Basic Communications Security

Communications Security

Communications security (COMSEC) means preventing unauthorized access to communications infrastructure and communicated messages, while still providing the communications service between intended parties

Once upon a time this subject was only of interest for military communications but it is now crucial to businesses and residential customers as well

In the early days of the Internet, the model was *trust everyone*

In the 1990s the model changed to *soft on the inside, hard on the outside*
trust employees and colleagues but not outsiders
which led to development of access policies, firewalls, etc.

Today standard operation procedures dictate

- *trust no-one*
- constantly monitor everything
- pro-actively search for vulnerabilities

which requires layers of protection

Threats

Before adopting a security measure one must understand the *threat*

In this talk we will assume that a *user* who consumes services from a *server*

Potential threats include :

- denial of service (DoS) to the user
- theft of service by an unauthorized user
- access to confidential information by an unauthorized user
- modification of information by an unauthorized user
- control of restricted resources by an unauthorized user
- physical damage to resources

Security experts build *threat models*

- what are the risks ?
- who are the potential attackers
- what at vulnerabilities and attack vectors

before putting *countermeasures* into effect

Countermeasures

We are going to deal only with these in this talk!

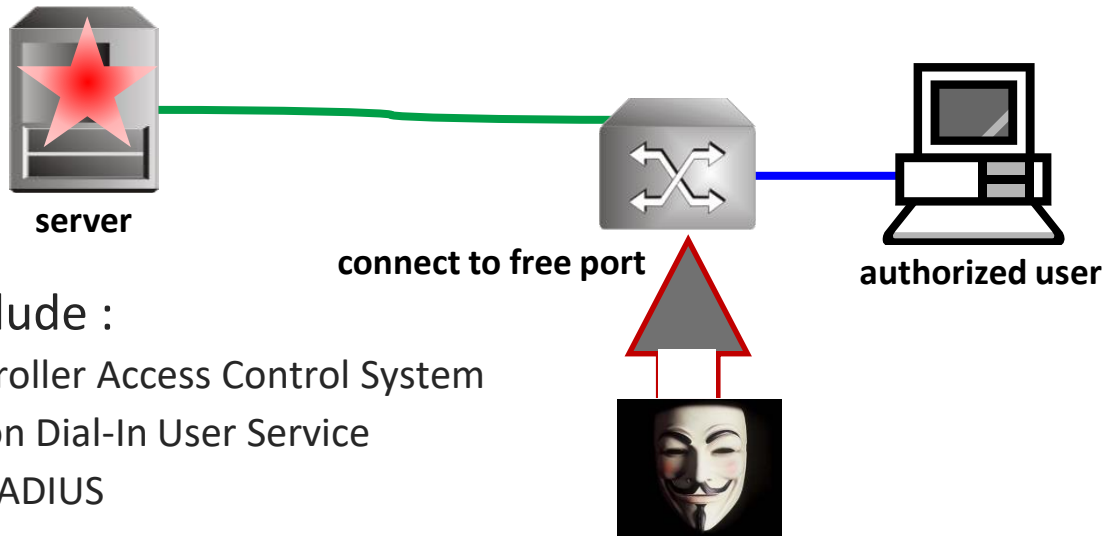
What can we do to combat threats ?

- Physical security – preventing access to communications devices and links
- Emission security – preventing interception and jamming
- Authorization – preventing unauthorized access to resources
- Source authentication – confirming the source of a message
- Integrity – preventing tampering with messages
- Confidentiality – preventing eavesdropping
- DoS blocking – preventing Denial of Service
- Topology hiding – thwarting traffic analysis
- Anti-hacking – preventing injection of computer malware
- Privacy – protecting user's personal data from mining and directed collection

Authorization

The first threat to consider is unauthorized access to resources

AAA (user) Authentication, Authorization, and Accounting
means any mechanism for controlling access to resources



Well-known AAA systems include :

- TACACS : Terminal Access Controller Access Control System
- RADIUS : Remote Authentication Dial-In User Service
- DIAMETER : twice as good as RADIUS

Such systems require

- a *supplicant* (the party requesting service)
- a method of proving identity (such as a password or biometric characteristic)
- an *authenticator* (the entity approving access) not necessarily the server itself

Passwords

A password is a string provided by the supplicant and verified by the server

Threats to the use of passwords:

- guessing of password
- theft of password
- brute-force (exhaustive search) discovery

Countermeasures

- use complex passwords (*at least one capital, one numeral, one punctuation*)
- use long passwords (*thisisareasonablygoodpassword,* x!A0 isn't)
- delay after wrong guess
- CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) are no longer considered useful
- three-factor authentication
  - something you are
  - something you have
  - something you know

password

Knowing the password

How does the server know that the supplicant entered the correct password?

There are two parts to this question:

1. the supplicant and authenticator need to agree on a password
   typically during *registration* and *password change*
2. the authenticator must remember the password

The first part is a special case of *key exchange* and will be discussed later

For the meanwhile assume that either :

- the authenticator assigns the password
  and sends it to the user over a secure channel (e.g., via trusted courier)
- the supplicant supplies the password over a secure channel

The second part sounds obvious – the server can simply store the password!

But passwords must never be stored *in the open*
due to the threat of theft of the password file

Instead the authenticator should store something else
that still enables it to check that the supplicant entered the correct password

This *something* is called a *hash*

Crypto-hashes

In computer science a *hash* is a function that

1. maps strings (vectors) of arbitrary length to strings (vectors) of fixed length
2. ensures that small change in input map to large changes in output

The output length may be much smaller than the input length,
meaning that many inputs map to the same output (collisions!)

A crypto-hash has a further property of being a *1-way function*

3. calculating the hash function is computationally easy
   finding an input that produces a given output is computationally hard

You may already know about check-sums and CRC hashes
these are good for *random* error correction
but are *not* crypto-hashes and are not good against malicious modification

Well-known crypto-hashes include :

- MD5 (no longer considered secure)
- Secure Hash Algorithm SHA1 (widely used, no longer considered secure)
- Secure Hash Algorithm SHA2 (actually 6 hashes SHA-224,256,384,512,512/224,512/256)
- Secure Hash Algorithm SHA3 (new)

Using hashes for the password problem

How does a crypto-hash help with the passwords problem?

- during registration the password is crypto-hashed and the hash is stored
- the password file/database contains only hash values
  from which passwords can not be recovered (due to the 1-way function!)
- the supplicant enters the password
- the password is hashed and the hash value is compared to the stored hash
- if the hashes match then access is granted
  otherwise access is denied

This mechanism is fine for logging on to a local computer
but not for requesting access over a communications link or network

- if the password were to be sent over the link *in the clear*
  an eavesdropper could intercept it and know the password!
- if the password were locally hashed and the hash sent over the link
  an eavesdropper could intercept the hash and send it to gain access!

We need something better!

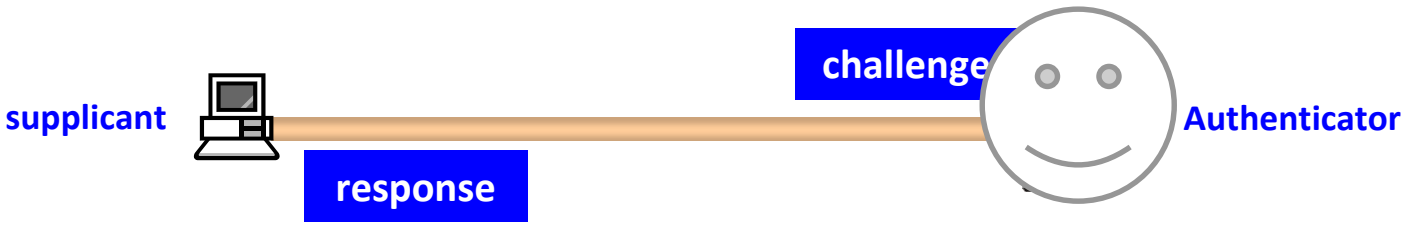
CHAP

The solution is a **C**hallenge **H**andshake **A**uthentication **P**rotocol
in which the supplicant answers a *challenge*
that proves that he knows the password

In the simplest challenge scenario the challenge is just a random string

- authenticator sends random challenge message to the supplicant
- supplicant concatenates challenge + password and crypto-hashes it
- supplicant sends crypto-hash as response to authenticator
- authenticator compares response with expected response
- if the hashes match then access is granted
  otherwise access is denied

Note that an eavesdropper can view the response, but that won't help him
since the authenticator sends a different random challenge each time



EAP

Extensible **A**uthentication **P**rotocol is a authentication *framework*
and runs over various link layers (PPP, Ethernet, WiFi) without needing IP

Originally developed for PPP (extends PPP's original CHAP)

Dozens of specific methods (EAP-PSK, EAP-MD5, EAP-TLS, EAP-IKEv2, EAP-SIM, …)

Used as a link layer authentication for WiFi (WPA, WPA2) and IEEE 802.1X

EAP provides *1-sided authentication* (supplicant by authenticator)
but it can be run in both directions for *mutual authentication*

EAP operation

- optionally authenticator sends *Identity Request* to supplicant
- optionally supplicant sends *Identity response*
- authenticator sends *EAP request* with challenge data
- supplicant sends *EAP response*
- authenticator sends *success* or *failure*

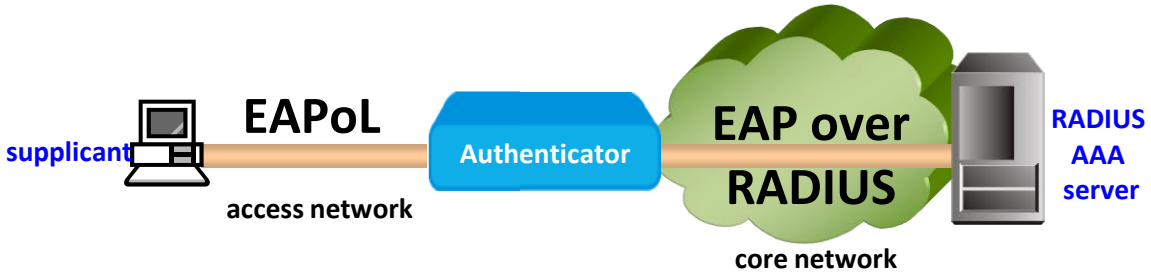
802.1X

802.1X is an IEEE standard for authenticating users of a LAN or WLAN

1X defines

- 3 parties: supplicant, authenticator, and authentication server
- an encapsulation of EAP called EAPoL (EAP over LAN)
- a Port Access Entity (PAE)
  - before authentication only EAPOL traffic can pass through the port
  - after authentication all traffic can pass through the port

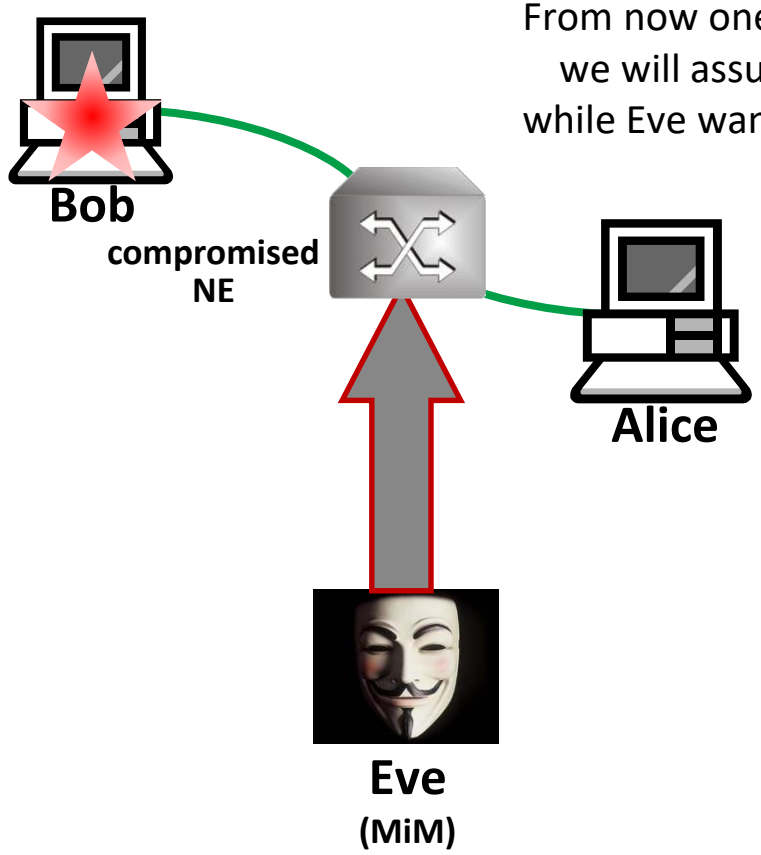
802.1X was extended in 2010 not only to open ports
but to authorize security associations and services
in order to support MACsec (802.1AE) and Secure Device Identity (802.1AR)



Integrity (anti-tampering) protection

The next threat we need to consider is a *Man in the Middle* (MiM) attack

Here someone gains access to a **N**etwork **E**lement
and can tamper with packets on-the-fly



From now one, instead of talking about a user and a server we will assume that *Alice* and *Bob* want to communicate while Eve wants to eavesdrop on their communications

MAC

We can provide integrity protection
using a **M**essage **A**uthentication **C**ode (MAC)

Warning: Don't confuse this MAC
with Ethernet's **M**edia **A**ccess **C**ontrol
or with DSP's **M**ultiply and **AC**cumulate

A MAC is a short block of information that

- is uniquely determined by the message
- verifies that the message has not been modified
  i.e., it is highly unlikely that a modified packet has the same MAC (collision)
- is difficult to forge

The MAC is inserted into packet headers and can be verified upon receipt
but only the authorized sender knows how to create the MAC

So if Eve modifies the packet

- he does not know how to find the new MAC
- if he leaves the MAC unchanged the verification will probably not succeed

HMAC

A Hash-based Message Authentication Code (HMAC)
uses a crypto-hash as a MAC
but other mechanisms (such as block ciphers) can be used to build MACs

The simplest way to prevent forging MACs is by using a *shared key (password)*
which once again returns us to the *key exchange* problem
which we'll discuss later

HMAC operation:

- Alice calculates HMAC by crypto-hashing message + key, and inserts into packet
- Bob calculates HMAC using message + key, and compares to HMAC in packet
- if the HMACs match then packet is accepted
  otherwise packet is discarded

Of course Eve, not knowing the key, can not forge the HMAC

Source Authentication

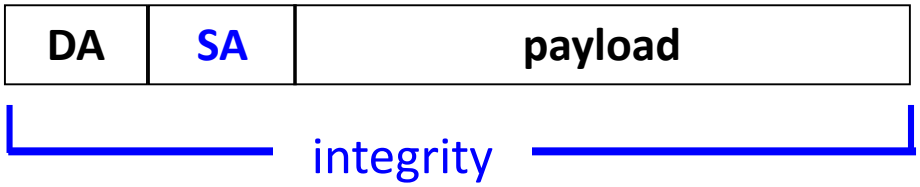
How can Bob be sure that a packet was actually sent by Alice ?

Ethernet and IP packets contain Source Addresses (SA)
but these can be readily forged

MACs can also be used to authenticate a packet's source address
that is, to prove that the SA correctly indicates the packet's source

We can re-use integrity mechanisms (e.g., MACs) to solve this problem !

All that is needed is to have the (H)MAC protect the SA
if the MAC is correct, then the SA indeed belongs to the claimed sender



Eve doesn't know how to forge MACs (e.g., not knowing the shared key)
and so can't fool us into believing that Alice sent the packet

Replay Attacks

Another type of attack is the *replay attack*

Here the MiM intercepts a packet and resends it multiple times
(transfer$100 → transfer$100, transfer$100, transfer$100, transfer$100)

Integrity and source authentication mechanisms do not detect replay attacks since the MACs calculate correctly every time!

We can reuse integrity mechanisms (e.g., MACs) to solve this problem too

To defend against replay attacks

- add or utilize a packet sequence number field
- have the MAC protect the sequence number field
- if the same SN is received again, discard packet

Confidentiality

Another threat is eavesdropping
that is observation of a packet's content by unintended parties

The standard countermeasure is *encryption*

This threat is so obvious that many people equate *security* with *encryption*

In fact, confidentiality is often the *least important threat*
and many types of communications do not need it

On the other hand, it is hard to conceive of communications
that do not require source authentication and integrity protection
after all, what is the value of incorrect information (AKA fake news)

Encryption takes *plaintext* and produces *ciphertext*
while decryption takes the *ciphertext* and retrieves the original *plaintext*

There are two very different types of encryption:

- *symmetric key* (requires a shared key)
- *public key* (no shared key needed)

Symmetric key encryption

Symmetric key encryption is based on the sides having a shared key
    and so, once again, requires solving the key exchange problem

There is one symmetric key algorithm that is provably safe, that is
    Eve can recover no information from observing the communications

Alice and Bob have an identical copies of a *one-time pad*
    i.e., a list of random bits at least as long as the message
Alice can xor her message with the one-time pad
    creating a message unreadable to Eve (who does not have the one-time pad)
Bob can xor the encrypted message with the one-time pad
    recovering the original message

X	Y	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

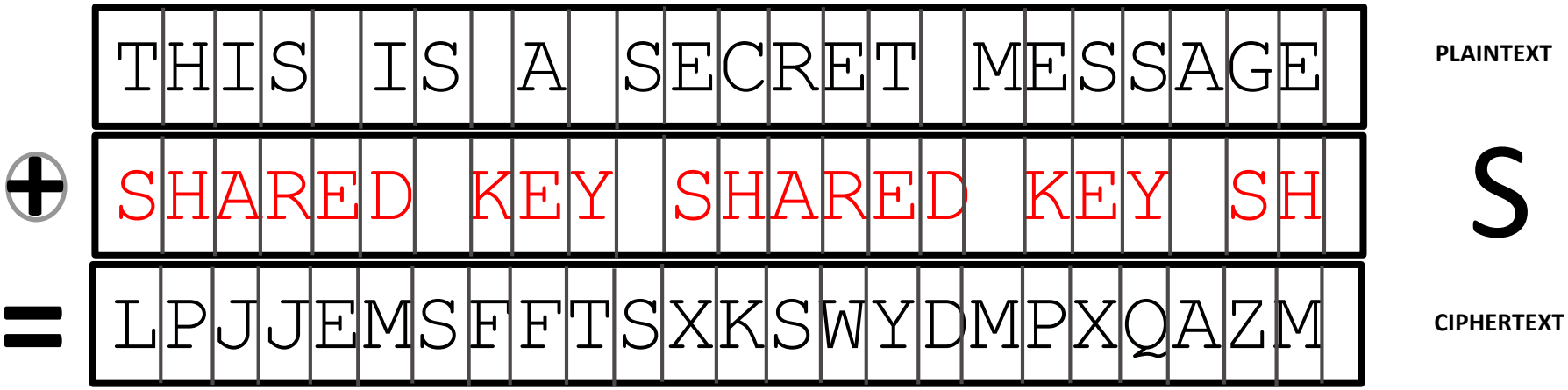
The one-time pad must never be used again
    re-use leads to leakage of secret information

So one-time pads need to be very long and thus cumbersome to use

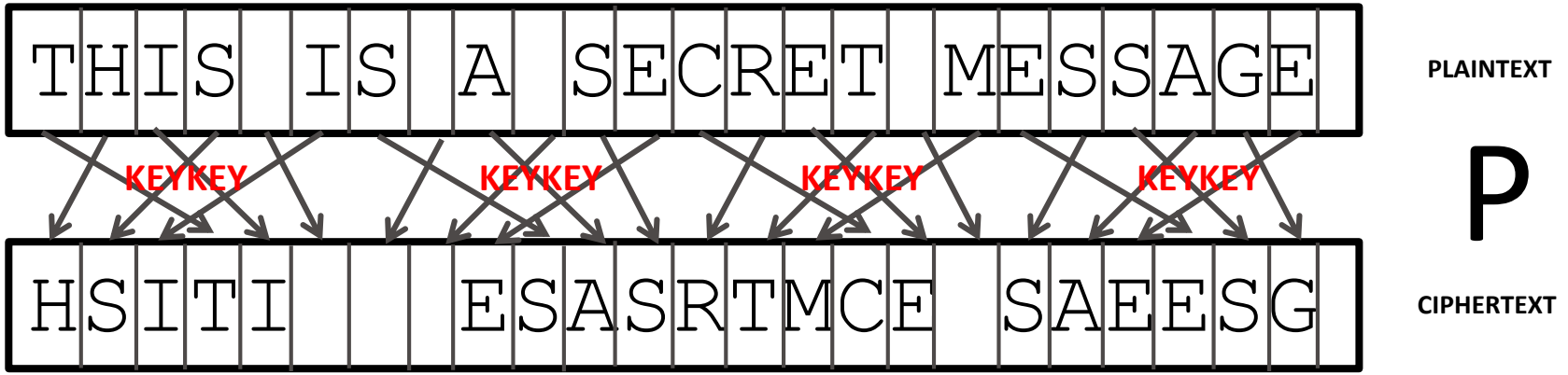
Instead most symmetric encryption algorithms use shared *keys*

Symmetric Encryption Using a Shared Key

Simple encryption algorithms are either substitution ciphers



Or permutation ciphers



Block ciphers

Block ciphers break a message into blocks of length n bits (n=64, 128, 256, …)

For each block:

- substitution ciphers substitute ciphertext for plaintext
  the substitution is a key-dependent 1-1 function from n-bits to n-bits
  the function is usually called an S-box

- permutation block ciphers permute n bits to n bits
  the permutation is called a P-box

It turns out that both substitution ciphers and permutation ciphers
are relatively easy to *break*
That is, given enough ciphertext a code-breaker can deduce the plaintext

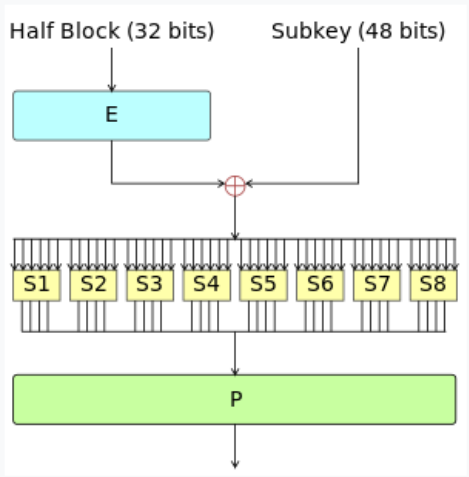
In 1949 Claude Shannon (in the paper Communication Theory of Secrecy Systems)
introduced substitution-permutation (S-P) ciphers
which encode each block by alternating rounds of S-boxes and P-boxes
and showed that it was much harder to break

Shannon's idea was first exploited by Feistel (from IBM)

DES

In 1977 NIST published the **D**ata **E**ncryption **S**tandard
based on the Feistel algorithm

- Inputs
  - 64 bits of plaintext
  - a 56 bit key
- Performs 16 rounds of S-boxes and P-boxes
- Outputs 64 bits of ciphertext



Due to using a short 56-bit key
since the late 1990s DES is not considered secure against brute-force attacks
(and possibly never was secure against sophisticated attacks …)

In order to save DES, a method called triple-DES (3DES) was proposed

3DES uses a key of length 3*56=168 bits, but its effective length is 112 bits

NIST allows use of 3DES use until 2030, but there is something better …

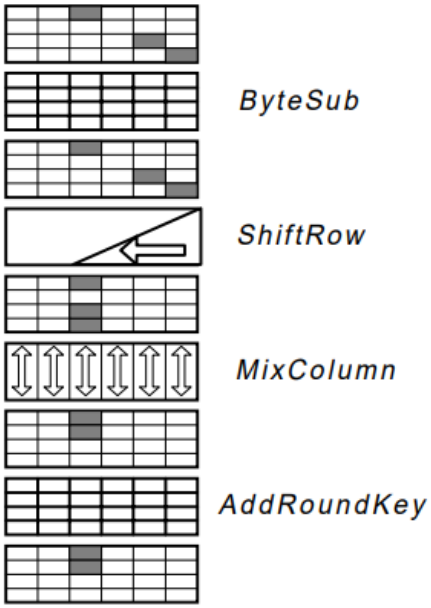
AES

In 2001 NIST adopted an algorithm called Rijndael
(after its Belgian creators Vincent Rijmen and Joan Daemen)
as the Advanced Encryption Standard

This choice was the result of a 5-year process
in which 15 candidate algorithms were compared

AES

- Inputs
  - 128 bits of plaintext
  - a 128/192/256 bit key
- Performs 10/12/14 rounds of S-boxes and P-boxes
- Outputs 128 bits of ciphertext



AES is approved by the NSA for protection of ***top secret*** data (w/ 256 bit key)

AES-GCM is a *combined algorithm*, providing
source authentication, integrity protection, and encryption
using a single algorithm

Block Cipher Modes (1)

The encryption mechanisms we have discussed so far are *block ciphers*
that is, they operate on blocks of N bits
But how do we use a block cipher to encrypt an arbitrary sized message ?

The simplest method is Electronic codebook (ECB)
which performs the block cipher independently on each N bits
(if needed, the last block is padded with zeros)
If the input repeats itself, ECB encrypts in the same way
revealing patterns in the plaintext (*leakage* - aiding the cipher breaker)
ECB is also susceptible to replay attacks

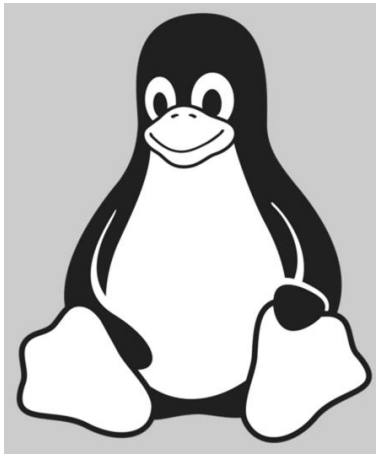
Thus, ECB should never be used

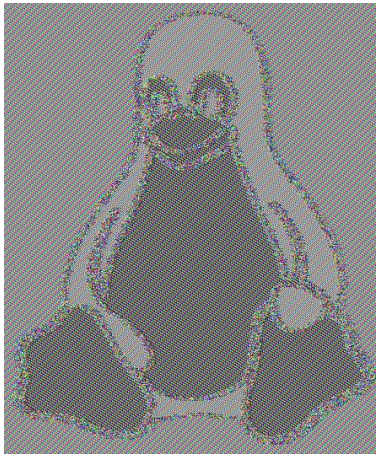
What can be done ?

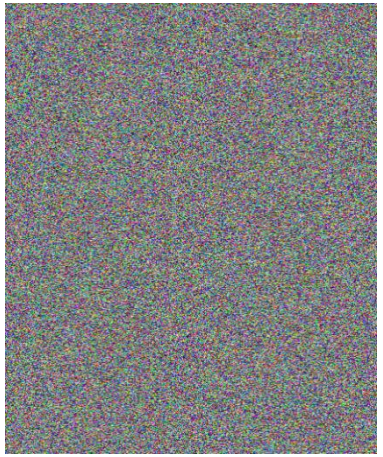
Instead of encrypting the plaintext,

- we can xor the plaintext with the previously encrypt block before encrypting
- encrypt the previous ciphertext and then xor with the plaintext
- or even encrypt something else entirely (e.g., a counter) and xor with the plaintext

Example of ECB's failure







pictures by Phillip Wang

original AES-ECB AES-CBC

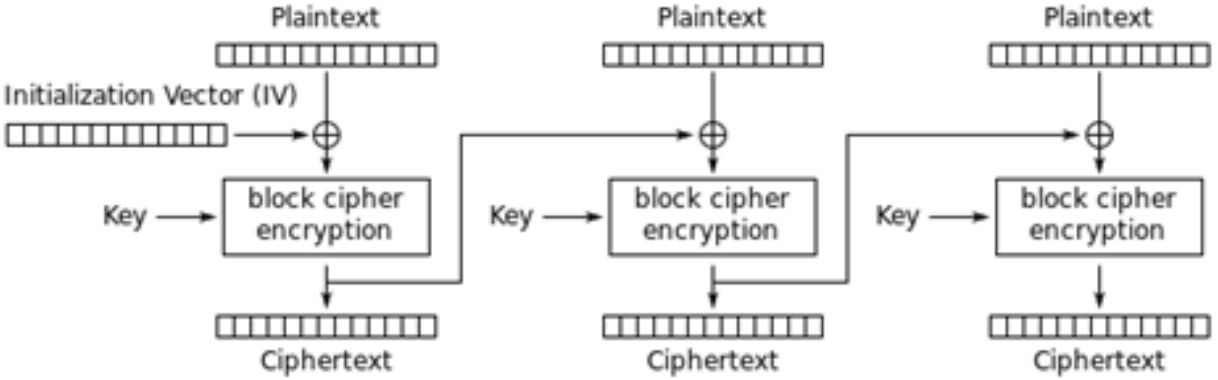
All patches of the same color in the original picture
are encrypted by ECB to the same value

This results in significant *leakage*
which can be exploited to recognize the picture and recover the secret key

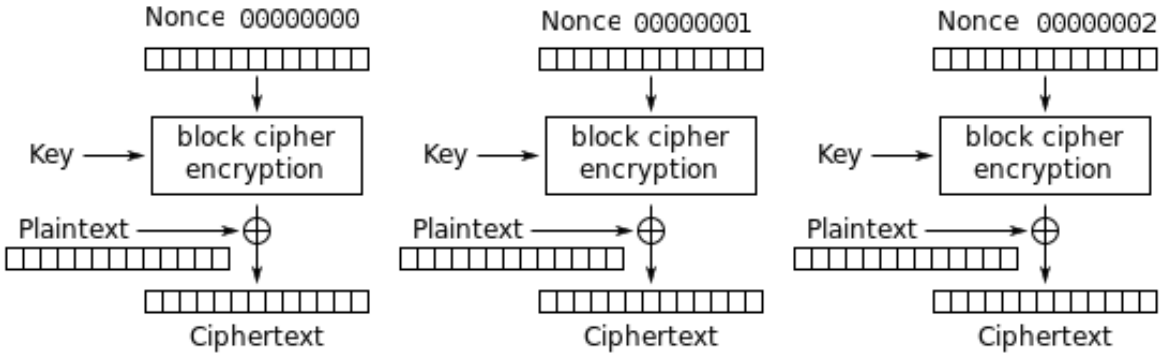
Other block cipher modes do not exhibit this leakage

Block Cipher Modes (2)

Cipher block chaining CBC



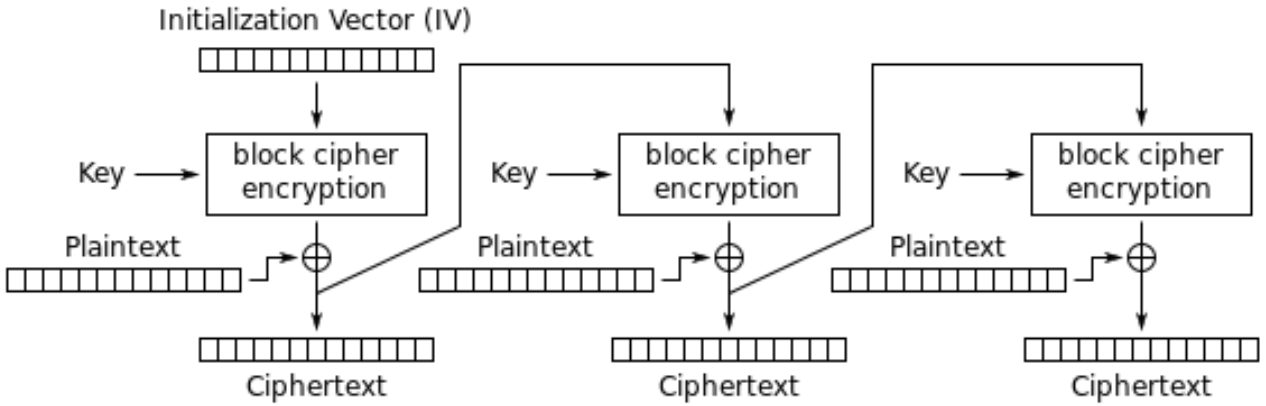
Counter mode CTR



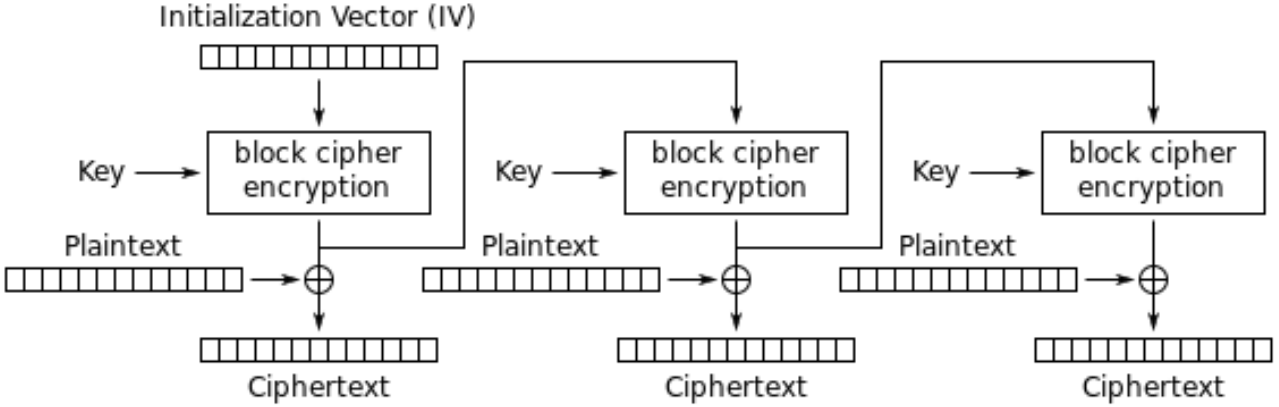
Galois Counter Mode GCM is CTR using Galois field operations

Block Cipher Modes (3)

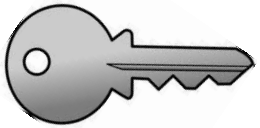
Cipher feedback CFB

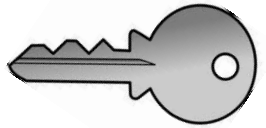


Output feedback OFB



Key Exchange (AKA Key Distribution)





The time has finally come to discuss *key exchange*
which must be used whenever we need a *shared key*

Some ways of sharing keys include:

- face-face meetings
- trusted couriers
- synchronized key generators
- using an existing (but perhaps compromised) secure channel

This makes symmetric encryption logistics a nightmare

For many years mathematicians searched for a solution to this problem

It was first solved in principle in GCHQ by James Ellis and Chris Cocks
but remained classified and unknown to the public at large

Key exchange was solved again by Martin Hellman

Use of public keys for this was solved in theory by Whitfield Diffie
and Ron Rivest provided the first 1-way function

My classification

We can explain four types of encryption by the following analogies

Shared-key (symmetric) encryption (e.g., DES, AES)

- Alice and Bob both have identical copies of the key to a strong-box
- Alice puts her message in the box, locks it with her key, and sends to Bob
- Bob uses his identical key to open the box and recovers the message

One private key encryption (e.g., RSA or ECC)

- Bob has the key to open a box that locks automatically upon closing (1-way function)
- Bob sends open box to Alice, Alice puts message in box, closes, sends back to Bob
- Bob opens the box with his key

Two private key encryption (e.g., RSA or ECC with signature)

- Alice has a *locking key* and Bob has an *unlocking key* to a strong-box
- Alice puts her message in the box, locks it with her locking key, and sends to Bob
- Bob unlocks the box with his unlocking key, and recovers *Alice's* message

Multi-exchange (e.g., DH)

- Alice puts her message in the box, locks with her padlock, and sends to Bob
- Bob places his padlock alongside Alice's, locks, and sends back to Alice
- Alice removes her padlock, and sends back to Bob
- Bob removes his padlock, and recovers the message

First attempts to encrypt without keys

Can we use one-time pads to communicate without shared secrets ?

ATTEMPT 1 – substitution code

- Alice xors message M with a one-time pad A and sends $D_1 = M + A$ to Bob
- Bob xors D_1 with his one-time pad B and sends $D_2 = M + A + B$ to Alice
- Alice xors D_2 with her old one-time pad A and sends $D_3 = M + A + B + A = M + B$
- Bob xors D_3 with his old one-time pad B and recovers $D_3 + B = M + B + B = M$

While the data of each transmission D_n is safe
if Eve observes all the transmissions, she can recover the message

- Eve observes $D_1 = M + A$, $D_2 = M + A + B$, and $D_3 = M + B$
- Eve xors $D_1 + D_2 + D_3 = B + D_3 = M$ BROKEN

ATTEMPT 2 – permutation code

- Alice permutes M with a one-time permutation A and sends $D_1 = A*M$ to Bob
- Bob permutes D_1 with his one-time permutation B and sends $D_2 = B*A*M$ B to Alice
- but permutations don't commute!
  so if Alice permutes D_2 with the inverse of her old one-time permutation
  she obtains $D_3 = A^{-1} * B * A * M \neq B*M$ DOESN'T WORK

Public key cryptography

Two things are missing to make this idea work:

1. one-way functions
2. greater mathematical sophistication

The breakthrough is *public key cryptography* (AKA asymmetric cryptography)

With this method each party has a *public key* and a *private key*

Public keys may be advertised
but private keys are kept private

Operation

- Alice encrypts the message using Bob's public key
  - and optionally signs with her private key if authentication is desired
- Bob decrypts it using his private key (and optionally Alice's public key)
- Eve, not knowing the private key, can not decrypt the message

So no key distribution is needed
although you still have to ensure that Alice's public key is authentic

Digital Signatures

Another application of public key cryptography is digital signatures
If a message is digitally signed by Alice
    then Bob can verify that Alice really signed it, and not Eve
The idea behind the use of public key methods is
    the signature proves that the signer had access to the private key
    the private key is not divulged

Operation

- Alice signs the message with her private key and sends to Bob
- Bob can verify the signature using Alice's public key
- as a side effect, the verification also ensures integrity

Note that this is a kind of source authentication
    but different from a MAC
    since a MAC relies on a shared secret

Public key algorithms

Public-key cryptography (asymmetric cryptography)
relies on mathematical calculations that are hard to perform, such as

- **factoring large integers**
  it is easy to multiple 2 large primes p and q to find $n = pq$
  but it is hard to factor n to find p and q
- **finding discrete logarithms** (and related : computational DH, decisional DH)
  given a finite group G
  it is easy to multiply an element b with itself n times to find $g = b^n$
  but it is hard to find given b and g to find n such that $b^n = g$
- **elliptic curve logarithms**
  given an elliptical curve $y^2 = x^3 + ax + b$ over the field F_p
  (elements 0 … p-1 with all operations modulo p)
  it is easy to perform the EC multiplication operation
  of element b with itself n times to obtain $b^n = b \cdot b \cdot b \ldots = g$
  but it is hard given b and g to find n such that $b^n = g$

RSA

RSA was one of the first public key algorithms to be discovered
Named after Ron **R**ivest, Adi **S**hamir, Leonard **A**dleman who published it in 1977

RSA is based on the fact that

- it is easy to multiple 2 large prime numbers
- it is hard to factor the product to recover the prime numbers

(It is an open problem in mathematics whether factoring is indeed hard)

To use:

- find two large prime numbers and multiply them
- based on number theory create public and private keys
- sign messages using private key or
- encrypt messages using recipient's public key

RSA *was* patented (by RSA Security Inc., founded by R, S, and A) but expired in 2000

- there are rumors that RSA placed backdoors in their products for the NSA
- RSA Security was acquired by EMC in 2006, which was acquired by Dell in 2016

RSA is still often used instead of stronger elliptical curve methods
due to the latter having patents in force

RSA algorithm (for the mathematically inclined)

Preparation

- Bob selects two large primes p and q (there are many methods to find primes)
- Bob calculates $n = pq$ and $\Phi=(p-1)(q-1)$
- Bob selects e between 2 and Φ-1 which is co-prime to Φ
- Bob computes d such that $ed = 1 \bmod \Phi$ (using Euclid's algorithm)
- Bob publishes $\{n, e\}$ as his public key, and keeps d as his private key

Mathematical background

- Raising M to the e power modulo n : $M^e \bmod n$
  - is a 1:1 transform
  - is a one-way function
- if $C = M^e \bmod n$ then $M = C^d \bmod n$ (follows from *Fermat's little theorem* and *CRT*)

Operation

- Alice encrypts her message M into ciphertext thus: $C = M^e \bmod n$ and sends to Bob
- Bob decrypts thus : $C^d \bmod n = M$
- Eve, not knowing p and q and thus not d, can not recover M

El Gamal (for the mathematically inclined)

Discrete logarithm cryptosystem published by Taher Elgamal in 1985

PhD under Hellman, chief scientist at Netscape where he was father of SSL
director engineering and afterwards CEO of RSA Security Inc., now CTO for security at SalesForce

and used in NIST's **D**igital **S**ignature **S**tandard

Preparation

- Bob chooses a cyclic group G of order q with generator g
- Bob chooses a random number x between 1 and q-1
- Bob computes $h = g^x$
- Bob publishes $\{G, q, g, h\}$ as his public key, and keeps x as his private key

Operation

- Alice randomly chooses an ephemeral key y between 1 and q-1
- Alice calculates $c_1 = g^y$
- Alice calculates the shared secret $s = h^y$
- Alice encodes her message as an element m of G and calculates $c_2 = ms$
- Alice sends the ciphertext (c_1, c_2) to Bob
- Bob calculates the shared secret $s = c_1^x = g^{yx} = g^{xy} = h^y$
- Bob calculates the inverse of s : $s^{-1} = c_1^{q-x} = g^{(q-x)y}$
- Bob recovers the message $m = c_2 s^{-1} = ms\, s^{-1}$
- Eve, not knowing x, can not find s or s^{-1}

Certificates

I can be sure of Alice's public key if she personally hands it to me
    if I am just given it – it may be forged by Eve !

How can we be sure of someone else's public key ?
The idea is to have someone trusted (i.e., for whom we already have a public key) vouch for it
How can that trusted party be sure ? Someone he trusts vouches for it!

There are two methods to implement this

- **P**ublic **K**ey **I**nfrastructure (e.g., X.509)
- web of trust (e.g., PGP)

X.509 is an ITU-T standard for PKI
It specifies the format of certificates
    and a strict hierarchical system of Certificate Authorities that can issue them

With the web of trust model anyone (not just special CAs) may sign certificates

Secure key exchange

Using public key cryptography no key distribution is required

However, public key encryption is computationally much more expensive than symmetric encryption

One solution to this problem is to re-use public key cryptography

We only use public key encryption to encrypt a (symmetric) key
all the messages are sent using inexpensive symmetric cryptography

Operation

- Alice encrypts a random key using her private key and Bob's public key
- Bob decrypts the key using his private key and Alice's public key
- Alice and Bob now communicate using symmetric encryption
- after some amount of key use, the process is repeated

Diffie–Hellman key exchange

We do not need full public key cryptography to distribute symmetric keys

The original DH was published by Whitfield Diffie and Martin Hellman in 1976
although discovered earlier in GCHQ but classified

DH is based on the field F_p (numbers 0 … p-1, with operations modulo p)

Operation (for the mathematically inclined)

- Alice and Bob agree to use a *prime number* ***p*** and a *primitive root* ***g***
  (**g** is a primitive root, if *every* number is congruent to g^n for some n)
- Alice chooses a secret integer ***a*** and sends to Bob $\boldsymbol{A} = g^{\boldsymbol{a}} \bmod p$
- Bob chooses a secret integer ***b*** and sends to Alice $\boldsymbol{B} = g^{\boldsymbol{b}} \bmod p$
- Alice now computes $\boldsymbol{s} = B^{\boldsymbol{a}} \bmod p = g^{ba} \bmod p$
- Bob now computes $A^{\boldsymbol{b}} \bmod p = g^{ab} \bmod p = \boldsymbol{s}$
- Alice and Bob now share the secret **s**
- Alice and Bob now use ***s*** as a symmetric key !
- Eve, not knowing either ***a*** or ***b***, can not deduce ***s***

Diffie Hellman Groups (RFC 7296)

0	NONE		[RFC7296]
1	768-bit MODP Group	[RFC6989], Sec. 2.1	[RFC7296]
2	1024-bit MODP Group	[RFC6989], Sec. 2.1	[RFC7296]
3-4	Reserved		[RFC7296]
5	1536-bit MODP Group	[RFC6989], Sec. 2.1	[RFC3526]
6-13	Unassigned		[RFC7296]
14	2048-bit MODP Group	[RFC6989], Sec. 2.1	[RFC3526]
15	3072-bit MODP Group	[RFC6989], Sec. 2.1	[RFC3526]
16	4096-bit MODP Group	[RFC6989], Sec. 2.1	[RFC3526]
17	6144-bit MODP Group	[RFC6989], Sec. 2.1	[RFC3526]
18	8192-bit MODP Group	[RFC6989], Sec. 2.1	[RFC3526]
19	256-bit random ECP group	[RFC6989], Sec. 2.3	[RFC5903]
20	384-bit random ECP group	[RFC6989], Sec. 2.3	[RFC5903]
21	521-bit random ECP group	[RFC6989], Sec. 2.3	[RFC5903]
22	1024-bit MODP Group with 160-bit Prime Order Subgroup	[RFC6989], Sec. 2.2	[RFC5114]
23	2048-bit MODP Group with 224-bit Prime Order Subgroup	[RFC6989], Sec. 2.2	[RFC5114]
24	2048-bit MODP Group with 256-bit Prime Order Subgroup	[RFC6989], Sec. 2.2	[RFC5114]
25	192-bit Random ECP Group	[RFC6989], Sec. 2.3	[RFC5114]
26	224-bit Random ECP Group	[RFC6989], Sec. 2.3	[RFC5114]
27	brainpoolP224r1	[RFC6989], Sec. 2.3	[RFC6954]
28	brainpoolP256r1	[RFC6989], Sec. 2.3	[RFC6954]
29	brainpoolP384r1	[RFC6989], Sec. 2.3	[RFC6954]
30	brainpoolP512r1	[RFC6989], Sec. 2.3	[RFC6954]
31	Curve25519	[RFC8031], Sec. 3.2	[RFC8031]
32	Curve448	[RFC8031], Sec. 3.2	[RFC8031]

MODP = **MOD**ular **P**rime
ECP = **E**lliptical **C**urve mod **P**rime
brainpool – see RFC 6954

Quantum computers can break RSA

Public key cryptography depends
on the computational complexity of calculations such as factorization

RSA can be broken by factoring an integer with b bits, for which

- no algorithm than runs in polynomial time $O(b^k)$ is known
- the best factorization algorithms are subexponential $O((1 + \varepsilon)^b)$
  (faster than polynomial but slower than true exponential)

But for a *quantum computer*, Shor's algorithm runs in polynomial time $O(b^3)$

Similarly, discrete logarithm and ECC cryptosystems
can be broken by quantum computers

The largest quantum computer built to date has 53 (72?) qubits
and to date no number larger > 100 has been factored by Shor's algorithm
So these algorithms are still safe, but they won't be safe forever

Google recently announced attaining quantum supremacy
that is performing some calculation not possible on a classical computer

Workarounds

Since the world's communications and financial infrastructure
    is so dependent on public key infrastructure
there is an urgent need for a solution to this problem

More accurately, there *will be* an urgent need
    the day after the announcement of a quantum computer
    capable of quantum supremacy for these problems

Recently new types of public key algorithms have been devised

A *quantum-resistant* public key system is based on an intractable problem
  for which there is *no known* simplifying quantum computer algorithm

A *quantum-safe* public key system is based on an intractable problem
  for which one can *prove* that there is no simplifying quantum algorithm

Quantum resistant cryptography

Also called post-quantum or quantum-resistant cryptography
uses 1-way functions not known to be compromised by quantum computers

Recently NIST has called for proposals of such systems to be standardized

Some proposed methods:

- **Lattice-based**
  - **NTRU** is an open source (GPL) lattice based system being considered by several standard's bodies
- **Code-based**
  - **McEliece** has been recommended by EU Post Quantum Cryptography SG for long term protection against quantum attacks
- **Multivariate**
  - **Rainbow** has resisted attack attempts since begin proposed in 2005
- **Hash-based**
  - the Merkle signature scheme is being considered by NIST

QKD

An alternative to public key cryptography is **Q**uantum **K**ey **D**istribution

QKD enables Alice and Bob to exchange a random shared key
with physical (not mathematical) proof that it has not been intercepted
This key may then be used by symmetric encryption as usual

Two somewhat different quantum mechanical phenomena may be exploited

- observation collapse
- entanglement (AKA spooky action at a distance)

In either case Eve's observing an exchanged bit can be readily discovered

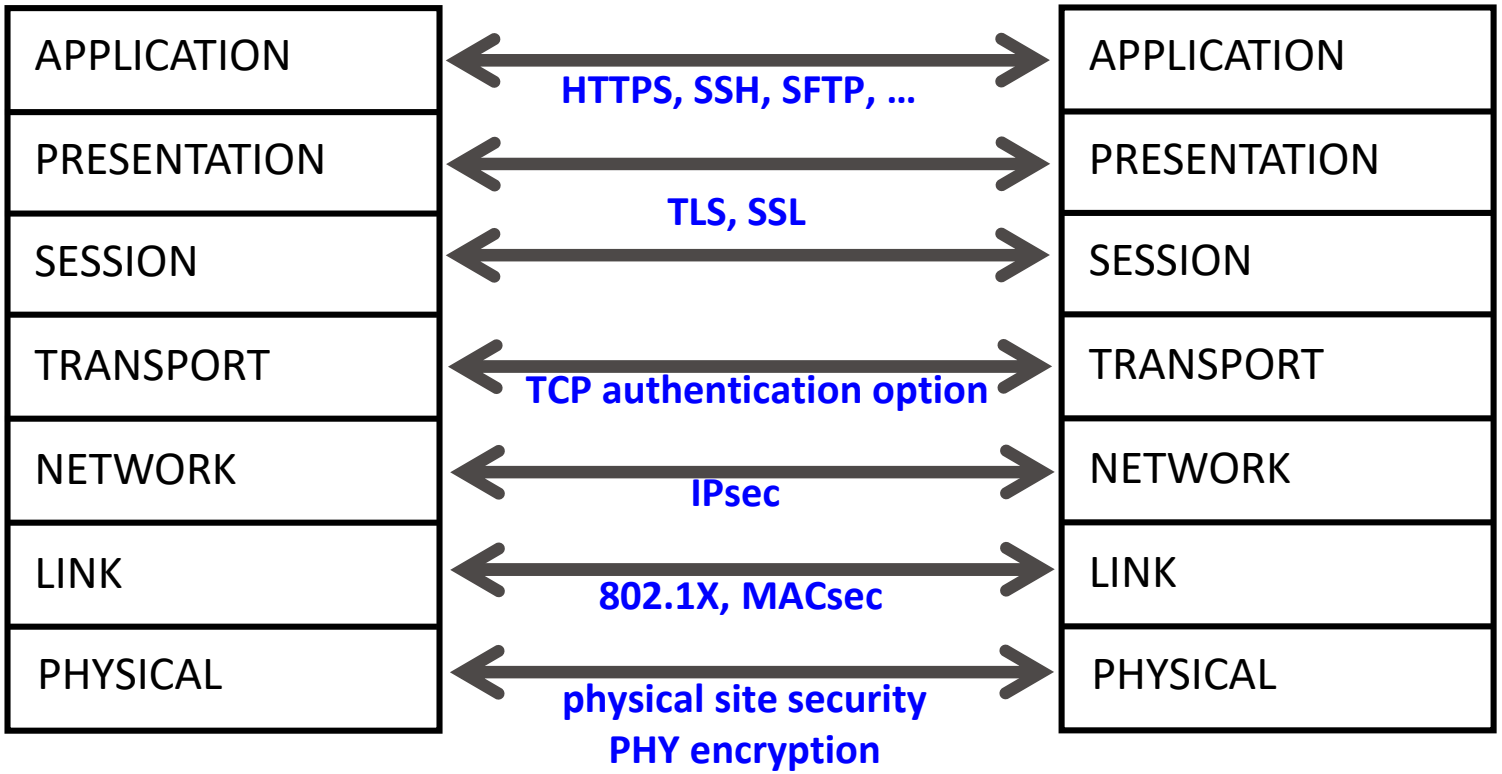
QKD systems are commercially available from several sources
and have been successfully deployed for extended periods of time

Fiber-based QKD system typically work over tens of kilometers
but have been demonstrated to operate over 100s

An experimental satellite-based system has operated over 7,500 km
between China and Vienna

Where to apply security ?

Every layer of the user plane can benefit from security mechanisms



Control and management planes have their own mechanisms

IPsec

IP was designed when security was not an issue

IPsec is a set of open standards that rectify many of IP's deficiencies

IPsec is transparent to applications (apps needn't know that it will be used)

IPsec is based on the concept of a *security association* (SA)

The SA is a relationship between two or more entities

- performs mutual authorization
- specifies which security features (authentication, integrity, encryption) will be used
- specifies algorithms (DES, AES, SHA-1, RSA, …) and options
- takes care of key exchange (ISAKMP Internet Security Association and Key Management Protocol)
  - pre-shared keys
  - Internet Key Exchange (IKE, IKEv2)
  - IPSECKEY DNS records

Internet **K**ey **E**xchange (UDP port 500) is used for establishing IPsec sessions
(performing mutual authentication, establishing and maintaining SAs, key exchange)

IKE and IPsec stages

IPsec *tunnel* operation typically has 4 steps

- **IKE Phase 1** (main and aggressive modes)
  mutually authenticate IPsec peers
  sets up a secure channel between them to enable IKE exchanges:
  - authenticates IPsec peer identities
  - negotiates IKE SA policy to protect the IKE exchange
  - performs authenticated DH exchange to share keys
  - sets up secure tunnel to negotiate IKE phase 2 parameters
- **IKE Phase 2**
  negotiates IPsec SA parameters protected by existing IKE SA
  establishes IPsec security associations (tunnels)
  periodically renegotiates IPsec SAs to ensure security
  optionally performs additional DH exchange if perfect forward security is desired
- **IPsec tunnel usage**
- **IPsec tunnel tear-down**

IPsec modes

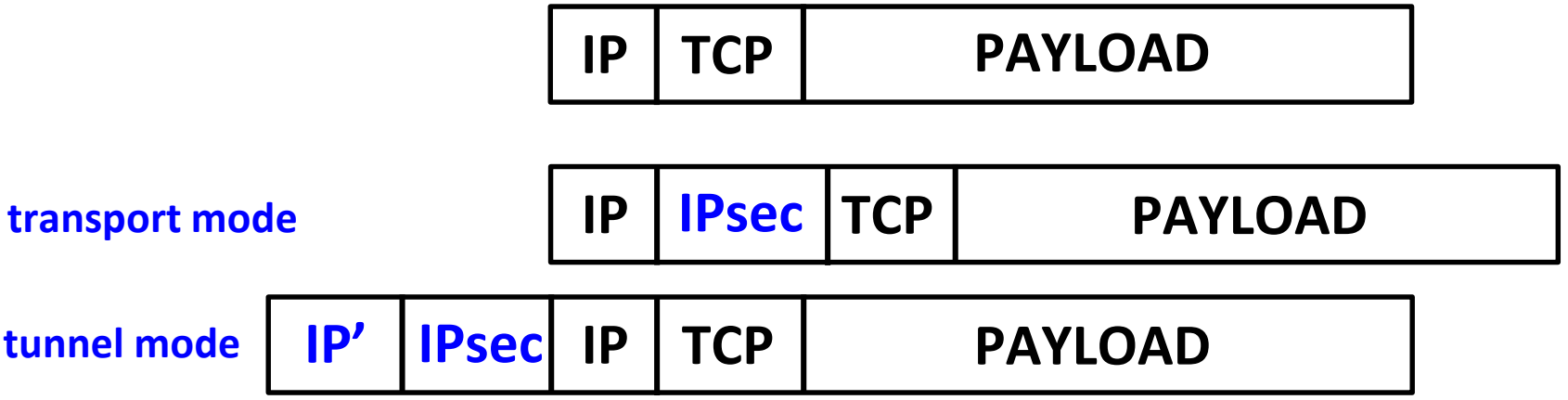
IPsec has two different modes of operation

Transport Mode

- IPsec header is inserted into an existing packet (between IP and TCP headers)
- packet routing is unaffected, but NAT traversal may be (new protocol numbers)
- adds security functionalities to existing flow
- if encrypting, TCP (or UDP) header is encrypted

Tunnel Mode (between Security Gateways - SEGs)

- entire IP packet is encapsulated (and optionally encrypted)
- can be used to create an *IPsec VPN*



AH and ESP

IPsec has two different formats

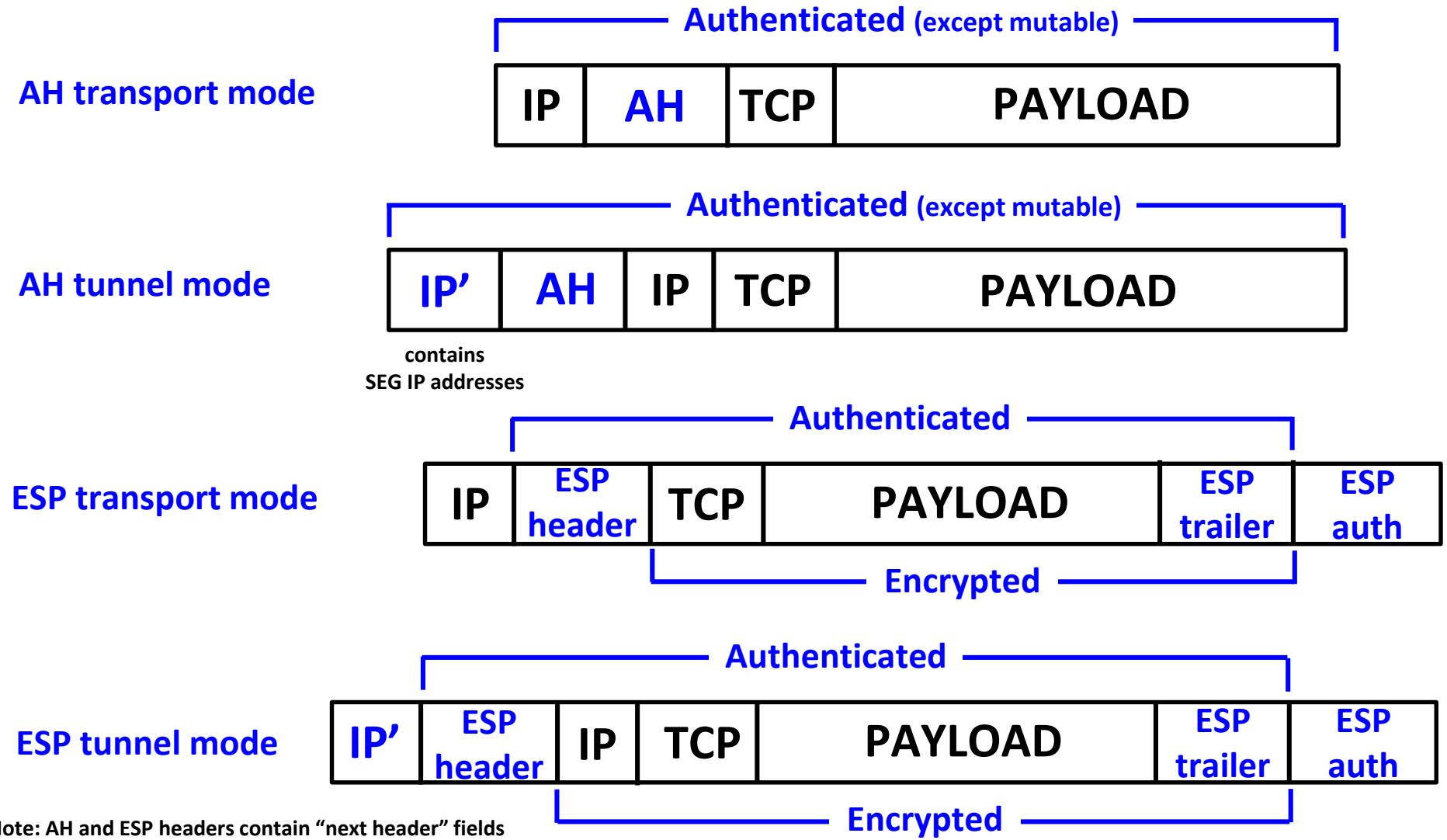
Authentication Header (AH) – no longer recommended

- supports source authentication and data integrity only
- protection
  - in transport mode protects payload and header fields except mutable (e.g., TTL, DSCP, header checksum, offset, flags)
  - in tunnel mode protects entire packet
- uses protocol number 51

Encapsulating Security Payload (ESP)

- supports source authentication, data integrity, and encryption
  - similar to AH if null encryption is used
- adds *trailer(s)* in addition to IPsec *header*
- protection
  - in transport mode doesn't protect IP header (but does TCP header)
  - in tunnel mode entire packet is protected
- uses IP protocol 50

IPsec packet formats



Note: AH and ESP headers contain "next header" fields

- for tunnel mode these are set to 4 (IPv4) or 41 (IPv6)
- for transport mode these are set to 6 (TCP) or 17 (UDP)

RFC 8221

RFC 8221 is the most recent IPsec implementation document
updating all previous specifications

It specifies:

- manual shared key entry SHOULD NOT be used
  and if used only AES-CBC may be used for encryption
- unauthenticated encryption MUST NOT be used
  always use ESP mode with a combined algorithm (e.g., AES-GCM)
  or ESP mode both encryption and authentication algorithms
- IPsec AH is no longer recommended for use

The RFC also specifies which algorithms must be *supported* for interop purposes
(but not necessarily used)

These specifications are updated as compared to the previous RFC 7321

Some algorithms are allowed only for IoT

RFC 8221 algorithms

Compliant implementations implement specific algorithms as follows

Encryption algorithms

DES_IV64	MUST NOT
DES	MUST NOT
3DES	SHOULD NOT
BLOWFISH	MUST NOT
3IDEA	MUST NOT
DES_IV32	MUST NOT
NULL	MUST
AES_CBC	MUST
AES_CCM_8	SHOULD IoT
AES_GCM_16	MUST
CHACHA20_POLY1305	SHOULD

Authentication algorithms

NONE	MUST NOT
HMAC_MD5_96	MUST NOT
HMAC_SHA1_96	MUST
DES_MAC	MUST NOT
KPDK_MD5	MUST NOT
AES_XCBC_96	MAY
AES_128_GMAC	MAY
AES_256_GMAC	MAY
HMAC_SHA2_256_128	MUST
HMAC_SHA2_512_256	SHOULD

Suite B

RFC 6379 and 6380 detail a suite of IPsec algorithms

This suite mandates AES-GCM with 128 or 256 bit keys

- GCM-128
  - IPsec with AES-GCM-128 including encryption
  - IKE with AES-CBC-128, HMAC-SHA-256 and DH group 19 (256 bit ECP)
- GCM-256
  - IPsec with AES-GCM-256 including encryption
  - IKE with AES-CBC-256, HMAC-SHA-384 and DH group 20 (384 bit ECP)

- GMAC-128
  - IPsec with AES-GCM-128 without encryption
  - IKE with AES-CBC-128, HMAC-SHA-256 and DH group 19 (256 bit ECP)
- GMAC-256
  - IPsec with AES-GCM-256 without encryption
  - IKE with AES-CBC-256, HMAC-SHA-384 and DH group 20 (384 bit ECP)

MACsec

802.1AE MACsec was approved in June 2006

based on well known AES-128 encryption

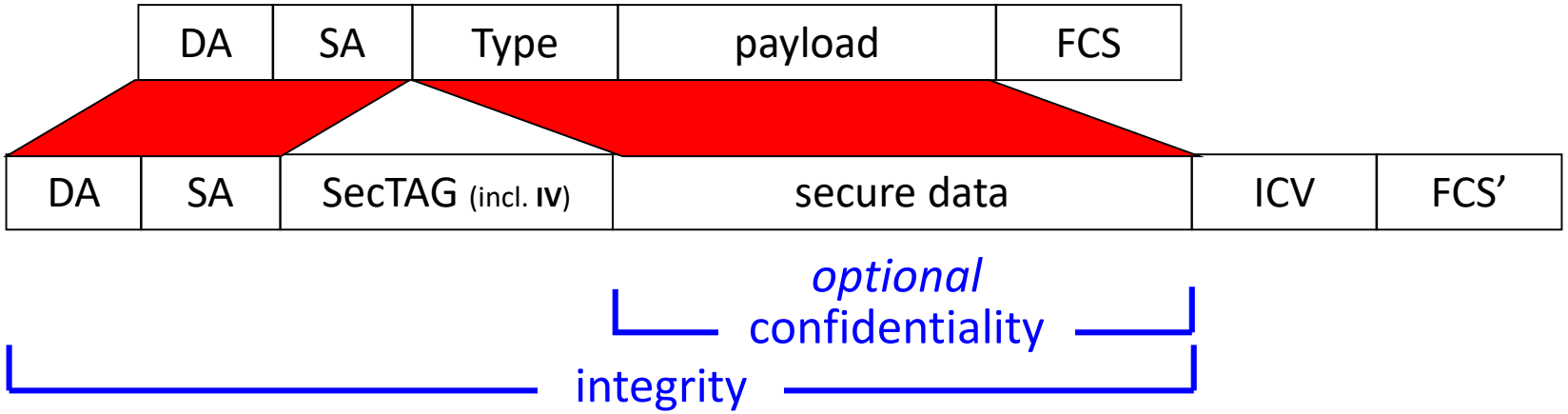
but with a new mode - **G**alois **C**ounter **M**ode

AES/GCM uses a single combined algorithm for integrity and encryption

MACsec

- works over Connectionless network by forming secure associations
- integrated into Ethernet frame format
- provides
  - source authentication
  - confidentiality
  - connectionless data integrity
  - replay protection
  - limited blocking of DoS attacks
- may degrade some QoS attributes (e.g. introduces bounded delay)

MACsec format



SecTAG contains

- MACsec Ethertype (88E5)
- 4B Packet Number (sequence number)
- 8B Secure Channel Identifier

} 12 B **I**nitialization **V**ector

ICV is a 16B **I**ntegrity **C**heck **V**alue

TLS (and SSL)

Sometimes it is better to provide security at a layer higher than L3

For these cases there is **T**ransport **L**ayer **S**ecurity (and previously versions of SSL)

Heartbleed bug in OpenSSL scared the whole world in April 2014

TLS

- adds integrity and encryption to client/server applications
  - https, smtp/pop/imap, sips, mysql, EAP-TLS…
- is based on SSL (originally developed by Netscape)
- is widely used to secure web browsing (https), email, ecommerce, etc.
- client picks a random number, encrypts with server's public key and sends to server
  client and server can now communicate using symmetric encryption and MACs

Compared to IPsec, TLS

- can be implemented in a browser (e.g., for online banking or commerce)
  and doesn't require/trust OS kernel support like IPsec VPNs
- only runs over connection oriented TCP (but there is DTLS is for UDP)
- can authenticate individual users, rather than IP source addresses
- is better than IPsec at NAT traversal