Basics of Cryptology
MAC and hashes

Security system's requirements

- **Confidentiality.** Privacy or the ability to control or restrict access so that only authorized individuals can view sensitive information.
- **Integrity.** Information is accurate and reliable and has not been subtly changed or tampered with by an unauthorized party.
  - **Authenticity**: The ability to verify content has not changed in an unauthorized manner.
  - **Non-repudiation & Accountability**: The origin of any action on the system can be verified and associated with a user.
- **Availability:** Information and other critical assets are accessible to customers and the business when needed.

Integrity

- Avoid the (direct or indirect) modification of the information by user that are not allowed or due to unintentional threats.
- Authenticity
  - Allow to check whether data have been modified.
  - Any user should be allowed to verify the authenticity of the information (even not confidential).
- Non-repudiation & Accountability
  - Each document must be associated with the user that created/modified it.
    - Avoid that a user repudiates a document.
    - Avoid that a user pretend to be the “owner” of a document.

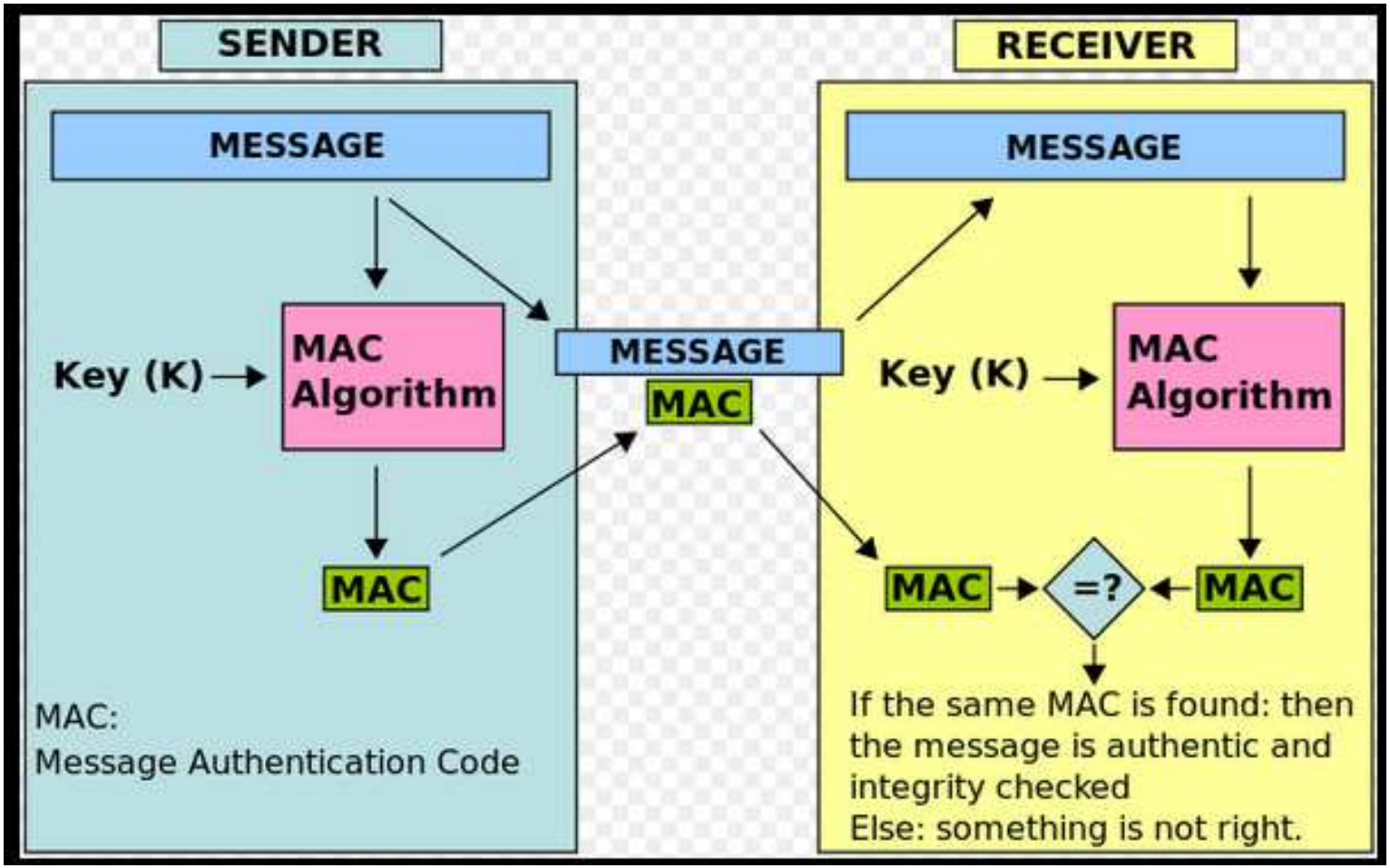
Message Authentication Code (MAC)

- A message authentication code (often MAC or MIC = Message Integrity Code) is a short piece of information attached to some message which is used to authenticate the message.
- A MAC algorithm takes as input a message and a secret key, and outputs a MAC (or a tag).
- MAC algorithms are usually ciphers.
- The MAC value ensures integrity and authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content.

General authentication algorithm

- Given a message m:
  - Compute $A_k(m)$
    - Where A is a cipher and k is a secret key
  - Send (m, $A_k(m)$)
- To authenticate:
  - Receive (m, $A_k(m)$)
  - Take m and compute $A_k(m)$
  - Compare the computed $A_k(m)$ with the received one
- Sender and receiver must share a key

General authentication algorithm



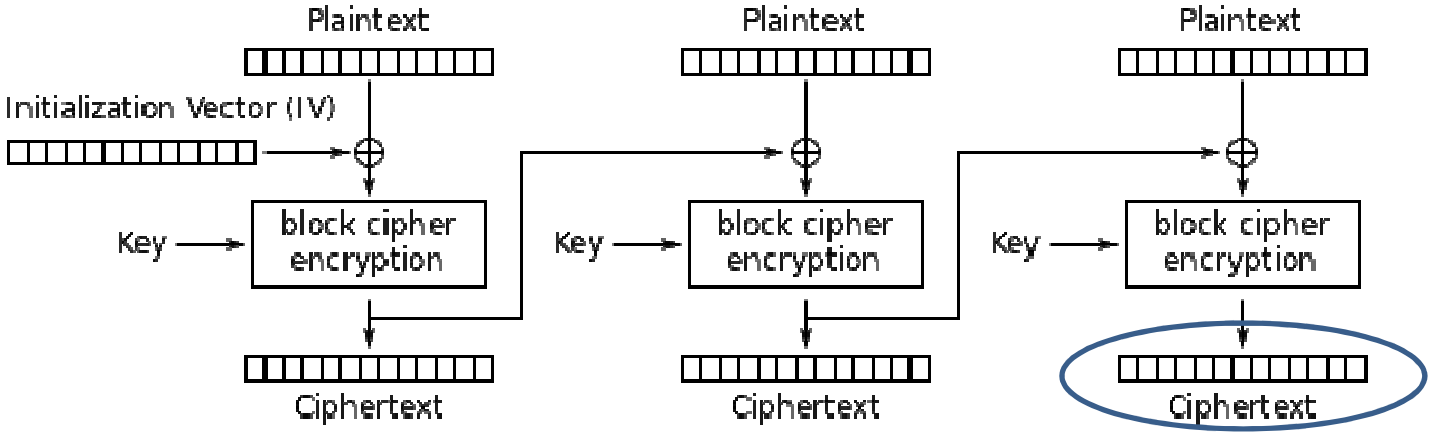
Requirements

- An attacker cannot compute a pair $(m, A_k(m))$ even if he sniffed $(m', A_k(m'))$ for many other messages m'.
- MACs must be short.
- MAC are not 1-to-1 functions.

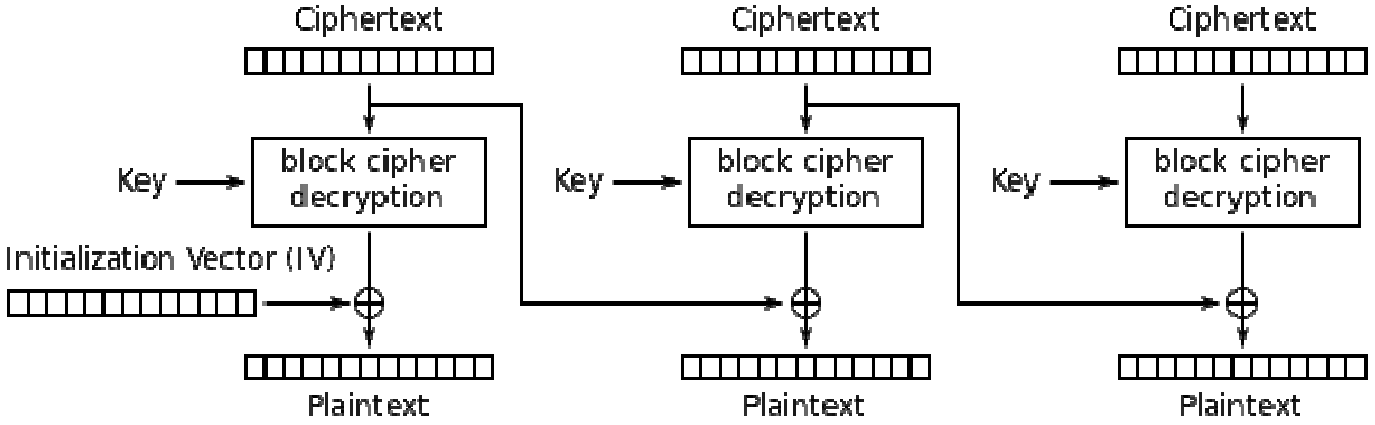
CBC (Cipher-block chaining) Residue

- A standard way for protecting against undetected modifications is to compute the CBC but send only the last block along with the plaintext message.

CBC (Cipher-block chaining) Residue



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

CBC (Cipher-block chaining) Residue

- The last block is called the CBC residue.
- In order to compute the CBC residue, the sender has to know the secret key.
- If an attacker modifies any portion of a message, the residue will no longer be the correct value.
- The attacker will not be able to compute the residue for the modified message without knowing the secret key.

Confidentiality + authentication

- CBC ensures confidentiality by ciphering the message.
- CBC ensures authentication by using the CBC residue as a MAC.

- For some applications we require only confidentiality or only authentication.
- What if we require both?

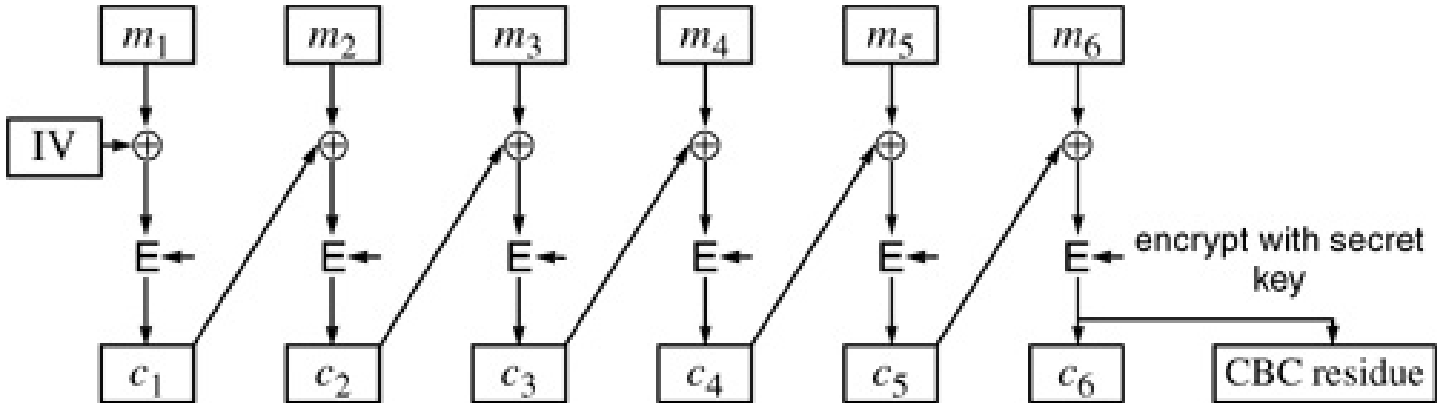
Confidentiality + authentication Solution 1

- Send the entire CBC sequence.

- Drawback:
  - We would like the receiving computer to automatically be able to tell if the message has been tampered with.
  - With CBC alone there is no way to detect tampering automatically, since CBC is merely an encryption technique. Any random string of bits will decrypt into something, and the final block bits of that random string will be a "correct" CBC residue.
  - So it is easy for anyone to modify the ciphertext, and a computer on the other side will decrypt the result, come up with garbage, but have no way of knowing that the result is garbage.
  - Notice that if the message was an English message, and a human was going to look at it, then modification of the ciphertext would probably get detected, but if it is a computer merely loading the bits onto disk, there is no way for the computer to know, with this scheme, that the message has been modified.

Confidentiality + authentication Solution 2

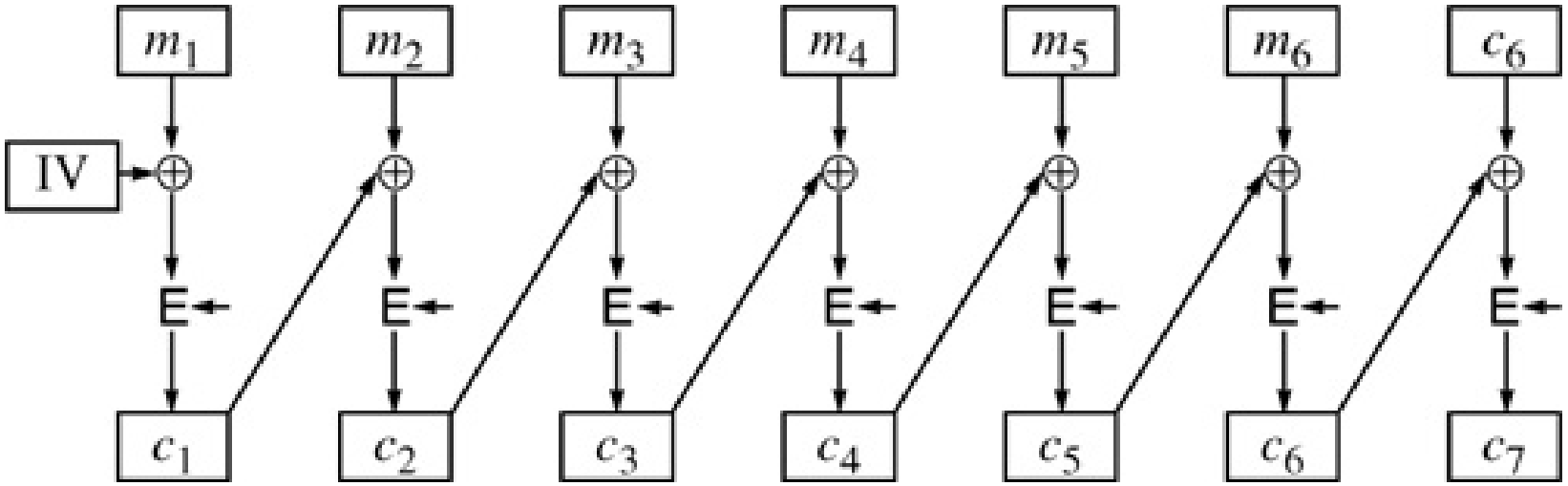
- Send the CBC encrypted message and just repeat the final block.



- Drawback:
  - An attacker could simply modify the message and send the last block twice.
  - Again, there is no way to know if the message is garbage.

Confidentiality + authentication Solution 3

- Compute the CBC residue, attach it to the plaintext, and then do a CBC encryption of the concatenated message.



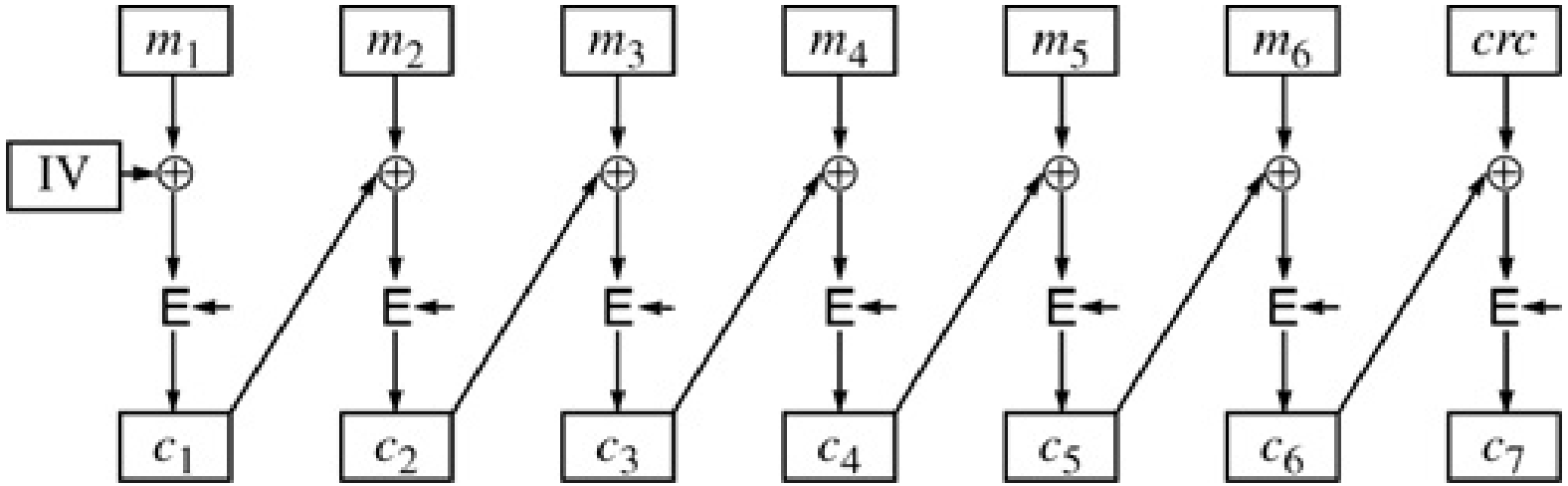
- Does it work?
- C6 is XORed with itself
- The output is always zero

Confidentiality + authentication Solution 4

- Compute a non-cryptographic checksum, attach it to the plaintext, and then do a CBC encryption of the concatenated message.
- Note: A non-cryptographic checksum is a value which is used to detect whether the message contains error.
  - It only provides error-detection.
  - It does not provide authentication because it is not based on a key and hence it is not possible to verify who is the sender.

Confidentiality + authentication Solution 4

- Compute a non-cryptographic checksum, attach it to the plaintext, and then do a CBC encryption of the concatenated message.



- It works if the checksum is long enough.
- Attacks are known for short checksums.

Confidentiality + authentication Solution 5

It is generally accepted as secure to compute a message integrity code by computing a CBC residue and then encrypting the message using an independently chosen key.

- We use two different keys:
  - One to obtain the residue;
  - One to encode.

- Drawback:
  - It requires two keys and twice the cryptographic power of encryption.
  - Various shortcuts have been proposed and even deployed, but generally they have subtle cryptographic flaws.

Confidentiality + authentication

- Another possible approach is
  - Compute a cryptographic hash of the plaintext, attach it to the plaintext, and then do a CBC encryption of the concatenated message.

- It still requires two cryptographic passes but it is more efficient than doing CBC twice if the hash function is faster than the encryption algorithm.

Hash functions

- A **hash** or a **message digest** is a one-way function.
  - Function: it takes an input message and produces an output.
  - One-way: it is not practical to figure out what input corresponds to a given output.
- For a hash function to be considered cryptographically secure,
  - It must be computationally infeasible to find a message that has a given hash.
  - It must be computationally infeasible to find two messages that have the same message digest.
    - It must be computationally infeasible to find a different message with the same message digest.
- We recall: Computationally infeasible = Impossible
- Informally, all of the digest/hash algorithms take an arbitrary-length message and wind up with a fixed-length quantity.

Hash functions

- Two main classes of algorithms
  - The NIST (National Institute of Standards and Technology) message digest function is called SHA-1, (Secure Hash Algorithm).
  - Rivest's hash functions: MD2, MD4, and MD5 (MD stands for Message Digest.)

Hash functions

- As in symmetric cryptography, there is intuitive notion of randomness
  - In symmetric cryptography, it is desirable for the mapping from input to output to appear randomly chosen.
  - In hash functions, the output should look random, that is:
    - It should not be possible to predict any portion of the output.
    - It should be true that, for any subset of bits in the message digest, the only way of obtaining two messages with the same value in those bits would be to try messages at random and compute the message digest of each until two happened to have the same value.
- A secure message digest function with n bits should be derivable from a message digest function with more than n bits merely by taking any particular subset of n bits from the larger message digest.

Hash functions

- Many messages yield the same message digest, because a message can be of arbitrary length and the message digest will be some fixed length,
  - 1000-bit messages
  - 128-bit message digest
  - 2^{1000} possible messages 2^{128} possible message digests
  - $2^{1000}/2^{128} = 2^{1000-128} = 2^{872}$ messages that map to any each message digest.
- By trying lots of messages, one would eventually find two that mapped to the same message digest.
- "lots" = so many that it is essentially impossible.

The birthday problem

- What is the probability that, in a set of n randomly chosen people, at least one pair of them will have the same birthday? (assume 365 days per year).
- If $P(A)$ is the probability of at least two people in the room having the same birthday, it may be simpler to calculate $P(A')$, the probability of there not being any two people having the same birthday. Then, because A and A' are the only two possibilities and are also mutually exclusive, $P(A) = 1 - P(A')$.

The birthday problem

- How to compute $P(A')$? (suppose n <= 365)

$P(A')=364/365 \times 363/365 \times 362/365 \times \ldots \times (365-n+1)/365=$

$= (1/365)^{n-1} \times 364 \times 363 \times 362 \times \ldots \times (365-n+1)$

n	*P(A)= 1 – P(A')*
1	0%
5	2.7%
10	11.7%
20	41.1%
23	50.7%
30	70.6%
40	89.1%
50	97.0%
60	99.4%
70	99.9%
100	99.99997%
200	99.99999999999999999999999998%
366	100%

The birthday problem

- If there are 23 or more people in a room, the probability that two of them have the same birthday is >=0.5.
- We will assume that a birthday is an unpredictable function taking a human and outputs one of 365 values.
- In general, we have n inputs (humans) and k possible outputs (birthdays), and an unpredictable mapping from input to output.
- With n inputs, there are $n(n-1)/2 = O(n^2)$ pairs of inputs.
- For each pair there's a probability of 1/k of both inputs producing the same output value.
- We need about k/2 pairs in order for the probability to be about 50% that you'll find a matching pair.
- It implies that, if you consider a group of humans that is greater than the square root of k, there's a good chance (about 50%) of finding a matching pair.

Hash functions

- Given a message digest with m bits:
  - Find a message that has a given message digest requires 2^m randomly chosen messages.
  - Find two messages with the same message digest requires about $2^{m/2}$ randomly chosen messages.
- 64-bit message digest function: feasible
- 256-bit message digest function: not feasible

Hash functions

Threats motivating “It must be computationally infeasible to find two messages that have the same message digest”:

- Suppose Alice wants to send a message to Bob.
- Alice asks her secretary to prepare the message.
- Alice computes the message digest, and encode it by using a private key to use it as a MAC.
- Trudy can prepare two messages m and m’ that have the same message digest h.
  - She gives m to Alice, obtain the MAC=E(h) from Alice
  - Then, she sends (m’,E(h)) to Bob.
- Bob receives (m’,E(h)), computes the hash of m’, encode it, and verifies that it is equal to E(h).

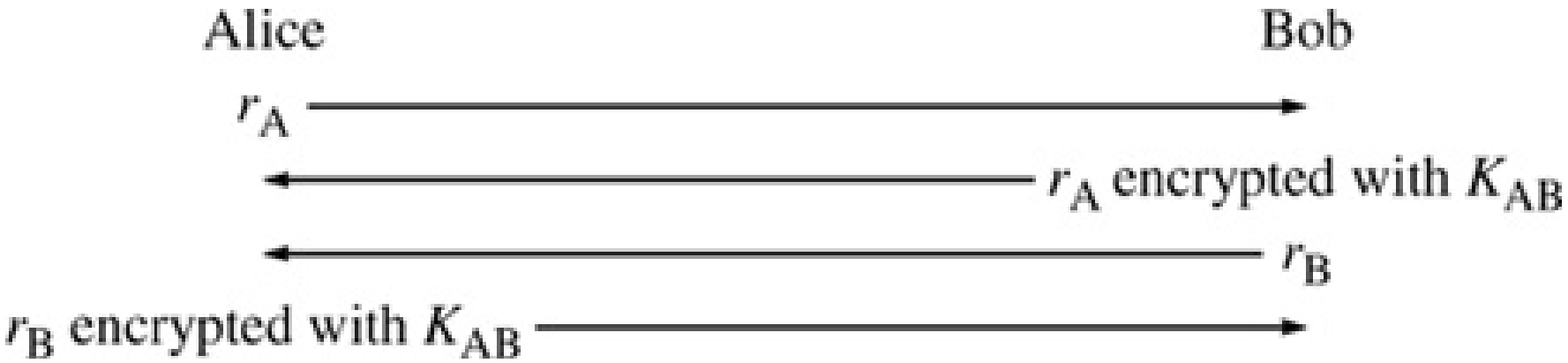
Applications of hash functions

- We have already seen applications of hash functions (Password Hashing, Message fingerprint).
- If there is a shared secret, message digest (or hash) algorithms can be used in all the ways that secret cryptography is used (notice we will use a shared secret and not secret cryptography!).
- The significant difference between a secret key algorithm and a message digest algorithm is that a secret key algorithm is designed to be reversible and a message digest algorithm is designed to be impossible to reverse.
- In the following we will use MD as a "generic" message digest (cryptographic hash) algorithm.

Applications of hash functions

Recall: Authentication of Sender and receiver.

- Alice and Bob share a key K_{AB} and they want to verify they are speaking to each other over a communication channel.
- Using cryptogaphy:
  - They each pick a random number, which is known as a **challenge**. Alice picks r_A. Bob picks r_B.
  - The value x encrypted with key K_{AB} is called the **response** to the challenge x.
  - Alice sends r_A and receives the response to r_A; Bob sends r_b and receives the response to r_b;



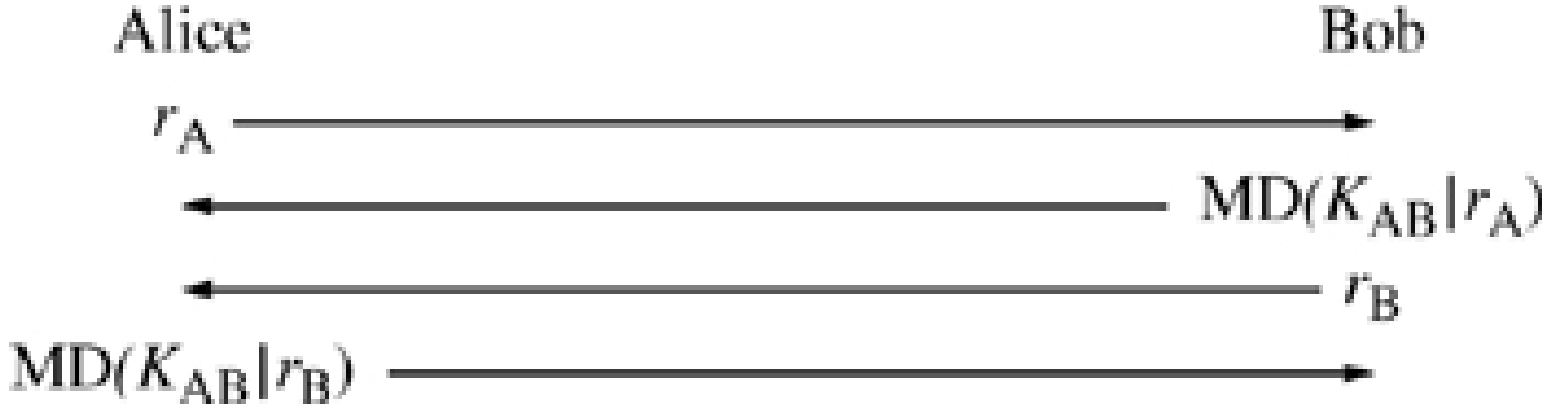
Applications of hash functions

- Imagine a world without secret key cryptography.
- Could we use a message digest function in some way to accomplish the same thing?
- Message digest algorithms aren't reversible, so it can't work quite the same way. In the above example, in which secret key cryptography is used for authentication, Bob encrypts something, and Alice decrypts it to make sure Bob encrypted the quantity properly.
- A hash function will do pretty much the same thing. Alice still sends a challenge. Bob then concatenates the secret he shares with Alice with the challenge, takes a message digest of that, and transmits that message digest. Alice can't "decrypt" the result. However, she can do the same computation, and check that the result matches what Bob sends.
- Notice that Bob and Alice will still need to share a secret.

Applications of hash functions

Application 1: Authentication of Sender and receiver.

- Alice and Bob share a key K_{AB} and they want to verify they are speaking to each other over a communication channel.
- Without cryptogaphy:
  - They each pick a random number, which is known as a **challenge**. Alice picks r_A. Bob picks r_B.
  - The **response** is computed as the message digest of the random number received concatenated with the key K_{AB}



- Note that the message digest is smaller than the ciphertext used before.

Applications of hash functions

Application 2: Computing a MAC with hash

- Note: the message digest of a message (i.e., MD(m)) is not a MAC because anyone can compute the message digest (it is not based on a secret key).
- We can use the same method used for the sender and receiver authentication:
  - We concatenate a shared secret K_{AB} to the message m, and use MD(m|K_{AB}) as the MAC.
  - Alice then sends the hash and the message (without the secret) to Bob. Bob concatenates the secret to the received message and computes the hash of the result. If that matches the received hash, Bob can have confidence the message was sent by someone knowing the secret.

Applications of hash functions

Application 3: Encryption with hash

- Similarly to Output feedback (OFB), an hash function can be used to compute a keystream
- It is similar to One-Time pad, but the keystream is generated starting from a shared key K which is shorter than the message
  - O_1 = MD(K) is the first block of the keystream
  - O_2 = MD(K|O_1)
  - …
  - O_i = MD(K|O_{i-1})

Applications of hash functions

Application 3: Encryption with hash

- Alice and Bob need a shared secret K.
- Alice wants to send Bob a message. She computes MD(K). That gives the first block of the bit stream, O_1. Then she computes MD($K_{AB}|O_1$) and uses that as O_2, and in general O_i is MD($K_{AB}|O_{i-1}$).
- Alice and Bob can do this in advance, before the message is known. Then when Alice wishes to send the message, she XORs it with as much of the generated bit stream as necessary. Similarly, Bob decrypts the ciphertext by XORing it with the bit stream he has calculated.

Hash functions

- Two main classes of algorithms
  - The NIST (National Institute of Standards and Technology) message digest function is called SHA-1, SHA-2, SHA-3, (Secure Hash Algorithm).
  - Rivest's hash functions: MD2, MD4, and MD5 (MD stands for Message Digest.)

MD5 - Overview

- The message is processed in 512-bit blocks (sixteen 32-bit words).
- The message digest is a 128-bit quantity (four 32-bit words).
- Each stage consists of computing a function based on the 512-bit message chunk and the message digest to produce a new intermediate value for the message digest.
- The value of the message digest is the result of the output of the final block of the message.

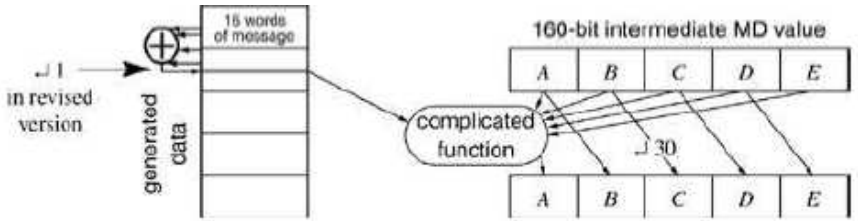
MD5 - Overview

- Each stage in MD5 takes four passes over the message block.
- Like any other message digest functions, the way how the passes are performed are like alchemy and not quite understandable (in the next slide we will see briefly what happens for SHA1).
- All the passes are different.

SHA1 - Overview

- Like MD5, SHA-1 operates in stages.
- The message is processed in 512-bit blocks (sixteen 32-bit words).
- The message digest consists of five 32-bit words (160 bits). Let's call them A, B, C, D, and E.
- After the last stage, the value of A|B|C|D|E is the message digest for the entire message.
- At the start of each stage, the 512-bit message block is used to create a 5x512-bit chunk.
- The first 512 bits of the chunk consist of the message block.
- The rest gets filled in, a 32-bit word at a time, according to the bizarre rule that the nth word (starting from word 16, since words 0 through 15 consist of the 512-bit message block) is the XOR of words n-3, n-8, n-14, and n-16. The XOR of words n-3, n-8, n-14, and n-16 is rotated left one bit before being stored as word n.
- Now we have a buffer of eighty 32-bit words (5x512 bits). Let's call the eighty 32-bit words W_0, W_1, ...W_{79}.
- Then, the digest continues by modifying A, B, C, D, and E with strange rules by using such words W_i.

SHA1 – Overview (not for the exam)



Now we have a buffer of eighty 32-bit words (5x512 bits). Let's call the eighty 32-bit words W_0, W_1, ...W_{79}. Now, a little program:

For t = 0 through 79, modify A, B, C, D, and E as follows:

B = old A C = old $B \lrcorner 30$ D = old C E = old D

"Hmm," you're thinking. "This isn't too hard to follow." Well, we haven't yet said what the new A is! Since the new A depends on the old A, B, C, D, and E, a programmer would first compute the new A into a temporary variable V, and then after computing the new values of B, C, D, and E, set the new A equal to V. For clarity we didn't bother. In the following computation, everything to the right of the equal refers to the old values of A, B, C, D, and E:

$$A = E + (A \lrcorner 5) + W_t + K_t + f(t,B,C,D)$$

Let's look at each of the terms. E and $A \lrcorner 5$ are easy. W_t is the t^{th} 32-bit word in the 80-word block. K_t is a constant, but it varies according to which word you're on (do you like the concept of a variable constant?):

$K_t = \lfloor 2^{30}\sqrt{2} \rfloor = 5a827999_{16}$ $(0 <= t <= 19)$
$K_t = \lfloor 2^{30}\sqrt{3} \rfloor = 6ed9eba1_{16}$ $(20 <= t <= 39)$
$K_t = \lfloor 2^{30}\sqrt{5} \rfloor = 8f1bbcdc_{16}$ $(40 <= t <= 59)$
$K_t = \lfloor 2^{30}\sqrt{10} \rfloor = ca62c1d6_{16}$ $(60 <= t <= 79)$

$f(t,B,C,D)$ is a function that varies according to which of the eighty words you're working on:

$f(t,B,C,D) = (B \wedge C) \vee (\sim B \wedge D)$ $(0 <= t <= 19)$
$f(t,B,C,D) = B \oplus C \oplus D$ $(20 <= t <= 39)$
$f(t,B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$ $(40 <= t <= 59)$
$f(t,B,C,D) = B \oplus C \oplus D$ $(60 <= t <= 79)$

Remarks on MD5

- The security of the MD5 has been severely compromised.
- In 2007 a collision attack (finding two inputs producing the same hash value) that can find collisions within seconds on a computer with a 2.6 GHz Pentium 4 processor, has been showed. Further, there is also a chosen-prefix collision attack that can produce a collision for two inputs with specified prefixes within hours.

- Still in 2015 MD5 was demonstrated to be still quite widely used.

Remarks on SHA1

- Cryptographic weaknesses were discovered in SHA-1 in 2010.

- The more secure SHA-3 was released on 2015.
- Unfortunately we do not have time to see it. However consider that you have enough knowledge to study it for yourself!