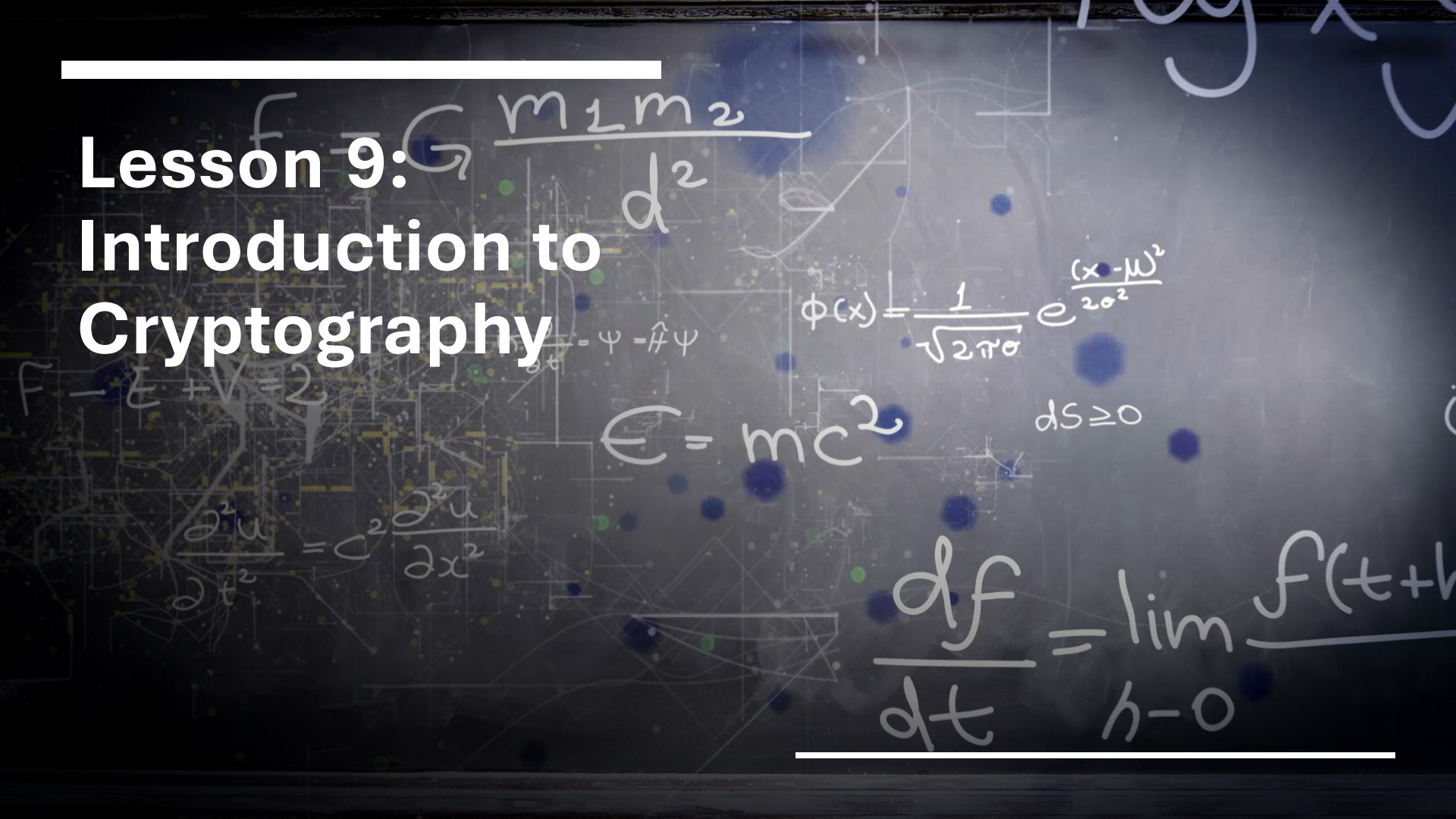
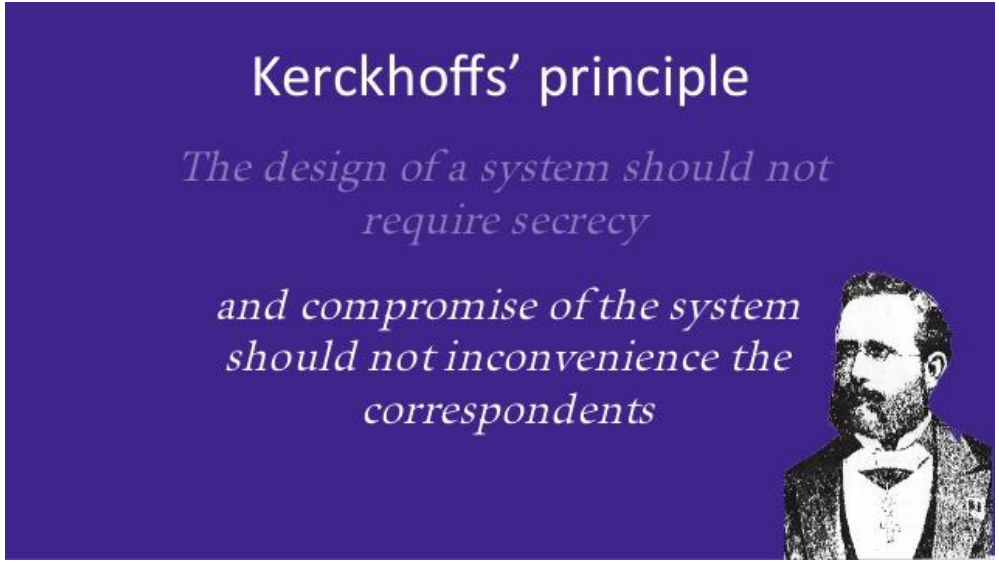


# Lesson 9: Introduction to Cryptography

# Kerckhoffs's Principle/Shannon’s Maxim




- Auguste Kerckhoff's first articulated this in the 1800s, stating that the security of a cipher depends only on the secrecy of the key, not the secrecy of the algorithm.

- Claude Shannon rephrased this, stating, “One ought to design systems under the assumption that the enemy will ultimately gain full familiarity with them.” This is referred to as Shannon’s maxim and states essentially the same thing Kerckhoffs’s principle.

# Information Entropy

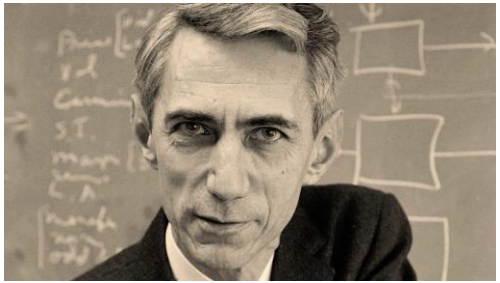
- In information theory, *entropy* is the amount of information in a given message.

- It is sometimes described as the number of bits required to communicate information.

- Another way to describe information entropy, used in many texts, is the measure of uncertainty in a message.

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

# Confusion and Diffusion



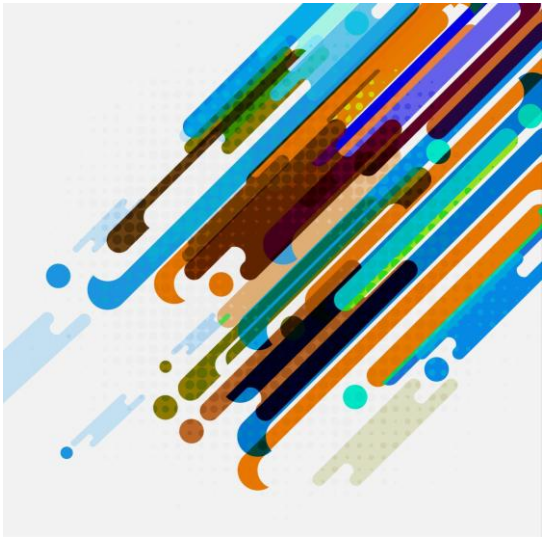
*Claude Shannon*

The concept of *confusion*, as relates to cryptography, was outlined in Claude Shannon's 1949 paper "Communication Theory of Secrecy Systems." In general, this concept attempts to make the relationship between the statistical frequencies of the cipher text and the actual key as complex as possible.

*Diffusion* literally means having changes to one character in the plain text affect multiple characters in the cipher text.

*Avalanche* means that a small change yields large effects in the output, like an avalanche. This is Horst Fiestel's variation on Claude Shannon's concept of diffusion.

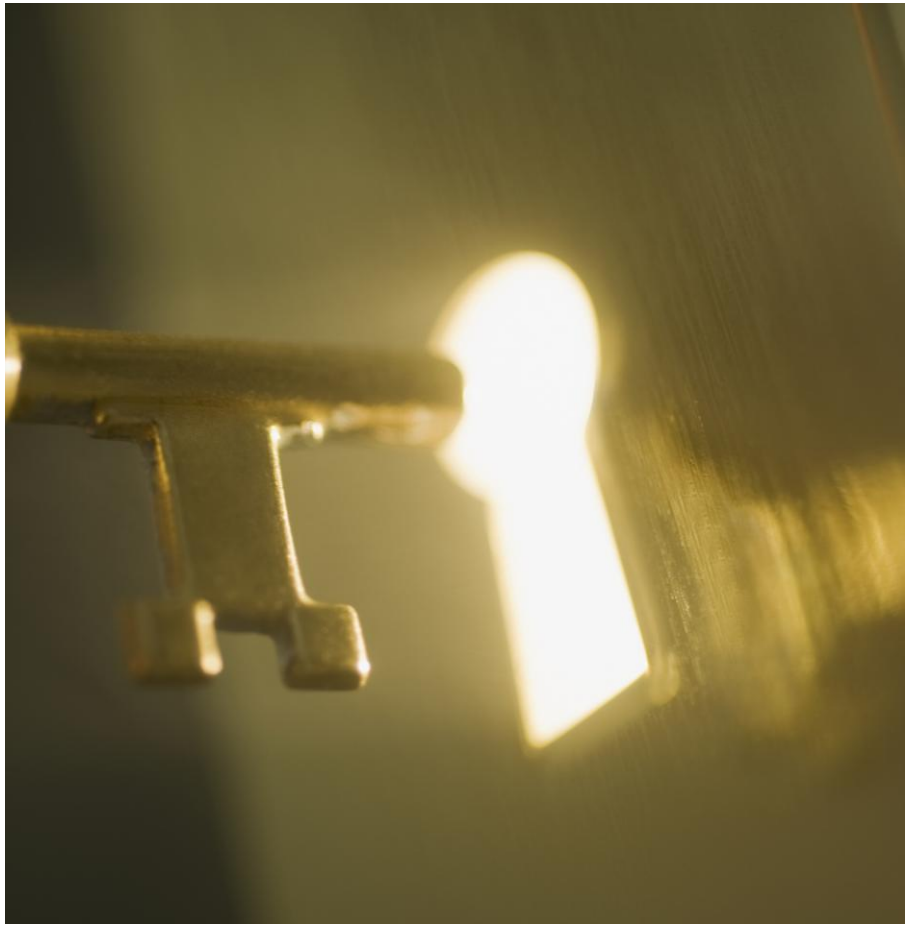
# Hamming Distance and Weight



- The Hamming distance is the number of characters (or bits) that are different between two strings. This can be expressed mathematically as

h(x, y)

- The concept of Hamming weight is closely related to Hamming distance. It compares the string to a string of all 0's. Put more simply, it is how many 1's are in the binary representation of a message. Some sources call this the *population count* or pop count.



# Key Schedule

In addition to the cipher key, all block ciphers have a second algorithm called a key schedule. The key schedule derives a unique key from the cipher, called a round key, for each round of the cipher.

Using a key schedule, each round uses a key that is slightly different from the key used in the previous round. Because both the sender (who encrypts the message) and the receiver (who must decrypt the message) are using the same cipher key as a starting point and are using the same key schedule, they will generate the same round keys.

# Feistel History

Feistel is used in DES, CAST-128, Blowfish, Twofish, RC5, and others.

First seen in IBM’s Lucifer algorithm (the precursor to DES).

Michael Luby and Charles Rackoff analyzed the Feistel cipher construction and proved that if the round function is a cryptographically secure pseudorandom function, then three rounds is sufficient to make the block cipher a pseudorandom permutation, while four rounds is sufficient to make it a “strong” pseudorandom permutation.

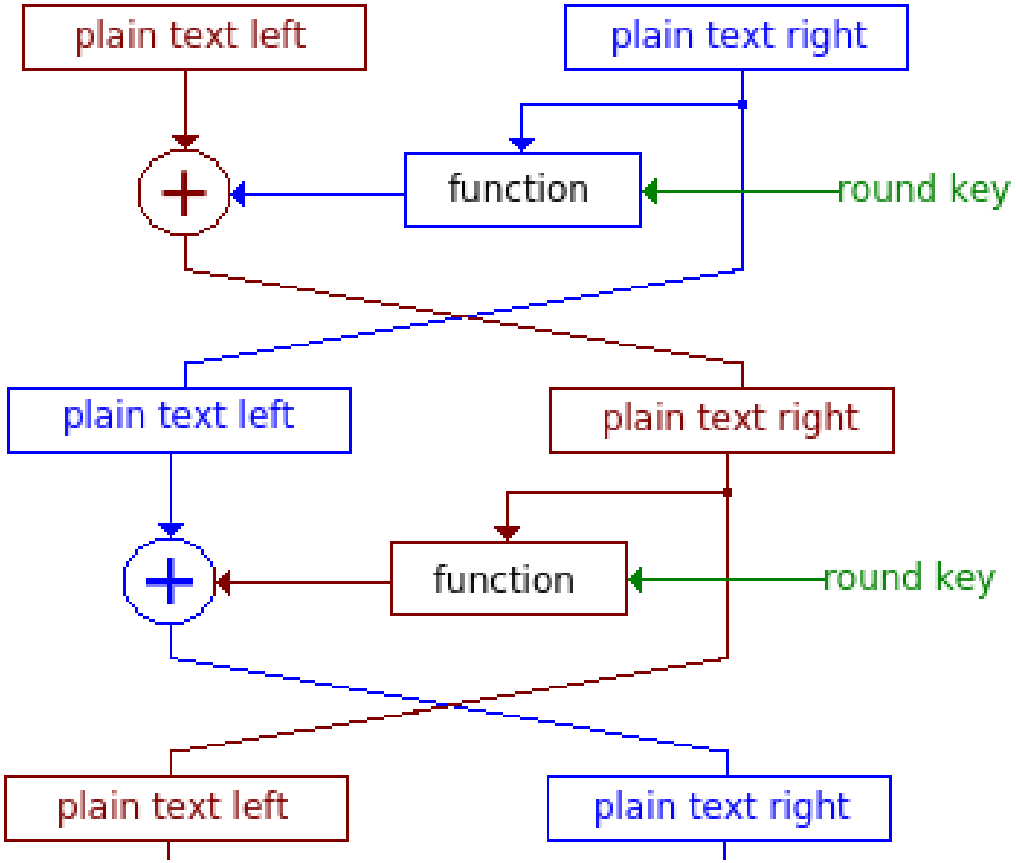
# THE FEISTEL FUNCTION

This function is named after its inventor, the German-born physicist and cryptographer Horst Feistel.

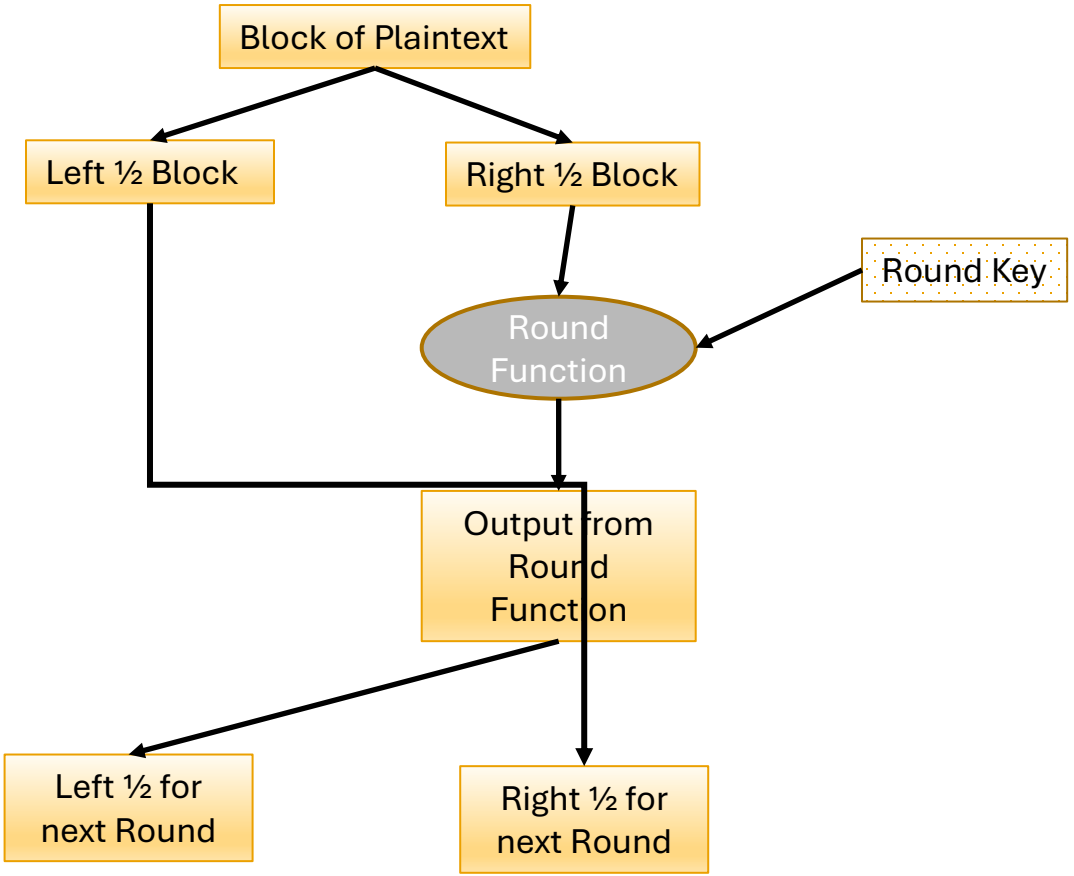
At the heart of many block ciphers is a Feistel function, which forms the basis for most block ciphers. This makes it one of the most influential developments in symmetric block ciphers. It is also known as a Feistel Network or a Feistel cipher.

# The Feistel Function

*Here is a general overview of a basic round of a Feistel cipher:*



# DES Illustrated




Left and Right Blocks are Swapped

This process goes on for 16 rounds!!

# Unbalanced Feistel

- A variation of the Feistel network is called an Unbalanced Feistel cipher.

- These ciphers use a modified structure, where $L_0$ and $R_0$ are not of equal lengths. This means that $L_0$ might be 32 bits and $R_0$ could be 64 bits (making a 96-bit block of text). This variation is used in the Skipjack algorithm.

# 3DES

- Eventually it became obvious that DES would no longer be secure. The U.S. government began a contest seeking a replacement cryptography algorithm. In the meantime, 3DES was created as an interim solution. Essentially it does DES three times, with three different keys.

- Triple DES uses a “key bundle” that comprises three DES keys: K1, K2, and K3. Each key is standard 56-bit DES key. There were some variations that would use the same key for K1 and K3, but three separate keys is considered the most secure.

# GOST

GOST is a DES-like algorithm developed by the Soviets in the 1970s. It was classified but was released to the public in 1994. It uses a 64-bit block and a key of 256 bits. It is a 32-round Feistel cipher.

GOST is an acronym for *gosudarstvennyy standart*, which translates into English as “state standard.”

The official designation is GOST 28147-89. It was meant as an alternative to the U.S. DES algorithm and has some similarities to DES.

# Blowfish

Blowfish is a symmetric block cipher.

This algorithm was published in 1993 by Bruce Schneier. This cryptography algorithm was intended as a replacement for DES.

Like DES, it is a 16-round Feistel cipher working on 64-bit blocks. However, unlike DES, it can have varying key sizes ranging from 32 bits to 448 bits.

Early in the cipher, the cipher key is expanded. Key expansion converts a key of at most 448 bits into several sub key arrays totaling 4168 bytes.

# Twofish

Twofish was one of the five finalists of the AES contest

It is related to the block cipher Blowfish, and Schneier was also part of the team that worked on this algorithm.

Twofish is a Feistel cipher that uses a 128-bit block size and key sizes of 128, 192, and 256 bits. It also has 16 rounds, like DES.

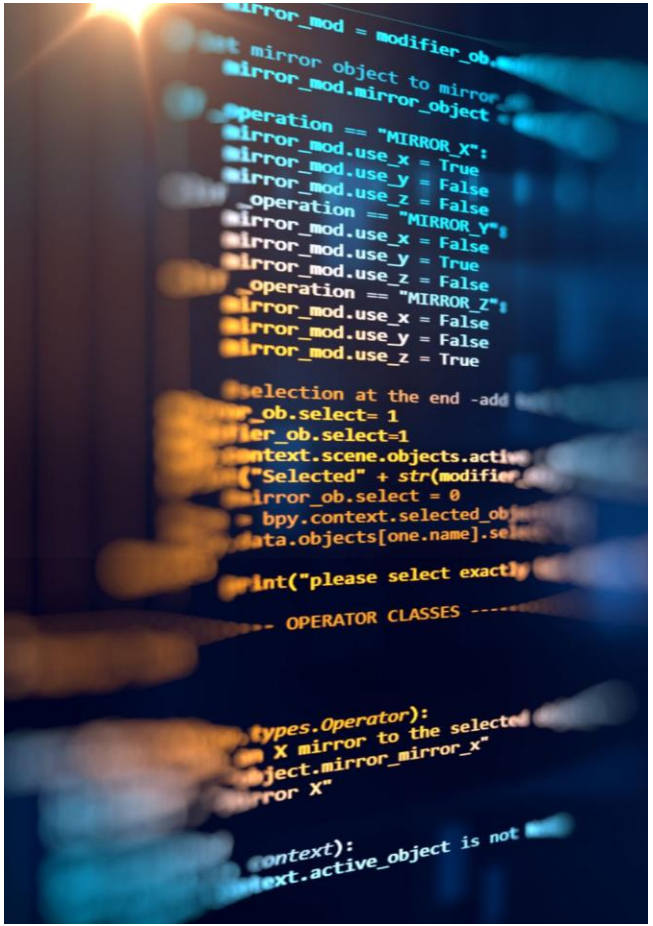
Like Blowfish, Twofish is not patented and is in the public domain. It can be used without restrictions by anyone.

# Electronic Codebook (ECB)

The most basic encryption mode is the electronic codebook (ECB) mode.

The message is divided into blocks and each block is encrypted separately.

However, if you submit the same plain text more than once, you always get the same cipher text. This gives attackers a place to begin analyzing the cipher to attempt to derive the key.

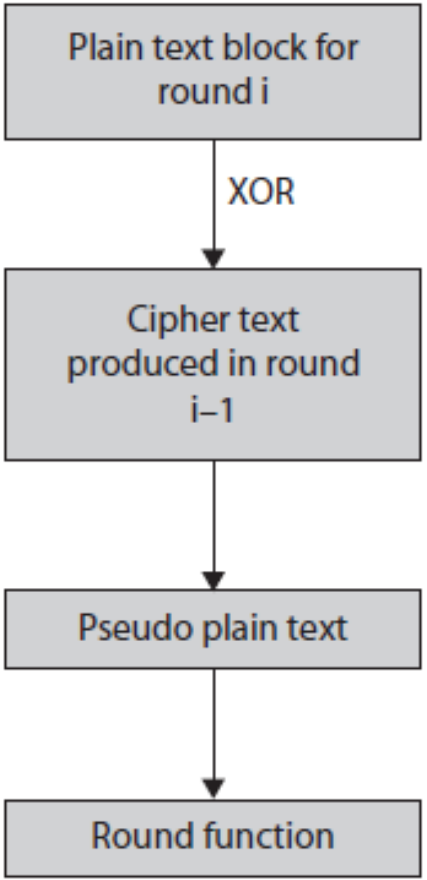


# Cipher-Block Chaining (CBC)

When using cipher-block chaining (CBC) mode, each block of plain text is XOR'd with the previous cipher text block before being encrypted.

This means there is significantly more randomness in the final cipher text.

This is much more secure than electronic codebook mode and is the most common mode.



# Propagating Cipher-Block Chaining (PCBC)

The propagating cipher-block chaining mode was designed to cause small changes in the cipher text to propagate indefinitely when decrypting and encrypting.

This method is sometimes called plain text cipher-block chaining.

The PCBC mode is a variation on the CBC mode of operation. It is important to keep in mind that the PCBC mode of encryption has not been formally published as a federal standard.

# AES

AES can have three different key sizes: 128, 192, or 256 bits.

The three different implementations of AES are referred to as AES 128, AES 192, and AES 256.

The block size is always 128 bit

The original Rijndael cipher allowed for variable block and key sizes in 32-bit increments. However, the U.S. government uses these three key sizes with a 128-bit block as the standard for AES.

# AES Steps

**Key Expansion:** Round keys are derived from the cipher key using Rijndael's key schedule

**Initial Round:** AddRoundKey—each byte of the state is combined with the round key using a bitwise XOR

**Rounds:**

- 1. SubBytes—a nonlinear substitution step in which each byte is replaced with another according to a lookup table.
- 2. ShiftRows—a transposition step in which each row of the state is shifted cyclically a certain number of steps.
- 3. MixColumns—a mixing operation that operates on the columns of the state, combining the four bytes in each column.
- 4. AddRoundKey

**Final Round (no MixColumns):**

- 1. SubBytes
- 2. ShiftRows
- 3. AddRoundKey

# S-Box (the only one in AES)

(a) S-box

| | | y | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| x | 0 | 63 | 7C | 77 | 7B | F2 | 6B | 6F | C5 | 30 | 01 | 67 | 2B | FE | D7 | AB | 76 |
| | 1 | CA | 82 | C9 | 7D | FA | 59 | 47 | F0 | AD | D4 | A2 | AF | 9C | A4 | 72 | C0 |
| | 2 | B7 | FD | 93 | 26 | 36 | 3F | F7 | CC | 34 | A5 | E5 | F1 | 71 | D8 | 31 | 15 |
| | 3 | 04 | C7 | 23 | C3 | 18 | 96 | 05 | 9A | 07 | 12 | 80 | E2 | EB | 27 | B2 | 75 |
| | 4 | 09 | 83 | 2C | 1A | 1B | 6E | 5A | A0 | 52 | 3B | D6 | B3 | 29 | E3 | 2F | 84 |
| | 5 | 53 | D1 | 00 | ED | 20 | FC | B1 | 5B | 6A | CB | BE | 39 | 4A | 4C | 58 | CF |
| | 6 | D0 | EF | AA | FB | 43 | 4D | 33 | 85 | 45 | F9 | 02 | 7F | 50 | 3C | 9F | A8 |
| | 7 | 51 | A3 | 40 | 8F | 92 | 9D | 38 | F5 | BC | B6 | DA | 21 | 10 | FF | F3 | D2 |
| | 8 | CD | 0C | 13 | EC | 5F | 97 | 44 | 17 | C4 | A7 | 7E | 3D | 64 | 5D | 19 | 73 |
| | 9 | 60 | 81 | 4F | DC | 22 | 2A | 90 | 88 | 46 | EE | B8 | 14 | DE | 5E | 0B | DB |
| | A | E0 | 32 | 3A | 0A | 49 | 06 | 24 | 5C | C2 | D3 | AC | 62 | 91 | 95 | E4 | 79 |
| | B | E7 | C8 | 37 | 6D | 8D | D5 | 4E | A9 | 6C | 56 | F4 | EA | 65 | 7A | AE | 08 |
| | C | BA | 78 | 25 | 2E | 1C | A6 | B4 | C6 | E8 | DD | 74 | 1F | 4B | BD | 8B | 8A |
| | D | 70 | 3E | B5 | 66 | 48 | 03 | F6 | 0E | 61 | 35 | 57 | B9 | 86 | C1 | 1D | 9E |
| | E | E1 | F8 | 98 | 11 | 69 | D9 | 8E | 94 | 9B | 1E | 87 | E9 | CE | 55 | 28 | DF |
| | F | 8C | A1 | 89 | 0D | BF | E6 | 42 | 68 | 41 | 99 | 2D | 0F | B0 | 54 | BB | 16 |

# Stream Ciphers

- There are two types of stream ciphers: synchronous and self-synchronizing.

- In a *synchronous* stream cipher (sometimes called a binary additive stream cipher), the key is a stream of bits produced by some pseudo-random number generator. The production of this key stream is completely independent of the plain text. The key stream is XOR'd with the plain text to produce the cipher text. It is called a synchronous cipher because the sender and receiver need to be in synch. If digits are added or removed from the message during transmission (such as by an attacker), then that synchronization is lost and needs to be restored.

- *Self-synchronizing* ciphers use parts of the current cipher text to compute the rest of the key stream.

# RC4

- RC4 is a widely known and used stream cipher, perhaps the most widely known. The RC stands for Ron's Cipher or Rivest cipher, as it was designed by Ron Rivest—a name that is very familiar in cryptography. He is the R in RSA, which we will explore in Chapter 10.

- RC4 is widely used in many security situations, including WEP (Wired Equivalent Privacy) and TLS (Transport Layer Security). The algorithm was designed in 1987, and some experts have expressed concerns about its security. There has been speculation that it can be broken, and many people recommend no longer using it in TLS.

- However, it is the most widely known stream cipher and has a place in the history and study of stream ciphers, similar to that of DES in block ciphers.

# RC4

- It uses variable length key from 1 to 256 bytes, which constitutes a *state table* that is used for subsequent generation of pseudorandom bytes and then to generate a pseudorandom string of bits which is XORed with the plaintext to produce the ciphertext

# FISH

This algorithm was published in 1993 by the German engineering firm Siemens.

The FISH (Fibonacci Shrinking) cipher is a software-based stream cipher that uses a Lagged Fibonacci generator (LFG) along with a concept borrowed from the shrinking generator ciphers.

A Lagged Fibonacci generator is a pseudo-random number generator based on the Fibonacci sequence.

# PIKE

This algorithm was published in a paper by Ross Anderson as an improvement on FISH.

In that paper, Anderson showed that FISH was vulnerable to known plain-text attacks.

PIKE is both faster and stronger than FISH. The name PIKE is not an acronym, but rather a humorous play on the name FISH, a pike being a type of fish.

# RANDOM NUMBER GENERATORS

Random numbers are a key part of cryptography. When you generate a key for a symmetric algorithm such as AES, Blowfish, or GOST, you need that key to be very random. Random numbers are also required for initialization vectors used with a variety of algorithms. A true totally random number is not possible to generate from a computer algorithm. It is only possible to generate a truly random number using other means including hardware, but not by using a software algorithm.

What is normally used in cryptography are algorithms called pseudo random number generators. Pseudo-random number generators (PRNGs) are algorithms that can create long runs of numbers with good random properties but eventually the sequence repeats.

# RANDOM NUMBER GENERATORS

- There are three types of pseudo random number generators
- Table look-up generators. Literally a table of pre-computed pseudo random numbers is compiled, and numbers are extracted from it as needed.
- Hardware generators. Some hardware process, perhaps packet loss on the network card, or fluctuations from a chip, are used to produce pseudo random numbers.
- Algorithmic (software) generators. This is the type most commonly used in cryptography, and what we will focus our attention on in this chapters.

# RANDOM NUMBER GENERATORS

Any pseudo random number generator is going to generate a sequence of numbers. That sequence should have certain properties:

- **Uncorrelated Sequences –**This simply means that the sequences are not correlated. One cannot take a given stretch of numbers (say 16 bits) and use that to predict subsequent bits. There, quite literally, is no correlation between one section of output bits and another.
- **Long Period**–Ideally the series of digits (usually bits) should never have any repeating pattern. However, the reality is that there will eventually be some repetition. The distance (in digits or bits) between repetition's is the period of that sequence of numbers. The longer the period the better. Put another way: we accept that there will be repeated sequences, but those should be as far apart as possible.
- **Uniformity**– It is most often that pseudo random numbers are represented in binary format. There should be an equal number of 1's and 0's, though not distributed in any discernable pattern. The sequence of random numbers should be uniform, and unbiased. If you have significantly more (or significantly less) 1's than 0's then the output is biased.
- **Computational Indistinguishability** –Any subsection of numbers taken from the output of a given PRNG should not be distinguishable from any other subset of numbers in polynomial time by any efficient procedure. The two sequences are indistinguishable. That does not, however mean they are identical. It means there is no efficient way to determine specific differences.

The third category is the one most often used in cryptography. It does not produce a truly random number but rather a pseudo random number.

# RANDOM NUMBER GENERATORS

How do you know if a given pseudo random number generator is random enough? There are a variety of tests that can be applied to the output of any algorithm to determine the degree of randomness.

The 1-D TEST is a frequency test. It is a rather simple test, and essentially used a first pass. In other words, simply passing the 1-D test does not mean a given algorithm is suitable for cryptographic purposes, However, if a given PRNG fails the 1-D test, there is no need for further testing. Imagine a number line stretching from 0 to 1, with decimal points in between. Use the random number generator to plot random points on this line. First divide the line into a number of "bins". These can be of any size, in the graph below, there are four bins, each size .25

|---------|---------|---------|---------|

0 .25 .50 .75 1.0

Now as random numbers (between 0 and 1.0) are generated, count how many fit into each bin. Essentially if the bins fill evenly that is a good sign that you have random dispersal. If there is a significant preference for one bin over another, then the PRNG is not sufficiently random. That PRNG has a bias and further testing is not required to determine that it is not useful for cryptographic purposes.

# RANDOM NUMBER GENERATORS

A run is an uninterrupted sequence of identical bits.
The focus of this test is the total number of runs in the sequence.
A run of length k consists of exactly k identical bits and is bounded before and after with a bit of the opposite value.
The purpose of the runs test is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence.
Determines whether the oscillation between such zeros and ones is too fast or too slow.
fast oscillation occurs when there are a lot of changes, e.g., 010101010 oscillates with every bit.
(input) E = 11001001000011111101101010100010001000010110100011000
01000110100110001001100011001100010100010111000
(input) n = 100
(output) P-value = 0.500798
A p-value is a statistical test. A small p-value (typically $\leq 0.05$) indicates strong evidence against the null hypothesis, so you reject the null hypothesis. A large p-value ($> 0.05$) indicates weak evidence against the null hypothesis, so you fail to reject the null hypothesis. You can use CryptTool to apply the Runs test to a sequence of numbers

# RANDOM NUMBER GENERATORS

- The poker test for PRNG's is based on the frequency in which certain digits are repeated in a series of numbers. It is best understood by considering a trivial example, a 3-digit number.
- In a three-digit number, there are only three possibilities. The first possibility is that the individual digits can be all different. The second possibility is that all three are different, and the third is that there is one pair of digits with one digit that does not match the other two.

- The tests actually assumes sequences of five numbers, as there are five cards in a hand of poker. The actual five numbers are analyzed to determine if any given sequence appears more frequently than the other possible sequences.

# RANDOM NUMBER GENERATORS

This is a very old algorithm. It was first described by a Franciscan friar in the 13th century. It was then re-introduced by John Von Neumann. It is a rather simple method and easy to follow. The following is a step-by-step description of the Mid square method:

Start with an initial seed (for example a 4-digit integer).

Square the number.

Take the middle 4 digits.

This value becomes the new seed. Divide the number by 10,000. This becomes the random number. Go step 2. The following is a concrete example:

$x_0 = 1234$

$x_1$: $1234^2 = 01\underline{5227}56 \rightarrow x_1 = 5227$, $R_1 = 0.5227$

$x_2$: $5227^2 = 27\underline{3215}29 \rightarrow x_2 = 3215$, $R_2 = 0.3215$

$x_3$: $3215^2 = 10\underline{3362}25 \rightarrow x_3 = 3362$, $R_3 = 0.3362$

The process is repeated indefinitely, generating a new random number each time. The middle four bits of each output is the seed for the next iteration of the algorithm.

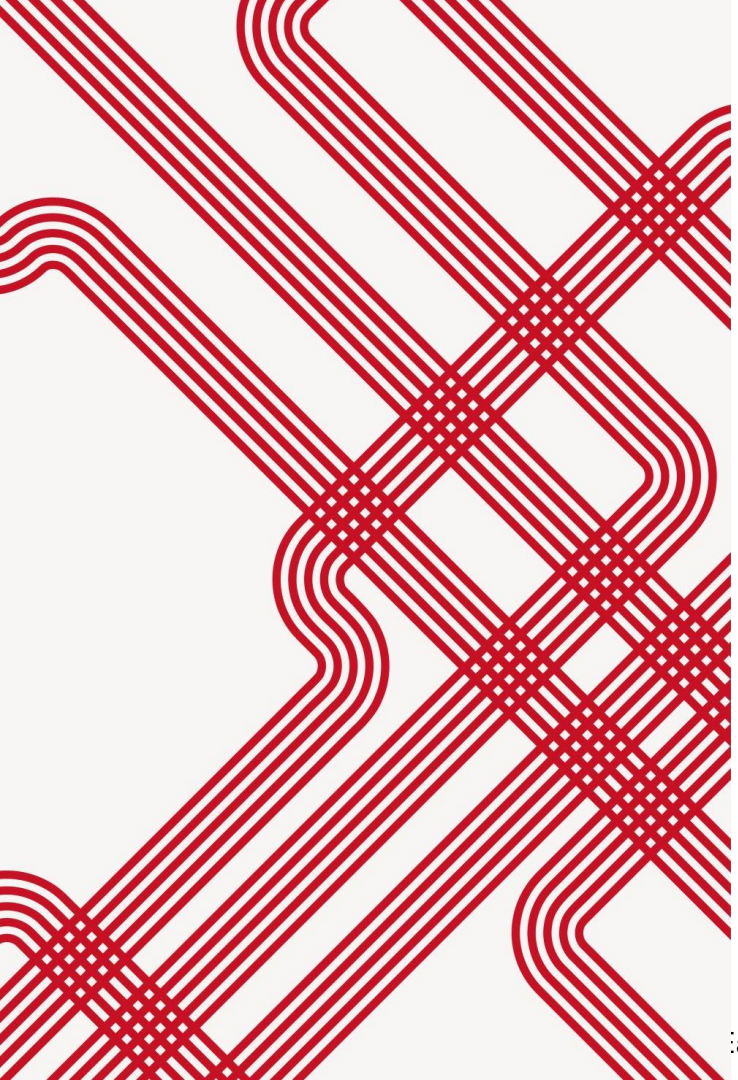
# Three Properties of a Hash

- The algorithm is one way; it is not reversible.
- Variable-length inputs produce fixed-length outputs.
- Few or no collisions should occur.

# Message Integrity

- One common use of hashing algorithms is to ensure integrity of messages. Messages can be altered in transit, either intentionally or accidentally. Hashing algorithms can be used to detect that such an alteration has occurred.

- Consider the example of an e-mail message. If you put the body of the message into a hashing algorithm, let's just say SHA-1, the output is a 160-bit hash. That hash can be appended at the end of the message.



# Password Storage

Cryptographic hashes also provide a level of security against insider threats. It is common to store passwords in a cryptographic hash. When the user logs into the system, whatever password she types is hashed and then compared to the hash in the database. If it matches exactly, the user is logged into the system.

Given that the database stores a hash of the password only, and hashes are not reversible, even a network or database administrator cannot retrieve the password from the database. If someone attempted to type in the hash as a password, the system will hash whatever input is input into the password field, thus yielding a hash different from the one stored in the database.

Storing passwords as a hash is widely used and strongly recommended.

# Merkle-Damgård Function

A Merkle-Damgård function (also called a Merkle-Damgård construction) is a method for building hash functions.

Merkle-Damgård functions form the basis for MD5, SHA1, SHA2, and other hashing algorithms.

This function was first described in Ralph Merkle's doctoral dissertation in 1979.

# Checksum

One of the simplest checksum algorithms is the longitudinal parity check.

It breaks data into segments (called words) with a fixed number of bits. Then the algorithm computes the XOR of all of these words, with the final result being a single word or checksum.

Here's an example. Assume this text:

Euler was a genius

Convert that to binary:

**01000101 01110101 01101100 01100101 01110010 00100000**
**01110111 01100001 01110011 00100000 01100001 00100000**
**01100111 01100101 01101110 01101001 01110101 01110011**

# Checksum *(continued)*

The segments can be any size, but let's assume a 2-byte (16-bit word). This text is now divided into nine words (shown here separated with brackets).

**[01000101 01110101] [01101100 01100101] [01110010 00100000] [01110111 01100001] [01110011 00100000] [01100001 00100000] [01100111 01100101] [01101110 01101001] [01110101 01110011]**

The next step is to XOR the first word with the second:

**01000101 01110101 XOR 01101100 01100101 = 00101001 00010000**

Then that result is XOR'd with the next word:

**00101001 00010000 XOR 00100000 01110111 = 00001001 01100111**

This process is continued with the result of the previous XOR then XOR'd with the next word, until a result is achieved. That result is called the longitudinal parity check.

This type of checksum (as well as others) works well for error detection.

# MD5

This 128-bit hash is specified by RFC 1321.

Designed by Ron Rivest in 1991 to replace and improve on the MD4 hash function, MD5 produces a 128-bit hash, or digest.

As early as 1996, a flaw was found in MD5, and by 2004 it was shown that MD5 was not collision-resistant.

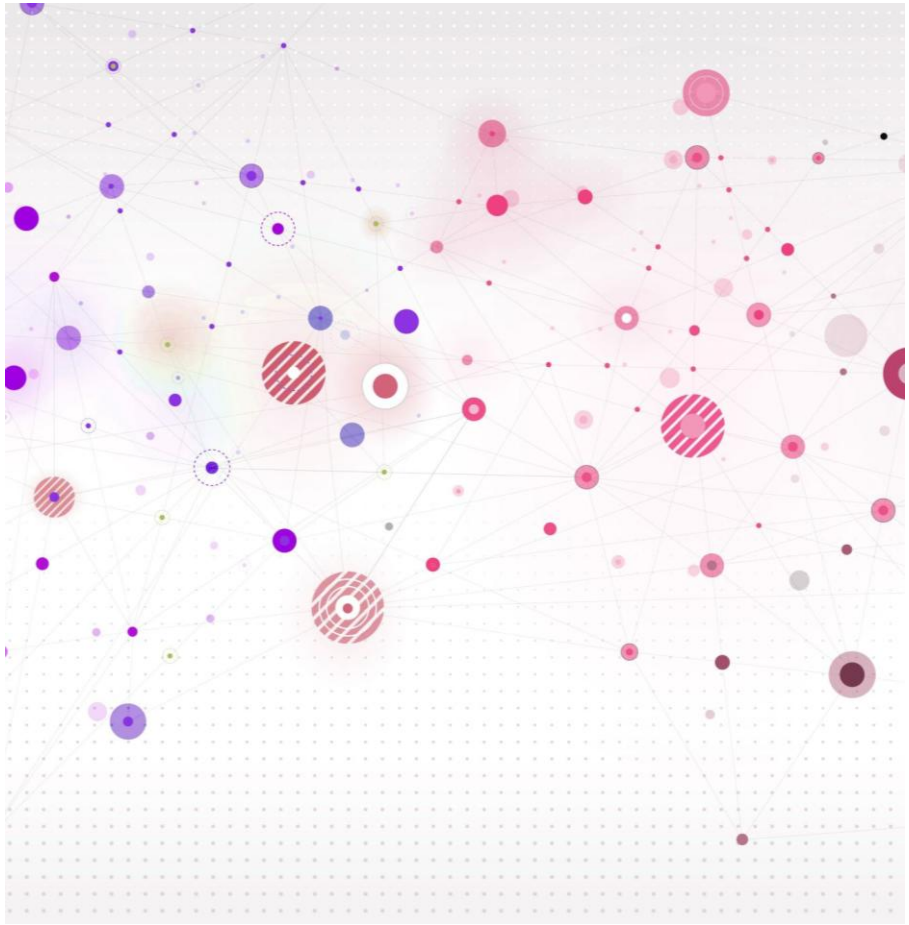
# SHA

The Secure Hash Algorithm is perhaps the most widely used hash algorithm today. There are several versions of SHA, all of which are considered secure and collision-free.

**SHA-1:** This 160-bit hash function resembles the earlier MD5 algorithm. It was designed by the National Security Agency (NSA) to be part of the Digital Signature Algorithm.

**SHA-2:** This is actually two similar hash functions, with different block sizes, known as SHA-256 and SHA-512. They differ in the word size; SHA-256 uses 32-byte (256 bits) words, and SHA-512 uses 64-byte (512 bits) words. There are also truncated versions of each standardized, known as SHA-224 and SHA-384. These were also designed by the NSA.

**SHA-3:** This is the latest version of SHA. It was adopted in October 2012.



# RIPEMD

- RACE Integrity Primitives Evaluation Message Digest is a 160-bit hash algorithm developed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel.

- There exist 128-, 256-, and 320-bit versions of this algorithm, called RIPEMD-128, RIPEMD-256, and RIPEMD-320, respectively.

- These all replace the original RIPEMD, which was found to have collision issues. The larger bit sizes make this far more secure that MD5 or RIPEMD.

# Tiger

Tiger is designed using the Merkle-Damgård construction (sometimes called the Merkle-Damgård).

Tiger produces a 192-bit digest.

This cryptographic hash was invented by Ross Anderson and Eli Biham and published in 1995.

# GOST

This hash algorithm was initially defined in the Russian national standard GOST R 34.11-94 Information Technology – Cryptographic Information Security – Hash Function. It is based on the GOST block cipher.

This hash algorithm produces a fixed-length output of 256 bits. The input message is divided into chunks of 256-bit blocks.

If a block is less than 256 bits, the message is padded by appending as many 0's to it as are required to bring the length of the message up to 256 bits. The remaining bits are filled up with a 256-bit integer arithmetic sum of all previously hashed blocks, and then a 256-bit integer representing the length of the original message, in bits, is produced.

# Message Authentication Code (MAC)

- A MAC function is used to add a secret key to protect integrity
- Two types of MACs
  - Hash Message Authentication Code (HMAC)
  - Cipher Block Chaining Message Authentication Code(CBC-MAC)

# MAC

A Message Authentication Code (MAC), particularly the Hashing Message Authentication Code (HMAC), is away to detect intentional alterations in a message. A MAC is also often called a keyed cryptographic hash function. That name should tell you how this works.

Assume you are using MD5 to verify message integrity. To detect an intercepting party intentionally altering a message, both the sender and recipient must previously exchange a key of the appropriate size (in this case, 128 bits). The sender will hash the message and then XOR that hash with this key. The recipient will hash the received message and XOR that computed hash with the key. Then the two hashes are exchanged. Should an intercepting party simply recompute the hash, he or she will not have the key to XOR it with (and may not even be aware that it should be XOR'd) and thus the hash the interceptor creates won't match the hash the recipient computes, and the interference will be detected.

# MAC/HMAC

So, imagine you are using a SHA 1 hash (160 bits)

Your message is :  this is my message

The SHA 1 Hash of that is: 589205d6d61d7c2f0c8d2f4b29d6db9aca3a91d7

But rather than simply send that as a hash, you first exclusively or it with some random number you have decided upon:

Your SHA 1 value  589205d6d61d7c2f0c8d2f4b29d6db9aca3a91d7

The random number a8a0c642a3fb7e14e50c9a22706d3094db2fed53

The output f032c39475e6023be981b56959bbeb0e11157c84

# MAC

- Or you want to use MAC with CBC. Your message is:  this is my message
- You put that through any symmetric cipher using CBC, but only use the last block. You send
- 69c4e0d86a7b0430d8cdb78070b4c55a

- Anyone without the key cannot create the same MAC and won't be able to perform a man in the middle attack.

- The output f032c39475e6023be981b56959bbeb0e11157c84

# Rainbow Table

In 1980 Martin Hellman described a cryptanalytic technique which reduces the time of cryptanalysis by using precalculated data stored in memory. This technique was improved by Rivest before 1982. Basically, these types of password crackers are working with pre-calculated hashes of all passwords available within a certain character space, be that a-z or a-zA-z or a-zA-Z0-9 etc, This is called a Rainbow table.  If you search a Rainbow table for a given hash, whatever plaintext you find, must be the text that was input into the hashing algorithm to produce that specific hash.

# Rainbow Table

Clearly such a Rainbow table would get very large very fast. Assume that the passwords must be limited to keyboard characters. That leaves 52 letters (26 upper case and 26 lower case), 10 digits, and roughly 10 symbols, or about 72 characters. As you can imagine even a 6-character password has a very large number of possible combinations. This means there is a limit to how large a Rainbow table can be, and this is why longer passwords are more secure that shorter passwords.

# VERY Simple Illustration of Rainbow Tables

| Password | MD5 Hash (in Hex) | Password | MD5 Hash (in Hex) |
|---|---|---|---|
| aaaa | 74b87337454200d4d33f80c4663dc5e5 | aaaaa | 594f803b380a41396ed63dca39503542 |
| aaab | 4c189b020ceb022e0ecc42482802e2b8 | aaabb | 120858a7016efcfab66967b834e9153c |
| aaac | 3963a2ba65ac8eb1c6e2140460031925 | aaacc | ee43671d755ac457cfe6e32d1894788e |
| aaa1 | 39dc4f1ee693e5adabddd872247e451f | aaa1a | 5bbac29650eb36b4de16885c190a9fa3 |
| aaa2 | 0ad346c93c16e85e2cb117ff1fcfada3 | aaa2a | 597f0ce6d11567cc691b3f5df35594cb |
| aaa4 | ee93fca7c150d9c548aff721c87d0986 | aaa4a | 4305dc076b3ba2bf8d55524cddf5a72d |

# Hash - Salt

Random bits added to further secure encryption or hashing. Most often encountered with hashing, to prevent Rainbow Table attacks.

Essentially the salt is intermixed with the message that is to be hashed.  Consider this example. You have a password that is

pass001

in binary that is

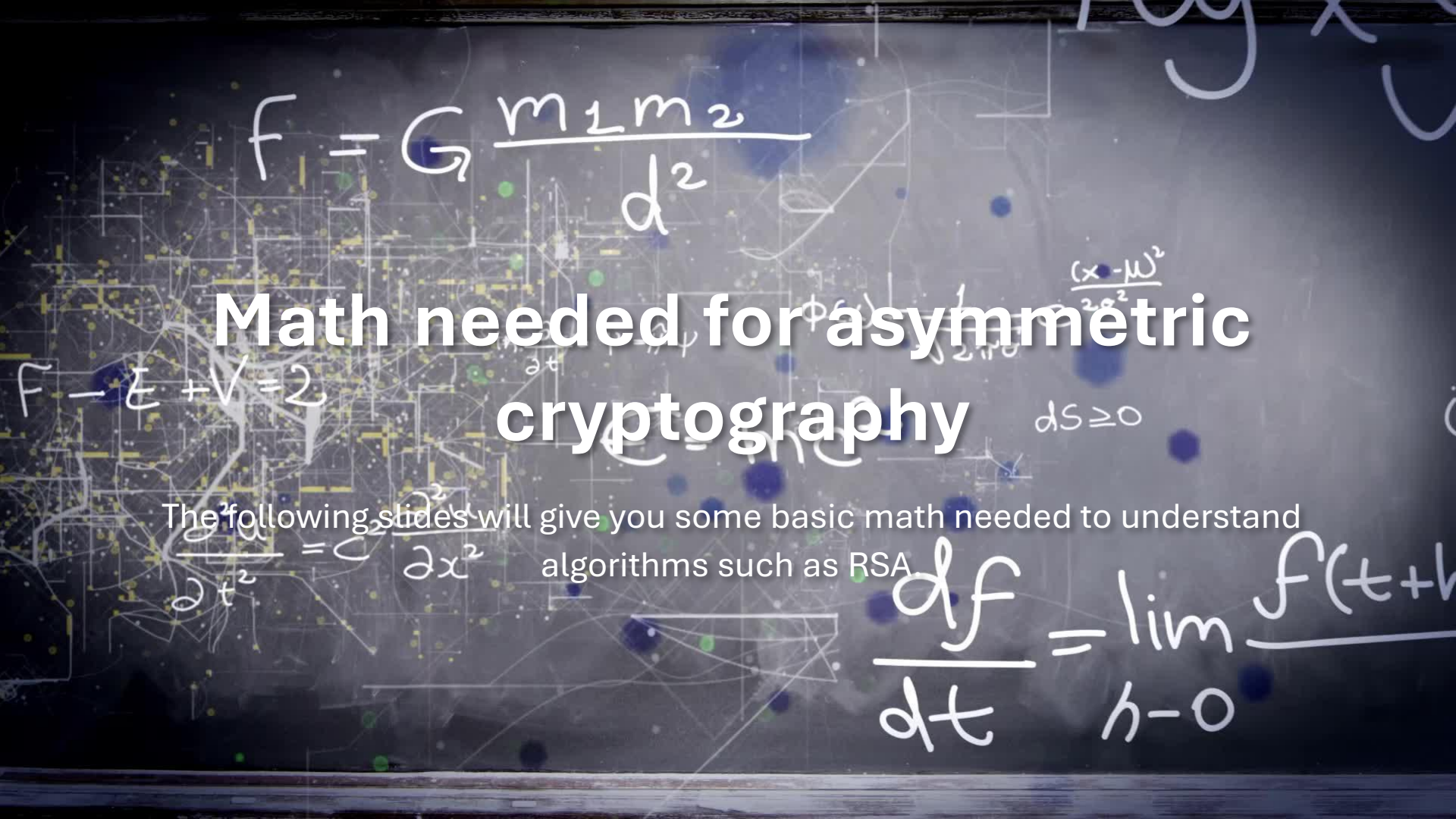
01110000 01100001 01110011 01110011 00110000 00110000 00110001

A salt algorithm would insert bits periodically, lets assume for this example, we just duplicate the first 8 bits every fourth byte:

01110000 01100001 01110011 01110011 01110000  00110000 00110000 00110001 10111001 01110000

If you convert that to text, you will get

passp001p



# Math needed for asymmetric cryptography

The following slides will give you some basic math needed to understand algorithms such as RSA

# Relatively Prime

- Two numbers x and y are relatively prime (co-prime) if they have no common divisors other than 1. So, for example, the numbers 8 and 15 are co-prime. The factors of 8 are 1, 2, and 4. The factors of 15 are 1, 3, and 5. Since these two numbers (8 and 15) have no common factors other than 1, they are relatively prime.

- The term for the integers that are smaller than *n* and have no common factors with *n* (other than 1) is *totative*. For example, 8 is a totative of 15.

- Another term for Euler's totient is the Euler phi function. $\varphi(n)$

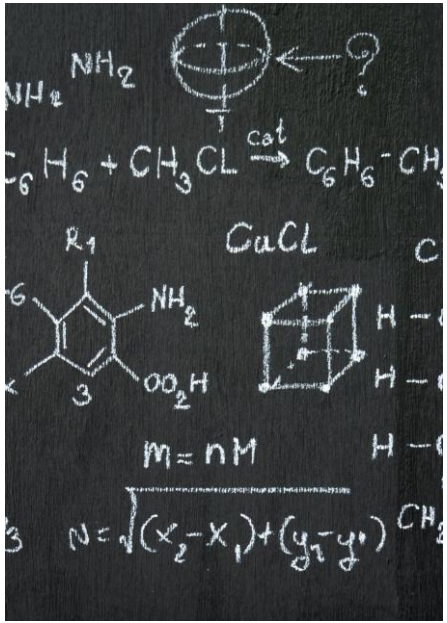
# Relatively Prime

- What if a given integer, *n*, is a prime number? How many totatives will it have (that is, what is the totient of *n*)?

- Let's take a prime number as an example, 7. Given that it is prime, we know that none of the numbers smaller than 7 have any common factors with it. So 2, 3, 4, 5, and 6 are all totatives of 7. And since 1 is a special case, it is also a totative of 7, so we find there are six numbers that are totatives of 7.

- It turns out that for any *n* that is prime, the Euler's totient of *n* is n – 1. So if *n* = 13, then the totient of *n* is 12.

# Modulus




- Simply divide **A** by **N** and return the remainder
- So **5** mod **2** ≡ **1**
- So **12** mod **5** ≡ **2**
- Sometimes symbolized as **%** as in **5 % 2** ≡ 1

# Modulus

- One way to think about modulus arithmetic is to imagine doing any integer math you might normally do but bound your answers by some number. A classic example is the clock. It has numbers 1 through 12, so any arithmetic operation you do has to have an answer that is 12 or less. If it is currently 4 o'clock and I ask you to meet me in 10 hours, simple math would say I am asking you to meet me at 14 o'clock. But that is not possible, because our clock is bounded by the number 12. The answer is simple: take your answer and use the mod operator with a modulus of 12 and look at the remainder:

- $14 \bmod 12 \equiv 2$

- So, I am asking you to meet me at 2 o'clock (whether that is a.m. or p.m. depends on the original 4 o'clock, but that is irrelevant to understanding the concepts of modular arithmetic). This is an example of how you use modular arithmetic every day.

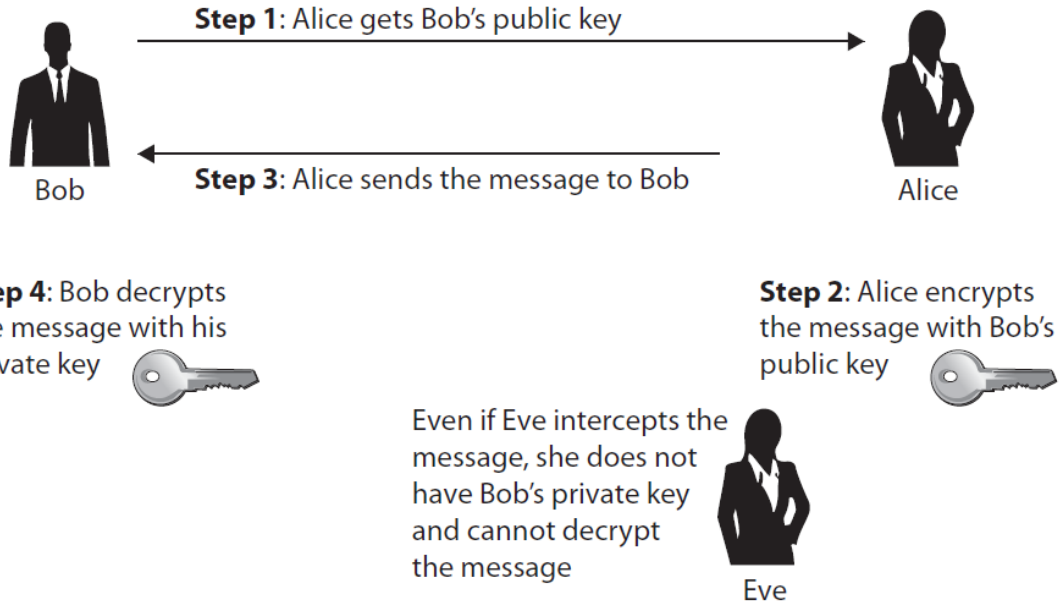
# Congruence

Congruence in modulus operations is a very important topic, and you will see it applied frequently in modern cryptographic algorithms.

Two numbers, a and b, are said to be “congruent modulo n” if

$$(a \bmod n) = (b \bmod n) \longrightarrow a \equiv b \pmod{n}$$

# How Public/Private Key Encryption Works



# RSA

RSA may be the most widely used asymmetric algorithm.

This public key method was publicly described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT. The letters RSA are the initials of their surnames.

# HOW DOES RSA WORK?

**Step 1**

Generate two large random primes, p and q, of approximately equal size such that their product n = pq is of the required bit length (such as 2048 bits, 4096 bits, and so on):

Let $n = pq$

Let $m = (p - 1)(q - 1)$

**Step 2**

Choose a small number e, co-prime to m. (Note: Two numbers are co-prime if they have no common factors.)

Find d, such that $de \bmod m \equiv 1$

**Step 3**

Publish e and n as the public key.

**Step 4**

Keep d as the secret key.

# RSA *(continued)*

Encrypt:

- $C = M^e \bmod n$

Put another way: Computes the cipher text $c = m^e \bmod n$.

Decrypt:

- $P = C^d \bmod n$

Put another way: Uses his private key (d,n) to compute $m = c^d \bmod n$.

# RSA *(continued)*

Normally RSA would be done with very large integers. To make the math easy to follow, this example uses small integers. (This example is from Wikipedia.)

1. Choose two distinct prime numbers, such as $p = 61$ and $q = 53$.
2. Compute $n = pq$, giving $n = 61 \times 53 = 3233$.
3. Compute the totient of the product as $\varphi(n) = (p - 1)(q - 1)$ giving $\varphi(3233) = (61 - 1) \times (53 - 1) = 3120$.
4. Choose any number $1 < e < 3120$ that is co-prime to 3120. Choosing a prime number for $e$ leaves us only to check that $e$ is not a divisor of 3120. Let $e = 17$.
5. Compute $d$, the modular multiplicative inverse of $e$, yielding $d = 2753$.

The public key is ($n = 3233$, $e = 17$). For a padded plain text message $m$, the encryption function is $m^{17} \pmod{3233}$. The private key is ($n = 3233$, $d = 2753$). For an encrypted cipher text $c$, the decryption function is $c^{2753} \pmod{3233}$.

# Another RSA Example

1. Select primes: $p = 17$ and $q = 11$.
2. Compute $n = pq = 17 \times 11 = 187$.
3. Compute $\phi(n) = (p - 1)(q - 1) = 16 \times 10 = 160$.
4. Select e: gcd(e,160) = 1; choose $e = 7$.
5. Determine d: $de = 1 \bmod 160$ and $d < 160$.
   Value is d = 23 since $23 \times 7 = 161 = 10 \times 160 + 1$.
6. Publish public key KU = {7,187}
7. Keep secret private key KR = {23,187}

# RSA Continued

Now use the number 3 as the plain text. Remember e =7, d=23, and n =187

- Ciphertext= $\text{Plaintext}^{e}$ mod n  or
- Ciphertext = $3^{7}$ mod 187
- Ciphertext =2187 mod 187
- Ciphertext = 130

Decrypt

- Plaintext = $\text{Ciphertext}^{d}$ mod n
- Plaintext = $130^{23}$ mod 187
- Plaintext = 4.175390541341311636704579 7e+48 mod 187
- Plaintext = 3


# RSA why the public key won't decrypt

Remember from the last slide the ciphertext is 130. Remember e =7, d=23, and n =187. What if you tried the public key (e,n) again, instead of the private key (d,n)?

- Decrypt
  - Plaintext = Ciphertext $^{e}$ mod n
  - Plaintext = $130^{7}$ mod 187
  - Plaintext = 627485170000000 mod 187
  - Plaintext = 37

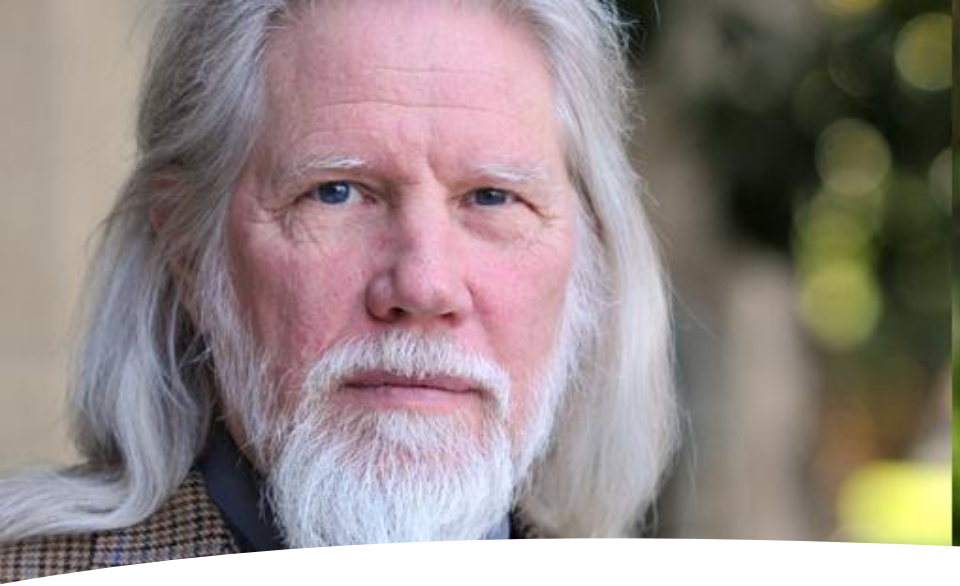

# RSA Continued

- Now take 3 as the plain text. Remember e =7, d=23, and n =187. But what if even one of these numbers is wrong? What if we did not pick an e that is coprime to m (160)? So, let's pick e = 6 which is not coprime to 160
  - Ciphertext= Plaintext$^{e}$ mod n or
  - Ciphertext = $3^{6}$ mod 187
  - Ciphertext =729 mod 187
  - Ciphertext = 168
- Decrypt
  - Plaintext = Ciphertext $^{d}$ mod n
  - Plaintext = $168^{23}$ mod 187
  - Plaintext = 1.52094489562674867628545902396666e+51 mod 187
  - Plaintext = 93

  - See it does not work! Only by using numbers that have the coprime relationships can this work

# RSA Continued

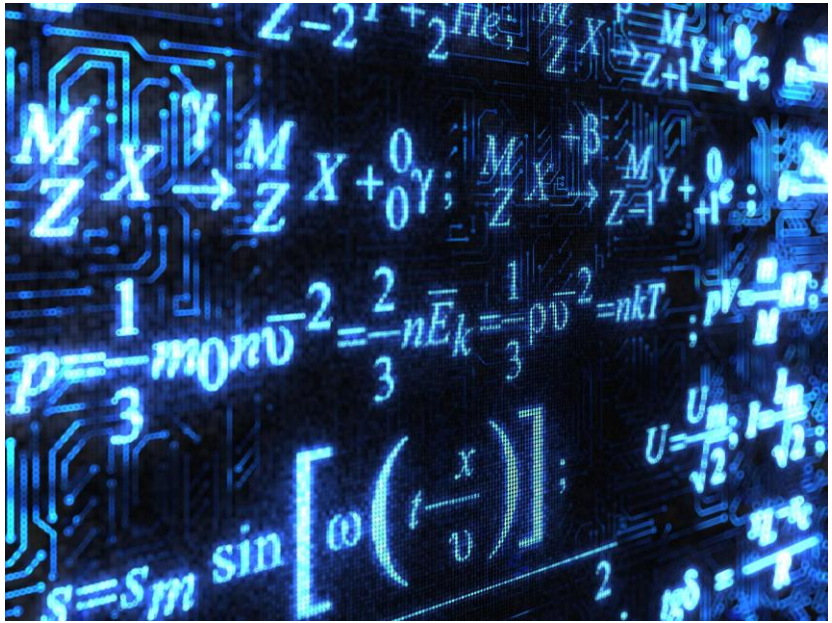
Also note RSA cannot encrypt any plaintext larger than the modulus. If you do, then the public key will decrypt as well as encrypt!



# Diffie-Hellman

Diffie-Hellman is a cryptographic protocol that allows two parties that have no prior knowledge of each other to establish a shared secret key jointly over an insecure communication channel.

# Diffie-Hellman *(continued)*



The system has two parameters, called *p* and *g*. Parameter *p* is a prime number, and parameter *g* (usually called a *generator*) is an integer less than *p*, with the following property: for every number *n* between 1 and *p* – 1 inclusive, there is a power *k* of *g* such that $n = g^k \bmod p$.

1. Alice generates a random private value *a* and Bob generates a random private value *b*. Both *a* and *b* are drawn from the set of integers.

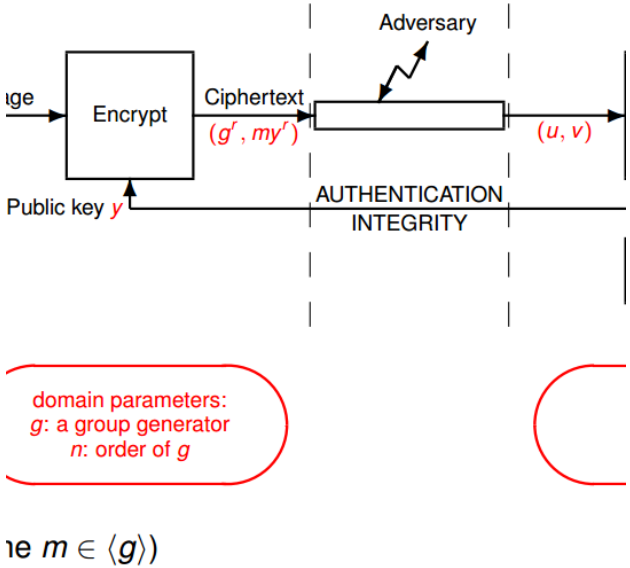
2. They derive their public values using parameters *p* and *g* and their private values. Alice's public value is $g^a \bmod p$ and Bob's public value is $g^b \bmod p$.

3. They exchange their public values.

4. Alice computes $g^{ab} = (g^b)^a \bmod p$, and Bob computes $g^{ba} = (g^a)^b \bmod p$.

5. Since $g^{ab} = g^{ba} = k$, Alice and Bob now have a shared secret key *k*.

# ElGamal




First described by Taher Elgamal in 1984, ElGamal is based on the Diffie-Hellman key exchange algorithm. It is used in some versions of PGP.

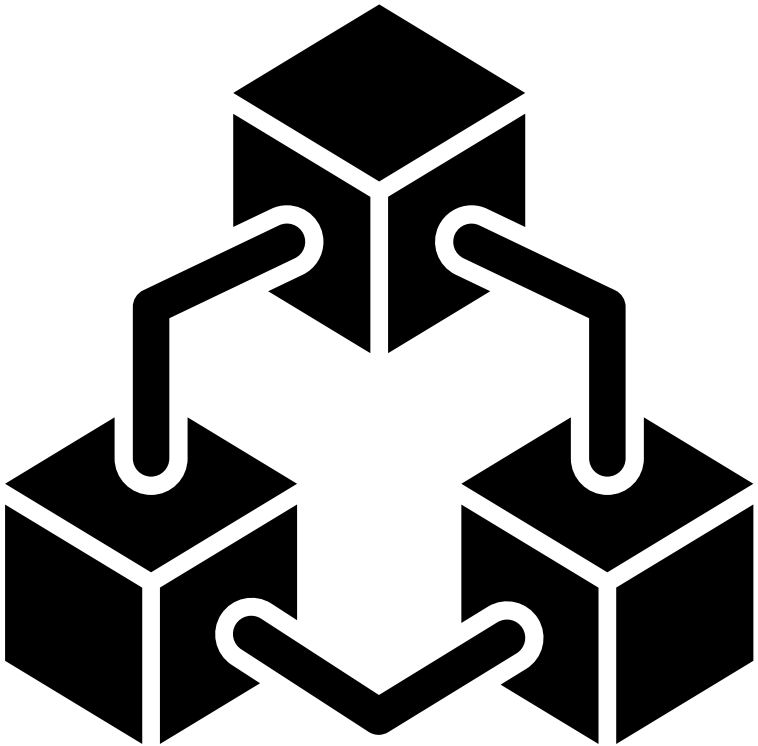
The ElGamal algorithm has three components: the key generator, the encryption algorithm, and the decryption algorithm.

# MQV

Like ElGamal, MQV (Menezes–Qu–Vanstone) is a protocol for key agreement that is based on Diffie–Hellman.

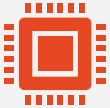
It was first proposed by Menezes, Qu, and Vanstone in 1995 and then modified in 1998.

MQV is incorporated in the public-key standard IEEE P1363. HQMV is an improved version.



# Elliptic Curves

Elliptic curves have been studied, apart from cryptographic applications, for well over a century.



As with other asymmetric algorithms, the mathematics have been a part of number theory and algebra, long before being applied to cryptography.

# Elliptic Curves in Cryptography

The use of elliptic curves for cryptographic purposes was first described in 1985 by Victor Miller (IBM) and Neil Koblitz (University of Washington).

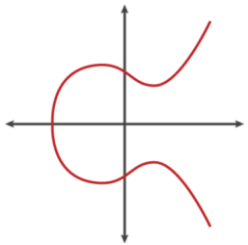
The security of elliptic curve cryptography (ECC) is based on the fact that finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point is so difficult that it is impractical.

# ECC Basics

An elliptic curve is the set of points that satisfy a specific mathematical equation. The equation for an elliptic curve looks something like this:

$$y^2 = x^3 + Ax + B$$

And graphed like this:



Another way to describe an elliptic curve is this: it is the set of points that satisfy an equation that has two variables in the second degree and one variable in the third degree. The first thing you should notice from the graph is the horizontal symmetry. Any point on the curve can be reflected about the x-axis without changing the shape of the curve.

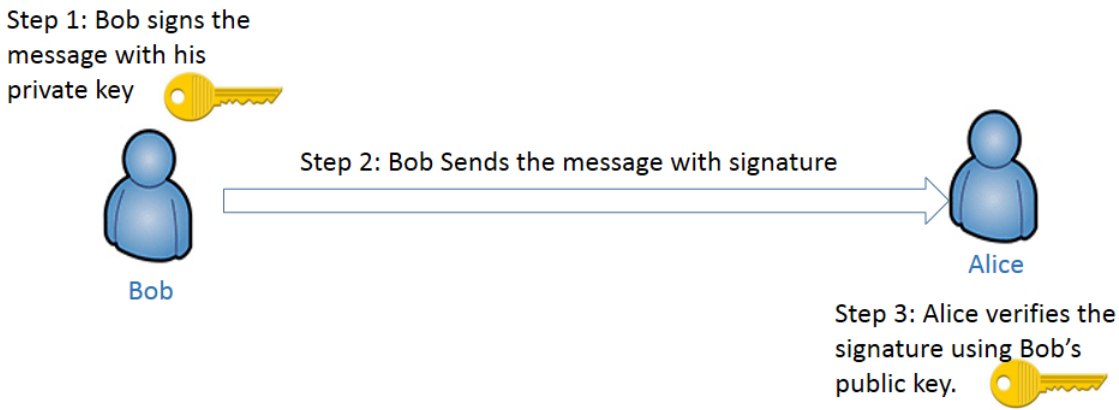
# Digital Signatures/Certificates

Digital Signature is usually the encryption of a message or message digest with the sender's private key. To verify the digital signature, the recipient uses the sender's public key. Good digital signature scheme provides:

- authentication
- integrity
- non-repudiation

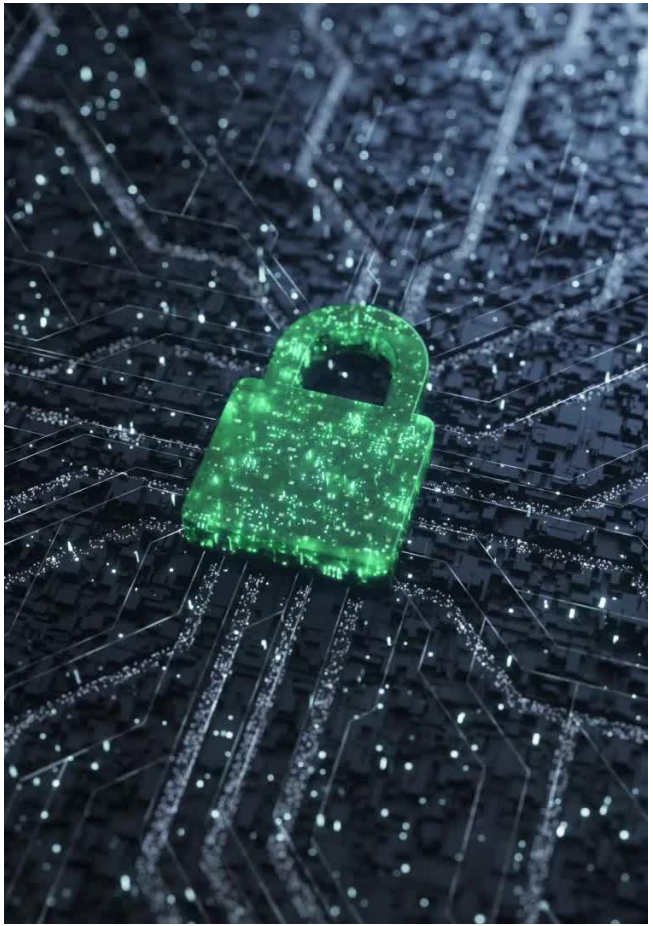
RSA algorithm can be used to produce and verify digital signatures; another public-key signature algorithm is DSA.

# Digital Signature Basics



# Good cryptography vs Bad cryptography

It is important to realize that this particular area of the computer industry is replete with frauds and charlatans. One need only scan a search engine for *encryption* to find a plethora of advertisements for the latest and greatest “unbreakable” encryption. If you are not knowledgeable about encryption, how do you separate legitimate encryption methods from frauds?



# Good cryptography vs Bad cryptography

There are many fraudulent cryptographic claims out there. You do not have to be a cryptography expert to be able to avoid many of those fraudulent claims. Here are some warning signs:

- **Unbreakable:** Anyone with experience in cryptography knows that there is no such thing as an unbreakable code. There are codes that have not yet been broken. There are codes that are very hard to break. But when someone claims that a method is "completely unbreakable," you should be suspicious.
- **Certified:** Guess what? There is no recognized certification process for encryption methods. Therefore, any "certification" a company claims is totally worthless.
- **Inexperienced people:** A company is marketing a new encryption method. What experience do the people working with it have? Does the cryptographer have a background in math, encryption, or algorithms? If not, has he submitted his method to experts in peer-reviewed journals? Or, is he at least willing to disclose how his method works so that it can be fairly judged? Recall that PGP's inventor had decades of software engineering and encryption experience.

# Good cryptography vs Bad cryptography

Auguste Kerckhoffs first articulated what has come to be called Kerckhoffs's principle in the 1800s. He stated that the security of a cipher depends only on the secrecy of the key, not on the secrecy of the algorithm. Claude Shannon rephrased, this stating that, "One ought to design systems under the assumption that the enemy will ultimately gain full familiarity with them." This idea, referred to as Shannon's maxim, states essentially the same idea as Kerckhoffs's principle.

# Top 10 Developer Crypto Mistakes

- Hard-coded keys
- Improperly choosing an IV
- ECB mode of operation
- Wrong use or misuse of a cryptographic primitive for password storage
- MD5 just won't die. And SHA1 needs to go too!
- Passwords are not cryptographic keys
- Assuming encryption provides message integrity
- Asymmetric key sizes too small
- Insecure randomness
- "Crypto soup" "use this term to mean a developer mixing a bunch of cryptographic primitives together without a clear goal" use this term to mean a developer mixing a bunch of cryptographic primitives together without a clear goal

https://littlemaninmyhead.wordpress.com/2017/04/22/top-10-developer-crypto-mistakes/

# Top 10 Amateur Crypto Mistakes

- Improper or inadequate seeding of the random number generator
- Using the MD5 hashing algorithm
- Storing SHA1(password) in the database (without salt)
- Using DES
- Using ECB mode
- Using the same keystream to encrypt two different documents
- Not verifying the hostname of the SSL certificate
- Not using certificate pinning
- Not testing your SSL configuration

https://webencrypt.org/10cryptographymistakes/

# Quantum Computing – NIST

Post quantum cryptography standards working group.
NIST has initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms. Full details can be found in the Post-Quantum Cryptography Standardization page.
The submission deadline of November 30, 2017, has passed. Please see the Round 1 Submissions for the listing of complete and proper submissions.

- In recent years, there has been a substantial amount of research on quantum computers – machines that exploit quantum mechanical phenomena to solve mathematical problems that are difficult or intractable for conventional computers. If large-scale quantum computers are ever built, they will be able to break many of the public-key cryptosystems currently in use. This would seriously compromise the confidentiality and integrity of digital communications on the Internet and elsewhere. The goal of *post-quantum cryptography* (also called quantum-resistant cryptography) is to develop cryptographic systems that are secure against both quantum and classical computers and can interoperate with existing communications protocols and networks.
- The question of when a large-scale quantum computer will be built is a complicated one. While in the past it was less clear that large quantum computers are a physical possibility, many scientists now believe it to be merely a significant engineering challenge. Some engineers even predict that within the next twenty or so years sufficiently large quantum computers will be built to break essentially all public key schemes currently in use. Historically, it has taken almost two decades to deploy our modern public key cryptography infrastructure. Therefore, regardless of whether we can estimate the exact time of the arrival of the quantum computing era, we must begin now to prepare our information security systems to be able to resist quantum computing.

# Quantum Computing – NIST Round 3

Asymmetric Cryptography

- Classic McEliece
- CRYSTALS-KYBER
- NTRU
- SABER

Digital Signature

- CRYSTALS-DILITHIUM
- FALCON
- Rainbow

Alternate Public Key

- BIKE
- FrodoKEM
- HQC
- NTRU Prime
- SIKE

Alternate Digital Signature

- GeMSS
- Picnic
- SPHINCS+

# NIST Cryptography

| |
|---|
| NIST has selected algorithms to be standardized |
| CRYSTALS-KYBER for Public-key Encryption and Key-establishment Algorithms |
| CRYSTALS-DILITHIUM for Digital Signatures |
| FALCON for Digital Signatures |
| SPHINCS+ for Digital Signatures |
| For more on CRYSTALS-KYBER |
| https://pq-crystals.org/kyber/ |

# NIST Cryptography

- **FIPS 203 (ML-KEM or CRYSTALS-Kyber):** This algorithm is designed for key establishment, ensuring that sensitive information can be securely exchanged, even in the presence of quantum-capable adversaries. It stands out for its efficiency in encryption and decryption, making it suitable for a wide range of applications, from secure communications to cloud storage. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf
- **FIPS 204 (ML-DSA or CRYSTALS-Dilithium):** Targeting digital signatures, ML-DSA provides a robust mechanism for verifying identities and ensuring the integrity of messages and documents. Its balance of speed and security makes it a strong candidate for use in software updates, code signing, and any scenario where the authenticity of information is critical. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf
- **FIPS 205 (SLH-DSA or SPHINCS+):** Also focused on digital signatures, SLH-DSA offers an alternative that emphasizes resilience against attacks, including those leveraging quantum computing. While it is slightly less efficient than ML-DSA, its stateless nature provides an additional layer of security, particularly for applications requiring long-term integrity. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf