CS306: Introduction to IT Security

Fall 2020

Lab

Instructor: **Nikos**

October 8 & 15, 2020

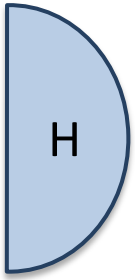
STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Cryptographic hash functions

Basic cryptographic primitive

- maps **"objects"** to a **fixed-length** binary strings
- core security property: mapping **avoids collisions**
  - **collision**: distinct ob
  - although collisions

input
arbitrarily long string



output
short digest, fingerprint, "secure" description

hash value (H(x) = H(y))

to find

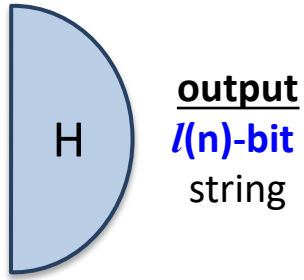
Important role in modern cryptography

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

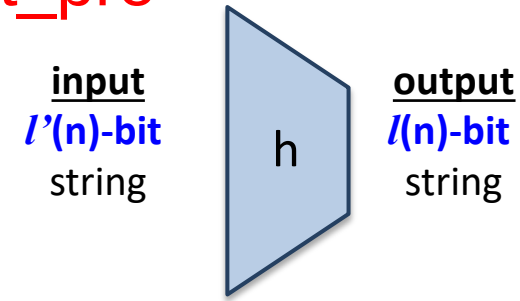
Hash & compression functions

Map messages to short digests

- a **general** hash function H() maps
  - a message of an **ar**

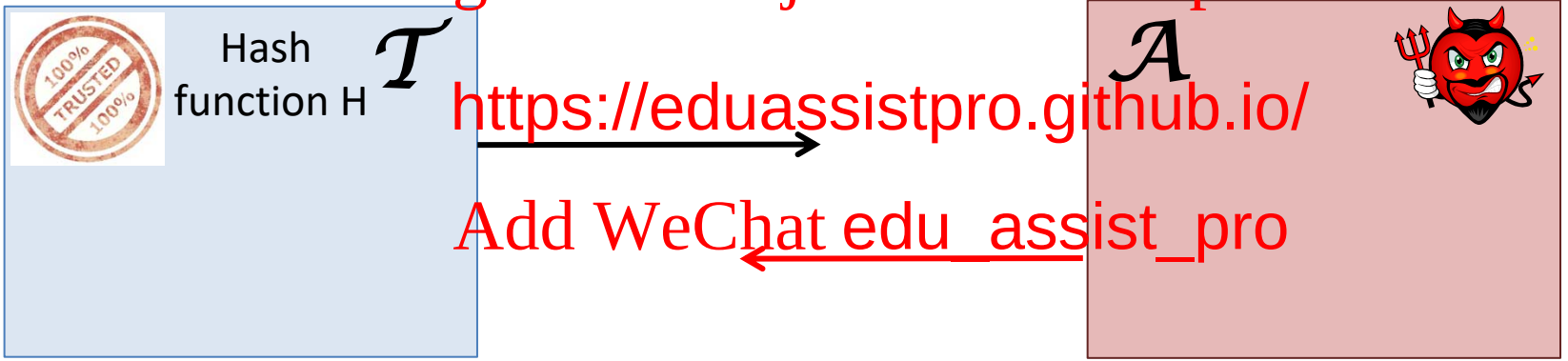


- a **compression** (hash) function h() maps
  - a **long** binary string to a **shorter** binary string
  - an ***l'*(n)-bit string** to a ***l*(n)-bit string**, with ***l'*(n) > *l*(n)**



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 $\rightarrow$ 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 resistan**
  - if given a **uniform** x 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

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 collis pression 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: $\mathcal{M} = \{x : |x| < 2^n\} \rightarrow \{0,1\}^{n}$, defined as

Merkle-Damgård desig

- H(x) is computed by a 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 **$h(x||k) = F_k(x)$ 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 H hms standardized by NIST)
  - output 160 bits, co ance
  - **broken** in 2017 by r
- 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

Generic attacks against cryptographic hashing

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

- **brute-force** attack
  - for each string x in the domain
    - compute and rec
    - if h(x) equals a p
      halt and output collision on x ≠ 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 "**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

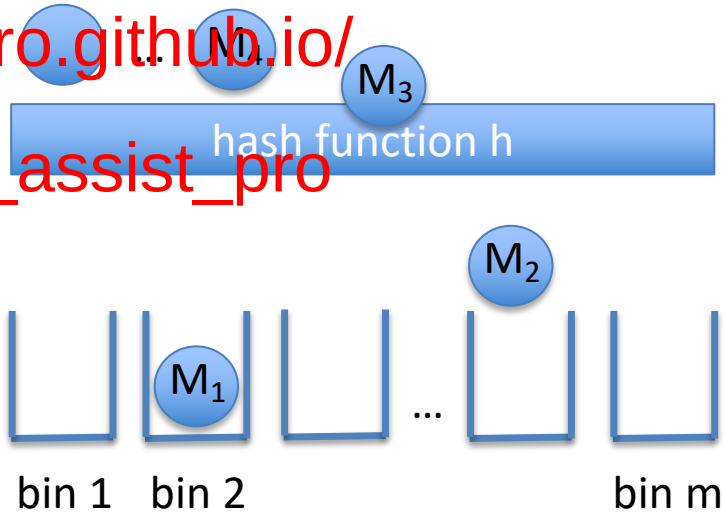
m=365

k

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 mes
  - m bins: number of possible hash values
  - independent & random 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 1 - (i - 1)/m

- the probability F_k that af ame bin is:

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

- by the standard approximation $1 - x \approx e^{-x}$: $F_k \approx e^{-(1/m + 2/m + 3/m + \ldots + (k-1)/m)} = e^{-k(k-1)/2m}$
- 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

n m

- 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" **a collision** is found
- **birthday** attack
  - evaluate h on (much) **fewer** distinct input **than 2^n** 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**

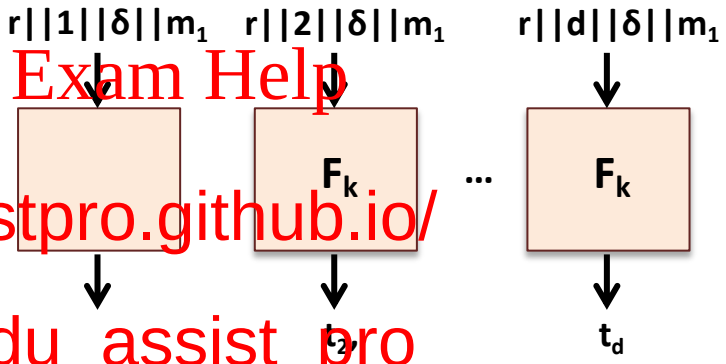
lications of

o cryptography

Hash functions enable efficient MAC design!

Back to problem of designing **secure MAC for messages of arbitrary lengths**

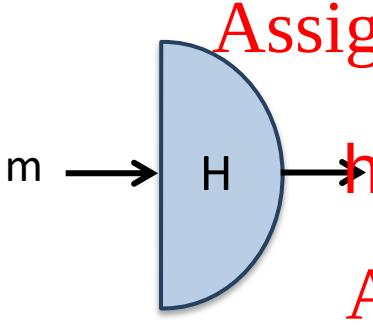
- so far, we have seen two solutions
  - block-based "tagging"
    - based on PRFs
    - inefficient
  - CBC-MAC
    - also based on PRFs
    - more efficient

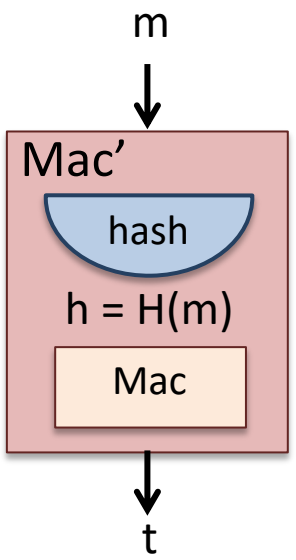


[1] Hash-and-MAC: Design

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

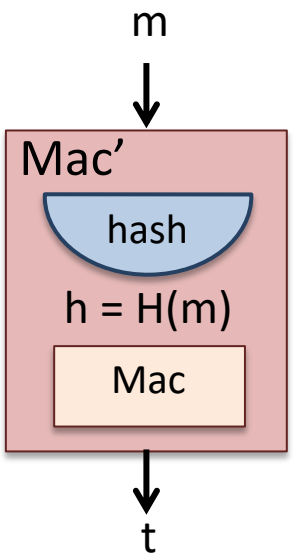
[1] Hash-and-MAC: Security

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 as good as aut **elf**!



[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:

$$Mac_k(m) = \mathbf{H}(k||\mathbf{m})$$

- 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:

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

- intuition: **instantiation of hash & sign paradigm**
- two layers of hashing
  - **upper layer**
    - y = H((k ⊕ ipad) || m)
    - y = H'(m), i.e., "hash"
  - **lower layer**
    - t = H ((k ⊕ 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

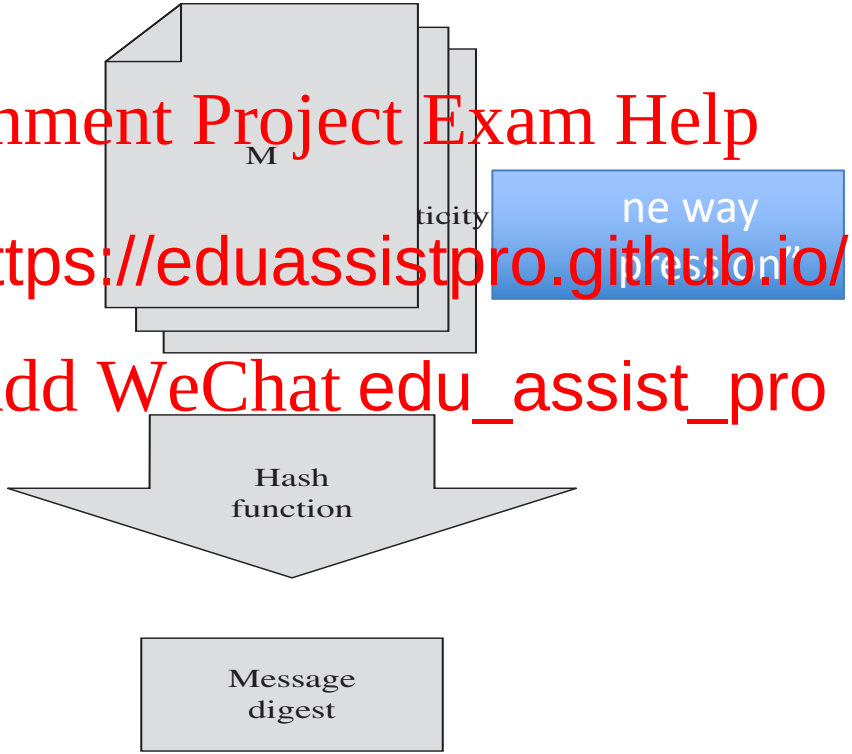
Recall: Weaker security notions

Given a hash function H: X $\rightarrow$ 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 resistan**
  - if given a **uniform** x 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



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 rand
- Every time Bob **reads** a file f_i, he also reads ies 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
- solution
  - Alice, Bob, Charlie submit **hashes** h(A), h 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
- 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 s r)
  - e.g., commit(x, r) =
  - **hiding property**: you cannot see throug nvelop
- 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 randoml ness and) commitment C_i of x_i
  - challenger gives C_b ind 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 commit(x_1, r_1) = commit(x_2, r_2)

Example 2: Fair decision via coin flipping

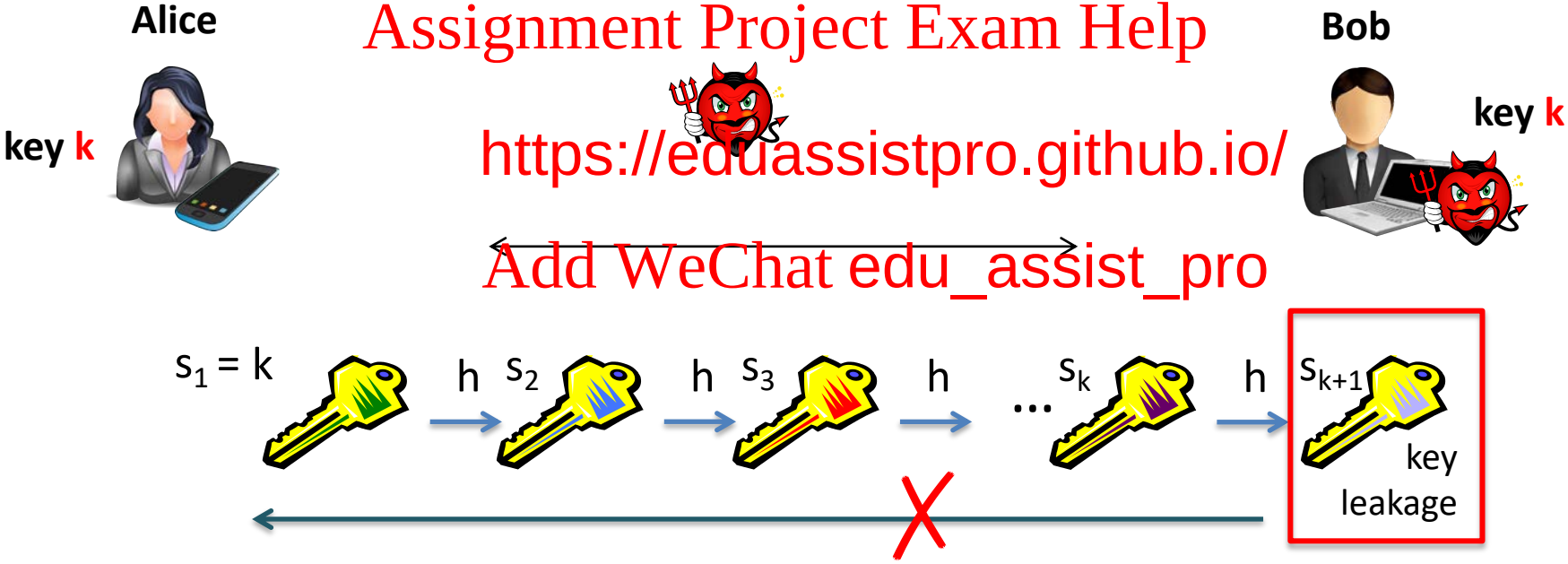
Alice is to “call” the coin flip and Bob is to flip the coin

- to decide who will do the dishes…
- problem: Alice may change her mind, Bob may skew the result
- protocol
  - Alice "calls" the coin f her call
  - Bob flips the coin and
  - Alice reveals what she committed to
  - Bob verifies that Alice's call matches her com
  - 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' adicts 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 the digest of your files agains 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 nodes (e.g., their IP addresses) are into identifiers in a hash range
- Wh dded in the system it is can peer nodes that are resp es 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 d
  e.g., think of how many us
  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.
  - s ready to upload a new file to the
    's digest is first uploaded.
  - cks to find a possible duplicate,
    ointer 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 u access to some computing
- 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 passwords are stored at the server in the clear, an attacker can steal the password file after the authentication server – this type pens routinely nowadays…
- involved having the server s 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

Plaintext **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

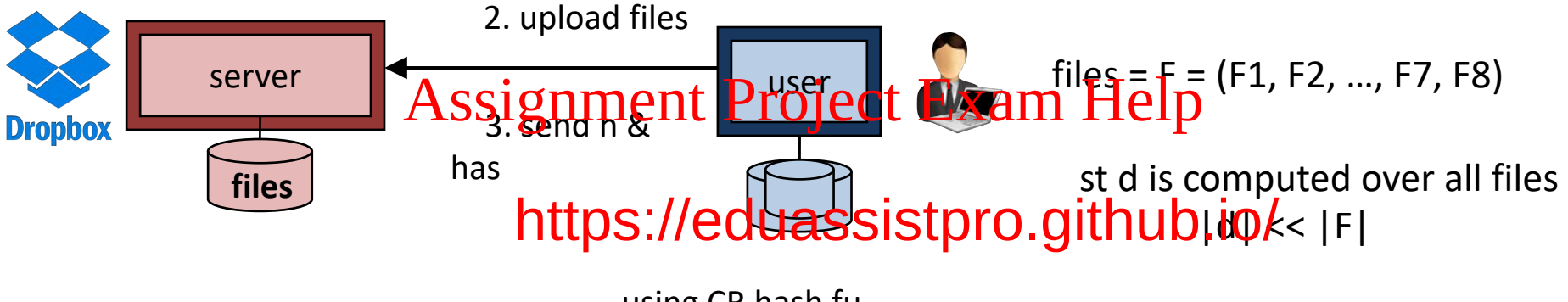
Signing message M

- let h be a cryptogra tting (n, d, e)
- compute signature
- send σ, M

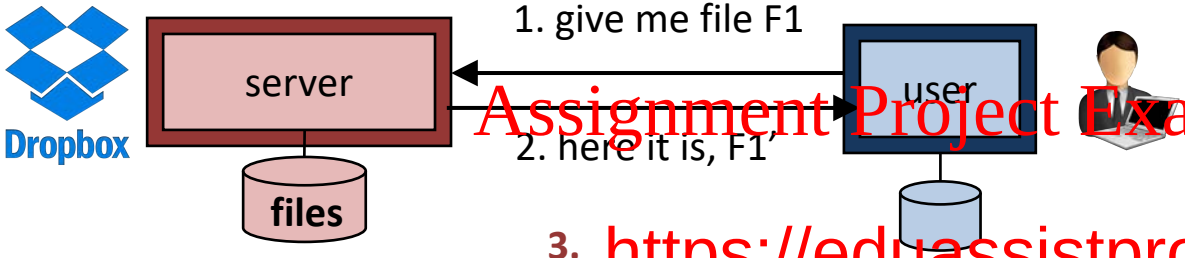
Verifying signature σ

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

Securing your cloud storage optimally – setup



Securing your cloud storage optimally – data authentication



F = (F1, F2, …, F7, F8) **(or helpe**

goals

security: Step 4 is reliable test

efficiency:

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

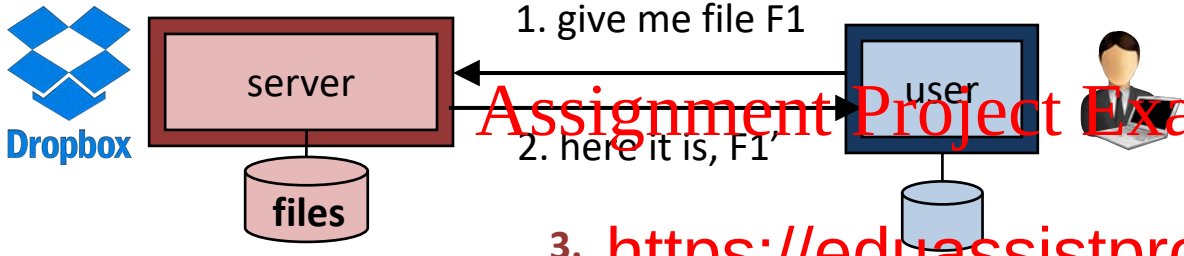
user has

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

v s

- 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?



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

let $h_i =$

goals

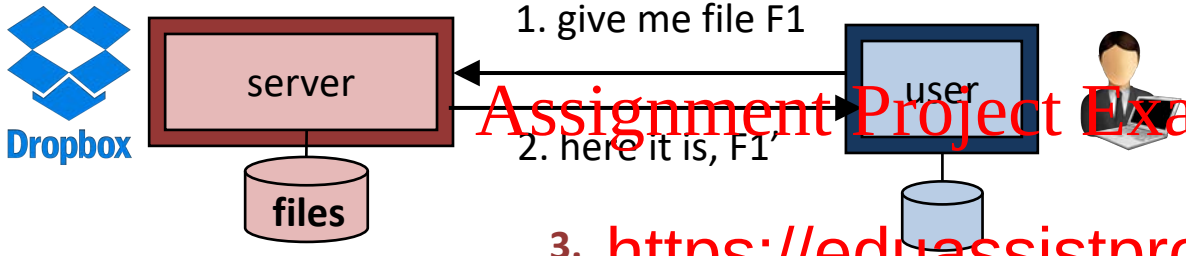
security: Step 4 is reliable test

efficiency:

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

$d = (h_1, h_2, \ldots, h_n)$ 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?



3.

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

let h_i =

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

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

Vs.

k disjoint subsets each of n/k size

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

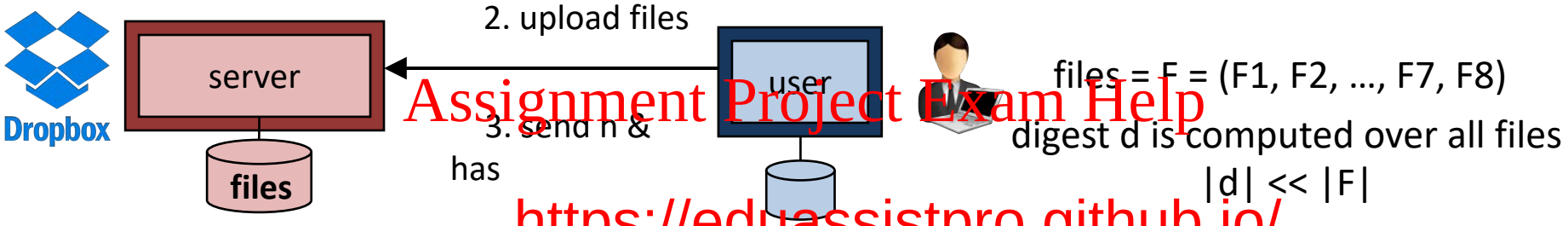
subsets & their digests are hashed in chains

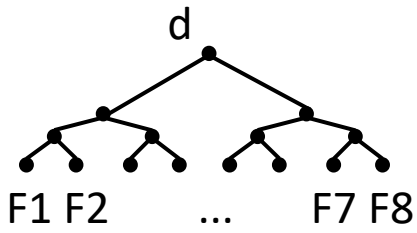
Towards an optimal hashing scheme

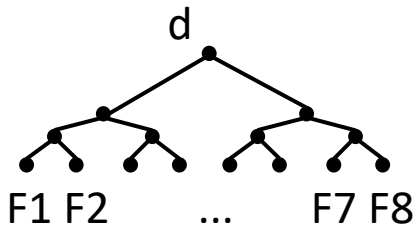
Lessons learned

- files should better be individually hashed as $h_i = h(F_i)$, $1 \leq i \leq n$, over which the final digest should be computed
- for k > 2:
  - hashing k files in a has s a whole
  - hash chains over k objects (hash values/files) **ed** verification costs
    - e.g., proof size/verification time are sensit tion 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

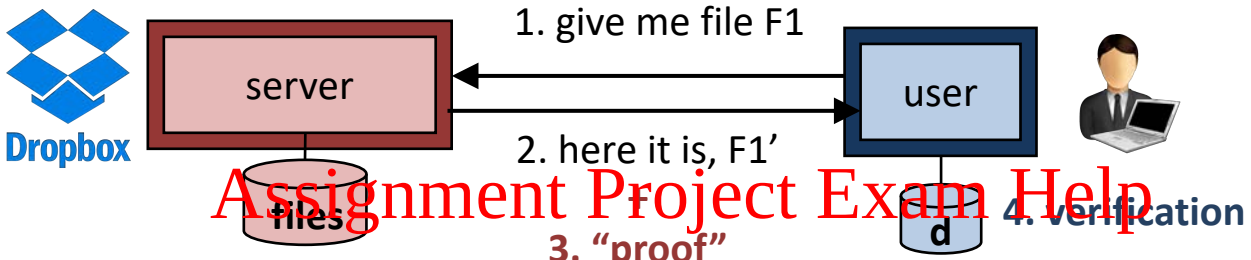
Hashing via the Merkle tree



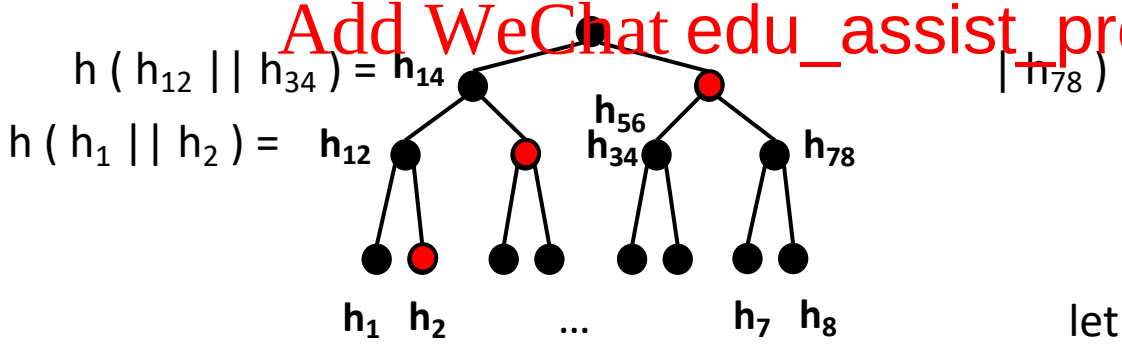




The Merkle tree

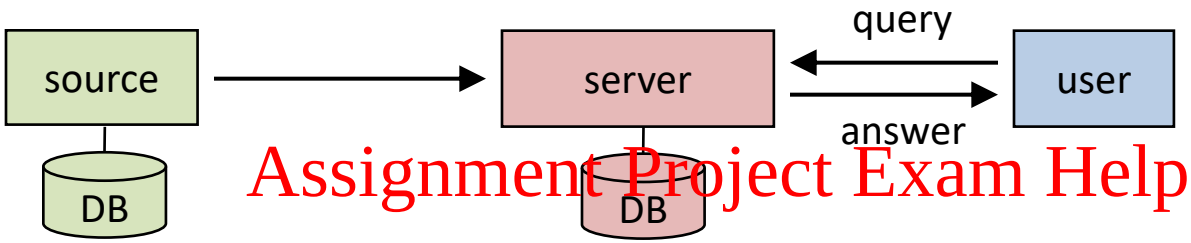


F = (F_1, F_2, …, F



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

Generalization: DB-as-a-service authentication model

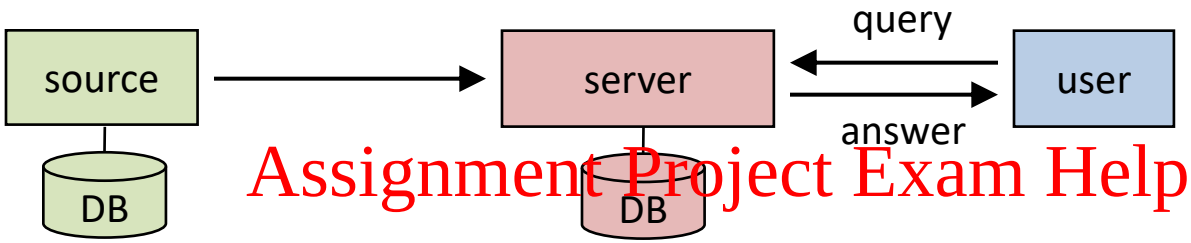


"is answer correct?"

cloud-based DB

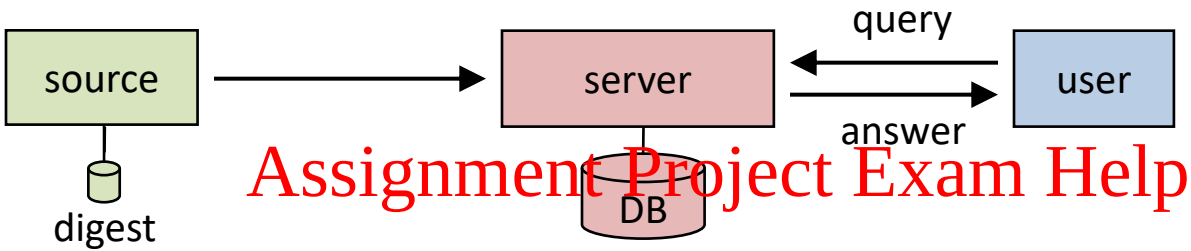
portals, hubs

Generalization: DB-as-a-service authentication model



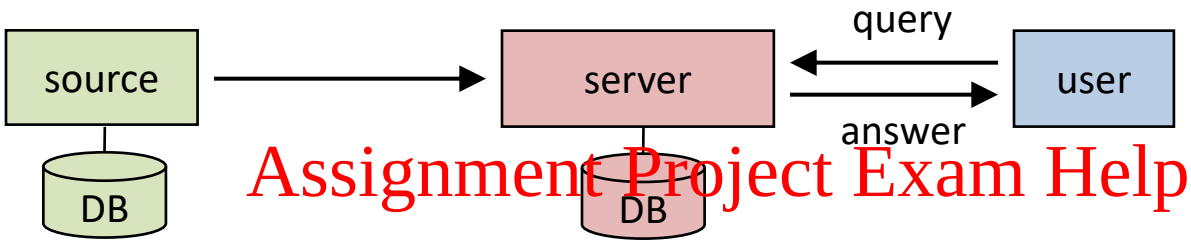
Internet protocols
(DNS, OCSP)

Generalization: DB-as-a-service authentication model



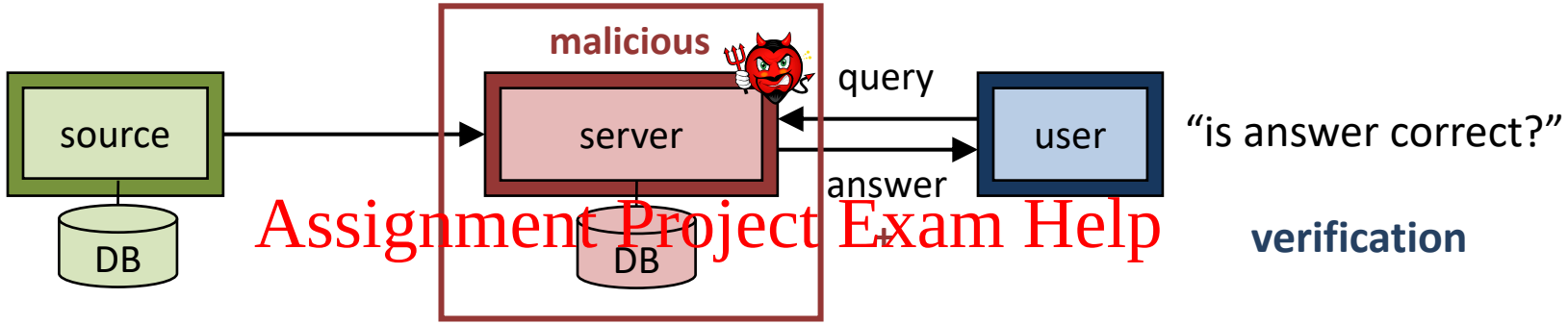
file hosting

DB-as-a-service authentication model



Integrity checks offer a guarantees correct answer, as if
g directly from the source!

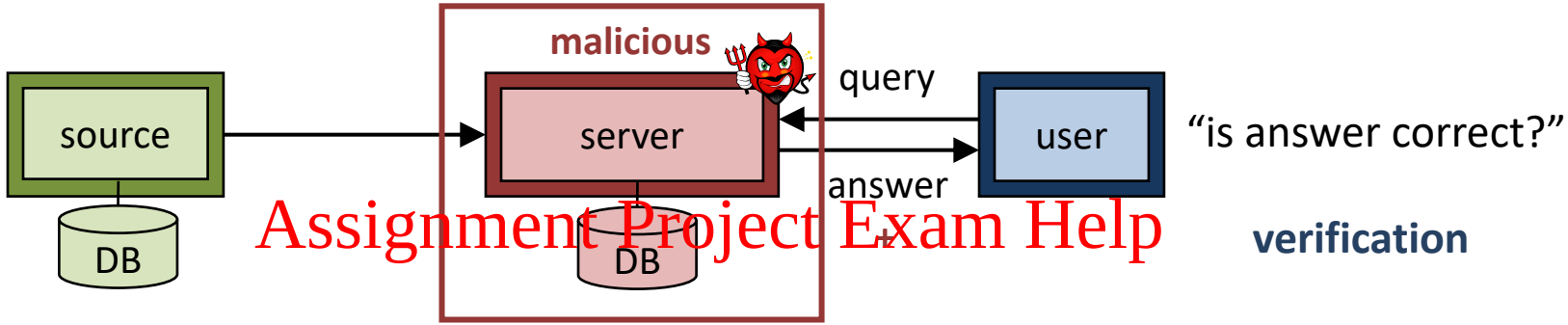
DB-as-a-service authentication model



Integrity checks offer a

- crypto-based: harden data/computations t ble answers
- reliable: allow no false positives or negatives pto does its work!

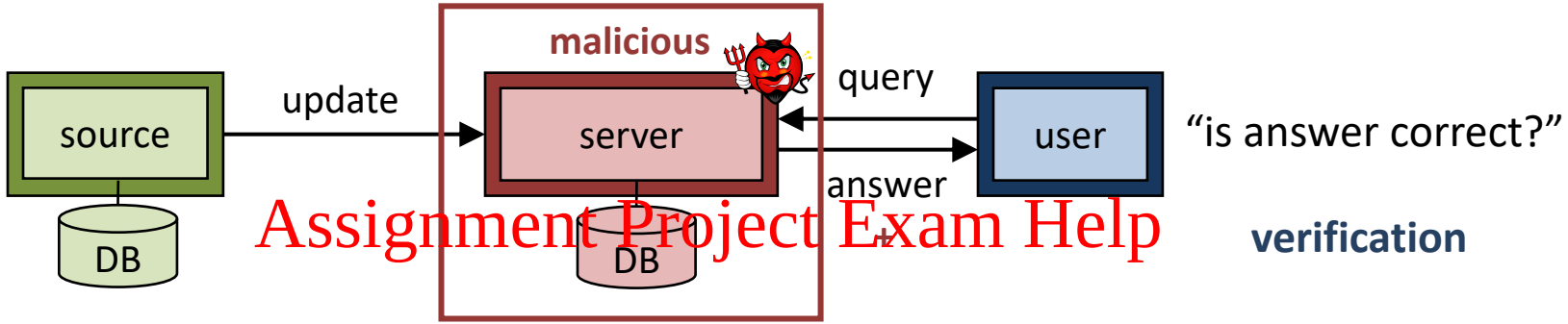
DB-as-a-service authentication model



Integrity checks offer a

- crypto-based: harden data/computations t ble 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

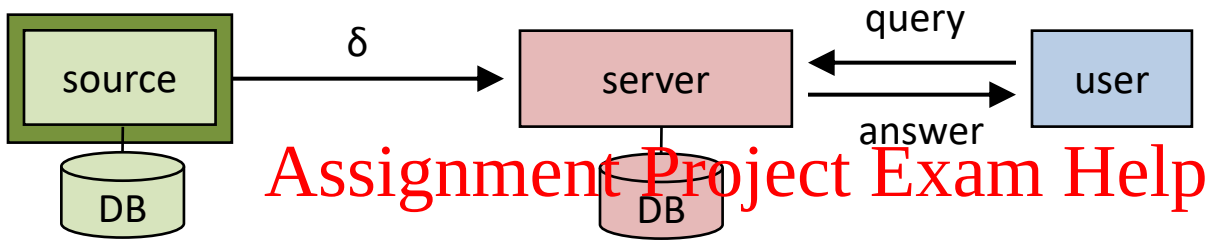
DB-as-a-service authentication model



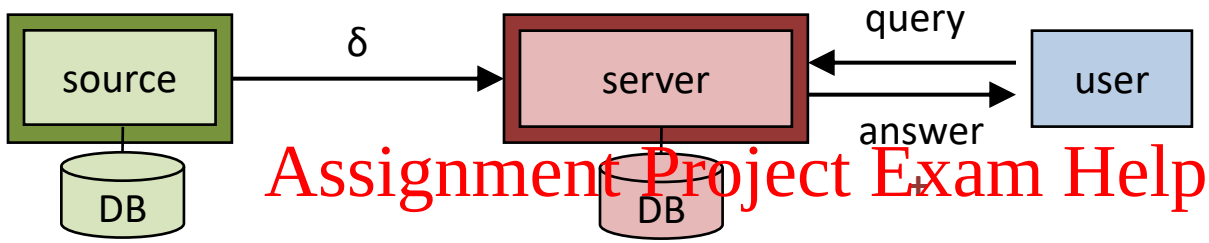
Integrity checks offer a

- crypto-based: harden data/computations t ble 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

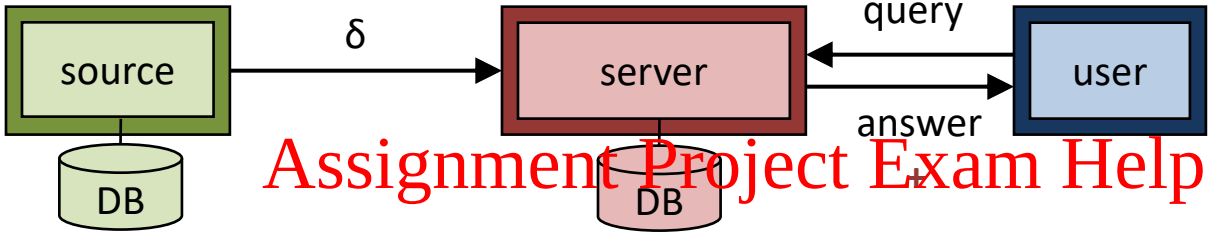
Intro to authenticated querying (AQ101)



Intro to authenticated querying (AQ101)



Intro to authenticated querying (AQ101)



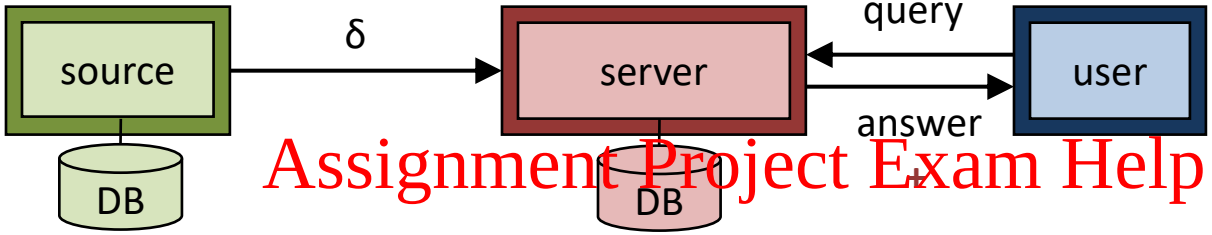
Assumption: user possesses authentic δ

compute compact & secure digest δ of DB

verification

use proof to securely link answer to authentic digest

Intro to authenticated querying (AQ101)



Assumption: user possesses authentic δ

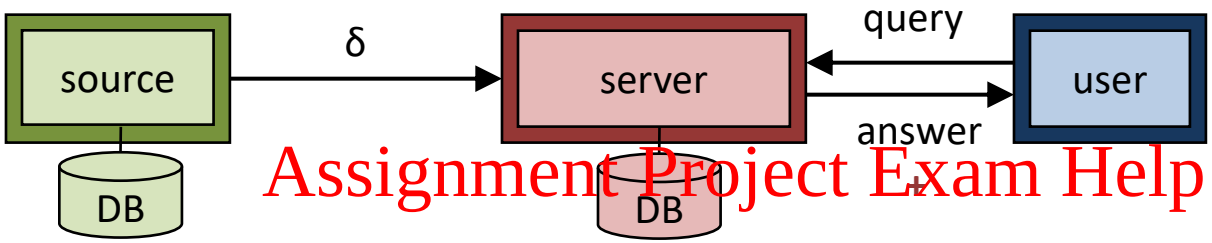
easily removed using PKI, digital signatures

compute compact & secure digest δ of DB

verification

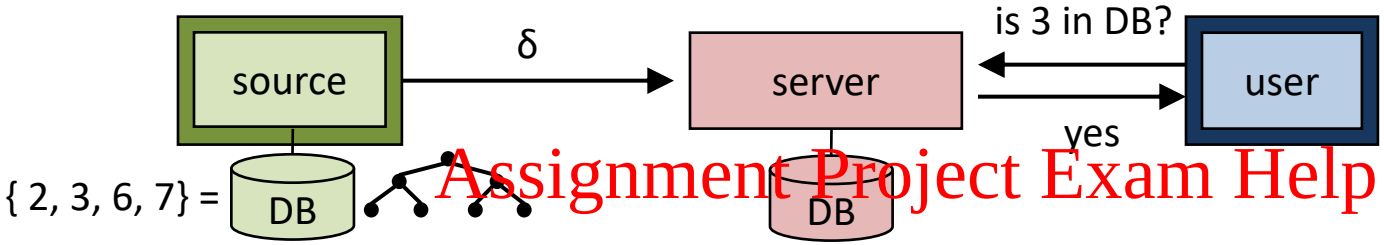
use proof to securely link answer to authentic digest

Intro to authenticated querying (AQ101)



Using digital signatures to provide au

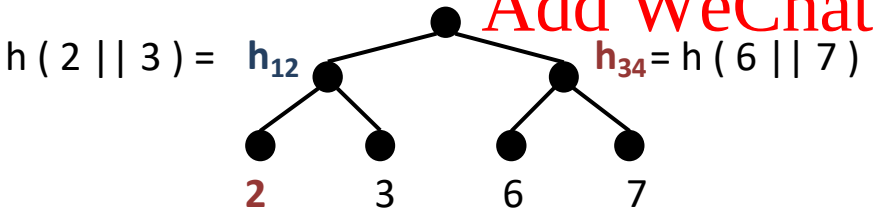
Example: Set membership – digest computation



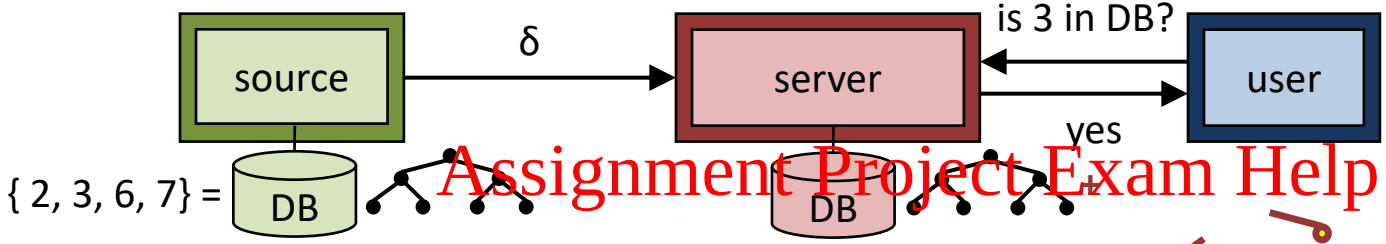
compute compact & secure digest δ of DB

Merkle tree hash $\delta = h(h_{12} || h_{34})$

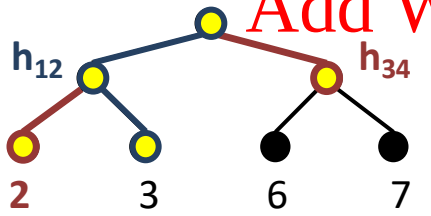
terative hash function (e.g., SHA-2)



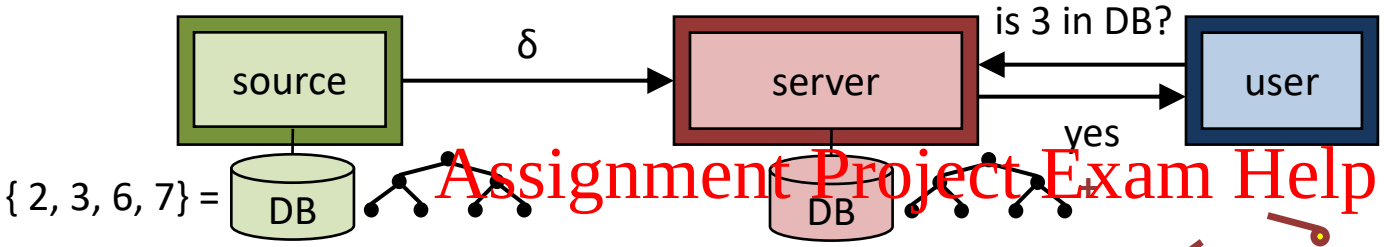
Example: Set membership – proof computation



Merkle tree hash δ



Example: Set membership – proof verification

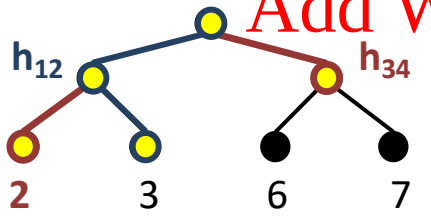


verification
(given δ)

compute compact & secure digest δ of DB

use proof to securely link answer to authentic digest

Merkle tree hash δ



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