UNIVERSITÄT DUISBURG ESSEN

Sicherheit in Kommunikationsnetzen
(Network Security)

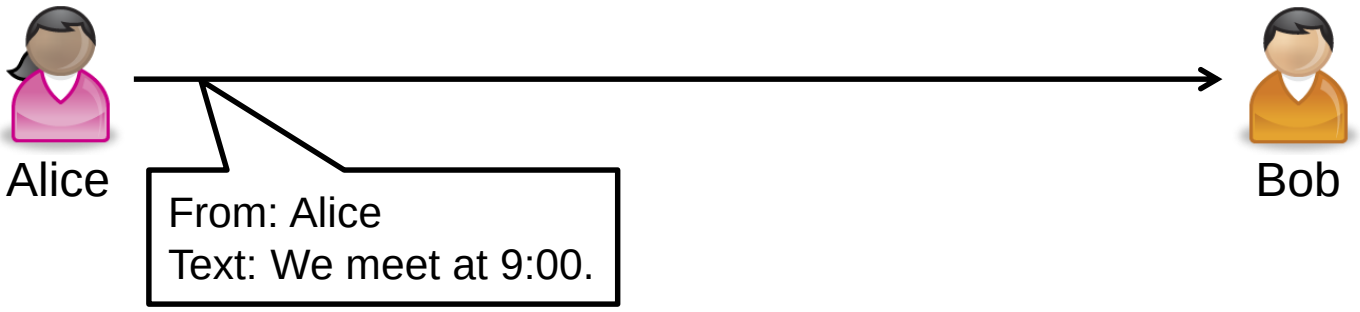
Authentication and Integrity

Dr.-Ing. Matthäus Wander

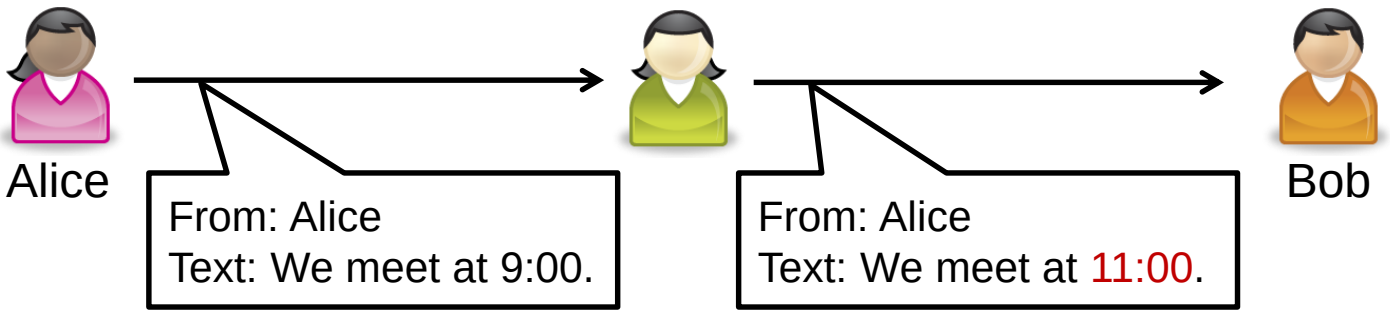
Universität Duisburg-Essen

Attack Model

- Alice sends a message to Bob

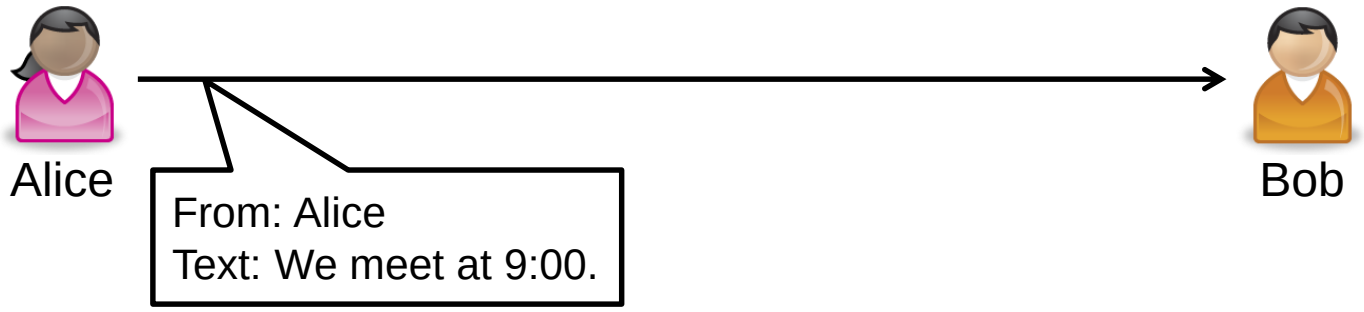


- Attack on integrity: change message contents

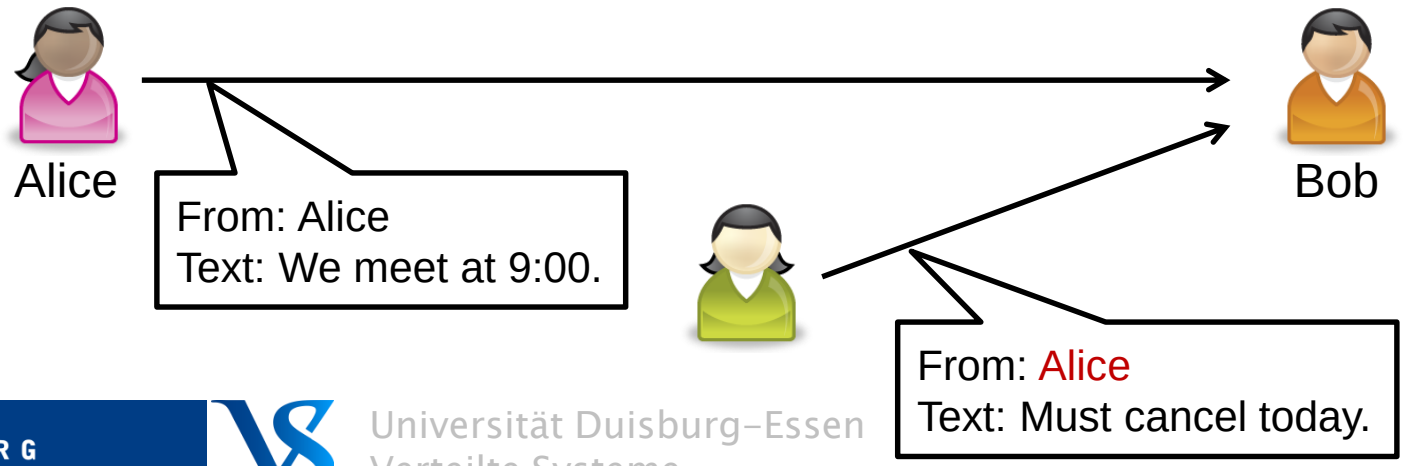


Attack Model (2)

- Alice sends a message to Bob



- Attack on authenticity: claim to be someone else



Example: Email over plain SMTP

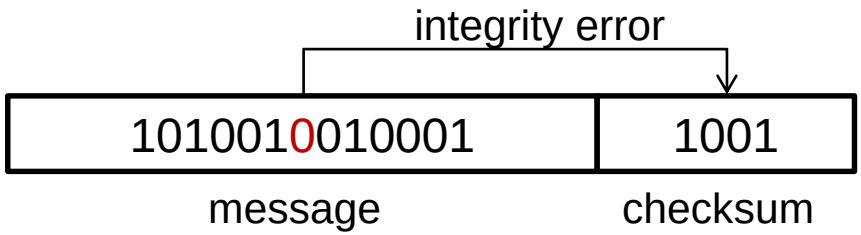
```
> 220 mail.example.com SMTP Foo Mailserver
< HELO mail.example.org
> 250 Ok
< MAIL FROM: <alice@example.org>
> 250 Ok
< RCPT TO: <bob@example.com>
> 250 Ok
< DATA
> 354 End data with <CR><LF>.<CR><LF>
< From: <alice@example.org>
< To: <bob@example.com>
< Subject: Testmail
<
< Testmail
< .
> 250 Ok
< QUIT
> 221 Bye
```

Example: Email over plain SMTP (2)

- In its original form, there was no authentication of the sender in SMTP
  - Anyone can send an email from any email address
  - Servers had no means to check whether sender is real
- … and no integrity check of the contents
  - Man-in-the-middle can change the message
- Preventive measures
  - Digital signatures over emails with S/MIME or PGP
  - For other measures ⇨ see lecture Internet Technology

Not suitable: Error-Detecting Codes

- Idea: use error-detecting code
  - e.g. checksum, parity bit, CRC, Hamming code
- Purpose: detect integrity errors in network messages or data storages
  - If bit flips in message, it won‘t match checksum

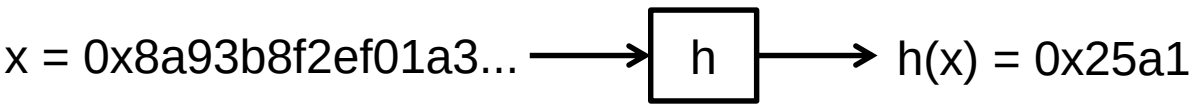


- But: protects from accidental modification only
  - Attacker can easily adapt checksum to match message

Hash Functions

- Definition: a hash function h is a function with the following properties:

1. Compression: h maps an input of arbitrary finite bitlength to a fixed-length output h(x)
   - The output h(x) is called hash value
2. Ease of computation: given h and x, it is easy to compute h(x)



Hash Functions (2)

- Usually additional properties are required
  - e.g. for key-value stores and other use cases
- Deterministic: the same input x generates always the same output y=h(x)
- Uniform distribution: all output values y=h(x) occur with the same probability
  - Even if we use only a subset of all possible inputs as x
- For security use, we need one-way functions
  - Deriving x from y=h(x) should be hard

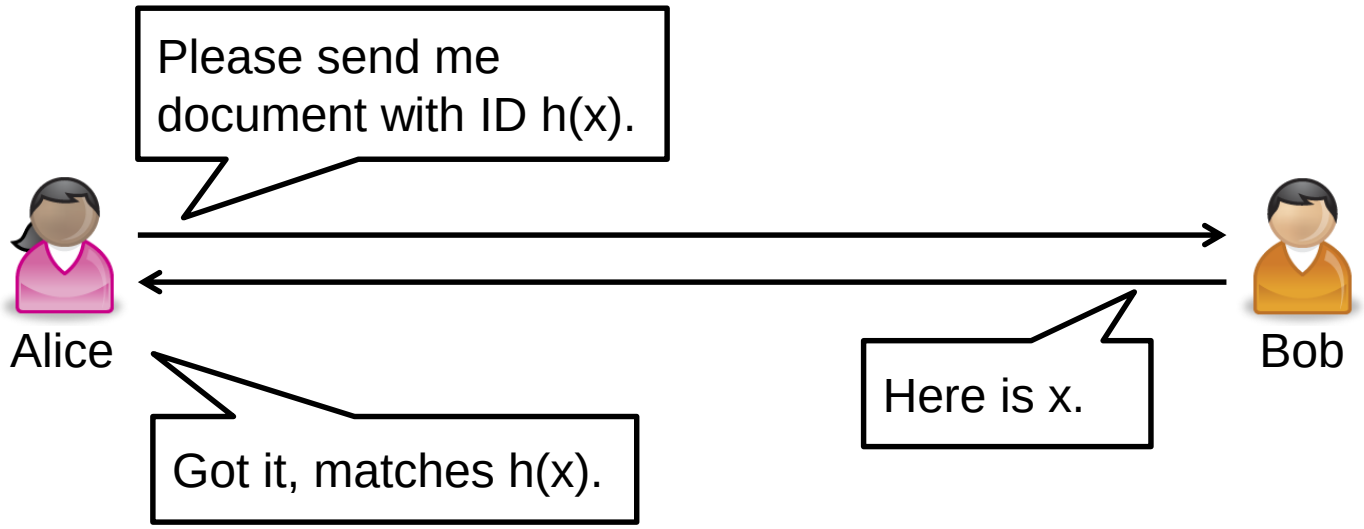
Cryptographic Hash Functions

- Definition: a cryptographic hash function is a hash function with additional properties:

1. Pre-image resistance: for a given hash value y, it is infeasible to find an input x so that $h(x)=y$
2. Second pre-image resistance: for a given x, it is infeasible to find an x‘ so that $h(x)=h(x')$
   - This is also called *„weak collision resistance“*
3. Collision resistance: it is infeasible to find any two x_1, x_2 so that $h(x_1)=h(x_2)$

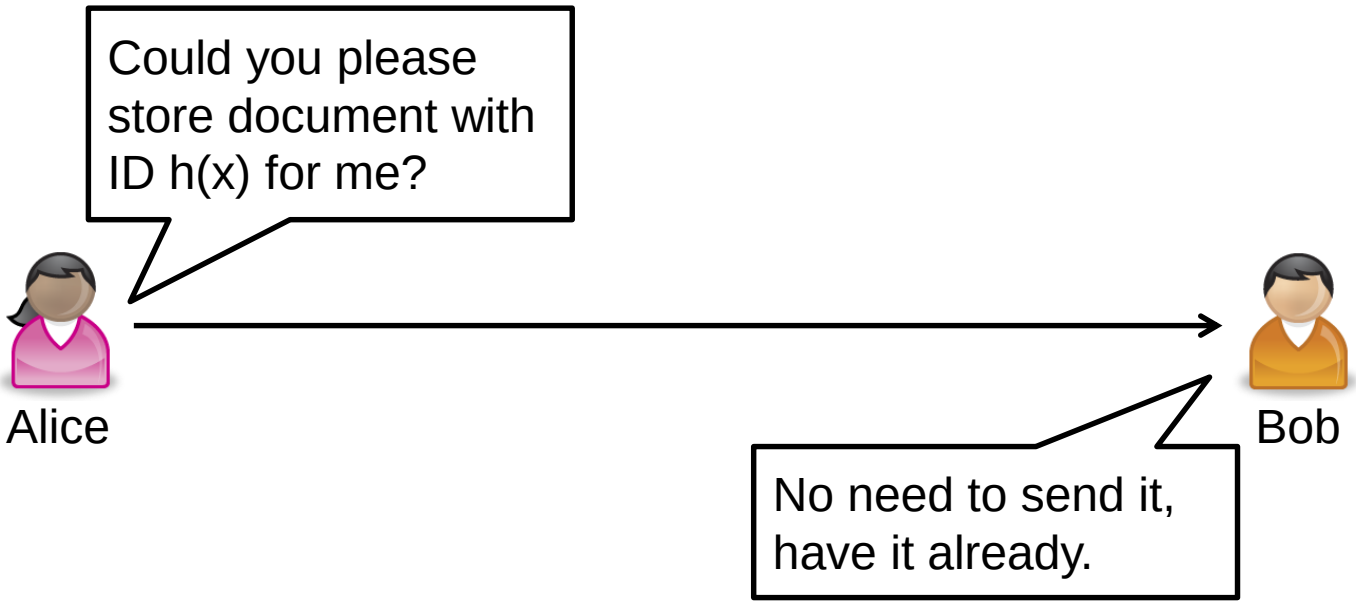
Use Case: Data Identifier

- A cryptographic hash value h(x) is a fingerprint or distinct identifier of message x



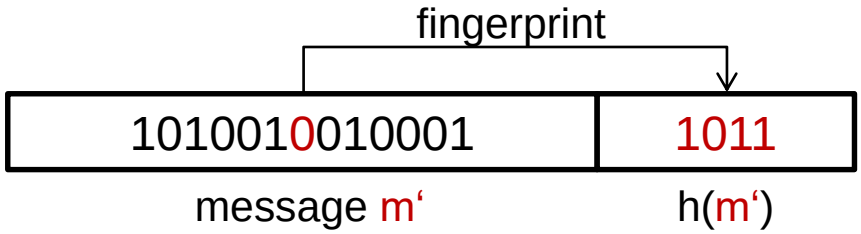
Use Case: Data Deduplication

- Data deduplication: if two x_1 and x_2 have the same $h(x_1)=h(x_2)$, we can deduce that $x_1=x_2$



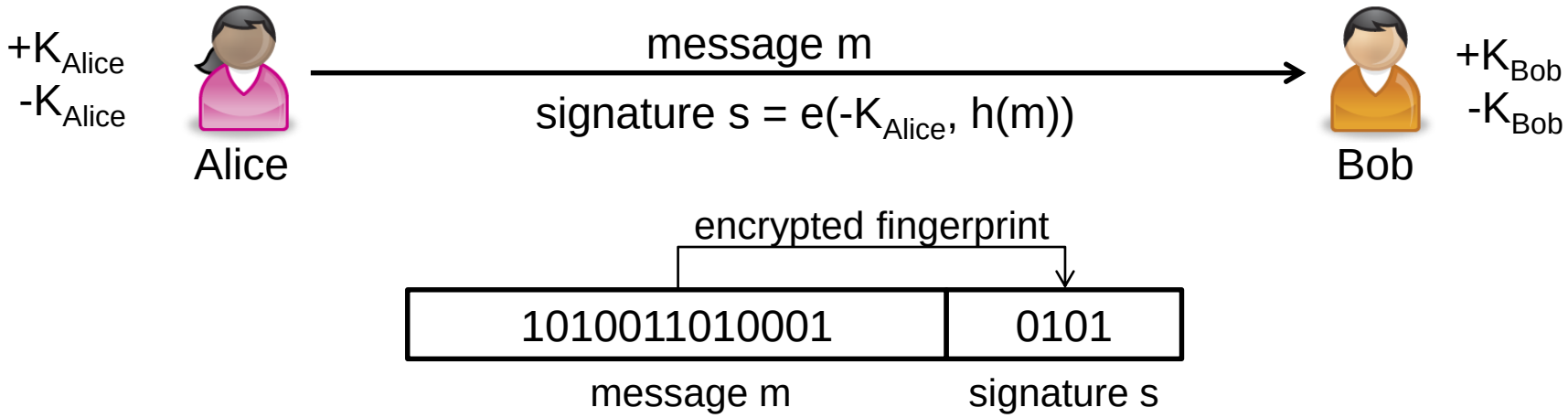
Use Case: Data Integrity

- Idea: send x and h(x) for integrity verification



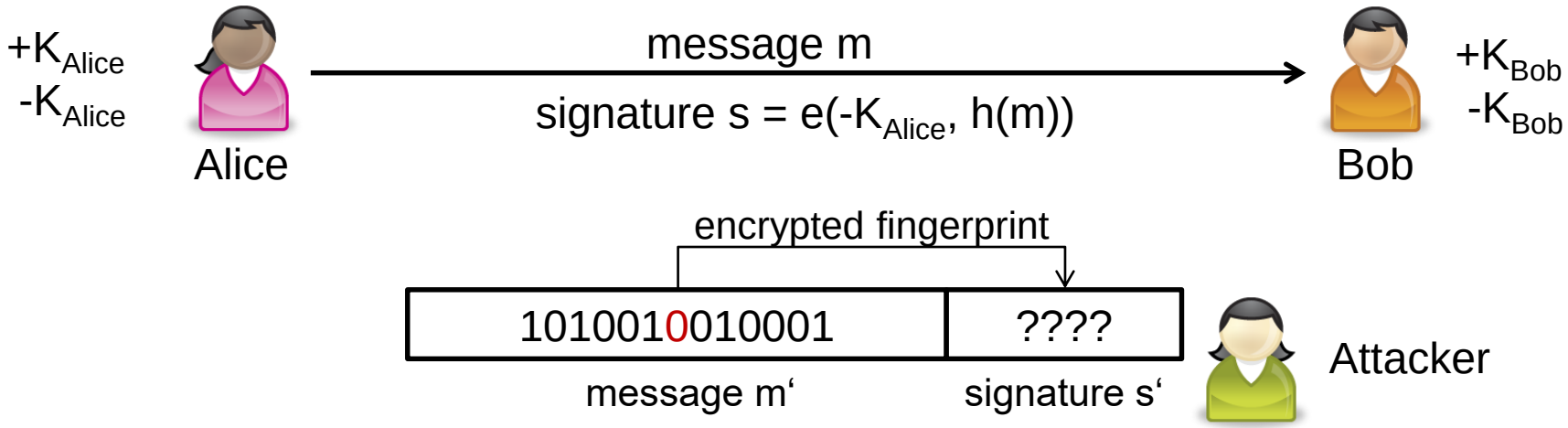
- Insecure: attacker can replace m with m‘ and h(m) with h(m‘)
- We need to authenticate the fingerprint h(x)
  - with asymmetric cryptography: digital signature
  - with symmetric cryptography: message authentication code (MAC)

Digital Signatures



- Bob receives m and s
- Decrypts $d(+K_{Alice}, s) = h(m)$
- Computes h(m) and verifies that h(m)=h(m)
  - If Bob receives m‘ ≠ m, then h(m‘)≠h(m)

Digital Signatures (2)



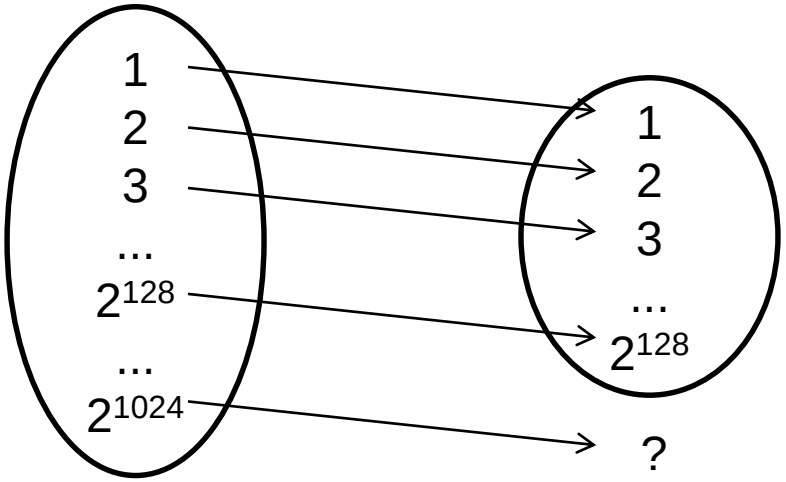
- Attacker can replace m with m‘ and can compute h(m‘) but won‘t be able to encrypt it properly without knowledge of $-K_{Alice}$
  - Security based on asymmetric cipher, e.g. RSA
  - ... and second pre-image resistance of hash function

Digital Signatures (3)

- Do we need collision resistance for signatures?
  - Finding any two x_1, x_2 with $h(x_1)=h(x_2)$
- Yes, otherwise the following attack is possible:
  - Mallory finds $x_{harmless}$,$x_{harmful}$ with $h(x_{harmless})=h(x_{harmful})$
  - Sends $x_{harmless}$ to Alice and let her sign it
  - Mallory claims Alice has signed $x_{harmful}$
- Example attack scenarios:
  - Rogue certificates that wouldn‘t be signed otherwise
  - False contract documents

Collisions in Hash Functions

- Idea: design a hash function with no collisions
  - Input: arbitrary x of up to 1024 bits
  - Output: 128-bit hash value h(x)=y
- How to map 2^{1024} input elements to 2^{128} output elements with an injective function?
  - Impossible
  - We cannot avoid collisions
  - But finding them should be hard



Attacking Cryptographic Hash Functions

- Let h be a hash function that produces 128-bit hash values
  - We can always attempt a brute-force attack
- Pre-image attack
  - Given h(x)=y, it will take 2^{128} hash operations to find a matching x (on average after 2^{127} operations)
- Second pre-image attack
  - Given x, it will take 2^{128} hash operations to find another x‘ with h(x)=h(x‘)
- What about a collision attack?

Birthday Problem

- For collision attacks, the attacker can exploit the birthday problem or birthday paradox
- In a room with 23 people, what is the probability that at least two people share the same birthday?
  - Consider 365 days, disregard years
  - Assume uniform birthday distribution
- Answer: slightly more than 50%
  - Intuitively, one may expect a single-digit percentage

Birthday Problem (2)

- The answer seems surprising because we do not match 1 person to 22 others
  - Instead, we match any pair out of 23 people

$$P = 1 - \frac{365 \cdot 364 \cdot 363 \ldots \cdot 343}{365^{23}} \approx 0{,}507$$

- Count the non-matching birthday combinations
  - First person has birthday on any of 365 days
  - Second person on any except for the first chosen...
- Divide by all possible birthday combinations

Birthday Attack on Hash Functions

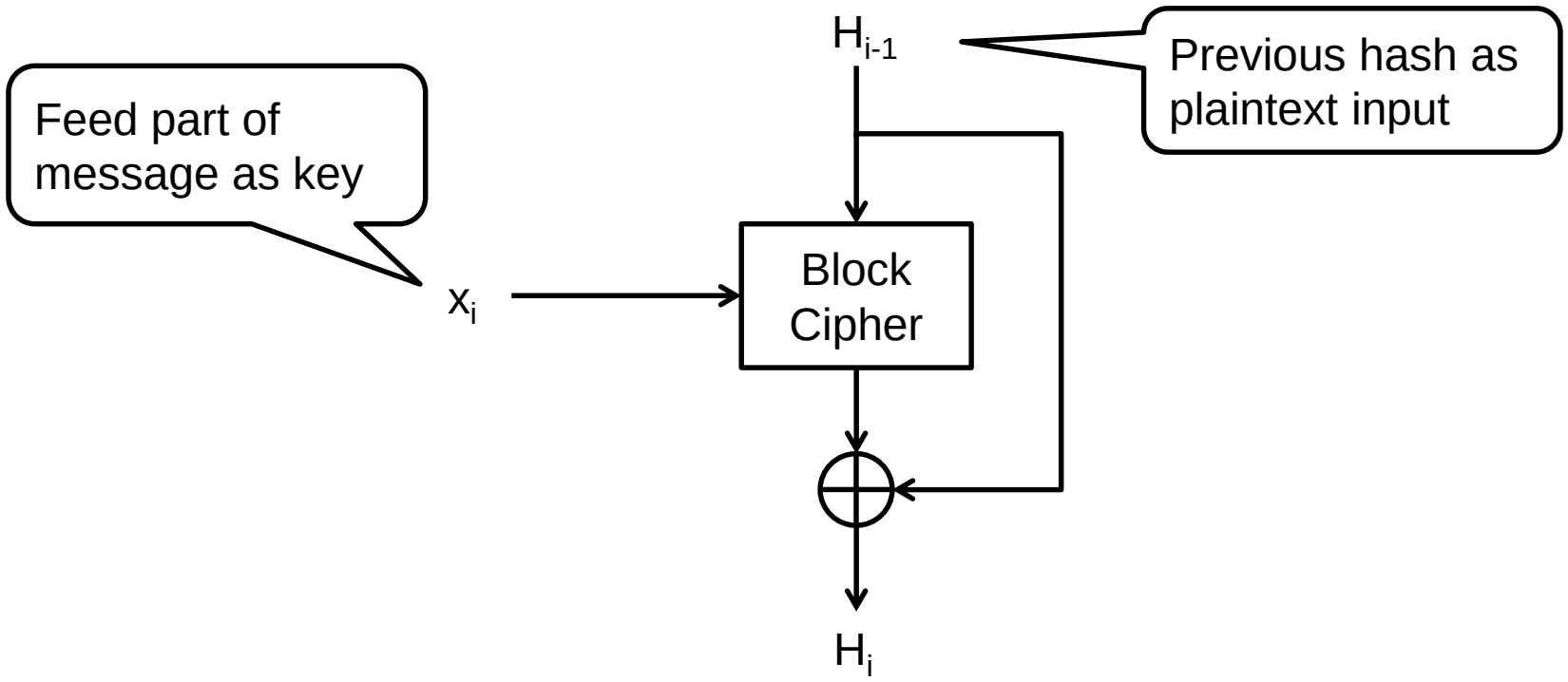
- Birthday attack: chose arbitrary inputs x_1, x_2, x_3, ... and compute their hash values
  - Store all $x \rightarrow h(x)$ pairs in a table
- Given N possible hash values, a matching pair will be found after $\sqrt{N}$ choices on average
  - i.e. probability for finding a match is $>50\%$
- After $2^{(n/2)}$ operations on average for hash functions with n-bit hash value
  - e.g. with 2^{128} hash values after 2^{64} operations

Construction of Cryptographic Hash Functions

- Two general ways to build hash functions

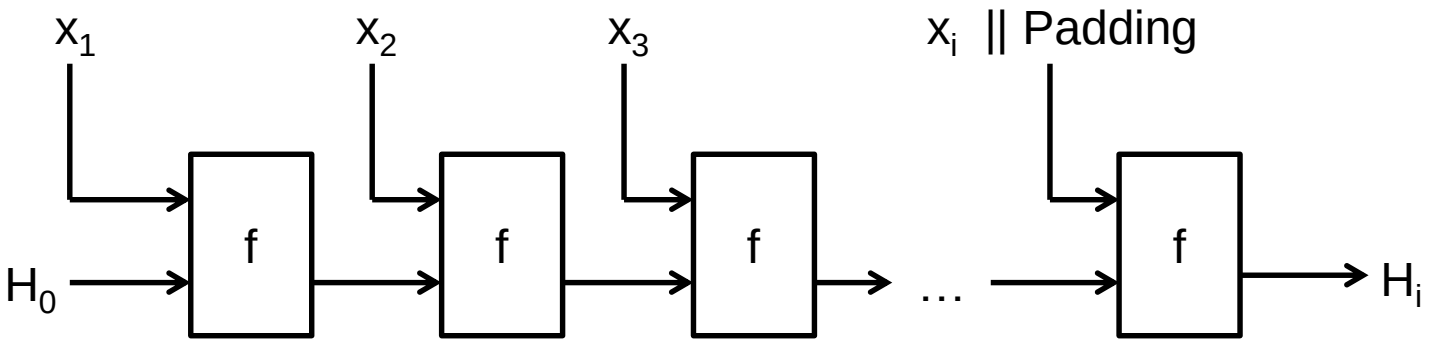
1. Hash functions based on block ciphers
   - Encrypt input message with any block cipher
   - Hash function is keyless, thus derive key from input
   - XOR output ciphertexts to produce a fixed-length hash value
2. Dedicated hash functions
   - Designed specifically for cryptographic hashing
   - Some functions have similar structures

Davis-Meyer Construction



- $H_i = H_{i-1} \oplus e_{x_i}(H_{i-1})$
- H_0 is a public constant (initial value)

Merkle-Damgård Construction



- Chained call of a compression function *f*
  - Last input block is padded (aligned to block length)
- H_0 is a public constant (initial value)
- h is collision-resistant if *f* is collision-resistant

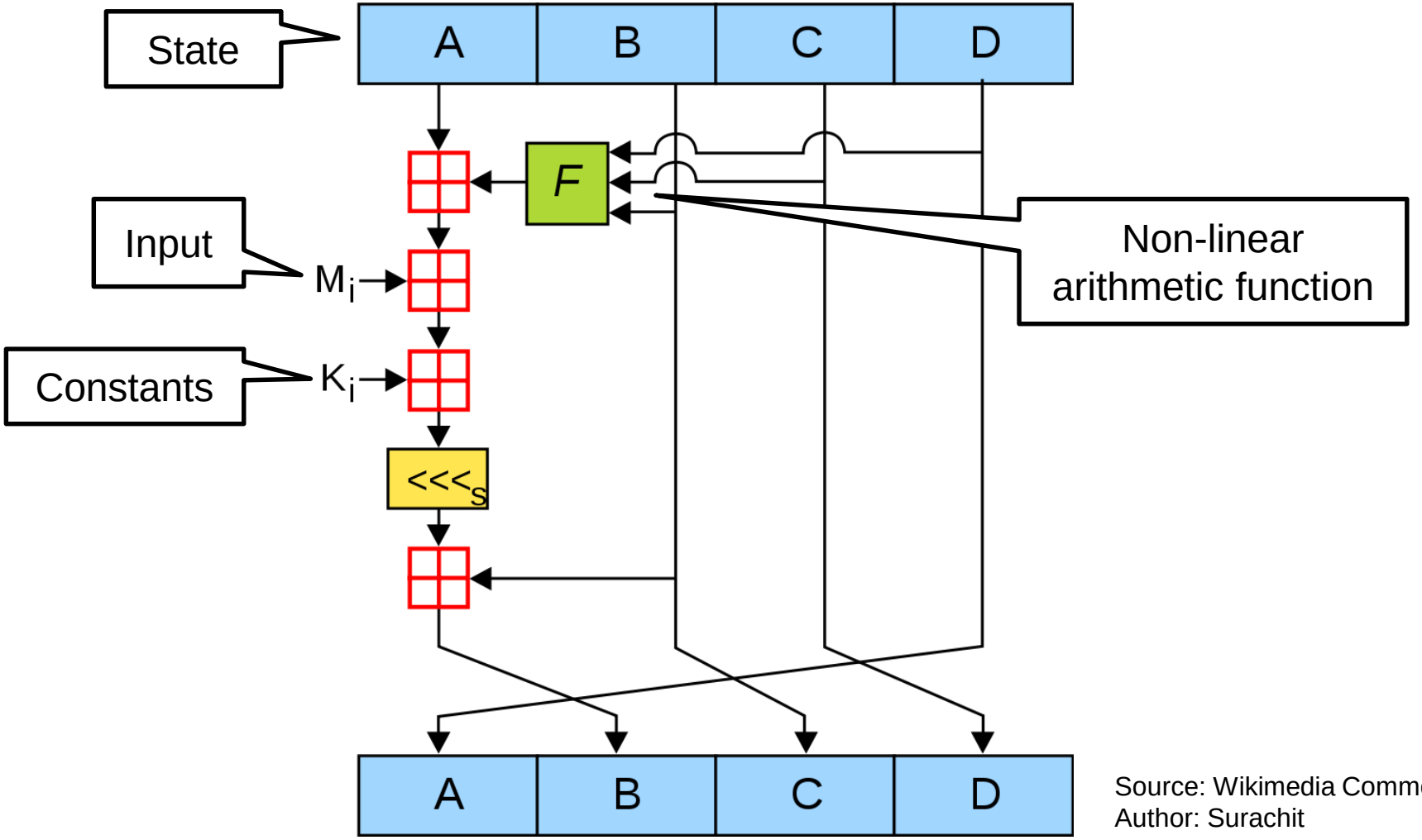
Cryptanalysis of Cryptographic Hash Functions

- Cryptanalysis concentrates on the function *f*
  - Cryptanalysts try to find efficient techniques to produce collisions for a single execution of *f*
- Length of a hash value
  - Larger is better (as with cipher keys)
  - Primarily motivated by birthday attacks
  - State of the art: ≥256 bit
  - A birthday attack requires effort of order 2^{128}
  - Today, 2^{128} is considered infeasible

MD5

- Designed by Ron Rivest, published in 1992
- 128-bit hash value (insecure)
- Merkle-Damgård construction
- Process input in 512-bit blocks
  - Four 32-bit words for internal state
  - *f* consists of 64 rounds
- Simple 32-bit arithmetic, very fast

MD5: Schematic of one Round



Source: Wikimedia Commons
Author: Surachit

MD5: Security

- MD5 is not collision-resistant
- Finding collisions
  - 2005: 8h @ 1.6 GHz PC (Vlastimil Klima)
  - 2006: 5 min @ 3 GHz Pentium4 (Marc Stevens)
  - 2006: 17s @ 3.2 GHz Pentium4 (Vlastimil Klima)
- MD5 is not pre-image resistant
  - $2^{123.4}$ operations (Sasaki & Aoki, 2009)
- ⇨ Insecure, don't use

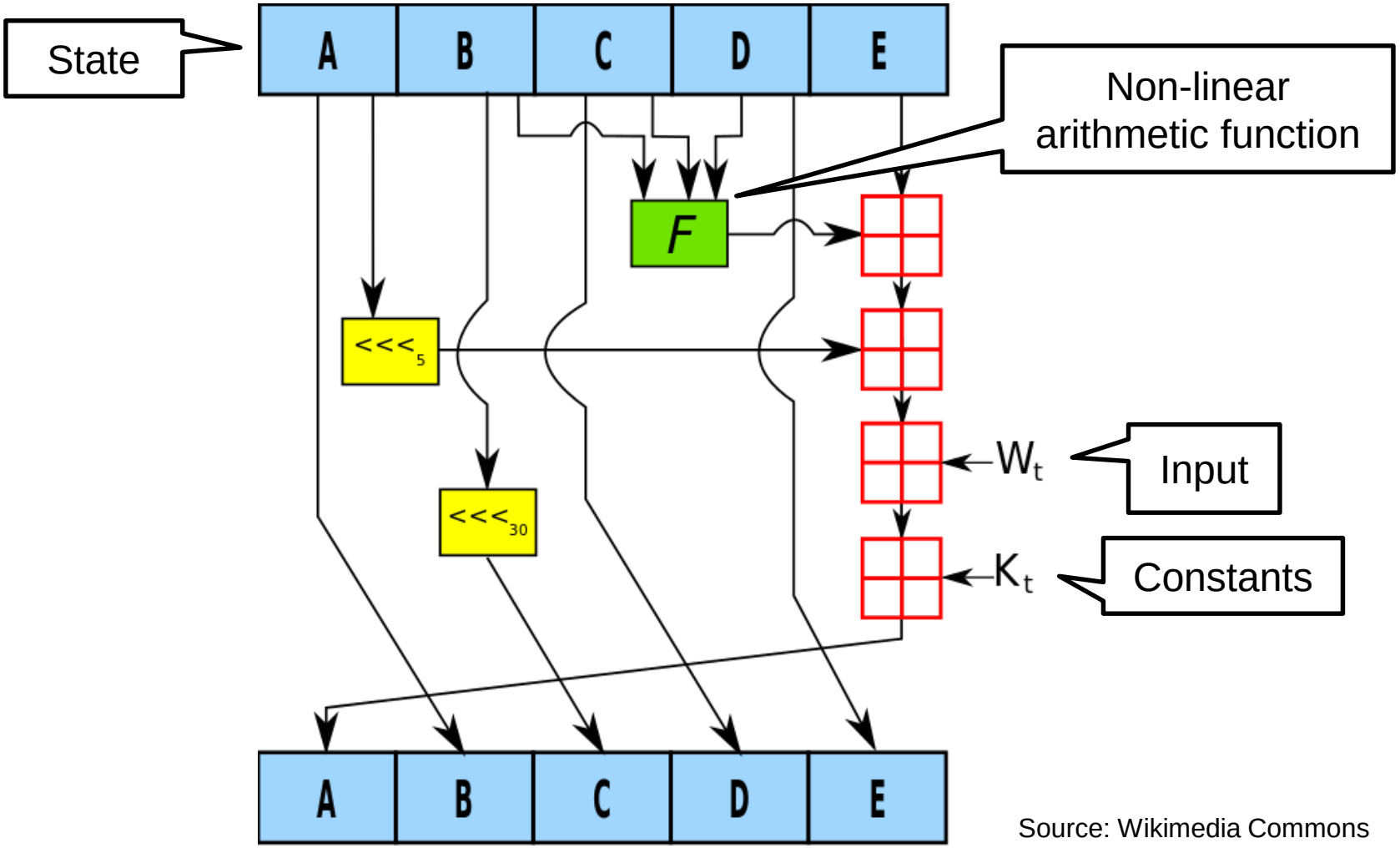
MD5: Security (2)

- Although security deficiencies were known, various applications continued to support MD5
- 2008: Stevens et al. used a chosen-prefix collision to create a rogue CA certificate
  - Signed by RapidSSL due to MD5 hash collision
  - Can be used to issue trusted certificates
- 2012: Flame malware used a chosen-prefix collision for a rogue code signing certificate
  - Signed by Microsoft due to MD5 hash collision
  - Flame malware appears like legitimate software

SHA-1

- Designed by NSA, published by NIST in 1995
  - Based on earlier version (SHA-0) which was flawed
- 160-bit hash value (weak)
- Merkle-Damgård construction
- Process input in 512-bit blocks
  - Five 32-bit words for internal state
  - *f* consists of 80 rounds
- Similar to MD5, but ~25% slower

SHA-1: Schematic of one Round



Source: Wikimedia Commons

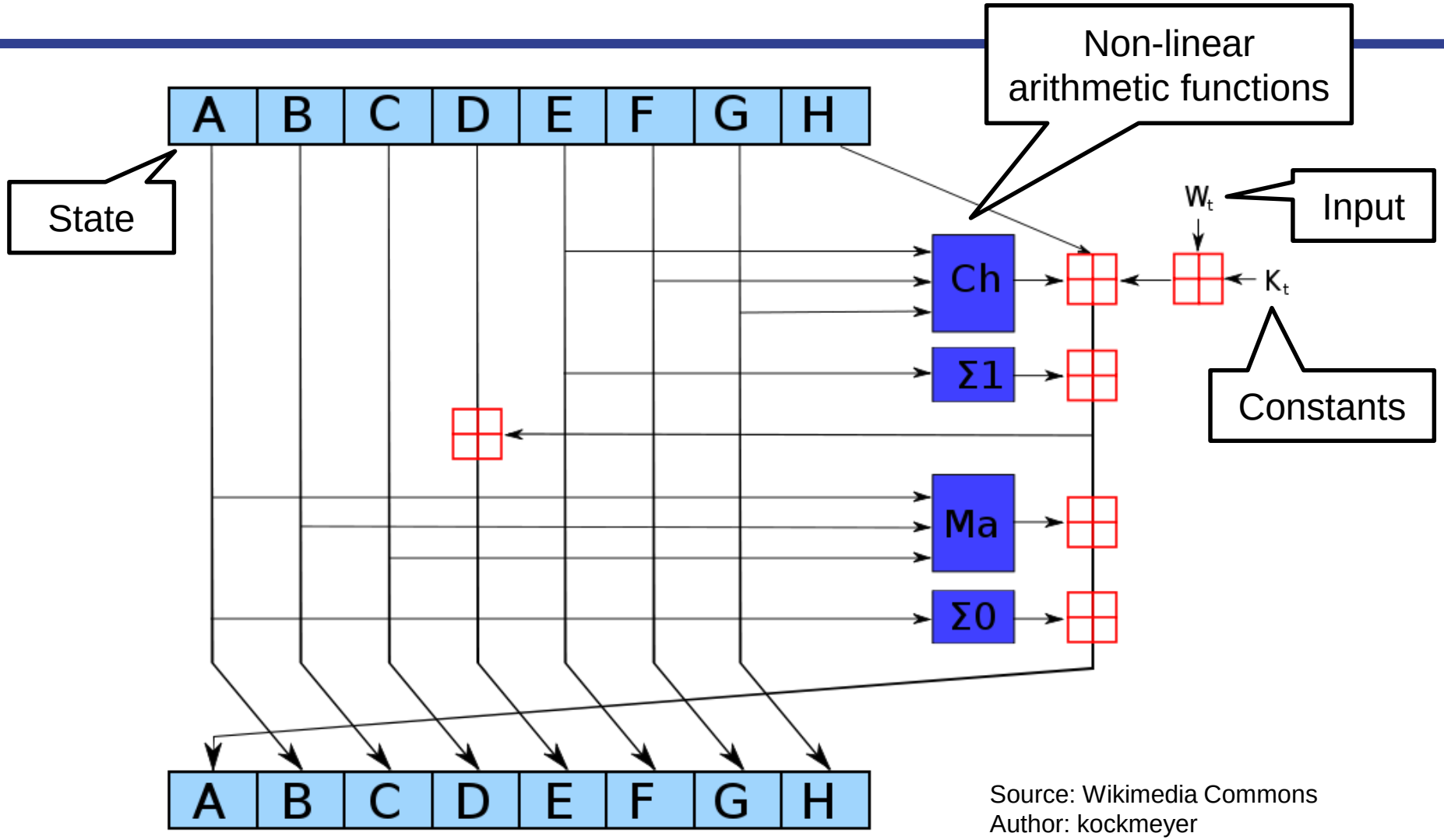
SHA-1: Security

- SHA-1 is not collision-resistant
  - 2005: first 2^{69}, then 2^{63} operations (Wang et al.)
- 2017: SHA-1 collision in practice (Stevens et al.)
  - Two different PDF documents with identical SHA-1 hash value
  - Identical-prefix collision
  - Computation required 2^{63} SHA-1 operations
- SHA-1 is insecure and being deprecated
  - However, no pre-image attacks known yet

SHA-2

- Designed by NSA, published by NIST in 2001
  - Family of similar hash functions
- Hash value: 224, 256, 384 or 512 bit
- Merkle-Damgård construction
- Process input in 512-bit or 1024-bit blocks
  - Eight 32-bit or 64-bit words for internal state
  - f consists of 64 or 80 rounds
- Similar to SHA-1, but 30-50% slower

SHA-2: Schematic of one Round



Source: Wikimedia Commons
Author: kockmeyer

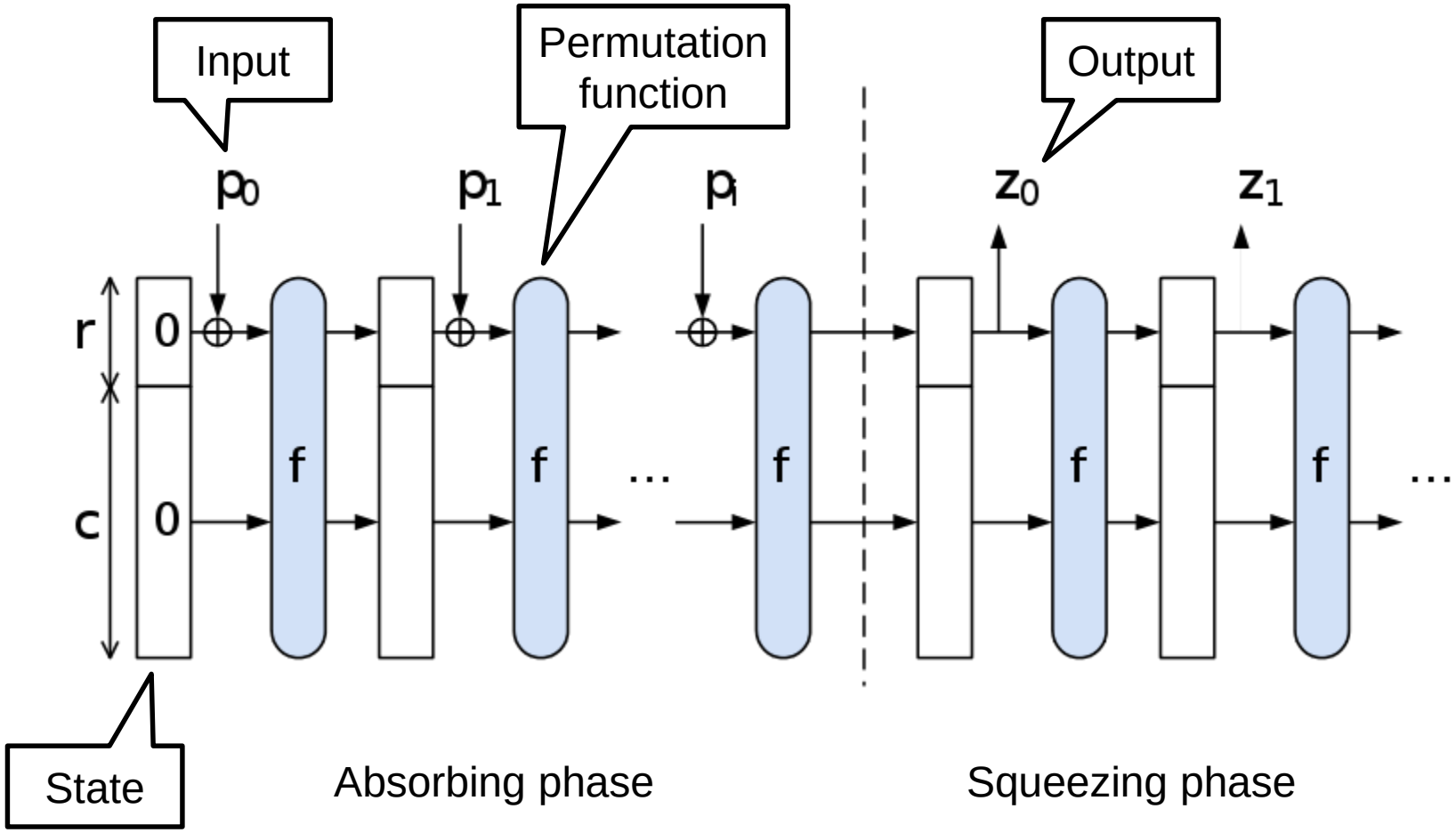
SHA-2: Security

- No attacks on collision resistance known yet
  - Some research work on reduced round versions exists
- No attacks on pre-image resistance known yet
- Secure so far ☺
- But: yet another Merkle-Damgård hash function
  - Does SHA-2 inherit weaknesses from using a similar design as MD5/SHA-1?

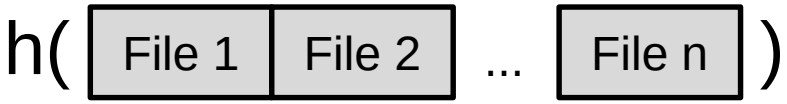
SHA-3

- Designed by Bertoni et al. as Keccak, published by NIST as SHA-3 in 2015
- Can output any hash value size
  - Including the ones from SHA-2 for easy replacement
- Sponge construction
  - Consists of absorbing phase (reading input)
  - … and squeezing phase (writing output)
- Process input in variable block sizes
  - 5x5 array of 64-bit words for internal state

Sponge Construction



Merkle Hash Tree

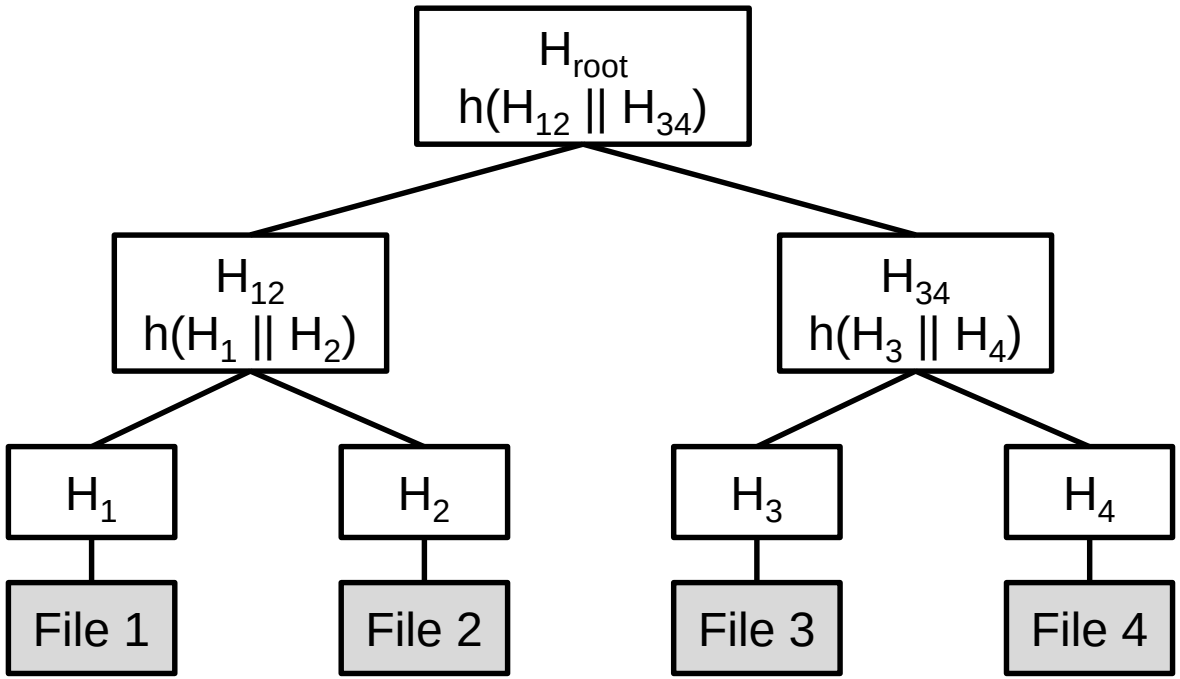


- How to compute hash value over lots of data?
  - Append data to list, compute hash value over list
- What if parts of the data change?

  - Recompute hash value over complete list
- Idea: use intermediate hash values

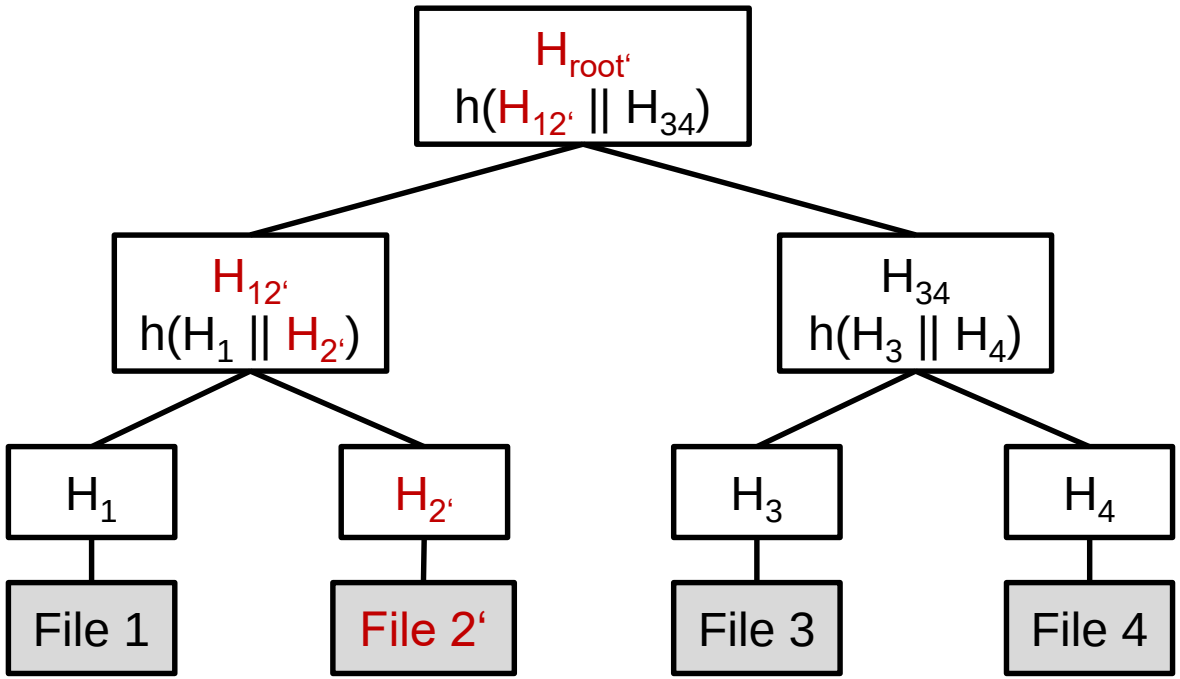
Merkle Hash Tree (2)

- Merkle hash tree: create a tree of hash values
- In addition to *n* leaf hash values, we need *n−1* hash values for the inner nodes



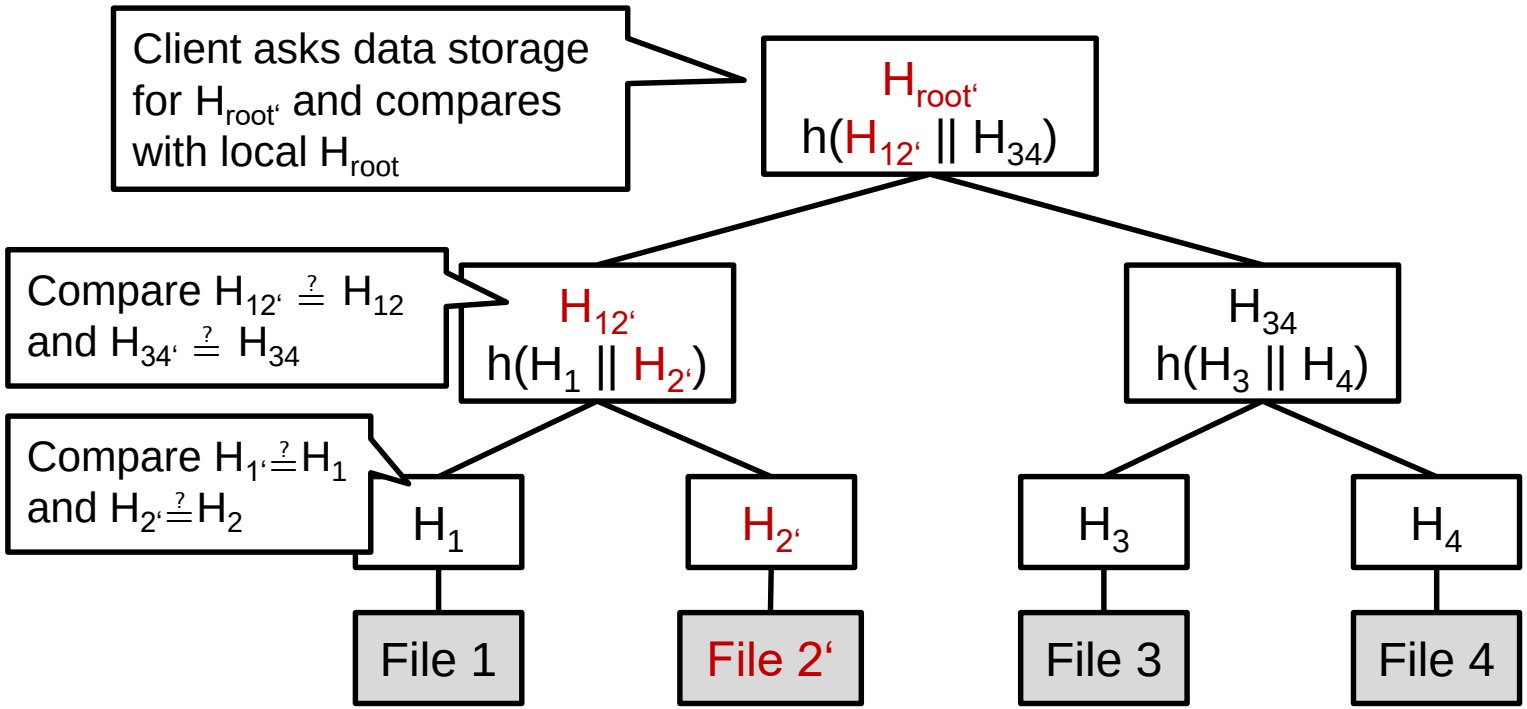
Merkle Hash Tree (3)

- When one leaf hash value changes, update log(n) hash values of inner nodes

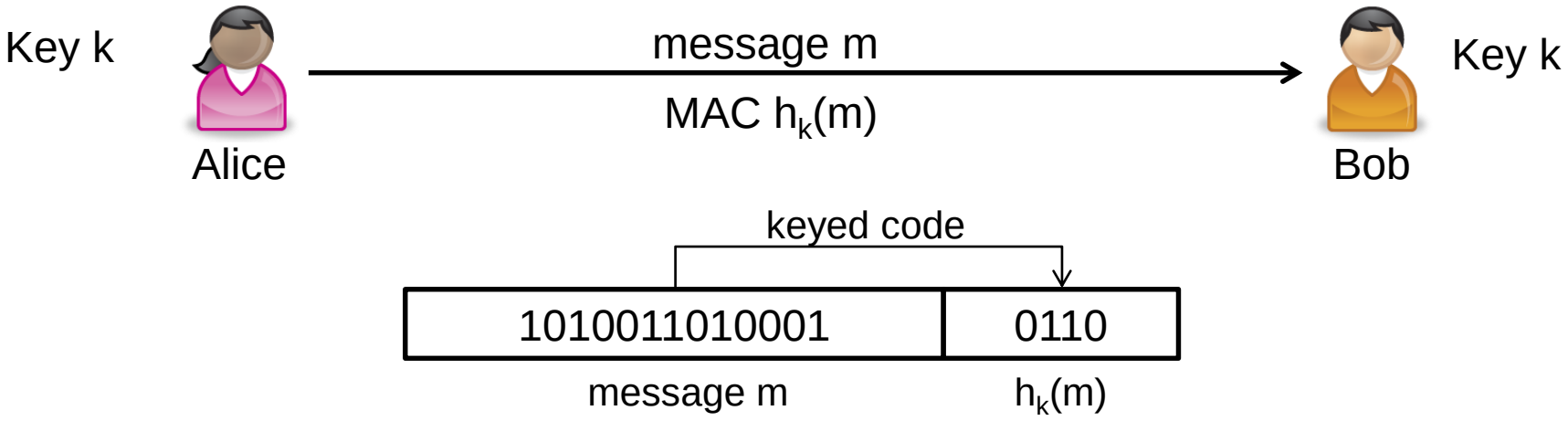


Merkle Hash Tree (4)

- Use case: synchronization of data storages
- Efficient identification which file has changed

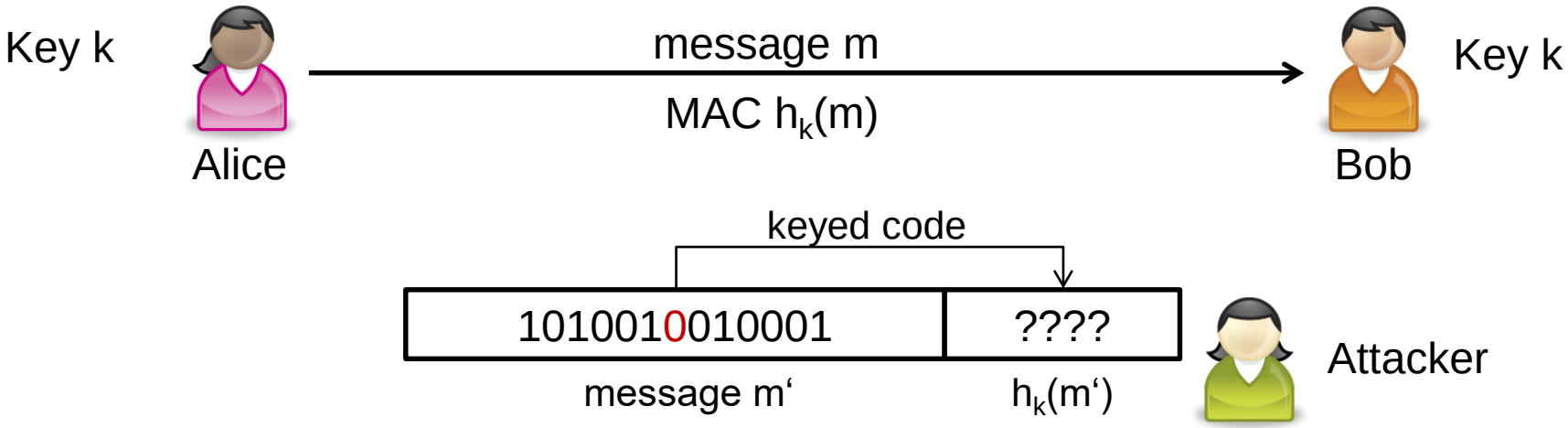


Message Authentication Codes



- Similar to digital signatures, we can use a code with a symmetric key as secret parameter
  - Message authentication code (MAC)
- Purpose: authenticity/integrity

Message Authentication Codes (2)



- Knowledge of k required to verify MAC
- Attacker cannot forge valid MAC without k
  - But Bob can! Authenticity between Alice/Bob only
- MAC functions are usually faster than signatures

MAC Properties

- Definition: a message authentication code (MAC) function h_k is a function with these properties:

1. Compression: h_k maps an input of arbitrary finite bitlength to a fixed-length output $h_k(x)$
2. Ease of computation: given h, k and x, it is easy to compute $h_k(x)$
   - 1 and 2 similar to non-cryptographic hash functions
3. Computation resistance: given any number of input and MAC pairs $(x_i, h_k(x_i))$, it is infeasible to compute $h_k(x')$ for another $x' \neq x_i$ without k

MAC Properties (2)

- Computation resistance implies that we cannot recover the key k from known $(x_i, h_k(x_i))$ pairs
  - With a known k, computing new $h_k(x')$ would be easy
- But: key non-recovery alone is not enough
  - We want to avoid forging MACs $h_k(x')$ for new x'
- This is different than encryption
  - Encryption achieves confidentiality: hide the plaintext
  - Attacker can manipulate message without knowing k
  - We're not trying to hide x, we want to protect its integrity so that the attacker cannot manipulate x

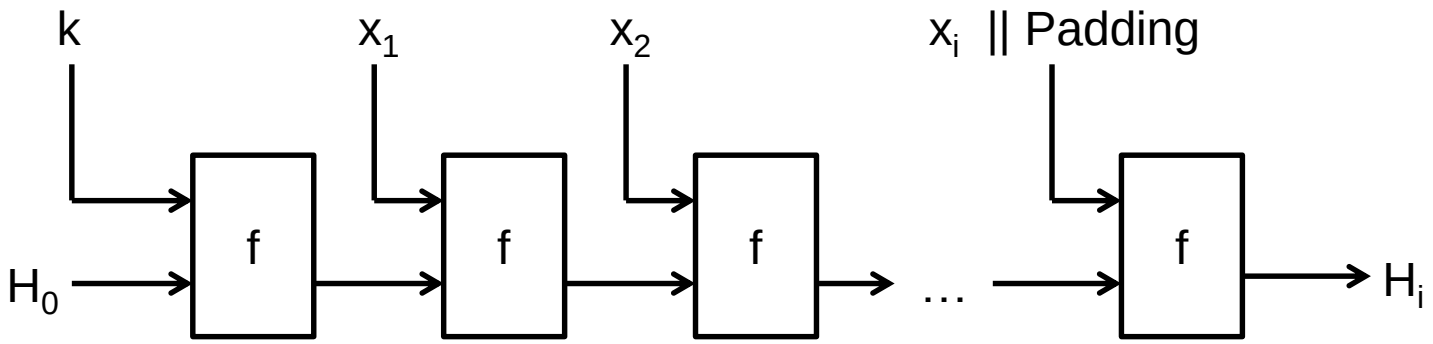
Construction of MAC Functions

- Two schemes for constructing MAC functions

1. Based on block ciphers
2. Based on cryptographic hash functions
   - Hash functions usually faster than block ciphers
   - Mix secret key with input and compute hash value

- Idea: $h_k(x) = h(k \,||\, x)$
  - Hash secret key concatenated with input message

Hash Function with Secret Prefix

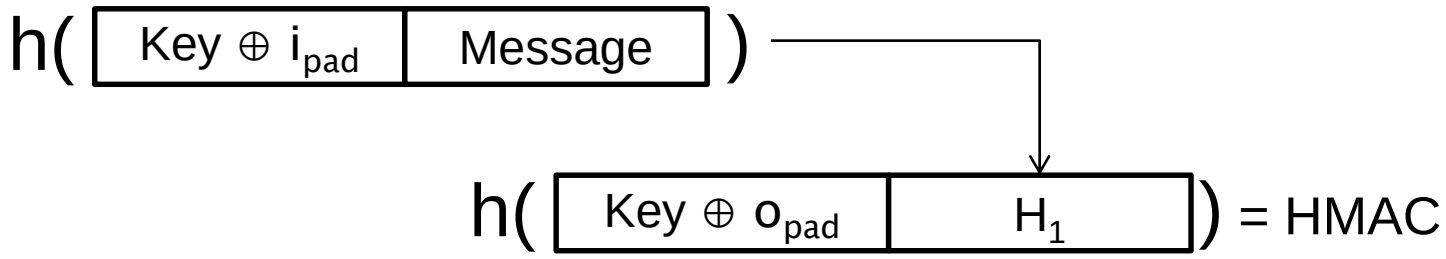


- Suppose a Merkle-Damgård hash function
- Input message: $x_1 x_2 \ldots x_i$ (known to attacker)
- MAC: $H = h(k \,||\, x_1 \,||\, x_2 \,||\, \ldots \,||\, x_i)$
- An attacker can append the message with x‘
  - Compute new MAC as h(H || x‘) without knowing k

Hash Function with Secret Suffix

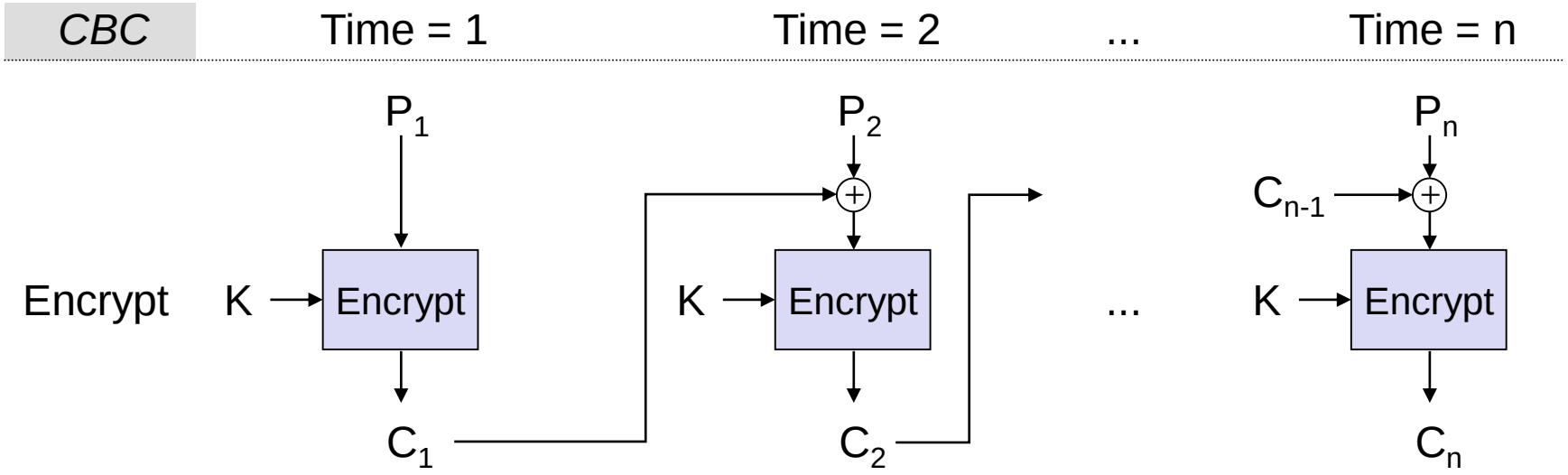
- Idea: $h_k(x) = h(x \,||\, k)$
  - Hash message first, then secret key
- Can be exploited with collision attacks
  - Attacker finds collision h(x)=h(x‘)
  - Lets Alice compute MAC $h(x \,||\, k)$
  - Attacker knows MAC for $h(x \,||\, k) = h(x' \,||\, k)$
- Example: SHA-256 with 256-bit secret key
  - 2^{128} operations for birthday attack
  - Considerably less than with a 256-bit cipher

Keyed-Hash Message Authentication Code



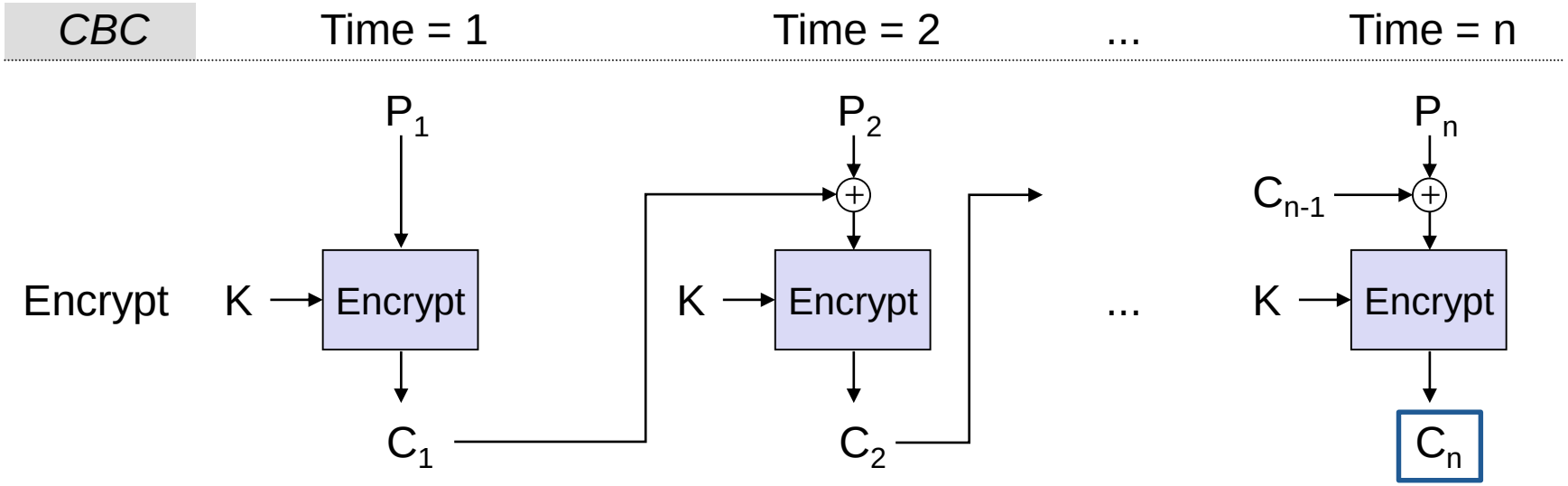
- HMAC = $h(k \oplus o_{pad} \,||\, h(k \oplus i_{pad} \,||\, x))$
  - Two nested hash function calls
  - But only one over potentially long message ⇨ efficient
- Key padded with public constant to block length
  - Two pads: results in different inner and outer keys
- HMAC is secure if hash function h is secure

CBC-MAC



- Construct MAC based on block ciphers
- Utilize Cipher Block Chaining (CBC) with zero IV
  - Encrypt an input message x_1, ... x_n with a secret k
  - Yields ciphertext blocks c_1, ..., c_n

CBC-MAC (2)

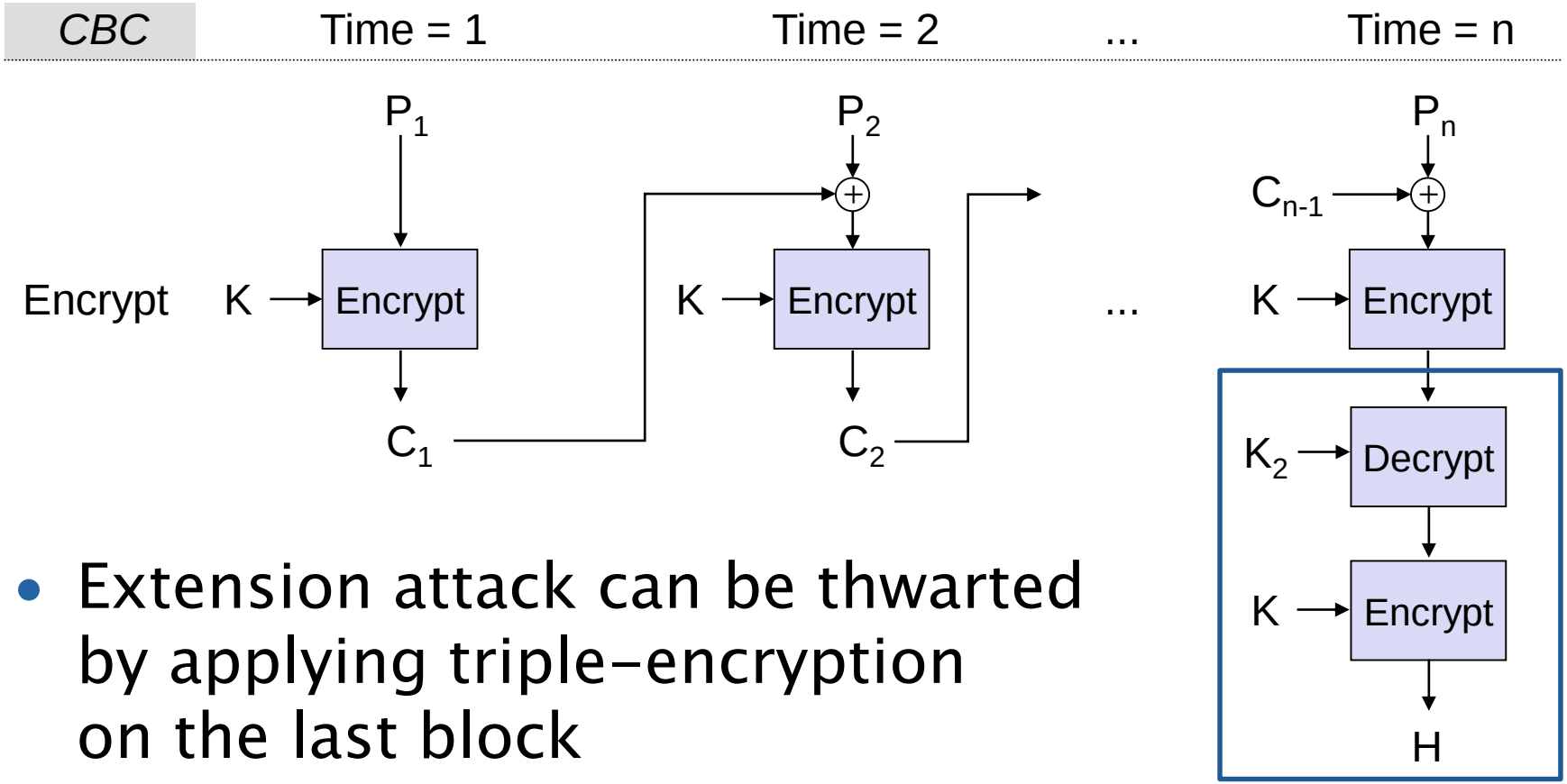


- Use last ciphertext block c_n as CBC-MAC
  - Each ciphertext block depends on all previous blocks
  - Previous blocks c_1, ..., c_{n-1} are not required
  - Verify CBC-MAC by encrypting input message (decryption is not used)

Security of CBC-MAC

- Without k, it‘s difficult to run a birthday attack
  - This allows for shorter MACs than hash values
  - Collisions may occur though: different plaintext blocks may lead to one identical ciphertext block
  - This allows to exchange the colliding message prefix without changing the CBC-MAC
- Message extension attack is possible
  - Given x with $H_x = h_k(x)$ and y with $H_y = h_k(y)$
  - Attacker crafts new message $x \,||\, H_x \oplus y$ with H_y
  - Identical MAC $H_y = h_k(y) = h_k(x \,||\, H_x \oplus y)$

Security of CBC-MAC (2)

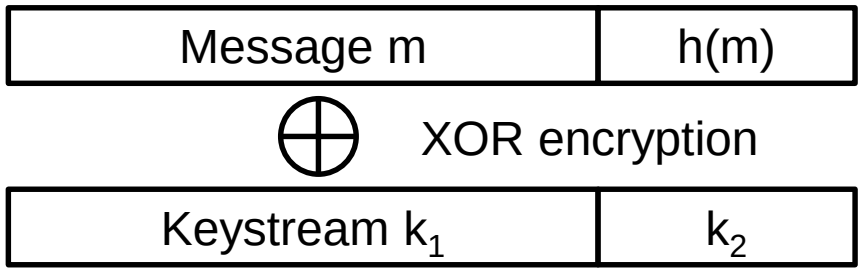


- Extension attack can be thwarted by applying triple-encryption on the last block
- Doubles key-length due to second key K_2

Confidentiality combined with Integrity

- So far we've seen how to achieve either one:
  - Confidentiality (with encryption)
  - ... and authenticity/integrity (with signature or MAC)
- How to achieve authenticated encryption?
- Idea: encrypt data and cryptographic hash value
  - $e_k(m \,||\, h(m))$, e.g. by using block cipher in CBC mode
  - Plain hash value would be insecure but here it is hidden by symmetric-key encryption
  - Security depends solely on encryption cipher: broken confidentiality ⇒ broken integrity

Confidentiality combined with Integrity (2)



- Insecure if cipher is vulnerable to known-plaintext attacks, e.g. stream ciphers
  - Given a known message m, recover keystream k_1
  - Compute h(m), recover k_2
  - Change to m‘, encrypt with k_1
  - Compute h(m‘), encrypt with k_2

Authenticated Encryption

- Idea: Encrypt data and MAC
- Problem: Combination of mechanisms may have undesired side effects
- Example: CBC encryption and CBC-MAC
  - $e_k(m \,||\, h_k(m))$ with identical k and zero IV
  - CBC-MAC equals last ciphertext block $c_n = e_k(c_{n-1} \oplus m_n)$
  - When encrypting c_n, this yields $e_k(c_n \oplus c_n) = e_k(0)$
  - Insecure: encrypted MAC lost relationship to message
- Solution: use separate, independent keys

Authenticated Encryption (2)

1. Encrypt-then-MAC (e.g. IPSec)
   - Encrypt data $c = e_{k_1}(m)$
   - MAC over ciphertext $H = h_{k_2}(c)$
2. MAC-then-encrypt (e.g. TLS 1.2)
   - MAC over plaintext $H = h_{k_2}(m)$
   - Encrypt data and MAC $c = e_{k_1}(m \,||\, H)$
3. Encrypt-and-MAC (e.g. SSH)
   - Encrypt data $c = e_{k_1}(m)$
   - MAC over plaintext $H = h_{k_2}(m)$

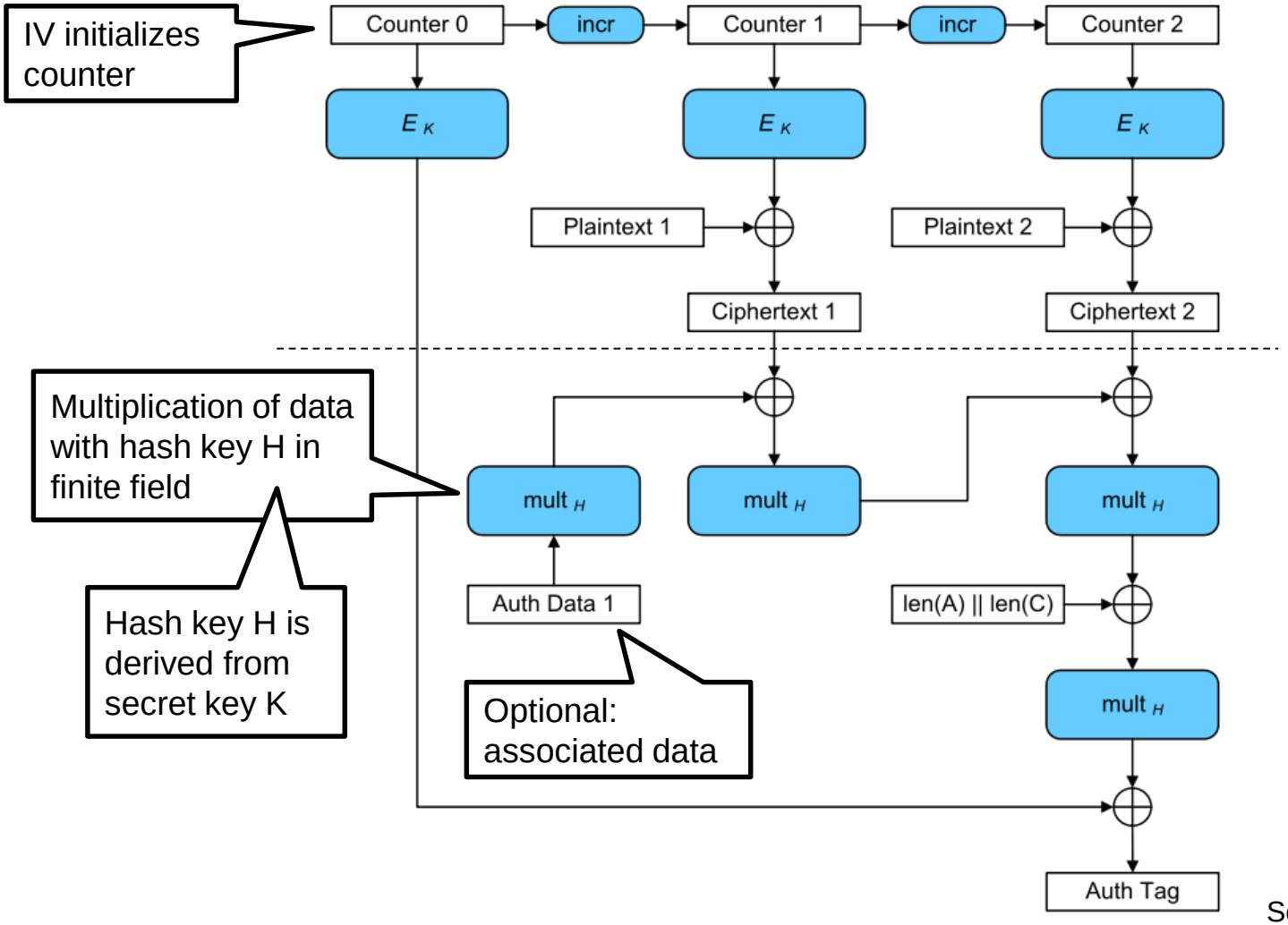
Authenticated Encryption (3)

- Each of the approaches
  - ... is used in practice
  - ... can be implemented securely
  - ... has slightly different properties with different consequences if implemented insecurely
- MAC-then-encrypt implementations prone to padding oracle attacks
- Modern block cipher modes offer authenticated encryption with associated data (AEAD)
  - Encryption and authentication in one operation

Galois/Counter Mode

- GCM: AEAD-type block cipher mode
  - Combines CTR mode for encryption with GHASH function for authentication
- Input:
  - Plaintext data to be encrypted
  - Secret key
  - Initialization vector (IV)
  - Optional: associated data which is authenticated only
- Output:
  - Ciphertext
  - Authentication tag (i.e. MAC)

Galois/Counter Mode (2)

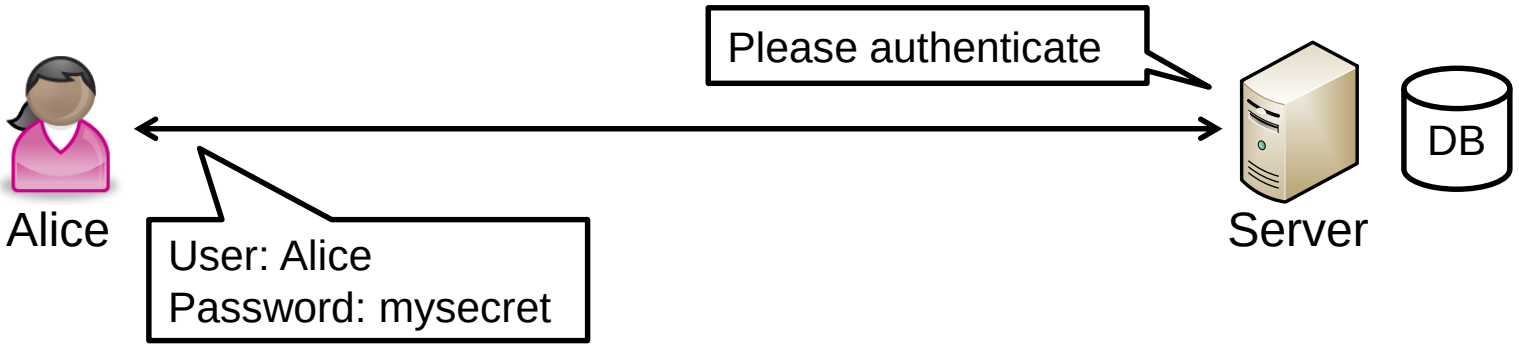


Encryption in CTR mode

Authentication with GHASH function

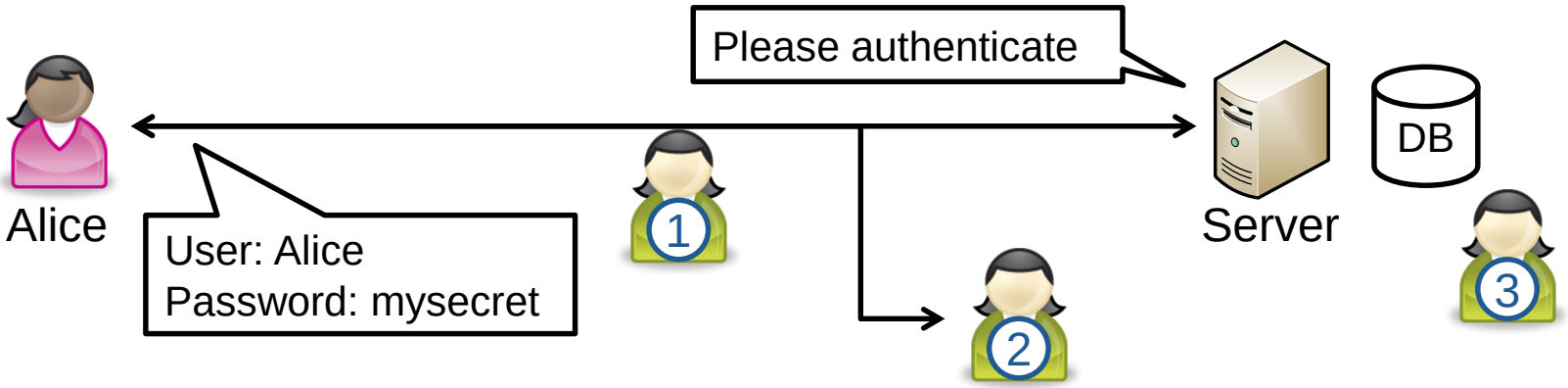
Source: Wikimedia Commons

Password Authentication



- We can authenticate users via:
  - Public-key cryptography
  - Symmetric-key cryptography
  - Both require possession of a key, which can be impractical in some scenarios
- Idea: authenticate users via password

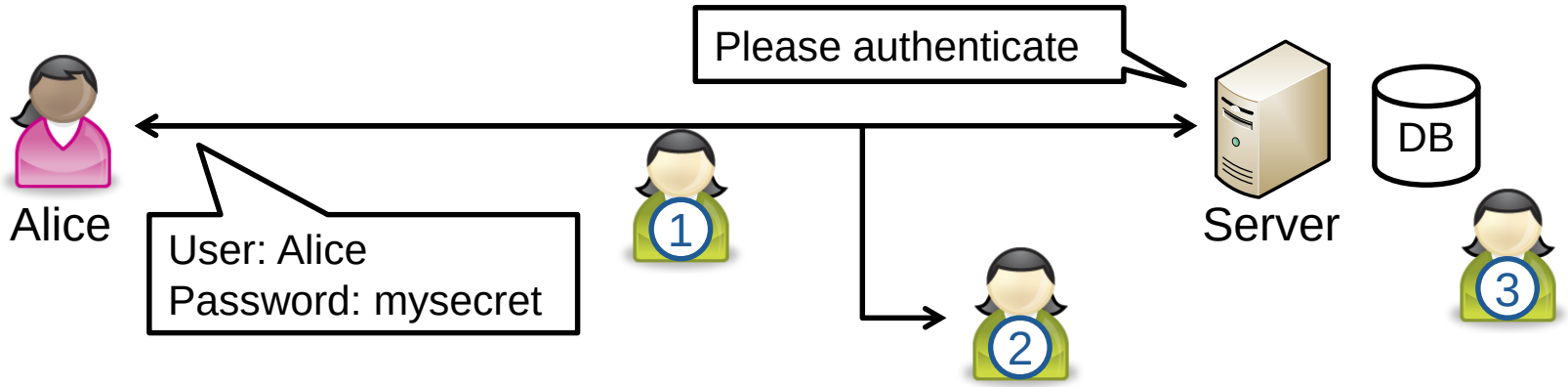
Password Authentication (2)



- Possible Attacks include:

1. Passive attacker eavesdropping the network
2. Active attacker impersonates as server
3. Attacker stealing password database

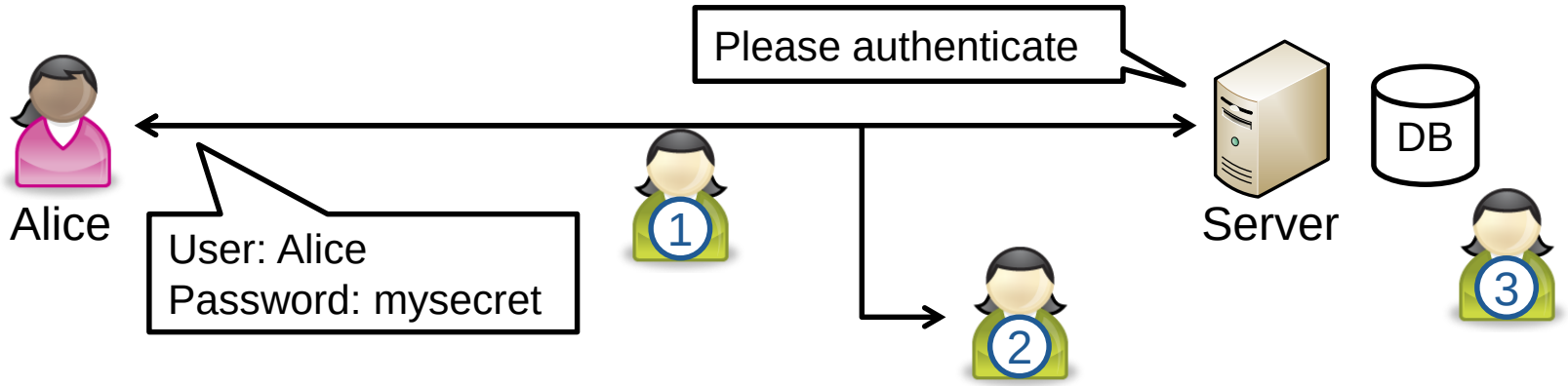
Password Authentication (3)



1. Protection against eavesdropping:

- Encrypt password. How?
  - Establish secret key with Diffie-Hellman key exchange
  - Or with asymmetric cryptography

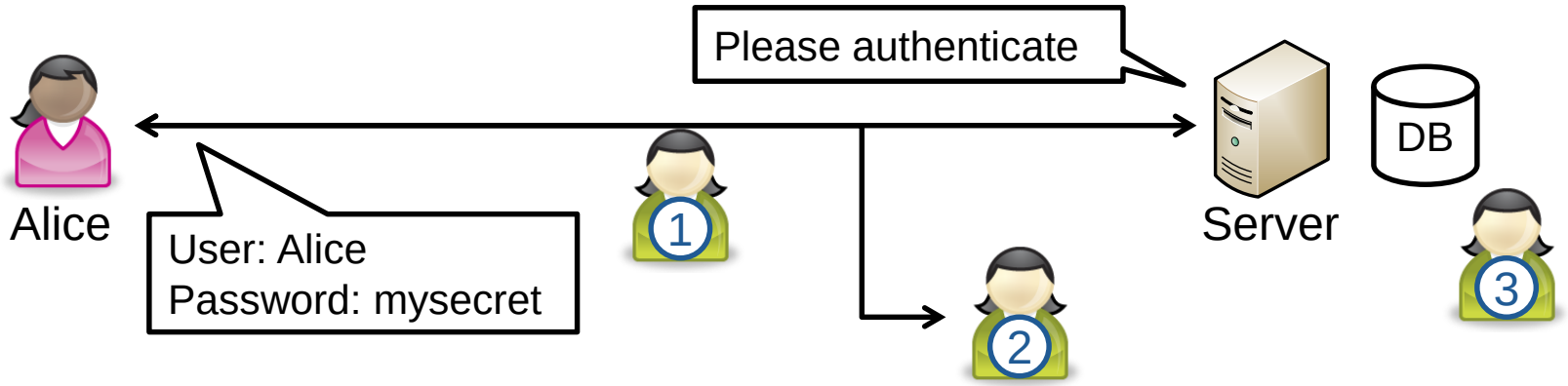
Password Authentication (4)



2. Protection against impersonation:

- Authenticate server. How?
  - Public-key cryptography
  - Retrieve public key securely from digital certificate (we will cover this topic later)

Password Authentication (5)



3. How to protect the password database?

- Idea: encrypt the passwords
  - Problem: how to protect the decryption key?
- Idea: apply one-way function on passwords

Password Hashing

- Store hash values of passwords in database
  - e.g. „Alice“ → h(p)
- Server asks Alice for password
  - Alice sends password p‘ (over encrypted connection)
  - Server computes h(p‘) and checks whether $h(p') \stackrel{?}{=} h(p)$
- Attacker might steal h(p) from database
  - But can‘t authenticate with h(p), because server checks $h(h(p)) \neq h(p)$
  - Attacker can‘t recover p due to one-way function

Password Hashing (2)

- Given the following SHA-256 hash value:

```
2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b
```

- Despite the pre-image resistance of SHA-256, we can easily „break“ the hash value
  - e.g. with Dictionary attack
- Measures to increase security:
  - Strong passwords
  - Salted password hashing
  - Iterated hashing

Salted Password Hashing

- Add an individual „salt“ value to each password
  - Random string s, which can be saved in cleartext
  - e.g. „Alice“, salt s → h(p||s)
- Salt prevents attacks with pre-computed tables
  - e.g. rainbow table attack or Google attack
- Individual salts per user slow down batch cracking against databases with multiple users
  - Without salt: attacker can compute h(w) for a word and compare it against all password hashes
  - With salt: compute h(w||s) for each s ∈ (s_1, …, s_i)

Iterated Hashing

- Problem: attacker can check many hash values efficiently with lots of computing power
- Iterated hashing: call hash function repeatedly
  - e.g. „Alice“, salt s, 5 iterations → h(h(h(h(h(p||s)))))
- Problem: attacker can increase efficiency with GPU computing or custom-hardware ASICs
- Memory-hard hashing: require lots of memory
- Password-based key derivation functions
  - e.g. PBKDF2, scrypt, Argon2