Block Cipher Modes of Operation and Hash Functions

CS 161 Spring 2022 - Lecture 8

Review: Block Ciphers

- Encryption: input a k-bit key and n-bit plaintext, receive n-bit ciphertext
- Decryption: input a k-bit key and n-bit ciphertext, receive n-bit plaintext
- Correctness: when the key is fixed, $E_K(M)$ should be bijective
- Security
  - Without the key, $E_K(m)$ is computationally indistinguishable from a random permutation
  - Brute-force attacks take astronomically long and are not possible
- Efficiency: algorithms use XORs and bit-shifting (very fast)
- Implementation: AES is the modern standard
- Issues
  - Not IND-CPA secure because they're deterministic
  - Can only encrypt n-bit messages

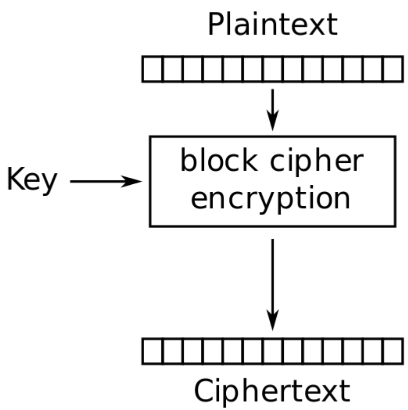
Block Cipher Modes of Operation

Textbook Chapter 6.6–6.9

Scratchpad: Let's design it together

Here's an AES block. Remember that it can only encrypt 128-bit messages.

How can we use AES to encrypt a longer message (say, 256 bits?)

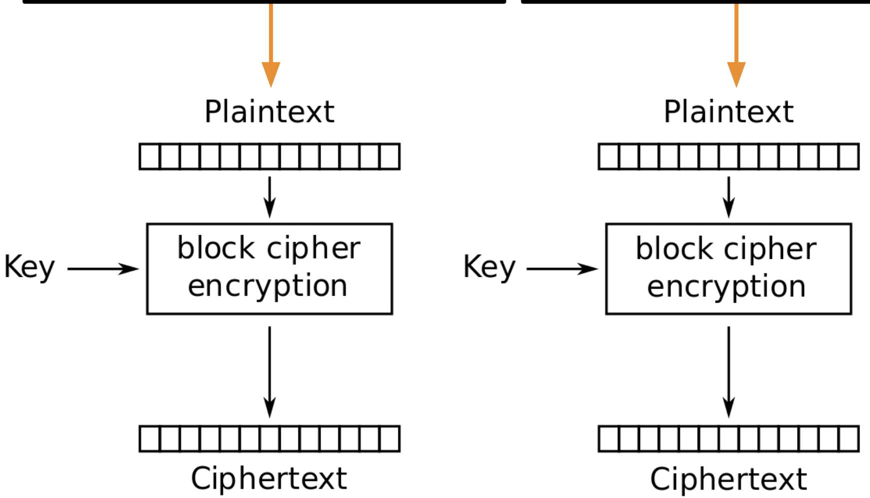


Scratchpad: Let’s design it together

Idea: Let’s use AES twice!

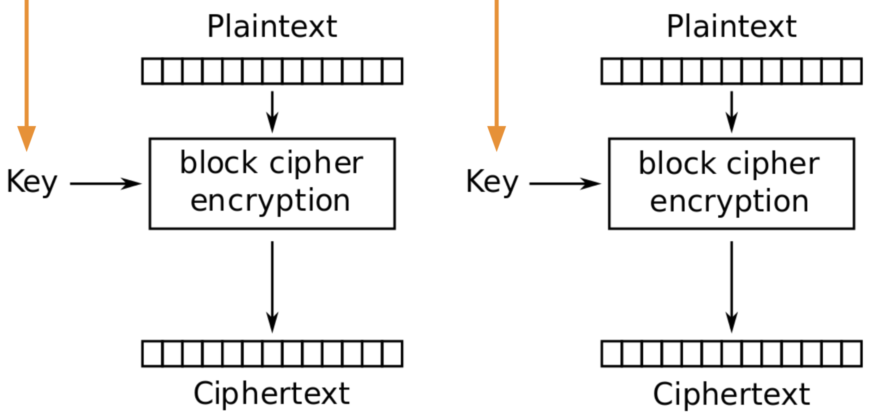
First 128 bits of message

Second 128 bits of message



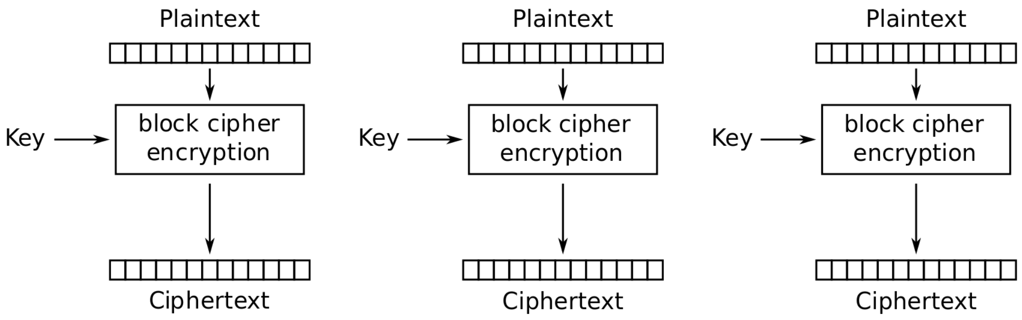
Scratchpad: Let's design it together

Note that we are using the same key twice. We want to avoid a situation like one-time pads where we need very long keys.



ECB Mode

- We've just designed **electronic code book (ECB) mode**
  - Enc(K, M) = C_1 || C_2 || … || C_m
  - Assume m is the number of blocks of plaintext in M, each of size n
- AES-ECB is not IND-CPA secure. Why?
  - Because ECB is deterministic

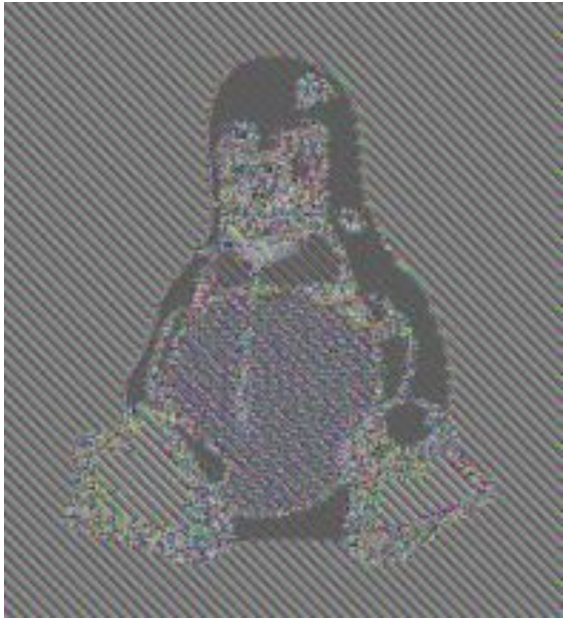


Electronic Codebook (ECB) mode encryption

ECB Mode: Penguin

Original image

ECB Mode: Penguin

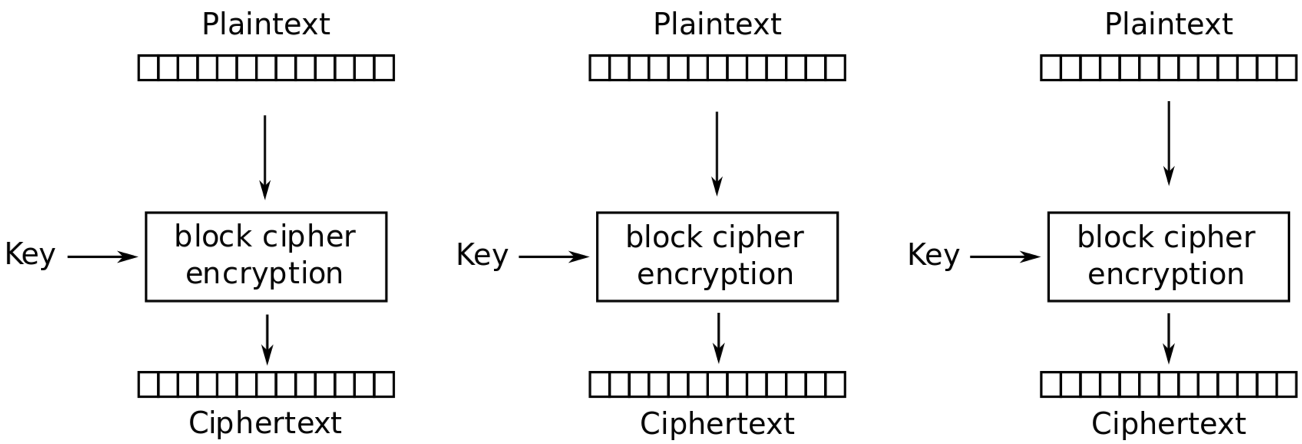


Encrypted with ECB

Scratchpad: Let's design it together

Here's ECB mode. It's not IND-CPA secure because it's deterministic.

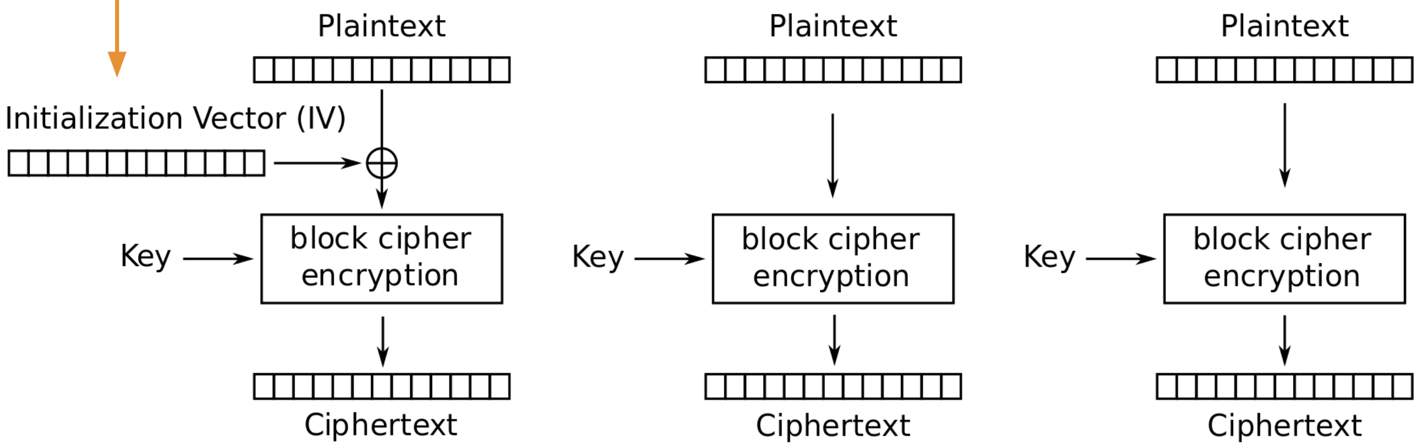
Let's fix that by adding some randomness.



Scratchpad: Let's design it together

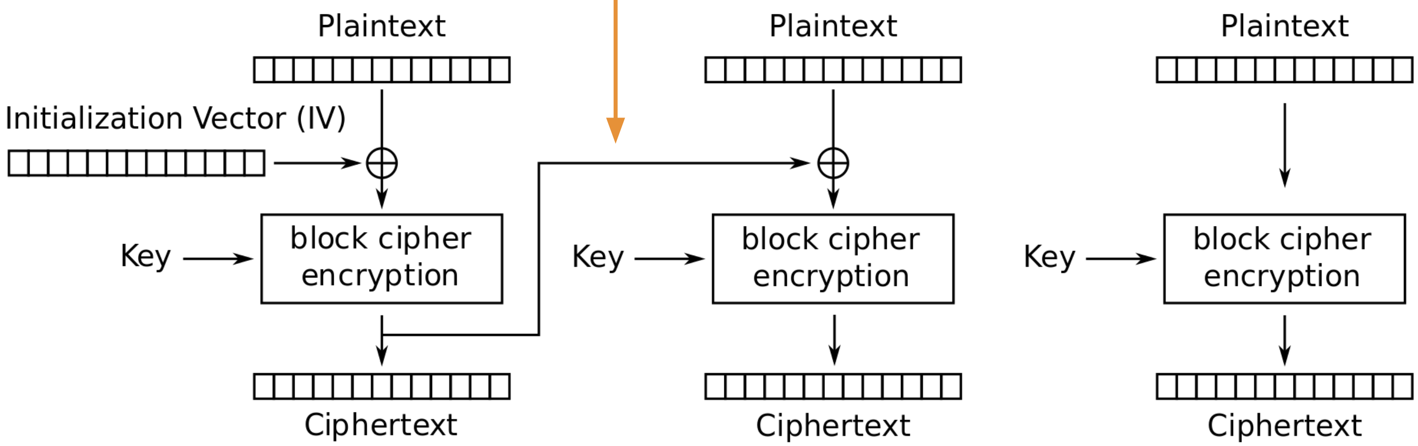
The **Initialization Vector** (**IV**) is different for every encryption. Now the first ciphertext block will be different for every encryption!

Okay, but the other blocks are still deterministic...



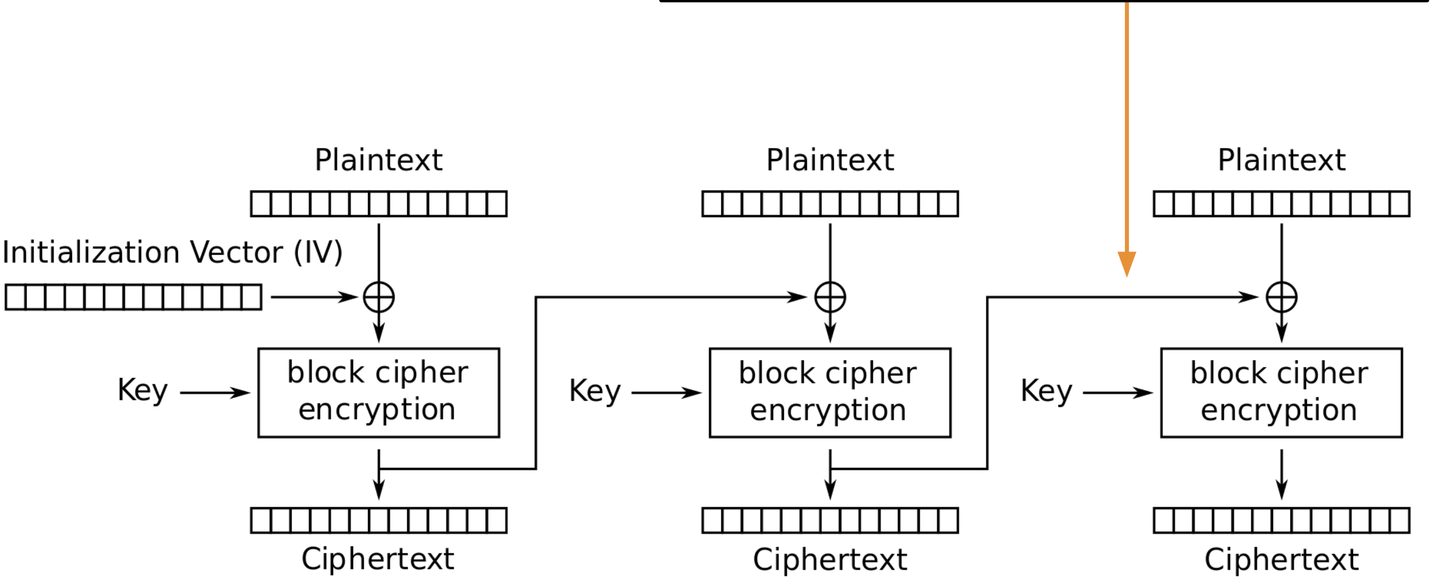
Scratchpad: Let's design it together

Idea: The first ciphertext block was computed with some randomness. Let's use it to add randomness to the second block.



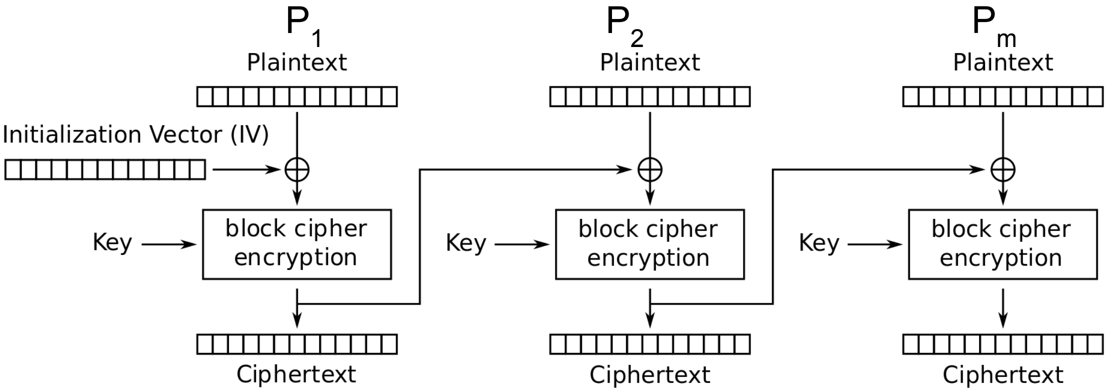
Scratchpad: Let’s design it together

Now the second ciphertext block has some randomness in it. Let’s use it to add randomness to the third block.



CBC Mode

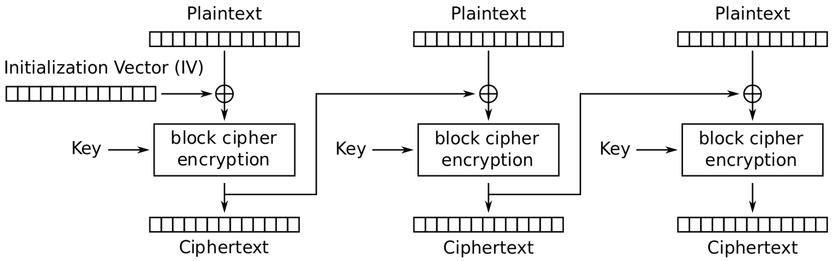
- We've just designed **cipher block chaining (CBC) mode**
- $C_i = E_K(M_i \oplus C_{i-1})$; $C_0 = IV$
- Enc(K, M):
  - Split M in m plaintext blocks $P_1 \ldots P_m$ each of size n
  - Choose a random IV
  - Compute and output $(IV, C_1, \ldots, C_m)$ as the overall ciphertext
- How do we decrypt?



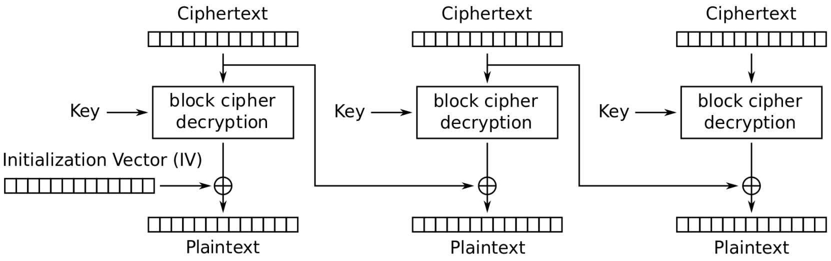
Cipher Block Chaining (CBC) mode encryption

CBC Mode: Decryption

- How do we decrypt CBC mode?
  - Parse ciphertext as (IV, C_1, …, C_m)
  - Decrypt each ciphertext and then XOR with IV or previous ciphertext



Cipher Block Chaining (CBC) mode encryption



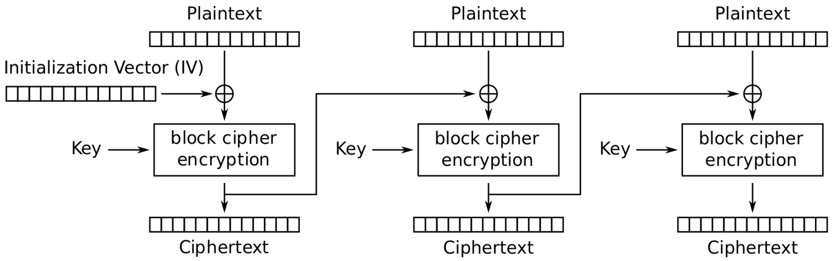
Cipher Block Chaining (CBC) mode decryption

CBC Mode: Decryption

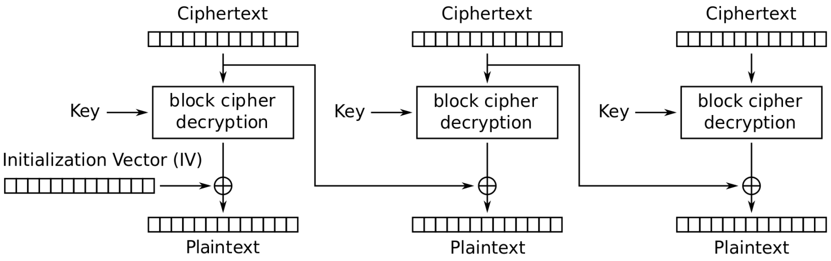
C_i	$= E_K(M_i \oplus C_{i-1})$	Definition of encryption
$D_K(C_i)$	$= D_K(E_K(M_i \oplus C_{i-1}))$	Decrypting both sides
$D_K(C_i)$	$= M_i \oplus C_{i-1}$	Decryption and encryption cancel
$D_K(C_i) \oplus C_{i-1}$	$= M_i \oplus C_{i-1} \oplus C_{i-1}$	XOR both sides with C_{i-1}
$D_K(C_i) \oplus C_{i-1}$	$= M_i$	XOR property

CBC Mode: Efficiency & Parallelism

- Can encryption be parallelized?
  - No, we have to wait for block i to finish before encrypting block i+1
- Can decryption be parallelized?
  - Yes, decryption only requires ciphertext as input



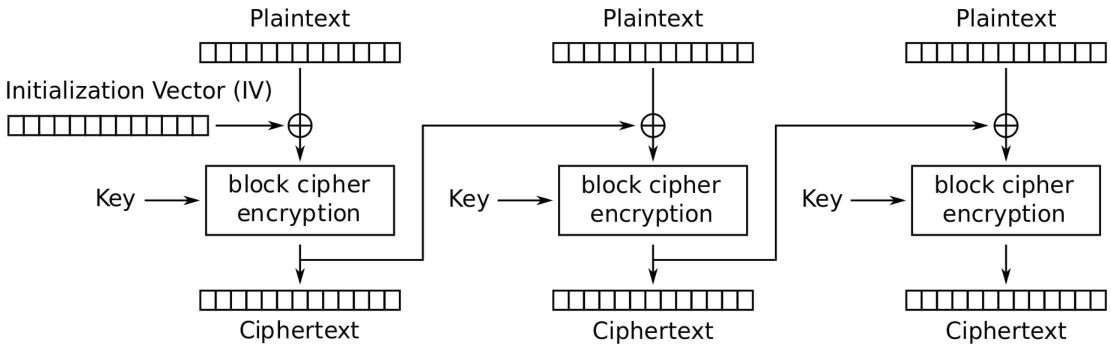
Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

CBC Mode: Padding

- What if you want to encrypt a message that isn't a multiple of the block size?
  - AES-CBC is only defined if the plaintext length is a multiple of the block size
- Solution: Pad the message until it's a multiple of the block size
  - **Padding**: Adding dummy bytes at the end of the message until it's the proper length



Cipher Block Chaining (CBC) mode encryption

CBC Mode: Padding

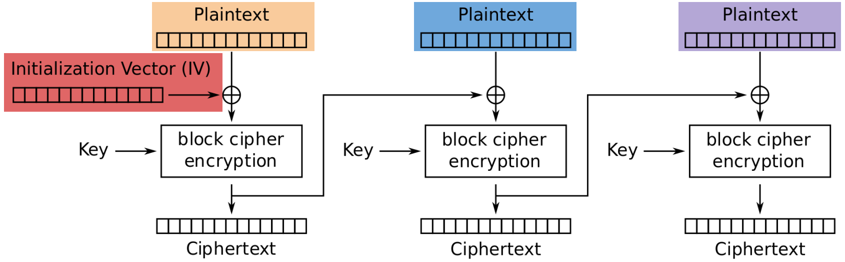
- What padding scheme should we use?
  - Padding with 0’s?
    - Doesn’t work: What if our message already ends with 0’s?
  - Padding with 1’s?
    - Same problem
- We need a scheme that can be unpadded without ambiguity
  - One scheme that works: Append a 1, then pad with 0’s
    - If plaintext is multiple of n, you still need to pad with an entire block
  - Another scheme: Pad with the number of padding bytes
    - So if you need 1 byte, pad with **01**; if you need 3 bytes, pad with **03 03 03**
    - If you need 0 padding bytes, pad an entire dummy block
    - This is called PKCS #7

CBC Mode: Security

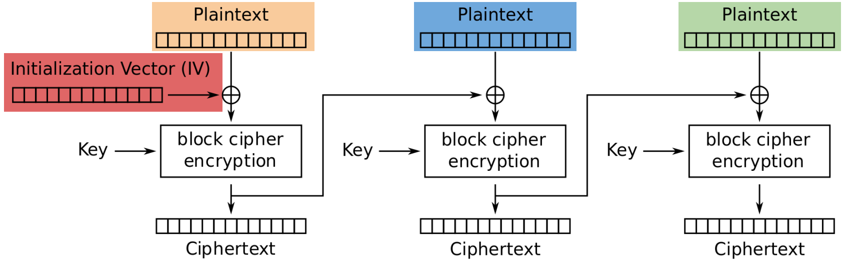
- AES-CBC is IND-CPA secure. With what assumption?
  - The IV must be randomly generated and never reused
- What happens if you reuse the IV?
  - The scheme becomes deterministic: No more IND-CPA security

CBC Mode: IV Reuse

- Consider two three-block messages: $P_1P_2P_3$ and $P_1P_2P_4$
  - The first two blocks are the same for both messages, but the last block is different
  - What if we encrypt them with the same IV?
- When the IV is reused, CBC mode reveals when two messages start with the same plaintext blocks, up to the first different plaintext block



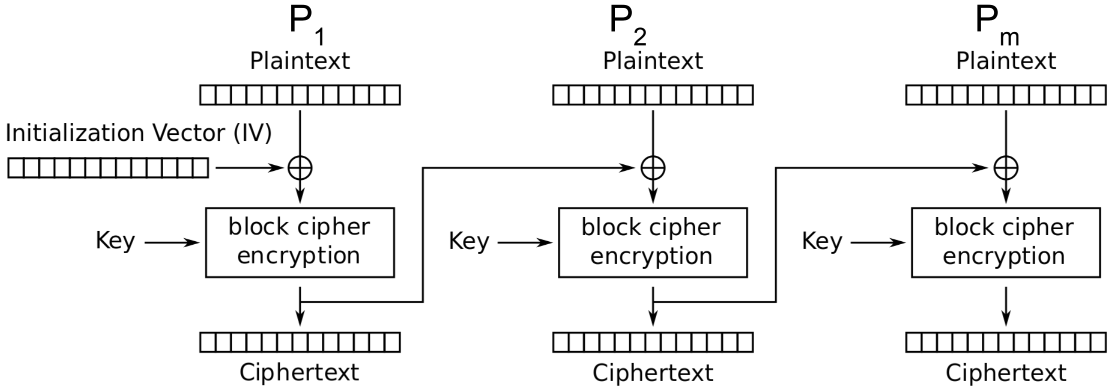
Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode encryption

CBC Mode is IND-CPA (when used correctly)

- Enc(K, M):
  - Split M in m plaintext blocks $P_1 \ldots P_m$ each of size n
  - Choose random IV, compute and output (IV, C_1, …, C_m) as the overall ciphertext
- Why IND-CPA?
  - If there exists an attacker that wins in the IND-CPA game, then there exists an attacker that breaks the block cipher security. Proof is out of scope.

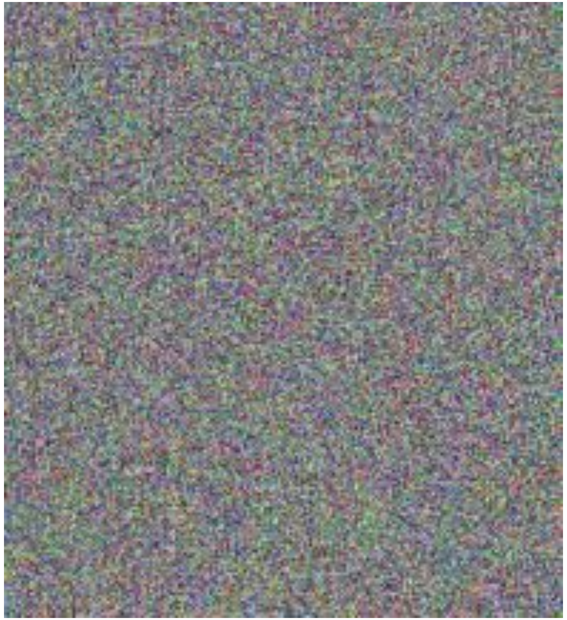


Cipher Block Chaining (CBC) mode encryption

CBC Mode: Penguin

Original image

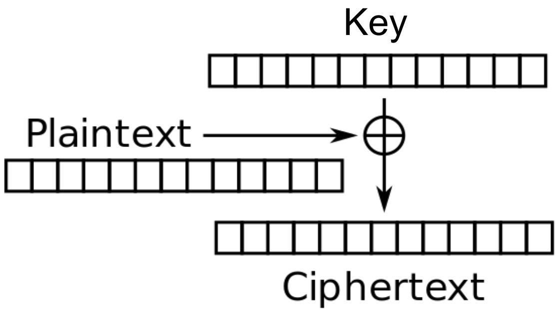
CBC Mode: Penguin



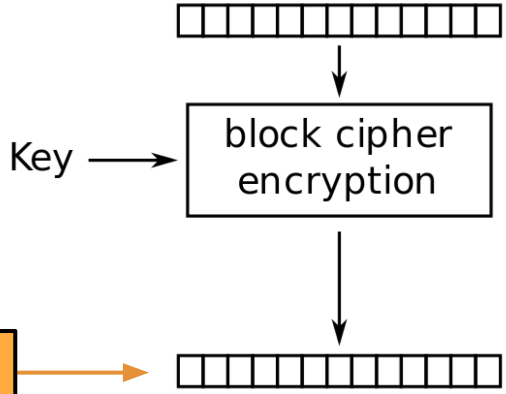
Encrypted with CBC, with random IVs

CTR Mode Scratchpad: Let's design it together

One-time pads are secure if we never reuse the key.



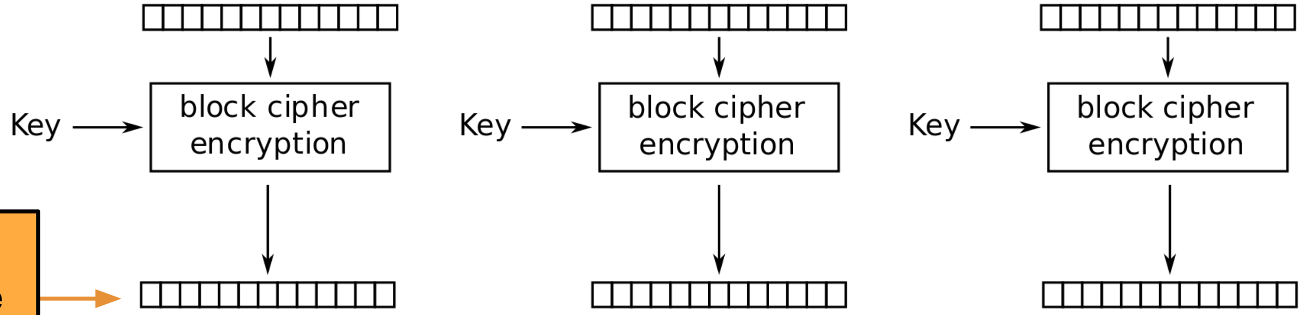
CTR Mode Scratchpad: Let's design it together



If the attacker doesn't know the key, the block cipher output looks random.

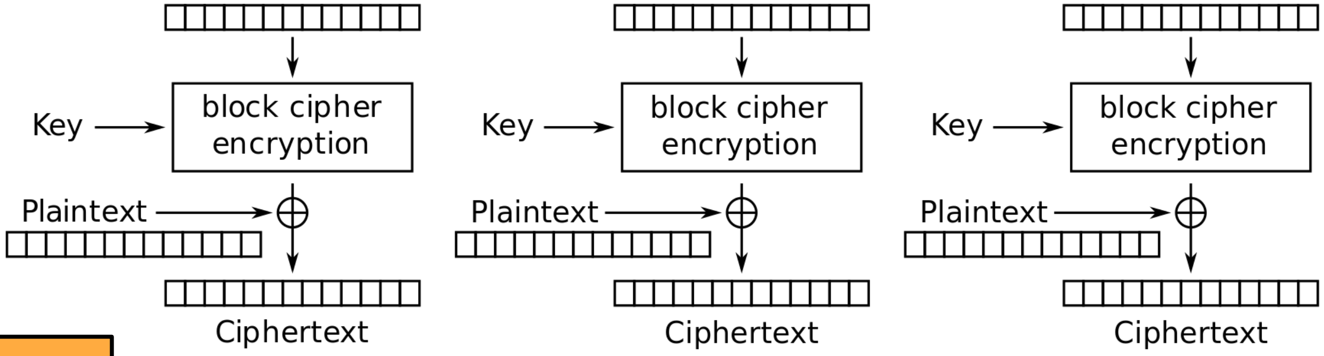
New idea: Can we use block ciphers to simulate a one-time pad?

CTR Mode Scratchpad: Let's design it together



If the attacker doesn't know the key, all of these outputs look random.

CTR Mode Scratchpad: Let's design it together

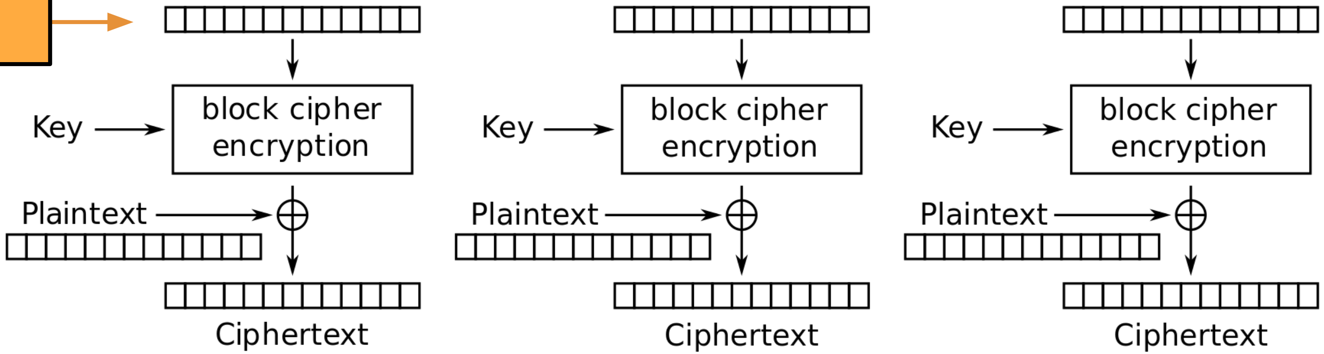


Idea: Use this random-looking output as a one-time pad!

Remember one-time pads: XOR the pad with plaintext to get ciphertext

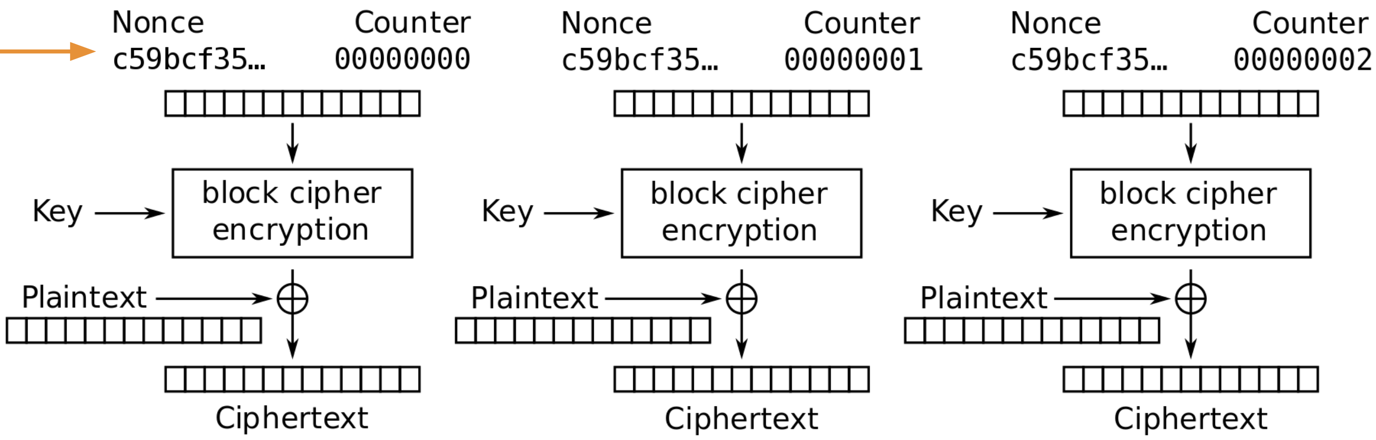
CTR Mode Scratchpad: Let’s design it together

What do we use as input to the block cipher?



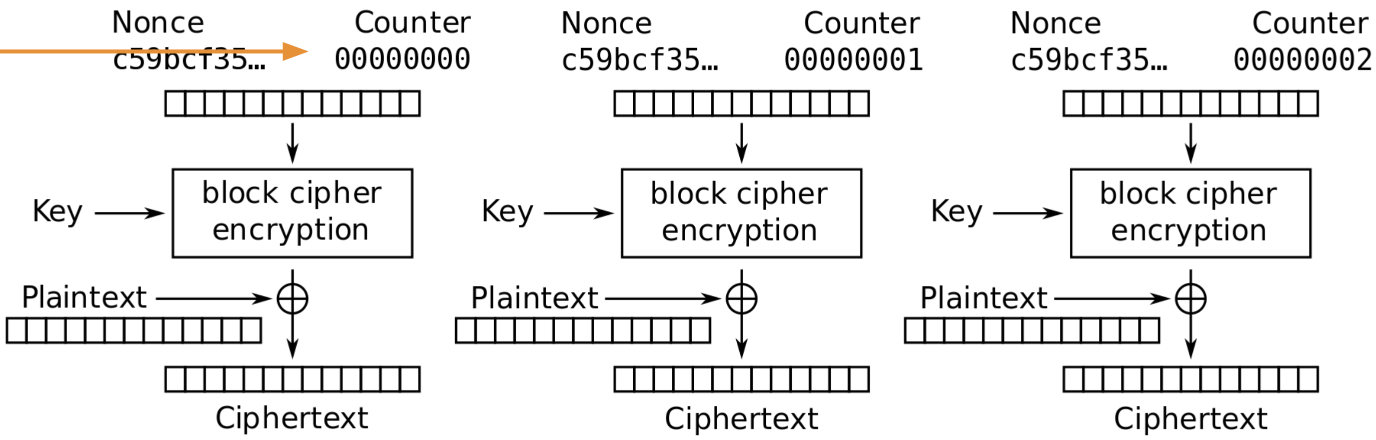
CTR Mode Scratchpad: Let's design it together

IND-CPA schemes need randomness, so let's put a random **nonce** here!



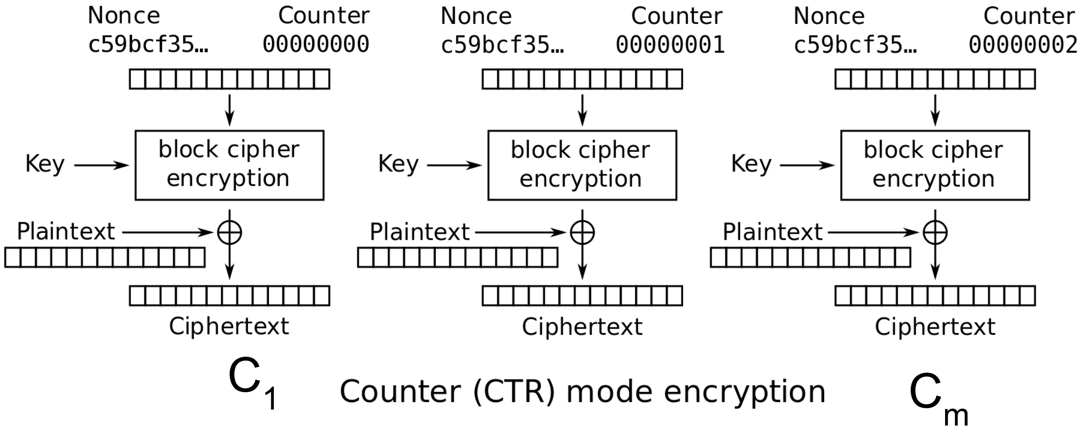
CTR Mode Scratchpad: Let’s design it together

The **counter** increments per block to ensure each block cipher output is different.



CTR (Counter) Mode

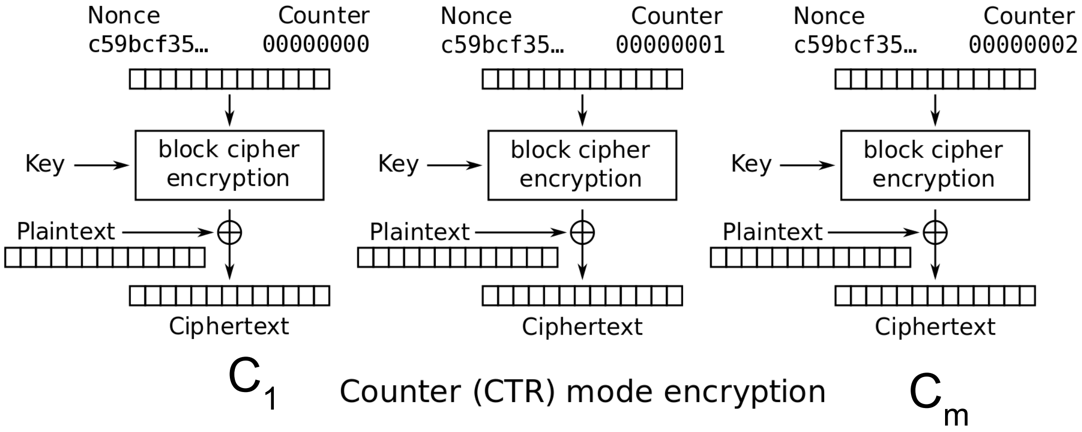
- Note: the random value is named the nonce here, but the idea is the same as the IV in CBC mode
- Overall ciphertext is (Nonce, C_1, …, C_m)



Counter (CTR) mode encryption

CTR Mode

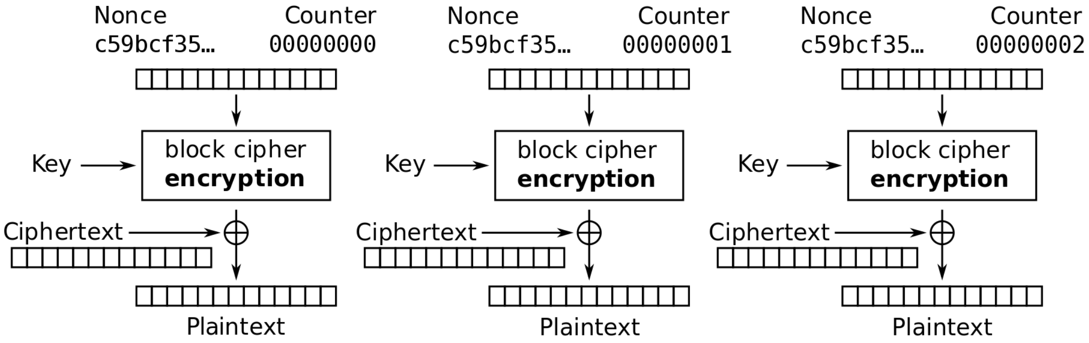
- Enc(K, M):
  - Split M in plaintext blocks $P_1 ... P_m$ (each of block size n)
  - Choose random nonce
  - Compute and output (Nonce, C_1, …, C_m)
- How do you decrypt?



Counter (CTR) mode encryption

CTR Mode: Decryption

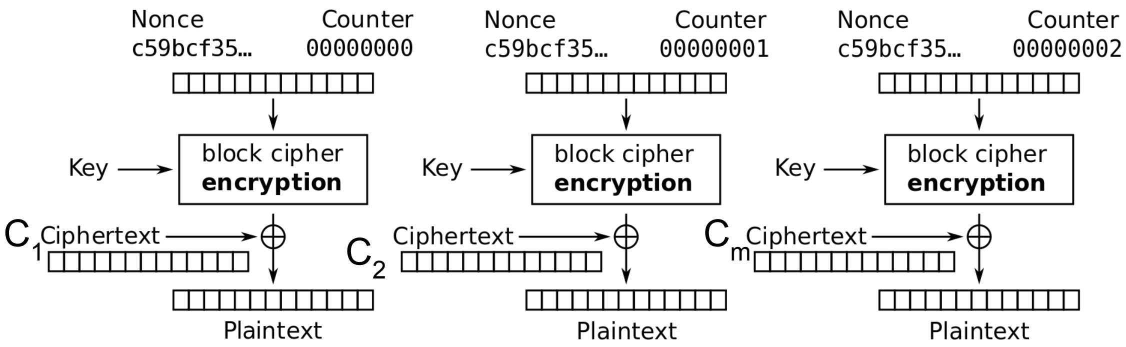
- Recall one-time pad: XOR with ciphertext to get plaintext
- Note: we are only using block cipher encryption, not decryption



Counter (CTR) mode decryption

CTR Mode: Decryption

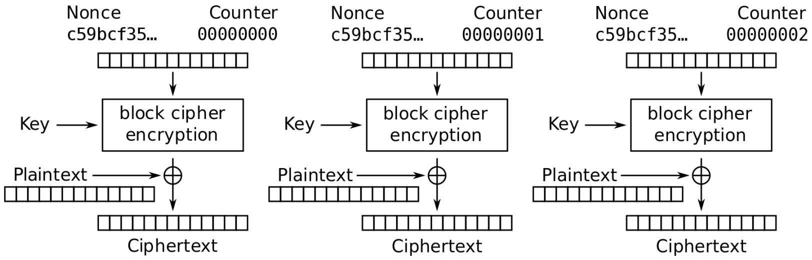
- Dec(K, C):
  - Parse C into (nonce, C_1, …, C_m)
  - Compute P_i by XORing Ci with output of E_k on nonce and counter
  - Concatenate resulting plaintexts and output M = P_1 … P_m



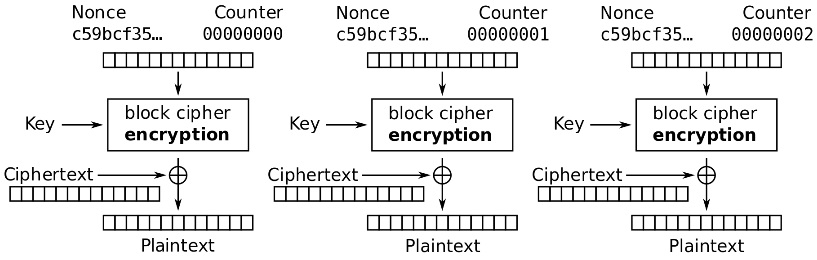
Counter (CTR) mode decryption

CTR Mode: Efficiency

- Can encryption be parallelized?
  - Yes
- Can decryption be parallelized?
  - Yes



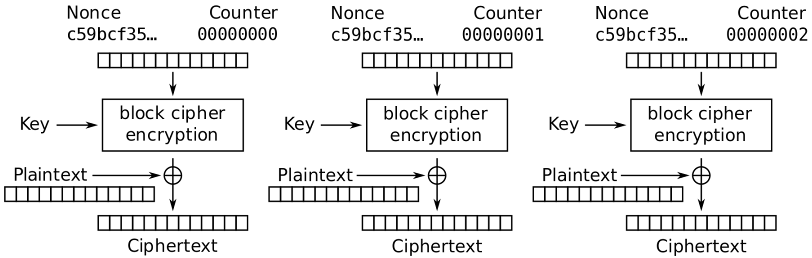
Counter (CTR) mode encryption



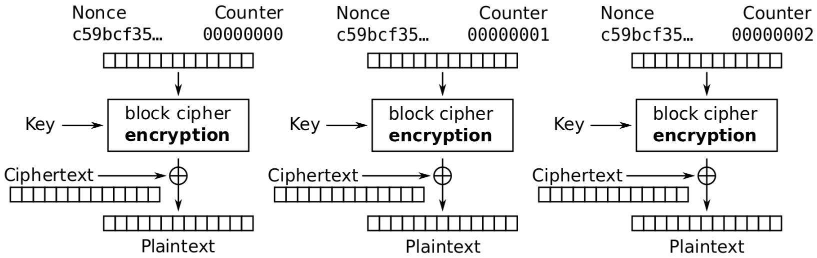
Counter (CTR) mode decryption

CTR Mode: Padding

- Do we need to pad messages?
  - No! We can just cut off the parts of the XOR that are longer than the message.



Counter (CTR) mode encryption



Counter (CTR) mode decryption

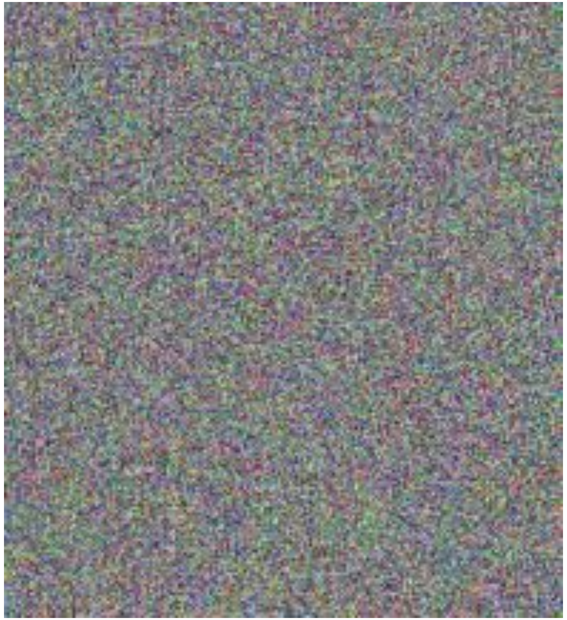
CTR Mode: Security

- AES-CTR is IND-CPA secure. With what assumption?
- The nonce must be randomly generated and never reused
  - And in general less than $2^{n/2}$ blocks are encrypted
- What happens if you reuse the nonce?
- Equivalent to reusing a key in a one-time pad
  - Recall: Key reuse in a one-time pad is catastrophic: usually leaks enough information for an attacker to deduce the entire plaintext

CTR Mode: Penguin

Original image

CTR Mode: Penguin

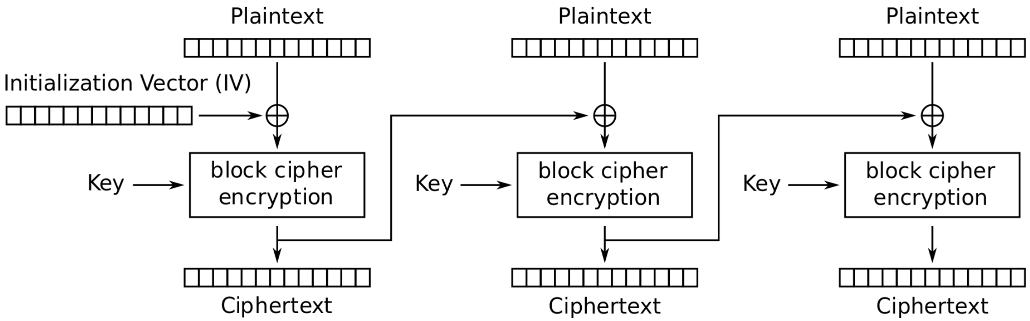


Encrypted with CTR, with random nonces

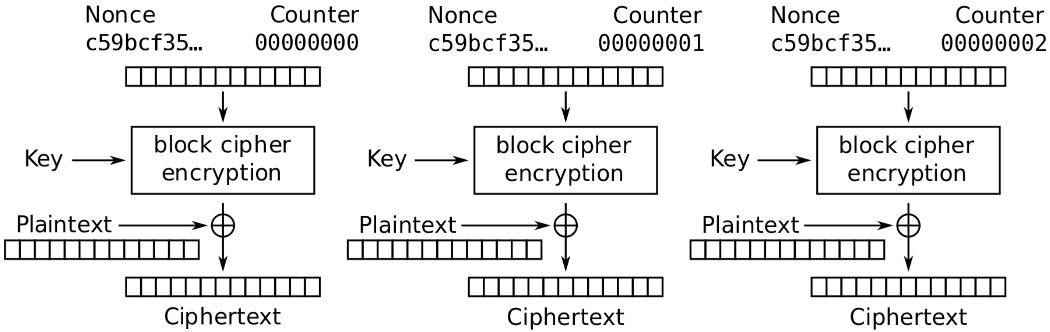
What happens if we reuse IV?

CBC: Attacker can tell that two messages have the same initial plaintext blocks all the way until the first block of difference

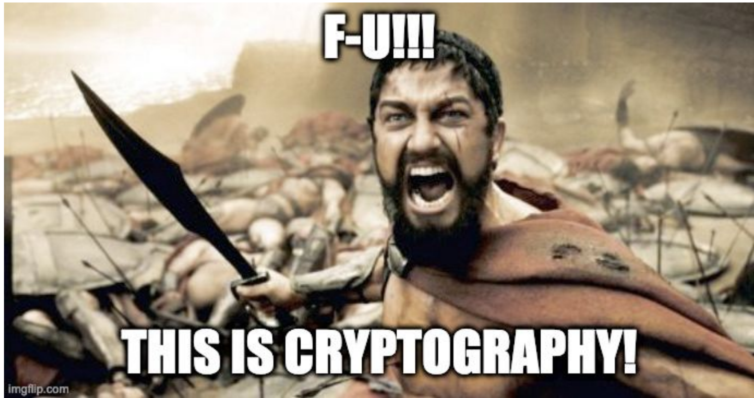
CTR: Attacker can compute the XOR between two blocks of two messages on the same positions



Cipher Block Chaining (CBC) mode encryption

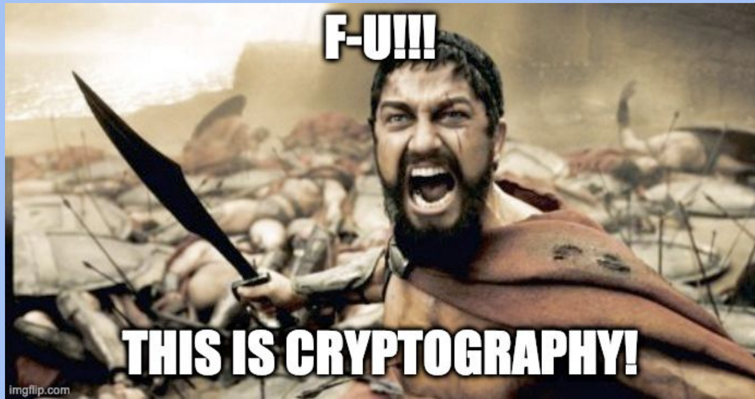


Counter (CTR) mode encryption



The summer 2020 CS 61A exam mistake

- The TAs used a Python library for AES
  - A bad library for other reasons besides this example
- When they invoked CTR mode encryption, they didn't specify an IV
  - Assumption: the crypto library would add a random IV for them
  - Reality: the crypto library defaulted to IV = 0 every time
- The same IV was used to encrypt multiple exam questions
- All security was lost!
  - Any CS 161 student could have seen the exam beforehand
- **Takeaway**: Do not reuse IVs
- **Takeaway**: Real world cryptosystems are hard. You do *not* have the skills necessary to build real world cryptosystems. I don't either.



IVs and Nonces

- **Initialization vector** (**IV**): A random, but public, one-use value to introduce randomness into the algorithm
  - For CTR mode, we say that you use a **nonce** (number used once), since the value has to be unique, not necessarily random.
  - In this class, we use IV and nonce interchangeably
- **Never reuse IVs**
  - In some algorithms, IV/nonce reuse leaks limited information (e.g. CBC)
  - In some algorithms, IV/nonce reuse leads to catastrophic failure (e.g. CTR)

IVs and Nonces

- Thinking about the consequences of IV/nonce reuse is hard
- What if the IV/nonce is not reused, but the attacker can predict future values?
  - Now you have to think about more attacks
  - We'll analyze this more in discussion: it really depends on the encryption function
- Solution: Randomly generate a new IV/nonce for every encryption
  - If the nonce is 128 bits or longer, the probability of generating the same IV/nonce twice is astronomically small (basically 0)
  - Now you don't ever have to think about IV/nonce reuse attacks!

Comparing Modes of Operation

- If you need high performance, which mode is better?
  - CTR mode, because you can parallelize both encryption and decryption
- If you're paranoid about security, which mode is better?
  - CBC mode is better
- Theoretically, CBC and CTR mode are equally secure if used properly
  - However, if used improperly (IV/nonce reuse), CBC only leaks partial information, and CTR fails catastrophically
    - Consider human factors: Systems should be as secure as possible even when implemented *incorrectly*
  - IV failures on CTR mode have resulted in multiple real-world security incidents!

Other Modes of Operation

- Other modes exist besides CBC and CTR
- Trade-offs:
    - Do we need to pad messages?
    - How robust is the scheme if we use it incorrectly?
    - Can we parallelize encryption/decryption?

Lack of Integrity and Authenticity

- Block ciphers are designed for *confidentiality* (IND-CPA)
- If an attacker tampers with the ciphertext, we are not guaranteed to detect it
- Remember Mallory: An **active** manipulator who wants to tamper with the message

Lack of Integrity and Authenticity

- Consider CTR mode
- What if Mallory tampers with the ciphertext using XOR?

	P	a	y		M	a	l		$	1	0	0
M	0x50	0x61	0x79	0x20	0x4d	0x61	0x6c	0x20	0x24	0x31	0x30	0x30

$\oplus$

$E_K(i)$	0x8a	0xe3	0x5e	0xcf	0x3b	0x40	0x46	0x57	0xb8	0x69	0xd2	0x96

=

C	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	0x58	0xe2	0xa6

Lack of Integrity and Authenticity

- Suppose Mallory knows the message *M*
- How can Mallory change the *M* to say **`Pay Mal $900`**?

	P	a	y		M	a	l		$	**1**	0	0
M	0x50	0x61	0x79	0x20	0x4d	0x61	0x6c	0x20	0x24	**0x31**	0x30	0x30

$\oplus$

$E_K(i)$	0x8a	0xe3	0x5e	0xcf	0x3b	0x40	0x46	0x57	0xb8	**0x69**	0xd2	0x96

=

C	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	**0x58**	0xe2	0xa6

Lack of Integrity and Authenticity

$C_i = M_i \oplus Pad_i$	$\mathtt{0x58} = \mathtt{0x31} \oplus Pad_i$	Definition of CTR
$Pad_i = M_i \oplus C_i$	$Pad_i = \mathtt{0x58} \oplus \mathtt{0x31}$ $= \mathtt{0x69}$	Solve for the *i*th byte of the pad
$C'_i = M'_i \oplus Pad_i$	$C'_i = \mathtt{0x39} \oplus \mathtt{0x69}$ $= \mathtt{0x50}$	Compute the changed *i*th byte

C	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	**0x58**	0xe2	0xa6
C'	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	**0x50**	0xe2	0xa6

Lack of Integrity and Authenticity

- What happens when we decrypt *C'*?
  - The message looks like “Pay Mal $900” now!
  - Note: Mallory didn’t have to know the key; no integrity or authenticity for CTR mode!

C'	0xda	0x82	0x27	0xef	0x76	0x21	0x2a	0x77	0x9c	**0x50**	0xe2	0xa6

⊕

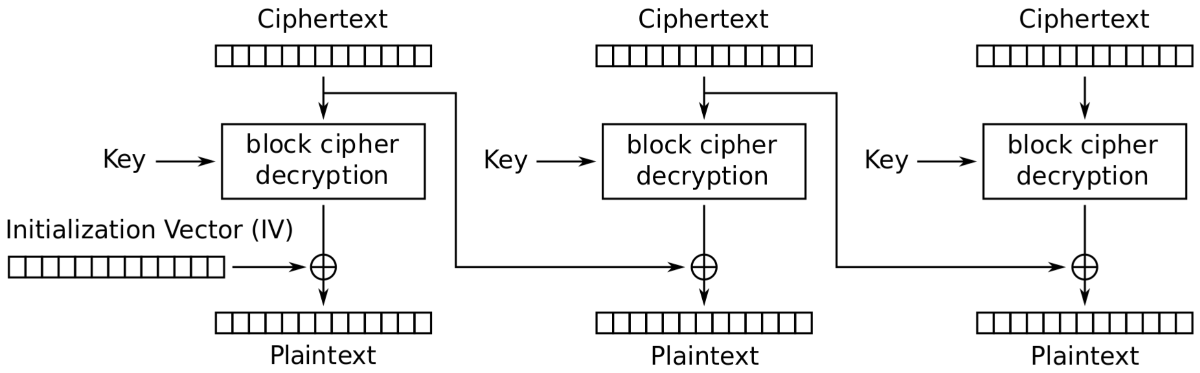
$E_K(i)$	0x8a	0xe3	0x5e	0xcf	0x3b	0x40	0x46	0x57	0xb8	**0x69**	0xd2	0x96

=

P'	0x50	0x61	0x79	0x20	0x4d	0x61	0x6c	0x20	0x24	**0x39**	0x30	0x30
	P	a	y		M	a	l		$	**9**	0	0

Lack of Integrity and Authenticity

- What about CBC?
  - Altering a bit of the ciphertext causes some blocks to become random gibberish
  - However, Bob cannot prove that Alice did not send random gibberish, so it still does *not* provide integrity or authenticity



Cipher Block Chaining (CBC) mode decryption

Preview: The Alternative: Stream Ciphers

- Recall: **Block ciphers** encrypt a block of bits at a time
- The other main symmetric encryption idea is **stream ciphers**
  - Overall idea: If I XOR my message with a bunch of random bits, the resulting message looks random and is secure
  - If key *K* = the random bits, this is a one-time pad
  - If key *K* is used to *generate* a stream of pseudorandom (“random-looking”) bits, this is a stream cipher
    - Notice: CTR mode does exactly this! CTR mode turns block ciphers into stream ciphers.
- We’ll talk more about stream ciphers and generating random and pseudorandom bits later

Block Cipher Modes of Operation: Summary

- ECB mode: Deterministic, so not IND-CPA secure
- CBC mode
  - IND-CPA secure, assuming no IV reuse
  - Encryption is not parallelizable
  - Decryption is parallelizable
  - Must pad plaintext to a multiple of the block size
  - IV reuse leads to leaking the existence of identical blocks at the start of the message
- CTR mode
  - IND-CPA secure, assuming no IV reuse
  - Encryption and decryption are parallelizable
  - Plaintext does not need to be padded
  - Nonce reuse leads to losing all security
- Lack of integrity and authenticity

Next: Cryptography Hashes and MACs

- Hashing
  - Definition
  - Security: one-way, second preimage resistant, collision resistant
  - Examples
  - Length extension attacks
  - Application: Lowest-hash scheme
  - Do hashes provide integrity?
- MACs
  - Definition
  - Security: unforgeability
  - Example: HMAC
  - Do MACs provide integrity?

Cryptographic Hashes

Textbook Chapter 7.1–7.3

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC)	• RSA encryption • ElGamal encryption
Integrity, Authentication	• MACs (e.g. HMAC)	• Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)
- Key management (certificates)
- Password management

Cryptographic Hash Function: Definition

- Hash function: $H(M)$
  - Input: *Arbitrary* length message M
  - Output: *Fixed* length, n-bit hash
  - Sometimes written as $\{0, 1\}^* \rightarrow \{0, 1\}^n$
- Properties
  - **Correctness**: Deterministic
    - Hashing the same input always produces the same output
  - **Efficiency**: Efficient to compute
  - **Security**: One-way-ness (“preimage resistance”)
  - **Security**: Collision-resistance
  - **Security:** Random/unpredictability, no predictable patterns for how changing the input affects the output
    - Changing 1 bit in the input causes the output to be completely different
    - Also called “random oracle” assumption

Hash Function: Intuition

- A hash function provides a fixed-length “fingerprint” over a sequence of bits
- Example: Document comparison
  - If Alice and Bob both have a 1 GB document, they can both compute a hash over the document and (securely) communicate the hashes to each other
  - If the hashes are the same, the files must be the same, since they have the same “fingerprint”
  - If the hashes are different, the files must be different

Hash Function: One-way-ness or Preimage Resistance

- **Informal:** Given an output y, it is infeasible to find **any** input x such that $H(x) = y$
- **More formally:** For all polynomial time adversary,

  Pr[x chosen randomly from plaintext space; $y = H(x)$:

  Adv(y) outputs x' s.t. $H(x') = y$] is negligible
- Intuition: Here's an output. Can you find an input that hashes to this output?
  - Note: The adversary just needs to find *any* input, not necessarily the input that was actually used to generate the hash

Is this function one-way?

- The constant function $H(x) = 1$
  - No, because an attacker can output any x, and that leads to 1.
  - It does not have to be the original x some challenger thought about
- Analogy: Take a cow and make a burger: yes
- A believed secure block cipher, for K random and secret: E_K
  - Yes, if an attacker can invert E_K, then an attacker can distinguish it from a random permutation which breaks the security of the block cipher
  - But if the attacker knows K then it is no longer a one-way function

Hash Function: Collision Resistance

- **Collision**: Two different inputs with the same output
  - $x \neq x'$ and $H(x) = H(x')$
  - Can we design a hash function with no collisions?
    - No, because there are more inputs than outputs (pigeonhole principle)
  - However, we want to make finding collisions *infeasible* for an attacker
- **Collision resistance**: It is infeasible to (i.e. no polynomial time attacker can) find any pair of inputs $x' \neq x$ such that $H(x) = H(x')$
- Intuition: Can you find *any* two inputs that collide with the same hash output for *any* output?

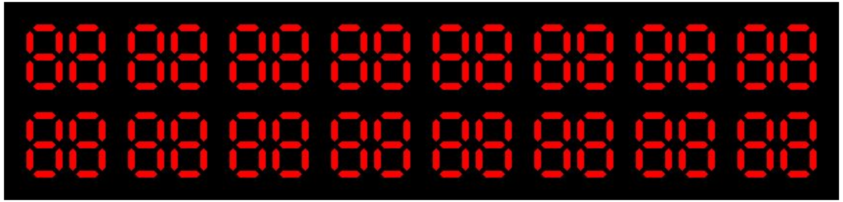
Hash Function: Collision Resistance

- **Birthday attack**: Finding a collision on an *n*-bit output requires only $2^{n/2}$ tries on average
    - This is why a group of 23 people are >50% likely to have at least one birthday in common

Hash Function: Random Oracle/Unpredictability

- Intuitively, the hash function behaves like a random mapping, so $H(x)$ maps x to a random value
  - An attacker should not be able to predict how a change in input affects the output

Hash Function: Examples



A GIF that displays its own MD5 hash

- MD5
    - Output: 128 bits
    - Security: Completely broken
- SHA-1
    - Output: 160 bits
    - Security: Completely broken in 2017
    - Was known to be weak before 2017, but still used sometimes
- SHA-2
    - Output: 256, 384, or 512 bits (sometimes labeled SHA-256, SHA-384, SHA-512)
    - Not currently broken, but some variants are vulnerable to a length extension attack
    - Current standard
- SHA-3 (Keccak)
    - Output: 256, 384, or 512 bits
    - Current standard (not meant to replace SHA-2, just a different construction)

Length Extension Attacks

- **Length extension attack**: Given $H(x)$ and the length of x, but not x, an attacker can create $H(x \,||\, m)$ for any m of the attacker's choosing
  - See Homework 3 for a demo
  - Note: This doesn't violate any property of hash functions but is undesirable in some circumstances
- SHA-256 (256-bit version of SHA-2) is vulnerable
- SHA-3 is not vulnerable

Do hashes provide integrity?

- It depends on your threat model
- Scenario
  - Mozilla publishes a new version of Firefox on some download servers
  - Alice downloads the program binary
  - How can she be sure that nobody tampered with the program?
- Idea: use cryptographic hashes
  - Mozilla hashes the program binary and publishes the hash on its website
  - Alice hashes the binary she downloaded and checks that it matches the hash on the website
  - If Alice downloaded a malicious program, the hash would not match (tampering detected!)
  - An attacker can’t create a malicious program with the same hash (collision resistance)
- Threat model: We assume the attacker cannot modify the hash on the website
  - We have integrity, as long as we can communicate the hash securely

Do hashes provide integrity?

- It depends on your threat model
- Scenario
  - Alice and Bob want to communicate over an insecure channel
  - Mallory might tamper with messages
- Idea: Use cryptographic hashes
  - Alice sends her message with a cryptographic hash over the channel
  - Bob receives the message and computes a hash on the message
  - Bob checks that the hash he computed matches the hash sent by Alice
- Threat model: Mallory can modify the message *and the hash*
  - No integrity!

Do hashes provide integrity?

- It depends on your threat model
- If the attacker can modify the hash, hashes don't provide integrity
- Main issue: Hashes are *unkeyed* functions
  - There is no secret key being used as input, so any attacker can compute a hash on any value
- Next: Use hashes to design schemes that provide integrity