Security in Computing

Chapter 2

Elementary Cryptography (part 3)

Chapter Outline

- 2.1 Terminology and Background
- 2.2 Substitution Ciphers
- 2.3 Transpositions (Permutations)
- 2.4 Making "Good" Encryption Algorithms
- 2.5 The Data Encryption Standard (DES)
- 2.6 The AES Algorithm
- 2.7 Public Key Encryption
- 2.8 Uses of Encryption
- 2.9 Summary

Characteristics of "Good" Ciphers

- Shannon *Communication Theory of Secrecy Systems* (1949), pg. 15
  - Amount of secrecy should be proportional to value
  - Key needs to be transmitted/memorized → should be as short as possible
  - Encryption/decryption should be as simple as possible
  - Errors shouldn't propagate
  - Size of the ciphertext should be the same as plaintext

Trustworthy Encryption Properties

- Encryption systems should:
  - be based on sound mathematics
  - be analyzed by experts
  - stand the test of time

Block Ciphers We've Done

Cipher	Block Size
transposition with period *p*	*p*
simple substitution	1 character
homophonic substitution	1 character
playfair	2 characters

- Not stream ciphers?

Block Ciphers We've Done

Cipher	Block Size
transposition with period *p*	*p*
simple substitution	1 character
homophonic substitution	1 character
playfair	2 characters

- Not stream ciphers?
  - No.
  - Stream ciphers use the i^{th} part of the keystream to encrypt symbol i.
  - These use the same key for all plaintext chars.

Stream and Block Ciphers

- **stream ciphers**
  - encrypt one symbol (bit, byte, or word) at a time
  - encrypt the i^{th} symbol with the i^{th} part of the keystream
- **block ciphers** encrypt larger blocks of plaintext
  - block size → usually 64 bits or more
  - encrypt all blocks with the same key

Block Ciphers We've Done

Cipher	Block Size
transposition with period *p*	*p*
simple substitution	1 character
homophonic substitution	1 character
playfair	2 characters

Question

- When we do a simple substitution cipher
  - We map a character in P to a character in C
- Question:
  - *Is it possible for two different chars in P to map to the same character in C?*

Question

- When we do a simple substitution cipher
  - We map a character in P to a character in C
- Question:
  - *Is it possible for two different chars in P to map to the same character in C?*
- Answer:
  - no. otherwise, how would you decrypt? example:

P	A	B	C	D	E	F	G	H	I	J	K	L	M	N	...
C	J	E	F	K	M	E	E	P	M	D	S	T	L	A	...

Stream Ciphers We've Done

Cipher	Period
Vigenere with period *p*	*p*
Rotor machine with *r* rotors	26**r
Vernam	none

Stream vs. Block Ciphers

	Stream Ciphers	Block Ciphers
Advantages	fast	high diffusion
	low error propagation	more immunity to insertior
Disadvantages	low diffusion	slower
	vulnerable to insertions and modifications	error propagation

Cryptographic Functions

- The cryptographic algorithms that we've been discussing (*except maybe the random homophonic ciphers*) are functions.
- Plaintext alphabet is P
- Ciphertext alphabet is C
- The cryptographic algorithm maps the characters in P to C
- f: $P \rightarrow C$

Cryptographic functions are 1→1

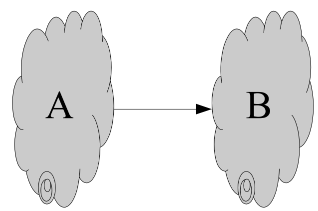
- *Question*:
  - Why must cryptographic functions be 1→1?

and now ...

same discussion sounding like you ate a math book

Math Review: Functions

- *Recall* - A function is defined by two sets A, and B, and a rule that maps the elements in A to elements in B
  - A is called the domain
  - B is called the co-domain



- Notation – f: $A \rightarrow B$
- A function is **one-to-one** (1→1) if for every element in B, there is at most one element in A

more complex crypto

- for $y=x^2$ it's easier to define function without drawing the map
- we'd like the same thing for crypto function

Block Ciphers

- Plain substitution ciphers that we've discussed
  - example:
    - A→K
    - B→D
    - C→Q
    - ...
- ciphers that operate on 64-bit blocks
  - example:
    - 0x0000 0001 → 0x81A7 C961
    - 0x0000 0002 → 0xB132 8DC5
    - ...

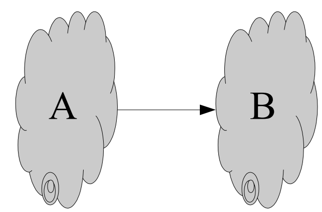
Cryptographic functions are 1→1

- *Question*:
  - Why must cryptographic functions be 1→1?
- *Answer:*
  - If they weren't 1→1 this would mean that there are elements in *C* for which there are more than one element in *P*.
  - How would we do decryption?
  - *Example:*

P	A	B	C	D	E	F	G	H	I	J	K	L	M	N	...
C	J	E	F	K	M	E	E	P	M	D	S	T	L	A	...

A simple function

- $y = x^2$



- what's A and B?
- is it practical to specify the function like this?

A	1	2	3	4	5	6	7 ...
B	1	4	9	16	25	36	49 ...

Bits to encode 64-bit block ciphers

- ciphers that operate on 64-bit blocks
  - example:
    - 0x0000 0001 → 0x81A7 C961
    - 0x0000 0002 → 0xB132 8DC5
    - ...
- How many bits would it take to encode this?

Bits to encode 64-bit block ciphers

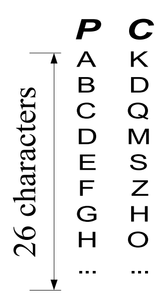
- ciphers that operate on 64-bit blocks
  - example:
    - 0x0000 0001 → 0x81A7 C961
    - 0x0000 0002 → 0xB132 8DC5
    - ...
- How many bits would it take to encode this?
  - If we made a table, there would be:
    - 2^{64} entries
    - each entry would be 64 bits long
    - $2^{64} * 2^{6} = 2^{70}$ bits

Block Ciphers

- Plain substitution ciphers that we've discussed
  - example: A→K, B→D, C→Q, ...
  - *how many bits are required to specify the mapping?*

Block Ciphers

- Plain substitution ciphers that we've discussed
  - example: A→K, B→D, C→Q, ...
  - *how many bits are required to specify the mapping?*



- *Answer:*
  - *There are 26 characters*
  - *It takes 5 bits per character*
  - *26*5 = 130 bits*

Background

- Early 70s non-military crypto research unfocused
- National Bureau of Standards (now NIST) wanted algorithm which:
  - is secure
  - open
  - efficient
  - useful in diverse applications
- IBM Lucifer algorithm submitted
- DES based on Lucifer
- controversies over:
  - reduced key size
  - design (of S-boxes)

Description of DES

- block cipher. 64-bit blocks
- same algorithm used for encryption, decryption
- 56-bit keys
  - represented as 64-bit number
  - but every 8^{th} bit is for parity only → usually ignored
- symmetric: receiver uses same key to decrypt
- uses basic techniques of encryption. provides
  - confusion (substitutions)
  - diffusion (permutations)
- same process 16 times/block
- uses standard arithmetic and logical operators
  - efficient hardware implementations

Bits to encode 64-bit block ciphers

- So for larger block sizes, we have to do something different
- Goal:
  - generate a 1→1 mapping
  - make it look as random as possible
  - don't store all possible input/output pairs

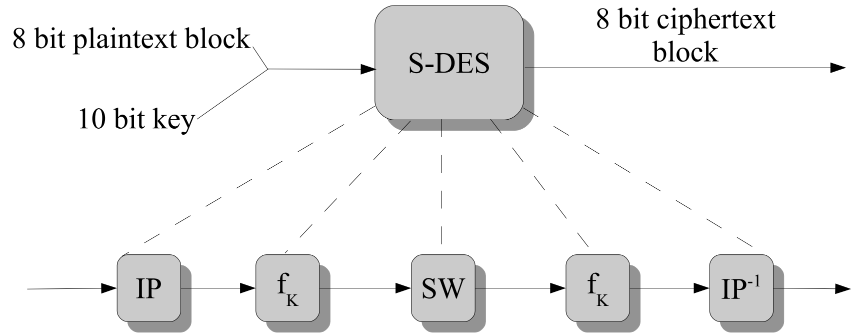
Chapter Outline

- 2.1 Terminology and Background
- 2.2 Substitution Ciphers
- 2.3 Transpositions (Permutations)
- 2.4 Making "Good" Encryption Algorithms
- 2.5 The Data Encryption Standard (DES)
- 2.6 The AES Algorithm
- 2.7 Public Key Encryption
- 2.8 Uses of Encryption
- 2.9 Summary

S-DES overview

- for each block, permutations and substitutions
- 5 functions:
  1) initial permutation (IP)
  2) a complex function f_K
     - consists of permutations and substitutions
     - key is applied
  3) special permutation: switch the left and right sides
  4) f_K again
  5) inverse of initial permutation (IP^{-1})

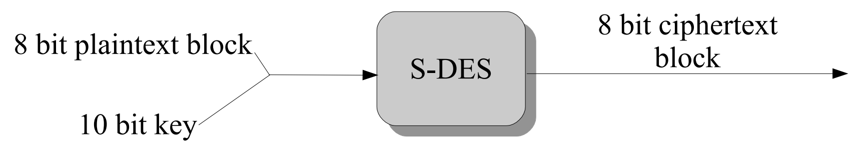
S-DES: more detailed look



But first ...

- DES is very complicated
- Simplified DES first.
  - educational protocol
  - similar to DES
  - works with much smaller units
  - easier to see

S-DES



S-DES: function f_K

$$f_k(L, R) = (L \oplus F(R, SK), R)$$

left 4 bits

right 4 bits

XOR

subkey

key choice

S-DES initial permutation

$p_2\ p_6\ p_3\ p_1\ p_4\ p_8\ p_5\ p_7$

example:

1 0 1 0   1 1 1 0

becomes:

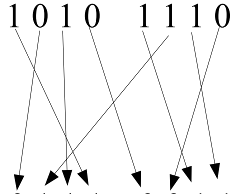
0 1 1 1   0 0 1 1

S-DES initial permutation

$p_2\ p_6\ p_3\ p_1\ p_4\ p_8\ p_5\ p_7$

example:

1 0 1 0   1 1 1 0

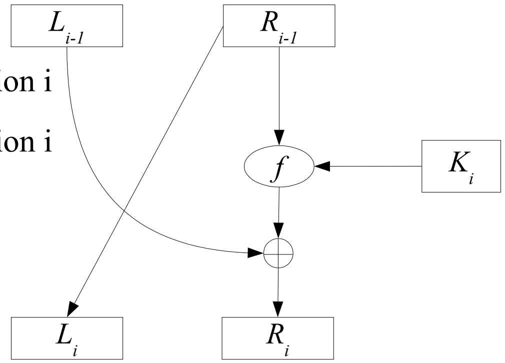


becomes:

0 1 1 1   0 0 1 1

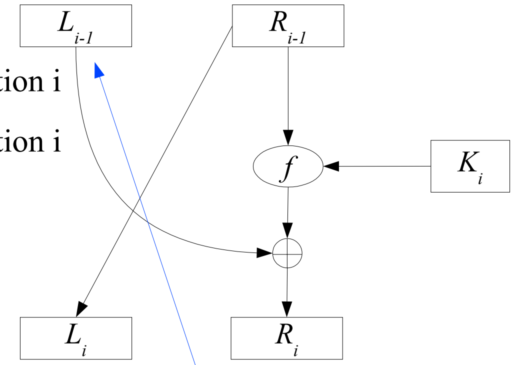
Description of DES

- Break up plaintext into 64-bit blocks
- Each block goes through 16 rounds
  - B_i = block after iteration i
  - L_i = LHS of block after iteration i
  - R_i = RHS of block after iteration i
- For each block of plaintext:
  - initial permutation
  - for (i=1 to 16)
    - $L_i = R_{i-1}$
    - $R_i = L_{i-1} \oplus f(R_{i-1}, k_i)$
  - final permutation



Description of DES

- Break up plaintext into 64-bit blocks
- Each block goes through 16 rounds
  - B_i = block after iteration i
  - L_i = LHS of block after iteration i
  - R_i = RHS of block after iteration i
- For each block of plaintext:
  - initial permutation
  - for (i=1 to 16)
    - $L_i = R_{i-1}$
    - $R_i = L_{i-1} \oplus f(R_{i-1}, k_i)$
  - final permutation



combining LHS-RHS: Feistel Structure

Back to Real World

now back to real DES ...

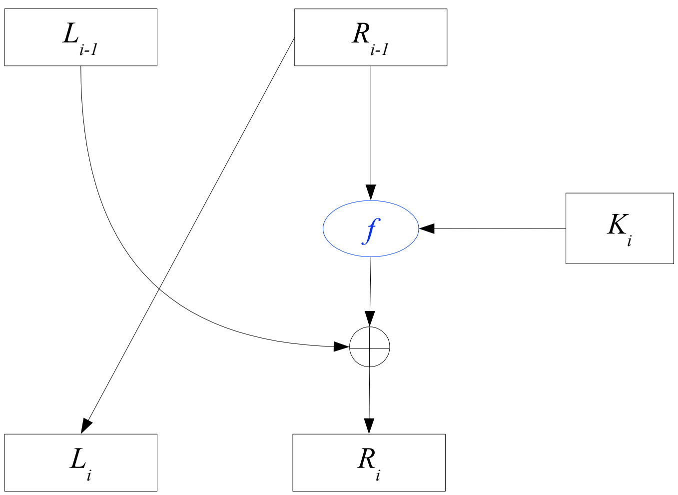
for more details on S-DES, check out supplement to Stallings' *Cryptography and Network Security*
http://williamstallings.com/Crypto/Crypto4e.html

Back to Real World

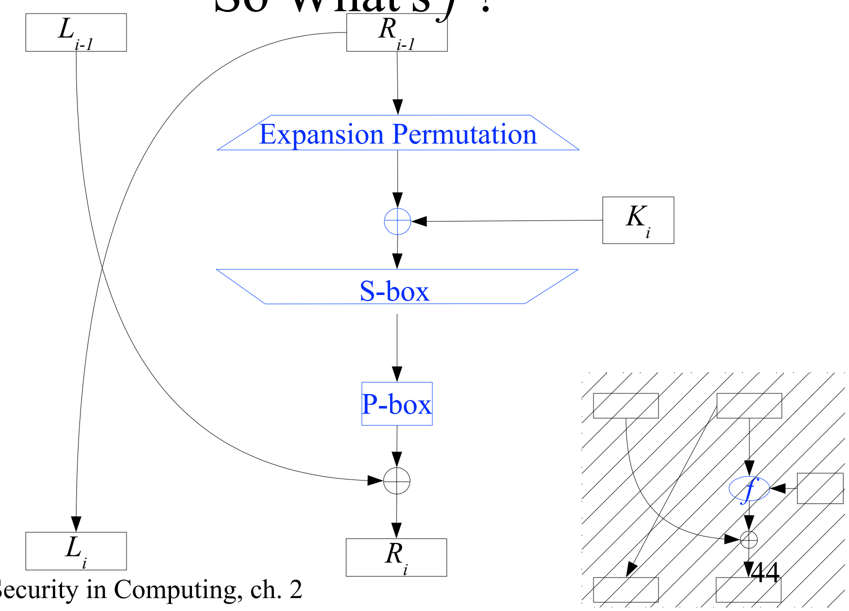
now back to real DES ...

for more details on S-DES, check out supplement to Stallings' *Cryptography and Network Security*
http://williamstallings.com/Crypto/Crypto4e.html

So What's f?



So What's f?



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

- Done before the 16 rounds
- Read: "put bit 58 into the 1st position, put 50 into the 2nd position ..."
- Reversed by Inverse Initial Permutation (after round 16)
- Problem with this?

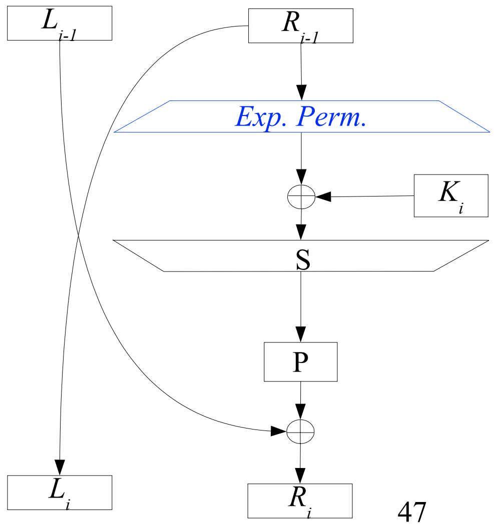
Initial Permutation

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

- Done before the 16 rounds
- Read: "put bit 58 into the 1st position, put 50 into the 2nd position ..."
- Reversed by Inverse Initial Permutation (after round 16)
- Problem with this?
  - *Not really, but it doesn't add to the security*

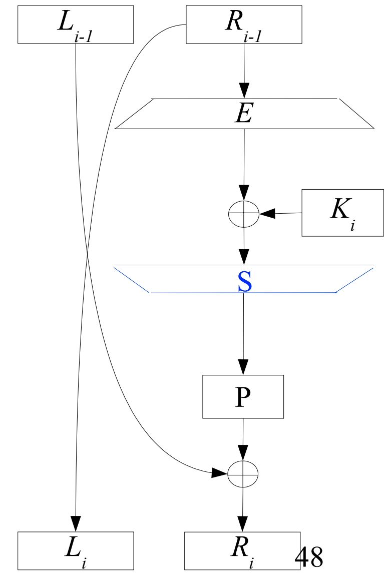
Expansion Permutation

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



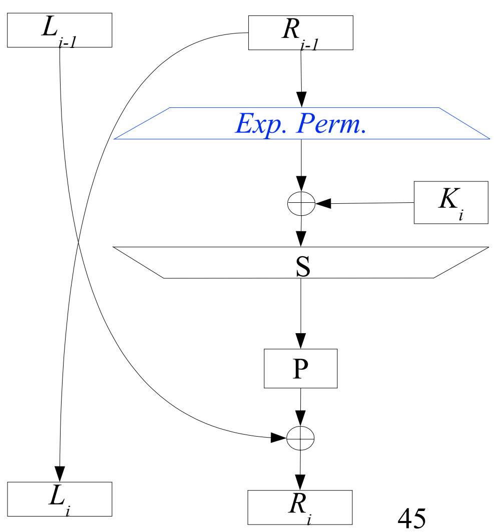
S-boxes

- take 48-bits from result of
  - expansion permutation $\oplus$ K_i
- break into 8 6-bit blocks
  - block 1 → box S_1
  - block 2 → box S_2
  - etc.



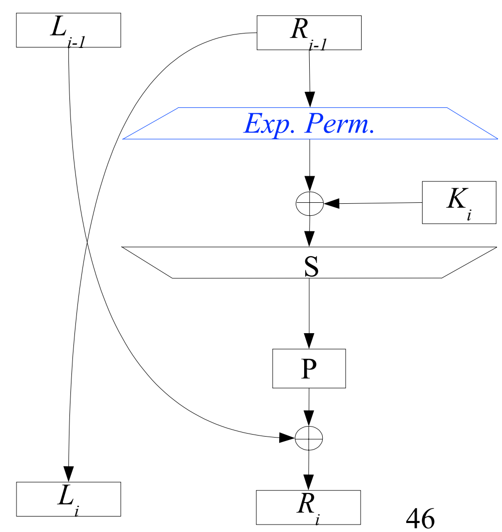
Expansion Permutation

- expand R_i: 32→48 bits
- all bits used at least once. some twice.
- R_i becomes same length as key for XOR
- **avalanche effect**
  - few bits of plaintext affects many bits of ciphertext



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



Example: S box 1

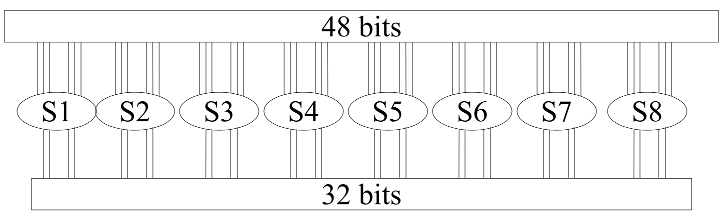
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

- bit 1 and 6 define the row.
- bit 2-5 define col.
- example: 010011
  - bit 1,6 = 01 → row 1
  - bit 2,3,4,5 = 1001 → col 9
  - output = 6, *i.e.* 0110

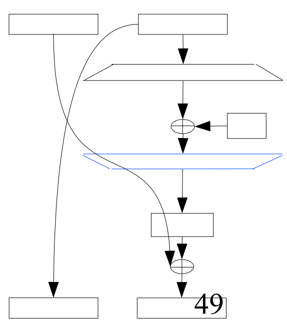
Avalanche Effect

- good ciphers:
  - change few plaintext bits → change many in ciphertext
- pronounced in DES
  - big changes to block after only a few rounds

S boxes



- Each box defines a substitution
  - 6-bit input
  - 4-bit output

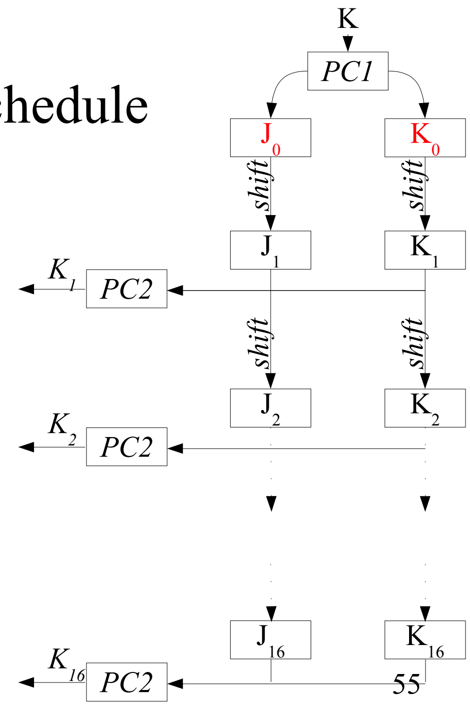


Example: S box 1

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

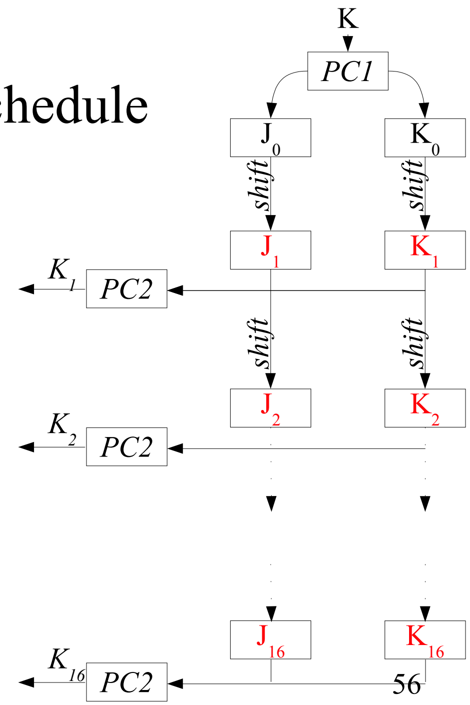
- bit 1 and 6 define the row.
- bit 2-5 define col.
- example: 010011

Key Schedule



- *PC1* – just a simple permutation
- key split in half
  - each half 28 bits
- at round i, J_i and K_i shifted either 1 or 2 bits (depending on round)
- result of shift fed to *PC2*
  - bits are permuted
  - 48 of the 56 bits chosen

Key Schedule

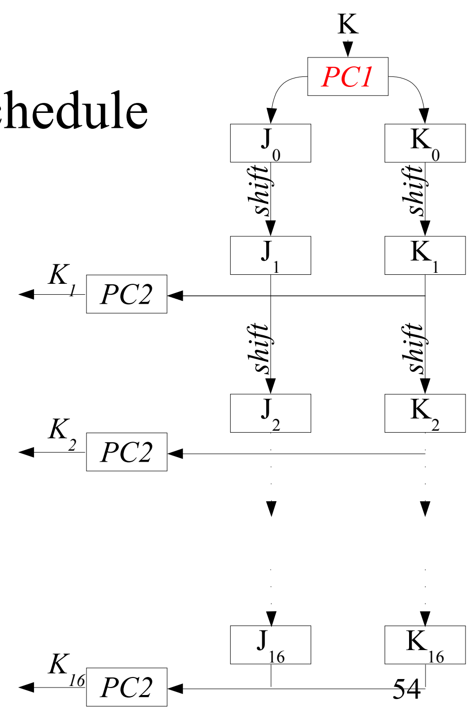


- *PC1* – just a simple permutation
- key split in half
  - each half 28 bits
- at round i, J_i and K_i shifted either 1 or 2 bits (depending on round)
- result of shift fed to *PC2*
  - bits are permuted
  - 48 of the 56 bits chosen

Key Schedule

- Key is 56 bits (64 – 8 parity bits)
- Goes through a permutation before round 1
- Then for each round:
  - divide into two halves
  - circular shift of each half (shift 1 or two bits depending on round)
  - select 48 of the 56 bits

Key Schedule



- *PC1* – just a simple permutation
- key split in half
  - each half 28 bits
- at round i, J_i and K_i shifted either 1 or 2 bits (depending on round)
- result of shift fed to *PC2*
  - bits are permuted
  - 48 of the 56 bits chosen

Strength of DES

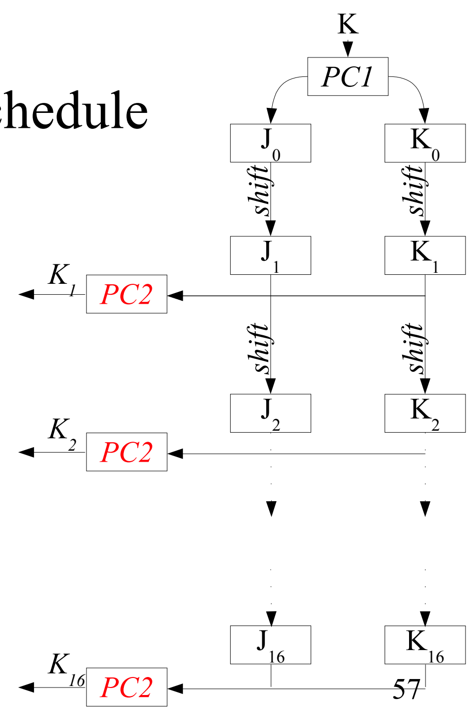
- Strong in 70s. Very weak today.
  - 56-bit keys
  - exhaustive search → average 2^{55} attempts
- DES crackers
  - 1977 - $20,000,000
  - 1998 - $150,000
  - 2004 – ?
  - Now ???

Multiple Encryption with DES

- how about doing DES twice?
  - probably not more secure than doing DES once
    - Merkle and Hellman paper
- 3DES
  - usually use two keys. (but 3 keys also common)
  - effective key strength of 112 bits
  - break through exhaustive search:
    - if we can do 10^9 tries per second, on average
      - 56-bit keys ≈ 800 days
      - 112-bit keys ≈ 6 * 10^{19} years

Key Schedule

- *PC1* – just a simple permutation
- key split in half
  - each half 28 bits
- at round *i*, J_i and K_i shifted either 1 or 2 bits (depending on round)
- result of shift fed to *PC2*
  - bits are permuted
  - 48 of the 56 bits chosen



DES Decryption

- Same as encryption, but done in reverse
  - key schedules, etc.

Electronic Codebook Mode (ECB)

- chop the plaintext into 64 bit blocks
- encrypt each block separately
- pros, cons?

Electronic Codebook Mode (ECB)

Pros

- simple
- encrypt in any order
- encrypt in parallel
- example (database):
  - database stored in encrypted form
  - can change a single record without having to re-encrypt the other records
- no error propagation

Cons

- plaintext block always encrypts to the same ciphertext block
  - could theoretically create a codebook of plaintext → ciphertext pairs
- patterns aren't hidden
  - tcp headers, mail headers, etc., long strings of 0's.
- insertion attacks
- replay attacks

Triple DES Operation: Typical Case

- for each block:
  - encrypt with key 1
  - decrypt with key 2
  - encrypt with key 1
  - *i.e.* $C = E_{K1}(D_{K2}(E_{K1}(P)))$
- *Bonus*: interoperates with DES
  - $E_{K1}(D_{K1}(E_{K1}(P))) = E_{K1}(P)$
- Can also use 3DES with 3 keys

Modes of Operation

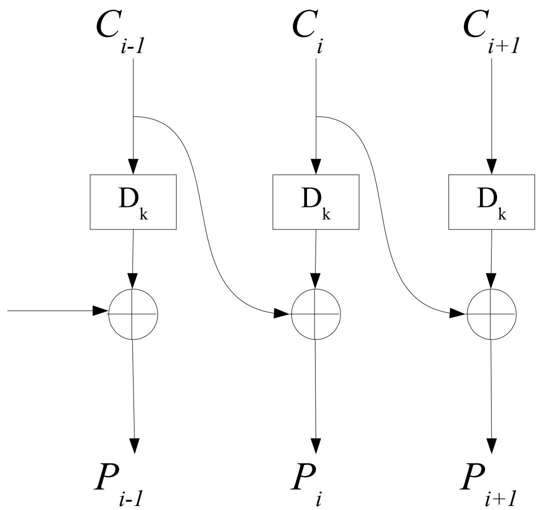
- Not in the textbook (but useful. used in many contexts.)
- Suppose that we have a message longer than 64 bits.
- How do we use a 64-bit block cipher to encrypt it?
- Modes of operation:
  - Electronic Code Book Mode (ECB)
  - Cipher Block Chaining Mode (CBC)
  - Ouput Feedback Mode (OFB)
  - Cipher Feedback Mode (CFB)
  - Counter Mode (CTR)

CBC Decryption

- The *ciphertext* of block *i* is decrypted and then XOR'ed with the ciphertext of block *i-1*

so:

$$P_i = C_{i-1} \oplus D_k(C_i)$$



CBC: Why it works

<u>*Encryption*</u>

$$C_i = E_k(P_i \oplus C_{i-1})$$

<u>*Decryption*</u>

$$P_i = C_{i-1} \oplus D_k(C_i)$$

$$... = C_{i-1} \oplus (P_i \oplus C_{i-1})$$

$$... = P_i$$

Cipher Block Chaining Mode (CBC)

- The *plaintext* of block *i* is XOR'ed with the *ciphertext* of block *i-1* before it is encrypted
- Decryption is just the opposite

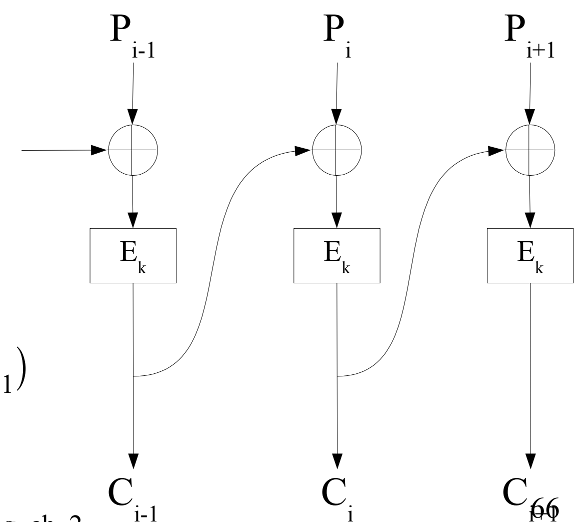
$$C_i = E_k(P_i \oplus C_{i-1}) \qquad P_i = C_{i-1} \oplus D_k(C_i)$$

CBC Encryption

- The *plaintext* of block *i* is XOR'ed with the *ciphertext* of block *i-1* before it is encrypted

so:

$$C_i = E_k(P_i \oplus C_{i-1})$$



CBC: The Point

- Make two identical plaintext blocks encrypt to two different ciphertext blocks
- but if all of the preceeding ciphertext blocks are also the same, we're in trouble
- what if the entire message is the same?

CBC: The Point

- Make two identical plaintext blocks encrypt to two different ciphertext blocks
- but if all of the preceeding ciphertext blocks are also the same, we're in trouble
- what if the entire message is the same?
  - the entire ciphertext will be the same
- fix?

Initialization Vector

- To form the ciphertext of block *i*
  - XOR the plaintext of block *i* with the ciphertext of block *i-1*.
- What do we do with the 1st block?

Initialization Vector

- To form the ciphertext of block *i*
  - XOR the plaintext of block *i* with the ciphertext of block *i-1*.
- What do we do with the 1st block?
  - use block of random data known to both the sender and receiver
  - called **initialization vector (IV)**

CBC: Error Propagation

- What happens if there is an error in block *i*?
  - Error affects block *i* and block *i+1*?
- Why does it only affect block *i* and *i+1* and nothing later?

CBC Error Propagation

- block *i* is flawed: so C_i becomes C_i^1

$$C_{i-1} \oplus D_k(C_i^1) = P_i^1$$

CBC: The Point

- Make two identical plaintext blocks encrypt to two different ciphertext blocks
- but if all of the preceeding ciphertext blocks are also the same, we're in trouble
- what if the entire message is the same?
  - the entire ciphertext will be the same
- fix?
  - use different IVs

CBC: Error Propagation

- What happens if there is an error in block *i*?

CBC Error Propagation

- block *i* is flawed: so C_i becomes C_i^1

$$C_{i-1} \oplus D_k(C_i^1) = P_i^1$$

- block *i*+1 arrives

$$C_i^1 \oplus D_k(C_{i+1}) = C_i^1 \oplus P_{i+1} \oplus C_i = P_{i+1}^1$$

- block *i+2* arrives

OK

$$C_{i+1} \oplus D_k(C_{i+2}) = C_{i+1} \oplus P_{i+2} \oplus C_{i+1} = P_{i+2}$$

CBC Security Problems

- Attacker can still
  - add blocks to the end
  - modify particular bits in block *i* to affect plaintext in block *i+1*
- Point of CBC is to hide patterns
  - but birthday paradox says that even with CBC, duplicates will eventually happen → $2^{\text{blockSize}/2}$ blocks
  - for 64 bit blocks → 32 gigabytes

CBC Error Propagation

- block *i* is flawed: so C_i becomes C_i^1

$$C_{i-1} \oplus D_k(C_i^1) = P_i^1$$

- block *i*+1 arrives

$$C_i^1 \oplus D_k(C_{i+1}) = C_i^1 \oplus P_{i+1} \oplus C_i = P_{i+1}^1$$

CBC Error Propagation

- block *i* is flawed: so C_i becomes C_i^1

$$C_{i-1} \oplus D_k(C_i^1) = P_i^1$$

- block *i*+1 arrives

$$C_i^1 \oplus D_k(C_{i+1}) = C_i^1 \oplus P_{i+1} \oplus C_i = P_{i+1}^1$$

garbage again

Problem

- Suppose that we're doing telnet and we'd like to use CBC mode?
- Blocks are 64 bits

  1)We'd have either:
  - wait until we've typed several characters OR
  - pad each so that we have a full block

  2) We'd have to transmit 64-bits of ciphertext for every 8 bits of plaintext

Cipher Feedback Mode (CFB)

- Stream ciphers – can encrypt small amounts of plaintext
- Block ciphers – have to encrypt an entire block's worth of data
- CFB Idea: implement a block cipher as a type of stream cipher

CBC Security Problems

- Attacker can still
  - add blocks to the end
  - modify particular bits in block *i* to affect plaintext in block *i+1*
- Point of CBC is to hide patterns
  - but birthday paradox says that even with CBC, duplicates will eventually happen → $2^{blockSize/2}$ blocks
  - for 64 bit blocks → 32 gigabytes

$$\frac{2^{32}\ blocks * 64\ bits/block}{8\ bits/byte * 1024\ bits/Kbit * 1024\ Kbits/Mbit * 1024\ Mbits/Gbit} = 32\ GBytes$$

Problem

- Suppose that we're doing telnet and we'd like to use CBC mode?

CFB: How it works

1) Fill up a block sized IV

2) Encrypt it

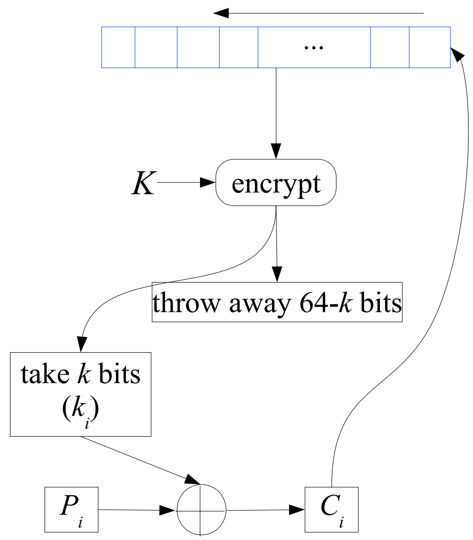
3) Take the left-most k bits

- throw away the rest
- left bits are next bits of keystream

4) XOR with plaintext

5) Result is ciphertext

6) Feed it back into queue



CFB: How it works

1) Fill up a block sized IV

2) Encrypt it

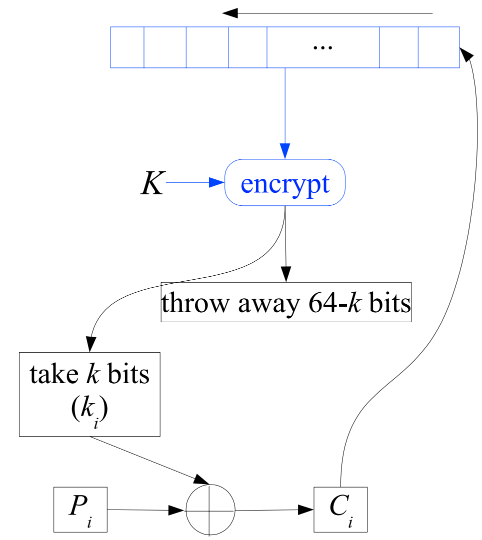
3) Take the left-most k bits

- throw away the rest
- left bits are next bits of keystream

4) XOR with plaintext

5) Result is ciphertext

6) Feed it back into queue

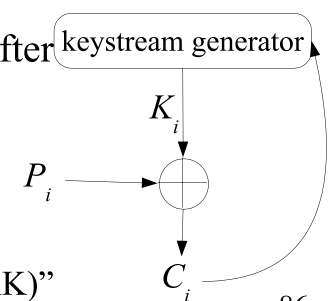


An Idea

- Take your name
- Encrypt it with DES → looks like random garbage
- Can take the garbage, and encrypt that too
  - Looks like more random garbage
- The point:
  - Can use garbage as a key stream
  - Reproduceable

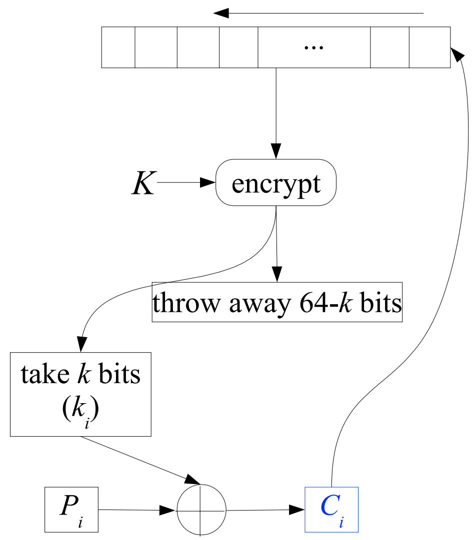
Self-Synchronizing Stream Ciphers

- Recall how stream ciphers work
- Self-synchronizing stream ciphers:
  - each bit in the keystream is a function of n previous bits of the ciphertext
- "self-synchronizing" because after receiver's key generator has received n bits of text, it is synchronized with the sender's keystream generator
- military: "ciphertext auto key (CTAK)"



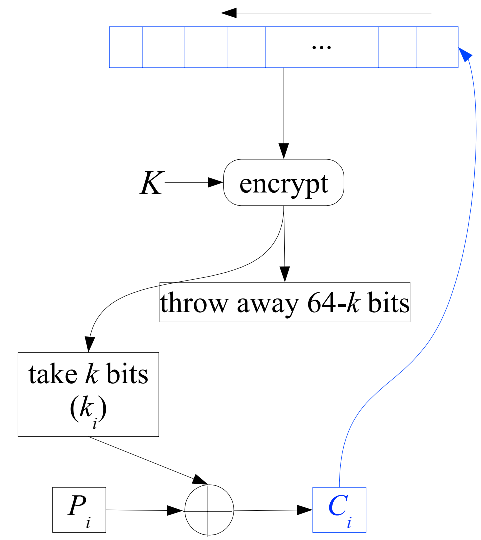
CFB: How it works

1) Fill up a block sized IV
2) Encrypt it
3) Take the left-most *k* bits
   - throw away the rest
   - left bits are next bits of keystream
4) XOR with plaintext
5) Result is ciphertext
6) Feed it back into queue



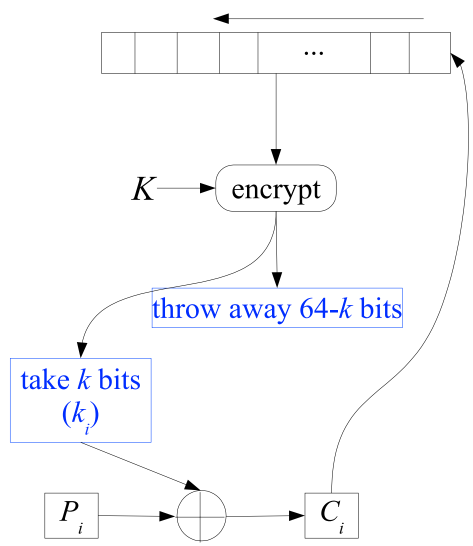
CFB: How it works

1) Fill up a block sized IV
2) Encrypt it
3) Take the left-most *k* bits
   - throw away the rest
   - left bits are next bits of keystream
4) XOR with plaintext
5) Result is ciphertext
6) Feed it back into queue



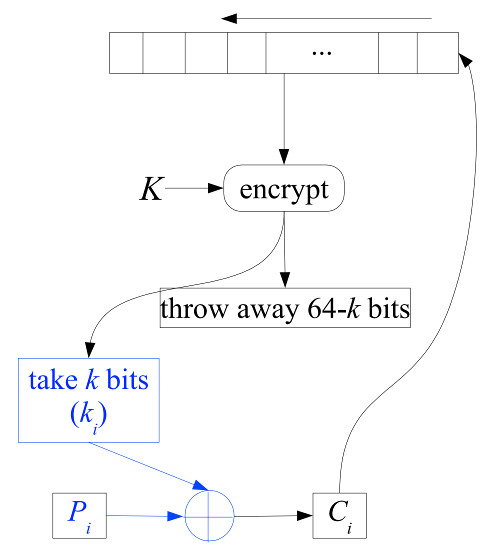
CFB: How it works

1) Fill up a block sized IV
2) Encrypt it
3) Take the left-most *k* bits
   - throw away the rest
   - left bits are next bits of keystream
4) XOR with plaintext
5) Result is ciphertext
6) Feed it back into queue



CFB: How it works

1) Fill up a block sized IV
2) Encrypt it
3) Take the left-most *k* bits
   - throw away the rest
   - left bits are next bits of keystream
4) XOR with plaintext
5) Result is ciphertext
6) Feed it back into queue



CFB: Additional Notes

- When we take k bits, it's called *k-bit* CFB
- If k is the block size

$$C_i = P_i \oplus E_K(C_{i-1})$$

$$P_i = C_i \oplus E_K(C_{i-1})$$

CFB: Additional Notes

- When we take k bits, it's called *k-bit* CFB
- If k is the block size

$$C_i = P_i \oplus E_K(C_{i-1})$$

$$P_i = C_i \oplus E_K(C_{i-1})$$

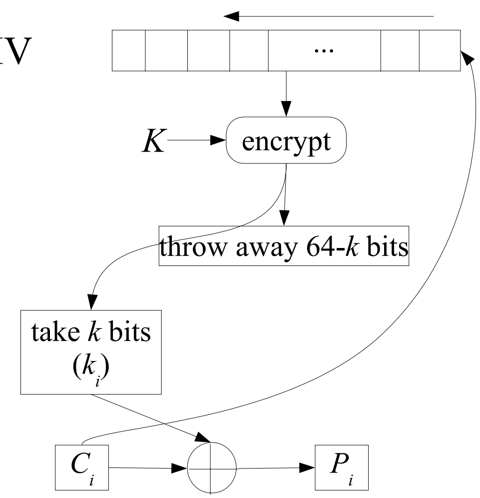
Really E_K *not* D_K

CFB Decryption

- Recall: with stream ciphers
  - decrypt by XOR'ing the keystream with ciphertext
- CFB decryption:
  - receiver starts with the same IV
  - encrypt IV
  - select left-most k bits
  - XOR with ciphertext to recover plaintext
  - feed k bits of ciphertext back into queue

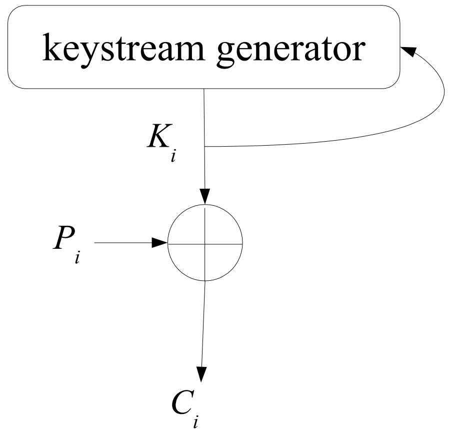
CFB Decryption

- receiver starts with same IV
- encrypt IV
- select left-most k bits
- XOR with ciphertext to recover plaintext
- feed k bits of ciphertext back into queue

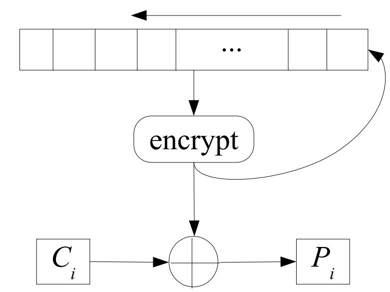


Synchronous Stream Ciphers

- *Feedback comes from the keystream itself*



Output Feedback Mode



- *Idea*: run a block cipher as a synchronous stream cipher
- *Encryption*

$$C_i = P_i \oplus S_i$$
$$S_i = E_K(S_{i-1})$$

- *Decryption:*

$$P_i = C_i \oplus S_i$$
$$S_i = E_K(S_{i-1})$$

Update internal state

- IV should be unique, but doesn't have to be secret

CFB Errors

- Plaintext error:
  - affects *all* ciphertext
  - but fixes itself in decryption
- Ciphertext error:
  - causes a single error in corresponding plaintext
  - enters the feedback register
    - causing all ciphertext to be garbled until it leaves the queue
  - then everything is fine
- Attacker can add to the end

Synchronous Stream Ciphers

- *Recall*: Self-synchronizing stream ciphers
  - keystream generated by feeding back previous ciphertext
- Synchronous stream ciphers:
  - keystream totally independent of:
    - previous plaintext
    - previous ciphertext
  - why bother?
    - can pre-compute the keystream
    - no error propagation

OFB Security Problems

- Don't want keystream to repeat
- Should chose the feedback size to be the same as the block size
  - *e.g.* so if you're using a 64-bit block size, you should use 64-bit OFB
  - the smaller the block size, the more often the keystream will repeat

Counter Mode (CTR)

- Use sequence numbes as input to the algorithm
- Just like OFB, except:
  - you don't feed the output back into the shift register
  - just add a counter to the register
- It doesn't matter
  - what the starting counter value is
  - what the increment amount is
- Only requirement: sender and receiver must agree

OFB Errors

- Error propagation
  - no error extension
  - single bit error in ciphertext causes single bit error in corresponding plaintext
- What happens if the sender and receiver lose sync?

OFB Errors

- Error propagation
  - no error extension
  - single bit error in ciphertext causes single bit error in corresponding plaintext
- What happens if the sender and receiver lose sync?
  - *disaster*
  - must be able to:
    - detect sync errors
    - automatically recover with a new IV to regain sync

Counter Mode (cont'd)

- Synchronization problems: same as OFB
- Why use it?
  - compute keystream in parallel
  - precompute the keystream
  - random access
  - simple

Summary

- Block ciphers encrypt chunks of plaintext at a time all with the same key
- Stream ciphers encrypt symbol *i* of the plaintext by combining it with symbol *i* of the key
- With very simple primitive ops (substitutions, permutations, shifts, XORs) DES was strong
- DES insecure by today's standards (56-bit keys too short). 3DES strong but slow.
- CBC, OFB, CFB, CTR → hide patterns
  - Additionally OFB, CFB, CTR fast
  - Get the best of both stream and block ciphers