Terminology and Principles:

* Cryptography: Science of making code
* Cryptanalysis: science of breaking codes
* Cryptology = Cryptography + Cryptanalysis
* Plaintext: Regular Text
* Ciphertext: Output of cryptosystem
* Kerckhoffs’ Principle: Cryptosystem known to attacker but key isn’t
* Cipher/Cryptosystem: Something used to encrypt plaintext into ciphertext
* Key: A parameter for a cryptosystem
* Symmetric systems: Systems in which Encrypt Key = Decrypt Key
* Confusion: relationship between plain and cypher text must be involved (i.e. each cyphertext bit must depend on many parts of key)
* Diffusion: Plaintext statistics should be spread through ciphertext (For same key, changing one bit of plaintext should change ½ of cyphertexts’ bits on average
* Asymmetric systems: Uses a key pair: One key for encryption, another for decryption
* Stream Cipher: Short key is stretched into long keystream and the used like one-time pad.
* Block Cipher: Each key determines different codebook for mapping blocks of plaintext into blocks of cipher text
* Salt: Random bits used in hashing
* SSO (Single sign on): Allows users to connect to different services provided they own account of one service
* Password Manager: Programs which manage many passwords via master password
* Cookie: Piece of data which maintains state across sessions when accessing associated website
* Asymmetric password-authenticated key exchanges (asymmetric PAKEs): cryptographic protocol that allows two parties to establish a shared cryptographic key securely, even if only one party knows the password

Notations & Formulae:

* P = plaintext block
* C = Ciphertext block
* K = Key
* E = Encrypt
* D = Decrypt
* Encrypt: C=E(P,K)
* Decrypt: P=D(C,K)
* P=D(E(P, K),K) & C=E(D(C,K),K)
* F (. , . ): Round Function
* ⊕: XOR

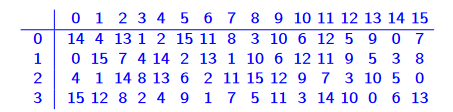
Cyphers:

* Caesar’s Cypher: Shift plain text by n positions (0 to 25). This is the key
  + Cypher = (plain text + shift) mod 26 and plain text = (cypher – n) mod 26
  + Brute search/Exhaustive Key analysis: try all possible 26 possible keys
  + Special case of simple substitution
* Simple substitution: Key can be any permutation of alphabet (i.e. not necessarily in alphabetical order)
  + 26! possible keys
  + Exploit statistics of letters (think of how e is the most common letter in words)
  + Monoalphabetic: One permutation of alphabet
* Vigenère Cipher: Polyalphabetic substitution
  + Circular shift by the characters in key word.
  + e.g. CAT indicates a shift on the first character by 2, the 2nd character by 0 and the 3rd by 19
  + 26t keys where t is keyword length
* Double Transposition: Encryption on permutations of plaintext
  + Arrange into a m x n matrix, permute rows then permute columns then concatenate rows
  + Key is matrix dimension plus column and row permutations
* Vernam Cypher/one-time pad cipher:
  + Encrypt: Cyphertext= plaintext XOR key
  + Decrypt: plaintext = Cyphertext XOR key
  + Letters in text are converted into binary (based on position in word)
  + Key is randomly generated, same size as message, known only to sender and receiver and used only once
* A5/1:
  + 3 shift registers
    - X : 19 bits (X0🡪x18)
    - Y: 19 bits (Y0🡪Y21)
    - Z: 19 bits (Z0🡪Z22)
  + Key is a 64 bit string, used to initialize 3 registers
  + Bits used like 1-time pad
  + Each iteration:
    - Take majority of x8, y10 and z10 and depending on which is majority perform following:
      * If x8:
        + t= x13 XOR x16 XOR X17 XOR X18   
           xi= xi -1 for i 18🡪1, x0=t
      * If y10:
        + t= Y20 XOR Y21

yi= yi -1 for i 21🡪1, y0=t

* + - * If z10:
        + t= Z7 XOR Z20 XOR Z21 XOR22   
           zi= zi -1 for i 22🡪1, z0=t
    - Keystream bit is X18 XOR Y21 XOR Z22 regardless of step or not
* RC4:
  + Self-modifying lookup table
  + Contains permutation of values 0🡪255
  + Key is used to initialise permutation in table
  + Each step:
    - Swaps elements I lookup table so new permutation is chosen
    - Selects keystream byte from table
  + Initialization:
    - S[i] contains permutation of 0🡪255
    - Key[i] contains N bytes of key for i🡪N-1 of byte length N which ranges from 1🡪256
    - Pseudocode:
    - For i in range (0,256):   
       s[i]=i   
       K[i]=key[i% N] this is mod n  
      J=0  
      For i in range (0,256):   
       J= (j+ S[i] +k[i]) % 256  
       Swap s[i] and s[j]  
      i=j=0
  + Keystream generation:
    - In each stop, two elements are swapped, and keystream byte is selected:
      * Swap elements
        + i = (i + 1) %256  
          j = (j + S[i]) %256  
          Swap s[i] and s[j]
      * Select keystream byte from table
        + Byte= S[ (S[i]+s[j]) % 256 ]
* Feistel Cipher:
  + Plaintext/Cyphertext split into left and right halves:
  + Encrypt: For each round 1🡪n:  
    Li = Ri-1   
    Ri = Li-1 XOR F( Ri-1 ,Ki)  
    Ciphertext is Ln,Rn
  + Decrypt: For each round 1🡪n:  
    Ri-1 = Li   
    Li-1 = Ri XOR F(Ri-1 ,Ki) = Ri XOR F(Li ,Ki)  
    Plaintext is L0,R0
* Data Encryption Standard (DES)
  + Feistel cipher with a 64-bit block length and a 56-bit key length, 16 rounds and 48 round subkeysA diagram of a flowchart

    Description automatically generated
  + For expand
    - Permute 32-bit input by repeating 16 random bits
  + For S-box
    - 8 Total S-Boxes
      * S boxes are non-linear substitution boxes, they map 6 bits to 4 bits. Have a set standard
      * Takes the first and last bit from input to create a new binary and does the same for the remaining bits, giving us a new 4-bit output depending on combination
      * Essentially a lookup table
      * S-Box 1 with the columns and rows indicating the decimal of the bits (row being bits 1 and 6 and columns are the remaining)



* + P- Box
    - Each bit corresponds to an index
    - They permute 32 bits giving the following bit-index order (left to right):
      * + 15 6 19 20 28 11 27 16 0 14 22 25 4 17 30 9  
          1 7 23 13 31 26 2 8 18 12 29 5 21 10 3 24
  + Key Schedule/Key generation
    - 56-bit DES drives the key schedule which helps us produce the 48 bit skeys needed in the 16 rounds
    - To generate a subkey, we split the key in two 28 half key bits (LK and RK) after permutating them
      * For rounds 1,2,9,16 we shift LK and RK by 1 bit while for the other rounds, its 2 bits
      * Each bit corresponds to an index (0🡪52), as we shift the binary, its corresponding index moves with it
      * Subkey is produced from LK and RK after removing specific bits (compression permutation) reducing the 56 bit to 48 bits
    - Initial bit permutation after first round applied to plain text and we inverse this to get cipher text
    - RK and LK are swapped after final round
    - S-BOX dependant level of security (Provide non-linearity)
    - Only way to attack is with exhaustive key search
    - As a 56-bit DES Key is small, we use 3DES or Triple DES which has a 112 bit key in which we use two different keys to encrypt/decrypt
      * C = E(D(E(P, K1), K2), K1) & P = D(E(D(C, K1), K2), K1)
      * We don’t do E(E(P,K1),K2) = C because a meet-in the middle attack is possible
        + Attacker knows a pair (P,C) and then computes a table of E(P,K1) for every possible key K1 and does the same for K2 and D(C,K2). If a match between the two tables is found, they will get the keys (only for one match though)
        + Only needs 257 tries as opposed to 2112
* AES (Advances Encryption Standard)
  + Replacement for DES
  + Iterated block cipher and not a Feistel Cipher
  + Permutation-Substitution Cipher
  + 128 bits expressed as 4x4 matrix of bytes
  + Key length of either 128,192 or 256 bits
  + 10-14 rounds which is a concat of 4 function belonging to 3 layers
    - ByteSub (non-linear layer)
      * Similar to S-Box; it’s a non-linear composition of 2 math operations or a sub operation
    - ShiftRow (linear mixing layer)
      * Cyclic shift of rows
      * For row 1, we shift the 1st element one, for row 2 we do it twice, and row 3 thrice.
    - MixColumn (linear mixing layer): Implemented like a lookup table
      * Matrix multiplication in which same matrix used for all columns
    - AddRoundKey (key addition layer)

Attacks:

* Related Key attack: An attack that exploits relationships between cryptographic keys to deduce information about the cipher. Occurs when keys aren’t completely independent. To prevent this, first 256 bytes of keystream is discarded
* Meet-in the middle attack : If an attacker knows a pair (P,C) they can then computes a table of E(P,K1) for every possible key K1 and does the same for K2 and D(C,K2). If a match between the two tables is found, they will get the keys. Only needs 257 tries as opposed to 2112
* Cryptosystem is secure if best attack is to try all possible keys. If shortcut is known, not secure
  + - Efficient in hardware, slower in software
    - Used in resource constrained devices
    - Rare due to block ciphers
* Block Cipher: Each key determines different codebook for mapping blocks of plaintext into blocks of cipher text
  + Plain and cyphertext consist of fixed size blocks, where latter is usually obtained by iterating a round function
    - Input to function is a round subkey and output of previous round
    - Key schedule: Production of round keys from overall key
  + Idea is to implement an avalanche effect
  + Feistel Cipher: general class of block ciphers
    - Plaintext/Cyphertext split into left and right halves
      * Encrypt:
        + For each round 1🡪n:

Li = Ri-1

Ri = Li-1 XOR F( Ri-1 ,Ki)

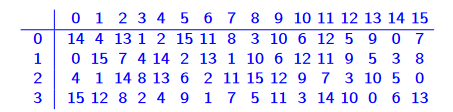
F is a round function, and Ki is subkey for current round

* + - * + Ciphertext is Ln,Rn
      * Decrypt
        + For each round 1🡪n:

Ri-1 = Li

Li-1 = Ri XOR F(Ri-1 ,Ki) = Ri XOR F(Li ,Ki)

* + - * + Plaintext is L0,R0
    - Only secure for certain round functions
  + Data Encryption Standard (DES)
    - It is a Feistel cipher with a 64-bit block length and a 56-bit key length, 16 rounds and 48 round subkeys
    - A diagram of a flowchart

      Description automatically generated
    - For expand
      * We take in 32 bits and we permute the input with expansion, giving us a 48 bit output in which 16 bits are repeated from the 32 bit input
    - For S-box
      * 8 Total
      * S boxes are non-linear substitution boxes
      * They map 6 bits to 4 bits
      * Takes the first and last bit from input to create a new binary and does the same for the remaining bits
      * Depending on the above combination, we get a new 4-bit output
      * Set in standard
      * Essentially a lookup table
      * Below is S-Box 1 with the columns and rows indicating the decimal of the bits (row being bits 1 and 6 and columns are the remaining)
      * 
    - P- Box
      * Each bit corresponds to an index
      * They permute 32 bits giving the following bit-index order (left to right):
        + 15 6 19 20 28 11 27 16 0 14 22 25 4 17 30 9  
          1 7 23 13 31 26 2 8 18 12 29 5 21 10 3 24
    - Key Schedule/Key generation
      * 56-bit DES drives the key schedule which helps us produce the 48 bit skeys needed in the 16 rounds
      * To generate a subkey, we split the key in two 28 half key bits (LK and RK) after permutating them
        + For rounds 1,2,9,16 we shift LK and RK by 1 bit while for the other rounds, its 2 bits
        + Each bit corresponds to an index (0🡪52), as we shift the binary, its corresponding index moves with it
      * Subkey is produced from LK and RK after removing specific bits (compression permutation) reducing the 56 bit to 48 bits
    - Initial bit permutation after first round applied to plain text and we inverse this to get cipher text
    - RK and LK are swapped after final round
    - S-BOX dependant level of security
    - Only way to attack is with exhaustive key search
    - As a 56-bit DES Key is small, we use 3DES or Triple DES which has a 112 bit key in which we use two different keys to encrypt/decrypt
      * C = E(D(E(P, K1), K2), K1) & P = D(E(D(C, K1), K2), K1)
      * We don’t encrypt, Encrypt for C because a meet-in the middle attack is possible
        + Attacker knows a pair (P,C) and then computes a table of E(P,K1) for every possible key K1 and does the same for K2 and D(C,K2). If a match between the two tables is found, they will get the keys (only for one match though)
        + Only needs 257 tries as opposed to 2112
  + AES (Advances Encryption Standard)
    - Replacement for DES
    - Iterated block cipher
    - Not a Feistel Cipher
    - Permutation-Substitution Cipher
    - Not many strong shortcut attacks known
    - 128 bits expressed as 4x4 matrix of bytes
    - Key length of either 128,192 or 256 bits
    - 10-14 rounds which is a concat of 4 function belonging to 3 layers
      * ByteSub (non-linear layer)
        + Similar to S-Box
        + it’s a non-linear composition of 2 math operations or a sub operation
      * ShiftRow (linear mixing layer)
        + Cyclic shift of rows
        + For row 1, we shift the 1st element one, for row 2 we do it twice, and row 3 thrice.
      * MixColumn (linear mixing layer)
        + Implemented like a lookup table
        + Matrix multiplication in which same matrix used for all columns
      * AddRoundKey (key addition layer)
        + XOR elements
    - To decrypt everything must be invertible
    - Subkeys used in reverse to decrypt
    - Decryption != Encryption
* To encrypt multiple plaintext blocks, we first turn the plain text into blocks of the size accepted by block cipher and use the same key for each block and pad any partial blocks
* PKCS#7 padding was when we set the value of the padded byte to the length of required padding. It is always applied
  + If plaintext length happens to be a multiple of 16, we add a full block of padding set to value 16
  + We remove padding after decryption
* Modes of operations used for Block Ciphers:
  + Electronic Codebook(ECB):
    - Encrypt each block independently
    - Seen like a single codebook cipher
    - Assume 64-bit blocks and 8-bit ASCII
    - Mallet can cut and paste cipher text in transits, can switch order
    - Integrity issues and if two plaintext blocks are similar, giving Mallet information to attack
    - If using an image, structure will leak in ciphertext due to repeated blocks
  + Cipher Block Chaining:
    - Blocks are chained together and as such, aren’t encrypted independently
    - Uses an initialisation vector (IV), a block-sized bit string shared that is shared between the two people, and the following equation to encrypt/decrypt
      * Ci = E (C i-1 XOR Pi, K) Pi = C i-1 XOR D( Ci , K), where C -1 is IV
    - IV should not be reused because if it is, the same first cyphertext will be produced if the first plaintext block is repeated
    - IV can be randomly chosen and sent as the first ciphertext block
    - Identical plaintext blocks have different cipher text blocks
    - If transmission errors occur, one cipher block error creates two block errors in decryption
      * i.e. If C1 happens to corrupted to but rest is correct then P 1 C0 XOR D( , P2 but P 3 C2 XOR D(
  + Counter (CTR)
    - Makes block cipher act like stream cipher
      * Used when random access to encrypt data is required like read or write
    - Encryption: Ci = Pi XOR E(IV+I,K) Decryption: Pi = Ci XOR E(IV+I,K)
* Preserving data integrity is critical and encryption alone doesn’t provide this, it only provides security
* Message Authentication Code (MAC) is sent with ciphertext for integrity
  + To produce this, we can use CBC-MAC which is done by computing the CBC encryption of the plaintext but save the final ciphertext block, the MAC
  + MAC is CN−1 = E(CN−2 ⊕ PN−1, K) = MA, we then send the P0 🡪 PN-1 and the Mac in which the receiver computes the MAC and checks if it’s the same as the same
  + Receiver and sender must share K and IV which is fixed
  + If error occurs due to man in the middle, the MAC would be different thus the receiver would know the message is compromised
  + Thus, we usually combine encryption and integrity with CBC as if authentication fails we can just discard the data
  + Encrypt and produce MAC with 2 different keys
  + Twice as much work
* In public key/asymmetric cryptography, key pairs are used
  + A uses B’s public key to encrypt and B uses their private key to decrypt message
  + Must be related via a function. i.e not independently chosen of each other
  + Two key pairs needed for duplex communication
  + Based on trapdoor one way function
    - One way = Easy to compute in one way but hard in the other
    - Trapdoor = function is easy to revert given one owns some information
  + Uses digital signatures
    - Since B’s public key is public, anyone can encrypt a message M to produce C, but only B can decrypt C to give M using their private key
    - A signature is produced by encrypting M to C with his private key
    - A signature is then verified by decrypting C using B’s public key and checking it matches M and if so only B could have sent it
    - Recall modulo operation:
      * a mod n = where is the floor (i.e. round down)
      * ( a + b) mod n = ( ( a mod n) + (b mod n) ) mod n
      * ( a x b) mod n = ( ( a mod n) x (b mod n) ) mod n
      * as mod n = (a \* …\* a) mod n
      * if a != 0 and there exists a-1 ∈ {1, . . . , n − 1} s.t a \* a-1 mod n =1 , then a1 is the inverse of a modulo n.
  + Knapsack problem/subset sum problem: Given a set of N positive integer weights (W0🡪Wn-1 ) and a target sum S, find coefficients ai ∈ {0, 1} such that S = a0 W0 + a1 W1 + … an-1 Wn-1
    - General Knapsack (GK) is hard to solve as it is NP (nondeterministic polynomial) complete
    - A super increasing knapsack (SIK) is easy to solve despite a large N
      * Each weight is greater than the sum of previous weights i.e.
      * Given S and starting with j = N – 1 and ending j < 0:
        + if Wj<=S:

then aj = 1 and S 🡨 S - Wj

* + - * + Else aj=0 and j 🡨 j-1
      * This is a greedy algorithm
    - Given the above 2, we can use them to create a public key cryptosystem by creating an SIK then converting it to a GK via parameters
      * Modulus n is any positive integer greater than sum of elements in SIK
      * Multiplier m an integer that is coprime with n (gcd is 1)
      * Example let the SIK be (2,3,4,30,57,120,251). We compute the GK by multiplying each of the elements in the SIK by m then using mod 491, we get the GK
      * The public key is not the GK and the modulus
      * The private key is the SIK and the SIK and the inverse of m mod n
        + To find the inverse of a \*b mod c is to find the value of b (range from 0 to c-1) in which a \* b mod c = 1
        + To encrypt an 8-bit plain text, multiply each bit by the corresponding GK weight
        + To decrypt we multiply the GK value by the modular inverse then mod it n (in this case 491)
      * Forward attack search: Encrypt all plaintext bits and see which yields the GK
      * Can be decrypted in polynomial time
  + RSA :
    - Using two large prime numbers p and q, we set a modulus N such that N = p \* q. We choose e such that e is co-prime to (p-1)(q-1). We then try to find such that e \* d = 1 mod (p-1)(q-1)
    - Public key is (N,e) and private key is d
    - Plaintext M is treated as an integer between 0 and N-1
      * We need to chop the plaintext into block
      * To encrypt it we do: Me mod N and to decrypt we do: Cd mod n
    - If Mallet can find N = pq, easily can find d as e\*d = 1 md (p-1)(q-1)
    - Cd = (Me)d = M mod N
    - Eulers Theorem: If gcd(x, n) = 1 then xΦ(n) = 1 (mod n), where Φ(n) is Euler’s totient function
      * As e \* d = 1 mod (p-1)(q-1) 🡪 ed 1 + k(p-1)(q-1) = kΦ(n) as we can use that fact (p-1)(q-1) = Φ(n)
      * Thus Cd = Med =M \* MΦ(n)\* k = M \* (MΦ(n)) k = M\*1k =M mod N
    - Repeated Squaring:
      * Take one bit a time (i.e. we try to fin x20 we take the exponent in base 2 which is 10100 we build the following (1, 10, 101, 1010, 10100) = (1, 2, 5, 10, 20). The second list is the list of indexes we calculate
      * From there we find the mod each x index where the index is the indexes we found above. We take them in terms of each other and since we know answers, we substitute them in . For example let x be 5 :
        + 5 mod 25 is 5
        + 52  mod 25 is 52 = 25
        + 55 mod 25 = 52 \* 5 mod 35 = 252 \*5 mod 25 = 10
        + 510 mod 25 = (55)2 = 102 mod 35 = 30
        + 520 mod 25 = (510)2 = 302 mod 35 = 25
    - To speed up RSA, we can use a small e value as we only need 2 seps for Me like 3 but e=3 leads to a cube root attack
    - Cube root attack: If message is < cube root of N, then M3 is also smaller than N 🡪 C = M3 thus M is the cube root of C
      * To prevent this, we pad the message by using a value like e = 216 +1. Its larger than 3 and it is still relatively small
  + Diffie Hellman Key exchange:
    - Key exchange alg
    - Based on discrete logarithm problem
      * Consider v= gk mod p
      * The problem consists in getting the exponent k given v, g and p
    - How it works:
      * Let p be prime and g (generator) be a number between 1 an p-1 s.t for any x, there is n such that x = gn mod p. g and p are both public
      * A selects a private value a s.t. a is between 1 and p -2 and B selects a private value b which is also between 1 and p – 2
      * A send ga to B and B sends gb to A
      * They both then compute gab mod p and is then used as a symmetric key
    - Mallet can pretend to be the sender and receiver in a Man in the Middle attack (MiM)
      * Can encrypt DH with symmetric or public key to prevent this or sign the DH values with a private key
  + It is confidential and makes use of authentication protocols
  + They also use digital signatures which provide integrity and non-repudiation
  + Notation:
    - {.} involves a public key
      * {M} Alice means encrypt M using Alices public
    - [.] involves a private key
      * [D] Bob means Decrypt with bobs Private
    - {[M] Alice}Alice= M = [{M}Alice]Alice
  + Order matters:
    - [M]Alice}Bob
      * Issue: after decrypting with his private key, Bob can forward  
        [M]Alice to a third party (Charlie) as follows:  
        {[M]Alice}Charlie which leads Charlie to think that he is the destination of Alice’s message, but the original destination was Bob
    - [{M}Bob]Alice
    - Issue: MiM (man in the middle). The MiM can be somebody that Bob trusts, such as Charlie, who obtains {M}Bob using Alice’s public key and then sends [{M}Bob]Charlie to Bob
  + Digital Certificate: Essentially acts like a way to verify someone is who they are using their public key
    - Signed by a CA (Certificate Authority) which is a trusted source
    - signature S = [M]CA which is then used to make A’s Certificate = (M,S)
    - To verify the signature, M = {S}CA
  + PKI (Public Key Infrastructure): Something needed to securely use Public key Cryptography
    - No General Standard
    - Compromised of:
      * Key Generation and management
      * CA
      * Certificate Revocation lists
    - Has 3 Trust Models:
      * Monopoly model:
        + Single trusted organisation is the CA for known universe
        + Who to trust?
      * Oligarchy Models:
        + Multiple trusted Cas
        + User decides who to trust
        + Chaining: CA can get certificate for its public key from another CA and do the same with its public key and again with another CA

Root certificate leading to same safety as self-signed certificates

* + - * Anarchy Model:
        + Everyone is a CA and users decide who to trust
        + More signatures a public key has, more trustable
* Symmetric keys tend to have better speed and no need for PKI (but we need to generate and distribute shared keys) whereas for public key they make use of signatures and there isn’t a need for a shared secret but you need to communicate public keys in a trusted way
* Hybrid cryptosystems use public key cryptography to establish a symmetric key and then symmetric key cryptography is used to encrypt data

A close up of a text

Description automatically generated

* If the message M is large, the signature [M]Alice is costly to compute, send and store
  + Might be divided into many blocks for encryption and so S will be same size in blocks
* Instead we can hash a message such that A signs h(M) s.t. h(M) is a smaller fingerprint of M
  + A sends M and S = [h(M)]Alice to B
  + B then computes the shash and verifies it is equal to {S}Alice
  + Efficient
* Collision: When Mallet finds that M’ != M s.t. h(M’)=h(M)
  + Mallet can replace (M,s) with (M’,S)
    - B wont detect the tampering as h(M’) = h(M) = {S}Alice and thinks M’ was signed by Alice
    - Hash is broken = Collision is found
* Features needed in a cryptographic hash function h(x):
  + Compression: the size of the output should be small
  + Efficiency: h(x) should be easy to compute for any x
  + One way: given a hash value v, hard to find x s.t. h(x) = v
  + Weak collision resistance, given x and h(x), unfeasible to find x!=y s.t. h(y) = h(x)
  + Strong collision resistance, unfeasible to find x and y with x!=y s.t. h(x) = h(y)
    - Just hard to find as hashes aren’t one to one
* In cryptographic hashes, hash vales are unrelated for different messages and are uniformly distributed
  + If h(x) is n bits long, p(h(x)) = 1/2n
* Birthday paradox and birthday problem:
  + Think of each person as a different message with birthday being result of a hash with each birthday is equally likely for each person
  + Given N people in the room, probability no one has a given birthday is (364/365)N and if we subtract 1 from that, that means at least one person has that given birthday . IN this case N=about 253 to yield a 50% chance someone has that as their birthday (Weak collision resistance analyses)
  + To see how many people need to be in a room for it to be likely that two share a birthday:
    - γ= 354/365 \* 363/365 \* … (365-N-1)/365 i.e. this is for N <= 365, if N>365 then γ = 0
    - γ = 1 – γ is the probability two people share the same and if we set rho to ½ for N we get about N is around 23
    - Strong collision resistance analyses
  + This leads to a paradox since the number of people is vastly different
  + Total number of comparisons with N people in a room around N2 / 2
  + If h(x) is n bits long, there are 2n possible hash values and according to the 3nd point, one can expect a 50% collision after 2n/2 hashes.
    - Compared to a secure n-bit symmetric cipher which require 2n-1 attempts to break with a probability of ½ , it appears weaker
* Cyclic Redundancy check (CRC): error detection code
  + A CRC is the remainder of a long division with a certain function of the message, could use this as a hash value
* Different hashes are:
  + MD5 (Message Digest 5)
    - 128 bit output
  + SHA (Secure Hashing Algorithm)
* Different hash functions take an arbitrary length input and produces a fixed length output . Always same sized output in spite of input size, a tiny change in input completely changes output.
* Hash functions is deterministic
* Hashing algorithms divide message into blocks where each block passes through some rounds, leading to an avalanche effects
* Merkle-Damgard Scheme:
  + Message is divided into b-bit blocks M1🡪Mt, may need padding
  + Then processed block by block, using intermediate n-bit changing variables hk for k = 1🡪t+1
  + Procedure:
    - h1 = IV (initial value)
    - hk+1 = g(hk , Mk) 1<=k<=t where g is a compression function
    - h(M) = ht+1
  + If g is collision resistant, so is the iterated hash
  + Compression functions used here can be built using block ciphers
  + Schemes for compression function are based on encryption function E( . , .)
    - Matyas-Meyer-Oseas: hk+1  = Mk ⊕ E( Mk , f(hk))
    - Davies-Meyer: hk+1 = hk ⊕ E( hk , f(hk))
    - Miyaguchi-Preneel: hk+1 = hk ⊕ E( Mk , f(hk))
    - F acts as a mapping to the key size if needed
* SHA-2:
  + Four algorithms, called SHD-d where d ∈ {224, 256, 384, 512} which produce d-bit hashes
  + NO efficient attacks found yet
* SHA-3
  + Shares structures and operation as predecessors
  + Mode for arbitrary long hashes called SHAKE
* HMAC:
  + Alternative to CBC-MAC
  + Symmetric key hash
  + Hash functions are public
  + We need a key for the same reason as CBC-MAC: if we just send M and M =h(M), mallet can replace M and create a new MAC
  + Before applying hash function, we can concact the key and message or the other way area
  + If we follow Merkle scheme, we compute in blocks
    - M=(M1,M2) then h(m) = g(g(IV,M1),M2) = g(h(M1),M2)
    - This is an issue because:
      * If HMAC is h(K,M). Mallet can produce M’ s.t. M’=(M1,M2,X) leading to h(K,M’) = g( h(K,M),X). This means that Mallet can get the HMAC of M’ with needing the key
      * If HMAC is h(M,K). Mallet can produce M’ s.t. h(M’) = h(M). If M is a multiple of the block size then h(M,K) = g(h(M),K) = g(h(M’),K) = h(M’K) leading to mallet finding a keyed collision as they found a collision without a key
  + To compute it securely:
    - Let B be message block length used in hash
    - ipad = byte 0x36 which is repeated B times
    - opad = byte0x5c repeated b times
    - HMAC(M,K) = h(K ⊕ opad, h(K ⊕ ipad, M))
* Hashes are used in:
  + Authentication
  + Message integrity
  + Fingerprint
  + Corruption detection
  + Signature efficiency
* Hashes in practice:
  + Suppose we have 3 bidders and they each want to bid a certain amount. They could hash the bids, submit them and then reveal them but are prone to a forward search attack
  + Subresource integrity
  + BitTorrent
* Visual Cryptography:
  + Secret Sharing
  + Imagine C wants to divide an image between A and B by giving them a part of the image, A and B must work together to reveal image by pooling resources
  + No one can learn anything from one share alone
  + Possible combinations:

A row of squares with black squares

Description automatically generated

* + - To recover OG pixel, we use a logical OR
  + No key
  + Brute force attacks are last resort but not possible with one time pad or only given one search(in the alters case, its because shares are chosen uniformly at random so possible images could be made by overlapping all possible shares which is alot)
* Random Numbers:
  + Used in key generation, nonce generation (Numbers only used once), Monte Carlo
  + Must be unpredictable because if pseudorandom, can be possible to determine a key via pooling other generated keys
  + Can generate pseudorandom numbers, with a seed, via:
    - Linear feedback shift registers and linear congruential generators where the seed is the initial value of register or modular recurrence
    - Keystreams where the seed is the key
    - Hash function where the seed is the input to hash function
  + The above 2 are more secure due to non-linearity but issue of a random seed is still an issue
  + Truly random sources:
    - Entropy sources
      * hardware devices which exploit randomness occurring in physical process and phenomena
      * Software methods like mouse movements, keyboard dynamics
      * Slow
    - True random number generators (TRNGs)
  + Raw randomness can be poor quality
  + Thus we usually use PRNGs because the Mixing and Entropy sources are slow, we use a single uniformly random value to seed a PRNG which then produces numbers at a high rate
* Digital Data hiding:
  + Adds unobtrusive info to data either via
    - Watermarking
      * Hidden info is linked to host
      * Can be imperceptible in which the mark is not evinced by looking at the file or perceptible
    - Steganography
      * Host carries hidden info which is undetectable
      * If perfect, Mallet will never know when A or B communicate
  + A and B Share a symmetric key which embeds and retrieves info
    - May need two keys for data hiding and cryptographic key
* Least Significant Bit(LSB) is an example of data hiding:
  + Imperceptible changes to human but computer can see
  + If image is a x b pixels, we can hide 3 a x b pixels
* Authentication: Are you say who you are and thus determine if access is allowed
* Based on something you know, have or are
  + Something you know is more popular than something you have or are as its more convenient and is free
* Recall Cryptographic keys have 2n keys where n is number of bits
  + If key is chosen at random, we must try about 2n-1keys
* If passwords are 8 characters long with 256 possible characters, we have 2568 passwords which is the same as 264
  + Since passwords aren’t randomly selected, an attacker has far less passwords to try when attacking
  + Good password: Difficult to guess but east to remember
* To attack a password, one can either target a particular account, any account on a system or any account on any system
  + Common path for mallet: Outsider🡪User🡪Admin
  + One single weak password is all that’s needed
* Depending on how long a system locks after 3 password attempts, issues occurs:
  + 5 seconds wont deter an automated attack
  + 5 minutes allows mallet to cycle over accounts and then moving to a new one after locked. Could also exploit temporary lock for a DoS (Denial of Service Attack)
  + If one waits until the system admin restores the service, it becomes very inconvenient
* If storing password (no encryption or hashing) in a file, if system is compromised, they’ll be exposed. If we encrypt the file, the decryption key must be accessible for verifying passwords which lead to the issue of Mallet can access the password file, access to key
  + Thus, store a hashed password rather than password
    - i.e y=h(password)
* Mallet can perform a forward search attack in that they can guess x and check y = h(x)
* Dictionary attack: Given a dictionary of common passwords, precompute a hashed dictionary. Compare hashed password to precomputed hashed dictionary if they get access to password
  + Can reuse, no need to re-hash
* Salts help make these more difficult
  + Compute hashed password s.t. y = (s, password)
  + Not Secret
* Password Cracking Maths
  + Assumptions:
    - Password is 8 characters with 128 choices per character then there are 1288 = 256 possible passwords
    - Password file has 210 hashed passwords
    - Mallet has a dictionary of 220 common passwords
    - P(Password in Dictionary) is ¼
    - Work = Number of hashes needed by Mallet. comparisons of hashes needs no work
  + If we want to attack one specific password :
    - no dictionary: we can only perform a brute force attack which means we must try 256 / 2 = 255 passwords. Hash must be computed for each one as well
    - Have a dictionary:
      * With Salt:
        + Average work done is ¼ \* 219 + ¾ \* (256) /2 which is around 254.6
        + This is because probability password is in dictionary is ¼ so we multiply it by possible number. We repeat the same with probability that isn’t in dictionary and the possible passwords not in dictionary.
      * Without Salt :
        + One time work of 220
        + So no work with trying dictionary only the ¾ \* (256) /2
        + Succeeds with ¼ probability with precomputed dictionary hashes with no work
  + Attack any 210 passwords:
    - Without Dict:
      * No salt:
        + Sequentially compute all possible password hashes and compare for Yiwhere i is on possible password
        + Avg work is 255/210  as each computed hash allows for 210 comparisons
      * Salt:
        + Avg work is 255 as each comparison needs a hash computation so Mallet can’t recycle computed hashes done earlier
    - With Dict:
      * Assume P(¾)1024= 0 is the probability that no password in dictionary, thus we can assume at least one password in in the dictionary
      * No salt:
        + Hash dictionary (220 ) and expect at least one match
      * If salt is used:
        + Let s0 🡪s210 -1  for each of the 210 password hashes in password file and Let d0 🡪d210 -1  be the dictionary words
        + Mallet computes h(s0,d0) and compares it to y0 then repeats the same with the other dictionary values.
        + If y0 is in the dictionary then its found after 219 hashes on average otherwise 220
        + If password isn’t recovered, repeats process with s1 and y1
        + For n-th attempt, if yk is in dict avg work is 219 with p(1/4) else it is n \* 220 + 219
        + Expected work is : ¼\*219 + ¼ \* ¾ \*( 220 + 219 ) + ¾2 \* ¼ \*(2\*220 + 219 ) + … + ¼ \* ¾n\* (n \*220 + 219) < 222
        + avg work = size of dictionary/Pr(password is in dictionary)
* Password threats:
  + Failure to change
  + Keystroke logging/spyware
  + Social engineering
  + Error longs which may contain passwords
* Two main SSO protocols:
  + SAML (Security Assertion Markup Language): Based on XML encodings
  + OIDC (Open ID Connect): An extension to OAuth2 authorisation protocol which is based on JSON encoding
* Can use cookie to authenticate user
* To avoid to share users to random string associations, webpage could store username in cookie but is easily modifiable🡪impersonation
  + Can be avoided if key is hashed thus using a hash along with username (HMAC CBC-MAC)
* Example of asymmetric PAKE is Secure Remote Protocol, integrated into TLS
  + Prone to Man in the Middle
* Oblivious PAKE (OPAQUE): Protocol based off an oblivious pseudorandom function (OPRF) in which one party does computation of a pseudorandom function (PRF) for another party
  + PRF is similar to MAC which uses a uniformly random output
  + How it works:
    - Alice blinds input with a random value and sends it to Bob
    - Bob then runs the PRF on the blinded input using his secret key
    - Alice unblinds the input to get output
  + Output only depends on input
  + Can use exponentiation modulo and a large prime p
    - Alice’s input is an element x ∈ {1, ... , p − 1}
    - They then generate a random binding factor r ∈ {1, . . . , p − 2}
    - Blinds input xb= xr (mod p) and is sent to Bob
    - Bob computes blinded output s.t. yb = (xb)k mod p (k is secret key)
    - Alice then unblinds the result: y = (yb)inverse r = ( (xb)k )inverse r  = ( xkr)inverse r = xk (mod p)
  + Can be used to register to a server
    - Alice generates key pair and sends public key to server who stores it
    - Then uses OPRF with her password to get a symmetric key which is used to encrypt their key pair, later sent to server
    - When logging in, they obtain keypair from server and then uses OPRF with password to get a symmetric key to decrypt key pair then signs a challenge