confidentiality: private or confidential info is not made available or disclosed to unauthorized individuals
- sometimes privacy also lumped under confidentiality

integrity: assures that data has not been altered by unauth. similar to non-repudiation
availability: systems work, service is not denied to auth.

Alice	Mallory	Eve	Bob
Sender	Malicious (can modify, sniff and drop)	Sniff only	Receiver

symmetric key encryption
properties
1. **correctness:** for any pt. x and key k, $D_k(E_k(x)) = x$
2. **security:** ciphertext should be indistinguishable from random stream
3. **probabilistic:** for same pt, could be diff. ct, decrypts to x

attack model
attacker goals

total break	partial break	distinguishability
attacker wants key	not interested in key, interested in CT./info on CT.	most modest goal, with probability > 50% attacker can correctly distinguish CTs of a given PT from another

indistinguishability: semantically secure
distinguishability ← partial break ← total break (*worst*)
always design system to prevent attacker from achieving the "weakest" goal

attacker's capability

CT-only	attkr analyse ciphertext itself, brute force - time inconclusive, weakest attkr capability
known-PT	attkr has a collection of PT and corresponding CT, may capture (PT,CT) pairs - certain PT patterns will appear - may find key depending on how PT is transformed
chosen-PT	attkr choose arbitrary PTs to be encrypted and obtains corresponding CTs (via encryption oracle)
chosen-CT	attkr chooses CT and decryption oracle outputs PT (attkr has access to Dec. Oracle) - worst!

Kerckhoffs' Principle: a system should be secure even if everything about the system, except the *secret key*, is public knowledge

modern ciphers
key space: set of all possible keys
key space size: size of key space
key size: number of bits required to represent a key

substitution cipher
- a substitution table, S, represents a 1-1 onto function from each letter in PT to CT
- in this case, the *secret key* is S
- key size: $\kappa!$, where κ represents all symbols used in the key

permutation cipher
- first groups PT into blocks of t characters, then applies a secret permutation to each block by shuffling the chars
- *secret key*: permutation which is a 1-1 onto function, t could also be kept secret (e.g. $t = 5, p = (1, 5, 2, 4, 3)$)
- key size: $n!$, where n is number of bits being permuted

attacks on substitution and permutation ciphers
1. brute force: exhaustively search the keys
- to be secure, brute force must be computationally infeasible
2. frequency analysis - mapping frequent characters in English to CT

- substitution cipher is not secure under CT only attack, when PT are English sentences

- known plaintext attack on permutation cipher causes the key to be easily found

XOR operations
commutative: $A \oplus B = B \oplus A$
associative: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
identity element: $A \oplus 0 = A$
self-inverse: $A \oplus A = 0$

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

one time pad (OTP)
pure substitution cipher, key cannot be re-used and key should be as long as the PT

Plaintext	0	0	1	0	1	1	0
	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Key	1	1	0	0	1	1	1
Ciphertext	1	1	1	0	0	0	1

$\forall x, k: (x \oplus k) \oplus k = x \oplus (k \oplus k) = x (x: PT)$

secure as leaks no info of PT but only if:
- OTP should consist of truly random characters (not pseudorandom!!!)
- OTP must have same length as PT
- only two copies of OTP should exist
- OTP should be used only once
- both copies of the OTP are destroyed immediately after use

however, OTP is not practical due to long keys

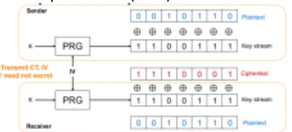
stream cipher
use a pseudo-random number stream for keystream
use short keys to generate long p-rand keystream
- encrypts PT 1 byte at a time

keystream reuse attack
- if two messages are identical and if the same keystream is used to encrypt the message, the corresponding CT is identical

stream cipher with same keystream
consider same key used to encrypt two diff. PTs
$U = X \oplus K; V = Y \oplus K$
- attkr eavesdropped and got U, V
attkr computes: $U \oplus V = (X \oplus K) \oplus (Y \oplus K)$
$= X \oplus Y \rightarrow$ this reveals partial info about the PT

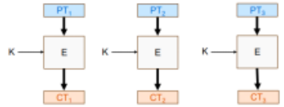
stream cipher with IV
initialization vector is random value used to generate keystream
- ensures that even if same key is used for multiple encryptions, generated keystream is different if unique IV is used

stream cipher with IV (cont.)



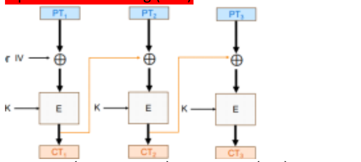
modern ciphers - DES and AES
Data Encryption Standard (DES)
- symmetric key block cipher
block size: 64 bits, key size: 56 bits
Advanced Encryption Standard (AES)
block size: 128 bits, key size: 128/192/256 bits
block cipher
block cipher has a fixed size input/output

Electronic Code Book (ECB)
divides PT into blocks and applies block cipher to each block, all with the same key



same PT always corresponds to same CT

Cipher Block Chaining (CBC)



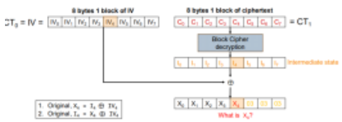
$CT_j = E_k(PT_j \oplus CT_{j-1})$ $PT_j = E_k^{-1}(CT_j) \oplus CT_{j-1}$
+: semantic security -: cannot be parallelized

Counter (CTR)
similar to stream cipher, uses IV as X_1, subsequently, $X_j = X_1 + j - 1$
+: semantic security, can be parallelized

meet in the middle attack on DES
DES is not secure, try to improve using multiple encrypt
consider $[m, c = DES_{k2}(DES_{k1}(m))]$, for known PT m
- attacker has sniffed $[m, c = DES_{k2}(DES_{k1}(m))]$
- wants to find k_1 and k_2
- attacker can encrypt using k_1 and decrypt using k_2 and find the corresponding keys that product c'/m'
- for k-bit keys, crypto operations reduced to 2^{k+1}
- remedy: 3-DES with 2 keys
$E_{k1}\left(E_{k2}(E_{k1}(x))\right)$ or $E_{k1}\left(D_{k2}(E_{k1}(x))\right)$

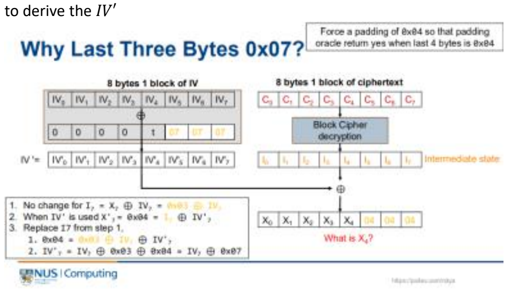
padding oracle attack (on AES CBC)
padding format
- block size of AES is 128 bits (16 bytes)
- length of PT is 25 bytes, so there are 7 remaining bytes that need to be padded with some values
- we must encode the number of padded bits else receiver will not know the length of the PT
- padded bytes' value is number of added bytes

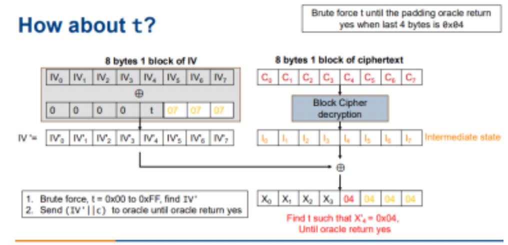
attacker has access to padding oracle

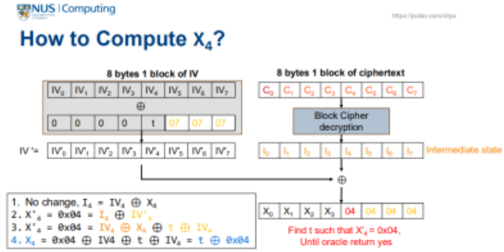


padding oracle attack (cont.)
attacker uses padding oracle by submitting modified CTs and seeing the oracle's response

how to tell the values of the padded IV bytes?
- since we are using CBC, we need the previous CT block to derive the IV'







algo needs to know initial padding length
padding oracle is a weaker form of decryption oracle
- prevention: deny access to the oracle, change the padding standard to mitigate attack
- note: CTR mode also vulnerable to padding oracle attack

PKC and data integrity
public key cryptography
k_e: public key, k_d: private key
given k_e and CT but not k_d, difficult to determine the PT
- must be difficult to get k_d from k_e
- encryption oracle is always accessible, so anyone can encrypt → chosen plaintext attack must be considered

symmetric key setting
individual secure channel for each pair (sender + recvr)
- 1 key for each pair of entities
- total keys: $\frac{n(n-1)}{2}$

public key setting
secure broadcast channel, each entity publishes k_e and keeps it's k_d secret
- total public keys: n, total private keys: n
- fewer keys, entities don't need to know each other before broadcasting keys

RSA (Rivest, Shamir, Adleman)
based on the mathematical properties of large primes and modular arithmetic
- computationally difficult to factor the product of two large primes

phase 1: key generation
1. randomly choose 2 large primes p, q and compute $n = pq, \phi(n) = (p-1)(q-1)$, where $\phi(n)$ is Euler's totient function
2. randomly choose an encryption exponent e s.t. gcd $\left(e, (p-1)(q-1)\right) = 1$
3. find decryption exponent d (multiplicative inverse of e) s.t. $d * e * mod\ \phi(n) = 1$
4. publish $[n, e]$ as public key, keep d as private key

phase 2: encryption/decryption
1. encryption
- given m and $[n, e]$ CT = $m^e mod\ n$
2. decryption
- given c and $[n, d]$ PT = $c^d\ mod\ n$
- difficult to derive k_d from k_e unless you can factor n to find p & q

correctness: for any +ve $m < n$, any pair of k_e, k_d
- $D(E(m)) = m$
- $(m^e)^d\ mod\ n = m$

security of RSA
- factoring large n back to p & q is computationally infeasible

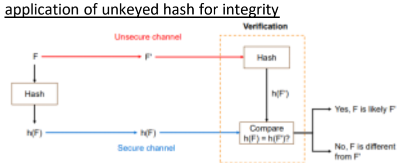
IV and padding of RSA
- IV needed to ensure semantic security - same PT at different time → different CT
- "homomorphic" property: multiplication of CTs (e.g. $CT_1 \times CT_2$) gives multiplication of PTs modulo n (i.e. $PT_1 \times PT_2\ mod\ n$)
- RSA is malleable because of this property, hence, padding is used to prevent such attacks

efficiency: 128-bit AES and 3072-bit RSA has equivalent key strength

data authenticity (digest), unkeyed
hash function
takes an arbitrarily large message and outputs a fixed size digest, if message is altered, hash won't match
- small change will result in a large change of hash

security requirements of hash
1. collision resistant: computationally infeasible to find $[F, F']\ s.t. F' \neq F\ and\ H(F') = H(F)$
2. preimage resistant: computationally infeasible to find $F\ s.t. H(F) = h$ i.e. H is one-way
3. second-preimage resistant: computationally infeasible to find $F \neq F' s.t. H(F') = H(F)$

application of unkeyed hash for integrity



attkr goal: make Alice accept a file other than F, example of 2^{nd} pre-image resistance issue

issues with unkeyed hash
1. no authentication: provides integrity verification but not authentication (of sender's identity)
2. vulnerability to collision attacks/2^{nd} preimage (depending on hash function quality)

birthday attack - a straightforward way to find collision
suppose $H()$ is a hash with n-bit digest, produces distinct 2^n hash values
- randomly pick two messages, hash both and repeat
- if more than 2^n messages are hashed, at least two distinct messages must have the same output (Pigeonhole principle), if $n = 256$ this is computationally infeasible

birthday paradox
- in a room of 23 people, there is a probability greater than 50% that at least 2 people will share the same birthday
- not looking for a specific match but any match
- for 23, people there are $23\ C\ 2 = \frac{23!}{2!*21!}$ pairs

formula for probability
suppose we have M messages and each message is tagged with a value randomly picked from $\{1,2,3, \dots T\}$
- probability that there is a pair of messages tagged with the same value (i.e. one collision) is approx:

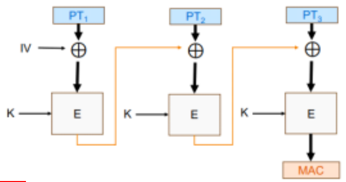
$prob(collision) \approx 1 - e^{\left(-\frac{M^2}{2T}\right)}$, in particular: when $M > 1.17 * T^{0.5}, prob(collision) > 0.5, T = 2^n; M > 1.17 * 2^{\frac{n}{2}}$

birthday attack steps
suppose $H()$ is a hash giving n-bit digest
- construct a set S of $\sim 1.17 * 2^{\frac{n}{2}}$ unique randomly chosen messages
- compute the digest of each message m_i in S
- check if there are two messages in S w/ same digest
- if so, output $m1, m2$,else Fail

we want to design a hash so that known attacks can't do better than birthday attack

keyed-hash (aka MAC - Message Authentication Code)
keyed-hash is a func. that takes an arbitrarily large message and a secret key as input and outputs a fixed size MAC
- data origin authenticity achieved via secret key between sender and receiver

CBC-MAC:



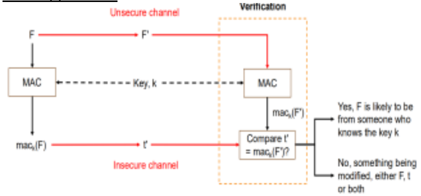
HMAC
1. key preparation
- key K is the secret key between sender and recver
- if K is longer than the hash func. block size, it is first hashed and the hash is used as the key
- if K is shorter than the block size, it's padded with zeros

2. inner padding
- `ipad` is a fixed-length string consisting of repeated "5c" bytes (in hexadecimal) which has the same length as the block size of the hash func.
- $K \oplus$ `ipad`

3. outer padding
- `opad` is a fixed-length string consisted of repeated "36" bytes + same length as block size of hash func.
- $K \oplus$ `opad`

`HMAC(K,x)=SHA-1(K⊕opad)||SHA-1(K⊕ipad||x)),` where || is concatenate

MAC application



attkr can forge a valid pair of message, mac w/o the key
- example of non-repudiation violated

data origin authenticity (signature), asym. key
PKC version of MAC is Digital Signature
- owner uses private key to generate signature
- public can use public key to verify signature

hash and sign
RSA is slow and inefficient for signing large files, soln:
1. hash the file to reduce size of data being signed
2. sign the hash value → produces a digital signature

generation
`s = RSA_encrypt (private key, Hash(X))`
where s is the signed hash

verification
`if Hash(X)=RSA_dec(public key, s), OK else reject`

signature ensures non-repudiation
- a message signed using Alice's private key cannot be interpreted as from another user as only Alice knows her private key

password authentication
authentication
1. communicating entity: verifying the identity of communicating parties
2. data origin: verifying the source of data recv.ed

to verify communicating entity, we need **credentials** to prove their identity
- something the user knows
- something the user is
- something the user has

passwords
passwords vs secret key
both used for entity auth.
- passwords generated by humans and is human remembered, secret key is long binary seq. that is not human readable or remember-able

password system
1. bootstrapping - establish a common pwd between user and server, server keeps password file
- mail, initial pwd is sent for single first login, WPS

2. authentication - user sends pwd to server and server verifies pwd

3. password reset - need to authenticate entity before allowing for change of password
- IT support, recovery email acc., security questions

secure password file storage
use of hashed passwords and a fixed-length random salt value (pre-image resistant: cannot retrieve password from hash value)
- salt prevents rainbow table attacks (two users have the same password but hash values will be completely different)

attacks on bootstrapping and resetting
attkr may intercept pwd during bootstrapping
- attkr uses the default passwords (such as those included on a wifi router for example)
- mitigation: require user to change pwd after login

attacks on pwd reset - pwd reset poisoning
goal: OTP which is included in reset link
vulnerability: how application constructs pwd reset URLs
mitigation:
1. input validation: robust validation and sanitization of HTTP headers and user input
2. secure URL generation: server-side configuration for generating URLs
3. 2FA
4. HTTPS

searching for password
1. brute force (exhaustive search)
2. dictionary attacks (restricted name space)
- pwds chosen are not random

online dictionary attack
- attkr interacts with system to test pwd
- mitigation: rate limiting, acc. lockout, strong password policy

offline dictionary attack
- attkr carries out dict. attk. without interacting with system
- attkr generates large dictionary of possible pwds and its hash values
- compare with hashed words with table
- mitigation: adding salt to password and then hash to increase attk complexity

password strength
entropy
- a measurement of randomness
- Alice chose a password of length, L randomly and uniformly from a set of N possible symbols
- number of possible pwds: N^L
- increasing N or L will ↑ pwd strength

$H = L \times \log_2 N$, where H is entropy

Symbol set	Symbol count N	Entropy per symbol H
Arabic numerals (0–9) (e.g. PIN)	10	3.322 bits
Hexadecimal numerals (0–9, A–F) (e.g. WEP keys)	16	4.000 bits
Case insensitive Latin alphabet (a–z or A–Z)	26	4.700 bits
Case insensitive alphanumeric (a–z or A–Z, 0–9)	36	5.170 bits
Case sensitive Latin alphabet (a–z, A–Z)	52	5.700 bits
Case sensitive alphanumeric (a–z, A–Z, 0–9)	62	5.954 bits
All ASCII printable characters except space	94	6.555 bits
All Latin-1 Supplement characters	94	6.555 bits
All ASCII printable characters	95	6.570 bits
All extended ASCII printable characters	218	7.768 bits
Binary (0–255 or 8 bits or 1 byte)	256	8.000 bits
Diceware word list	7776	12.925 bits per word

- online password: at least 29 bits of entropy
- offline password: at least 128 bits

something the user is/has + 2FA

$false\ match\ rate = \frac{B}{B + D}$

$false\ non-match\ rate = \frac{C}{A + C}$

	Accept	Reject
Genuine	A	C
False	B	D

ensure 2FA is: something you have + something you are

public key infrastructure
need a mechanism to securely distribute public keys, entity only needs to broadcast its k_e once and doesn't need to know the receiver

methods of key distribution
1. public announcement/hardcoded
- owner broadcasts their public key (e.g. game might have developer's public key hardcoded)
-ve: not standardized
-ve: no systematic way to search/verify k_e
-ve: difficult to update keys

2. publicly available directory
- keys stored in a directory publicly accessible
-ve: anyone can post their public keys in the server
-ve: not everyone trusts the directory/server
-ve: server is overwhelmed by load

3. public key infrastructure
- a standardized system to distr. public keys
- deployable on a large scale

PKI: certificate and CA
key features:
- key pair generation (public/private key pair)
- digital certificate: bind key to identity, signed by Trusted Third Party (CA)
- certificate revocation: invalidate a cert before its expiration

digital certificate
binds entity to public key, CA is a TTP, CA verifies and issues certificate to entity requesting for it, CA has its own public key

certificate issuance:
- Alice key pair:(K_A, K_A^{-1}), Charlie: (K_{CA}, K_{CA}^{-1})
- $cert_{C\to A}$: cert for Alice's key issued by Charlie
- with K_A, Charlie computes: $cert_{C\to A} = \{Alice, K_A\}_{K_{CA}^{-1}}$

certificate verification:
- Alice→Bob: $(K_A, cert_{CA\to A})$, Bob verifies $cert_{CA\to A}$
- if Bob knows CA's key is K_{CA} and trusts CA, Bob can believe that K_A is indeed Alice's
- no one but the CA can produce the signature

issuance of certificate
1. request for a cert from a CA
2. CA verifies the type of certificate request
3. CA issues certificate signed by CA to requester

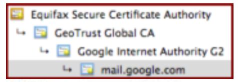
$cert_{CA\to A} = \{alice@x.com, x1sj93,1\ Sep\ 2025\}_{K_{CA}^{-1}}$

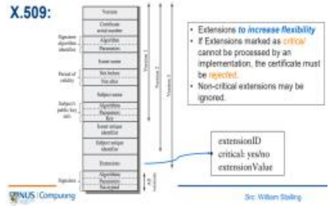
- messages to be signed will have entity identity, public key, validity
- CA hashes the message and then signs
- CA's public key is securely distributed, most OS and browser have a few pre-loaded CA public keys → root CAs
- other CA keys can be added through chain-of-trust

single CA PKI model
CA responsible for issuing and managing cert
+: easy to build and maintain
+: consistent approach to certificate issuance
+: users trust only a single CA
-: single point of failure
-: does not scale well

multiple CAs
system has many trusted CAs in the form of a list, cert issued by any is accepted
+: redundancy
+: geographically distributed
-: complexity

CA chain of trust
- hierarchy of trust
- root CAs are trusted, intermediate CAs can be verified



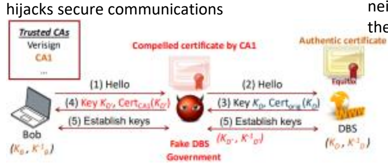


obtaining a certificate
1. self-signed certificate
- self generated, not CA signed, free
- signed with organization's own private key
- by accepting a self-signed cert, user instructs machine to accept the binding by accepting responsibility in ensuring cert. info is correct
+: easy and fast, not dependent on others
-: if compromised, security risk
-: not recommended for public facing sites

2. domain validated certificate
- entry-level SSL certificate
- only verification check: if application owns the domain associated with the cert., no actual identity verification
+: easy, quick and cheap
-: only confirms domain ownership, not identity

3. extended validation certificate
stringent identity validation, takes time to issue
- additional verification
+: highest level of trust, rigorous identity verification
-: high cost, time-consuming

security issues with PKI
number of trusted CAs is high nowadays
1. CA is a single point of failure → security of the weakest link, comprises security of sites protected by any other CA
2. compelled certificates → govt. agencies can compel a CA to issue false certificates for website to be spoofed, covertly intercepts and hijacks secure communications



social engineering: typo squatting, subdomain takeover (attacker rightfully owns a domain and creates a subdomain look-a-like)

certificate revocation
1. certificate revocation list (CRL)
CA periodically publishes CRL, contains a complete list of unexpired, revoked certificates
-ve: large size, high overhead
- delta CRLs only contain changes from the last complete CRL

2. short-lived certificates
- short validity lifetime certs
+ve: risk minimised
-ve: too much overhead to CAs and domain owners

3. OCSP extension
client query CA with Certificate Revocation Server to get status
1. client connect to site
2. server sends cert. to client
3. client send OCSP request to OCSP responder to check status
4. CA returns good, revoked or valid

+ve: timely info, reduced latency
-ve: traffic overhead, user privacy

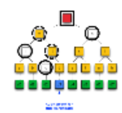
stapling
1. periodically refresh OCSP response to site
2. client connects to site
3. server sends certificate + OCSP to client

certificate transparency
uses certificate log: public, verifiable, append-only
- reject if certificate is not in CLs

1. domain requests cert from CA
2. CA requests $cert_{CA\to D}$ certificate log
3. CLog returns proof of inclusion
4. CA sends POI and $cert_{CA\to D}$ to domain
5. browser requests domain
6. domain returns POI and $cert_{CA\to D}$

Merkle tree
- leaves: hashes of individual certificates
- nodes: hashes of paired child leaves or paired child nodes
- root: summary of all certs
- distributed root/tree head: $\{TH\}_{K_{logg}^{-1}}$, this is public

- proof of inclusion: get hash of that leaf and a list of neighbouring hashes working up the tree



SCT: signed certificate timestamp
- a promise that log server will insert a new cert to its Merkle tree within a maximum merge delay (MMD)