CT5191
CRYPTOGRAPHY AND NETWORK SECURITY

HASH FUNCTIONS AND MESSAGE AUTHENTICATION CODES

Dr. Michael Schukat

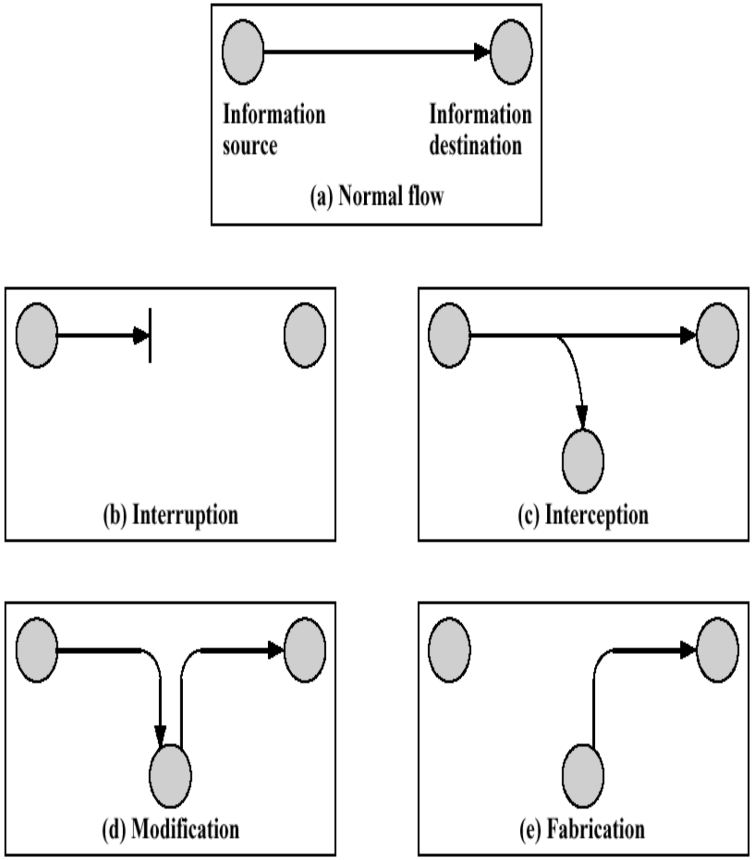
Ollscoil na Gaillimhe
University of Galway

Lecture Overview

- In the previous lectures we have covered block and stream ciphers that provide data confidentiality
- In this slide deck we focus on data integrity, i.e., "*Guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity*"
- Such integrity protection can be provided via
  - Message authentication codes
  - Hash functions

Recap: Types of Security Attacks on Information in Transit

- Integrity checks are particularly important for data in transit
- Here we need to consider the following **active** and **passive** attacks:
  - **Interception** - of info-traffic flow, attacks confidentiality
  - **Interruption** - of service, attacks availability
  - **Modification** - of info, attacks integrity
  - **Fabrication** - of info, attacks authentication
- In all these scenarios the attacker is a "Man-in-the-Middle" (MitM)



Recap: Passive Attacks

- Passive attacks are in the nature of eavesdropping or the monitoring of transmissions:
  - Release of plaintext message content
  - Traffic analysis of encrypted data communication
    - Allows to analyse patterns of message exchange (sender, receiver, timing) rather than content
- Tools like Wireshark allow for passive attacks

Recap: Active Attacks

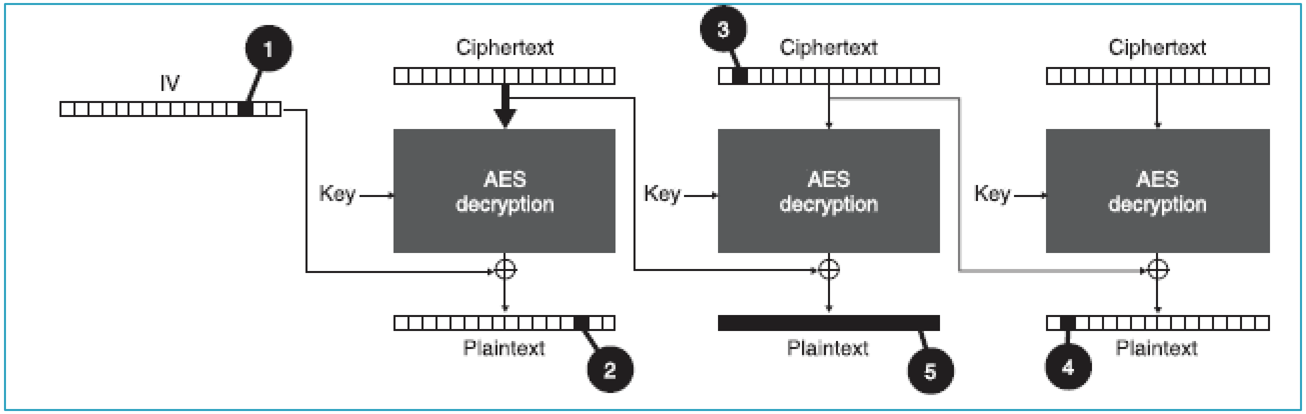
- Active attacks involve the modification or the creation of data in a stream:
  - **Masquerade**
    - **Attacker pretends to be a legitimate sender or receiver of data**
  - Replay
    - Attacker retransmits (encrypted) data which has been previously captured via eavesdropping
  - **Modification of message content**
    - **Attacker intercepts a message in transit, modifies it and forwards it to the receiver**
  - Denial of Service (DoS)
    - Attacker Inhibits the normal use of communication services

Attack Scenario

- Your company sends the software patch as email attachment to all the clients
- The patch is encrypted using a secret key, which is pairwise shared with your clients
- However, an attacker can
  - intercept these emails in transit, changes randomly a few bytes of the encrypted executable and forwards them to their destination, or
  - forge a similar looking email with some random file attached that claims to be a bug fix
- Your clients replace the executable on their local machines, which of course won't work and bring the entire factory floor to a halt
  - → financial losses for your clients, huge reputational loss for your company!
- **Therefore, your clients need some mechanism to validate the origin of the email, as well as the integrity of its content**

Case Study 2: Weakness of Mode Block Cipher Modes

- In CBC, the IV is tagged to an encrypted message as plaintext (thereby allowing the receiver to decrypt the message), a MitM attacker can do changes in transit. Here:
  - Flipping the i^{th} IV-bit (1) flips also the i^{th} plaintext bit (2)
  - Flipping a ciphertext bit (3) will change the entire plaintext block (5), and the corresponding bit of the next plaintext block (4)
- Other modes show similar weaknesses, i.e. changing one bit in a single block of an encrypted message (in transit) will corrupt the correct decoding of a following blocks
- **The receiver needs the ability to validate the integrity of the received message (blocks) !**

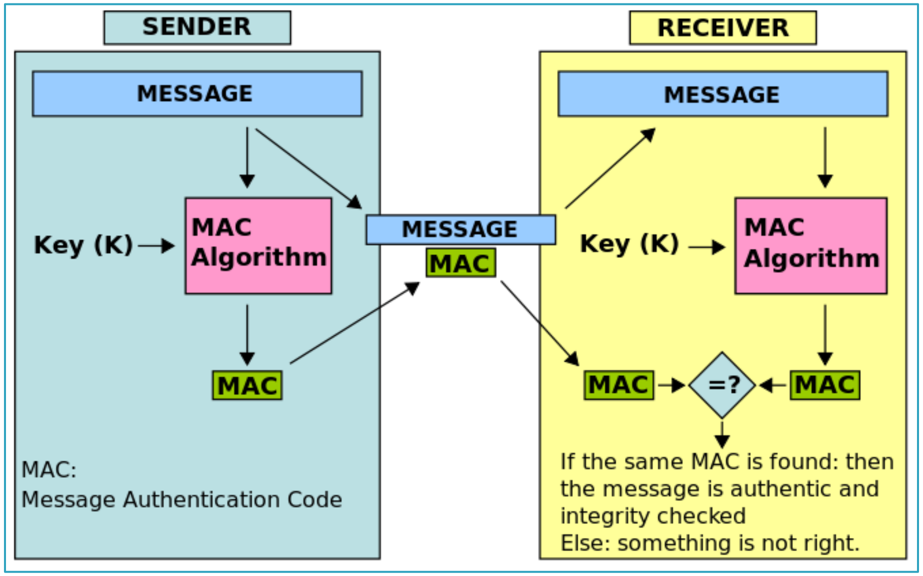


Message Authentication Code (MAC)

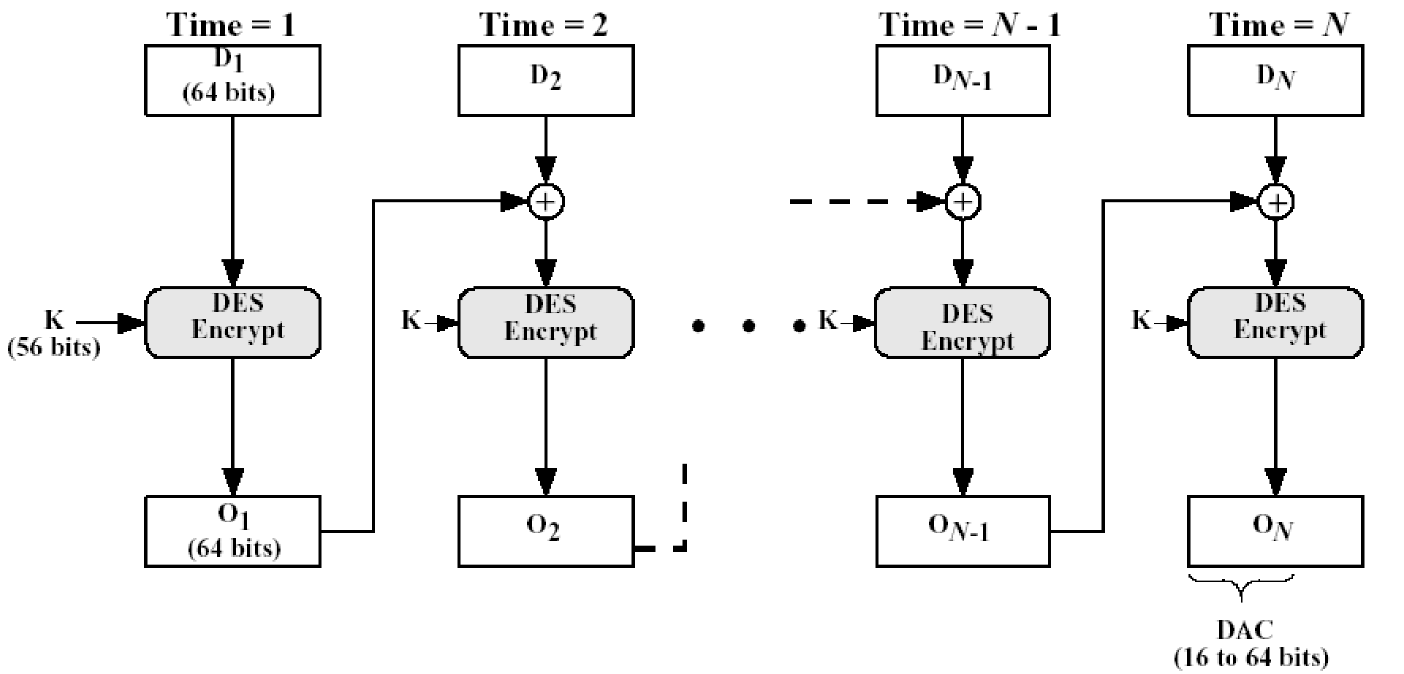
- Message authentication = message integrity [+ source authentication]
- A MAC (also called authentication tag, fingerprint, or cryptographic checksum), is a short piece of information used for authenticating and integrity-checking a message
- A MAC condenses a variable-length message M using a secret key K and some algorithm C to a fixed-sized authenticator: $MAC = C_K(M)$
- After its calculation, the MAC is appended to the message before it is sent
- Note that the message:
  - can have any length
  - is not automatically encrypted!

Typical Use of a MAC (Wikipedia)

- If both MACs are identical, the receiver knows, that
  - the message was not altered in transit,
  - the message was sent by the alleged sender, and
  - if the message includes a sequence number, that the sequence was not altered
- The term **CMAC** is used for a MAC that is calculated using a (block) cipher
- This contrasts to a **HMAC**, where a hash function (later) and a secret key is used



Typical CMAC Implementation



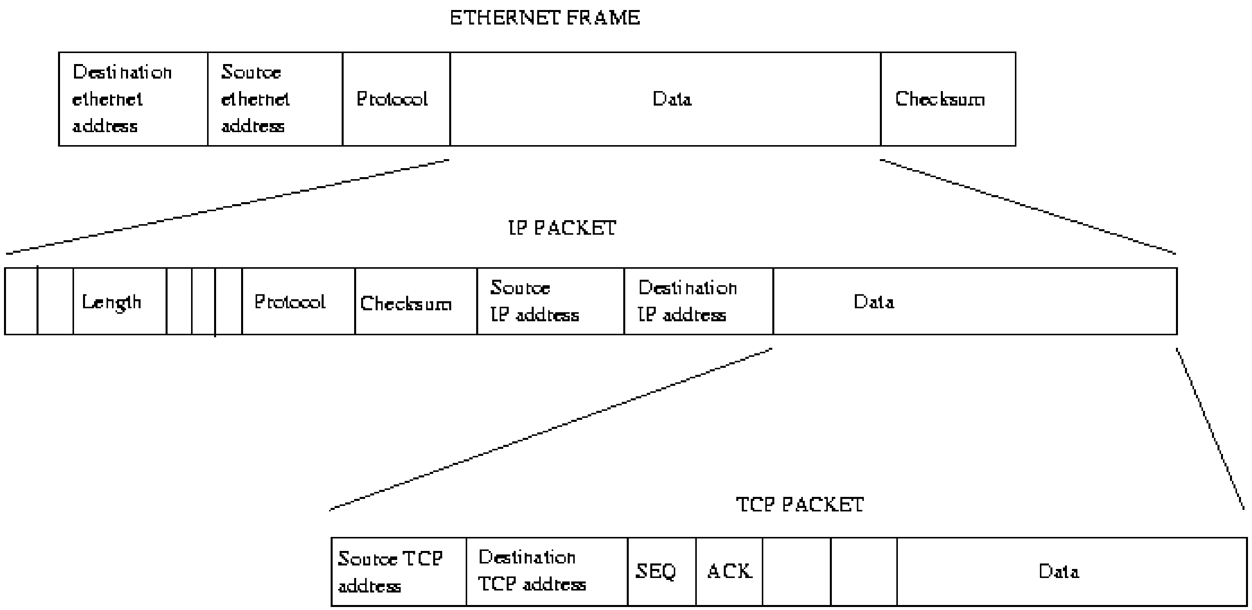
- Generally:
  - Any modern block cipher may be used (i.e., it's only DES in the example above)
  - Message padding shall apply as seen before
  - MAC = $C_K(M)$, where K is secret key and C is a symmetric block cipher (DES above)
  - MAC guarantees message integrity AND source authentication
  - This construction is also called **Encrypt-then-MAC**

Message Authentication Benefits

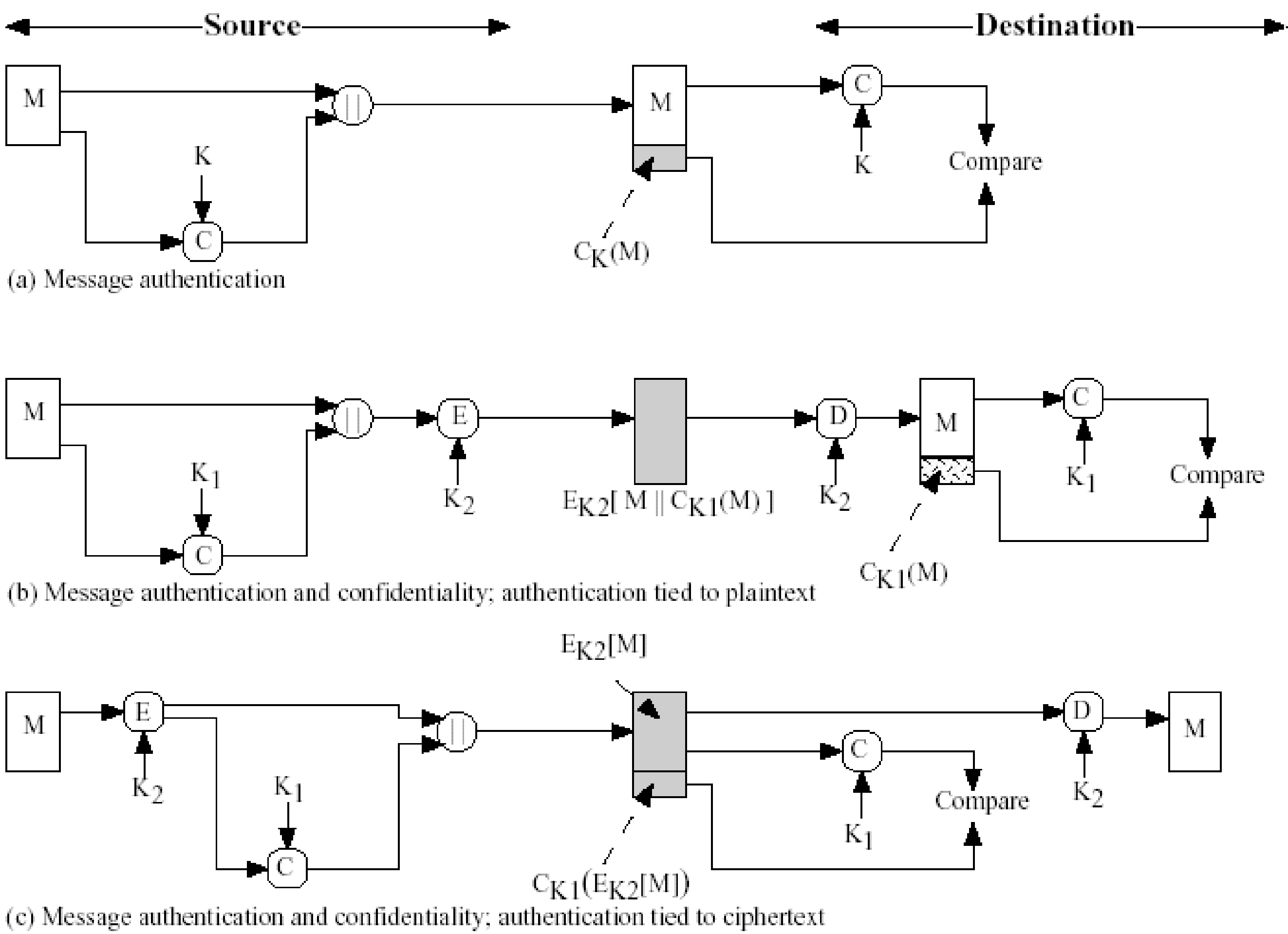
- In summary there are four types of attacks on data in transit, which are addressed by message authentication:
  - **Masquerade:** insertion of messages into the network from a fraudulent source
  - **Content modification**
  - **Sequence modification:** change the order of messages as they arrive
  - **Timing modification:** delete or repeat messages
- Note that the above may require a unique (i.e. incremented) sequence number in every message
- Therefore, message authentication is concerned with:
  - Protecting the integrity of a message
  - Validating identity of originator
  - Validating sequencing and timeliness
  - Non-repudiation of origin (dispute resolution)

Example: Authentication of TCP/IP Packets

- In TCP/IP data communication, a MAC cannot only cover the payload (i.e., the TCP Data field), but also the TCP header, as well as the non-modifiable fields of the IP header



Basic Use Cases of CMACs



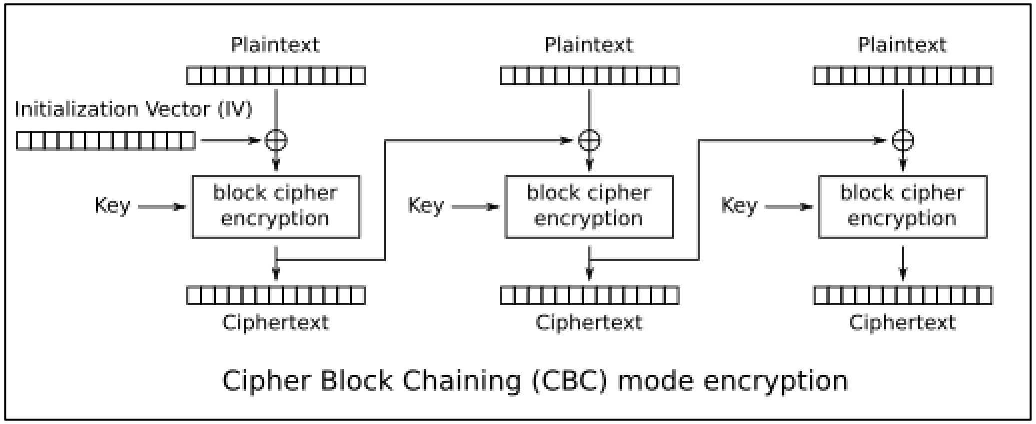
- M: Message
- K: Secret key
- C: Block cipher
- ¦¦:Concatenation operation

Case Study CMAC

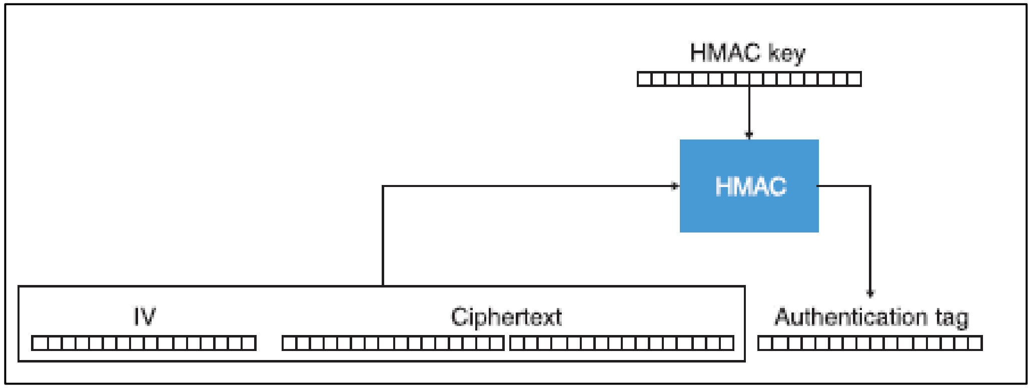
- Assume you operate a distributed weather station with battery-operated sensors located across Ireland
- You use "public" networks (i.e. Wi-Fi, Internet) to collect data and send it for processing to a central hub in Galway
- Which basic uses of a CMAC as shown in the previous slide would be most appropriate?
  - In your suggestion consider data privacy concerns and energy budget

The AES-CBC-HMAC Mode

- An example on how to combine authentication with a block cipher mode
- Based on CBC mode (top), but with additional authentication (bottom)
- Here the HMAC takes a single variable length input, i.e. the concatenation of IV + ciphertext + HMAC key, and creates a fix length authentication key
  - The diagram is misleading as it shows two separate inputs
- How many secret keys would this scheme require?

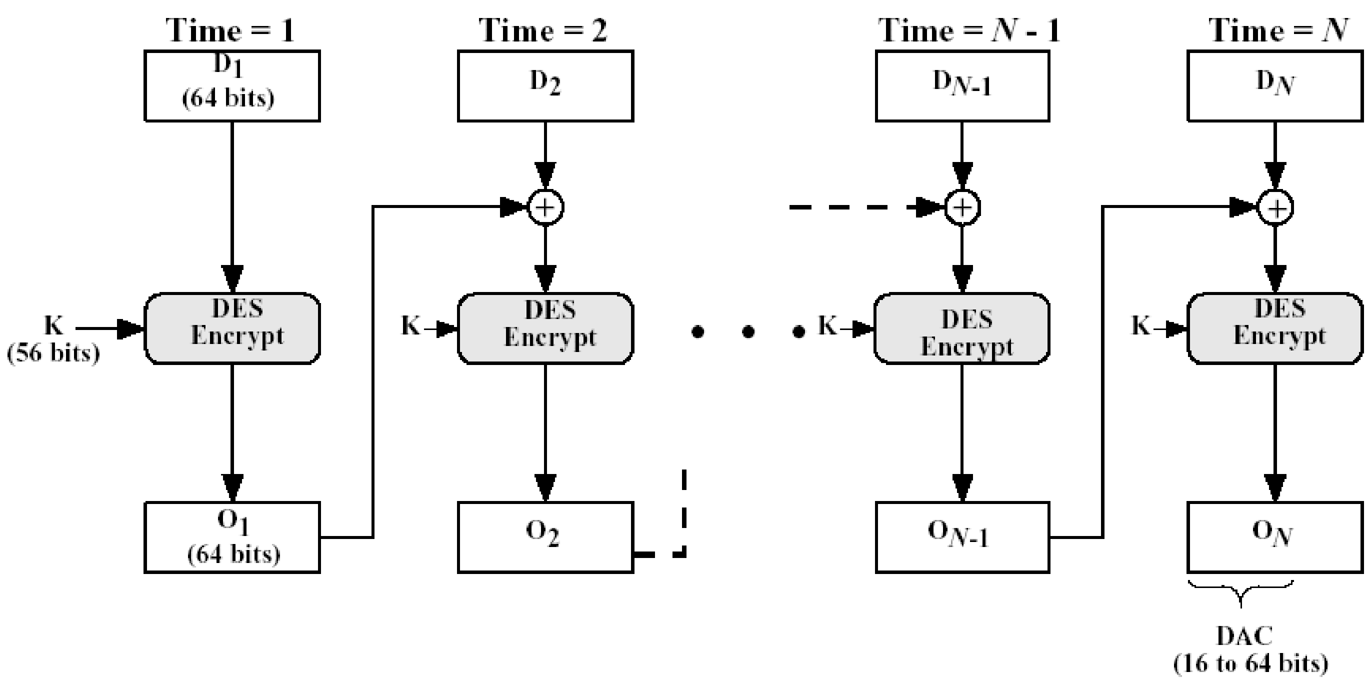


Cipher Block Chaining (CBC) mode encryption



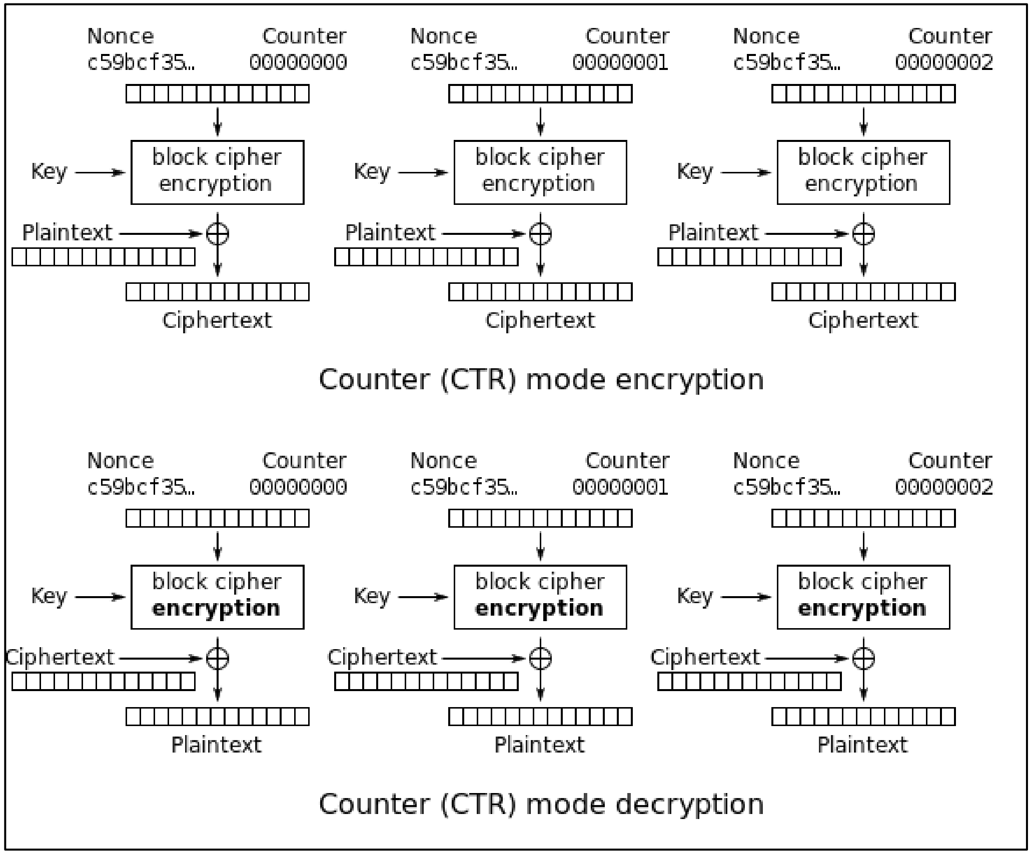
Block Cipher Mode of Operation: The Galois / Counter Mode

- What are weaknesses of the mode below and the AES-CBC-HMAC Mode (previous slide), i.e.
  - Can it be parallelised?
  - Is a 16- to 64-bit DAC sufficient?



Block Cipher Mode of Operation: The Galois / Counter Mode

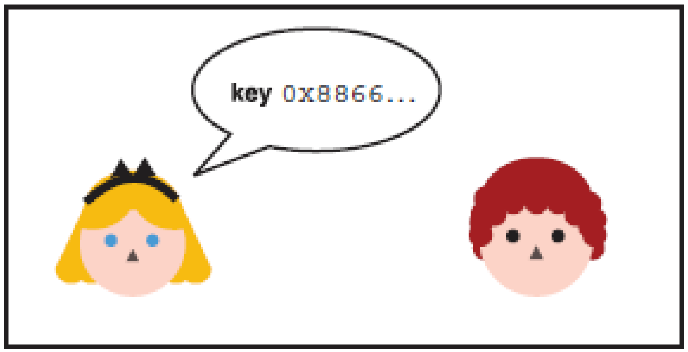
- Extension of counter mode
- Recall advantages of this mode?



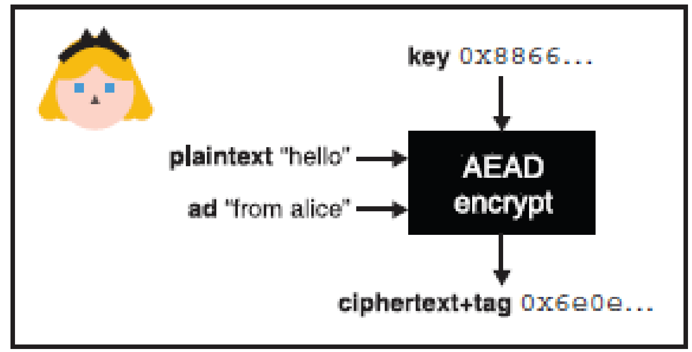
Block Cipher Mode of Operation: The Galois / Counter Mode

- GCM provides both data authenticity (integrity) and confidentiality
- It belongs to the class of **authenticated encryption with associated data (AEAD)** methods, i.e. it takes as an input
  - an initialisation vector IV
  - a **single** secret key K,
  - the plaintext P, and
  - some associated data AD
- It encrypts the plaintext (similar to counter mode) using the key to produce ciphertext C, and computes an authentication tag T from the ciphertext and the associated data (which remain unencrypted)
- A recipient with knowledge of K, upon reception of AD, C and T, can decrypt the ciphertext to recover the plaintext P and can check the tag T to ensure that neither ciphertext nor associated data were tampered with
- GCM uses a block cipher with block size 128 bits (i.e., AES-128), and uses arithmetic in the Galois field $GF(2^{128})$ to compute the authentication tag
  - That's modular arithmetic with a modulus of 2^{128}

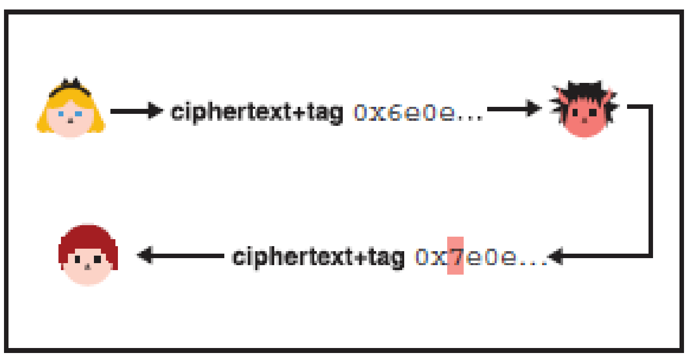
Features of AEAD



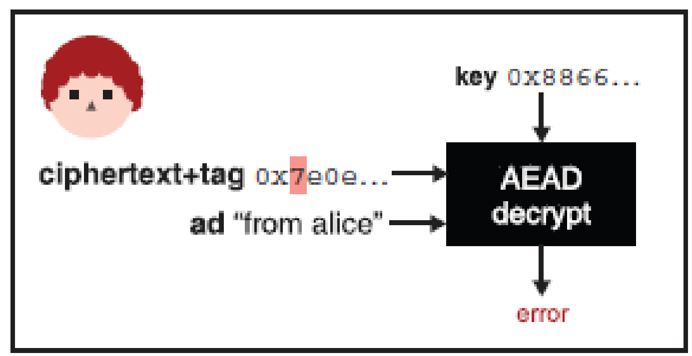
1. Alice and Bob meet in real life to **agree on a key**.



2. Alice can now use it to encrypt messages with an **AEAD algorithm** and the **symmetric key**. She can also add some optional associated data.



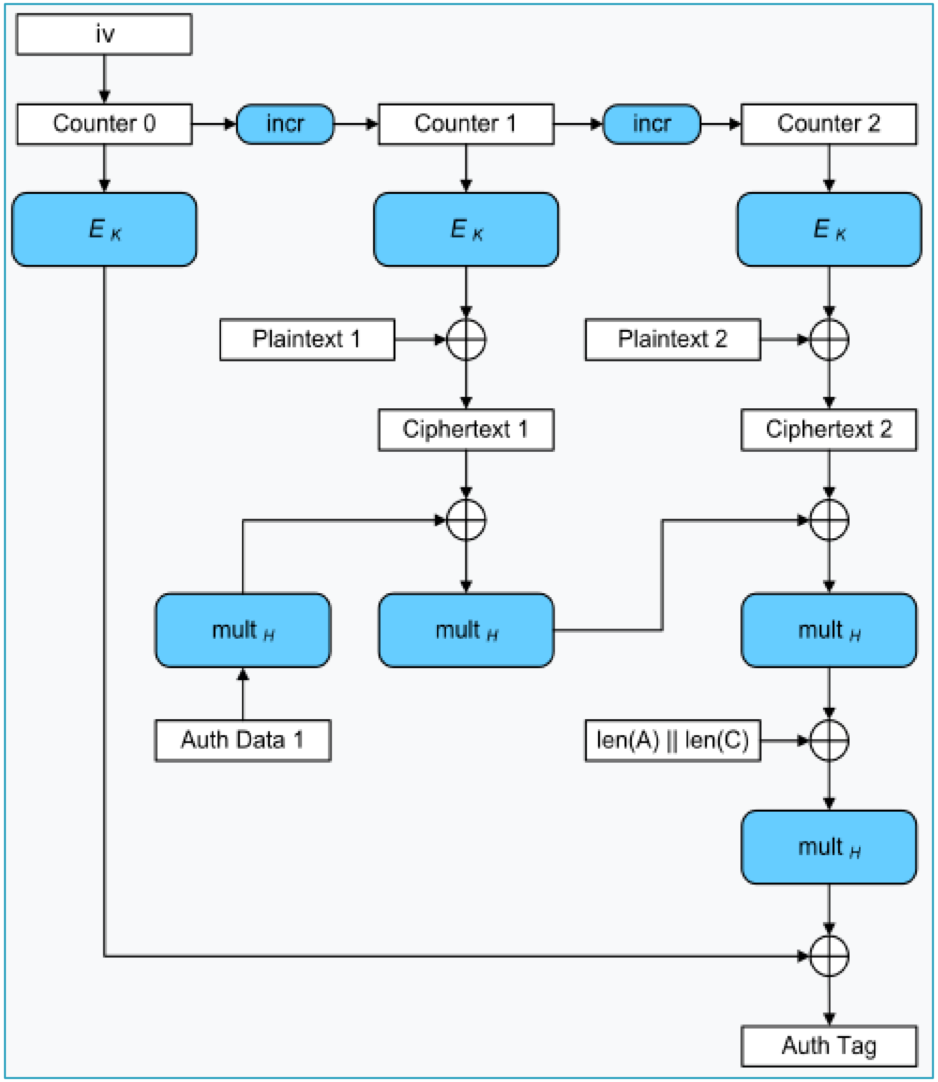
3. The ciphertext and tag are sent to Bob. An observer on the way **intercepts** them and **modifies** the ciphertext.



4. Bob uses the AEAD decryption algorithm on the **modified ciphertext** with the same key. **The decryption fails.**

Block Cipher Mode of Operation: The Galois / Counter Mode

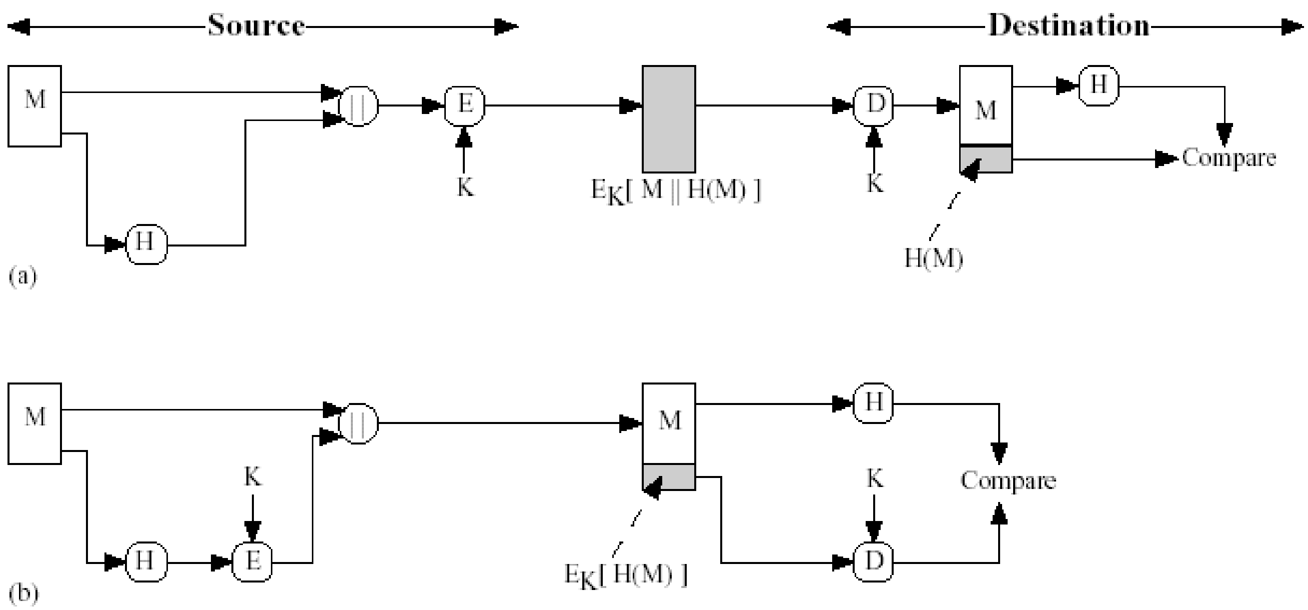
- A 96-bit IV is concatenated with a 32-bit counter (initialised with 0), i.e. (IV << 32) | | C
- E_K is AES with a 128 – 256 bit key (AES-128, AES-192 or AES-256)
- $mult_H$ is a hash-function (later) that produces a 128-bit (hash) output
- Auth_Data_1 has a variable length (but its hash is 128-bit wide)
- len(A) and and len(C) are 64-bit values that are the lengths (in bytes) of Auth_Data_1 and all ciphertext blocks respectively
- ⊕ is the bitwise XOR function



Hash Functions and HMAC

- A hash function produces a fixed size hash code (i.e. hash or fingerprint) based on a variable size input message
  - A hash function
    - does not need a key
    - guarantees the integrity of the message
- However, since a hash function is public and is not keyed, a hash value may have to be protected (i.e., encrypted)
  - A HMAC (hash-based message authentication code) is a specific type of MAC involving a cryptographic hash function and a secret cryptographic key
  - A HMAC verifies both message integrity and its authenticity
- Modern hash functions calculate 256 - 512-bit hashes

Basic Uses of HMACs



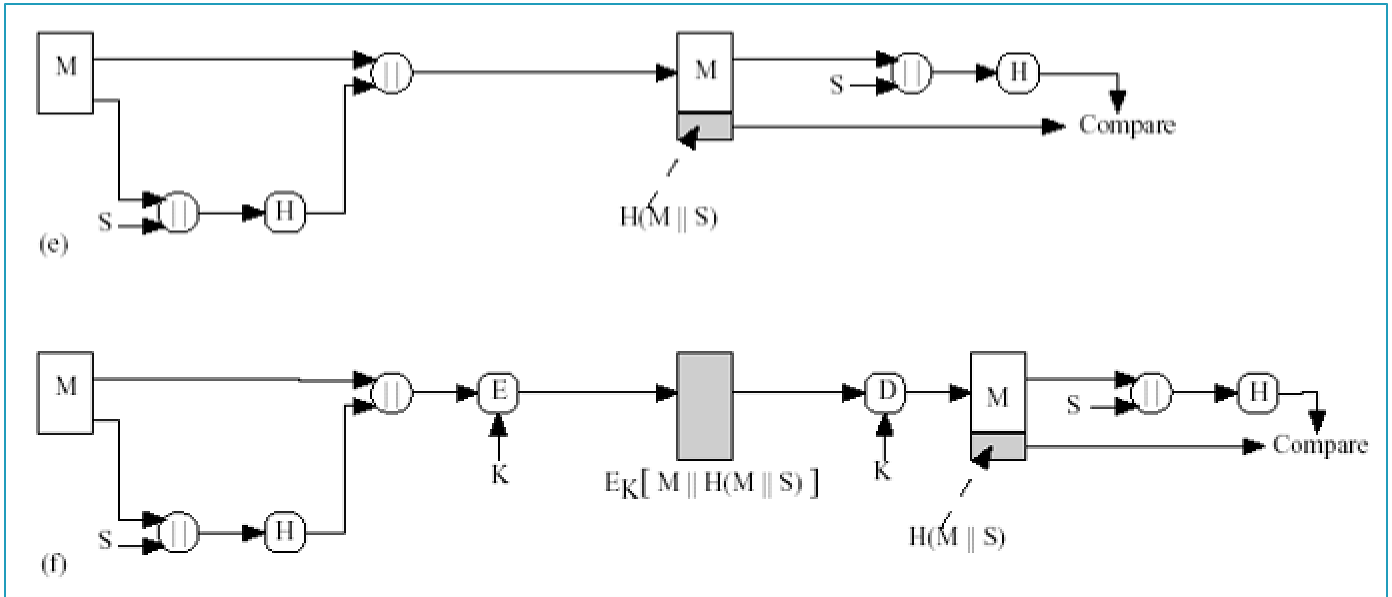
- M: Message
- H: Hash Function
- E: Block Cipher Encryption
- D: Block Cipher Decryption
- ¦¦: Concatenation operation

Note:

- Scenario (a) (and (f) provide confidentiality and message authentication
- Scenario (b) (and (c)) provide message authentication only

Basic Uses of HMACs

- In scenarios (e) and (f) a symmetric secret seed S is used, which is shared between sender and receiver
- S is used to authenticate all messages exchanged between both endpoints
- Scenario (f) also uses a symmetric key K for confidentiality, which is independent from S



Case Study HMAC

- Assume you operate a distributed weather station with battery-operated sensors located across Ireland
- You use "public" networks (i.e. Wi-Fi, Internet) to collect data and send it for processing to a central hub in Galway
- Which basic uses of a Hash function as shown in the previous slides would be most **appropriate and efficient**?

Requirements for a Hash Function H(x)

- **One-way property** (also called **pre-image resistance**): For a given hash function H and a hash value h it is infeasible to find x such that $H(x) = h$
  - I.e., it is virtually impossible to generate a message given a hash
  - Such a situation is also called a **hash collision**
- Why is the one-way property important?
  - See Figure (e): An opponent could intercept M || H(M, S), create inputs M || X (with some random value X), until a hash collision is found (i.e. S)

Requirements for a Hash Function H(x)

- **Weak collision resistance** (also called **second pre-image resistance)**:
  For a given hash function H and a known input x it is infeasible to find another input y with
  $y \ != x$ and $H(x) = H(y)$

- Why is the weak collision resistance important?
  - See Figure (b): An opponent could
    - calculate $h(M)$ (as both h and M are known)
    - find an alternate message with the same hash code (a hash collision), and
    - send it together with the encrypted (original) hash code to the receiver
  - The receiver would not be able to realise that the original message had been tampered with
    - Think of the previous software patch example

Requirements for a Hash Function H(x)

- **Strong collision resistance** (also called **collision resistance**): It is computational infeasible to find **any** pair of inputs (i.e., messages) (x, y) with $H(x) = H(y)$

- Why is the strong collision resistance important?
  - Again, see Figure (b), but this time the attack vector is different:
    - Rather than intercepting a hashed message in transit, the attacker presents the signing authority a crafted authentic message that has the same hash as a fraudulent message
    - Generating such a crafted message is accommodated by the **Birthday Paradox** discussed earlier

Birthday Paradox Attack

- Rather than thinking of birthdays, we consider messages and their hashes
- In the BPA the attacker does not intercept a hashed message in transit, but presents the signing authority a crafted authentic message that has the same hash as a fraudulent message (HMAC use case b)
- For a hash value that is m-bit long, the attacker creates a large number (i.e., in the order of $2^{0.5m}$) of variations of:
  - correct messages
  - fraudulent replacement messages
- The birthday paradox will make it more likely to find among both sets a correct message M_{nice} that has the same hash as a fraudulent message M_{nasty}
- M_{nice} is presented to the signing authority, who
  - hashes the message
  - encrypt the hash using the secret key (only known to the signing authority and the receiver)
  - concatenate message and hash
- Before the message is sent off, the attacker replaces M_{nice} with $_{Mnasty}$
- The receiver gets M_{nasty}, but will assume that it was signed (and send) by the signing authority

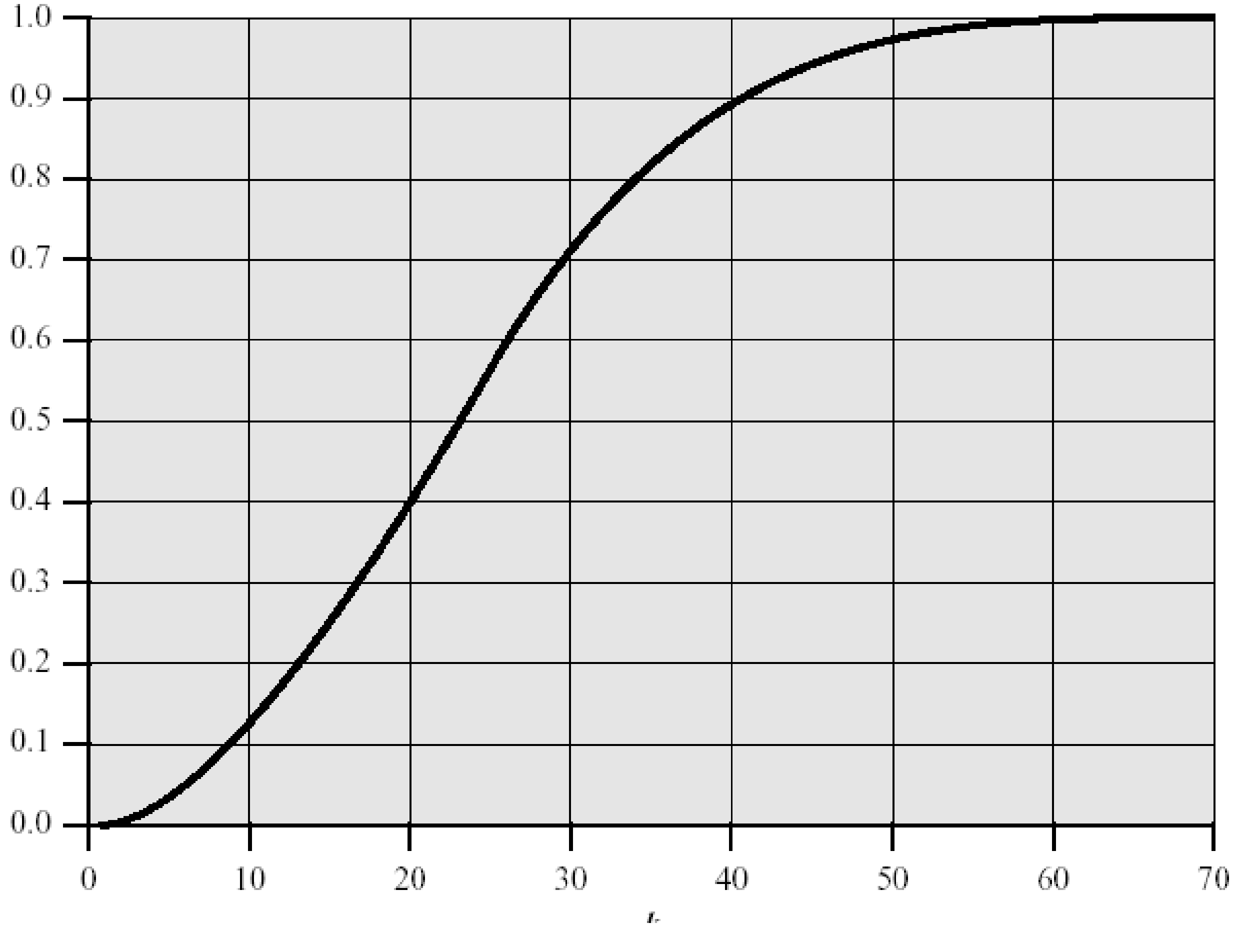
Birthday Paradox

- What is the minimum value k such that the probability is greater than 0.5 that at least 2 people in a group of k people have the same birthday, assuming that a year has 365 days?

- Intuitively someone would assume that $k = 365 / 2 = 183$

- **Probability theory shows, that $k = 23$ is sufficient!**

Birthday Paradox



BPA – How to create many Variations of a Message

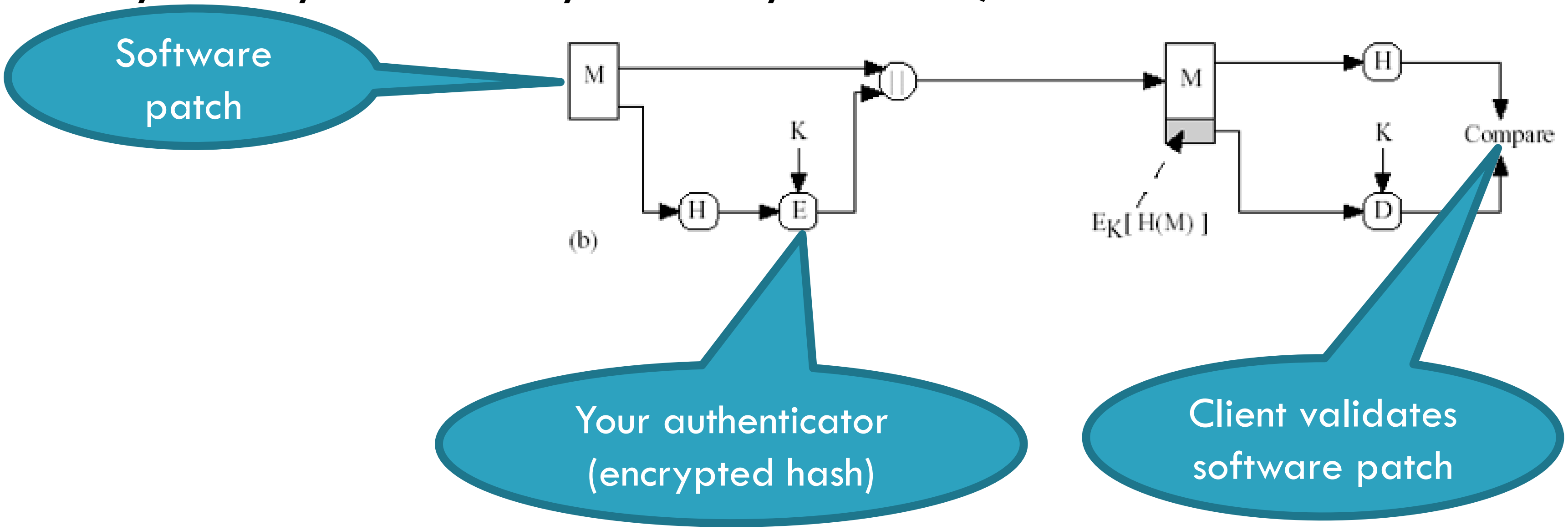
- The example gives a letter in 2^{37} variations

Dear Anthony,

{This letter is | I am writing} to introduce {you to | to you} {Mr. | --} Alfred {P. | --} Barton, the {new | newly appointed} {chief | senior} jewellery buyer for {our | the} Northern {European | Europe} {area | division}. He {will take | has taken} over {the | --} responsibility for {all | the whole of} our interests in {watches and jewellery | jewellery and watches} in the {area | region}. Please {afford | give} him {every | all the} help he {may need | needs} to {seek out | find} the most {modern | up to date} lines for the {top | high} end of the market. He is {empowered | authorized} to receive on our behalf {samples | specimens} of the {latest | newest} {watch and jewellery | jewellery and watch} products, {up | subject} to a {limit | maximum} of ten thousand dollars. He will {carry | hold} a signed copy of this {letter | document} as proof of identity. An order with his signature, which is {appended | attached} {authorizes | allows} you to charge the cost to this company at the {above | head office} address. We {fully | --} expect that our {level | volume} of orders will increase in the {following | next} year and {trust | hope} that the new appointment will {be | prove} {advantageous | an advantage} to both our companies.

Case Study: Circulating Software using the BPA

- This is a typical insider attack (here conducted by Grumpy George – GG – a disgruntled lead engineer in your team)
- Again, your team develops an urgent software patch, which is hashed
- The 32-bit hash value is encoded using a symmetric key K, which is shared with your client
- The key is only known to you and you client, but not to GG



Case Study: Circulating Software via a Birthday Paradox Attack

- GG as the lead engineer creates a large number of binary code versions for
  - software patches (to be presented to quality team)
  - malicious software patches (to be circulated)
- How can GG create > $2 * 2^{16}$ different source code variations?
  - GG introduces in both source code files a new constant variable (e.g. long int) that is not otherwise used, e.g.
    ...
    const unsigned long int var = 12; // possible values are 0 … 2^{64}-1
  - GG then creates different source codes by systematically incrementing var
    - GG is able to create 2^{64} different versions of both programs if needs to be
- GG compiles each of those software versions and calculates their hash
- GG looks for a hash collision, i.e. a software patch and a malicious patch that have the same hash code
- GG present this software patch to quality team, who sign it using key K
- GG replaces the software with the malicious patch before sending it to the client

Hash Function Execution (Example HAVAL)

- HAVAL creates a 256-bit fingerprint, for example:
  - "The quick brown fox jumps over the lazy **d**og“
    will be translated into the (256 bit) hash
    “b89c551cdfe2e06dbd4cea2be1bc7d557416c58ebb4d07cb
    c94e49f710c55be4”
  - “The quick brown fox jumps over the lazy **c**og”
    will be translated into the hash
    “60983bb8c8f49ad3bea29899b78cd741f4c96e911bbc272e
    5550a4f195a4077e”
- **I.e. very similar inputs result in totally different outputs, there is no correlation between a hash and its original input**

A naive Hash Function based on XOR

- Consider the XOR function ⊕:
- The input is broken into m blocks
- For the resulting hash value C, each bit C_i is calculated via

$$C_i = b_{i1} \oplus b_{i2} \oplus b_{i3} \oplus \ldots b_{im}$$

Where

- m = the number of n-bit blocks and
- b_{ij} is the i^{th} bit of the j^{th} block

EX-OR Gate Truth Table

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

A naive Hash Function based on XOR

	Bit 1	**Bit 2**	**…**	**Bit n**
Block 1	b_{11}	b_{21}		b_{n1}
Block 2	b_{12}	b_{22}		b_{n2}
…				
Block m	b_{1m}	b_{2m}		b_{nm}
Hash code	C_1	C_2		C_n

A naive Hash Function based on XOR

- Consider the ASCII-encoded input “ABC” and a hash function H that calculates an 8-bit hash h:
  - ASCII(A) = 65_{10} = 01000001_2
  - ASCII(B) = 66_{10} = 01000010_2
  - ASCII(C) = 67_{10} = 01000011_2
-

	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
A	0	1	0	0	0	0	0	1
B	0	1	0	0	0	0	1	0
C	0	1	0	0	0	0	1	1
h	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**

H(“ABC”) = h = 64_{10} = “@”

A naive Hash Function based on XOR

- Does this algorithm fulfil the requirements of a hash function:
  - One-way property?
  - Weak collision resistance?

	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
A	0	1	0	0	0	0	0	1
B	0	1	0	0	0	0	1	0
C	0	1	0	0	0	0	1	1
h	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**

H("ABC") = 64_{10} = "@"

Example: 8-bit Hash Function based on XOR

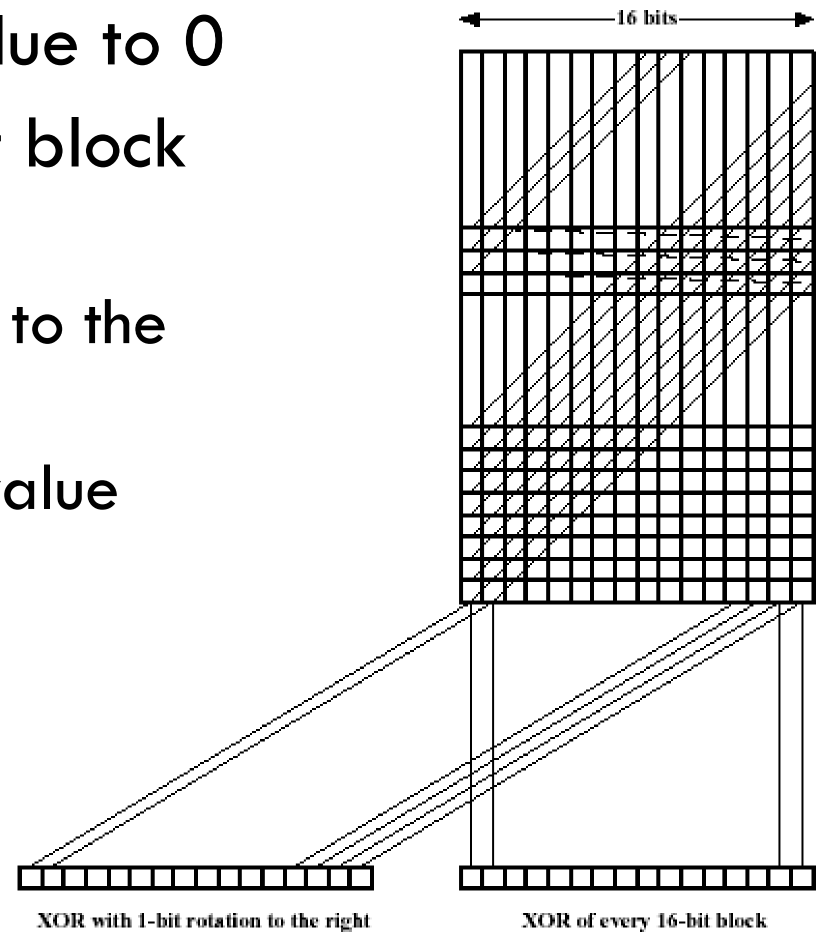
- Fulfils requirements of hash function?
  - One-way property? Certainly not!
  - Weak collision resistance? H(“ABC”) = H(“@@@”) = H(“@@@@@@“) = …

	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1
“@“	0	1	0	0	0	0	0	0
“@“	0	1	0	0	0	0	0	0
“@“	0	1	0	0	0	0	0	0
h	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**

H(“@@@”) = 64_{10} = “@”

A naive Hash Function based on rotating XOR

- Initially set the n-bit hash value to 0
- Process each successive n-bit block a follows:
  - Rotate the current hash value to the left by one bit
  - XOR the block into the hash value



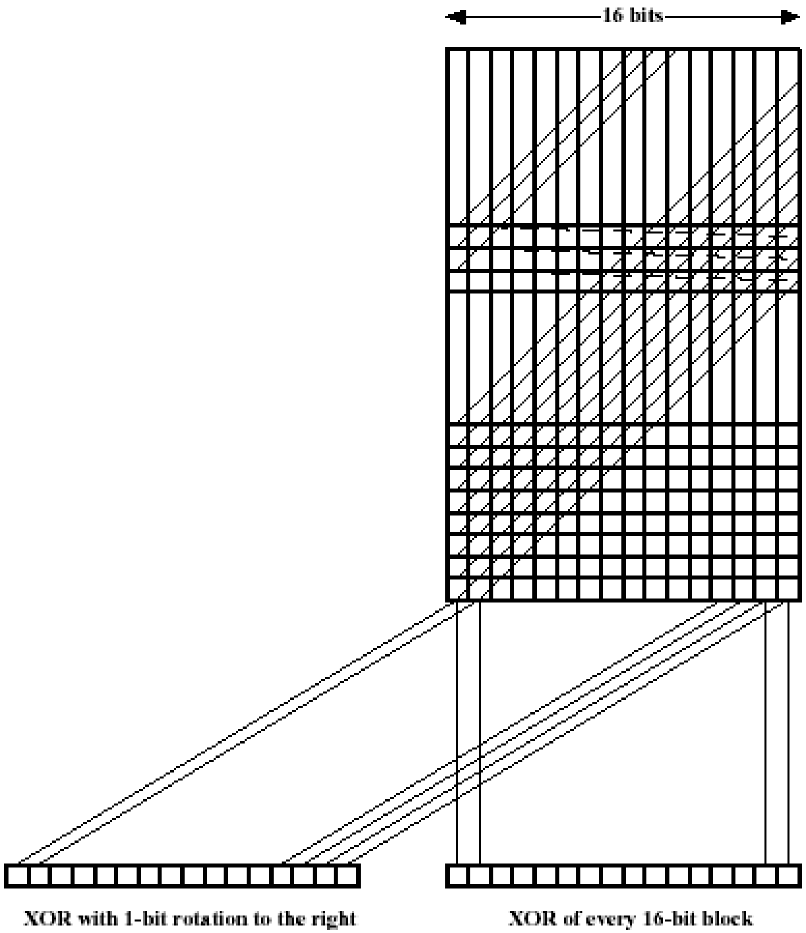
Example: Simple Hash Function based on Rotating XOR

- Consider "ABCD"
- "AB" = $01000010\ 01000011_2$
- "CD" = $01000100\ 01000101_2$
- "CD" left-rotated = $10001000\ 10001010_2$

	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
	0	1	0	0	0	0	1	1	0	1	0	0	0	0	1	1
	1	0	0	0	1	0	0	0	1	0	0	0	1	0	1	0
h	1	1	0	0	1	0	1	1	1	1	0	0	1	0	0	1

$h = CBC9_{16}$

Example: Simple Hash Function based on Rotating XOR

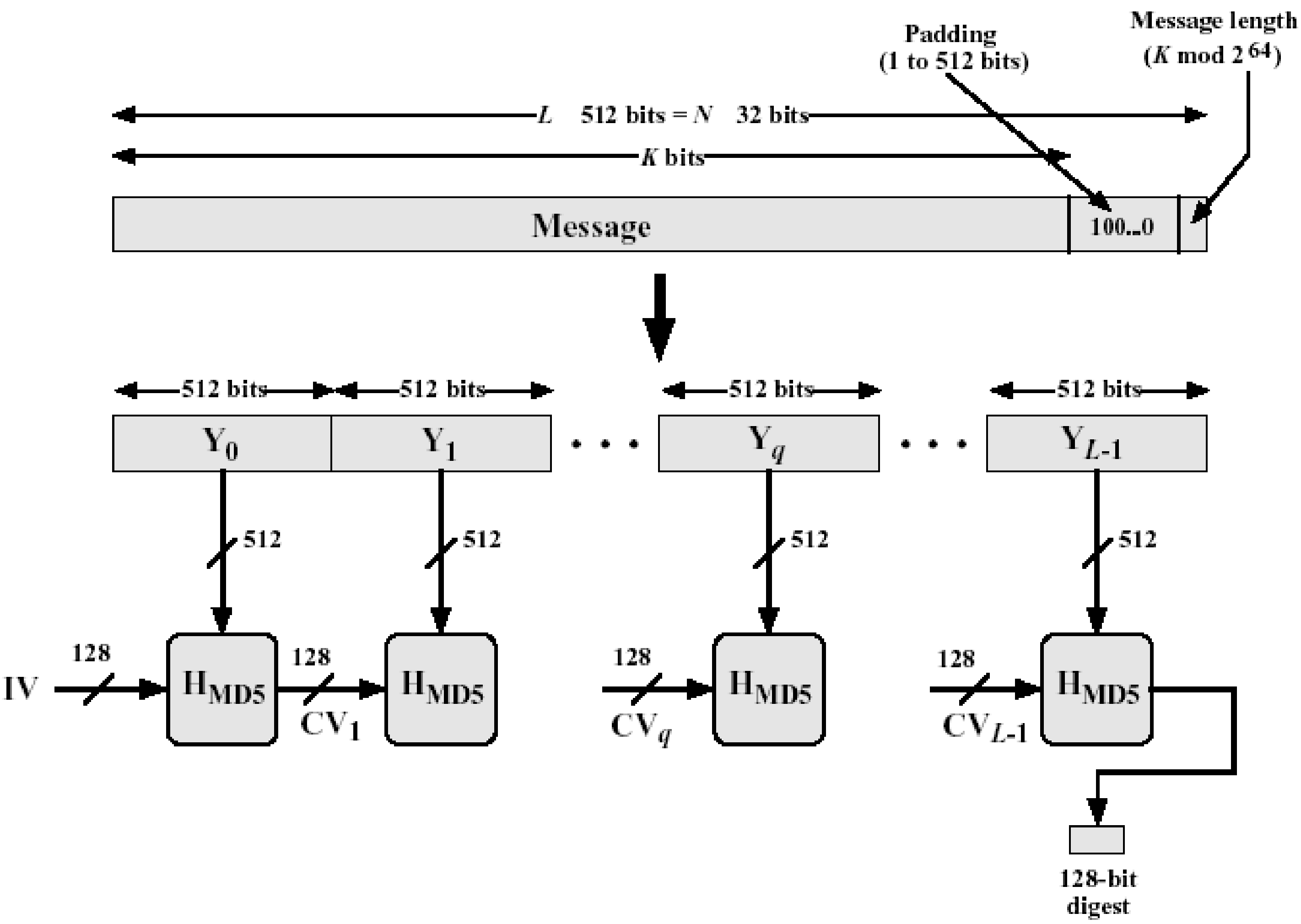


- Assume a password must be at least 2 ASCII-encoded characters long
- Fulfils requirements of hash Function?
  - One-way property?
  - Weak collision resistance?

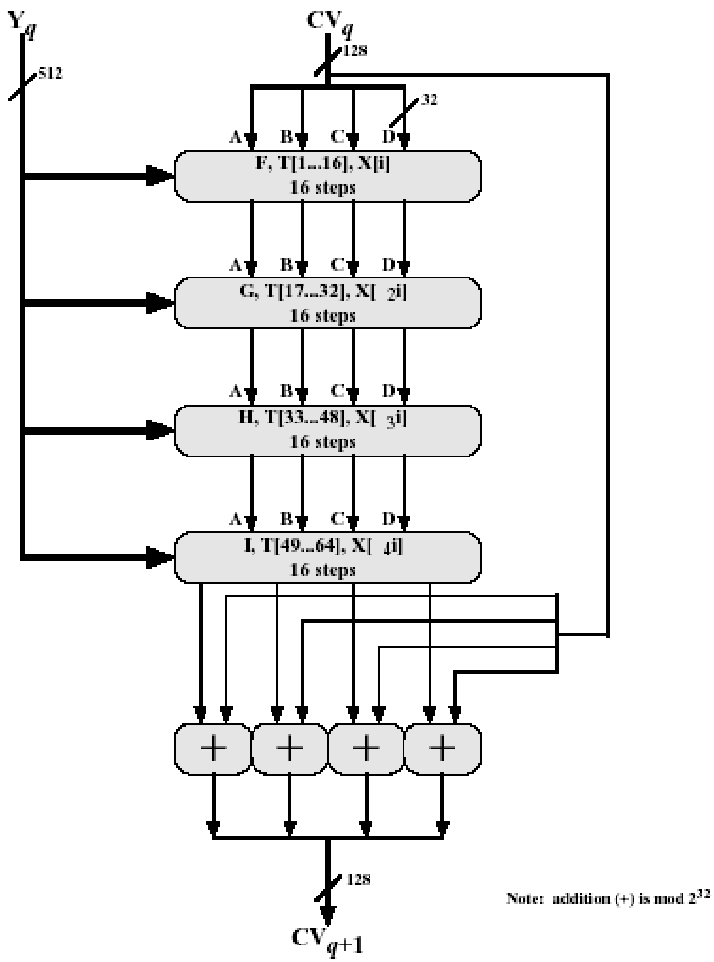
Examples for Hash Algorithms

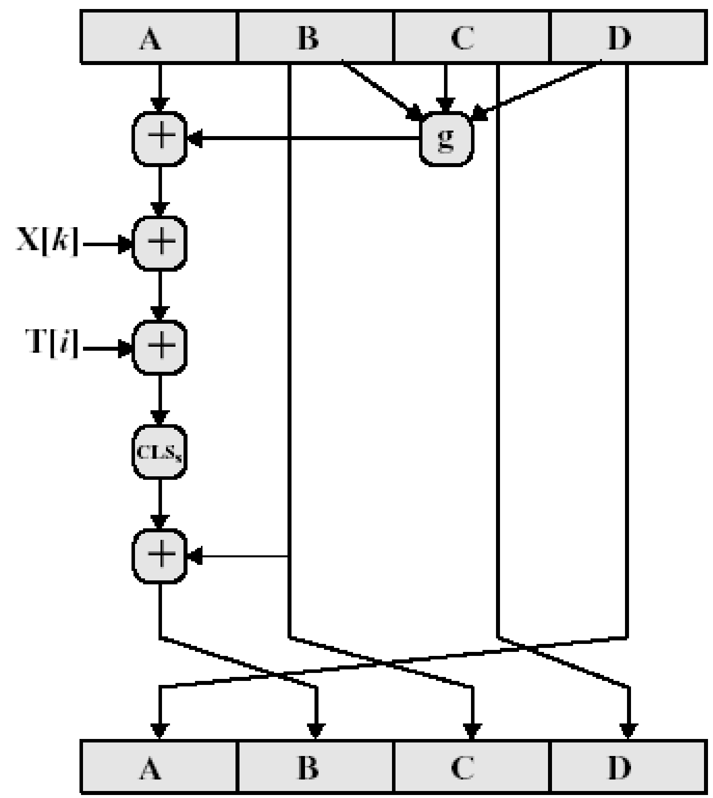
- In order to meet the aforementioned requirements, a hash algorithm must
  - be non-trivial
  - calculate long hash values
- Popular hash functions include:
  - MD5:
    - Produces a 128-bit hash value
    - Specified as Internet standards (RFC1321)
    - Still has some popularity, but unsafe for years (broken via collision attacks)
  - SHA (Secure Hash Algorithm) - X:
    - Family of hash functions, designed by NIST & NSA
    - SHA-3 (released 2015) produces 224-, 256-, 384- and 512-bits hash values
    - Internet standard
  - RIPEMD-160:
    - Creates a 160-bit hash value
    - Developed in Europe
- See https://defuse.ca/checksums.htm

FYI: MD5-An Overview



FYI: MD5-Processing of a Single 512 Bit Block (left) and Elementary MD5 Operation





FYI: MD5-Table T

T[1] = D76AA478	T[17] = F61E2562	T[33] = FFFA3942	T[49] = F4292244
T[2] = E8C7B756	T[18] = C040B340	T[34] = 8771F681	T[50] = 432AFF97
T[3] = 242070DB	T[19] = 265E5A51	T[35] = 699D6122	T[51] = AB9423A7
T[4] = C1BDCEEE	T[20] = E9B6C7AA	T[36] = FDE5380C	T[52] = FC93A039
T[5] = F57C0FAF	T[21] = D62F105D	T[37] = A4BEEA44	T[53] = 655B59C3
T[6] = 4787C62A	T[22] = 02441453	T[38] = 4BDECFA9	T[54] = 8F0CCC92
T[7] = A8304613	T[23] = D8A1E681	T[39] = F6BB4B60	T[55] = FFEFF47D
T[8] = FD469501	T[24] = E7D3FBC8	T[40] = BEBFBC70	T[56] = 85845DD1
T[9] = 698098D8	T[25] = 21E1CDE6	T[41] = 289B7EC6	T[57] = 6FA87E4F
T[10] = 8B44F7AF	T[26] = C33707D6	T[42] = EAA127FA	T[58] = FE2CE6E0
T[11] = FFFF5BB1	T[27] = F4D50D87	T[43] = D4EF3085	T[59] = A3014314
T[12] = 895CD7BE	T[28] = 455A14ED	T[44] = 04881D05	T[60] = 4E0811A1
T[13] = 6B901122	T[29] = A9E3E905	T[45] = D9D4D039	T[61] = F7537E82
T[14] = FD987193	T[30] = FCEFA3F8	T[46] = E6DB99E5	T[62] = BD3AF235
T[15] = A679438E	T[31] = 676F02D9	T[47] = 1FA27CF8	T[63] = 2AD7D2BB
T[16] = 49B40821	T[32] = 8D2A4C8A	T[48] = C4AC5665	T[64] = EB86D391

FYI: MD5-Primitive Functions and their Truth Tables

Round	Primitive function g	g(b, c, d)
1	F(b, c, d)	(b AND c) OR (NOT b AND d)
2	G(b, c, d)	(b AND d) OR (c AND NOT d)
3	H(b, c, d)	B EXOR c EXOR d
4	I(a, b, c)	C EXOR (b or NOT d)

b	c	d	F	G	H	I
0	0	0	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	0	1	1
1	0	1	0	1	0	1
1	1	0	1	1	0	0
1	1	1	1	1	1	0

Non-Cryptographic Hash Functions aka Checksums

- Checksums are designed to detect bit errors of files or data streams, e.g.
  - Hard disk storage errors
  - Data transmission errors
- CRC (Cyclic Redundancy Code) is a well know example
- Such checksums are too short and vulnerable to brute force attacks, and **are not suitable for cryptographic purposes**