CS306: Introduction to IT Security

Fall 2020

Labs 5 & 6: More on Hashing

Instructor: **Nikos Triandopoulos**

October 8 & 15, 2020

STEVENS
INSTITUTE *of* TECHNOLOGY
THE INNOVATION UNIVERSITY®

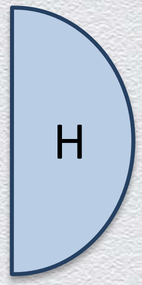
6.5 Hash functions

Cryptographic hash functions

Basic cryptographic primitive

- maps **"objects"** to a **fixed-length** binary strings
- core security property: mapping **avoids collisions**
  - **collision**: distinct objects ($x \neq y$) are mapped to the same hash value ($H(x) = H(y)$)
  - although collisions **necessarily exist**, they are **infeasible to find**

input
arbitrarily long string



output
short digest, fingerprint, "secure" description

Important role in modern cryptography

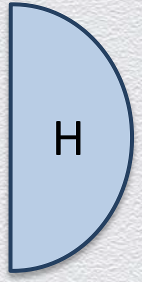
- lie between symmetric- and asymmetric-key cryptography
- capture different security properties of "idealized random functions"
- qualitative stronger assumption than PRF

Hash & compression functions

Map messages to short digests

- a **general** hash function H() maps
  - a message of an **arbitrary length** to a ***l*(n)-bit** string

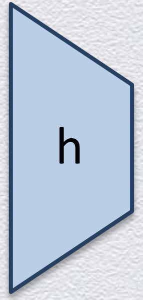
input
arbitrarily
long string



output
***l*(n)-bit**
string

- a **compression** (hash) function h() maps
  - a **long** binary string to a **shorter** binary string
  - an ***l'*(n)-bit string** to a ***l*(n)-bit string**, with ***l'*(n) > *l*(n)**

input
***l'*(n)-bit**
string



output
***l*(n)-bit**
string

Collision resistance (CR)

Attacker wins the game if $x \neq x'$ & $H(x) = H(x')$

H is collision-resistant if any PPT $\mathcal{A}$ wins the game only negligibly often.

Weaker security notions

Given a hash function $H: X \to Y$, then we say that H is

- **preimage resistant** (or **one-way**)
  - if given $y \in Y$, finding a value $x \in X$ s.t. $H(x) = y$ happens negligibly often
- **2-nd preimage resistant** (or **weak collision resistant**)
  - if given a **uniform** $x \in X$, finding a value $x' \in X$, s.t. $x' \neq x$ and $H(x') = H(x)$ happens negligibly often

- cf. **collision resistant** (or **strong collision resistant**)
  - if finding two distinct values $x', x \in X$, s.t. $H(x') = H(x)$ happens negligibly often

6.6 Design framework

Domain extension via the Merkle-Damgård transform

General design pattern for cryptographic hash functions

- reduces CR of general hash functions to CR of compression functions

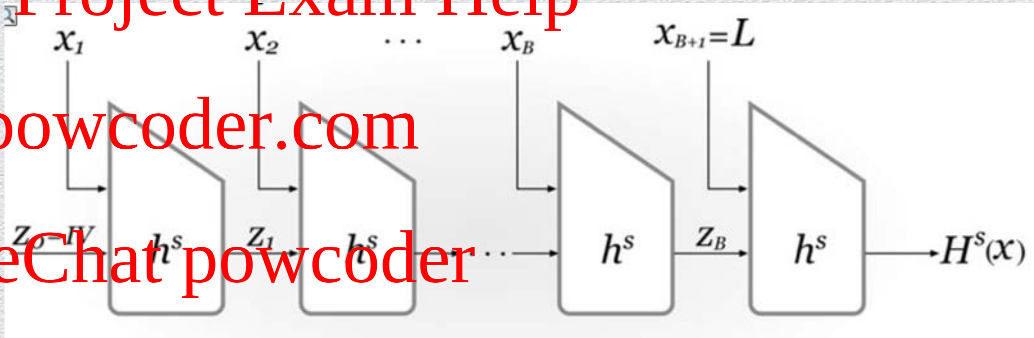
- thus, in practice, it suffices to realize a collision-resistant compression function h
- compressing by 1 single bit is a least as hard as compressing by any number of bits!

Merkle-Damgård transform: Design

Suppose that h: $\{0,1\}^{2n} \rightarrow \{0,1\}^n$ is a collision-resistant compression function

Consider the general hash function H: $\mathcal{M}$ **= {x : |x|<2ⁿ}** $\rightarrow \{0,1\}^n$, defined as

<u>Merkle-Damgård design</u>

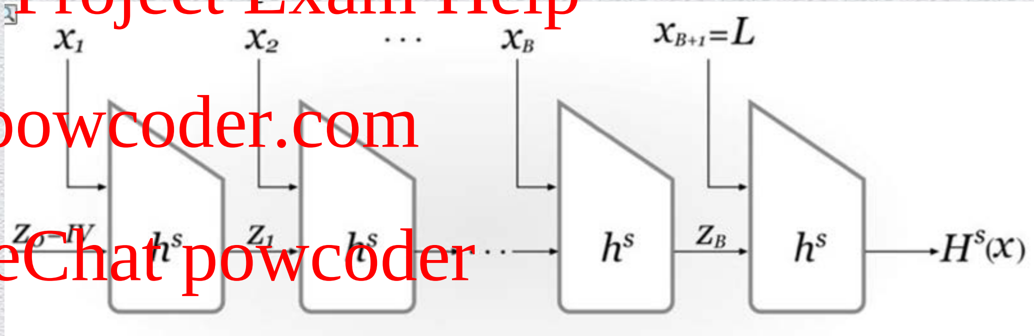


- H(x) is computed by applying h() in a **"chained" manner** over n-bit message blocks

  - pad x to define a number, say B, **message blocks $x_1, \ldots, x_B$**, with $|x_i| = n$
  - set extra, final, message block **x_{B+1} as an n-bit encoding L of $|x|$**
  - starting by initial digest **$z_0 = IV = 0^n$**, output **$H(x) = z_{B+1}$**, where **$z_i = h^s(z_{i-1} || x_i)$**

Merkle-Damgård transform: Security

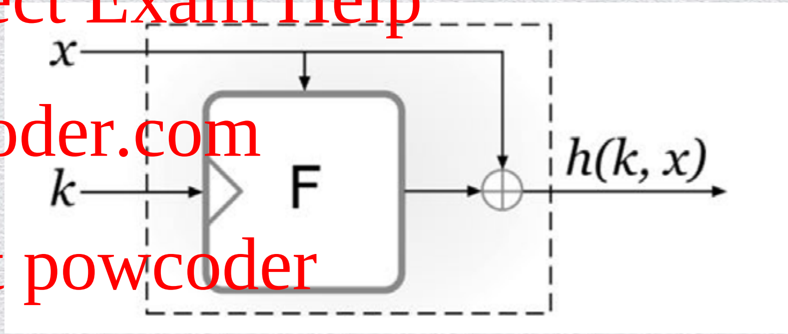
If the compression function h is CR,
then the derived hash function H is also CR!



Compression function design: The Davies-Meyer scheme

Employs PRF w/ key length m & block length n

- define h: $\{0,1\}^{n+m} \rightarrow \{0,1\}^n$ as **$h(x||k) = F_k(x) \text{ XOR } x$**



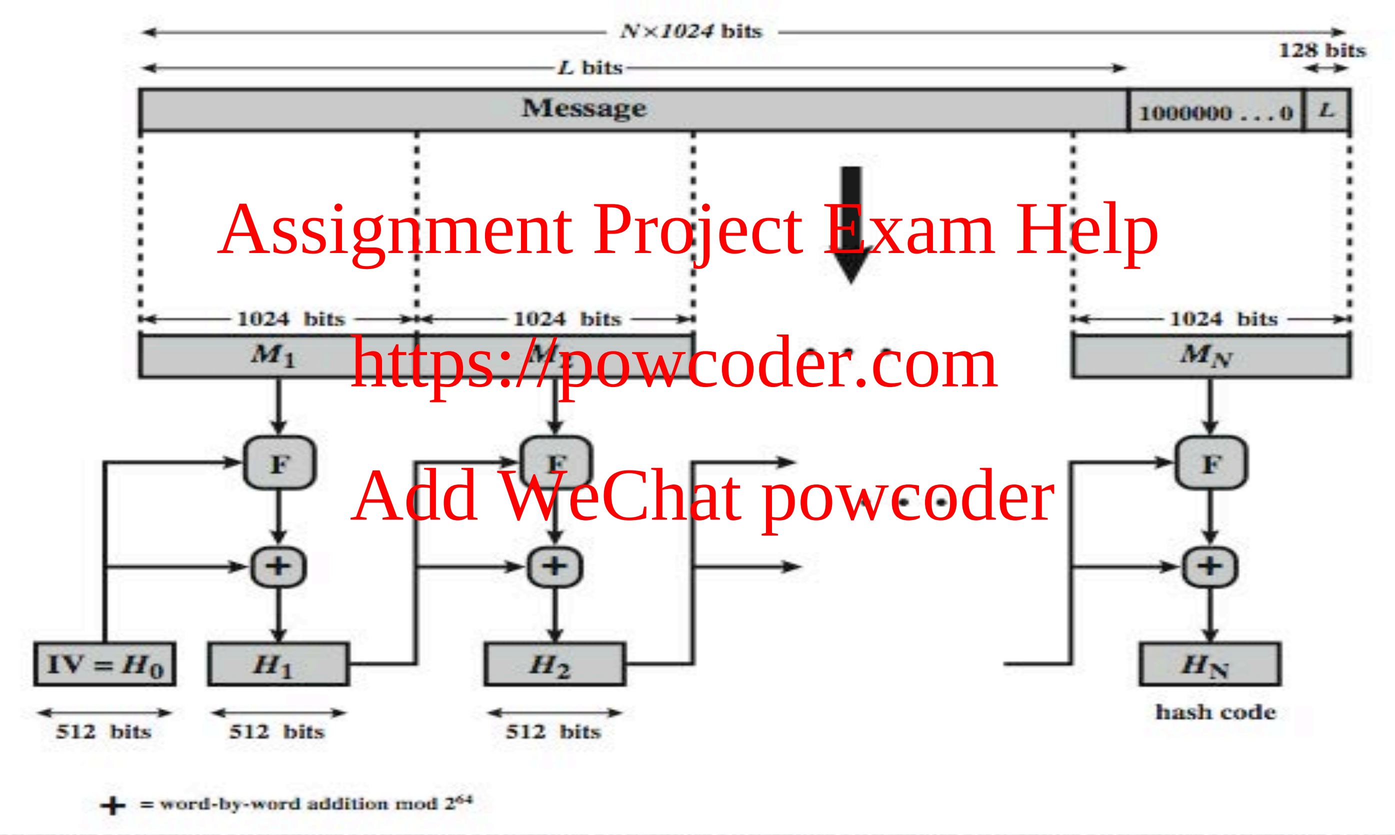
Security

- h is CR, if F is an **ideal cipher**

Well known hash functions

- MD5 (designed in 1991)
  - output 128 bits, collision resistance **completely broken** by researchers in 2004
  - today (controlled) collisions can be found in less than a minute on a desktop PC
- SHA1 – the Secure Hash Algorithm (series of algorithms standardized by NIST)
  - output 160 bits, considered **insecure** for collision resistance
  - **broken** in 2017 by researchers at CWI
- SHA2 (SHA-224, SHA-256, SHA-384, SHA-512)
  - outputs 224, 256, 384, and 512 bits, respectively, **no real security concerns yet**
  - based on Merkle-Damgård + Davies-Meyer generic transforms
- SHA3 (Kessac)
  - **completely new philosophy** (sponge construction + unkeyed permutations)

SHA-2-512 overview



Current hash standards

Algorithm	Maximum Message Size (bits)	Block Size (bits)	Rounds	Message Digest Size (bits)
MD5	2^{64}	512	64	128
SHA-1	2^{64}	512	80	160
SHA-2-224	2^{64}	512	64	224
SHA-2-256	2^{64}	512	64	256
SHA-2-384	2^{128}	1024	80	384
SHA-2-512	2^{128}	1024	80	512
SHA-3-256	unlimited	1088	24	256
SHA-3-512	unlimited	576	24	512

6.7 Generic attacks

Generic attacks against cryptographic hashing

Assume a CR compression function h : $\{0,1\}^{\ell'(n)} \rightarrow \{0,1\}^{\ell(n)}$

- **brute-force** attack
  - for each string x in the domain
    - compute and record hash value h(x)
    - if h(x) equals a previously recorded hash h(y) (i.e., $x \neq y$ but $h(x)=h(y)$), halt and output collision on $x \neq y$
- **birthday** attack
  - surprisingly, a more efficient generic attack exists!

Birthday paradox

"In any group of 23 people (or more), it is **more likely** (than not) that **at least two** individuals have their birthday on the **same** day"

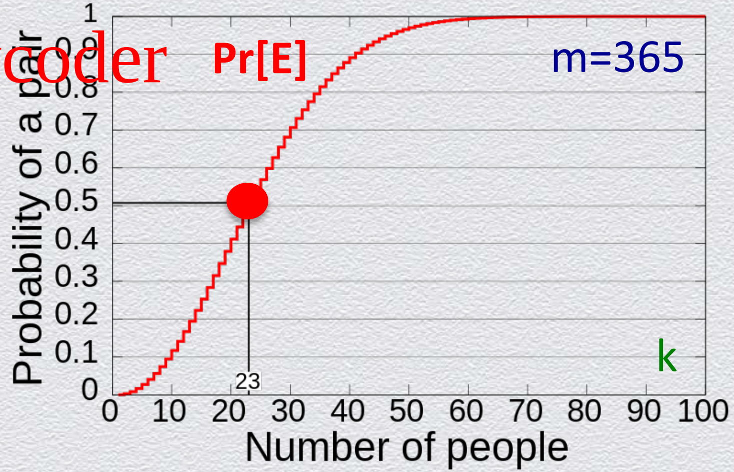
- based on probabilistic analysis of a random "balls-into-bins" experiment:

  "k balls are each, independently and randomly, thrown into one out of m bins"

- captures likelihood that event E = "**two balls land into the same bin**" occurs

- analysis shows: $\Pr[E] \approx 1 - e^{-k(k-1)/2m}$ (1)

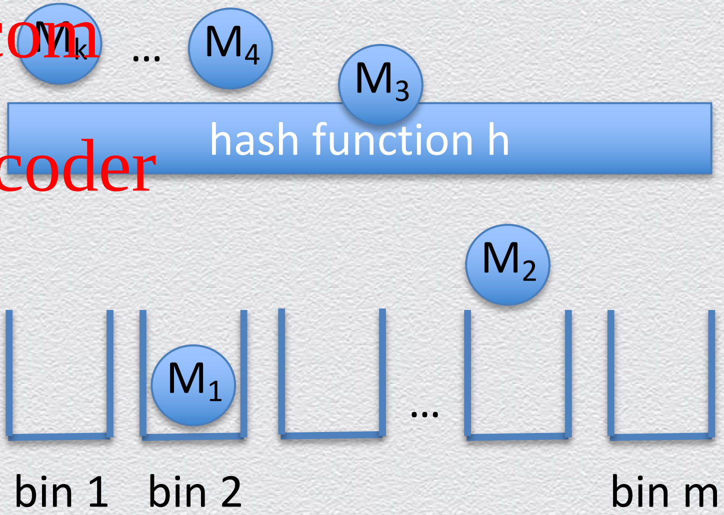
  - if **$\Pr[E] = 1/2$**, Eq. (1) gives **$k \approx 1.17\, m^{1/2}$**
  - thus, for $m = 365$, k is around 23 (!)
    - assuming a uniform birth distribution



Birthday attack

Applies “birthday paradox” against cryptographic hashing

- exploits the likelihood of finding collisions for hash function h using a **randomized** search, rather than an **exhaustive** search
- analogy
  - k balls: distinct messages chosen to hash
  - m bins: number of possible hash values
  - <u>independent & random</u> throwing
    - how is this achieved?
    - message selection, hash mapping



Probabilistic analysis

Experiment

- k balls are each, independently and randomly, thrown into one out of m bins

Analysis

- the probability that the i-th ball lands in an empty bin is: $1 - (i - 1)/m$

- the probability F_k that after k throws, no balls land in the same bin is:

$$F_k = (1 - 1/m)(1 - 2/m)(1 - 3/m) \dots (1 - (k - 1)/m)$$

- by the standard approximation $1 - x \approx e^{-x}$: $F_k \approx e^{-(1/m + 2/m + 3/m + \dots + (k-1)/m)} = e^{-k(k-1)/2m}$
- thus, two balls land in same bin with probability $Pr[E] = 1 - F_k = 1 - e^{-k(k-1)/2m}$
- **lower bound** – Pr[E] increases if the bin-selection distribution is not uniform

What birthday attacks mean in practice…

- # hash evaluations for finding collisions on n-bit digests with probability p

Bits n	Possible outputs (2 s.f.) (H) m	Desired probability of random collision (2 s.f.) (p)									
		10^{-18}	10^{-15}	10^{-12}	10^{-9}	10^{-6}	0.1%	1%	25%	50%	75%
16	65,536	<2	<2	<2	<2	<2	11	36	190	300	430
32	4.3×10^{9}	<2	<2	<2	3	93	2900	9300	50,000	77,000	110,000
64	1.8×10^{19}	6	190	6100	190,000	6,100,000	1.9×10^{8}	6.1×10^{8}	3.3×10^{9}	5.1×10^{9}	7.2×10^{9}
128	3.4×10^{38}	2.6×10^{10}	8.2×10^{11}	2.6×10^{13}	8.2×10^{14}	2.6×10^{16}	8.3×10^{17}	2.6×10^{18}	1.4×10^{19}	2.2×10^{19}	3.1×10^{19}
256	1.2×10^{77}	4.8×10^{29}	1.5×10^{31}	4.8×10^{32}	1.5×10^{34}	4.8×10^{35}	1.5×10^{37}	4.8×10^{37}	2.6×10^{38}	4.0×10^{38}	5.7×10^{38}
384	3.9×10^{115}	8.9×10^{48}	2.8×10^{50}	8.9×10^{51}	2.8×10^{53}	8.9×10^{54}	2.8×10^{56}	8.9×10^{56}	4.8×10^{57}	7.4×10^{57}	1.0×10^{58}
512	1.3×10^{154}	1.6×10^{68}	5.2×10^{69}	1.6×10^{71}	5.2×10^{72}	1.6×10^{74}	5.2×10^{75}	1.6×10^{76}	8.8×10^{76}	1.4×10^{77}	1.9×10^{77}

- for large $m = 2^n$, average # hash evaluations before finding the first collision is

$1.25(m)^{1/2}$

Overall

Assume a CR function h producing hash values of size n

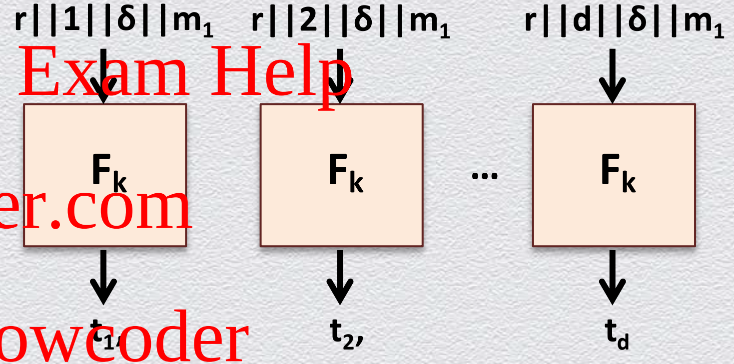
- **brute-force** attack
  - evaluate h on **$2^n + 1$** distinct inputs
  - by the "pigeon hole" **principle**, at least 1 collision **will be** found
- **birthday** attack
  - evaluate h on (much) **fewer** distinct inputs that hash to **random** values
  - by "balls-into-bins" **probabilistic analysis**, at least 1 collision will **likely** be found
  - when hashing **only half** distinct inputs, it's **more likely** to find a collision!
  - thus, in order to get **n-bit security**, we (**at least**) need **hash values of length 2n**

6.8 Applications of hashing to cryptography

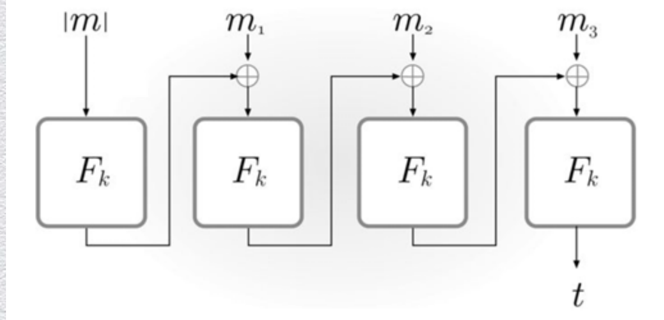
Hash functions enable efficient MAC design!

Back to problem of designing **<u>secure MAC for messages of arbitrary lengths</u>**

- so far, we have seen two solutions
  - block-based "tagging"
    - based on PRFs
    - inefficient



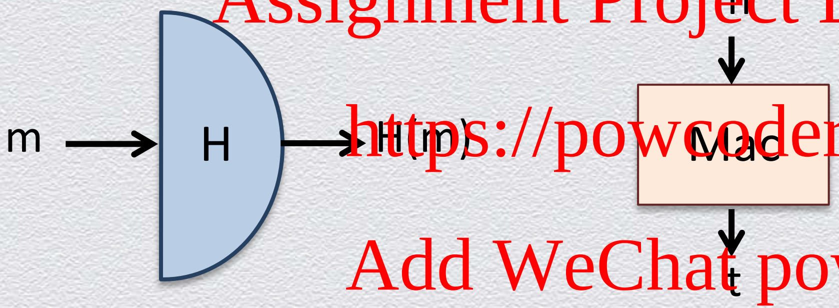
  - CBC-MAC
    - also based on PRFs
    - more efficient

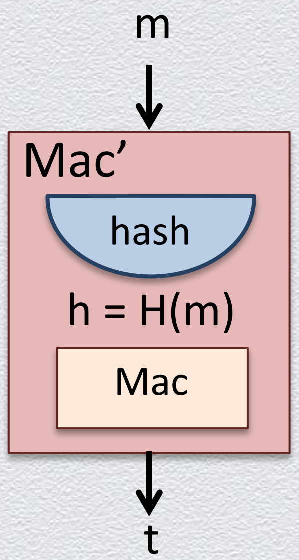


[1] Hash-and-MAC: Design

Generic method for designing **secure MAC for messages of arbitrary lengths**

- based on **CR hashing** and **any fix-length secure MAC**





- new MAC (Gen', Mac', Vrf') **as the name suggests**
  - Gen': **instantiate** H and Mac_k with key k
  - Mac': **hash** message m into h = H(m), output **Mac_k**-tag t on h
  - Vrf': **canonical** verification

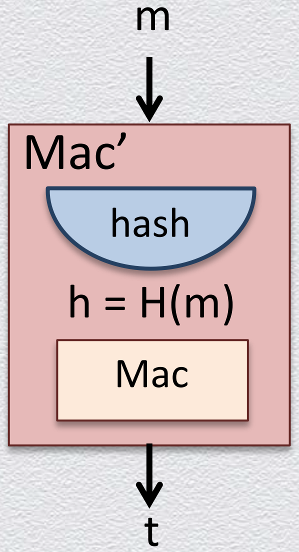
[1] Hash-and-MAC: Security

The Hash-and-MAC construction is a secure as long as

- H is **collision resistant**; and
- the underlying MAC is **secure**

Intuition

- since **H is CR**:
  authenticating **digest H(m)** is **a good as** authenticating **m itself**!



[2] Hash-based MAC

- so far, MACs are based on block ciphers
- can we construct a MAC based on CR hashing?

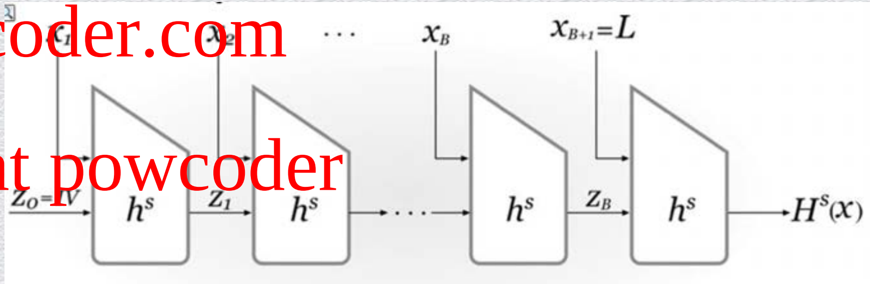
[2] A naïve, insecure, approach

Set tag t as:

$$\text{Mac}_k(m) = \mathbf{H}(k||\mathbf{m})$$

- intuition: given H(k||m) it should be infeasible to compute H(k||m'), m' ≠ m

Insecure construction



- practical CR hash functions employ the Merkle-Damgård design

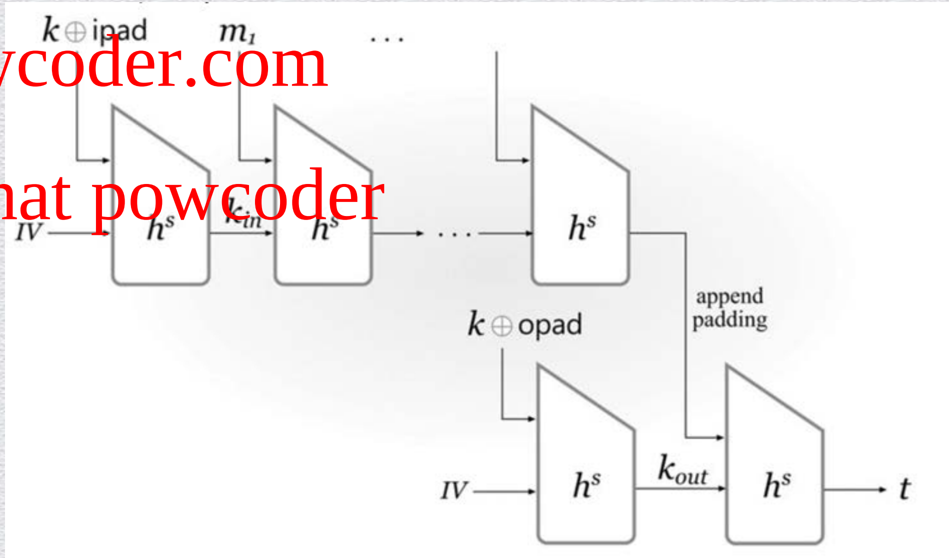
- **length-extension attack**
  - knowledge of $H(m_1)$ makes it feasible to compute $H(m_1||m_2)$
  - by knowing the length of m_1, one can learn internal state z_B even without knowing m_1!

[2] HMAC: Secure design

Set tag t as:

$$\mathrm{HMAC}_k[m] \;=\; \mathbf{H}\,[\;(k \oplus \mathrm{opad}) \;||\; \mathbf{H}[\;(k \oplus \mathrm{ipad}) \;||\; \mathbf{m}\;]\;]$$

- intuition: **instantiation of hash & sign paradigm**
- two layers of hashing H
  - **upper layer**
    - y = H($(k \oplus \mathrm{ipad})$ || m)
    - y = H'(m), i.e., "hash"
  - **lower layer**
    - t = H ($(k \oplus \mathrm{opad})$ || y')
    - t = Mac'(k_{out}, y'), i.e., "sign"



[2] HMAC: Security

If used with a secure hash function and according to specs, HMAC is secure

- no practical attacks are known against HMAC

6.9 Applications to security

Recall: Weaker security notions

Given a hash function $H: X \to Y$, then we say that H is

- **preimage resistant** (or **one-way**)

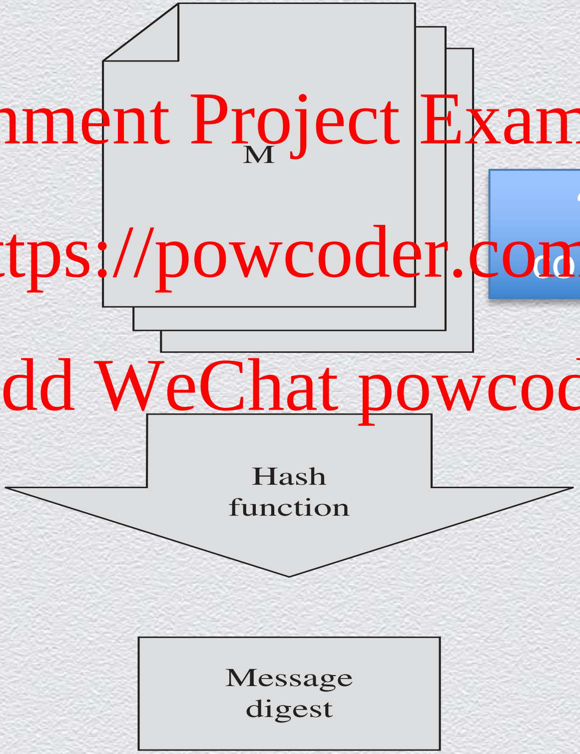
  - if given $y \in Y$, finding a value $x \in X$ s.t. $H(x) = y$ happens negligibly often
- **2-nd preimage resistant** (or **weak collision resistant**)

  - if given a **uniform** $x \in X$, finding a value $x' \in X$, s.t. $x' \neq x$ and $H(x') = H(x)$ happens negligibly often

- cf. **collision resistant** (or **strong collision resistant**)
  - if finding two distinct values $x', x \in X$, s.t. $H(x') = H(x)$ happens negligibly often

Generally: Message digests

Short secure description of data primarily used to detect changes



“one way compression”

Application 1: Secure cloud storage

- Bob has files $f_1, f_2,\ldots,f_n$
- Bob sends to a cloud storage provider
  - the hashes $h(f_1||r), h(f_2||r),\ldots, h(f_n||r)$
  - files $f_1, f_2,\ldots,f_n$
- Bob stores locally randomness r and keeps it secret
- Every time Bob **reads** a file f_i, he also reads $h(f_i||r)$ and verifies the integrity of f_i
- Any problems with **writes**?

Application 2: Fairness (I)

Suppose Alice, Bob, Charlie are bidders in an online auction

- Alice plans to bid A, Bob B and Charlie C
  - they do not trust that bids will be secret
  - nobody is willing to submit their bid
- solution
  - Alice, Bob, Charlie submit **hashes** h(A), h(B), h(C) of their bids
  - all received hashes are posted online
  - then parties' bids A, B and C revealed
- analysis
  - "hiding:" hashes do not reveal bids (which property?)
  - "binding:" cannot change bid after hash sent (which property?)

Application 2: Fairness (II)

A general issue with concealing private data via hashing

- due to the small search space, this protocol is not secure!
- a forward search attack is possible
  - e.g., Bob computes h(A) for the most likely bids A
- how to prevent this?
  - increase search space
  - e.g., Alice computes h(A||R), where R is randomly chosen
    - at the end, Alice must reveal A and R
    - but before he chooses B, Bob cannot try all A and R combination

Application 2: Digital envelops

Commitment schemes

- two operations
- commit(x, r) = C
  - i.e., put message x into an envelop (using randomness r)
  - e.g., commit(x, r) = h(x || r)
  - **hiding property**: you cannot see through an (opaque) envelop
- open(C, m, r) = ACCEPT or REJECT
  - i.e., open envelop (using r) to check that it has not been tampered with
  - e.g., open(C, m, r): check if h(x || r) =? C
  - **binding property**: you cannot change the contents of a sealed envelop

Application 2: Security properties

Hiding: perfect opaqueness

- similar to indistinguishability; commitment reveals nothing about message
  - adversary selects two messages x_1, x_2 which he gives to challenger
  - challenger randomly selects bit b, computes (randomness and) commitment C_i of x_i
  - challenger gives C_b to adversary, who wins if he can find bit b (better than guessing)

Binding: perfect sealing

- similar to unforgeability; cannot find a commitment “collision”
  - adversary selects two distinct messages x_1, x_2 and two corresponding values r_1, r_2
  - adversary wins if $\text{commit}(x_1, r_1) = \text{commit}(x_2, r_2)$

Example 2: Fair decision via coin flipping

Alice is to “call” the coin flip and Bob is to flip the coin

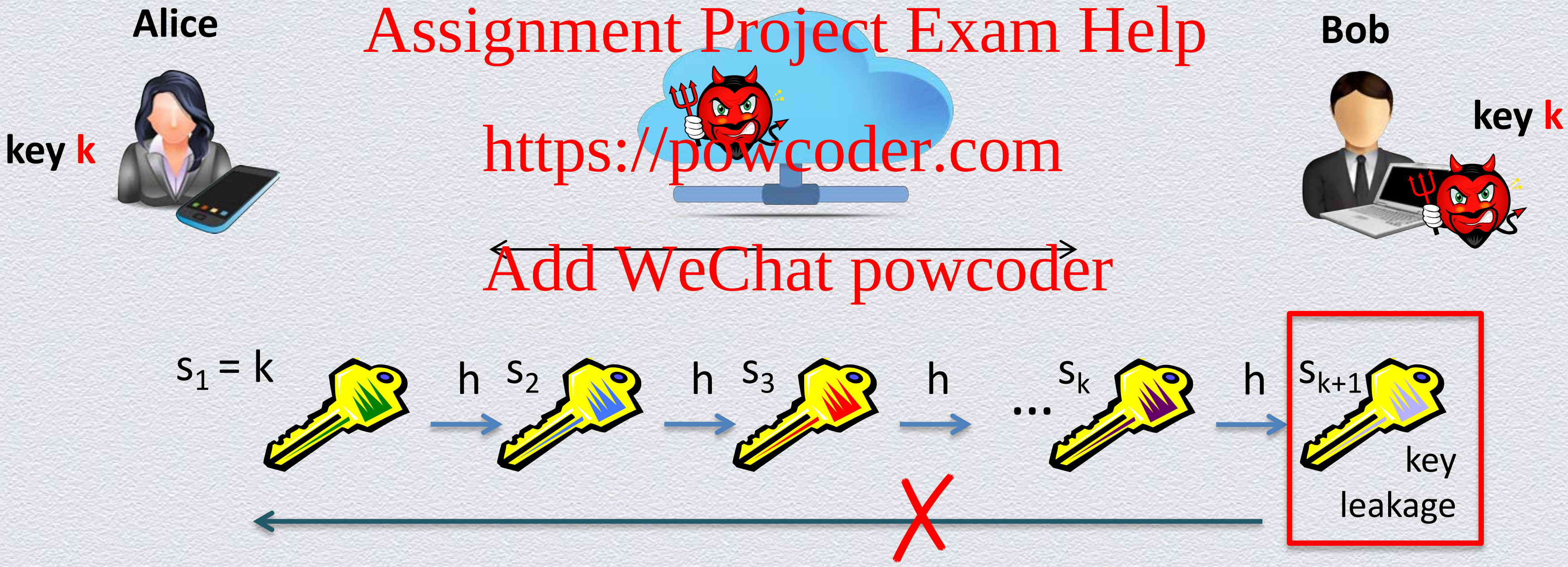
- to decide who will do the dishes…
- problem: Alice may change her mind, Bob may skew the result

- protocol
  - Alice "calls" the coin flip but only tells Bob a commitment to her call
  - Bob flips the coin and reports the result
  - Alice reveals what she committed to
  - Bob verifies that Alice's call matches her commitment
  - If Alice's revelation matches the coin result Bob reported, Alice wins
- hiding: Bob does not get any advantage by seeing Alice commitment
- binding: Alice cannot change her mind after the coin is flipped

Application 3: Forward-secure key rotation

Alice and Bob secretly communicate using symmetric encryption

- Eve intercepts their messages and later breaks into Bob's machine to steal the shared key



Application 4: Hash values as file identifiers

Consider a cryptographic hash function H applied on a file F

- the hash (or digest) H(M) of F serves as a **unique** identifier for F
  - "uniqueness"
    - if another file F' has the same identifier, this contradicts the security of H
  - thus
    - the hash H(F) of F is like a fingerprint
    - one can check whether two files are equal by comparing their digests

Many real-life applications employ this simple idea!

Examples

4.1 Virus fingerprinting

- When you perform a virus scan over your computer, the virus scanner application tries to identify and block or quarantine programs or files that contain viruses
- This search is primarily based on comparing the digest of your files against a database of the digests of already known viruses
- The same technique is used for confirming that is safe to download an application or open an email attachment

4.2 Peer-to-peer file sharing

- In distributed file-sharing applications (e.g., systems allowing users to contribute contents that are shared amongst each other), both shared files and participating peer nodes (e.g., their IP addresses) are uniquely mapped into identifiers in a hash range
- When a given file is added in the system it is consistently stored at peer nodes that are responsible to store files those digests fall in a certain sub-range
- When a user looks up a file, routing tables (storing values in the hash range) are used to eventually locate one of the machines storing the searched file

Example 4.3: Data deduplication

Goal: Elimination of duplicate data

- Consider a cloud provider, e.g., Gmail or Dropbox, storing data from numerous users.
- A vast majority of stored data are duplicates; e.g., think of how many users store the same email attachments, or a popular video…
- Huge cost savings result from deduplication:
  - a provider stores identical contents possessed by different users once!
  - this is completely transparent to end users!

Idea: Check redundancy via hashing

- Files can be reliably checked whether they are duplicates by comparing their digests.
- When a user is ready to upload a new file to the cloud, the file's digest is first uploaded.
- The provider checks to find a possible duplicate, in which case a pointer to this file is added.
- Otherwise, the file is being uploaded literally
- This approach saves both storage and bandwidth!

Example 4.4: Password hashing

Goal: User authentication

- Today, passwords are the dominant means for user authentication, i.e., the process of verifying the identity of a user (requesting access to some computing resource).
- This is a “something you know” type of user authentication, assuming that only the legitimate user knows the correct password.
- When you provide your password to a computer system (e.g., to a server through a web interface), the system checks if your submitted password matches the password that was initially stored in the system at setup.

Problem: How to protect password files

- If password are stored at the server in the clear, an attacker can steal the password file after breaking into the authentication server – this type of attack happens routinely nowadays…
- Password hashing involved having the server storing the hashes of the users passwords.
- Thus, even if a password file leaks to an attacker, the onewayness of the used hash function can guarantee some protections against user-impersonation simply by providing the stolen password for a victim user.

Example 4.4: Password storage

Identity	Password
Jane	qwerty
Pat	aaaaaa
Phillip	oct31witch
Roz	aaaaaa
Herman	guessme
Claire	aq3wm$oto4

Plaintext

Identity	Password
Jane	0x471aa2d2
Pat	0x13b9c32f
Phillip	0x01c142be
Roz	0x13b9c32f
Herman	0x5202aae2
Claire	0x488b8c27

Concealed via hashing

Application 5: Hash-and-digitally-sign (looking ahead)

Very often digital signatures are used with hash functions

- the hash of a message is signed, instead of the message itself

Signing message M

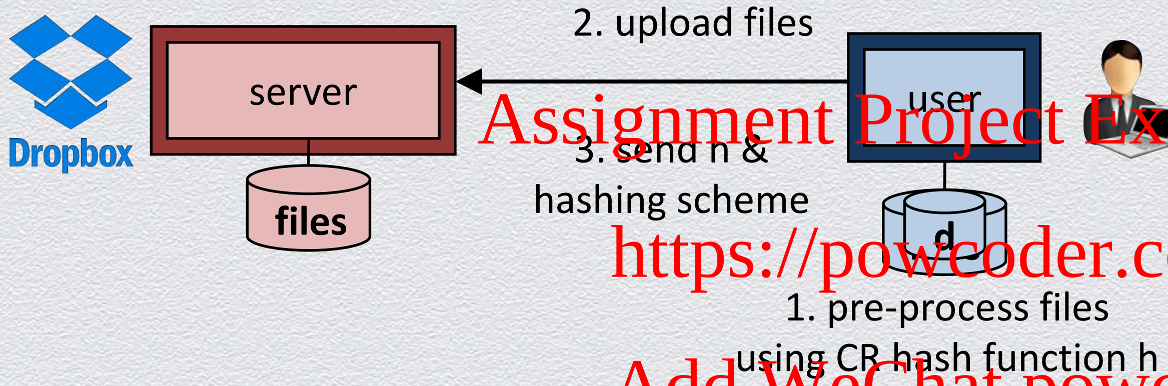
- let h be a cryptographic hash function, assume RSA setting (n, d, e)
- compute signature $\sigma = h(M)^d \bmod n$
- send σ, M

Verifying signature σ

- use public key (e,n)
- compute $H = \sigma^e \bmod n$
- if H = h(M) output ACCEPT, else output REJECT

6.10 Database-as-a-service authentication model

Securing your cloud storage optimally – setup

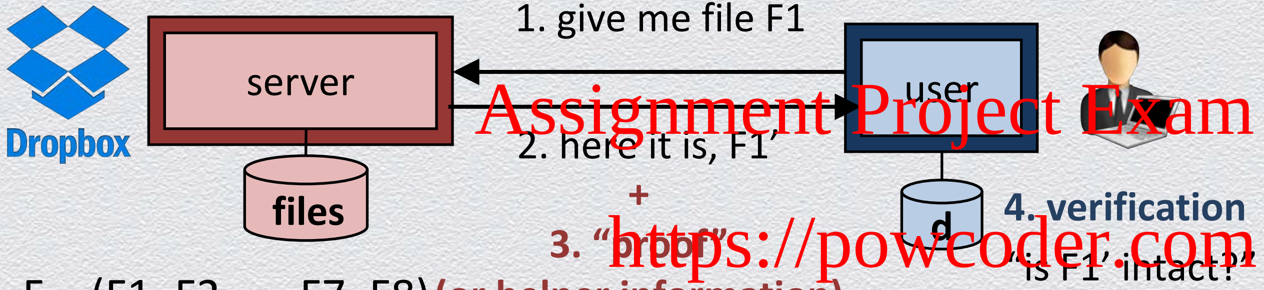


files = F = (F1, F2, …, F7, F8)

digest d is computed over all files

$|d| << |F|$

Securing your cloud storage optimally – data authentication



F = (F1, F2, ..., F7, F8) **(or helper information)**

goals

security: Step 4 is reliable test

efficiency:

- d is as small as possible
- proof is as small as possible
- verification is as fast as possible

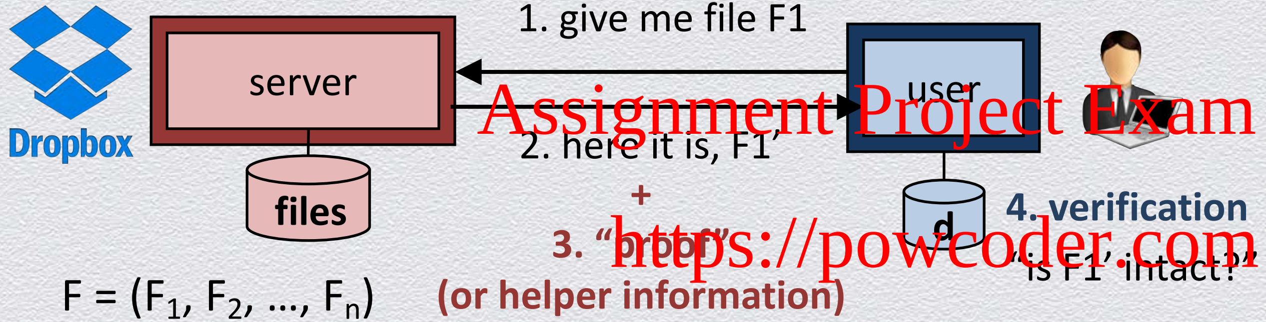
user has

- authentic digest d (locally stored)
- file F1' (to be checked)
- **proof** (to help checking integrity)

verification involves

- combine file F1' with the proof to re-compute candidate digest d'
- check if d' = d
- if yes, then F1 is intact; otherwise tampering is detected!

Hashing the files: Individually or as a whole?



$F = (F_1, F_2, \ldots, F_n)$

goals

security: Step 4 is reliable test

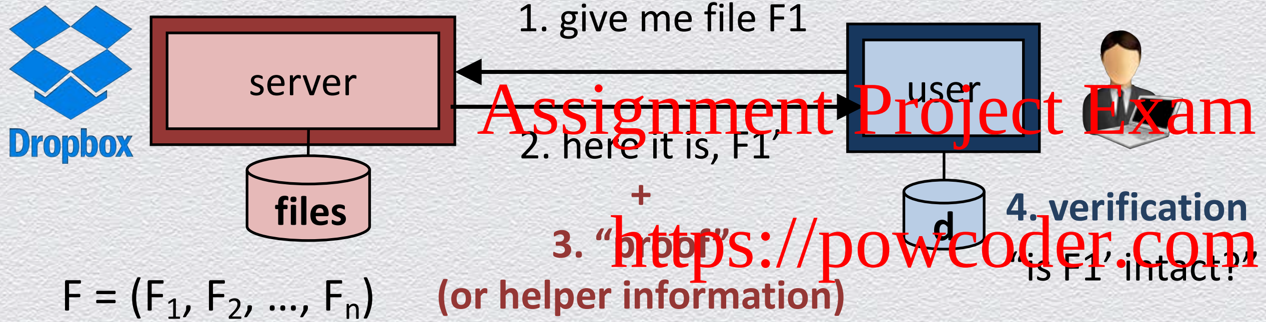
efficiency:

- d is as small as possible
- proof is as small as possible
- verification is as fast as possible

let $h_i = h(F_i)$, $1 \le i \le n$

$d = (h_1, h_2, \ldots, h_n)$ Vs. $d = h(h_1 || h_2 || \ldots || h_n)$ Vs. $d = h(F_1 || F_2 || \ldots || F_n)$

Hashing the files: In a chain or in a partition?



goals

security: Step 4 is reliable test

efficiency:

- d is as small as possible
- proof is as small as possible
- verification is as fast as possible

let $h_i = h(F_i)$, $1 \le i \le n$

$d = h_n$, $h_i = h(F_i || h_{i-1})$, $1 < i \le n$, and $h_1 = F_1$

$d = h_n$, $h_i = h(\mathbf{h(F_i)} || h_{i-1})$, $1 < i \le n$, and $h_1 = \mathbf{h(F_1)}$

Vs.

k disjoint subsets each of n/k size

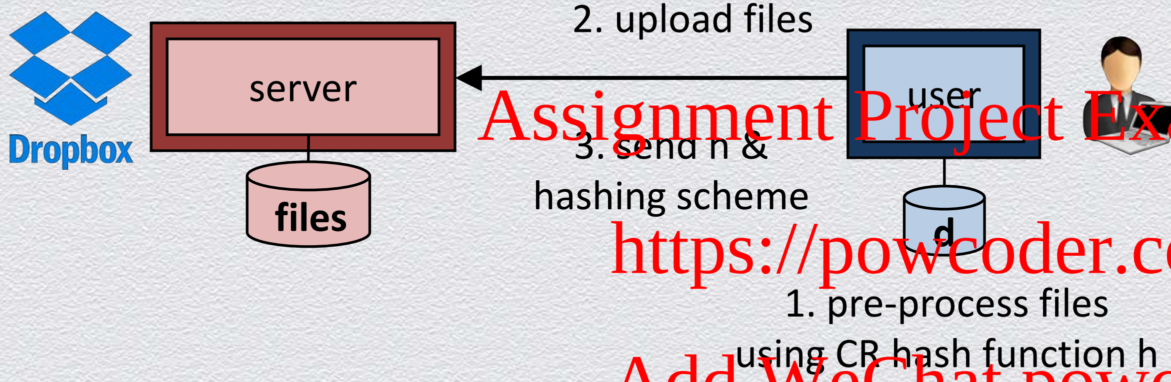
subsets & their digests are hashed in chains

Towards an optimal hashing scheme

Lessons learned

- files should better be individually hashed as $h_i = h(F_i)$, $1 \le i \le n$, over which the final digest should be computed
- for $k > 2$:
  - hashing k files in a hash chain is preferable to hashing files as a whole
  - hash chains over k objects (hash values/files) result in **unbalanced** verification costs
    - e.g., proof size/verification time are sensitive to a file's position in the hash chain
- hashing file subsets individually across a balanced partition of the files:
  - offers trade-offs between space and verification costs
  - allows for hierarchical hashing schemes

Hashing via the Merkle tree

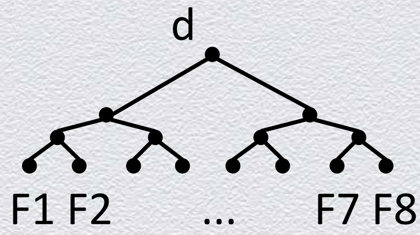


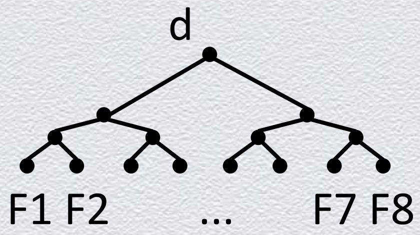
files = F = (F1, F2, ..., F7, F8)

digest d is computed over all files

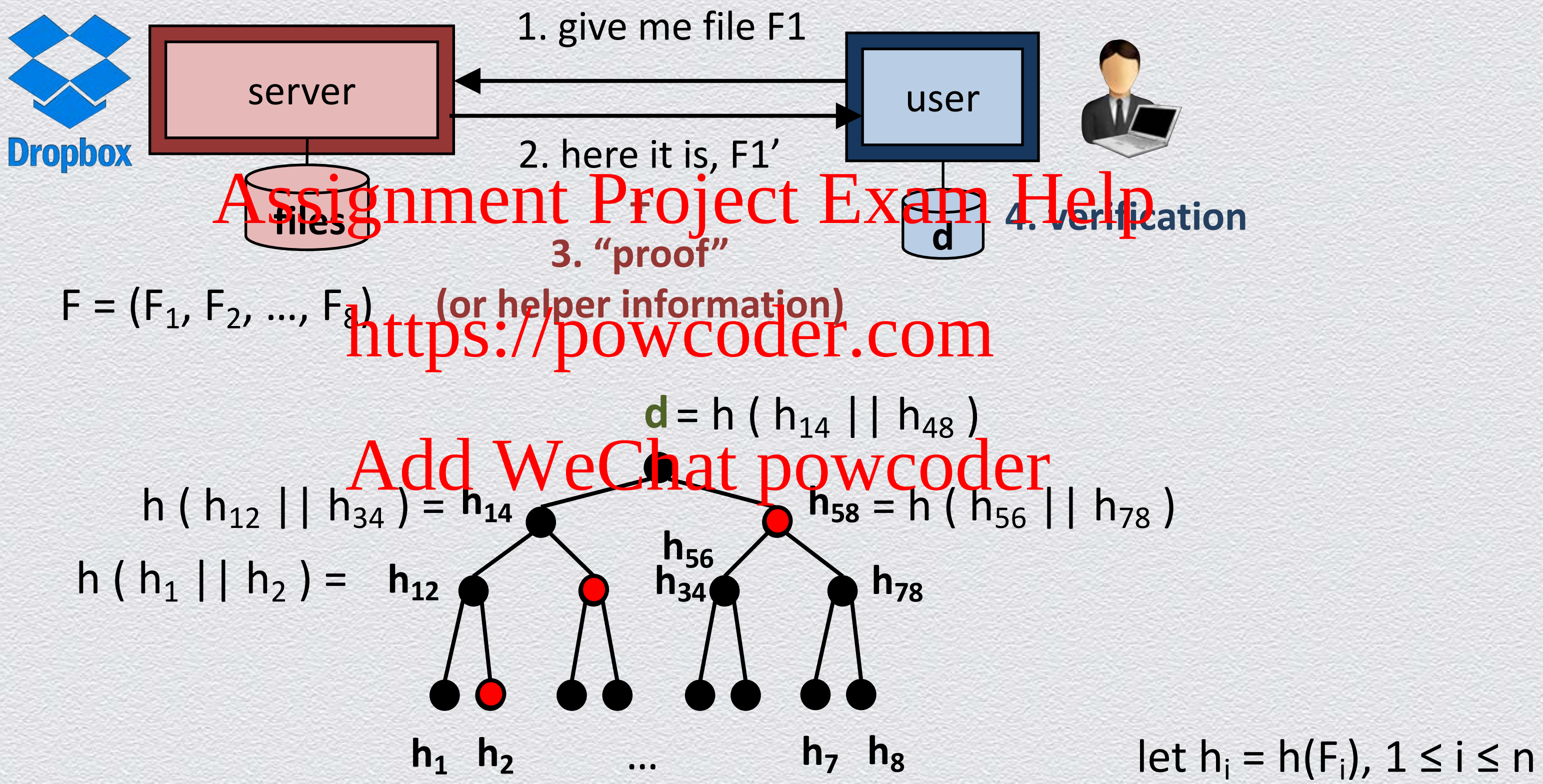
|d| << |F|

h: collision resistant iterative hash function (e.g., SHA-2)

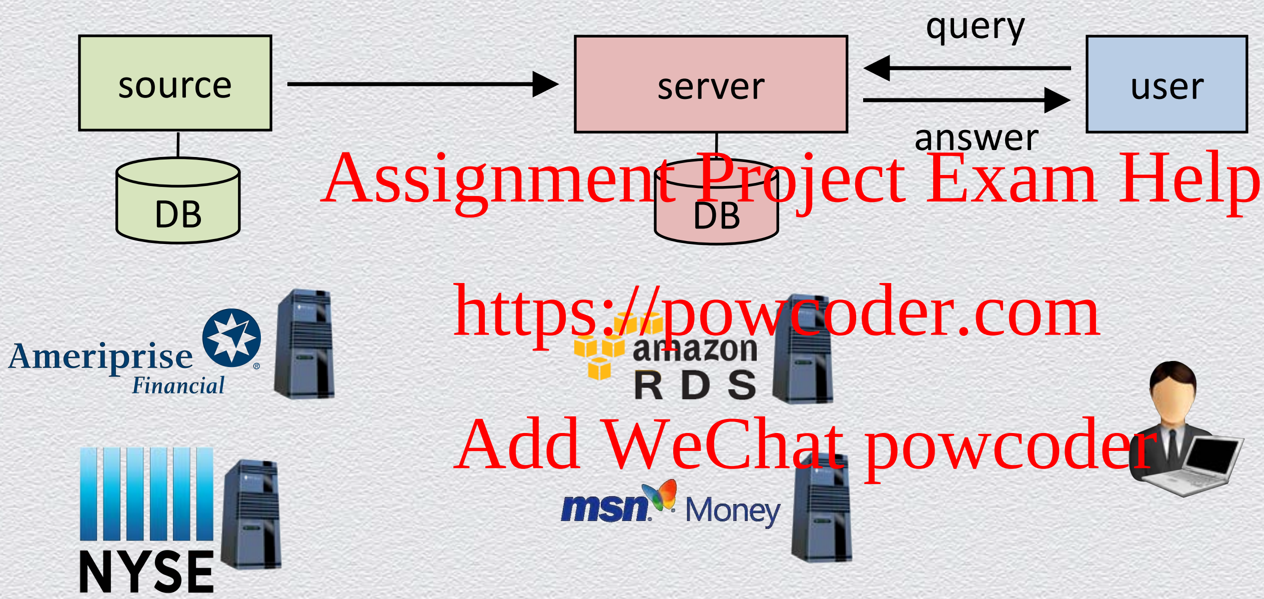




The Merkle tree



Generalization: DB-as-a-service authentication model

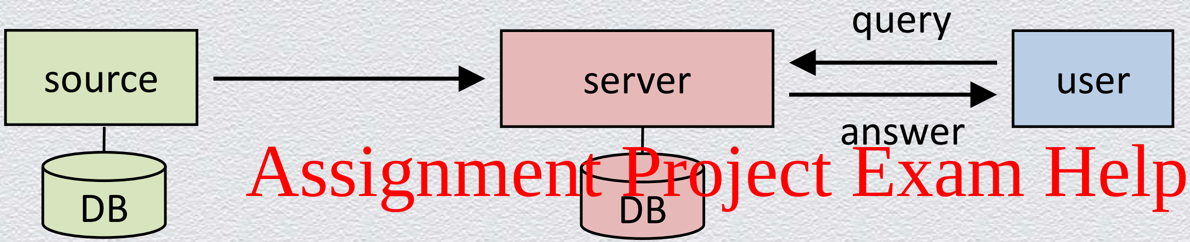


“is answer correct?”

cloud-based DB

portals, hubs

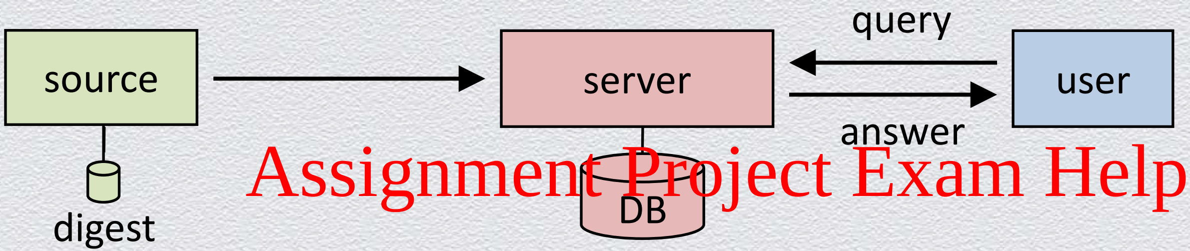
Generalization: DB-as-a-service authentication model



“is answer correct?”

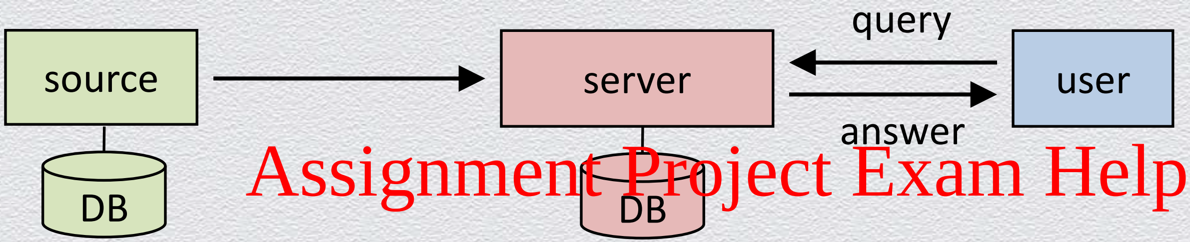
Internet protocols
(DNS, OCSP)

Generalization: DB-as-a-service authentication model



file hosting

DB-as-a-service authentication model

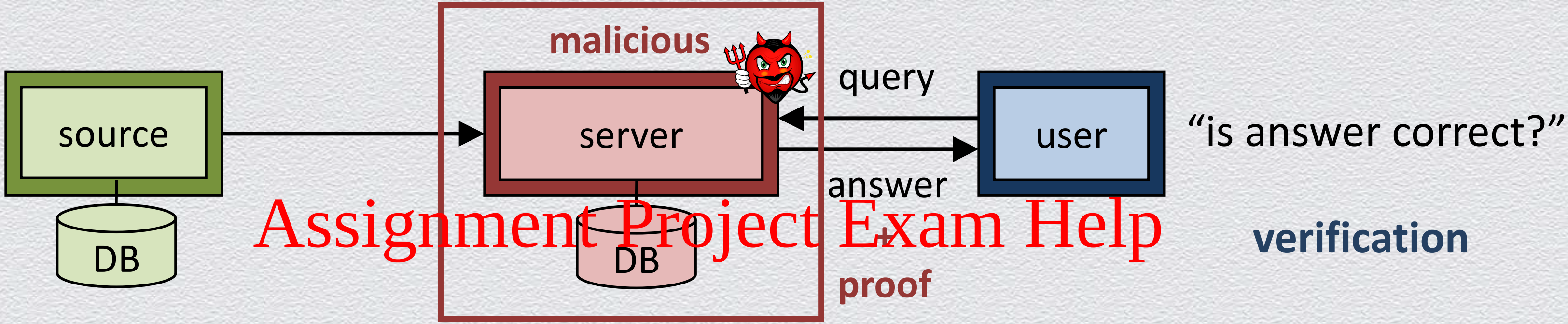


"is answer correct?"

Integrity checks offer authenticated queries

guarantees correct answer, as if coming directly from the source!

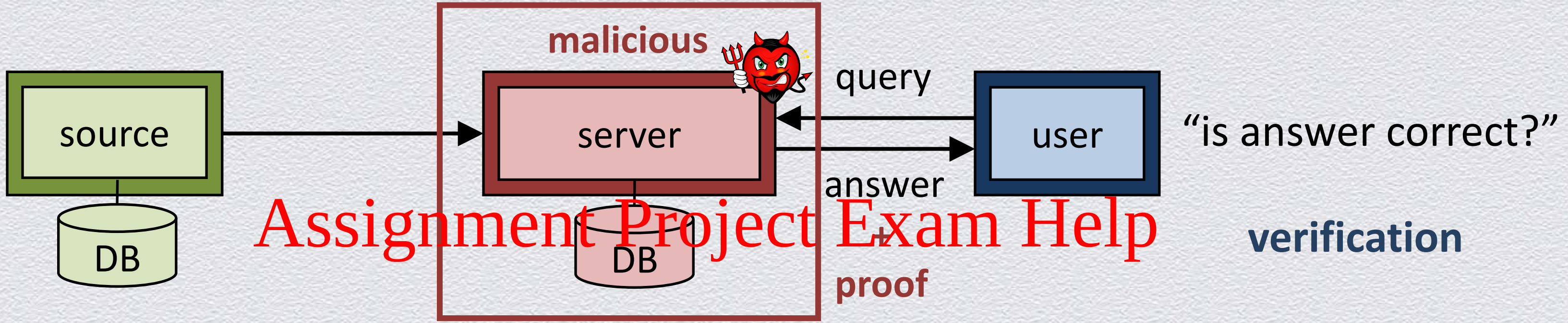
DB-as-a-service authentication model



Integrity checks offer authenticated queries

- crypto-based: harden data/computations to provide verifiable answers
- reliable: allow no false positives or negatives ⇨ crypto does its work!

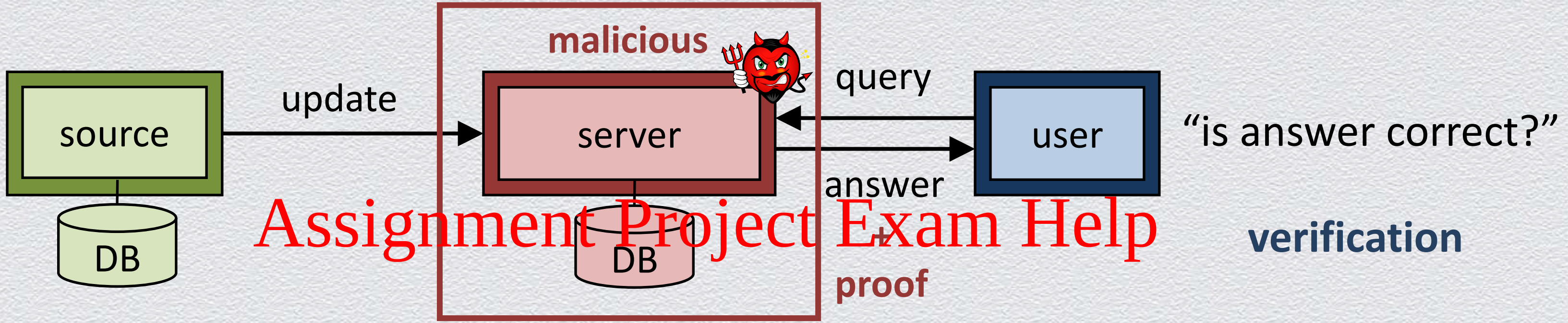
DB-as-a-service authentication model



Integrity checks offer authenticated queries

- crypto-based: harden data/computations to provide verifiable answers
- reliable: allow no false positives or negatives
- utility-preserving: practical & easy to adopt ⇨ minimize all overheads
  - fast times for query, verification
  - near total storage, low bandwidth usage for proof

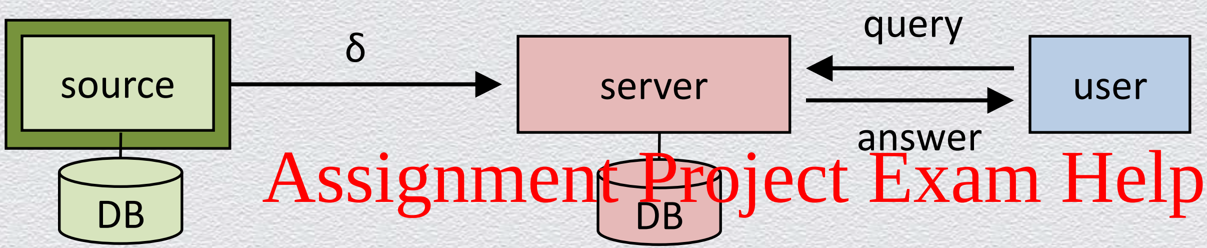
DB-as-a-service authentication model



Integrity checks offer authenticated queries

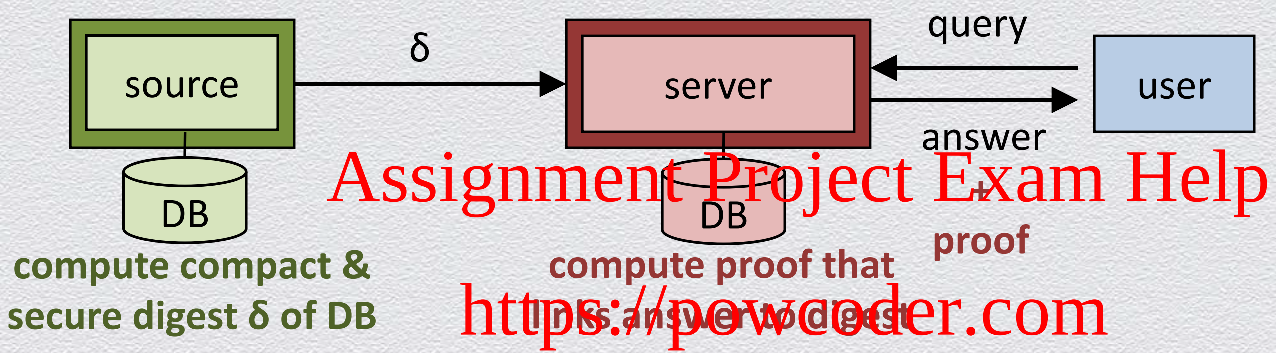
- crypto-based: harden data/computations to provide verifiable answers
- reliable: allow no false positives or negatives
- utility-preserving: practical & easy to adopt
  - fast times for query, verification, update
  - near total storage, low bandwidth usage for proof, update

Intro to authenticated querying (AQ101)



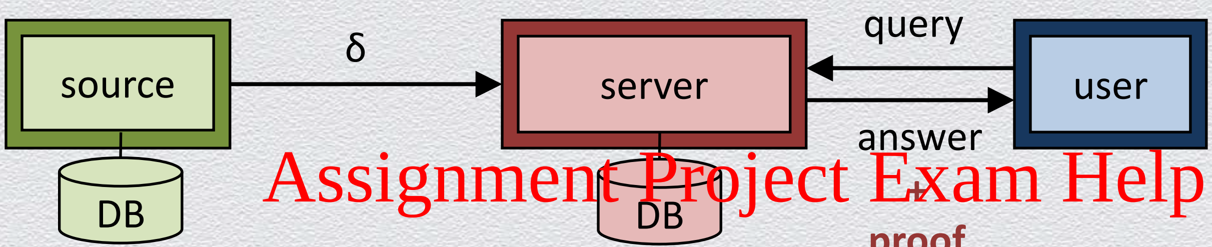
compute compact & secure digest δ of DB

Intro to authenticated querying (AQ101)



Intro to authenticated querying (AQ101)

Assumption: user possesses authentic δ



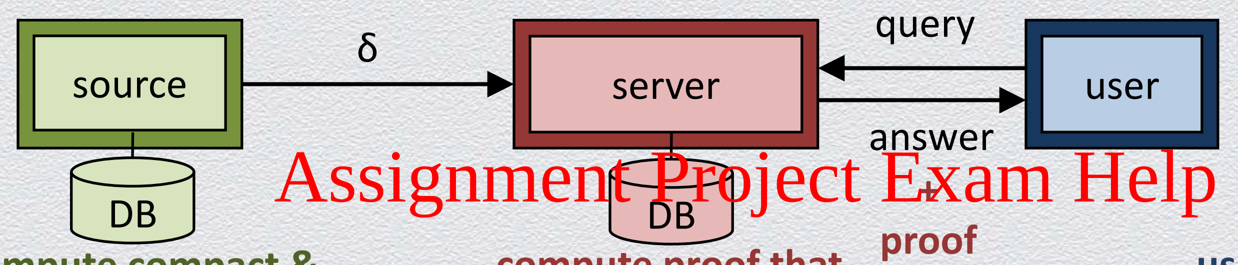
verification

compute compact & secure digest δ of DB

compute proof that links answer to digest

use proof to securely link answer to authentic digest

Intro to authenticated querying (AQ101)



compute compact & secure digest δ of DB

compute proof that links answer to digest

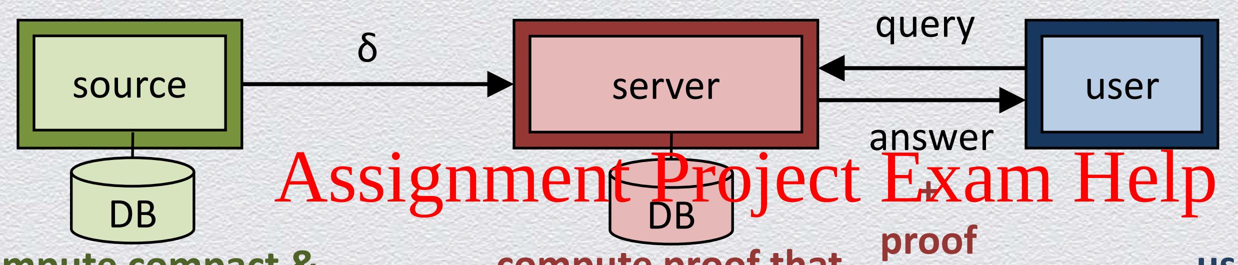
Assumption: user possesses authentic δ

easily removed using PKI, digital signatures

verification

use proof to securely link answer to authentic digest

Intro to authenticated querying (AQ101)



compute compact & secure digest δ of DB

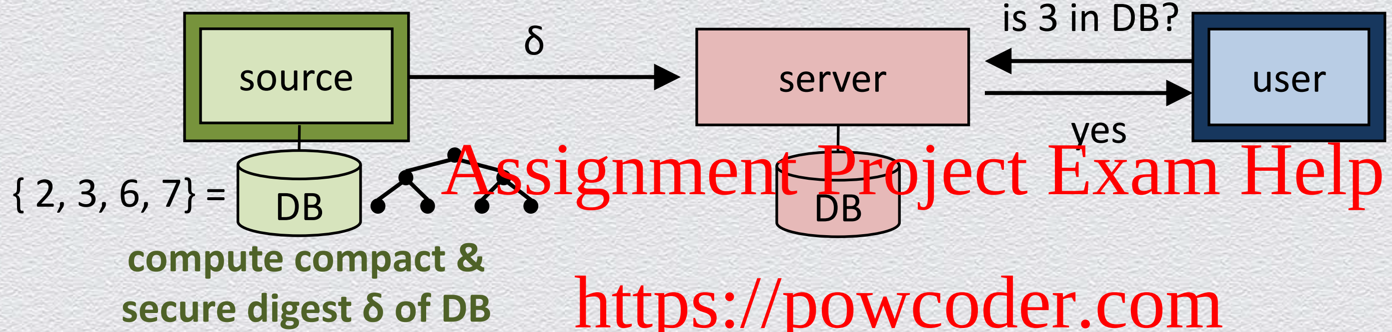
compute proof that links answer to digest

(given δ)
verification

use proof to securely link answer to authentic digest

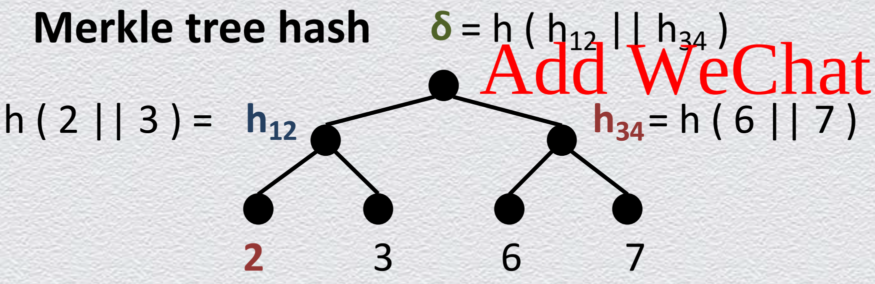
Using digital signatures to provide authentic digest δ

Example: Set membership – digest computation

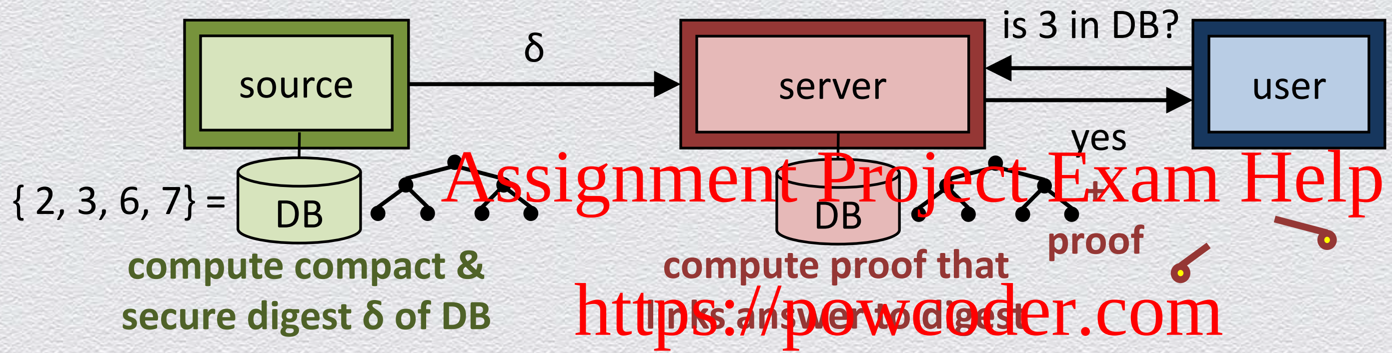


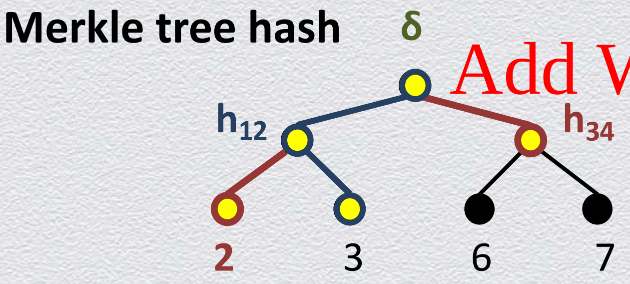
Merkle tree hash

h: collision resistant terative hash function (e.g., SHA-2)

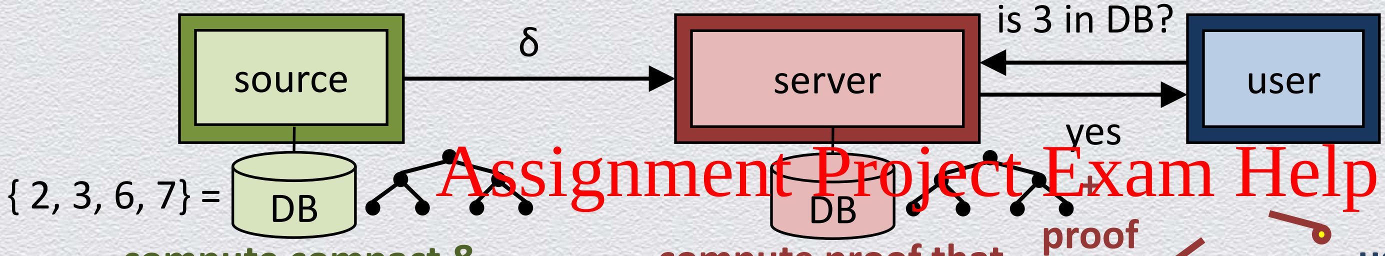


Example: Set membership – proof computation





Example: Set membership – proof verification



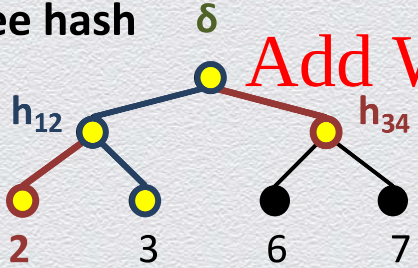
verification
(given δ)

compute compact & secure digest δ of DB

compute proof that links answer to digest

use proof to securely link answer to authentic digest

Merkle tree hash



i.e.:
recompute h12
recompute δ
compare against given δ