https://brown-csci1660.github.io

CS1660: Intro to Computer Systems Security
Spring 2026

Lecture 7: Authentication

Instructor: **Nikos Triandopoulos**

February 12, 2026

BROWN

CS1660: Announcements

- Course updates
  - Project 1 “Cryptography” is due next Thursday
  - HW 1 is going out tomorrow
  - (Tentative) Exams dates
    - Midterm exam: March 12
      - Covering primarily: Cryptography and Web Security
    - Final exam: April 28 (reading period) or May 12 (exam date)
      - Covering primarily: OS Security and Network Security

Last class

- Cryptography
  - Symmetric-key encryption in practice
    - Computational security, pseudo-randomness
    - Stream & block ciphers, modes of operations for encryption, DES & AES
    - Introduction to modern cryptography
  - Integrity & reliable communication
    - Message authentication codes (MACs)
    - Authenticated encryption
    - Cryptographic hash functions

Today

- Cryptography
  - Integrity & reliable communication
    - Message authentication codes (MACs)
    - Authenticated encryption
    - Cryptographic hash functions
    - Applications of cryptographic hash functions
    - User authentication: something you know, are, have
    - Password security and cracking

*On message authentication

Recall: Approach in modern cryptography

Formal treatment

- **fundamental notions** underlying the **design & evaluation** of crypto primitives

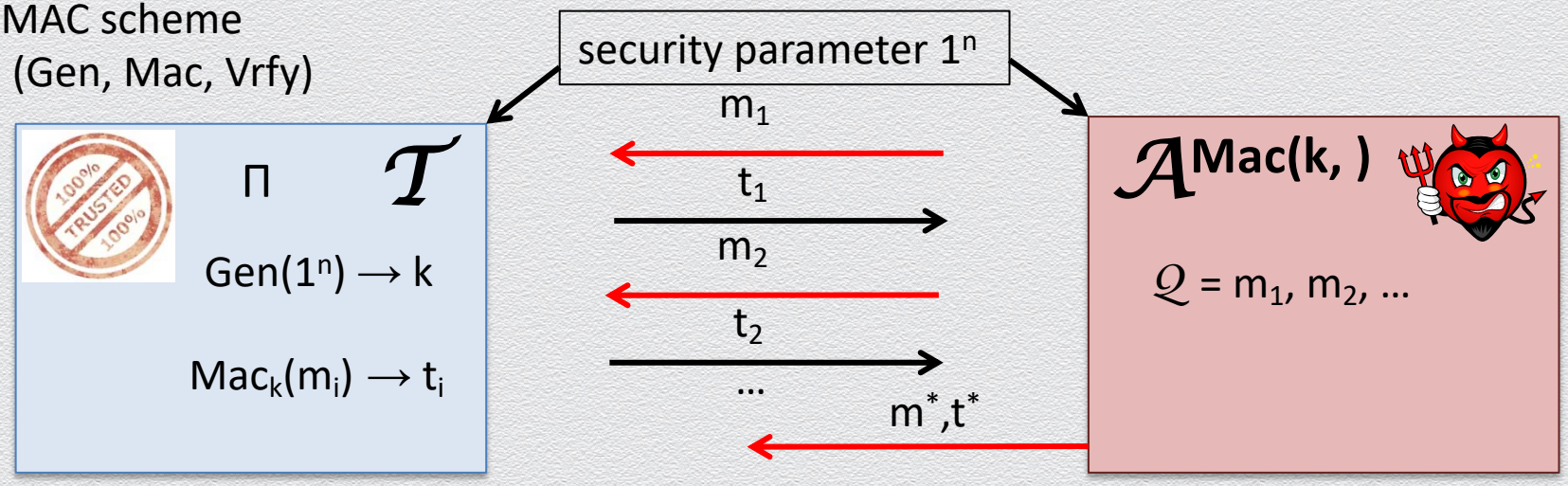
Systematic process

- A) **formal definitions** (what it means for a crypto primitive to be “secure”?)
- B) **precise assumptions** (which forms of attacks are allowed – and which aren’t?)
- C) **provable security** (why a candidate instantiation is indeed secure – or not)?

Computational MAC security

Game Mac-forge$_{\mathcal{A},\Pi}$(n) = 1 iff 1. $Vrfy_k(m^*,t^*) = 1$ &
2. m^* not in $\mathcal{Q}$

MAC scheme
$\Pi = (Gen, Mac, Vrfy)$



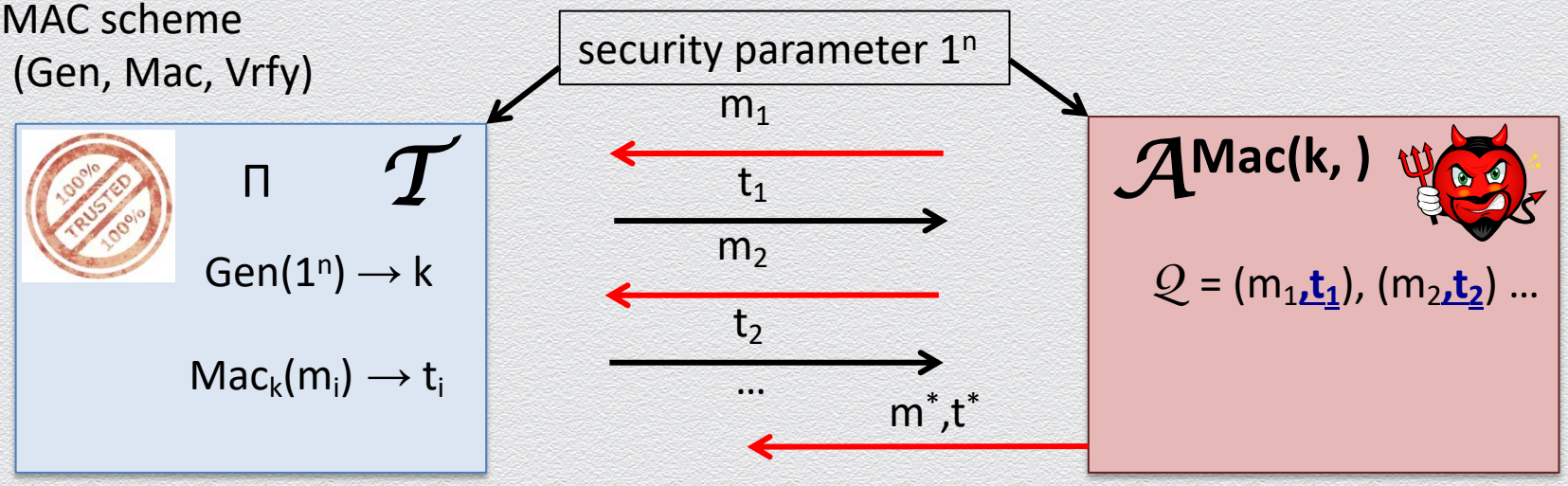
We say that Π is **secure** if for all PPT $\mathcal{A}$, there exists a negligible function negl so that

$$Pr[\ Mac\text{-}forge_{\mathcal{A},\Pi}(n) = 1\ ] \leq negl(n)$$

Strong MAC

Game Mac-sforge$_{\mathcal{A},\Pi}$(n) = 1 iff 1. $Vrfy_k(m^*,t^*) = 1$ &
2. (m^*,t^*) not in $\mathcal{Q}$

MAC scheme
$\Pi = (Gen, Mac, Vrfy)$



We say that Π is **strongly secure** if for all PPT $\mathcal{A}$, there exists a negligible function negl so that

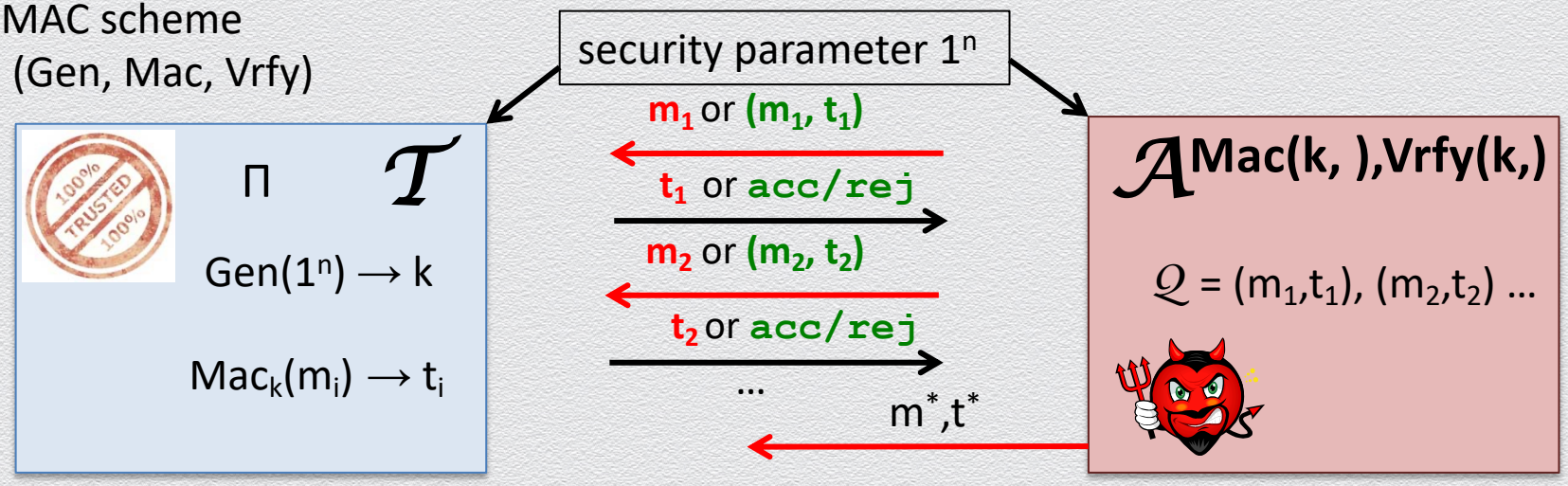
$$Pr[\ Mac\text{-}sforge_{\mathcal{A},\Pi}(n) = 1\ ] \leq negl(n)$$

(Strong) MAC w/ verification queries

Game Mac-s<u>V</u>forge$_{\mathcal{A},\Pi}$(n) = 1 iff 1. $Vrfy_k(m^*,t^*) = 1$ &
2. (m^*,t^*) not in $\mathcal{Q}$

MAC scheme
Π = (Gen, Mac, Vrfy)

security parameter 1^n



We say that Π is **strongly V-secure** if for all PPT $\mathcal{A}$, there exists a negligible function negl so that

$$Pr[\text{ Mac-s}\underline{V}\text{forge}_{\mathcal{A},\Pi}(n) = 1] \leq negl(n)$$

Verification queries Vs. timing attacks on MAC verification

In game Mac-sVforge$_{\mathcal{A},\Pi}$(n)

- queries to oracle Verf$_k$() return `acc/rej` (i.e., a **single bit**)

In real life

- implicit tag verification is **feasible** (e.g., by detecting a difference in verifier's behavior)
- but also an attacker may receive **more than this 1-bit info** via other "**side channels**"
  - Vrfy(m,t) of a **canonical** MAC returns `acc` only if t = Mac$_k$(m)
  - if implemented using `strcmp`, then comparison occurs **byte by byte until first mismatch**
  - thus, the time to return `rej` **depends on position** of the first unequal byte
  - i.e., that attacker may also receive **timing-related information**

Side-channel attack via tag verification w/ timing

- attacker $\mathcal{A}$ wishes to forge a **verifiable s-byte tag t** for target message m
  - assume that $\mathcal{A}$ knows t_i, i.e., the first i bytes of tag t (for some i = 0, 1, …, s-1)
  - for j = 0, …, 255
    - send verification query (m, t_j) where $t_j = t_i \,||\, j \,||\, (00)^{s-i-1}$
    - get response res_j (probably `rej`) and **measure $time_j$** spent for computation of res_j
  - if $time_{j^*}$ is the **maximum measured response time**, then set $t_{i+1} = t_i \,||\, j^*$
- **realistic** attack
  - forged code updates in Xbox 360 to load pirated games into the hardware
  - exploited differences of 2.2msec between rejection times!
  - 4096 queries are needed to recover a 16-byte tag!

Side-channel attack via tag verification (cont.)

Other side-channels can be used

- Padding Oracle Attacks
- Exploits leaked information about tag verification due to padding
  - PKCS#7 specifies how messages are unambiguously padded (in modes of operations)
- Attacker get additional information of whether the padding was correct
  - E.g., if padding is correct message processing results in longer response time

Summary of message-authentication crypto tools

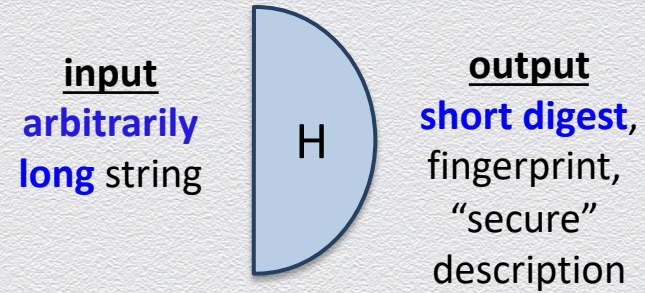
	Hash (SHA2-256)	MAC	Digital signature
Integrity	Yes	Yes	Yes
Authentication	No	Yes	Yes
Non-repudiation	No	No	Yes
Crypto system	None	Symmetric (AES)	Asymmetric (e.g., RSA)

7.0 Properties of cryptographic 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**



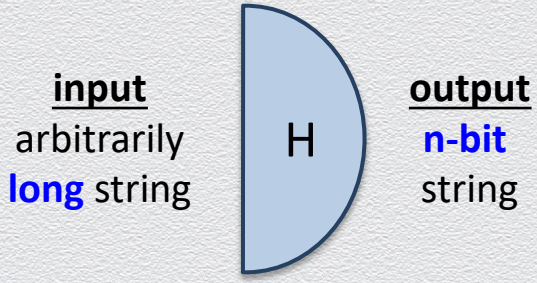
Important role in modern cryptography

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

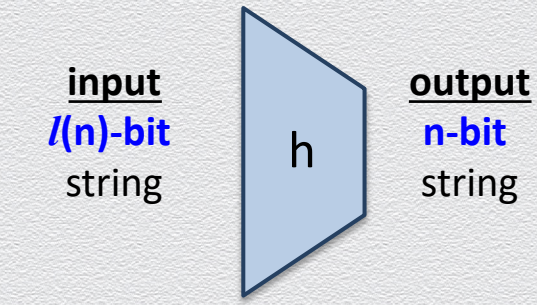
Hash & compression functions

Map messages to short digests

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

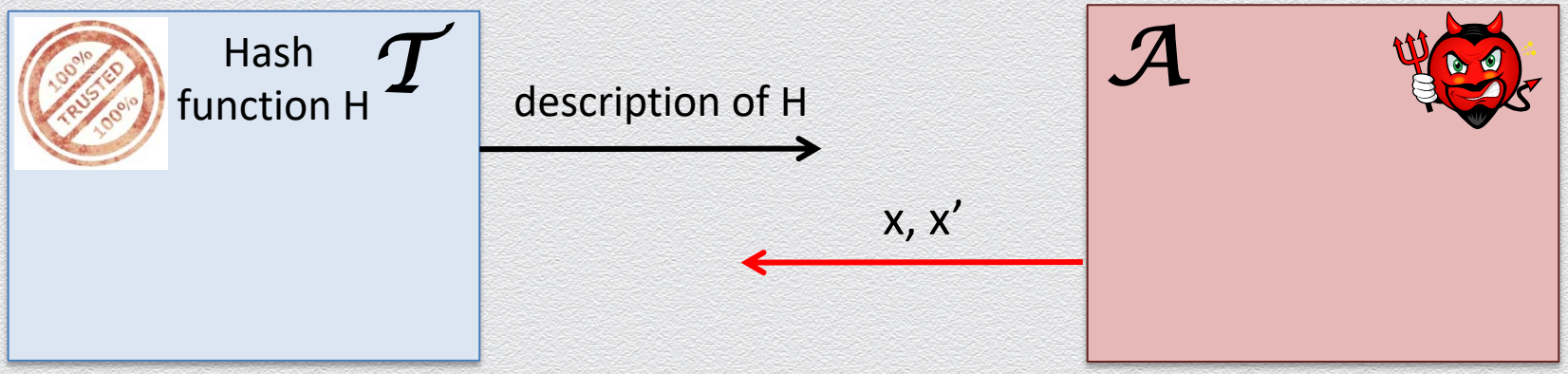


- a **compression** (hash) function h() maps
  - a **long** binary string to a **shorter** binary string
  - an ***l*(n)-bit string** to a **n-bit string**, with ***l*(n) > n**



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

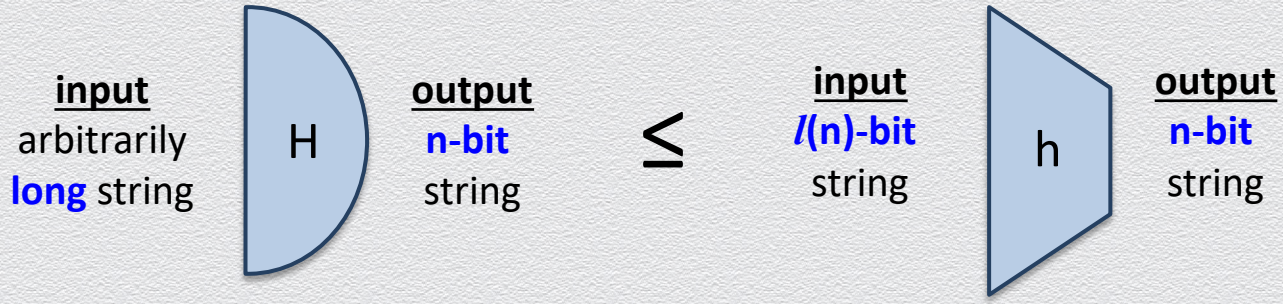
- **preimage resistant** (or **one-way**)
  - if given y $\in$ Y, finding a value x $\in$ X s.t. H(x) = y happens negligibly often
- **2-nd preimage resistant** (or **weak collision resistant**)
  - if given a **uniform** x $\in$ X, finding a value x' $\in$ X, s.t. x'$\neq$ x and H(x') = H(x) happens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

7.0.1 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



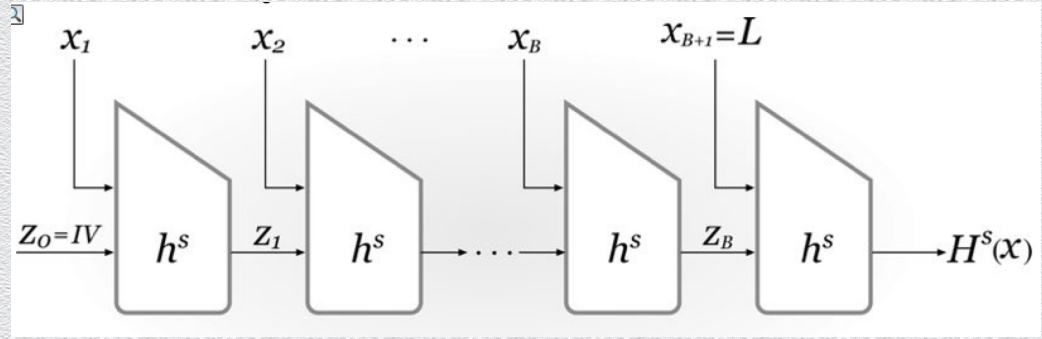
- thus, in practice, it suffices to realize a collision-resistant compression function h
- compressing by 1 single bit is a least as hard as compressing by any number of bits!

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

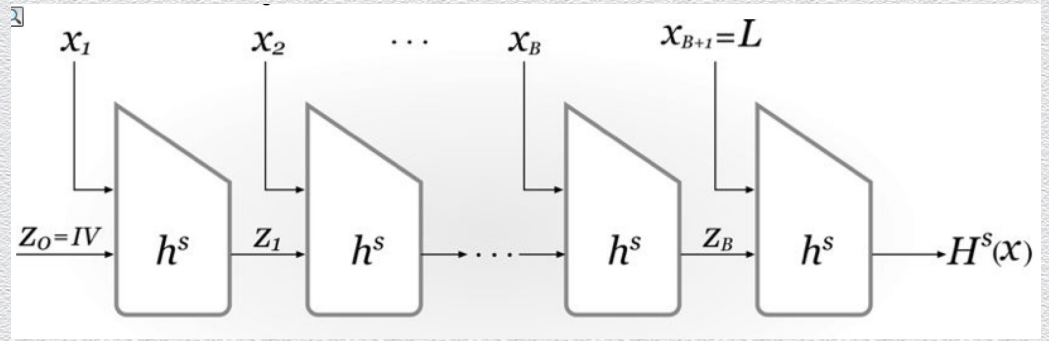
Merkle-Damgård design



- 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

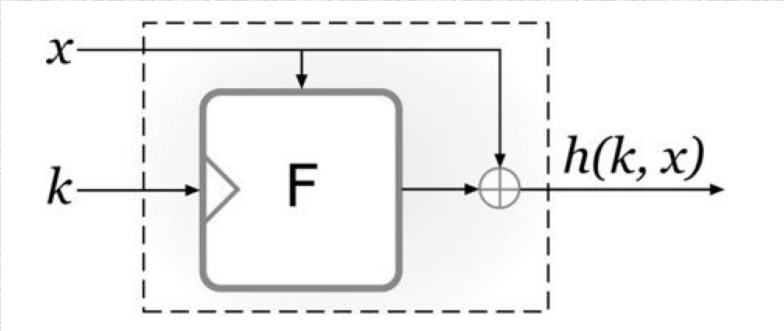
If the compression function h is CR,
then the derived hash function H is also CR!



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 **$h(x||k) = F_k(x)$ XOR x**



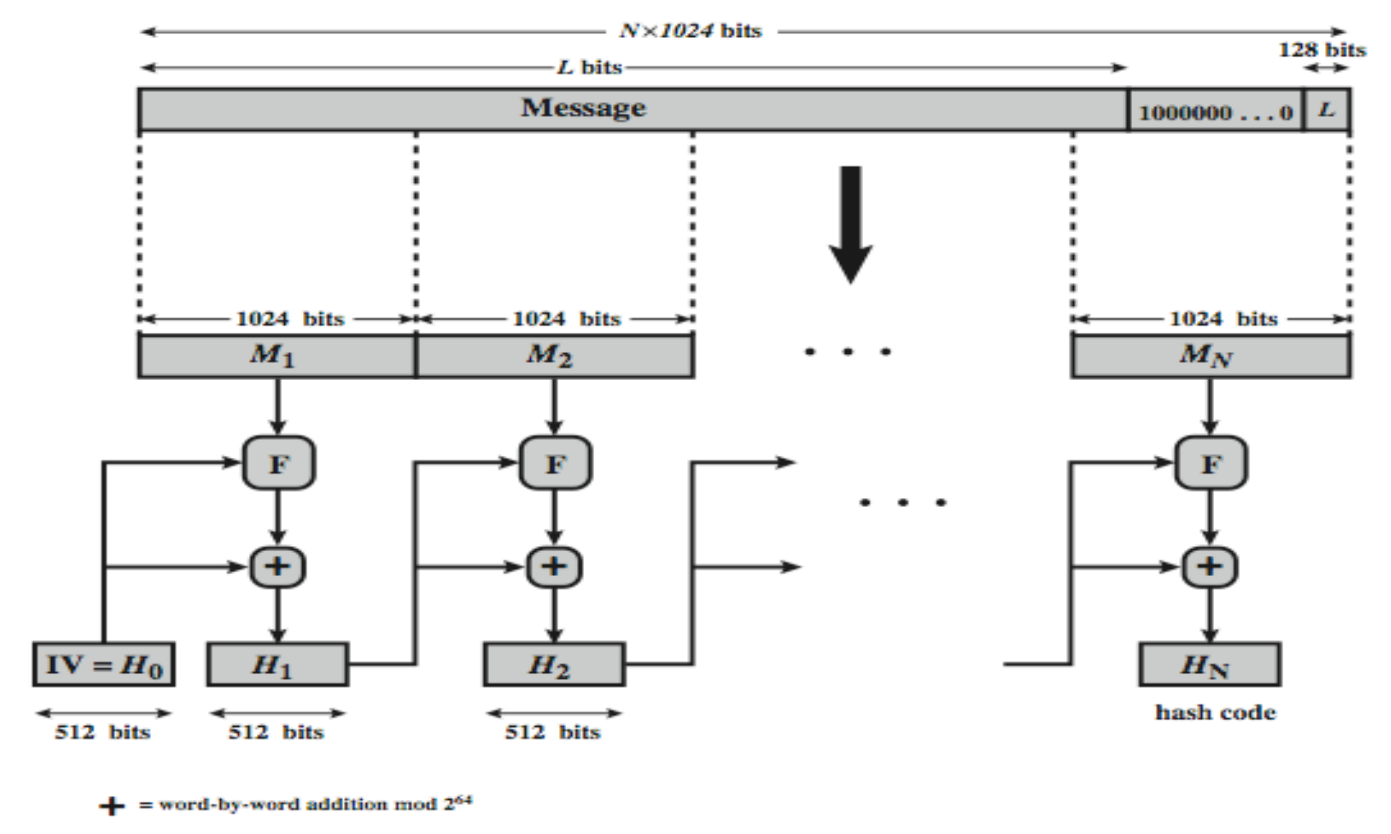
Security

- 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

7.0.2 Generic attacks

Generic attacks against cryptographic hashing

Assume a CR function $h : \{0,1\}^* \rightarrow \{0,1\}^n$

- **brute-force** attack
  - for x = 0 to 2^n-1 (sequentially, for each string x in the domain):
    - compute & record hash value h(x)
    - if h(x) equals a previously recorded hash h(y) halt & output collision on $x \neq y$
- **birthday** attack
  - surprisingly, a more efficient generic attack exists!

Birthday paradox

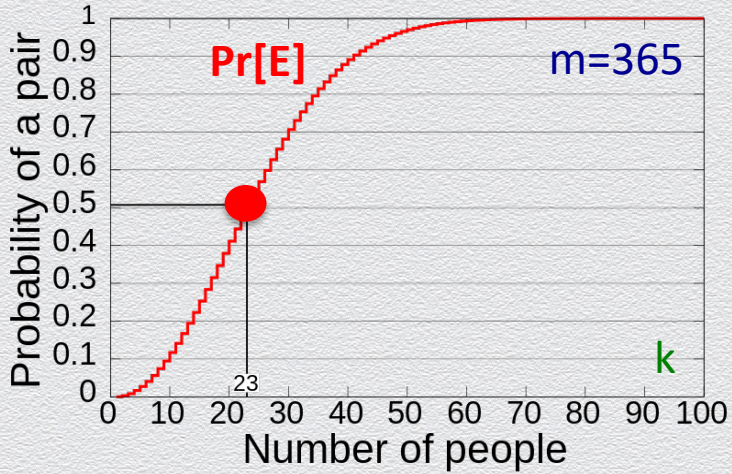
"In any group of 23 people (or more), it is **more likely** (than not) that **at least two** individuals have their birthday on the **same** day"

- 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

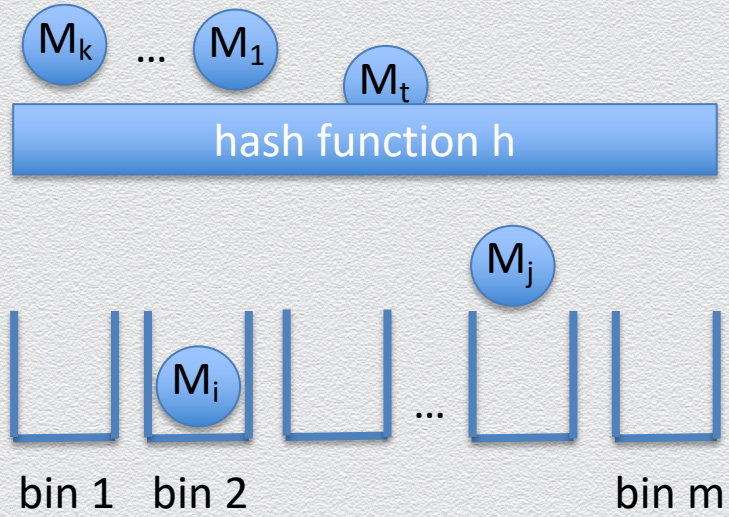
- analysis shows: $\Pr[E] \approx 1 - e^{-k(k-1)/2m}$ (1)
  - if **Pr[E] = 1/2**, Eq. (1) gives **$k \approx 1.17\, m^{1/2}$**
  - thus, for m = 365, k is around 23 (!)
    - assuming a uniform birth distribution



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 **exhausting** search
- analogy
  - k balls: distinct messages chosen to hash
  - m bins: number of possible hash values
  - independent & random throwing
    - random 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:

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

- 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

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

$$1.25(m)^{1/2} = 1.25 \times 2^{n/2}$$

Overall

Assume a CR function h producing hash values of size n

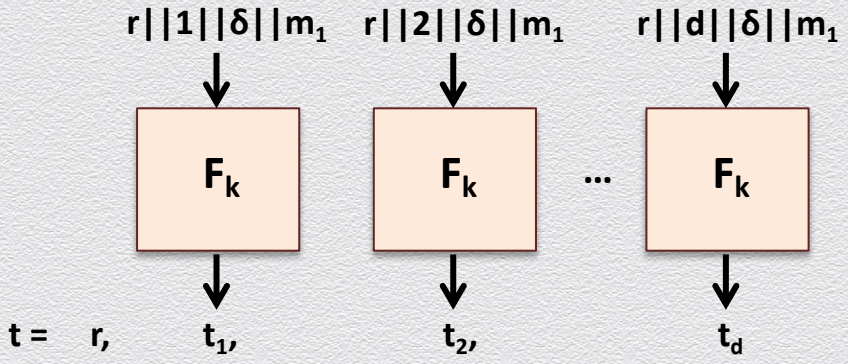
- **brute-force** attack
  - evaluate h on $2^n + 1$ distinct inputs, enumerated by counting
  - by the "pigeon hole" **principle**, at least 1 collision **will be** found
- **birthday** attack
  - evaluate h on (much) **fewer** distinct **randomly** selected inputs
  - by "balls-into-bins" **probabilistic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

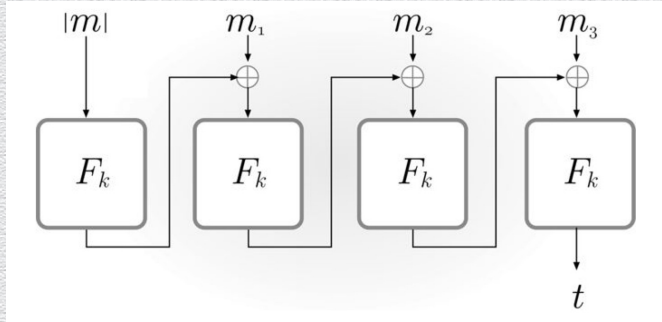
7.1 Applications to cryptography

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

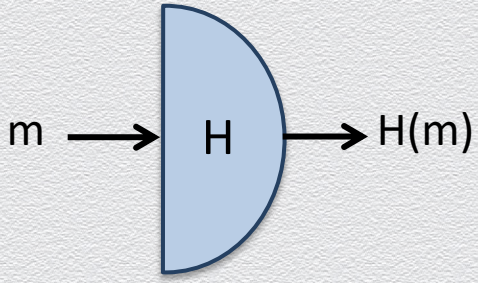


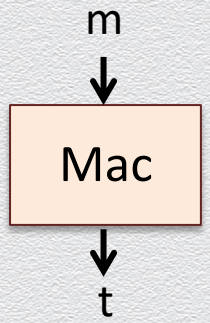


[1] Hash-and-MAC: Design

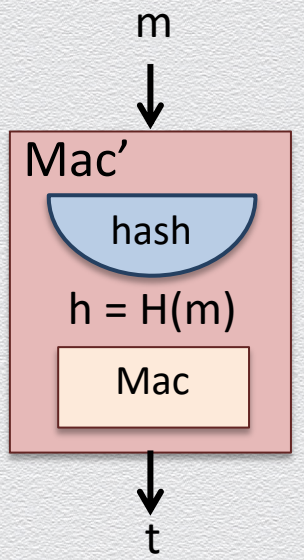
Generic method for designing **secure MAC for messages of arbitrary lengths**

- 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



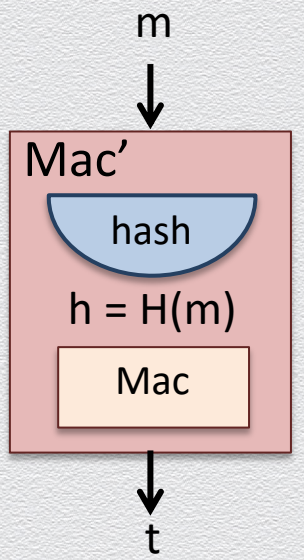
[1] Hash-and-MAC: Security

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 **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

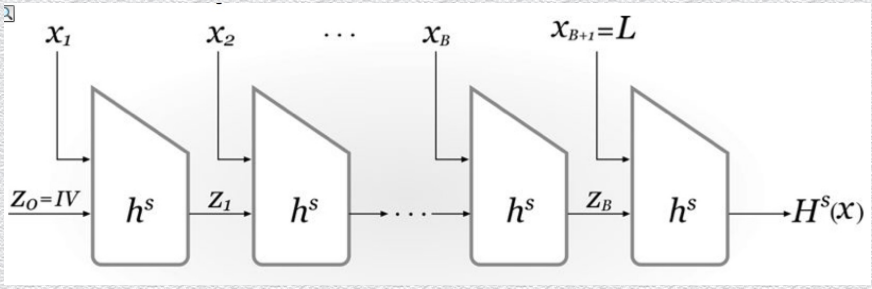
Set tag t as:

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

- 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!

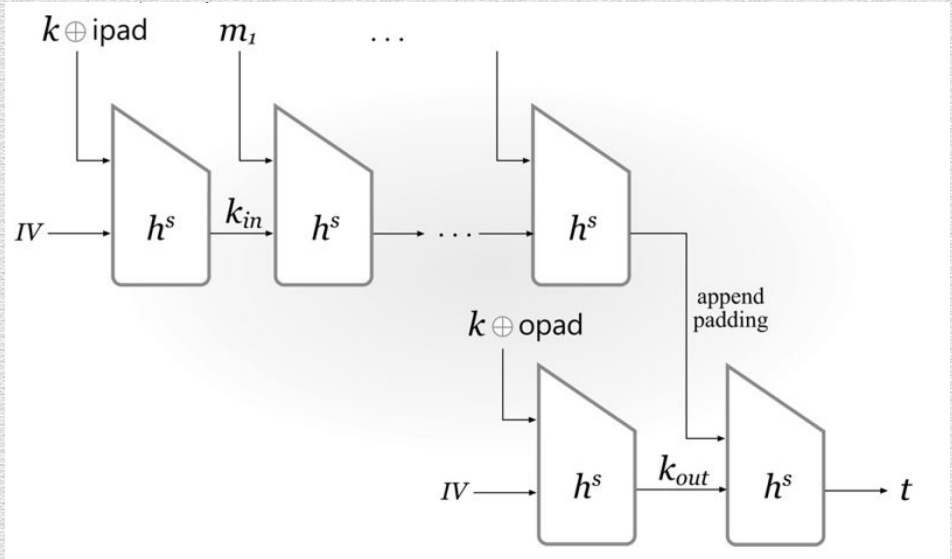


[2] HMAC: Secure design

Set tag t as:

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

- intuition: **instantiation of hash & sign paradigm**
- two layers of hashing H
  - **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

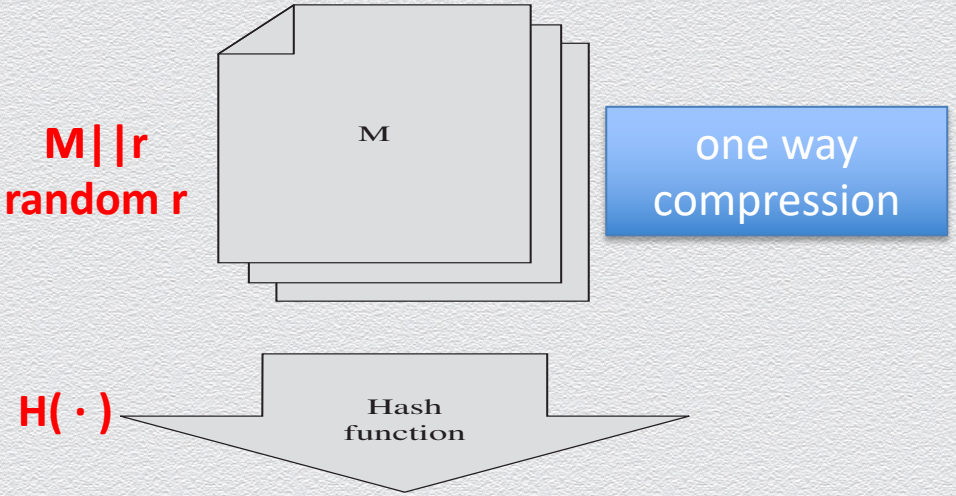
If used with a secure hash function and according to specs, HMAC is secure

- no practical attacks are known against HMAC

7.2 Applications to security

Generally: Message digest (= hash value = fingerprint)

Short secure description of data (primarily used to detect changes)



A crypto hash function is **not**
an encryption scheme,
a MAC tag or signature,
or anything else

other than a **random mapping**
(thus collision resistant) into
a **fixed-length** hash domain.

Question: Does H(M) "conceal" M?

Answer: It depends on M's message space & prob. distribution

Application 1: Digital envelops

A commitment scheme implements a physical envelop

- two operations
- commit(x, r) = C
  - i.e., put message x into an envelop (using randomness r)
  - 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
  - open(C, m, r): check if h(m || r) =? C
  - **binding property**: you cannot change the contents of a sealed envelop

Application 1: Security properties

Hiding: perfect/computational 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/computational 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 1: Fair digital coin flipping

Problem

- To decide who will do the dishes: Alice is to call the coin flip & Bob is to flip the coin
- But Alice may change her mind, Bob may skew the result

Protocol

- Alice calls the coin flip x but only tells Bob a commitment C of x
- Bob flips the coin & reports the result R
- Alice reveals her call x & Bob verifies that revealed call x “matches” commitment C
- If Alice’s verified call x matches Bob’s result, i.e., x = R, Alice wins; else Bob wins

Example 1: Fair digital coin flipping (cont.)

Protocol

- Alice calls the coin flip x but only tells Bob a commitment C of x
- Bob flips the coin & reports the result R
- Alice reveals her call x & Bob verifies that revealed call x “matches” commitment C
- If Alice’s verified call x matches Bob’s result, i.e., x = R, Alice wins; else Bob wins

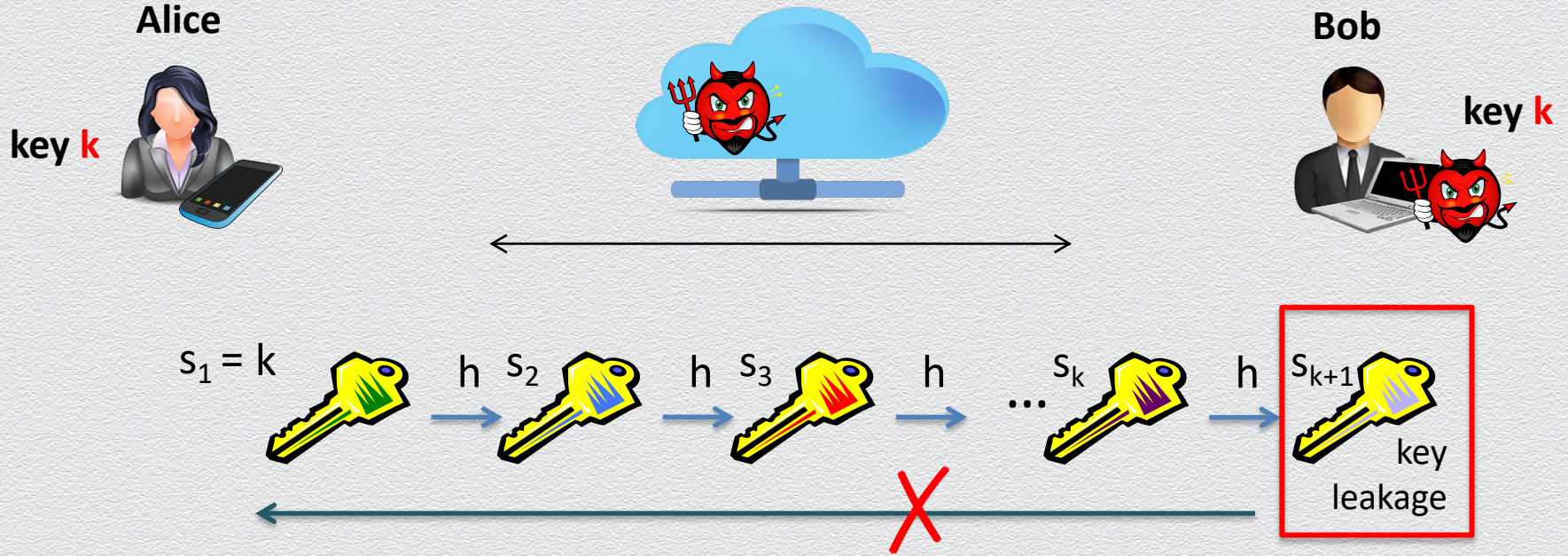
Security

- Hiding: Bob gains nothing by seeing Alice’s commitment C or skewing coin toss R
- Binding: Alice cannot change her mind x after the coin R is announced

Application 2: 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 3: 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

3.1 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

3.2 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 3.3: 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!

Application 4: Concealing stored passwords

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: Password storage

Identity	Password
Jane	qwerty
Pat	aaaaaa
Phillip	oct31witch
Roz	aaaaaa
Herman	guessme
Claire	aq3wm$oto!4

Plaintext

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

Concealed via hashing

Subject to “concealment” preconditions

If fully concealed, are we safe?

Any hash pre-image leads to impersonation

Application 5: Hash-and-digitally-sign

Very often digital signatures are used with hash functions

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

Signing message M

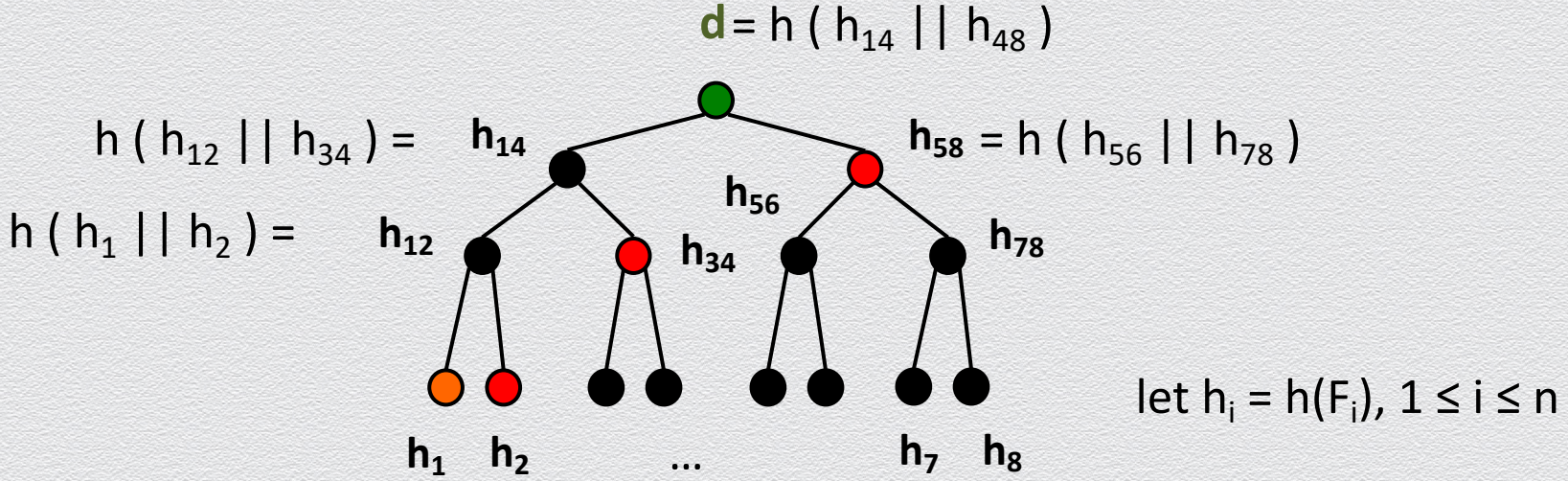
- let h be a cryptographic hash function, assume RSA setting (n, d, e)
- compute signature $\sigma = h(M)^d \bmod n$
- send σ, M

Verifying signature σ

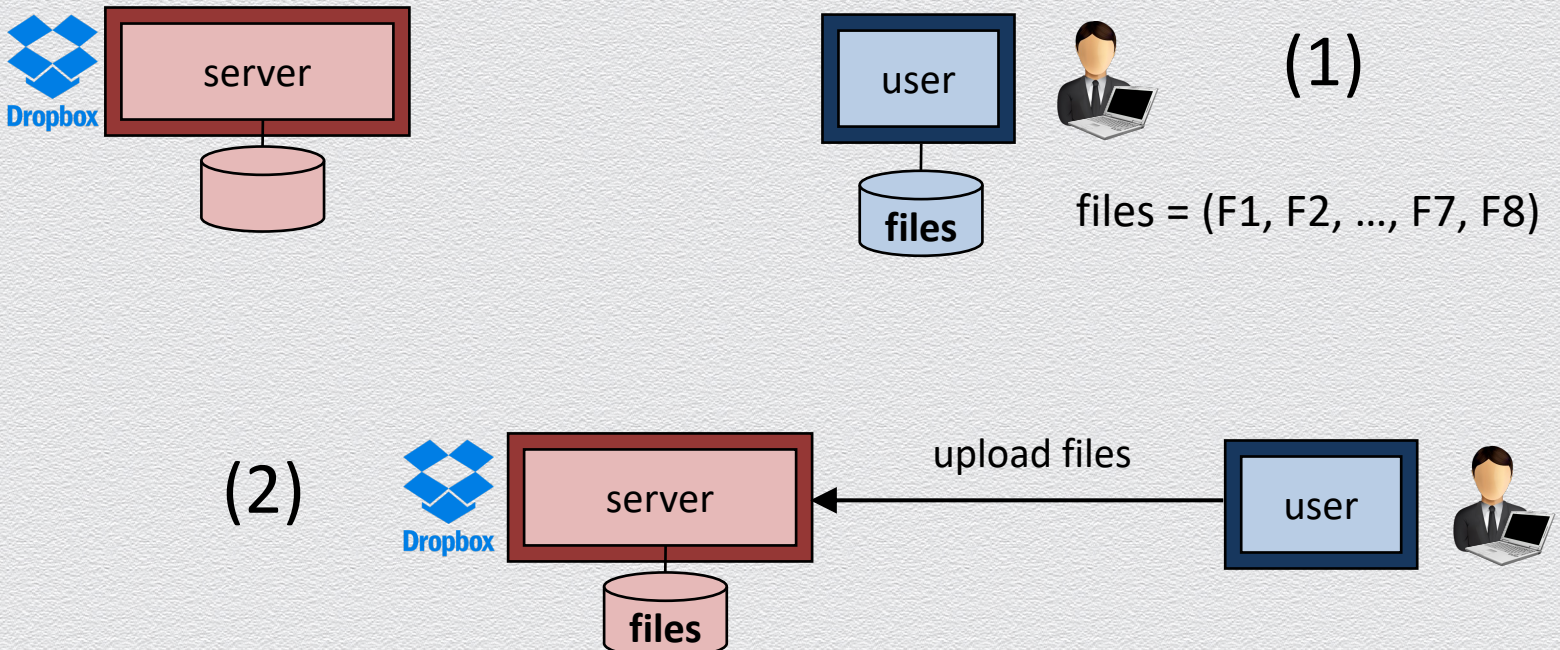
- use public key (e,n)
- compute $H = \sigma^e \bmod n$
- if H = h(M) output ACCEPT, else output REJECT

Application 6: The Merkle tree

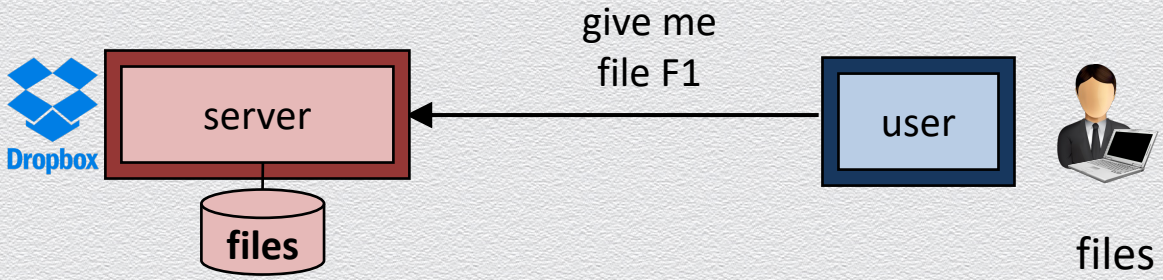
An alternative (to Merkle-Damgård) method to achieve domain extension



Example 6: Secure cloud storage



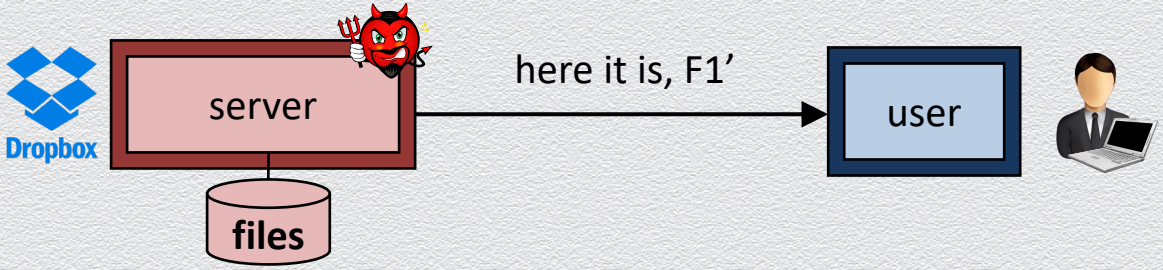
Example 6: Secure cloud storage



(3)

files = (F1, F2, …, F7, F8)

(4)

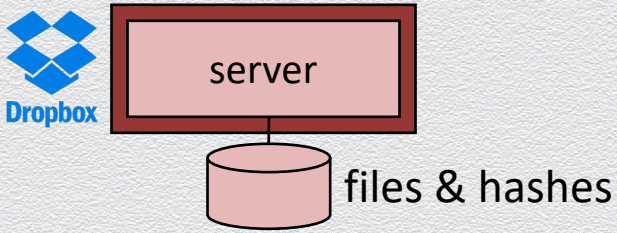


F1' = #@$@!#^@$^… (altered)

Example 6: Secure cloud storage – per-file hashing

Bob wants to outsource storage of files F_1, F_2,…,F_8 to Dropbox & check their integrity

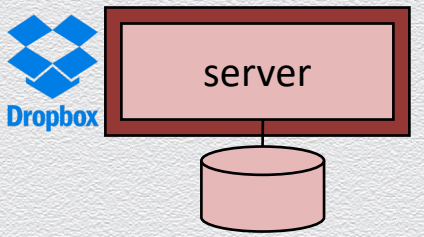
- Bob stores random r
  (& keeps it secret)
- Bob sends to Dropbox
  - files F_1, F_2,…,F_8
  - hashes $h(r||F_1)$, $h(r||F_2)$,…, $h(r||F_8)$

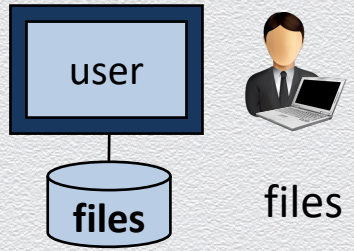


Every time Bob **reads** a file F_i, he also reads $h(r||F_i)$ to verify F_i's integrity

- any problems with **writes**?

Example 6: Secure cloud storage – per-file-set hashing



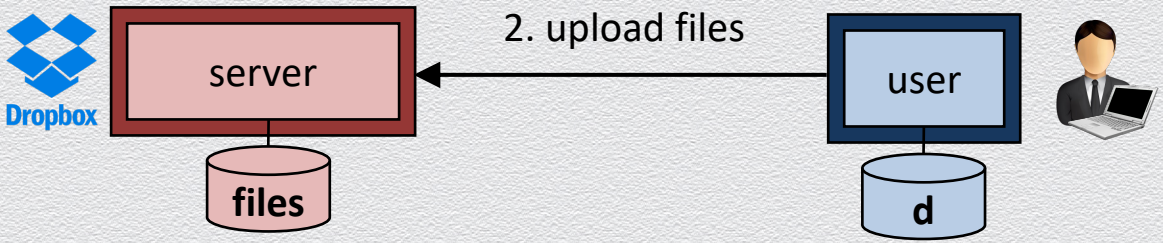


(1)

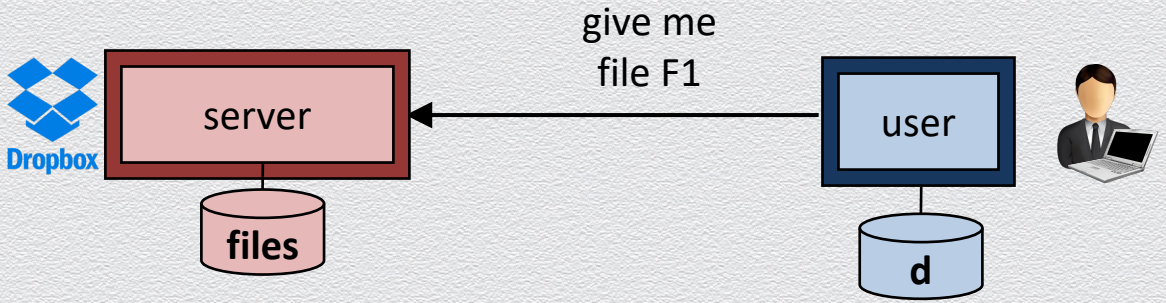
files = (F1, F2, …, F7, F8)

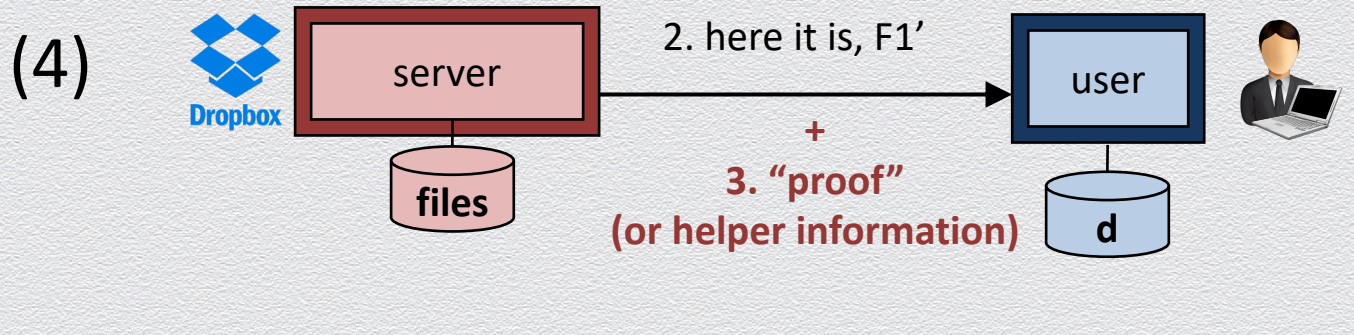
1. use CR hash function h to compute over all files a digest d, |d| << |F|

(2)

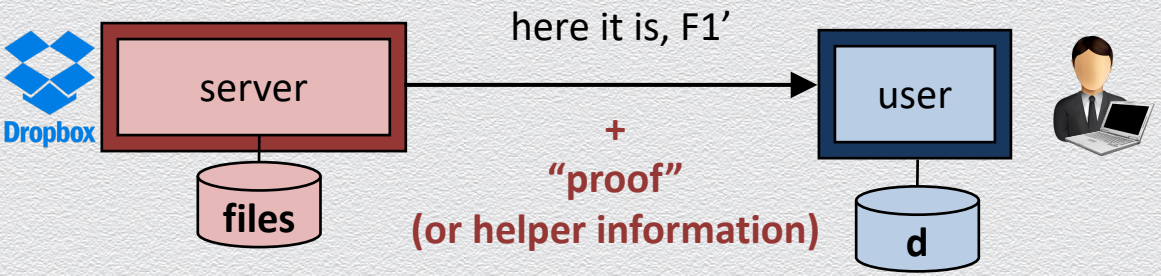


Example 6: Secure cloud storage – integrity checking



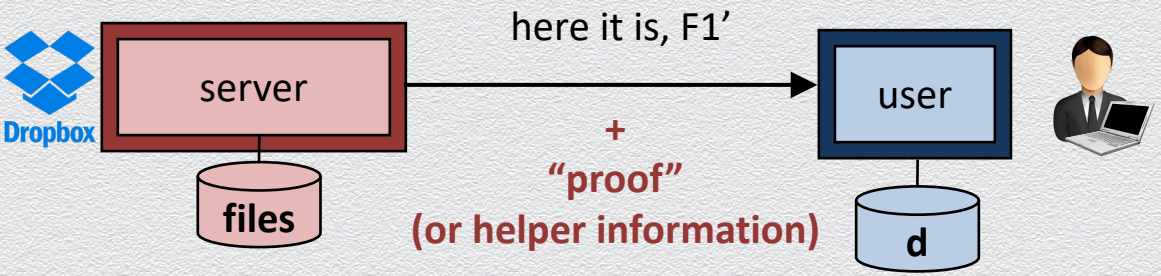


Example 6: Secure cloud storage – verification



- user has
  - authentic digest d (locally stored)
  - file F1' (to be checked/verified as it can be altered)
  - **proof** (to help checking integrity, but it can be maliciously chosen)
- user locally verifies received answer
  - combine the 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!

Example 6: Data authentication via the Merkle tree



(4)

verification

"is F1' intact?"

d' ≠ d rej

d' = d acc

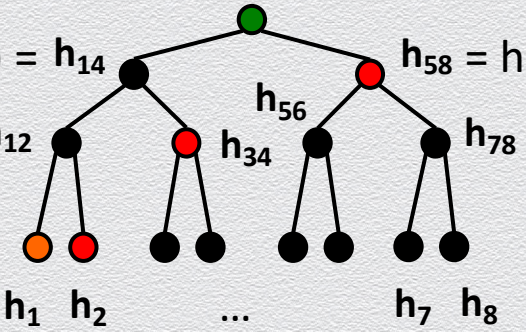
digest is the **green** root hash

$\mathbf{d} = h (h_{14} \,||\, h_{48})$

$h (h_{12} \,||\, h_{34}) = \mathbf{h_{14}}$

$\mathbf{h_{58}} = h (h_{56} \,||\, h_{78})$

$h (h_1 \,||\, h_2) = \mathbf{h_{12}}$



compute candidate d' based on **answer** & **proof**

answer is **orange** hash

proof is **red** hash path

let $h_i = h(F_i)$, $1 \le i \le 8$

7.3 User authentication

User identification & authentication

Identification

- asserting who a person is

Authentication

- proving that a user is who she says she is
- methods
  - something the user *knows*
  - something the user *is*
  - something user *has*

Does authentication imply identification?

Suppose that a user

- provides her (login) name and
- uses one of the three methods to authenticate into a computer system
  - either terminal or remote server via a web browser
- when does user authentication imply user identification?
  - not quite…

Example: Something you know

The user has to know some secret to be authenticated

- password, personal identification number (PIN), personal information like home address, date of birth, name of spouse ("security" questions)

But anybody who obtains your secret "is you…"

- impersonation Vs. delegation
- you leave no trace if you pass your secret to somebody else

What if there is a case of computer misuse?

- i.e., where somebody has logged in using your username & password…
- can you prove your innocence?
- can you prove that you have not divulged your password?

Thus…

- a password does not authenticate a person
- successful authentication only implies that the user knew a particular secret
- there is no way of telling the difference between the legitimate user and an intruder who has obtained that user's password
- **unfortunately: this holds true for almost all of authentication methods…**

7.3.1 Something you know – password authentication

Something you know

- passwords
  - or PINs
  - or answers to “security” questions (e.g., where did you meet your wife?)

Problems with passwords

Many attack vectors…

- password “live” in different “places:”
  1) user’s brain, 2) channel & 3) authentication server

1) password guessing

- predict weak passwords

2) phishing & spoofing or **cached passwords**

- deceive users to reveal their password

3) leaked password files

- steal user credentials

Password guessing

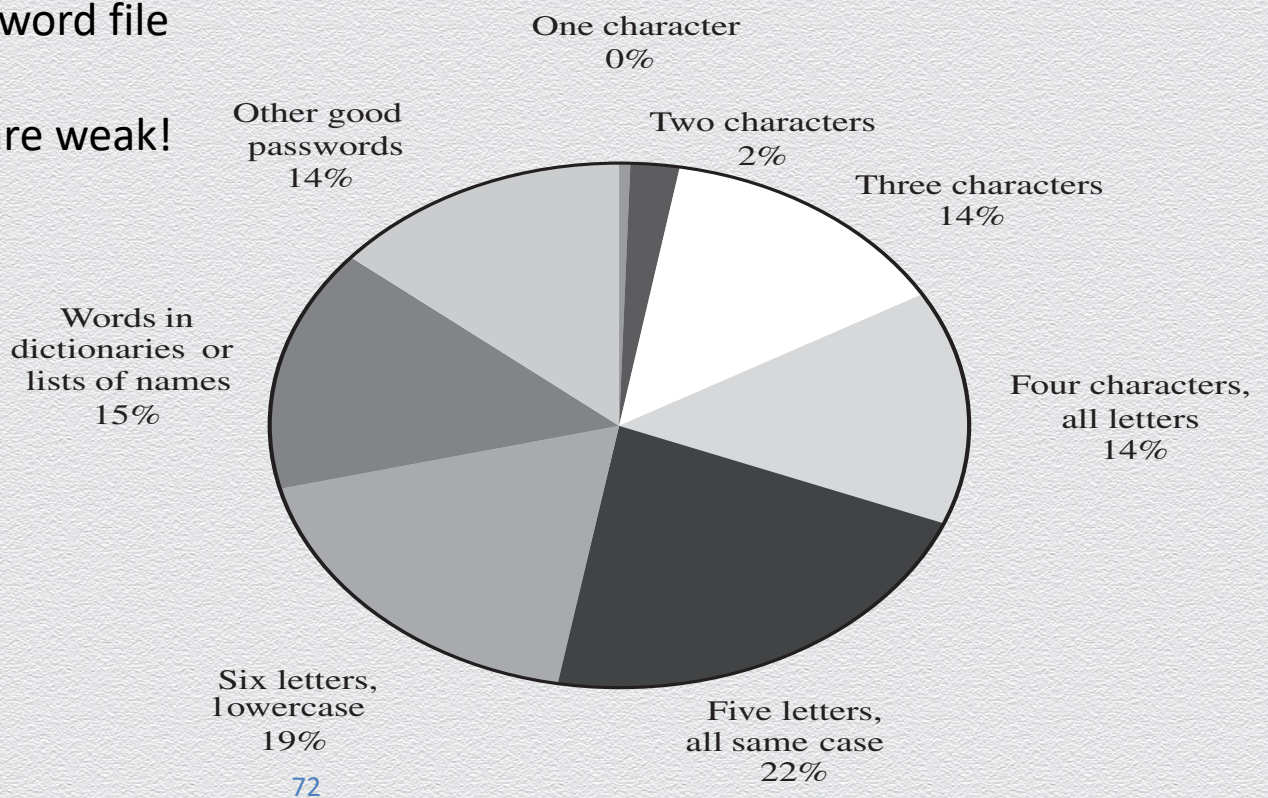
Infer passwords through guessing

- Low-entropy passwords
  - To be easy to remember, passwords are often weak easy-to-predict secrets
  - e.g., password is “Password1”
- Password reuse
  - To be easy to remember, passwords are often reused across many authentication servers
  - e.g., same password for all accounts

Distribution of password types

Graph from an old leaked password file

The point is: Most passwords are weak!



Online dictionary attacks

- Direct brute-force or dictionary attacks against passwords
  - employs only the authentication system
  - attacker tries to impersonate a victim by trying
    - all possible (short length) passwords or
    - passwords coming from a known dictionary
  - (cf. offline brute-force or dictionary attacks using leaked hashed passwords)
- Countermeasure
  - block login & lock account after many consecutive failed authentication attempts
  - false negatives…

Phishing & spoofing

- Identification and authentication through username and password provide **unilateral authentication**
- Computer verifies the user's identity but the user has no guarantees about the identity of the party that has received the password
- In **phishing** and **spoofing** attacks a party voluntarily sends the password over a channel, but is misled about the end point of the channel

Spoofing

- Attacker starts a malicious program that presents a fake login screen and leaves the computer
- If the next user coming to this machine enters username and password on the fake login screen, these values are captured by the malicious program
  - login is then typically aborted with a (fake) error message and the spoofing program terminates
  - control returns to operating system, which now prompts the user with a genuine login request
  - thus, the victim does not suspect that something wrong has happened
    - the victim may think that the password was mistyped…

Counteracting password spoofing

- display **number of failed logins**
  - may indicate to the user that an attack has happened
- **trusted path**
  - guarantee that user communicates with the operating system and not with a spoofing program
- **mutual authentication**
  - user authenticated to system, system authenticated to user

Phishing

- attacker impersonates the system to trick a user into releasing the password
- e.g.,
  - a message could claim to come from a service you are using
  - tell you about an upgrade of the security procedures
  - and ask you to enter your username and password
    at the new security site that will offer stronger protection
- attacker impersonates the user to trick a system operator into releasing the password to the attacker
  - **social engineering**

Cached passwords

- description of login has been quite abstract
  - password travels directly from user to the password checking routine
- in reality, it will be held temporarily in intermediate storage locations
  - e.g., like buffers, caches, or a web page
- management of these storage locations is normally beyond user's control
  - a password may be kept longer than the user has bargained for

Leaked password files

- Breach authentication server to steal user credentials
  - e.g., plaintext passwords

- Countermeasures
  - protect passwords via encryption (e.g., a symmetric-key cipher)
    - subject to keeping the secret key secure against the server's compromise…
    - hard to achieve in practice…
  - concealed password via hashing
    - subject to meeting conditions for secret concealment via hashing…

Protecting the password file

Operating system maintains a password file (with user names & passwords)

- attacker could try to compromise its confidentiality or integrity
- options for protecting the password file
  - cryptographic protection
  - access control enforced by the operating system
  - combination of cryptographic protection and access control, possibly with further measures to slow down dictionary attacks

Access control settings

- only privileged users must have write access to the password file
  - an attacker could get access to the data of other users simply by changing their password
  - even if it is protected by cryptographic means
- if read access is restricted to privileged users, passwords could be stored unencrypted
  - in theory – in practice, bad idea because of breaches
- if password file contains data required by unprivileged users, passwords must be “encrypted”; such a leaked file can still be used in dictionary attacks
  - typical example is **/etc/passwd** in Unix
  - many Unix versions store encrypted passwords in a shadow password file (not publicly accessible)

Example: Password storage via hashing

Identity	Password
Jane	qwerty
Pat	aaaaaa
Phillip	oct31witch
Roz	aaaaaa
Herman	guessme
Claire	aq3wm$oto!4

Plaintext

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

Concealed

Subject to “concealment” preconditions

If fully concealed, are we safe?

Any hash pre-image leads to impersonation

Hashing passwords is not enough

An immediate control against password leakage through stolen password files, involves concealing passwords stored at the authentication server via hashing

Why are offline dictionary attacks quite effective using leaked hashed passwords in practice?

- Most hashed passwords are weak passwords
- Thus, they can be “cracked”
  - Invert the hash
  - Find a 2nd preimage of the hash

Password cracking

Given leaked hashed passwords, recover passwords

- Use exhaustive search by hashing over guessed passwords…
  - brute-force attack: try all possibilities
  - dictionary attacks: try all words in a dictionary & variations of them
  - rainbow tables: try possibilities in a systematic way via a data structure
- These methods impose different time-space trade-offs on attacker's workload
  - preprocessing is often very useful, e.g.,
    - precompute a dictionary-based set of password-hash pairs
    - use precomputed set for cracking any newly leaked hashed passwords

Countermeasures

Password salting

U, h(PU||**SU**), SU

Now preprocessing is useless;
or it must be **user specific**!

- to slow down dictionary attacks
  - a user-specific **salt** is appended to a user's password before it is being hashed
  - each salt value is stored in the clear along with its corresponding hashed password
  - if two users have the same password, they will have different hashed passwords
  - example: Unix uses a 12 bit salt

Hash strengthening

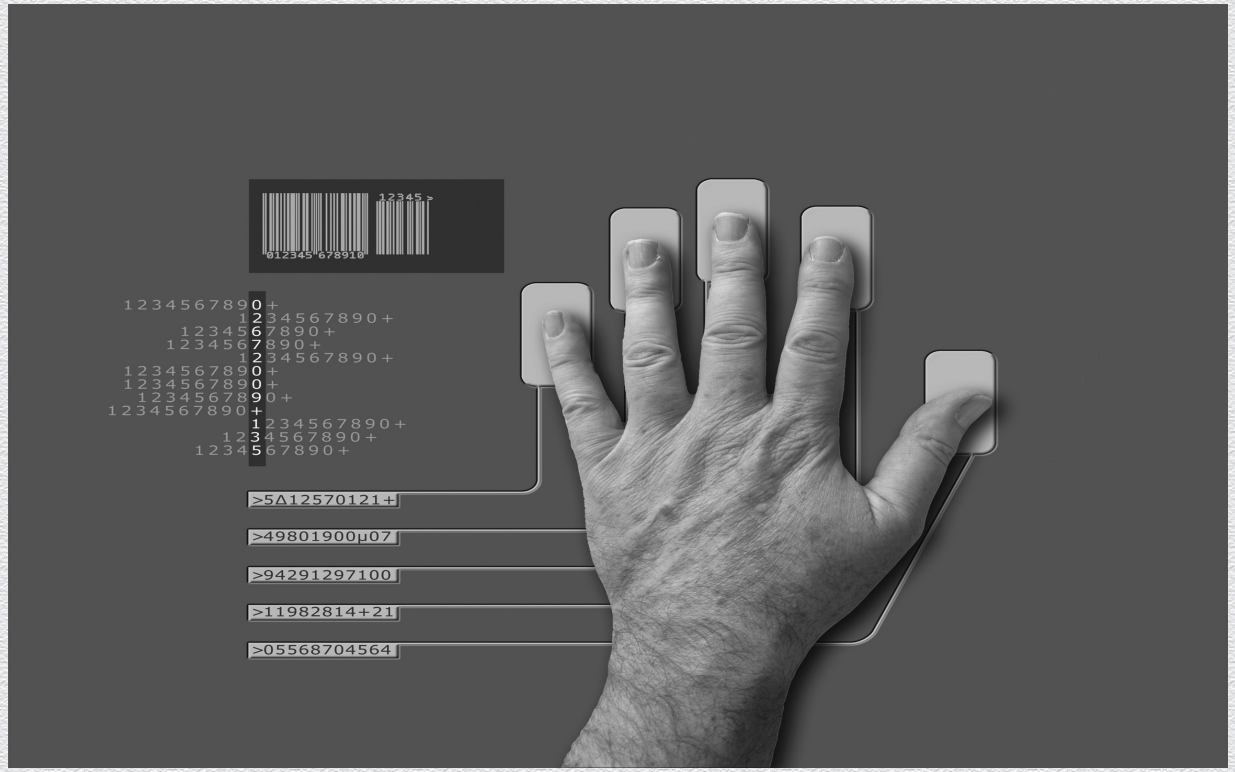
- to slow down dictionary attacks
  - a password is hashed k times before being stored

7.3.2 Something you are – biometric authentication

Something you are

- biometric schemes use people's unique physical characteristics
  - traits, features
  - face, finger prints, iris patterns, hand geometry
- biometrics may seem to be the most secure solution for user authentication
- biometric schemes are still quite new

Biometrics: Something you are



Problems with biometrics

- Intrusive
- Expensive
- Single point of failure
- Sampling error
- False readings
- Speed
- Forgery

Fingerprint

- Enrolment
  - reference sample of the user's fingerprint is acquired at a fingerprint reader
- Features are derived from the sample
  - fingerprint minutiae
    - end points of ridges, bifurcation points, core, delta, loops, whorls, …
- For higher accuracy, record features for more than one finger
- Feature vectors are stored in a secure database
- When the user logs on, a new reading of the fingerprint is taken
  - features are compared against the reference features

Identification Vs. verification

- Biometrics are used for two purposes
  - Identification: 1:n comparison, i.e., identify user from a database of n persons
  - Verification: 1:1 comparison, i.e., check whether there is a match for a given user
- Authentication by password
  - clear reject or accept at each authentication attempt
- Biometrics
  - stored reference features will hardly ever match precisely features derived from the current measurements

Failure rates

- Measure similarity between reference features and current features
- User is accepted if match is above a predefined threshold
- **New issue: false positives and false negatives**
- Accept wrong user (false positive)
  - security problem
- Reject legitimate user (false negative)
  - creates embarrassment and an inefficient work environment

Forgeries

Fingerprints, and biometric traits in general, may be unique but they are no secrets!

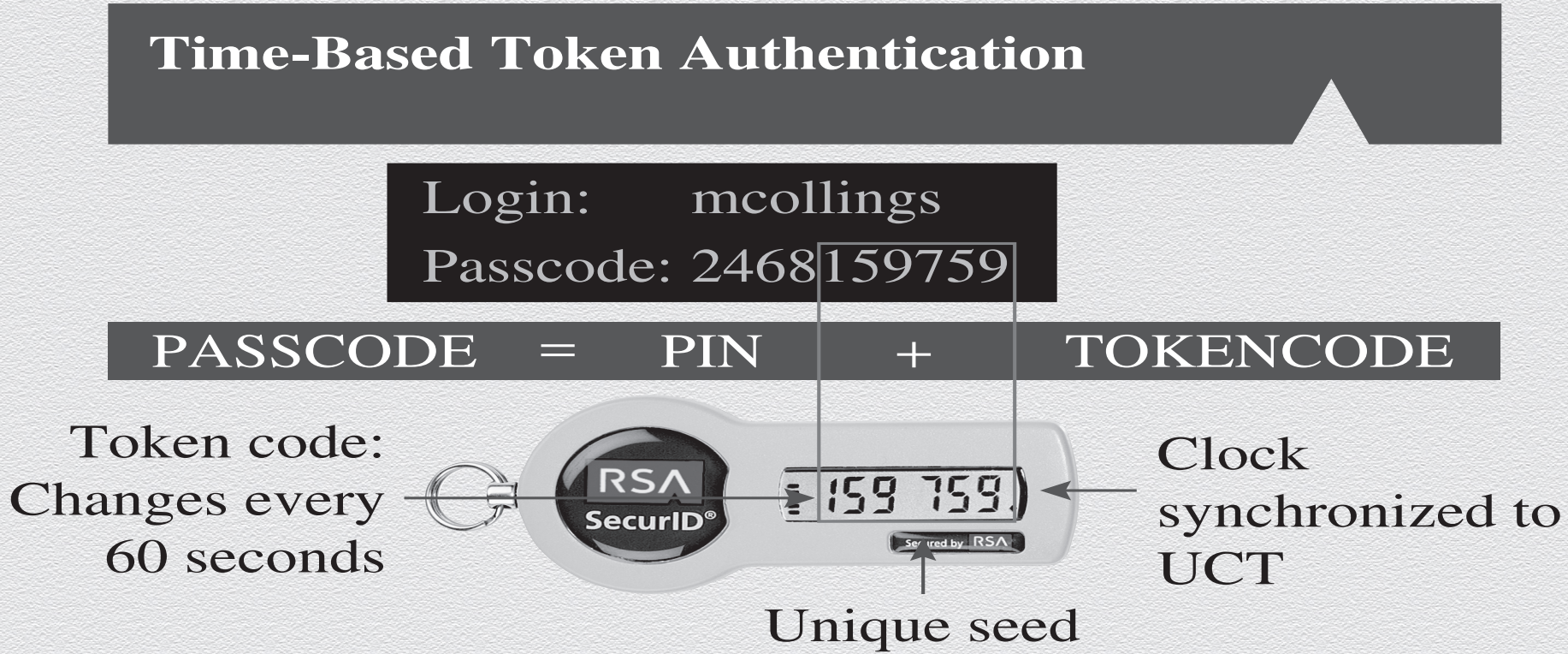
- you are leaving your fingerprints in many places
- rubber fingers have defeated commercial fingerprint-recognition
- minor issue if authentication takes place in the presence of security personnel
  - when authenticating remote users additional precautions have to be taken
- user acceptance: so far fingerprints have been used for tracing criminals

7.3.3 Something you have – authentication tokens

Something you have

- user presents a physical token to be authenticated
  - keys, cards or identity tags (access to buildings), smart cards
- limitations
  - physical tokens can be lost or stolen
  - anybody in possession of token has the same rights as legitimate owner
- physical tokens are often used in combination with something you know
  - e.g. bank cards come with a PIN or with a photo of the user
  - this is called: **2nd-factor authentication or multi-factor authentication**

Tokens: something you have

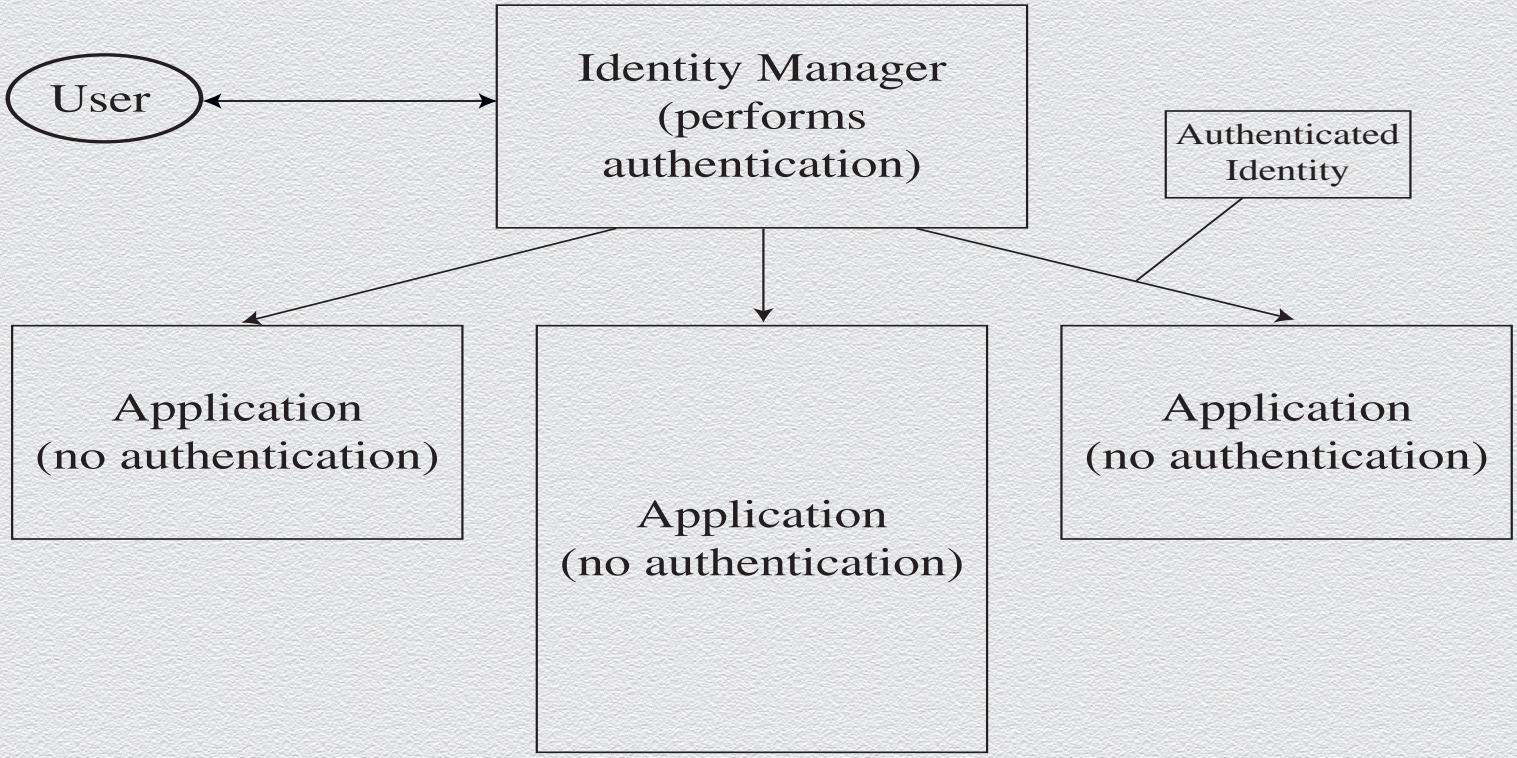


Problems with tokens

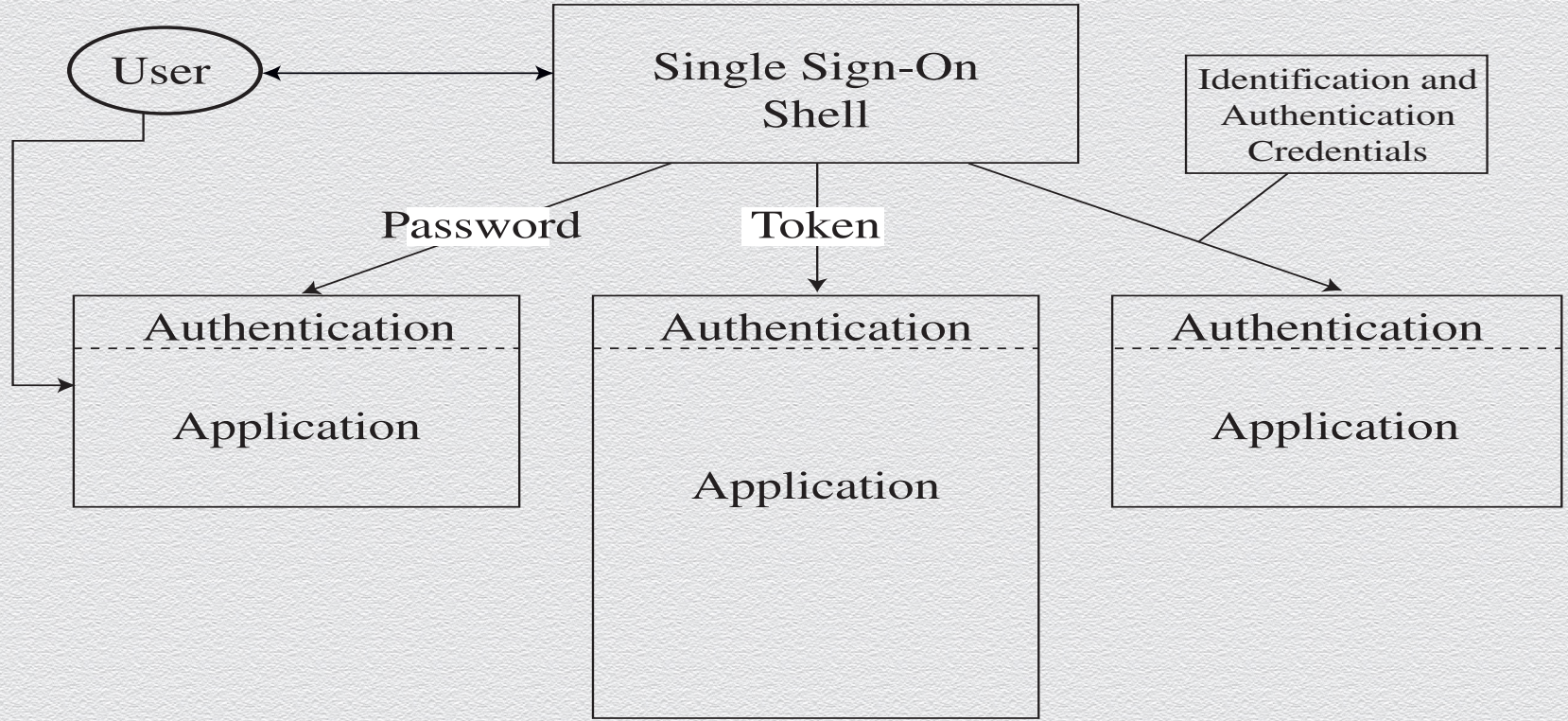
- Inconvenience
- Lost token
- Stolen token
- Cloned token
- Side-channel attacks (for key exfiltration)

7.3.4 Other methods

Federated identity management



SSO: Single Sign-On



More details on SSO

- Having to remember many passwords for different services is a nuisance
  - with a single sign-on service, you have to enter your password only once
  - an alternative solution: password managers
- A simplistic single-sign on service could store your password and do the job for you whenever you have to authenticate yourself
  - such a service adds to your convenience but it also raises new security concerns
- System designers have to balance convenience and security
  - ease-of-use is an important factor in making IT systems really useful
  - but many practices which are convenient also introduce new vulnerabilities

More on authentication

If dissatisfied with security level provided by passwords?

- you can be authenticated on the basis of
  - something you know
  - something you have
  - something you are
  - **what you do – behavioural**
  - **where you are – location based**

What you do

- people perform mechanical tasks in a way that is both repeatable and specific to the individual
- experts look at the dynamics of handwriting to detect forgeries
- users could sign on a special pad that measures attributes like writing speed and writing pressure
- on a keyboard, typing speed and key strokes intervals can be used to authenticate individual users
- more recently behaviours from one's mobile phone have been studied

Where you are

- some OSs grant access only if you log on from a certain terminal
  - a system administration may only log on from an operator console but not from an arbitrary user terminal
  - users may be only allowed to log on from a workstation in their office
- common method in mobile and distributed computing
- Global Positioning System (GPS) might be used to established the precise geographical location of a user during authentication

7.4 Password security & cracking

Password cracking methods

- Brute force
  - Try all passwords (in a search space) for inverting a specific password hash
  - Eventually succeeds given enough time & CPU power
- Dictionary
  - Precompute & store by hash (hash, password) pairs of a set of likely passwords
  - Fast look up for password given the hash
  - Large storage & preprocessing time
- Rainbow table
  - Partial dictionary of hashes
  - More storage, shorter cracking time

Brute force cracking: Method

- Try all passwords (for a given password space)
- Parallelizable
- Eventually succeeds given enough time & computing power
- Best done with GPUs and specialized hardware (e.g., FPGAs or Asic)
- Large computational effort for each password cracked

Brute force cracking: Search space

Assume a standard keyboard with 94 characters

Password length	Number of passwords
5	$94^5 = 7{,}339{,}040{,}224$
6	$94^6 = 689{,}869{,}781{,}056$
7	$94^7 = 64{,}847{,}759{,}419{,}264$
8	$94^8 = 6{,}095{,}689{,}385{,}410{,}816$
9	$94^9 = 572{,}994{,}802{,}228{,}616{,}704$

Brute force cracking: Computational effort

Say, the attacker has 60 days to crack a password by exhaustive search assuming a standard keyboard of 94 characters.

How many hash computations per second are needed?

- 5 characters: 1,415
- 6 characters: 133,076
- 7 characters: 12,509,214
- 8 characters: 1,175,866,008
- 9 characters: 110,531,404,750

Dictionary attack: Method

- Precompute hashes of a set of likely passwords
- Parallelizable
- Store (hash, password) pairs sorted by hash
- Fast look up for password given the hash
- Requires large storage and preprocessing time

Dictionary attack: Example

STEP 1: Make a plaintext password file of bad passwords (called `wordlist`):

```
triandop12345
letmein
zaq1zaq1
```

STEP 2: Generate MD5 hashes:

```
for i in $(cat wordlist); do
    echo -n "$i" | md5 | tr -d " *-"; done > hashes
```

STEP 3: Get a dictionary file.

E.g., using rockyou.txt which lists most common passwords from the RockYou hack in 2009.

Dictionary attack: Intelligent Guessing

Try the top N most common passwords

- e.g., check out several lists of passwords on known repositories

Try passwords generated by

- a dictionary of words, names, places, notable dates along with
  - combinations of words & replacement/interspersion of digits, symbols, etc.
- a syntax model
  - e.g., 2 words with some letters replaced by numbers: elitenoob, e1iten00b, …
- a Markov chain model or a trained neural network

Password Cracking Tradeoffs

1980 - Martin Hellman

- Achieves (possibly useful) time Vs. memory tradeoffs
- Idea: Reduce time needed to crack a password by using a large amount of memory
  - **Benefits**
    - Better efficiency than brute-forcing methods
  - **Flaws**
    - This kind of database takes tens of memory's terabytes

Password Cracking Tradeoffs (cont.)

Password Cracking Tradeoffs (cont.)

Brute-force: no preprocessing, no storage, very slow cracking

Dictionary: very slow preprocessing, huge storage, very fast cracking

Rainbow tables: **tunable** tradeoff between storage space & cracking time

- Trade more storage for faster cracking

Password space of **size n**

Method	Storage	Preprocessing	Cracking
Brute-force	~ 0	~ 0	n
Dictionary	n	n	~ 0
Rainbow table, $mt^2 = n$	mt	mt^2	$t^2/2$

All costs relate to **hashing**

Rainbow tables

- Use data-structuring techniques to get desirable time Vs. memory tradeoffs
- Main challenge
  - Cryptographic hashing is random and exhibits no patterns
  - E.g., no ordering can be exploited to allow for an efficient search data structure
- Main idea
  - Establish a type of “ordering” by randomly mapping hash values to passwords
  - E.g., via a “reduction” function that produces password “chains”

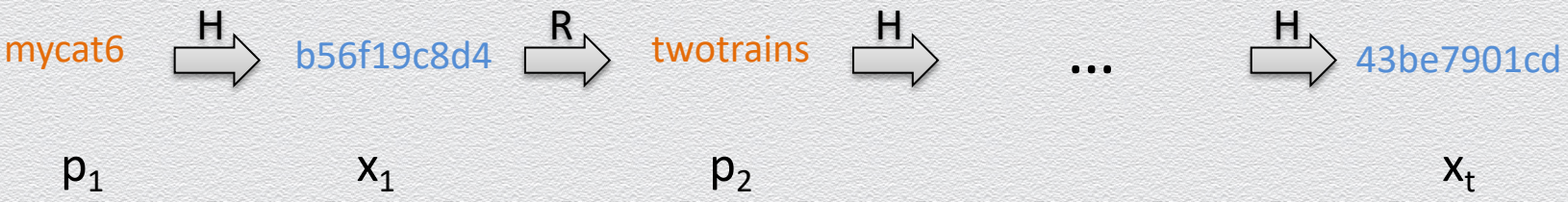
Reduction function

Maps a hash value to a pseudorandom password from a given password space

- E.g., reduction function p = R(x) for 256-bit hashes & 8-character passwords from a 64-symbol alphabet a_1, a_2, …, a_{64}
  - Split hash x into 48-bit blocks x_1, x_2, …, x_5 and one 16-bit block x_6
  - Compute $y = x_1 \oplus x_2 \ldots \oplus x_5$
  - Split y into 6-bit blocks y_1, y_2, …, y_8
  - Let $p = a_{y1}, a_{y2}, \ldots, a_{y8}$
- This method can be generalized to arbitrary password spaces

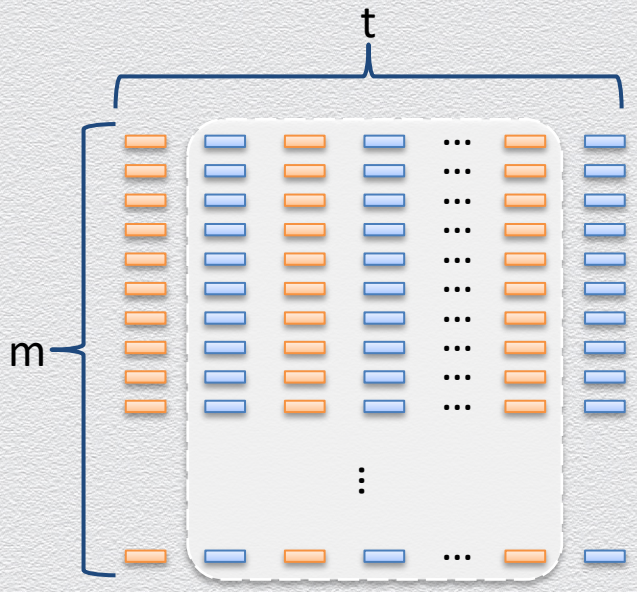
Password chain

- Sequence (of size t) alternating **passwords** & **hashes**
  - Start with a random password p_1
  - Alternate using cryptographic hash function H & reduction function R
    - $x_i = H(p_i)$, $p_{i+1} = R(x_i)$
  - End with a hash value x_t



Hellman's method

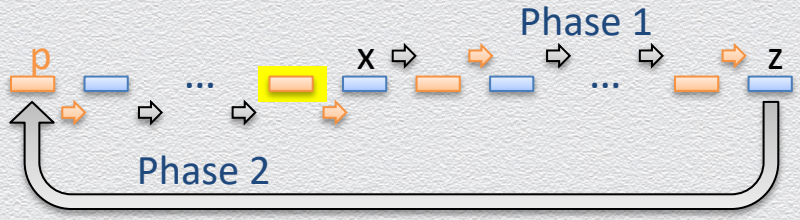
- Starting from m random passwords, build a table of m password chains, each of length t
- The expected number of distinct passwords in a table is $\Omega(mt)$
- Compressed storage:
  - For each chain, keep only the first password, p, and the last hash value, z
  - Store pairs (z, p) in a dictionary D indexed by hash value z



Classic password recovery

Recovery of password with hash value x

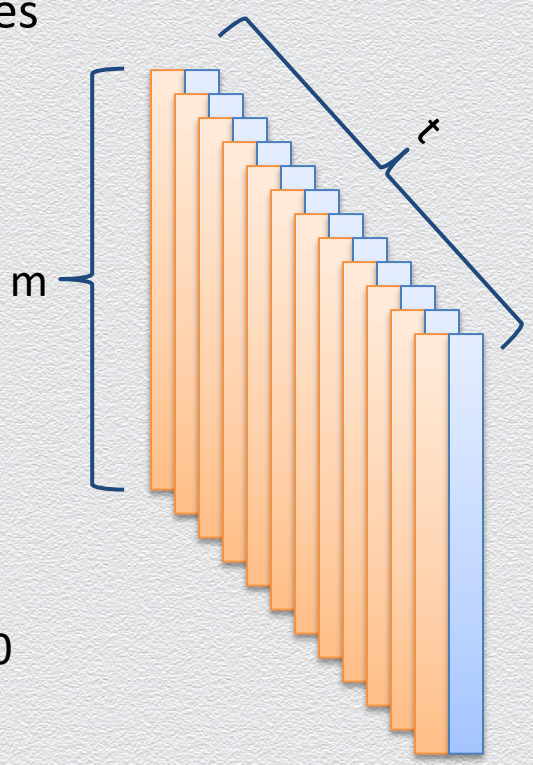
- Step 1: traverse the suffix of the chain starting at x
  - y = x;
  - while p = D.get(y) is null
    - y = H(R(y)) //advance
    - if i++ > t return “failure” //x not in the table
- Step 2: traverse the prefix of the chain ending at x
  - while y = H(p) ≠ x
    - p = R(y) //advance
    - if j++ > t return “failure” //x not in the table
  - return p //password recovered



High-probability recovery

Collisions in the reduction function result in recovery issues

- Mitigate the impact of collisions, using t tables with distinct reduction functions R
- If $m \cdot t^2 = O(n)$, n passwords are covered with high probability



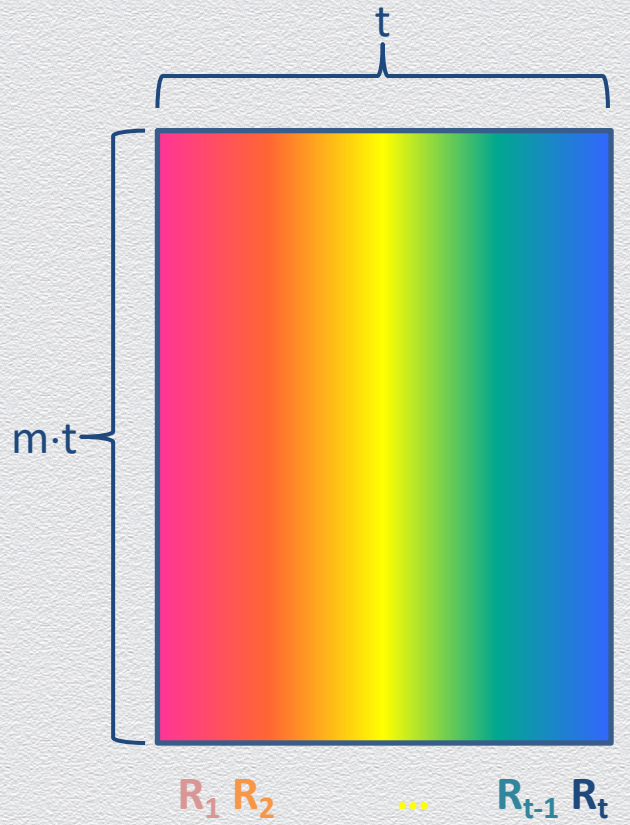
Performance

- Storage: mt cryptographic hash values
- Recovery: t^2 hash computations & t^2 dictionary lookups
- E.g., $n = 1{,}000{,}000{,}000$, $m = t = n^{1/3}$, $mt = t^2 = n^{2/3} = 1{,}000{,}000$

Rainbow table

Instead of t different tables, use a single table with

- O(m·t) chains of length t
- Distinct reduction function at each step
- Visualizing the reduction functions with a gradient of colors yields a **rainbow**



Performance

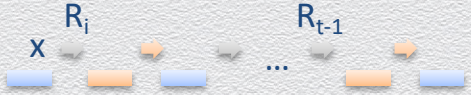
- Storage : mt hash values (as before)
- Recovery : $t^2/2$ hash computations & t dictionary lookups (lower than before)

Rainbow-table password recovery

```
for i = t, (t – 1), … , 1
    y = x //x is password hash we want to crack
    for j = i, …, t - 1 //traverse from i to t
        y = H(Rj(y)) //advance
    if p = D.get(y) is not null //candidate position i
        for j = 1 … i - 1 //traverse from 1 to i
            p = Rj(H(p)) //advance
        if H(p) = x   return p //password recovered
        else return “failure” //x not in the table
return “failure” //x not in the table
```

Final loop: from 1 to i

Inner loop: from i to t



Worst-case # of hashing

$1 + 2 + \ldots + (t-1) + 1 \approx t^2/2$