Modern Symmetric-Key Encryption:

Block Ciphers
Block Cypher

• A function \( E : \{0, 1\}^b \times \{0, 1\}^k \rightarrow \{0, 1\}^b \)

• Once we fix the key \( K \) (of size \( k \) bits), we get:
  – \( E_K : \{0,1\}^b \rightarrow \{0,1\}^b \) denoted by \( E_K(M) = E(M,K) \)
  – And also \( D(C,K) \), \( E(M,K) \)'s inverse

• Three properties:
  – **Correctness:**
    • \( E_K(M) \) is a permutation (bijective function) on \( b \)-bit strings
    • Bijective \( \Rightarrow \) invertible
  – **Efficiency:**
    • Computable in micro seconds
  – **Security:**
    • For unknown \( K \), behaves like a random permutation

• **Provides a building block for more extensive encryption**
Data Encryption Standard (DES)

• Developed in early 1970s by IBM
  – Block size 64 bits, key size 56 bits
  – NSA influenced two facets of its design:
    • Altered some subtle internal workings in “mysterious ways”
    • Reduced proposed key size 64 bits ⇒ 56 bits
      – Made brute-force attacks feasible for any attacker with massive (for the time) computational resources (can’t imagine why the NSA would want to do that 😃)

• Remains essentially unbroken 45 years later
  – The NSA’s tweaking hardened it against an attack “invented” a decade later
    • Self study: Find details on this

• However, modern computer speeds make DES completely unsafe due to the small key size
Advanced Encryption Standard (AES)

- Today’s “go-to” encryption standard
- 20 years old
- Block size 128 bits
- Key can be 128, 192, or 256 bits
  - 128 remains quite safe; sometimes termed “AES-128”
- As usual, includes encryptor and (closely-related) decryptor
- **How it works is beyond scope of this class**
- **NOT** proven secure:
  - But no known flaws
  - So we assume it is a secure block cipher
How strong is a 128-bit Key?

- $2^{128}$ possibilities
- Handy approximation: $2^{10} \approx 10^3$
- $2^{128} = 2^{10 \times 12.8} \approx (10^3)^{12.8} \approx (10^3)^{13} \approx 10^{39}$
- Say we happen to have lying around some massive hardware cluster that can try 109 keys every nanosecond...
  - So $10^{18}$ keys/sec
  - Thus, we’ll need $\approx 10^{21}$ sec

- How long is that?
  - One year $\approx 3 \times 10^7$ sec
  - So need $\approx 3 \times 10^{13}$ years
  - Or **30 trillion years!!**
Issues With Block Ciphers

• Block ciphers can only encrypt messages of a certain size
  – If $M$ is smaller, easy, just pad it (details omitted, beyond scope of class)
  – If $M$ is larger, repeatedly apply block cipher
    • Specific method known as a the “block cipher mode”
    • Tricky to get this right!

• If same data is encrypted twice, attacker knows it is the same
  – Solution: incorporate a varying, known quantity
    • The IV = “initialization vector”
Electronic Code Book (ECB)

- Simplest **block cipher mode**
- Split message into \( b \)-bit blocks \( P_1, P_2, \ldots \)
- Each block is enciphered independently
  \[
  C_i = E(P_i, K)
  \]
- Since key \( K \) is fixed, each block is subject to the same permutation
  - As if we had a “code book” to map each possible input value to its designated output
ECB Mode Encryption
ECB Mode Decryption

Problem? Relationships between $P_i$’s reflected in $C_i$’s
Encrypted Penguins: ECB Mode

Original image, RGB values split into many b-bit blocks

Encrypted with ECB and interpreting ciphertext directly as RGB

Later identical message, also encrypted with ECB
Building a Better Block Cipher Mode

• Ensure blocks incorporate more than just the plaintext to mask relationships between blocks

• Done carefully, either of these works:
  – Include elements of prior computation
  – Include positional information

• Plus... need some initial randomness
  – Prevent encryption scheme determinism
    • i.e. potential revealing relationships between messages
  – Introduce an initialization vector (IV) (a varying, but known quantity)

• Example:
  – Cipher Block Chaining (CBC)
  – Takes output from current computation as an input to next
Cipher Block Chaining : Encryption

• **E(plaintext, K):**
  – Split plaintext into blocks of size $b$ where $b$ is the block cipher block size
  – Choose a random IV (do not reuse for other messages)
  – Now compute:

  ![Cipher Block Chaining Diagram]

  - $C_1$ (Ciphertext)
  - $C_2$ (Ciphertext)
  - $C_3$ (Ciphertext)

• Final ciphertext (as seen by Eve) is $(IV, C_1, C_2, C_3)$
Cipher Block Chaining: Decryption

- **D(ciphertext, K):**
  - Take IV out of ciphertext
  - Split ciphertext into blocks of size $b$ where $b$ is the block cipher block size
  - Now compute:

  \[
  C_1 \rightarrow \text{Ciphertext} \rightarrow \text{block cipher decryption} \rightarrow \text{Initialization Vector (IV)} \rightarrow \oplus \rightarrow P_1 \rightarrow \text{Plaintext}
  \]

  \[
  C_2 \rightarrow \text{Ciphertext} \rightarrow \text{block cipher decryption} \rightarrow P_2 \rightarrow \text{Plaintext}
  \]

  \[
  C_3 \rightarrow \text{Ciphertext} \rightarrow \text{block cipher decryption} \rightarrow P_3 \rightarrow \text{Plaintext}
  \]

- Final plaintext (as seen by Bob) is $P_1, P_2, P_3$
Encrypted Penguins : CBC Mode

Original image, RGB values split into many b-bit blocks

Encrypted with CBC and interpreting ciphertext directly as RGB
Cipher Block Chaining: Summary

• Widely used

• Issue: **sequential encryption**, hard to parallelize

• Parallelizable alternative:
  – **Counter Mode** (CTR Mode)

• Security:
  – If no reuse of nonce (see next slide), CBC and CTR are provably secure
    • Assuming underlying block cipher is secure
Counter Mode: Encryption

Nonce is equivalent to IV

Important that nonce/IV does not repeat across multiple encryptions
=> choose nonce at random!
Counter Mode: Decryption

CTR decryption uses block cipher’s encryption, not decryption!
Pseudo Random Number Generators (PRNGs)

• Given a seed, outputs sequence of seemingly random bits - and keeps internal state
  – \text{PRNG}(\text{seed}) \Rightarrow \text{“random” bits}

• Can output arbitrarily many random bits

• \textbf{Can a PRNG be truly random?}
  – No
  – For seed length \(s\), it can generate at most \(2^s\) distinct possible sequences

• \textbf{Can a PRNG be sufficiently random?}
  – Yes
  – A cryptographically strong PRNG \textit{appears} truly random to an attacker
  – Attacker cannot distinguish it from a genuinely random sequence
Building Stream Ciphers

• **Encryption**, given key $K$ and message $M$:
  – Choose a random value $IV$
  – $E(M, K) = PRNG(K, IV) \oplus M$

• **Decryption**, given key $K$, ciphertext $C$, and initialization vector $IV$:
  – $D(C, K) = PRNG(K, IV) \oplus C$

• Can encrypt message of any length because $PRNG$ can produce any number of random bits
Using a PRNG to Build a Stream Cipher

(Small) K, IV

PRNG

Keystream

M_i: i^{th} message of plaintext

IV, C_i

M_i

UCSC BSOE CMPS122 Winter 2018. Copyright © 2018 David C. Harrison. All rights reserved.
Building a Cryptographically Strong PRNG

- A simple design for a PRNG that generates 128-bit numbers
  - Only state needed is SEED and N (No. of calls so far)
- $\text{PRNG(SEED)} = \{\text{return AES-128 SEED(}++\text{N)} \}$
  - i.e., encrypt counter of No. of calls using SEED as key
  - AES-128 acts as a random permutation of 128-bit bit-strings, so even a tiny input change such as $N$ vs. $N+1$ completely and unpredictably changes output
- A version that incorporates an IV
  - Only state needed is SEED and N (No. of calls so far), plus an IV
- $\text{PRNG(SEED, IV)} = \{\text{return AES-128 SEED(}++\text{N} \oplus \text{IV)} \}$
  - i.e., encrypt (counter of No. of calls, XOR’d with IV) using SEED as key
- Let’s look at using this PRNG to build a stream cipher with the block cipher “CTR” (counter) mode
Using a PRNG to Build a Stream Cipher

IV ⊕ (++n) → AES-128_K

Keystream

M_i: i^{th} message of plaintext

IV, C_i → IV ⊕ (++n) → AES-128_K

Keystream

Bob

Alice
Counter Mode: AES-128 Encryption

Nonce is equivalent to IV

- Only difference from our stream cipher built on AES-128 is use of a different operator (concatenation vs. XOR) to combine IV and counter
- Both are equally secure as long as IV is random