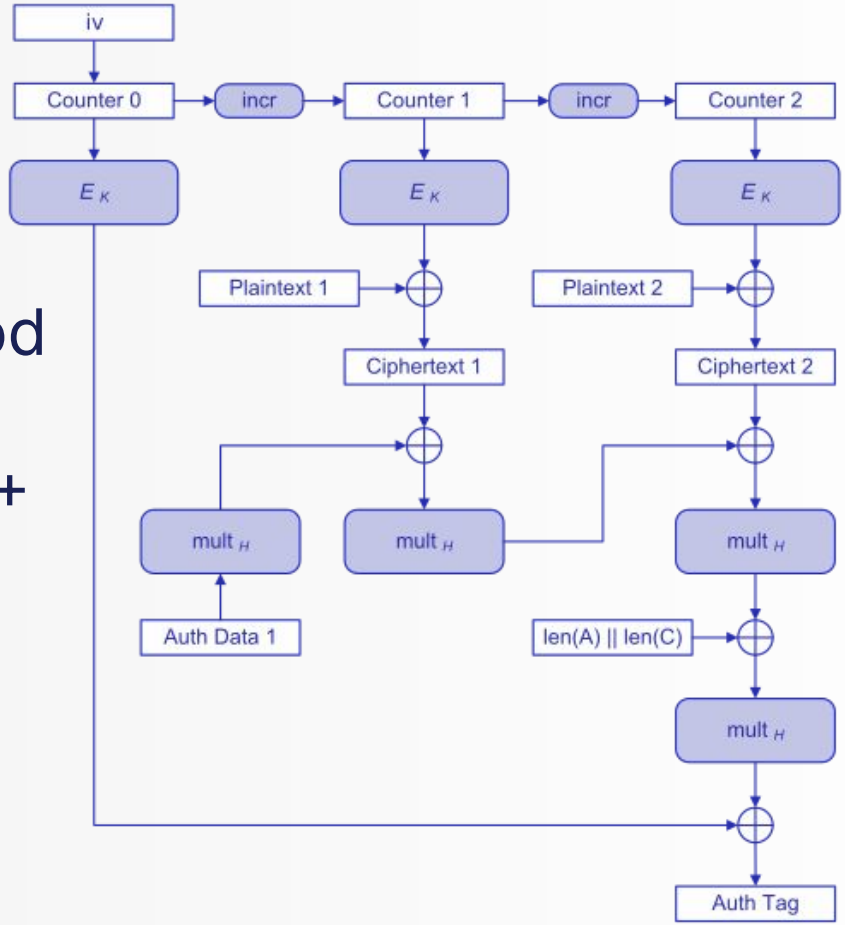
- Encrypt plaintext using CTR mode → ciphertext
- GHASH(A, C) → authentication value
- Compute J_0 from IV
- Tag = $AES_K(J_0) \oplus$ "GHASH output"

NIST description



A 2-blocks scheme

- $mult_H(X) = (X \cdot H) \bmod P(x)$
  - $P(x) = x^{128} + x^7 + x^2 + x + 1$
- is part of GHASH



Decryption

Element	Purpose
1. Ciphertext	The encrypted message
2. Authentication Tag (T)	Ensures integrity & authenticity
3. Nonce / IV	Builds counter blocks & J_0
4. AAD (Associated Data)	Authenticated, not encrypted
5. Key (K)	Used for both AES and GHASH

Security and pitfalls

- Strong confidentiality + integrity
- Efficient in hardware
- Critical: never reuse IV with same key
- Tag forgery risk if GHASH is misused

Adoption and performance GCM

- TLS, IPsec, MACsec, disk encryption
- Optimized via AES-NI, ARM crypto extensions
- Parallelizable (GHASH is tricky): good for high throughput

AEAD: Poly1305

What is Poly1305?

- Poly1305 is a MAC algorithm
- Created by Daniel J. Bernstein in 2005
- Designed to be highly secure and incredibly fast, especially on modern processors
- It is often paired with the ChaCha20 stream cipher to form ChaCha20-Poly1305, a powerful **authenticated encryption** scheme

Poly1305's core components

- One-Time Key (k). A 256-bit secret key, divided into two 128-bit parts
    - r (polynomial key): used in the modular multiplication steps
    - s (addition key): added at the final step to produce the authentication tag
- Message (M): the data to authenticate
- Nonce: a unique per-message value (counter or random number)
    - Used together with the encryption key to derive the one-time key k = (r, s)

ChaCha20 and Poly1305

- When using ChaCha20 together with Poly1305, a 256-bit one-time key (k) is derived from ChaCha20’s output
- The key is generated by encrypting a 256-bit block of zeros with counter = 0 and the given key and nonce
- The resulting 256 bits are split into two 128-bit halves:
    - first half → r (polynomial key)
    - second half → s (addition key)

Poly1305's core components

- The core of Poly1305 is the evaluation of a polynomial over a finite field
- The message M is divided into 128-bit blocks m_1, ..., m_{n+1}
- Each block is treated as a coefficient of a polynomial p(x)
- The polynomial is evaluated at a secret point r modulo a prime $p = 2^{130} - 5$:
  $p(r) = m_1 r^n + m_2 r^{n-1} + \cdots + m_n r + m_{n+1} \pmod p$
- The result is then added to the second part of the one-time key s to produce the authentication tag
- Security relies on the difficulty of forging a valid

Step-by-Step Procedure (Part 1)

Start

- Divide M into 16-byte (128-bit) blocks
  - If the last block is shorter, pad it with zeros to 16 bytes
  - Append one extra byte 0x01 to each block: total 17 bytes per block
- Extract the 128-bit r (polynomial key) from the one-time key k
- Initialize the accumulator: acc = 0

Loop: for each padded 17-byte block m_i

- Interpret m_i as a little-endian integer $block_i$
- Update the accumulator:
  $acc = ((acc + block_i) \times r) \bmod p,\ p = 2^{130} - 5$

After all blocks are processed: p(r) = acc

Step-by-Step Procedure (Part 2)

- After loop, the other part of the secret key, s, is extracted
- tag=(acc+s) mod 2^{128}
- The s key is added to prevent simple brute-force attacks on p(r)
- **Verification**
    - Recipient performs the exact same steps
    - If their newly calculated tag matches the received tag, the message is authenticated

Key Generation and constraints

- Key k is 256 bits
  - First 128 bits → r, the polynomial key
  - Second 128 bits → s, the addition key
- Constraints on r (clamping)
  - For security and correctness, specific bits of r are cleared to avoid small-subgroup attacks and simplify modular reduction
  - In little-endian form, bits 0–1, 2, and bits 126–127, 123–122, 119–118 are cleared
  - This ensures r is always less than 2^{130} and divisible by 4, making multiplication mod 2^{130}-5 efficient and deterministic
- Security requirement
  - Each Poly1305 key k = (r, s) is one-time use
  - Reusing the same key with the same nonce across messages is a critical vulnerability that allows tag forgery

Security and performance

Security

- Poly1305 is provably secure based on standard cryptographic assumptions (specifically, that ChaCha20 is a secure pseudorandom function family)
- The probability of an attacker forging a valid tag is extremely low, approximately 8×2^{-106} per attempt
- This is a "one-time MAC" and should only be used with a unique nonce for each message

Performance

- Poly1305 is exceptionally fast
- It performs very few multiplications, additions, and modular reductions per message block
- These operations are highly optimized for modern 64-bit processors, making it a performance leader among MAC algorithms
- It is often cited as being faster than HMAC-SHA256

Real-World Applications

- **ChaCha20-Poly1305**: the most common usage. It combines the speed of the ChaCha20 stream cipher for encryption with the speed of Poly1305 for authentication
- **Transport layer security (TLS 1.2 and 1.3)**: the ChaCha20-Poly1305 cipher suite is a highly recommended and widely used option for securing web traffic
- **OpenSSH**: Used as part of the chacha20-poly1305@openssh.com cipher
- **Android**: part of the Android's BoringSSL library.
- **Internet of things (IoT)**: due to its low resource requirements and high speed, it is an excellent choice for devices with limited computing power