# Cryptanalysis - Caesar Cipher

The cipher is based on shifting each character in the plaintext by x many characters in the alphabet using [modular arithmetic](https://andystabler.co.uk/blog/cryptanalysis-caesar-cipher/number-theory-part-1" \l "modular-arithmetic), where x is some predetermined shift value known only to the sender and receiver. The cipher uses a shared secret, which means it is a symmetric cryptosystem.

## Choosing a shift value

Let's say we've chosen the message we want to send to our recipient, Alice. The next thing we now need to do is pick a shift value that will be used when encrypting the message. If we use the English alphabet, we know there are 26 possible values for each character in the message. It makes sense therefore, to pick a shift value between 1 and 25, since after the modulo operation the result would be within that range anyway for all numbers (unless you chose a shift of 26 or 0, which would result in a shift of 0 after the modulo – don't do that).

## Encryption

For every character in the plaintext we need to add the shift value to its position in the alphabet to get the encrypted character. We are essentially shifting all characters in the plaintext along by x in the alphabet.

Ci=(Mi+shift)mod26Ci=(Mi+shift)mod26

Plain text: ILOVECAKE

Shift value: 3

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

|0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11|12|13|14|15|16|17|18|19|20|21|22|23|24|25|

Our plain text has the following positions in the alphabet:

I = 8

L = 11

0 = 14

V = 21

E = 4

C = 2

A = 0

K = 10

E = 4

We can now add our shift value to these positions and perform the modulo operation to get the encrypted values:

I = (8 + 3) mod 26 = 11 = L

L = (11 + 3) mod 26 = 14 = O

O = (14 + 3) mod 26 = 17 = R

V = (21 + 3) mod 26 = 24 = Y

E = (4 + 3) mod 26 = 7 = H

C = (2 + 3) mod 26 = 5 = F

A = (0 + 3) mod 26 = 3 = D

K = (10 + 3) mod 26 = 13 = N

E = (4 + 3) mod 26 = 7 = H

Ciphertext: LORYHFDNH

Now we've encrypted our secret message we can send it along to Alice, resting assured that any adversary seeing it on its way will be unable to understand the content (actually messages encrypted using Caesar's cipher are very easy to decipher without the key, but more on that later).

Here's the Java code to encrypt:

public static String encrypt(String plaintext, int shift) {

*// only interested in the alphabet*

plaintext = plaintext.replaceAll("[^a-zA-Z]", "").toUpperCase();

StringBuilder ciphertext = new StringBuilder();

for (char c : plaintext.toCharArray()) {

*// all upper case chars are in the ascii range 65-90.*

*// Subtracting A (65) from from the character gives us a value in the range of 0 25*

int newPos = c - 'A';

*// add the shift to the position*

newPos += shift;

*// perform the modulo to make sure the result is in the range of 0-25*

newPos = Math.floorMod(newPos, LetterFrequencyUtils.ALPHABET\_COUNT);

*// add A (65) to the value to get the uppercase character*

newPos += 'A';

ciphertext.append((char) newPos);

}

return ciphertext.toString();

}

## Decryption

Alice receives a message from Bob that has been encrypted using the Caesar cipher. Since both Alice and Bob previously agreed that they would use a shift value of 3, Alice can easily compute the original plaintext.

The method used to decrypt the ciphertext is the opposite to the one used for encrypting the plaintext. That is, instead of adding the shift value to each character, we need to subtract it, while still performing the modulo operation to wrap round to the end of the alphabet if result value is less than 0.

Mi=(Ci−shift)mod26Mi=(Ci−shift)mod26

The ciphertext characters have the following indexes in the alphabet:

L = 11

O = 14

R = 17

Y = 24

H = 7

F = 5

D = 3

N = 13

H = 7

By subtracting the shift value from each character in the cipher text we can calculate the original values:

L = (11 - 3) mod 26 = 8 = I

O = (14 - 3) mod 26 = 11 = L

R = (17 - 3) mod 26 = 14 = O

Y = (24 – 3) mod 26 = 21 = V

H = (7 – 3) mod 26 = 4 = E

F = (5 – 3) mod 26 = 2 = C

D = (3 – 3) mod 26 = 0 = A

N = (13 – 3) mod 26 = 10 = K

H = (7 – 3) mod 26 = 4 = E

Plaintext: ILOVECAKE

There we have it- Bob encrypted his message, sent it to Alice and she successfully decrypted it to reveal his secret food of choice. But how do we know that the message wasn't intercepted? With this method of encryption, we don’t!

Here's the Java code to decrypt:

public static String decrypt(String ciphertext, int shift) {

*// only interested in the alphabet*

ciphertext = ciphertext.replaceAll("[^a-zA-Z]", "").toUpperCase();

StringBuilder plaintext = new StringBuilder();

for (char c : ciphertext.toCharArray()) {

*// all upper case chars are in the ascii range 65-90.*

*// Subtracting A (65) from from the character gives us a value in the range of 0 25*

int newPos = c - 'A';

*// subtract the shift from the position*

newPos -= shift;

*// perform the modulo to make sure the result is in the range of 0-25*

newPos = Math.floorMod(newPos, LetterFrequencyUtils.ALPHABET\_COUNT);

*// add A (65) to the value to get the uppercase character*

newPos += 'A';

plaintext.append((char) newPos);

}

return plaintext.toString();

}

## Cryptanalysis

Mallory knows that Bob will be sending his secret food of choice to Alice encrypted using Caesar's cipher. She also happens to know that Bob is allergic to nuts, and wants to maliciously modify his message to Alice to say his food preference is in fact the deadly peanut.

In order to decipher the message, she first needs to determine the shift value used. Thankfully (for her), the English alphabet has only 26 characters, which means the key space is 26. This means the shift value could be any integer between 0 and 26, but since 0 and 26 would result in no change of plaintext it is realistic to consider the possible keys as {1, 2, … , 25}.

### Brute Force

Brute force attacks are usually the simplest to implement, but have a very low computational efficiency. This, however, is not so bad in the case where our key space is so small.

Mallory needs to apply every possible shift value to each character in the ciphertext until she finds the value that successfully transforms it into the original plaintext.

The method I demonstrate here requires human intervention in that for every possible shift value the decryption is applied and the result printed. The user (Mallory) must then look at each result and decide which is the original plain text.

public static void bruteForceAttack(String ciphertext) {

*// shift of 0 or 26 would result in no change*

for (int shift = 1; shift < LetterFrequencyUtils.ALPHABET\_COUNT; shift++)

System.out.println("shift: " + shift + ", " + decrypt(ciphertext, shift));

}

Mallory executes the brute force attack, and the following is printed:

Brute Force Attack

shift: 1, KNQXGECMG

shift: 2, JMPWFDBLF

shift: 3, **ILOVECAKE**

shift: 4, HKNUDBZJD

shift: 5, GJMTCAYIC

shift: 6, FILSBZXHB

shift: 7, EHKRAYWGA

shift: 8, DGJQZXVFZ

shift: 9, CFIPYWUEY

shift: 10, BEHOXVTDX

shift: 11, ADGNWUSCW

shift: 12, ZCFMVTRBV

shift: 13, YBELUSQAU

shift: 14, XADKTRPZT

shift: 15, WZCJSQOYS

shift: 16, VYBIRPNXR

shift: 17, UXAHQOMWQ

shift: 18, TWZGPNLVP

shift: 19, SVYFOMKUO

shift: 20, RUXENLJTN

shift: 21, QTWDMKISM

shift: 22, PSVCLJHRL

shift: 23, ORUBKIGQK

shift: 24, NQTAJHFPJ

shift: 25, MPSZIGEOI

Now that Mallory knows the shift value used was 3, she can encrypt the message "PEANUTS" with the same shift value and send it along to Alice.

System.out.println(CaesarCipher.encrypt("PEANUTS", 3)); *// prints SHDQXWV*

### Frequency Analysis

The brute force approach seemed to work rather well. Mallory simply needed to glance at 25 strings, each the result of shifting characters in the ciphertext down the alphabet by an incrementing shift value. She could easily see the plaintext because it would be the only string that resembled the English language. But say our alphabet wasn't 26 characters long, say we were using some alien alphabet, which was hundreds of characters long- Mallory would potentially have to look at hundreds of strings to determine which one was the plaintext, and what if she was trying to decode hundreds of these messages, all with random shift values? Ain't nobody got time for that!

By comparing the frequency of characters that we'd expect to find in the language used with those in the ciphertext, we are able to calculate a best guess at the shift value used.

A chi-squared test is a way of measuring how likely it is that any difference between two sets of values occurred by chance. This happens to be very useful for calculating the shift value. We can have one set of values containing the expected number of occurrences of characters for a string the length of the ciphertext, and another set containing the actual number of occurrences. We can then shove the values into the chi-squared formula:

∑i=Zi=A(Ci−Ei)2Ei∑i=Ai=Z(Ci−Ei)2Ei

Here, CiCi is the number of times the iith letter occurred in the ciphertext, and EiEi is the expected number of times the iith letter should occur in a string whose length is that of the ciphertext.

We can call the decrypt function and make a note of the chi-square for increasing shift values. After we've tried all shifts, the most likely key length is the one that produced the lowest chi-squared value.

public static String frequencyAnalysis(String ciphertext) {

ciphertext = ciphertext.replaceAll("[^a-zA-Z]", "");

int bestShift = calculateShift(ciphertext);

return decrypt(ciphertext, bestShift);

}

public static int calculateShift(String ciphertext) {

ciphertext = ciphertext.replaceAll("[^a-zA-Z]", "");

int shift = 0;

double fitness = Integer.MAX\_VALUE;

for (int i = 0; i < LetterFrequencyUtils.ALPHABET\_COUNT; i++) {

*// shift the ciphertext by i characters and compute the chi-square for the result*

double tempFitness = LetterFrequencyUtils.chiSquareAgainstEnglish(CaesarCipher.decrypt(ciphertext, i));

*// if the chi-square was lower than the previous value, make a note of it*

if (tempFitness < fitness) {

fitness = tempFitness;

shift = i;

}

}

return shift;

}

To demonstrate the frequency analysis, in the following section I encrypt Douglas Adams' [The Restaurant at the End of the Universe](http://www.textfiles.com/stories/hitch2.txt) using a shift of 3, and then try to decrypt it without providing the shift value:

System.out.println("FREQUENCY ANALYSIS");

System.out.println("Encrypting The Restaurant at the End of the Universe...");

String bigText = CaesarCipher.readFile(CaesarCipher.class.getResource("/hitch2.txt"));

String bigCipherText = CaesarCipher.encrypt(bigText, 3);

System.out.println(bigCipherText.substring(0, 200) + "...");

System.out.println("Performing frequency analysis...");

String decryptedBigCipherText = CaesarCipher.frequencyAnalysis(bigCipherText);

System.out.println(decryptedBigCipherText.substring(0, 200) + "...");

Which prints:

FREQUENCY ANALYSIS

Encrypting The Restaurant at the End of the Universe...

GRXJODVDGDPVWKHUHVWDXUDQWDWWKHHQGRIWKHXQLYHUVHGRXJODVDGDPVWKHKLWFKKLNHUVJXLGHWRWKHJDODABGRXJODVDGDPVWKHUHVWDXUDQWDWWKHHQGRIWKHXQLYHUVHGRXJODVDGDPVOLIHWKHXQLYHUVHDQGHYHUBWKLQJGRXJODVDGDPVVRORQJDQGWKDQN...

Performing frequency analysis...

DOUGLASADAMSTHERESTAURANTATTHEENDOFTHEUNIVERSEDOUGLASADAMSTHEHITCHHIKERSGUIDETOTHEGALAXYDOUGLASADAMSTHERESTAURANTATTHEENDOFTHEUNIVERSEDOUGLASADAMSLIFETHEUNIVERSEANDEVERYTHINGDOUGLASADAMSSOLONGANDTHANK...

Since this method of deciphering the message is completely automatic, it can be very useful. However, if the message is a short string, it may not show frequencies that match the language. For example, the string "HELLOWORLD" has L as the most common letter, followed by O, and when encrypted with a shift of 3, and then decrypted using the frequency analysis method it becomes "EBIILTLOIA", which is clearly wrong.

# Vigenère Cipher

Vigenere Cipher is a method of encrypting alphabetic text. It uses a simple form of [polyalphabetic substitution](https://en.wikipedia.org/wiki/Polyalphabetic_cipher" \t "https://www.geeksforgeeks.org/vigenere-cipher/_blank). A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets .The encryption of the original text is done using the [Vigenère square or Vigen](https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher" \l "/media/File:Vigen%C3%A8re_square_shading.svg)*[èr](https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher" \l "/media/File:Vigen%C3%A8re_square_shading.svg)*[e table](https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher" \l "/media/File:Vigen%C3%A8re_square_shading.svg).

* The table consists of the alphabets written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible[Caesar Ciphers](https://www.geeksforgeeks.org/caesar-cipher/).
* At different points in the encryption process, the cipher uses a different alphabet from one of the rows.
* The alphabet used at each point depends on a repeating keyword.

**Example:**

Input : Plaintext : GEEKSFORGEEKS

Keyword : AYUSH

Output : Ciphertext : GCYCZFMLYLEIM

For generating key, the given keyword is repeated

in a circular manner until it matches the length of

the plain text.

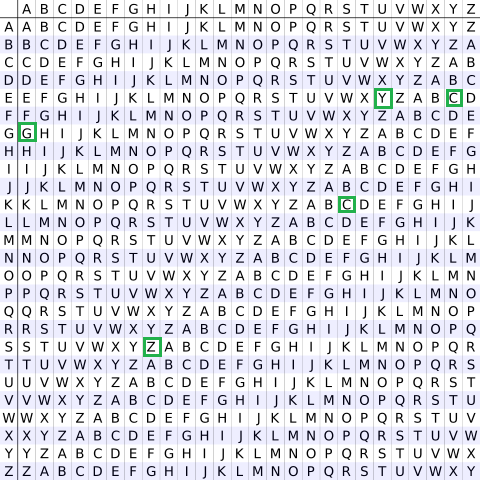
The keyword "AYUSH" generates the key "AYUSHAYUSHAYU"

The plain text is then encrypted using the process

explained below.

**Encryption**  
The first letter of the plaintext, G is paired with A, the first letter of the key. So use row G and column A of the Vigenère square, namely G. Similarly, for the second letter of the plaintext, the second letter of the key is used, the letter at row E and column Y is C. The rest of the plaintext is enciphered in a similar fashion.

**Table to encrypt –**



**Decryption**  
Decryption is performed by going to the row in the table corresponding to the key, finding the position of the ciphertext letter in this row, and then using the column’s label as the plaintext. For example, in row A (from AYUSH), the ciphertext G appears in column G, which is the first plaintext letter. Next we go to row Y (from AYUSH), locate the ciphertext C which is found in column E, thus E is the second plaintext letter.

A more **easy implementation** could be to visualize Vigenère algebraically by converting [A-Z] into numbers [0–25].

**Encryption**

The the plaintext(P) and key(K) are added modulo 26.

Ei = (Pi + Ki) mod 26

**Decryption**

Di = (Ei - Ki + 26) mod 26

# Vernam Cipher in Cryptography

**Vernam Cipher** is a method of encrypting alphabetic text. It is simply a type of substitution cipher. In this mechanism we assign a number to each character of the Plain-Text, like (a = 0, b = 1, c = 2, … z = 25).

**Method to take key:**  
In Vernam cipher algorithm, we take a key to encrypt the plain text which length should be equal to the length of the plain text.

**Encryption Algorithm:**

1. Assign a number to each character of the plain-text and the key according to alphabetical order.
2. Add both the number (Corresponding plain-text character number and Key character number).
3. Subtract the number from 26 if the added number is grater than 26. otherwise left it.

**Example:**

**Plain-Text:** RAMSWARUPK**Key:** RANCHOBABA

Now according to our encryption algorithm we assign a number to each character of our plain-text and key.

**PT:** R A M S W A R U P K**NO:** 17 0 12 18 22 0 17 20 15 10

**KEY:** R A N C H O B A B A **NO:** 17 0 13 2 7 14 1 0 1 0

Now add the number of Plain-Text and Key and after doing the addition and subtraction operation (if required), we will get the corresponding Cipher-Text character number.

**CT-NO:** 34 0 25 20 29 14 18 20 16 10

In this case, there are two numbers which are greater than the 26 so we have to subtract 26 from them and after applying the subtraction operation the new Cipher text character numbers are as follow:

**CT-NO:** 8 0 25 20 3 14 18 20 16 10

New Cipher-Text is after getting the corresponding character from the number.

**CIPHER-TEXT:** I A Z U D O S U Q K

# **Stream Ciphers**

A Stream Cipher is used for symmetric key cryptography, or when the same key is used to encrypt and decrypt data. Stream Ciphers encrypt pseudorandom sequences with bits of plaintext in order to generate ciphertext, usually with XOR. A good way to think about Stream Ciphers is to think of them as generating one-time pads from a given state.

## **Definitions**

* A ****keystream**** is a sequence of pseudorandom digits which extend to the length of the plaintext in order to uniquely encrypt each character based on the corresponding digit in the keystream

## **One Time Pads**

A one time pad is an encryption mechanism whereby the entire plaintext is XOR'd with a random sequence of numbers in order to generate a random ciphertext. The advantage of the one time pad is that it offers an immense amount of security BUT in order for it to be useful, the randomly generated key must be distributed on a separate secure channel, meaning that one time pads have little use in modern day cryptographic applications on the internet. Stream ciphers extend upon this idea by using a key, usually 128 bit in length, in order to seed a pseudorandom keystream which is used to encrypt the text.

## **Types of Stream Ciphers**

### Synchronous Stream Ciphers

A Synchronous Stream Cipher generates a keystream based on internal states not related to the plaintext or ciphertext. This means that the stream is generated pseudorandomly outside of the context of what is being encrypted. A binary additive stream cipher is the term used for a stream cipher which XOR's the bits with the bits of the plaintext. Encryption and decryption require that the synchronus state cipher be in the same state, otherwise the message cannot be decrypted.

### Self-synchronizing Stream Ciphers

A Self-synchronizing Stream Cipher, also known as an asynchronous stream cipher or ciphertext autokey (CTAK), is a stream cipher which uses the previous N digits in order to compute the keystream used for the next N characters.

**Note**

Seems a lot like block ciphers doesn't it? That's because block cipher feedback mode (CFB) is an example of a self-synchronizing stream ciphers.

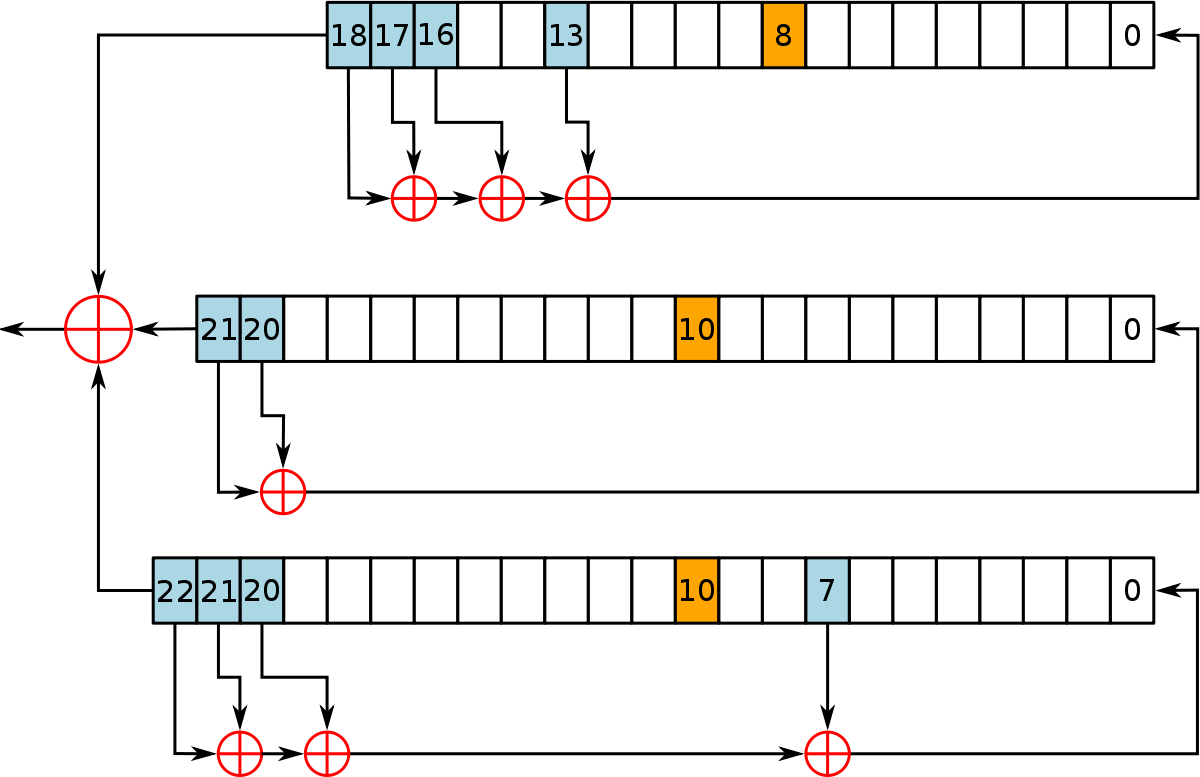
## **Stream Cipher Vulnerabilities**

### Key Reuse

The key tenet of using stream ciphers securely is to ****NEVER**** repeat key use because of the communative property of XOR. If C1 and C2 have been XOR'd with a key K, retrieving that key K is trivial because C1 XOR C2 = P1 XOR P2 and having an english language based XOR means that cryptoanalysis tools such as a character frequency analysis will work well due to the low entropy of the english language.

### Bit-flipping Attack

Another key tenet of using stream ciphers securely is considering that just because a message has been decrypted, it does not mean the message has not been tampered with. Because decryption is based on state, if an attacker knows the layout of the plaintext, a Man in the Middle (MITM) attack can flip a bit during transit altering the underlying ciphertext. If a ciphertext decrypts to 'Transfer $1000', then a middleman can flip a single bit in order for the ciphertext to decrypt to 'Transfer $9000' because changing a single character in the ciphertext does not affect the state in a synchronus stream cipher.



Block Cipher

A block cipher consists of two paired [algorithms](https://en.wikipedia.org/wiki/Algorithm" \o "Algorithm), one for encryption, *E*, and the other for decryption, *D*.[[4]](https://en.wikipedia.org/wiki/Block_cipher" \l "cite_note-4) Both algorithms accept two inputs: an input block of size *n* bits and a [key](https://en.wikipedia.org/wiki/Key_(cryptography)" \o "Key (cryptography)) of size *k* bits; and both yield an *n*-bit output block. The decryption algorithm *D* is defined to be the [inverse function](https://en.wikipedia.org/wiki/Inverse_function" \o "Inverse function) of encryption, i.e., *D* = *E*−1. More formally,[[5]](https://en.wikipedia.org/wiki/Block_cipher" \l "cite_note-HAC-5)[[6]](https://en.wikipedia.org/wiki/Block_cipher" \l "cite_note-modern-crypto-6) a block cipher is specified by an encryption function

{\displaystyle E\_{K}(P):=E(K,P):\{0,1\}^{k}\times \{0,1\}^{n}\rightarrow \{0,1\}^{n},}IMG_256

which takes as input a key *K* of bit length *k*, called the *key size*, and a bit string *P* of length *n*, called the *block size*, and returns a string *C* of *n* bits. *P* is called the [plaintext](https://en.wikipedia.org/wiki/Plaintext" \o "Plaintext), and *C* is termed the [ciphertext](https://en.wikipedia.org/wiki/Ciphertext" \o "Ciphertext). For each *K*, the function *EK*(*P*) is required to be an invertible mapping on {0,1}*n*. The inverse for *E* is defined as a function

{\displaystyle E\_{K}^{-1}(C):=D\_{K}(C)=D(K,C):\{0,1\}^{k}\times \{0,1\}^{n}\rightarrow \{0,1\}^{n},}IMG_257

taking a key *K* and a ciphertext *C* to return a plaintext value *P*, such that

{\displaystyle \forall K:D\_{K}(E\_{K}(P))=P.}IMG_258

For example, a block cipher encryption algorithm might take a 128-bit block of plaintext as input, and output a corresponding 128-bit block of ciphertext. The exact transformation is controlled using a second input – the secret key. Decryption is similar: the decryption algorithm takes, in this example, a 128-bit block of ciphertext together with the secret key, and yields the original 128-bit block of plain text.[[7]](https://en.wikipedia.org/wiki/Block_cipher" \l "cite_note-7)

For each key *K*, *EK* is a [permutation](https://en.wikipedia.org/wiki/Permutation" \o "Permutation) (a [bijective](https://en.wikipedia.org/wiki/Bijective" \o "Bijective) mapping) over the set of input blocks. Each key selects one permutation from the set of {\displaystyle (2^{n})!}IMG_259 possible permutations.

### Comparison Chart

| **BASIS FOR COMPARISON** | **BLOCK CIPHER** | **STREAM CIPHER** |
| --- | --- | --- |
| Basic | Converts the plain text by taking its block at a time. | Converts the text by taking one byte of the plain text at a time. |
| Complexity | Simple design | Complex comparatively |
| No of bits used | 64 Bits or more | 8 Bits |
| Confusion and Diffusion | Uses both confusion and diffusion | Relies on confusion only |
| Algorithm modes used | ECB (Electronic Code Book) CBC (Cipher Block Chaining) | CFB (Cipher Feedback) OFB (Output Feedback) |
| Reversibility | Reversing encrypted text is hard. | It uses XOR for the encryption which can be easily reversed to the plain text. |
| Implementation | Feistel Cipher | Vernam Cipher |

### Definition of Block Cipher

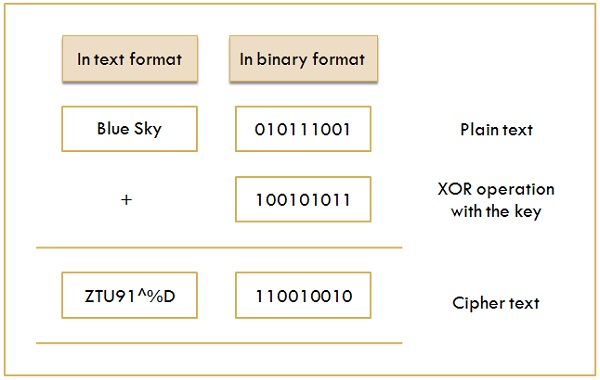
Block Cipher takes a message and break it into a fixed size of blocks and converts one block of the message at an instant. ****For example,**** we have a message in plain text “STREET\_BY\_STREET” required to be encrypted. Using bock cipher, “STREET” must be encrypted at first, followed by “\_BY\_” and finally at last “STREET”.  
In actual practice, communication takes place only in bits. Therefore, STREET actually means the binary equivalent of ASCII character of STREET. Subsequently, any algorithm encrypts these; the resultant bits are transformed back into their ASCII equivalent.

An evident problem regarding the usage of Block ciphers is ****repeating text****, for which the same cipher is generated. Hence, it would give a hint to the cryptanalyst which makes it is easier to figure out the recurring strings of plain text. As a result, it can reveal the whole message.

To overcome from this problem the ****chaining mode**** is used. In this technique, the preceding block of cipher text is mixed with the current block, so as to the cipher text vague, this avoids the recurring patterns of blocks with the same content.

### Definition of Stream Cipher

Stream Cipher typically encrypts one byte of the message at that moment instead of using blocks. Let’s take an ****example,**** suppose the original message (plain text) is “blue sky” in ASCII (i.e. text format). When you convert these ASCII into equivalent binary values, it will give the output in 0’s and 1’s form. Let it be translated in 010111001.



For encryption and decryption, a ****pseudorandom bit generator**** is used in which a key and plain text are loaded. A pseudorandom bit generator creates a stream of 8-bit numbers that are seemingly random known as ****keystream****. Let the input key is 100101011. Now the key and plaintext are XORed. The XOR logic is simple to understand.  
XOR produces an output of 1 when one input is 0, and the other is 1. The output is 0 if either both the inputs are 0 or both the inputs are 1.

****Confusion**** is a method which guarantees that a cipher text gives no clue about the original plain text.  
****Diffusion**** is a strategy used to enhance the redundancy of the plain text by spreading it across rows and columns.

## Key Differences Between Block Cipher and Stream Cipher

1. Block cipher technique involves encryption of one block of text at a time, i.e. singly. Similarly, decrypt the text by taking one block after another. In contrast, Stream cipher technique involves encryption and decryption of one byte of the text at a time.
2. Block cipher uses both confusion and diffusion while stream cipher relies only on confusion.
3. The usual size of the block could be 64 or 128 bits in the Block cipher. As against, 1 byte (8 bits) at a time is converted in the stream cipher.
4. Block cipher uses ****ECB (Electronic Code Book)**** and ****CBC (Cipher Block Chaining)**** algorithm modes. On the contrary, Stream cipher uses ****CFB (Cipher Feedback)**** and ****OFB (Output Feedback)**** algorithm modes.
5. Stream cipher uses XOR function for converting the plain text into cipher text, that is the reason why it is easy to reverse the XORed bits. Whereas Block cipher does not use XOR for doing so.
6. Block cipher uses the same key to encrypt each block while stream cipher uses a different key for each byte.

### Conclusion:

Block Cipher and Stream Cipher differ in the way in which plain text is encrypted and decrypted. The idea behind block cipher is to divide the plain text into blocks further encrypt those blocks. While stream cipher converts plain text bit by bit similar to stream.

AES(Advance encryption Standards)

The more popular and widely adopted symmetric encryption algorithm likely to be encountered nowadays is the Advanced Encryption Standard (AES). It is found at least six time faster than triple DES.

A replacement for DES was needed as its key size was too small. With increasing computing power, it was considered vulnerable against exhaustive key search attack. Triple DES was designed to overcome this drawback but it was found slow.

The features of AES are as follows −

* Symmetric key symmetric block cipher
* 128-bit data, 128/192/256-bit keys
* Stronger and faster than Triple-DES
* Provide full specification and design details
* Software implementable in C and Java

## Operation of AES

AES is an iterative rather than Feistel cipher. It is based on ‘substitution–permutation network’. It comprises of a series of linked operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations).

Interestingly, AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes. These 16 bytes are arranged in four columns and four rows for processing as a matrix −

Unlike DES, the number of rounds in AES is variable and depends on the length of the key. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys. Each of these rounds uses a different 128-bit round key, which is calculated from the original AES key.

The schematic of AES structure is given in the following illustration −



## Encryption Process

Here, we restrict to description of a typical round of AES encryption. Each round comprise of four sub-processes. The first round process is depicted below −



### Byte Substitution (SubBytes)

The 16 input bytes are substituted by looking up a fixed table (S-box) given in design. The result is in a matrix of four rows and four columns.

### Shiftrows

Each of the four rows of the matrix is shifted to the left. Any entries that ‘fall off’ are re-inserted on the right side of row. Shift is carried out as follows −

First row is not shifted.

Second row is shifted one (byte) position to the left.

Third row is shifted two positions to the left.

Fourth row is shifted three positions to the left.

The result is a new matrix consisting of the same 16 bytes but shifted with respect to each other.

### MixColumns

Each column of four bytes is now transformed using a special mathematical function. This function takes as input the four bytes of one column and outputs four completely new bytes, which replace the original column. The result is another new matrix consisting of 16 new bytes. It should be noted that this step is not performed in the last round.

### Addroundkey

The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key. If this is the last round then the output is the ciphertext. Otherwise, the resulting 128 bits are interpreted as 16 bytes and we begin another similar round.

## Decryption Process

The process of decryption of an AES ciphertext is similar to the encryption process in the reverse order. Each round consists of the four processes conducted in the reverse order −

* Add round key
* Mix columns
* Shift rows
* Byte substitution

Since sub-processes in each round are in reverse manner, unlike for a Feistel Cipher, the encryption and decryption algorithms needs to be separately implemented, although they are very closely related.

## AES Analysis

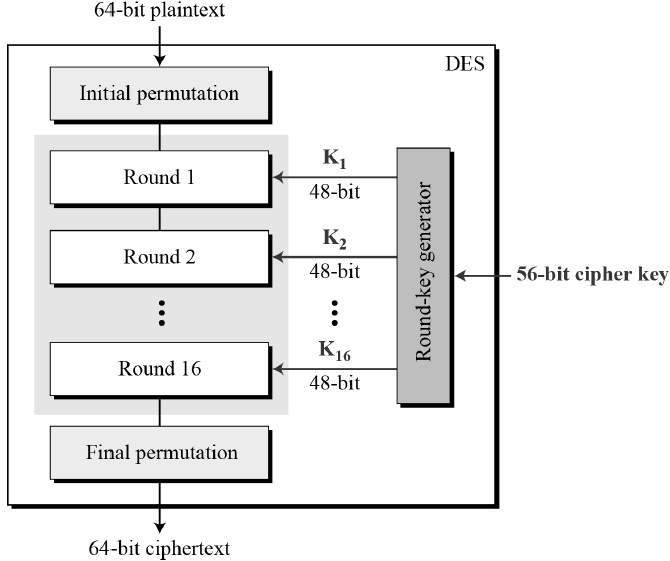
In present day cryptography, AES is widely adopted and supported in both hardware and software. Till date, no practical cryptanalytic attacks against AES has been discovered. Additionally, AES has built-in flexibility of key length, which allows a degree of ‘future-proofing’ against progress in the ability to perform exhaustive key searches.

However, just as for DES, the AES security is assured only if it is correctly implemented and good key management is employed.

# Data Encryption Standard

The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST).

DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only). General Structure of DES is depicted in the following illustration −

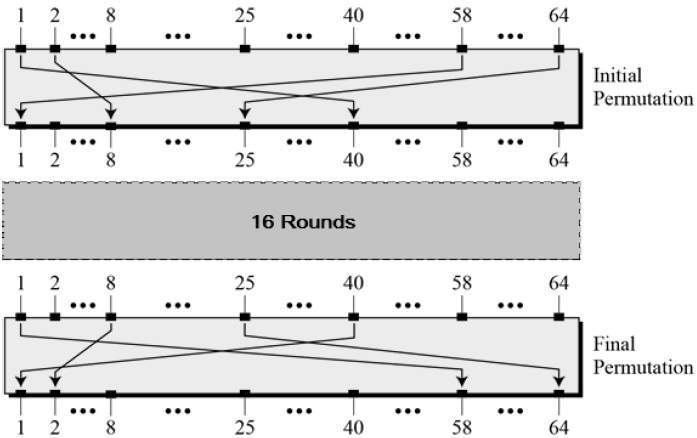


Since DES is based on the Feistel Cipher, all that is required to specify DES is −

* Round function
* Key schedule
* Any additional processing − Initial and final permutation

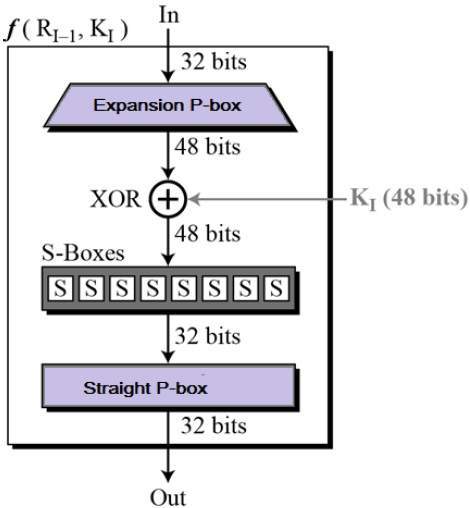
## Initial and Final Permutation

The initial and final permutations are straight Permutation boxes (P-boxes) that are inverses of each other. They have no cryptography significance in DES. The initial and final permutations are shown as follows −

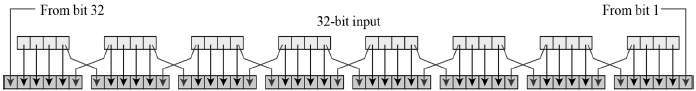


## Round Function

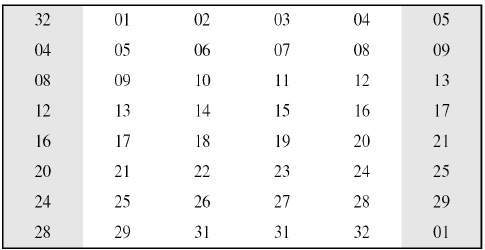
The heart of this cipher is the DES function, *f*. The DES function applies a 48-bit key to the rightmost 32 bits to produce a 32-bit output.



**Expansion Permutation Box** − Since right input is 32-bit and round key is a 48-bit, we first need to expand right input to 48 bits. Permutation logic is graphically depicted in the following illustration −

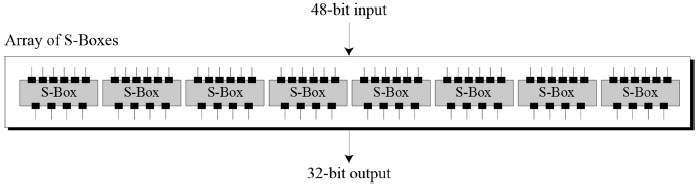


The graphically depicted permutation logic is generally described as table in DES specification illustrated as shown −

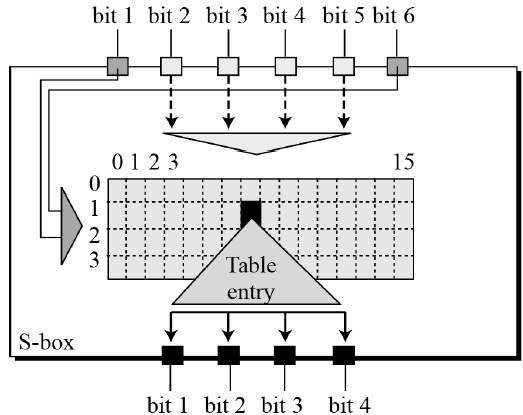


**XOR (Whitener).** − After the expansion permutation, DES does XOR operation on the expanded right section and the round key. The round key is used only in this operation.

**Substitution Boxes.** − The S-boxes carry out the real mixing (confusion). DES uses 8 S-boxes, each with a 6-bit input and a 4-bit output. Refer the following illustration −

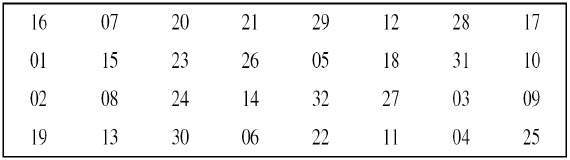


The S-box rule is illustrated below −



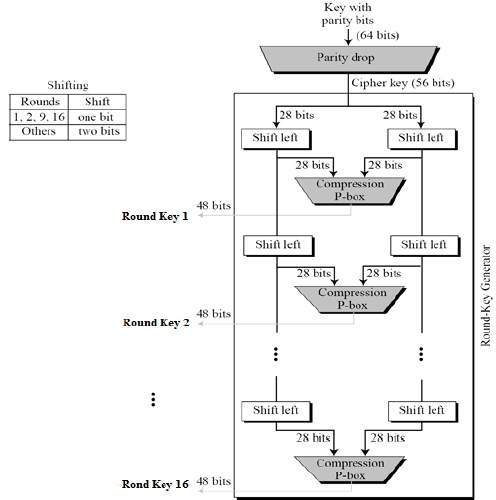
There are a total of eight S-box tables. The output of all eight s-boxes is then combined in to 32 bit section.

**Straight Permutation** − The 32 bit output of S-boxes is then subjected to the straight permutation with rule shown in the following illustration:



## Key Generation

The round-key generator creates sixteen 48-bit keys out of a 56-bit cipher key. The process of key generation is depicted in the following illustration −



The logic for Parity drop, shifting, and Compression P-box is given in the DES description.

## DES Analysis

The DES satisfies both the desired properties of block cipher. These two properties make cipher very strong.

**Avalanche effect** − A small change in plaintext results in the very great change in the ciphertext.

**Completeness** − Each bit of ciphertext depends on many bits of plaintext.

During the last few years, cryptanalysis have found some weaknesses in DES when key selected are weak keys. These keys shall be avoided.

DES has proved to be a very well designed block cipher. There have been no significant cryptanalytic attacks on DES other than exhaustive key search.