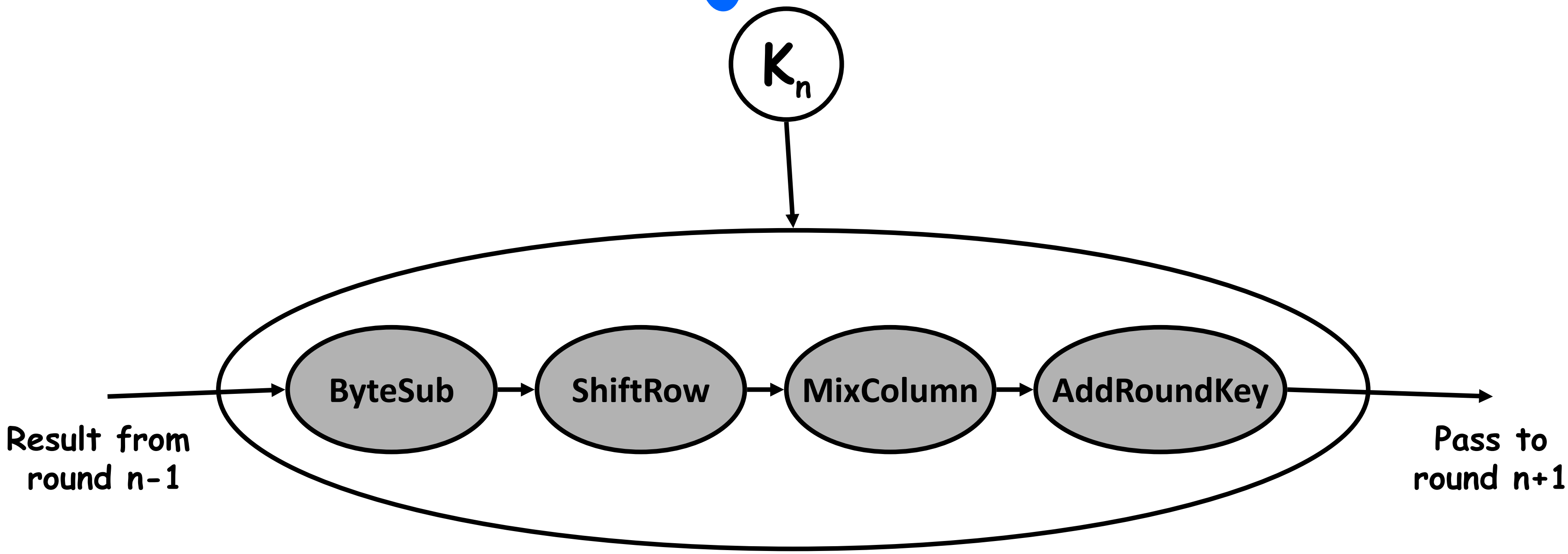
# Announcements

About Homework 1

- Available on the **course website**
    - If you cannot see it, it could be due to caching --- so try *refreshing the webpage*
- Due in ***two weeks***: 10/22/19 11:59pm
- Submit through **GradeScope**

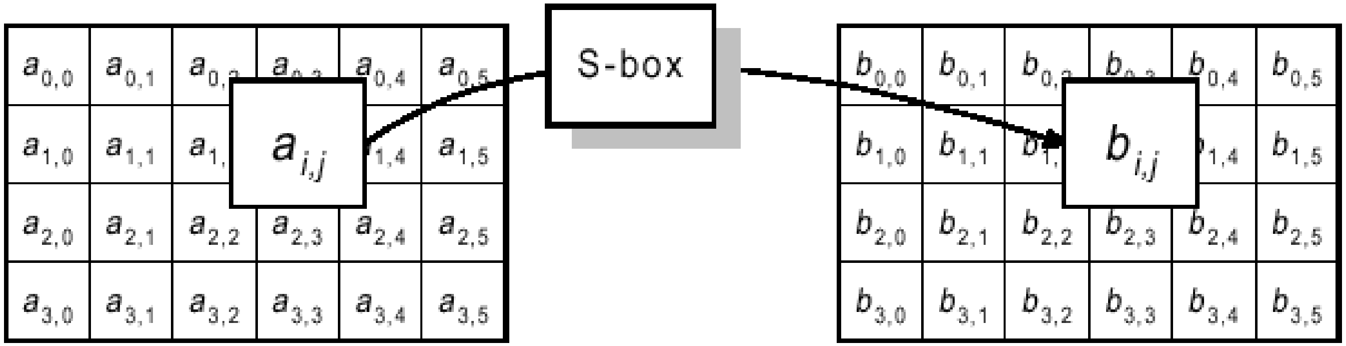
# Rijndael




**Detailed view of round n**

- **Each round performs the following operations:**
  - **Non-linear Layer:** No linear relationship between the input and output of a round
  - **Linear Mixing Layer:** Guarantees high diffusion over multiple rounds
    - Very small correlation between bytes of the round input and the bytes of the output
  - **Key Addition Layer:** Bytes of the input are simply XOR'ed with the expanded round key

# Rijndael: ByteSub




**Each byte at the input of a round undergoes a non-linear byte substitution according to the following transform:**

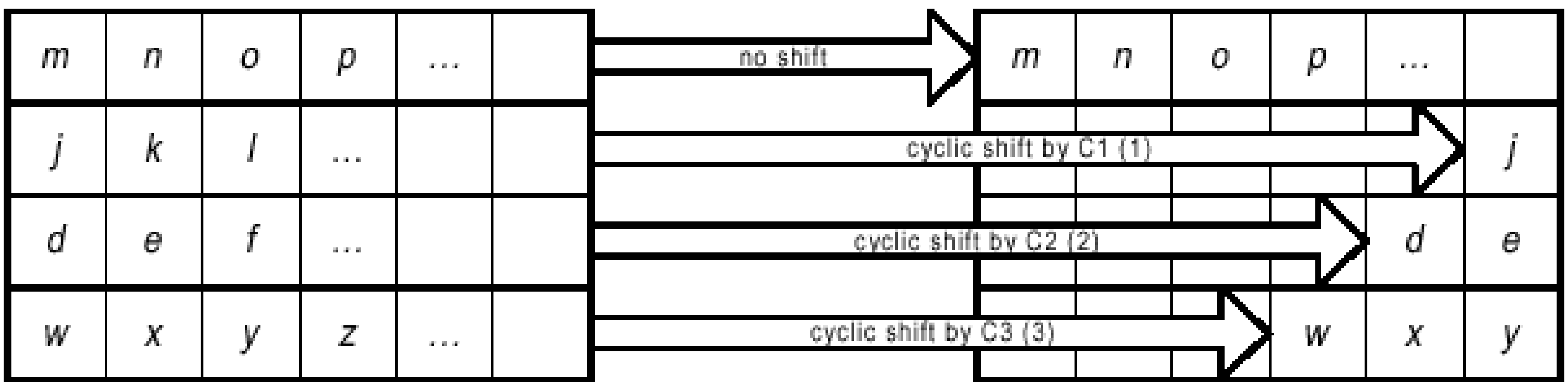
$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

**Substitution (“S”)-box**

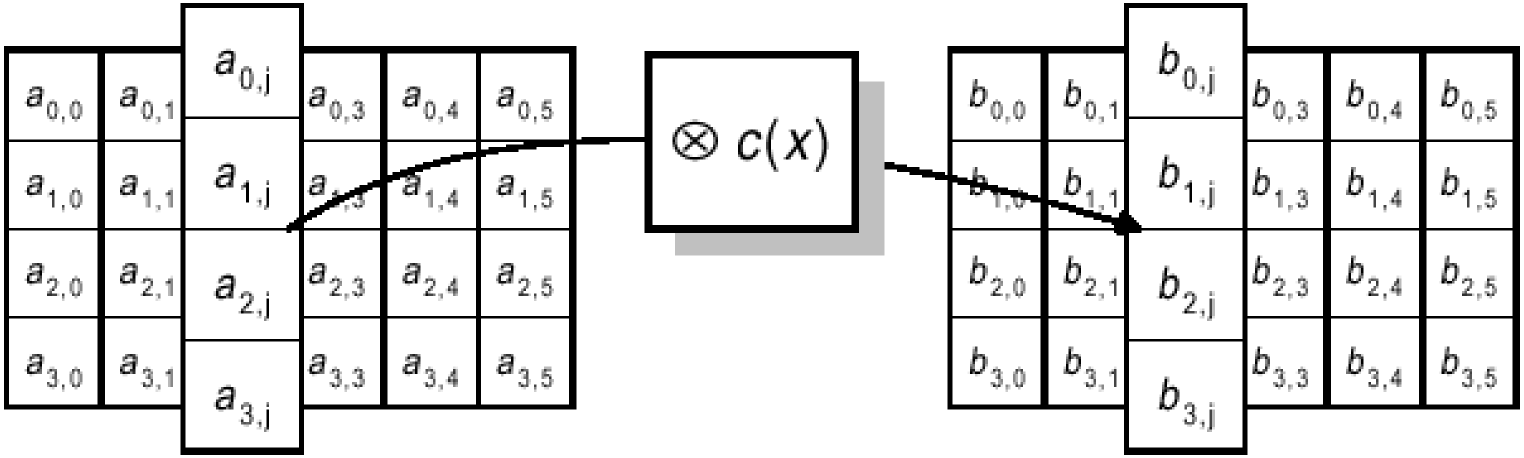
# Rijndael: ShiftRow

| Nb | C1 | C2 | C3 |
|---|---|---|---|
| 4 | 1 | 2 | 3 |
| 6 | 1 | 2 | 3 |
| 8 | 1 | 3 | 4 |

**Depending on the block length, each "row" of the block is cyclically shifted according to the above table**



# Rijndael: MixColumn




**Each column is multiplied by a fixed polynomial**
**$C(x) = '03'*X^3 + '01'*X^2 + '01'*X + '02'$**

**This corresponds to matrix multiplication $b(x) = c(x) \otimes a(x)$:**

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Not XOR

# Rijndael: Implementations

- Well-suited for software implementations on 8-bit processors (important for "Smart Cards")
  - Atomic operations focus on bytes and nibbles, not 32- or 64-bit integers
  - Layers such as ByteSub can be efficiently implemented using small tables in ROM (e.g., < 256 bytes).
  - No special instructions are required to speed up operation, e.g., barrel-shifting registers on some embedded device microprocessors

- For 32-bit implementations:
  - An entire round can be implemented via a fast table lookup routine on machines with 32-bit or higher word lengths
  - Considerable parallelism exists in the algorithm
    - Each layer of Rijndael operates in a parallel manner on the bytes of the round state, all four component transforms act on individual parts of the block
    - Although the Key expansion is complicated and cannot benefit much from parallelism, it only needs to be performed *once* when the two parties switch keys.

# Rijndael: Implementations

- Hardware Implementations
  - Rijndael performs very well in software, but there are cases when better performance is required (e.g., server and VPN applications).
  - Multiple S-Box engines, round-key XORs, and byte shifts can all be implemented efficiently in hardware when absolute speed is required
  - Small amount of hardware can vastly speed up 8-bit implementations

- Inverse Cipher
  - Except for the non-linear ByteSub step, each part of Rijndael has a straightforward inverse and the operations simply need to be undone in the reverse order.
  - However, Rijndael was specially written so that the same code that encrypts a block can also decrypt the same block simply by changing certain tables and polynomials for each layer. The rest of the operation remains identical.

# Conclusions and The Future

- Rijndael is an extremely fast, state-of-the-art, highly secure algorithm
- Amenable to efficient implementation in both hw and sw; requires no special instructions to obtain good performance on any computing platform
- Triple-DES: officially being retired by NIST.

# Lecture 5

## Cryptographic Hash Functions

**Read:** Chapter 5 in KPS

[lecture slides are adapted from previous slides by Prof. Gene Tsudik]

# Purpose

- CHF – one of the most important tools in modern cryptography and security

- CHF-s are used for many authentication, integrity, digital signatures and non-repudiation purposes

- Not the same as “hashing” used in DB or CRCs in communications

# Cryptographic HASH Functions

**Purpose:** produce a fixed-size "fingerprint" or digest of arbitrarily long input data

Why? To guarantee integrity of input

Properties of a "good" cryptographic HASH function H():

1. Takes on input of any size
2. Produces fixed-length output
3. Easy to compute (efficient)
4. Given any h, computationally infeasible to find any x such that H(x) = h
5. For a given x, computationally infeasible to find y: H(y) = H(x) and y≠x
6. Computationally infeasible to find any (x, y) such that H(x) = H(y) and x ≠ y

# Same Properties Re-stated:

- Cryptographic properties of a “good” HASH function:
  - One-Way-ness (#4)
  - Weak Collision-Resistance (#5)
  - Strong Collision-Resistance (#6)

- Non-cryptographic properties of a “good” HASH function
  - Efficiency (#3)
  - Fixed Output (#2)
  - Arbitrary-Length Input (#1)

# Simple Hash Functions

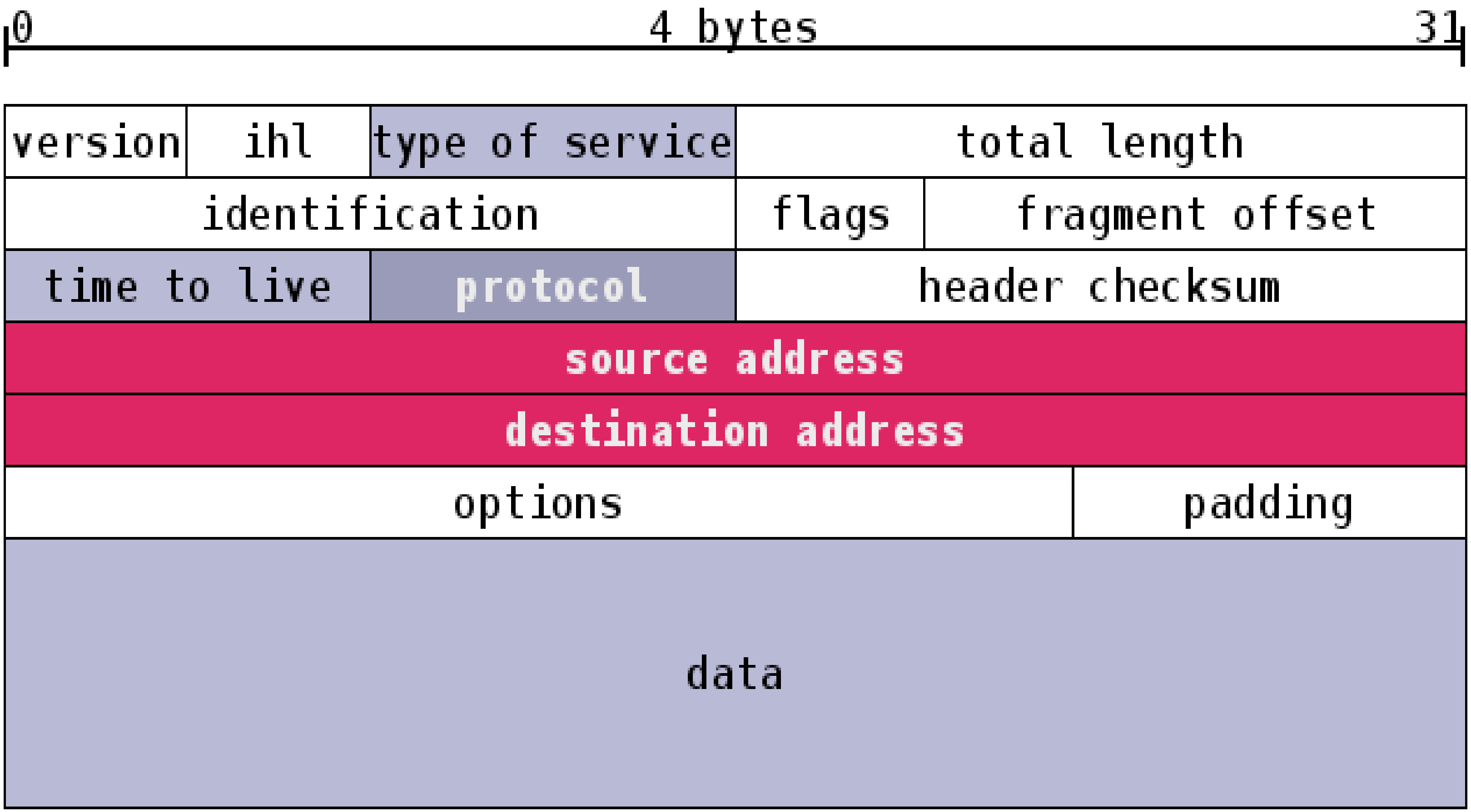
- Bitwise-XOR

| | 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** | $C_1$ | $C_2$ | | $C_n$ |

- Not secure, e.g., for English text (ASCII<128) the high-order bit is almost always zero
- Can be improved by rotating the hash code after each block is XOR-ed into it
- If message itself is not encrypted, it is easy to modify the message and append one block that would set the hash code as needed
- Another weak hash example: IP Header CRC

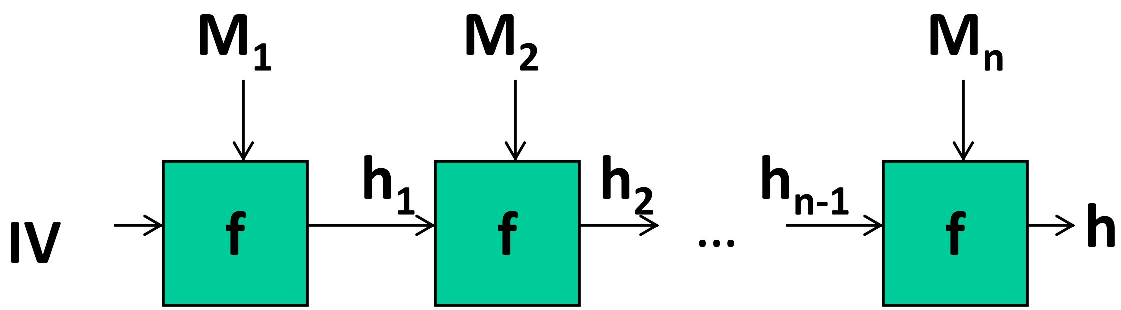
# Another Example

- IPv4 header checksum
- One’s complement of the one’s complement sum of the IP header's 16-bit words



# Construction

- A hash function is typically based on an internal compression function f() that works on fixed-size input blocks (Mi)
  - Merkle-Damgard construction:
  - A fixed-size "compression function".
  - Each iteration mixes an input block with the previous block's output




- Sort of like a **Chained Block Cipher**
  - Produces a hash value for each fixed-size block based on (1) its content and (2) hash value for the previous block
  - "Avalanche" effect: 1-bit change in input produces "catastrophic" and unpredictable changes in output


# The Birthday Paradox

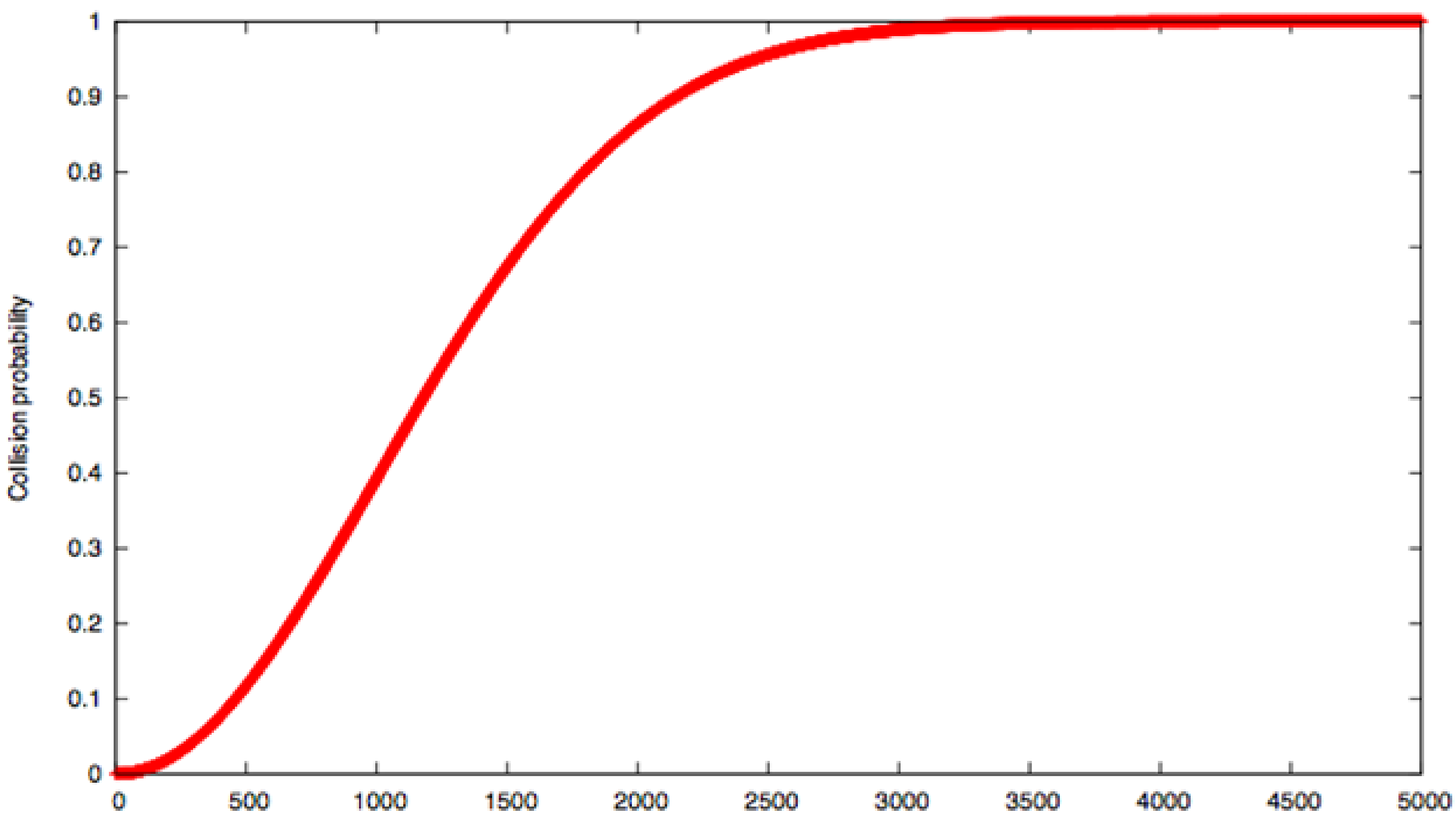

- Example hash function: **y=H(x)** where: x=person and H() is Bday()
- y ranges over set Y=[1…365], let n = size of Y, i.e., number of distinct values in the range of H()
- How many people do we need to 'hash' to have a collision?
- Or: what is the probability of selecting at random k DISTINCT numbers from Y?

- probability of no collisions:
  - $P0=1*(1-1/n)*(1-2/n)*\ldots*(1-(k-1)/n)) <= e^{(k(1-k)/2n)}$
    (use $1-x <= e^{-x}$)
- probability of at least one:
  - $P1=1-P0$
- Set P1 to be at least 0.5 and solve for k:
  - k == 1.17 * SQRT(n)
  - k = 22.3 for n=365

**Surprisingly small!**

# “Birthday Paradox”

Example: $N = 10^6$



# The Birthday Paradox

$m = \log(n) = \text{size of } H()$

$\sqrt{2^m} = 2^{m/2}$ trials must be computationally infeasible! Otherwise, finding collisions is easy.

# How Long Should a Hash be?

- Many input messages yield the same hash
  - e.g., 1024-bit message, 128-bit hash
  - On average, $2^{896}$ messages map into one hash
- With m-bit hash, it takes about $2^{m/2}$ trials to find a collision (with ≥ 0.5 probability)
- When m=64, it takes $2^{32}$ trials to find a collision (doable in very little time)
- Today, need **at least** m=160, requiring about $2^{80}$ trials (180 is better)

# CHF from a Block Cipher

One direct option:

- Split input into a sequence of *keys*: $M_1, \ldots M_p$
- Encrypt a constant plaintext (e.g., block of zeros) with this sequence of keys:

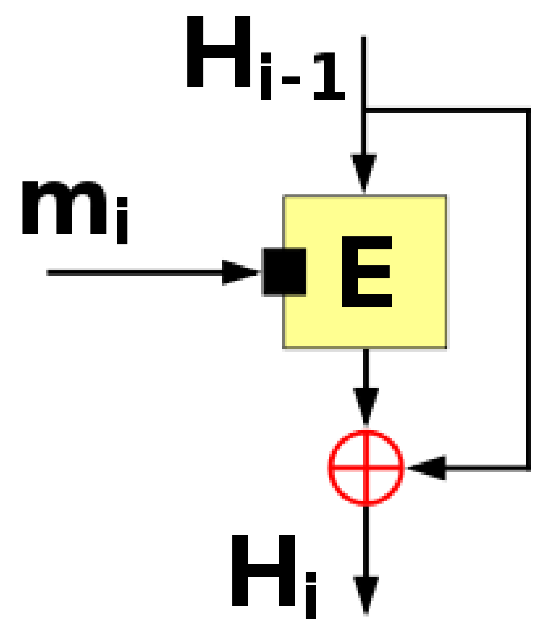
$$\mathbf{H_i = E\,(\,M_{i,}\,H_{i-1}\,), \quad M_o = 0}$$

- Final ciphertext $H_p$ is the hash output
- Secure?

# CHF from a Block Cipher

Davies-Meyer CHF:

- $H_i = H_{i-1} \oplus E(M_{i,} H_{i-1})$, $H_o$=0
- Compression function is secure if is a secure block cipher



# Hash Function Examples

| | **MD5 (defunct)** | **SHA-1 (weak)** | **SHA-256 (SHA-2 family, used today)** |
|---|---|---|---|
| Digest length | 128 bits | 160 bits | 256 bits |
| Block size | 512 bits | 512 bits | 512 bits |
| # of steps | 64 | 80 | 64 |
| Max msg size | $\infty$ | $2^{64}$-1 bits | $2^{64}$-1 bits |
| Security against collision attacks | <=18 bits | <= 63 bits | 128 bits |

# Latest standard: SHA-3

- Public competition by NIST, similar to AES:
- NIST request for proposals (2007)
- 51 submissions (2008)
- 14 semi-finalists (2009)
- 5 finalists (2010)
- Winner: Keccak (2012)
  - Designed by Bertoni, Daemen, Peeters, Van Assche.
  - Based on “sponge construction”, a completely different structure from prior CHF-s.

What are hash functions good for (besides integrity)?

# Message Authentication Using a Hash Function

Use symmetric encryption (AES or 3-DES) and a hash function

- Given message M
- Compute H(M)
- Encrypt H(M) in ECB or CBC mode
- Result is: $E_K(H(M)) = MAC$
- Alice sends to Bob: MAC, M
- Bob receives MAC',M' decrypts MAC' with K, hashes result and checks if: $D_K(MAC') =?= H(M')$

**Collision → MAC forgery!**

# Using Hash for Authentication

Alice and Bob share a secret key $K_{AB}$

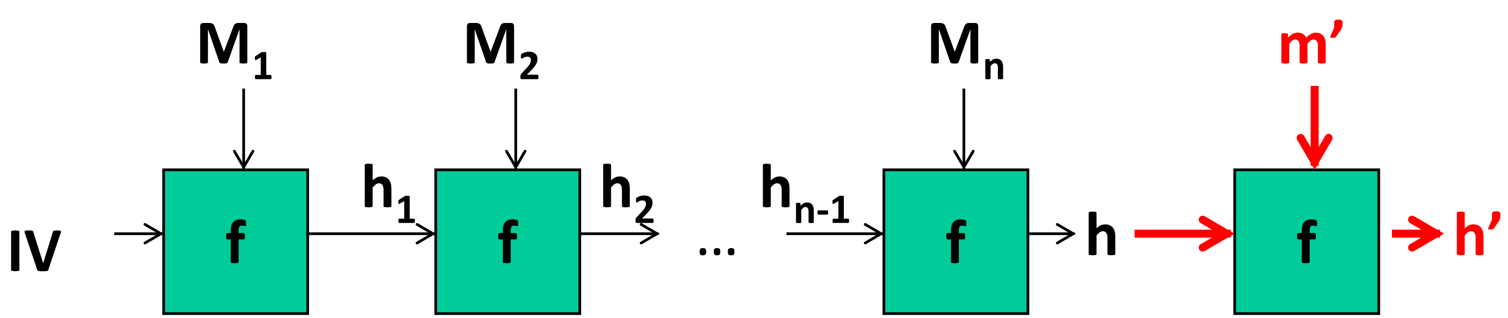
1. Alice ➔ Bob: random challenge $r_A$
2. Bob ➔ Alice: $H(K||r_A)$, random challenge $r_B$
3. Alice ➔ Bob: $H(K||r_B)$

Only need to compare H() results

# Using Hash to Compute a MAC: message integrity **and authentication**

- Just computing and appending H(m) to m is enough for integrity but not for authenticity
- Need a "Keyed Hash":
  - Prefix:
    - MAC: H(K || m), almost works, but …
    - Allows concatenation with arbitrary message:

$$H(\,K \,||\, m \,||\, m'\,)$$



# Using Hash to Compute a MAC: message integrity **and authentication**

- Just computing and appending H(m) to m is enough for integrity but not for authenticity
- Need a “Keyed Hash”:
  - Prefix:
    - MAC: H(K || m), almost works, but …
    - Allows concatenation with arbitrary message:

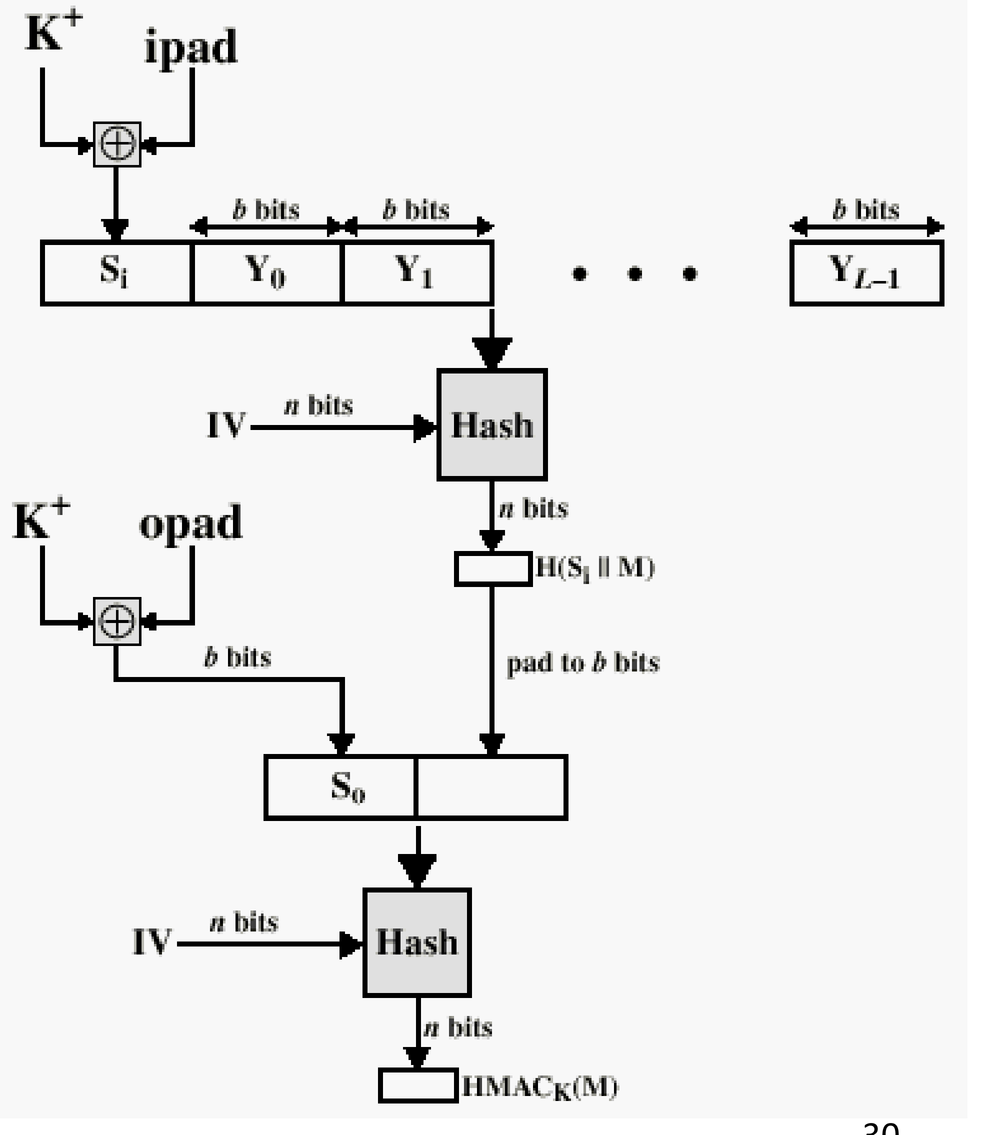
      H( K || m || m’ )
  - Suffix:
    - MAC: H(m || K)
    - Works better, but what if m’ is found such that H(m)=H(m’)?
  - HMAC:
    - H ( K || H (K || m) )

# Hash Function-based Keyed MAC (HMAC)

- **Main Idea**: Use a MAC derived from any CHF
  - hash functions do not use a key, therefore cannot be used directly as a MAC
- **Motivations for HMAC:**
  - Cryptographic hash functions run faster in software than many encryption algorithms such as 3-DES
  - No need for the function to be reversible
  - No US Government export restrictions (was important in the past)
- **Status**: designated as mandatory for IP security
  - Also used in TLS, IPSec, etc.

# HMAC Algorithm

- Compute H1 = H() of the concatenation of M and K1
- To prevent an "additional block" attack, compute again H2= H() of the concatenation of H1 and K2
- Notation:
  - $K^+$ = K padded with 0's
  - ipad=00110110 x b/8
  - opad=01011100 x b/8
- Execution:
  - Same as H(M), plus 2 blocks



# Hash Function Examples

| | **MD5 (defunct)** | **SHA-1 (weak)** | **SHA-256 (SHA-2 family, used today)** |
|---|---|---|---|
| Digest length | 128 bits | 160 bits | 256 bits |
| Block size | 512 bits | 512 bits | 512 bits |
| # of steps | 64 | 80 | 64 |
| Max msg size | $\infty$ | $2^{64}$-1 bits | $2^{64}$-1 bits |
| Security against collision attacks | <=18 bits (2013) | <= 63 bits (2005) | 128 bits |