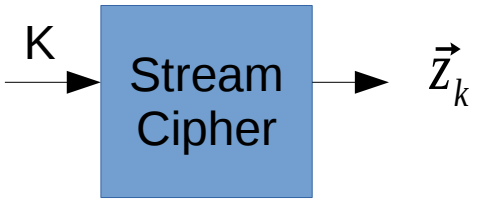
Symmetric Cryptography

Encryption/Decryption
Hashing

Stream Ciphers
Using Pseudo Random Functions



Seeded by a key K the stream cipher generates a random bit-stream $\bar{z}$.
A stream of plain-text bits $\bar{p}$ is XORed with the pseudo-random stream to obtain the cipher text stream $\bar{c}$

$$\vec{c} = \vec{p} \oplus \vec{z}_k \text{ and } \vec{p} = \vec{c} \oplus \vec{z}_k$$

$\vec{c}$ cipher text stream

$\vec{p}$ plain text stream

$\vec{z}_k$ key stream derived from seed K

The same stream generator
(using the same seed)
Is used for encryption and decryption

Stream Ciphers: Need for IV

$$K \rightarrow \bar{z}_k \text{ (Stream Generation)}$$

$$\bar{c}_i = \bar{p}_i \oplus \bar{z}_k$$

$$\bar{c}_j = \bar{p}_j \oplus \bar{z}_k$$

Attacker has access to $\bar{c}_i$ and $\bar{c}_j$

$$\bar{c}_i \oplus \bar{c}_j = (\bar{p}_i \oplus \bar{z}_k) \oplus (\bar{p}_j \oplus \bar{z}_k) = \bar{p}_i \oplus \bar{p}_j$$

- XORing two cipher-texts encrypted using the **same seed** results in XOR of corresponding plain-texts
- Redundancy in plain-text structure can be easily used to determine both plain-texts (and the key stream)
- Never reuse seed? Impractical (key setup is expensive)
- Extend seed using an initial value (IV) which can be sent in the clear
- K'=K||IV used as the seed (|| denotes concatenation)
- **Never reuse IV**

Block Ciphers

- C=E(P,K)
- P=D(C,K)
- E() and D() are *algorithms*
- P is a block of “plain text” (m bits)
- C is the corresponding “cipher text” (also m bits)
- K is the key (k bits long)
- (k,m) block cipher – k-bit keysize, m-bit blocksize
- **(m+k)-bit input, m-bit output**

Desired Properties

- The most efficient attack should be the brute-force attack (attack complexity depends only on key length)
- Knowledge of *any* number of plain-cipher text pairs, still does not reveal *any* information regarding *any* bit of the key.
  - Even if attacker has the ability to *choose* plain-text/cipher-text
  - Think of the cipher as encryption/decryption black boxes (with key inside the boxes). Attacker with access to the black-boxes can input any plain text to encryption block to get cipher text, and can input any cipher text to get corresponding plain text
    - The attacker should still not be able to determine the key

Confusion and Diffusion

- Confusion is “making the relationship between the cipher-text and the symmetric key as complex and involved as possible.”
- Diffusion refers to “dissipating the statistical structure of plain-text over bulk of cipher-text.”
- A block cipher with good confusion and diffusion properties will meet the desired goals

Confusion and Diffusion: Another Perspective

$$C = E(P, K), P = D(C, K)$$

$m + k$ input bits

m output bits

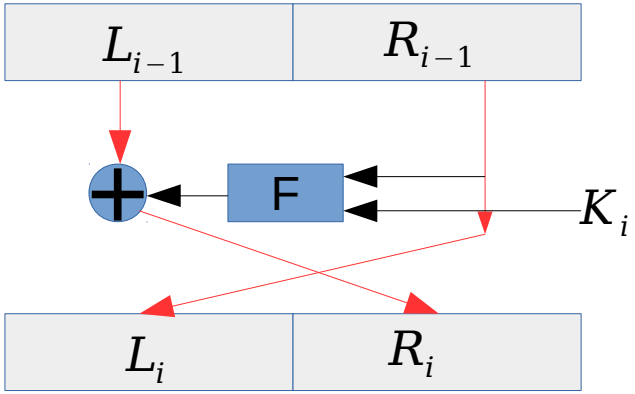
Let $p_{ij}, 1 \leq i \leq m + k, 1 \leq j \leq m$ be the probability that flipping input bit i flips output bit j

For a good cipher we desire $p_{ij} \approx 0.5 \forall i, j$

Block Cipher Construction

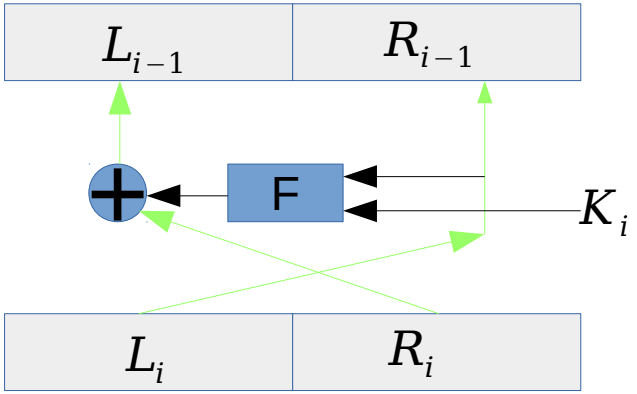
- Desire thorough mangling of plain-text and key
  - But, we also need to reverse the process
- Non reversible approaches can achieve better confusion and diffusion
  - Can we use non-reversible components in a reversible block cipher?
  - Feistel Structure.

Block Cipher Construction: Feistel Structure



Encryption

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$



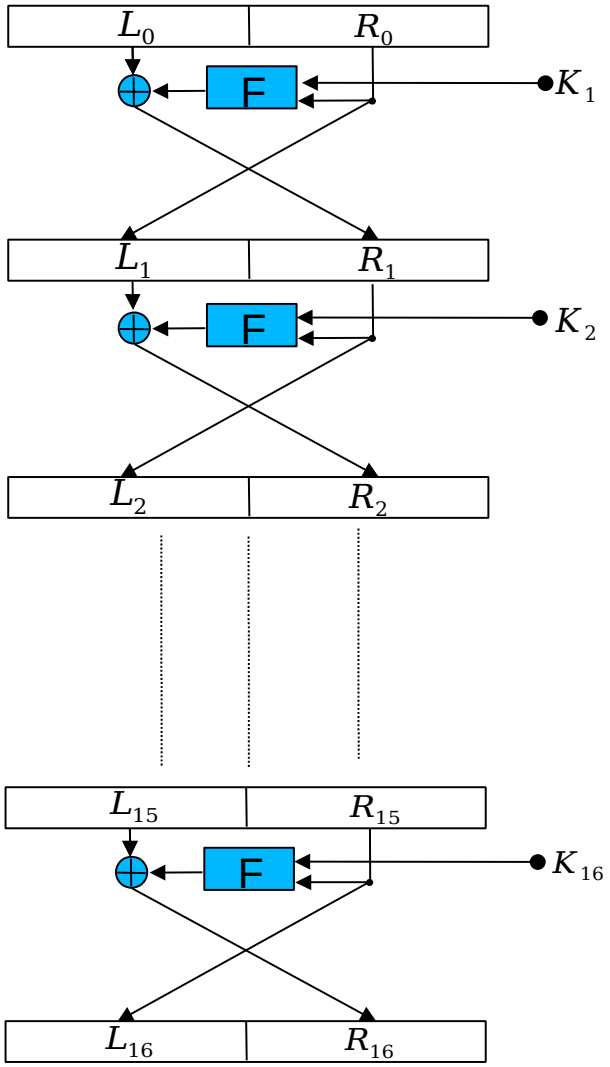
Decryption

$$R_{i-1} = L_i$$
$$L_{i-1} = R_i \oplus F(L_i, K_i)$$

- Block ciphers constructed from repeated Feistel rounds
- Plain-text block split into 2 halves (left and right)
- Each round has the same F block, but a different round key
- Trivially invertible (only red arrows flipped for decryption)
- F() need **not** be invertible for the block cipher to be invertible!
  - F() can be made as complex/non-linear as desired
- Example Feistel cipher: DES (Data Encryption Standard)

DES Uses 16 F-Rounds

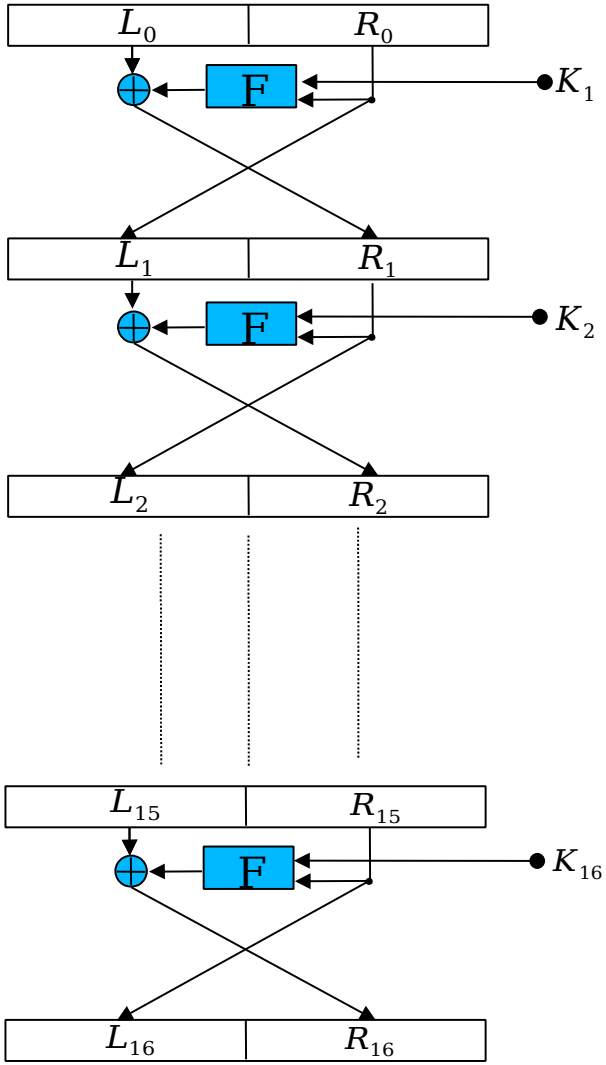
$P = L_0 \| R_0$



$K \rightarrow K_1 \cdots K_{16}$ (round keys)

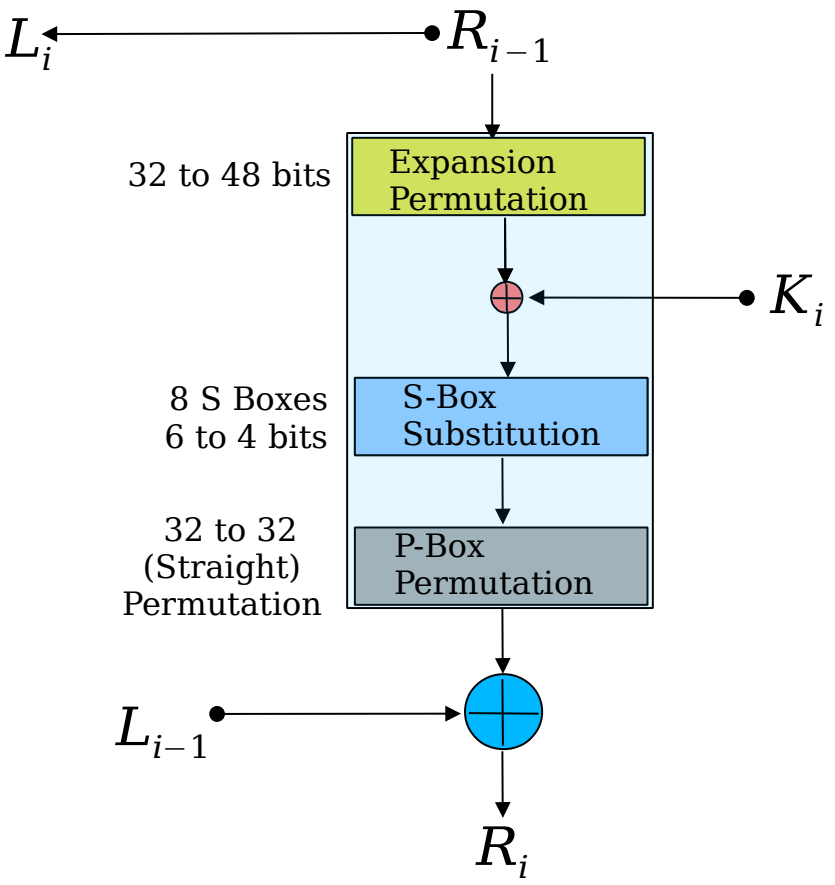
$C = L_{16} \| R_{16}$

F-Block in DES

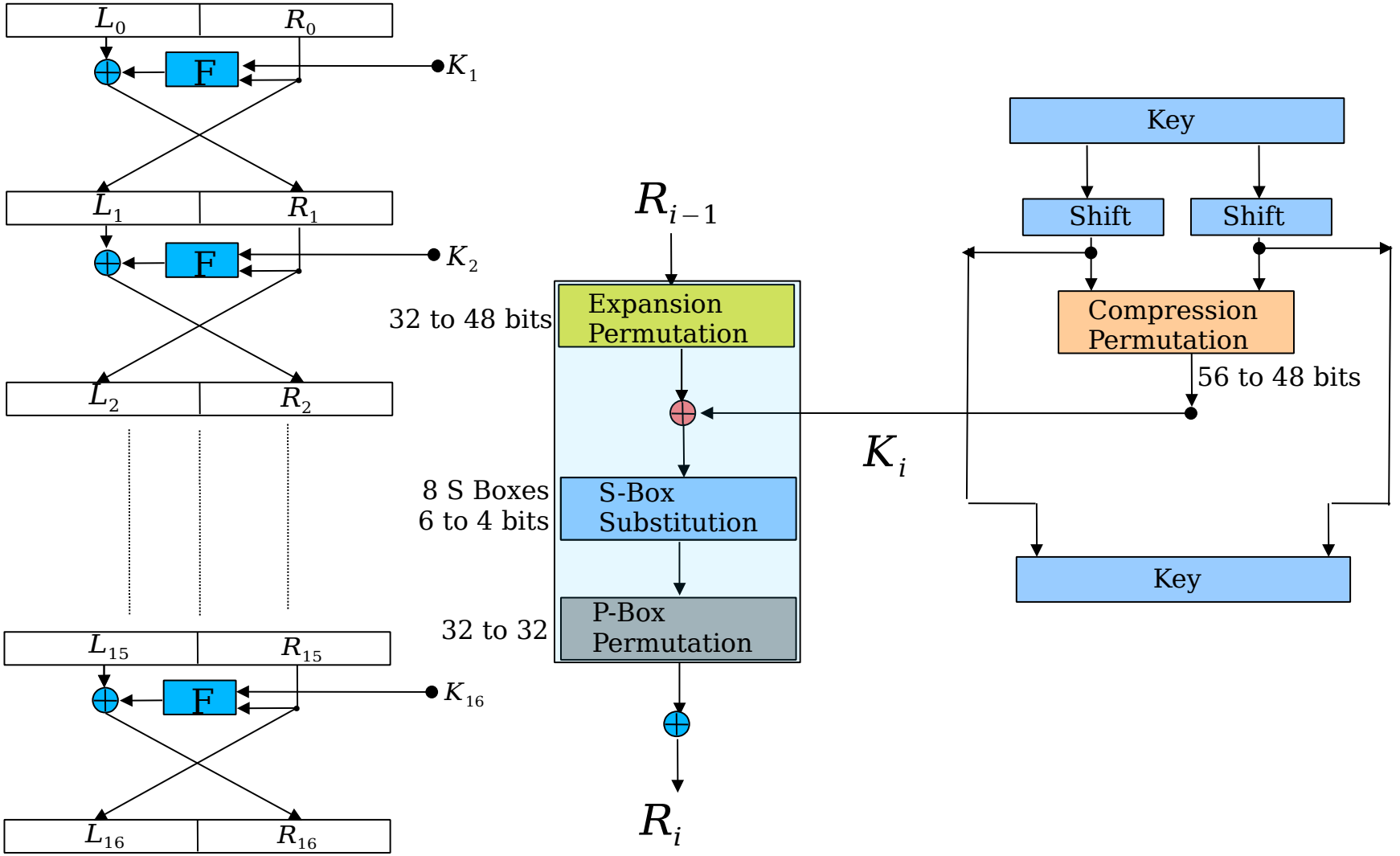


$$L_i = R_{i-1}$$

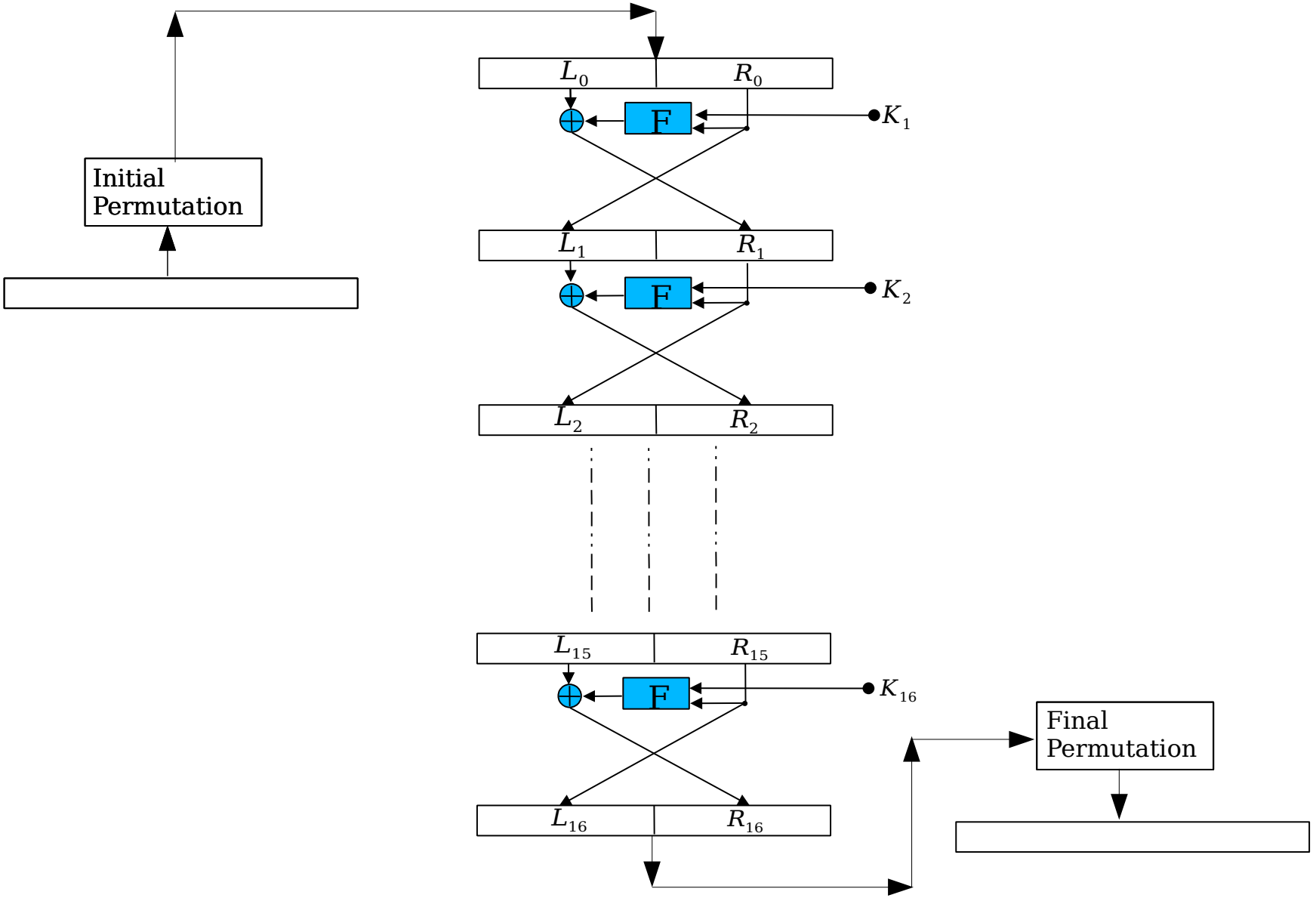
$$R_i = L_{i-1} \oplus F\left(R_{i-1}, K_i\right)$$



DES – Round Key Generation



DES – Initial and Final Permutation



DES – Algorithmic Overview

$$T - \textit{64 bit input}$$

$$K - \textit{64 bit key with parity - leads to } K_0 - \textit{56 bit key}$$

$$K_1, K_2, \ldots, K_{16} \textit{ (generated by round key generation)}$$

$$T_1 = IP(T) \textit{ (Initial Permutation)}$$

$$(L_0, R_0) = T_1 \textit{ (split into two 32 bit quantities)}$$

$$(L_1, R_1) = (R_0, L_0 \oplus F(R_0, K_1))$$

$$(L_2, R_2) = (R_1, L_1 \oplus F(R_1, K_2))$$

$$\vdots$$

$$(L_{16}, R_{16}) = (R_{15}, L_{15} \oplus F(R_{15}, K_{16}))$$

$$C_1 = (R_{16}, L_{16}) \textit{ (swapping)}$$

$$C = FP(C_1) \textit{ (Final Permutation)}$$

IP and FP

Initial Permutation

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Final Permutation

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

DES – Round Function

$$R_1 = F(R_{0,k})$$

R_0 – *32 bit round input*

k – *48 bit round key*

$X = E(R_0)$ *(Expansion Permutation)*

$X_1 = X \oplus k$ *(XOR with round key)*

$X_2 = S(X_1)$ *(apply S-Box substitution - output 32 bits)*

$R_1 = P(X_2)$ *(apply round permutation)*

E – Expansion Permutation

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

P – Round Permutation

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

DES – S-Boxes

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

$X \rightarrow Y$ *48 bit to 32-bit*

$S_1 \cdots S_8$ *8 S-Boxes*

$X = X_1 \| X_2 \| \cdots \| X_8$ *(input)*

Each S-box converts 6-bits to 4-bits

$Y = (S_1(X_1) \| \cdots \| S_1(X_8))$ *(output)*

Each S-Box has 4 rows and 16 columns

Each row is a permutation of 0 to 15

$b_1 b_6$ *of* X_i *chooses the row of* S_i

$b_2 b_3 b_5 b_4$ *of* X_i *chooses the column of* S_i

DES – Key Schedule

K 64 bit key

r_i left shifts in round i

$r_i = 1$ for $i = 1,2,9,16$

$r_1 = 2$ for all other i

$K_1 = PC_1(K)$ (Permuted Choice)

(Effective Key length is 56)

$$(C_0, D_0) = K_1$$

$$(C_1, D_1) = (r_1(C_0), r_1(D_0))$$

$k_1 = CP(C_1 D_1)$ (Compression Permutation)

$$(C_2, D_2) = (r_2(C_1), r_2(D_1))$$

$$k_2 = CP(C_2, D_2)$$

$$\vdots$$

$$k_{16} = CP(C_{16}, D_{16})$$

PC

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

CP

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

DES – At a glance

T 64 bit input

K_0 64 bit key - leads to $K-$ 56 bit key

$K_1, K_2, \ldots, K_{16}$ generated by round key generation

$T_1 = IP(T)$ Initial Permutation

$(L_0, R_0) = T_1$ split into two 32 bit quantities

$$(L_1, R_1) = (R_0, L_0 \oplus F(R_0, K_1))$$
$$(L_2, R_2) = (R_1, L_1 \oplus F(R_1, K_2))$$
$$\vdots$$
$$(L_{16}, R_{16}) = (R_{15}, L_{15} \oplus F(R_{15}, K_{16}))$$

$C_1 = (R_{16}, L_{16})$ (swapping)

$C = FP(C_1)$ Final Permutation

Round function $R_1 = F(R_0, k)$

R_0 32 bit round input

k 48 bit round key

$X = E(R_0)$ Expansion Permutation

$X_1 = X \oplus k$ XOR with round key

$X_2 = S(X_1)$ apply S-Box substitution (output 32 bits)

$R_1 = P(X_2)$ apply round permutation

S-Box Function

X input - 48 bit data

$S_1 \cdots S_8$ 8 S-Boxes

$(X_1, X_2, \cdots, X_8)$ split X

$Y = (S_1(X_1), \cdots, S_1(X_8))$

Each S-Box has 4 rows and 16 columns

Each row is a permutation of 0 to 15

$b_1 b_6$ of X_i chooses the row of S_i

$b_2 b_3 b_5 b_4$ of X_i chooses the column of S_i

Key Schedule

K_0 64 bit key

r_i left shift in round i

$r_i = 1$ for $i = 1, 2, 9, 16$ and 2 for all other i

$K = PC(K_0) = (C_0, D_0)$

K is 56 bits

$$(C_1, D_1) = (r_1(C_0), r_1(D_0))$$
$$k_1 = CP(C_1, D_1)$$
$$(C_2, D_2) = (r_2(C_1), r_2(D_1))$$
$$k_2 = CP(C_2, D_2)$$
$$\vdots$$
$$k_{16} = CP(C_{16}, D_{16})$$

DES Description & History

https://en.wikipedia.org/wiki/Data_Encryption_Standard

Current standard for encryption is AES

AES (Advanced Encryption Standard) is not a Fiestel cipher

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

Encrypting Bulk Data

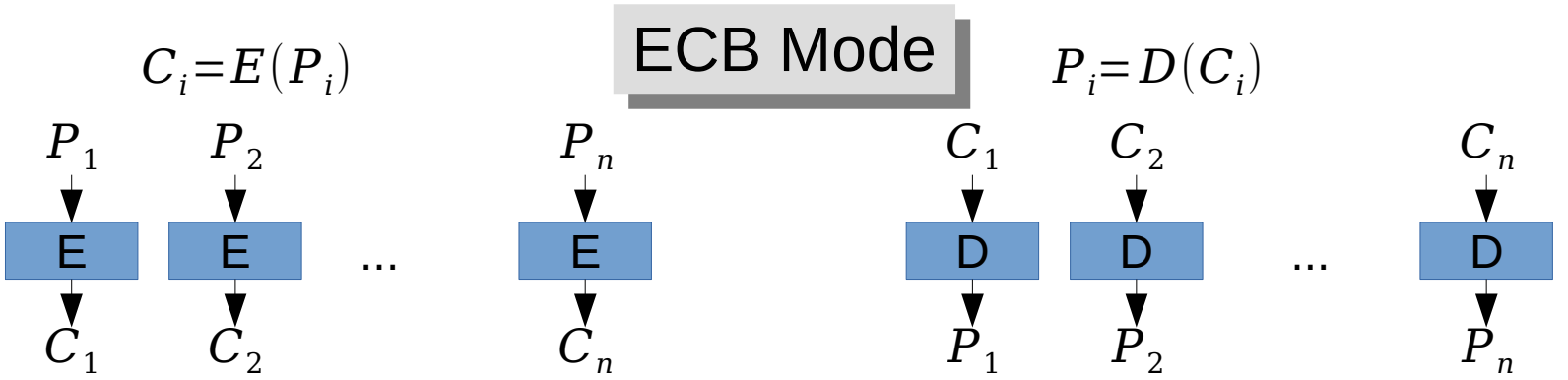
- For example, a file, or a packet
- Segment data into blocks of size m bits (block size)
- Encrypt each block using the same key
  - As key set-up is expensive
- Important Considerations
  - Encrypted file/packet should reveal as little information as possible regarding the contents of the file/packet
  - What happens if there is a transmission error?
    - Accidental error?
    - Deliberate error?
  - What happens to the encrypted blocks if one-bit of some input block is changed?

Block Cipher Modes

- Electronic Codebook (ECB)
- Cipher Block-chaining (CBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter mode (CTR)
- CBC and CFB modes are also used for **key based message authentication code (MAC)**

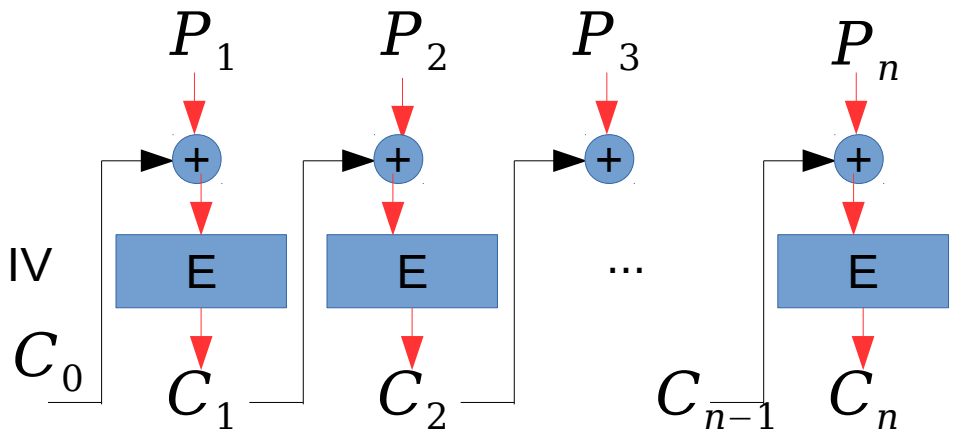
Key Based MAC

- CBC and CFB modes also used for key based message authentication code (MAC)
- For a message M (of any size)
  - *a=MAC(M,K)* is a MAC with key *K*
  - Size of *a* is the block-size *m*
  - If the sender and receiver share a key *K* sender can send message M along with MAC *a*
  - The receiver can verify that *a=MAC(M,K)* and thus be assured of
    - The integrity of the message *M*, and
    - The message *M* came from an entity with knowledge of key *K*



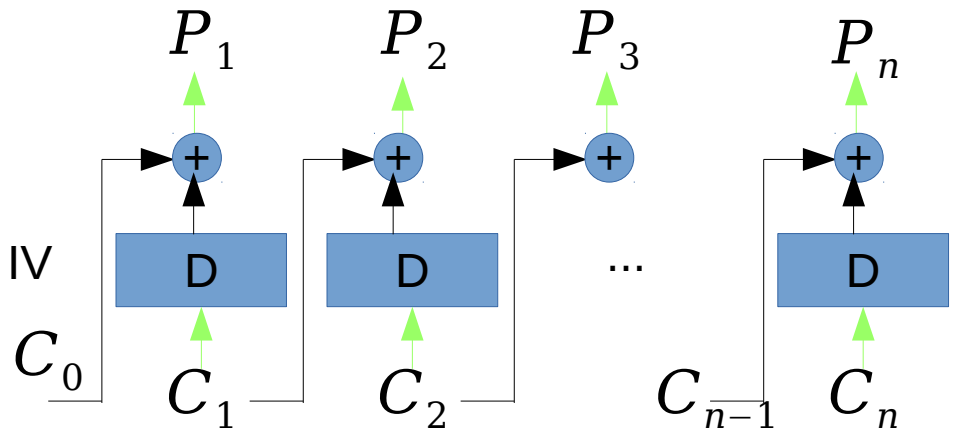
- Sender to receiver: n blocks C_1 to C_n
- Identical plain-text bocks produce identical cipher-text blocks
- This can reveal some information regarding the plain text
- Encryption/Decryption can be parallelized

CBC Mode

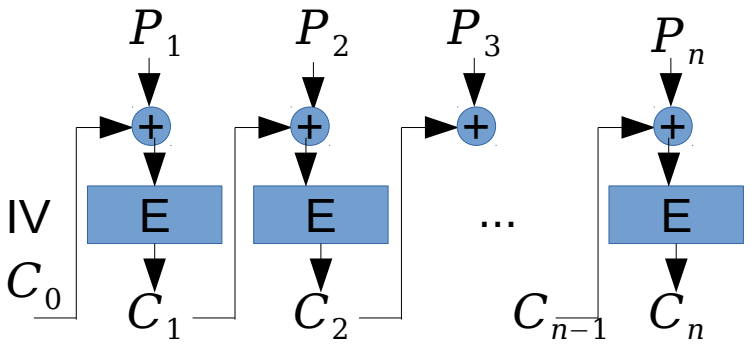


$$C_i = E(P_i \oplus C_{i-1}) \text{ where } C_0 = IV$$

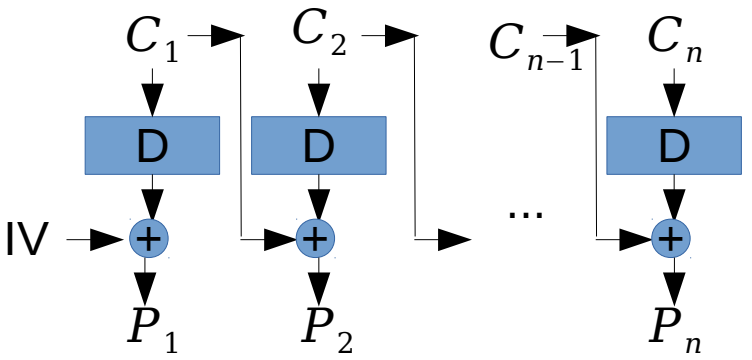
$$P_i = D(C_i) \oplus C_{i-1} \text{ where } C_0 = IV$$



$C_i = E(P_i \oplus C_{i-1})$ where $C_0 = IV$



$P_i = D(C_i) \oplus C_{i-1}$ where $C_0 = IV$

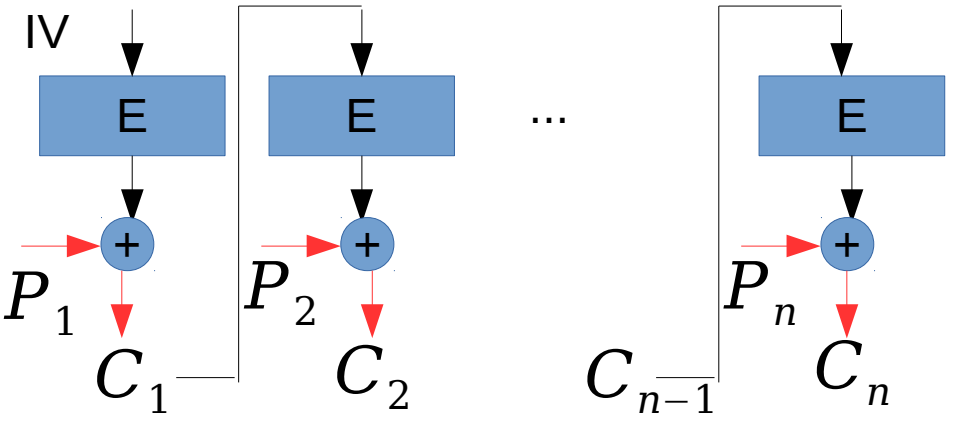


CBC Mode

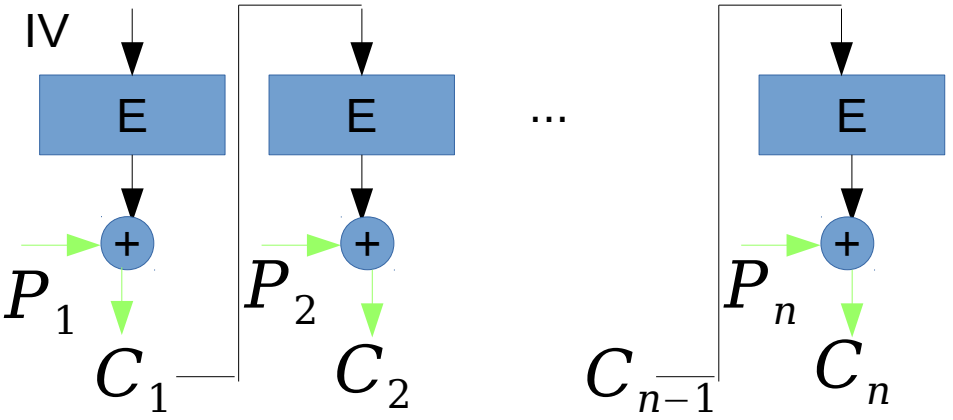
- Sender to Receiver: IV, and C_1 to C_n
- When used for MAC, IV, C_n is sent with plain text
- Tx error in C_k affects decryption of P_k and P_{k+1}
- Encryption/decryption can not be parallelized
- A change in any bit of any plain-text block will dramatically modify the all following cipher text blocks
  - Desirable property for MAC.
- What happens if a bit of the IV is modified in transit?
- IV should be encrypted in ECB mode (recommended)

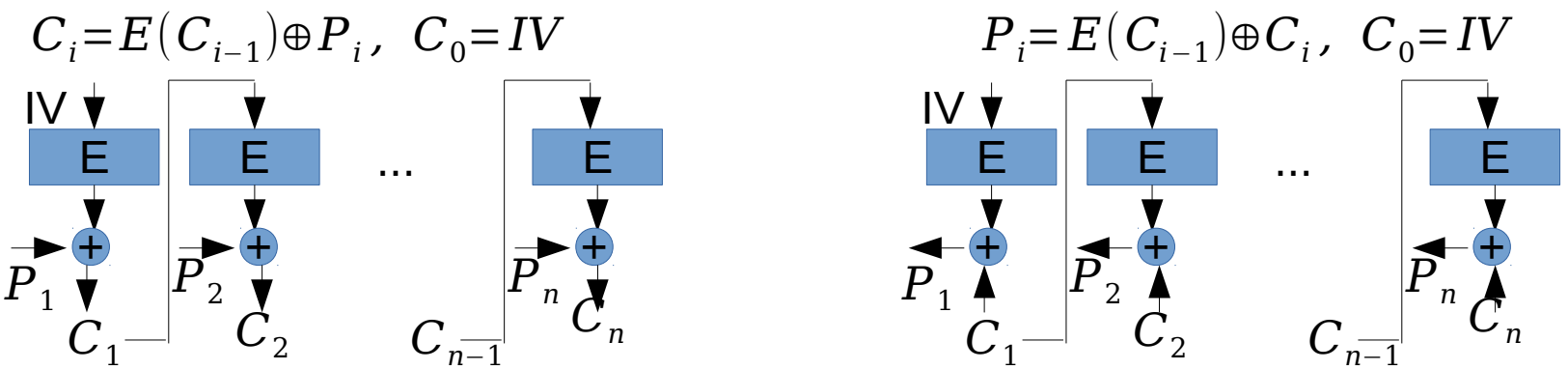
CFB Mode

$$C_i = E(C_{i-1}) \oplus P_i, \quad C_0 = IV$$



$$P_i = E(C_{i-1}) \oplus C_i, \quad C_0 = IV$$



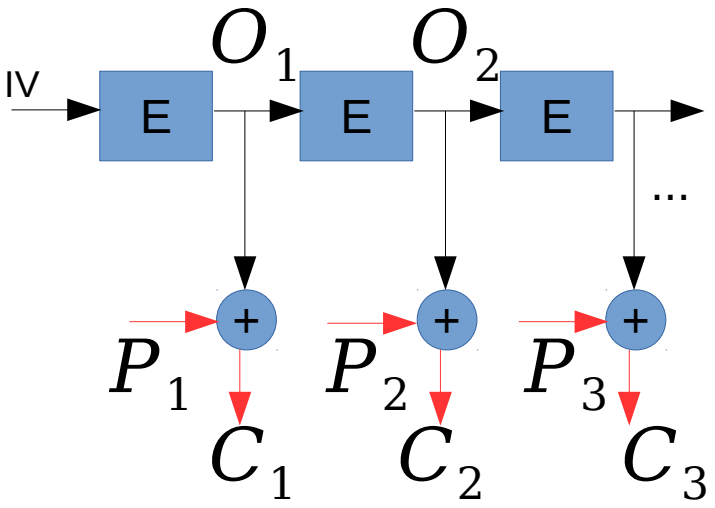


CFB Mode

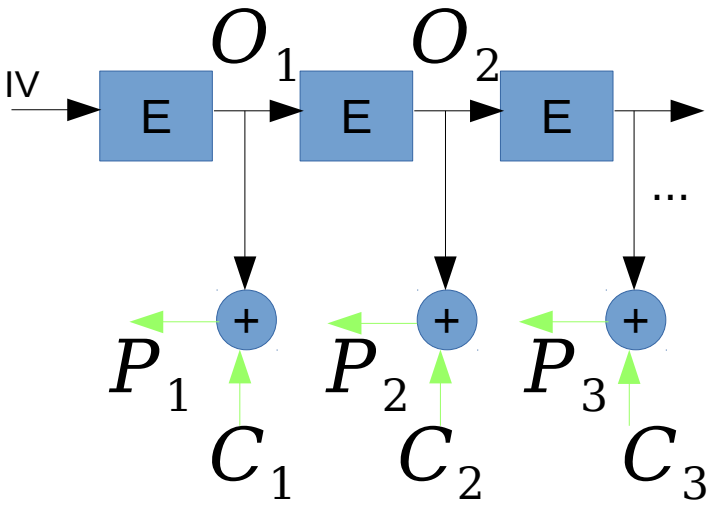
- Sender to Receiver: IV, and C_1 to C_n
- When used for MAC, IV, C_n sent with plain-text
- Tx error in C_k affects decryption of P_k and P_{k+1}
- Encryption/decryption can not be parallelized
- A change in any bit of any plain-text block will dramatically modify the all following cipher text blocks
- Block cipher used in encryption mode for both encryption and decryption (advantages?)

OFB Mode

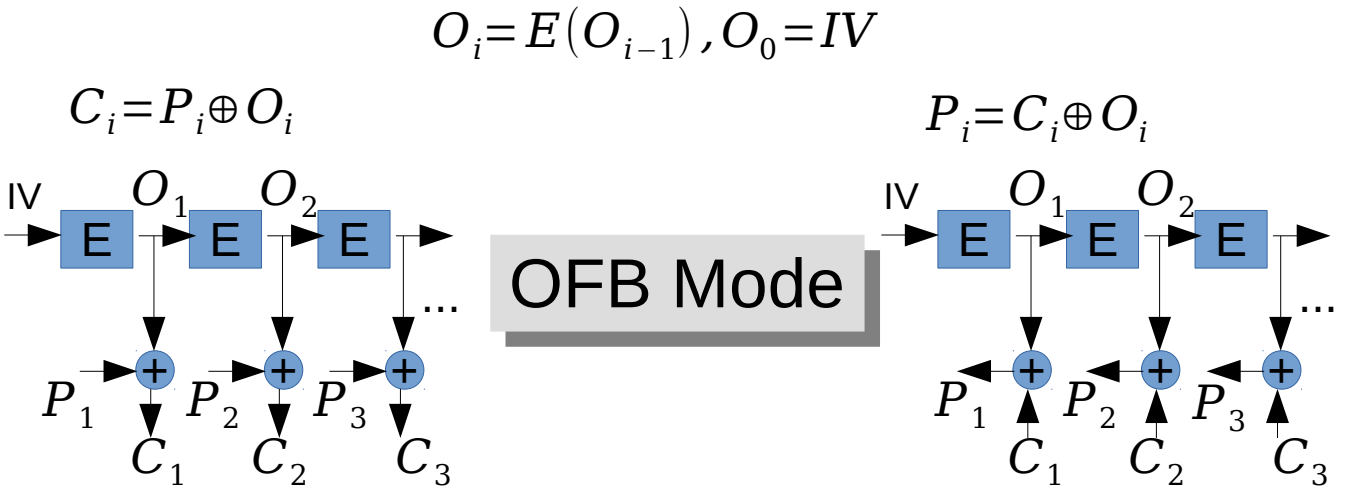
$$O_i = E(O_{i-1}), O_0 = IV$$



$$C_i = P_i \oplus O_i$$



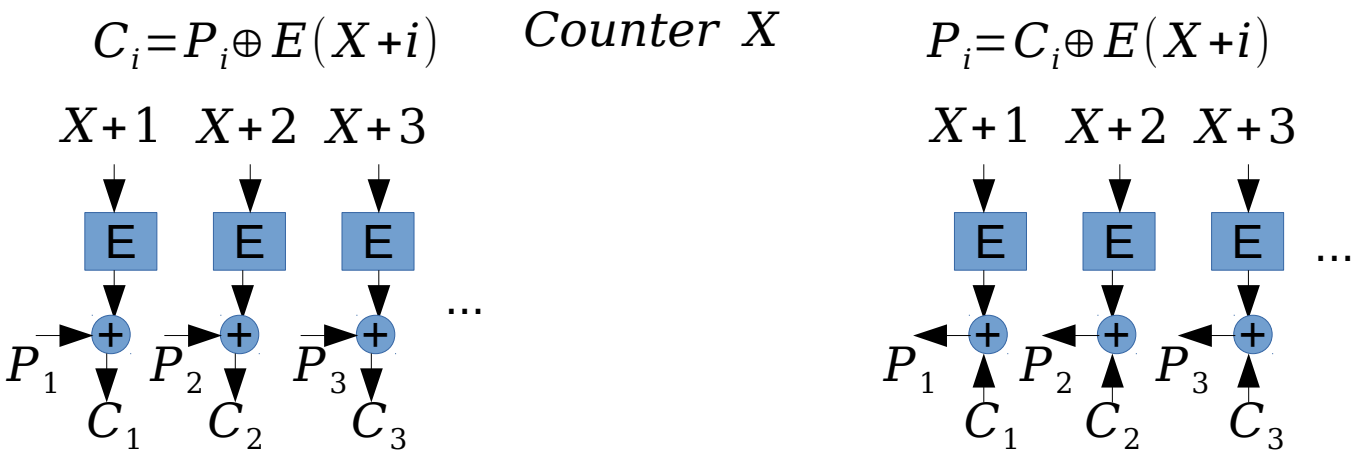
$$P_i = C_i \oplus O_i$$



- Converts a block cipher into a stream cipher
- Not parallelizable
- If a bit of any cipher text is inverted the corresponding plain-text bit will be inverted.
- Preferable for encrypting streaming data over noisy channels
- If data integrity is crucial then some additional mechanism should be used to ensure that.

CTR Mode

- Can be parallelized (like ECB)
- Same plain-text will not produce same cipher text (unlike ECB)
- Two of the most recommended modes are currently CTR and CBC



Block Cipher Modes: Key Concerns

- For all modes
  - Encryption
    - Input: P_1, P_2, … P_n, IV (IV for all modes except ECB)
    - Output: C_1, C_2, … C_n
  - C_1, C_2, … C_n and IV sent over a open channel
  - Decryption
    - Input: C_1, C_2, … C_n, and IV
    - Output: P_1, P_2, … P_n
- Concerns
  - What happens if there are repetitions in plain-text blocks?
  - What happens if there is a random channel error? What is the result of a bit error in block C_k?
  - Can attackers take advantage of their ability to perpetrate *deliberate* errors?
  - What happens if any plain-text block is modified?

Block Cipher Modes: Key Concerns

Concerns

- What happens if there are repetitions in plain-text blocks?
  - In all modes except ECB this is not an issue. Same p[lain text blocks produce different cipher text blocks.
- What happens if there is a random bit error in block C_k?
  - In CBC and CFB two plain text blocks (P_k and P_{k+1}) will be affected
  - In OFB (stream cipher) only the same bit of P_k will be affected
- Can attackers take advantage of their ability to perpetrate *deliberate* errors?
  - Yes, in OFB, CTR, and first block in CBC.
  - To a lesser extent in CFB as changing a specific bit in P_k will affect the same bit in C_k but affects C_{k+1} in an unpredictable manner.
- What happens if any plain-text block is modified?
  - In CBC and CFB a change in P_k unpredictably affects all blocks C_k and later.

Summary

- ECB: Random access, reveals plain-text patterns
- CBC: Useful for MAC. Encrypt IV
- CFB: Useful for MAC.
- OFB: Stream Cipher
- CTR: Random access, does **not** reveal plain-text patterns.

Useful Thumb Rules

- Do not use stream cipher if integrity is crucial
  - Attacker can modify specific bits
  - Use only if noise resiliency is important
  - If integrity is also necessary an additional mechanism should be used
- For the same reason watch out for CTR mode
  - Use only if random access is necessary
  - If integrity is also essential it can be achieved with an extra cost
    - An additional block cipher operation instead of XOR
    - Use *E(X+i)* as a key for encrypting block i.
- CBC/CFB for message authentication

Brute-force Attacks on Ciphers

C = E(P,K). Attacker has C, no K

Try every possible key K

$P_i = D(C,K_i)$

How do we know when to stop? Under *any* key there will be *some* P_i

How do we know that a particular P_i is the correct plain-text?

Does this mean brute force attacks are not possible?

Brute-force Attacks are Always On-the-Table

- Natural redundancy of plain-text
- Deliberately introduced redundancy for authentication

  (for example, MACs)
- Known plaintext-ciphertext pairs

Redundancy in Plain-Text

Think of all possible 100 character strings that “make sense”

For example, say a billion books, each with 1 billion “strings that make sense” - still makes it only 10^{18} possible phrases!

How many total strings of length 100?

26^{100}. That is more than 3×10^{141} !

Say we encrypt a meaningful string with a 64 bit key,

the cipher-text is decrypted with another key

What is the probability that the wrong key results in a string that makes sense?

$2^{64} * 10^{18}/(3*10^{141}) < 6 * 10^{-105}$

Which is good news for the attacker...

Vernam Cipher

What if we make the *number of possible keys* the same as the *number of possible plain text messages?*

One-time pad – Vernam Cipher

Cannot try out keys any more! There is always a key which maps cipher text to **every possible** plain text

No way an attacker can eliminate any message – all messages are equally likely

The attacker learns NOTHING!
Perfect Secrecy
Not very practical. Why?

Good Cipher vs Strong Cipher

- A good cipher meets its design goals
  - Only possible attack is the brute-force attack (determined by key-length k)
- A strong cipher is a good cipher with sufficiently large key-length k
- Can we get a strong cipher from a good cipher?
  - Yes, multiple encryption
  - A good cipher with key length k can be converted to a strong cipher with key length nk by performing (n+1) repeated encryptions with (n+1) independent keys.

Multiple Encryption

- Double Encryption
- $C = E_{K1}(E_{K2}(P))$
- Is there a K3 such that $C = E_{K3}(P)$
- If there is, there is no point doing multiple encryption
  - Useless for Caeser cipher
  - Or any cipher based on permutation **or** substitution
- Multiple encryption is indeed useful for modern ciphers

Double Encryption

- Why is it impractical to find K3?
- Consider a k-bit block cipher with b-bit blocks
- Each of the 2^k keys defines a random one-to-one mapping between two tables of size 2^b
- Total number of possible mappings is factorial(2^b)
  - which is an impossibly large number (for example, factorial(2^{64}) > 10^x where x is a 20-digit number!
- Only 2^k out of factorial(2^b) possible mappings are used by the cipher
- Likelihood that a composite mapping (mapping twice) is the same as one of the permitted mappings is 2^k/factorial(2^b)

Meet-in-the-Middle

An attack that weakens the strength of multiple encryption (n+1 encryptions with n+1 independent keys required to increase strength by factor n)

$$C = E_{K1}\left(E_{K2}\left(P\right)\right)$$

Let us assume attacker knows some P-C pairs

Compute $D_{K2'}\left(C_1\right)$ for all 2^k possible $K2$

Compute $E_{K1'}\left(P_1\right)$ for all 2^k possible $K1$

Values for which $E_{K1'}\left(P_1\right) = D_{K2'}\left(C_1\right)$ are possible candidates

On an average $2^{2k}/2^b$ key-pairs will work for a specific $P_{1,}\, C_1$

With two known P-C pairs prob. of false alarm falls to $2^{2k}/\left(2^b\right)^2$

With n known P-C pairs false alarm probability is $2^{2k}/\left(2^b\right)^n$

Meet-in-the-Middle

On an average $2^{2k}/2^b$ key-pairs will work for a specific $P_{1,}C_1$

Why? Given $P_{1,}$ 2^{2k} possible keys can produce only 2^b outcomes

In DES with $k=56, b=64, 2^{112}$ keys can produce only 2^{64} different outputs

So several keys will map P_1 to the same $C_{1,}$

$2^{112-56}=2^{48}$ keys will yield the same C_1 for a given P_1

Given 2 pairs the possible outcomes are $\left(2^b\right)^2$

With two known P-C pairs prob. of false alarm falls to $2^{2k}/\left(2^b\right)^2$

With n known P-C pairs false alarm probability is $2^{2k}/\left(2^b\right)^n$

Triple Encryption

- Triple DES is widely used
- $C = E_{K3}(D_{K2}(E_{K1}(P)))$
  - Using decryption with an independent key does not compromise security in any way
  - Backward compatibility with single DES (Triple DES with all three keys the same becomes single DES)
- Triple DES with two keys also used (less widely)
- $C=E_{K1}(D_{K2}(E_{K1}(P)))$
- Estimated strength of triple DES is 112-bit security. 80-bit security for triple DES with 2 keys.

Good Cipher to Strong Cipher

- You can design a strong good cipher
- Or design a simple “weak good cipher” and use multiple encryption
- The latter may actually be a good idea
- Simple ciphers may be easier to test thoroughly to make sure there are no weaknesses

Hash Functions

- *d=H(M); H()* is a one-way function
- *M* is can be of any size; *d* is the digest of *M*, and is of a fixed size (say, *n*-bits)
- Several possible inputs will yield the same *n*-bit digest
- *M* is called the pre-image of digest *d*
- What is the difference between a “hash function” and a “cryptographic hash function”?
  - Pre-image resistance and collision resistance

Pre-image Resistance

- Given pre-image *x*, with digest *d=H(x)*, it is impractical to find for another pre-image x' != x that yields the same digest (*d=H(x')*)
  - Even while several candidates for *x'* surely exist, the only way to find one is by brute-force search
    - pick some x' and check if *H(x')=d*. If not try again till you find one that satisfies the requirement
  - Before you actually compute *H(x'),* every *n*-bit digest is an equally likely output.
  - Probability that a given *x'* will yield the sought digest is $1/2^n$
  - On an average we need to make $2^{(n-1)}$ attempts before we find a suitable *x'*.
- This property is more aptly called second pre-image resistance

Collision Resistance

- A collision is finding two pre-images with the same digest (whatever the common digest may be)
- For an ideal hash function with n-bit digest the only way to find a collision is by brute-force search
  - Pick a random pre-image and compute and store the digest
  - Pick another and compute and store the digest (check if it is the same as the previous. If not continue)
  - Pick another and check if it the same as either of the previous two
  - And so on

Collision Resistance

- Finding a collision is a lot simpler than finding a pre-image
- The brute force complexity for finding a collision in a hash function with *n*-bit digest is $1/2^{n/2}$
- Brute force complexity $1/2^{80}$ for $n=160$: We need make a million billion billion attempts to succeed.
- Birthday paradox:
  - In a room of 50 randomly chosen people what is the probability that at least one of them was born on Jan 1st?
  - In a room of 50 randomly chosen people what is the probability that at least two people have the same birthday?

Digest is a Commitment

- The digest *d=H(X)* is a **commitment** to *X*
  - Suppose I want to prove to you that I know *X* **now**
  - But I am only allowed to reveal *X* tomorrow
  - I can give you the digest *d* today. When I release *X* tomorrow you will know that I had to have known *X* today
  - There is no way I can find an *X* given *d*. I should have known *X* to compute *d*
  - This is the property that makes hash functions useful in practice

Hash Functions vs Ciphers

- As long as we can guarantee integrity of the digest *d* (a succinct **commitment** to *X)*
  - We can guarantee the integrity of *X*
- Encryption schemes can be used to assure the **privacy** of an unlimited number of values (by repeated use of block ciphers using the same key)
  - Hash functions are used to guarantee the integrity of a large number of values (by assuring integrity of the commitment)
- Both cater for *trust-amplification*

Compression Function

- Hash functions are implemented using *compression functions*
- A compression function with *n*-bit output has the form *d=c(p,B)*
  - Input *p* (previous state) and output *d* (next state) are *n*-bits long
  - Input *B* is a block of fixed size (say *b*-bits)
  - Typically *n*=160-bits; b=512 bits
- Looks very similar to a block cipher with *d* and *p* as cipher-text and plain-text and *B* as key?
  - Except that we do **not** need reversibility

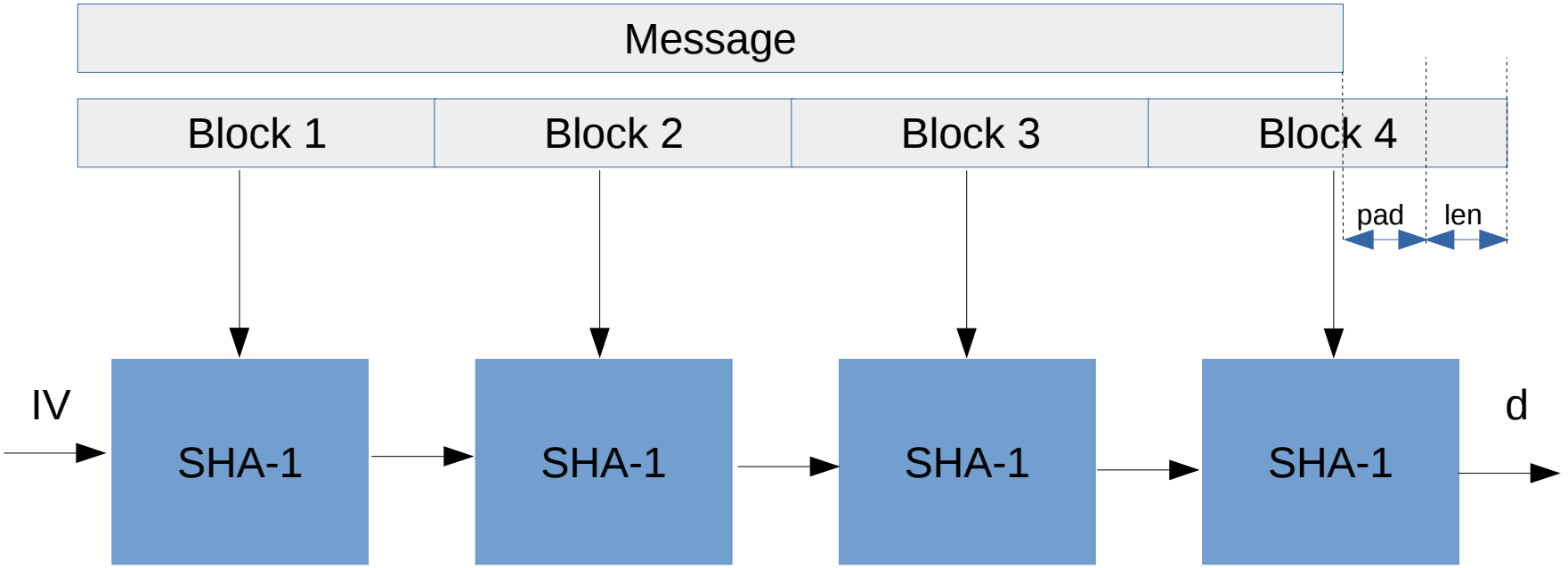
Merkle-Damagard Construction

- Compression function “compresses” a *(n+b)*-bit input to *n*-bit digest
- Hash function *H()* (with unlimited input size and output size *n*-bits) constructed by repeated use of a compression function
- Merkle-Damagard construction ensures that if the compression function *c()* is pre-image resistance and collision resistant, the hash function *H()* will be too.
- Easier to analyze the security properties of *c()* (with fixed size inputs/outputs) than that of *H()*

Merkle-Damagard Construction

- Let the input size be *L*-bits
- Digest size *n*, block size be *b*
- Divide the input into *u b*-bit blocks such that *ub>L+64*
- The last block will include
  - *L-(n-1)b* remaining bits of the input,
  - Zero padding (at least 1 bit, at most *b*-bits)
  - 64 bits to represent the actual length *L* of the input,
- Compression function is applied to each block sequentially

Hash Function Construction



Standard Compression Blocks

SHA-1 (b=512, n=160)
MD5SUM (b=512, n=128)
SHA-2(b=512, n=256)

Last Block

Remaining message bits (can be 0)
Zero pad (1 to b bits)
Message Length (64 bits)

Merkle-Damagard Construction

- The (next state) output of one block is the (previous state) input to the next block.
- The next-state output of the final block is the desired digest
- The previous state input to the first block is a standard fixed value (IV)
- The bit-crunching operations inside the compression function blocks are similar to operations in block-ciphers
  - Also need to satisfy confusion and diffusion properties to achieve pre-image and collision resistance

Other Useful Constructions

- Hash trees
- Instead of large continuous chunk of bits (like a file) we sometimes want to deal with tons of discrete independent pieces of data (like a data-base with several records)
- Hash functions (*H()*) allow one to compute a commitment for the former
- How do we compute the commitment for a database of records?

Merkle Binary Hash Tree

- Assume the database has *N* records
- And a way of computing a commitment (a digest) *d* for the entire database
- Once the digest *d* has been computed
  - Any one should be able to easily verify that a specific record *R* belongs to a database with commitment *d* (without having to bother with the other *N-1* records)
  - If there is a reason to change a record *R*, we should be able to update the digest accordingly to *d'* (again, without worrying about other records)
- As long as we can guarantee the integrity of the commitment *d*, we can guarantee the integrity of every record in the database.

Binary Tree

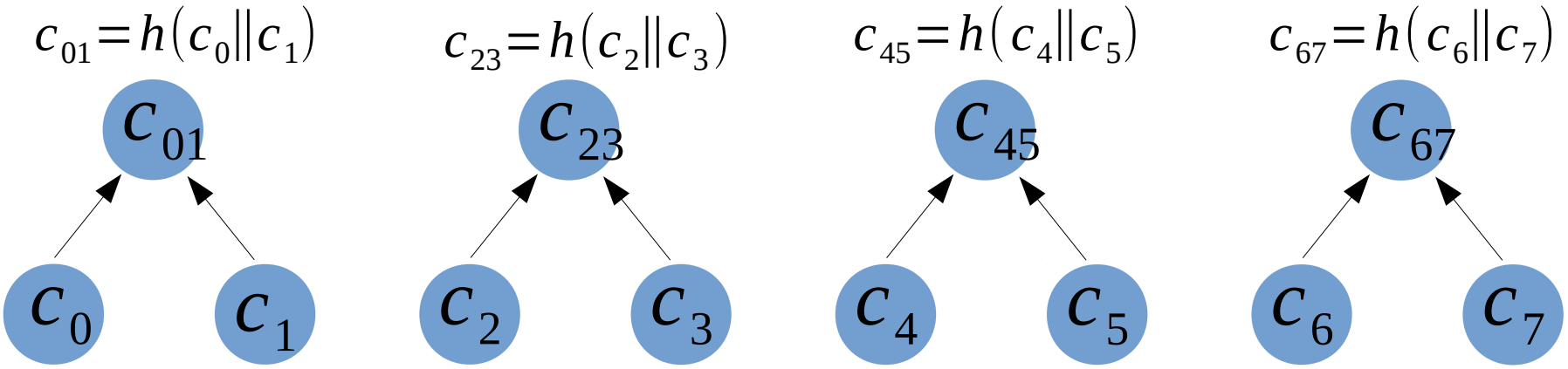
- A tree of commitments (hashes)
- *N* “leaf hashes” at the lowest level of the tree (level 0). Each hash corresponds to a record
- At the next higher level (level 1) we have *N/2* hashes. Each hash in level 1 computed by combining two hashes in level 0
  - **Extending** one hash with another
- At level 2 we have *N/4* hashes obtained by hashing together two hashes from level 1, and so on
- At level r we have $N/2^r$ hashes
- At level L where $L = log_2(N)$, we have a single hash (**root of the tree**), which is a commitment to the entire tree (every record in the database)

Merkle Tree

Begin with N leaf (level-0) nodes
In this example N=8

N/2=4 Level-1 Nodes

$h(c_0 \| c_1)$ can be a compression function $c(p, B)$
where $B = (c_0 \| c_1) \| pad$, $p =$ constant



$c_{01} = h(c_0 \| c_1)$ is

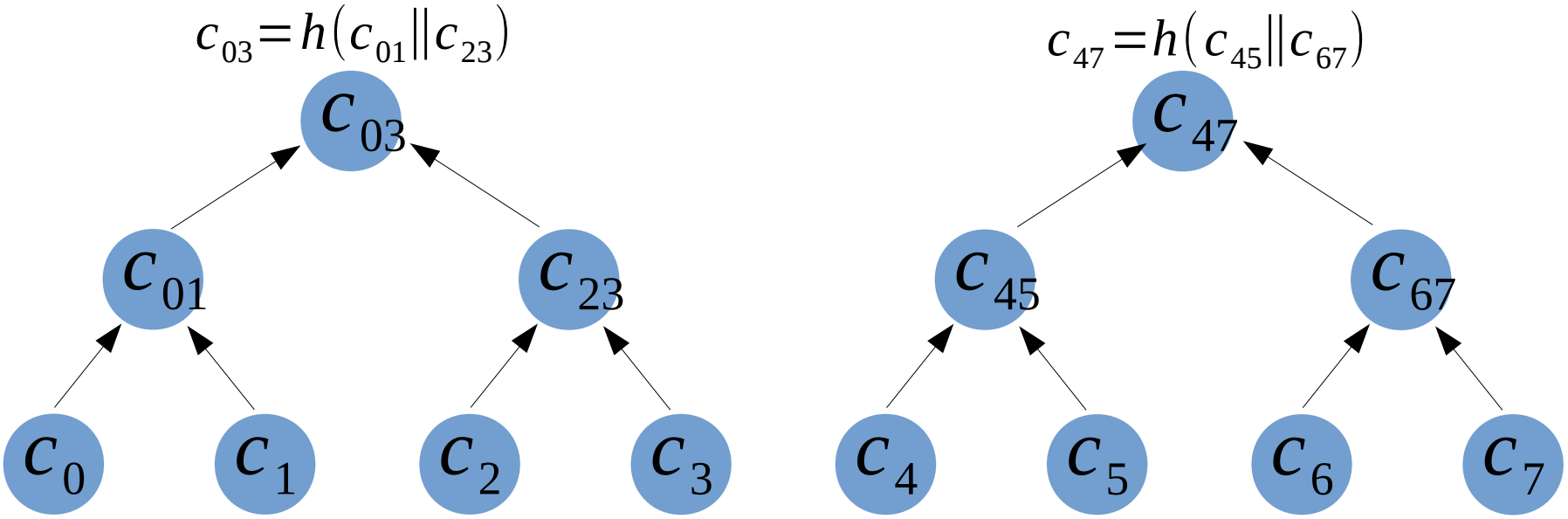
1. Right hash extension of c_0 with c_1

or

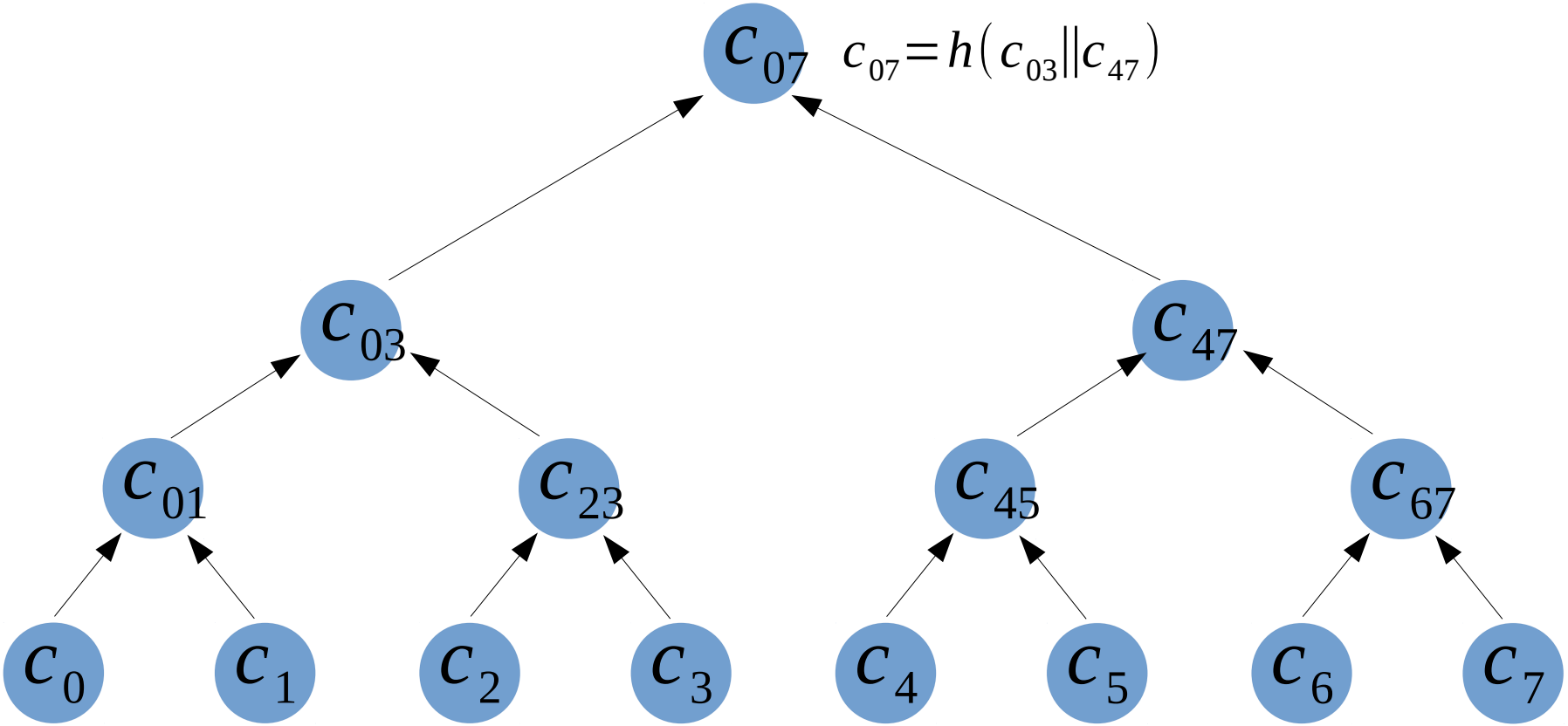
2. Left hash extension of c_1 with c_0

$$h(c_0 \| c_1) \neq h(c_1 \| c_0)$$

N/4=2 Level-2 Nodes



Complete Merkle Tree

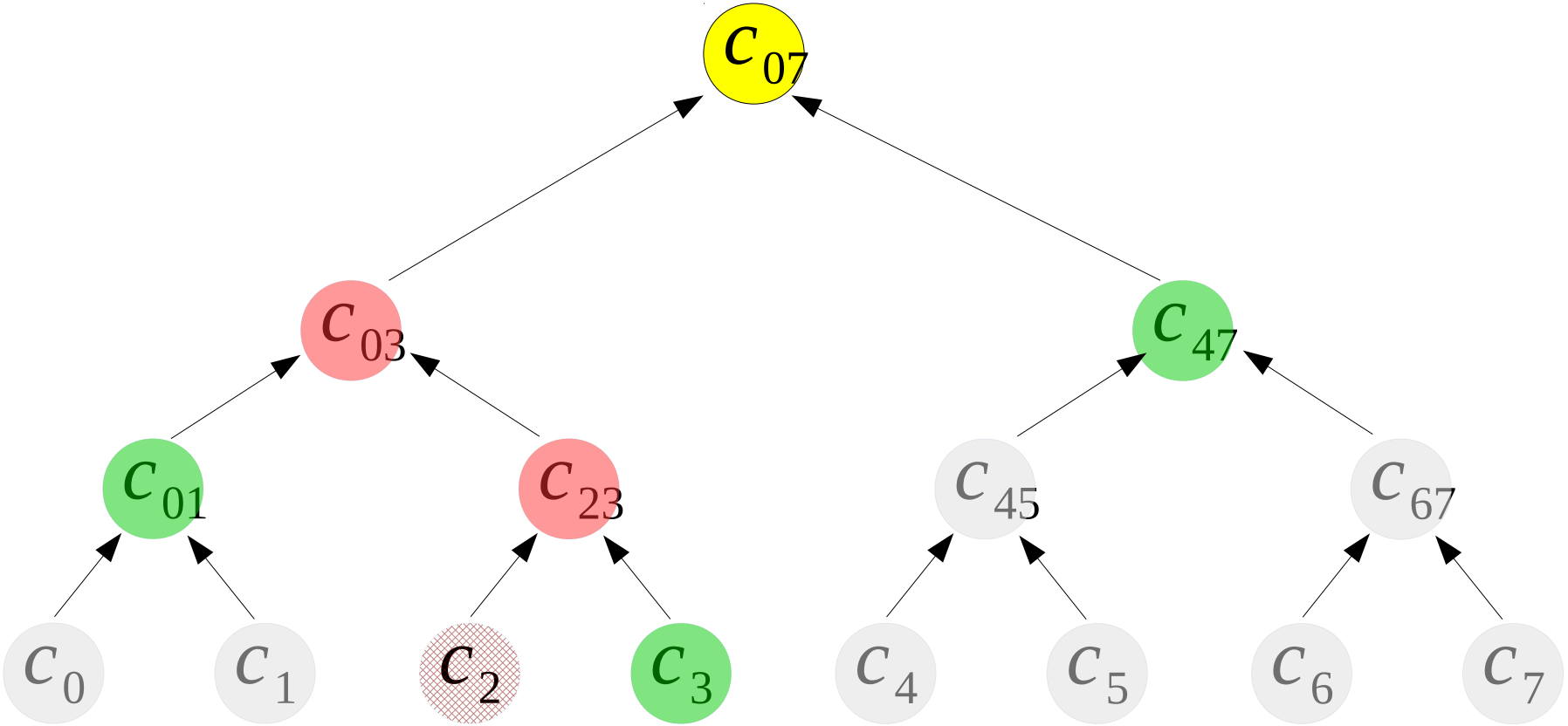


A tree with N=8 nodes has $L=\log_2(8)=3$ levels

Every node has a sibling and 'ancestors' in the path to root

The root c_{07} is the end-point for all N=8 nodes

Complementary Nodes (commitment to **all other** nodes)

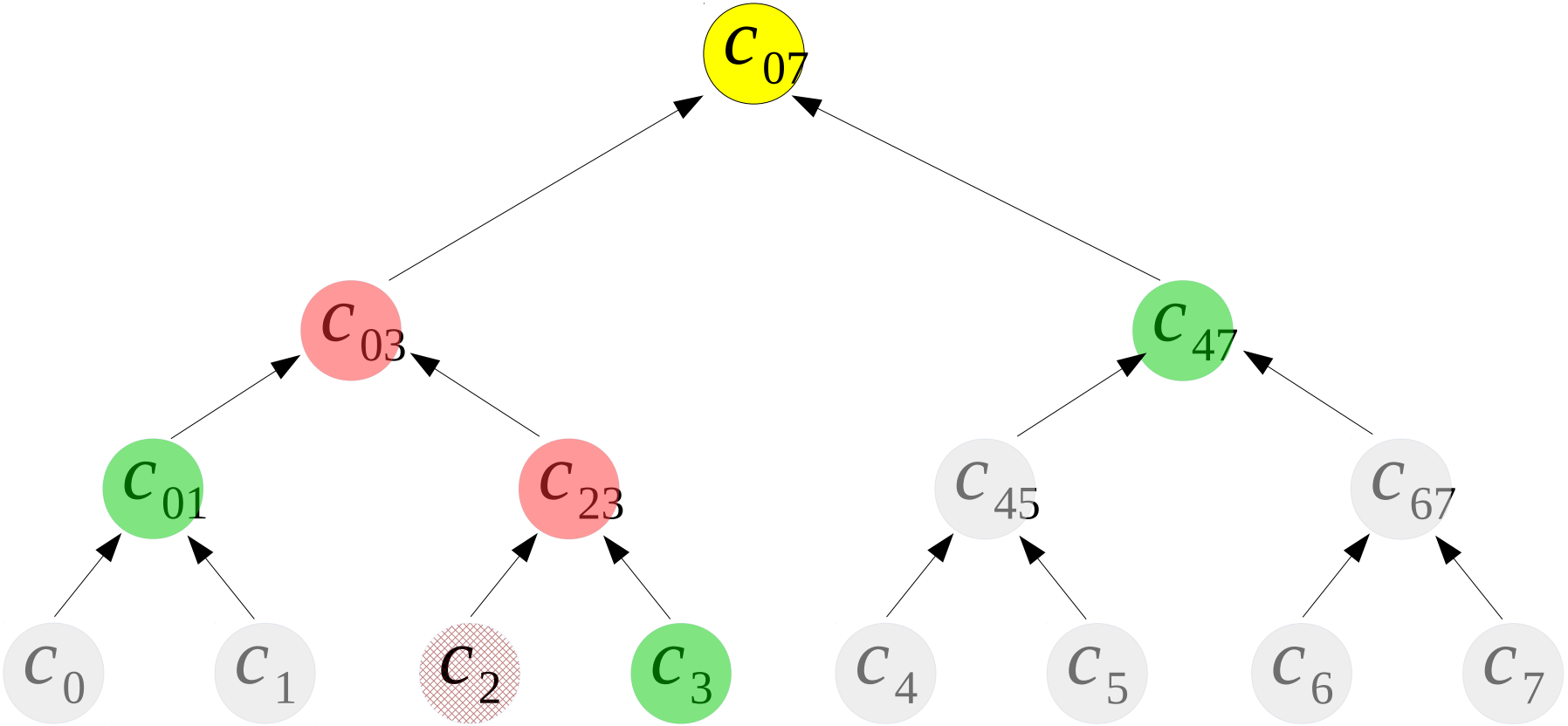


Every leaf node has L complementary nodes

Path from $c_2 : \{c_2, c_{23}, c_{03}\}$

Their respective siblings $\{c_3, c_{01}, c_{47}\}$ are the complementary nodes of c_2

Hash Extension Using Complementary Hashes



$c_2 \Rightarrow c_{3,} c_{01}, c_{47}$

Perform $L=3$ hash extensions of c_2

first with $c_{3,}$ the result with c_{01} then with $c_{47} \cdots$

$$x=h(c_2 \| c_3), x=h(c_{01} \| x), x=h(x \| c_{47})$$

to reach the root (final value of $x=c_{07}$)

Verifying Records

- The records and the entire tree can be stored in an open (untrusted) database server
  - The tree has *2N-1* hashes including the root
- The leaf nodes are (regular) hashes of records $c_2=H(R_2)$
- The root (commitment c_{07}) is protected in a secure location
- To a user who requests record R_2, the database server provides
  - record R_2 (with hash c_2)
  - the L complementary nodes of c_2

Verifying a Record

- Hash the record (R_2) to get the leaf node c_2
- Use the $L=3$ commitments c_3, c_{01} and c_{47}
- Set $tmp= c_2$ and perform $L=3$ **hash extensions**
- Iterate
  - $tmp=h(tmp \,||\, c_3)$ (right extension)
  - $tmp=h(c_{01}||tmp)$ (left extension)
  - $tmp=h(tmp||c_{47})$ (right extension)
- Only if $tmp = c_{07}$ accept record R_2 as genuine

Right vs Left Hash Extension

- Note that *h(x,y) != h(y,x)*
- Right extension is not the same as left extension
- Server can also send what extension to use (1 additional bit for each complementary hash)
- Or send the index of the record in the tree
  - Flipped binary representation of record index tells us when to do right / left extension (0-right, 1-left)

$$c_6 \Rightarrow c_{7,}\, c_{45}, c_{03}$$
$$6 = 110 \leftarrow (\text{ right, left, left })$$
$$tmp = c_6$$
$$tmp = h(tmp \| c_7) \text{ right extension}$$
$$tmp = h(c_{45} \| tmp) \text{ left exension}$$
$$tmp = h(c_{03} \| tmp) \text{ left extension}$$

$$c_3 \Rightarrow c_{2,}\, c_{01}, c_{47}$$
$$3 = 011 \leftarrow (\text{ left, left, right })$$
$$tmp = c_3$$
$$tmp = h(c_2 \| tmp) \text{ left extension}$$
$$tmp = h(c_{01} \| tmp) \text{ left exension}$$
$$tmp = h(tmp \| c_{47}) \text{ right extension}$$

Scalability of Hash tree

- What about a database with 1000 records? L=10
- Million (L=20); Billion (L=30); Trillion (L=40)
- We simply need to perform 30 hashes to verify any record in a database with a billion records
- Only one hash (root) needs to be protected

What about Intermediate hashes?

- Do they need to be protected?
  - No. Pre-image resistance of hash functions guarantees that it is not possible fabricate a set of complementary nodes for a leaf hash.
- After we successfully verify a leaf against the root we can be
  - Assured of the integrity of the leaf node
  - Assured of the integrity of all complementary nodes
- The latter is very important for **updating** the database

Updating a Record

- Given a leaf node *x*, its index, and its set of *L* complementary nodes $v_1, v_2, \ldots v_L$
  - starting with the leaf node we can reach the root by L hash extensions
- If we want to modify *x* to *x′* (to modify a record) all we need to do is to start with *x′* and do the same hash extensions with the **same** complementary nodes $v_1, v_2, \ldots v_L$ to compute the **new root**
- All other records will still be consistent with the root. Why?
  - The complementary nodes are actually commitments to **all other** nodes.

Hashed Message Authentication Code (HMAC)

- MACs using hash function instead of block cipher modes like CBC and CFB
- M = H(M || K) (simplistic representation)
- Standard HMAC
  - HMAC=H((K + opad)||H((K+ipad)||M))
  - opad=0x5c5c... 5c5c
  - ipad=0x3636... 3636

HMAC Properties

- Brute force strength depends only on length of key
  - (not on collision resistance of hash function)
  - Not on the size of the MAC (usually the MAC is truncated)
- General MAC related issues
  - How does MAC length affect security?