Introduction to Cryptography

Alan Cao

Disclaimers

- Cryptography is rooted in both theoretical (math) and technical components.
  - **I AM NOT A MATH MAJOR, NOR AN EXPERT CRYPTOGRAPHER!!**

- What this talk **WILL**:
  - Help you make the best choices to make in selecting cryptographic primitives
  - Give you a whirlwind tour of the internals that make a cryptographic primitive work
  - Make you comfortable working with cryptographic software

Disclaimers

- Cryptography is rooted in both theoretical (math) and technical components.
    - **I AM NOT A MATH MAJOR, NOR AN EXPERT CRYPTOGRAPHER!!**

- What this talk **WILL**:
    - Help you make the best choices to make in selecting cryptographic primitives
    - Give you a whirlwind tour of the internals that make a cryptographic primitive work
    - Make you comfortable working with cryptographic software
- What this talk **will NOT**:
    - Teach you how to roll your own crypto (please don’t do this)
    - Cover post-quantum cryptography
    - Give in-depth looks into modern-day cryptography schemes and protocols

Cryptography Primer

- Cryptography is the study of protocols that can help effectively transmit information
  - Encryption and decryption occur with a **cipher**, which takes **plaintext** and turns it into **ciphertext** using a **secret key/password**.
  - We want to **take advantage** of the invertibility of certain mathematical properties to construct cryptographic schemes

Crypto and Security

- Our goal in secure modern cryptography: w*hat do we want to guarantee the users of cryptographic software?*

    - Kerckhoffs's Principle
        - A cryptographic scheme should rely on the *secrecy of the key,* rather than the secrecy of the cipher
        - If an attacker understands the all the intricacies of a cipher, he/she should still not be able to get key *K* if it is kept secret.

Crypto and Security

- Our goal in secure modern cryptography: w*hat do we want to guarantee the users of cryptographic software?*

    - Kerckhoffs's Principle
        - A cryptographic scheme should rely on the *secrecy of the key,* rather than the secrecy of the cipher
        - If an attacker understands the all the intricacies of a cipher, he/she should still not be able to get key *K* if it is kept secret.

    - IND-CPA (semantic security)
        - Ciphertext should not leak information about plaintext if key is kept secure
        - Randomness is an important factor in adhering to IND-CPA

Objectives

Building Blocks

- OTPs
- RNGs
- Hashing

Symmetric Cryptography

Block Ciphers

Stream Ciphers

Authenticated Encryption

Asymmetric Cryptography

RSA

Elliptic Curves

Diffie-Hellman

SSL/TLS

...and more!

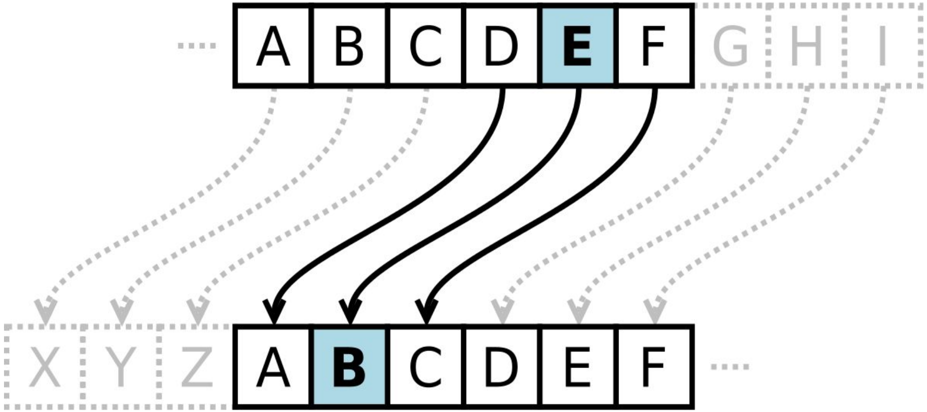
Classical Cryptography

- Substitution Ciphers
  - *Bijective map* from each character to another
- Classic example: **Caesar's Cipher** (monoalphabetic)
- Later implementation: **Vigenere Cipher** (polyalphabetic)

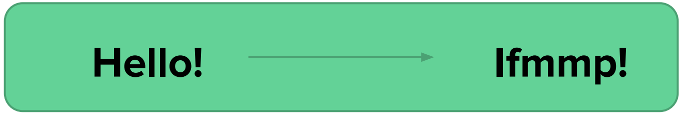
- Efficient/sufficient when transmission of information wasn't digital
  - *We'll see why these all fail now, and how we've adapted from problems we've seen in them.*

Classical Crypto - Caesar's Cipher

- Simple family of shift ciphers
  - Each character maps to the character some number of positions down the alphabet
  - Key = index used for shifting
- "Modern" implementation
  - ROT13
- Bruteforce is relatively easy!
  - Search space: 2^88 permutations
  - Demonstration



Caesar Cipher (fixed shift of 3)



Classical Crypto - Vigenère Cipher

- Slightly more complex, since it is **polyalphabetic**
  - Instead of a fixed shift value, a string is used instead
- More difficult to break, but still possible with **frequency analysis**

Vigenère Cipher (with key "LEMON")

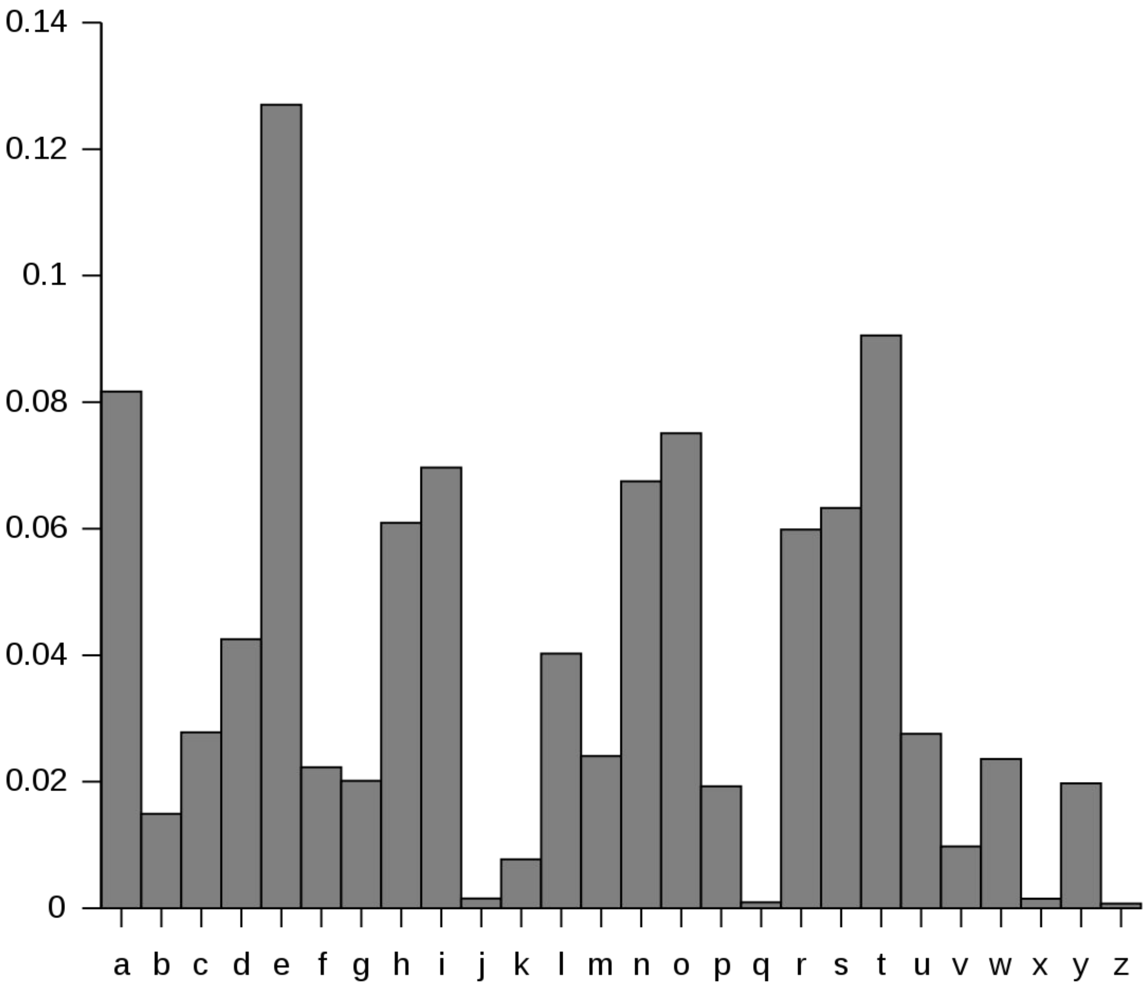
Plaintext	A	T	T	A	C	K	A	T	D	A	W	N
Key	L	E	M	O	N	L	E	M	O	N	L	E
Shift	+11	+4	+12	+14	+13	+11	+4	+12	+14	+13	+11	+4
Ciphertext	L	X	F	O	P	V	E	F	R	N	H	R

Vigenère square

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Frequency Analysis

- We know that some letters are used more than others. For example:
  - E is most commonly used
  - J and Z are almost never used
- For a large enough ciphertext, the **frequency distribution** should roughly match the general English distribution
- Extensions:
  - N-gram / bigram analysis - use pairs of letters rather than just one individual


0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
a b c d e f g h i j k l m n o p q r s t u v w x y z

Problems with Classical Cryptography

- Classical ciphers **FAIL** because
  - They don't bode well against modern-day computational power
  - Don't align with Kerckhoff's Principle!
- Two techniques:
  - Bruteforce is possible with very simple programs
  - Frequency analysis can be used in order to identify common patterns within large ciphertexts

Building Block: XOR

- Let's try solving this problem with the **eXclusive OR (XOR)** operation!
- Output is 1 if A or B is 1, but not if both are
    - Therefore: $A \oplus A = 0$
- Properties:
    - $A \oplus B \oplus A = B$
    - $(A \oplus B) \oplus C = A \oplus (B \oplus C)$

INPUT		OUTPUT
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XOR Truth Table

Building Block: XOR and OTPs

- XOR Ciphers can be used as **One-Time Pads (OTPs)**
- One Time Pads introduce *perfect secrecy*
  - Impossible to learn anything about plaintext from ciphertext, even with immense computational power, other than the length.

<u>Encryption</u>: $C = P \oplus K$

<u>Decryption</u>: $P = C \oplus K$

Where a ciphertext C is produced by XORing plaintext P and random key K.

Building Blocks: OTPs

- Why are they secure? How do they uphold IND-CPA?
  - If K is *guaranteed* to be **random**, then the resultant C should also appear to be **random**
    - Therefore, the XOR cipher used **should not be** single-byte XOR. since it can be brute-forced and/or analyzed for character frequency!
      - (Demonstration)

Building Blocks: OTPs

- Why are they secure? How do they uphold IND-CPA?
  - If K is *guaranteed* to be **random**, then the resultant C should also appear to be **random**
    - Therefore, the XOR cipher used **should not be** single-byte XOR. since it can be brute-forced and/or analyzed for character frequency!
- Caveat:
  - One-time Pads should only be used once! If two ciphertexts are received by an attacker, the following can be done:

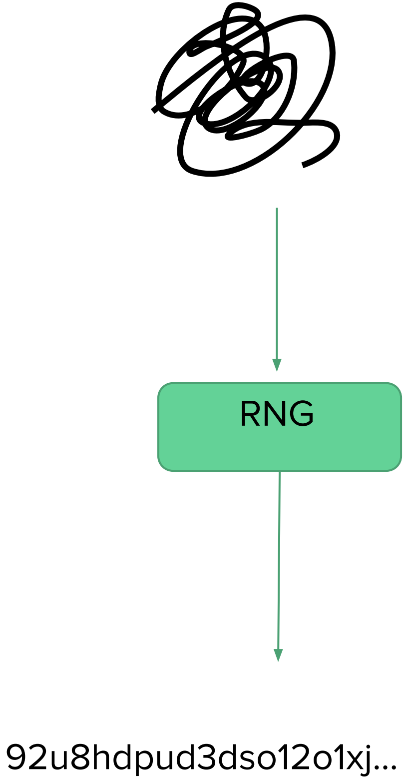
$$\mathbf{C1 \oplus C2} = (P1 \oplus K) \oplus (P2 \oplus K) = (P1 \oplus P2) \oplus (K \oplus K) = \mathbf{(P1 \oplus P2)}$$

Since K ⊕ K = 00000.

Building Block: Randomness

- Many secure ciphers utilize good sources of **randomness.**
- Getting random bits from a reliable source is an important problem!
  - Solution? Use *sources of entropy*

- **Random Number Generators (RNGs)**
  - General “umbrella term”
  - Harness sources of entropy to generate reliable random bits of information



Building Block: Randomness

- What if the RNG itself is not always the most reliable?
  - ie what if the mouse / keyboard is not in use, but randomness is needed somewhere?
- **Pseudorandom Number Generators (PRNGs)**
  - Provide artificial bits from RNGs if analog source becomes unreliable
  - `/dev/urandom`

Building Block: Randomness

- What if the RNG itself is not always the most reliable?
  - ie what if the mouse / keyboard is not in use, but randomness is needed somewhere?
- **Pseudorandom Number Generators (PRNGs)**
  - Provide artificial bits from RNGs if analog source becomes unreliable
  - Seeds from *entropy pools* stored with bits from RNGs, and uses a DRBG (deterministic random bit generator) to expand the bits
    - DRBGs only help expand the bits, b/c same input == same output

Building Block: Randomness

- What do we ultimately want for cryptography?
    - Non-cryptographic PRNGs, like the *Mersenne Twister* may be predictable!
- **Cryptographically secure pseudorandom number generator (CSRNGs)**
    - “Crypto-reliable” PRNGs, meaning they are nearly unpredictable
    - Reliable source(s) of randomness available
    - Cryptographic algorithm to produce reliable bit
    - Should preserve both *backward* and *forward* secrecy
- Recommendations?
    - Stick with `/dev/urandom` or interfaces that rely on it!

Building Block: Hashes

- “One-way functions”
  - Protect data *integrity* rather than guaranteeing *confidentiality*
  - Input given to hash function results in a random *hash*
  - Useful for
    - Validating files (MD5 checksums for integrity)
    - Persisting login credentials without storing cleartext
    - Digital signatures

Building Block: Hashes

- **Secure** hash functions are
    - Avalanche Property - can’t predict changes in hash from subtle changes in input
    - Preimage Resistant - encrypted output **cannot** be reverted back to original input!
    - Collision Resistant - highly unlikely to find hash collisions derived from different inputs

AAAAAAAAA

↓

8430894cfeb54a3
625f18fe24fce272
e

AAAACAAAA

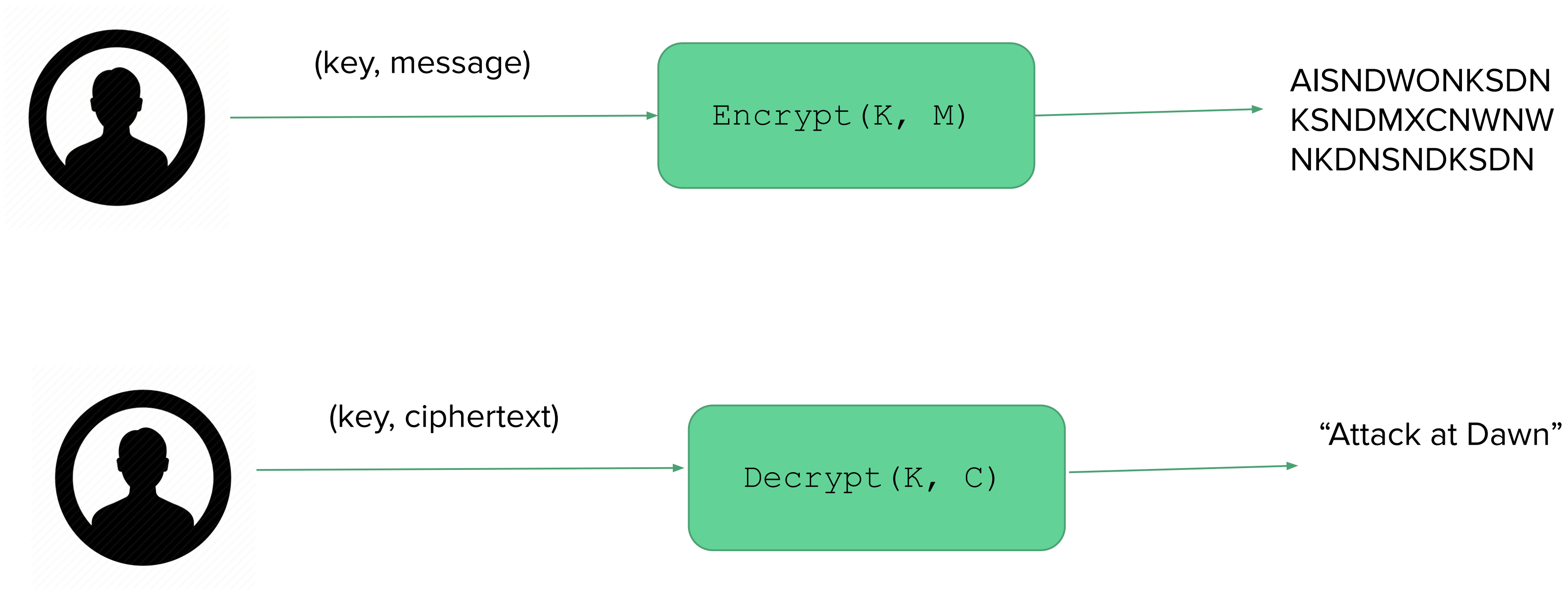
↓

d1d4a244fa5b3b085
662d591e50d5908

Building Block: Hashes

- Hash functions to use/not use:
  - Merkle-Damgard Constructions (MD*)
    - Very performant, but meaning attackers can bruteforce quickly
  - SHA family of hashing functions
    - SHA 1 is not collision-resistant
    - SHA 2 is vulnerable to length extension attacks
    - **SHA-3 / Keccak**
  - **BLAKE2 hash function**
    - Performance + security balanced
    - Modern day standard!!!

Symmetric Cryptography

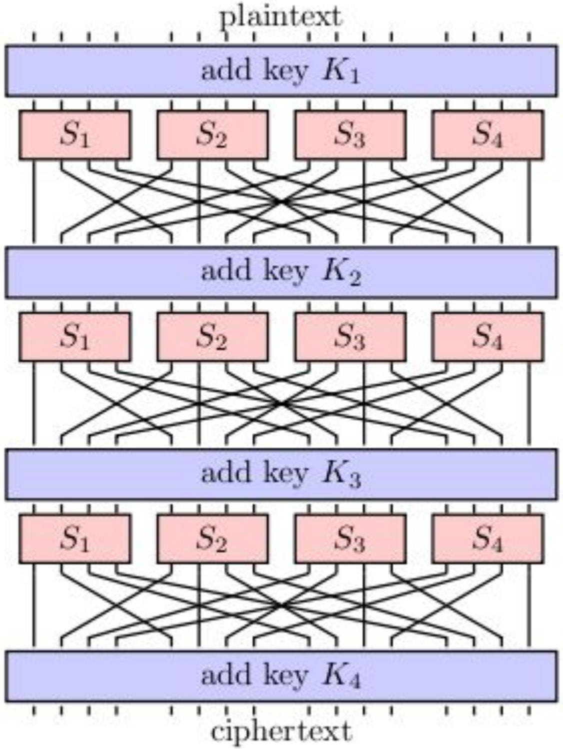


Symmetric Crypto - Block Ciphers

- XOR cipher was an example of a simple block cipher!
- **Block ciphers**
    - Encryption (E(K, P)) and decryption (D(K,C)) algorithm
    - *Pseudorandom permutations* - if key is secure, attacker should never be able to compute output from input (IND-CPA!!)

Symmetric Crypto - Block Ciphers

- How do they work?
  - Input is chunked into sizable blocks of 16, 24, or 32 bytes
  - Blocks are passed into several *rounds* of computation to transform the data
    - `C = R3(R2(R1(P)))`
  - **Substitution-Permutation Networks (S-Boxes)** are lookup tables that help mutate small chunks of data in some way during each round

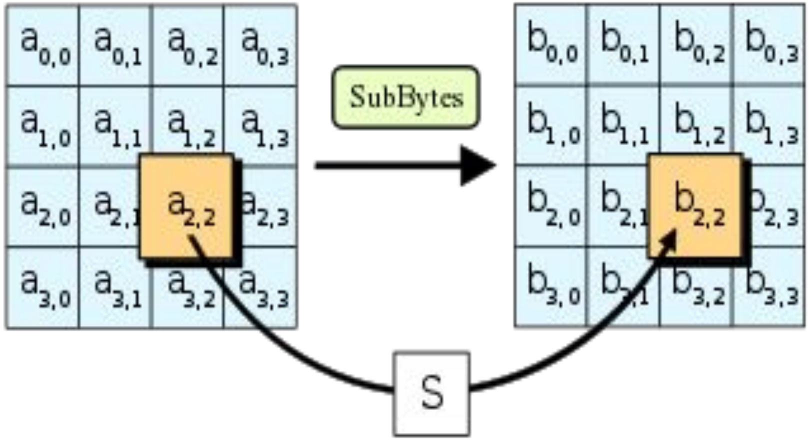


Symmetric Crypto - Block Ciphers

- Does not account for secure key-agreement between parties!
- Does not account for message integrity!
- Other common symmetric crypto schemes:
  - DES / 3DES
  - Blowfish
  - **AES**

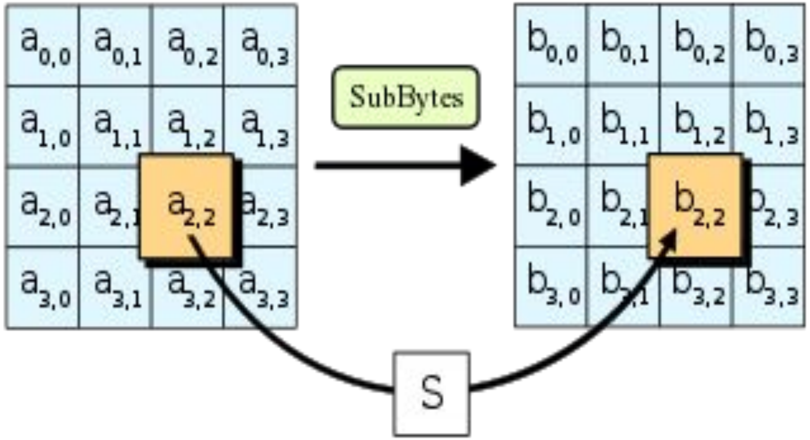
AES - Advanced Encryption Standard (Rijndael)

- Current modern standard (succeeded the DES / 3DES design)
- Processes blocks of **128** bits with a key of **128, 192, or 256** bits
    - Utilizes an internal 4x4 matrix array of 16 bytes



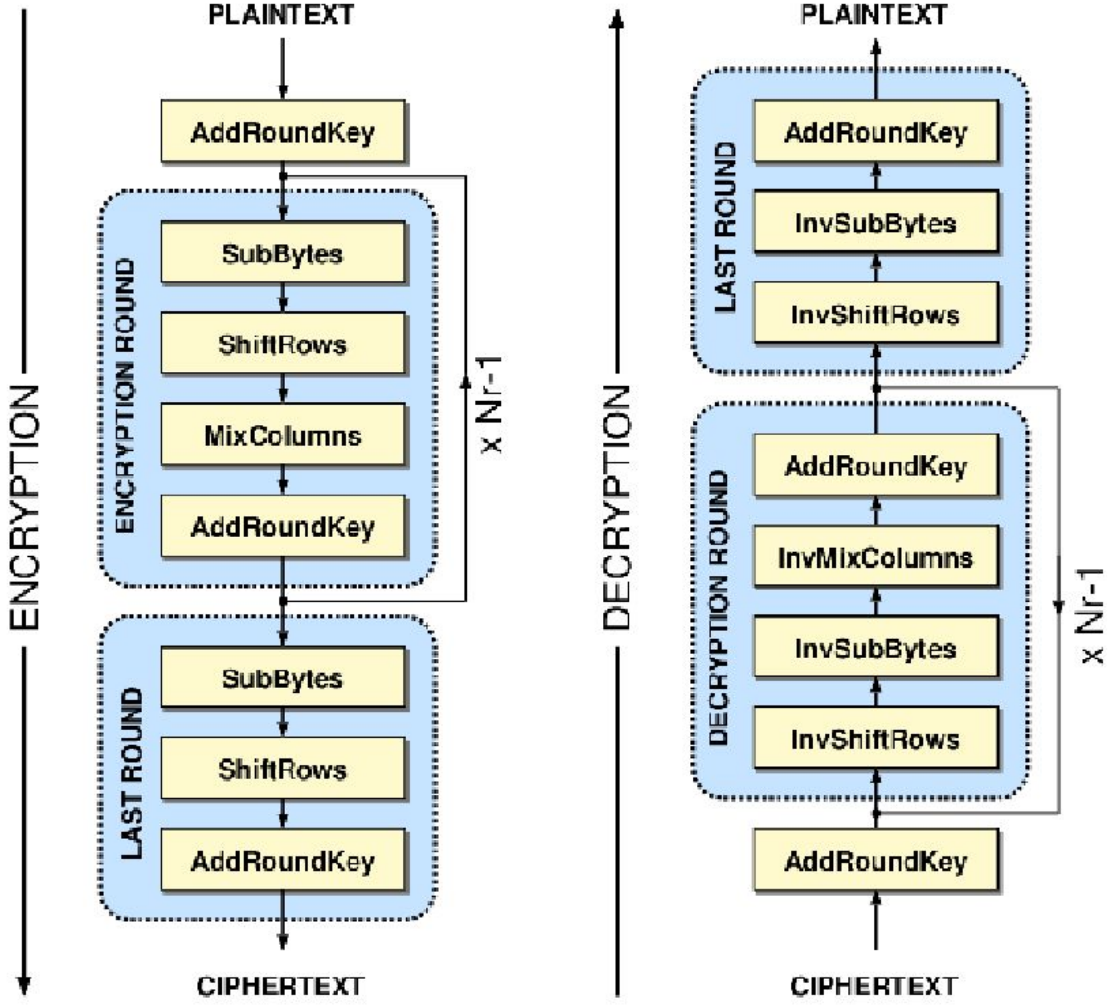
AES - Advanced Encryption Standard (Rijndael)

- Current modern standard (succeeded the DES / 3DES design)
- Processes blocks of **128** bits with a key of **128, 192, or 256** bits
  - Utilizes an internal 4x4 matrix array of 16 bytes



- Inputs of arbitrary sizes are *padded* using the **PKCS#7 standard**
  - `| DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD` **`01`** `|`
  - `| DD DD DD DD DD DD DD DD DD DD DD DD DD DD` **`02 02`** `|`
  - `| DD` **`0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F 0F`** `|`

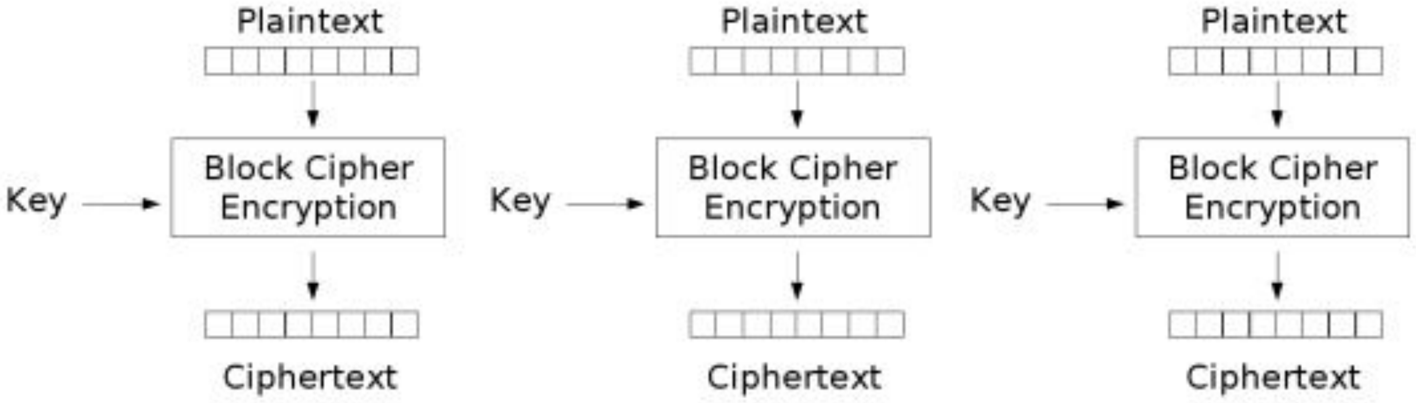
AES Internal Rounds



AES Block Cipher Modes

- Block ciphers harness different *modes of operations* that provide different security guarantees
  - **ECB** - Electronic Codebook Mode
  - **CBC** - Cipher Block Chaining Mode
  - CTR - Counter Mode *(stream cipher)*
  - GCM - Galois Counter Mode *(authenticated encryption)*

Modes of Operation: Electronic Codebook (ECB)

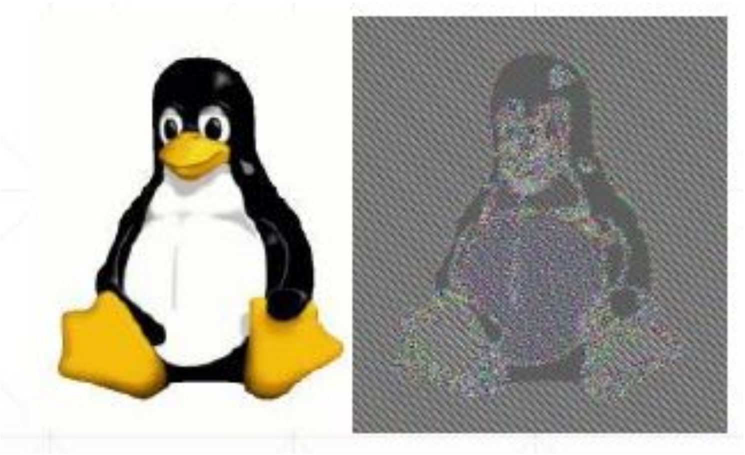


Electronic Codebook (ECB) mode encryption

ECB Mode

- Plaintext blocks are processed independently:
    - $C1 = E(K, P1)$, $C2 = E(K, P2)$ ...

- **KNOWN TO BE SEMANTICALLY INSECURE!!!**
    - Same input blocks in a plaintext result in same output ciphertext blocks
    - Chosen plaintext attacks

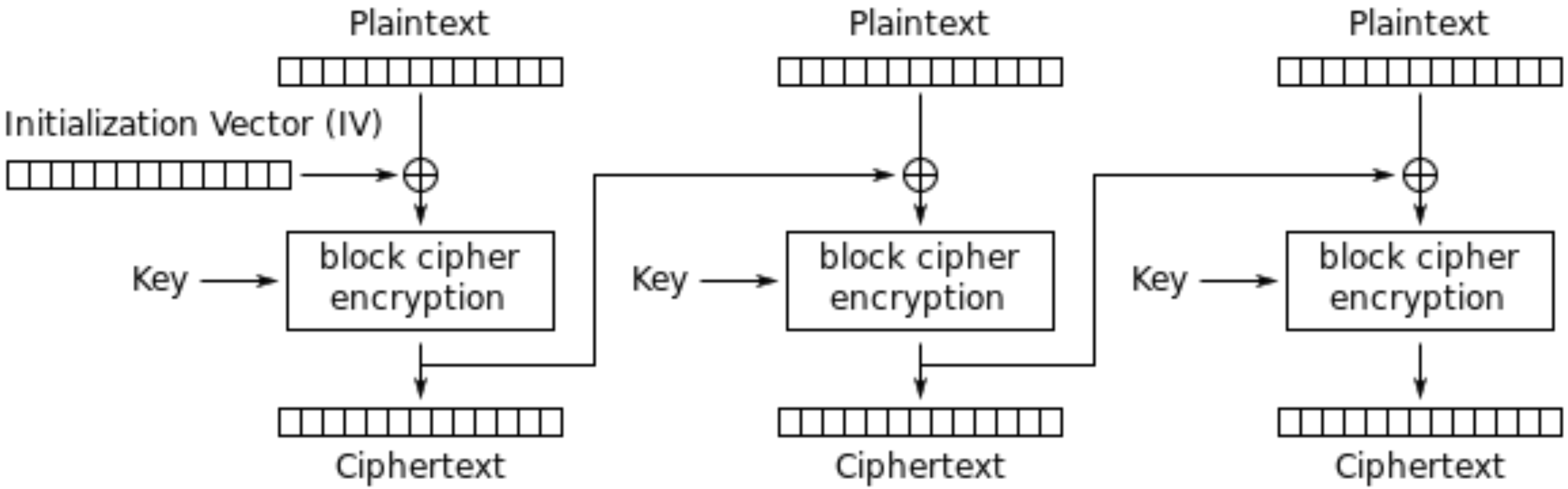


The Infamous ECB Penguin

ECB Mode - Chosen Plaintext Attack

- **Chosen plaintext attack**
  - Observing patterns in ciphertexts can help recover plaintext
  - Works if we have control over some part of the input being encrypted, with sensitive information also residing within some block
  - Demonstration!

Modes of Operation: Cipher Block Chaining



Cipher Block Chaining (CBC) mode encryption

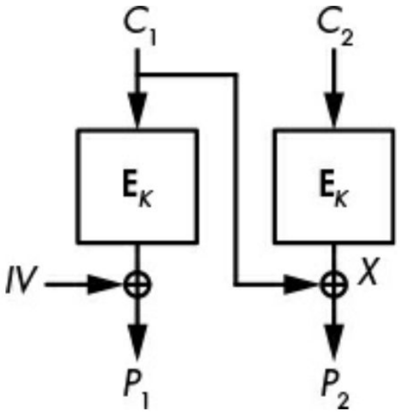
CBC Mode

- Alleviates problems with ECB by:
  - Incorporating an *Initialization Vector (IV)* that can be publicly known
  - Instead of independently encrypting blocks, the result of one encryption is chained with the input of the next encryption
    - $C_i = E_k(P_i \oplus C_{i-1})$, and $C_0 = IV$
    - $P_i = D_k(C_i) \oplus C_{i-1}$, and $C_0 = IV$
  - Encrypting same inputs = different outputs now!

- **PROBLEMS!**
  - IVs that are constant and non-random will make CBC like ECB
  - Padding Oracle Attacks

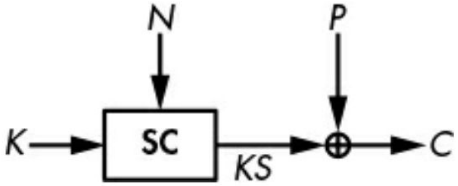
CBC Mode - Padding Oracle Attack

- Think of program doing crypto as an **black-box oracle** that gives results that dictate success/fail based on inputs
  - Given a padding oracle, we want to see which inputs we throw have *valid padding and which don't* in order to try to figure out plaintext



- CBC modes use padding oracles to check to ensure that padding during encryption is valid
  - Pass in a C1 where C1[15] ⊕ X[15] = 01, and so on with each byte from the end in order to determine C2's decryption

Symmetric Crypto - Stream Ciphers



- **Stream ciphers**
  - Pseudorandom bits are generated from a keystream, and encrypted against plaintext through XOR
  - Similar to the earlier concept of DRBGs!
- How it works:
  - A keystream is generated with a key and a one-time 64-128 bit nonce.
  - Keystream XORed against clear/ciphertext for encryption/decryption
  - Allow repeating inputs, as long as nonce is uniquely generated each time!

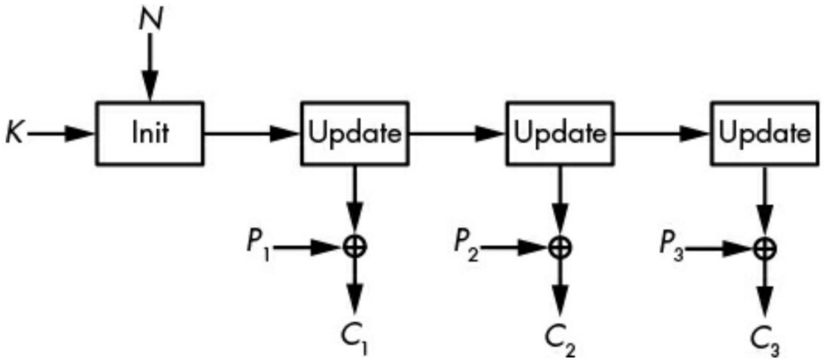
$KS = SC(K, N)$

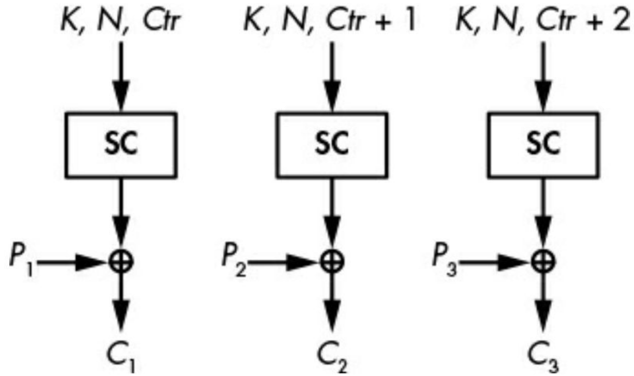
$C = P \oplus KS$

$P = C \oplus KS$

Symmetric Crypto - Stream Ciphers

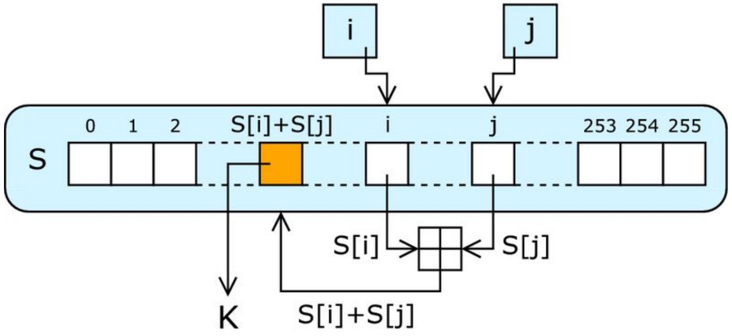
- Software stream ciphers
  - Alleviate padding oracle attacks
  - Work with 32/64-bit words
- Stateful stream ciphers
  - Internal state that changes throughout keystream generation
- Counter-based stream ciphers
  - Involves a counter that increments rather than an internal state

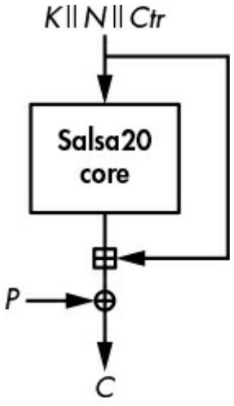




Symmetric Crypto - Stream Ciphers

- RC4
  - State-based
  - Used in WEP and SSL/TLS
  - Internal state involves byte swaps as part of its key scheduling algorithm
  - Broken, but because of various implementation-level reasons
    - Demonstration!
- Salsa20
  - Counter-based
  - Modern standard!!
  - Security and performance balanced



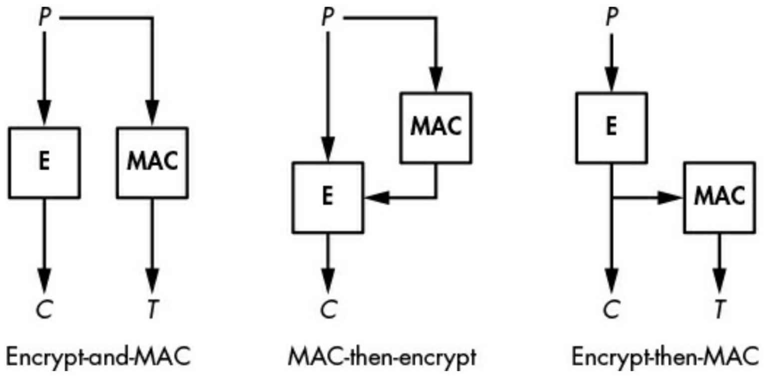


Symmetric Crypto - Authenticated Encryption

- If symmetric ciphers help preserve *confidentiality,* and hash functions help preserve *integrity*, can we combine both?
- **Authenticated Encryption** (or Authenticated Encryption with associated data, AE, AEAD) combine BOTH!
    - Utilize **MACs (Message Authentication Codes)** in order protect integrity of encrypted data in transmission plus a strong block/stream cipher
        - Aka “keyed hashing”
        - An authentication tag is generated with K and M: `T = MAC(K, M)`
        - Sent along with message, and if tampered, recomputed tag won’t be the same!
        - Hash-based MACs (HMACs) are used in crypto-schemes

Symmetric Crypto - Authenticated Encryption

- Choices of HMACs
  - Poly1305
  - SipHash
- Performing AEAD with Cipher + HMAC
  - Encrypt-and-MAC
  - MAC-then-encrypt
  - Encrypt-then-MAC



Symmetric Crypto - Authenticated Encryption

- Choices of AEAD Schemes
  - AES-GCM
  - OCB
  - **Salsa20-Poly1305**
  - **ChaCha20-Poly1305**

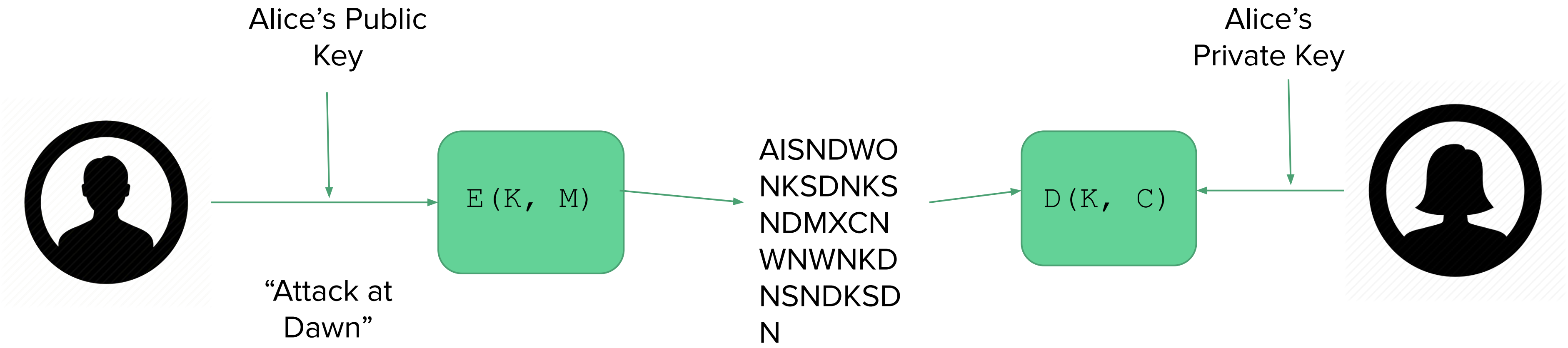
```
import os
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305

data = b"a secret message"
aad = b"authenticated but unencrypted data"

# generate a random key and object instance
key = ChaCha20Poly1305.generate_key()
chacha = ChaCha20Poly1305(key)

# create a random nonce
nonce = os.urandom(12)
ct = chacha.encrypt(nonce, data, aad)

print(chacha.decrypt(nonce, ct, aad))
~
```

Quick Look: Asymmetric Cryptography

Quick Look: Asymmetric Cryptography

"What if we want to share our keys with others to be able to decrypt transmitted info?"

Public-Key Cryptography



Closing Thoughts

- Cryptography is well-developed and studied
  - Attacks have been existing for very long, so the choices we make today are currently the best standards!
- Never implement your own crypto!
  - Understand the best design choices in different implementations and how they fit your needs

Questions?