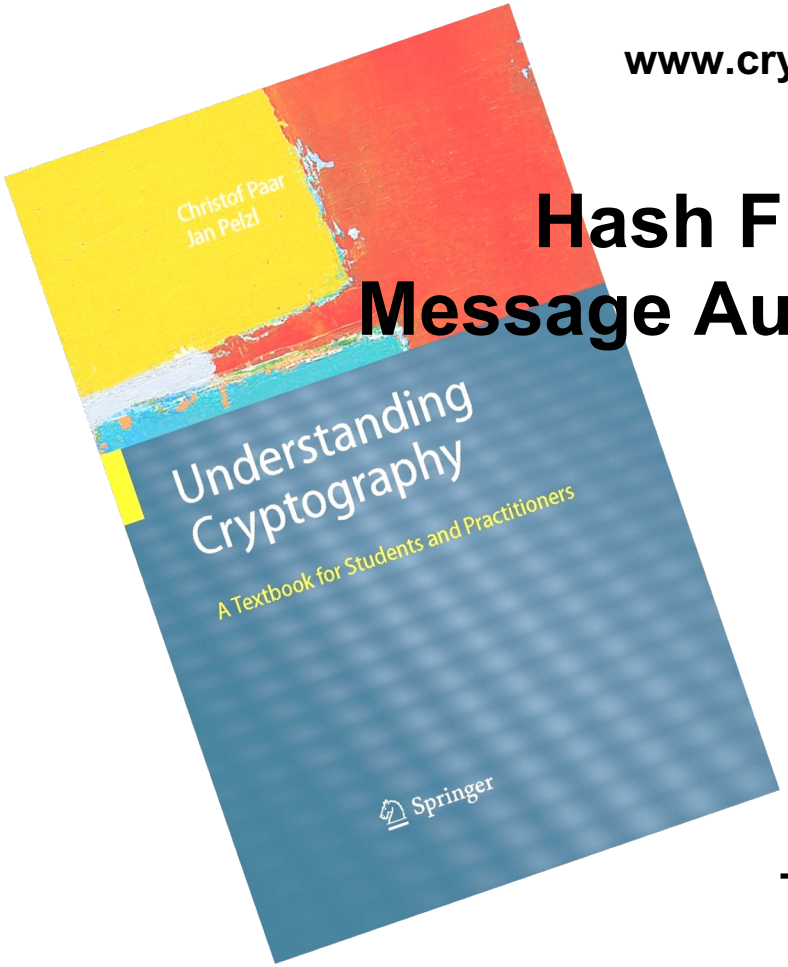
Understanding Cryptography – A Textbook for Students and Practitioners

by Christof Paar and Jan Pelzl

www.crypto-textbook.com



Hash Functions and Message Authentication Code

These slides are a modified version of the slides prepared by Stefan Heyse and Christof Paar and Jan Pelzl. Modified by Alwen Tiu (09/2022).

Some legal stuff (sorry): Terms of Use

- The slides can used free of charge. All copyrights for the slides remain with Christof Paar and Jan Pelzl.
- The title of the accompanying book “Understanding Cryptography” by Springer and the author’s names must remain on each slide.
- If the slides are modified, appropriate credits to the book authors and the book title must remain within the slides.
- It is not permitted to reproduce parts or all of the slides in printed form whatsoever without written consent by the authors.

Hash Functions

Content of this Chapter

- Why we need hash functions
- How does it work
- Security properties
- Algorithms
  - Hash functions from block ciphers
  - The Secure Hash Algorithm SHA-1

Content of this Chapter

- **Why we need hash functions**
- How does it work
- Security properties
- Algorithms
  - Hash functions from block ciphers
  - The Secure Hash Algorithm SHA-1

Motivation: message integrity

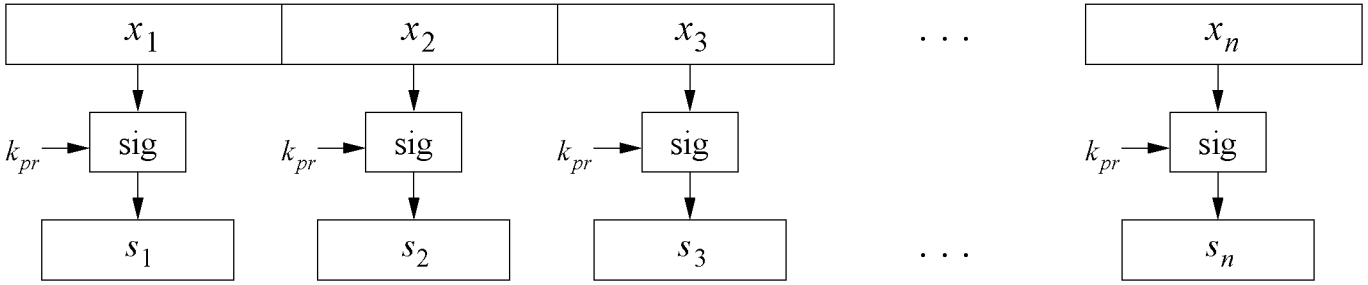
- Consider the problem of checking the integrity of a large file stored on the cloud.
  - Keep a local copy version of the file, and compare the local and the cloud version bit-by-bit: impractical, defeat the purpose of cloud storage.
  - Better solution: keeps a small *digest* of the file locally, and compute the digest of the cloud version and compare the digests.
- What are the requirements for the digests?
  - If two files have the same digest, they are very likely to be the same file.
  - It is infeasible, given a file and its digest, to construct a different file that has the same digest.

Motivation: computing digital signatures

- Digital signatures:
  - A method for 'signing' messages using asymmetric encryption (to be covered later in the course).
  - Use a pair of *public key* and *private key* (k_{pub}, k_{pr}).
  - Two functions: **signing** (requires private key k_{pr}) and **signature verification** (requires public key k_{pub}).
- Signing: $sig_{k_{pr}}(m)$ takes a message m of **fixed length** and produces a fixed length signature.
- Signature verification: $ver_{k_{pub}}(s, m)$ takes a signature *s* and a message *m*, returns true if and only if $s = sig_{k_{pr}}(m)$.
- Problem: how do we sign messages of **arbitrary length**?

Motivation: computing digital signatures

Naive signing of long messages generates a signature of same length.



Three Problems

- Computational overhead
- Message overhead
- Security limitations

Solution:

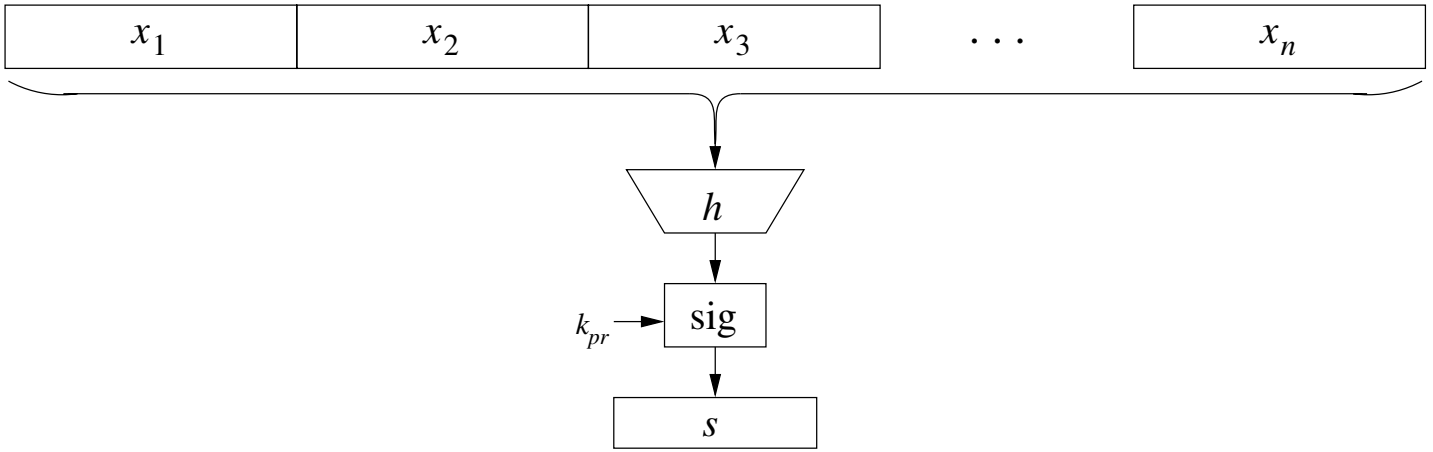
Instead of signing the whole message, sign only a digest (=hash)

Also secure, but much faster

Needed:

Hash Functions

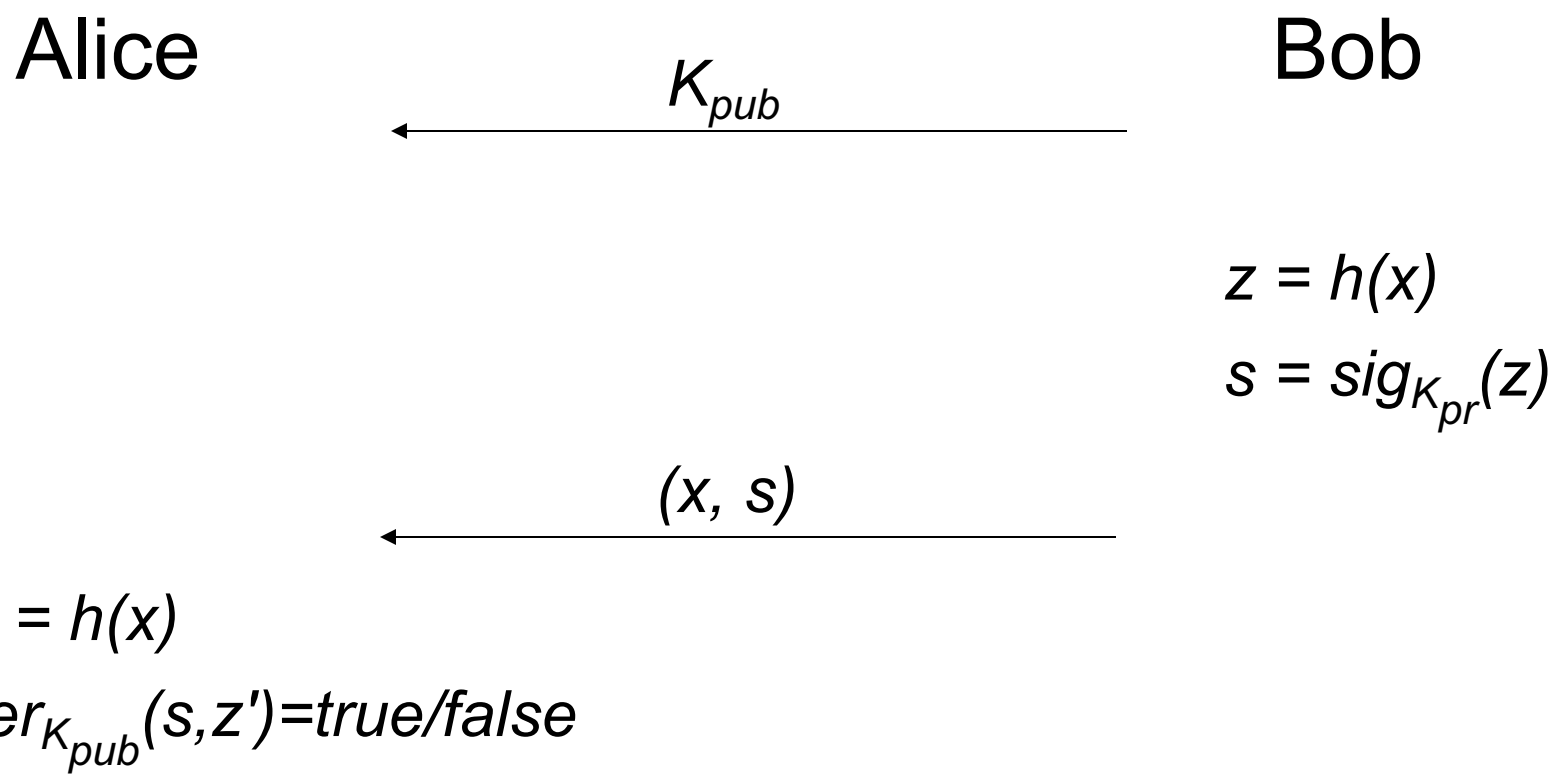
Digital Signature with a Hash Function



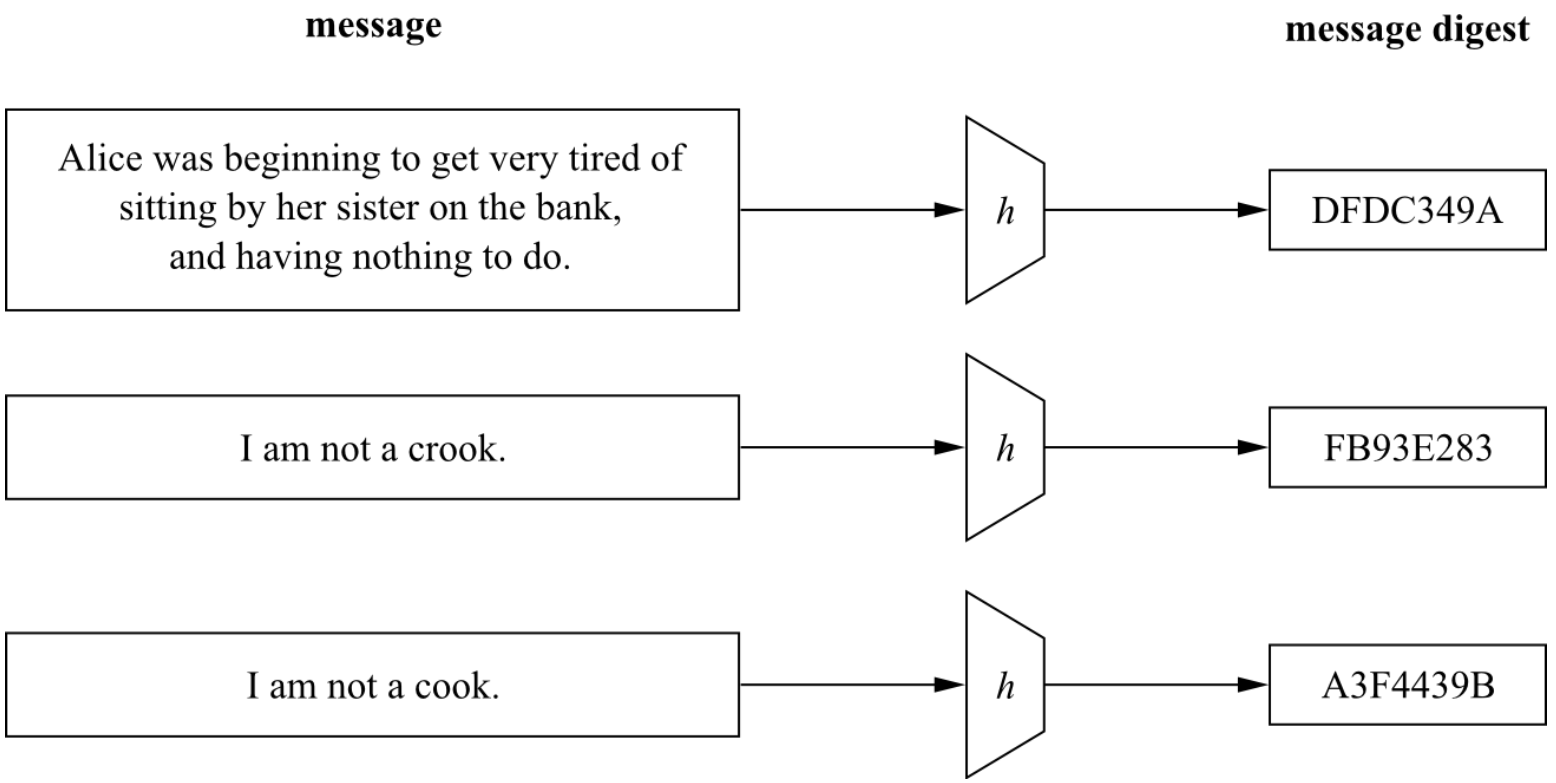
Notes:

- The hash function *h(x)* does not require a key.
- *h(x)* is public.
- Sign the hash of the message rather than the message itself.

- **Basic Protocol for Digital Signatures with a Hash Function:**



Principal input–output behavior of hash functions

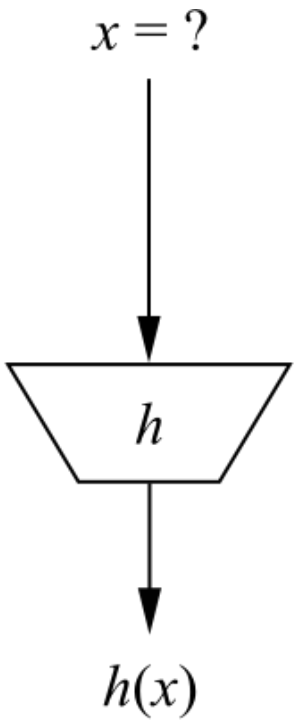


- Variable length input, fixed length output.
- Highly sensitive to changes in input: small changes in input results in very different hashes.

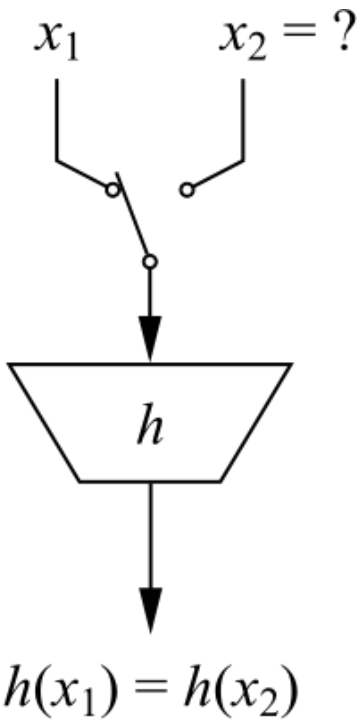
Content of this Chapter

- Why we need hash functions
- How does it work
- **Security properties**
- Algorithms
  - Hash functions from block ciphers
  - The Secure Hash Algorithm SHA-1

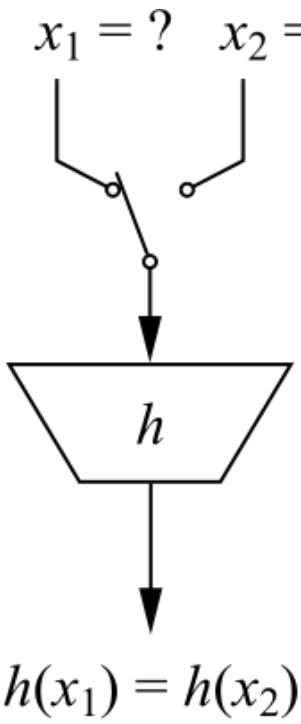
The three security properties of hash functions



preimage resistance



second preimage resistance



collision resistance

Hash Functions: Security Properties

Preimage resistance: For a given output z, it is computationally infeasible to find any input x such that $h(x) = z$, i.e., $h(x)$ is one-way.

Second preimage resistance: Given x_1, and thus $h(x_1)$, it is computationally infeasible to find any x_2 such that $x_1 \neq x_2$ and $h(x_1) = h(x_2)$.

Collision resistance: It is computationally infeasible to find any pairs $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$.

Hash Functions: Security

- How hard is it to find a collision with a probability of 0.5 ?
- Related Problem: How many people are needed such that two of them have the same birthday with a probability of 0.5 ?
  - Answer: 23.
- This is called the ***birthday paradox***: given a hash function with n-bit output (so there are 2^n possible hash values), to have a λ chance of collision, the number of hash values needed to be explored is given by:

$$t \approx \sqrt{2^{n+1} \ln\left(\frac{1}{1-\lambda}\right)} \approx p(\lambda) \cdot \sqrt{2^n} \quad \text{where} \quad p(\lambda) = \sqrt{2 \cdot \ln(\frac{1}{1-\lambda})}$$

See *Understanding Cryptography*, Section 11.2.3 for the exact derivation of this formula.

Hash Functions: Security

- For $\lambda = 0.5$, we have $p(\lambda) \approx 1.177$, and for $\lambda = 0.9$, we have $p(\lambda) \approx 2.146$.
- For a hash function with n bits output:
  - To have a 50% chance of collision, the number of hashes needed to be explored is approximately $1.177\sqrt{2^n}$.
  - To have a 90% chance of collision, the number of hashes needed to be explored is approximately $2.146\sqrt{2^n}$.

Hash Functions: Security

Table 11.1 Number of hash values needed for a collision for different hash function output lengths and for two different collision likelihoods

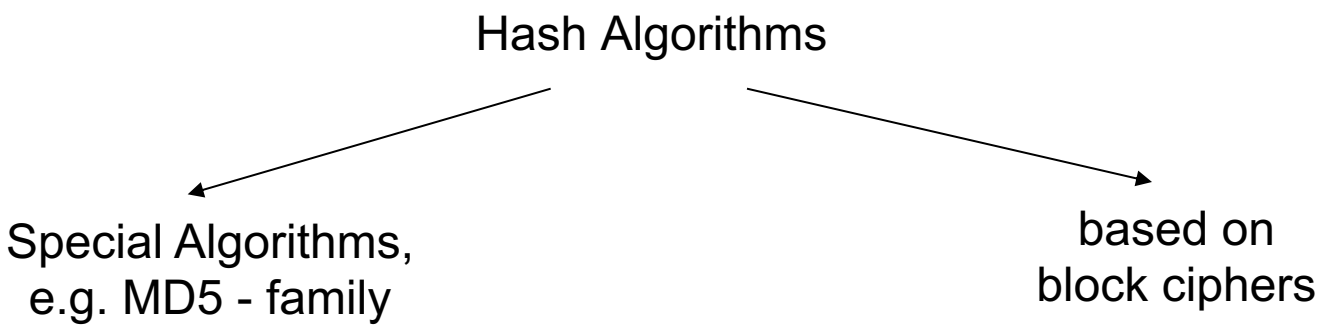
	Hash output length				
λ	128 bit	160 bit	256 bit	384 bit	512 bit
0.5	2^{65}	2^{81}	2^{129}	2^{193}	2^{257}
0.9	2^{67}	2^{82}	2^{130}	2^{194}	2^{258}

- For a hash function with 160 bit output, a brute force attack would need to generate approximately 2^{81} hash values to have a 50% chance find a collision.
- A hash function with 160 bit output may no longer be secure against collision.
  - E.g., if one takes all the hashes computed by cryptocurrency miners (which rely on computing as many hash values as fast as possible), the cumulative hash generated may exceed 2^{93} hashes per year.

Content of this Chapter

- Why we need hash functions
- How does it work
- Security properties
- **Algorithms**
  - Hash functions from block ciphers
  - The Secure Hash Algorithm SHA-1

Hash Functions: Algorithms



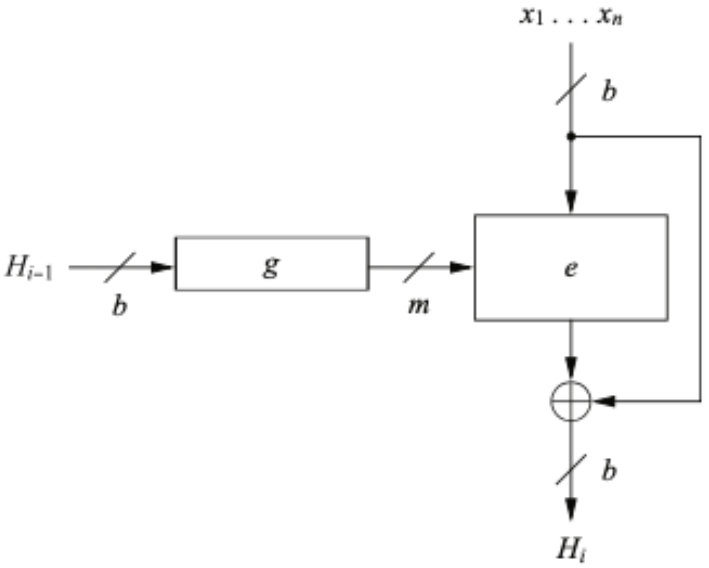
- **MD5 –** family
  - **SHA-1**: output - 160 Bit; input - 512 bit chunks of message *x;* operations - bitwise AND, OR, XOR, complement and cyclic shifts.
  - Variants of SHA-1: **SHA-256**, **SHA-384**, **SHA-512**, with output of length 256, 384 and 512, respectively.
  - **RIPEMD 160:** output - 160 Bit; input - 512 bit chunks of message *x*; operations – like in SHA-1, but two in parallel and combinations of them after each round.

Hash Functions: Algorithms

Algorithm		Output [bit]	Input [bit]	No. of rounds	Collisions found
MD5		128	512	64	yes
SHA-1		160	512	80	yes
SHA-2	**SHA-224**	224	512	64	no
	SHA-256	256	512	64	no
	SHA-384	384	1024	80	no
	SHA-512	512	1024	80	no

Hash functions from block ciphers

- Using a construction by Matyas, Meyer and Oseas (MMO):
  - Assume block cipher e with key size m and block size b.
  - The input blocks $(x_1, .., x_n)$ are encrypted and chained.
  - The first block is encrypted using a key generated from a fixed H_0.
  - The output from each block (H_i) is used to create the encryption key for the next block.



$$H_i = e_{g(H_i-1)}(x_i) \oplus x_i$$

- A function g is used to map a b-bit input to m-bit output.
- The hash value is the output for the final block (H_n).

MMO hash function: example

- If the block cipher AES-128 is used, then the block size is 128 bits and the key size 128 bits.
- The output of the hash function is 128 bits (block size of AES).
- We could let g be the identity function (i.e., $g(x) = x$) and let H_0 be a block of 0s. Then this instance of the MMO hash function can be expressed as:

$$H_0 = 00 \ldots 000$$
$$H_i = e_{H_{i-1}}(x_i) \oplus x_i$$

- For n-block input $s = x_1 .. x_n$, the hash value would then be H_n.

MMO hash function: security

- One issue with MMO is that the size of the hash output is the same as the block size of the block cipher.
- For AES, this is 128 bits, which does not provide enough security against birthday attack for collision.
- This can be remedied by combining two or more block encryptions.
- For AES-256 (key size 256 bits), this can be done by producing two encrypted blocks for each input block, yielding a hash function with 256 bits output.
    - See *Understanding Cryptography*, Chapter 11.3.2, for an example.

Padding

- If the input to a hash function is not a multiple of block size, some paddings need to be added.
- How padding is done can be critical to the security of the hash function.
- Generally, to prevent easy attack on second preimage or collision.
- An (insecure) example: pad input with 0s until its length a multiple of block size. Then collision attack (and second preimage attack) become trivial:

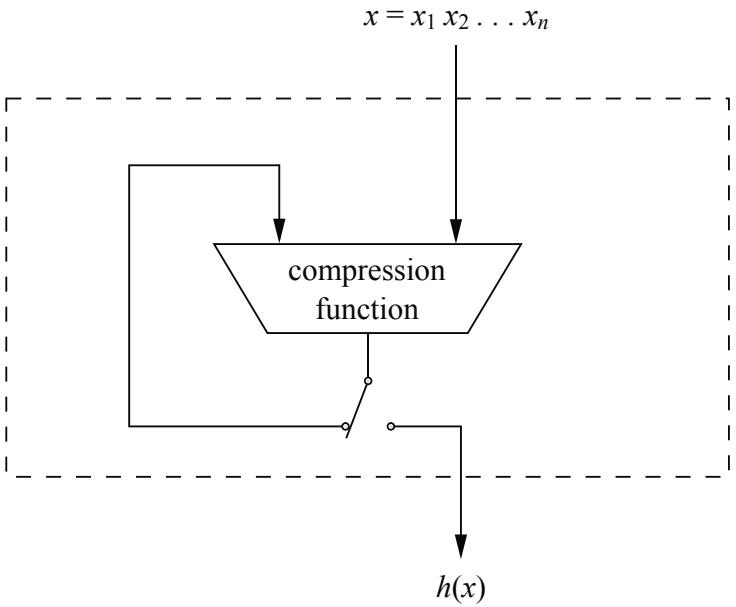
  X = 111 --[padding]--> 11100..00 --[hash]--> Y

  X' = 1110 --[padding]--> 11100..00 --[hash]--> Y

- Generally, to avoid ambiguity introduced by padding, the length of the input needs to be encoded in the padding scheme. (We'll see an example with SHA-1 padding scheme).

SHA-1

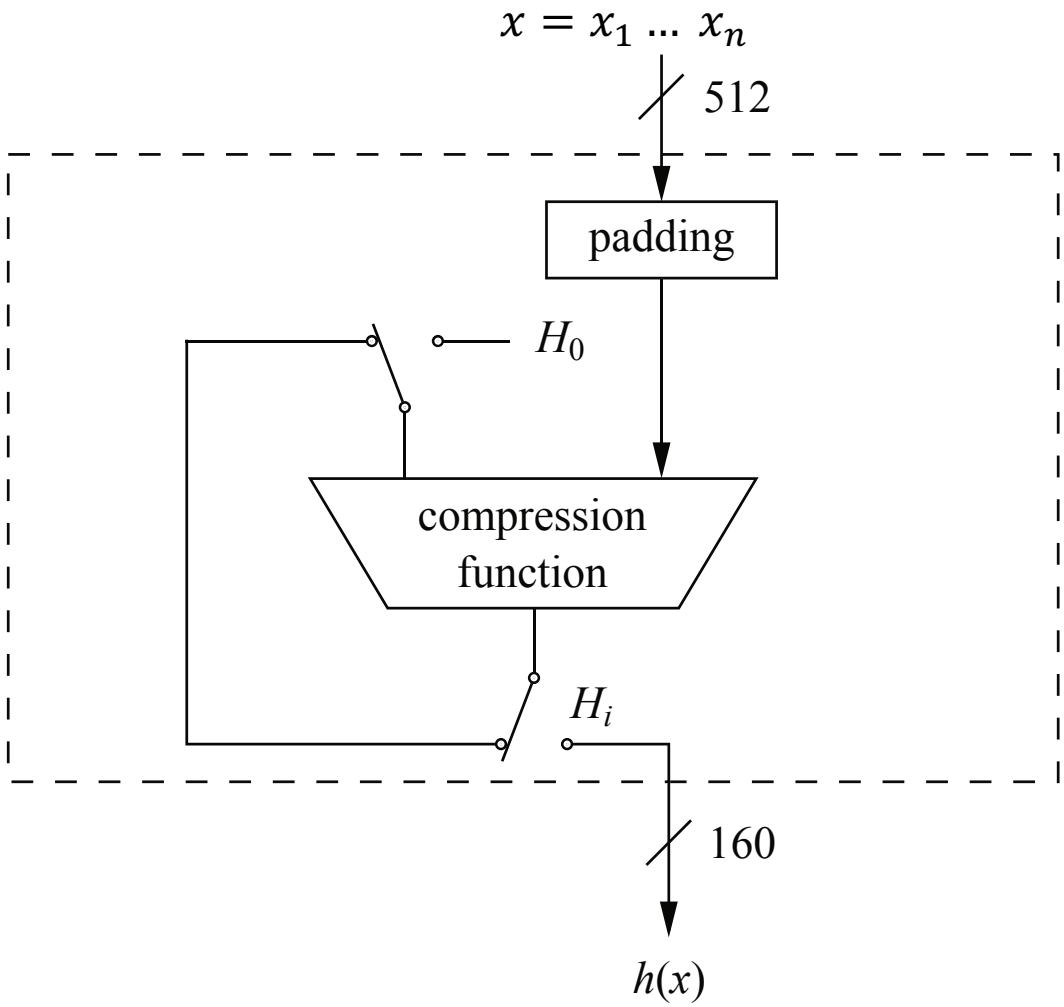
- Part of the MD-4 family.
- Based on a Merkle-Dåmgard construction.
- 160-bit output from a message of maximum length 2^{64} bit.
- Widely used (even tough some weaknesses are known)
- First collision (for the full SHA-1) found in 2017 (by researchers at CWI Amsterdam and Google), but there‘s a long history of theoretical weakness.
  - See: https://shattered.io



Merkle-Dåmgard construction

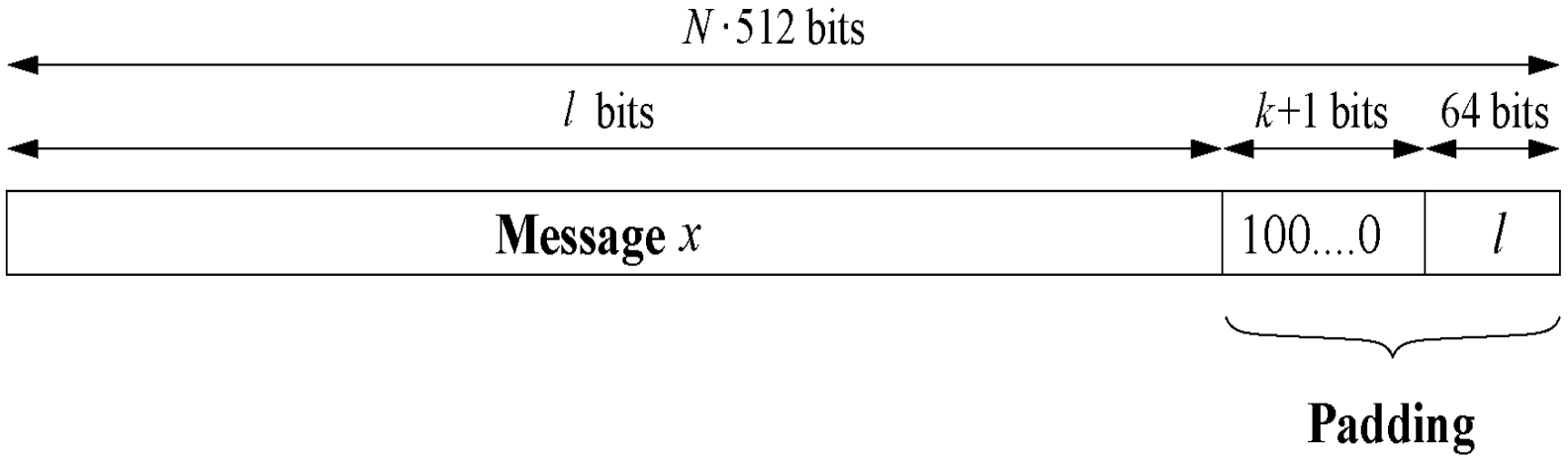
SHA-1 High Level Diagram

- SHA-1 uses a Merkle-Dåmgard construction that iterates a compression function.
- Compression Function consists of 80 rounds which are divided into four stages of 20 rounds each



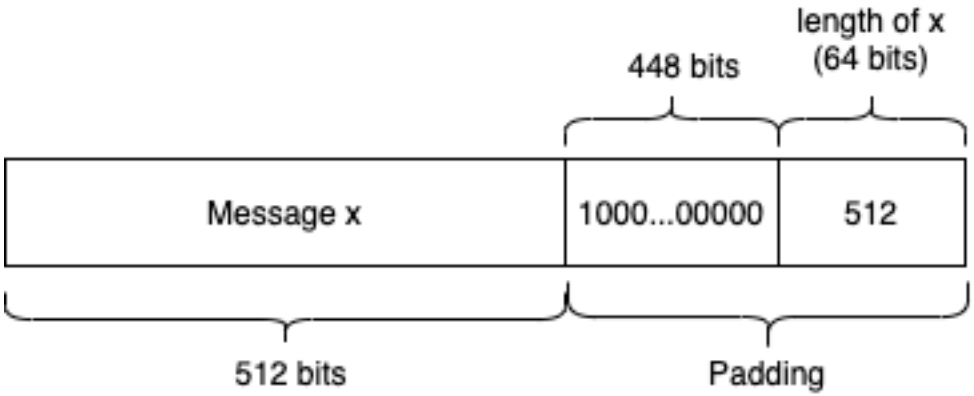
SHA-1: Padding

- Message x has to be padded to fit a size of a multiple of 512 bit.
- The padding adds the length l of the message (encoded as a 64-bit integer), and some $k + 1$ filler bits (with pattern 1000...00).
- $k \equiv 512 - 64 - 1 - l = 448 - (l + 1) \, mod \, 512.$



SHA-1: Padding

- Note that the information about the length of the original message x needs to be included in the input.
- This means that if the original message x is 512 bit long, the final padded input will be twice as long.



SHA-1: compression function

- The compression function of SHA-1 processes each input block x_i (512 bits) in four stages.
- Each stage takes 160 bits input and produces 160 bits output.
- Each stage consists of 20 rounds.

SHA-1 uses:

- A message schedule which computes a 32-bit word W0,W1,...,W79 for each of the 80 rounds
- Five working registers of size of 32 bits A,B,C,D,E
- A hash value H_i consisting of five 32-bit words $H_i^{(0)}$, $H_i^{(1)}$, $H_i^{(2)}$, $H_i^{(3)}$, $H_i^{(4)}$
- In the beginning, the hash value holds the initial value H_0, which is replaced by a new hash value after the processing of each single message block.
- The final hash value H_n is equal to the output h(x) of SHA-1.

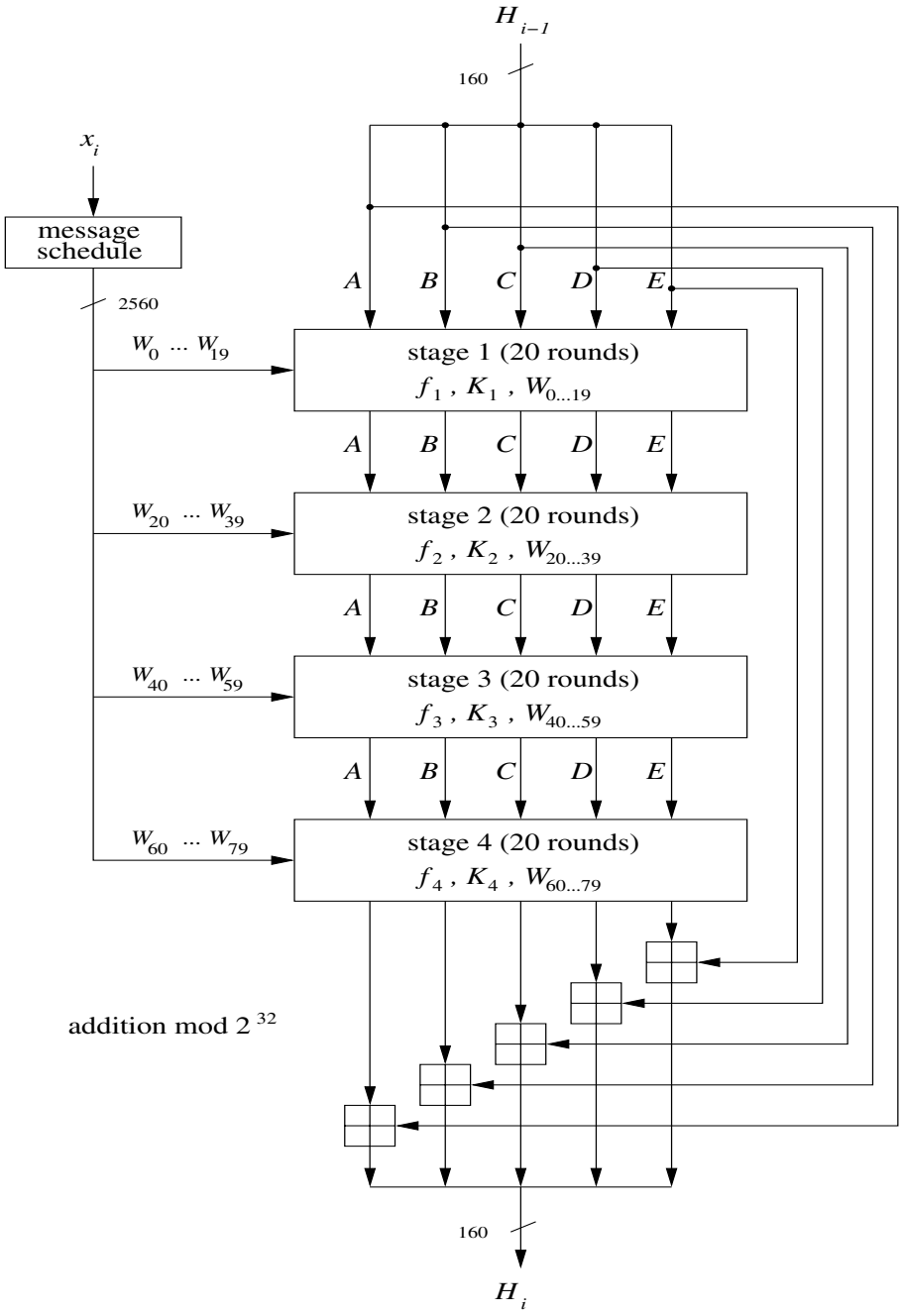
SHA-1: All four stages

Message schedule:

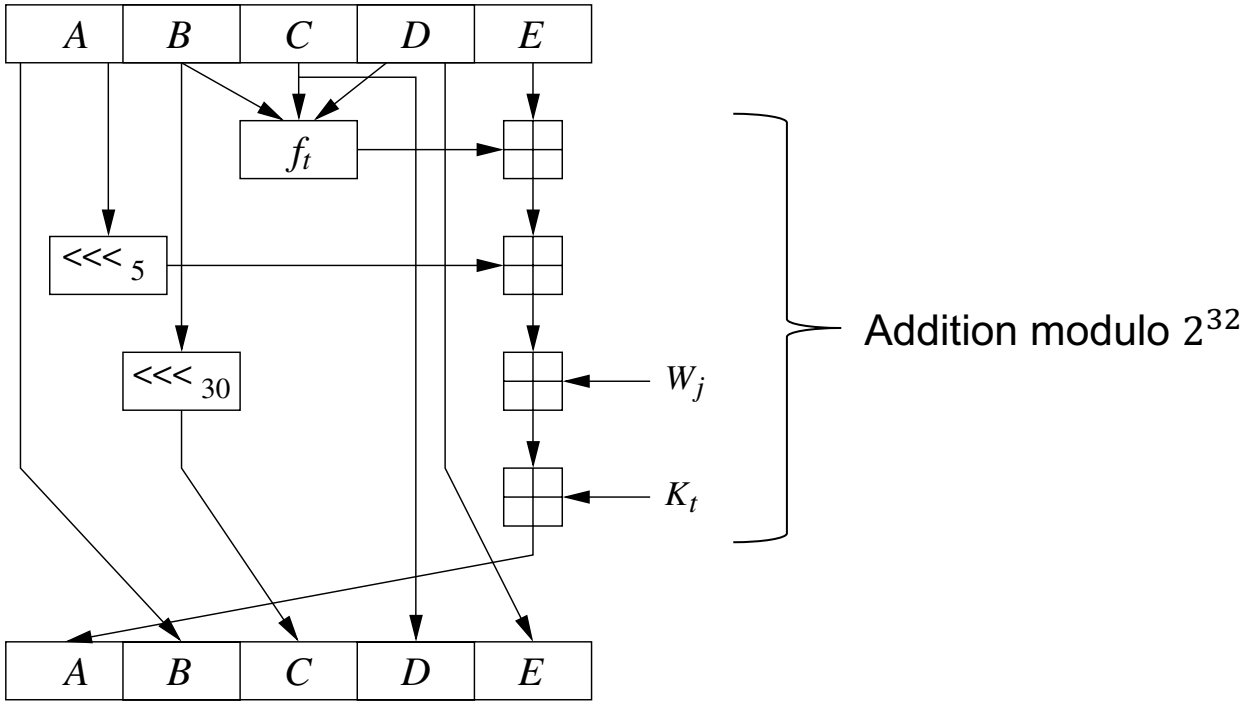
$$W_j = \begin{cases} x_i^{(j)} & 0 \le j \le 15 \\ (W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3}) \lll 1 & 16 \le j \le 79, \end{cases}$$

where $X_{\lll n}$ indicates a circular left shift of word X by n bit positions, and the input message block x_i is split into 16 32-bit blocks:

$$x_i = (x_i^{(0)}\ x_i^{(1)} \cdots x_i^{(15)})$$



SHA-1: Internals of a Round



Stage t	Round j	Constant K_t	Function f_t
1	0...19	$K_1 = \texttt{5A827999}$	$f_1(B,C,D) = (B \wedge C) \vee (\bar{B} \wedge D)$
2	20...39	$K_2 = \texttt{6ED9EBA1}$	$f_2(B,C,D) = B \oplus C \oplus D$
3	40...59	$K_3 = \texttt{8F1BBCDC}$	$f_3(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
4	60...79	$K_4 = \texttt{CA62C1D6}$	$f_4(B,C,D) = B \oplus C \oplus D$

Lessons Learned: **Hash Functions**

- Hash functions are keyless. The two most important applications of hash functions are their use in digital signatures and in message authentication codes such as HMAC.
- The three security requirements for hash functions are one-wayness, second preimage resistance and collision resistance.
- Hash functions should have at least 160-bit output length in order to withstand collision attacks; 256 bit or more is desirable for long-term security.
- MD5, which was widely used, is insecure. Serious security weaknesses have been found in SHA-1, and the hash function should be phased out. The SHA-2 algorithms all appear to be secure.
- The SHA-3 algorithm is standardized in 2015, after a competition that ran for a number of years. It uses a different construction from the Merkle-Damgard construction.

Further Information: **Hash Functions**

- Overview over many Hash Functions with Specifications:
  - http://ehash.iaik.tugraz.at/wiki/The_Hash_Function_Zoo
- Birthday Paradox: Wikipedia has a nice explanation
  - http://en.wikipedia.org/wiki/Birthday_problem
- SHA Standards
  - SHA1+2: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf
  - SHA3 Overview: http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo
- CrypTool is a learning program which also includes hash functions:
  - http://www.cryptool.org/
- Statistics on the total hash rate for the bitcoin network:
  - https://www.blockchain.com/en/charts/hash-rate

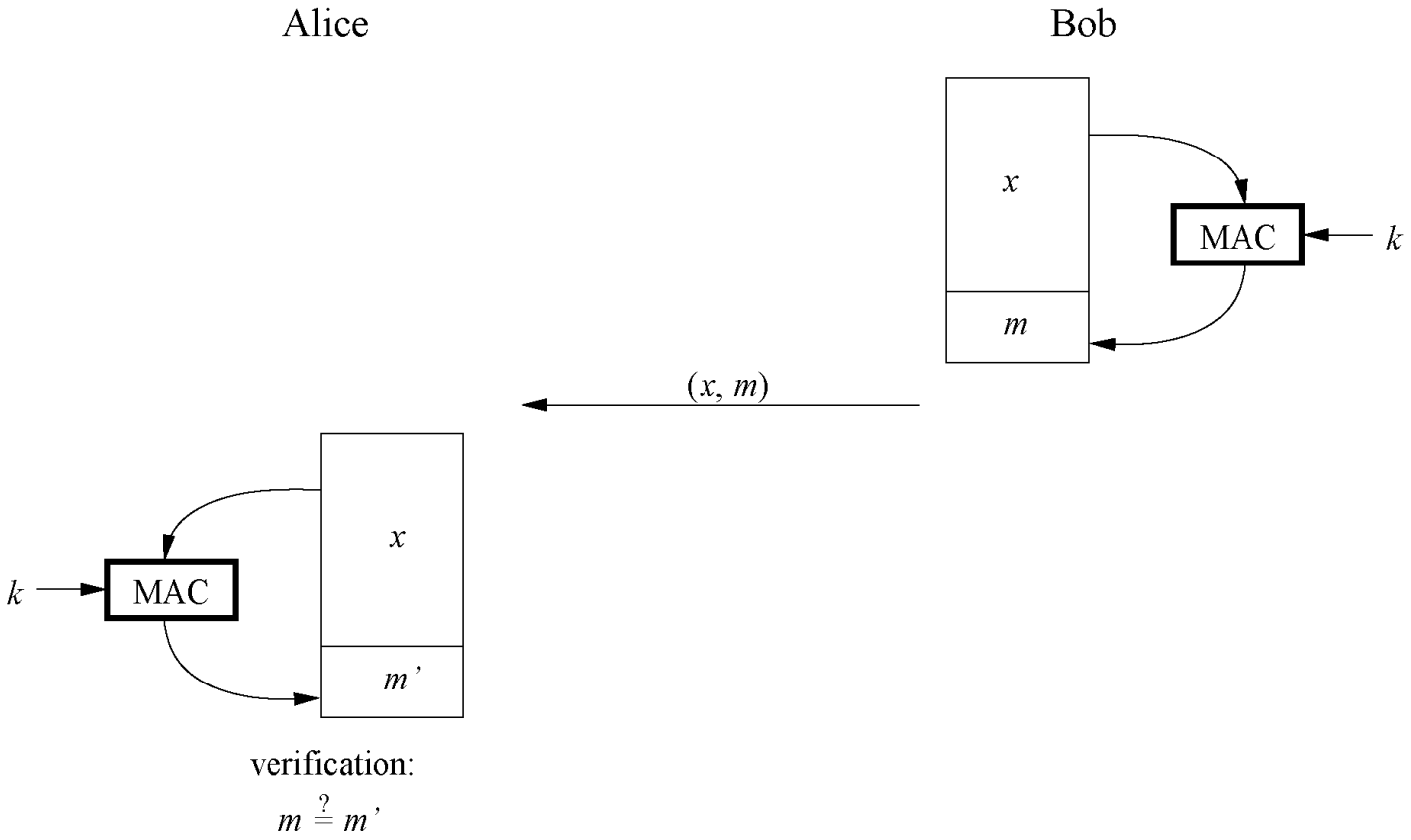
Message Authentication Code

Content of this Chapter

- The principle behind MACs
- The security properties that can be achieved with MACs
- How MACs can be realized with hash functions and with block ciphers

Principle of Message Authentication Codes

- Similar to digital signatures, MACs append an authentication tag to a message
- MACs use a symmetric key *k* for generation and verification
- Computation of a MAC: $m = \text{MAC}_k(x)$



Properties of Message Authentication Codes

1. **Cryptographic checksum**
   A MAC generates a cryptographically secure authentication tag for a given message.
2. **Symmetric**
   MACs are based on secret symmetric keys. The signing and verifying parties must share a secret key.
3. **Arbitrary message size**
   MACs accept messages of arbitrary length.
4. **Fixed output length**
   MACs generate fixed-size authentication tags.
5. **Message integrity**
   MACs provide message integrity: Any manipulations of a message during transit will be detected by the receiver.
6. **Message authentication**
   The receiving party is assured of the origin of the message.
7. **No nonrepudiation**
   Since MACs are based on symmetric principles, they do not provide nonrepudiation.

MACs from Hash Functions

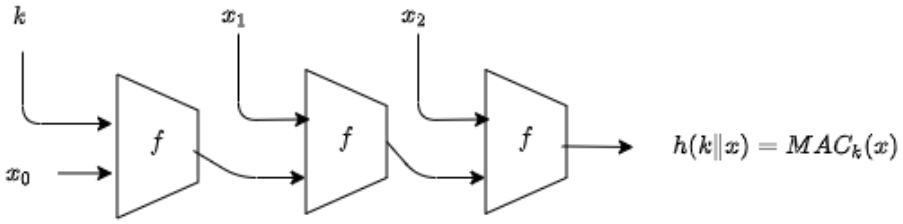
- MAC is realized with cryptographic hash functions (e.g., SHA-1)
- HMAC is such a MAC built from hash functions
- Basic idea: Key is hashed together with the message
- Two possible (naïve) constructions (where || means message concatenation):
  - secret prefix MAC: $m = MAC_k(x) = h(k \parallel x)$
  - secret suffix MAC: $m = MAC_k(x) = h(x \parallel k)$
- Attacks:
  - secret prefix MAC: Given x, construct x' such that $h(k \parallel x \parallel x')$ can be computed from $h(k \parallel x)$ *without knowing the* secret key k.
  - secret suffix MAC: find collision x and y such that $h(x) = h(y)$, then $m = h(x \parallel k) = h(y \parallel k)$.
- Idea: Combine secret prefix and suffix: HMAC.

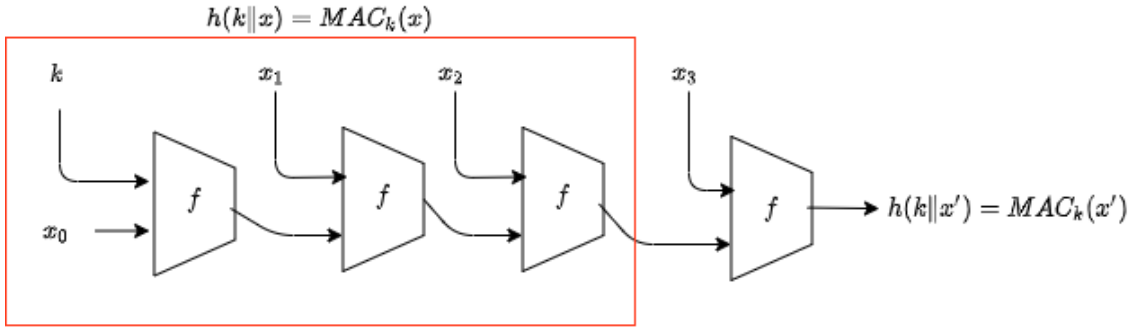
Secret prefix attack

- Consider the naïve MAC construction using secret prefix:

$$MAC_k(x) = h(k \parallel x)$$

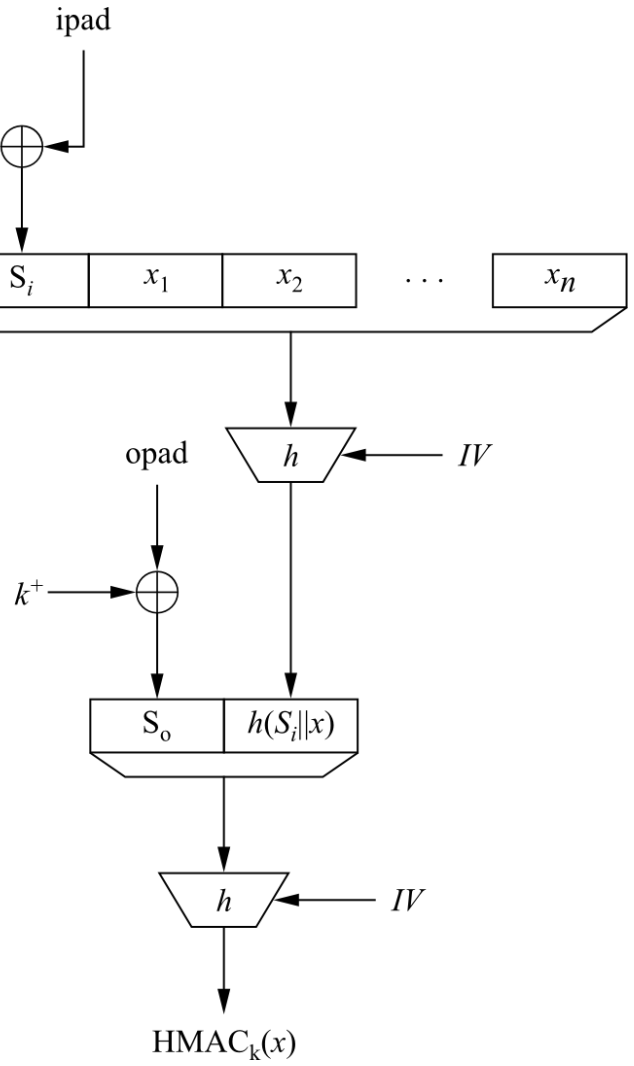
- An attack is possible if the hash function h uses the Merkle-Damgard construction.
- Suppose $x = x_1 x_2$ (a two-block message). Let f be the compression function that h is built from, and x_0 be the initial value for the hash function h. Then the attacker can compute the MAC for $x' = x_1 x_2 x_3$ for any arbitrary block x_3 by computing directly the compression function f using $MAC_k(x)$ and x_3.





HMAC

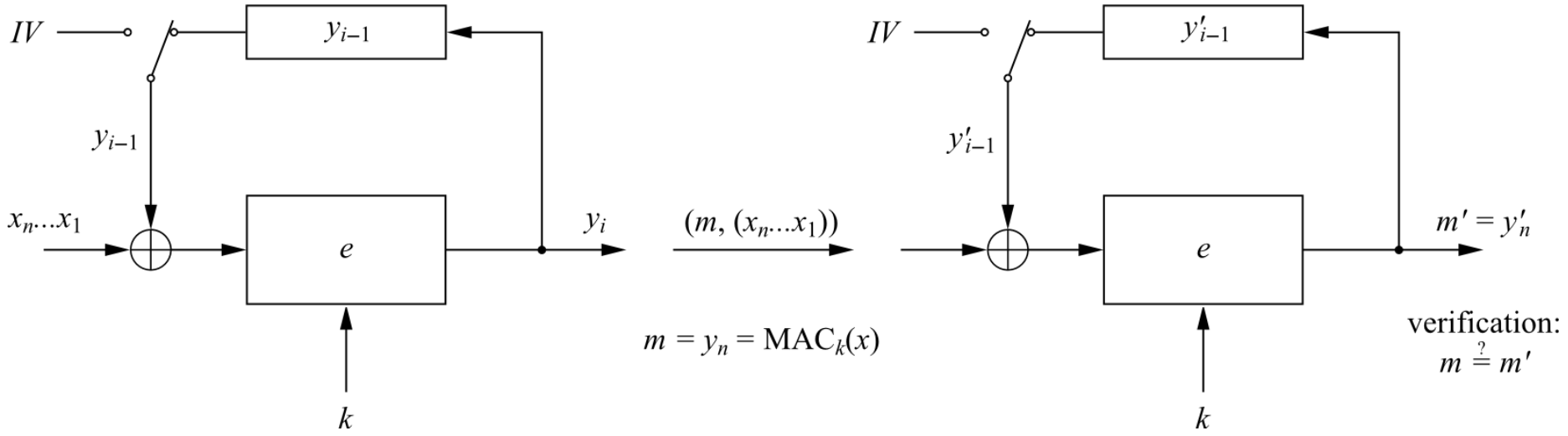
- Proposed by Mihir Bellare, Ran Canetti and Hugo Krawczyk in 1996
- Scheme consists of an inner and outer hash
  - k^+ is expanded key k
  - expanded key k^+ is XORed with the inner pad
  - ipad = 00110110,00110110, . . .,00110110
  - opad = 01011100,01011100, . . .,01011100
  - $\text{HMAC}_k(x) = h[(k^+ \oplus \text{opad})||h[(k^+ \oplus \text{ipad})||x]]$
- HMAC is provable secure which means (informally speaking):
  If HMAC is broken then a collision attack can be constructed for the underlying hash function.
- Note that the reverse does not necessarily hold; collision attack in h does not necessarily mean HMAC is insecure.



MACs from Block Ciphers

- MAC constructed from block ciphers (e.g. AES)
- Popular: Use AES in CBC mode

- CBC-MAC:



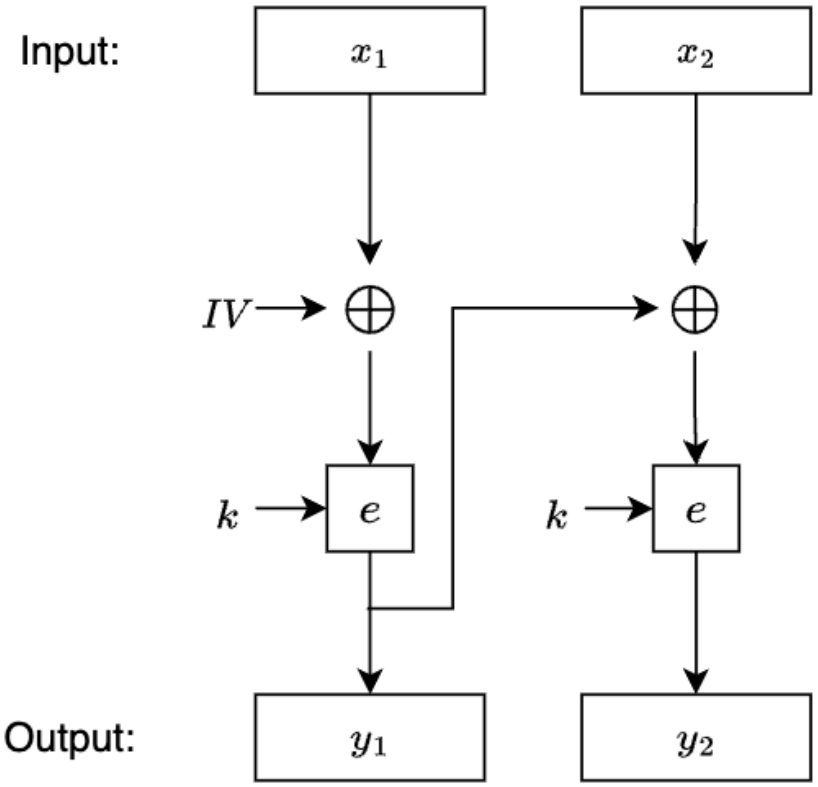
CBC-MAC

- MAC Generation
  - Divide the message x into blocks x_i
  - Compute first iteration $y_1 = e_k(x_1 \oplus IV)$
  - Compute $y_i = e_k(x_i \oplus y_{i-1})$ for the next blocks
  - Final block is the MAC value: $m = \text{MAC}_k(x) = y_n$

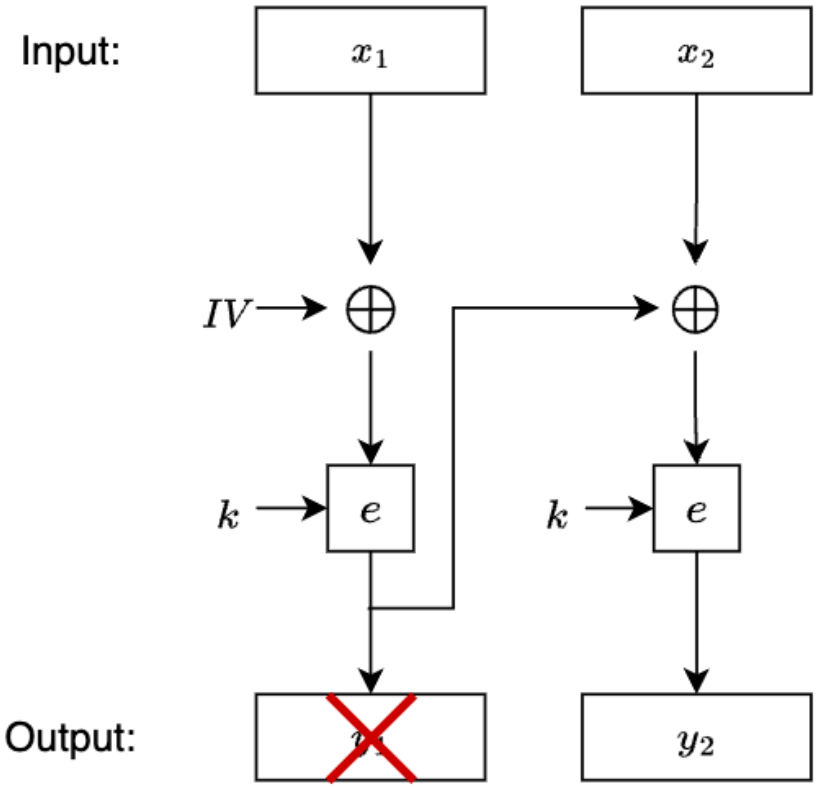
- MAC Verification
  - Repeat MAC computation (m')
  - Compare results:In case $m' = m$, the message is verified as correct
  - In case $m' \neq m$, the message and/or the MAC value m have been altered during transmission

CBC-MAC: an example

CBC encrypt:



CBC-MAC:



Discard

$MAC_k(x_1x_2) = y_2$

Attack on CBC-MAC

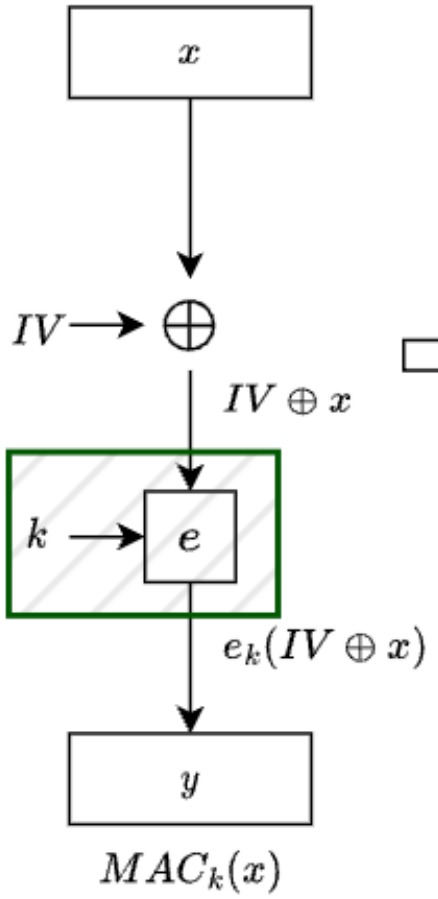
- Suppose the attacker knows a pair of one-block message and its MAC (x, m) where $m = MAC_k(x)$.
- The attacker can construct a MAC for a two-block message as follows: Let $x' = x \| (x \oplus m \oplus IV)$ where || denotes message concatenation.
- Then m is also the MAC of x'.
- Proof: let $x_1 = x$ and $x_2 = x \oplus m \oplus IV$. So $x = x_1 x_2$

$$MAC_k(x') = e_k(x_2 \oplus m) = e_k(x \oplus m \oplus IV \oplus m) = e_k(x \oplus IV) = m.$$

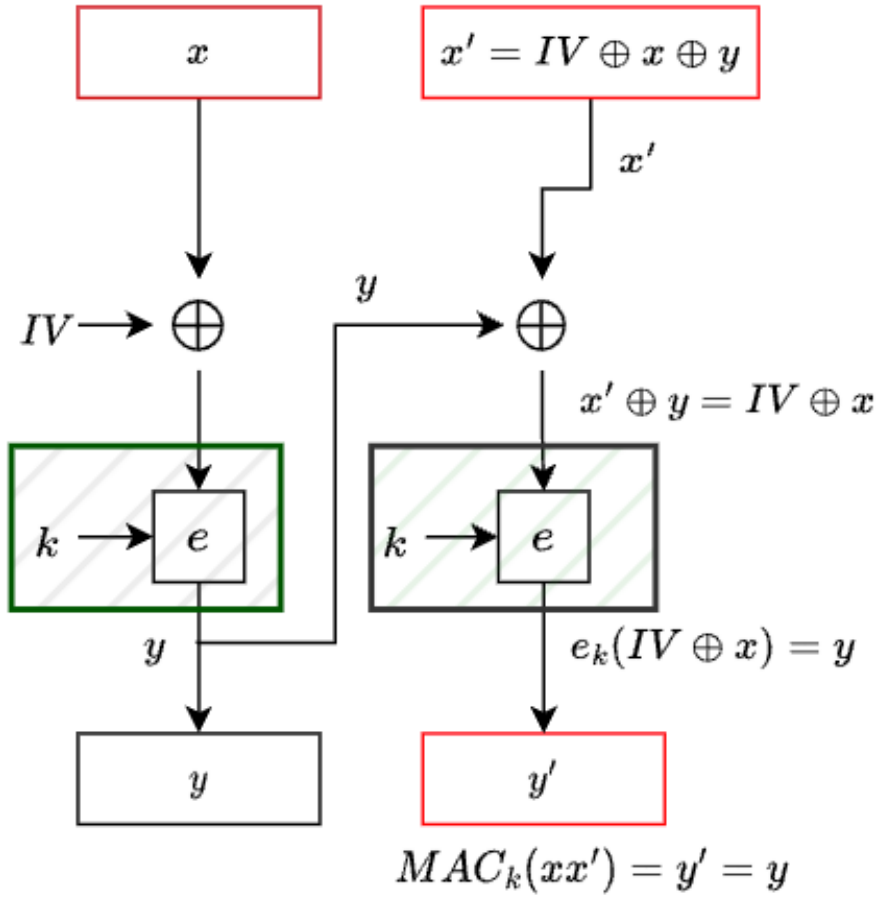
- To protect against this attack, the last cipher block in the MAC computation needs to be protected with additional (derived) keys.
- This gives rise to a variant called CMAC algorithm.
  - See https://tools.ietf.org/html/rfc4493

Attack on CBC-MAC

Attacker obtains x and $MAC_k(x)$



Attacker constructs xx' and computes $MAC_k(xx')$ without knowing k.



The shaded parts are inaccessible to the attacker.

Lessons Learned

- MACs provide two security services, *message integrity and message authentication,* using symmetric techniques. MACs are widely used in protocols.
- Both of these services also provided by digital signatures, but MACs are much faster as they are based on symmetric algorithms.
- MACs do not provide nonrepudiation.
- In practice, MACs are either based on block ciphers or on hash functions.
- HMAC is a popular and very secure MAC, used in many practical protocols such as TLS.