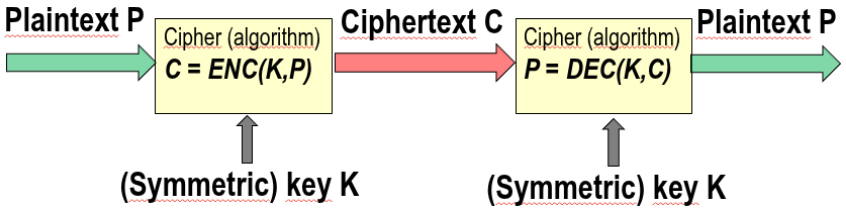
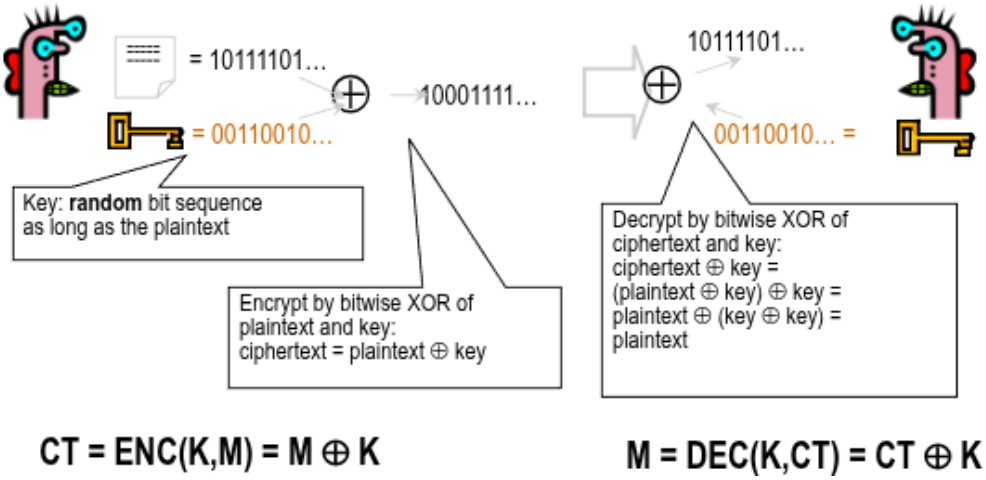
Encryption
Goal: protect data confidentiality by TRANSFORMING data into something incomprehensible, the transformation must be reversible.
Symmetric encryption means that we use the same key for encryption and decryption.
Enc/Dec algorithms should be PUBLICLY know the only secret should be the key.



Substitution Chiper is a chiper (bad chiper) where for every letter u replace with another letter, it's easy to decipher thanks to a frequency analysis that shows how many times a vowel or consonant appears or some repeating patterns.

If we want the best encryption we should use the One-time-pad (Vernam cipher)



This scheme is unconditionally secure but only if:
- Use as many keys as msg,
- Keys must be as long as the plaintext
- Keys are random
But there is no integrity because everyone can change the msg, it's insecure if keys are reused and must use random keys and not pseudorandom: PRNG (Pseudo Random Number Generator) can be good/bad in three ways: Statistical properties of the output, Predictability and Periodicity; instead TRNG (Truly Random Number Generator) randomness extracted from a physical phenomenon has noise but it's not reproducible. So the Vernam Cipher is not realistic.

Security must be defined against an attacker.
(A cipher may be secure against an attacker able to access only the ciphertext but might be trivially broken by an attacker able to see few plaintext-cipher pairs).

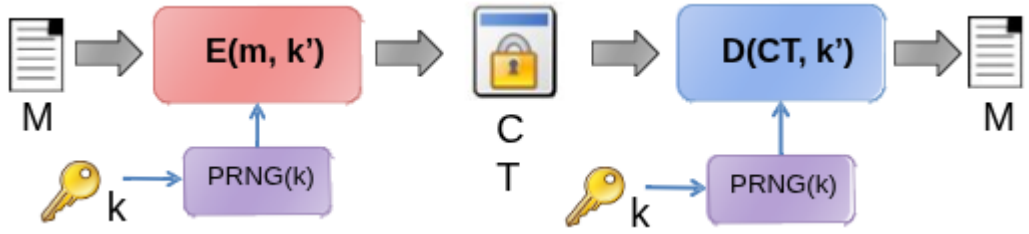
Baseline requirement for any good cipher: IND-CPA (semantic Security).
INDistinguishability under Chosen Plaintext Attack
Adversary must not be able to compute any information about a plaintext from its ciphertext, even if it has access to an encryption oracle: given two plaintexts of equal length and given a ciphertext which contains a randomly chosen msg among these two, should NOT be able to distinguish which one it is.

IND-CPA consequence 1: encryption MUST be randomized
A same msg must always encrypt to a different ciphertext, and the ciphertext must be indistinguishable from random;
IND-CPA consequence 2: if a substring repeats in the msg, in the ciphertext we shouldn't have repeated substrings.

Semantic security != Perfect security
Perfect Security: Even with unlimited computational power, the ciphertext reveals no information about the plaintext. (One-Time Pad)
Semantic Security: Means that no efficient (real-world) computer can figure out anything useful about the msg from the ciphertext. But if someone had infinite power, they could break it, it's just computationally hard to do so.

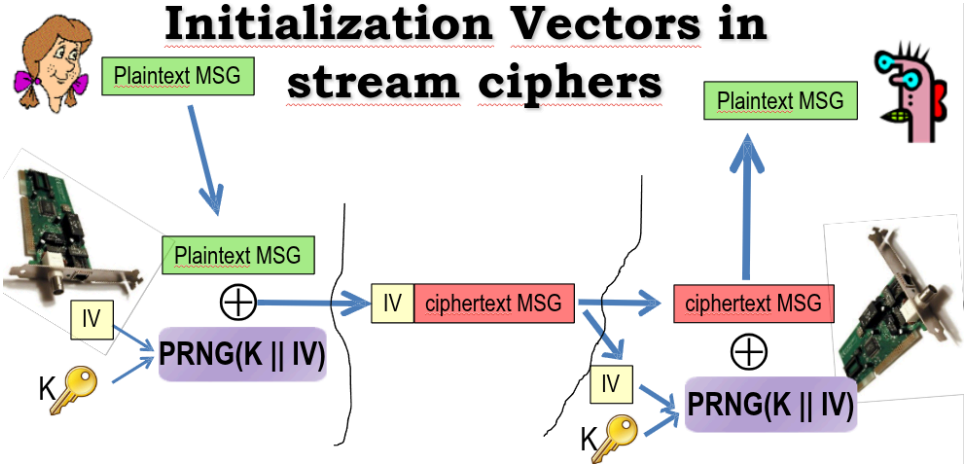
We have two major categories: STREAM and BLOCK ciphers, plus there are BLOCK ciphers used in STREAM mode.

Stream ciphers
The goal is to approximate a one-time-pad



In the one-time-pad the key must be as long as the msg, so we take a random key k (not as long as the plaintext) and expand it with a PRNG (it becomes a keystream != key) and ENC the msg.
The one-time-pad has a random key while the stream cipher has a random keystream.
This way if there is a substring in the key this dosen't repeat in the ciphertext, but if we ENC the same msg twice the ciphertext will be the same, that's because PRNG is deterministic.

Initialization Vectors IV



The plaintext is send to the wifi card that creates a Truly Random Key, the IV, then uses a PRNG (with the IV as key) at every new msg and send the IV in clear along the ciphertext. The receiver now gets the ciphertext and the IV than uses the same PRNG to reconstruct the keystream and extract the plaintext.
This is semantically secure but only if the IV will never repeat.

WEP
Good cipher is not enough u must make good use of the cipher and must design good protocols.
WEP goals are:
- Authentication (prove that is the actual one who sent the msg)
- Integrity (nobody can modify my message)
- Confidentiality (not let other see the msg transmitted)
WEP did 3/3 wrong, assuming that it uses a perfect cipher (it used RC4 a specific PRNG, today weak) u can also broke it.

The packets dosen't arrive in order and the Stream must be synchronized, so the WEP solution is to send and change IV per each frame so the receiver can independently decrypt each frame, also WEP transmitted in clear and that's not a problem in a good stream cipher.

If we assume RC4 strong WEP would be secure as long as IV never repeats, but same keystream iff same (Key,IV), so if the IV repeats the weakness is PRACTICAL.

Two blunders:
- IV size is 24 bits so 2^24 possible combinations = 16 millions frames. Assuming 1500 bytes frames at 7mbps wifi throughput in 8 hours u go through all the possible combinations.
With the birthday paradox we have a 50% collision probability just after 2^12 = 4000 frames.
- The standard doesn't say how to generate the IV so it's left to the implementation.

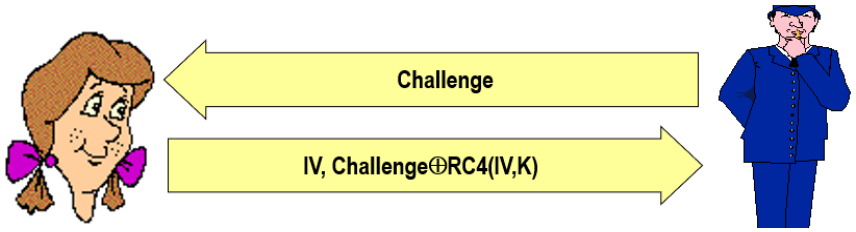
A passive attack can be to create a dictionary and use known message to recover the keystream and then try all the possible IVs (u must send the IV, and the msg with the key (RC4(IV,key)).

$(MSG \oplus RC4(IV,key)) \oplus MSG \rightarrow RC4(IV,key)$

Increasing the WEP key size is irrelevant cause the IV will still be 24 bits.

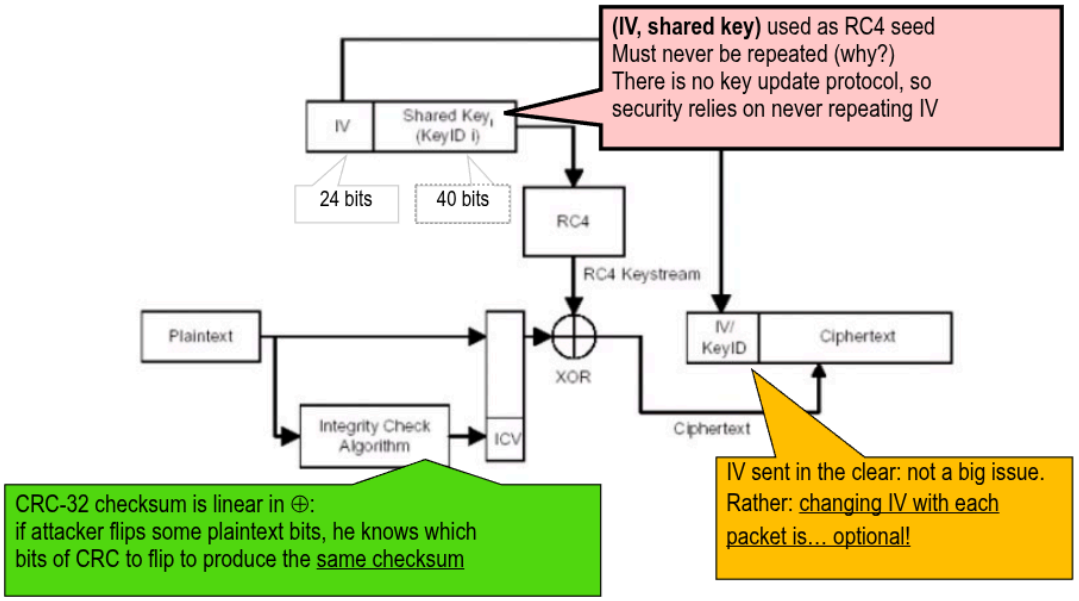
This get even worse cause there were attacks to RC4. One found some weak IVs leaks key information into keystream, and with 5% probability a byte in the keystream will be equal to the one in the key, another one focus on the keystream byte cause the first byte of each plaintext packet was 0xAA (thanks to the LLC encapsulation) and this information was a weakness that accelerated the attacks made on it.

WEP authentication
Great network access only to users who know a pre-shared secret. A challenge-handshake is used to prove that the secret is know, to prove knowledge of K by showing ability to encrypt the challenge. WEP helps the attacker cause along the encrypted challenge with the keystream it sends also the IV.



So it uses the same key as frame encryption, the challenge is palintext so it's known. It’s easy to send multiple challenges.

Also there is no need to know the key to enter the network cause just one [IV, RC4(IV,K)] pair and its game over!

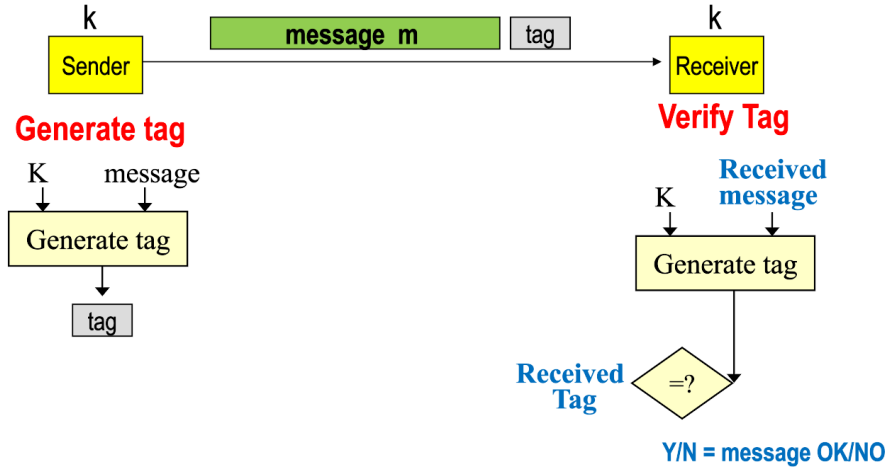


So either confidentiality, authentication are wrong and the integrity also cause they used a CRC32 as integrity check value thinking that the external encryption wrapping should prevent attacker to perform valid alteration, but it is linear in xor so the attacker can modify the message.

Integrity check must be NON LINEAR and must explicitly include a key, external encryption may not provide ANY guarantee.

Confidentiality != Integrity
- Confidentiality: Only the legitimate destination should be able to see the content of a message.
- Integrity: Nobody should be able to modify a message while in transmission, only the legitimate source should be able to forge a message.
Encryption does NOT guarantee any integrity! (only the AEAD ciphers does guarantee both of them).

Message authentication with symmetric key



We have a key K known by both sender and receiver (authentication and encryption use different keys).
Using the msg and the key K the tag is created, then is sent both the msg and the tag to the receiver.
The receiver now can verify the tag by recreating it with the same message and the same key K, if the tag created corresponds to the one sent thn the message was the original.
The tag is NOT reversible.

Message authentication does protect from:
- MITM attacker: u cant change the tag without knowing the key and u cant extract the key from the tag.
- Spoofing: Cannot compute the tag without knowing the key.

Message authentication doesn’t protect from:
- Replay attacks: The sender sends a msg (along with the tag) (like a payment of 1000$), that message can be resend to the receiver (and this way make a payment of $2000).

We can prevent replay attacks by applying a nonce to each packet.
Nonce (Number used ONCE) = fresh information, guaranteed to differ in each message, doesn't need to remain secret so can be added in clear, must be included in the TAG computation.
Can be a sequence number but u need to manage reboots, random number but u need to control that the packet is a fresh one, timestamp but u need to guarantee when now is really now.

MAC
- Digital signature (DS): Nobody can modify a digitally signed message except the creator.
- Message Authentication Code (MAC): Nobody except TX and RX (which have the key) can modify a MAC-authenticated message.
Both have an identical purposes, message/data integrity!
Unlike MAC, DS further guarantees non repudiation so it's a stronger authentication than MAC, useful in legal scenarios.

Defining Security for MAC
Security: Attacker should not be able to create or modify a message so it shouldn't be able to extract key from pair (msg, tag(k, msg)).
From weaker to stronger: Know Message Attack, Chosen Message Attack, Adaptively Chosen Message Attack.
Even if the attacker is given a number of past msg/tag pairs he must be unable to forge a tag, more specifically the probability to forge a valid pair must be NEGLIGIBLE.

Hash function
Transforming any length message x in a fixed size digest Y=H(x) should be relatively easy to compute for any given x.

Cryptographic hash function properties
- Preimage resistance (one way): Given Y = result of a hash, it is hard to find ANY X such as H(X) = Y. You shouldn't be able to find ANY of such msg (in reasonable time).
- Second Preimage resistance: Given X, it's hard to find another X' such that H(X) = H(X').
- Collision resistance: It is hard to find two generic X1 and X2 such that H(X1) = H(X2).

Hash functions should also resist to brute-force so the digest cannot be small and must be large. A large digest is not enough tho (SHA1 of 160bits was broken with 2^61 attempts).

Birthday Paradox

1. **What is the probability that none of you N=22 is born in my same day?**

$$\left(1-\frac{1}{365}\right)^{22}=\left(\frac{364}{365}\right)^{22}=\cdots \textbf{ 94.1\%}$$

= you have my same birthday in less than 6% cases

2. **What is the probability that no two+ of us N=23 are born the same day?**

$$1\cdot\left(1-\frac{1}{365}\right)\cdot\left(1-\frac{2}{365}\right)\cdots\left(1-\frac{22}{365}\right)=$$

$$=\frac{365\cdot(365-1)\cdots(365-22)}{365^{23}}=$$

$$=\frac{365!\ /\ (365-23)!}{365^{23}}=\cdots \quad \textbf{49.3\%}$$

In more than 50% cases two of us have the same birthday!!! Not so obious...

Birthday Paradox math

n bit digest N=2^n **K = n. messages** **p = collision probability**

$$P(no\ collision) = 1 - p = \frac{N!/(N-K)!}{N^K} =$$

$$= \frac{N}{N} \cdot \frac{N-1}{N} \cdot \frac{N-2}{N} \cdot \cdots \cdot \frac{N-(K-1)}{N} =$$

$$= 1 \cdot \left(1 - \frac{1}{N}\right) \cdot \left(1 - \frac{2}{N}\right) \cdot \cdots \cdot \left(1 - \frac{K-1}{N}\right) =$$

Remember: for x small, 1-x approx e^(-x)

$$= \prod_{i=1}^{K-1} \left(1 - \frac{i}{N}\right) \approx \prod_{i=1}^{K-1} e^{-i/N} = e^{-\sum_{i=1}^{K-1} i / N}$$

$$= e^{-\frac{K(K-1)/2}{N}} \approx e^{-\frac{K^2}{2N}}$$

(no need to learn all the process just the result e)

So, we concluded that: $1 - p \approx e^{-\frac{K^2}{2N}}$

Let's solve in K: $\log(1-p) \approx -\frac{K^2}{2N} \Rightarrow K = \sqrt{2N} \cdot \sqrt{\log \frac{1}{1-p}}$

50% (p=1/2) collision probability: $K \approx \sqrt{2} \cdot \sqrt{\log 2} \sqrt{N} \approx 1.177\sqrt{N}$

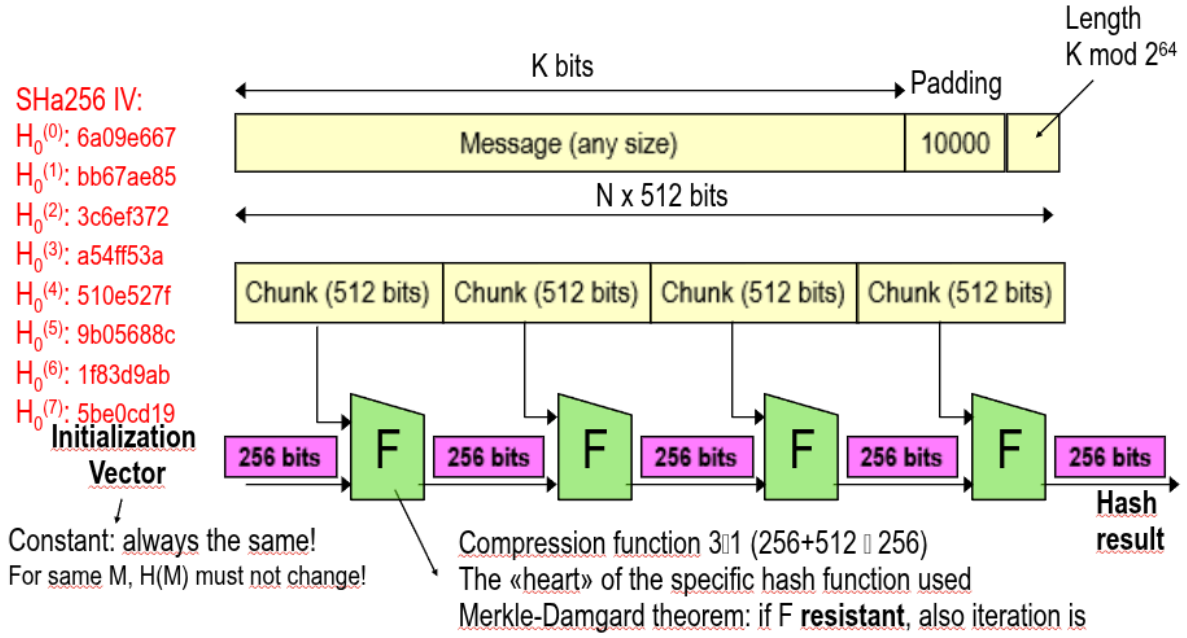
And since N=2^n, using n bits: $K \approx 1.177\sqrt{2^n} \approx 2^{n/2}$

MD5 (128bit) 50% collision after 2^64 msg (much worse was broken via cryptoanalysis since 2005)
SHA256 (256bit) 50% collision after 2^128 msg, (1 billion centuries of work so its ok today)

Message Authentication Code using Hash Functions
1: Good Hash: SHA-2 family
2: Include secret in hash: secret is authentication key, the problem is where to put the auth key (beginning or end?)

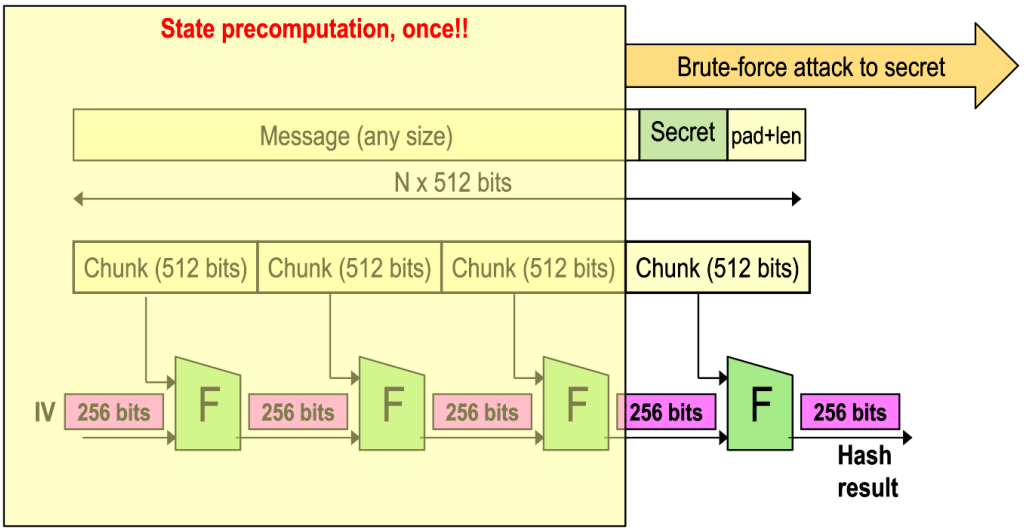
If Hash Function were prefect (random oracle) that given any distinct input X, H(X) will be a truly random value but with repeatability property in case we put the same input X. But no practical hash function can be a random oracle.

Hash construction:
Merkle-Damgard construction



F resistant means that he has all the 3 properties as the hash function. The IVs, in this case are constants not the usual IVs.
Now we need to see where to put the secret, if at the start or in the end.
The length at the end is necessary cause otherwise there are attacks that can exploit this.

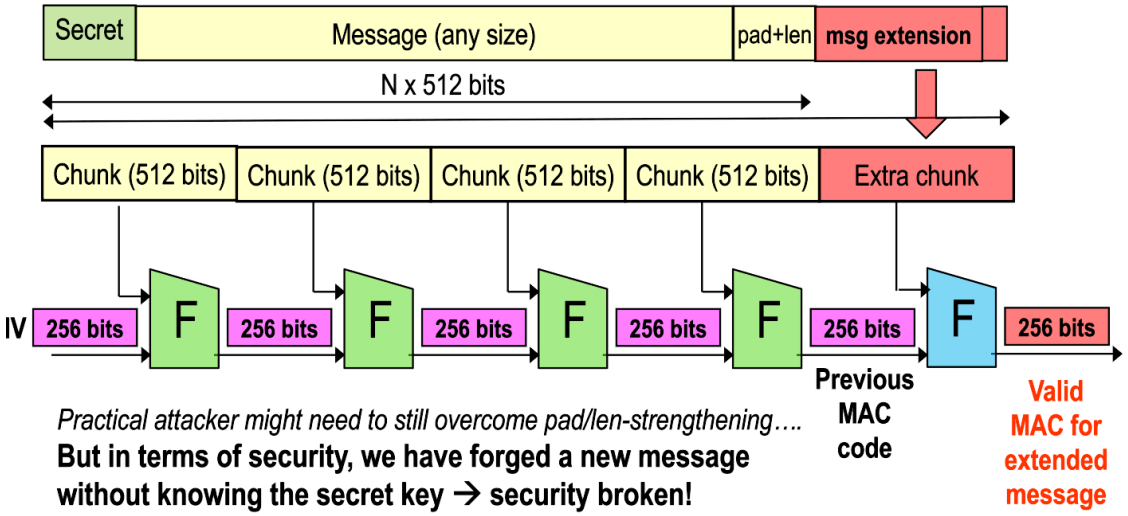
Let's now see if we add a secret to the message, is it secure?
Assume we put it at the end between the msg and the pad+len.



Weakened security: from N compression blocks per trial, to just 1!
Even worse: collision on msg → collision ALSO on MAC!

We can precompute all the F until the lust chunk and then we can bruteforce the last part.
Now assume that we have M_1 and M_2 such that $H(M_1) = H(M_2)$ to find the secret i just need to analyze the differences in the iterations between those two, when we found different values we know that there is the secret key.

Assume we put it at the start of the msg. This is now vulnerable to length-extension attacks so we add a msg extension and forge a valid MAC without knowing the secret key. So we now have a forged message with a valid tag (the hashed) without ever knowing the key.

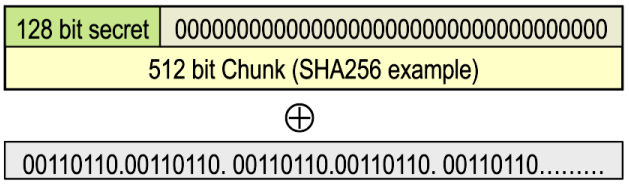


Practical attacker might need to still overcome pad/len-strengthening….
But in terms of security, we have forged a new message without knowing the secret key → security broken!

HMAC
If u need to create an auth tag use a HMAC, standardized in 1996, it has more or less the same performance of the hash and has a provably secure construction:

$$\mathbf{HMAC_K(M) = H(K^+ \oplus opad \,||\, H(K^+ \oplus ipad \,||\, M))}$$

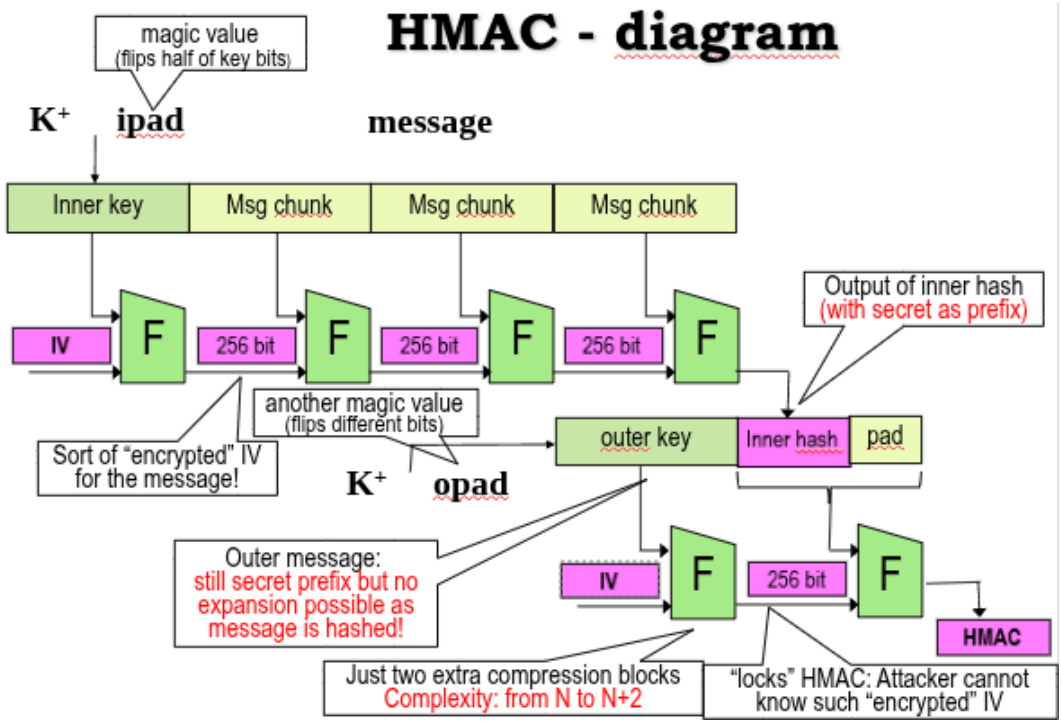
→K^+ = shared key, but "extended" to block size
» E.g. SHA256 → padded with zeros up to 512 bits
» *Makes sure secret uses the entire first hash chunk*



→opad = 0x36 = 00110110 repeated as needed
→ipad = 0x5C = 01011100 repeated as needed
» *We would need TWO different (inner and outer) secrets, but since we have only one, let's "pseudo-randomly" derive inner and outer keys from the single key*

(roughly) equivalent to NMAC construction: $H(K_o \,||\, H(K_i \,||\, M))$

In the formula we can see that its computed first the tag internally with the message M, the result is then used as a tag and computed externally.
K^+ is the expanded secret (secret padded with zeros so it goes to 512) and it's used opad and ipad which hamming distance is about 50%, obtaining the inner and outer key.



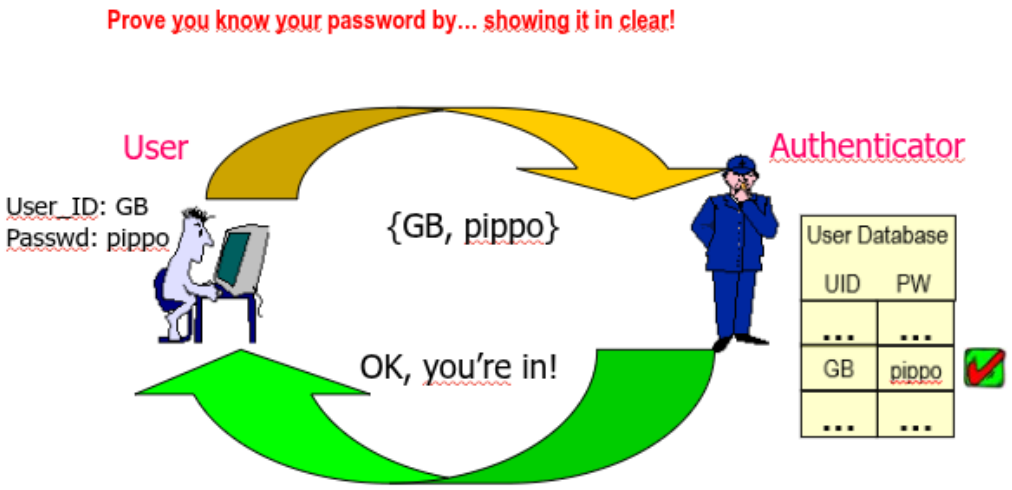
(xor negli spazi bianchi)
Its done the hash of the inner key with the constant IV that gives a pseudorandom value that goes in input in the next block where starts the hash of the msg chunks. After obtaining the inner hash u use also the outer key as prefix to hash and obtain the HMAC.
The complexity goes from N to N+2 cause the second hash only uses two compression blocks.

User Authentication
Prove you are really the one u claim to be. Not to confuse with identification where u show your digital identity (email, ID) and not to confuse with authentication where u prove your digital identity is controlled by u (u know a secret password).

Authentication means something u and only u know (password), something u can only u have (smart card), something u are (fingerprint), something u do (dynamic biometrics like handwriting).

To prove u know a password doesn't necessarily imply revealing it (even if that's one approach). There are different techniques that varies the disclosure of the information:
- ZKP = no information leakage.

- PAP = full disclosure



- CHAP = some information leaks

- **Secret passwd never transmitted in clear - Hash is the usual approach, but it is NOT the only one**

(example: is sent the challenge 1234, the user hash it with the password and generate the hash H, the authenticator will do the same thing using the same challenge and pass in its database and confront the results. The challenge must be always fresh and not a used one)

Password and secret conceptually are the same but practically there is a HUGE difference: the secret is a random string of N bit so the probability of guess it is 1 in 2^N,
while the password is a low-entropy string the probability to guess is much lower than that.

PAP is not better than CHAP because that depends on the attacker's model
Attack to the communication channel: if u want to use PAP u must protect communication channel.
Attack to backend UID-passwd database: attacker penetrates system and steals entire passwd DB. A mitigation can be to hash the passwd database.
Other attacks can be insiders, key loggers, …

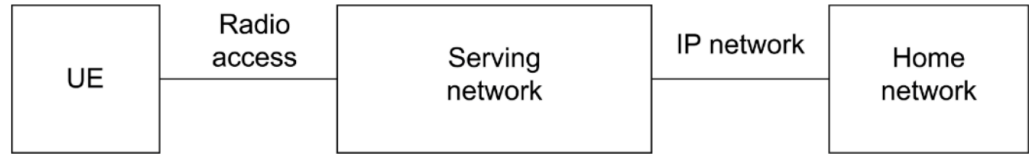
Hashed passwd database in PAP
Store the H(passwd) in the database so if an attacker steals the DB it has to invert the passwd, must choose a good hash (in order to be one-way). Actually is used a salted hash, same as hashed passwd but the hashed part is the salt+passwd not only the passwd. This is done for a few reasons, one of them is that there are some passwd that are commonly used like 123456789, doing the hash of this one and checking with the DB can lead to a match, but with the salted hash this doesn't happen.

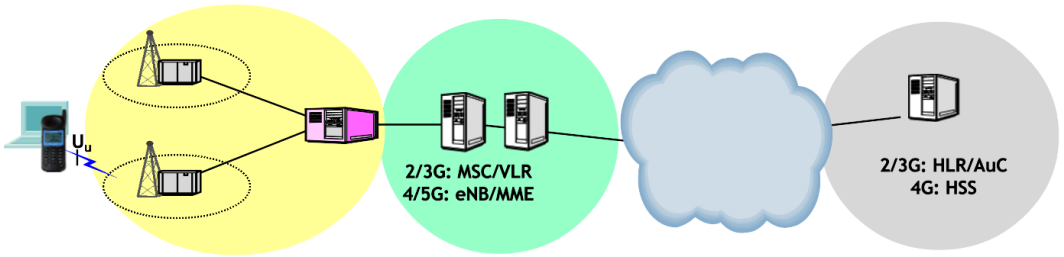
In security there is NO best algorithm if attacker is backend then PAP is better otherwise CHAP can be better. There isn't a single solution, but depends on which attack u aim at defending.

CHAP mitigation can be done by sending the salt along with the challenge and then the user sends H(H(salt, pass), challenge) so attacker cannot anymore reuse stolen passwd.

The "best" hash today is SHA256 but its not the best one to use in this case cause its designed to be fast, collision resistance and pseudorandomness. We want to use a hash that is slow cause faster computation leads to faster cracking. Especially cause SHA256 is used in the mining of bitcoin so there are ultra-high performance HW for SHA256 computation. So its best to use slow hash, including also salt and cost parameters.

(DON'T STUDY until X)
Architecture of wireless cellular system

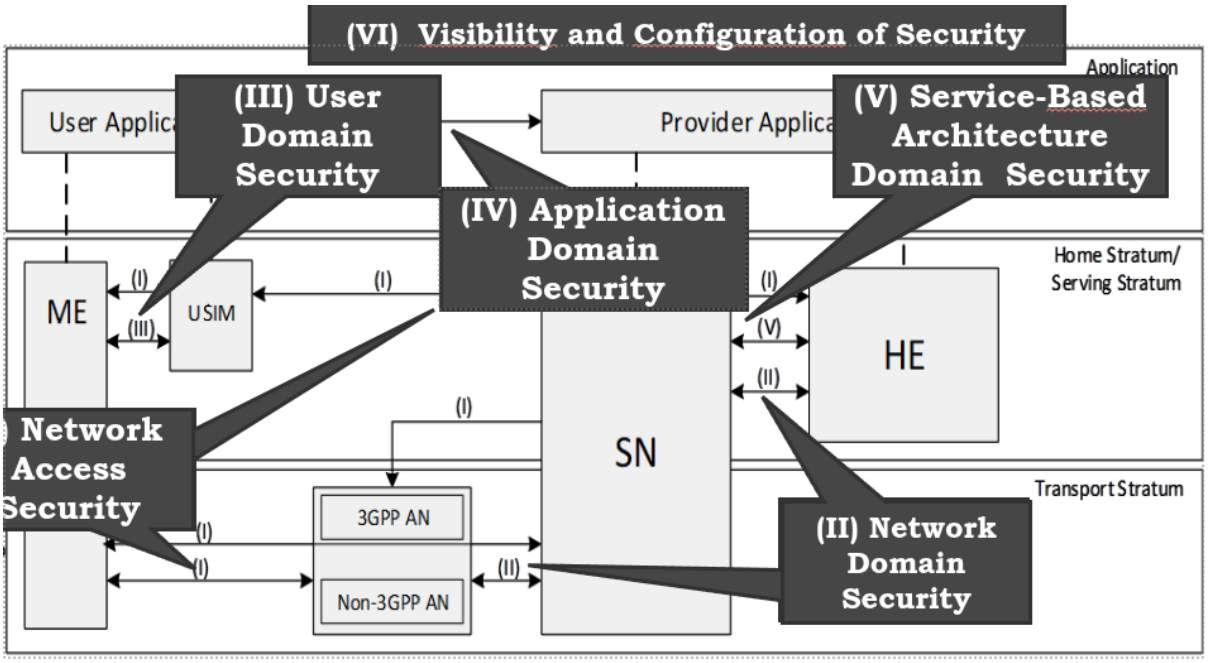




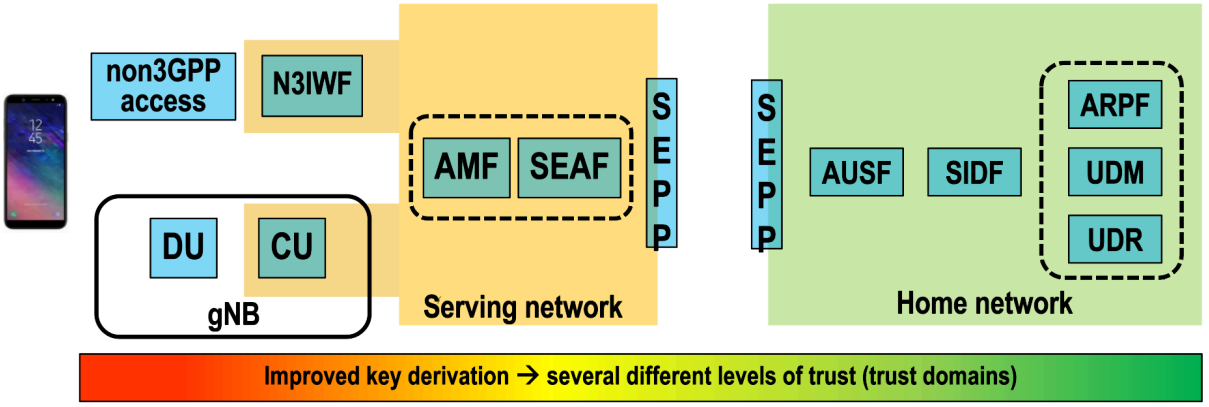
There is a sort of hierarchic structure where u go from your home network to the external network.

Security in 1G: there wasn't, anybody could intercept the conversation with anyone else.
Security in 2G: they invented a crypto system that wasn't public (COMP-128) so security by obscurity. They didn't support mutual authentication and no core security network.
Security in 3G: Started to use good ciphers (public scrutiny) and to support mutual authentication, they also added core network security.
Security in 4G and 5G: (I) They added security in the Access of the Network, so the communication between the device and the radio part. (II) Then they added security, protocols, between the device and the SN (Serving Network). (III) All the security in the device is done by the SIM module, so they also added an interface between that (USIM) and

the mobile equipment (ME), the mobile alone shouldn't have any cryptographic part. (IV) There is also security at the level of application. (V) Specific for 5G, security for the architecture. (VI) Decided how to configure the security of the network.

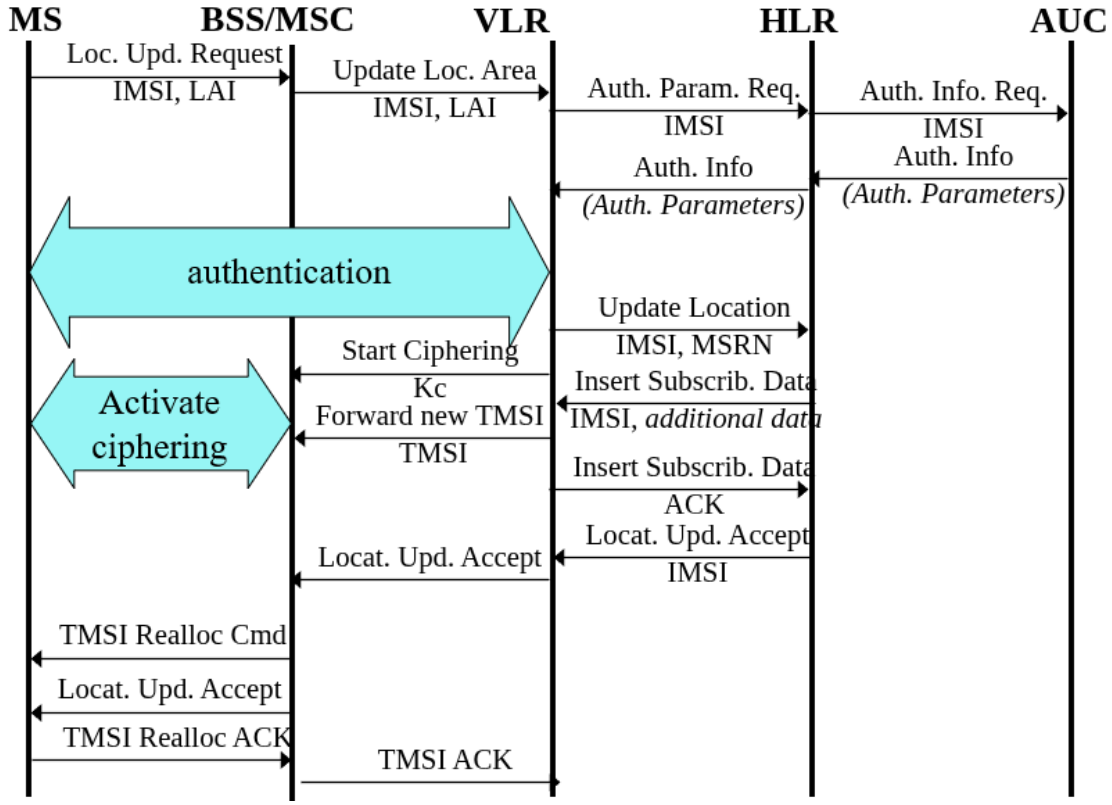


5G Security Architecture - Components



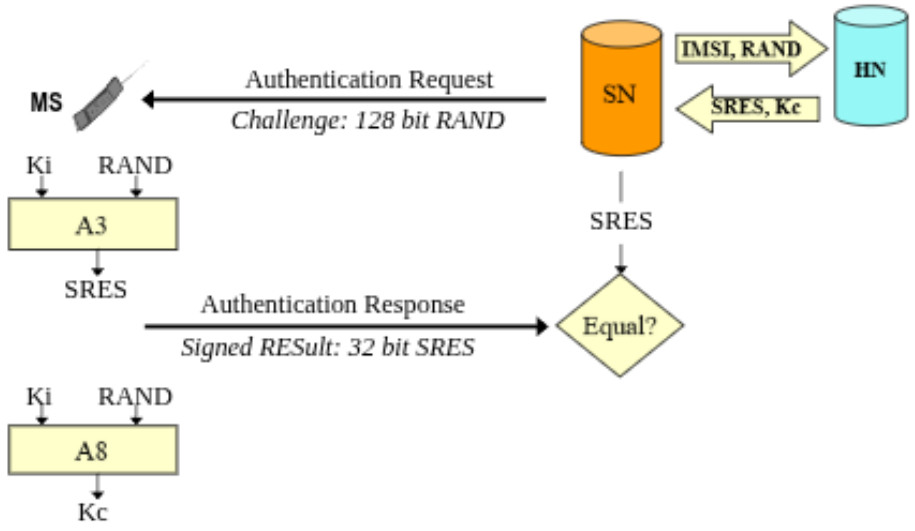
DU	Distributed Unit	AMF	Access Management Function	AUSF	AUthentication Server Function
CU	Central Unit	SEAF	SEcurity Anchor Function	SIDF	Subscription Identifier Deconcealment Fct
				ARPF	Auth credential Repository & Processing Fc
N3IWF	Non 3GPP Inter Working Function	SEPP	Security Edge Protection Proxy	UDM	Unified Data Management
				UDR	Unified Data Repository

Authentication in 2/3G (dont go in details)



First the location update goes from the mobile through the BSS and all the connection arriving to the AUC that sends Auth info so the mobile can auth itself, Then starts the authentication and the encryption.

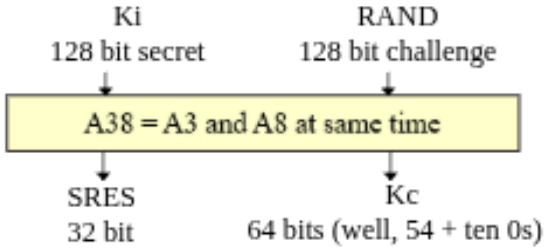
Authentication in 2G



After the MS wants to connect the Service Network sends to the Home Network the IMSI (your identity) and RAND that is use as a challenge to authenticate the user. After the Home Network HN provides the answer SRES that si for sure correct, the SN sends the authentication request to the mobile MS. Now MS need to produce the same SRES in order to connect to the network. Its also used a second function to generate the K_C (enc key)

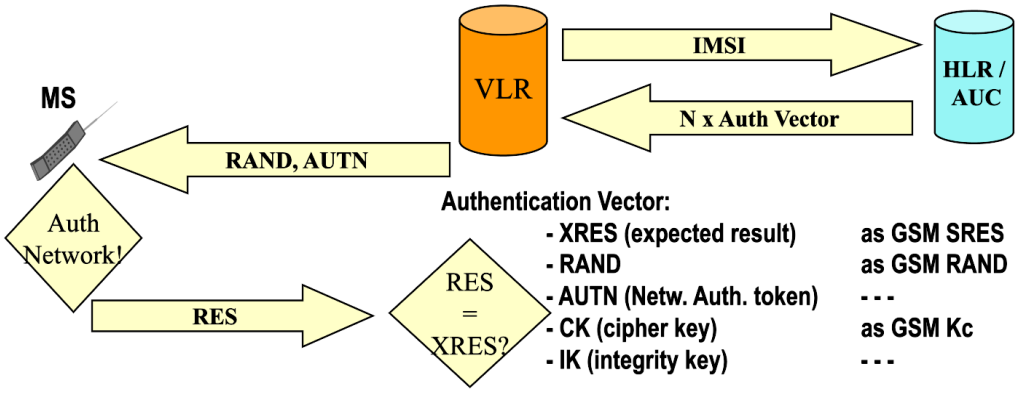
Triplets (Authentication Vector)
If we want to connect multiple mobiles we create some traffic between the SN and the HN asking every single time for the SRES. To resolve this, the SN will send only the IMSI and the HN will deliver N different triplets (RAND, SRES, K_c) to be used for N distinct authentications. This way the security is completely in charge on the home network, cause he generates the challenge that need to be completed, also the SN doesn't need to know the algo used to generate SRES and K_c (A3, A8).

Usually the one that need authentication after using A3 to generate the SRES it right away uses A8 to produce the K_c. An idea is to use A38 = A3 and A8 at the same time, with input K_i and RAND to produce the SRES and K_C together.

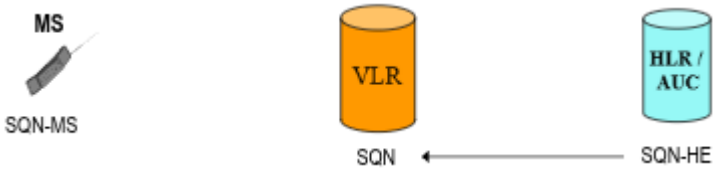


The A3/A8 algorithms used security by obscurity, actually most vendors used COMP128 that was not disclosed but also broken in 98. Security by obscurity in fact is really bad cause noone can see and test those algo, leave the hash design to crypto experts.

3G/4G/5G authentication: AKA (Authentication Key Agreement)
Introduction of mutual authentication, more keys one for integrity and one for encryption and several extra details used, algo first scrutinized by research community and then selected.

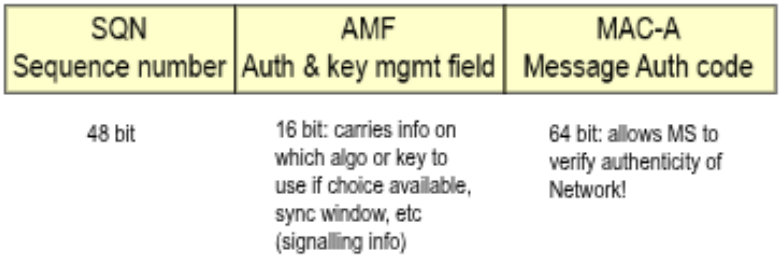


For the Network Authentication a Sequence number is used as implicit nonce.



The current SQN-MS is stored. The SQN is send (included in AUTN) and given by AuC and it's checked that SQN = SQN-MS + 1 (with appropriate tolerance range to cope with lost message or range used for more authentications). Once auth OK, the local SQN is updated.

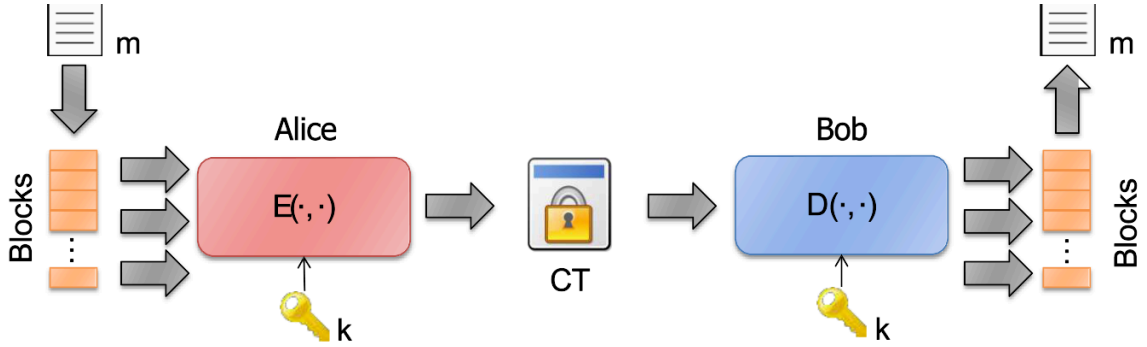
AUTN format



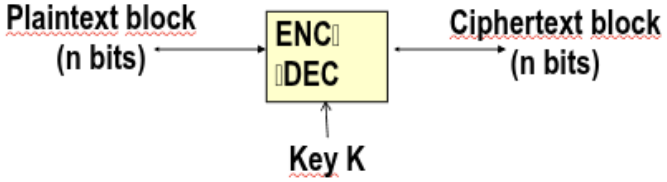
The MS has all info needed to check that MAC-A transmitted by network is the same of MAC-A locally computed, so SQN guarantees defense against replay. But by looking at SQN (stepwise increasing), eavesdropper may discriminate and track user, the solution to this problem is to mask SQN with Anonymity Key.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Block ciphers



The goal is generalize substitution ciphers.

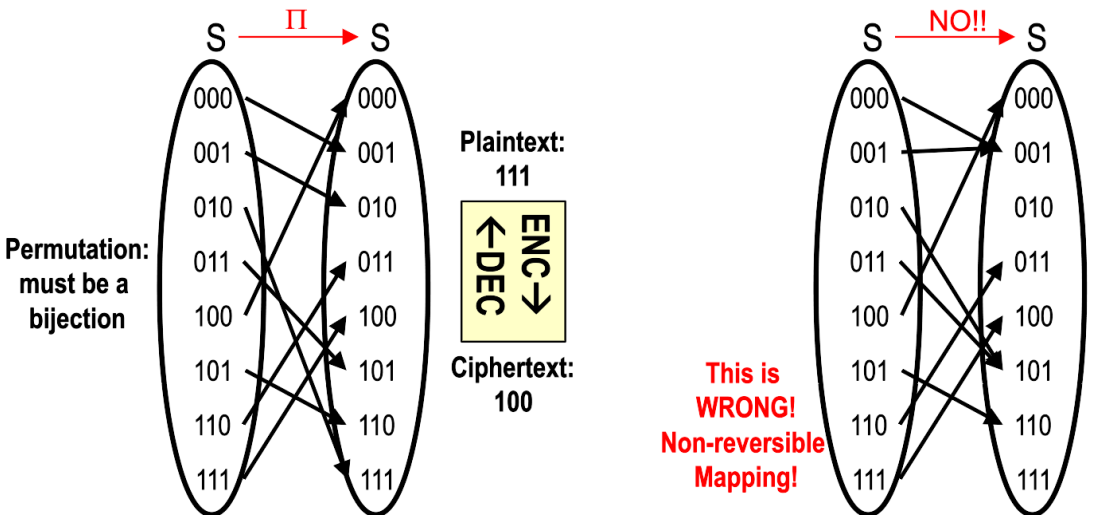


Block algorithm should implement a Pseudorandom permutation but in reality the key permits to select only among $2^{keysize}$ permutations.

Pseudo Random Permutation (PRP):
Be S the set of all possible plaintexts:

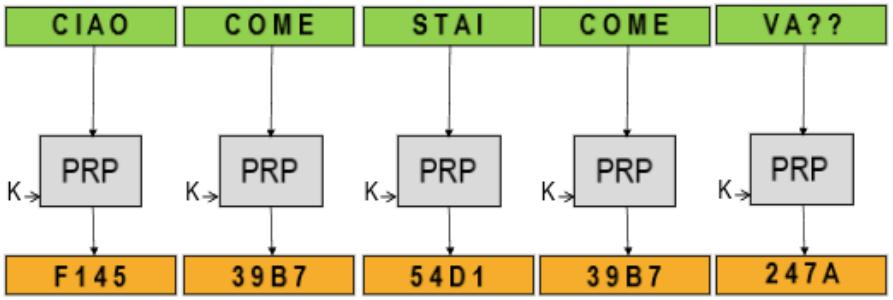
→n = 3 bits, S = {000,001,010,011,100,101,110,111}, |S| = $2^n = 8$

Permutation: bijective function Π: S→S (1-to-1 mapping)



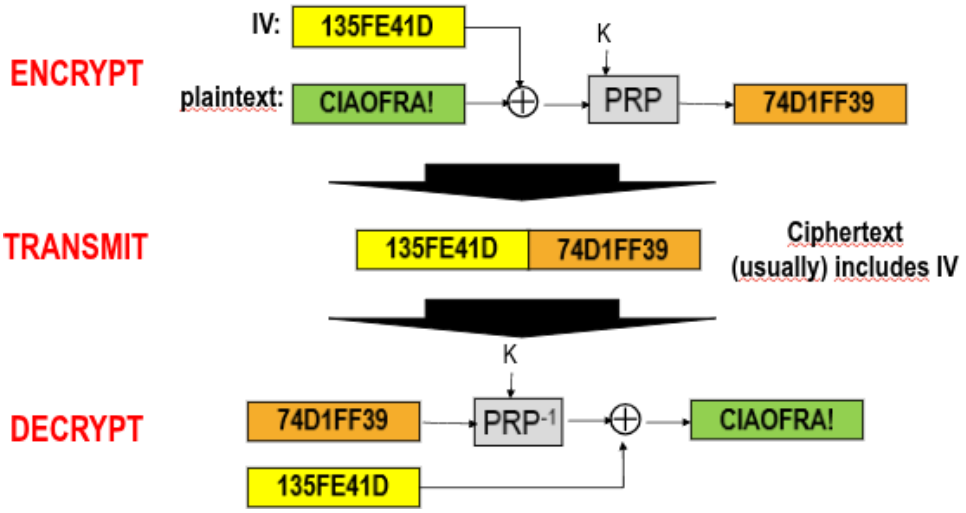
Pseudo-Random means that block cipher should uniformly select one of the possible permutations, and this one is selected by the secret key K. But there are 2^n! (factorial) permutations. In AES n = 128 so 2^{128}! that's like 2^{135}. (unbelievably large number). AES permutations are way less than ideal PRP (but still OK).

If the plaintext is longer than the block size it is divided in block



There is an independent encryption of each block = Electronic Code Book (ECB Mode) means that the same plaintext block implies the same ciphertext block (repeated message) so there isn't semantic security and that leads to trivial cryptanalysis. ECB should almost never be used if not in special occasions.

The best thing to do is to use a fresh initialization vector (IV) for every new encryption.



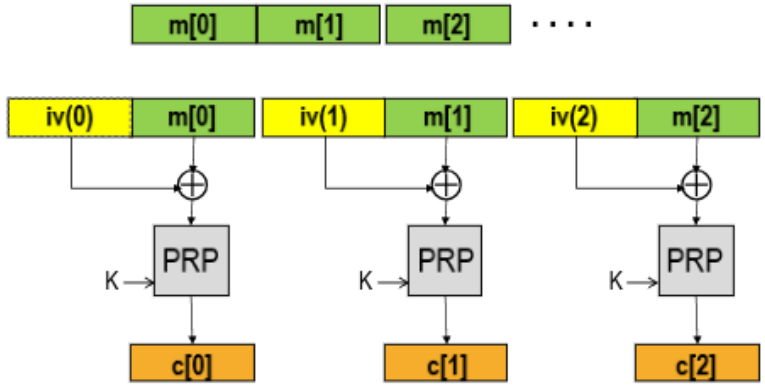
Plain ECB is only OK if the msg is smaller than one block and the msg will never repeat. For repeated msg it's used an IV that must never repeat and must be unpredictable so it has to be a random value with the same size of the block.
For longer messages we need to have a mode of operation: a secure (semantic security, IND-CPA) way to combine blocks, avoiding that a same plaintext block is encrypted twice as the same ciphertext.

The most used modes of operations are:
- Cipher Block Chaining (CBC)
- Counter Mode (CTR)

Nist recommended:
- Cipher Feedback Mode (CFB)
- Output Feedback Mode (OFB)

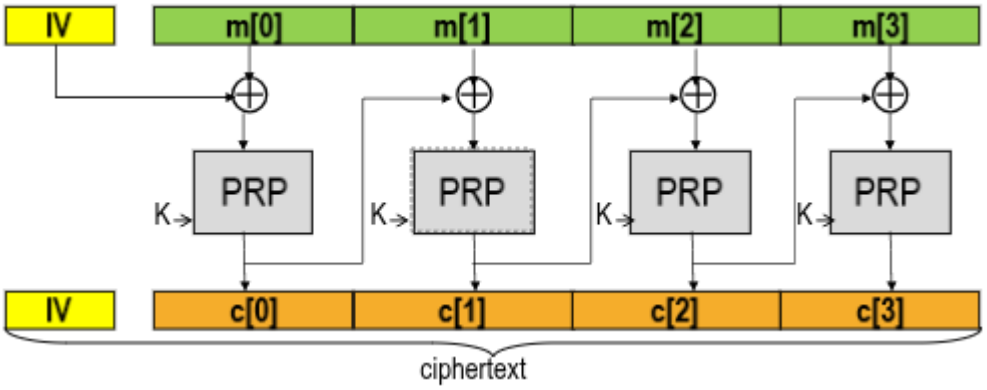
More advanced properties:
- Galois Counter Mode (GCM)
- Offset Codebook Mode (OCB)

Semantic Security can be done by applying a different IV for every block.

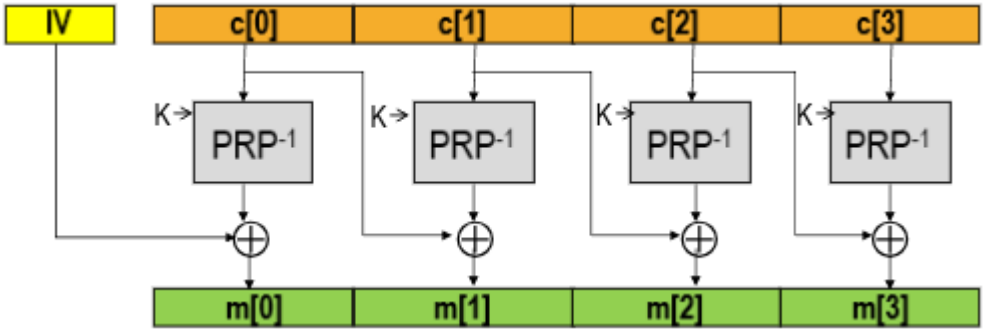


The only problem is that there is a lot of overhead cause of the double size.

Cipher Block Chaining (CBC)
Good PRP means random output means that c[i] looks as random. The CBC idea is to use previous ciphertext block as IV for next block.
CBC Encryption must be done sequentially:

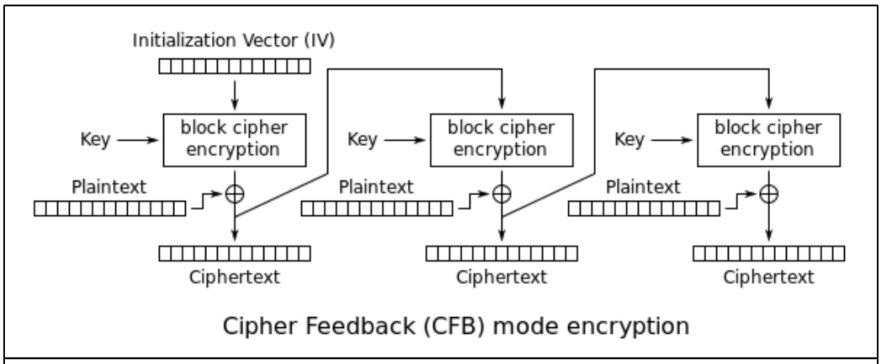


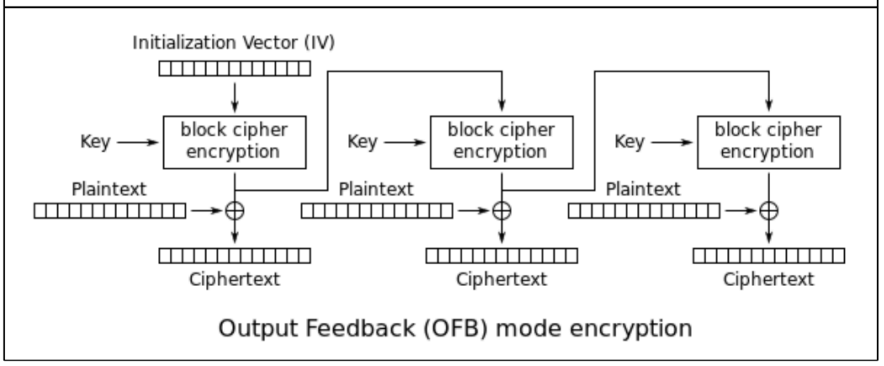
CBC Decryption can be parallelizable:



CBC is the most common & secure, secure only if nonce is random and unpredictable cause predictable IV leads to CPA attacks like the one (BEAST) exploited in TLS in 2011.
It's slow during encryption and requires two different circuits Enc = PRP, Dec = PRP^{-1}, other modes use the PRP either for End and Dec.
Plaintext must be multiple of block size, so there is some padding used to fill the space. (Padding is also used in other modes).

Other modes: CFB, OFB:





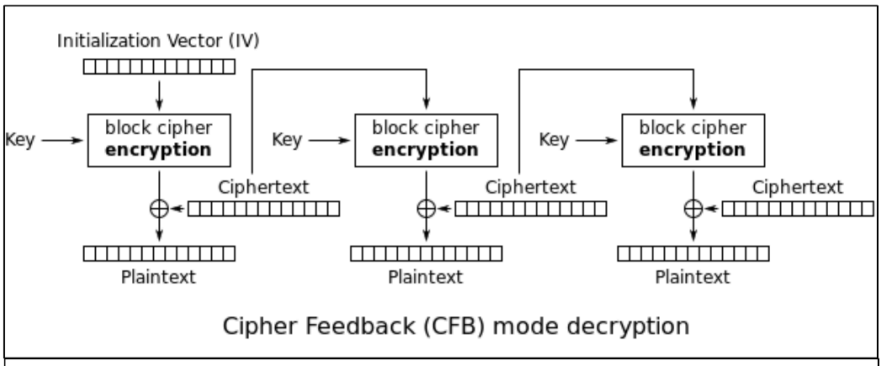
Block cipher used in «stream» manner
- PT encrypted with XOR
- No need for padding

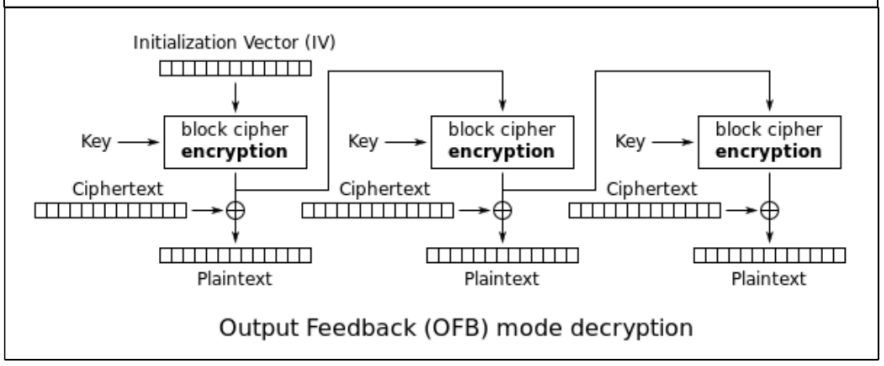
Non parallelizable in encryption
- But OFB advantage: permits preprocessing!

Reason: CFB, like CBC, depends on previous ciphertexts
- $c[0] = m[0] \oplus ENC_K(IV)$
- $c[i] = m[0] \oplus ENC_K(c[i-1])$

OFB «keystream» depends only on IV!
- **Robust to errors**

Looks like a classical stream cipher!



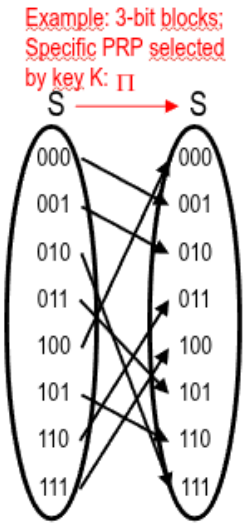


DECRYPTION advantage:
Reuse encryption block!
No need for «inverse» PRP implementation

Further CFB advantage:
Parallel decryption
OFB still serial decrypt

CFB is parallelizable because when u receive the ciphertext u don't need to permutate anymore the IV but u permutate directly the ciphertext u have.

Chaining block: short cycle problem



1) OFB keystream, with IV=010?

010 111 100 000 001 010 111 …

cycle = 5

2) OFB keystream, with IV=011?

011 101 110 011 101 110 011 …

cycle = 3 – shorter! Cycle may depend on IV…

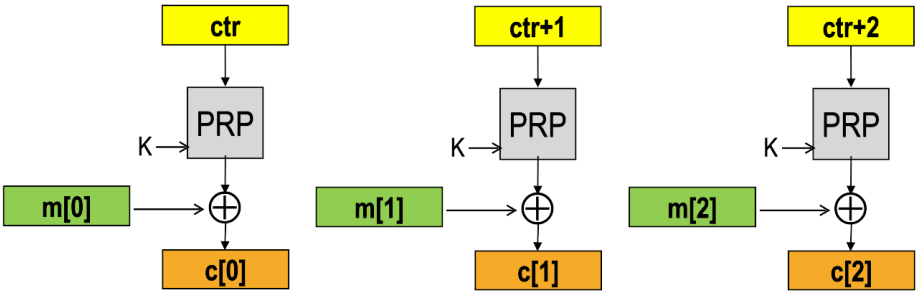
3) Same problem in CBC when encrypting a «regular» plaintext.

Example: encrypt 011 011 011 with IV=010

CT = (010) 010 010 010

Counter Mode (CTR)

Initialize counter ctr, and increase it at every new block. Encrypt the counter with block cipher (independent of plaintext, can be precomputed). XOR that output with plaintext block. In practice build a PRNG keystream out of a block PRP, it's provably secure: if PRP is secure then PRNG is secure as well.



AES-CTR is the standard used today.

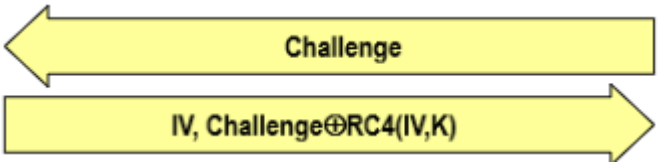
This is the mode with most of the advantages: turn block cipher into a stream one and combine all the advantages of CFB and OFB.
It is efficient cause u can do parallel encryption and decryption.
Requires the implementation of encryption block only unlike ECB and CBC which require also inverse block.
The decryption of the i^{th} block does not depend on previous blocks (unlike CBC or CFB).
It's secure: Counter if properly used doesn't repeat and there is guaranteed no short cycle problems. Since block is a permutation we have the same output block when the same input block repeats with a periodicity of 2^n.

Back to CHAP-type authentication

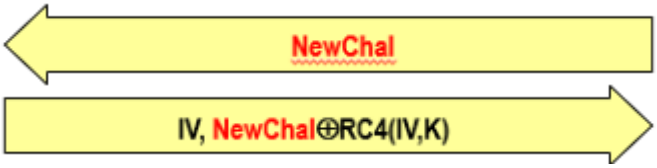
There are two way to prove u know a secret:
- Hash(secret, challenge)
- ENC_{secret}(challenge)

Wifi WEP
Let's now replace RC4 (that is broken) with AES.

If we use AES-CTR the problem is the same:
The attacker sees:

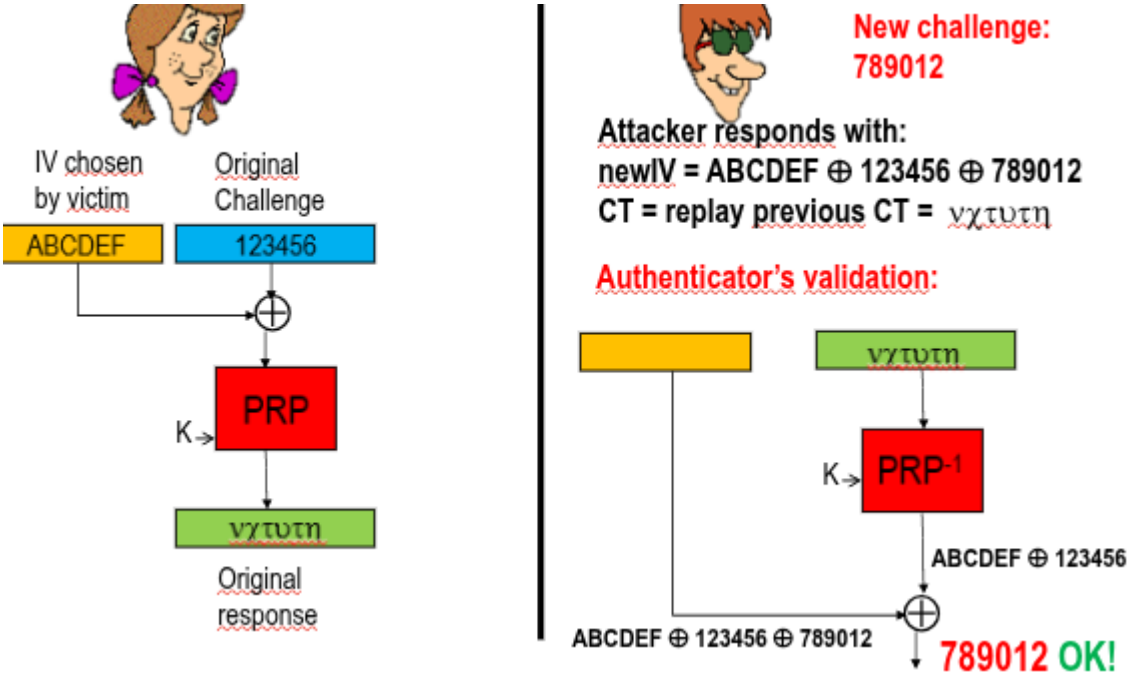


So he extracts the keystream RC4(IV,K) (know plaintext attack situation) and encrypts a NEW challenge by reusing previous IV and keystream:



So even if we use the AES-CTR we can use the identical attack pattern.

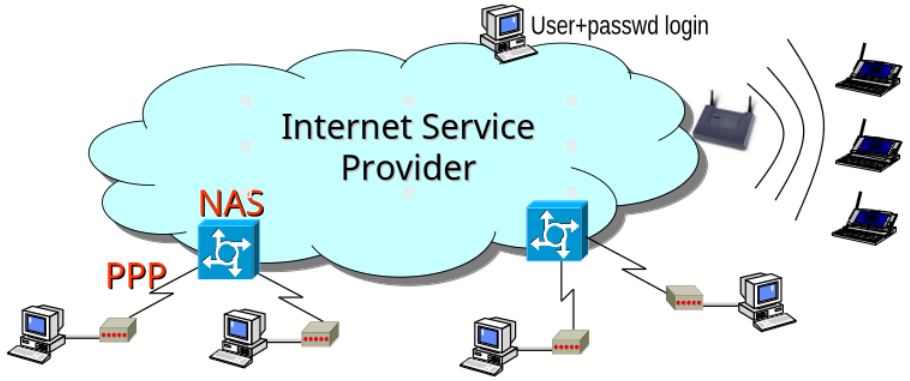
If we use AES-CBC:



If we use AES-ECB
This is the most secure one despite being the worst algorithm (there is no best or worst, depends on attacker's model). As long as challenge never repeats there is no way for attacker to forge a valid auth.

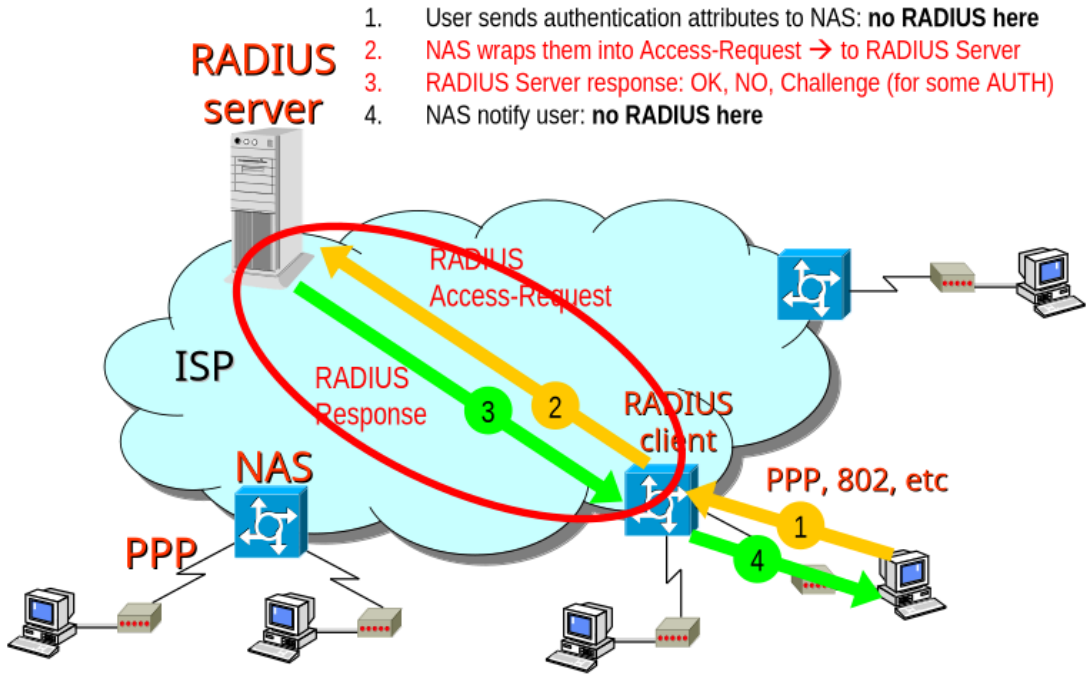
Golden auth rule: leave zero freedom to authenticating device! And nonce (e.g. challenge) decided by authenticator; devices must only respond to authenticator requests, with no extra parameters to decide (e.g. IV).

Handling Remote Access: RADIUS (Remote Authentication Dial In User Service)
Managing large scale networks is really difficult, multiple NAS, multiple access services. Not only authentication, but also service-specific configuration assignment.



This is best achieved by managing single user database.

RADIUS

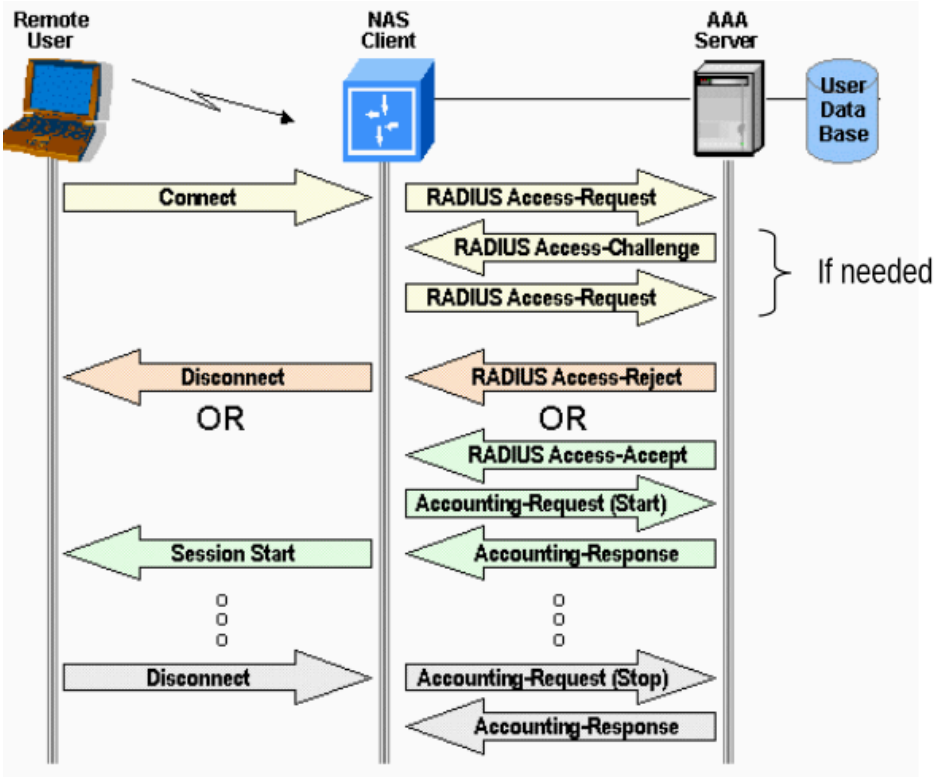


It provides centralized AAA functionalities:
- Authentication: r u really the one u claim to be.
- Authorization: do u have permission to access a service.
- Accounting: what are u currently doing / paying.

Radius: client-server protocol
Based on UDP/IP, uses server port 1812 and logically centralized with one primary server other servers may act as proxy.
The client is the NAS (don't confuse it with the end user).
Registered User Database: for each entry (user_name), it contains (at least): authentication information (secret), authentication method, authorization attributes;

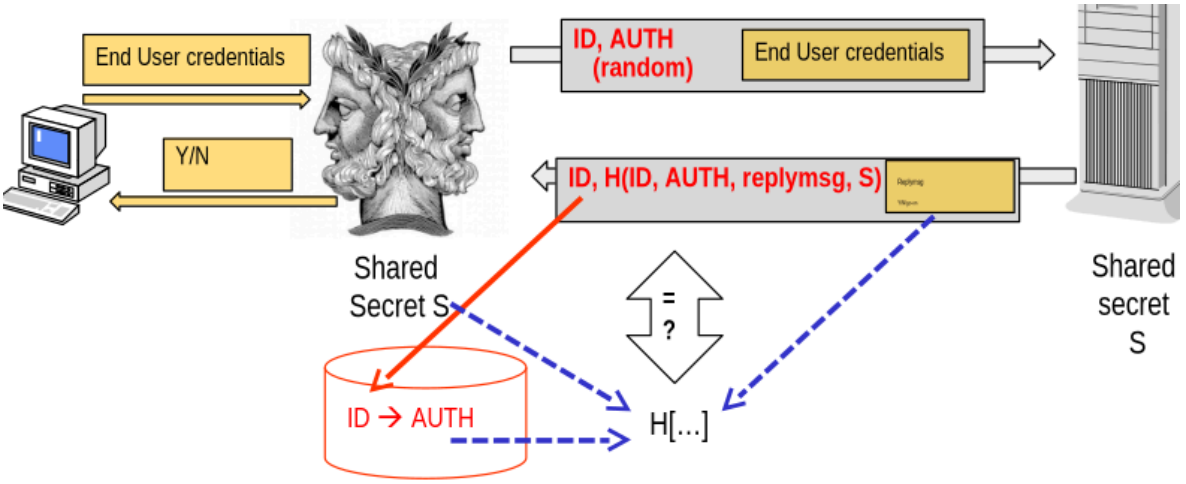
Client Database: contains the clients which are entitled to communicate with the server;
Accounting Database: When radius needs accounting.

Message exchange



RADIUS Security features
- Pre-packet authenticated reply: Shared-secret based so no transmission of secrets is needed but only the reply is authenticated, hash-based not HMAC-based, MD5, and often low-entropy shared key.
- Encrypted user password transmission: Uses a custom transmission but the same shared key used for authentication.

RADIUS authenticated reply

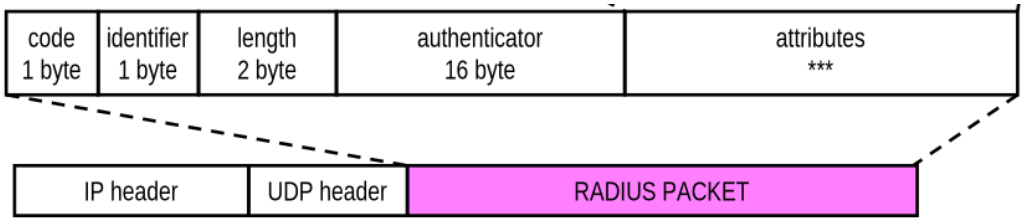


(auth is the same as nonce)

Sort of challenge-response: challenge is the request authenticator, response also include reply message.

User - Nas scheme (above):
The end user sends credentials (use, passwd) to the NAS.
NAS - RADIUS server:
The NAS itself does not know whether these credentials are correct, its job is to relay them securely to the RADIUS server. The NAS prepares a RADIUS Access-Request message containing:
- ID: identifies this request uniquely.
- Authenticator (AUTH): a 16-byte random challenge.
- User credentials: encrypted.
Everything is protected using the shared secret S (known only to NAS and server).
RADIUS server - NAS:
The RADIUS server: Verifies the user credentials, builds a reply (e.g. Access-Accept or Access-Reject), computes a Response Authenticator to prove authenticity (the hashed one).
NAS - User
Depending on the verified RADIUS response, the NAS allows or denies access.

Packet format



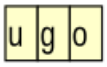
Code: type of radius packet (1 Access-Request, 2 Access-Accept, 3 Access-Reject, 4 Accounting-Request, 5 Accounting-Response, 11 Access-Challenge);
Identifier: match request with response;
Length: minimum 20, maximum 4096;
Authenticator: used to authenticate reply from server, also used as nonce for passwd encryption;
Attributes: extensible information field.

Authentication field
In Access-Request (C to S): 16 random generated bytes, unpredictable and unique to avoid reply attack.
In response packets (S to C): one-way MD5 hash of: request ID, request authenticator, shared secret, packet response information.
Specifically:
MD5(Code | ID | Length | RequestAuth | Attributes | Secret)

Attributes:
Any number of attributes in a packet, order of them doesn't matter, some attributes may appear more than once, we can have up to 2^8 attributes (1 byte) where the 0 is reserved, 1-191 are IANA assigned/assignable, 192-223 are experimental and 241-255 are reserved. It is an extensible protocol, new attribute values can be added without disturbing existing implementations.

Access-Request
Typically contains: who is the user (user-name), passwd (user-passwd e CHAP-passwd), an identifier of the RADIUS client (NAS-IP or NAS-Identifier), an identifier of the port the user is accessing (NAS-Port).

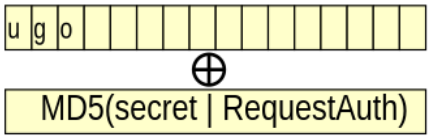
Password encryption
Native password:



Step 1: padding to 16 bytes

Step 2: generate a 16 bytes hash using key and the content of the authenticator field of the request

Step 3: XOR padded passwd and hash



If passwd longer than 16 characters:
Step 4: compute MD5(secret | result of previous XOR)
Step 5: XOR with next segment of the passwd

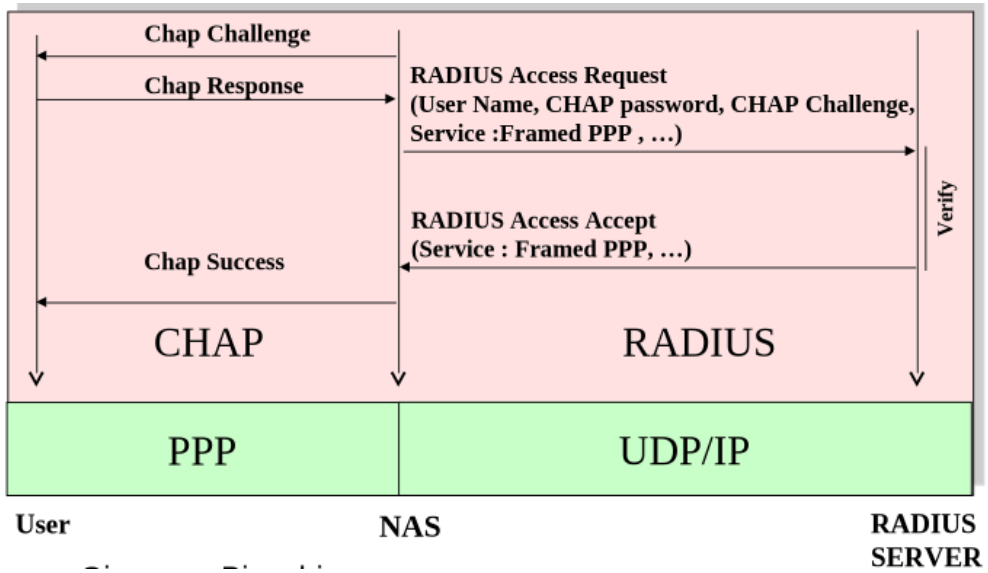
Access-Accept
It's the positive server response (when user authentication credentials are ok) and contains all the service-specific configuration like the Service-Type attribute and other service-related configuration parameters (IP address, mask, …).

Access-Reject
Two main reasons: Authentication failed or 1+ attributes in the request were not considered acceptable.

Access-Challenge
Used whenever the server needs the user to send a further response, typically contains one or more reply-message attributes like text or an explicit authentication challenge. NAS collects responses from the user and sends a NEW Access-Request. Based on this, the server accepts or rejects or sends another challenge.

PPP CHAP support with RADIUS
A CHAP challenge locally generated by NAS is sent to the User, it responds with the CHAP response to the NAS that sends in it’s turn the to the RADIUS server the RADIUS Access Request (includes the CHAP informations). The RADIUS server first retrieves the user passwd from database, computes and compares CHAP response that can be an Accept /

Rejects / Challenge, sent to the NAS that responds in its turn to the User.



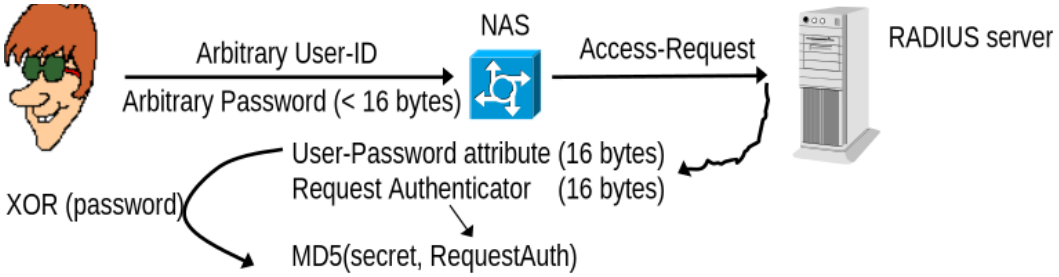
Possible attack:
Lisa request a challenge from the NAS (1234) and responds (abcd), the NAS now sends a RADIUS Access Request with a challenge and the payload of Lisa (challenge of NAS and response).
The attacker requests a challenge (5678) and responds with a random msg, the NAS doesn't verify anything so sends the attacker (user, challenge, passwd) directly to the RADIUS server. The response from the RADIUS Server will be negative.
Now the attacker can intercept the RADIUS Accept Request before it arrives at the NAS and change the internal payload with the response (abcd) and challenge (1234) of Lisa. In this way the request is valid and grants the access to the attacker.

RADIUS Security Weakness
It is vulnerable to message sniffing and modification cause it’s a clear-text protocol (user-name, calling-station-id, nas-identification, local attributes are in clear) and Access-Request are not authenticated so may be forged or modified. To resolve this problem they added a Message-Authenticator attribute and now is mandatorily used with EAP.
Authenticator = HMAC-MD5(whole packet).

Dictionary attack to shared secret
Usually there is a low-entropy shared secret, many implementations only allows ASCII and often a single one is used for the entire network.
Is obviously a weakness if all the NAS share the same secret S, its better to a different secret for every NAS. Now it’s a problem to remember all the secrets, it’s better to remember only one.
In order to generate a lot of different secret and remember just one, what can be done is take the static identifier of the NAS and a secret S and generate the hash.
That hash is used as the secret between the NAS and the RADIUS Server.

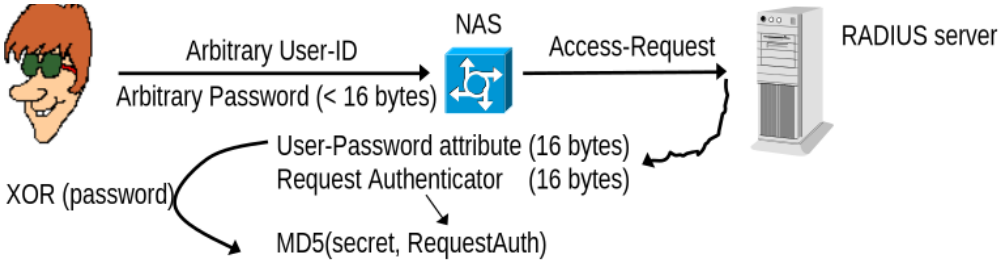
Many hooks for offline dictionary attacks:
Intercept any request-response pair, the request gives u random ReqAuth

ReqAuth = MD5(Code | ID | Length | ReqAuth | Attributes | Secret)
(the secret place at the end makes pre-computation attack easier!)
Don't even need to get the a pair, just a request with user passwd suffices cause the same secret is also used for password encryption.

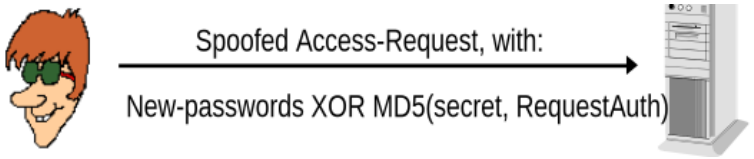


Now u can attack the secret by bruteforce, cause all the other fields are known so u just bruteforce the secret until the MD5 is the same.

Attacking the password of a user
Now that we know the secret we can exploit the password cause
b2 = MD5(Secret | c1)
c2 = Passwd2 XOR b2
Step 1: like the previous case, but with valid user ID:



Step 2: Attacker is now able to encrypt the user password, May exploit:
1) lack of upper limit on authentication rate at server-side
2) RADIUS servers typically do not check for authenticator reuse
Works only with 16 or less byte passwd cause it divides into blocks of 16 bytes.



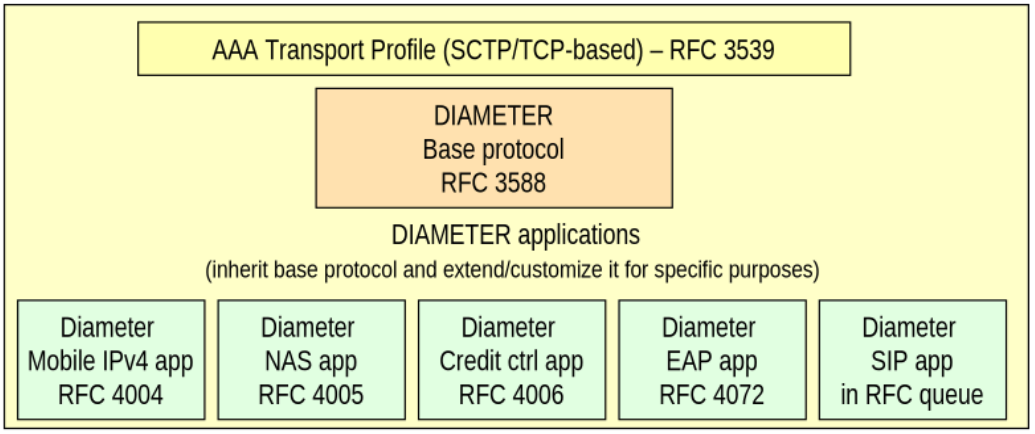
Poor PRNG Implementations
Security of radius: requires uniqueness of the Request Authenticator so must be a nonce. Some implementations as poor PRNG generators are often non-crypto and have short cycles and are predictable. So when facing a PRNG u need to verify what is the cycle, what about predictability and if it has unique or repeated values.

Assume PRNG cycle = 2^N, at best we have 2^N different values, thanks to the birthday attack we have ½ the probability with approx $2^{N/2}$.
Bad implementation idea is to start from a 32bit random number and generate a 128bit authenticator by appending 4 subsequent 32 bit random numbers so the cycle is reduced to 30bits.

Lesson learned from RADIUS
AAA protocols should NOT implement their own security mechanisms. The current trend is to rely on an underlying security layer like TLS or IPsec.

RADIUS today
RADIUS initially developed to mainly support dial-in PPP users and terminal login users, today is a de-facto standard for AAA but has severe functional limits:
- Scalability: from a few thousand to several M users in cellular operator today (not solvable by adding many NAS because AAA servers and their networks may experience congestion as well as packet drop).
- Extensibility: new technologies came out.
- Interoperability: lot left unspecified.

IETF standardization
- RADext: Started on aug 2003, limitations of RADIUS can be circumvented with the extensions like RADext: RADIUS extensions with mandatory backward compatibility, doesn't use transport and security enhancements (still UDP).
- Diameter: Started on dec 1998, now completed, is a brand new protocol, much more powerful, activities moved to DiME (Diameter Maintenance and Extensions WG).
Diameter picture:



Diameter improvements:
- reliable transport: (vs RADIUS unreliable UDP) essential in accounting, SCTP preferred (otherwise TCP)
- standardized error and fail-over control: (vs RADIUS implementation dependent approaches).
- extension of functional limits: 24bit AVP (attribute-value-pair) field vs 8 attribute field, no more 8 bit ID but 32, duplicate detection, capability negotiation: (vs RADIUS reject answer if attributes not supported) mandatory / non-mandatory (M) AVPs allow to restrict and reject

only to serious lack of capabilities.

Diameter header

Version: 0x01	Message length (3B)
R P E T res-flags	Command-Code (3B)
Application-ID (4B)	
Hop-By-Hop Identifier (4B)	
End-To-End Identifier (4B)	

Flags:
R: 1=request, 0=answer
P: proxiable
E: this is an error message
T: potentially retransmitted message

AVPs

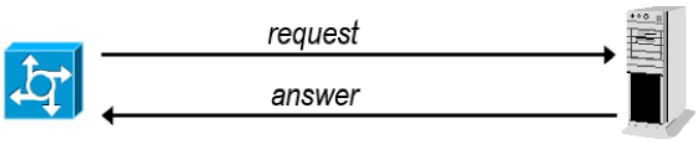
AVP code (4B)	
V M P res-flags	AVP length (3B)
Vendor-ID (optional, 4B)	
..... DATA	

Flags:
V: vendor-specific bit: vendor ID follows, code is from vendor
M: mandatory bit: reject if this AVP unsupported
P: privacy bit: need for e2e encryprion

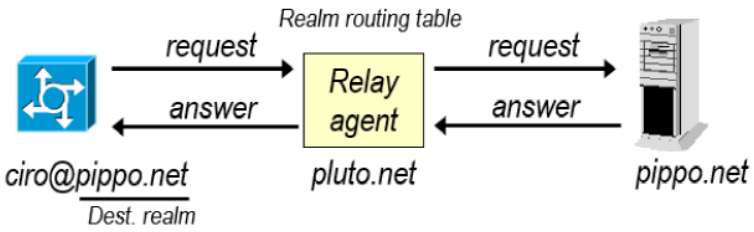
- peer discovery, configuration, capability detection: (vs RADIUS where clients are manually configured) uses specific IETF protocols, C/S exchange capabilities when they set-up a transport connection.
- supports unsolicited S to C messages: (vs RADIUS pure client-server paradigm) allows unsolicited abort/disconnect, reauthentication/reauthorization.
- very detailed management of intermediate entities: (vs RADIUS sloppy specification) clear distinction between hop-by-hop and end-to-end, explicit specification of Proxies, Redirects, Relay agents.

Diameter agents operations

No intermediate agent

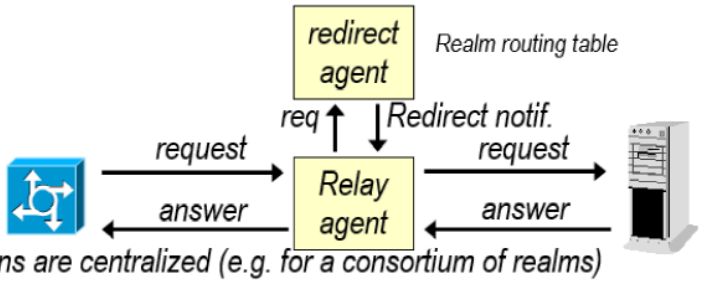


Relay agent: *accept request and routes it to the proper server based on info contained In the message (destination realm)*



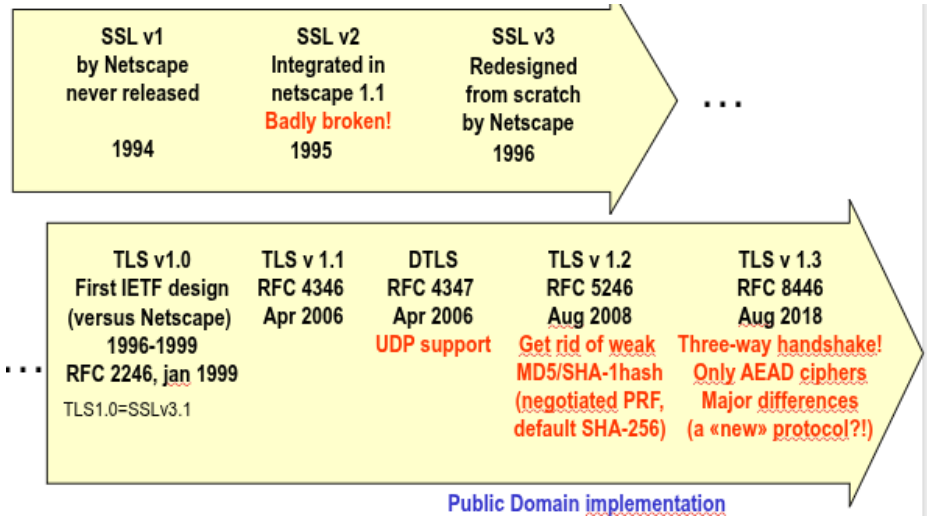
Proxy agent: *as relay but modifies message (hence e2e message authentication broken)*

Redirect agent: *provide routing decision on incoming request but does not actually route request (returns redirect)*



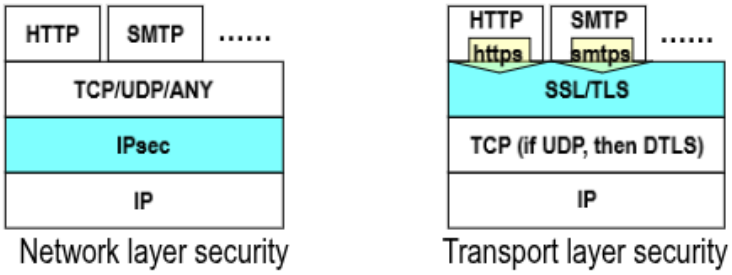
Useful when Diameter routing decisions are centralized (e.g. for a consortium of realms)
Typical usage: individual relays default-route to redirect agent

SSL (Secure Socket Layer) / TLS (Transport Layer Security)



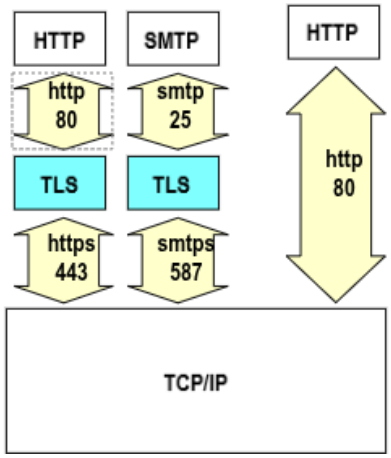
SSL is the name of the product of Netscape, TLS is the name of the standard that turns SSL into an official standard. Both are the same.

SSL/TLS: layered view



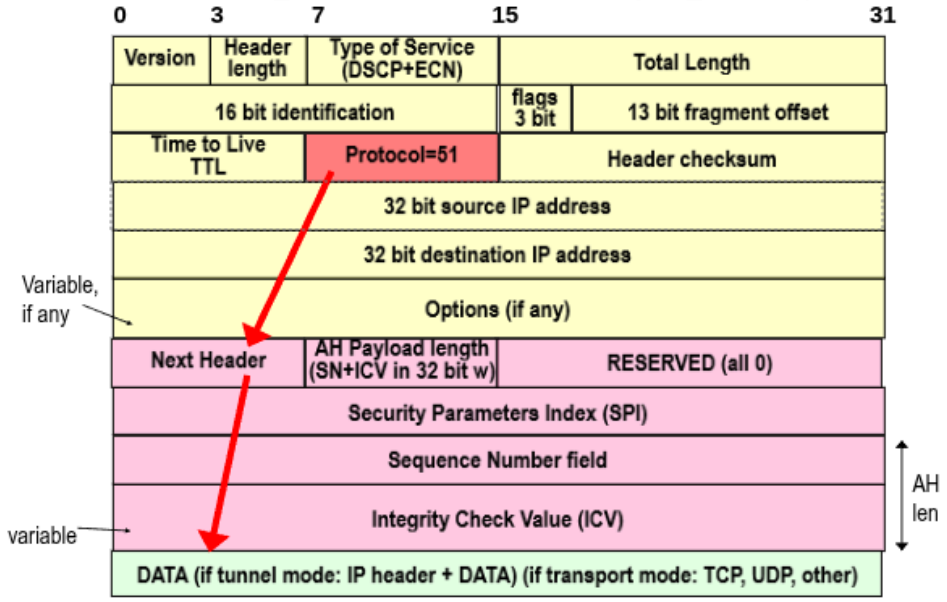
SSL/TLS is on top of the TCP but below the application layer so in between the transport and the application layer (it can be considered a Session layer protocol), Its not a security enhancement of TCP and its not necessarily limited to Internet transport (L4).

Application support



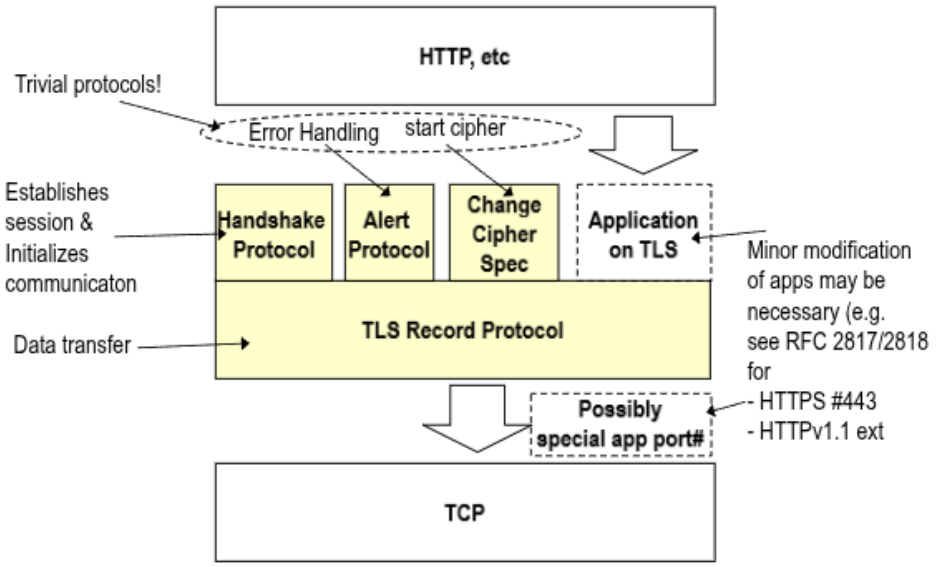
Bad historical idea: reserve special port number for HTTP over SSL/TLS (http 80, https 443), if TLS used for other applications then we have special port numbers as well (smtps 465, spop3 995, …), works well cause is a standard de facto but we have 2 reserved port numbers for same service, deprecated by IETF.

An alternative approach is to slightly adapt application's internals so the app reuse same port number.
Compare with IPsec (Authentication Header AH)
The IP header has 51 as protocol and indicates that encapsulate the AH that encapsulate in its turn the DATA.

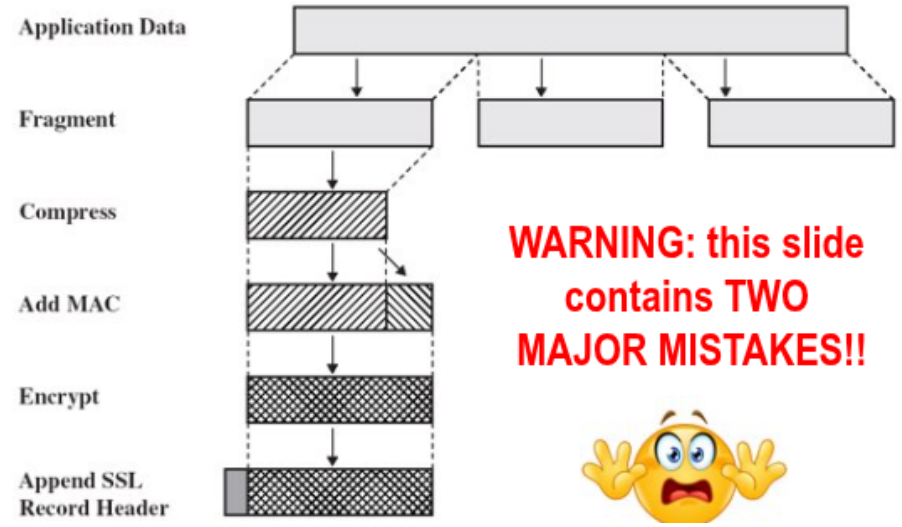


TLS goals
Establish a session (TLS Handshake phase): agree on algorithms, share secret, perform authentication;
Transfer application data: communication privacy with symmetric encryption and data integrity (HMAC);
Can be seen as two protocols in one, one for establishing a session (e.g. IPsec) and one for delivering the data and enforcing security services.

TLS protocol stack



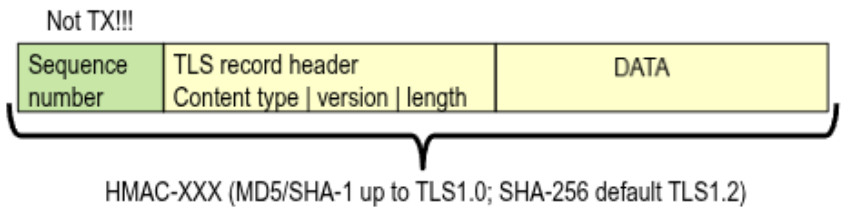
TLS up to the 1.2 ver



First mistake: Compress and then encryption is vulnerable. On the contrary doing first encryption and then compression is correct.
Second mistake: The order of adding encryption and integrity is wrong.

Fragmentation
At application, the msg is fragmented (not to confuse with TCP segmentation) in blocks of 2^{14} byte size.
Compression
Lossless compression formerly used in SSLv2, Considered in TLSv1.0 but not specified, in 2004 is introduced DEFLATE and with the diffusion of verbose languages like XML was considered appealing but it suddenly died in september 2012 after the CRIME attack (so its not used anymore).
MAC
Secret (symmetric) key derived from security parameters negotiated during handshake along with the hash function used. The computation uses HMAC construction.
(Note: A network protocol should be independent on the cryptographic algorithm used)
Encryption
Fragment Encryption applies to both (compressed) fragment and MAC, can be used a stream or block cipher, the algorithm, necessary padding and derived secret key are negotiated during the handshake (differs from key used in MAC). Encryption algorithm cannot increase size of more than 1024 bytes.

To avoid reply attacks sequence numbers are used, these are kept at both connection extremes (C-S), initialized to 0 and up to 2^{64} and not transmitted. It bounded with the TCP, (cause if we use for example UDP packages can be lost: packet 3 can never arrive), and if we DTLS we need to send also the sequence number.



It's computed the HMAC of the Sequence number + TLS record header + DATA but the Sequence Number its not transmitted.

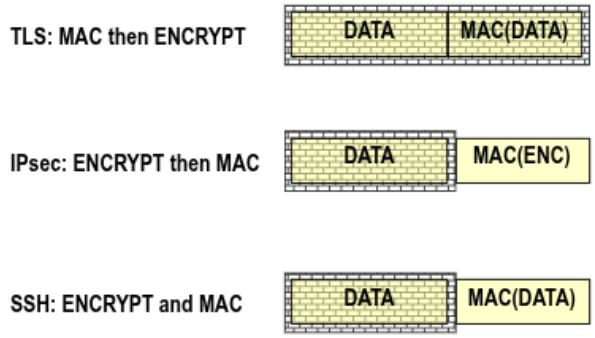
Encryption vs Integrity
Integrity: Prevents message spoofing (injection) and tampering (modification).
Integrity may be the only requirement if everybody should see but nobody should be able to change.

Encryption doesn't guarantee integrity.
One time pad (one time random key K): perfect secrecy but

- ⇨ Encryption: ENC(M) → C = M⊕K
- ⇨ But:
  - →C⊕M' = (M⊕K)⊕M' = (M⊕M')⊕K=ENC(M§ M')
  - →"pay 1000 $" § "…(1 § 9)…" → "pay 9000 $"

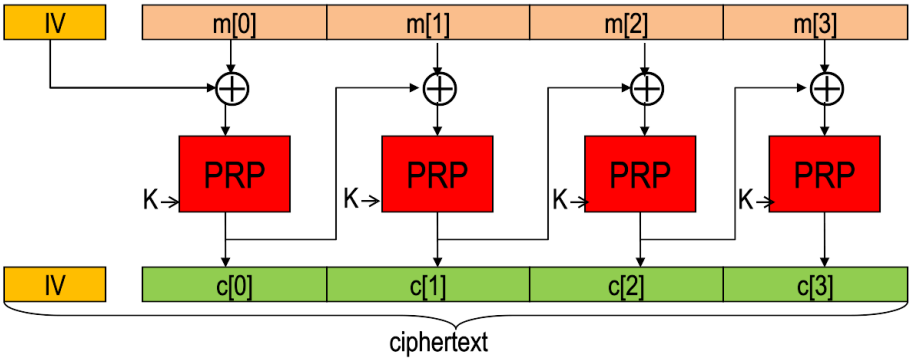
In general don't trust encryption mechanisms for integrity unless they are explicitly designed also for it. (TLS 1.3 only AEAD).

How to combine ENC+MAC

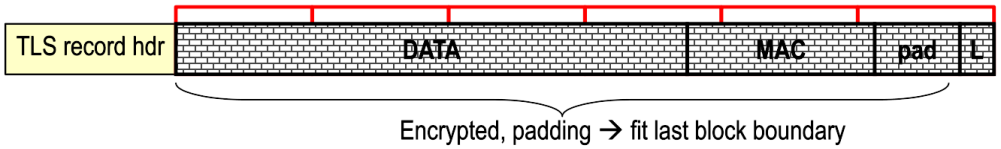


SSH is the most insecure: unforgeable MAC construction doesn't guarantee anything about information leakage! MAC (not encrypted and applied to plaintext) may reveal something about the message.
TLS is not recommended, there are pathological counterexamples and practical attacks found.
IPsec is provably secure if semantically secure Encryption scheme and unforgeable MAC scheme, so always use ENC then MAC

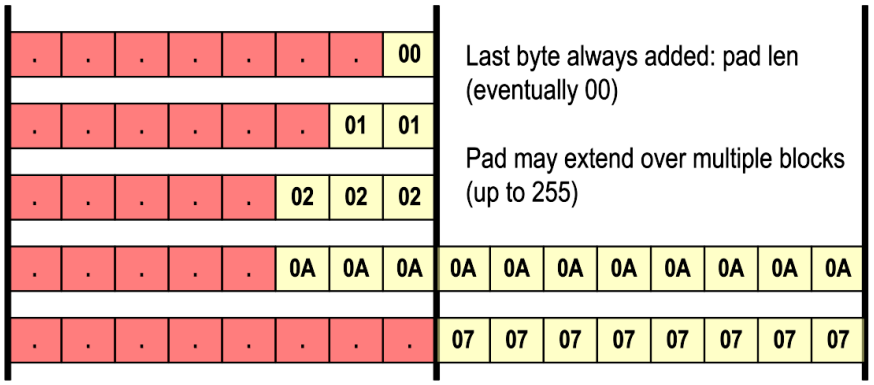
CBC encryption



TLS is vulnerable with CBC using the CBC Padding Oracle attack in TLS 1.0



When the text is padded u need to fill the msg and need to tell the receiver how many bytes must be removed. Last byte represent the length of the padding, ((01) means that the padding except the block itself is 1). To verify that the padding is correct the last byte is repeated along all the padding. (a padding even longer that the block can be made (like 3)).



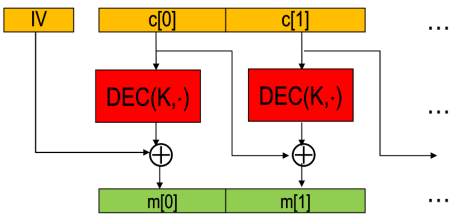
CBC decryption steps
Decrypt and verify that the #bytes are multiple of the block size otherwise the decryption is failed; Then read the last byte to know the # of padding to remove and remove the last bytes unless there is an invalid pad (like bytes that aren't equal to # like (02) only one time at the end); Lastly its checked the MAC if it's different there is a problem.

In network protocol we explain the error reason but it's opposite in crypto cause the attacker can use that vulnerability.
The attack:

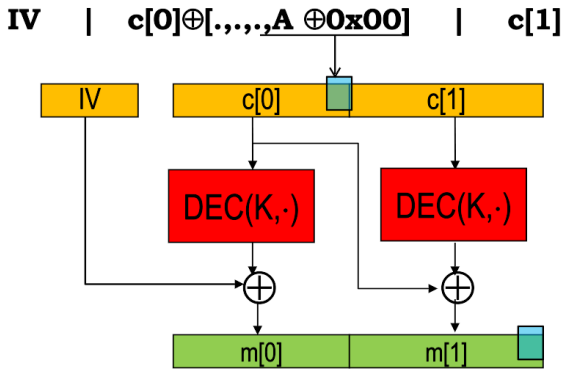
Goal: decryption of a block like c[1] (c[0] needs a different approach)
The attacker is able to submit another ciphertext (chosen ciphertext attack) and get the result of Decryption Failed and also Bad MAC. If I receive either of these two this means that the padding was found Ok, its like to have a padding oracle that tells us if the padding is ok or not.

➔c[i] = ENC(K, c[i-1] ⊕ m[i])

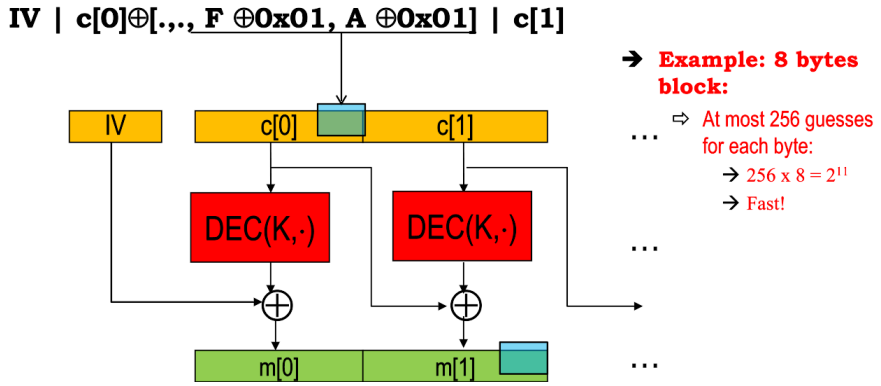
➔m[i]= c[i-1] ⊕ DEC(K, c[i])



Assume that the message is only 2 blocks. I can control what's the value of the last plaintext block m[1] by tampering with the previous ciphertext block c[0].



To guess the last byte what I do is taking the last byte of c[0] and make it the xor between the guess (A) and the padding expected (0x00).
If the guess is correct then in the plaintext I will have as the last byte of m[1] = 0x00, and the oracle will give u bad MAC, so the padding is OK. (if the guessed is wrong than we will have in the m[1] a wrong padding and not 0x00).
We now know last byte is A, what we do now is guess the previous byte.



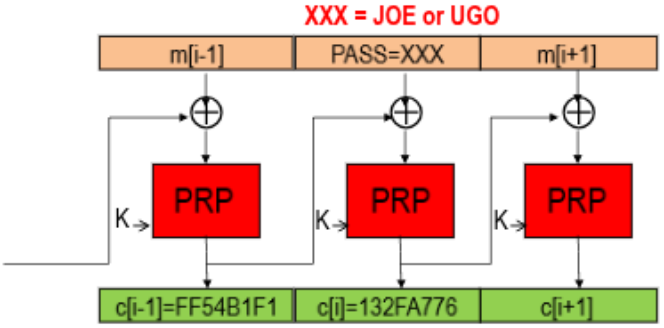
We can iterate that until we get all the entire message.

This attack was discovered in 2002 and the TLS protocol was corrected to avoid such padding oracle, but in 2003 was discovered a side channel "padding oracle" so the implementation was corrected and in TLS1.1 the MAC always performed even if malformed msg.

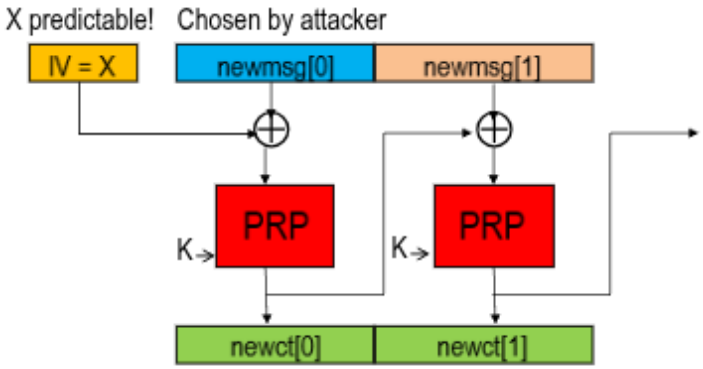
This attack apparently is not practical cause the bad MAC and the decryption failed are fatal alerts so every time the TLS connection is aborted and the next connection will have a different key.
Instead was used in IMAP where the client sends periodically login/passwd every few minutes and with a MITM attack the TLS msg was taken.

Fixes for these attacks
TLS 1.2: if padding fails, validate MAC in any case, but there is no way to know the message size if the padding fails so the whole data is used for validation, but this requires extra time and thanks to that were created other attacks.

BEAST attack: CBC IV in TLS1.0
When sending multiple msg the IV must differ, otherwise is not secure. Different IV is not sufficient: IV must also be unpredictable.



Attacker know m[i] contains the passwd and guess its either UGO or JOE, but dosen't know which. Attacker can also see the ciphertext and can convict the implementation to encrypt some data of his choice (chosen plaintext attack):



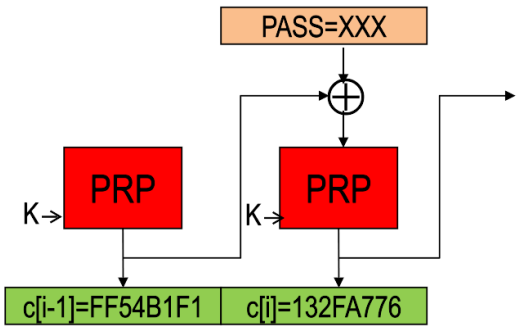
Attacker predicts X, sets newmsg[0]

Sets newmsg[0] = X⊕c[i-1]⊕PWGUESS

⇨Example: X§ "FF54B1F1"§ "PASS=UGO"

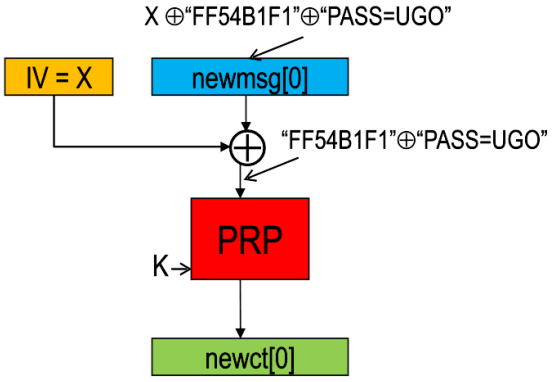
if newct[0] == c[i] then the correct guess its made. So if the IV is predictable a dictionary attack can be mounted.

Collected data, including target password



C[i]=ENC(K, "FF54B1F1"⊕"PASS=XXX")

Chosen plaintext attack



newct[0]=ENC(K, "FF54B1F1"⊕"PASS=UGO")

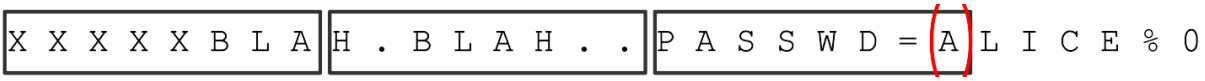
If equal to c[i], then guess right otherwise guess is wrong.

Exploited in TLS
The predictable IV is a well known vulnerability and in TLS1.0 the IV of a new msg was equal to the last c[i] of the previous message. That was corrected in TLS v1.1 where a new explicit IV was used for every message, this was also mandatory in DTLS (where the packets doesn't arrive in order).
This was a theoretical vulnerability but the attack was apparently hard to mount, until september 2011 with the BEAST attack.

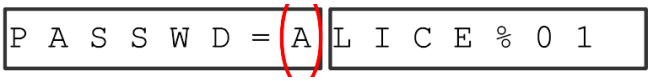
The attack was believed to be unpractical cause there were software issues: the chosen plaintext attack is active so was necessary an active agent code running on user browser or an injection of chosen plaintext. Found feasible with modern web technologies like Websockets and HTML5.
Another issue was the complexity one, cause a brute force attack for AES (128bits block) has 2^{128} guesses, and auth cookies may be long. The breakthrough was the chosen boundary attack made the attack linear with auth cookie size.

Chosen Boundary Attack
The authentication token is not inside a single block

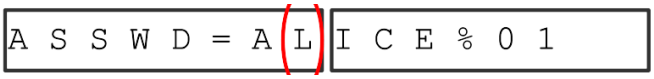
Insert a preamble text (controlled by attacker) and align auth token so that only first character is in block and read the relevant ciphertext



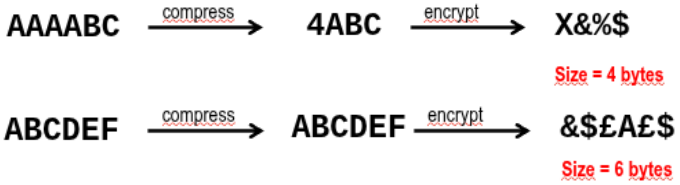
Now perform the attack: there are 256 guesses at most (less if we use base64 or if letter/numbers only used)



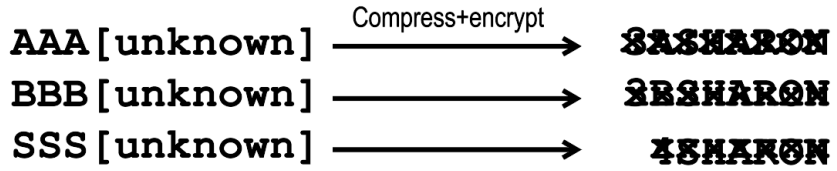
Once first character found, re-align block and perform the attack



CRIME attack: compression leaks (usable) information.
TLS Compression is harmful so it was disabled in Chrome after september 2012 CRIME attack and similar attacks in other settings. Not a problem of TLS but a more general problem (vulnerability known since 2002).
Let's say that compression works as follow: if x chars C are seen together they are compressed in xC



Your data is unknown to attacker but he can see size of compressed+encrypted data so he can repeatedly inject chosen plaintext prior to your data and see size of encrypted+compressed result. Let's make un example with SHARON:



Now we know that the first letter of the data is S.

CRIME (Compression Ration Info-Leak Made Easy) works independent on the cipher.
Injection of chosen text prior to user data reusing BEAST software tools.
Attack relies on DEFLATE that uses two subalgorithms: Huffman coding and LZ77 that replaces repeated 3+ char strings with (offset; size) pointers:

```
GIUSEPPE BIANCHI AND MARCO BIANCHINI      36B

GIUSEPPE BIANCHI AND MARCO (-18,7)NI      31B
```

Attacker inject chosen text so known header + unknown secret
ex: passwd=alice%01
Message crafted by attacker Att: input + [header + secret] and repeated many times
The attacker sees Len(encrypt(compress(att_input+header+secret)))
Example:

```
GET /comment:twid=a HTTP/1.1 Cookie: twid=flavia\r\n
GET /comment:twid=a HTTP/1.1 Cookie: (-24,5)flavia\r\n
```

Repeat guesses... until

```
GET /comment:twid=f HTTP/1.1 Cookie: twid=flavia\r\n
GET /comment:twid=f HTTP/1.1 Cookie: (-24,6)lavia\r\n
```

1 byte shorter!!

And now start guessing second character as well!!

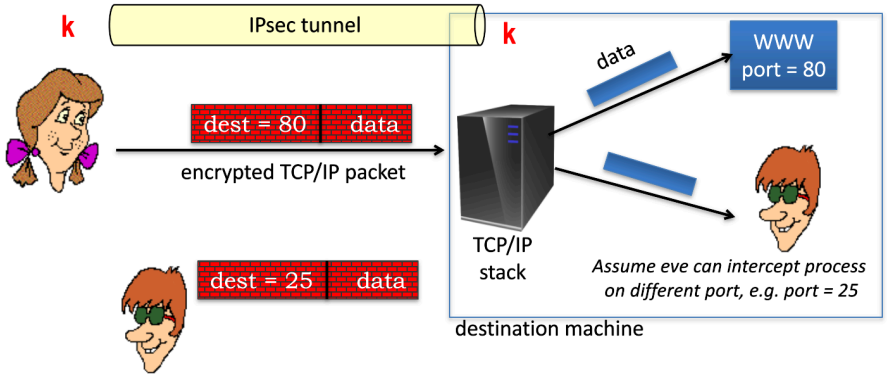
```
GET /comment:twid=fa HTTP/1.1 Cookie: twid=flavia\r\n
GET /comment:twid=fa HTTP/1.1 Cookie: (-24,6)lavia\r\n

GET /comment:twid=fl HTTP/1.1 Cookie: twid=flavia\r\n
GET /comment:twid=fl HTTP/1.1 Cookie: (-24,7)avia\r\n
```

And so on, linearly with secret size (again!)

Compression works in bits, not bytes and also uses huffman coding so u must be careful to get at least a byte difference. Works on two protocols either on TLS and SPDY. There are several optimizations to make the attack faster.

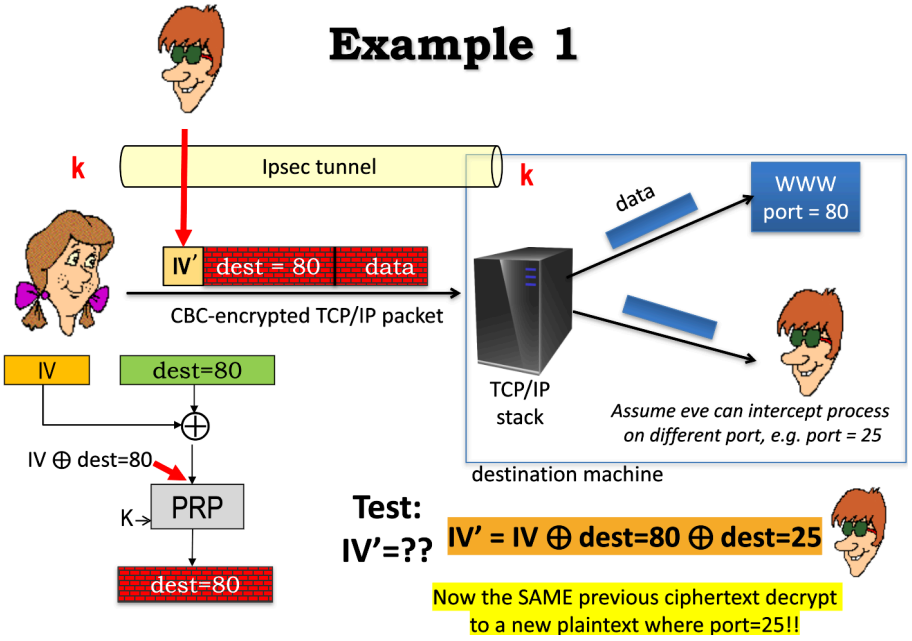
Confidentiality: semantic security against CPA
Integrity: unforgeability under CCA
The Authenticated Encryption is a cipher designed to be secure against active attacks (tampering), ensuring both confidentiality and integrity.

Tampering attacks can also break confidentiality:
When sending a package is specified the destination port, but let's say that the attacker can monitor a different port. Even if we protect the communication on the network (like with IPsec tunnel) if the attacker could just change the port he could read data in clear.



TRIVIAL, with CBC encryption!!!

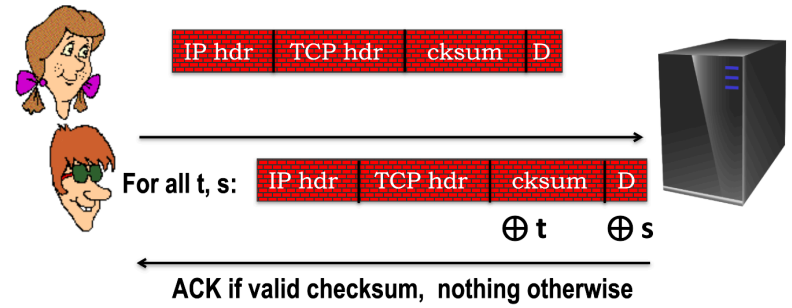
Let's try a CBC attack: Just tamper the initial IV in order to change the destination port.



The attacker model is too strong cause the attacker has access to the backend.

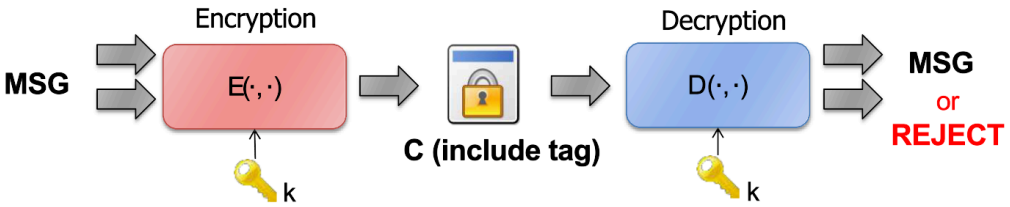
Let's try CTR: The attacker has only network access and uses a remote terminal app so each keystroke is encrypted with CTR mode. TCP will ACK only packets with correct cksum and we can use that as an oracle.

The cksum is 16bit the keystroke is 1 byte so what we can do is:



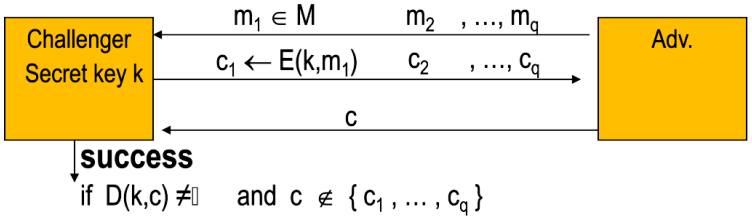
MANY (!) valid equations: {checksum(hdr, D⊕ s) = t ⊕ checksum(hdr, D)}
hdr often known, {cksum, D} unknown → may be solved!

CPA security cannot guarantee secrecy under active attacks. If message needs integrity but no confidentiality use a MAC, instead if the message needs both integrity and confidentiality use authenticated encryption modes.

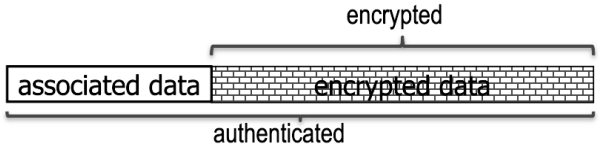


Unlike standard ENC which always returns a msg, AuthEnc may output a reject, it is fundamentally stronger than basic cipher: decrypts ONLY IF authentication tag is valid. Prevents attackers from performing CCA (Chosen Ciphertext Attack).

Definition: AE is secure if
1. is semantically secure under CPA;
2. guarantees ciphertext integrity (ciphertext integrity informally means that the probability of success is negligible, note: success even if forged c decrypting to completely random msg).



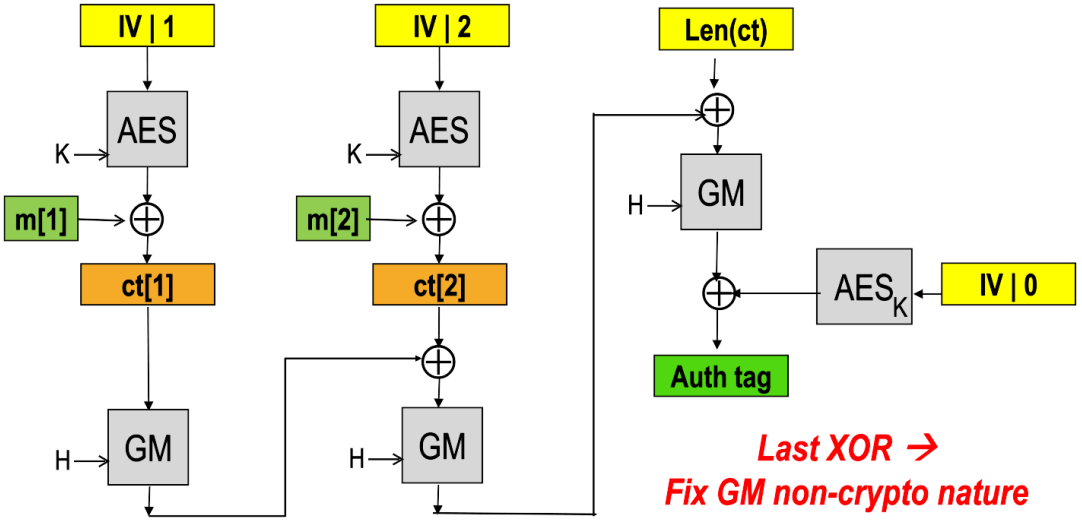
AEAD: AE with Associated Data



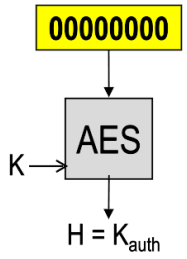
Associated data: often packet header that must usually remain in plaintext, there are also extreme cases.
With no associated data it becomes only AE and this means CCA-secure encryption, no encrypted data means it remains only the associated data that needs to be authenticated so its just a secure MAC.
Modern protocols only use AEAD, if not that Encrypt-then-MAC that is also CCA-secure.

Design choices for AEAD
Structure: can be two-layer: first ENC then MAC(AES-GCM) or just one-layer: all in one (OCB).
Performance: first encrypt then mac is slow and full parallelizability is hard.
Functional requirements: where to put the associated data?
Misuse resistance: what happens if we reuse IV?

AES-GCM: Galois Counter Mode
First standardized algorithm used in most security protocols (IPsec, TLS, …) uses Encrypt-then-MAC structure and encrypts with AES-CTR and mac with GHASH (not a crypto hash but much faster).
Construction
In the construction must be included the Len and the IV in the auth tag.



The key used for the encryption (K) is different from the key used in MAC (H):



Only key K is needed, and with that we can generate H.

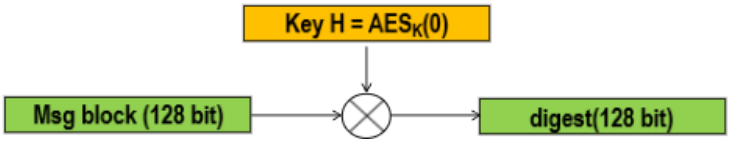
Wegman-Carter MAC
Crypto hash are slow we can use a two building block: Universal Hash Function and Pseudo Random Function to make it secure like a crypto hash function.

Universal Hash Function

A keyed hash function is a family of function: $H_k(msg) \rightarrow \{0,1,\ldots, m-1\}$ → k = key
The family is universal if it's hard for an attacker which does NOT know the key to find a collision $H_k(M_1)=H_k(M_2)$.
Many differences respect to crypto hash: if u know the key finding a collision can be easy and the output may not be pseudo-random.

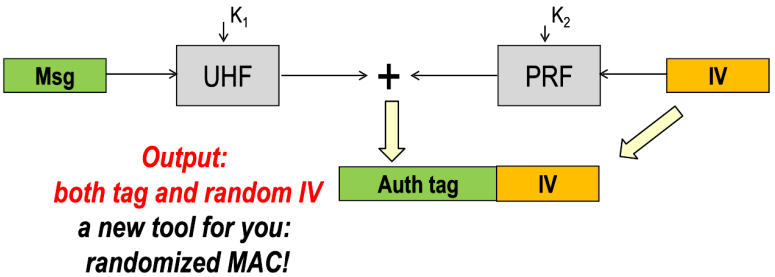
For any pair (y,x) with y not equal to x the Prob{$H_x(M) = H_y(M)$} is <= 1/m. Changing the key should randomly change digest, and there should be no pair (M1,M2) that gives the same hash for many different keys.

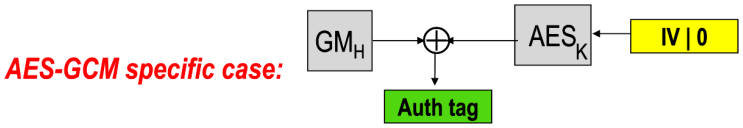
AES-GCM: GHASH
GHASH is not crypto, it's just a universal hash function

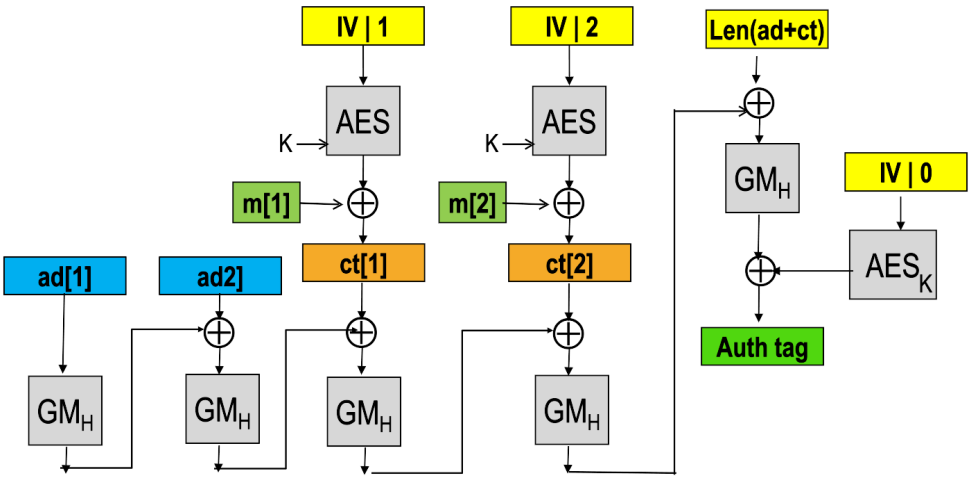


Building block: multiplication in GF(2^{128}), meaning polynomial carry-less multiplication (much faster than usual multiplication).

A secure MAC can be constructed from an UHF (universal hash function) and a PRF (pseudo random function) in this way:







Len adjusted to include also associated data

GCM critical issue: nonce reuse

AES-GCM tag: $GHASH_H(CT) \oplus AES_K(IV,0)$

Nonce reused (same IV):

$tag1 = GHASH_H(CT1) \oplus AES_K(IV,0)$

$tag2 = GHASH_H(CT2) \oplus AES_K(IV,0)$

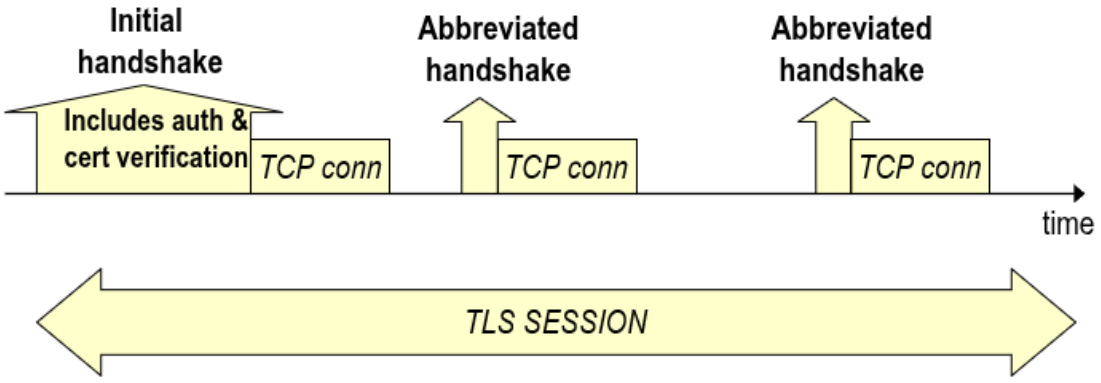
$tag1 \oplus tag2 = GHASH_H(CT1) \oplus GHASH_H(CT2)$

But GHASH is linear (poly mult) so the attacker can now recover the auth key H and forge valid messages! There is AES-GCM-SIV (Symmetric IV) that is more robust to nonce reuse.

TLS Handshake Protocol

In a TLS session there are multiple TCP connection and especially the first has to do a lot of things: it does an initial negotiation and exchange of parameters in which they mutually authenticate, agree on algorithms, exchange random values and exchange secrets or information to compute secrets.

The next TCP connection are only for regenerate the session keys (refreshing secrets).



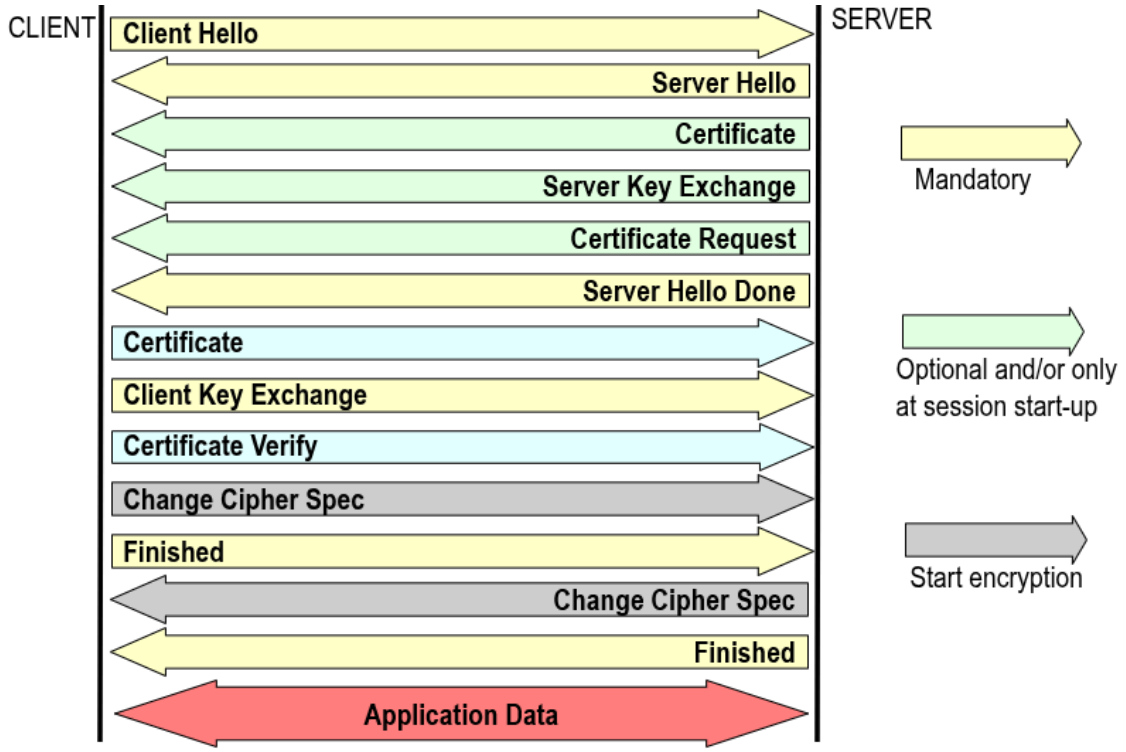
The handshake goals are:

Secure negotiation of shared secrets via asymmetric cryptography that is never transmitted in clear and derived from exchanged crypto parameters;

Optional authentication for both C/S, in practice always required for server and robust to MITM attacks;

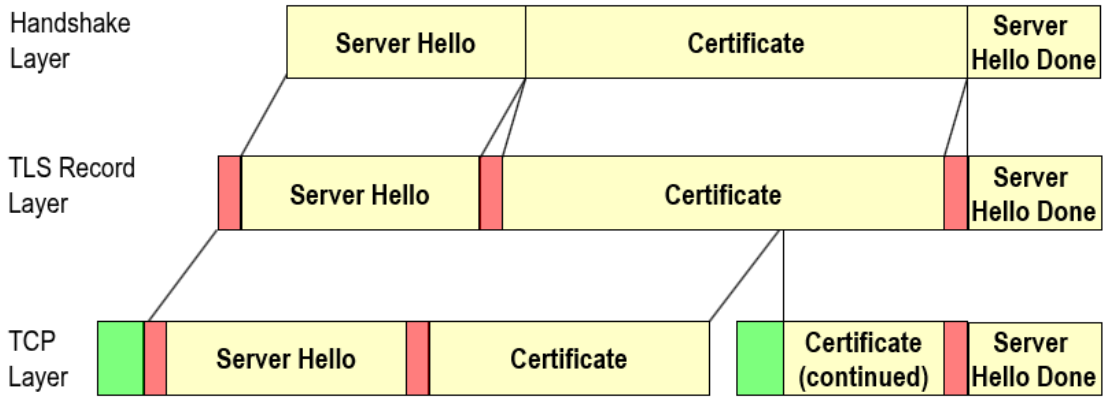
Reliable negotiation, so the attacker cannot tamper communication and affect or alter the negotiation outcome without being detected by the involved C/S parties, the negotiation is amenable to downgrade attacks.

Handshake messages



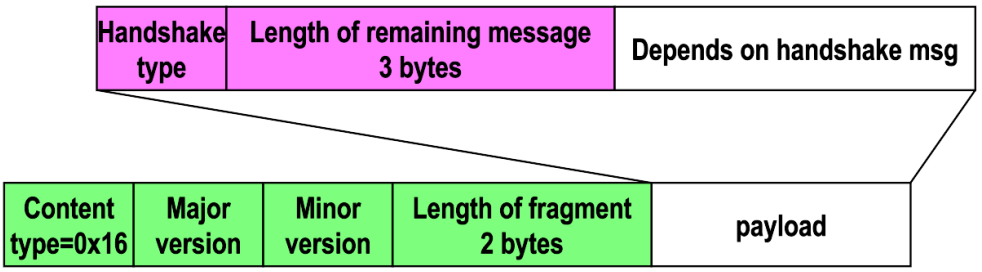
(blue part used in mutually authentication)

TCP segmentation

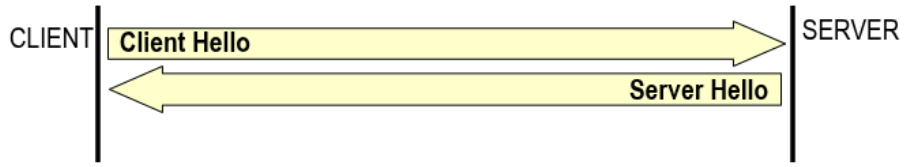


Handshake message format

it's encapsulated in TLS Record, and there are several handshake types: hello_request, client_hello, server_hello and so on.



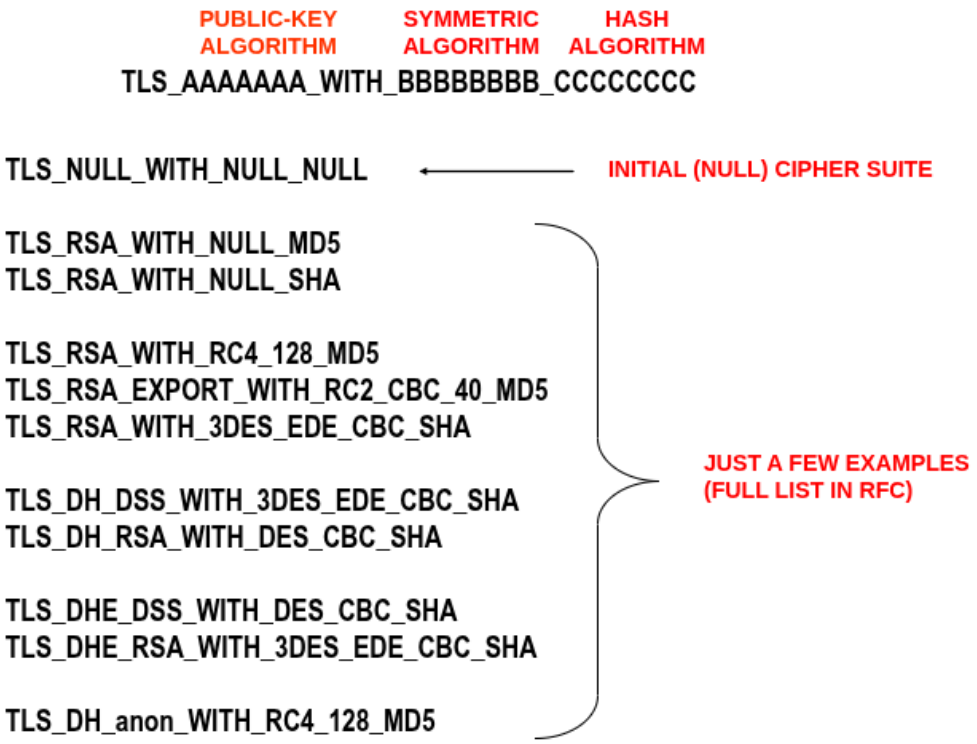
Handshake phase 1



General goal: create the TLS/SSL connection between C/S, agreeing on TLS/SSL version, defining session ID, exchanging nonces (timestamp + random values used to generate keys), agreeing on cipher algorithms and on compression algorithms.

The first message is sent in plain text, incapsulated in 5 bytes TLS Record and contains: Handshake type, Length, Version, 32 bytes Random (4 bytes Timestamp + 28 random), Session ID Length, Session ID, Cipher Suites length, Cipher suites, Compression length, Compression algorithms.

Cipher suites
The cipher suite is made in this way: TLS_ public-key algorithm_ WITH_ symmetric algorithm_ hash algorithm.

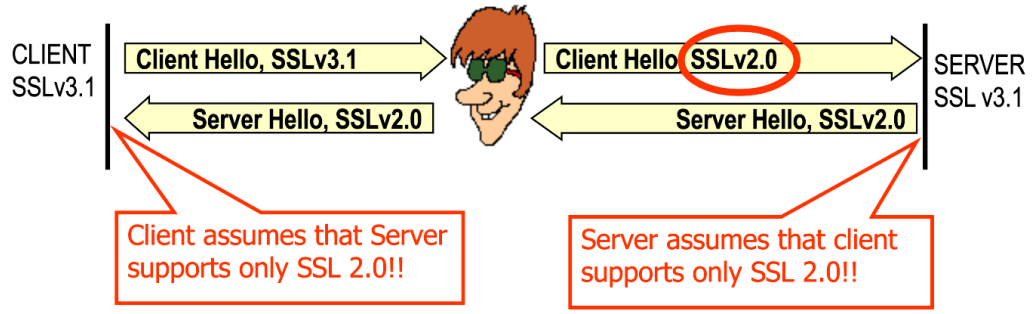


An arbitrary combination is not possible, must choose from a small list of combinations. Its impossible in TLS to support integrity and not encryption.

In reply to client hello, the server hello is sent in plain text and contains:
Handshake type, Length, Version, 32 bytes Random (4 bytes Timestamp + 28 random), Session ID Length, Session ID, Cipher Suites length, Cipher suites, Compression length, Compression algorithms.

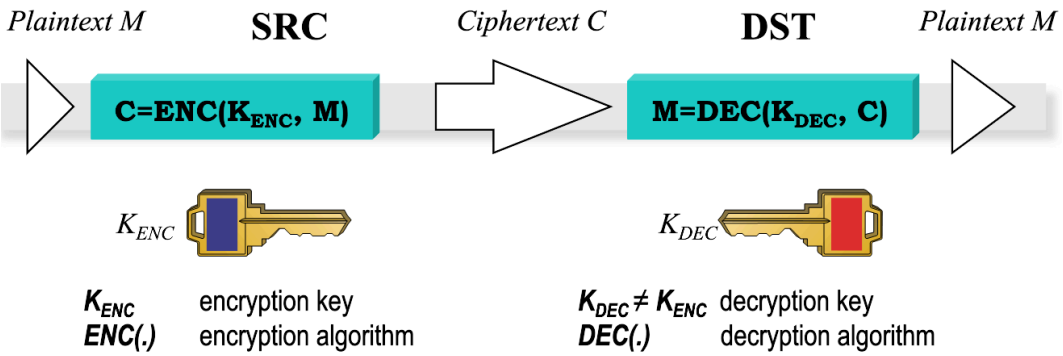
Downgrade attack
Let's suppose a MITM attack (the MITM attack dosen't break SSL3.x but knows how to

break SSL2.0 that uses 40 bit encryption)



The Client Hello SSL3.1 is intercepted by the MITM and it sends to the server an SSL2.0, the server will reply to the MITM with a Server Hello SSL2.0 that will be forwarded to the client. In this way MITM removes strong cipher suites and leave only the ones he knows he can break.

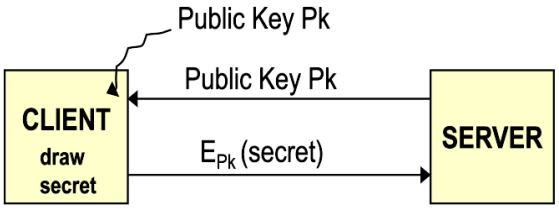
How public key cryptography is used in TLS
A symmetric cryptography is not bad cause it isn't a good idea to transfer and store the symmetric key, the problem is solved with asymmetric cryptography where it's used an encryption key and a decryption key.



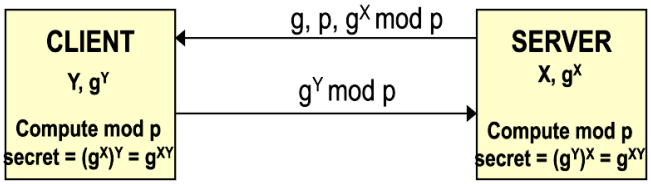
The ENC and DEC Key are linked but cannot be determined from each other so only the K_{ENC} can be public, in this way only the server is able to decrypt and everyone is able to encrypt.

Asymmetric and Symmetric address different problems and both can be (in)secure, it depends on algorithms and keysize. Asymmetric is more flexible cause there is no need to preshare keys but u need more complex protocols. And asymmetric is computationally heavier than symmetric.
The best thing to do is to use asymmetric crypto only to exchange a shared key and THEN transfer massive data using symmetric encryption.

There are actually two basic approaches to key management
1. Key transport (e.g. RSA)
Server sends Public Key, Client draws a random secret which is encrypted using the P_k and transported to the server, both peers now have the same secret.

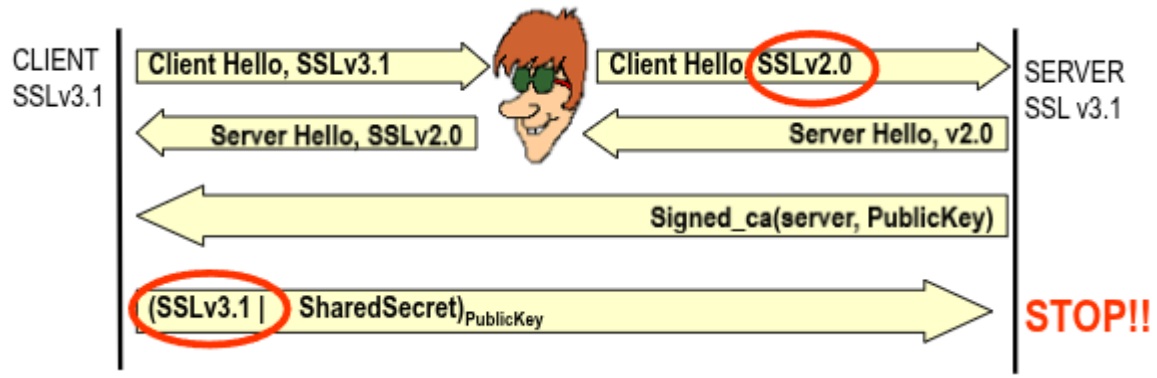


2. Key agreement (e.g. DH)
Parties independently generate a private and a public quantity, then exchange only the public quantity, this combined with other party's private part, permits both of them to compute the same secret so now both peers have the same secret.



Handshake phase 2 and 3

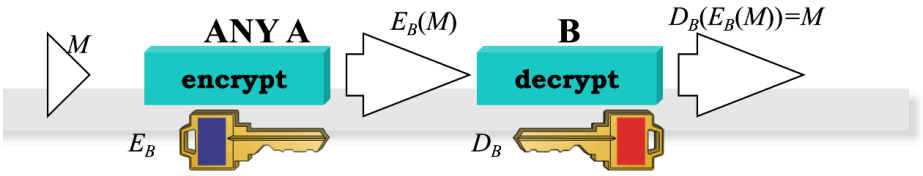
Phase 2: server sends authentication information (certificate) along with public key (in certificate or in extra msg).
Phase 3: the client generates a shared secret in which the length depends on agreed cipher suite and transmits it to the server, encrypted with Public Key.
So against the downgrade attack



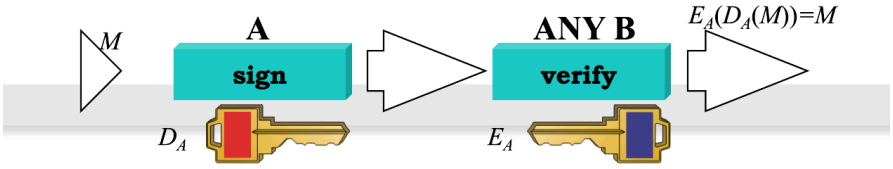
Encrypted SSL version is the one initially proposed, not the one negotiated in server hello.

Since the signed certificate cannot be tampered and since MITM cannot decrypt SharedSecret, downgrade attack on SSL version are easily discovered.

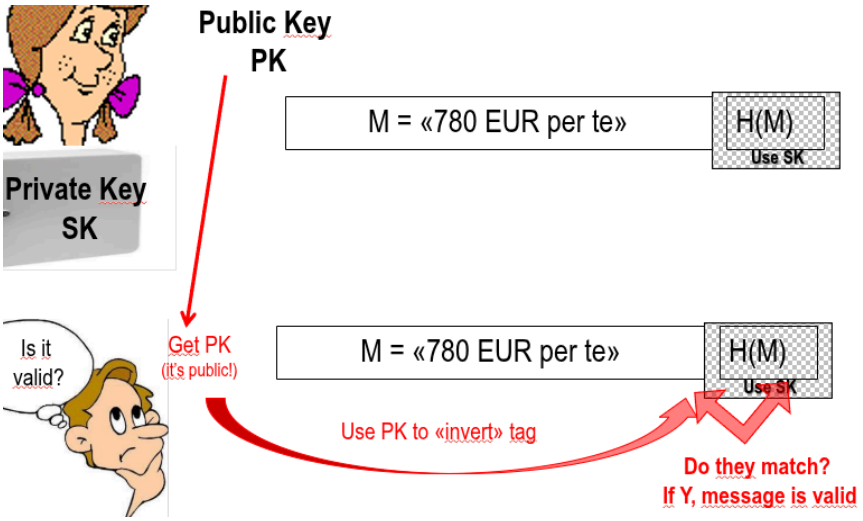
PubKey crypto: two ways to use it
ENC and DEC are inverse operations: DEC(ENC(M)) = M also ENC(DEC(M)) = M,
- Public key encryption: anyone can encrypt M, but only B can decrypt



- Digital signature: DUAL approach! Only A has K_{DEC} and can apply DEC (sign) but anyone else has K_{ENC} and can apply ENC (verify).



Digital signature example



So digital signature is message integrity with source authentication
Unlike symmetric MAC (e,g, HMAC): where K is the symmetric key shared by SRC and DST in the Digital signature, K is the SRC private key and DST just needs to know SRC PubKey to verify.

HARD (asymmetric) PROBLEM
It is computationally EASY in one direction, but computationally HARD in the opposite direction.
- Diffie-Hellman:
Discrete logarithm problem in prime fields, given large prime p, g, x, compute $y = g^x \bmod p$ is easy, but given $y = g^x \bmod p$ compute $x = DLog_g(y) \bmod p$ is hard
- RSA:
Factoring a product of two large primes, given p and q large primes, compute N = pq is easy, but given N find the p and q factors is hard.

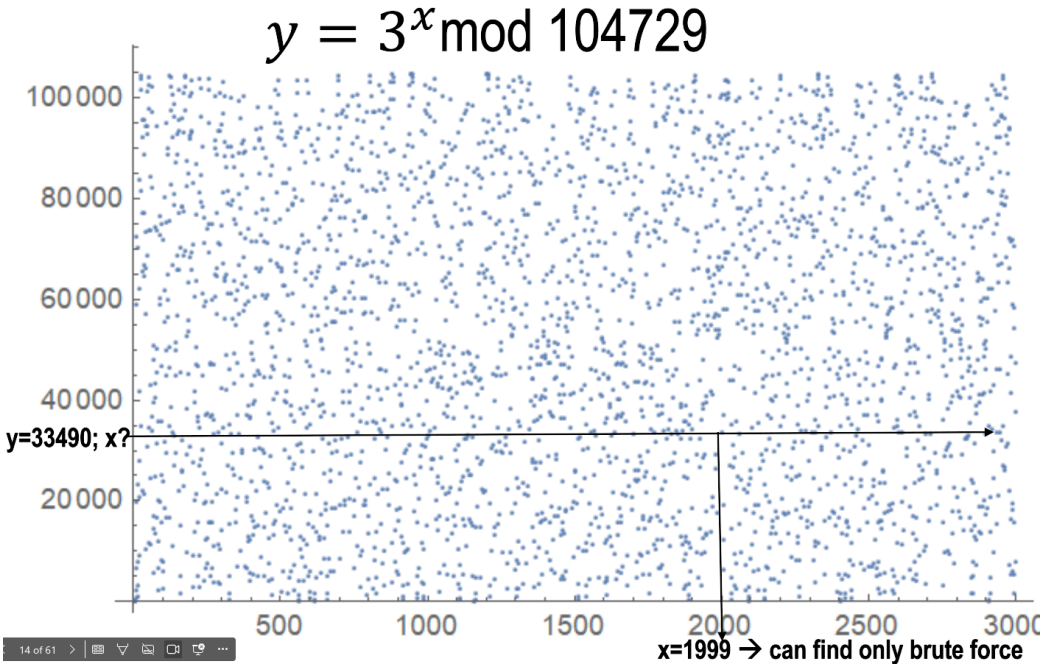
Modular exponentation is easy (Square&Multiply)

- ➔ **Goal: compute g^{1437} mod p**
- ➔ **Express x in bits**
  - ⇨ $1437_{10}=10110011101_2$
- ➔ **list bits from lsb to msb**
- ➔ **First line: initialize**
  - ⇨ Column Square = g
  - ⇨ Column Multiply = 1 (if b=0) or g (if b=1)
- ➔ **Start from 2° lsb to msb**
  - ⇨ For every bit
    - →Square;
    - →If 1: multiply to result
- ➔ **Complexity:**
  - ⇨ O(nbit) squares
  - ⇨ O(nbit/2) multiplications

$b: lsb->msb$	*Square*	*Multiply*
1	g	$\times g \rightarrow g$
0	g^2	.
1	$\left(g^2\right)^2 = g^4$	$\times g^4 \rightarrow g^5$
1	$\left(g^4\right)^2 = g^8$	$\times g^8 \rightarrow g^{13}$
1	$\left(g^8\right)^2 = g^{16}$	$\times g^{16} \rightarrow g^{29}$
0	$\left(g^{16}\right)^2 = g^{32}$	.
0	$\left(g^{32}\right)^2 = g^{64}$	.
1	$\left(g^{64}\right)^2 = g^{128}$	$\times g^{128} \rightarrow g^{157}$
1	$\left(g^{128}\right)^2 = g^{256}$	$\times g^{256} \rightarrow g^{413}$
0	$\left(g^{256}\right)^2 = g^{512}$	.
1	$\left(g^{512}\right)^2 = g^{1024}$	$\times g^{1024} \rightarrow g^{1437}$

No algorithm such as this for the opposite problem!! That's why it is hard!

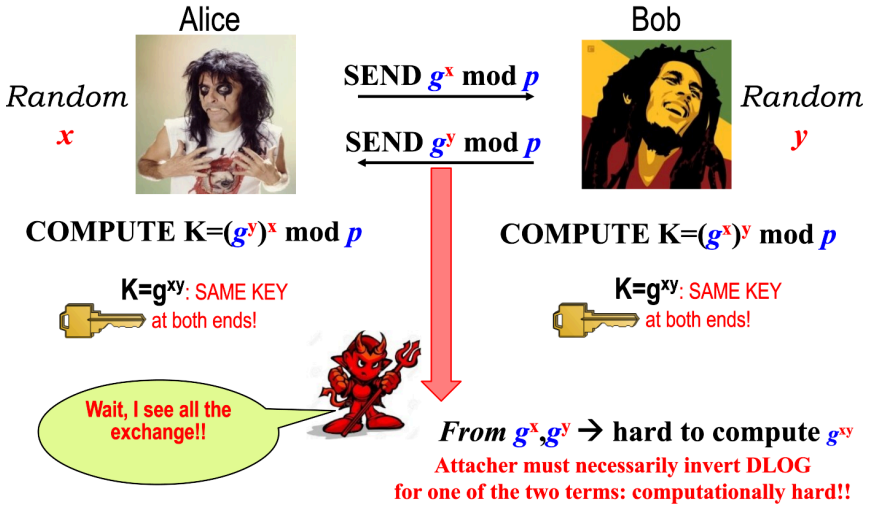
DLOG is hard



the only way to find x is through bruteforce

Diffie-Hellman Key Agreement

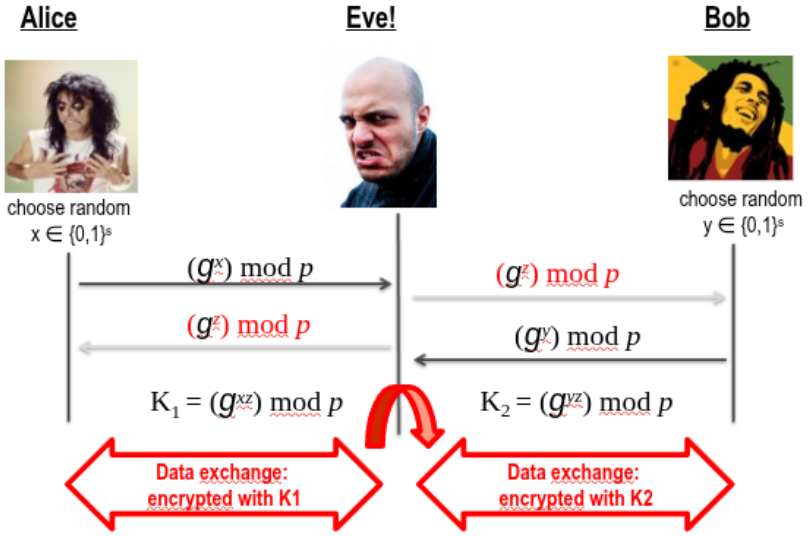
The Diffie-Hellman protocol was created in 1976 and didn't solve the PubKey cryptosystem problem (RSA solved it the next year). DH is a key agreement protocol and says how to setup a shared secret at both ends by only exchanging public values,without having a secure channel.



Be p a prime number and g a generator.

Both ends generate a random value x and y and sends each other g^x mod p and g^y mod p in this way Alice can compute K = $(g^y)^x$ mod p and Bob K = $(g^x)^y$ mod p. The key is now the same at both ends K = g^{xy}. If an attacker intercept g^x and g^y it will be hard to compute g^{xy} cause must necessarily invert DLOG from one of the two terms.

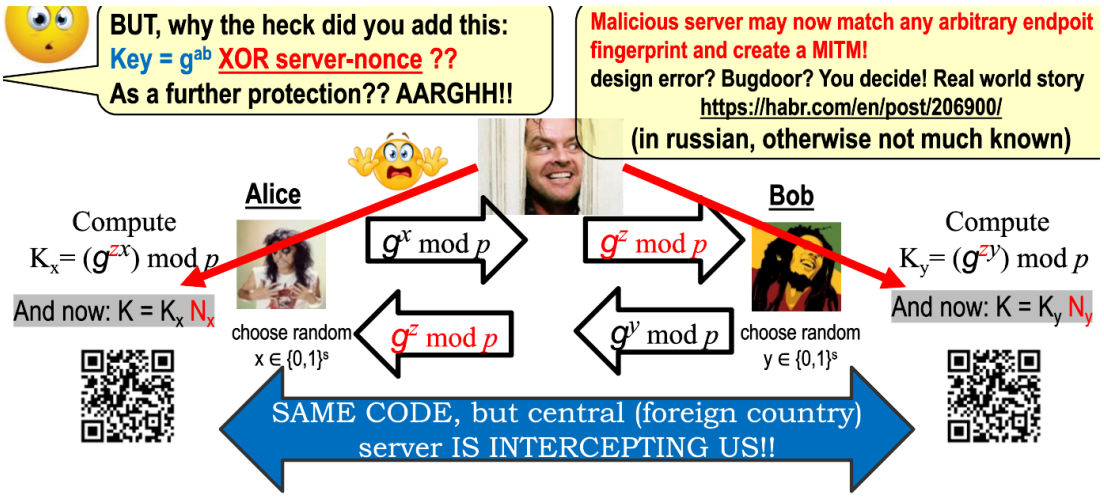
Vanilla DH can still be subject of MITM attacks and its called Anonymous DH:



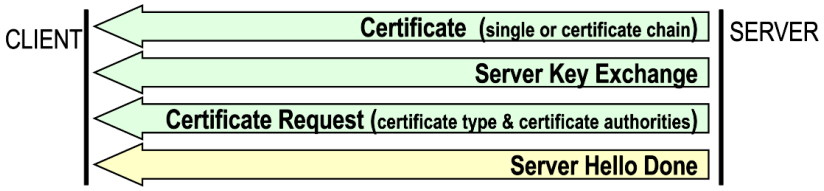
To fix this Telegram in 2013 made sure for both ends to be sent an Hash of the key K so they can compare it with the Hash of the same key K they have. The hash will be a qr code that needs to be scanned.

To make it even more secure to make the Key a Master Key it will be XORed with a truly random NONCE, but this nonce must be the same even for Alice and for Bob.

A malicious server now can do a MITM attack, intercept the g^x and g^y and sends g^z mod p to both ends. Now the server can give Alice the nounce N_x to xor it with K_x and Bob the nonce N_y to xor it with K_y in order to generate the same key K so they will have the same hash.



Let's get back to TLS Handshake
Handshake phase 2



The server sends the Certificate, the Server Key Exchange a Certificate request and the Server Hello Done. To protect this communication what can be done is to use DH with three basic approaches: Anonymous, Fixed and Ephemeral.

Anonymous (basic) DH
X, Y generated on the fly and not authenticated, trivial MITM attack

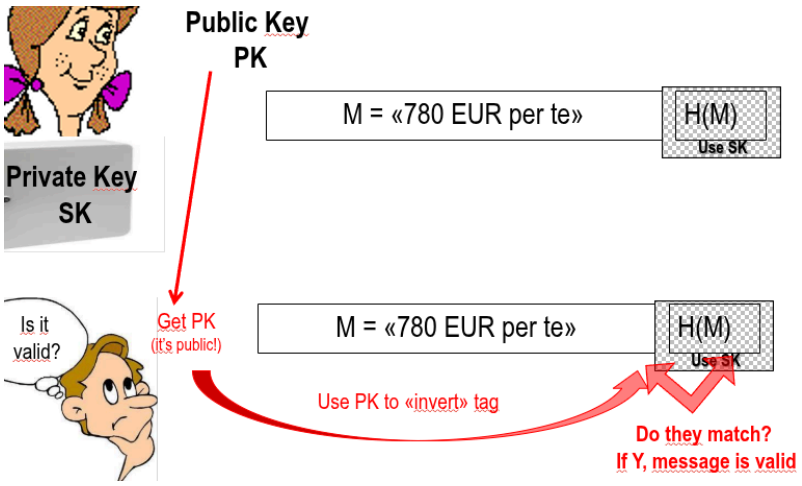
Digital Certificates
The main role is to bind a public key to a subject, the digital signature by a trusted third part, a Certification Authority that guarantees integrity of the binding.
In a MITM attack that uses a certificate, the Diffie–Hellman exchange is prevented because you can't modify the key, if you change it, the CA will no longer match. But using this method means you end up using the same key every time; otherwise, the CA wouldn't correspond.

Fixed DH
DH public parameters gx and gy are static and signed by a certification authority, no MITM anymore but they are long-lived consequence to brute-force attack.

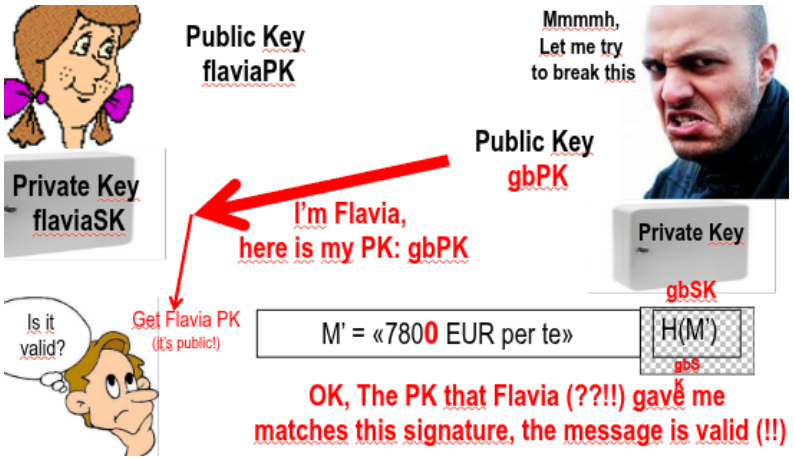
Ephemeral DH
DH parameters can vary but unlike anonymous DH, they are SIGNED by the party, hence parties must have an RSA or DSS secret key for signature purposes.

The Anon and Ephemeral DH is transmitted in the ServerKeyExchange.

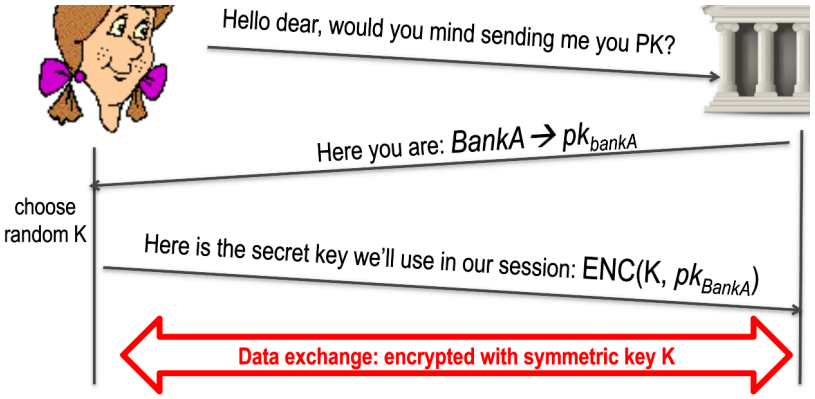
Digital Certificates and Public Key Infrastructure
The problem that Digital Certificate address is impersonation problem



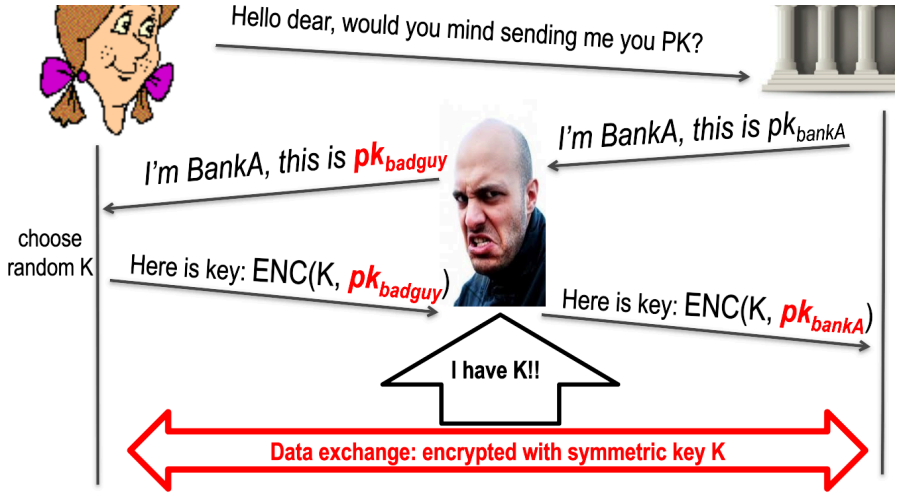
What an attacker can do is to change the message and use his own Private Key to Hash the message. The attacker now impersonates the real person and gives his Public Key to the .



RSA Key Transport

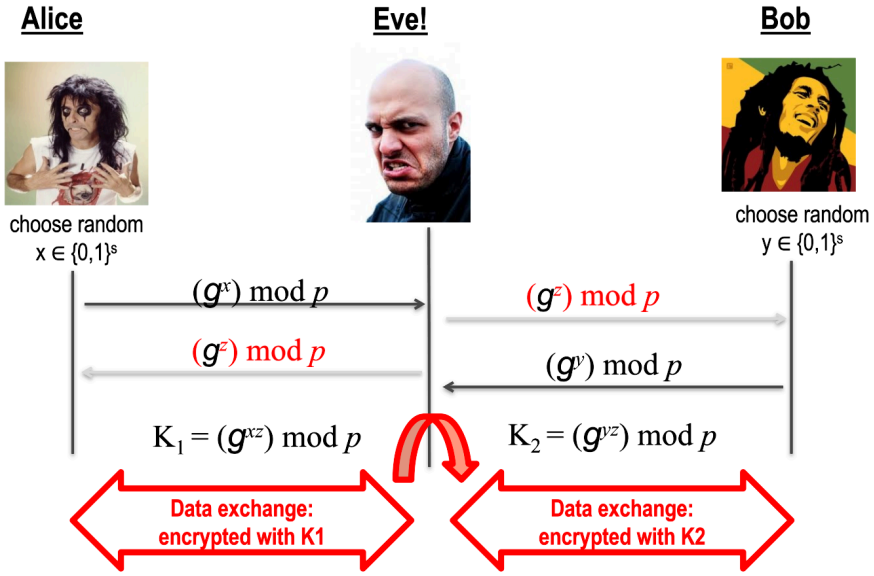


With this transport a MITM attack can be done



What the attacker do is intercept the response of the bank and respond with his key, and then the person sends his own key encrypted with the key of the bad guy. In this way the bad guy has the key K.

Using DH we have the same MITM problem



We have the same problem in all the scenarios so we need to cryptographically bind a public key to an identity, the solution is Digital Certificates.

Digital Certificates
The main role is to bind a public key to a subject, the digital signature by a trusted third part, a Certification Authority that guarantees integrity of the binding.
In a MITM attack that uses a certificate, the Diffie–Hellman exchange is prevented because you can't modify the key, if you change it, the CA will no longer match. But using this method means you end up using the same key every time; otherwise, the CA wouldn't correspond.

Let's suppose i send a message with my Identity and my Public Key, this is guaranteed by a CA, but now this CA must be verified by a CA in its turn and so on creating a chain. At the

beginning of the chain we have the Root CA, these are CA that are configured in every device. Or the user X knows all the public Keys, not a scalable solution, or he uses a transitivity trust model, so u dont need to have every possible CA but u just need that a CA is signed by a CA that u trust.

Issuing a Certificate

Verifying Certificate validity

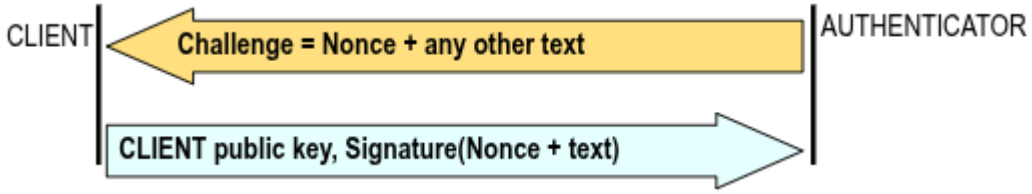
Now the Bank has the CERT and sends me the name and PK (BankName, BankPK)$_{CAsign}$ to the client. If the CA is in the trusted one and its correct then ok but we are not still sure that we are talking to the bank, cause the CERT can be sent by an attacker.

So we need to verify that we are talking to the bank, so we ask the bank to prove possession of the private key SK associated to the cert, and there are two ways to do this:

1) Ask bank to sign something fresh
2) Ask bank to decrypt something fresh

TLS uses Both.

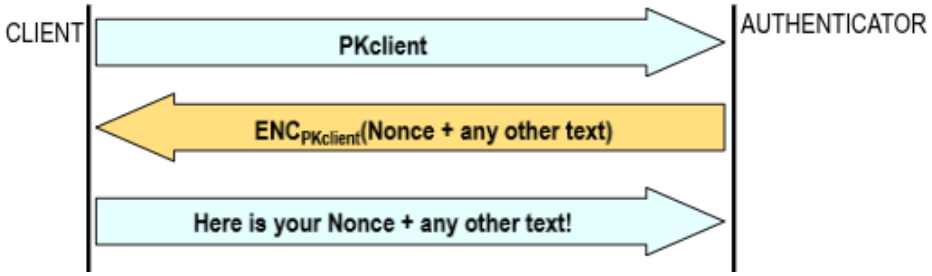
Use digital signature



The client sends the PK with the Signature of the nonce (Signature is done with SK).

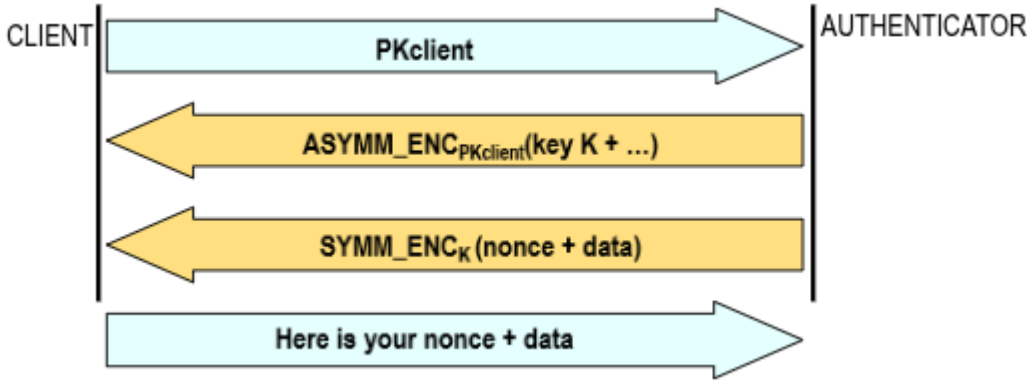
I can sign only if I own the private key corresponding to a given public key, PubKey must be bound to client.

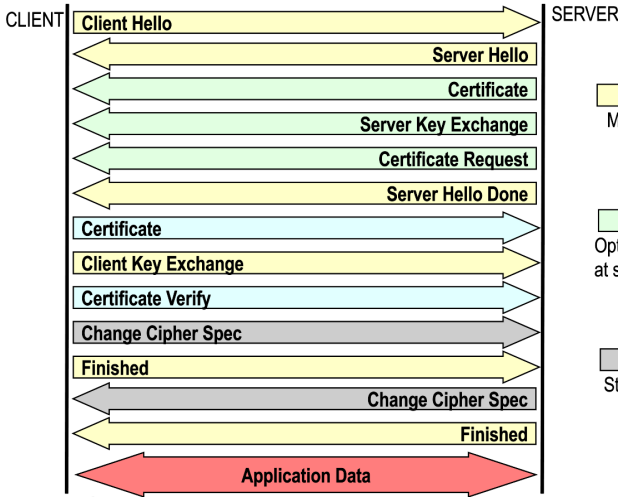
Use encryption



The PK is sent to the Authenticator that sends and he sends back the Nonce encrypted by the PK, the Client now must sends back the Nonce to the Auth to prove that he knows the SK.
I can decrypt only if I own the private key corresponding to the public key you used to encrypt.

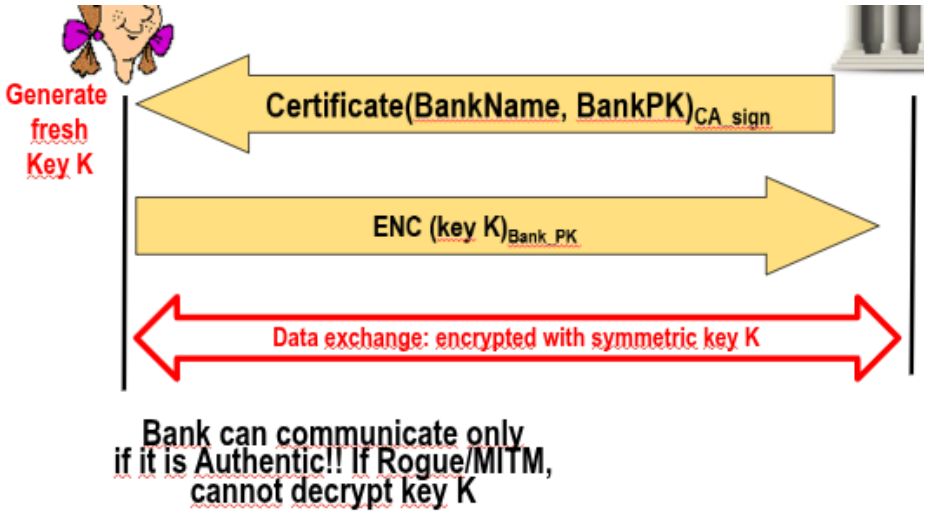
Prove knowledge of transferred key





In the Handshake (seen before) “Certificate Verify” means sign something fresh using the random bytes present in “Server Hello” as nonce. In normal TLS the client may not even be authenticated
We use signature on the side of the client and encryption on the side of the server.

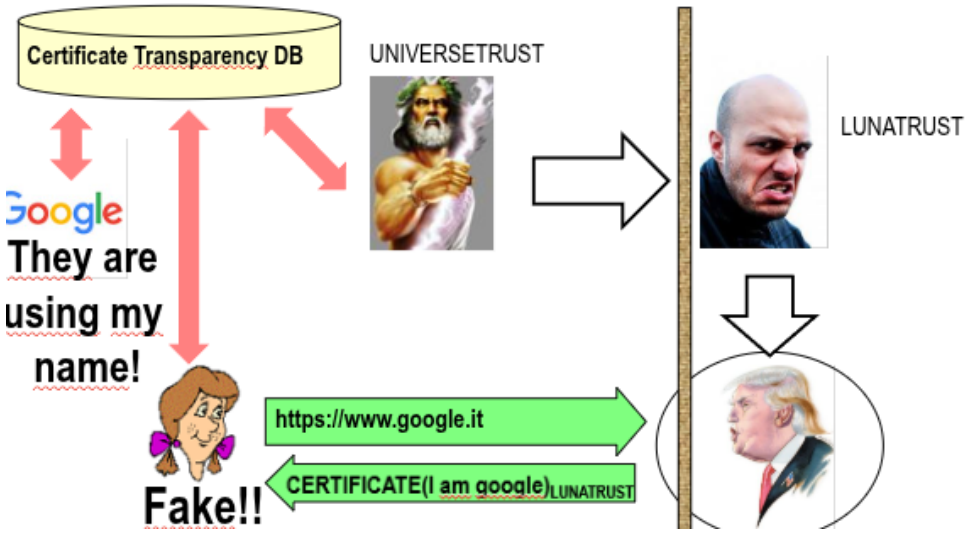
Practical TLS approach (when RSA Key Transport used)



Public Key Infrastructure (brief intro)
A PKI specifies protocols, policies, and technical mechanisms needed to support exchange of public keys. A PKI requires: standard format for certificates, relation among CAs and with end users, Policies for issuing and revoking certificates and directory services.

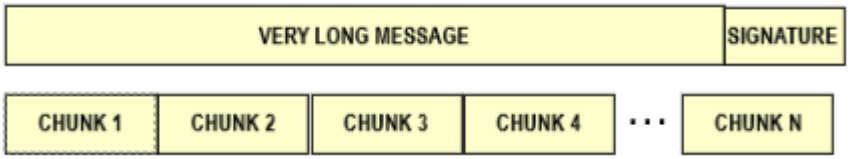
The failure of PKI: Certificate Transparency
A very serious problems are fake certificates, the web security pillar is based on the fact that CA are trusted but there are various Fake valid Certificates released for major sites and Compromised CAs, in consequence the security of PKI is getting weaker and weaker.

To cope with malicious CAs Google created a gigantic worldwide DB which anyone can check: the Certificate Transparency DB.

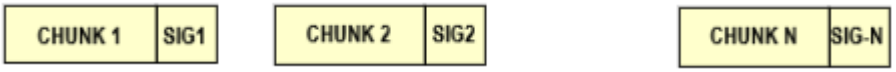


This dosen't guarantee that what is stored is correct but let's everyone check autonoma
Doing that now everyone can independently if a Certificate is valid or not.
To implement this DB is used the Merkle Trees.

To secure a file u make a fingerprint of the file (from a large file to a small fingerprint) via a cryptographic hash function and secure the fingerprint by copying it in a secure storage or by digitally signing the hash function.

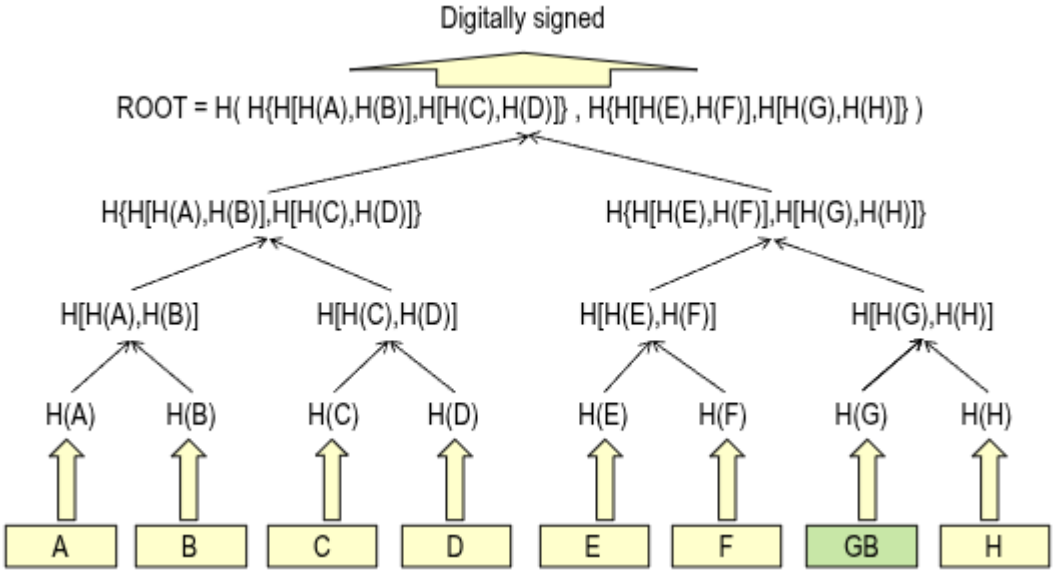


Very long messages are divided into independent chunks and frequently received out of order and from different peers so for the signature verification needs to wait until the complete message reconstruciton.



We can think of doing pre-chunk signature but doing that there is a computational and storage overhead, and also is prone to strip-type attacks to don't make u see part of the DB.

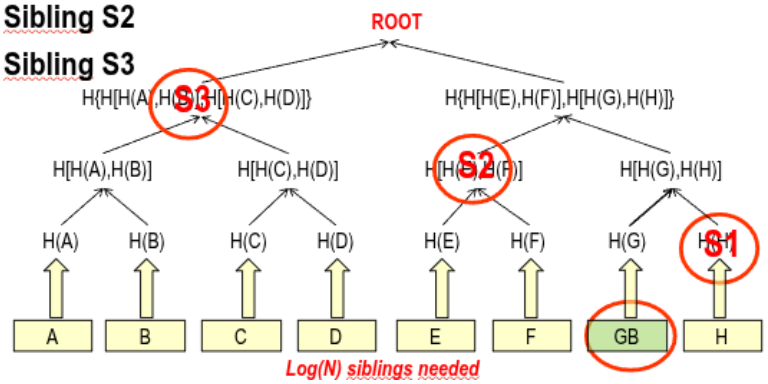
Thanks to the merkle trees we can verify both the signature of the whole DB and of the single chunks.



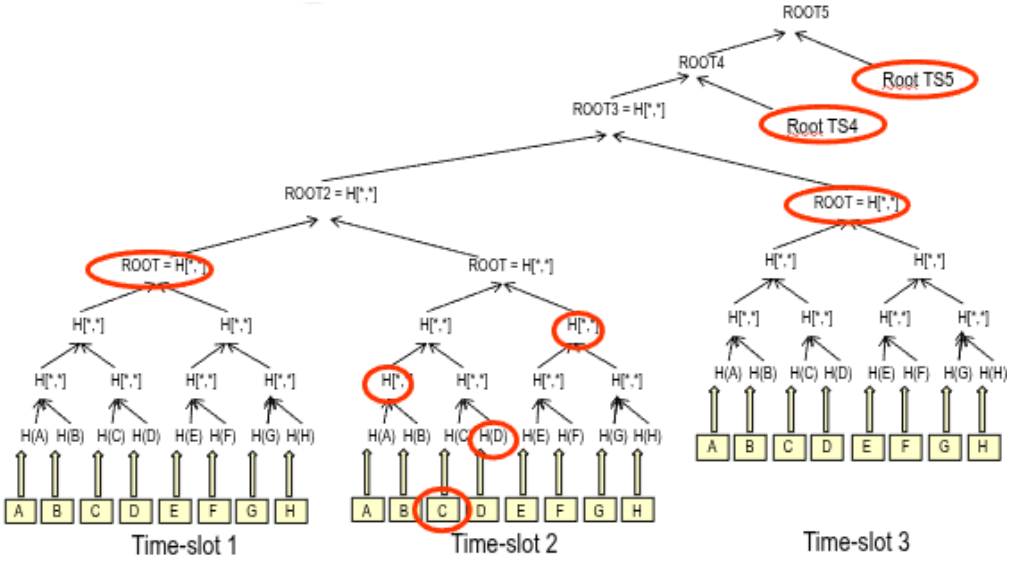
To verify the single chunk (GB) we can calculate the Root to check authenticity by using only Logn siblings, doing that i will download not the entire database but only the part and the

siblings i need:

- **Cert GB**
- **Sibling S1**
- **Sibling S2**
- **Sibling S3**

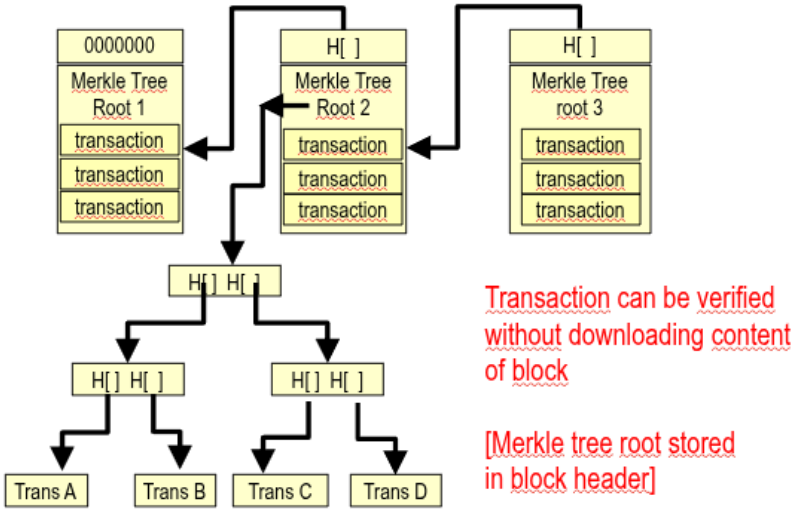


Merkle's tree can be extended w. time



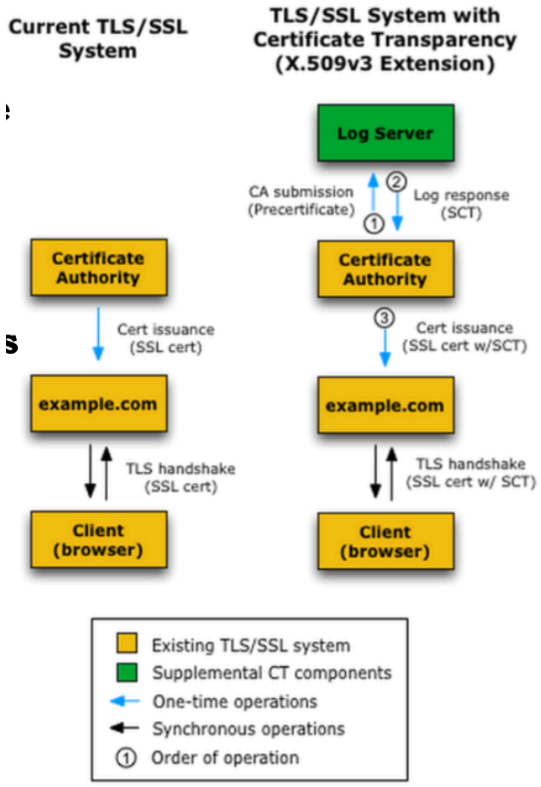
The validation now needs: the siblings for your tree and the root of the previous and of the next timeslots.

Merkle trees and blockchains



The Merkle Tree are used in efficient storage in blockchains bitcoin in ethereum and in many parts of many blockchain, google's certificate transparency, file chunk validation and many more.

Certificate Transparency
Before and After:

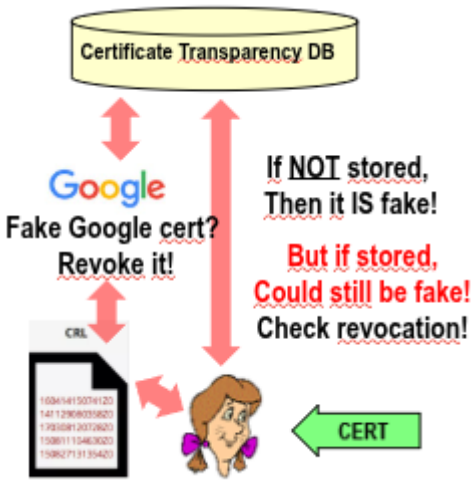


The CA sends it's Certificate to the Certificate Transparency DB that responds with the coordinates of your Certificate (time + siblings) once the CA have that part it sends it to the site.

The Certificate Transparency was a Google initiative started experimental in 2013 and endorsed and supported by IETF, supported also by major sites and integrated in major browsers.

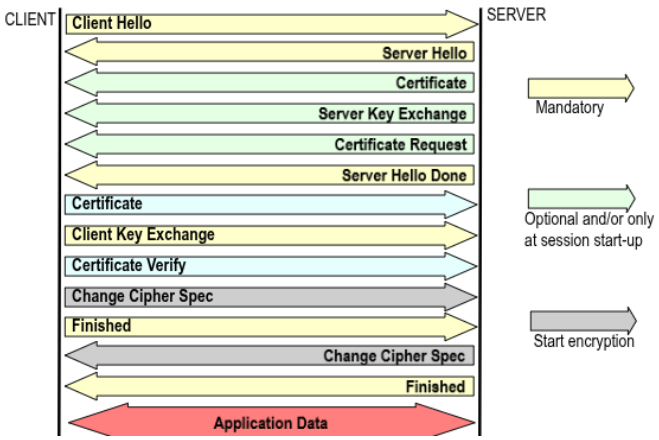
Cert Trans and Blockchain look alike, identical architecture as Bitcoin ledger, and it could be distributed as well, but isn't a blockchain, it doesn't necessarily contain only VALID certificates, it can also contain expired certificates, that's why security comes from

transparency.



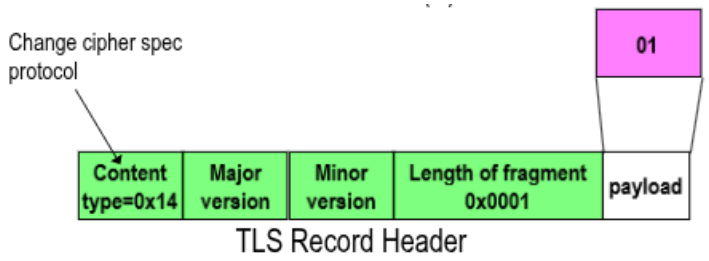
With Certificate Transparency we make it almost impossible for a CA to issue a SSL certificate for a domain without the certificate being visible to the owner of that domain by putting all certificates in a gigantic public list that everyone can access and consult and make sure that we all see the same list.

Going back to the Handshake msg

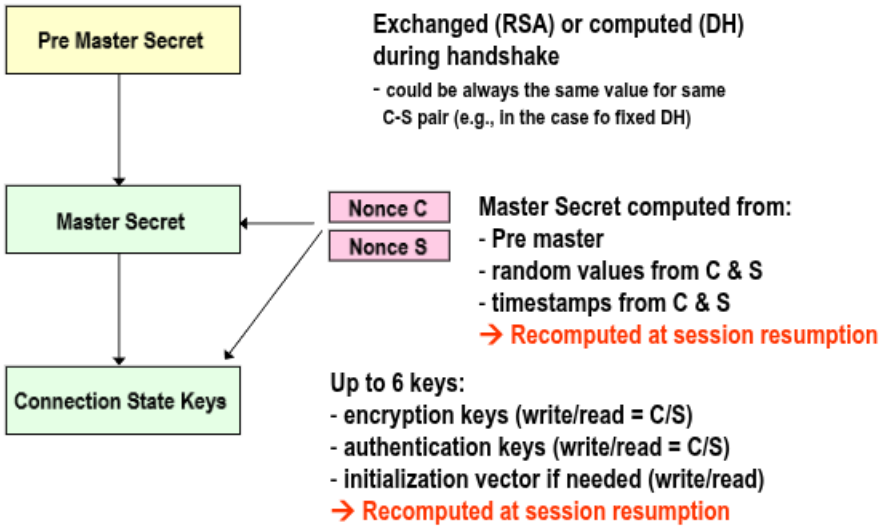


If we use RSA Key transport we only need the Certificate from the server, if we use the Fixed DH we only need the Certificate, if we use the Anonymous DH then we need the Server Key Exchange, and with Ephemeral DH we use both.

Change Cipher Spec
It is defined as a separate protocol (obsolete now) with only one msg: 1 single bit (or encrypted or not)

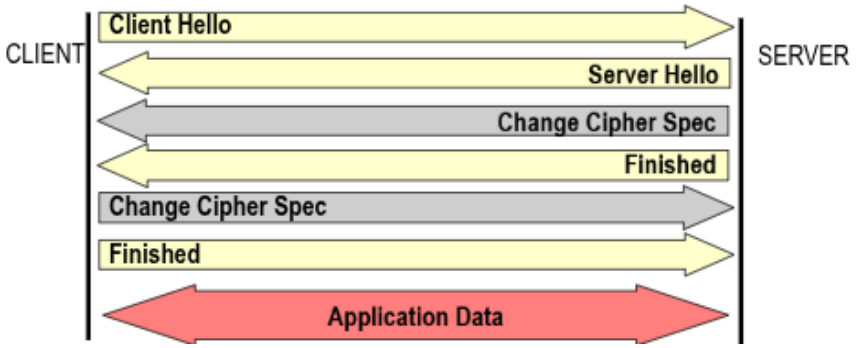


TLS Key Computation



The Connection State Keys are up to 6 and created thanks to the expanded method.

Abbreviated handshake (3-way)
Session resumption: Let's reuse the Pre Master Secret generated before and regenerate only the Master Secret.

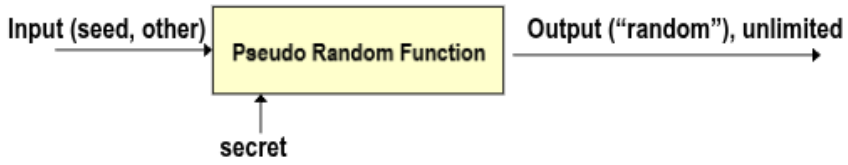


By resending the client hello and the server hello we have the two nounce needed to regenerate the Master Key and in consequence the next Connection State Keys.

Initial keying, extract then expand
1) Extract: Add randomness in key generation
2) Expand: initially limited secret to needed amount of crypto material

Extract then Expand paradigm
Extract: We want that the key is equiprobable, if for eg we use DH g^{xy} mod 7919, the key will never be 8000, a fixed non-secret string chosen at random.
Expand we want to use a secure PRF (pseudo random function) key to expand k.
The construction rely on the assumption that key k is random this means uniform distribution and that means that we cant use the Fixed DH where the key is the same, so the extraction guarantees random master key that is salted with nonces.

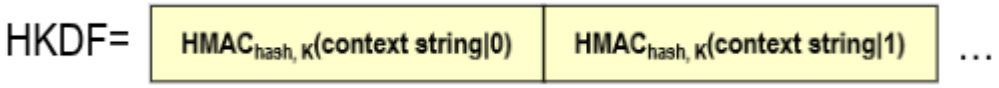
Basic building block: PRF



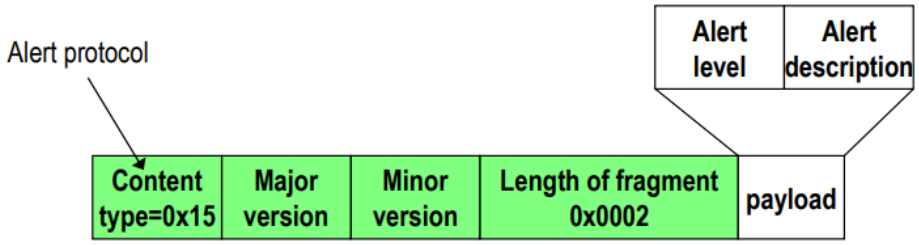
TLS1.1 uses a PRF constructed using MD5 and SHA-1, the first broken and the second very weak. In TLS1.2 we have a negotiable PRF with the default one that uses SHA256.

So the key generation its done in this way:
Master Key (48 bytes):
PRF(pre-master-secret, "master secret", Client-Random | Server-Random)
Key Block (size depends):
PRF(master-key, "key expansion", Server-Random | Client-Random)
Individual Keys:
Partition the key block into up to 6 fields in order: Client MAC, Server MAC, Client key, Server Key, Client IV, Server IV.
PRF used also in computation of finished msg instead of normal MD5 or SHA1.

PRF in TLS: HKDF
HMAC-based Key Derivation Function, provably secure and standardized now named KDF its included in the newer version of TLS



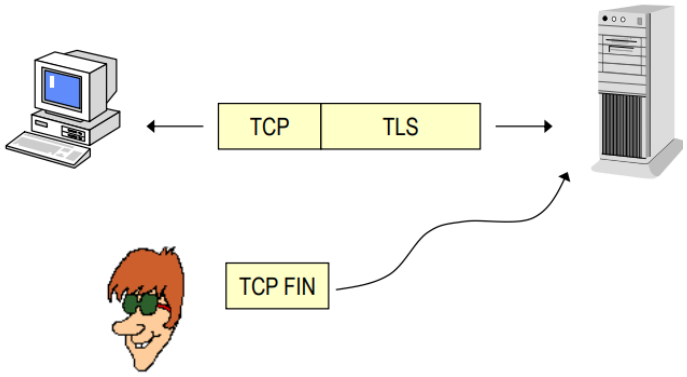
TLS connection management
Alert Protocol define special messages to convey "alert" information between the involved fields, those are encapsulated encrypted authenticated into TLS Records. Its formed on 2 bytes: the alert level and the alert description.



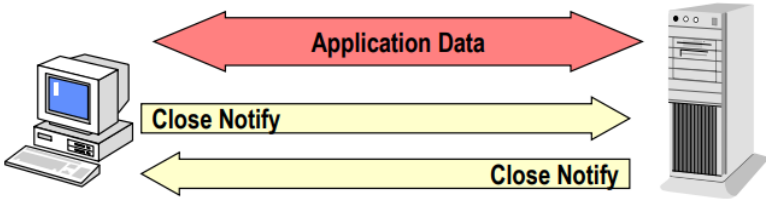
There are 23 possible alerts some of them are:
unexpected_message, bad_record_mac, record_overflow, handshake_failure, bad_certificate, and so on. If fatal, terminate and do not allow resumption with same security parameters.

TLS dosen't protect TCP (IP | TCP | Payload, TLS only protects the payload).
It's open to DoS attacks, anyone can kill a TCP connection, to protect it we can do TCP over TLS over TCP (use as last resort) or TCP over TCP tunnels has performance impairment, best to use DTLS tunnel.

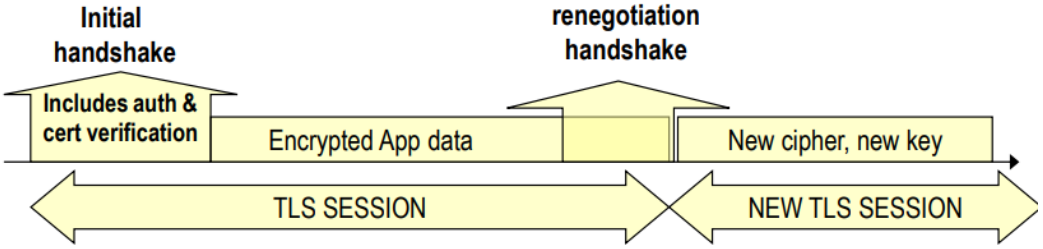
Truncation attack



An attacker may end connection any time by sending a TCP FIN.
The solution is to notify the closure of the connection using an Alert, by both ends. This will informs that no more data will be transmitted. A connection that ends abruptly without the close notify may be a truncation attack.

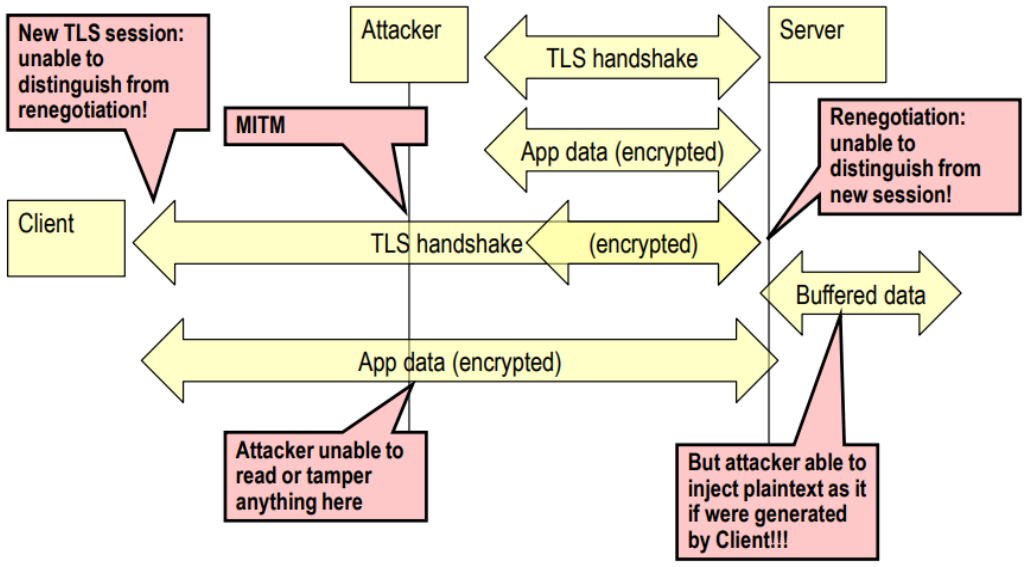


Renegotiation (re-handshake)
Not limited to rekeying and sent encrypted via previously established ciphersuite (its sent in the Encryption App data) with the purpose of increasing the security level, authenticating client and other things. Renegotiation generates a new session ID.



Besides the fact that it is encrypted, renegotiation handshake otherwise isn't distinguishable from the initial handshake. In security is best to keep things simple.

Renegotiation attack (plaintext injection attack)
Discovered by Marsh Ray in 2009 used in stealing attack against twitter users.



The attacker opens a TLS connection with the Server and send some App data encrypted in the meantime the attacker is a MITM between C/S. When the client wants to open a TLS connection it will be tunnelled in the attacker TLS previous connection because the Client TLS connections is undistinguishable from renegotiation. All the data sent by the server is encrypted and only the C/S can read it and the attacker can do nothing about that, but what he can do is being able to inject plaintext as it if were generated by Client.

With prefix data injection the attacker can sends query like:
GET /pizza?toppings=pepperoni;address=attacker_address HTTP/1.1
X-Ignore-This: (no carriage return)
The victim now sends
GET /pizza?toppings=sausage;address=victim_address HTTP/1.1
Cookie: victim_cookie
The result are is:
GET /pizza?toppings=pepperoni;address=attacker_address HTTP/1.1
X-Ignore-This: GET /pizza?toppings=sausage;address=victim_address HTTP/1.1
Cookie: victim_cookie
So the user victim's account sends a pizza to the attacker and dosen't receive anything.

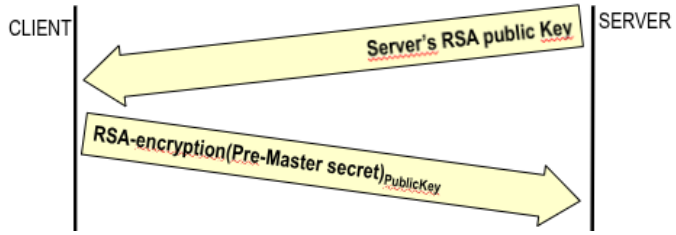
Thanks to this type of attack the Server can also disclose information generated by the client upon request.

Preventing TLS renegotiation attacks
The immediate solution is to disable it but it was standardized to patch the renegotiation, the idea is that the Server needs to recognize the renegotiations. Was forbidden in TLS1.3.

DTLS vs TLS
DTLS is TLS over UDP and the design goal is to be as similar to TLS. TLS assumes orderly delivery while DTLS sequence number explicitly added in record header, TLS assumes reliable delivery timeouts while DTLS added timeouts for manage datagram loss, TLS may

generate large fragments while DTLS includes fragmentation capabilities to fit into single UDP datagram and recommends Path MTU discovery, TLS assumes connection oriented protocol while DTLS connection = (TLS handshake - TLS closure Alert).

RSA key transport



We remember that RSA is based on factoring a product of two large primes so given p and q large primes compute N = pq is easy but given N, find the factors p and q is hard (crivello di erastotene calcola tutti i numeri primi fino a n).

The principle behind RSA
m^x mod N is a periodic function

$$3^x \bmod 10 = \{3,9,7,1,3,9,7,1,\dots\}$$
$$7^x \bmod 10 = \{7,9,3,1,7,9,3,1,\dots\}$$
$$9^x \bmod 10 = \{9,1,9,1,9,1,9,1,\dots\}$$

Euler Thm (Euler-Fermat) is a explicit rule to compute the maximum period of m^x mod N. If m coprime with N GCD(m,N) = 1 then $m^{\varphi(N)}$ mod N = 1

p prime → $\Phi(p) = p-1$

$$p = 7, \Phi(7) = 6 \rightarrow 3^6 \bmod 7 = 729 \bmod 7 = 1$$

p,q primes → $\Phi(p \cdot q) = \Phi(p) \cdot \Phi(q) = (p-1)(q-1)$ **RSA case**

$$N = 10, \Phi(10) = 4 \rightarrow 3^4 \bmod 10 = 81 \bmod 10 = 1$$

Prime p^k → $\Phi(p^k) = \Phi(p) \cdot p^{k-1} = (p-1)\,p^{k-1}$

$$N = 25, \Phi(25) = \Phi(5^2) = 4 \cdot 5$$
$$\rightarrow 3^{20} \bmod 25 = 3486784401 \bmod 25 = 1$$

general case: $N = p_1^{k_1} \cdot \dots \cdot p_z^{k_z}$

$$\Phi(N) = (p_1 - 1)p_1^{k_1 - 1} \cdot \dots \cdot (p_z - 1)p_z^{k_z - 1}$$

$$N = 100, \Phi(100) = \Phi(5^2\,2^2) = 4 \cdot 5 \cdot 1 \cdot 2$$
$$\rightarrow 3^{40} \bmod 100 = 12157665459056928801 \bmod 100 = 1$$

RSA Construction
Generate two LARGE primes p, q (must remain secret)
Compute RSA module: N = pq (can be made public)
Compute φ(N) = (p-1)(q-1) (must remain secret)
Generate public key 1<*e*<φ(N) with *e* coprime with φ(N)
Generate private key *d* such that *ed* = 1 mod φ(N)

Security assumption: given N, must be hard to find factors p,q hence hard to find φ(N) and without φ(N) hard to compute d from e.
The public key in RSA is (N,e) while the private key is d.

Encrypt: ENC(M): $C = M^e \bmod N$

Decrypt: DEC(C):
$$M = C^d \bmod N =$$
$$= (M^e)^d \bmod N =$$
$$= M^{ed} \bmod N =$$
$$= M^1 \bmod N = M$$

Consequence of periodicity

$m^x \bmod N$ is a periodic function **with period $\Phi(N)$**

$2^x \bmod 11 = \{2,4,8,5,10,9,7,3,6,1\}$ $\quad \Phi(11)=10$

Compute 9·7 mod 11=?

Using multiplications mod 11 → 9·7 mod 11= 63 mod 11 = 8
Alternative approach: since $9=2^6$ mod 11 and $7=2^7$ mod 11:
$9 \cdot 7 \bmod 11 = 2^6 \cdot 2^7 \bmod 11 = 2^{6+7} \bmod 11 = 2^{13} \bmod 11 = \ldots$
$\ldots = 2^{10+3} \bmod 11 = 2^3 \bmod 11 = 2^{13 \bmod 10} \bmod 11$
→ arithmetic @ exponent = modulo Φ(N), not N!!!

Consequence: $m^x = m \bmod N$ if $x = 1 \bmod \Phi(N)$

While dealing with exponents in modular arithmetic
a^x modN = $a^{(x \bmod \varphi(N))}$ modN if gcd(a,N)=1

RSA works thanks to the Trapdoor function
Given N, e, encrypted msg m^e computing the description key x is normally t.c. $(m^e)^x$ mod N = m is normally a hard problem (exponential complexity, brute force on all possible x) but becomes easy when u know φ(N) because ex mod φ(N) can be solved in polynomial time, $x = e^{-1}$ mod φ(N)

Extended Euclidean algo

➔GCD[51,11]=1 coprime

➔Find a,b s.t. 51 a + 11 b = 1

a	b	val	rem	
1	0	51		
0	1	11	4	Subtract 4x
1	-4	7	1	Subtract 1x
-1	5	4	1	Subtract 1x
2	-9	3	1	Subtract 1x
-3	14	1	=	

51x(-3)+11x(14)=1

How to compute RSA inverse

➔Public key: e

⇨*Example: e=13*

➔RSA modulus: 77=11×7

⇨$\Phi(77)=10\times6=60$

⇨*GCD(e, Φ)=1* (coprime, OK)

➔Find a,b such that Φ a + e b = 1

⇨*ExtendedGCD[60,13]={1,{5,-23}}*

⇨**60 × 5 + 13 × (-23)** = *1* → **13 × (-23)** = *1* - **60 × 5**

⇨*Mod[***13 × (-23)***, 60] = Mod[1 -* **60 × 5***, 60] = 1*

➔Hence 13^{-1} mod 60 = -23 = 37

RSA recap:

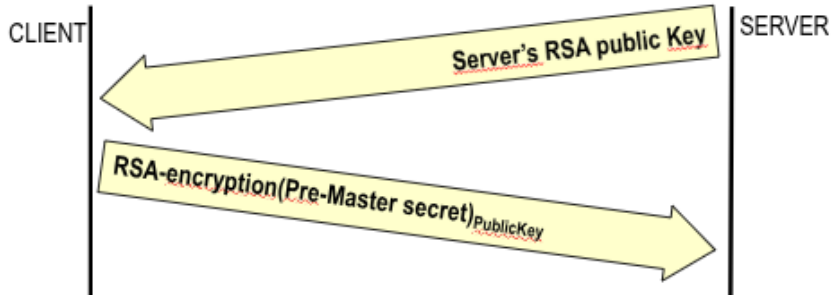
p = 11, q = 17 (*secret*)
N = p×q = 11×17 = 187 (*public*)
Φ = (p−1)×(q−1) = 10×16 = 160 (*secret*)

e = 7 (public, prime wrt 160)
d = 7^{-1} mod 160 = 23 (secret)
- Note: 23×7 ≡ 161 = 160+1 = 1 (mod 160)

Public Key Encryption: C=M^7 mod 187
- Decrypt: $(M^7)^{23}$ mod 187= M^{7x23} mod 187= M

Digital Signature: send M, TAG=$H(M)^{23}$ mod 187
- Verify sign: TAG^7 mod 187= $(H(M)^{23})^7$ mod 187= H(M)
  - check hash matches with message M provided

RSA key transport

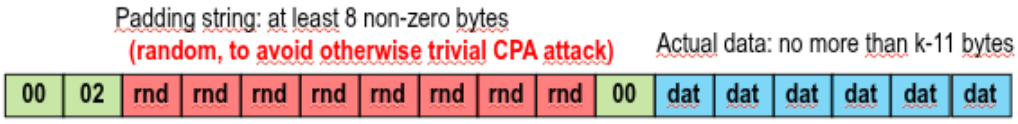


Let'see if RSA is robust against Chosen ciphertext attacks:
The ciphertext $C = M^e \bmod n$, and the Goal is ot decrypt it so $M = C^d \bmod n$, the attacker has the access to a decryption oracle (obviously cannot submit the original data).

Attack chose a random value r
Construct the new ciphertext: $X = (r^e\ C)$ and ask the oracle to decrypt X (chosen ciphertext attack).
The oracle will return the decryption of X: $X^d = (r^e\ C)^d = r^{ed}\ C^d = r\ C^d$
Attacker now cancels r and finds M: $X^d\ r^1 = r\ C^d\ r^1 = C^d = M$

The problem is that RSA is malleable
Non-malleability: given an encryption c of some message m, the attacker should NOT be able to create a different ciphertext c' that decrypts to a message m' that is somehow related to m.
RSA padding: should at least mitigate such problem.

RSA padding:
PKCS #1 = first standard for the family PKCS = Publick-Key Cryptography Standard.
Specifies RSA encryption and decryption operation, including basic RSA padding approach.
It works as follows:
The padding of course must be lower than n (cause then we do mod n), the first bytes are 00 02 that specifies the start of the padding after those bytes we have random bytes non 00, cause 00 is used as the start of the actual data.

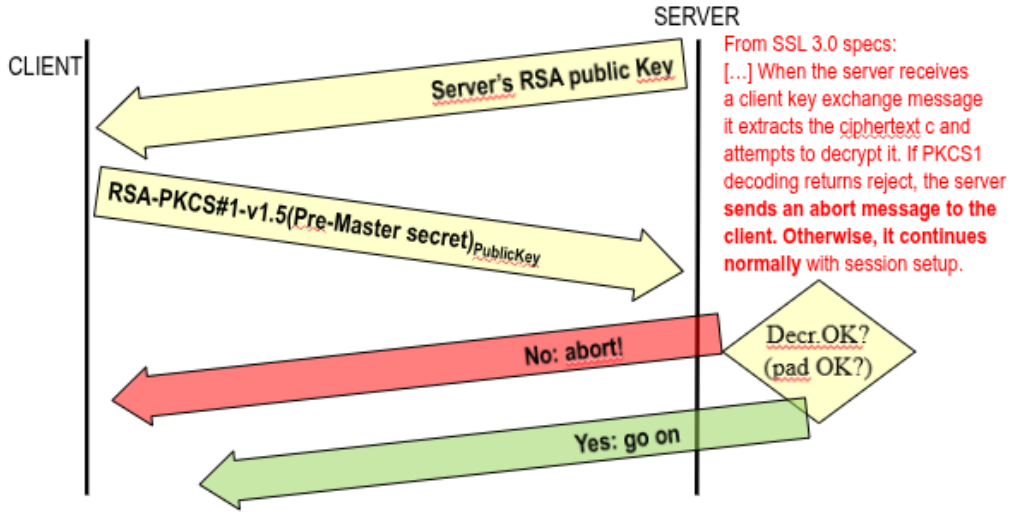


k = size of the RSA modulus in bytes – formally: k = integer value which satisfies

$$2^{8(k-1)} \leq n < 2^{8k}$$

e.g. with RSA-1024, k = 128 (in most cases) you can encrypt up to 117 bytes

RSA key transport (until SSL v3.0)



Looks like a padding oracle.

Bleichenbacher's Oracle
Target up to SSL3.0 corrected in TLS1.0 and removed TLS1.3, but still around until 2018.
Attacker's goal: decrypt ciphertext C
$C = M^e \bmod n$
Pick value r and construct new ciphertext
$C' = C\, r^e \bmod n = (M\, r)^e \bmod n$
Send to SSL3.0 server
The oracle will tell u if (M r) starts with 00 02 or not, this leaks information about M. The Bleichenbacher result tells us that with a sufficient number of attempts (order of 2^{20} with RSA-1024) u can completely decrypt M. And u can pick r based on previous results.

Bleinchenbacher example
Assume a simpler Oracle
Msg = $\log_2 n$ bits
Oracle tells if first bit is 0 or 1
Doing this we use the same concept of the bleichenbacher attack, but instead of telling us if the first 2 bytes are 00 02 it tells us if the first bit is 0 or 1.
With order of $\log_2 n$ attempts u can decrypt the whole message.

RSA parameters
p = 13, q = 19, n = 247 (8 bit)
Let's pick e = 29 and $d = e^{-1} \bmod 216 = 149$
Assume u see C = 90
Send to server:

```
Mod[c* PowerMod[2^Range[1, 8], e, n], n] =
{20, 224, 187, 69, 180, 40, 201, 127}
```

It returns (0,1,1,1,0,1,1,1)

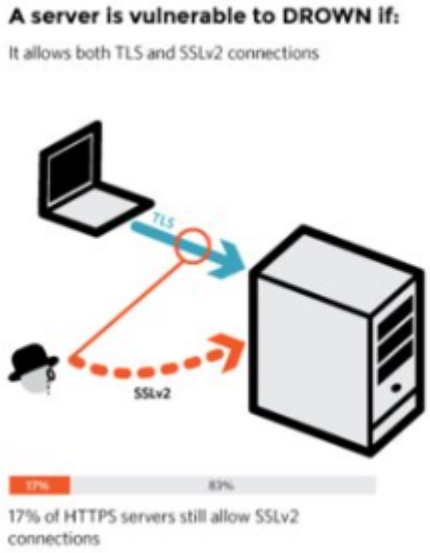
In this attack u scroll through the bits by multiplying by 2 the message in this way u shift the message by one.

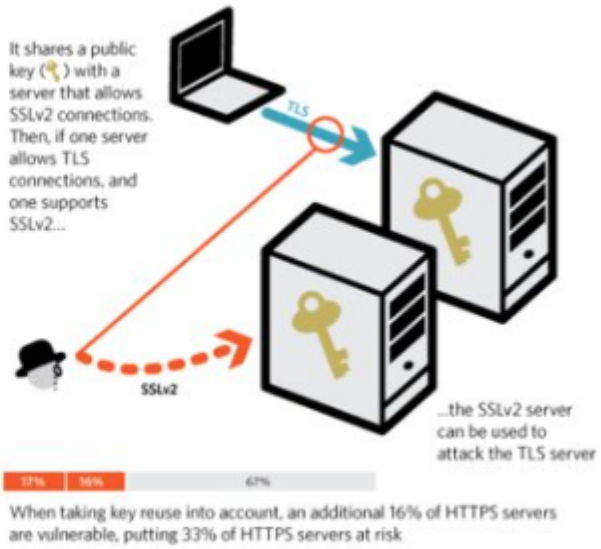
2M starts with 0 if
□M in (0, 63) or (124,187)
Of this, 4M starts with 1 if
□M in (32, 61) or (156,185)
Of this, 8M starts with 1 if
□M in (47, 61) or (171,185)

On the 8th attempt will break the last tie.
As long as one bit gets known, all the ciphertext gets known as well.

Corollary: attacking parity
Let's say a shop reject RSA-PKCS encrypted requests for an even number of items, Parity side channel = parity oracle, so we have the same situation as before: as log as just 1 bit gets revealed ALL plaintext is at stake.

DROWN attack
An important attack did in 2016





Uses the same principle of Bleinchenbacher Oracle

Countermeasures
Change RSA-PKCS#1 into a more secure padding (like RSA-OAEP), but that requires way too major changes, or we can do a careful implementation, but that's also hard, or get rid of RSA key transport, that is what TLS1.3 has done.
Implementation is very tricky and errors last for long so once a crypto error is made it's hard to incrementally patch it.

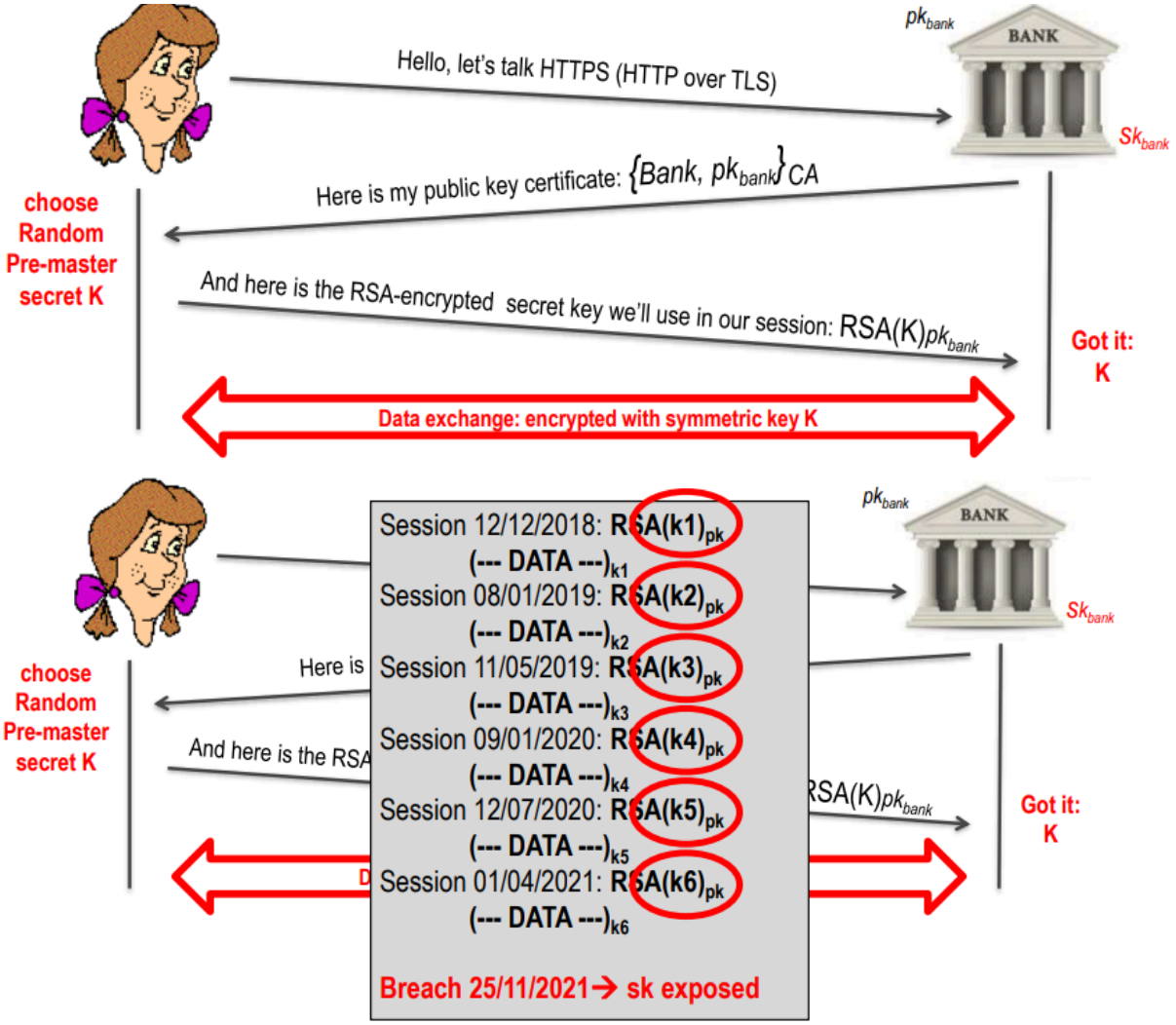
TLS1.3: Approved in 2018
They removed all symmetric algorithms broken like SHA1, MD5, AES-CBC and so on. All remaining ciphers are AEAD (Authenticated Encryption with Associated Data).
TLS1.2 has four pubkey handshake methods supported: RSA key transport, Anonymous Diffie-Hellman, Fixed Diffie-Hellman, Diffie-Hellman Ephemeral. In TLS1.3 they removed all

except Diffie-Hellman Ephemeral thanks to Forward Secrecy and also to get rid of RSA issues.

Perfect Forward Secrecy is:
If a long-term private key is compromised at some time, this shouldn't affect data delivered before nor after such time.
In essence PFS guarantees that session keys won't be compromised even if long-term secrets used in the session key exchange are compromised.

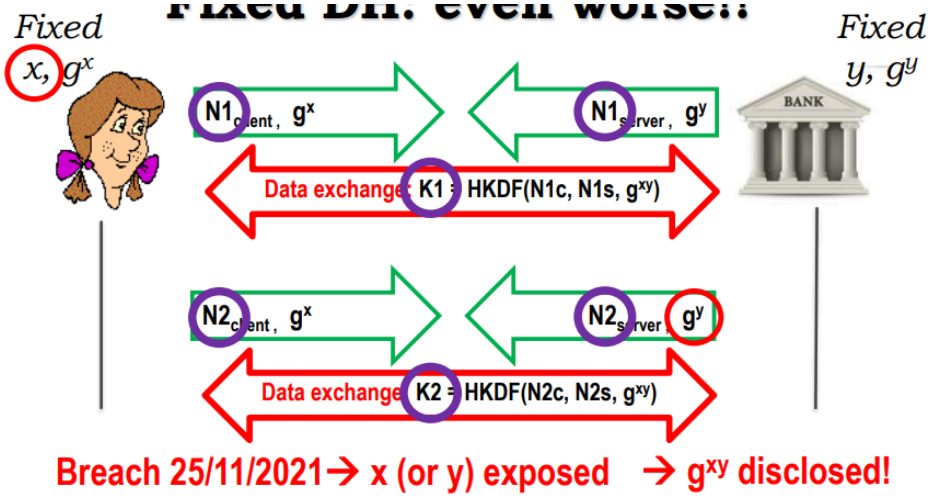
PFS is important because it protects against a more powerful attacker model capable of logging huge amounts of data and capable of occasionally break a public/private key pair.

Traditional RSA key transport:



After i discover the pk i can decrypt all the previous data.

Fixed DH: even worse

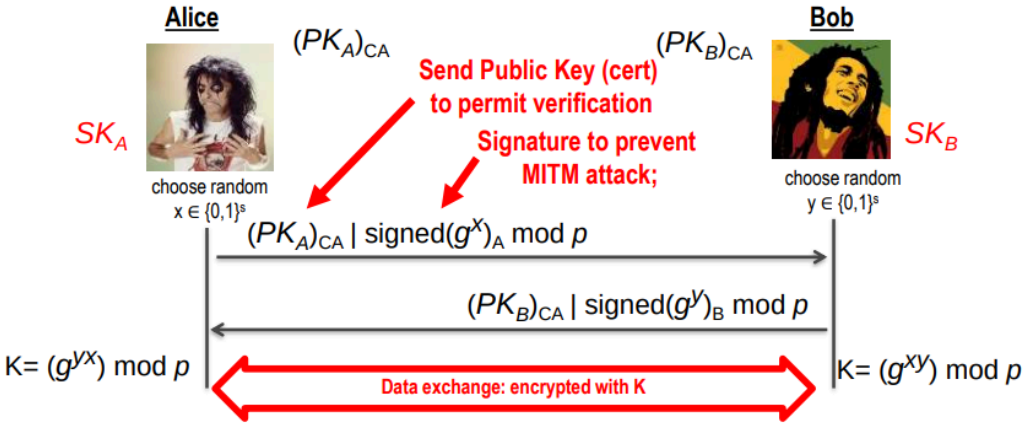


Breach 25/11/2021→ x (or y) exposed → g^{xy} disclosed!

→ compute K2 (nonces in hello msg) → And all past K!!

If x or y is exposed then g^{xy} is disclosed so we can compute all the past messages thanks to the nonces in the hello messages

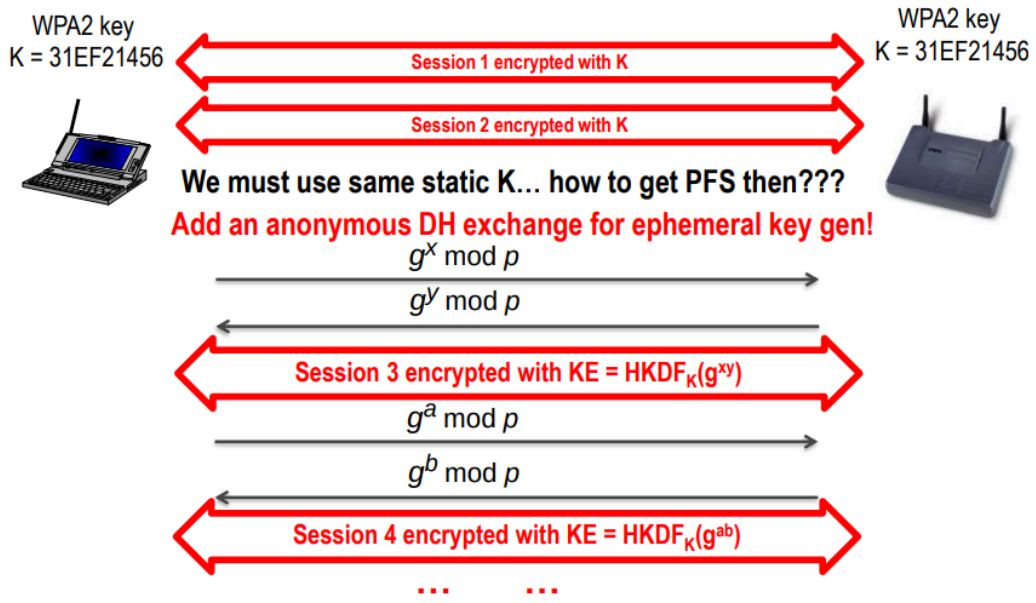
Ephemeral Diffie-Hellman: PFS



If SK breaks, (ephemeral) session keys are not revealed! Forward Secrecy!

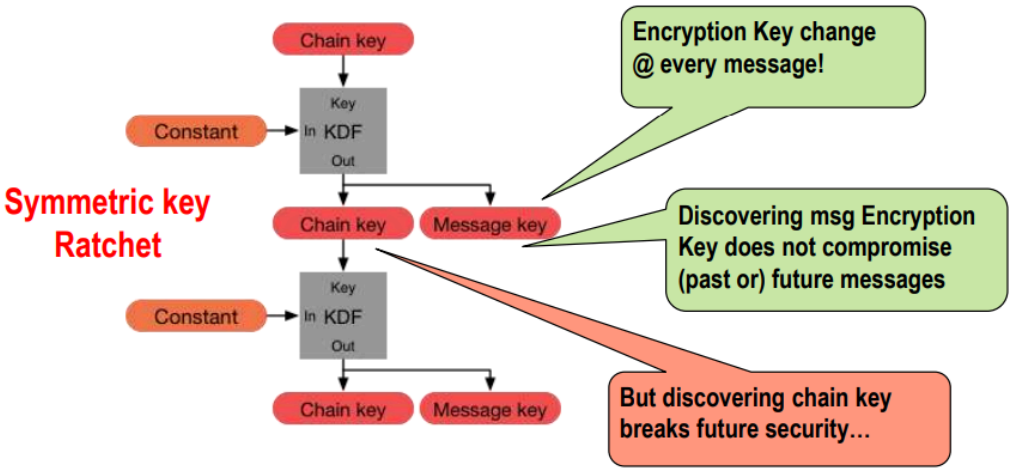
Amd even AFTER breach, if attacker if only passive (but MITM now possible)

PFS vs pre-shared key

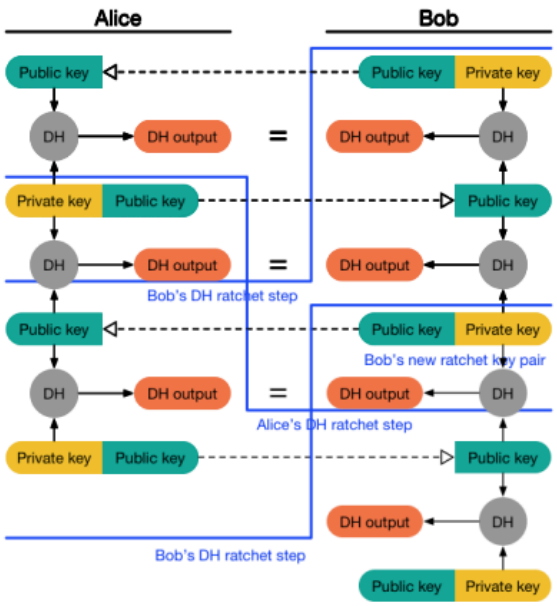


Messaging apps uses E2e (end-to-end) encryption with PFS: not only the provider cannot see, but even if it logs all data and gets access later on to your key, he cannot decrypt past messages. This is done thanks to the Double Ratchet.

Double ratchet combines a KDF chain



With a DH ratchet, to guarantee forward secrecy

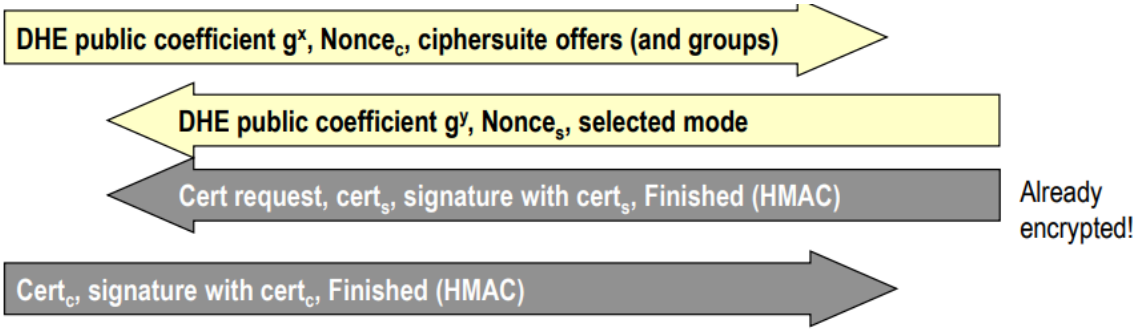


Diffie-Hellman Ratchet

Double ratchet = Sym key ratchet + DH ratchet

In the DH Ratchet, initially they perform one DH exchange.

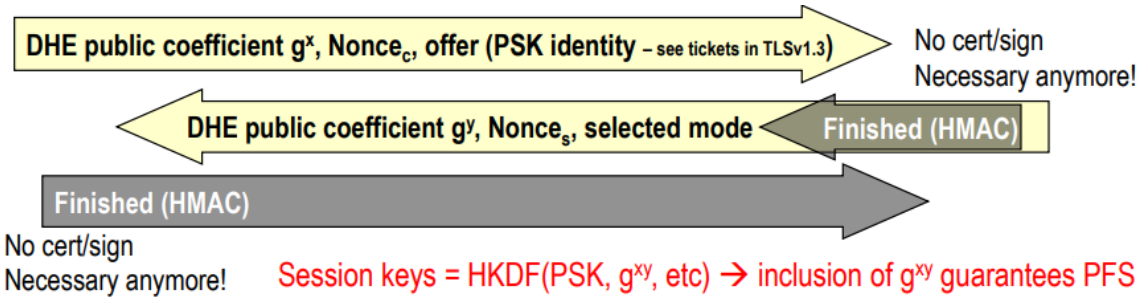
TLS1.3 handshake: faster and more secure
With only DHE left there is a 3 way handshake with PFS



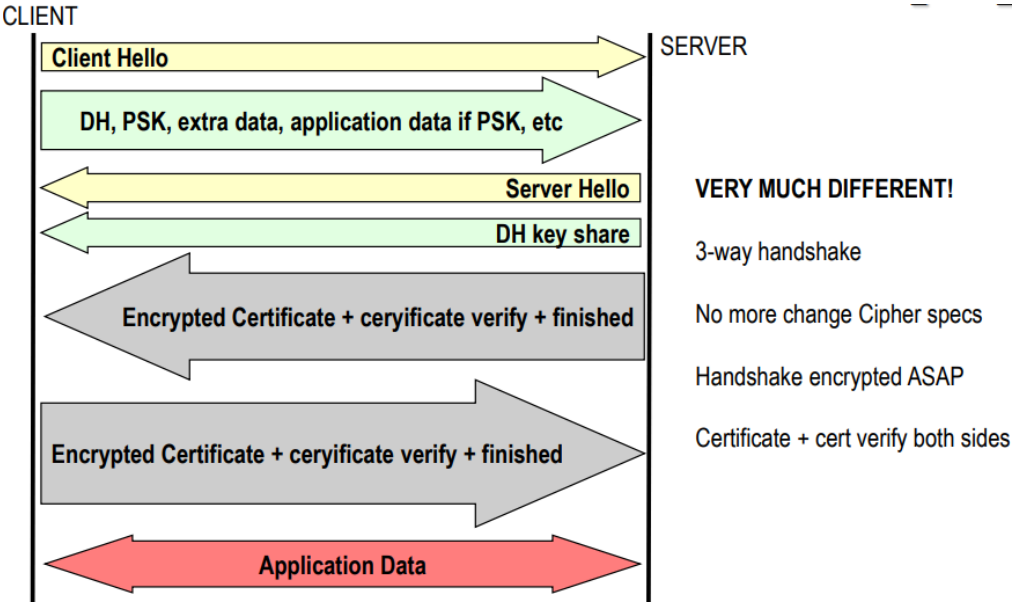
Side effect of early encryption: IDENTITY PROTECTION!
(certificates never transmitted in clear)
(not very useful in web, but keep in mind for different TLS scenarios…)

We can also use a pre–shared key, this option was added in TLS1.2 where the PSK is either an agreed offline in constrained environments or computed during the first handshake and then reused in subsequent TLS connection.
In TLS1.3 there is an optional support for forward secrecy that combines the PSK with a

fresh DHE.



The new handshake is now done in this way



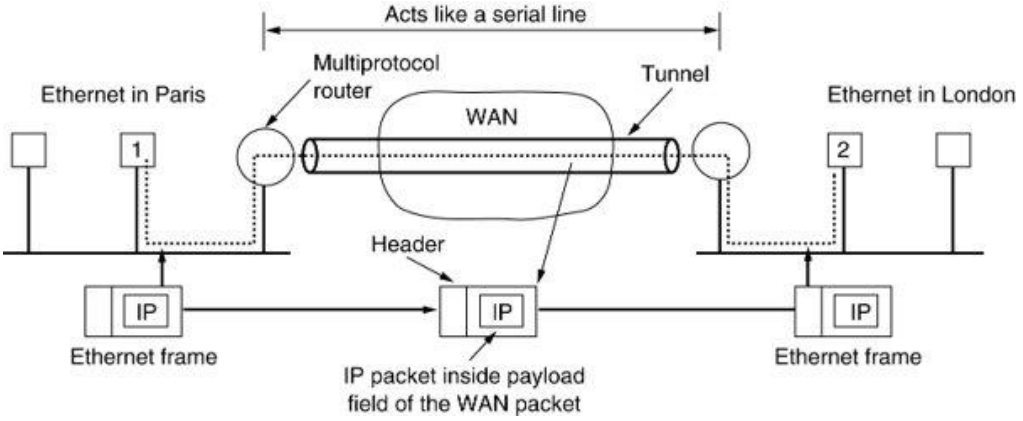
in the handshake of the 1.3 we have the 0-RTT (round trip time) data: when PSK agreed (in previous handshake), send application data directly in first message, use $HKDF_{PSK}$(Client-Hello-content) as key in this way is much faster and also very useful in application relying on short data transfer. But doing this way we have less security, there is an obvious replay attack.

The problem is that the key used in the first msg does not include server nonce so to mitigate the replay we can use a time window mitigation so include a maximum lifetime after the first session, doing this we don't solve the problem but reduce the vulnerability period. Another way to mitigate the replay attack is to control if the nonce is reused so keep a record of all client nonces.

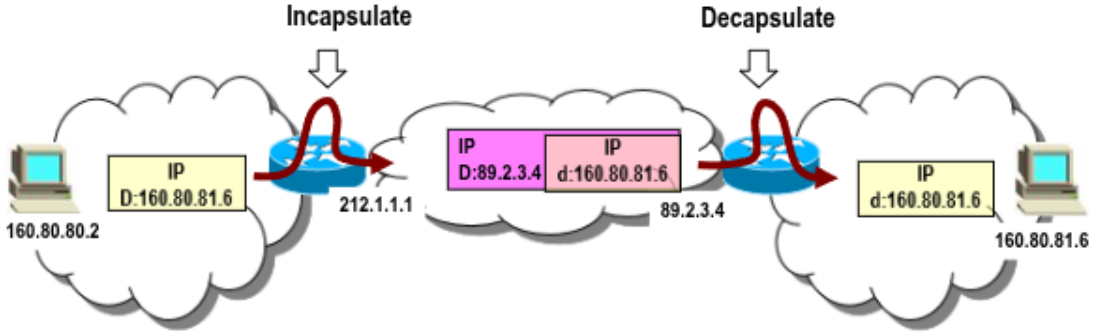
So in TLS1.3 we don't have negotiation anymore, the HKDF is officially included, there are a few simplifications in EC management, there is the removal of compression and exported key.

VPNs
How it works is: we encapsulate one packet in another packet that goes through the internet

and then decapsulate the inner packet.

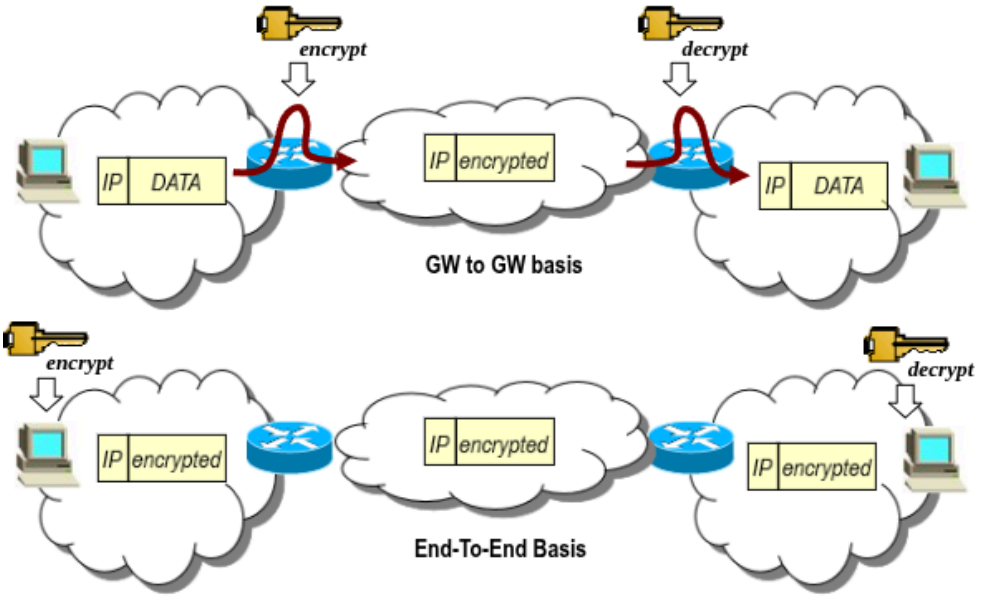


VPN main ingredient is tunnels



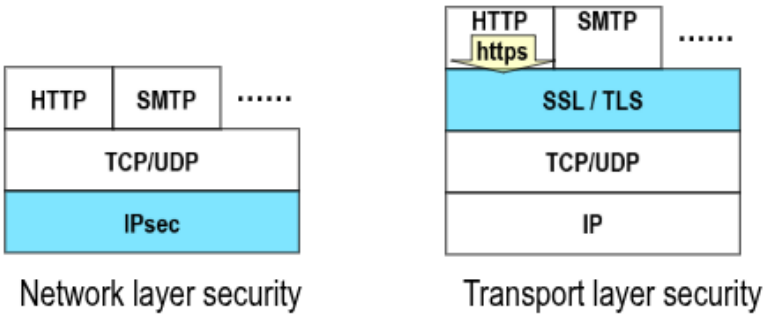
We have an IP in IP tunnels which is not the most effective approach, the outer packet has the IP source and destination of the gateways and the inner packet has the IP source and destination of the nodes.
MPLS tunnels are far more performance effective.

Another VPN ingredient is encryption



VPN and IPsec
IPsec is a possible tool for building VPN but its not the same, there are some VPN alternatives like PPTP (Layer 2), MPLS (Layer 3), DTLS tunnels (Layer 4), SSH tunnels (Layer 7).

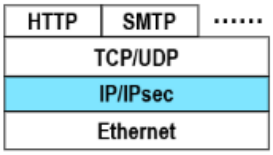
IPsec Components



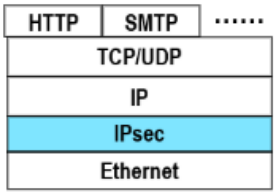
TLS is a transport layer security that operates between transport and application, while IPsec is a network layer security that operates with and within IP, the application / terminal is unaware of IPsec and protects all protocols that rely on IP.

IPsec implementation approaches

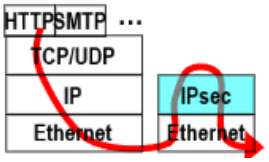
Inside the native IP code
- Best approach
- But hard to deploy as requires to access and modify IP source code



Bump in the Stack (BITS)
- Between native IP code and device driver
- Deployed in legacy systems

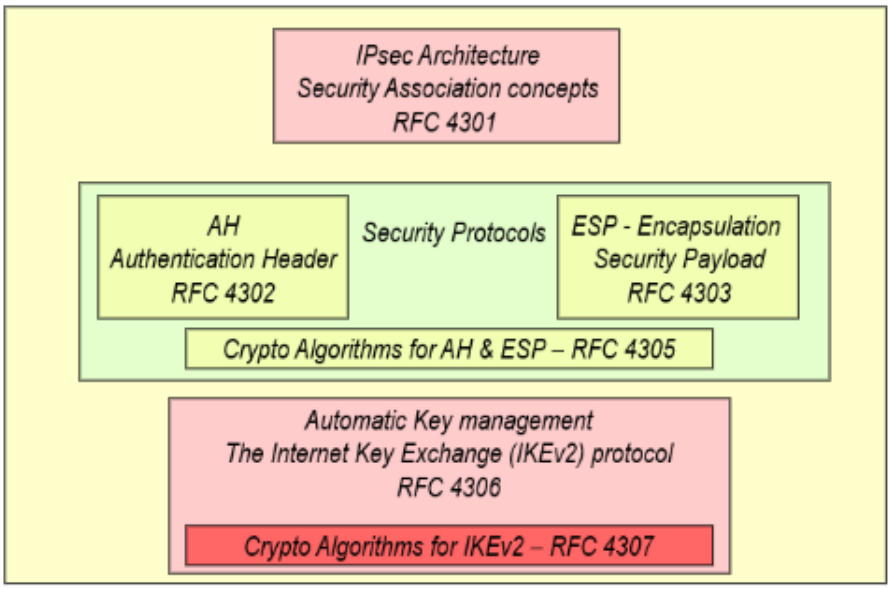


Bump in the Wire (BITW)
- Implemented in dedicated hardware
- External security processor, acts as gw



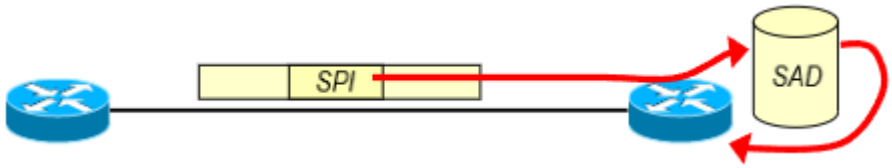
IPsec standardization has three major series
Serie 1: 1995 where the IPsec concepts were first drafted.
Serie 2: 1998 had a significant revision of ALL the IPsec architecture.
Serie 3: 2005 extends all the IPsec architecture and had a major revision of IKE (Internet Key Exchange).

IPsec RFCs



Security Association
Is a fundamental concept in IPsec and may involve host to host, host to intermediate router (security gateways) and security gateway to security gateway. The security association defines the boundaries for IP packets authentication / encryption.

Security Parameters Index and Security Association DB



To differentiate the connection is used a 32 bit index for the lookup in the SAD at destination, the lookup also uses destination address, source address and security protocol. Retrieves algorithms and parameters that allow to process received packet.

The SA is monodirectional

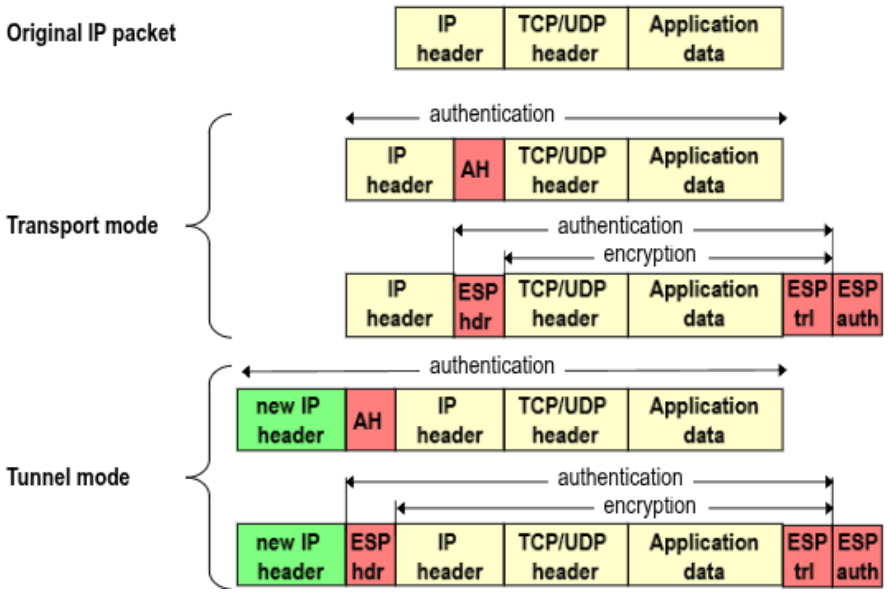
When setuping the connection between two nodes we need to setup SA1 and SA2 for the traffic going in different directions. One SA protects packets in one direction only.
The SPI (Security Parameters Index) is a 32bit identifier that names a specific SA, so when a packet arrives is used the SPI to figure out which SA it belongs to based also on the SAD (Security Association Database).

The Security Association and Key management can be configured
Manually: manually configure each SA and related crypto keys, typical in small scale VPNs

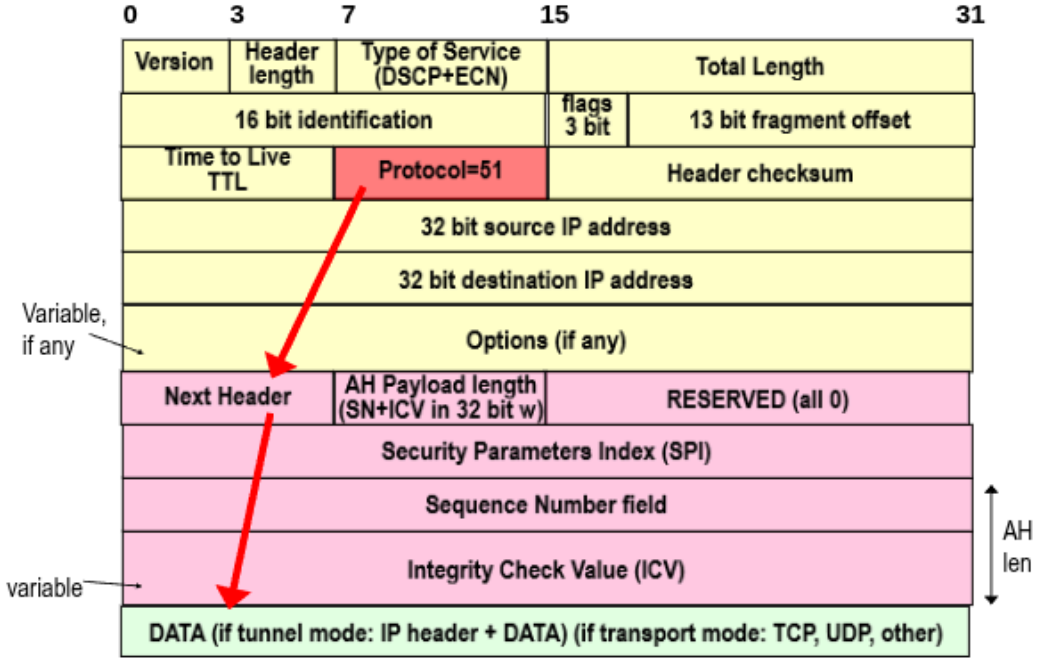
Automatically: SA management through IKEv2, SA creation on-demand and session oriented keying.

IPsec Security Protocols
Authentication Header (AH): Integrity and data origin authentication, so it guarantees, this one covers both payload and parts of IP header and make sure they doesn't modify in transfer. It also guarantees Protection against replays.
Encapsulated Security Payload (ESP): Same service as AH, confidentiality through encryption, it also has traffic flow confidentiality.
ESP in most cases is the only one needed and mandatorily supported in any IPsec implementation, unlike AH, the can be combined together if needed.

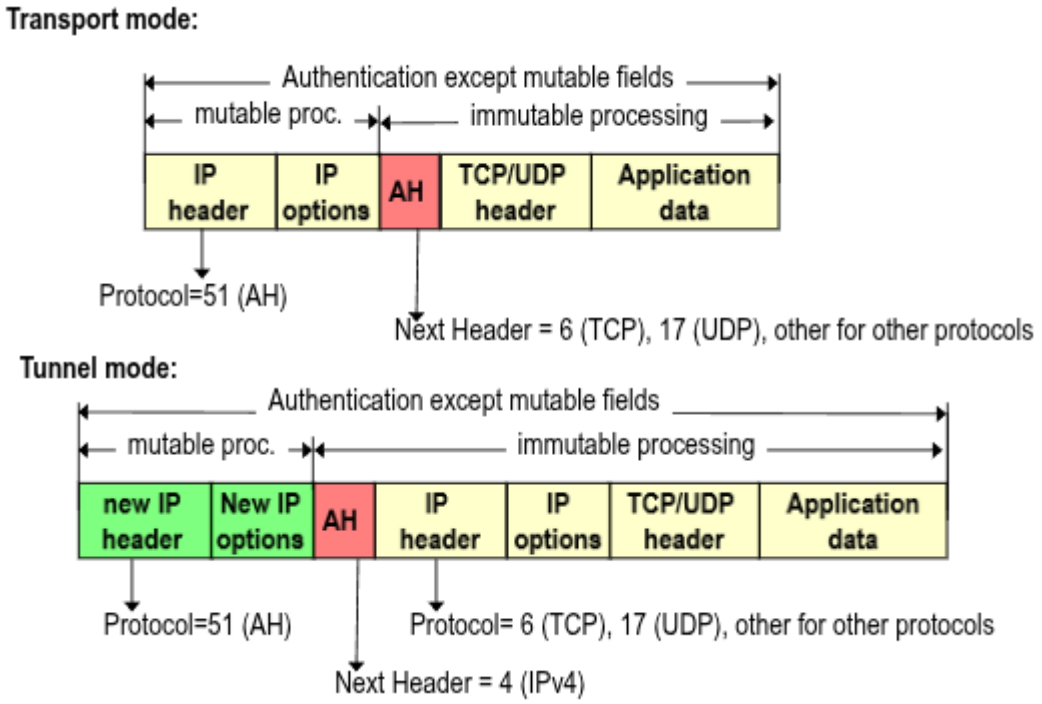
Transport vs Tunnel - AH and ESP



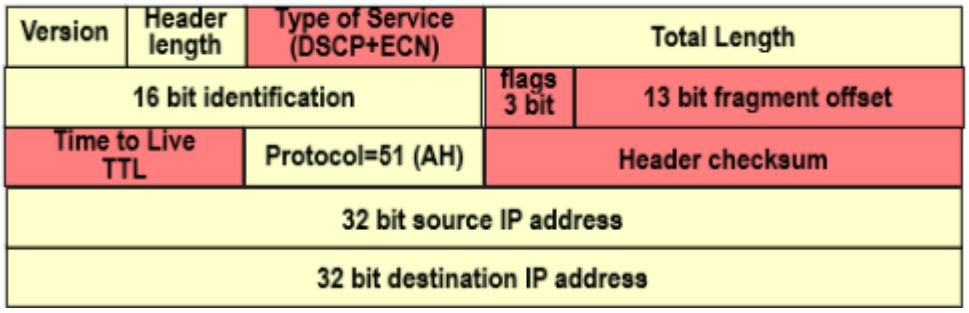
Authentication header



Transport mode, tunnel mode



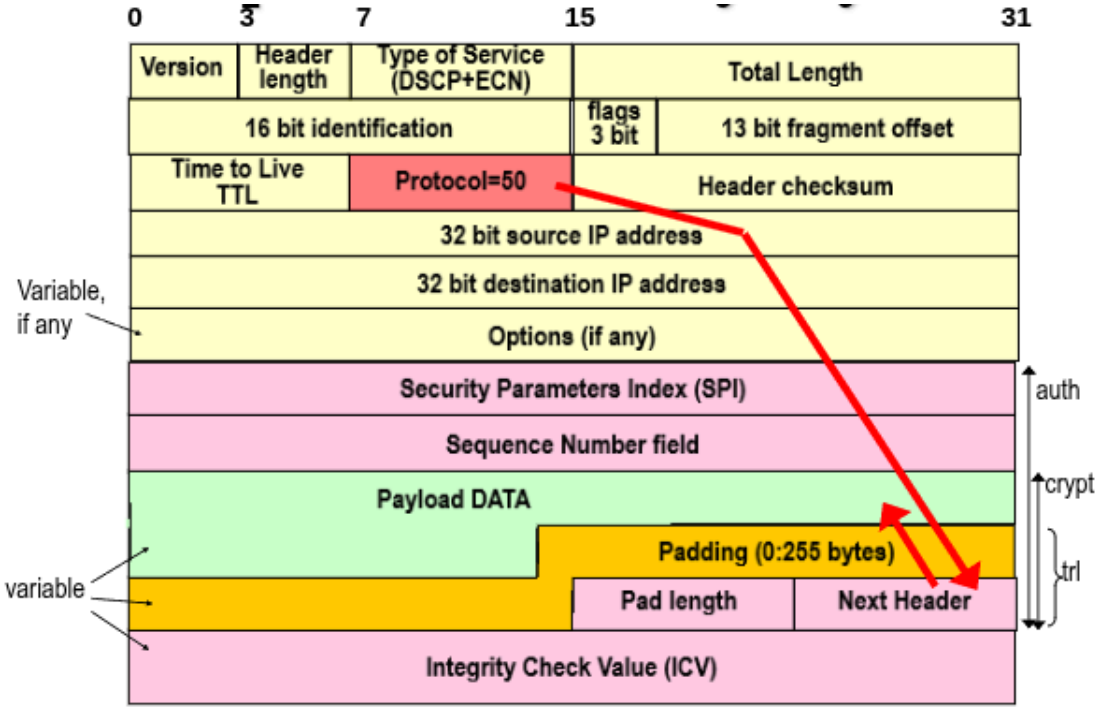
Integrity Check Computation



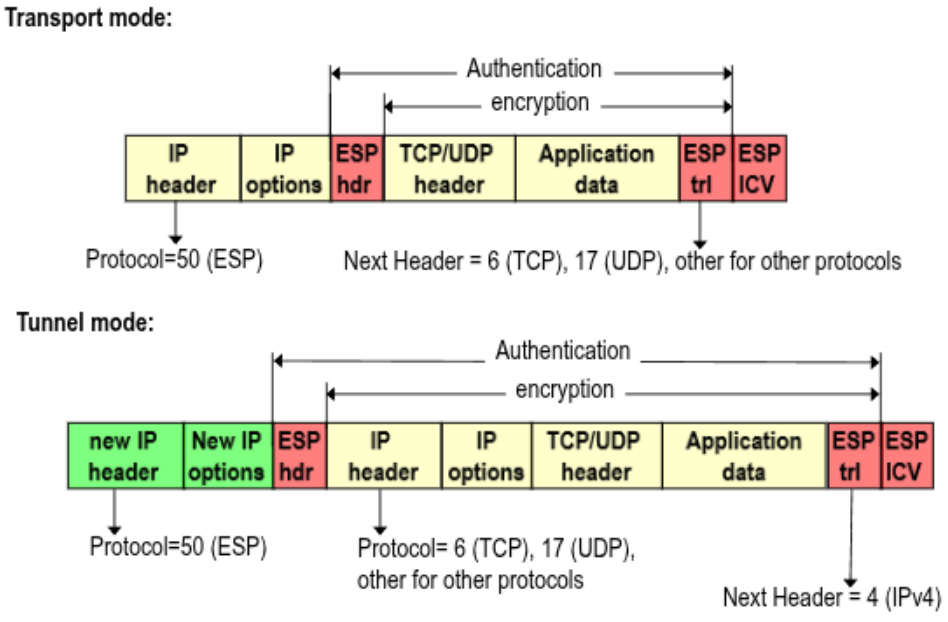
some fields of the packet (the red ones) are mutable during the transfer of the packet so the integrity check is done only on immutable fields of the IP header, or mutable but predictable. During the MAC computation the mutable fields are set to 0.

In the Authentication Header we see that the sequence number has 32bits, taking into account that a SA can last even one year and, having only 4 million packets is an issue. This is initialized to 0 when the SA is established and increments by 1 per each packet transmitted, after that the sequence number reach the value of 2^{32} - 1 the SA must be terminated. The extended sequence number can also take 64bits that should be more than enough.

Encapsulated Security Payload



Transport mode, tunnel mode



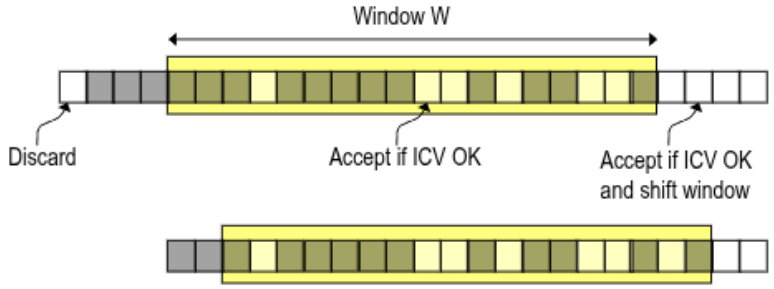
Traffic Flow Confidentiality Service

Encryption doesn't guarantee unlikability, cause thanks to statistical traffic pattern analysis we can see for example that amazon has a fingerprint (transmission of the same packets every single time) than another site.

Anti-replay

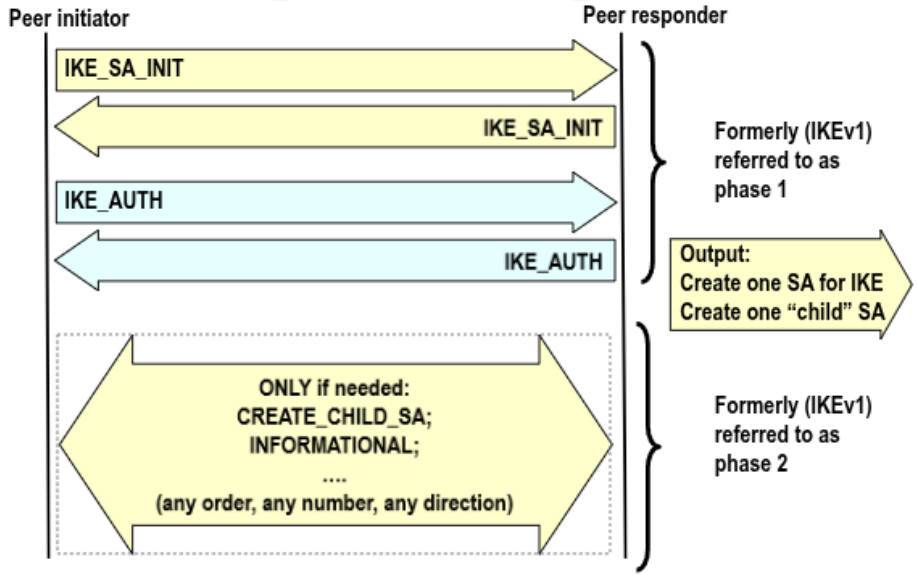
IPsec also comes with mechanisms for anti-replay. Thanks to a sliding window the only packets accepted are the ones in this sliding window, where the size of the window is

decided at receiver from a minimum of 32 to a maximum of 2^{31} -1. The duplicates and the packets out of left window edge are discarded while the packets greater than right window margin make the window shift.



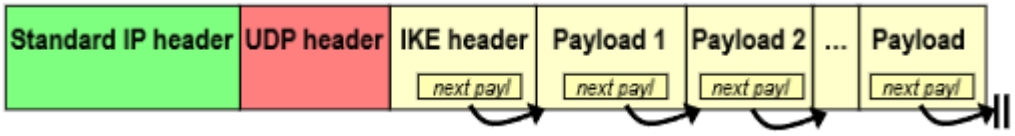
ESP Traffic Flow Confidentiality
ESP TFC: Two counter-measures against traffic analysis attacks:
Ability to alterate packet size by supplementary padding added to data payload.
Ability to generate dummy packets by putting the next header = 59.

IKEv2
Shared state must be maintained between source and sink, and manual maintenance is not scalable so it's used a IKE = Internet Key Exchange protocol for dynamically establish and maintain SA.
IKE phases:



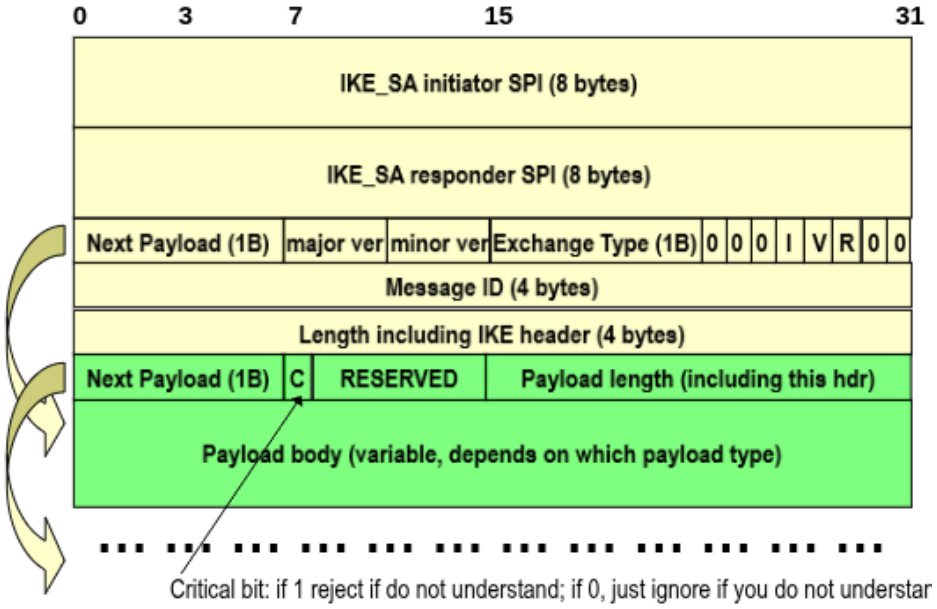
IKE SA and CHILD SA
The IKE SA: Security association to exchange IKE messages
The Child SA: Security association to exchange data messages, many Child SA may be setup between two peers.

IKE message format



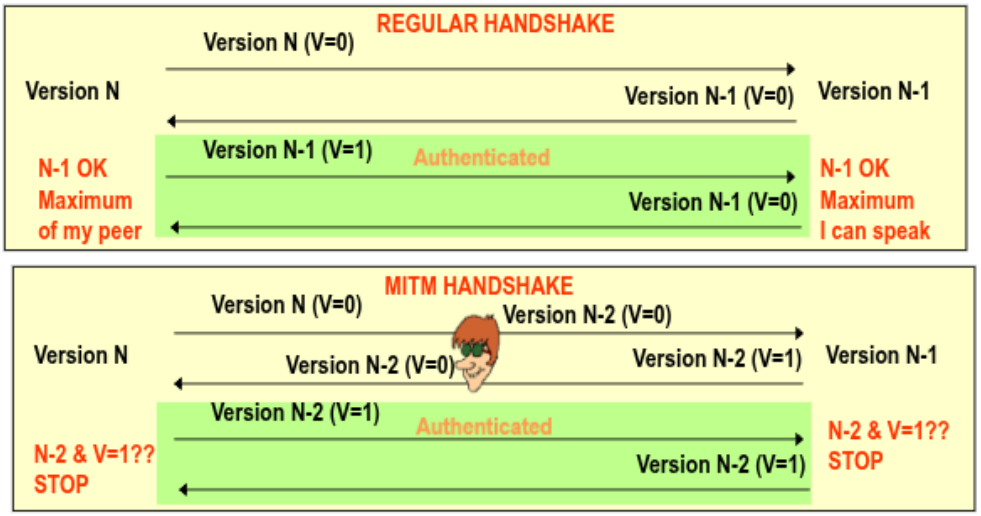
We the IKE header and then all the IKE payloads that manages each one single things.

IKE Header



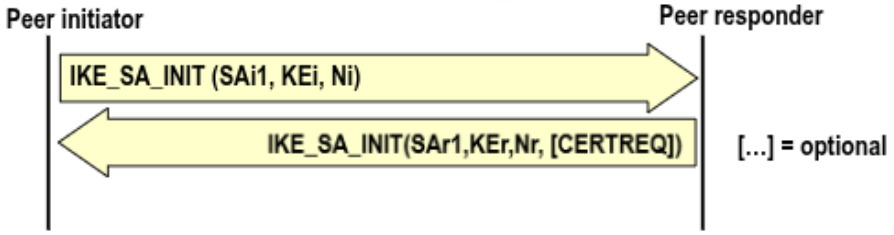
If u don't understand a payload the bit C in the hdr tells us if we reject it or we ignore it.

Protection against version rollback



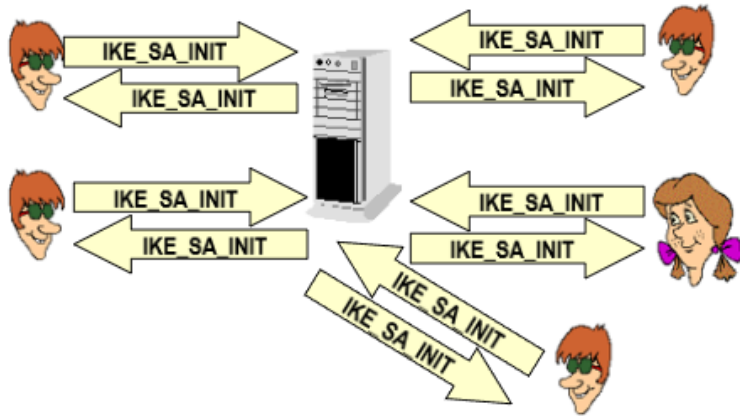
A say to B to use the version N with flag V=0 that is the maximum i can stand B responds with using N-1 with V=0 so that is the maximum B can stands.
Now the exchange of packets is A sends the version N-1 with flag V=1 saying that he can use a higher version, while B responds with version N-1 and V=0.
With a MITM attack:
A say to B to use the version N with V=0, and the MITM intercept the packet and saying to B to use version N-2 with V=0, after that B responds to A using N-2 and V=1 but the MITM responds to A with version N-2 V=0.
Now the exchange of packets is A sends the version N-2 with V=1 while B responds with version N-2 and V=1. They both can use a higher version so they stop.

Exchanged Info

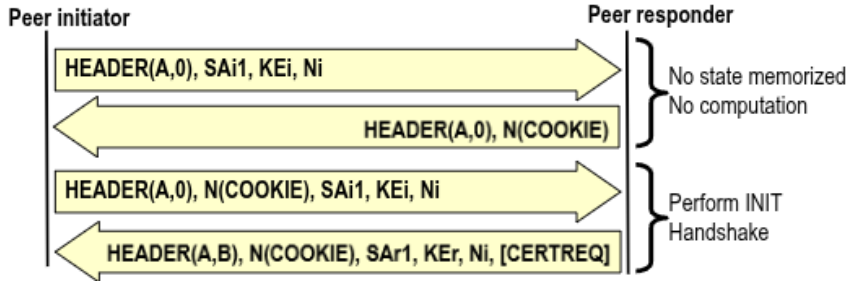


The SAi1 SAr1 is the Security Association payload where the initiator sends a list of crypto algorithms supported and the responder choose one algo per each function.
KEi and KEr are the Key Exchange payloads: DHE values.
Ni Nr payloads are random values used in crypto parameters
CERTEQ is a request for certificate.

Protection against DoS attacks
DH computation and key generation is a computational expensive process because the state must be motorized. In a Denial of Service attack there are many spoof INIT request and those overload the server.

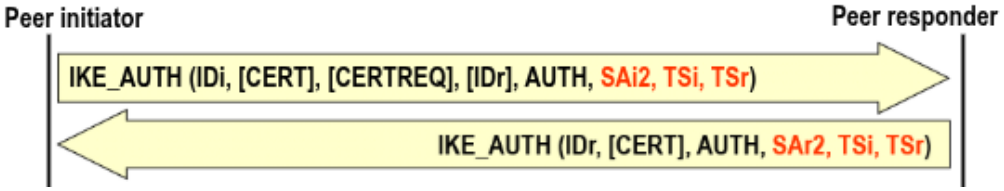


The solution is a cookie-based 4-way INIT handshake



The idea is that the responder first replies with a cookie so only the real initiators will reformulate the request including the cookie. Only at this point a state is initialized.
Now the DoS attacker can spoofs random cookies too, so the server must recognize valid cookies hence it must store a state fro the cookies and use memory.
So the best idea is to use stateless cookies, cookies that doesn't require any state memorization, cookies that can be checked without any lookup at a database but only looking at the request and at the cookies itself.

IKE Auth Phase



The IKE message is encrypted and authenticated, except for the IKE header and sends AUTH payload.

TERZA PARTE

Secret Sharing

Trivial Secret Sharing
The goal of the secret sharing is to share a secret among two persons such that it is revealed if BOTH reveal their part. An example can be to split the secret in two part and then reconstruct it, but that's a bad idea cause the secret is WEAKENED when one share revealed.
For example let's say that the goal is: 00101101 we split it in half 0010 and 1101, the probability to guess secret right is before seeing anything 1/256 (2^8) and after seeing one share 1/16 (2^4).

A better solution is to generate a random sequence to XOR it with the actual secret and give to one part the random sequence and to the other part the XOR value

Secret: *0010.1101*

Generate random sequence, ex. *1011.0100*

XOR Secret & random *1001.1001*

Give alex the random sequence
1011.0100

Give Bob the XOR value
1001.1001

Secret = Ashare XOR Bshare
1011.0100 ⊕ 1001.1001 = 0010.1101

If the bad guy gets to know alex's share no information on the secret is revealed cause it is a random value, on the other hand if gets to know about bob's share it is also secure cause is a XOR between a random sequence and the secret.

Perfect secrecy:

Secret bit
Probability:
0=p
1=(1-p)

Random bit
Probability:
0=1/2
1=1/2

Secret bit	Random bit	XOR result bit	
0	0	0	p/2
0	1	1	p/2
1	0	1	(1-p)/2
1	1	0	(1-p)/2

the probability to guess secret after seeing ONE share is the same as a guess without seeing any share.

The XOR is not strictly needed we can use (modular (not necessarily with a prime number)) sums:

SECRET 0010.1101→ 45

RAND mod 256: 180

S – RAND mod 256: 121

180 121

Secret = Ashare + Bshare

180+121 mod 256 = 45

Module not necessary, but practical (e.g. 32 bit numbers)

It's trivial to extend this to n parties

S → 45

R1 → 135

R2 → 6

R3 → 67

S-R1-R2-R3 → 93

135 6 67 93

Reconstruction:

$(135+6+67+93) \bmod 256 = 45$

In this case we have a perfect secrecy up to n-1 shares revealed, adversary knowing n-1 shares has still 1/256 prob to guess secret

Shamir Secret Sharing

(n, n) secret sharing scheme

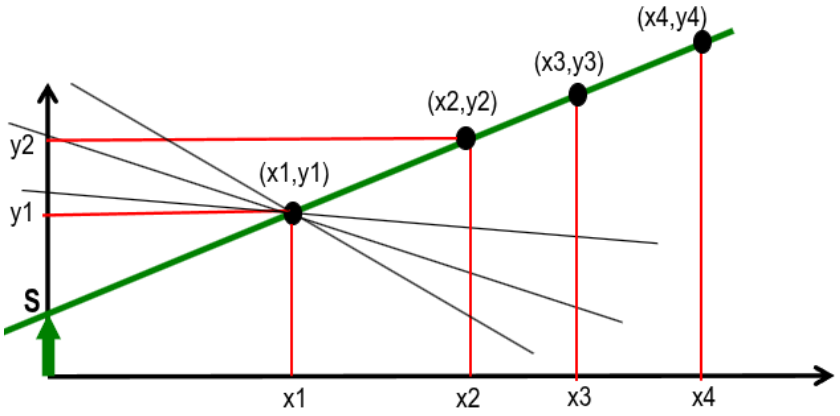
Secret revealed only if ALL n parties provide a share

(t, n) secret sharing scheme

Secret revealed when ANY t OUT of n parties provide a share (if t=1 then every partner has the secret)

Shamir created it in 1979 and also did it Blakley, these were different solution but the Shamir one was better in terms of performance

A (2, n) scheme idea



We only need to points to define a line, the secret S is the y-value at x=0, so with one share S remains unknown (we can have infinite line in one point) but with two shares we can determine S.

Procedure:
The dealer conduct the line with the coefficient a: randomly chosen and the secret S as known term so y= S + ax (es: y = 39 + 15x)
then it distributes the share to n participants by choosing randomly x_i:
Participant 1: 1 -> share = (1, 54)
Participant 2: 2 -> share = (2, 69)
Participant 3: 3 -> share = (3, 84)
…

Receive two shares
$P_i = (x_i, y_i)$, $Pj = (x_j, y_j)$

Interpolate points (equation of line)

$$\frac{y-y_j}{y_i-y_j}=\frac{x-x_j}{x_i-x_j} \Rightarrow y=y_j+\frac{x-x_j}{x_i-x_j}(y_i-y_j)$$

Set x=0 for deriving y=S

$$S=y_j+\frac{0-x_j}{x_i-x_j}(y_i-y_j)=y_j\frac{-x_i}{x_j-x_i}+y_i\frac{-x_j}{x_i-x_j}$$

- **Example: Pi=(3,84), Pj=(1, 54)**

$$\frac{y-54}{84-54}=\frac{x-1}{3-1} \Rightarrow y=54+15(x-1) \Rightarrow S=54+15(0-1)=54-15=39$$

$$S=y_1\frac{-3}{1-3}+y_3\frac{-1}{3-1}=54\cdot\frac{3}{2}-84\cdot\frac{1}{2}=39$$

Extension to (t, n)
Line -> 2 points
Quadratic -> 3 points
Cubic -> 4 points
…
Need formula to interpolate t shares, the most convenient is the Lagrange interpolation.

Any polynomial of degree t-1 with t known points (x_1, y_1), …, (x_t, y_t) can be decomposed as

$$y = \sum_{i=1}^{t} y_i \Lambda_i(x)$$

] **Where $\Lambda_i(x)$ is the basis polynomial**

$$\Lambda_i(x) = \prod_{m=1, m\neq i}^{t} \frac{x - x_m}{x_i - x_m} = \frac{(x-x_1)}{(x_i-x_1)} \cdots \frac{(x-x_{i-1})}{(x_i-x_{i-1})} \frac{(x-x_{i+1})}{(x_i-x_{i+1})} \cdots \frac{(x-x_t)}{(x_i-x_t)}$$

] **Clearly a basis:** $\Lambda_i(x_i) = 1; \quad \Lambda_i(x_m) = 0 \quad for\ m \neq i;$

(t, n) scheme

Dealer: Generate random polynomial p(x) with degree (t-1) and set secret S as known term in the polynomial

$$p(x) = s + a_1 x + a_2 x^2 + \cdots + a_{t-2} x^{t-2} + a_{t-1} x^{t-1}$$

Distribute one share to earache of the n parties

$$(x_i, y_i) \quad y_i = p(x_i)$$

Reconstruction: Collect the t shares out of the n available and compute the secret using Lagrange interpolation.

$$s = \sum_{shares\ x_i} y_i \Lambda_{x_i} \quad with \quad \Lambda_{x_i} = \Lambda_{x_i}(0) = \prod_{shares\ x_k \neq x_i} \frac{-x_k}{x_i - x_k}$$

Example:

Secret: value between 1 and 100

$$\text{dealer} : s = 32$$

$$\text{dealer} : p(x) = 32 + 52x + 3x^2$$

$$\text{shares} : (1,87), (2,148), (3,215), (4,288)$$

Collected shares: 1,2,3 (4 missing)

$$\Lambda_1 = \frac{-x_2}{x_1 - x_2} \cdot \frac{-x_3}{x_1 - x_3} = \frac{-2}{1-2} \cdot \frac{-3}{1-3} = 3$$

$$\Lambda_2 = \frac{-x_1}{x_2 - x_1} \cdot \frac{-x_3}{x_2 - x_3} = \frac{-1}{2-1} \cdot \frac{-3}{2-3} = -3$$

$$\Lambda_3 = \frac{-x_1}{x_3 - x_1} \cdot \frac{-x_2}{x_3 - x_2} = \frac{-1}{3-1} \cdot \frac{-2}{3-2} = 1$$

$$\text{secret} : s = y_1\Lambda_1 + y_2\Lambda_2 + y_3\Lambda_3 = 87 \cdot 3 + 148 \cdot (-3) + 215 \cdot 1 = 32$$

The scheme presented so far is flawed cause we don't have the perfect secrecy (we don't need n-1 shares). I know that the secret S is between 1 and 100 now i can do a range of D in order to get many possible secrets S.

Example: shares 1 and 3 collected; can we say something about s??

$$\text{compute as before}: \Lambda_1 = 3; \Lambda_2 = -3; \Lambda_3 = 1$$

$$\text{share 2 unknown}: \text{set to D}$$

$$\text{secret}: s = y_1\Lambda_1 + D\Lambda_2 + y_3\Lambda_3 = 87 \cdot 3 + D \cdot (-3) + 215 \cdot 1 = 476 - 3D$$

D=integer → s=98 (D=126), 95, 92, 89, 86, 83, … 2/3 of values s EXCLUDED!

Knowledge of shares 1 and 3 permits to triplicate guess probability!

Doing this we now have ⅔ of the values excluded and more probability to guess the secret.

The real Shamir scheme uses modular arithmetic instead of real arithmetic, in this way the secret and polynomial are in prime field F_p and the interpolation formula still holds mod p. The result is an unconditionally secure scheme with the condition that the prime p must be greater than the domain for the secret and doesn't strictly need to be large because security is not computational but is perfect (theory security).

Example:

Secret: value between 1 and 100

$$\text{dealer}: s = 32$$

$$\text{dealer}: p(x) = 32 + 52x + 3x^2 \mod p = 101$$

$$\text{shares}: (1,87), (2,47), (3,13), (4,86)$$

Collected shares: 1,2,3 (4 missing)

$$\Lambda_1 = 3; \Lambda_2 = -3; \Lambda_3 = 1;$$

$$s = y_1\Lambda_1 + y_2\Lambda_2 + y_3\Lambda_3 = 87 \cdot 3 + 47 \cdot (-3) + 13 \mod 101 = 32$$

$$s = 87 \cdot 3 + D \cdot (-3) + 13 \mod 101 = 72 - 3D \mod 101$$

S uniformly distributed in F_p!

Ideality

The share cannot be smaller than the secret, given t-1 shares no information can be determined about secret. Thus, the final share must contain as much information as secret itself.

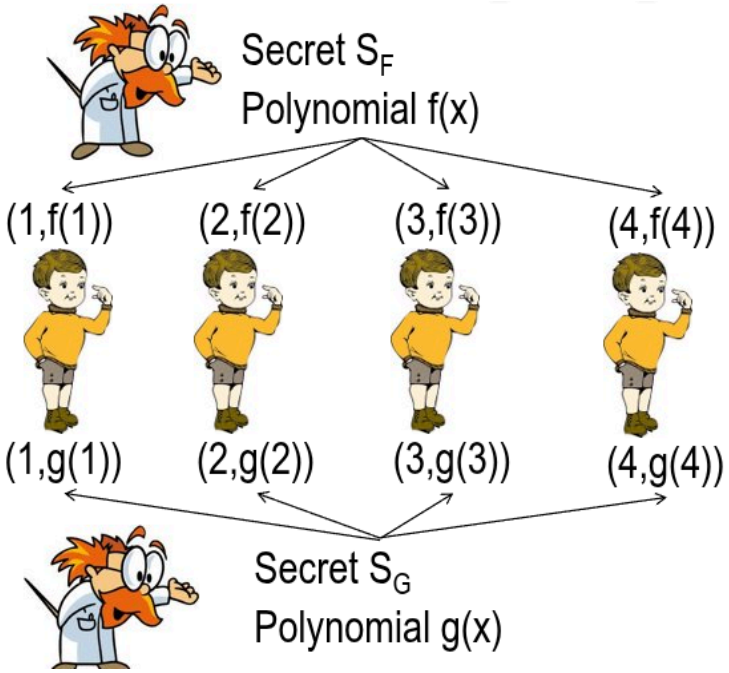
The Shamir scheme is ideal cause the length of the share is equal to the length of the secret (Blakley scheme shares are t-time secret so a share larget than secret).

Secret Sharing for Secure Multiparty Computation

Homomorphic property

There are two secrets and two polynomials: f(1) to f(4) are the share for S_F g(1) to g(4) are

the share for S_G and f(1)+g(1) is a share of the secret S_F+S_G.



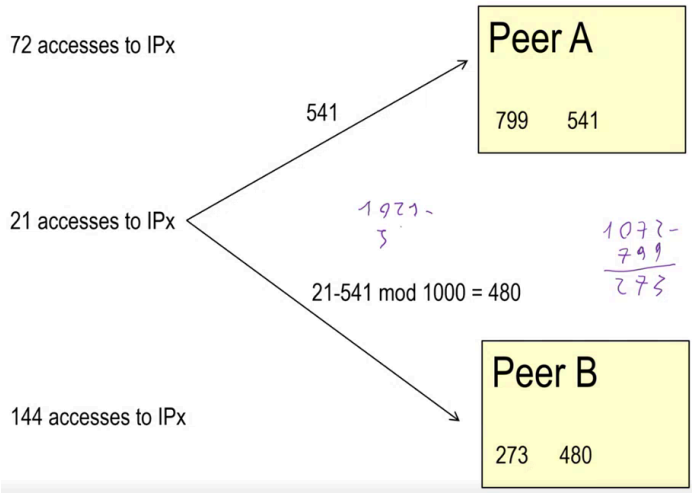
Homomorphic property:

SUM of the SHARES = Share of the SUM S_F+S_G

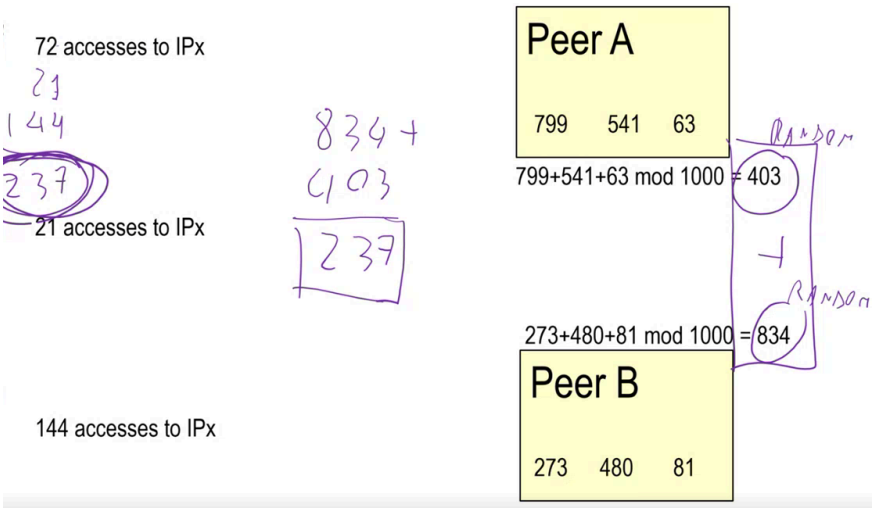
f(i), g(i) → f(i)+g(i) → → (f+g)(i)

Revealing t composite shares reveals SUM of secrets but does NOT reveal individual secrets

Example:



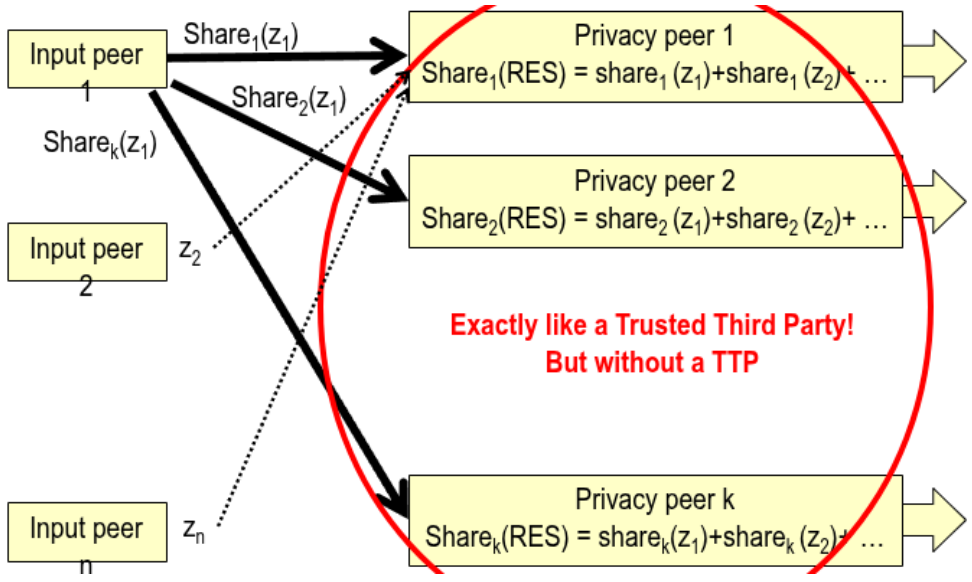
We have the tree secrets (72, 21, 144) and two privacy peers, we send to the first one A a random number and to the second one B the (secret - random number)mod x (x can be any number, prime not needed) now the Peer A and Peer B can both do the sum of the receiving number mod x and get two different number:



If we sum those number we get the sum of the secrets.

In the SMC (Secure Multiparty Computation) we compute the result of a function without revealing input data and is used in several scenario like business, medical and so on. Born as 2-party computation and then created multiparty (Yao's million problem, 1982 to revel which is richer).

We have N parties P_1, …, P_n each with value z_i
Compute the function $f(z_1, ..., z_n)$ such that result is public and no information is given on input values z_i.
Our interest in the special case of addition is a statistic aggregation, it also works with weighted addition. If f = addition then we have a very efficient SMC from secret sharing scheme

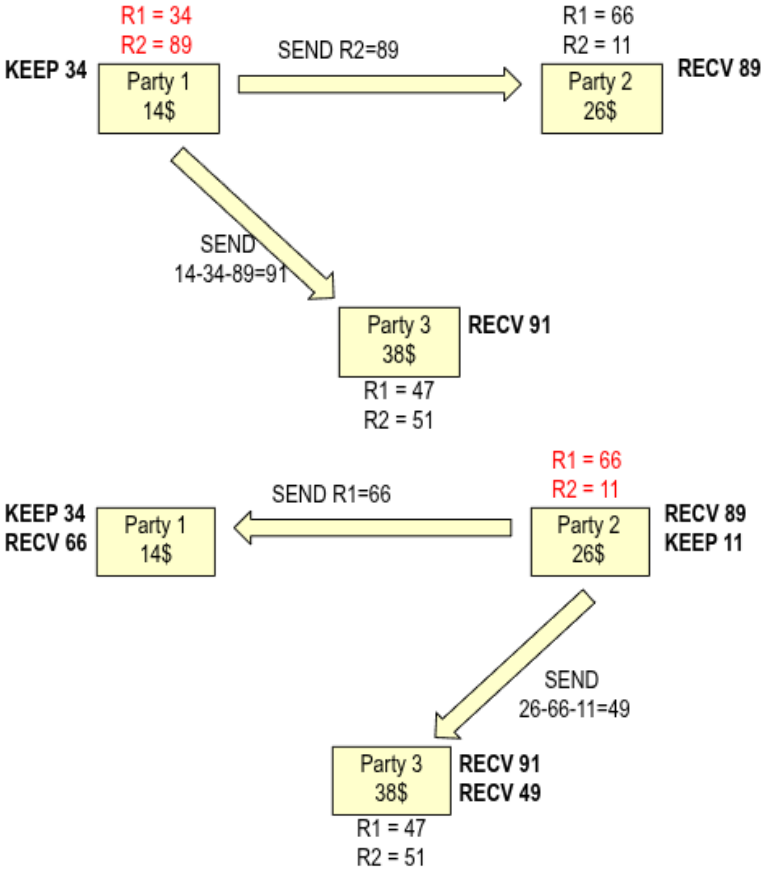


In this way we can have a (k,k) scheme or like (3, k) scheme so we only need the shares of 3 privacy peers and not all of all the k peers.

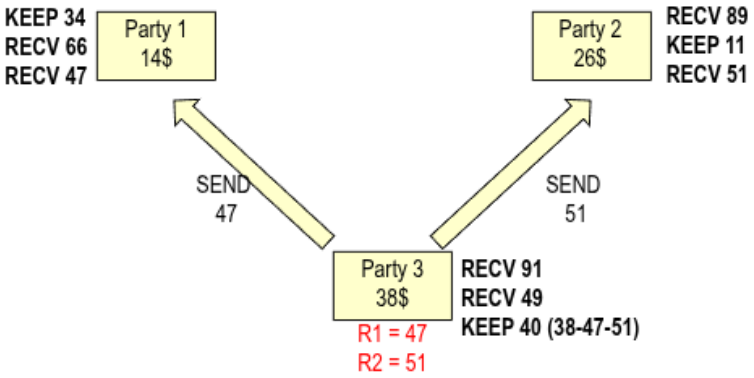
Deployment issues:
The input peers must be at least 3 cause if 2 then one party can know other party data by

subtracting from final result. The privacy peers must be at least 2 but the more the better for security.

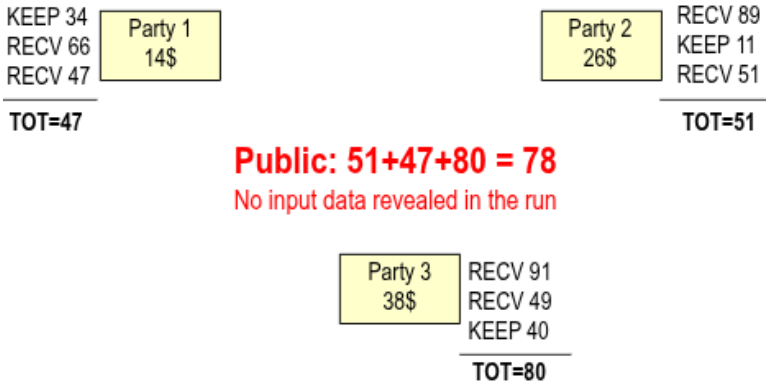
Input peers may also be privacy peers.
Example: Let's say 3 parties want to donate some money but they don't want to say how much but only know the total.
Each one generate two random values and only keeps one. The other random value is sent to one party and the operation of S-R1-R2 is sent to the other party (assume mod 100)



The last peer will generate two random numbers and sent both of the to the parties and he only keeps the S-R1-R2.



In this way they all sum up the numbers received and they sum all the sums giving out the total of the donation without revealing any input data.



Verifiable Secret Sharing
The model so far was “honest-but-curious” so both parties and peers respected the rules but there is no way so far to check that protocol rules are followed so we need to detect if we have a malicious dealer or if we have cheating parties.

Recap Math:
(G, o) is a group where
G = set of elements (group members)
o = operation (group operation)
It has 4 properties:
- Closure: for any g_1, g_2 we have that $g_x = g_1$ o g_2 must be a group member
- Identity: there is a group member I such that g o I = I o g
- Inverse: for any g, there is g^{-1} such that g o g^{-1} = I
- Associativity for any g_1, g_2, g_3: (g_1 o g_2) o g_3 = g_1 o (g_2 o g_3)
If also commutative then it is an Abelian Group

The Zp* group is the multiplicative group modulo prime p
Set of p-1 elements {1, 2, …, p-1}, multiplicative cause we only care about multiplications mod p (if we had the sum then it would be F_P), It has the same properties as an Abelian group. For the inverse properties note that if mod N, then x has inverse iff gcd(x,N) = 1 but in this case N = p = prime, then all elements have the inverse (0 isn’t a group member)

Example Z_{11}*

Elements: {1,2,3,4,5,6,7,8,9,10}

Inverses:

- ⇨1→1
- ⇨2→6 6→2
- ⇨3→4 4→3
- ⇨5→9 9→5
- ⇨7→8 8→7
- ⇨10→10 (analogous to -1)

To compute inverses we can use the Extended euclidean algorithm

Exponentiation is x^k = x o x o … o x (k times)
Generator of group of order m means that exist a g such that $\{g^0, g^1, \ldots, g^{m-1}\}$ = all m group members. A group is a Prime-order group if m is prime and any member is generator except the identity
Zp* isn't a prime order group cause the length of Zp* = p-1 that cannot be prime cause p is.
Example Z_{11}*

Elements: {1,2,3,4,5,6,7,8,9,10}

Generators? $\{g^1, g^2, g^3, \ldots, g^{10}\}$=?

- ⇨ g=2 → {2,4,8,5,10,9,7,3,6,1} — OK, generator
- ⇨ g=3 → {3,9,5,4,1,3,9,5,4,1} — NO! Subgroup order 5
- ⇨ g=4 → {4,5,9,3,1,4,5,9,3,1} — NO! Subgroup order 5
- ⇨ g=5 → {5,3,4,9,1,5,3,4,9,1} — NO! Subgroup order 5
- ⇨ g=6 → {6,3,7,9,10,5,8,4,2,1} — OK, generator
- ⇨ g=7 → {7,5,2,3,10,4,6,9,8,1} — OK, generator
- ⇨ g=8 → {8,9,6,4,10,3,2,5,7,1} — OK, generator
- ⇨ g=9 → {9,4,3,5,1,9,4,3,5,1} — NO! Subgroup order 5
- ⇨ g=10 → {10,1,10,1,10,1,10,1,10,1} — NO! Subgroup order 2

Either g is a generator or generates a subgroup and Zp* as well as all subgroups are cyclic.

Strong prime
Those are prime such that p = 2q+1 being q a prime and the order of Zp*: p-1 and p-1 = 2q, hence any member x (except 1 and p-1) either generates the whole group or generates subgroup of prime order p, both large if p and q large

Quadratic residue subgroup
x is a member of Zp* and it's called a quadratic residue if it admits square root in Zp*, there exists *a* such that a^2 mod p = x. The QR forms a subgroup of order (p-1)/2 cause we have a 2 to 1 mapping:

Indeed, $(p-x)^2 \bmod p = p^2 - 2px + x^2 \bmod p = x^2$

a test for the QR is the legendre symbol where a is a QR if a^((p-1)/2) mod p = 1 otherwise is -1.

VSS (Verifiable SS)
First (non interactive) scheme proposed in 1987 known as Feldman VSS scheme, frequently used as basis for distributed schemes and the basic approach presented discrete logs.

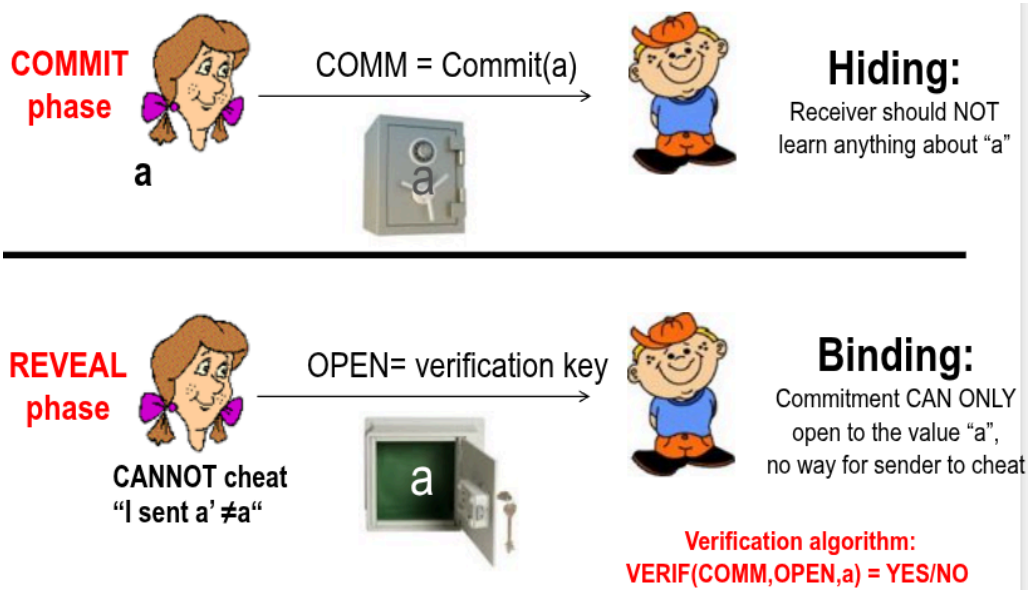
Feldman scheme: dealer
Start with an ordinary Shamir scheme so generate a p(x) with degree (t-1) with P(0) = s and distribute one share to each of the n parties.
Now, for each coefficient of the polynomial as well as secret, further broadcast:

$$c_0 = g^s; c_1 = g^{a_1}; \cdots c_{t-1} = g^{a_{t-1}} \mod p$$

(p is a strong prime so where p - 1 = 2q and q is a prime)

To verify if g is the generator of the group or the subgroup u just need to compute g^((p-1)/2) mod p= g^q mod p if its 1 is a generator of the subgroup if its -1 then it generates the whole group. This is called Quadratic Residue Subgroup.
These are special type of commitments cause they are discrete log so hard to compute.

What is a commitment:
In the first phase, the commit phase, A declares a value “a” and sends the Commit “Commit(a)” to B, this commit should not leak any info about “a”.
In the second phase, the reveal phase, A reveals his information and B verify the commit.
The commitment can only open to the value “a”, there is no way for the sender to cheat.



$c = g^x$ is a commitment
(computationally) Hiding: given $c = g^x$ mod p, it is computationally hard for the receiver to get x.
(perfectly) Binding: sender cannot find any other x’ such that $g^{x'} = c$, x’ cannot be greater than group order otherwise is trivial cause $g^x = c = g^{x+(p-1)}$ mod p.

Feldman scheme: verifier
Party i receives share (x_i, y_i), when party i reveals its share check if that party i is honest.
Verifies that share is valid (hence dealer honest) by computing (mod p)

$$c_0 \cdot c_1^{x_i} \cdot c_2^{x_i^2} \cdot c_3^{x_i^3} \cdot \ldots \cdot c_{t-1}^{x_i^{t-1}} =$$
$$= \left(g^{s}\right) \cdot \left(g^{a_1}\right)^{x_i} \cdot \left(g^{a_2}\right)^{x_i^2} \cdot \left(g^{a_3}\right)^{x_i^3} \cdot \ldots \cdot \left(g^{a_{t-1}}\right)^{x_i^{t-1}} =$$
$$= g^{s} \cdot g^{a_1 x_i} \cdot g^{a_2 x_i^2} \cdot g^{a_3 x_i^3} \cdot \ldots \cdot g^{a_{t-1} x_i^{t-1}} =$$
$$= g^{s + a_1 x_i + a_2 x_i^2 + a_3 x_i^3 + \ldots + a_{t-1} x_i^{t-1}} = g^{p(x_i)}$$

And checks that this is equal to g^{y_i}, where y_i is the party’s share.

So in the end in the Feldman scheme dealer its better to use q as the mod of p(x) and to use q as p=2q+1 (p strong prime), and for g we should use a generator of the subgroup q.

Shamir Scheme is unconditionally secure but Feldman's VSS is only computationally (Hiding) secure. But $c_0 = g^s$ leaks information about secret s, if s is small we can use a trivial dictionary attack: for x 1 to 1000 repeat g^x until $g^x = c_0$ doing this we get to know s.

Feldman approach works because:
g^a mod p is a Commit(a) and it's homomorphic: Commit(a+b) = Commit(a)Commit(b), indeed $g^{a+b} = g^a g^b$ mod p.
We need a perfectly hiding commitment scheme which is homomorphic and it was found by Pedersen in 91.
Given g and h (public): Commit(a; r) = $g^a h^r$ mod p

$$Commit(a+b; r_a + r_b) = g^{a+b} h^{r_a + r_b} =$$
$$= g^a h^{r_a} \cdot g^b h^{r_b} = Commit(a, r_a) \cdot Commit(b, r_b)$$

Pedersen commitment
(perfectly) Hiding: $c = g^a h^r$ mod p may be a commitment for any possible value, for any *a* different from *a'* we can find unique r' such that

$$commit(a';r') = g^{a'} h^{r'} = g^a h^r = commit(a;r)$$

(computationally) Binding:
Sender shouldn't be able to find such an a' (and related r') which is the case if sender doesn't know $\log_g h$.

Perfectly hiding + perfectly binding commitment is impossible, u can't have both just one or another.

Proof (base of)

let $h = g^w$ i.e. $w = \log_g h$

we know a, r, we are given a', and we look for r' s.t.

$$g^a h^r = g^{a'} h^{r'} \Rightarrow g^a g^{wr} = g^{a'} g^{wr'} \Rightarrow$$
$$\Rightarrow g^{a+wr} = g^{a'+wr'} \Rightarrow$$
$$\Rightarrow a + wr = a' + wr' \mod q \text{ (group order, prime s.t. } 2q + 1 = p) \Rightarrow$$
$$\Rightarrow r' = w^{-1}(a - a' + wr)$$

i.e., given w, we can compute ANY "colliding" committment, hence we would not commit to anything!! (no binding)

Pedersen VSS -dealer

Generate two random polynomials

$$f(x) = s + a_1 x + a_2 x^2 + \cdots + a_{t-2} x^{t-2} + a_{t-1} x^{t-1}$$

$$f'(x) = r + b_1 x + b_2 x^2 + \cdots + b_{t-2} x^{t-2} + b_{t-1} x^{t-1}$$

Give party i double share (x_i, y_i, z_i)

$$y_i = f(x_i) = s + a_1 x_i + a_2 x_i^2 + \cdots + a_{t-1} x_i^{t-1}$$

$$z_i = f'(x_i) = r + b_1 x_i + b_2 x_i^2 + \cdots + b_{t-1} x_i^{t-1}$$

Publish commitments

$$c_0 = g^s h^r$$

$$c_1 = g^{a_1} h^{b_1}$$

$$\ldots$$

$$c_{t-1} = g^{a_{t-1}} h^{b_{t-1}}$$

Party i receives share (x_i, y_i, z_i) and verifies (mod p) that:

$$c_0 \cdot c_1^{x_i} \cdot c_2^{x_i^2} \cdot \ldots \cdot c_{t-1}^{x_i^{t-1}} =$$

$$= \left(g^s h^r\right) \cdot \left(g^{a_1} h^{b_1}\right)^{x_i} \cdot \left(g^{a_2} h^{b_2}\right)^{x_i^2} \cdot \ldots \cdot \left(g^{a_{t-1}} h^{b_{t-1}}\right)^{x_i^{t-1}} =$$

$$= g^s \cdot g^{a_1 x_i} \cdot g^{a_2 x_i^2} \cdot \ldots \cdot g^{a_{t-1} x_i^{t-1}} \cdot h^r \cdot h^{b_1 x_i} \cdot h^{b_2 x_i^2} \cdot \ldots \cdot h^{b_{t-1} x_i^{t-1}} =$$

$$= g^{s + a_1 x_i + a_2 x_i^2 + \ldots + a_{t-1} x_i^{t-1}} \cdot h^{r + b_1 x_i + b_2 x_i^2 + \ldots + b_{t-1} x_i^{t-1}} =$$

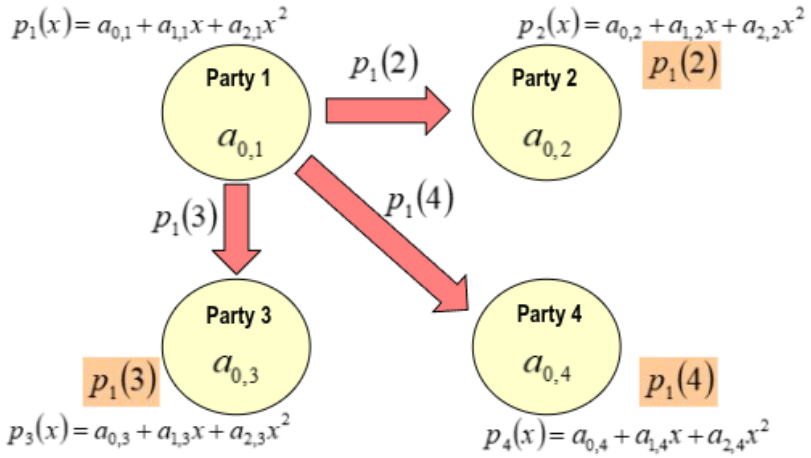
$$= g^{y_i} h^{z_i}$$

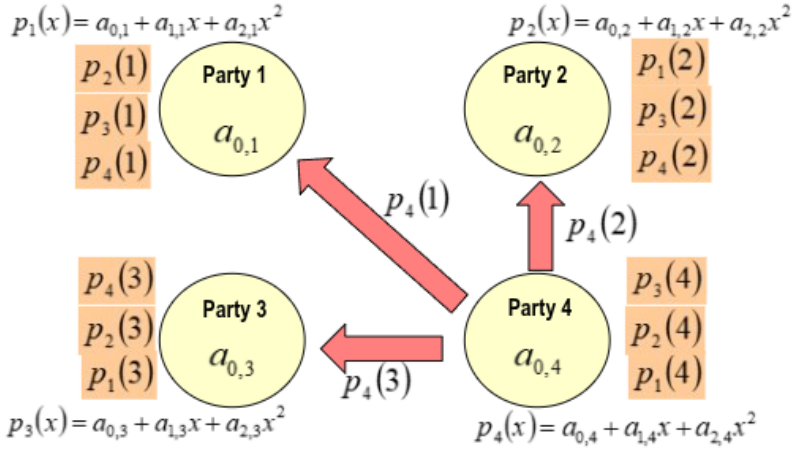
We must now generate $h = g^w$ such that nobody knows w, neither the dealer, TTP or the parties.

Distributed Key Generation

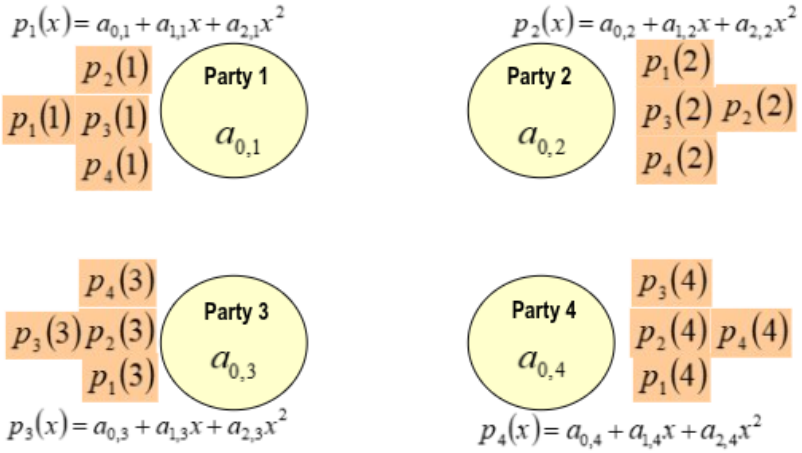
We must generate (Pub_k, $Priv_k$) pair s.t. all know Pub_k but nobody knows $Priv_k$ but if necessary it can be revealed later on.

This scheme is called DKG (Distributed Key Generation) the basic one is Pedersen DKG.
Example (3,4)
Step 1: each party independently generates random secret $a_{0,i}$.
Step 2: generates random polynomial of degree (t-1) with known term $a_{0,i}$.
Step 3: securely exchanges share of own poly with every other party.

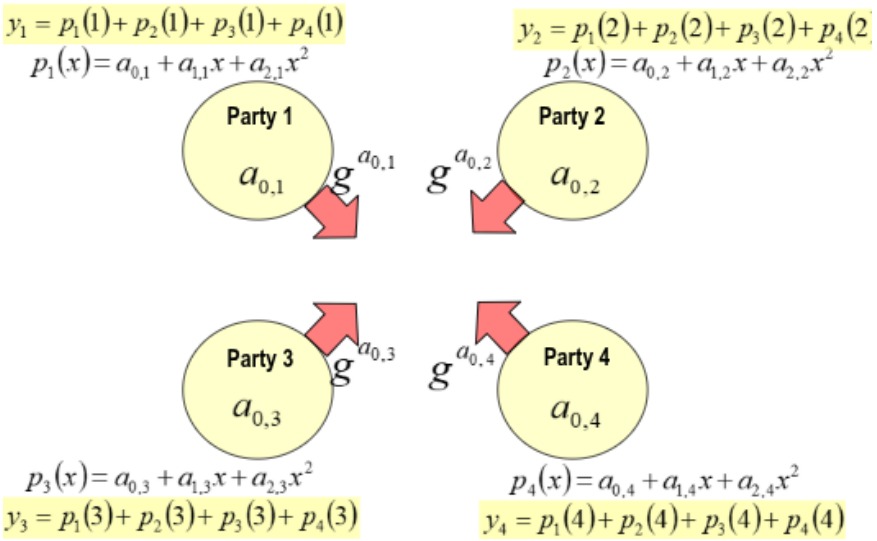




Step 4: each party computes “self” share



Step 5: each party computes “full” share (and keeps it))
Step 6: and broadcast commitment for $a_{0,i}$.



We have distributely built a random polynomial P(x) where a random S is unknown to ALL...

$$\begin{aligned} p_1(x) &= a_{0,1} + a_{1,1}x + a_{2,1}x^2 \\ p_2(x) &= a_{0,2} + a_{1,2}x + a_{2,2}x^2 \\ p_3(x) &= a_{0,3} + a_{1,3}x + a_{2,3}x^2 \\ p_4(x) &= a_{0,4} + a_{1,4}x + a_{2,4}x^2 \\ \sum \quad & \\ P(x) &= s + \alpha_1 x + \alpha_2 x^2 \end{aligned}$$

Being: $s = \sum_{i=1}^{4} a_{0,i}$, $\alpha_1 = \sum_{i=1}^{4} a_{1,i}$, $\alpha_2 = \sum_{i=1}^{4} a_{2,i}$

... but where each party has a valid SHARE of such random secret S...

$$y_i = p_1(i)+p_2(i)+p_3(i)+p_4(i) = P(i)$$

... and (our target!) where each party can compute g^s using the broadcast committments!

$$g^{a_{0,1}} \cdot g^{a_{0,2}} \cdot g^{a_{0,3}} \cdot g^{a_{0,4}} = g^{a_{0,1}+a_{0,2}+a_{0,3}+a_{0,4}} = g^s$$

Threshold and policy-based cryptography (ELGamal)
Public Key Encryption with DLOG:
1976: Diffie-Hellman used DLOG only for key agreement.
1977: RSA solved public key encryption, but with different assumption.
1985: Public Key Encryption with DLOG turning DH key agreement into an encryption protocol using a construction named Columbus egg.

Using DLOG cryptosystem is convenient cause Elliptic Curve Groups are used, and we want EC cryptography because it scales better than ordinary modular groups with increased security parameter.

Symm key equiv	modulus size	Elliptic Curve
80 bits	1024 bits	163 bits
128 bits	3072 bits	283 bits
256 bits	15360 bits	571 bits

ElGamal

Is a Public Key system created by ElGamal in 1985 based on Discrete log. It's a Dlog based Public Key (asymmetric) cryptosystem.

Inspired by DH because it turns DH idea into an asymmetric cipher.

p large prime — **All operations mod p**

g Group generator

Creative way to "reuse" Diffie-Hellman!

g^s, g^r : known to all (g^s public, g^r in ciphertext)

s: known only by RX (secret key)

r: known only by TX (random)

g^{sr} : No way for anybody else to compute

s Private key

$h = g^s$ Public key

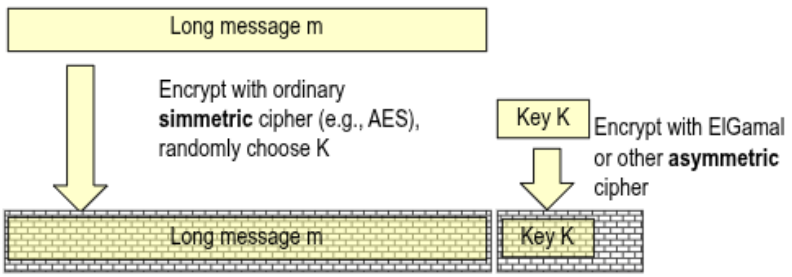
r random

Encrypt $(R, c) = (g^r, m \cdot h^r)$

Decrypt $m = c \cdot R^{-s} = \frac{c}{(g^r)^s} = \frac{m \cdot h^r}{g^{rs}} = \frac{m \cdot g^{sr}}{g^{rs}}$

Asymmetric ciphers:

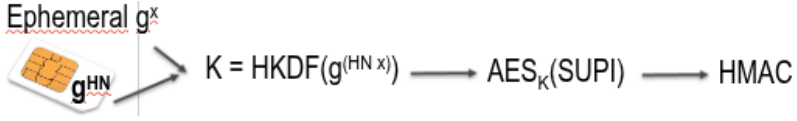
Message must be point in group, so a number between 1 and p-1 if using mod p integers, but messages can be much longer than that so we use an Hybrid encryption: encrypt the message with an ordinary symmetric cipher and randomly choose K then encrypt the key using ElGamal or other asymmetric cipher.



Today El-Gamal-type of crypto is used in many fields:

ECIES (Elliptic Curve Integrated Encryption Scheme) = Hybrid Encryption in 5G

SUPI is Subscriber Concealed Identifier.



Threshold ElGamal

Decryption when threshold # receivers cooperate, so the message can be read only by t cooperating parties and nobody has the private key. The first idea is to distribute shares of private key s and reconstruct s when decryption is needed but this only works once cause once the secret key is reconstructed anything can be decrypted. The goal is to decrypt one message and guarantee confidentiality for any other message.

Encrypt $\left(g^{r_1}, m_1 \cdot h^{r_1}\right)$

Decrypt $m_1 = \frac{c}{\left(g^{r_1}\right)^s}$ Decrypt $m_2 = \frac{c_2}{\left(g^{r_2}\right)^s}$

To be able to decrypt m_1 and not m_2 and to compute denominator for r_1 such that computation for a different r_2 is not revealed we use the property of exponentiation:
$A^x = A^{x1}\ A^{x2} = A^{x1+x2}$

We need to interpolate shares at exponent

Usual polynomial

$$p(x) = s + a_1 x + a_2 x^2 + \cdots + a_{t-2} x^{t-2} + a_{t-1} x^{t-1}$$

Usual shares

$$(x_i, y_i) \quad y_i = p(x_i)$$

Usual reconstruction formula...

$$s = \sum_{shares\ x_i} y_i \Lambda_{x_i} \quad with \quad \Lambda_{x_i} = \Lambda_{x_i}(0) = \prod_{shares\ x_k \neq x_i} \frac{-x_k}{x_i - x_k}$$

What if done at exponent?

$$\prod A^{y_i \Lambda_{x_i}} = A^{\sum y_i \Lambda_{x_i}} = A^s$$

Each party owns one share $(x_i, y_i) \quad y_i = p(x_i)$

Party gets g^r from ciphertext $\left(g^r, m \cdot h^r\right)$

Computes as usual Lagrange coefficients

- Depend only on (known) x-axis share values $\Lambda_{x_i} = \prod_{shares\ x_k \neq x_i} \frac{-x_k}{x_i - x_k}$

Computes exponent share $\left(g^r\right)^{y_i \Lambda_{x_i}}$

Sufficient # of shares permit to reconstruct decryption term $\prod \left(g^r\right)^{y_i \Lambda_{x_i}} = \left(g^r\right)^{\sum y_i \Lambda_{x_i}} = \left(g^r\right)^s = g^{rs}$

Finally factor away decryption term $\frac{m \cdot h^r}{g^{rs}} = \frac{m \cdot g^{sr}}{g^{rs}} = m$

Threshold signature (use-case: RSA)

Any t out of n members of a group can sign a message like validity of (signed) message endorsed by multiple “notaries” or group member certified by other t members or place trust

in more than one Certification Authority, etc.
If less than t members, impossible to forge a signature.
Further requirements are to reuse existing signature approaches and that size shouldn't blow up with t.

RSA signature
Large prime p, q where N = pq, φ(N) = (p-1)(q-1), then we pick e such that is coprime to φ(N) and compute $d = e^{-1} \bmod \varphi(N)$. To sign a message we do $[m, H(m)^d]$ and to verify the signature we make sure that $H(m) = (H(m)^d)^e$.

Threshold RSA

Dealer: $f(x) = d + a_1 x + a_2 x^2 + \cdots + a_{t-2} x^{t-2} + a_{t-1} x^{t-1}$

Share of P_i $(x_i, y_i = f(x_i)) \bmod \phi(N)$

Message to be signed $m, H(m)$

Signature share $H(m)^{y_i \Lambda_{x_i}} \bmod N$

Valid signature $\prod H(m)^{y_i \Lambda_{x_i}} = H(m)^{\sum y_i \Lambda_{x_i}} = H(m)^d$

Party i must compute $H(m)^{y_i \Lambda_{x_i}} \bmod N$

Where $\Lambda_{x_i} = \prod_{shares\ x_k \neq x_i} \frac{-x_k}{x_i - x_k} = \frac{\alpha_{x_i}}{\beta_{x_i}}$

Hence $H(m)^{y_i \Lambda_{x_i}} = H(m)^{y_i \alpha_{x_i} \beta_{x_i}^{-1}} \bmod N$

where $\beta_{x_i} \cdot \beta_{x_i}^{-1} = 1 \bmod \phi(N)$

This doesn't work cause we cannot compute the inverse without φ(N) so we need factorization but with factorization the RSA breaks, also if beta happens to be even (may be), the inverse doesn't exist.

We must avoid inverses
Assume $x_i = i$, and L players. The worst case is interpolation on all L shares:

$$\Lambda_i(x) = \prod_{shares\ k \neq i} \frac{x-k}{i-k} = \frac{something(x)}{(i-1)(i-2)\cdots(i-(i-1))(i-(i+1))\cdots(i-L)}$$

Denominator surely divides i!(L-1)! which in turn surely divides L! (all ! = factorial) and in consequence:

$$L! \cdot \Lambda_i(x) = \text{surely integer} = \overline{\Lambda_i}(x)$$

Dealer: $f(x) = d + a_1 x + a_2 x^2 + \cdots + a_{t-2} x^{t-2} + a_{t-1} x^{t-1}$

Share i: $(i, y_i = f(i)) \mod \phi(N)$

Compute signature share as: $H(m)^{y_i \overline{\Lambda_{x_i}}} \mod N$

- no inverse needed anymore
- Largange are NOW integers

Construct "signature" (?!) as

$$\prod H(m)^{y_i \overline{\Lambda_{x_i}}} = H(m)^{L! \sum y_i \Lambda_{x_i}} = H(m)^{d \cdot L!} \mod N$$

Now we have an extra factor L! at the exponent and we need to get rid of that.
We know that in the RSA command modulus attack Alice encrypt message m with public key e_a and Bob encrypt the same message m with different public key e_b, the module N is the same and the $\gcd(e_a,e_b)=1$, Then message can be easily decrypted using extended euclidean algorithm: find r,s such that $e_a r + e_b s = \gcd(e_a,e_b)=1$.
Given encrypted messages m^{ea} and m^{eb} we don't need the decrepit key cause:

$$\left(m^{e_a}\right)^r \cdot \left(m^{e_b}\right)^s = m^{e_a \cdot r + e_b \cdot s} = m$$

let $L! = \Delta$

we have $H(m)^{d \cdot L!} = H(m)^{d \cdot \Delta}$

we want $H(m)^d = y$

but we have $H(m)^{d \cdot \Delta} = y^{\Delta}$

we also have $H(m) = H(m)^{de} = y^e$

We have cast our problem as an RSA common modulus attack!

if e and Δ are coprime, we can apply "attack"

$\Rightarrow$ and hence derive our signature $H(m)^d = y$

If public key e is a prime larger than L, then this surely works, besides this RSA signature remains perfectly standard.

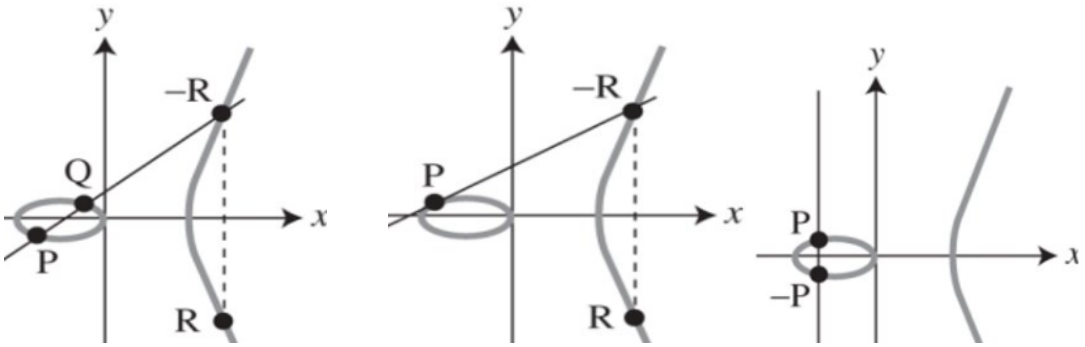
Elliptic Curve Crypto
Computational security = Hard problems
Must be easy on one way and hard on the other way
Example: discrete logarithm in Z*(p)
p is a prime number, given x finding g^x mod p is easy but given g^x finding x mod p is hard.
This isn't restricted to Z*(p) but can be any cyclic group Gp.

The same problem (Dlog) has different hardness in different groups:
Z*(p): exp(O(n^(⅓)))
EC: exp(O(n^(½)))
(Z(p), +): poly!
EC scales well with increased security parameter that's why there are so many new EC ciphers.

In reality these are not Elliptic curves, they are special cubic curves named from elliptic integrals, the general (Weierstrass) expression is $y^2=x^3+ax+b$ where $4a^3+27b^2 \neq 0$

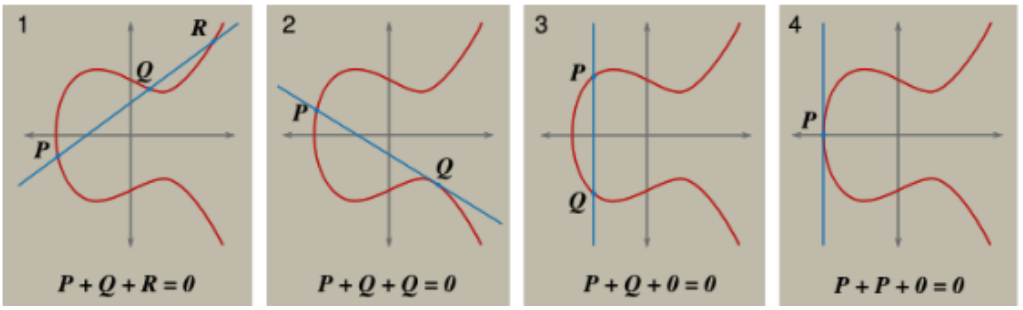
To come out with a group we also need an operation o (also called + but its not a plus), we want one that makes P(point of group) o Q(point of curve)= R and R belongs to the curve.

P+Q, P+P, and P-P



We complete the curve by adding point O = -O to infinity so that P+O=P, O is (additive) zero for group of EC points.

If three points of an elliptic curve lay on a same line, their sum is 0

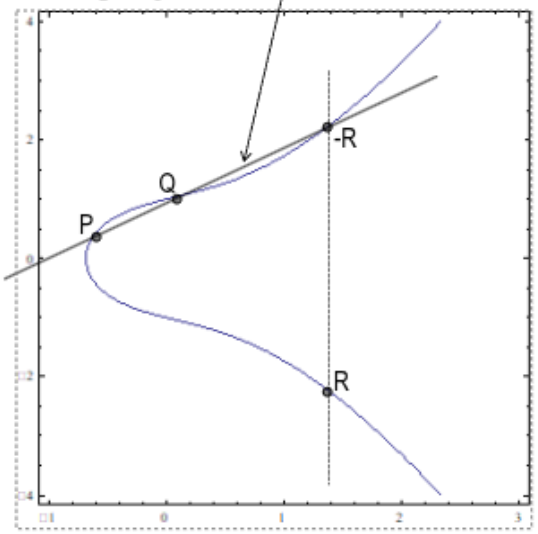


The algebraic expression can be derived from the geometric interpretation

$$P=(x_1,y_1)$$
$$Q=(x_2,y_2)$$
$$R=P+Q=(x_3,y_3)$$
$$x_3=\lambda^2-x_1-x_2$$
$$y_3=\lambda(x_1-x_3)-y_1$$
$$\lambda=\begin{cases}\dfrac{y_2-y_1}{x_2-x_1} & P\neq Q\\ \dfrac{3x_1^2+a}{2y_1} & P=Q\end{cases}$$

P different from Q example

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) = \lambda(x - x_1) + y_1$$



$$y^2 = x^3 + ax + b$$

$$(\lambda(x - x_1) + y_1)^2 = x^3 + ax + b$$

$$0 = x^3 - \lambda^2 x^2 + \ldots$$

recall :

$$(x - x_1)(x - x_2)(x - x_3) =$$

$$= x^3 - (x_1 + x_2 + x_3)x^2 + \ldots$$

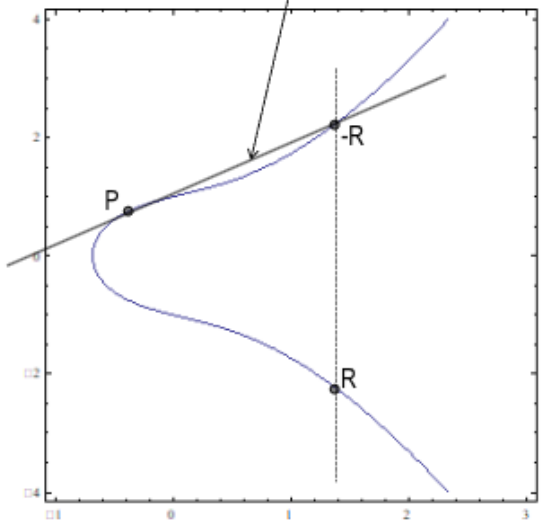
hence

$$\lambda^2 = x_1 + x_2 + x_3 \rightarrow x_3 = \lambda^2 - x_1 - x_2$$

y_3 now trivial from line eq.

P equal to Q example

$$y = y_1 + y'_1(x - x_1) = \lambda(x - x_1) + y_1$$



$$y^2 = x^3 + ax + b$$

$$2y\frac{dy}{dx} = 3x^2 + a \rightarrow \frac{dy}{dx} = \frac{3x^2 + a}{2y}$$

$$\lambda = y'_1 = \frac{3x_1^2 + a}{2y_1}$$

Follow up as before...

The geometric interpretation is over real numbers but our application needs to be over modular integers so we need to define a “curve” as $E_p(x,y)$ such that x,y are integers in Zp and y^2 mod p = x^3+ax+b mod p, we have now a finite field so a finite # of points.

Example:

- **All points:**
  - pairs (i,j), i in [0,4], j in [0,4] s.t. $j^2=i^3+i+1 \bmod 5$
  - And O

$$E(Z_5)=\{O,(0,1),(0,4),(2,1),(2,4),(3,1),(3,4),(4,2),(4,3)\}$$

- Result: 9 points
- Let P=(0,1)
  - k P=P+P+…+P generates all the points

$$P=(0,1)$$
$$2P=(0,1)+(0,1)=(4,2)$$
$$3P=(4,2)+(0,1)=(2,1)$$
$$4P=(2,1)+(0,1)=(3,4)$$
$$5P=(3,4)+(0,1)=(3,1)$$
$$6P=(3,1)+(0,1)=(2,4)$$
$$7P=(2,4)+(0,1)=(4,3)$$
$$8P=(4,3)+(0,1)=(0,4)$$
$$9P=(0,4)+(0,1)=O$$

(0,1) -> $1^2=0^3+0+1 \bmod 5$ -> 1=1 its ok
(0,2) -> $2^2=0^3+0+1 \bmod 5$ -> 4=1 not ok
(0,3) -> $3^2=0^3+0+1 \bmod 5$ -> 4=1 not ok
(0,4) -> $4^2=0^3+0+1 \bmod 5$ -> 1=1 its ok
example of computations:

$(0,1)+(0,1)$
step1: compute λ
$$\lambda=\frac{3x_1^2+a}{2y_1}=1\cdot 2^{-1}=1\cdot 3=3 \mod 5$$
step2: compute x_3
$$x_3=\lambda^2-x_1-x_1=3^2=4 \mod 5$$
step3: compute y_3
$$y_3=\lambda(x_1-x_3)-y_1=3(0-4)-1=2 \mod 5$$
result: $(4,2)$

$(4,2)+(0,1)$
step1: compute λ
$$\lambda=\frac{y_2-y_1}{x_2-x_1}=1\cdot 4^{-1}=4 \mod 5$$
step2: compute x_3
$$x_3=\lambda^2-x_1-x_1=2 \mod 5$$
step3: compute y_3
$$y_3=\lambda(x_1-x_3)-y_1=4(4-2)-2=1 \mod 5$$
result: $(2,1)$

($E(Z_p)$, +(not real addition, but operation)) is a group where Addition (is the name of the operation not a real plus) is closed on set E, Addition is commutative, O is identity with regard to addition, Every point P in E has inverse -P with regard to addition and the Associative property so this group is an Abelian group.

Easy / hard operation: [k]P where k is an integer mod group order and P is the elliptic curve point on group
Given P and k, it's easy to compute [k]P,
Given P and k[P], it's hard to compute k
This is the equivalent to Dlog problem on Zp
It depends on the chosen curve so its recommended to use standard curves that are not only on Zp but also in GF(2^m): binary curves so more convenient, more recently also Edwards curves. There are also different EC curves that satisfy different assumptions.

Example

Goal: compute [1437]P

Express in bits

- $1437_{10}=10110011101_2$

Initialize first line

- Differs if 0 or 1

Start from 2° lsb to msb

- For every bit
  - Double;
  - If 1: add to result

Complexity:

- O(nbit) x2
- O(nbit/2) additions

lsb-> msb	*double*	*result*
1	$1P$	$+1P = P$
0	$2P$	.
1	$4P$	$+4P = 5P$
1	$8P$	$+8P = 13P$
1	$16P$	$+16P = 29P$
0	$32P$	.
0	$64P$	.
1	$128P$	$+128P = 157P$
1	$256P$	$+256P = 413P$
0	$512P$	.
1	$1024P$	$+1024P = 1437P$

Elliptic Curves crypto

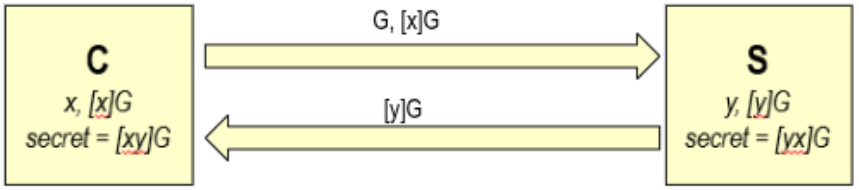
Invented in 1985 by Koblitz and Miller independently, the security is based on hardness of given P and [k]P it's hard to compute k and of ECDLP (Elliptic Curve Discrete Log Problem), uses much shorter keys than RSA/DLP and better scaling.

The most used algorithms use EC, it was also adopted in TLS, IPsec, etc, with algorithms like AES 128/256, Key Exchange: ECDH, Digital Signature: ECDSA, Hashing: SHA-256.

ECDH

Exactly as DH but using an EC group.

(prime p of Zp, curve parm a and b, group order q)



ECDSA (Elliptic Curve Digital Signature Algorithm)

EC version of DSA, invented by Vanstone and proposed NIST in august 91, its a variant of ElGamal and Dlog based.

DSA recap

Multiplicative group Z*(p), p>2 prime, there is a large prime q in group order and usually p=2q+1, the generator is chosen such that g^x mod p spans all group elements, the private and public keys are: d, y = g^d mod p

Signature for message m

Random k in (1,q)

Compute r = (g^k mod p) mod q

Compute s = k^{-1} (H(m)+ d r) mod q

Signature: (r,s)

(where k^{-1} and r are the nonces, H(m) is the hash of the msg and d is the private key)

Verification

Compute $u_1 = s^{-1} H(m) \bmod q$

Compute $u_2 = s^{-1} r \bmod q$

Verify $((g^{u1} y^{u2}) \bmod p) \bmod q = r$

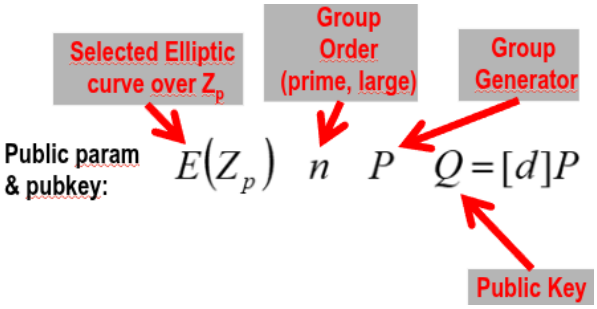
$$g^{u_1} y^{u_2} = g^{u_1} g^{d \cdot u_2}$$

$$u_1 + d \cdot u_2 = \frac{H(m)k}{H(m) + rd} + \frac{d \cdot r \cdot k}{H(m) + rd} = k$$

so it becomes $g^k = r$

ECDSA setup

Private key d = random integer in [1, n-1] while Q is a point



ECDSA: signature generation

k in [1, n-1] random integer must be unique per each signature and unpredictable,

compute kP = (x_1, y_1) this is a EC point,

compute r = $x_1 \bmod n$

compute $k^{-1} \bmod n$

Hash function H(.) like SHA

the signature is: (r,s) with $s = k^{-1}(H(m)+dr)$

ECDSA: verification

from message m compute H(m)

from s compute $w = s^{-1} \bmod n$

compute $u_1 = H(m)w \bmod n$

compute $u_2 = rw$

compute elliptic point: $u_1P + u_2Q = (x_0, y_0)$

compute $v = x_0 \bmod n$ and compare it to r to see if it's the same

it works because:

$$w = s^{-1} = k[H(m) + rd]^{-1}$$

$$u_1 P + u_2 Q =$$

$$= \frac{H(m)k}{H(m) + rd} P + \frac{r \cdot k}{H(m) + rd} Q =$$

$$= \frac{H(m)k}{H(m) + rd} P + \frac{rd \cdot k}{H(m) + rd} P = kP$$

k is never disclosed!

If k is predicted then the private key would be disclosed:

$$\text{from } (r,s) \text{ and knowledge of } k:$$
$$s = k^{-1}\left[H(m)+rd\right] \mod n$$
$$sk = H(m)+rd \mod n$$
$$\left[sk - H(m)\right]r^{-1} = d \mod n$$

If k is repeated then the private key would be disclosed:

$$\text{from } (r,s_1),(r,s_2) \text{ - same unknown } k = \text{same } r$$
$$\begin{cases} s_1 = k^{-1}\left[H(m_1)+rd\right] \\ s_2 = k^{-1}\left[H(m_2)+rd\right] \end{cases} \mod n \quad \Rightarrow \quad \begin{cases} k = s_1^{-1}\left[H(m_1)+rd\right] \\ k = s_2^{-1}\left[H(m_2)+rd\right] \end{cases} \mod n$$
$$s_1^{-1}\left[H(m_1)+rd\right] = s_2^{-1}\left[H(m_2)+rd\right]$$
$$s_2H(m_1) + s_2rd = s_1H(m_2) + s_1rd$$
$$d(s_2 - s_1)r = s_1H(m_2) - s_2H(m_1)$$
$$d = \left[s_1H(m_2) - s_2H(m_1)\right]r^{-1}(s_2 - s_1)^{-1} \mod n$$

Bilinear Maps

Bilinear Maps are the tool of pairing-based crypto they establish relationships between cryptographic groups, make DDH easy in one of the process and let's u solve CDH once.
Let G_1, G_2, and G_t be cyclic groups of the same order, a Bilinear Map from G_1 x G_2 to G_t is a function *e*: G_1 x G_2 -> G_t such that for all u in G_1, v in G_2, a AND b in Z then
$e(u^a, v^b) = e(u, v)^{ab}$

Bilinear maps are called pairing because they associate pairs of elements from G_1 and G_2 with elements in G_t, this definition admits degenerate maps which map everything to the identity of G_t.

Let g_1 and g_2 be generators of G_1 and G_2 respectively.
The map *e* is an admissible bilinear map if $e(g_1, g_2)$ generates G_t and *e* is efficiently computable.
(from now on bilinear maps will implicity means admissible bilinear map)

G_1 G_2 and G_t are isomorphic to one another since they have the same order and are cyclic, they are different group in the sense that we represent the elements and compute the operations differently, however normally $G_1 = G_2$, and denote both by G.
If $G = G_t$ called that is a self-binary map, is very powerful but there aren't known examples.
Sometimes G is written additively and G_t is written multiplicatively (we will use the last one).

Typically G is an elliptic curve defined by $y^2 = x^3 + 1$ over the finite field Fp, we can also have Supersingular curves and MNT curves that have different performance implications. More generally G is typically an abelian variety over some field and G_t is normally over a finite field.

Weil pairing and Tate pairing are more or less only known examples, very complicated math and non-trivial to compute but they use the Miller's algorithm.

Decisional Diffie-Hellman
(informally is given g^x and g^y (u cant compute g^{xy}, basic DH) and something, this something is either g^{xy} or either something random with a probability of 50%)
Let G be a group of order q with generator g. The advantage of an probabilistic algorithm A in solving the Decisional Diffie-Hellman (DDH) problem in G is

$$\mathrm{Adv}^{\mathrm{DDH}}_{\mathcal{A},G} = \left|\mathrm{P}\left[\mathcal{A}(g, g^a, g^b, g^{ab}) = 1\right] - \mathrm{P}\left[\mathcal{A}(g, g^a, g^b, g^z) = 1\right]\right|$$

The basic property of bilinear map is making DDH easy in G, with a bilinear map
e: G x G -> G_t and given g, g^a, g^b, g^c we can determine if c = ab mod q by just checking whether e(g^a, g^b) = e(g, g^c).
However if the map is from distinct groups G_1 and G_2, DDH may still be hard.

Computational Diffie-Hellman (CDH)
On the other hand CDH could still be hard in G, a bilinear map is not known to be useful for solving CDH.

Discrete Log
If there exists a bilinear map *e*: G x G -> G_t then the discrete log problem in G is no harder than the discrete log problem in G_t.
Given g in G and g^a in G, we can compute *e*(g, g) in G_t and *e*(g, g^a) = *e*(g, g)^a in G_t. Then we can use a discrete log solver for G_t to obtain a. This is called MOV reduction.

Informally, bilinear maps are useful cause they lets you cheat and solve a CDH problem, but only once, after that u are stuck in the group G_t.

Joux's 3-Party DH
Simple protocol, Let G be a group with prime order q, e: G x G -> G_t be a bilinear map, and g a generator of G. Let g^ = e(g, g) in G_t.

1. Alice picks $a \xleftarrow{R} \mathbb{Z}_q$, Bob picks $b \xleftarrow{R} \mathbb{Z}_q$, and Carol picks $c \xleftarrow{R} \mathbb{Z}_q$.
2. Alice, Bob, and Carol broadcast g^a, g^b, and g^c respectively.
3. Alice computes $e(g^b, g^c)^a = \hat{g}^{abc}$, Bob computes $e(g^c, g^a)^b = \hat{g}^{abc}$, and Carol computes $e(g^a, g^b)^c = \hat{g}^{abc}$.

From Alice's perspective, map lets you cheat to get g^^{bc} from g^b and g^c, then regular exponentiation gets you the rest of g^^{abc}. e cannot be used to get g^^{abc} from g^a, g^b, g^c, only one cheat allowed.

Boneh and Franklin's IBE Scheme
Let G be a group with prime order q, e: G x G -> G_t be a bilinear map, and g a generator of G. Let g^ = e(g, g) in G_t.

Let h_1: {0,1}* -> G and h_2: G_t -> {0,1}* be hash functions.

PKG picks $s \xleftarrow{R} \mathbb{Z}_q$. Then g^s is the public key of PKG.

Encryption

If Alice wants to send a message m to Bob, she picks $r \xleftarrow{R} \mathbb{Z}_q$ then computes the following.

$$\begin{aligned}\text{Encrypt}\,(g, g^s, \text{“Bob”}, m) &= (g^r, m \oplus h_2(e(h_1(\text{“Bob”}), g^s)^r) \\ &= (g^r, m \oplus h_2(e(h_1(\text{“Bob”}), g)^{rs})\end{aligned}$$

Making a Private Key

PKG may compute the private key of Bob as follows.

$$\text{MakeKey}\,(s, \text{“Bob”}) = h_1(\text{“Bob”})^s$$

Decryption

Given an encrypted message $(u, v) = (g^r, m \oplus h_2(e(h_1(\text{“Bob”}), g)^{rs})$ and a private key $w = h_1(\text{“Bob”})^s$, Bob may decrypt as follows.

$$\begin{aligned}\text{Decrypt}\,(u, v, w) &= v \oplus h_2(e(w, u)) \\ &= m \oplus h_2(e(h_1(\text{“Bob”}), g)^{rs}) \\ &\quad \oplus h_2(e(h_1(\text{“Bob”})^s, g^r)) \\ &= m \oplus h_2(e(h_1(\text{“Bob”}), g)^{rs}) \\ &\quad \oplus h_2(e(h_1(\text{“Bob”}), g)^{rs}) \\ &= m\end{aligned}$$

Let t be the discrete log of h_1(“Bob”) base g, The situation is like 3-party DH where Alice has public g^r private and r, PKG has public g^s and private s, Bob has public g^t and unknown t.

$e(h_1(\text{“Bob”}), g)^{rs} = e(g^t, g)^{rs} = \hat{g}^{rst}$ is like session key for encryption.

Alice and PKG could compute g^rst just like in Joux's scheme, Bob once PKG sends g^{st} can compute e(g^{st}, g^r) = g^rst. If it was g^st instead of g^{st} Bob couldn't apply e anymore.

Linear Secret Sharing & Access Control Matrix

Revisiting Shamir Scheme

We have a secret s and

Share i: $y_i = s + a_1 x_i + a_2 x_i^2 + \cdots + a_{t-2} x_i^{t-2} + a_{t-1} x_i^{t-1}$

Vector interpretation: scalar product

$$y_i = [1, x_i, x_i^2, \cdots, x_i^{t-2}, x_i^{t-1}] \bullet [s, a_1, a_2, \cdots, a_{t-2}, a_{t-1}]$$

The coefficients a_i are random so we can call them r_i.

Shamire scheme can also be represented in form of matrix Ax = b

$$\begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \end{pmatrix} \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$$

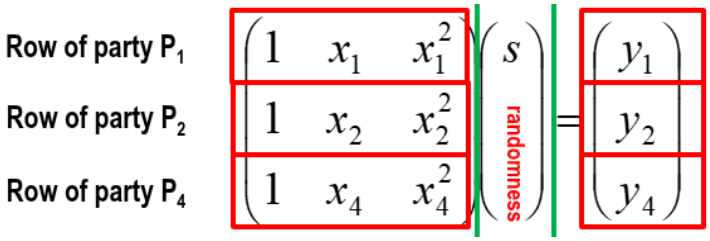
It is a special matrix: the Vandermonde matrix. For reconstructing the secret we use a linear system e.g. with 3 shares out of 4, previously we used the Lagrange formula.

$$\begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_4 & x_4^2 \end{pmatrix} \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_4 \end{pmatrix} \qquad \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_4 & x_4^2 \end{pmatrix}^{-1} \begin{pmatrix} y_1 \\ y_2 \\ y_4 \end{pmatrix}$$

$$\begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_4 & x_4^2 \end{pmatrix}^{-1} = \begin{pmatrix} \frac{x_2 x_4}{(x_1 - x_2)(x_1 - x_4)} & \frac{x_1 x_4}{(x_2 - x_1)(x_2 - x_4)} & \frac{x_1 x_2}{(x_4 - x_1)(x_4 - x_2)} \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{pmatrix}$$

$$s = y_1 \frac{x_2 x_4}{(x_1 - x_2)(x_1 - x_4)} + y_2 \frac{x_1 x_4}{(x_2 - x_1)(x_2 - x_4)} + y_4 \frac{x_1 x_2}{(x_4 - x_1)(x_4 - x_2)}$$

An alternative interpretation can be:



Each row is now a vector v_i, those vectors span a 3D space: $c_1v_1 + c_2v_2 + c_3v_3$.

$$c_1 \begin{pmatrix} 1 \\ x_1 \\ x_1^2 \end{pmatrix} + c_2 \begin{pmatrix} 1 \\ x_2 \\ x_2^2 \end{pmatrix} + c_3 \begin{pmatrix} 1 \\ x_4 \\ x_4^2 \end{pmatrix}$$

The vector [1,0,0] is included in the span, so we solve it with a linear system
$c_1v_1 + c_2v_2 + c_3v_3 = [1,0,0]$

$$\begin{pmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_4 \\ x_1^2 & x_2^2 & x_4^2 \end{pmatrix}\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

$$c_1 = \frac{x_2x_4}{(x_1 - x_2)(x_1 - x_4)}; \quad c_2 = \frac{x_1x_4}{(x_2 - x_1)(x_2 - x_4)}; \quad c_3 = \frac{x_1x_2}{(x_4 - x_1)(x_4 - x_2)}$$

But

$$\left\{c_1\begin{pmatrix}1 & x_1 & x_1^2\end{pmatrix} + c_2\begin{pmatrix}1 & x_2 & x_2^2\end{pmatrix} + c_3\begin{pmatrix}1 & x_4 & x_4^2\end{pmatrix}\right\}\begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix}1 & 0 & 0\end{pmatrix}\begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = s$$

$$c_1\begin{pmatrix}1 & x_1 & x_1^2\end{pmatrix}\begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} + c_2\begin{pmatrix}1 & x_2 & x_2^2\end{pmatrix}\begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} + c_3\begin{pmatrix}1 & x_4 & x_4^2\end{pmatrix}\begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = s$$

share

$$c_1y_1 + c_2y_2 + c_3y_4 = s$$

So the shamir secret sharing is treated as span problem.

Linear Secret Sharing Scheme (LSSS)
The matrix can be arbitrary:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{pmatrix}\begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$$

Amos Beimel theorem: LSSS = MSP

Trivial Secret Share is LSSS

Example: (3,3)

$$\begin{pmatrix} 1 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix} s - r_1 - r_2 \\ r_1 \\ r_2 \end{pmatrix}$$

Span program

$$c_1(1 \quad -1 \quad -1) + c_2(0 \quad 1 \quad 0) + c_3(0 \quad 0 \quad 1) = (1 \quad 0 \quad 0)$$
$$c_1 = c_2 = c_3 = 1$$

Share rec.

$$c_1(s - r_1 - r_2) + c_2 r_1 + c_3 r_2 = (s - r_1 - r_2) + r_1 + r_2 = s$$

Any LSSS is homomorphic

$$x_a = (s_a, r_{1a}, r_{2a}, \cdots) \quad y_a = (share_{1a}, share_{2a}, \cdots)$$
$$x_b = (s_b, r_{1b}, r_{2b}, \cdots) \quad y_b = (share_{1b}, share_{2b}, \cdots)$$
$$Ax_a = y_a$$
$$Ax_b = y_b$$
$$y_a + y_b = Ax_a + Ax_b = A(x_a + x_b) = A(s_a + s_b, rand, rand, \cdots)$$

Monotone Span Program (example in GF2, other fields ok)
(4 parties with 2 shares of a single party)

P_2	1	0	1	1
P_2	0	1	1	0
P_1	0	1	1	0
P_3	0	0	1	1
P_4	1	1	0	0
	0	0	0	1

The program accepts a set B iff the rows labelled by B span the target vector:

P_2	1	0	1	1	
P_2	0	1	1	0	
P_1	0	1	1	0	
P_3	0	0	1	1	
P_4	1	1	0	0	
	0	0	0	1	0 0 0 1

1 0 1 1 ⊕ 0 0 1 1

OK {P2,P4}

P_2	1	0	1	1	1 0 1 1
P_2	0	1	1	0	0 1 1 0
P_1	0	1	1	0	0 1 1 0
P_3	0	0	1	1	
P_4	1	1	0	0	
	0	0	0	1	0 0 0 1

REJECT {P1,P2}

Span Programs are analogous to Secret Sharing

P_2	1	0	1	1	s	$s+r_2+r_4$	P2
P_2	0	1	1	0	r_2	r_2+r_3	P2
P_1	0	1	1	0	r_3	r_2+r_3	P1
P_3	0	0	1	1	r_4	$s+r_2$	P3
P_4	1	1	0	0		r_3+r_4	P4

Example s=1,$r2$=$r3$=0, $r4$=1

0	P2
0	P2
0	P1
1	P3
1	P4

Slides from A. B

P_2	1	0	1	1	s	$s+r_2+r_4$	P2
P_2	0	1	1	0	r_2	r_2+r_3	P2
P_1	0	1	1	0	r_3	r_2+r_3	P1
P_3	0	0	1	1	r_4	$s+r_2$	P3
P_4	1	1	0	0		r_3+r_4	P4
	1	0	0	0		s	

{P2, P4}

LSSS and access structure

Every (monotone) access structure can be realized

$$\text{participants}: P = \{P_1, P_2, P_3, P_4\}$$

$$\text{monotone access structure}: A \subseteq 2^P$$

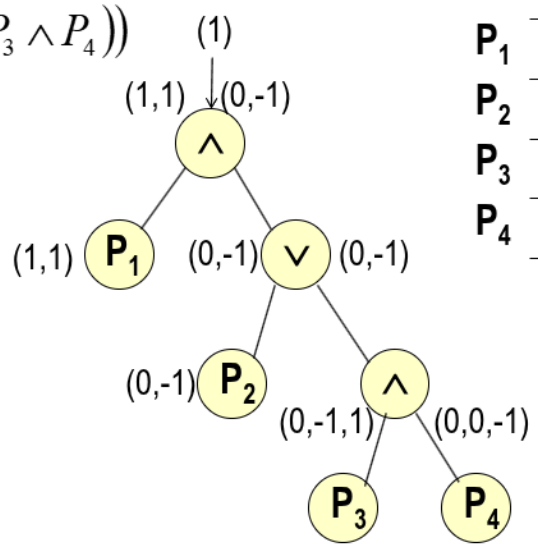
$$example: A = \{\{P_1, P_2\}, \{P_1, P_3, P_4\}\}$$

Every boolean predicate (without negation (without because we dont want that if u don't have a share u can see the secret)) may be supported: P1 and P2 …

Scheme may not be ideal cause may entail more than 1 share per partner.

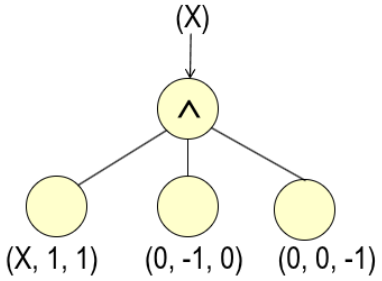
LSSS matrix from AC Predicate

$P_1 \wedge (P_2 \vee (P_3 \wedge P_4))$



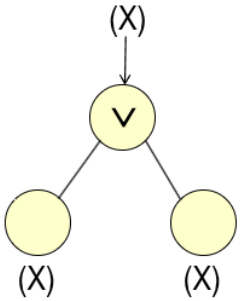
P_1	1	1	0
P_2	0	-1	0
P_3	0	-1	1
P_4	0	0	-1

Where the AND gate are formed like this:



$$\begin{pmatrix} X & 1 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

and the OR gate are formed like this:



$$\begin{pmatrix} X \\ X \end{pmatrix}$$

We also need to be careful about the padding cause in this predicate the tree creates 4 dimensions.

$$(A \wedge C \wedge D) \vee (B \wedge C)$$

$$\begin{pmatrix} A & 1 & -1 & -1 & . \\ C & 0 & 1 & 0 & . \\ D & 0 & 0 & 1 & . \\ B & 1 & . & . & -1 \\ C & 0 & . & . & 1 \end{pmatrix}$$

Mobile devices resilient to capture (MacKenzie & Reiter 2003) (NON studiare fino a X)
Capture Resilient devices are devices that cannot be used by other than the rightful owner, we assume the core of the device is a secret key that for example permit decryption. We make the assumption that the device is connected when used.
The solution of MacKenzie + Reiter is to involve a “capture protection server” in the network, the server confirms that device remains in owner’s possession before permitting usage of key, and its SW only, This server dosen’t need to be trusted, it can use two approaches one basic standard protocol and one extended that uses a (2,2) secret sharing

Scenario

Password P

Many possible attack scenarios
- **password known**
- **device stolen**
- **server cracked (no trusted server!)**

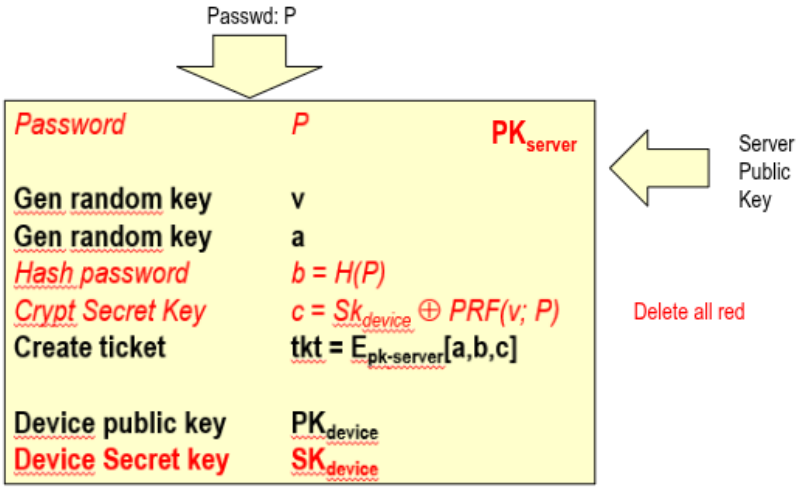
Good: Solution resilient to any crack
Better: Solution resilient to more than 1 crack!

Secret Key *SKd*
Public Key *PKd*

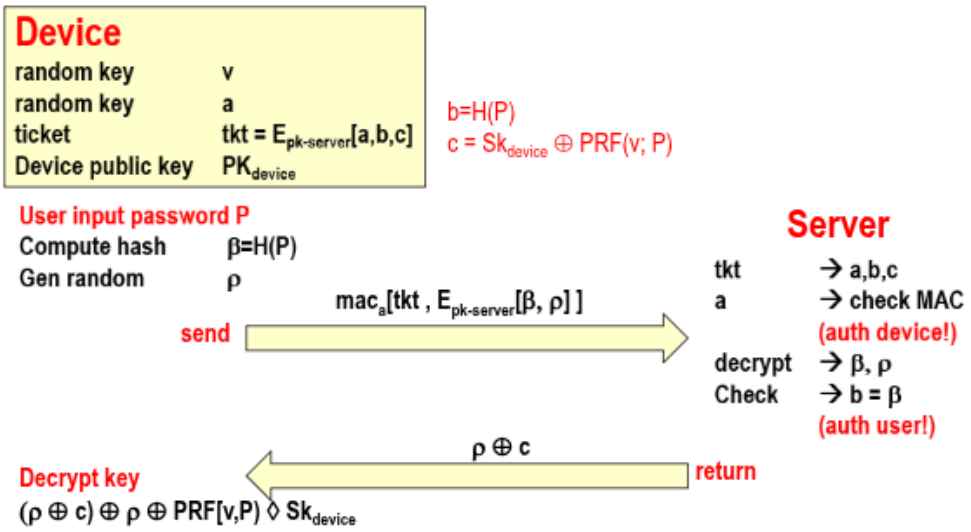
Secret Key *SKs*
Public Key *PKs*

Robust to this attacks:
Server cracked and password known, device cracked/stolen or device and server cracked but its not robust to device and password cracked.
The idea is to use “tickets”, the device has access to the network and we encrypt the device key so that it can be decrypted only via cooperation with the server.
So the idea is to send encrypted “ticket” to the server, those tickets contains authentication material for the user. Use the content of the ticket to authenticate the user and to partially decrypt device key.

Protocol: device initialization



Protocol: key retrieval



Attacks that can be done
Server cracked and password known: Key v in device is still a secret, SK is encrypted with PFR(v, pwd) so cannot be obtained.
Device cracked: attacker must send valid pwd hash and can only perform dictionary attack online but that's easy to detect.
Device and server cracked: pwd is still missing so its needed a dictionary attack offline against b.

The limitation appears when the device is stolen and pwd known: the attacker in this case can get SK and use it. the solution is to not reveal SK to the device itself using secret sharing.

Secret sharing (2,2) for RSA

$$n = p \cdot q$$

$$d \leftarrow \text{random,secretkey}$$

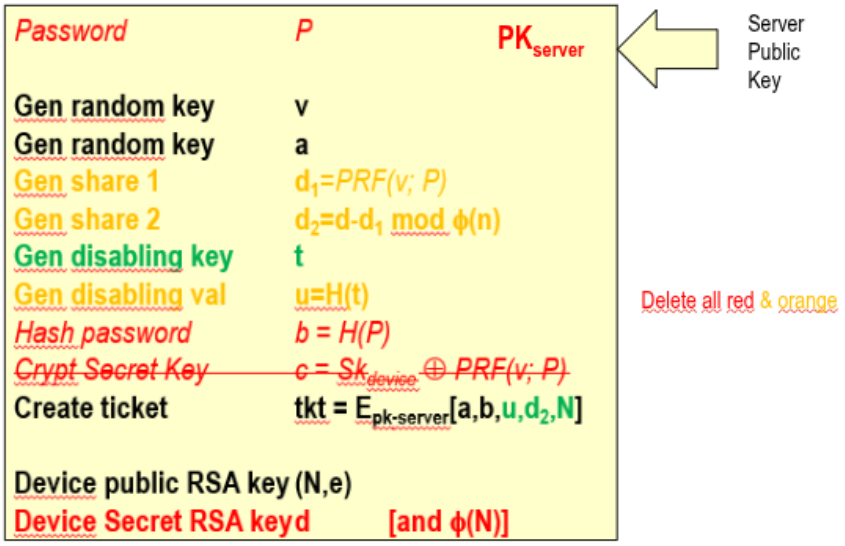
$$d_1 \leftarrow \text{random,share1}$$

$$d_2 = d - d_1 \bmod \phi(n) \leftarrow \text{share2}$$

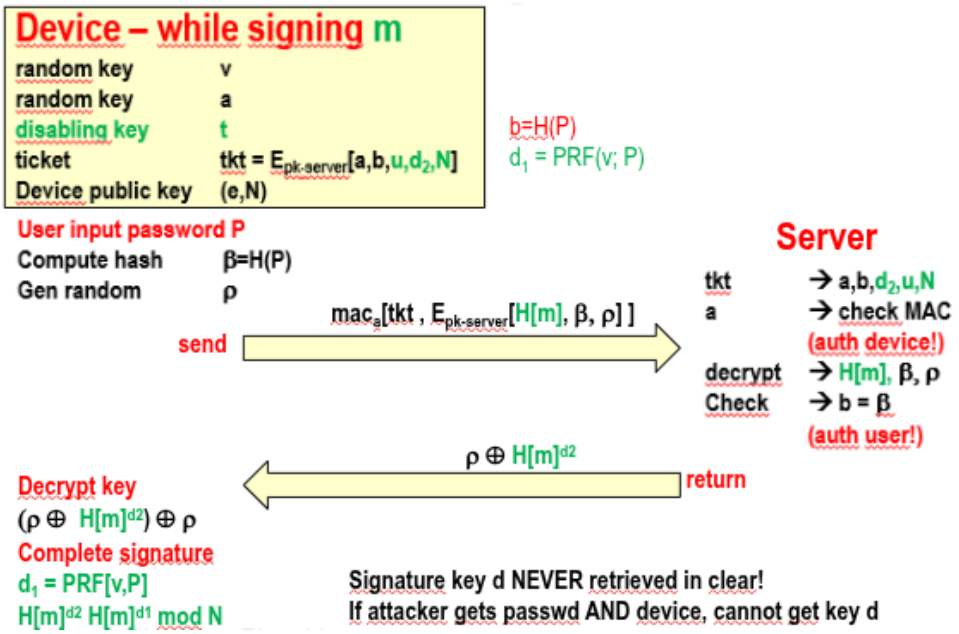
$$H(m)^{d_1} \cdot H(m)^{d_2} = H(m)^{d_1+d_2} = H(m)^{d_1+d-d_1} = H(m)^{d} \bmod n$$

Case (2,2) use trivial secret share.

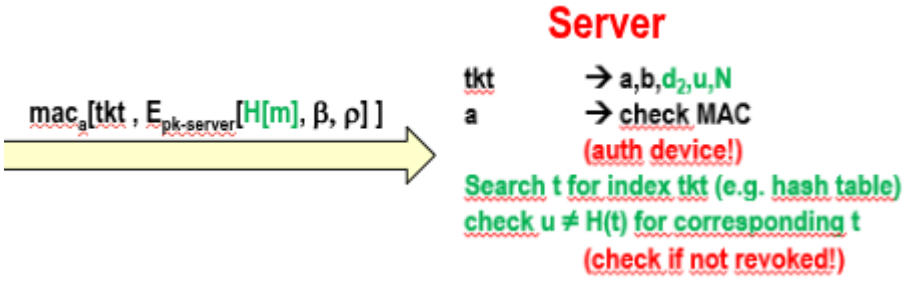
Protocol2: device initialization



Protocol2: key retrieval



Protocol2: key disabling
Sufficies to keep backup of t and tkt, and if device is stolen we just send them to the server that keep a blacklist.



(XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX)

IBE (Identity Base Encryption) discussion
Having the same ID is not sufficient, must register to PKG to get private key, the PKG (private key generator) is a powerful entity, but there can't be only one so the solution is to have distributed PKG or certificateless crypto.

ECQV (Elliptic Curve Qu-Vanstone)
Implicit certificate, identity-based. The idea is to self-certifying EC point, so don't use PKG but CA.
CA has:
private key: c
public key: C = cG
Before asking for a certificate, the device I_A generates a random positive integer r_A; then, it computes a point R_A on the chosen elliptic curve $R_A = r_A G$, and sends it to the CA. The CA extracts a random positive integer k, computes kG (another point on the elliptic curve, generates the implicit certificate P_A and the implicit signature Y_A:

$$\begin{cases} P_A = R_A + k \cdot G \\ \gamma_A = c + k \cdot H(P_A, I_A) \end{cases}$$

Now, it sends them to the device. Then, it can generate its private key $s_{v,A}$ and its public key $P_{b,A}$:

$$\begin{cases} s_{v,A} = \gamma_A + r_A \cdot H(P_A, I_A) \\ P_{b,A} = s_{v,A} \cdot G \end{cases}$$

The most important strength of ECQV is that the public key of a given device can be computed by any other third-party starting from the knowledge of the implicit certificate and the public key of the authority.

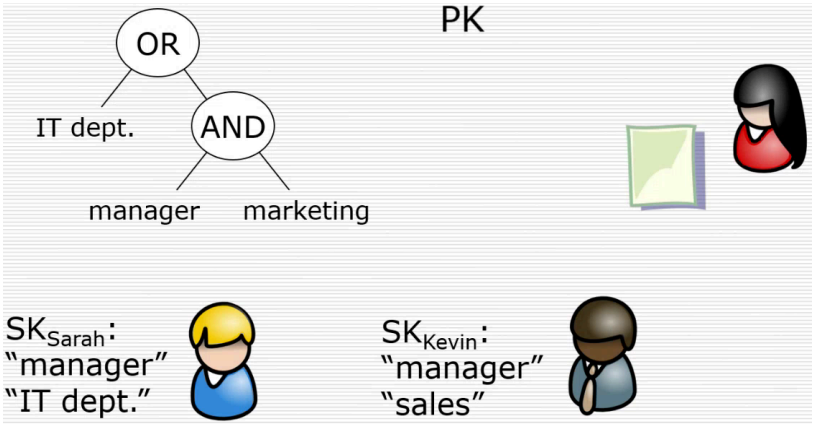
$$P_{b,A} = s_{v,A} \cdot G = C + P_A \cdot H(P_A, I_A).$$

Ciphertext-Policy, Attribute-Based Encryption (CP-ABE)
Type of identity-based encryption that uses one public key and a master private key to make more restricted private keys. Has very expressive rules for which private keys can decrypt

which ciphertexts, private keys have attributes or labels and ciphertexts have decryption policies.

Remote File Storage: we want reliability, scalability and also security
With a Server Mediated Access Control we have flexible access policies but the data is vulnerable and must trust the security of the server.
We could Encrypt the files, this way is more secure but we have a loss of flexibility and we need a new key for each file so we must be online to distribute the keys.
We want to: encrypt files for untrusted storage, setting up keys offline, no online, trusted party mediating access to files or keys, highly expressive fine grained access policies.

Access Control via CP-ABE



The Red encrypts a file where only the ones in the IT dept, or the ones either in manager and marketing can access, so the Blue one can but the Black one can't.
An important potential attack is when two combine their keys in order to access those files, but it was resolved.