EECS 388

Introduction to Computer Security

Lecture 5:

Combining Confidentiality and Integrity

September 12, 2023

Prof. Halderman

Padding and Block Cipher Modes

Challenge for block ciphers:

How to encrypt **arbitrary-sized messages**?

Padding: Add bytes to end of message to make it a multiple of block size

Flawed approach: add zeros [What's the issue?]

```
| MM MM MM MM MM 00 00 00 |
```

Don't know what to remove after decryption!

Better approach (**PKCS7**): Add **n** bytes of value **n**

```
| MM MM MM MM MM 03 03 03 |
```

Edge case: Message that ends at block boundary?

```
| MM MM MM MM MM MM MM MM | 08 08 08 08 08 08 08 08 |
```

Add an **entire block** of padding

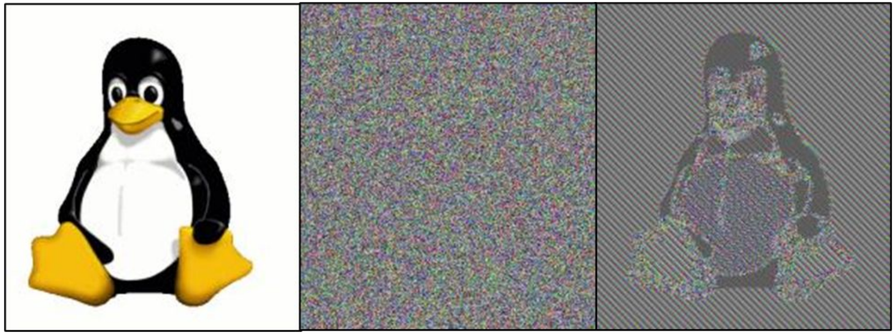
Ensures receiver can ***unambiguously*** distinguish the padding from the message after decrypting

Cipher modes: Algorithms for applying block ciphers to more than one block

Flawed approach: [What's the issue?]

Encrypted codebook (ECB) mode

Simply encrypt each block independently: $\mathbf{c}_i := E_k(\mathbf{p}_i)$



Plaintext Pseudorandom ECB mode

Cipher Modes

Cipher-block chaining (CBC) mode

"Chains" ciphertexts to obscure later ones

Choose a random **initialization vector IV**

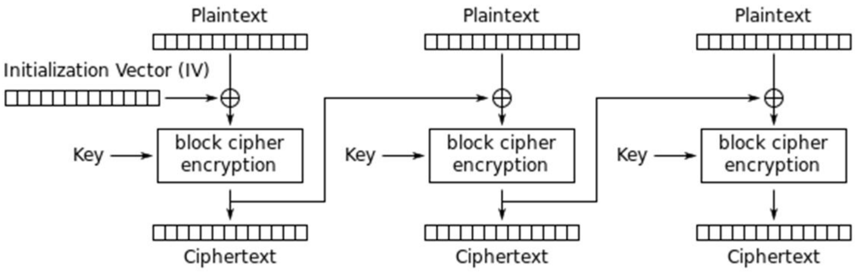
Encrypt: $\mathbf{c}_0 := \mathbf{IV};\ \mathbf{c}_i := E_k(\mathbf{p}_i \oplus \mathbf{c}_{i-1})$

Decrypt: $\mathbf{p}_i := D_k(\mathbf{c}_i) \oplus \mathbf{c}_{i-1}$,

[Why do we need the IV?]

Have to send IV with ciphertext

Can't encrypt blocks in parallel or out of order



Counter (CTR) mode

Turns a block cipher into a stream cipher

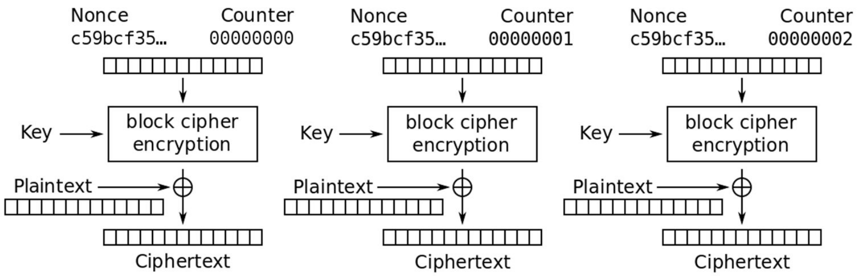
Generate **keystream s** for **k** and unique **nonce**:

$\mathbf{s} := E_k(\mathbf{nonce} \,//\, 0) \;//\; E_k(\mathbf{nonce} \,//\, 1) \;//\; E_k(\mathbf{nonce} \,//\, 2) \;//\; \ldots$

Encrypt: $\mathbf{c} := \mathbf{p} \oplus \mathbf{s}$ Decrypt: $\mathbf{p} := \mathbf{c} \oplus \mathbf{s}$

Benefits: Doesn't require padding
Efficient parallelism/random access

Caution: Never reuse **nonce** for same **k**!



Review: Integrity and Confidentiality

Integrity (~~tampering~~)

Let *f*() be a secure PRF.
In practice: e.g., **HMAC-SHA-256**

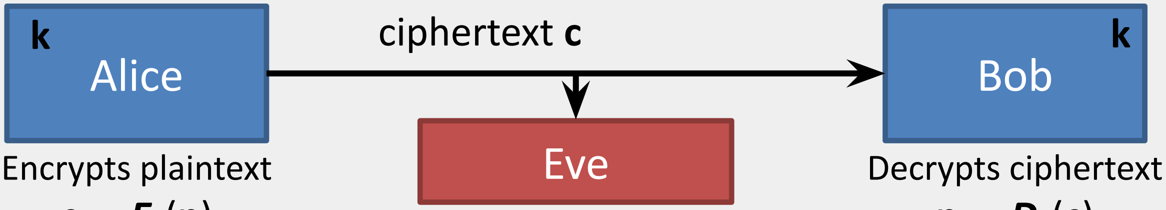
Computes *verifier*
$\mathbf{v} := f_k(\mathbf{m})$

"Malice-in-the-middle"
(an **active attacker**)

Rejects message if
$\mathbf{v}' \neq f_k(\mathbf{m}')$

Confidentiality (~~eavesdropping~~)

Construct ***E***() and ***D***() from secure PRG (a stream cipher) ***or*** secure PRP (a block cipher) with appropriate padding/cipher mode
In practice: e.g., **AES-128 in CTR mode**



Encrypts plaintext
$\mathbf{c} := E_k(\mathbf{p})$

Passive Eavesdropper
(a **passive attacker**)

Decrypts ciphertext
$\mathbf{p} := D_k(\mathbf{c})$

Today's lecture:

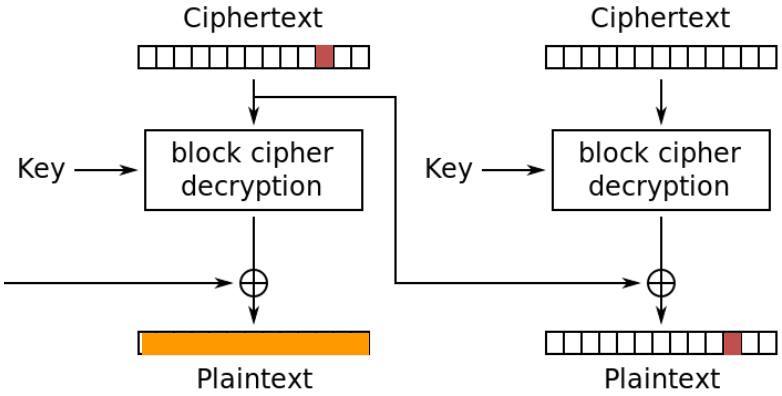
What if we want integrity and confidentiality *at the same time*?

Ciphertext Malleability

Caution: Many encryption methods are **malleable**: can transform a ciphertext into another ciphertext that decrypts to a related plaintext, without knowing the plaintext

Examples:

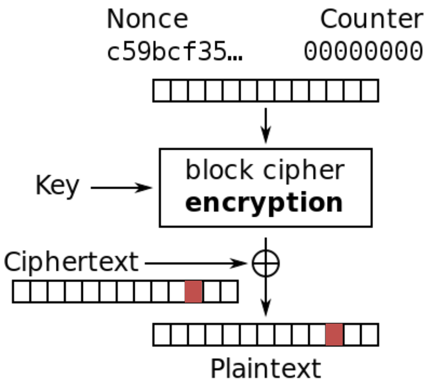
CBC mode decryption



Flipping bits in ciphertext block **i** will:

- completely corrupt decrypted block **i**
- flip corresponding bits in decrypted block **i**+1

Counter mode decryption



Flipping bits anywhere in the ciphertext will flip corresponding bits in decrypted plaintext

Need to use other methods to ensure integrity…

Authenticated Encryption (AE)

Two approaches:

1. Generically compose encryption and MAC
2. Build "all-in-one" primitive that does both

Syntax of AE:

c := E_k(p)

p/**"fail"** := D_k(c)

Important difference:
Decryption can fail!
Analogous to Bob rejecting verifier

Security definition:

1. Let **k** be a secret seed
2. Toss a coin (in secret) to get bit **b**
3. If **b**=0: ***G***() := E_k(); ***H***() := D_k()*

   *** Rejects previous *E*() outputs**

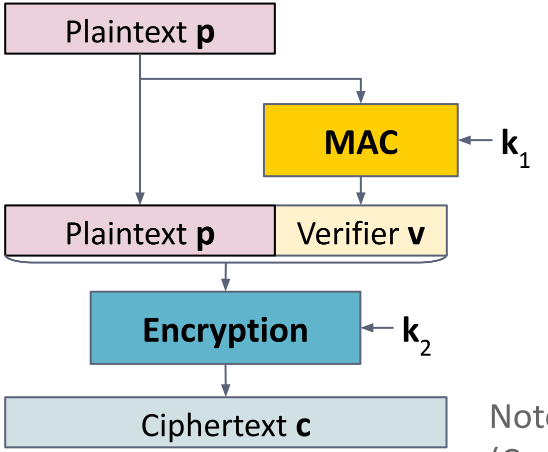
   If **b**=1: ***G***() := random bits; ***H***() := "fail"
4. Give Mallory ***G***()/***H***() oracles
   (Mallory gets to repeatedly probe them)
5. Mallory guesses **b** *in polynomial time*

We say AE is **secure** if Mallory can't do meaningfully better than random guessing.

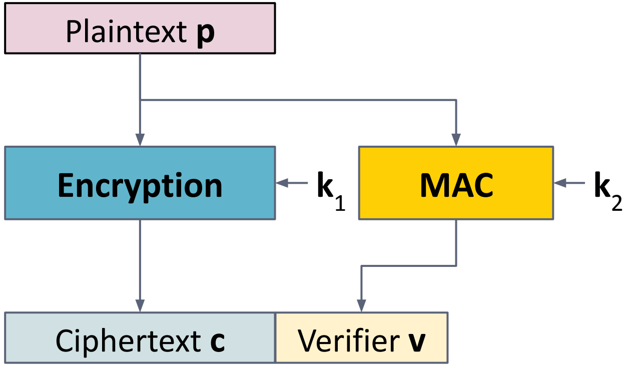
Composing Integrity and Confidentiality

How to ***compose*** our integrity and confidentiality protocols to achieve both? Three candidates:

MAC-then-Encrypt

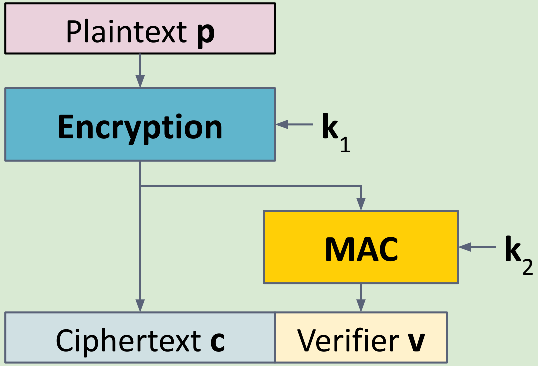


Encrypt-and-MAC



Note: **Use separate keys** for different purposes.
(Can derive from a single key using PRG.) [Why?]

Encrypt-then-MAC



Safest approach

[Which approach is safest?]

Our encryption methods (so far) only secure against passive eavesdroppers.
Only EtM can ensure ciphertext isn't tampered with before decryption.

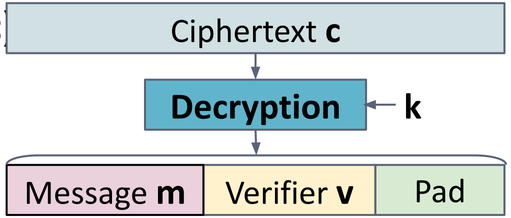
"Cryptographic Doom Principle": if you perform any cryptographic operations on a message you've received before verifying the MAC, it will somehow inevitably lead to doom

Example: CBC Padding Oracles

Common flaw when using **MAC-then-Encrypt**

Suppose an implementation uses **CBC mode**.
Decryption involves the following steps:

1. **m** // **v** // **pad** := D_k(**c**)
2. Check that **pad** is valid **PKCS7**, else raise PadError



3. Check that **v** = MAC_k(**m**), else raise MacError

This is how TLS 1.0 worked. Seems reasonable?

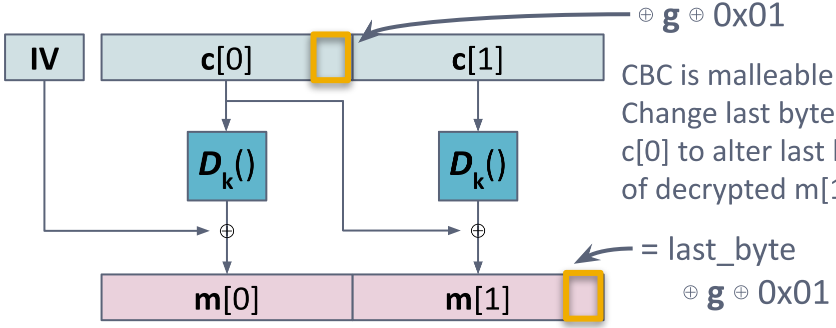
Any method to distinguish these two error types (even tiny timing differences) **leaks the plaintext!**

Padding oracle: attacker submits any ciphertext and learns if last bytes of plaintext are a valid pad

Example of a **chosen ciphertext attack**

Suppose attacker intercepts **c**, wants to learn **m**.

Step 1: Let **g** be a guess for last byte of block **m**[1]:



CBC is malleable: Change last byte of c[0] to alter last byte of decrypted m[1]

Step 2: Send modified ciphertext to padding oracle

If **g** = last_byte: **g** ⊕ last_byte ⊕ 0x01 = 0x01.
Modified plaintext ends in 0x01, so padding's valid; oracle returns MacError
else: Padding is invalid*, oracle returns PadError

(*Except for edge cases: e.g., what if m[1] ends in 0x02 0x01 and g = 0x02?)

Step 3: Repeat with **g** = 0,1, … 255 to learn last_byte.
Then use a 0x02, 0x02 pad to learn next byte, etc.

Lesson: Encrypt *then* MAC

You'll exploit in P1!

Authenticated Encryption with Associated Data

Preferred modern approach:

Authenticated encryption with associated data (AEAD)

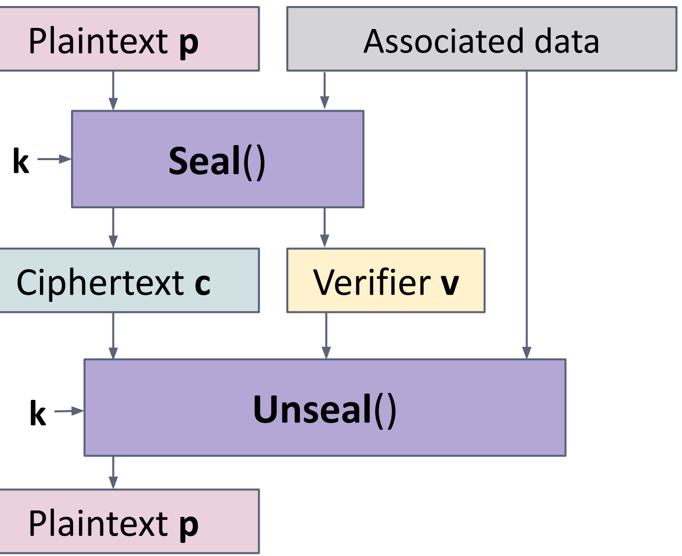
Integrity and encryption in a single primitive:

c, **v** := Seal(**k**, **p**, [**associated_data**])
encrypts plaintext **p** and returns
ciphertext **c** and a verifier **v** (called a "**tag**")

p, **err** := Unseal(**k**, **c**, **v**, [**associated_data**])
returns **p** or an error if **v** does not match the
supplied **c** and **associated_data**

Optional **associated_data** is covered by verifier but *not encrypted*.

Useful for binding data to its context:
e.g., counter, sender ID, etc.



Examples:

AES-GCM ("Galois Counter Mode")
hardware accelerated in recent CPUs

ChaCha20-Poly1305, common on mobile

Galois/Counter Mode (GCM)

Galois/Counter Mode (GCM)

Most widely used AEAD cipher mode

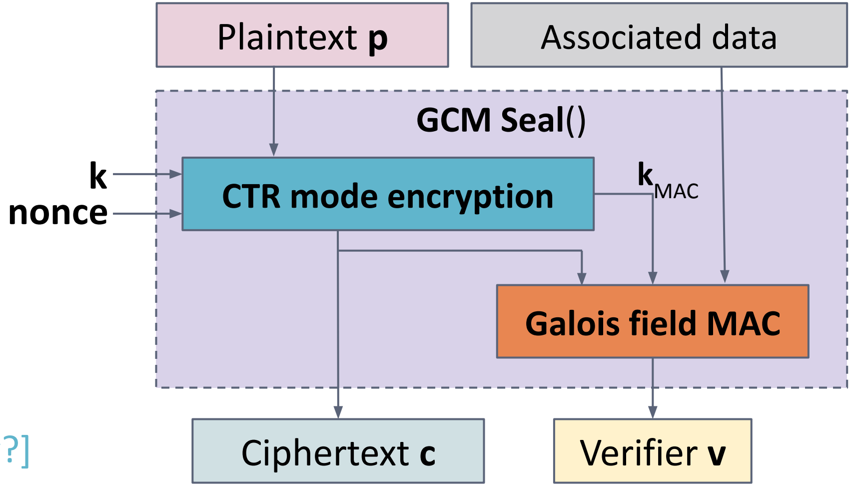
Developed by McGrew and Viega in 2004

Standardized by IETF, NIST, others

Non-generic composition [What kind?] of AES in CTR mode for encryption and a special MAC based on polynomials over finite ("Galois") fields.

Note: GCM violates principle of key separation [How?]

(We can prove it's ok, but it's delicate.)



Warning: Can construct GCM ciphertexts that decrypt (differently, but without error) under many keys

Prof. Grubbs crafted one that decrypts under 131,072 different keys

Warning: GCM nonce reuse is *catastrophic*!

Encrypting two ciphertexts with the same (**k**,**nonce**) leaks the plaintext ***and the MAC key.***

Why? Polynomial root-finding.

Details interesting but beyond our scope in 388.

Parameter Sizes

Issue: How should we set sizes?

Choose |**k**| to resist brute force attacks, even as computers become faster.

For ciphers/PRG keys:

- Want to resist exhaustive search for **k**
- 128 bits considered "classically" safe
  ($2^{128} \approx 10^{38} \approx$ number of silicon atoms in the earth)
- For quantum-resistance, use 256 bits
  (Grover's algorithm gets attacker "sqrt" speedup)

For hash function outputs:

- Want collision resistance (CR)
- Need $2^{\mathbf{n}}$ bits of output for **n** bits of CR, due to "birthday" attacks
  (e.g., SHA-256 has 128 bits of CR)

Estimating what's feasible to compute?

$2^{64} \approx 10^{19}$ $2^{128} \approx 10^{38}$ 1 year $\approx 3 \times 10^{7}$ s

CPU mining: $\approx 10^{8}$ SHA-256/s
GPU mining: $\approx 10^{11}$ SHA-256/s
ASIC mining: $\approx 10^{14}$ SHA-256/s

Bitcoin miners globally: 10^{20} SHA-256 blocks/s

"Birthday" Attack

Generate random values, look for collision

Requires $2^{|\mathbf{k}|/2}$ time, $2^{|\mathbf{k}|/2}$ space
[Puzzle: Do it in constant space?]

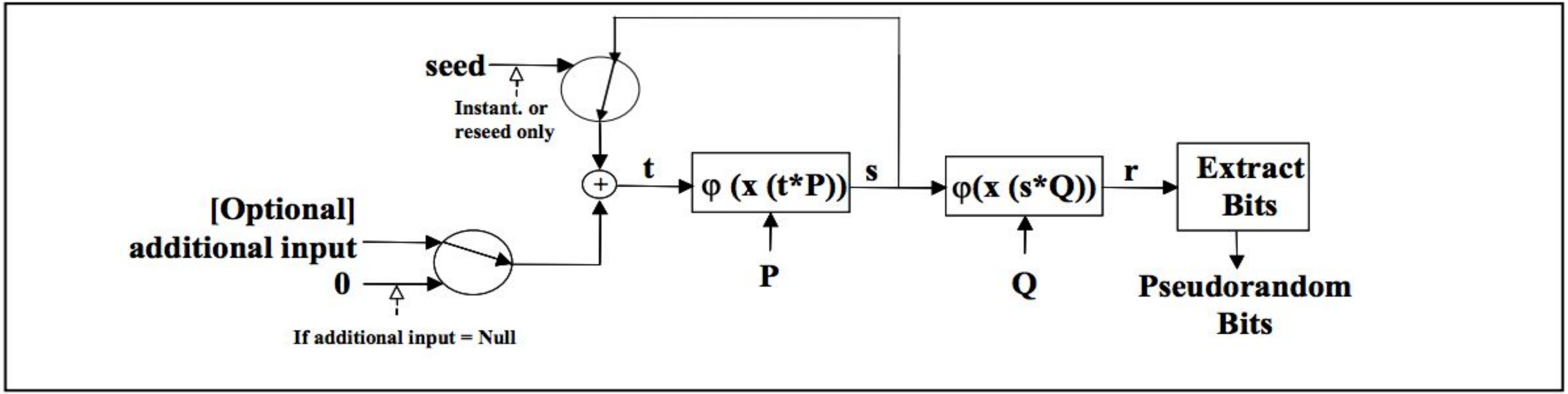
For 128-bit output, takes 2^{64} steps: doable!

Randomness as an Attack Target

Good randomness is needed everywhere in cryptography.
RNG is very good attack target!

Dual-EC DRBG: 2006 NIST standard that NSA (allegedly) **backdoored**. Evidence in Snowden documents.

Construction allows for the existence of a secret backdoor key that can be used to recover the internal RNG state (and determine future output) given knowledge of small amount of past output.



Coming Up

Reminders:

Project 1, Part 1 due Thursday at 6 p.m.
Project 1, Part 2 due 9/21 at 6 p.m.

Quiz on Canvas after every lecture

Thursday

Public Key Cryptography

Diffie-Hellman key exchange,
RSA encryption,
digital signatures

Starting Next Week

Web and Network Security Units