CS 370 Introduction to Security

Message Authentication Code and

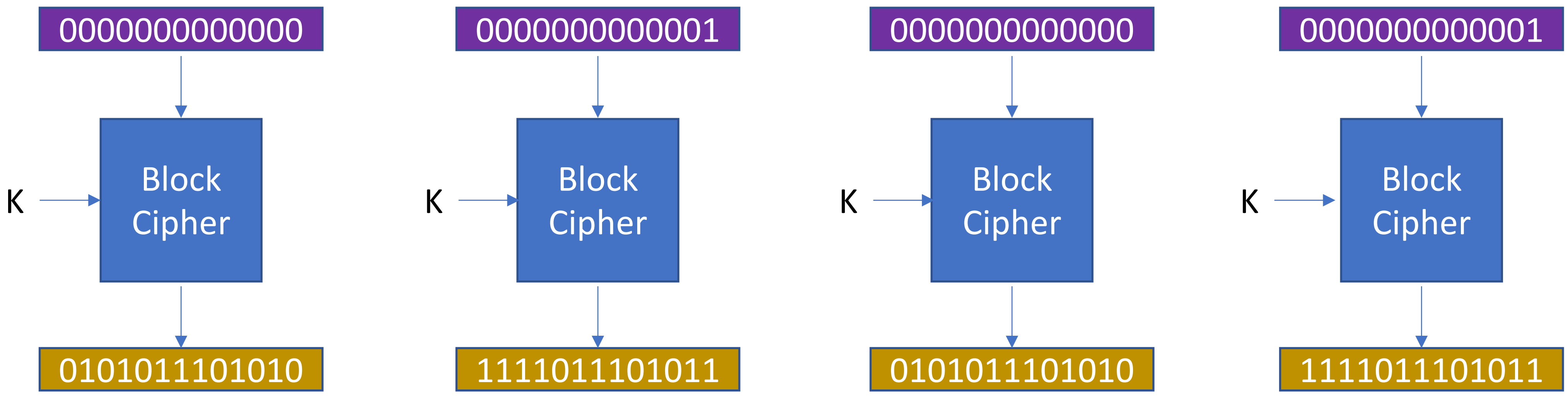
Asymmetric Encryption

Yeongjin Jang

Oregon State University

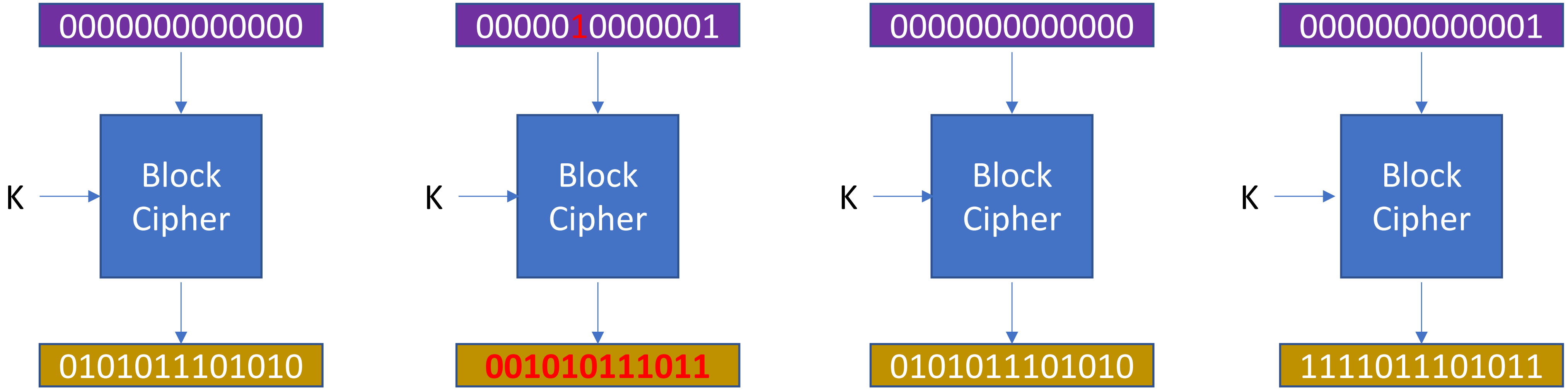
Recap: Electronic Code Book

- We can run encryption in parallel



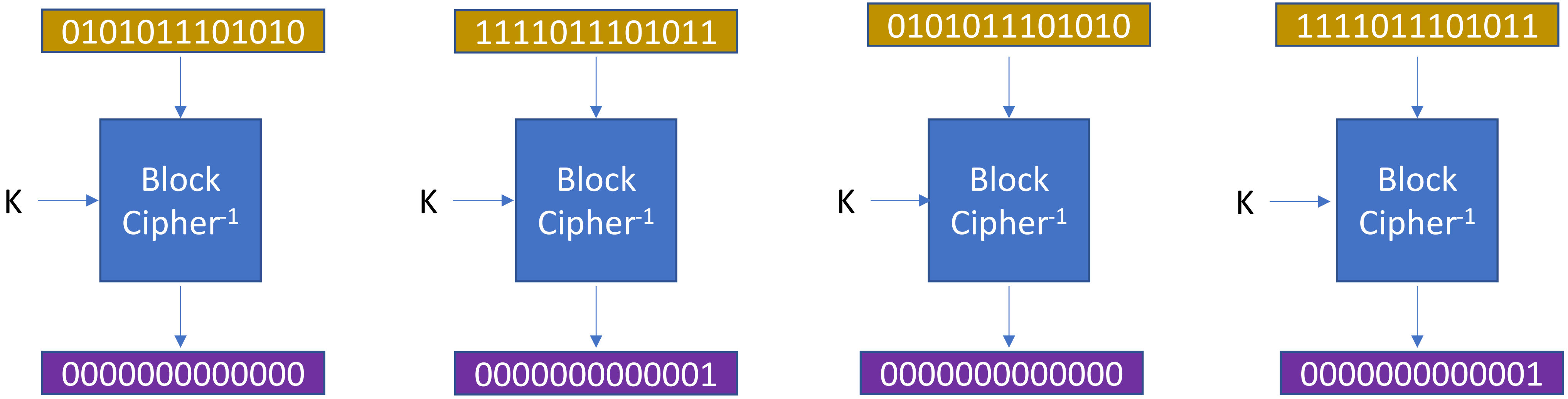
Recap: Electronic Code Book

- A specific bit error in ciphertext would result in a random error in plaintext



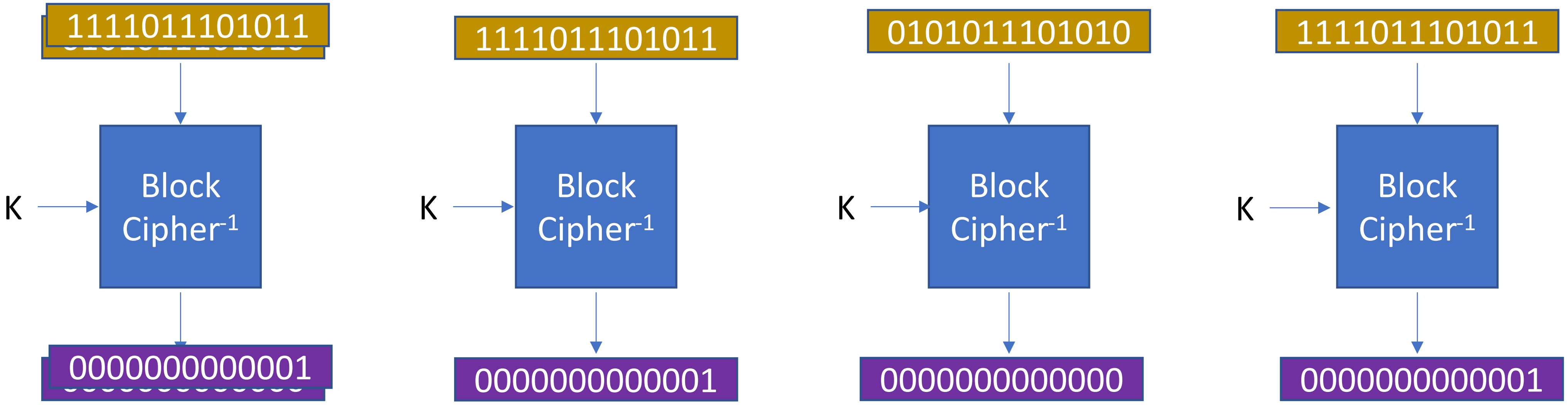
Recap: Electronic Code Book

- We can launch a message block substitution attack



Recap: Electronic Code Book

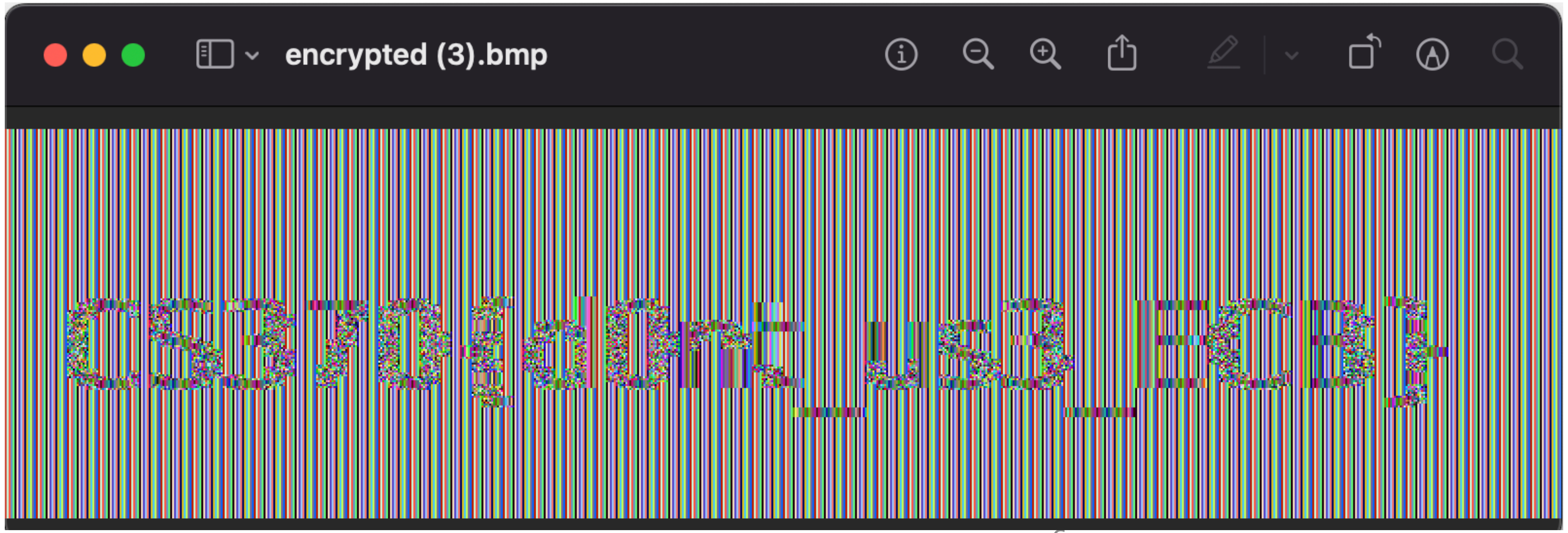
- We can launch a message block substitution attack



Recap: Electronic Code Book

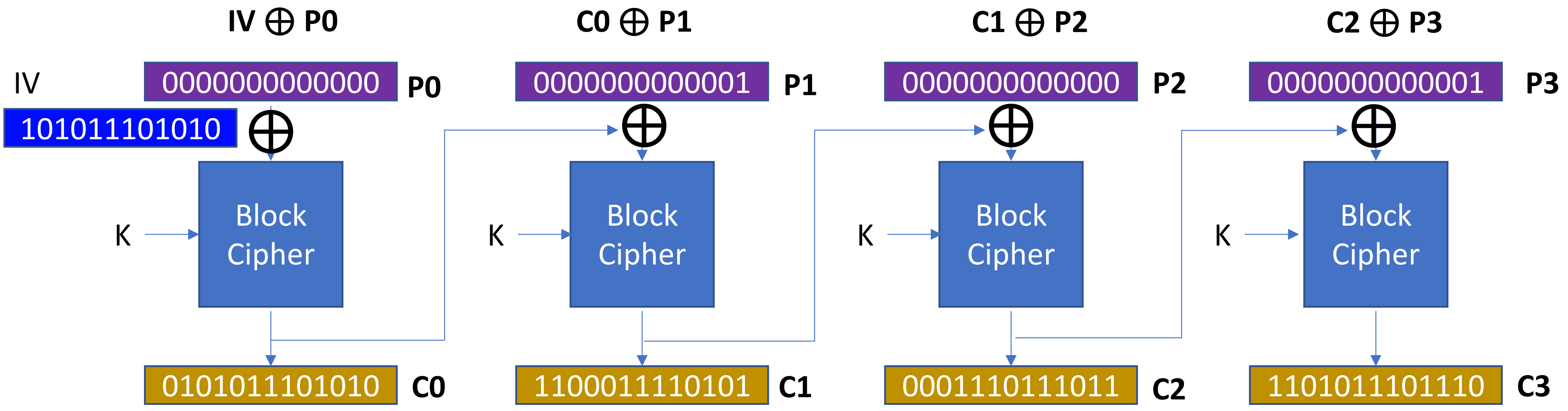
- Can encrypt in parallel
- Can decrypt in parallel

- Ciphertext block leaks plaintext block patterns



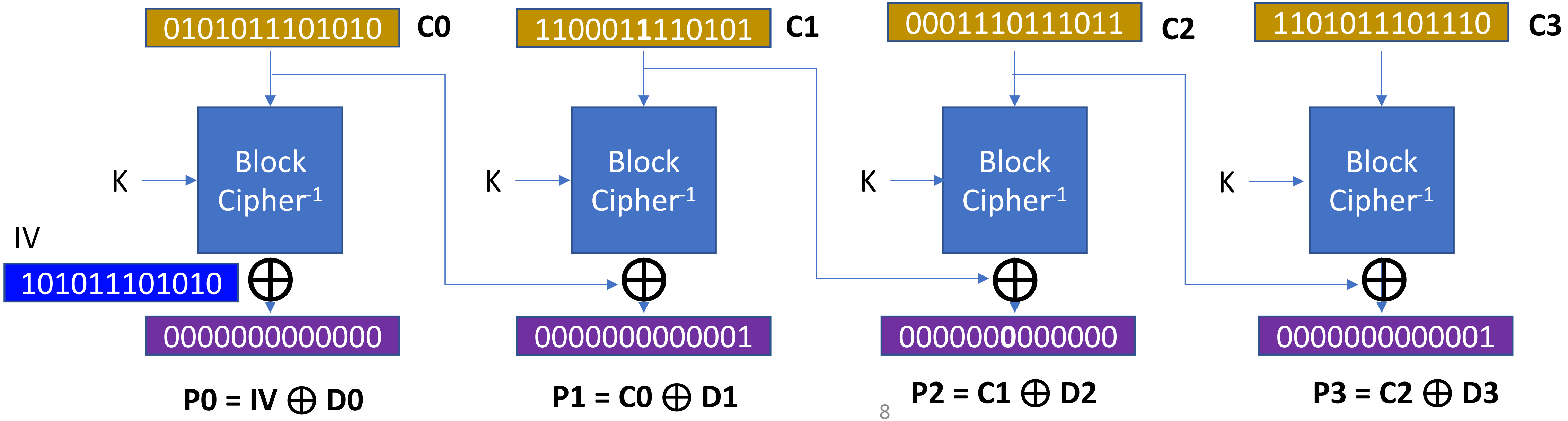
Recap: Cipher Block Chain (CBC)

- Apply XOR between the IV (Initialization Vector) and the plaintext
- Chain the previous ciphertext block to the plaintext with XOR
- Run Encryption on Xor'ed data



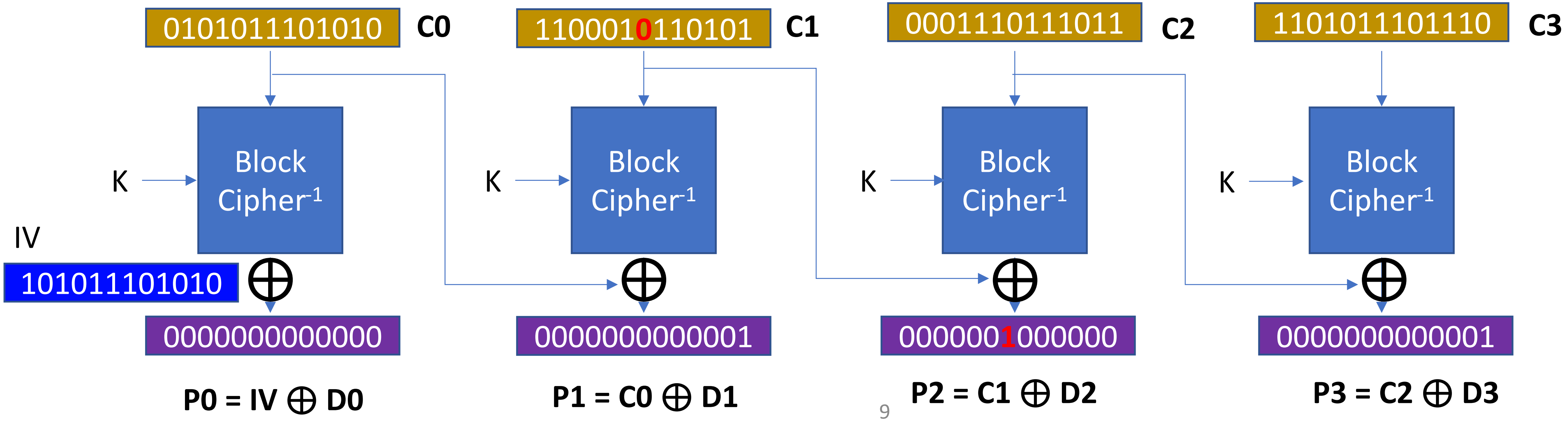
Recap: CBC Attack

- A specific bit error in cipertext-n will be a specific bit error in plaintext-(n+1)



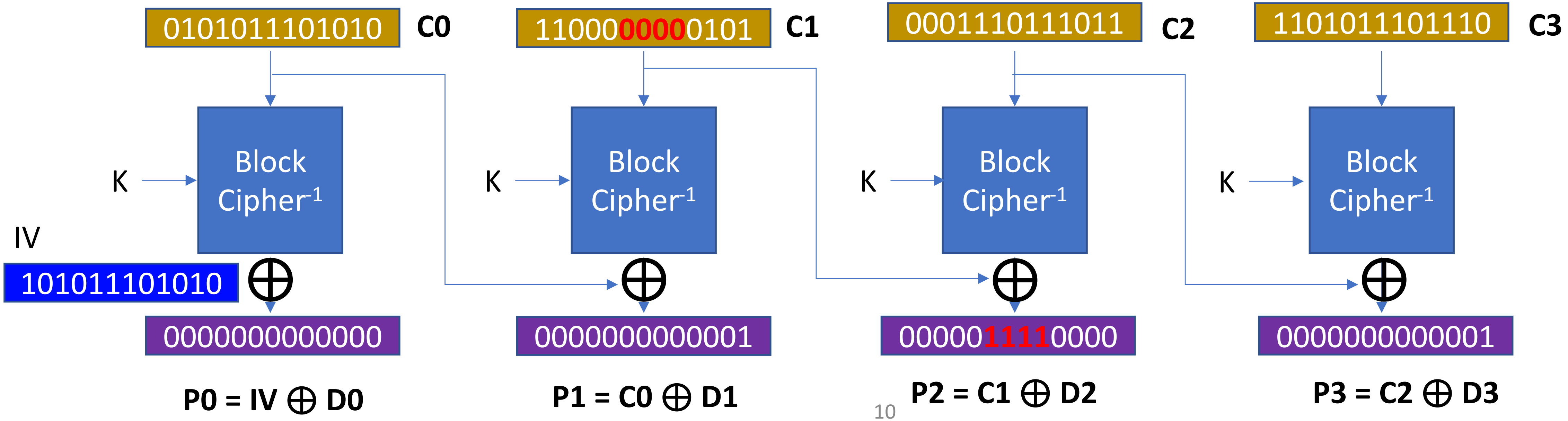
Recap: CBC Attack

- A specific bit error in cipertext-n will be a specific bit error in plaintext-(n+1)



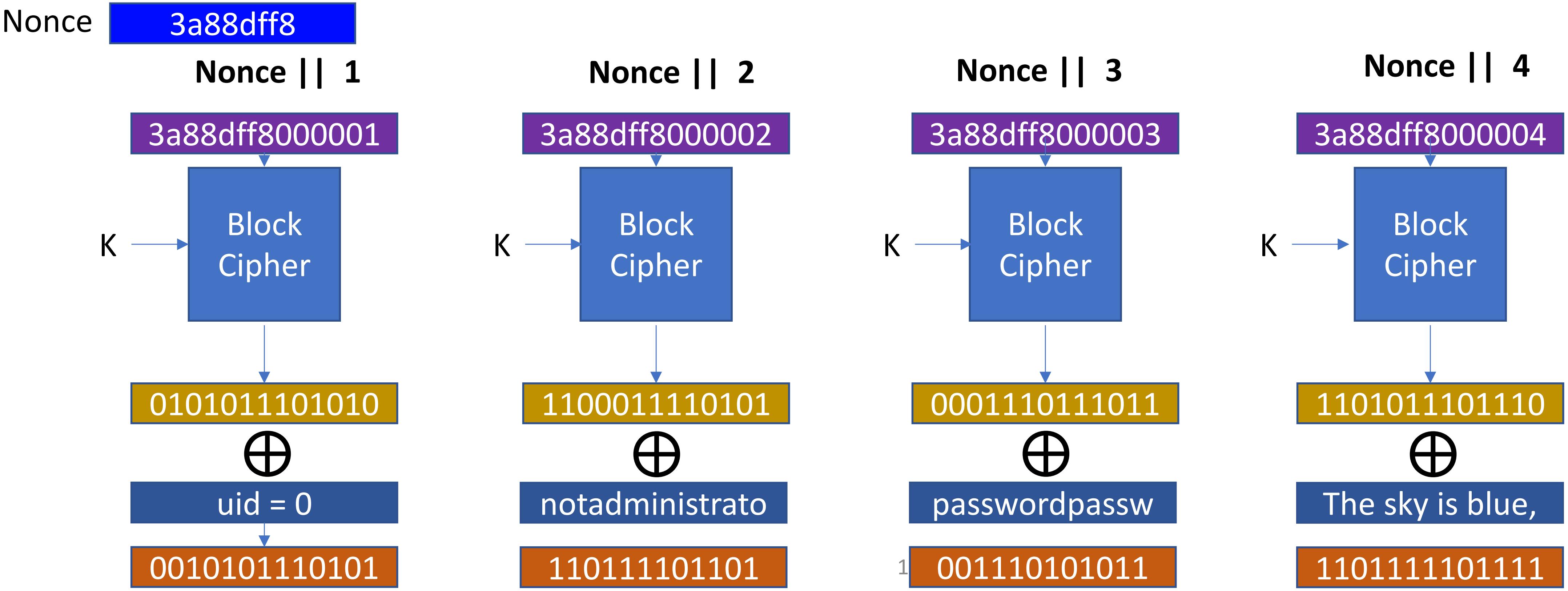
Recap: CBC Attack

- A specific bit error in cipertext-n will be a specific bit error in plaintext-(n+1)



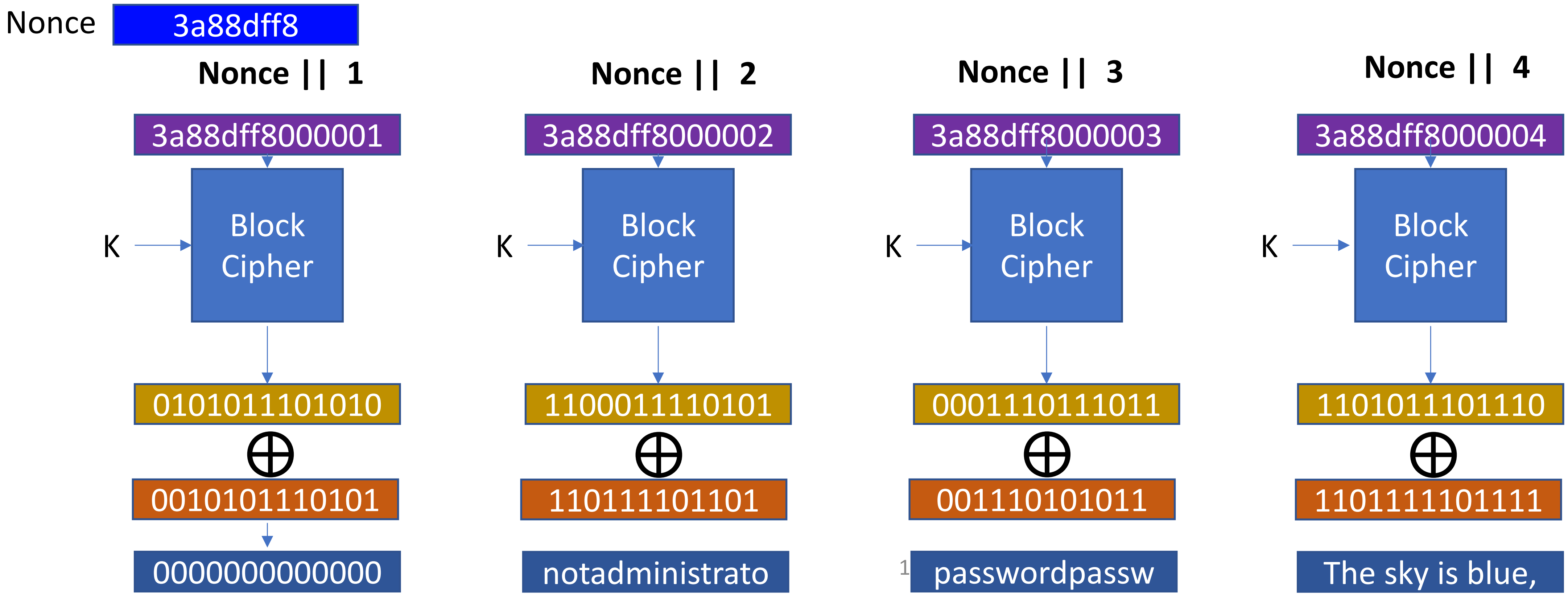
Recap: Counter Mode (CTR)

- CTR (Counter mode)
- Start with a random nonce || counter



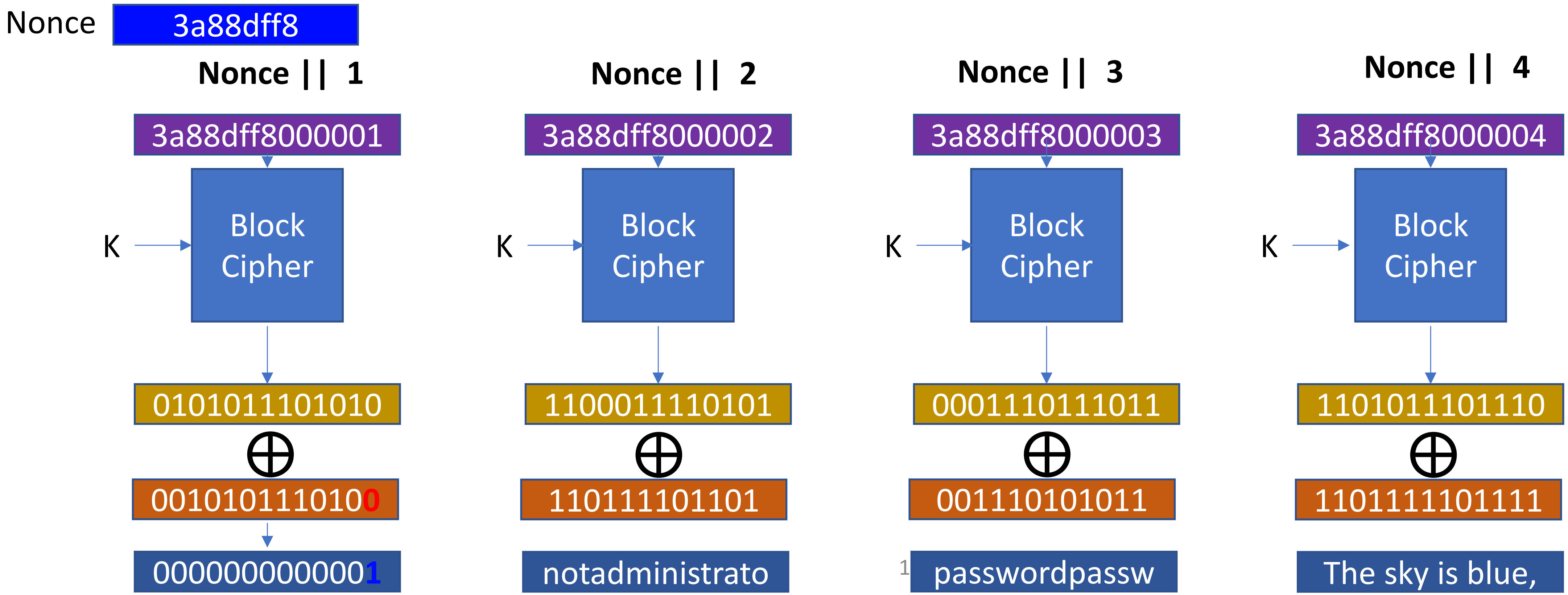
Recap: CTR Attack

- A specific bit error in cipertext-n will be a specific bit error in plaintext-n



Recap: CTR Attack

- A specific bit error in cipertext-n will be a specific bit error in plaintext-n



Block Cipher

- The block cipher itself cannot protect encrypted data modified by attackers
  - ECB, we can substitute blocks to known-plaintext-encrypted-block
  - CBC, we can apply XOR to the ciphertext that is one-block before the plaintext
  - CTR, we can apply XOR to the ciphertext then the result will show on the plaintext

- Why?
  - Block Cipher gives us data confidentiality
  - Not data integrity

ecb-, cbc-, ctr-attack

- Threat model
  - We have a verifier with the secret key
  - We have an encrypted user data, encrypted.user
  - We don’t have the key -> cannot arbitrarily generate valid encrypted file

- What can we do?
  - Edit the ciphertext in encrypted.user

- What can we achieve?
  - Change the plaintext at our own will

ecb-, cbc-, ctr-attack

- Threat model
  - We have a verifier with the secret key
  - We have an encrypted user data, encrypted.user
  - We don't have the key
- What can we do?
  - Edit the ciphertext in encrypted.user <- dangerous
- What can we achieve?
  - Change the plaintext at our own will

Is there any cryptographic tool that we can check the integrity of data?

- What we have
  - Encrypted data, e.g.,
  - encrypted.user

- What we want
  - Detect if the attacker has modified the file encrypted.user

- What tool can we use?
  - Cryptographic hash!

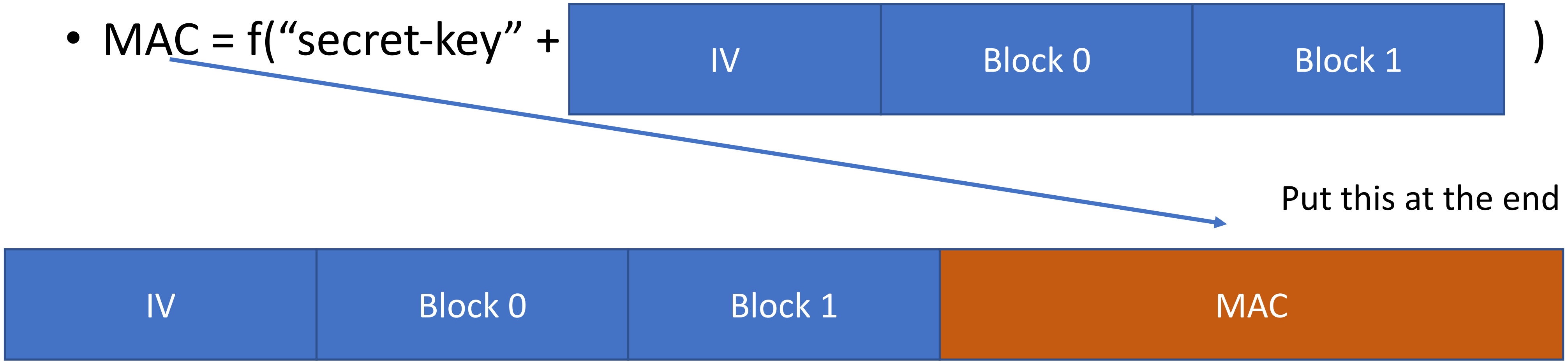
An Ideal Hash Function

- Suppose we have a function f(x) that
  - Generates a fixed length message (e.g., 32-byte)
  - You cannot get easily get $f^{-1}(y) = x$ from y
  - A slight value change in x for f(x) will result in drastic change in y
    - So you cannot correlate any f(x') = y' from f(x) = y

- What I will do:
  - f("secret-key" + encrypted_data) = message_authentication_code

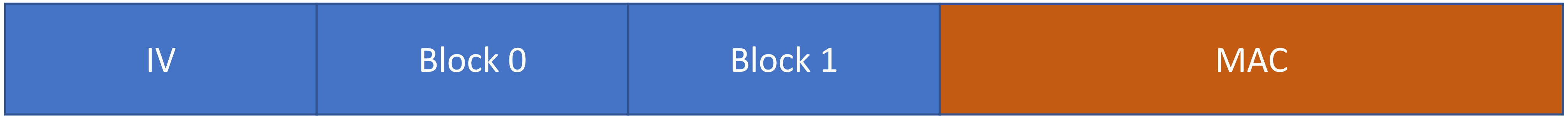
Create a MAC (Message Authentication Code)

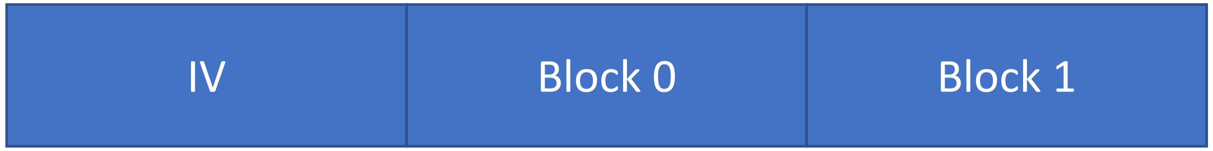
- What I will do:
  - f(“secret-key” + encrypted_data) = message_authentication_code
  - MAC = f(“secret-key” + IV Block 0 Block 1)



Checking a MAC

- When I read the encrypted data

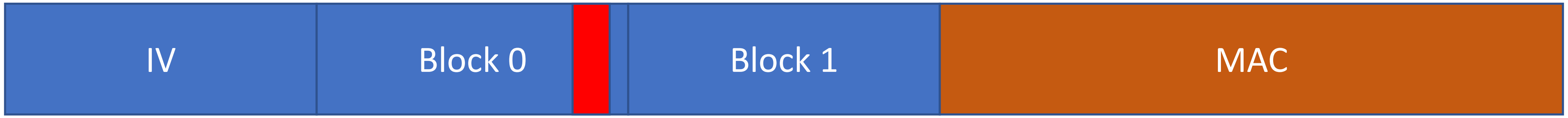


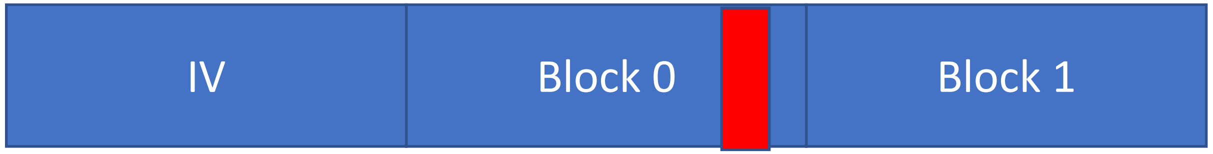
- MAC = f(“secret-key” + )

- MAC should be equal to 

What Attackers Can Do?

- What if they edited data?



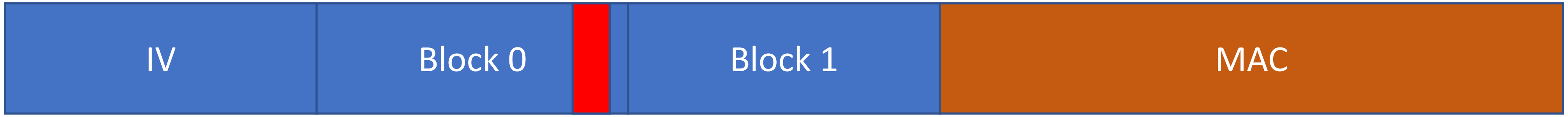
- MAC’ = f(“secret-key” + )

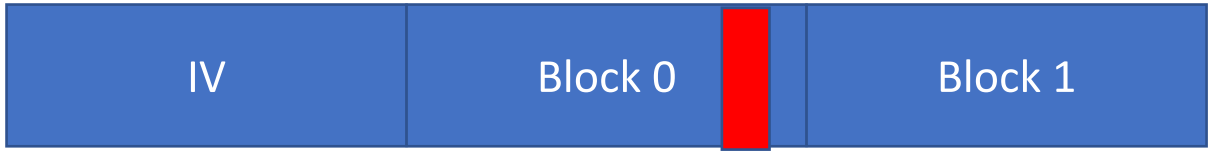
- MAC’ != 

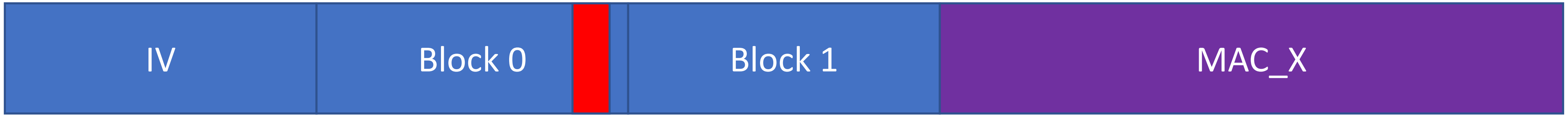
- Suppose we have a function f(x) that
  - A slight value change in x for f(x) will result in drastic change in y

They Can Generate Valid MAC if They know the key

- What if they edited data?



- MAC_X = f("**secret-key**" + )



- But they don't know the secret key; can't generate it
- Suppose we have a function f(x) that
  - You cannot get easily get $f^{-1}(y) = x$ from y
  - A slight value change in x for f(x) will result in drastic change in y

Cryptographic Hash

- A hash function that generates a fingerprint of a data
- SHA256(‘Hello, world’) =
  `03675ac53ff9cd1535ccc7dfcdfa2c458c5218371f418dc136f2d19ac1fbe8a5`

- With characteristics of:
  - One-way function
  - Hard to find x for given y where H(x) = y
  - Hard to find x’ for given x,y where x != x’, H(x) = y and H(x’) = y

SHA256

- Secure Hash Algorithm (SHA)
  - SHA256 is in the SHA2 standard
  - Input can be any length data
  - Output is 256-bit, 32-byte
- SHA256 is a cryptographic hash function that
  - It is one-way function
  - SHA256('Hello, world') = `03675ac53ff9cd1535ccc7dfcdfa2c458c5218371f418dc136f2d19ac1fbe8a5`
  - SHA256^{-1}(`03675ac53ff9cd1535ccc7dfcdfa2c458c5218371f418dc136f2d19ac1fbe8a5`) == ???? there could be many..

SHA256 Examples

```
┌─blue9057@blue9057-vm-ctf1 ~/t ‹ruby-head›
└─$ sha256sum *
9a271f2a916b0b6ee6cecb2426f0b3206ef074578be55d9bc94f6f3fe3ab86aa  0
4355a46b19d348dc2f57c046f8ef63d4538ebb936000f3c9ee954a27460dd865  1
53c234e5e8472b6ac51c1ae1cab3fe06fad053beb8ebfd8977b010655bfdd3c3  2
1121cfccd5913f0a63fec40a6ffd44ea64f9dc135c66634ba001d10bcf4302a2  3
7de1555df0c2700329e815b93b32c571c3ea54dc967b89e81ab73b9972b72d1d  4
f0b5c2c2211c8d67ed15e75e656c7862d086e9245420892a7de62cd9ec582a06  5
87428fc522803d31065e7bce3cf03fe475096631e5e07bbd7a0fde60c4cf25c7  a
0263829989b6fd954f72baaf2fc64bc2e2f01d692d4de72986ea808f6e99813f  b
a3a5e715f0cc574a73c3f9bebb6bc24f32ffd5b67b387244c2c909da779a1478  c
8d74beec1be996322ad76813bafb92d40839895d6dd7ee808b17ca201eac98be  d
a2bbdb2de53523b8099b37013f251546f3d65dbe7a0774fa41af0a4176992fd4  e
```

SHA256

- SHA256 is a cryptographic hash function that
  - Hard to find x for given y where H(x) = y
  - Find SHA256(x) for
    - `00000000000000000000000000000000000000000f418dc136f2d19ac1fbe8a5`
  - This task requires around the 2^{256} times of search…
- **Implication**
  - **If we know X, it is easy to get SHA256(X) = Y**
  - **But if we don’t know X, even if we know Y, it is hard to calculate X**

SHA256

- SHA256 is a cryptographic hash function that
  - Hard to find x’ for given x,y where x’ != x, H(x) = y, H(x’) = H(x)
  - SHA256(‘Hello, world’) = `03675ac53ff9cd1535ccc7dfcdfa2c458c5218371f418dc136f2d19ac1fbe8a5`

  - Can you find another x’ that produces SHA256(x’) = `03675ac53ff9cd1535ccc7dfcdfa2c458c5218371f418dc136f2d19ac1fbe8a5`
  - Other than ‘Hello, world’?

- **Implication**
  - **Even if we know X, Y where SHA256(X) = Y**
  - **It is hard to find SHA256(X’) = Y**

Avalanche Effect

- Even with a slight change in input, we want to have a huge change in output to not making attackers correlate output values based on their inputs…

```
┌─blue9057@blue9057-vm-ctf1 ~/t ‹ruby-head›
└─$ sha256sum *
9a271f2a916b0b6ee6cecb2426f0b3206ef074578be55d9bc94f6f3fe3ab86aa  0
4355a46b19d348dc2f57c046f8ef63d4538ebb936000f3c9ee954a27460dd865  1
53c234e5e8472b6ac51c1ae1cab3fe06fad053beb8ebfd8977b010655bfdd3c3  2
1121cfccd5913f0a63fec40a6ffd44ea64f9dc135c66634ba001d10bcf4302a2  3
7de1555df0c2700329e815b93b32c571c3ea54dc967b89e81ab73b9972b72d1d  4
f0b5c2c2211c8d67ed15e75e656c7862d086e9245420892a7de62cd9ec582a06  5
87428fc522803d31065e7bce3cf03fe475096631e5e07bbd7a0fde60c4cf25c7  a
0263829989b6fd954f72baaf2fc64bc2e2f01d692d4de72986ea808f6e99813f  b
a3a5e715f0cc574a73c3f9bebb6bc24f32ffd5b67b387244c2c909da779a1478  c
8d74beec1be996322ad76813bafb92d40839895d6dd7ee808b17ca201eac98be  d
a2bbdb2de53523b8099b37013f251546f3d65dbe7a0774fa41af0a4176992fd4  e
```

Cryptographic Hash: Implications

- SHA256 is a cryptographic hash that is included in the SHA2 standard

- SHA256 is a one-way function and

- It is hard to calculate x from y
  - where y = SHA256(x)

- It is hard to calculate x’ from x,y
  - where x’ != x, SHA256(x) = y, SHA256(x’) = y

- It is hard to correlate x and x’ from x, y, y’
  - where SHA256(x) = y, SHA256(x’) = y’

How can we use this?

- Hash-based Message Authentication Code (HMAC)
  - H = a hash function (e.g., SHA256)
  - HMAC = H(H(K) || M)
  - K: secret key
  - H(K): hash of the key
  - M: message or data

Encrypt Data with CBC

- CBC Data (32-byte blocks)

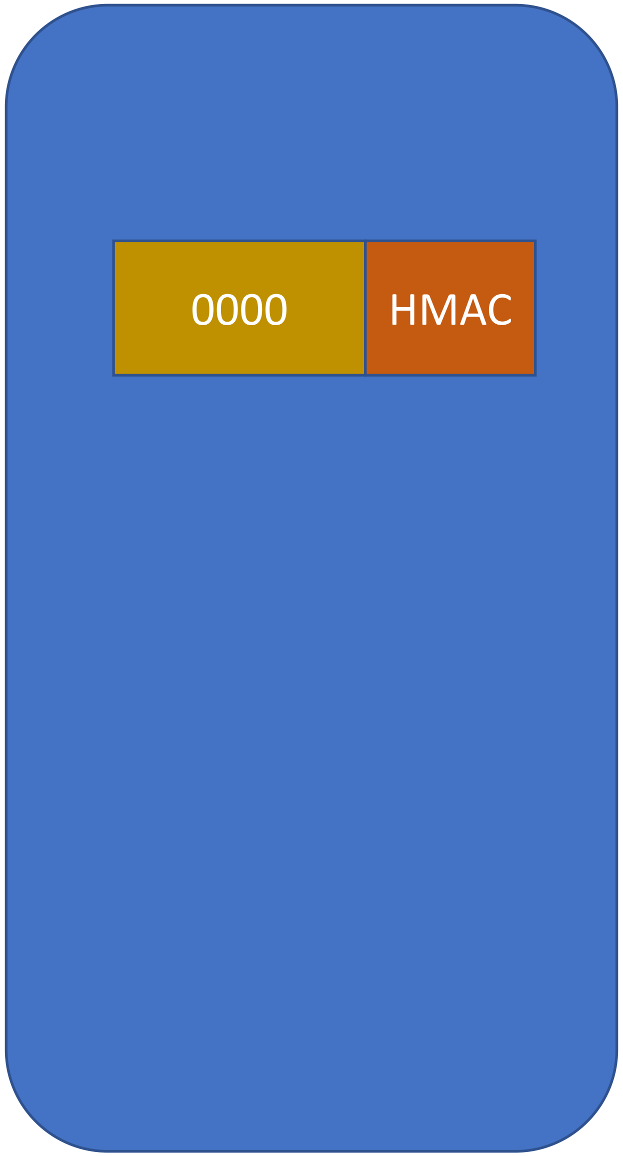
IV	Block 0	Block 1

- Suppose you have a hash key = ‘asdf’
  - HMAC = SHA256(SHA256(‘asdf’) || encrypted_data)
  - = `7624e1f89ce009f8ec7e6e39781a42c0a27fa38f94db4f05f78b0f301007e06a`

IV	Block 0	Block 1	HMAC (key || IV+Block0+Block1)

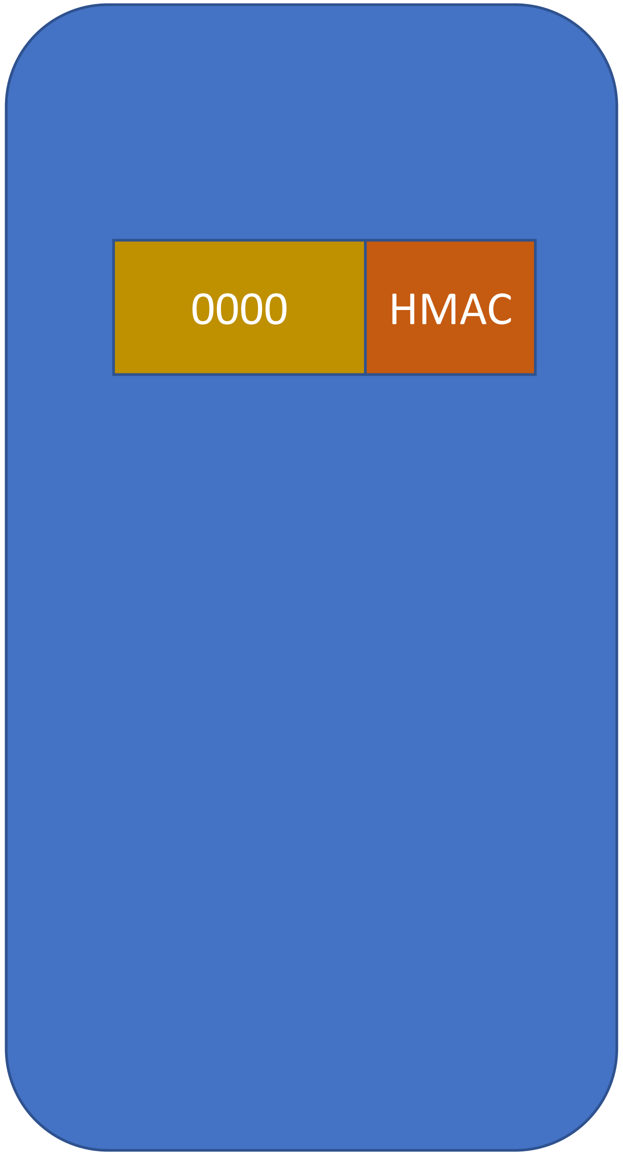
Workflow

I encrypt data
& added HMAC!
HMAC(key||0000)



Workflow

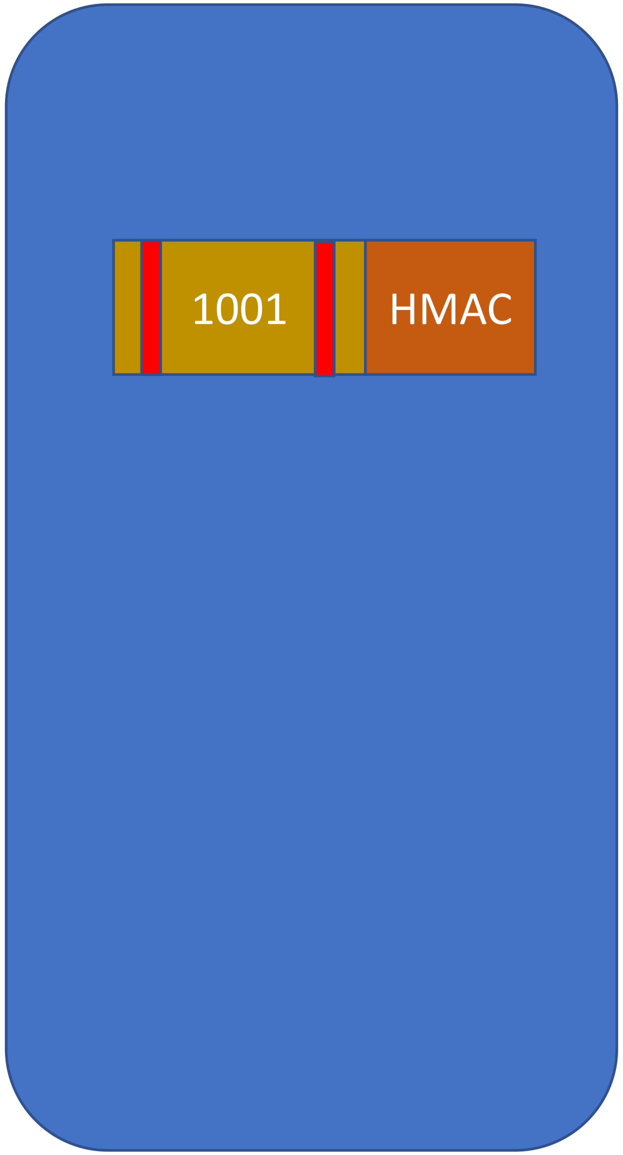
I encrypt data
& added HMAC!
HMAC(key||0000)



Edit data…

Workflow

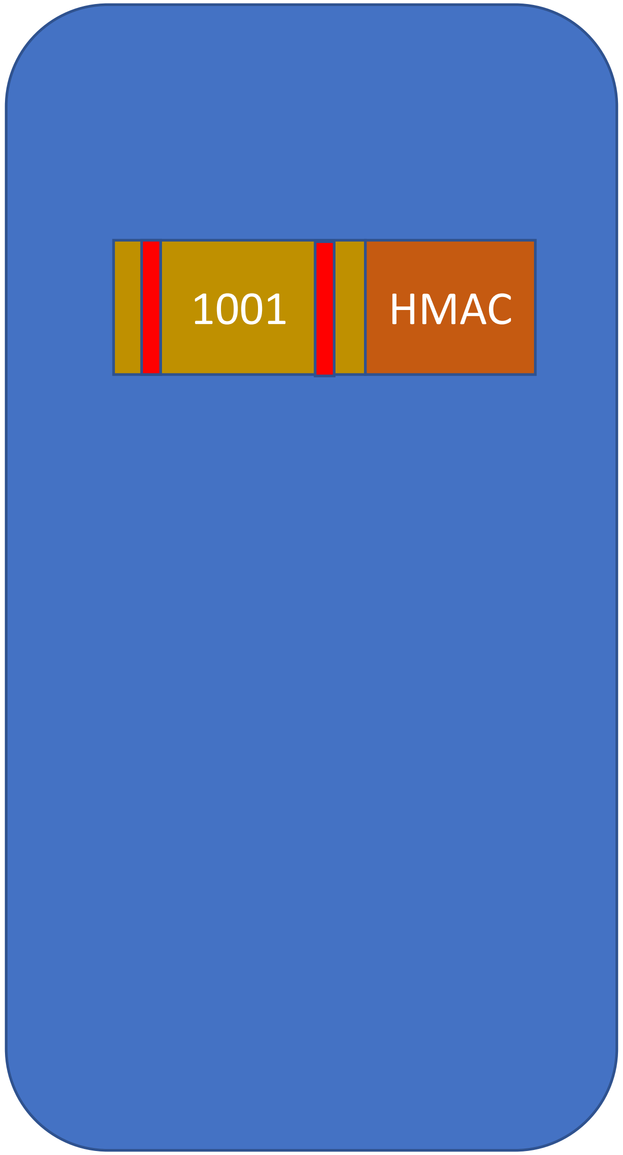
I encrypt data
& added HMAC!
HMAC(K||0000)



Edit data…

Workflow

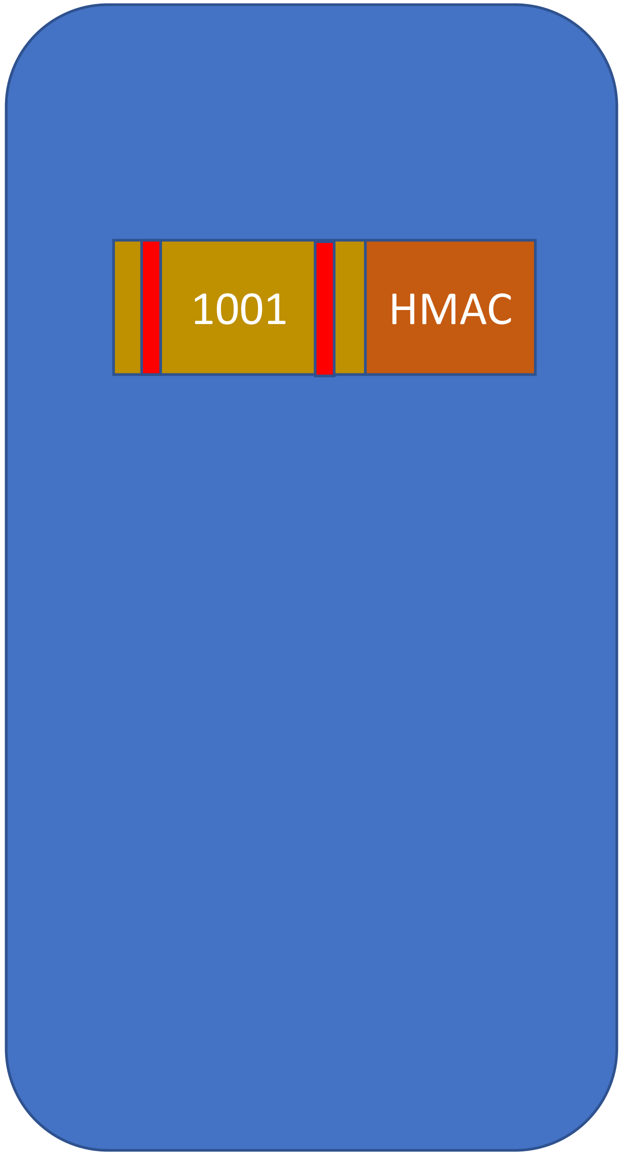
Want to check if
H(K||Data) = HMAC



Edit data...

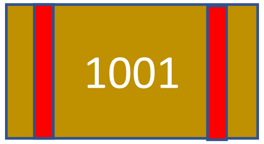
Workflow

I encrypt data
& added HMAC!
HMAC(K||0000)



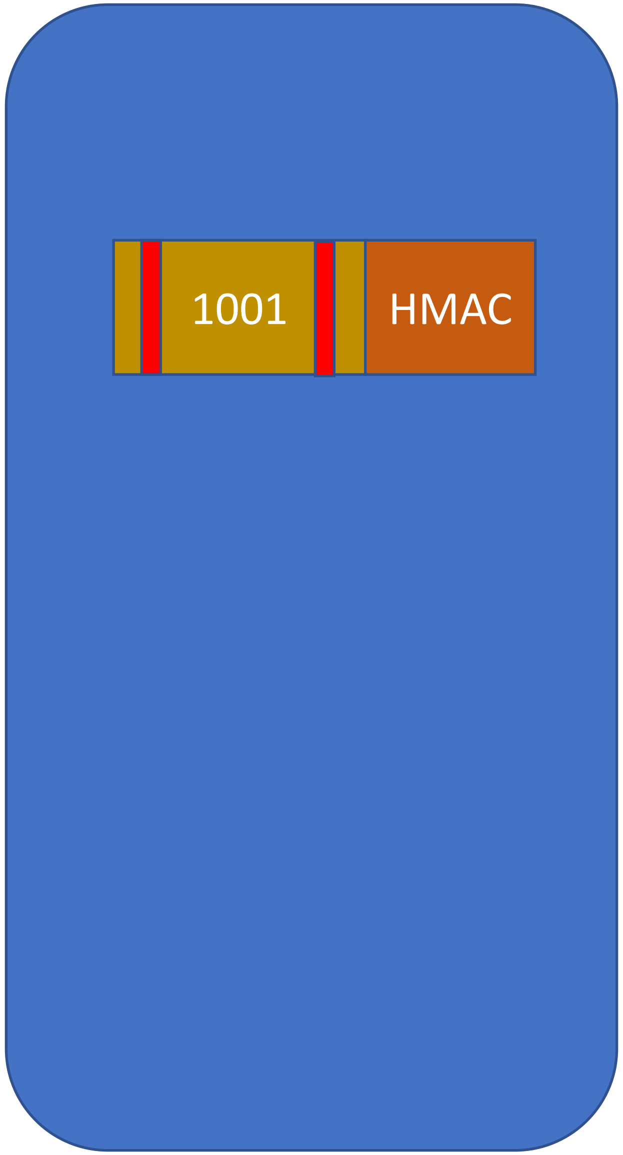
Edit data…

Want to check if
H(K||Data) = HMAC



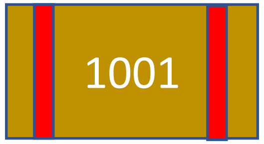
H(K || 1001) !=
H(K || 0000)

Workflow



Edit data…

Want to check if
H(K||Data) = HMAC

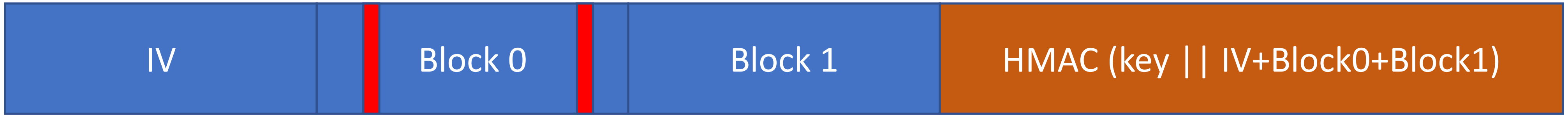


H(K || 1001) !=
H(K || 0000)

Reject!

Decrypt Data with CBC

- Suppose you have a hash key = ‘asdf’
  - HMAC = SHA256(SHA256(‘asdf’) || encrypted_data)
  - = `7624e1f89ce009f8ec7e6e39781a42c0a27fa38f94db4f05f78b0f301007e06a`
- Suppose the attacker changed the encrypted_data



  - HMAC = SHA256(SHA256(‘asdf’) || **encrypted_data**)
  - = `389205904d6c7bb83fc676513911226f2be25bf1465616bb9b29587100ab1414`
- Mismatch with HMAC!

Cannot edit data because we have HMAC

- Then, can attacker edit HMAC to match that to the edited ciphertext?

- HMAC = SHA256(SHA256('key') || edited_data)

- Attackers don't know the key
  - That's why we need to use key to SHA256.
  - Otherwise, anyone can generate valid MAC!

- Even they know SHA256(SHA256('key')|| encrypted_data)
  - They cannot generate a valid HMAC
  - They cannot correlate that value from this one…

Summary

- Block Cipher modes lets attacker play with ciphertext freely
  - They cannot be secure as we proved in challenges
- That's because Block Cipher protects only the data confidentiality
- Data Integrity left unprotected
- To protect data integrity, we can use cryptographic hash function
  - One way, it is hard to find an inverse of the result…
- HMAC, running cryptographic hash function with key on the data can protect data integrity…

Summary

- Encrypt-then-MAC

Encrypted data	HMAC: H(H(key)||Encrypted data)

- This is the only secure composition of using MAC with Encrypted data

- You must
  - Encrypt the data, and supply the entire encrypted data to HMAC
- No MAC-then-encrypt
  - Cryptanalysis exists (proven to be insecure)

Public (Asymmetric) Key Cryptography

- There is a scheme that we use different key to encryption and decryption
  - Why is it important? We will discuss this later about the ‘key exchange’

- RSA (Rivest, Shamir, Adleman)
  - A public-key cryptography algorithm
  - Based on the difficulty of prime factorization
    - i.e., if the prime factorization of a large prime number is difficult, then the cryptography scheme is secure
  - Can be used for digital signature

How RSA Works?

- Choose two large prime number, p and q
- $N = pq$
- $\phi = (p-1)(q-1)$

- Choose public key, say, $e = 65537$ (a prime), that is coprime to ϕ
- Find $de == 1 \pmod{\phi}$
  - d can be efficiently be computed if you know ϕ
  - Blue: public key, Red: private key
  - Attackers don't know ϕ, to know ϕ, you need to factor N

RSA Encryption

- Public key: e, N
- Message: M

$$M^e \bmod N$$

RSA Decryption

- Private key: d
- Public key: e N
- Ciphertext = C = M^e
- ed = 1

$$C^d \bmod N$$

$$(M^e)^d \bmod N$$

$$M^{ed} \bmod N$$

$$M \bmod N$$

In reality: RSA-4096

N

```
>>> n
94300162313076688501487627887743414045960390714020238212657873381998561681681396483075239596099727423614481972904672187684996447088678082779896359977776138389400094618
93811971402397228611383901773325321259010777553654865622984724944010604379009841301441764487806140346612303663579718455548165265742251289534980309219758481925957858787
99446168652865946938887547013421958335603541458898859985233102756405213301150045330555227176327316853262195678419436714942441571041724570176803145478446833917315101830
88856681940225944856271324669491185023754645727393394123350591112266076929457503053224635114890484540850553592509484360925706475886219500022922117482666285104831675845
83315533957568315104232320670250600708758347303059147821341336205419089515531617207836027717015263175059127264155564477809166344370523152038595667063337410819626147392
61504146573604212524025625329042730131363602682044377326795554520903135271401816609380989125787711356372203148662219805667048555875256480930486742228216374620641072794
39035295803547382839528902460696618996010706014197280097861310233823234888621192701394780193796900301510396063855789518617879808500828987759386908963639259712107524427
2314777803224755716476436654020264220108971589745725860770832311
```

pq = N

```
>>> p
32068656324685270630293083389512032043113021564958908104074082522961856175155506846871162451521328939255338658509282518527589856290667522356720088237485538660191295524
66763080165651305435511312859214781036375072132558927069941823437366007834867296636112996194202733617438183660982840127747653230903017526198432501279953198838940252489
07397379843971080938071066796458439488172567320447317484321566200567720344593323665558811276211955099547645761253947551466759960870349272433501174519655039023335818685
46315844543573625502694906678244707907306961176386182217799611306289079541727783724838441308692085353299440403808273
>>> q
29405710472654850958671060473587380272033499777349031897972774267355140300355329654820633532129222264029138740441499983175231104660614751766804322996494698305495888552
75830646534910098483628642910172110351604197235439018569363033804440884203852197669902443119162881776394993390408980913691022609261096791810408703245340903925873967271
31915873792822333318309536841669683057109843537055633086239079548671678418825449122417399716632535551668617851522599535745800792690727833156308278625231616345078778788
62534221887593666478665101149415792855605030513596595792804362412144174514663486222431955289564169364164960140372807
>>> p*q == n
True
```

e,d

```
>>> e
65537
>>> d
18849384718575836845896027058446964676474069889583977304207060764212294704213288583979822675876320692881116222058550218329698543675506704371830299154811288476755744686
33122309922202781555596519725201973060912784929930859509301614305911758508400276501043488636804559844677375802873099344914795687645505468558344783111507028292873460032
08458501447312875244509618473165199111897193846400889052085900271738228698980203454999061232737047023478870918523805193489875712660735033176924808974589217149348121427
35615339936478018182357360775902688920932997528554152051755386172249044170100756671761641820728219898456478814438777554613519848613000196946332897123652219249131582620
18419600479189059635429627251759483101314923287035484839408409780479225523102668224739334748234253874639242903082590778667288728135476436491806847150495168056124486222
04967505131689556757368561851417851628405515960727363119875634155780743786262756894688981786624749458184135230246829545964509695021029098701475660700811758053804335333
34248118321456892350272682042203653282995523452563819502703960150673047768164870581311920830986731370034223814398610511624078011843818974900250458094933776149648209294
864556932979130604479458312909282114054641455801867990797517633
```

In reality: RSA-4096

- Encryption

```
>>> m = 12345
>>> c = pow(m, e, n) # m ** e % n
>>> c
4437847063777873523960861181094228237820799745014796806317435352463184765865908695370736550887746388699345004929851287449210030488889432985334867098950587742373724841 7
8040079575874499290413374847062808224508941879682031434528457814176924662901174674588647695046357307850784382343408121122220430499568600055483447589046769763784724684 4
8632211687399983213712399982644387191965538396653162540063690301350947217017078364516165401139964366064459930930412289059144895011174628269542079488232243178919635809 3
9369824300739839583644448529240916312389333538226847670958792010806402061192128197464732116736141126317643655098310631571251255038145886608173936142768624376747440567 1
6969212531466451790154616008582626051573044131596636322530763889773664859797300591802077382373834580062670252137693091857292315461595441787257177342422993355220095280 4
2294247648106010082123894144581725049323951764142107487267859334789004372898889993104888145686482685014075098613704247499837173115420999867541470766944268437257194232 1
2253130274983980499647943007322662010113972847205263805344645817600350844304448669787454473077578613161392937377083287156455026552907697247941471786426217609201720684 4
55612835681708188451830976789314222414537395183724797676212931 11
```

- Decryption

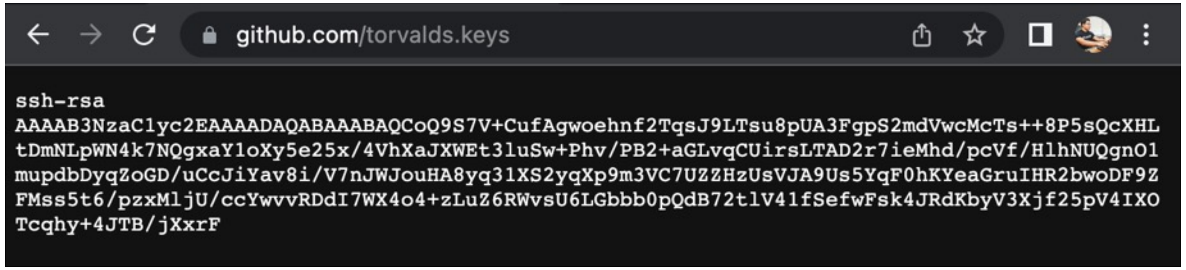
```
>>> _c = pow(c, d, n) # c ** d % n == m ** (de) % n == m % n == m
>>> _c
12345
>>>
```

Public Key Cryptography

- We can use separate key for encryption and decryption
  - Encryption key: public key (e, N)
  - Decryption key: private key (d)

- Attackers cannot guess the private key from the public key
  - In RSA, attacker must factor the prime number N = pq
  - In creating the key, we choose p and q as a big prime number
  - Factoring a multiplication of two big prime numbers is a difficult task

Characteristics

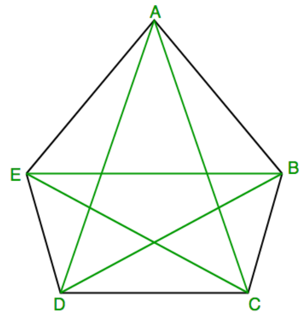


- We use a public key for encryption
  - We can publicize this key
  - If you publish your key, anyone who can access that can encrypt message
    - (e, N) is public, $m^e \bmod N$!
  - Only the holder of the private key can decrypt the message
    - d is private, $m^{ed} == m^1 == m \pmod N$

- Why is this important?
  - Let's talk about the key exchange problem

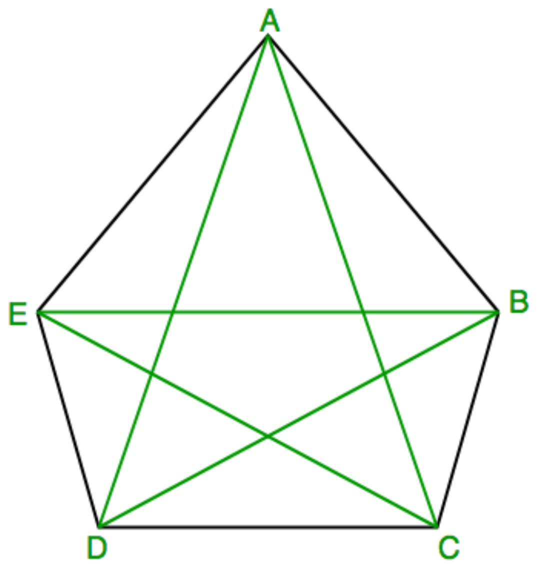
Key Exchange

- Suppose we have 5 people, A, B, C, D, E
  - How many keys do we need to have to make them communicate securely?
  - E.g., if A talks to B, C or others must not see the message
  - But anyone should be able to talk to anyone…
- A block cipher
  - We need 1 key for A and B can talk securely
- How many keys do we need to let them communicate securely?
  - A-B, A-C, A-D, A-E
  - B-C, B-D, B-E
  - C-D, C-E
  - D-E
  - 10 keys (5*4/2 = 10)



Symmetric Key Cryptography

- Encryption and the decryption operations are using the same key
  - Block Cipher – encryption key == decryption key
  - You cannot share that other than 2 people
- Key exchange complexity
  - We need 1 key per each pair of people
  - N (N – 1) / 2
  - $O(N^2)$

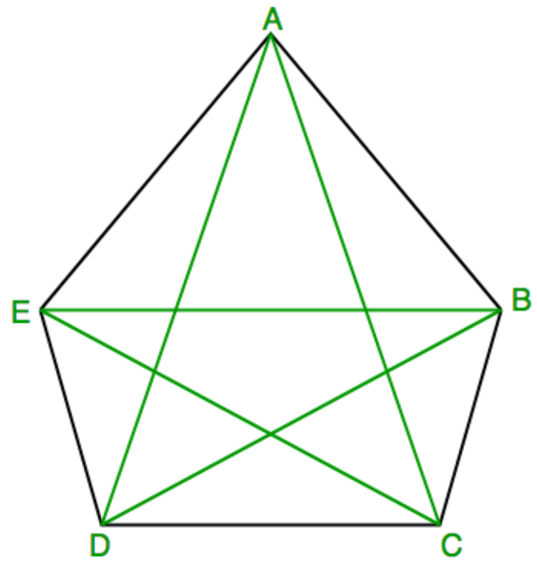


Asymmetric Key Cryptography

- Can we use a different key for the encryption and decryption?
  - K = k1, k2
  - Enc(k1, M) = C, Dec(k2, C) = M?
- Preferably, can we publish the encryption key to public?
  - While keeping the decryption key secret
- Then we need O(N) keys
  - Each member's public key, that's it.

Why O(N)?

- We need $O(N^2)$ keys for symmetric encryption



Why O(N)?

- Suppose each will generate public and private key

- Public_A, Private_A
- Public_B, Private_B
- Public_C, Private_C
- Public_D, Private_D
- Public_E, Private_E

Why O(N)?

- Each will have their own private key, and then,
  - publish all their public keys
- A: Private_A
- B: Private_B
- C: Private_C
- D: Private_D
- E: Private_E
- Public keys: Public_A, Public_B, Public_C, Public_D, Public_E

Can A Send an Encrypted message to B?

- Can A send an encrypted message to B?
  - Yes, encrypt data using Public_B; only B (holder of Private_B) can decrypt it
- Can C send an encrypted message to E?
  - Yes, encrypt data using Public_E; only E (holder of Private_E) can decrypt it

- Can X send an encrypted message to Y?
  - Yes, if X knows the public key of Y

- We only need to know the receiver's public key
  - Sender does not matter, that's why we have O(N)
  - Suppose we have N = 200, we need 19900 keys in symmetric, and we need 400 keys for asymmetric

RSA: Digital Signature

- RSA can be used as a digital signature scheme
- What is that?

- In RSA, encryption is applying the public exponent to the message
  - $M^e \bmod N$
- In RSA, decryption is applying the private exponent to the message
  - $C^d \bmod N$

RSA: What will be the meaning of private encrypt?

- Suppose A encrypts the following message with her private key
  - "I would like to donate $100 to OSU if I get A from CS 370"
- M = 53151406333611257093956293411584759988053228938724427101859883089254119711739486837784167497839141764612450119856395995171455585519613744
- $C = m^d \bmod N$

RSA: What will be the meaning of private encrypt?

- M = 531514063336112570939562934115847599880532289387244271018598830892541197117394868377841674978391417646124501198563959951714555855196137 44
- $C = m^d \bmod N$
- Anyone can have e. That means, anyone can decrypt C
  - $C^e == m^{de} == m^1 == m \pmod N$

RSA: What will be the meaning of private encrypt?

- M = 53151406333611257093956293411584759988053228938724427101859883089254119711739486837784167497839141764612450119856395995171455585519613744
- $C = m^d \bmod N$
- Anyone can have e. That means, anyone can decrypt C
  - $C^e == m^{de} == m^1 == m \pmod N$
  - m = 53151406333611257093956293411584759988053228938724427101859883089254119711739486837784167497839141764612450119856395995171455585519613744
  - "I would like to donate $100 to OSU if I get A from CS 370"

RSA: What will be the meaning of privat

- M =
  5315140
  1859883
  8563959

We can verify that the encrypted content C contains
The ciphertext that only the holder of private key can generate.
We all have public key, and if that is decrypted to
"I would like to donate $100 to OSU if I get A from CS 370”,
then, we know that the holder of private key ‘endorsed it’

- C = m^d mod N
- Anyone can have e. That means, anyone can decrypt C
  - $C^e == m^{de} == m^1 == m \pmod N$
  - m =
    5315140633361125709395629341158475998805322893872442710185988308925411971173948683778416749783914176461245011985639599517145 5585519613744
  - "I would like to donate $100 to OSU if I get A from CS 370"

RSA: private_encrypt

- RSA Encryption using the private key is so-called as ‘Signing’
- Why?
  - The ciphertext will be decrypted as a plaintext using the public key
    - Anyone can decrypt!
  - But the ciphertext can only be generated with the private key
    - Only the private key owner can generate it!

- Implication
  - Holder of the private key generated a ciphertext message of message M
  - M is signed, endorsed by the holder’s private key
  - (Because it can only be generated with the private key)

RSA Summary

- Public/Private key Scheme
  - We can publish the public key – encryption key
  - We must hide the private key – decryption key
- Based on the difficulty of prime factorization
  - You cannot correlate the private key from the public key unless
  - You can factor a big number (a multiple of 2 big prime numbers)
- Anyone can encrypt message to the private key owner
  - Enc(public_key, message)
- Only the private key owner can decrypt message
  - Dec(private_key, encrypted_message)

RSA Summary

- Encryption with private key could be a 'digital-signature'
  - Signed_message = Enc(private_key, message)
  - Message = Dec(public_key, signed_message)

- The correctly decrypted message using public key means that the private key holder have endorsed ('encrypted') the data
  - Anyone can verify this using the public key

Symmetric vs. Asymmetric

- Can we use symmetric key as digital signature?
  - A-B, A-C, A-D, A-E
  - B-C, B-D, B-E
  - C-D, C-E
  - D-E

- If a message was encrypted with key D-E
  - Then either D or E generated the message -> ambiguity
  - Only D or E can verify that -> result is not public
  - They must leak the key D-E for verification -> key need to be leaked to verify

Symmetric vs. Asymmetric

- In asymmetric key scheme
  - Public_A, Private_A
  - Public_B, Private_B
  - Public_C, Private_C
  - Public_D, Private_D
  - Public_E, Private_E

- If a message was encrypted with Private_D
  - We can decrypt the data using Public_D -> generated by D -> not ambiguous
  - Anyone can verify this -> result is public
  - Does not need to leak the private key -> don’t leak the key for verification