Constructing block ciphers: typically, relies on an "iterated cipher"

Difficult to design! Never invent your own crypto – use well-studied, standardized constructions and implementations!

We will look at two classic designs:

- DES / 3DES (Data Encryption Standard) 1977 (developed at IBM)
- AES (Advanced Encryption Standard) 2002 [most widely used block cipher, implemented in hardware in Intel processors]
  - on modern Intel processors, (with AES-NI), ~4 cycles/round

DES design uses 56-bit keys (and 64-bit blocks)

56-bit keys was a compromise between 40-bit keys (NIST/NSA) and 64-bit keys (cryptographers – notably Hellman)

↳ turned out to be insufficient
  - 1997: DES challenge solved in 96 days (massive distributed effort)
  - 1998: with dedicated hardware, DES can be broken in just 56 hours → not secure enough!
  - 2007: using off-the-shelf FPGAs (120), can break DES in just 12.8 days → anyone can now break DES!

↳ 2-DES: apply DES twice (keys now 112-bits)
  ↳ meet-in-the-middle attack gives no advantage (though space usage is high)

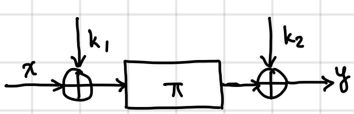
↳ 3-DES: apply DES three times $[\text{3DES}((k_1, k_2, k_3), x) := \text{DES}(k_3, \text{DES}^{-1}(k_2, \text{DES}(k_1, x)))]$
  ↳ 168-bit keys – standardized in 1998 after brute force attacks on DES shown to be feasible

AES (2002 – most common block cipher in use today):
- 3DES is slow (3x slower than DES)
- 64-bit block size not ideal (recall that block size determines adversary's advantage when block cipher used for encryption)

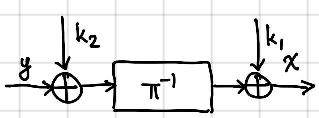
AES block cipher has 128-bit blocks (and 128-bit keys) ← also have 192-bit and 256-bit variants (but block size always 2^{128})

↳ follows another classic design paradigm: iterated Even-Mansour (also called alternating key ciphers)

Even-Mansour block cipher: keys (k_1, k_2), input x:



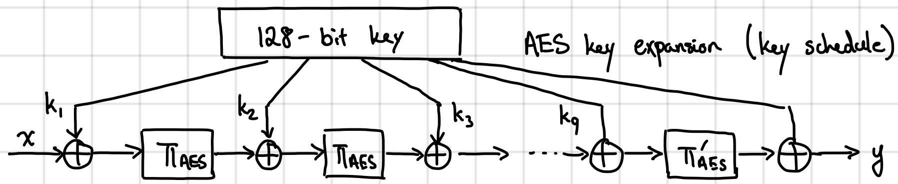
Evaluation



Inversion

Theorem (Even-Mansour): If π is modeled as a random permutation, then the Even-Mansour block cipher is secure (i.e., it is a secure PRP).

The AES block cipher can be viewed as an iterated Even-Mansour cipher:



key-size
- AES-128: 10 rounds
- AES-192: 12 rounds
- AES-256: 14 rounds

(block-size all 128 bits)

Permutations Π_{AES} and Π'_{AES} are fixed permutations and <u>cannot</u> be ideal permutations
↳ cannot write down random permutation over $\{0,1\}^{128}$

↳ Cannot appeal to security of Even-Mansour for security
  ↳ But still provides evidence that this design strategy is <u>viable</u> [similar to DES and Luby-Rackoff]

AES round permutation: composed of three invertible operations that each operate on a 128-bit block

a_0	a_1	a_2	a_3
a_4	a_5	a_6	a_7
a_8	a_9	a_{10}	a_{11}
a_{12}	a_{13}	a_{14}	a_{15}

128 bits arranged in 4-by-4 grid of bytes ($\{0,1\}^8$)

SubBytes: apply a <u>fixed</u> permutation $S: \{0,1\}^8 \to \{0,1\}^8$ to each cell
↳ hard coded in the AES standard (similar to S-box)
(chosen very carefully to resist attacks)

ShiftRows: cyclic shift the rows of the matrix
- 1st row unchanged
- 2nd row shifted left by 1
- 3rd row shifted left by 2
- 4th row shifted left by 3

MixColumns: the matrix is interpreted as a 4-by-4 matrix over $GF(2^8)$ and multiplied by a fixed <u>invertible</u> matrix (also carefully chosen and hard-coded into the standard)

elements are polynomials over $GF(2)$ ($\mathbb{F}_2$) modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$

<u>Observe</u>: Every operation is invertible, so composition is also invertible

Π_{AES} : SubBytes; ShiftRows; MixColumns

Π'_{AES} : SubBytes; ShiftRows No MixColumns for the last round [done so AES decryption circuit better resembles AES encryption]

<u>Security of AES</u>: Brute-force attack: 2^{128}

Best-known key recovery attack: $2^{126.1}$ time — only 4x better than brute force!

What does 2^{128}-time look like?

- Suppose we can try 2^{40} keys a second.
  - ↳ 2^{88} seconds to break 1 AES key $\sim 10^{19}$ years (710 million times larger than age of the universe!)
- Total computing power on Earth (circa 2015)
  - ↳ estimated to be $\sim 2^{70}$ operations/second (currently, bitcoin mining computes $\sim 2^{66}$ hashes/second)

Let's say we can do 2^{80} operations/second

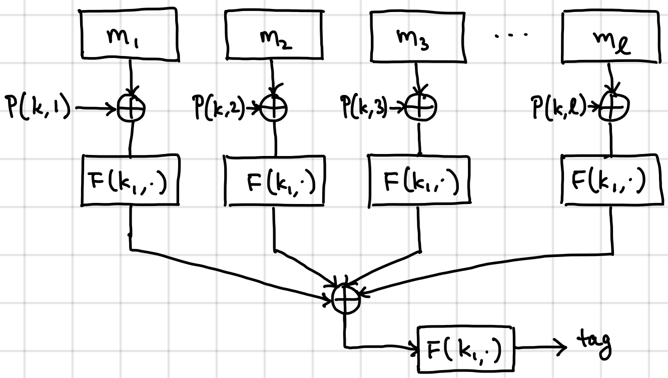
- ↳ still require 2^{48} seconds to break AES ~ 9 million years of compute

If we move to 256-bit keys, best brute force attack takes $2^{254.2}$ time (on AES-256)

In well-implemented systems, the cryptography is not the weak point — breaking the crypto requires new <u>algorithmic</u> techniques (e.g., quantum computers)

↳ But side channels / bad implementations can compromise crypto

A parallelizable MAC (PMAC) — general idea:



$P(k,\cdot)$ are important — otherwise, adversary can permute the blocks
(derived as $F(k_1, 0^n)$ — so key is just k_1)

↳ "mask" term is of the form $\gamma_i \cdot k$ where multiplication is done over $GF(2^n)$ where n is the block size (constants γ_i carefully chosen for efficient evaluation)

Can use similar ideas as CMAC (randomized prefix-free encoding) to support messages that is not constant multiple of block size

Parallel structure of PMAC makes it easily updateable (assuming F is a PRP)

↳ suppose we change block i from $m[i]$ to $m'[i]$:

compute $F^{-1}(k_1, tag) \oplus \underbrace{F(k_1, m[i] \oplus P(k,i))}_{\text{old value}} \oplus \underbrace{F(k_1, m'[i] \oplus P(k,i))}_{\text{new value}}$

} PMAC is "incremental": can make local updates without full recomputation

In terms of performance:

- On sequential machine, PMAC comparable to ECBC, NMAC, CMAC
- On parallel machine, PMAC much better

} Best MAC we've seen so far, but not used...
Reason: patents :([not patented anymore!]

Summary: Many techniques to build a large-domain PRF from a small-domain one (domain extension for PRF)

↳ Each method (ECBC, CMAC, PMAC) gives a MAC on variable-length messages

↳ Many of these designs (or their variants) are standardized

How do we combine confidentiality and integrity?

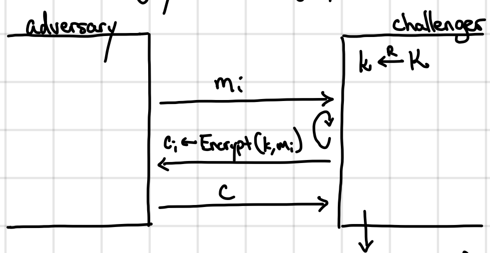
↳ Systems with both guarantees are called authenticated encryption schemes – gold standard for symmetric encryption

Two natural options:

1. Encrypt-then MAC (TLS 1.2+, IPsec) ← guaranteed to be secure if we instantiate using CPA-secure encryption and a secure MAC
2. MAC-then-encrypt (SSL 3.0/TLS 1.0, 802.11i) ← as we will see, not always secure

Definition. An encryption scheme $\Pi_{SE} = (\text{Encrypt}, \text{Decrypt})$ is an authenticated encryption scheme if it satisfies the following two properties:

- CPA security [confidentiality]
- ciphertext integrity [integrity]



output 1 if $c \notin \{c_1, c_2, \ldots\}$ and $\text{Decrypt}(k, c) \neq \bot$ ← special symbol $\bot$ to denote invalid ciphertext

Define $\text{CIAdv}[A, \Pi_{SE}]$ to be the probability that output of above experiment is 1. The scheme Π_{SE} satisfies ciphertext integrity if for all efficient adversaries A,

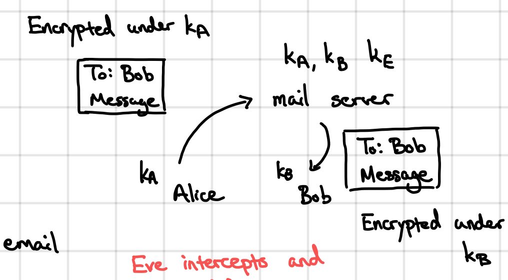
$$\text{CIAdv}[A, \Pi_{SE}] = \text{negl}(\lambda)$$

↑ security parameter determines key length

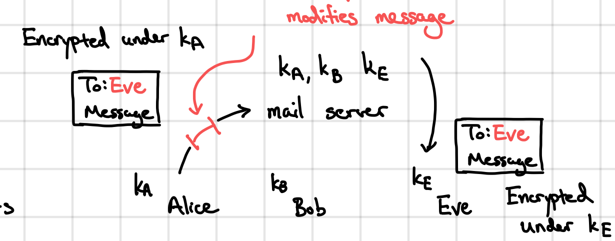
Ciphertext integrity says adversary cannot come up with a new ciphertext: only ciphertexts it can generate are those that are already valid. Why do we want this property?

Consider the following active attack scenario:

- Each user shares a key with a mail server
- To send mail, user encrypts contents and send to mail server
- Mail server decrypts the email, re-encrypts it under recipient's key and delivers email



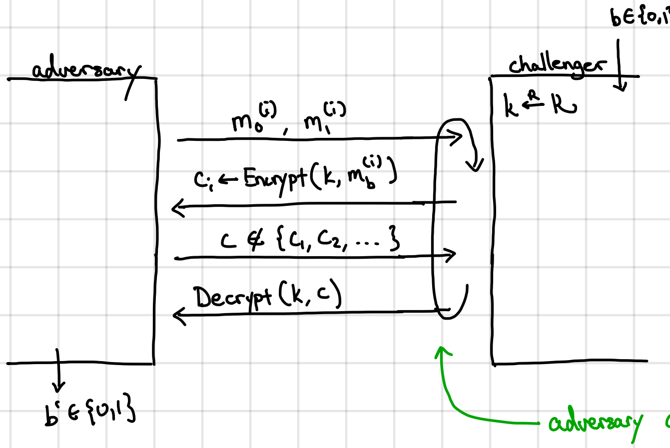
Eve intercepts and modifies message



If Eve is able to tamper with the encrypted message, then she is able to learn the encrypted contents (even if the scheme is CPA-secure)

↳ More broadly, an adversary can tamper and inject ciphertexts into a system and observe the user's behavior to learn information about the decrypted values – against active attackers, we need stronger notion of security

Definition. An encryption scheme $\Pi_{SE} = (\text{Encrypt}, \text{Decrypt})$ is secure against chosen-ciphertext attacks (CCA-secure) if for all efficient adversaries $\mathcal{A}$, $\text{CCAAdv}[\mathcal{A}, \Pi_{SE}] = \text{negl.}$ where we define $\text{CCAAdv}[\mathcal{A}, \Pi_{SE}]$ as follows:



adversary can make arbitrary encryption and decryption queries, but cannot decrypt any ciphertexts it received from the challenger (otherwise, adversary can trivially break security)
↳ called an "admissibility" criterion

$$\text{CCAAdv}[\mathcal{A}, \Pi_{SE}] = \left| \Pr[b' = 1 \mid b = 0] - \Pr[b' = 1 \mid b = 1] \right|$$

CCA-security captures above attack scenario where adversary can tamper with ciphertexts
- ↳ Rules out possibility of transforming encryption of $x \| z$ to encryption of $y \| z$
- ↳ Necessary for security against <u>active</u> adversaries [CPA-security is for security against <u>passive</u> adversaries]
- ↳ We will see an example of a real CCA attack in HW1

Theorem. If an encryption scheme Π_{SE} provide authenticated encryption, then it is CCA-secure.

Proof (Idea). Consider an adversary $\mathcal{A}$ in the CCA-security game. Since Π_{SE} provides ciphertext integrity, the challenger's response to the adversary's decryption query will be $\perp$ with all but negligible probability. This means we can implement the decryption oracle with the "output $\perp$" function. But then this is equivalent to the CPA-security game.
[Formalize using a hybrid argument]

Note: Converse of the above is not true since CCA-security $\not\Rightarrow$ ciphertext integrity.
- ↳ However, CCA-security + plaintext integrity $\Rightarrow$ authenticated encryption

simple counter-example: concatenate unused bits to end of ciphertext in a CCA-secure scheme (stripped away during decryption)

Take-away: Authenticated encryption captures meaningful confidentiality + integrity properties; provides <u>active</u> security

Encrypt-then-MAC: Let (Encrypt, Verify) be a CPA-secure encryption scheme and (Sign, Verify) be a secure MAC. We define Encrypt-then-MAC to be the following scheme:

$\text{Encrypt}'((k_E, k_M), m)$: $c \leftarrow \text{Encrypt}(k_E, m)$
$t \leftarrow \text{Sign}(k_M, c)$
output (c, t)

(k_E, k_M: independent keys)

$\text{Decrypt}'((k_E, k_M), (c, t))$: if $\text{Verify}(k_M, c, t) = 0$, output $\perp$
else, output $\text{Decrypt}(k_E, c)$

Theorem. If (Encrypt, Decrypt) is CPA-secure and (Sign, Verify) is a secure MAC, then (Encrypt', Verify') is an authenticated encryption scheme

Proof. (Sketch). CPA-security follows by CPA-security of (Encrypt, Decrypt). Specifically, the MAC is computed on ciphertexts and not the messages. MAC key is independent of encryption key so cannot compromise CPA-security.

Ciphertext integrity follows directly from MAC security (i.e., any valid ciphertext must contain a new tag on some ciphertext that was not given to the adversary by the challenger)

Important notes:
- Encryption + MAC keys must be independent. Above proof required this (in the formal reduction, need to be able to simulate ciphertexts / MACs — only possible if reduction can chose its own key).
  - ↳ Can also give explicit constructions that are completely broken if same key is used (i.e., both properties fail to hold)
  - ↳ In general, never reuse cryptographic keys in different schemes; instead, sample fresh, independent keys!
- MAC needs to be computed over the entire ciphertext
  - Early version of ISO 19772 for AE did not MAC IV (CBC used for CPA-secure encryption)
  - RNCryptor in Apple iOS (for data encryption) also problematic (HMAC not applied to encryption IV)

  ] means first block (i.e., "header") is malleable

MAC-then-Encrypt: Let (Encrypt, Verify) be a CPA-secure encryption scheme and (Sign, Verify) be a secure MAC. We define MAC-then-Encrypt to be the following scheme:

Encrypt'$((k_E, k_M), m)$: $t \leftarrow \text{Sign}(k_M, m)$
$c \leftarrow \text{Encrypt}(k_E, (m, t))$
output c

Decrypt'$((k_E, k_M), (c, t))$: compute $(m, t) \leftarrow \text{Decrypt}(k_E, c)$
if $\text{Verify}(k_M, m, t) = 1$, output m, else, output $\perp$

Not generally secure! SSL 3.0 (precursor to TLS) used randomized CBC + secure MAC
↳ Simple CCA attack on scheme (by exploiting padding in CBC encryption)
[POODLE attack on SSL 3.0 can decrypt all encrypted traffic using a CCA attack]

Padding is a common source of problems with MAC-then-Encrypt systems [see HW2 for an example]

In the past, libraries provided separate encryption + MAC interfaces — common source of errors
↳ Good library design for crypto should minimize ways for users to make errors, not provide more flexibility

Today, there are standard block cipher modes of operation that provide authenticated encryption
- One of the most widely used is GCM (Galois counter mode) — standardized by NIST in 2007

GCM mode: follows encrypt-then-MAC paradigm
- CPA-secure encryption is nonce-based counter mode
- MAC is a Carter-Wegman MAC
  - ↳ "encrypted one-time MAC"

} Most commonly used in conjuction with AES (AES-GCM provides authenticated encryption)

GCM encryption: encrypt message with AES in counter mode

compute Carter-Wegman MAC on resulting message using GHASH as the underlying hash function

and the block cipher as underlying PRF

(GHASH ← Galois Hash; hash function ← key derived from PRF evaluation at 0^n)

↳ GHASH operates on blocks of 128-bits

operations can be expressed as operations over $GF(2^{128})$ — Galois field with 2^{128} elements

implemented in hardware — very fast!

Typically, use AES-GCM for authenticated encryption

Oftentimes, only part of the payload needs to be hidden, but still needs to be authenticated

↳ e.g., sending packets over a network: desire confidentiality for packet body, but only integrity for packet headers (otherwise, cannot route!)

AEAD: authenticated encryption with associated data

↳ augment encryption scheme with additional plaintext input; resulting ciphertext ensures integrity for associated data, but not confidentiality (will not define formally here but follows straightforwardly from AE definitions)

↳ can construct directly via "encrypt-then-MAC": namely, encrypt payload and MAC the ciphertext + associated data

↳ AES-GCM is an AEAD scheme