Block Ciphers

AUTHENTICATED ENCRYPTION – ACTIVE ATTACKS

CREDITS: PROF. DAN BONEH

Recap: the story so far

Confidentiality: semantic security against a CPA attack

- Encryption secure against eavesdropping only
- Examples: Using CBC with a PRP (e.g. AES)

Integrity: Existential unforgeability under a chosen message attack

- Examples: CBC-MAC, HMAC, PMAC, CW-MAC

What we want: encryption secure against **tampering**

- Ensuring both **confidentiality and integrity**
- Against a more powerful attacker → **Active Attacker**

The lesson

CPA security cannot guarantee secrecy under active attacks.

If message needs integrity but no confidentiality:

- Use Message Authentication Code (MAC)

If message needs both integrity and confidentiality:

- Use Authenticated Encryption Modes

The lesson

CPA security cannot guarantee security under active attacks.

Integrity Only	→	✓ Secure MAC
Integrity + Secrecy	→	✗ Secure MAC + Secure Cipher
Integrity + Secrecy	→	✓ ***Authenticated Encryption***

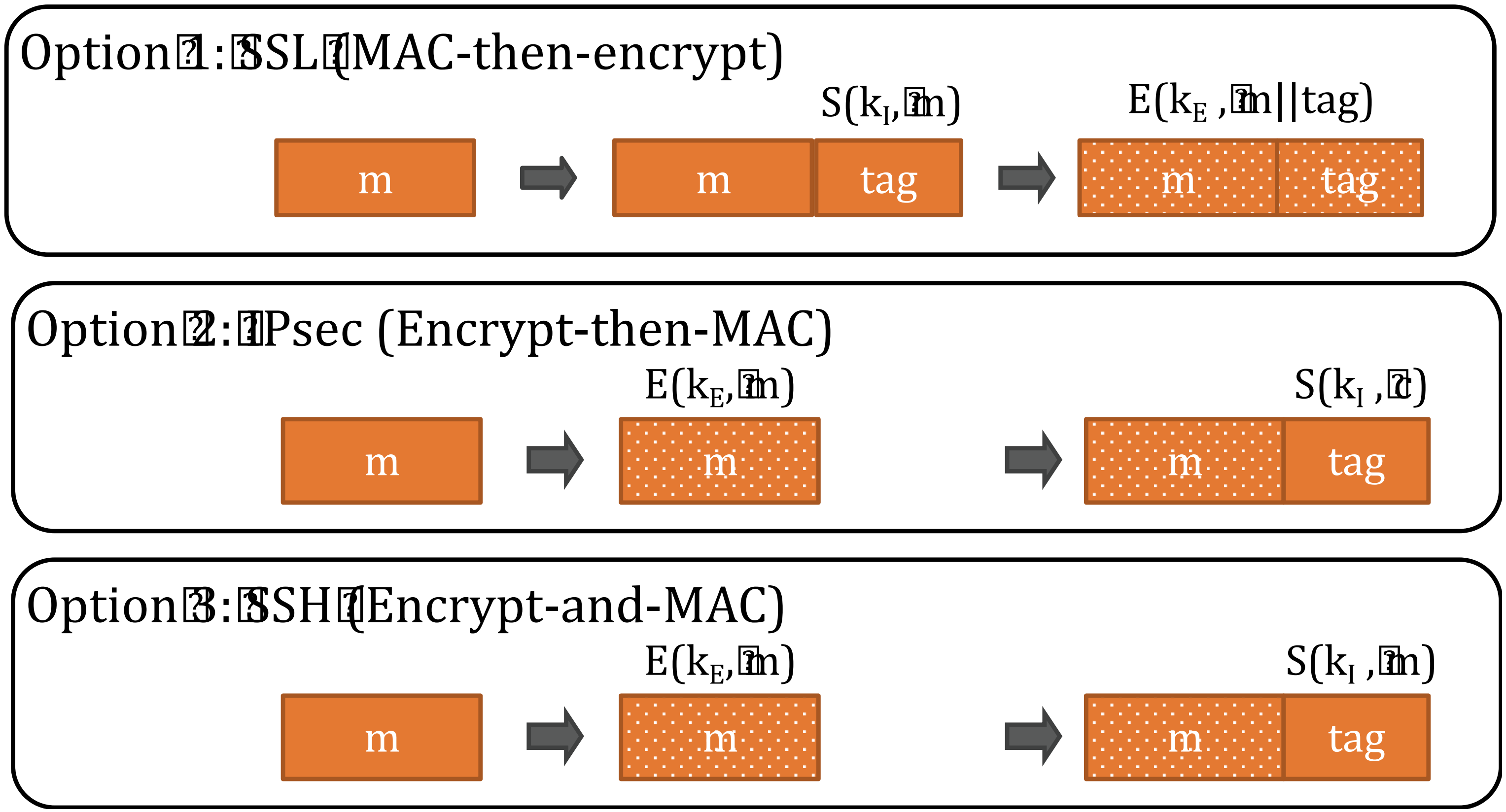
Associated Data

What if some parts of the message needs to be encrypted and some others need to be plaintext but want to ensure integrity on both?

Just think the payload of a packet and the TCP port!

The question: which one is better?

Encryption key k_E ***MAC key =*** k_I



Authenticated Encryption

Authenticated encryption (**AE**) and **authenticated encryption with associated data** (**AEAD**) are forms of encryption which simultaneously assure the confidentiality and authenticity of data.

These attributes are provided under a single, easy to use programming interface.

Authenticated Encryption

A cipher (E,D) provides authenticated encryption (AE) if it is

- semantically secure under CPA, and
- has ciphertext integrity

Bad example:

- CBC with rand. IV does not provide AE
- $D(k,\cdot)$ never outputs $\perp$, hence adv. easily wins the game

Programming Interface

A typical programming interface for an AE implementation provides the following functions:

Encryption

- Input: *plaintext*, *key*, and optionally a *header* in plaintext that will not be encrypted, but will be covered by authenticity protection.
- Output: *ciphertext* and *authentication tag* (MAC).

Decryption

- Input: *ciphertext*, *key*, *authentication tag*, and optionally a *header* (if used during the encryption).
- Output: *plaintext*, or an error if the *authentication tag* does not match the supplied *ciphertext* or *header*.

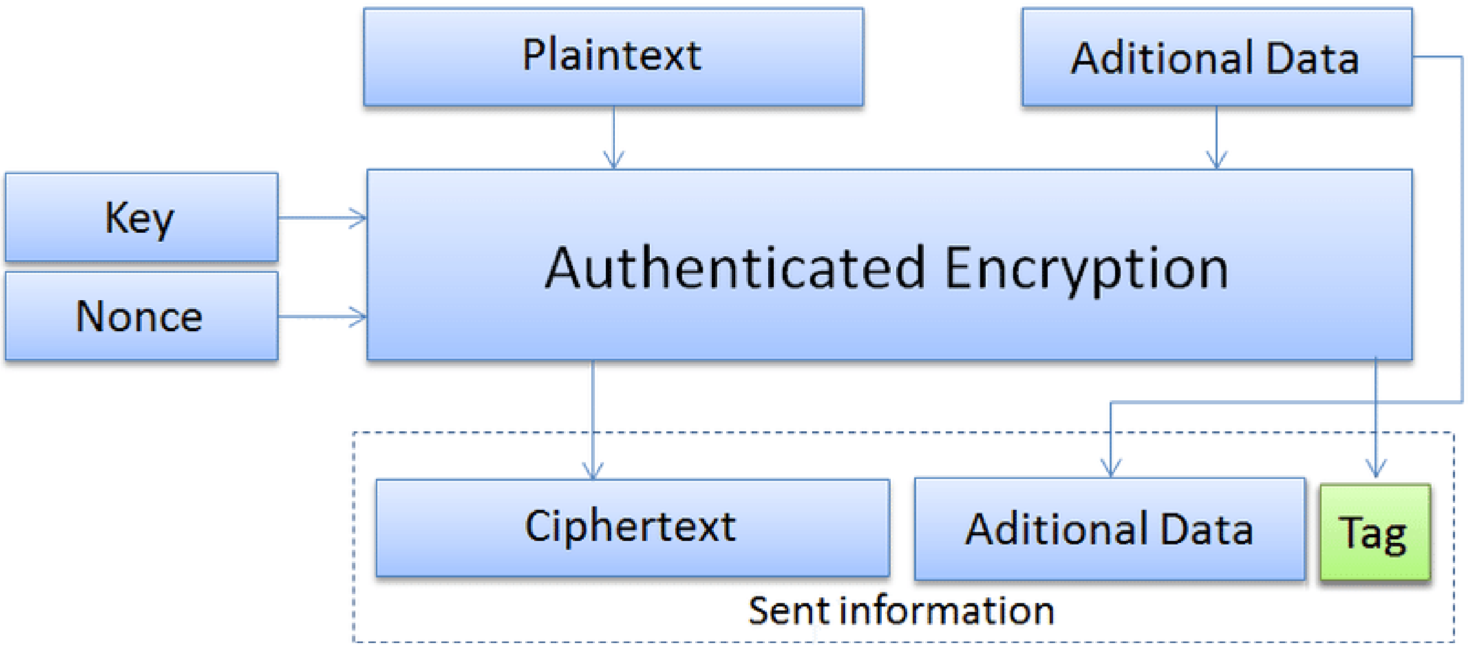
AEAD

AEAD is a variant of AE that allows a recipient to check the integrity of both the encrypted and unencrypted information in a message.

AEAD binds associated data (AD) to the ciphertext and to the context where it is supposed to appear, so that attempts to "cut-and-paste" a valid ciphertext into a different context are detected and rejected.

It is required, for example, by network packets. The header needs **integrity**, but must be visible; payload, instead, needs **integrity** and also **confidentiality**. Both need **authenticity**.

AEAD


Plaintext
Aditional Data
Key
Nonce
Authenticated Encryption
Ciphertext
Aditional Data
Tag
Sent information

So what?

Authenticated encryption:

- ensures confidentiality against an active adversary that can decrypt some ciphertexts

Limitations:

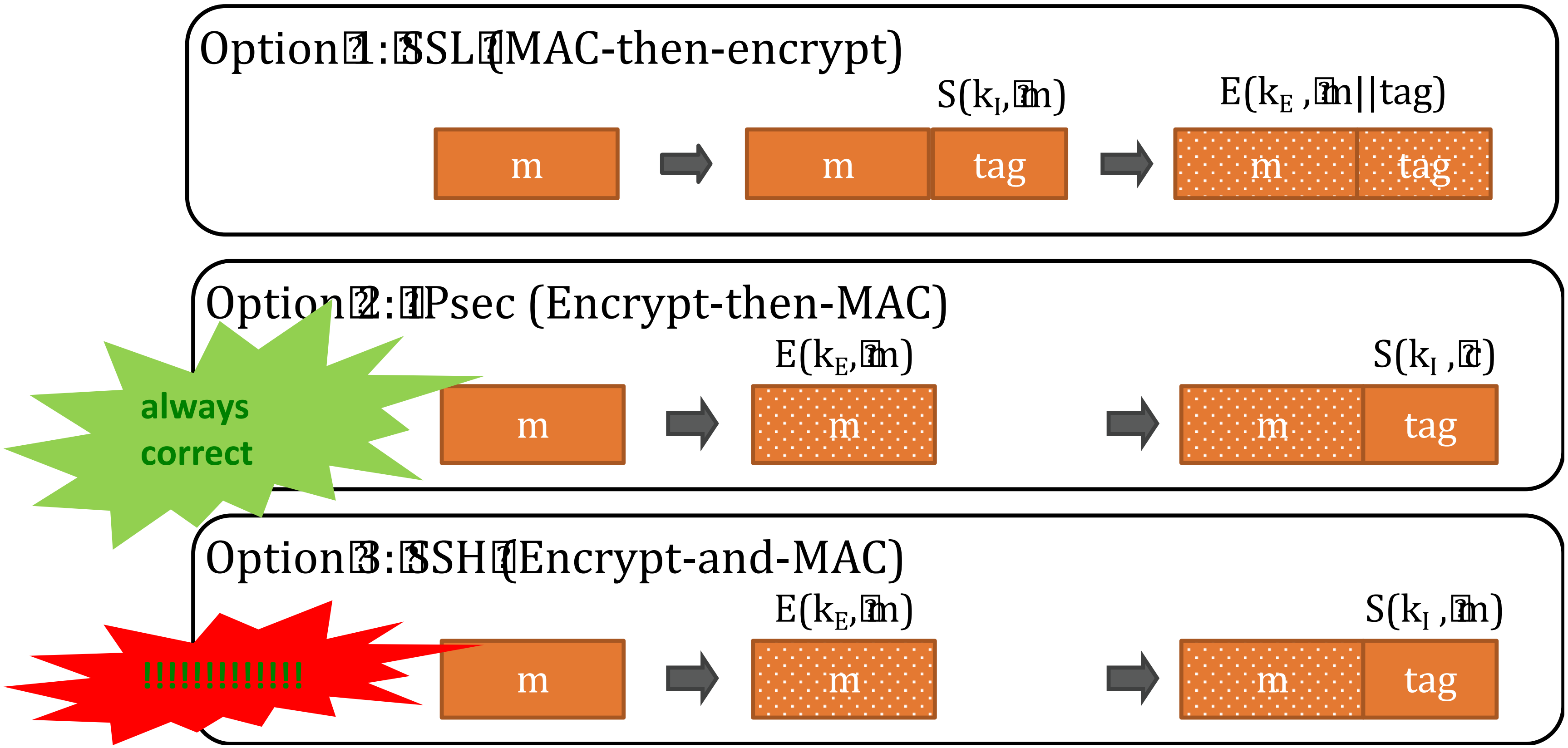
- does not prevent replay attacks
- does not account for side channel attacks

Authenticated Encryption

CONSTRUCTION

The question: which one is better?

Encryption key k_E ***MAC key =*** k_I



A.E. Theorems

Let (E,D) be CPA secure cipher and (S,V) secure MAC. Then:

Encrypt-then-MAC:

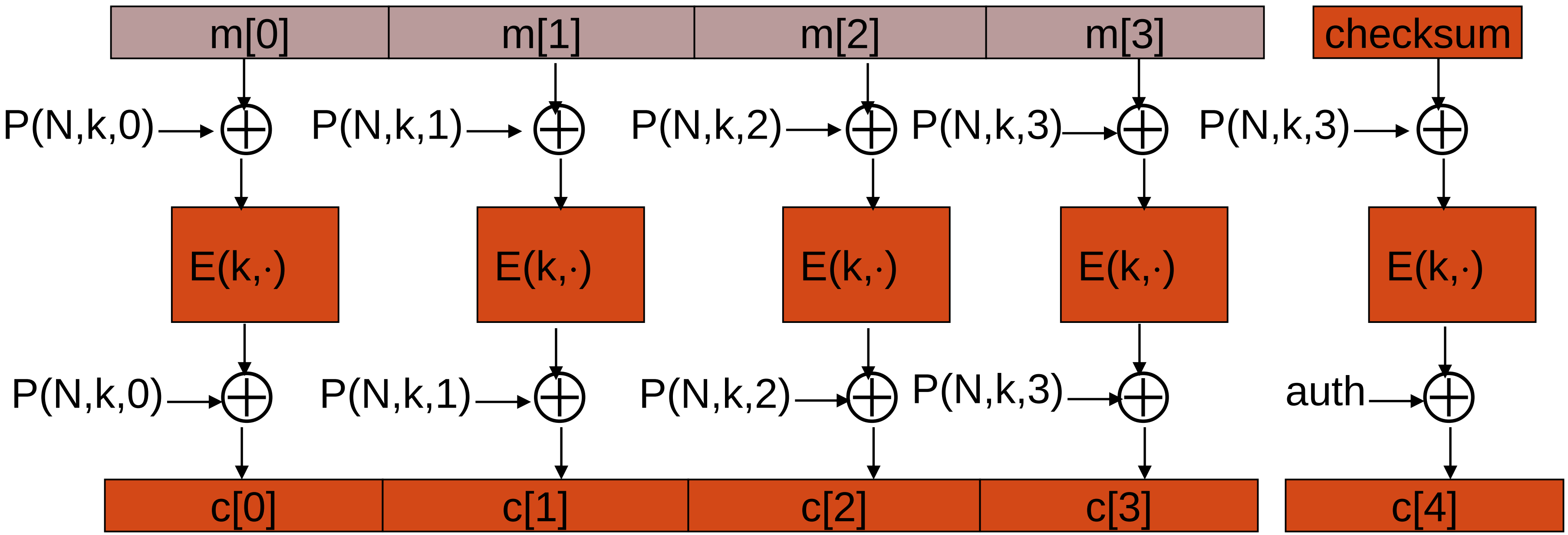
- always provides A.E.

MAC-then-encrypt:

- may be insecure against CCA attacks
- when (E,D) is rand-CTR mode or rand-CBC M-then-E provides A.E.
- for rand-CTR mode, one-time MAC is sufficient

OCB (Offset Codebook Mode)

More efficient authenticated encryption: one E() op. per block.



Standards (at a high level)

GCM

- CTR mode encryption then CW-MAC

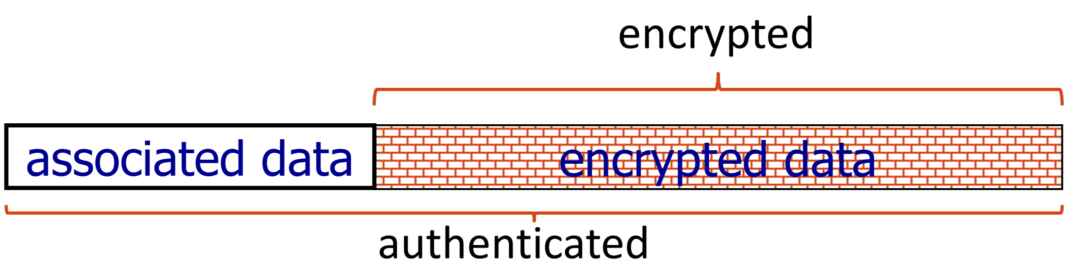
CCM

- CBC-MAC then CTR mode encryption (802.11i)

EAX

- CTR mode encryption then CMAC

All support AEAD: (auth. enc. with associated data) and ALL are nonce-based.



```c
int gcm_encrypt(unsigned char *plaintext, int plaintext_len,
                unsigned char *aad, int aad_len,
                unsigned char *key,
                unsigned char *iv, int iv_len,
                unsigned char *ciphertext,
                unsigned char *tag)
{
    EVP_CIPHER_CTX *ctx;

    int len;

    int ciphertext_len;


    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /* Initialise the encryption operation. */
    if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, NULL, NULL))
        handleErrors();

    /*
     * Set IV length if default 12 bytes (96 bits) is not appropriate
     */
    if(1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, iv_len, NULL))
        handleErrors();

    /* Initialise key and IV */
    if(1 != EVP_EncryptInit_ex(ctx, NULL, NULL, key, iv))
        handleErrors();

    /*
     * Provide any AAD data. This can be called zero or more times as
     * required
     */
    if(1 != EVP_EncryptUpdate(ctx, NULL, &len, aad, aad_len))
        handleErrors();

    /*
     * Provide the message to be encrypted, and obtain the encrypted output.
     * EVP_EncryptUpdate can be called multiple times if necessary
     */
    if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len))
        handleErrors();
    ciphertext_len = len;

    /*
     * Finalise the encryption. Normally ciphertext bytes may be written at
     * this stage, but this does not occur in GCM mode
     */
    if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len))
        handleErrors();
    ciphertext_len += len;

    /* Get the tag */
    if(1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag))
        handleErrors();

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    return ciphertext_len;
}
```

```c
int gcm_decrypt(unsigned char *ciphertext, int ciphertext_len,
                unsigned char *aad, int aad_len,
                unsigned char *tag,
                unsigned char *key,
                unsigned char *iv, int iv_len,
                unsigned char *plaintext)
{
    EVP_CIPHER_CTX *ctx;
    int len;
    int plaintext_len;
    int ret;

    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new()))
        handleErrors();

    /* Initialise the decryption operation. */
    if(!EVP_DecryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, NULL, NULL))
        handleErrors();

    /* Set IV length. Not necessary if this is 12 bytes (96 bits) */
    if(!EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, iv_len, NULL))
        handleErrors();

    /* Initialise key and IV */
    if(!EVP_DecryptInit_ex(ctx, NULL, NULL, key, iv))
        handleErrors();

    /*
     * Provide any AAD data. This can be called zero or more times as
     * required
     */
    if(!EVP_DecryptUpdate(ctx, NULL, &len, aad, aad_len))
        handleErrors();

    /*
     * Provide the message to be decrypted, and obtain the plaintext output.
     * EVP_DecryptUpdate can be called multiple times if necessary
     */
    if(!EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext_len))
        handleErrors();
    plaintext_len = len;

    /* Set expected tag value. Works in OpenSSL 1.0.1d and later */
    if(!EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, 16, tag))
        handleErrors();

    /*
     * Finalise the decryption. A positive return value indicates success,
     * anything else is a failure - the plaintext is not trustworthy.
     */
    ret = EVP_DecryptFinal_ex(ctx, plaintext + len, &len);

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    if(ret > 0) {
        /* Success */
        plaintext_len += len;
        return plaintext_len;
    } else {
        /* Verify failed */
        return -1;
    }
}
```