Hash Functions

Gerardo Pelosi

Department of Electronics, Information and Bioengineering (DEIB)
Politecnico di Milano

gerardo.pelosi - at - polimi.it

Overview

Lesson contents

- Definition and properties of cryptographic hash functions
- Design principles of hash functions
- Design principles of compression functions
- Common Hash functions
- Message Authentication Codes

Hash Functions

Outline

- Hash functions $h(\cdot)$ are the publicly known algorithms to provide **data integrity**
- Idea: compute a fingerprint of a ptx through a non-injective map
  - the hash computation must be efficient, deterministic and practically unforgeable
- The output size is **constant** (e.g., 160 bits)
- Common names for the output are: *message digest*, *hash* or *cryptographic checksum*

Hash Functions

Sample integrity-check scenario

- Given message m, compute $d = h(m)$ and store it in a safe place
- To check for integrity, recompute $h(m)$ and check against d
- If it matches, then m is still the same as before, otherwise
  - either m or d was tampered with
  - or an error occurred (resp. was injected) over the communication channel
- Note: the computation of $h(m)$ does **not include any key**!

Hash Functions

Formal Definition

A keyed hash function is a 4-uple (M, D, K, H), where:

1. M is a set of input messages (could be unbounded)
2. D is a finite set of digests, with $|\mathbf{M}| \geq |\mathbf{D}|$
3. K is a finite set of keys (the keyspace)
4. H is a finite set of hash functions

- For each key $k \in K$, there is a hash function $h_k : M \rightarrow D$ in H
- A pair (m, d) is called a valid pair under key k, if $h_k(m) = d$

An **unkeyed** hash function is a hash family with a **known fixed key** k

Security of Hash Functions

Unkeyed Hash Functions

- Consider $h : M \rightarrow D$, an unkeyed hash function: the following problems should be (computationally) impossible to solve:

1. **Preimage Problem**: given $d \in D$, find $m \in M$ s.t. $h(m) = d$
   - **One-way** property of h: you cannot reconstruct a valid m from d
2. **Second Preimage Problem**: given $m_1 \in M$, $d = h(m_1)$, find $m_2 \in M$ s.t. $m_1 \neq m_2$, $h(m_2) = h(m_1) = d$
   - **Weak Collision Resistance** property of h: you cannot find a message hashing to the same digest of an already known message
3. **Collision Problem**: find $m_1, m_2 \in M$ s.t. $m_1 \neq m_2$, $h(m_1) = h(m_2)$
   - **(Strong) Collision Resistance** property of h: you cannot find two arbitrary messages with the same digest

- **Weak Collision Resistance** is important because: If we could exhibit another message (or a small set of messages) with the same digest of a given input, the usefulness of the hash function could be at risk!
- **Collision Resistance** is important because: if we could find collisions at our will, we would have the ability to build messages we can subsequently repudiate – as we would be always able to show multiple plausible messages, of our choice, for a certain digest

Security of Hash Functions

A perfect hash: the Random Oracle

- What does a "perfect hash function" (i.e., one-way, weak collision resistant, and strong collision resistant) look like?
- Ideally, a perfect hash should be a **deterministic** function $o(\cdot)$ which returns a random string for every possible input string (Random Oracle)
  - Any call to the oracle with the same input m will yield $o(m)$
  - $o(m_1) = o(m_2)$ **if and only if** $m_1 = m_2$
  - Probability of picking randomly a message with a specific digest $\overline{d}$ is **0**
  - The message space is infinite $\Rightarrow$ the digest space is infinite too!
  - We need an infinite amount of memory to store all the mappings!
- Not practical, but useful to evaluate the security of real hash functions

Security of Hash Functions

Practical 1st Preimage Problem - (black box analysis)

- A real hash function $h(\cdot)$ does not have an infinite digest space
- Trying to find a preimage for $\overline{d} \in D$ calling the hash function with a random input, I will obtain the preimage with probability $\frac{1}{|D|}$
- Probability of calling $h(\cdot)$ q times without success: $\left(1-\frac{1}{|D|}\right)^q$
- Probability of getting **at least** one valid preimage, m_i, for d with q calls is:

$$\Pr(m_i \text{ s.t. } d = h(m_i)) = 1 - \left(1-\frac{1}{|D|}\right)^q$$

$(1-\frac{1}{|D|})^q = (1+\frac{-1}{|D|})^{|D|\frac{q}{|D|}} \overset{|D|\to\infty}{\approx} e^{-\frac{q}{|D|}}$ (notable limit), and

$e^{-\frac{q}{|D|}} \overset{\frac{q}{|D|}\to 0}{\approx} 1-\frac{q}{|D|}$ (notable limit), thus if q is much smaller than $|D|$ (i.e., $q \ll |D|$) we have:

$$\Pr(m_i \text{ s.t. } d = h(m_i)) \approx \frac{q}{|D|}$$

Security of Hash Functions

Practical 2nd Preimage Problem - (black box analysis)

- Let $m \in M$, $d = h(m)$ be the msg-digest pair for which we want a 2nd preimage (i.e., find $m' \in M$ s.t. $m' \neq m$, $h(m') = h(m) = d$)
- Pick q msg.s at random: $m_1, m_2, \ldots m_{q-1}$ s.t. $m \neq m$, $i \in \{1, \ldots, q-1\}$
  - For each chosen message m_i, compute $h(m_i)$
  - If one of the $h(m_i)$ is d, return m_i; otherwise fail
- Again, the probability of getting **at least** one valid preimage of d is:

$$\Pr(\forall\, i \;\; m_i \neq m,\;\; h(m_i) = h(m)) = 1 - \left(1-\frac{1}{|D|}\right)^{q-1} \approx \frac{q-1}{|D|}$$

Security of Hash Functions

Practical Collision Problem - (black box analysis)

- The approach to find a collision in a hash function is
  - Pick q messages at random m_i, $i \in \{1, \ldots, q\}$
  - If any two digests, $h(m_i)$ and $h(m_j)$ with $i \neq j$ ($i, j \in \{1, \ldots, q\}$), are equal then return (m_i, m_j); otherwise, fail
- $q = 1$, $\Pr(\text{no collision}) = 1$
- $q = 2$, $\Pr(\text{no collision}) = 1 \cdot \left(1-\frac{1}{|D|}\right)$
- $q = 3$, $\Pr(\text{no collision}) = 1 \cdot \left(1-\frac{1}{|D|}\right) \cdot \left(1-\frac{2}{|D|}\right)$
- $\ldots$
- q picks, $\Pr(\text{no collision}) = 1 \cdot \left(1-\frac{1}{|D|}\right) \cdot \left(1-\frac{2}{|D|}\right) \cdots \left(1-\frac{q-1}{|D|}\right)$

Security of Hash Functions

Finding the number of trials, q needed to get a collision

- From basic calculus (Taylor series), recall that $1 - \frac{a}{x} \leq e^{-\frac{a}{x}}$
- Consequentially, we can rewrite the previous result as follows:

$$\Pr(\text{no coll.}) = \prod_{i=0}^{q}\left(1-\frac{i}{|D|}\right) \leq \prod_{i=0}^{q-1} e^{-\frac{i}{|D|}} = e^{-\frac{1}{|D|}\sum_{i=0}^{q-1} i} = e^{-\frac{q(q-1)}{2|D|}}$$

- We look for: $\Pr(\textbf{no coll.}) \geq \textbf{50\%} \Leftrightarrow \Pr(\textbf{at least one coll.}) \leq \textbf{50\%}$
- we find that: $e^{-\frac{q(q-1)}{2|D|}} \geq \frac{1}{2} \Rightarrow \frac{q(q-1)}{2|D|} \leq \ln 2$
- Solving for q we conclude that $\Pr(\text{no collision}) \geq \frac{1}{2}$ if $0 < q \leq \frac{1}{2} + \sqrt{\frac{1}{4} + 2|D| \ln 2} \Leftrightarrow$ (roughly) $\mathbf{0 < q \leq 1.1774\sqrt{|D|}}$

Security of Hash Functions

Birthday Paradox

The Birthday paradox is the somewhat surprising answer to:
How many people must I gather in a room in order to have a probability >50% that **any two of them share** *the same birthday?*

- This is the probability to find at least one **collision** on the hash function $\textsc{BirthdayOf}(\cdot)$
- if we do not consider leap years and assume that the birthdays are uniformly distributed, the answer is: $\ldots \approx \lceil 1.1774 \cdot \sqrt{365} \rceil = 23$

Compare the above question with the following one:
how many people must I gather in a room in order to have a probability >50% that one of them has **the same** *birthday* **as me***?*

- This is the probability to find a **2nd preimage** of the $\textsc{BirthdayOf}(\cdot)$ function
- I need a lot more people: ... $\approx \frac{q-1}{365} \geq \frac{1}{2} \Leftrightarrow q \geq 1 + \frac{365}{2},\ q \geq 183$

Security of Hash Functions

Summing up

- To obtain a sound cryptographic hash function we need to find an efficiently computable map $h : M \to D$ such that:
  - It is computationally unfeasible to find a **first/second preimage** in **"less than"** $|D|$ calls to the hash function
  - It is computationally unfeasible to find a **collision** in **less than** $1.17\sqrt{|D|}$ calls to the hash function ($\approx \sqrt{|D|} = \sqrt{2^{\log_2 |D|}} = 2^{\frac{1}{2}\log_2 |D|}$ calls)
- Rule of thumb: take $|D| \geq 2^{160}$ (i.e., a hash with digest $\geq$160-bit) to avoid a bruteforce approach to finding collisions (80-bit security)
- **Note**: Collision Res. $\Rightarrow$ 2nd Preimage Res. $\Rightarrow$ Preimage Res.
- **Note**: Real hash functions have necessarily a very large number of collisions. Their unforgeability comes from the fact that it is just unfeasible to find collisions on purpose

Use Scenarios for Hash functions

Common Scenarios - 1

- **Integrity** of files: hash functions may be used to check the integrity of a network-transmitted file through
  - Downloading both the file f and $d{=}h(f)$ from the providing server
  - Computing $h(\cdot)$ on the downloaded file and checking that it matches with d
- **Pseudo-unique** file **indexing**: a unique index for a file f is obtained as $h(f)$ (e.g., video references on youtube, tiny-urls, Dropbox)
  - As collisions do not practically occur, then $h(f)$ is a good unique index for f

Use Scenarios for Hash functions

Common Scenarios - 2

- **Efficient digital signatures**: instead of signing the file f, sign $h(f)$
- **Safe password storage**: instead of storing a password p, store $d = h(p)$. Even if an attacker retrieves d, he cannot find p
  - having a very fast hash function may be an issue: it allows to quickly test a set of guesses for p extracted from a dictionary
- **Commitment Schemes**. Alice wants to commit to the choice of a value v, without revealing it instantly. She must be able to prove that she committed to that value
- Example: Alice and Bob want to perform a coin toss via e-mail:
  - Alice bets on a value, say, "heads", sends to Bob $h_a{=}h(\texttt{cat}(\text{"heads"}, r_a))$
  - Bob bets on a value say, "tails", sends to Alice $h_b{=}h(\texttt{cat}(\text{"tails"}, r_b))$
  - The coin is tossed, and then Alice and Bob broadcast the values of r_a, r_b and the values (h_a, h_b) they committed to, allowing the commitment check

Common mistakes

Why employing error correction codes is not a good idea?

- A common mistake is to believe that an *error correction code* (e.g., CRC-32) can be a good replacement for a cryptographic hash
- Despite providing integrity checking, their use is a **critical** mistake as it is trivial to find a preimage from a codeword!
- Encrypting the resulting code with a symmetric cipher may be a stopgap solution, as long as the encryption is not xor linear: **no stream cipher or CTR-like** modes of operation should be used[a]!
- Bottom line: if you need a cryptographic hash function, do not use makeshift solutions, as they will break

[a] As it was done, f.i., in WEP

Hash Functions Design

How to design an hash function?

- The design of an infinite domain (unlimited message length) hash function is done in two phases:
  1. designing a finite domain **compression function**
  2. designing a scheme to combine the compression function in order to act on an infinite message space
- Willing to formalize the problem:
  - for simplicity, we consider bit strings as inputs/outputs
  - given a **compression function** $h : M \mapsto D$, each msg is encoded by $l + t$ bits, while each digest is encoded using only l bits: $h : \{0,1\}^{l+t} \mapsto \{0,1\}^{l}$
  - we want to construct a hash function with unlimited domain $\widetilde{h} : \{0,1\}^{*} \mapsto \{0,1\}^{s}$, for some integer s (digest size in bits)

Hash Function Design

From an unlimited string to a sequence of blocks

- We want to compute $\widetilde{h} : \{0,1\}^{*} \mapsto \{0,1\}^{s}$ with a sequence of applications of $h : \{0,1\}^{l+t} \mapsto \{0,1\}^{l}$
- The input message should thus be preprocessed so that the length is a multiple of t
- Use an injective padding function to obtain an input string with length multiple of t (e.g.: concatenate enough zeros, or the size of the message followed by enough zeros)
- Split the padded input string $\overline{m}$ into substrings with t bits each: $\overline{m}_1, \overline{m}_2, \ldots, \overline{m}_r$

Hash Function Design

Digesting the blocks

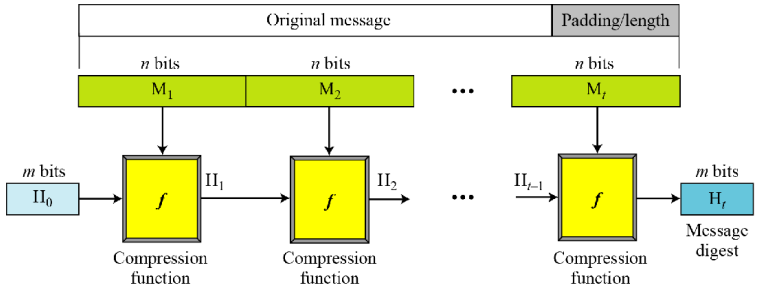
- Once we have the block sequence $\overline{m}_1, \overline{m}_2, \ldots, \overline{m}_r$, we need to devise a scheme to feed them into $h(\cdot, \cdot)$
- Let IV be some public initial value encoded with l-bit, then compute

$$z_0 \leftarrow \mathrm{IV}$$
$$z_i \leftarrow h\left(z_{i-1}, \bar{m}_i\right), \ 1 \leq i \leq r$$

  where the first l-bit parameter of $\mathrm{h}(\cdot, \cdot)$ is called *inner state*, while the second t-bit parameter is the next *message chunk*
- After the last sub-string, $\bar{m}_r$, has been processed, a further output transformation $g : \{0,1\}^{l} \rightarrow \{0,1\}^{s}$ can be applied (usually, it's the identity map, provided $l{=}s$)

Hash Function Design

Merkle-Damgård Construction



- The vast majority of currently used hash functions are based on this construction
- The most common attacks against this structure rely on appending a *chosen block* to the original message to obtain a collision

Compression Function Design

Key points

- The design of compression functions h is younger than the one for block ciphers, thus it tries to reuse some of its key ideas
- The high level structure is the same of block ciphers: employ a round based structure with a small analyzable round primitive
- The inner state of a hash is initialized with constant values, and contains the digest after the execution
- The padded message to be hashed is expanded in a larger message, known as *message schedule*, typically through combining the words via linear operations (XOR and shift/rotate)
- The purpose of the round in a hash function is to blend a part of the message schedule with the inner state (in a nontrivial way)

Compression Function Design principles

- The collision resistance is achieved through mixing the message into the state so that a random change in the message, triggers a bit flip in every bit of the digest with probability as close as possible to $\frac{1}{2}$ (this is commonly known as **avalanche effect**)
- The message mixing is usually performed through a nonlinear addition (f.i. the addition modulo 2^{32}) during the round function
- The message addition may involve the whole state (similar to the SPN design of a cipher), or modify only a part of it and shuffle the parts (similar to the Feistel Network design)
- The number of rounds is higher than the block ciphers (numbers in the $[64 - 80]$ range are not uncommon) as there is no secret key involved in the computation

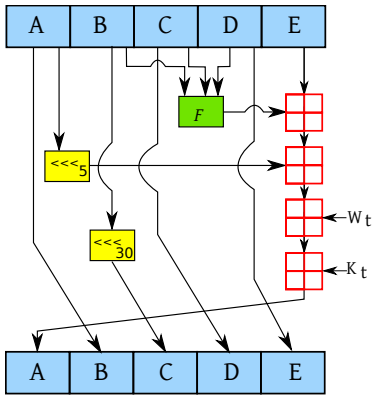
Secure Hash Algorithms

The most common hashing algorithms are:

- MD4 (1990): 512↦128 bit digest, collision attack with 2 msg
- MD5 (1992): 512↦128 bit digest, arbitrary collision attack appending one block
- SHA-0 (1993): 512↦160 bit digest, NSA reported it as weak, proposed a corrected version SHA-1
- SHA-1 (1995): 512↦160 bit digest, same as SHA-0 with an single bit rotation in the message schedule,
  collisions reported w/ $\approx 2^{69}$ **hash calls**
  ($\ll 2^{\text{DIGEST-BIT-LENGTH}/2} = 2^{80}$)
  (ref., X. Wang et al., Finding Collisions in the Full SHA-1, CRYPTO'05)
- SHA-2 (2001): $\{512, 1024\} \mapsto \{224, 256, 384, 512\}$ bit digest, shares structure with SHA-1, different mixing functions

Compression Function Design

SHA-1 Compression function



- Yellow boxes are bitwise rotations
- W_t is a 32-bit word of the message
- K_t is a 32-bit round-dependent constant word
- F is a round dependent bitwise function

Hash functions from block ciphers

- It is possible to employ a common block cipher in place of the ad-hoc designed compression function h
- To this end, the message should be padded to be a multiple of the length of the cipher block, and split accordingly in a sequence of blocks $x_1, \ldots, x_r$
- The proposed schemes maintain an inner state which is initialized to a constant known value $H_0 = \mathrm{IV}$
- The update rule of the inner state takes a block of the padded message x_i and computes $H_i = f(H_{i-1}, x_i)$
- The final digest of the message is: H_{r+1}.

Note: there are several ways to use the chosen block cipher primitive as a compression function f

Hash functions from block ciphers

DM (Davies-Meyer) Scheme

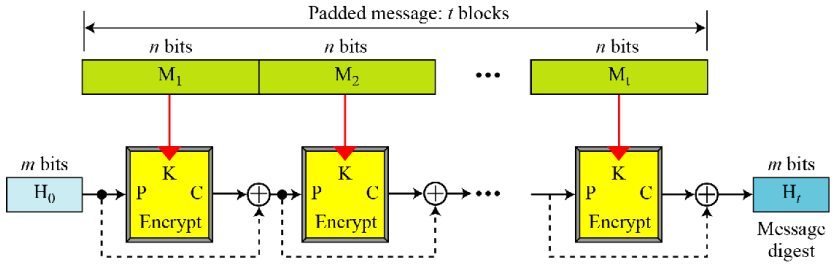


Figure: $f(x_i, H_{i-1}) = E_{x_i}(H_{i-1}) \oplus H_{i-1}$

- Solves the issue of generating collision via appending a block

Hash functions from block ciphers

MMO (Matyas-Meyer-Oseas) Scheme

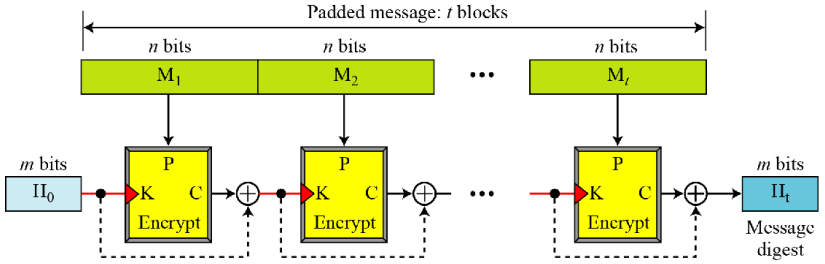


Figure: $f(x_i, H_{i-1}) = E_{H_{i-1}}(x_i) \oplus x_i$

- Avoids using the message as the key of the block cipher, uses the previous state as key

Hash functions from block ciphers

MP (Miyaguchi-Preneel) Scheme

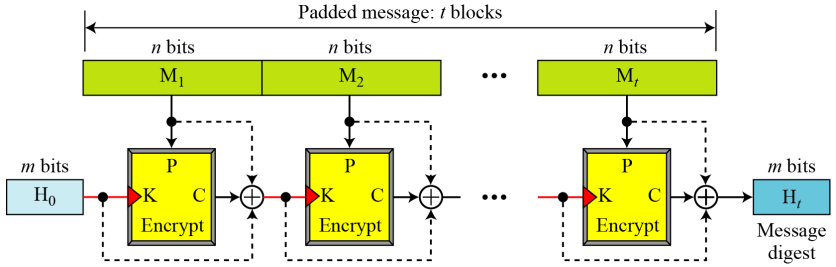


Figure: $f(x_i, H_{i-1}) = E_{H_{i-1}}(x_i) \oplus x_i \oplus H_{i-1}$

- Also adds the message to the output of the single compression function

Keyed Hash Functions Design

Keyed Hash Basics

- Incorporate a secret key into an unkeyed hash function by including the key as part of the message.
- If the employed hash function is constructed with the Merkle-Damgård structure, care is needed to choose **where** the key is added

$$H(m) = H(\langle m_1, m_2, \ldots, m_r \rangle) = \\ = h(\ldots h(h(\mathrm{IV}, m_1), m_2), \ldots, m_r) \quad |m_i| = t\text{-bit}$$

- The key should be added **neither as a prefix nor as a suffix** of the message.

Keyed Hash Functions

Key Prefix attack against the Merkle-Damgård structure

- **Given (m, d), with m being a (padded) message and d=$H(k||m)$ an attacker can provide a message-digest pair with a message of his choice without knowing the secret key.**

  Choosing an arbitrary message m', the attacker may forge a keyed-digest d' without knowing k as follows:

$$d' = h(d||m') \text{ as } h(d||m') = h(\ h(k||m)\ ||\ m'\ ) = H(k||m||m')$$

  (the symbol '||' means *concatenation*)
- $(m',\ d')$ is another valid message-digest pair!

Keyed Hash Functions

Key Suffix attack against the Merkle-Damgård structure

Given (m, d), d=$H(m||k)$, an attacker can re-use d as a digest for any message m' colliding with the original one, i.e.: $H(m)$=$H(m')$

- **If the length of** the known message m **is a multiple of** t **bits**: $m = m_1||m_2 \ldots ||m_r$, the attacker can consider the last block m_r of the msg and the l-bit block y derived from the MD construction and can observe that:
  $H(m)$=$h(y, m_r)$, with y=$h(\ \ldots h(\mathrm{IV}, m_1), \ldots, m_{r-1})$, while
  $H(m||k)$=$h(h(y, m_r), k)$
- If the attacker obtains a msg m' colliding with m, that is: $H(m')$=$H(m)$=$h(y, m_r)$
- The keyed digest of m' would be: $H(m'||k)$=$h(h(y||m_r)||k)$=d, thus
  $(m',\ d)$ is another valid message-digest pair!
- When m is shorter than t bits, the above collision attack cannot be mounted!... because part of the key k would become part of the first t-bit chunk in input to $h(\cdot, \cdot)$

Message Authentication Codes

A keyed hash function is often used as a message authentication code (MAC)

- A MAC can be appended to a sequence of plaintext blocks
- Used to prove to the receiver that the given plaintext originated from the rightful sender and was not tampered with
- This is the original scenario that I gave at the beginning of lecture

Common Ways to Create MAC (1)

A common and widely standardised way to construct a MAC is the HMAC (keyed-Hash Message Authentication Code):

- The construction allows to build a MAC from any unkeyed hash function
- Example based on SHA-1, with key size 512 bits:
  - ipad=512 bits constant 0x363636..36
  - opad=512 bits constant 0x5c5c5c..5c
  - $T = \text{SHA-1}((k \oplus \texttt{ipad}) || m)$
  - $\text{HMAC}_k(m) = \text{SHA-1}((k \oplus \texttt{opad}) || T)$
  - The scheme relies on the composition of two applications of a keyed hash function

Common Ways to Create MAC (2)

- If design constraints do not allow to implement a separate compression function (e.g., SHA-1) to build a HMAC, a possibility is to reuse a block cipher as "compression" function.
- A first attempt was the CBC-MAC: Use a block cipher in CBC mode, keep only the last encrypted block as MAC
  - Problem: if d is a valid MAC for $m = \texttt{cat}(m_1, \ldots, m_n)$, then d is a valid MAC also for $m = \texttt{cat}(m_1, \ldots, m_n, \text{IV} \oplus d \oplus m_1, \ldots, m_n)$
- To solve the problem with CBC-MAC it is possible to employ a *whitening step*, which involves adding with an XOR the key to the whole input message, before applying the block cipher encryption
- A proper way to do so has been standardized as CMAC, and is currently accepted for the IpSec checksums

Why design a MAC?

Why do we use MACs instead of encrypting the digest of an unkeyed hash function through employing a block cipher?

Why design a MAC?

Why do we use MACs instead of encrypting the digest of an unkeyed hash function through employing a block cipher?

- Block size of hash primitives is larger than block cipher ones, thus the schemes based on a general compression function can exhibit a better collision resistance. The computation of a MAC should be more efficient than "hash and then encrypt".
- Software and Hardware crypto-libraries usually includes hash functions because **there is no country with a regulation that explicitly forbid the export/import of them**
  - USA applies export restrictions on the design or implementation of ciphers
  - Russia forbids the import of any cryptographic solution for official or public use – The ISO standard "GOST R" was entirely designed in Russia to have a block cipher equivalent to either DES or AES