https://brown-csci1660.github.io

CS1660: Intro to Computer Systems Security
Spring 2025

Lecture 7: Cryptography VI

Co-Instructor: **Nikos Triandopoulos**

February 13, 2025

BROWN

CS1660: Announcements

- Course updates
  - Project 1 is due in 1 week from today (Thu, Feb 20)
  - Homework 1 is out and due in 2 weeks from today (Thu, Feb 27)
  - Ed Discussion, Top Hat (code: 821033), Gradescope (set up for Project 1 & HW1)
  - Where we are
    - ✓ Part I: Crypto – we will have a revision in our next class (Thu, Feb 27)
    - Part II: Web
    - Part III: OS
    - Part IV: Network
    - Part V: Extras

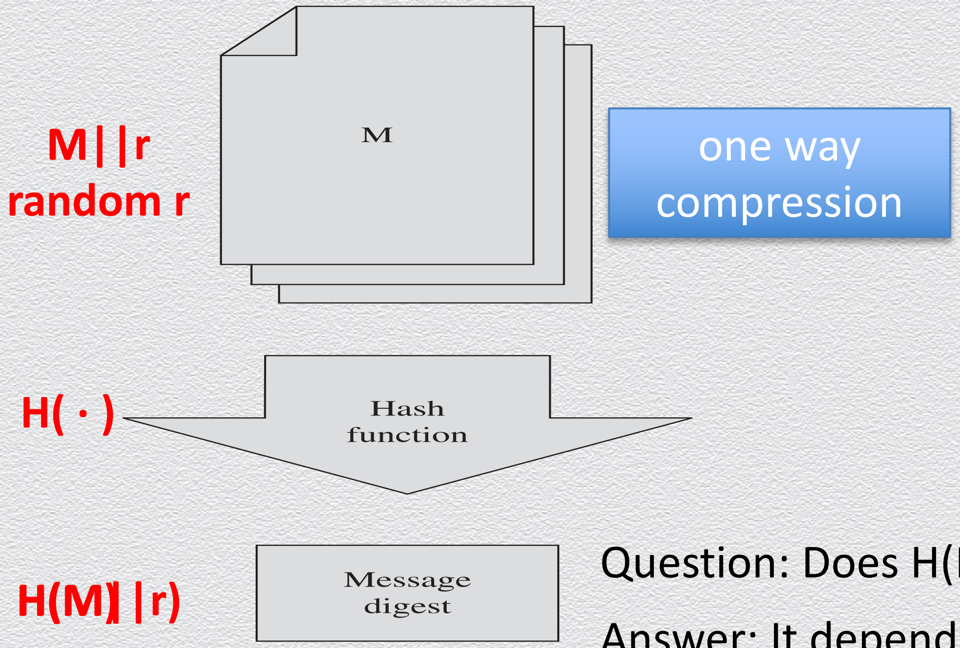
Today

- Cryptography
  - Hash functions
    - Applications to security
  - Public-key crypto
    - RSA crypto system
    - DH key agreement protocol
  - Authentication
    - System & user authentication
    - Passwords & password cracking

7.0 Cryptographic hash functions: Applications to security

Generally: Message digest (= hash value = fingerprint)

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



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

<u>other than</u> 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

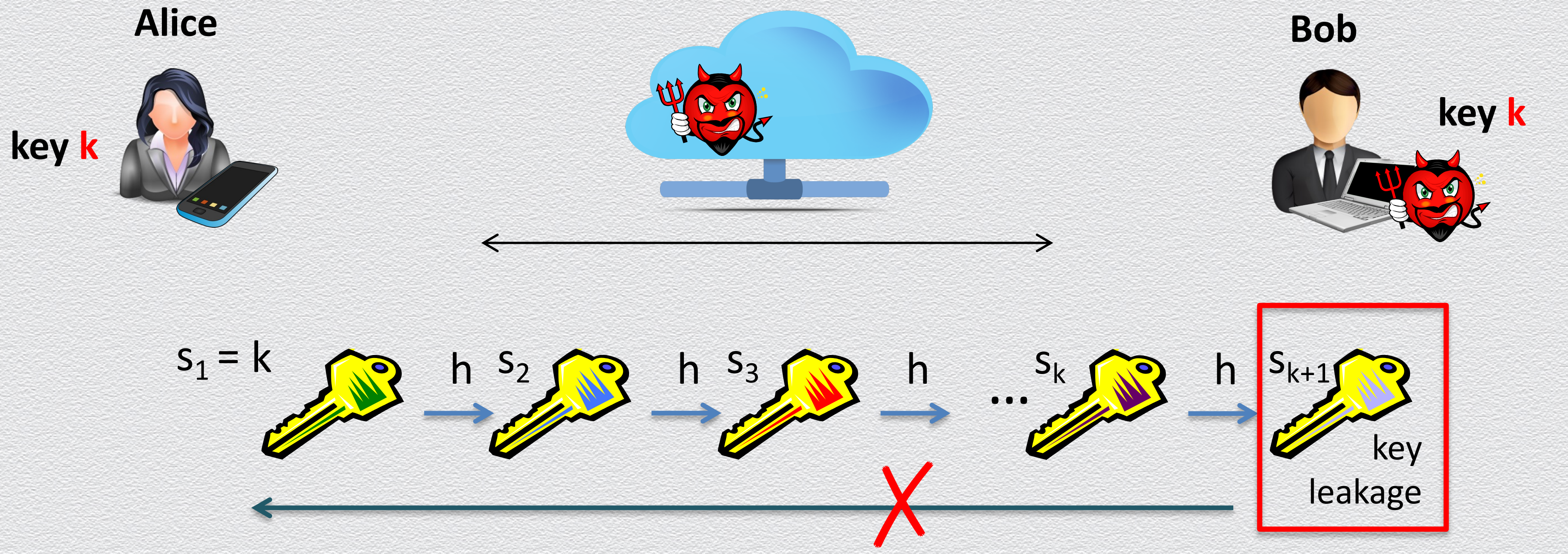
Security

- 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

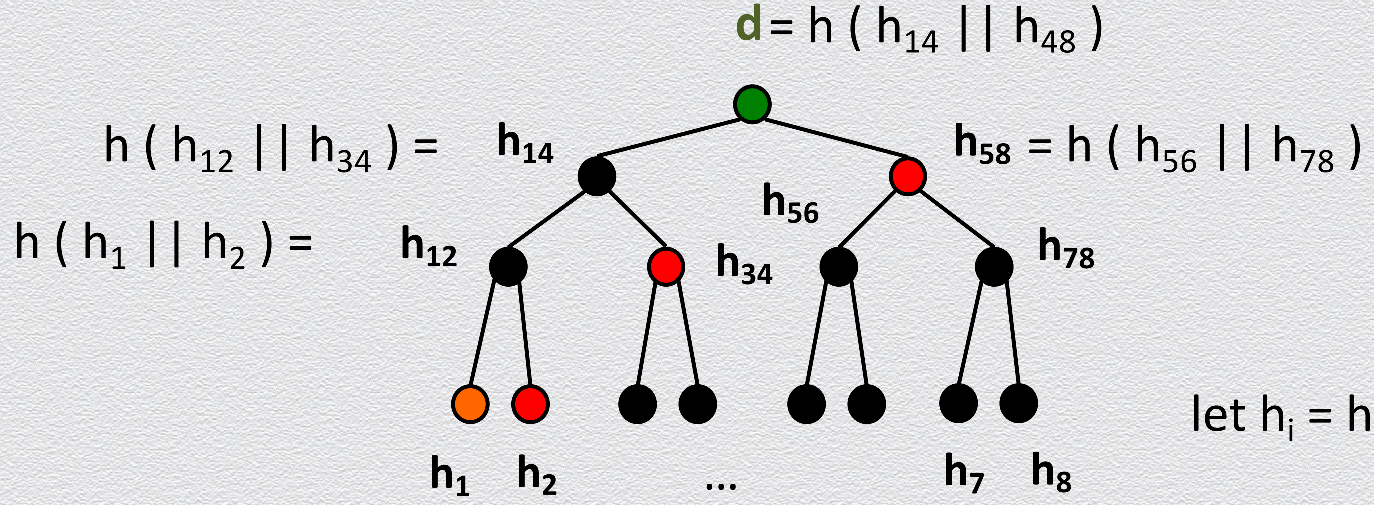
- let h be a cryptographic hash function, assume RSA setting (n, d, e)
- compute signature $\sigma = h(M)^d \bmod n$
- send σ, M

Verifying signature σ

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

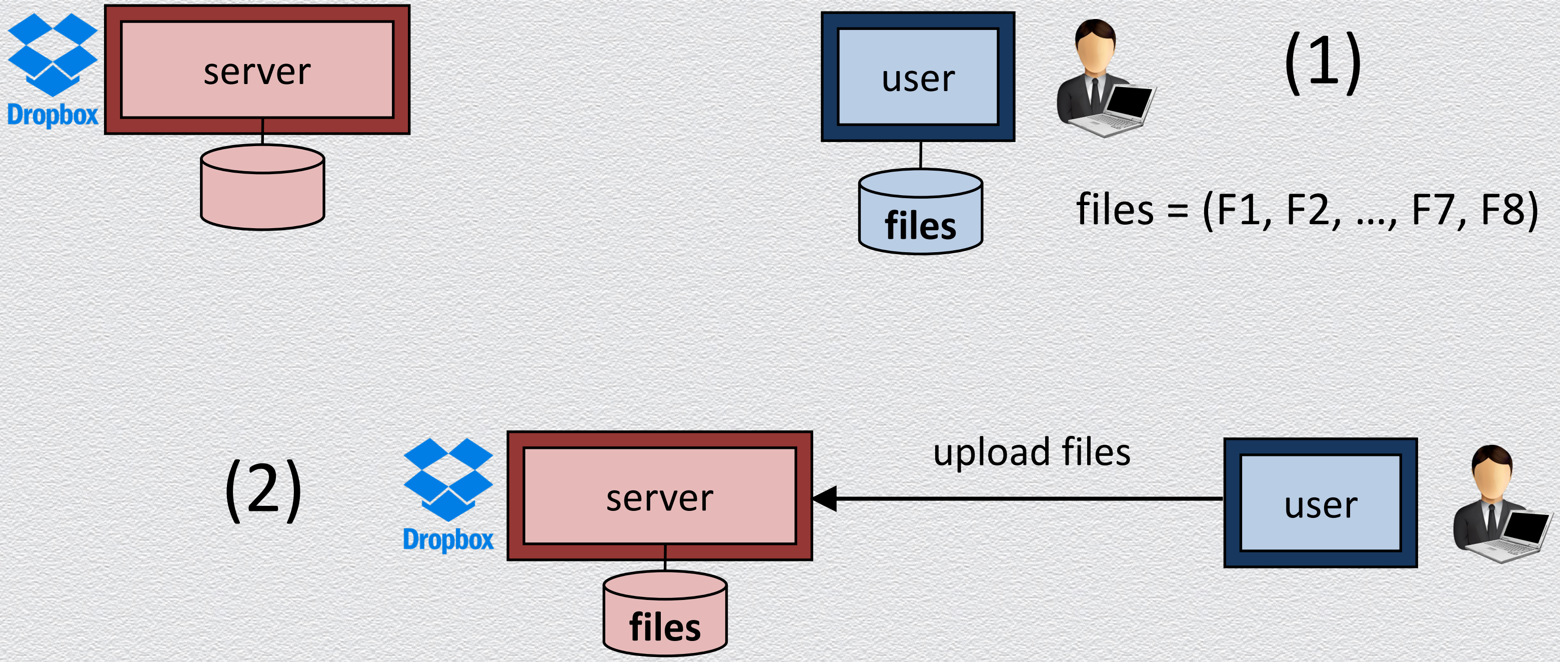
Application 6: The Merkle tree

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

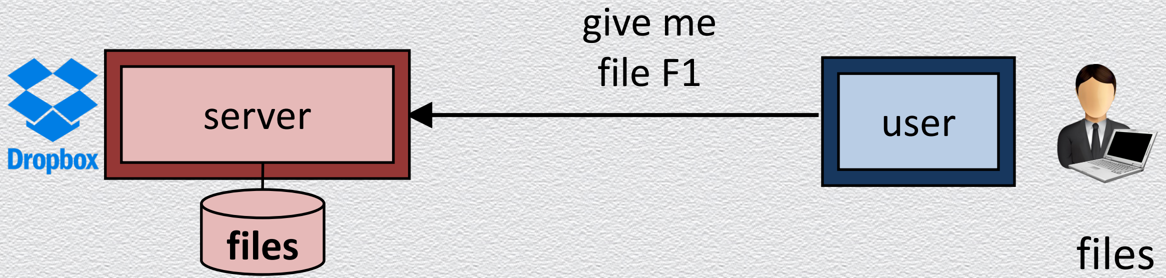


let $h_i = h(F_i)$, $1 \leq i \leq n$

Example 6: Secure cloud storage



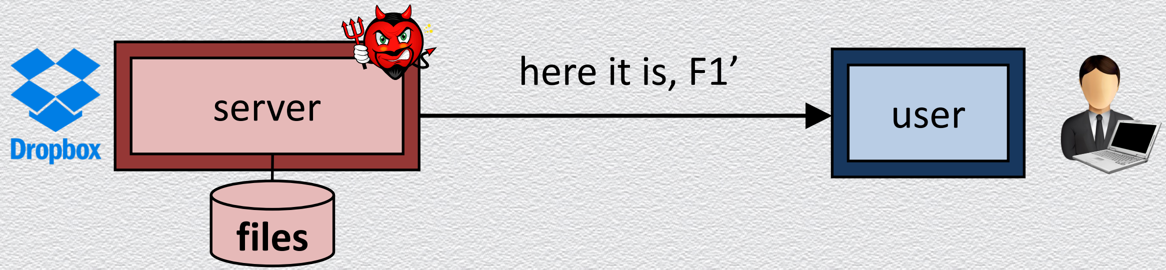
Example 6: Secure cloud storage



(3)

files = (F1, F2, …, F7, F8)

(4)

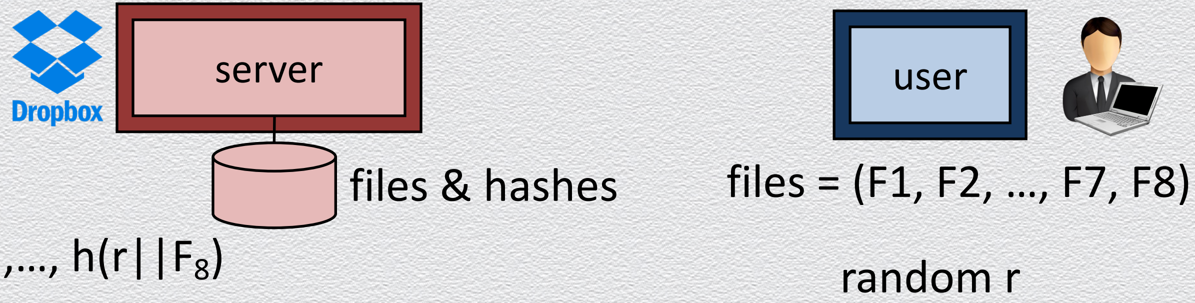


F1’ = #@$@!#^@$^… (altered)

Example 6: Secure cloud storage – per-file hashing

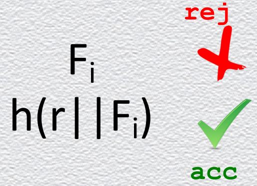
Bob wants to outsource storage of files F_1, F_2,…,F_8 to Dropbox & check their integrity

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

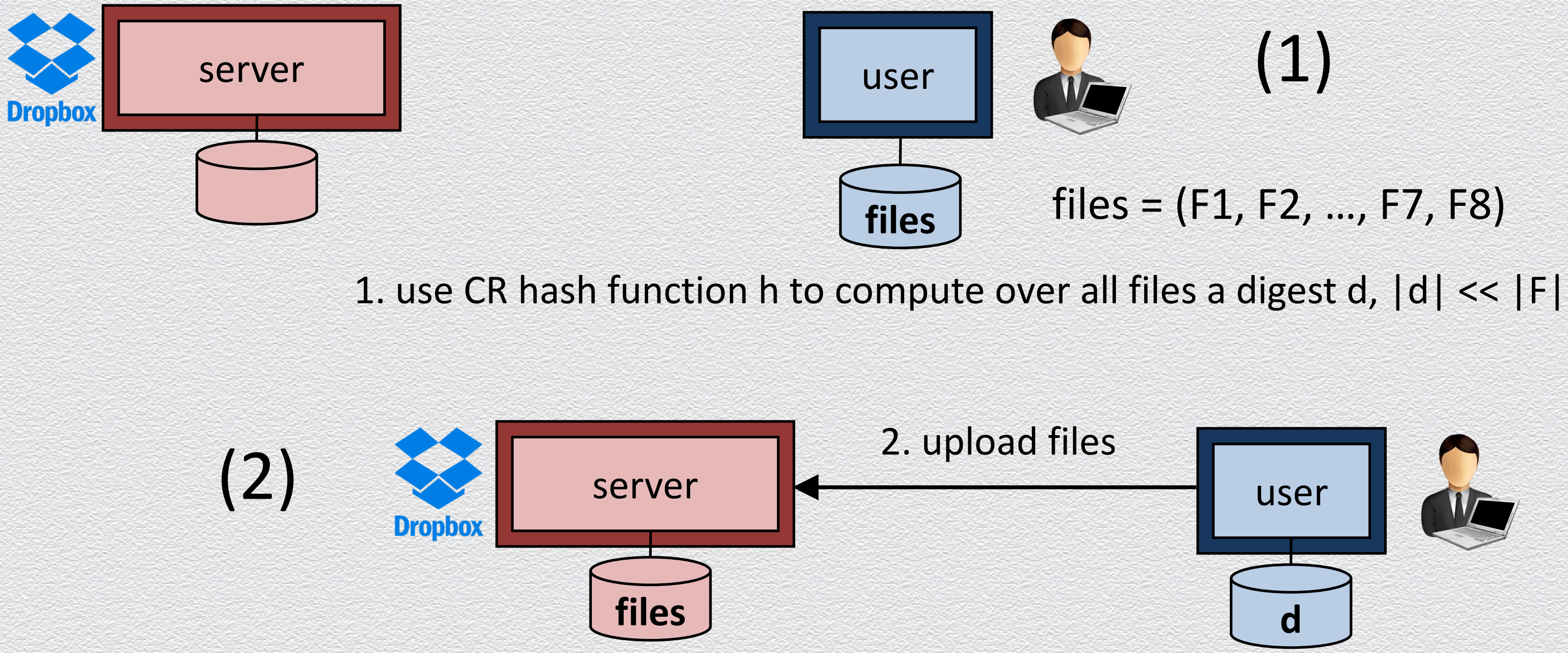


Every time Bob **reads** a file F_i, he also reads $h(r||F_i)$ to verify F_i's integrity

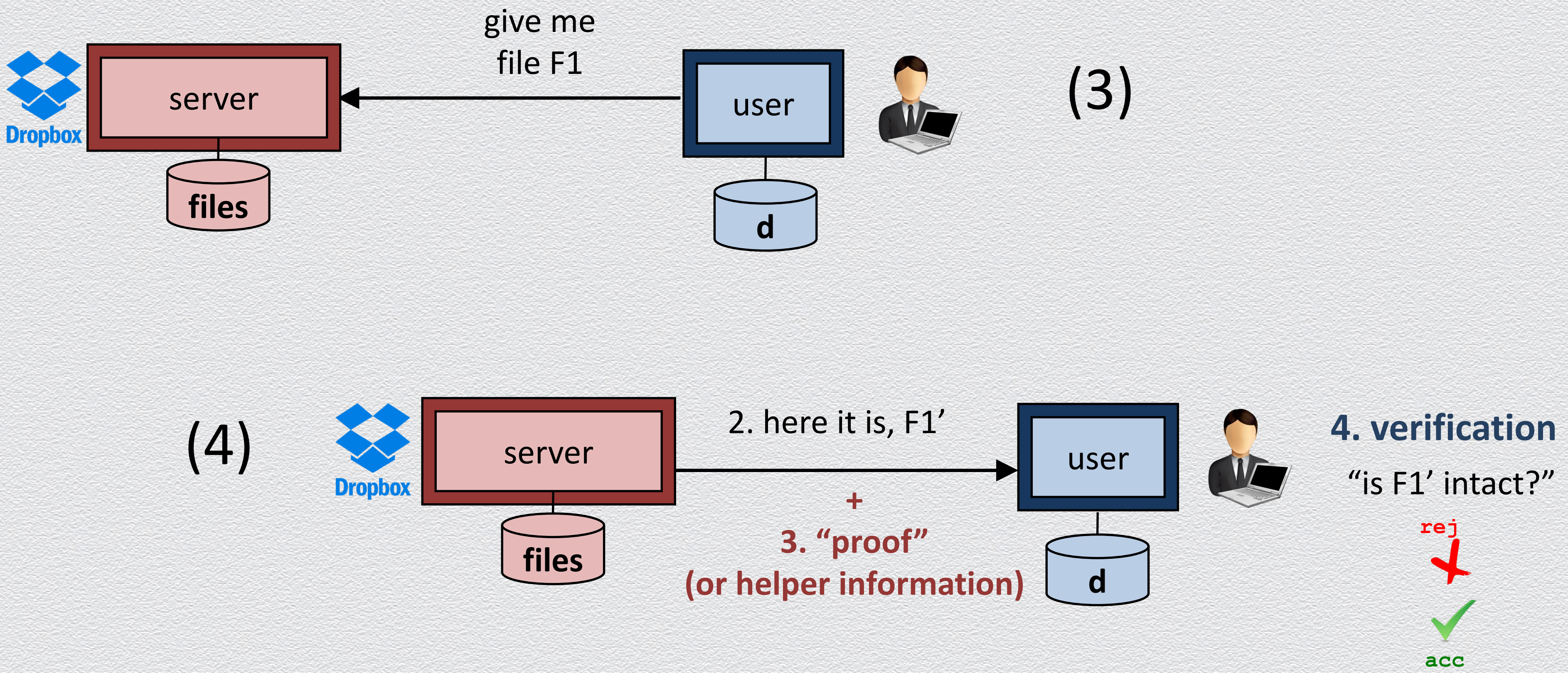
- any problems with **writes**?



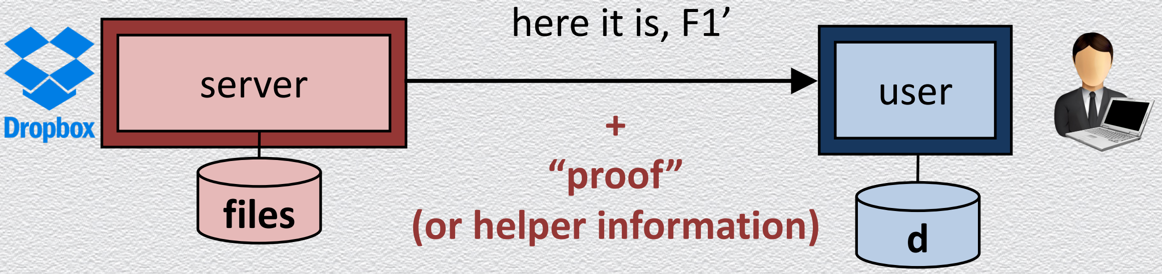
Example 6: Secure cloud storage – per-file-set hashing



Example 6: Secure cloud storage – integrity checking



Example 6: Secure cloud storage – verification



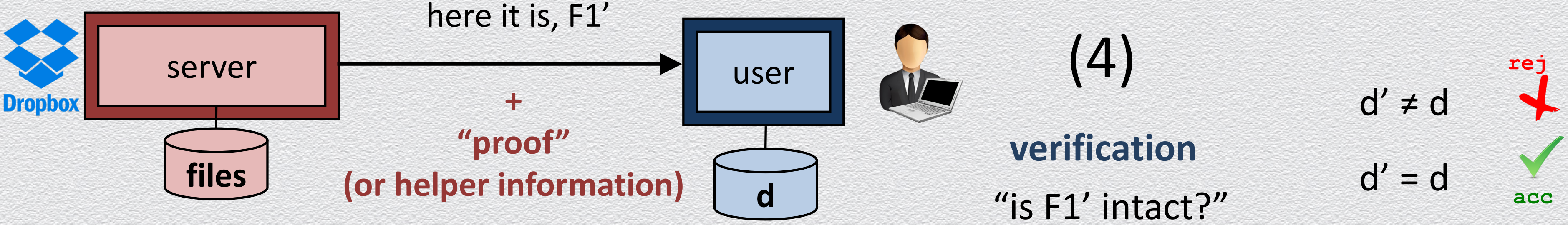
(4)

verification

“is F1’ intact?”

- 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!

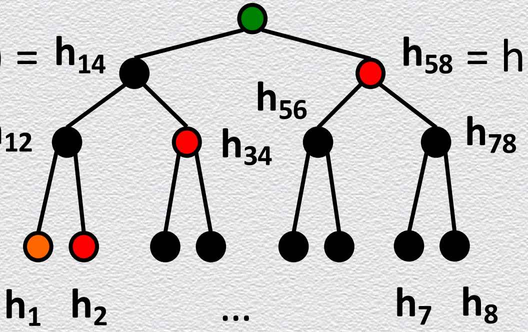
Example 6: Data authentication via the Merkle tree



digest is the **green** root hash $d = h(h_{14} \,||\, h_{48})$

$h(h_{12} \,||\, h_{34}) = h_{14}$ $\quad h_{58} = h(h_{56} \,||\, h_{78})$

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



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

answer is **orange** hash

proof is **red** hash path

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

7.1 Number theory (for public key crypto)

Multiplicative inverses

The residues modulo a positive integer n comprise set $Z_n = \{0,1,2,\ldots,n - 1\}$

- let x and y be two elements in Z_n such that x y mod n = 1
  - we say: y is the multiplicative inverse of x in Z_n
  - we write: $y = x^{-1}$

Theorem

An element x in Z_n has a multiplicative inverse iff x, n are relatively prime

Multiplicative inverses (cont.)

- e.g., multiplicative inverses of the residues **modulo 10** are 1, 3, 7, 9

x	0	1	2	3	4	5	6	7	8	9
x^{-1}		1		7				3		9

- e.g., multiplicative inverses of the residues **modulo 11** are all non-zero elements

x	0	1	2	3	4	5	6	7	8	9	10
x^{-1}		1	6	4	3	9	2	8	7	5	10

Computing multiplicative inverses

Fact

- given two numbers **a** and **b**, there exist integers x, y s.t.

$$\mathbf{x\ a + y\ b = gcd(a,b)}$$

which can be computed efficiently by the extended Euclidean algorithm.

Thus

- the multiplicative inverse of a in Z_b exists iff gcd(a, b) = 1
- i.e., iff the extended Euclidean algorithm computes x and y s.t. $\mathbf{x\ a + y\ b = 1}$
- in this case, the multiplicative inverse of a in Z_b is **x**

Euclidean GCD algorithm

Computes the greater common divisor by repeatedly applying the formula

gcd(a, b) = gcd(b, a mod b)

- example
  - gcd(412, 260) = 4

```
Algorithm EuclidGCD(a, b)
    Input integers a and b
    Output gcd(a, b)

    if b = 0
        return a
    else
        return EuclidGCD(b, a mod b)
```

a	412	260	152	108	44	20	4
b	260	152	108	44	20	4	0

Extended Euclidean algorithm

Theorem

If, given positive integers **a** and **b**, **d** is the smallest positive integer s.t. **d** = **ia** + **jb**, for some integers **i** and **j**, then **d** = gcd(**a, b**)

- example
  - **a** = 21, **b** = 15
  - **d** = 3, **i** = 3, **j** = -4
  - 3 = 3·21 + (-4)·15 = 63 - 60 = 3

```
Algorithm Extended-Euclid(a, b)
  Input integers a and b
  Output gcd(a, b), i and j
              s.t. ia+jb = gcd(a,b)
  if b = 0
     return (a,1,0)
  (d′, x′, y′) = Extended-Euclid(b, a mod b)
  (d, x, y) = (d′, y′, x′ - [a/b]y′)
  return (d, x, y)
```

Multiplicative group

A set of elements where multiplication • is defined

- closure, associativity, identity & inverses
- multiplicative groups Z^*_n, defined w.r.t. Z_n (residues modulo n)
  - subsets of Z_n containing all integers that are relative prime to n
  - **CASE 1**: if n is a prime number, then all non-zero elements in Z_n have an inverse
    - $Z^*_7 = \{1,2,3,4,5,6\}$, n = 7
    - 2 • 4 = 1 (mod 7), 3 • 5 = 1 (mod 7), 6 • 6 = 1 (mod 7), 1 • 1 = 1 (mod 7)
  - **CASE 2**: if n is not prime, then not all integers in Z_n have an inverse
    - $Z^*_{10} = \{1,3,7,9\}$, n = 10
    - 3 • 7 = 1 (mod 10), 9 • 9 = 1 (mod 10), 1 • 1 = 1 (mod 10)

Order of a multiplicative group

Order of a group = cardinality of the group

- multiplicative groups for Z^*_n
- the totient function $\phi(n)$ denotes the order of Z^*_n , i.e., $\phi(n) = |Z^*_n|$
  - if **n = p is prime**, then the order of $Z^*_p=\{1,2,\ldots,p-1\}$ is p-1, i.e., $\phi(n) = p-1$
    - e.g., $Z^*_7 = \{1,2,3,4,5,6\}$, n = 7, $\phi(7) = 6$
  - if **n is not prime**, $\phi(n) = n(1-1/p_1)(1-1/p_2)\ldots(1-1/p_k)$, where $n = p^{e1}{}_1 p^{e2}{}_2 \ldots p^{ek}{}_k$
    - e.g., $Z^*_{10} = \{1,3,7,9\}$, n = 10, $\phi(10) = 4$

- if n = p q, where p and q are distinct primes, then $\phi(n) = (p-1)(q-1)$ **Factoring problem**
  - difficult problem: given n = pq, where p, q are primes, find p and q or $\phi(n)$

Fermat's Little Theorem

Theorem

If **p is a prime**, then for each nonzero residue x in Z_p, we have $x^{p-1} \bmod p = 1$

- example (p = 5):

  $1^4 \bmod 5 = 1$

  $2^4 \bmod 5 = 16 \bmod 5 = 1$

  $3^4 \bmod 5 = 81 \bmod 5 = 1$

  $4^4 \bmod 5 = 256 \bmod 5 = 1$

Corollary

If **p is a prime**, then the multiplicative inverse of each x in Z^*_p is $x^{p-2} \bmod p$

- proof: $x(x^{p-2} \bmod p) \bmod p = xx^{p-2} \bmod p = x^{p-1} \bmod p = 1$

Euler's Theorem

Theorem

For each element x in Z^*_n, we have $x^{\phi(n)} \bmod n = 1$

- example (**n = 10**)
  - $Z^*_{10} = \{1,3,7,9\}$, n = 10, $\phi(10) = 4$
  - $3^{\phi(10)} \bmod 10 = 3^4 \bmod 10 = 81 \bmod 10 = 1$
  - $7^{\phi(10)} \bmod 10 = 7^4 \bmod 10 = 2401 \bmod 10 = 1$
  - $9^{\phi(10)} \bmod 10 = 9^4 \bmod 10 = 6561 \bmod 10 = 1$

Computing in the exponent

For the multiplicative group Z^*_n, we can reduce the exponent modulo ϕ(n)

- $x^y \bmod n = x^{k\,\phi(n) + r} \bmod n = (x^{\phi(n)})^k x^r \bmod n = x^r \bmod n = x^{y \bmod \phi(n)} \bmod n$

Corollary: For Z^*_p, we can reduce the exponent modulo p-1

- example
  - $Z^*_{10} = \{1,3,7,9\}$, $n = 10$, $\phi(10) = 4$
  - $3^{1590} \bmod 10 = 3^{1590 \bmod 4} \bmod 10 = 3^2 \bmod 10 = 9$
- example
  - $Z^*_p = \{1,2,\ldots,p - 1\}$, $p = 19$, $\phi(19) = 18$
  - $15^{39} \bmod 19 = 15^{39 \bmod 18} \bmod 19 = 15^3 \bmod 19 = 12$

Modular powers

Repeated squaring algorithm

Speeds up computation of $\mathbf{a}^{\mathbf{p}}$ mod $\mathbf{n}$

- write the exponent $\mathbf{p}$ in binary
  $\mathbf{p} = \mathbf{p}_{\mathbf{b}-1}\,\mathbf{p}_{\mathbf{b}-2} \ldots \mathbf{p}_1\,\mathbf{p}_0$
- start with $\mathbf{Q}_1 = \mathbf{a}^{\mathbf{p}_{\mathbf{b}-1}}$ mod $\mathbf{n}$
- repeatedly compute
  $\mathbf{Q}_\mathbf{i} = ((\mathbf{Q}_{\mathbf{i}-1})^2 \bmod \mathbf{n})\mathbf{a}^{\mathbf{p}_{\mathbf{b}-\mathbf{i}}} \bmod \mathbf{n}$
- obtain $\mathbf{Q}_\mathbf{b} = \mathbf{a}^\mathbf{p}$ mod $\mathbf{n}$

Total **O** (log **p**) arithmetic operations

Example

- 3^{18} mod 19 (18 = 10010)
- $\mathbf{Q}_1 = 3^1$ mod 19 = 3
- $\mathbf{Q}_2 = (3^2$ mod 19)3^0 mod 19 = 9
- $\mathbf{Q}_3 = (9^2$ mod 19)3^0 mod 19 = 81 mod 19 = 5
- $\mathbf{Q}_4 = (5^2$ mod 19)3^1 mod 19 =
  (25 mod 19)3 mod 19 = 18 mod 19 = 18
- $\mathbf{Q}_5 = (18^2$ mod 19)3^0 mod 19 = (324 mod 19) mod 19 = $17 \cdot 19 + 1$ mod 19 = 1

Powers

Let p be a prime

- the sequences of successive powers of the elements in Z^*_p exhibit repeating subsequences
- the sizes of the repeating subsequences and the number of their repetitions are the divisors of p – 1
- example, p = 7

x	x^2	x^3	x^4	x^5	x^6
1	1	1	1	1	1
2	4	1	2	4	1
3	2	6	4	5	1
4	2	1	4	2	1
5	4	6	2	3	1
6	1	6	1	6	1

7.1.1 DH key agreement

Computational assumption

Discrete-log setting

- cyclic $G = (Z_p^*, \cdot)$ of order $p - 1$ generated by g, prime p of length t ($|p|=t$)

Problem

- given G, g, p and x in Z_p^*, compute the discrete log k of x (mod p)
- we know that $x = g^k \bmod p$ for some unique k in {0, 1, …, p-2}… but

Discrete log assumption

- for groups of specific structure, **solving the discrete log problem is infeasible**
- any efficient algorithm finds discrete logs negligibly often (prob = $2^{-t/2}$)

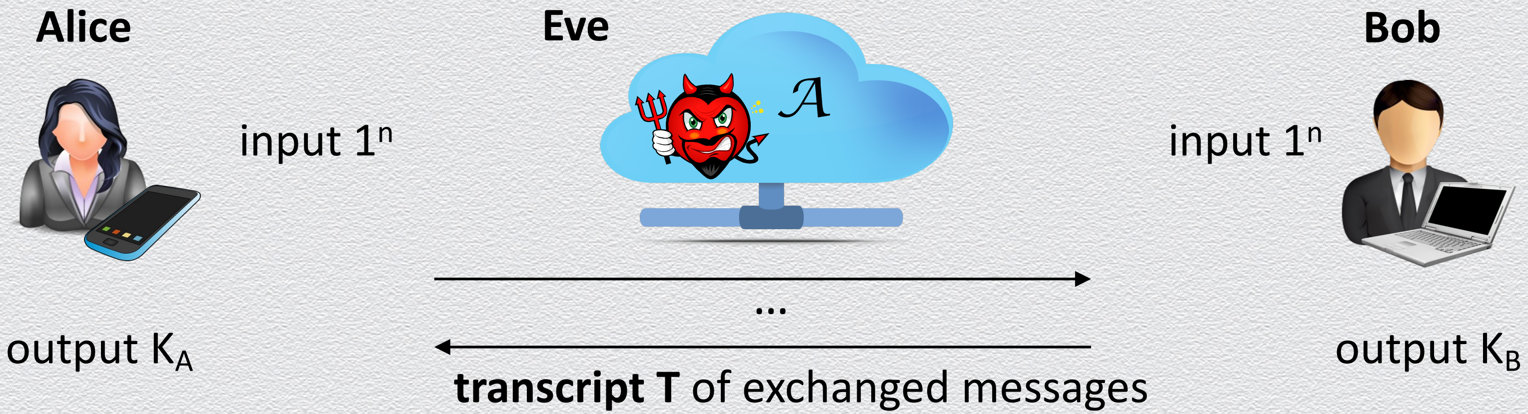
Brute force attack

- cleverly enumerate and **check $O(2^{t/2})$ solutions**

Application: Key-agreement (KA) scheme

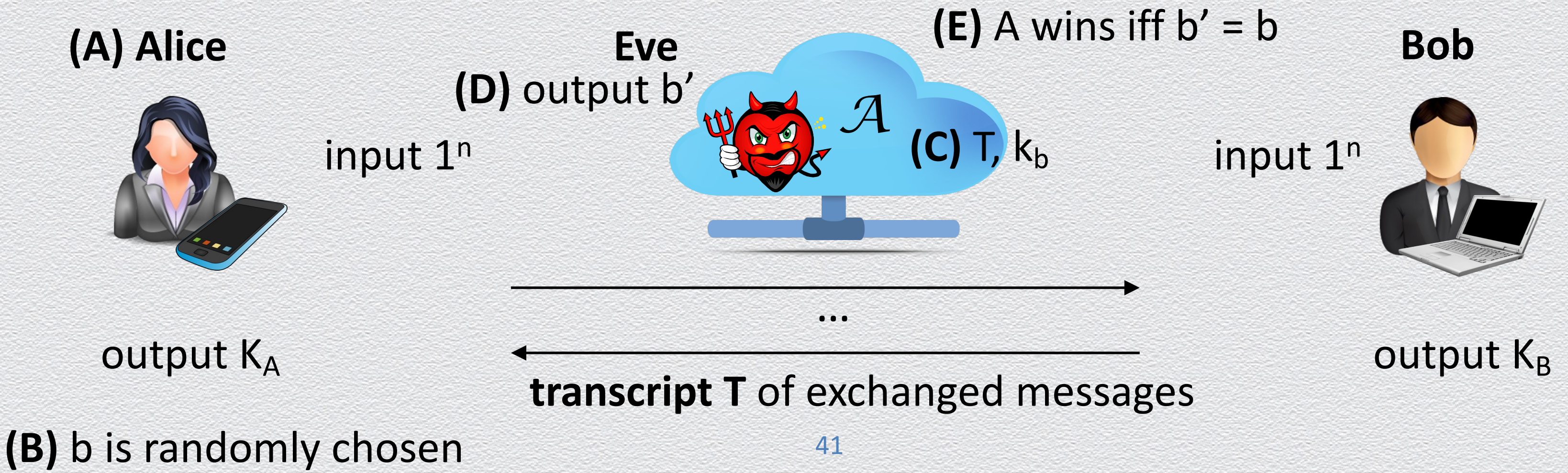
Alice and Bob want to securely establish a **shared key** for secure chatting over an **insecure** line

- instead of meeting in person in a secret place, they want to use the insecure line…
- KA scheme: they run a key-agreement protocol Π to contribute to a shared key K
- correctness: $K_A = K_B$
- security: no PPT adversary $\mathcal{A}$, given T, can distinguish K from a trully random one



Key agreement: Game-based security definition

- scheme $\Pi(1^n)$ runs to generate $K = K_A = K_B$ and transcript T; random bit b is chosen
- adversary $\mathcal{A}$ is given T and k_b; if b = 1, then k_b = K, else k_b is random (both n-bit long)
- $\mathcal{A}$ outputs bit b' and wins if b' = b
- then: **Π is secure if no PPT A wins non-negligibly often**



The Diffie-Hellman key-agreement protocol

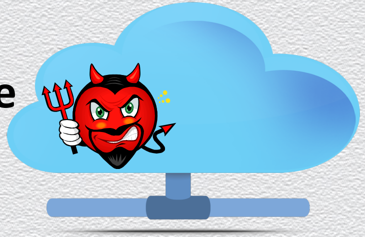
Alice and Bob want to securely establish a **shared key** for secure chatting over an **insecure** line

- DH KA scheme Π
  - discrete log setting: p, g public, where <g> = Z^*_p and p prime

Alice

input 1^n

Eve



Bob

input 1^n

(1) randomly pick secret a

(2) randomly pick secret b

→ **(3)** send g^a mod p

← **(4)** send g^b mod p

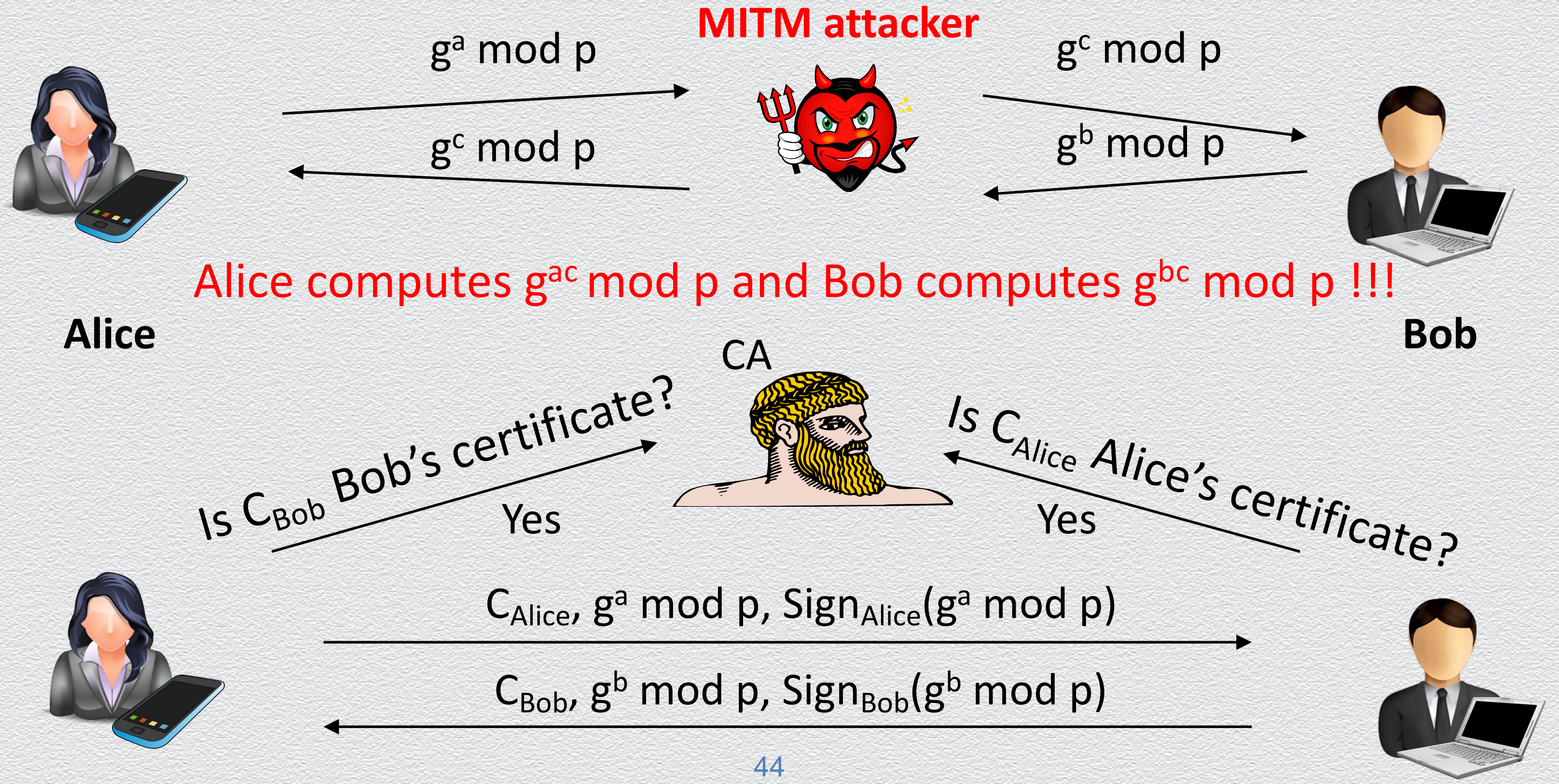
(5) set **K = g^{ab} mod p** = $(g^b \bmod p)^a \bmod p$

(6) set **K = g^{ab} mod p** = $(g^a \bmod p)^b \bmod p$

Security

- discrete log assumption is necessary but not sufficient
- decisional DH assumption
  - given g, g^a and g^b, g^{ab} is computationally indistinguishable from uniform

Authenticated Diffie-Hellman



7.1.2 The RSA algorithm

The RSA algorithm (for encryption)

General case

Setup (run by a given user)

- $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, with **p** and **q** primes
- **e** relatively prime to $\boldsymbol{\phi}(n) = (\mathbf{p} - 1)(\mathbf{q} - 1)$
- **d** inverse of **e** in $\mathbf{Z}_{\boldsymbol{\phi}(\mathbf{n})}$

Keys

- public key is $\mathbf{K_{PK}} = (\mathbf{n}, \mathbf{e})$
- private key is $\mathbf{K_{SK}} = \mathbf{d}$

Encryption

- $\mathbf{C} = \mathbf{M}^{e} \bmod \mathbf{n}$ for plaintext **M** in $\mathbf{Z_n}$

Decryption

- $\mathbf{M} = \mathbf{C}^{\mathbf{d}} \bmod \mathbf{n}$

Example

Setup

- $\mathbf{p} = 7$, $\mathbf{q} = 17$, $\mathbf{n} = 7 \cdot 17 = 119$
- $\mathbf{e} = 5$, $\boldsymbol{\phi}(\mathbf{n}) = 6 \cdot 16 = 96$
- $\mathbf{d} = 77$

Keys

- public key is (119, 5)
- private key is 77

Encryption

- $\mathbf{C} = 19^5 \bmod 119 = 66$ for $\mathbf{M} = 19$ in $\mathbf{Z_{119}}$

Decryption

- $\mathbf{M} = 66^{77} \bmod 119 = 19$

Another complete example

- Setup
  - **p** = 5, **q** = 11, **n** = 5 · 11 = 55
  - **ϕ(n)** = 4 · 10 = 40
  - **e** = 3, **d** = 27   (3·27 = 81 = 2·40 + 1)
- Encryption
  - **C** = **M**^{3} mod 55 for **M** in $\mathbf{Z}_{55}$
- Decryption
  - **M** = **C**^{27} mod 55

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
M	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
C	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
M	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
C	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

Correctness of RSA

Given

Setup

- $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, with **p** and **q** primes
- **e** relatively prime to $\phi(n) = (\mathbf{p} - 1)(\mathbf{q} - 1)$
- **d** inverse of **e** in $\mathbf{Z}_{\phi(n)}$ **(1)**

Encryption

- $\mathbf{C} = \mathbf{M}^e \bmod \mathbf{n}$ for plaintext **M** in $\mathbf{Z}_n$

Decryption

- $\mathbf{M} = \mathbf{C}^d \bmod \mathbf{n}$

Fermat's Little Theorem **(2)**

- for prime p, non-zero x: $x^{p-1} \bmod p = 1$

Analysis

Need to show

- $M^{ed} = M \bmod p \cdot q$

Use **(1)** and apply **(2)** for prime p

- $M^{ed} = M^{ed-1} M = (M^{p-1})^{h(q-1)} M$
- $\mathbf{M^{ed}} = 1^{h(q-1)} M \bmod p = \mathbf{M \bmod p}$

Similarly (w.r.t. prime q)

- $\mathbf{M^{ed} = M \bmod q}$

Thus, since p, q are co-primes

- $\mathbf{M^{ed} = M \bmod p \cdot q}$

A useful symmetry

[1] RSA setting

- modulo $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, p & q are **primes**, public & private keys (e,d): $\mathbf{d \cdot e = 1 \bmod (p-1)(q-1)}$

[2] RSA operations involve **exponentiations**, thus they are **interchangeable**

- $\mathbf{C} = \mathbf{M}^e \bmod \mathbf{n}$ (encryption of plaintext **M in $\mathbf{Z_n}$**)
- $\mathbf{M} = \mathbf{C}^d \bmod \mathbf{n}$ (decryption of ciphertext **C in $\mathbf{Z_n}$**)

Indeed, their order of execution does not matter: $\mathbf{(M^e)^d = (M^d)^e \bmod n}$

[3] RSA operations involve exponents that **"cancel out"**, thus they are **complementary**

- $\mathbf{x}^{(p-1)(q-1)} \bmod \mathbf{n} = \mathbf{1}$ (Euler's Theorem)

Indeed, they invert each other: $\mathbf{(M^e)^d = (M^d)^e = M^{ed} = M^{k(p-1)(q-1)+1} \bmod n}$

$\mathbf{= (M^{(p-1)(q-1)})^k \cdot M = 1^k \cdot M = M \bmod n}$

Signing with RSA

RSA functions are complementary & interchangeable w.r.t. order of execution

- **core property**: $M^{ed} = M \bmod p \cdot q$ for any message M in Z_n

RSA cryptosystem lends itself to a **signature scheme**

- 'reverse' use of keys is possible : $(M^d)^e = M \bmod p \cdot q$
- signing algorithm **Sign(M,d,n)**: $\sigma = M^d \bmod n$ for message M in Z_n
- verifying algorithm **Vrfy(σ,M,e,n)**: return $M == \sigma^e \bmod n$

The RSA algorithm (for signing)

General case

Setup (run by a given user)

- $\mathbf{n} = \mathbf{p} \cdot \mathbf{q}$, with **p** and **q** primes
- **e** relatively prime to $\boldsymbol{\phi}(n) = (\mathbf{p} - 1)(\mathbf{q} - 1)$
- **d** inverse of **e** in $\mathbf{Z}_{\boldsymbol{\phi}(\mathbf{n})}$

Keys (same as in encryption)

- public key is $\mathbf{K_{PK}} = (\mathbf{n}, \mathbf{e})$
- private key is $\mathbf{K_{SK}} = \mathbf{d}$

Sign

- $\boldsymbol{\sigma} = \mathbf{M}^{\mathbf{d}}$ mod **n** for message **M** in $\mathbf{Z}_{\mathbf{n}}$

Verify

- Check if $\mathbf{M} = \boldsymbol{\sigma}^{\mathbf{e}}$ mod **n**

Example

Setup

- $\mathbf{p} = 7$, $\mathbf{q} = 17$, $\mathbf{n} = 7 \cdot 17 = 119$
- $\mathbf{e} = 5$, $\boldsymbol{\phi}(\mathbf{n}) = 6 \cdot 16 = 96$
- $\mathbf{d} = 77$

Keys

- public key is (119, 5)
- private key is 77

Signing

- $\boldsymbol{\sigma} = 66^{77}$ mod $119 = 19$ for $\mathbf{M} = 66$ in $\mathbf{Z}_{119}$

Verification

- Check if $\mathbf{M} = 19^5$ mod $119 = 66$

Digital signatures & hashing

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 σ on message M as: $\sigma = h(M)^d \bmod n$
- send σ, M

Verifying signature σ

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

Security of RSA

Based on difficulty of **factoring** large numbers (into large primes), i.e., $n = p \cdot q$ into p, q

- note that for RSA to be secure, both p and q must be large primes
- widely believed to hold true
  - since 1978, subject of extensive cryptanalysis without any serious flaws found
  - best known algorithm takes exponential time in security parameter (key length $|n|$)
- how can you break RSA if you can factor?

Current practice is using 2,048-bit long RSA keys (617 decimal digits)

- estimated computing/memory resources needed to factor an RSA number within one year

Length (bits)	PCs	Memory
430	1	128MB
760	215,000	4GB
1,020	342×10^6	170GB
1,620	1.6×10^{15}	120TB

RSA challenges

Challenges for breaking the RSA cryptosystem of various key lengths (i.e., |n|)

- known in the form RSA-`key bit length’ expressed in bits or decimal digits
- provide empirical evidence/confidence on strength of specific RSA instantiations

Known attacks

- RSA-155 (**512-bit**) factored in **4 mo**. using 35.7 CPU-years or 8000 Mips-years (**1999**) and 292 machines
  - 160 175-400MHz SGI/Sun, 8 250MHz SGI/Origin, 120 300-450MHz Pent. II, 4 500MHz Digital/Compaq
- RSA-**640** factored in **5 mo**. using 30 2.2GHz CPU-years (**2005**)
- RSA-220 (**729-bit**) factored in **5 mo**. using 30 2.2GHz CPU-years (**2005**)
- RSA-232 (**768-bit**) factored in **2 years** using **parallel** computers 2K CPU-years (1-core 2.2GHz AMD Opteron) (**2009**)

Most interesting challenges

- prizes for factoring RSA-**1024**, RSA-**2048** is $100K, $200K – estimated at 800K, 20B Mips-centuries

Deriving an RSA key pair

- public key is pair of integers (e,n), secret key is (d, n) or d
- the value of n should be quite large, a product of two large primes, p and q
- often p, q are nearly 100 digits each, so n ~= 200 decimal digits (~512 bits)
  - but 2048-bit keys are becoming a standard requirement nowadays
- the larger the value of n the harder to factor to infer p and q
  - but also the slower to process messages
- a relatively large integer e is chosen
  - e.g., by choosing e as a prime that is larger than both $(p - 1)$ and $(q - 1)$
  - why?
- d is chosen s.t. $e \cdot d = 1 \bmod (p - 1)(q - 1)$
  - how?

Discussion on RSA

- Assume **p** = 5, **q** = 11, **n** = 5 · 11 = 55, **ϕ(n)** = 40, **e** = 3, **d** = 27
  - why encrypting small messages, e.g., **M** = 2, 3, 4 is tricky?
  - recall that the ciphertext is **C** = $\mathbf{M}^3$ mod 55 for **M** in $\mathbf{Z}_{55}$

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
M	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
C	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
M	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
C	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

Discussion on RSA

- Assume **p** = 5, **q** = 11, **n** = 5 · 11 = 55, **ϕ**(**n**) = 40, **e** = 3, **d** = 27
  - why encrypting small messages, e.g., **M** = 2, 3, 4 is tricky?
  - recall that the ciphertext is $\mathbf{C} = \mathbf{M}^3 \bmod 55$ for **M** in $\mathbf{Z}_{55}$
- Assume n = 2043439438435553434354542894348343435 6091 = p · q
  - can e be the number 4343253453434536?
- Are there problems with applying RSA in practice?
  - what other algorithms are required to be available to the user?
- Are there problem with respect to RSA security?
  - does it satisfy CPA (advanced) security?

Algorithmic issues

The implementation of the RSA cryptosystem requires various algorithms

- Main issues
  - representation of integers of arbitrarily large size; and
  - arithmetic operations on them, namely computing modular powers
- Required algorithms (at setup)
  - generation of **random numbers** of a given number of bits (to compute candidates **p**, **q**)
  - **primality testing** (to check that candidates **p**, **q** are prime)
  - computation of the **GCD** (to verify that **e** and **ϕ(n)** are relatively prime)
  - computation of the **multiplicative inverse** (to compute **d** from **e**)

Pseudo-primality testing

Testing whether a number is prime (**primality testing**) is a difficult problem

An integer **n** ≥ 2 is said to be a base-**x** **pseudo-prime** if

- $x^{n-1} \bmod n = 1$ (Fermat's little theorem)
- Composite base-**x** pseudo-primes are rare
  - a random 100-bit integer is a composite base-2 pseudo-prime with probability less than 10^{-13}
  - the smallest composite base-2 pseudo-prime is 341
- Base-**x** pseudo-primality testing for an integer **n**
  - check whether $x^{n-1} \bmod n = 1$
  - can be performed efficiently with the repeated squaring algorithm

Security properties

- Plain RSA is deterministic
  - why is this a problem?
- Plain RSA is also homomorphic
  - what does this mean?
  - multiply ciphertexts to get ciphertext of multiplication!
  - $[(m_1)^e \bmod N][(m_2)^e \bmod N] = (m_1 m_2)^e \bmod N$
  - however, not additively homomorphic

Real-world usage of RSA

- Randomized RSA
  - to encrypt message M under an RSA public key (e,n), generate a new random session AES key K, compute the ciphertext as [K^e mod n, $AES_K(M)$]
  - prevents an adversary distinguishing two encryptions of the same M since K is chosen at random every time encryption takes place
- Optimal Asymmetric Encryption Padding (OAEP)
  - roughly, to encrypt M, choose random r, encode M as $M' = [X = M \oplus H_1(r) , Y= r \oplus H_2(X)]$ where H_1 and H_2 are cryptographic hash functions, then encrypt it as $(M')^e$ mod n

7.2 On message authentication

Recall: Approach in modern cryptography

Formal treatment

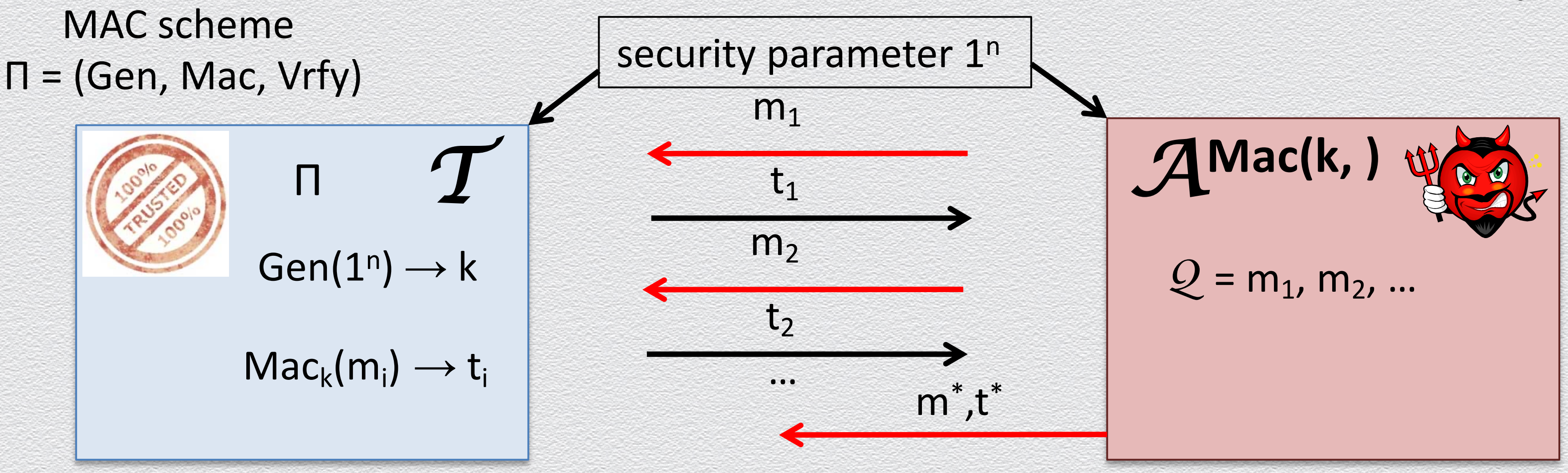
- **fundamental notions** underlying the **design & evaluation** of crypto primitives

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. $\mathrm{Vrfy}_k(m^*,t^*) = 1$ &
2. m^* not in $\mathcal{Q}$

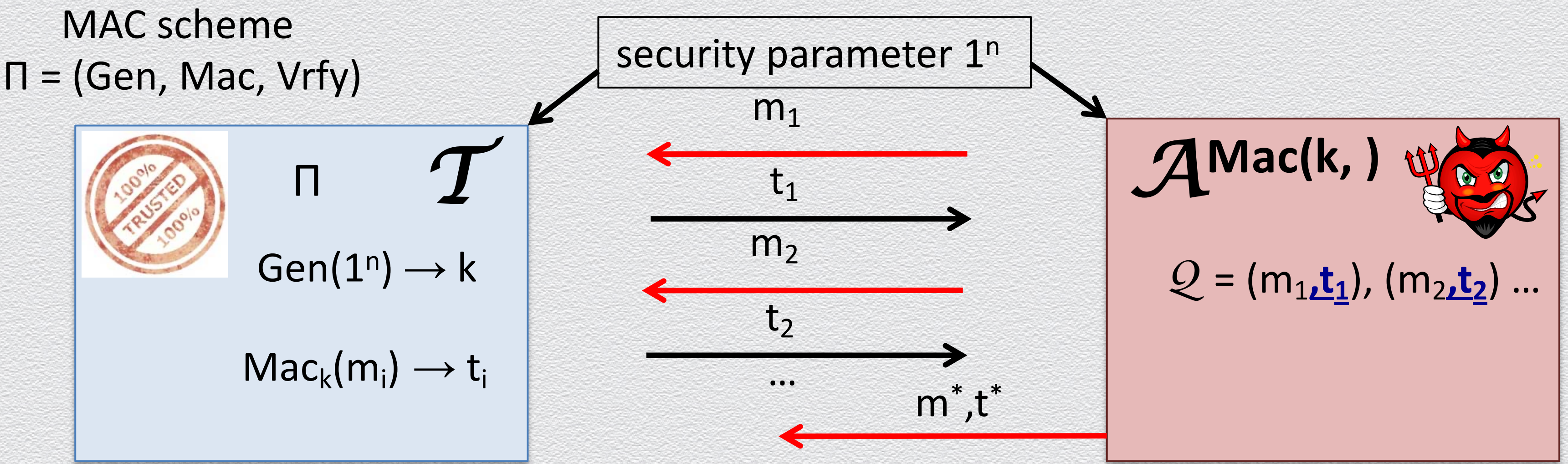


We say that Π is **secure** if for all PPT $\mathcal{A}$, there exists a negligible function negl so that

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

Strong MAC

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



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

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

(Strong) MAC w/ verification queries

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

MAC scheme
Π = (Gen, Mac, Vrfy)

security parameter 1^n

Π $\mathcal{T}$

$\text{Gen}(1^n) \rightarrow k$

$\text{Mac}_k(m_i) \rightarrow t_i$

m_1 or (m_1, t_1)

t_1 or acc/rej

m_2 or (m_2, t_2)

t_2 or acc/rej

…

m^*,t^*

$\mathcal{A}^{\text{Mac}(k,\ ),\text{Vrfy}(k,)}$

$\mathcal{Q} = (m_1,t_1), (m_2,t_2)$ …

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

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

Verification queries Vs. timing attacks on MAC verification

In game Mac-s<u>V</u>forge$_{\mathcal{A},\Pi}$(n)

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

In real life

- implicit tag verification is **<u>feasible</u>** (e.g., by detecting a difference in verifier's behavior)
- but also an attacker may receive **<u>more than this 1-bit info</u>** 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 **<u>byte by byte until first mismatch</u>**
  - thus, the time to return `rej` **<u>depends on position</u>** 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)

7.3 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

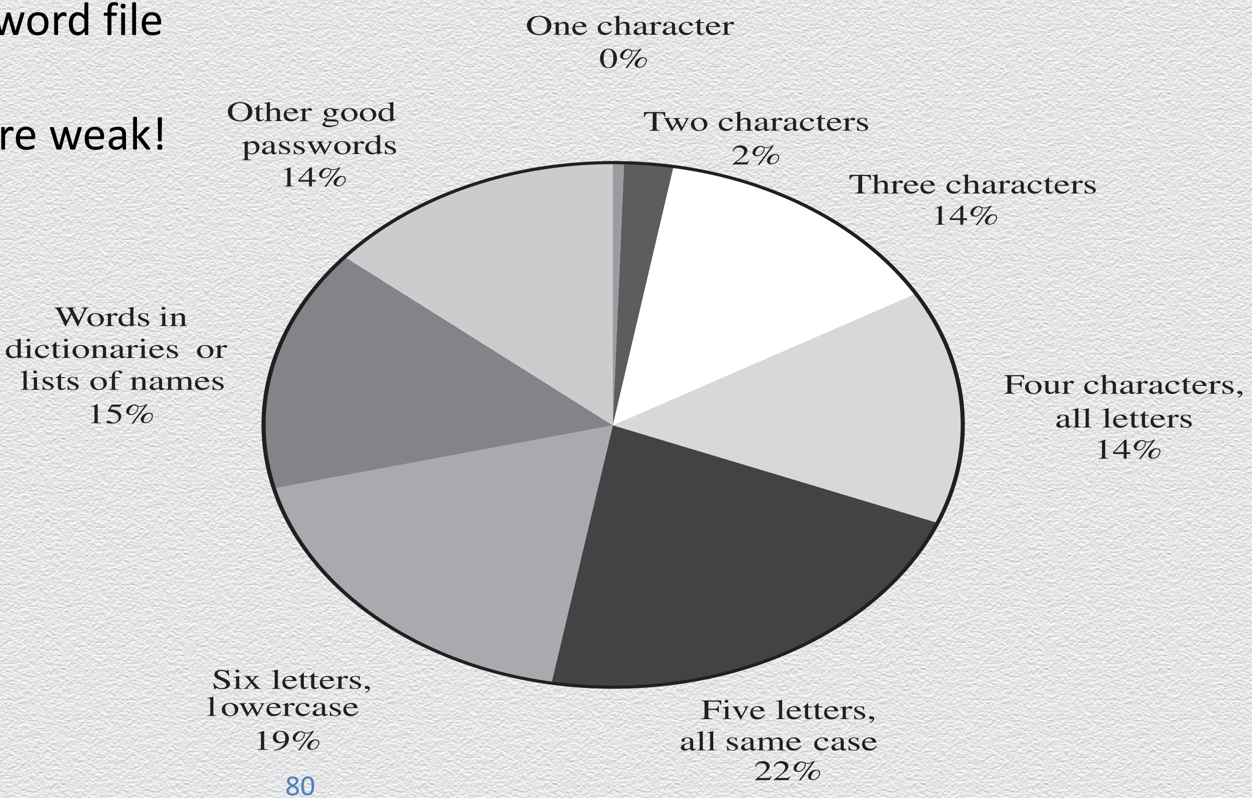
Infer passwords through guessing

- 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 2^{nd} 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

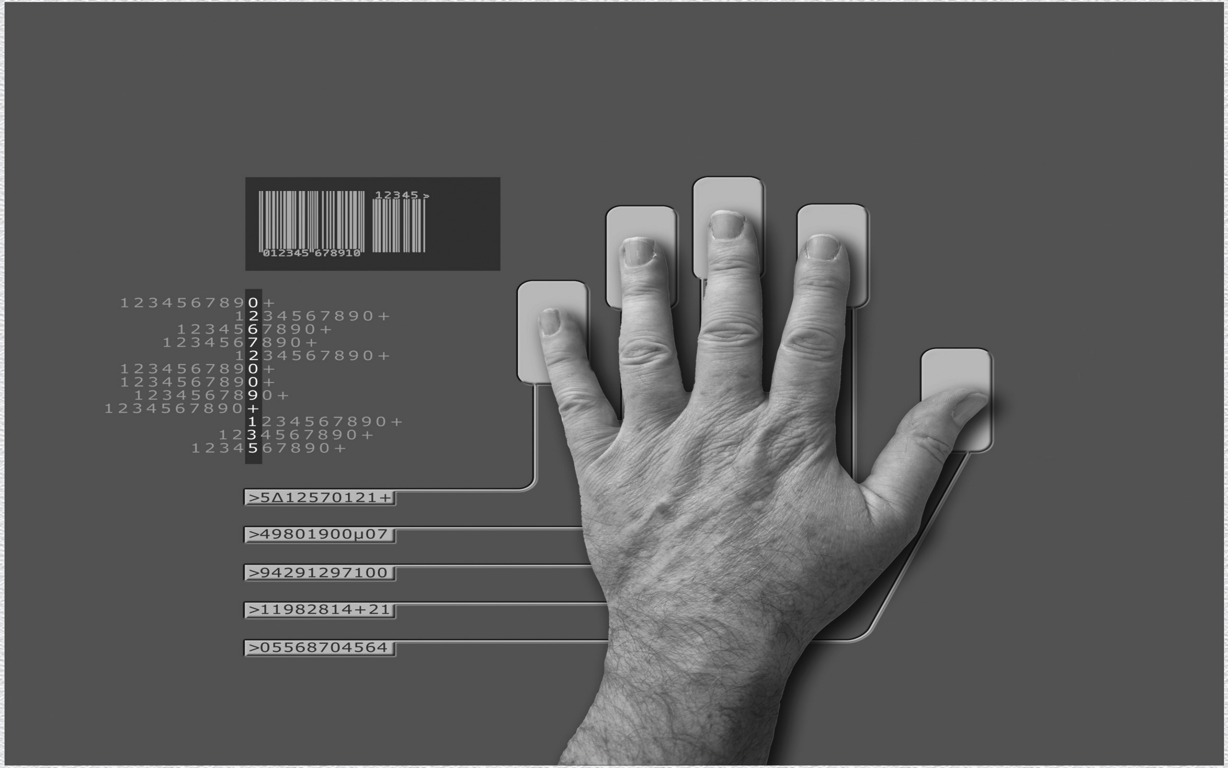
- to slow down dictionary attacks
  - a password is hashed k times before being stored

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!

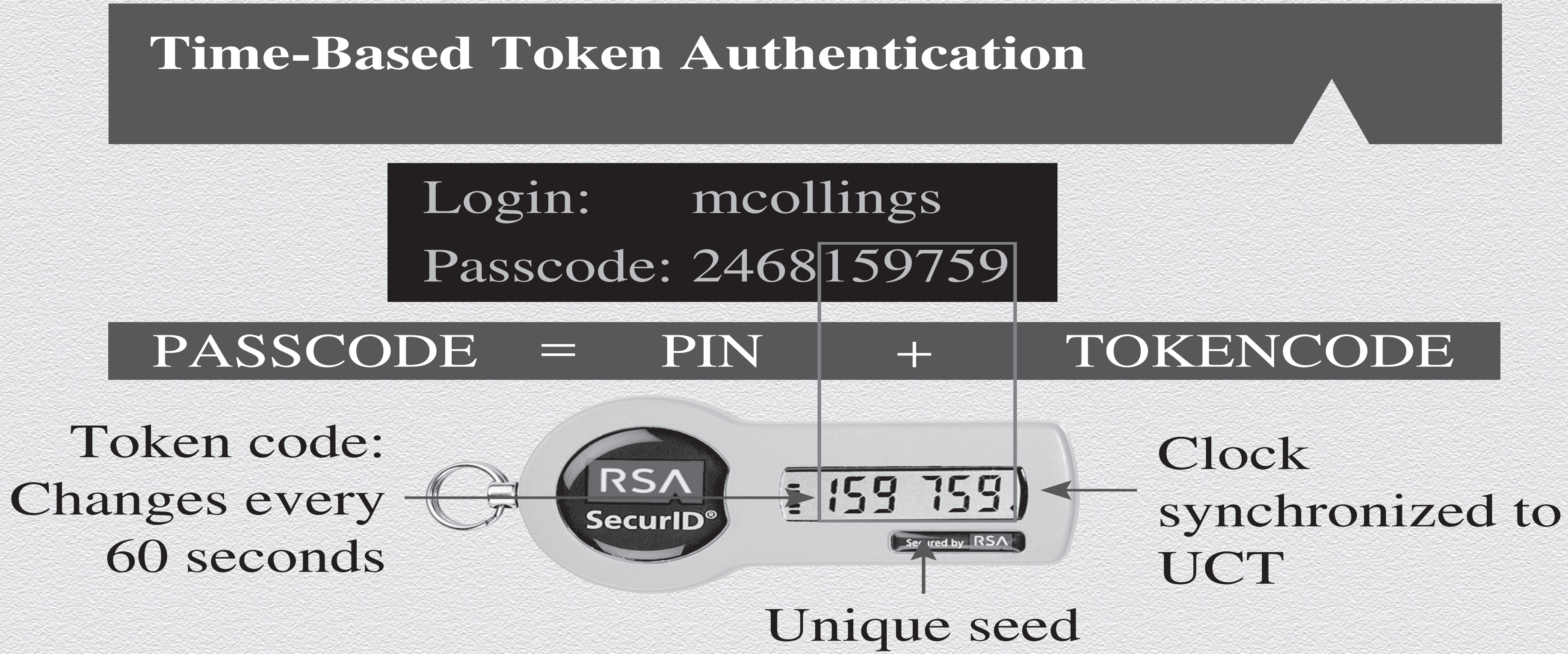
- you are leaving your fingerprints in many places
- rubber fingers have defeated commercial fingerprint-recognition
- minor issue if authentication takes place in the presence of security personnel
  - when authenticating remote users additional precautions have to be taken
- user acceptance: so far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

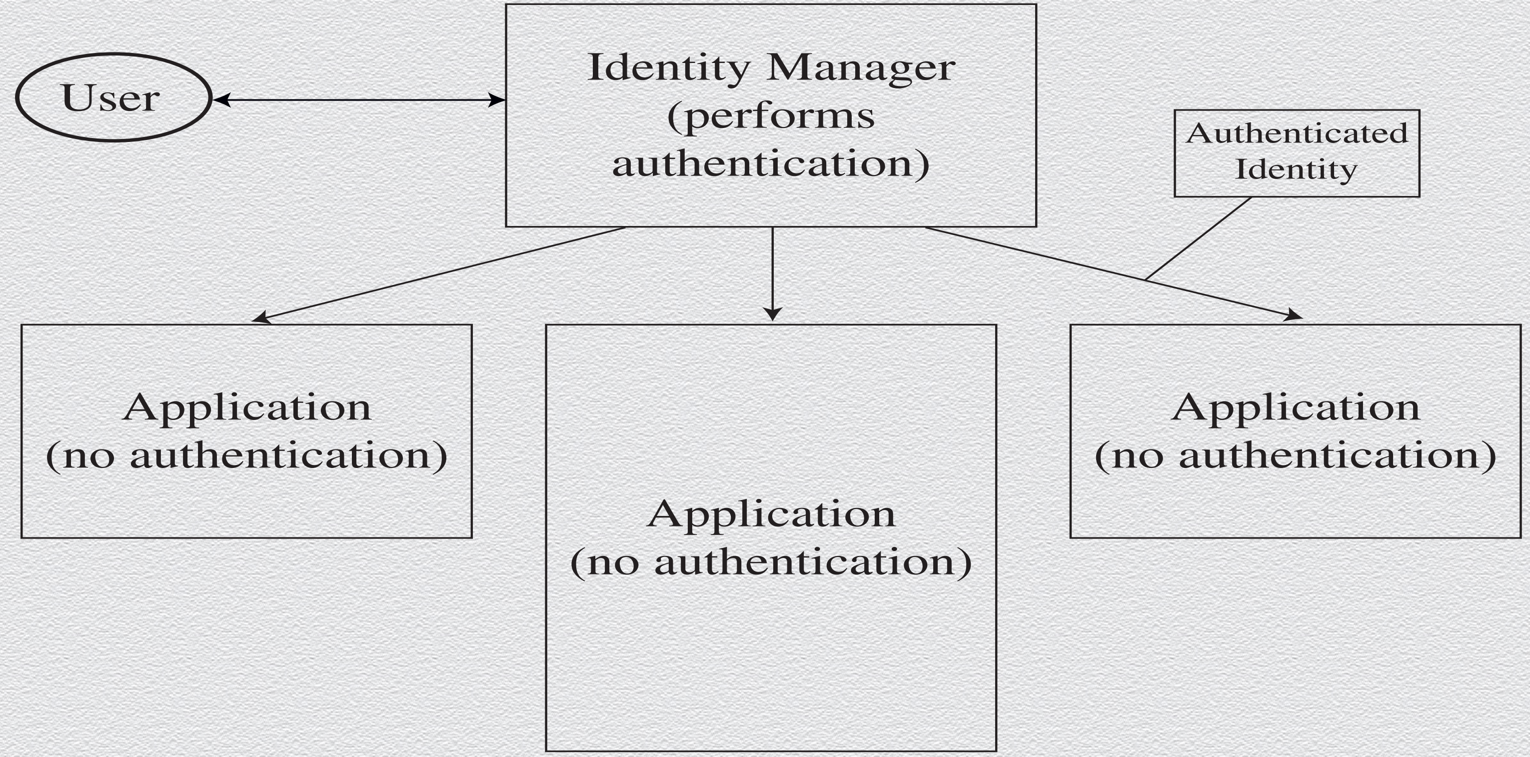


Problems with tokens

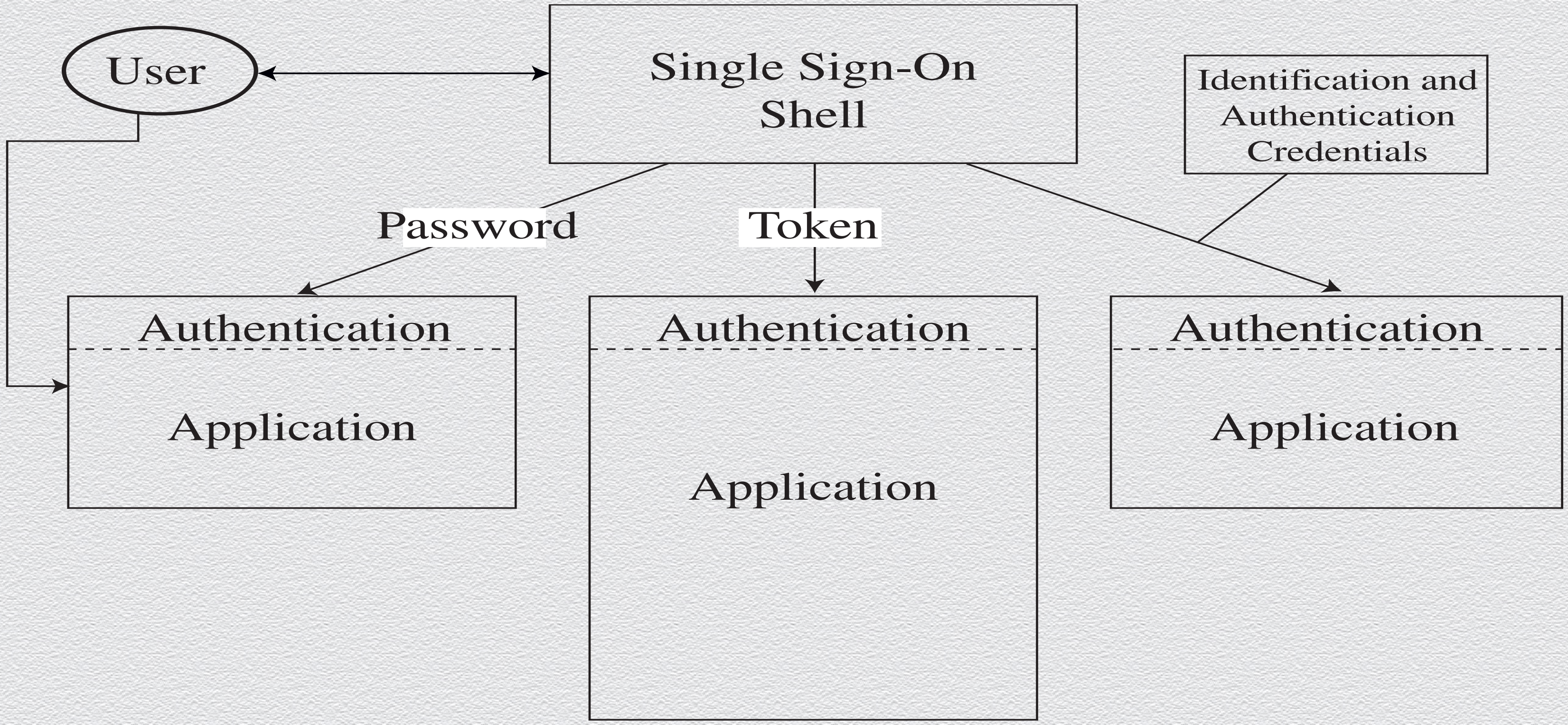
- Inconvenience
- Lost token
- Stolen token
- Cloned token
- Side-channel attacks (for key exfiltration)

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 More on password 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 $a_1, a_2, \ldots, a_{64}$
  - Split hash x into 48-bit blocks $x_1, x_2, \ldots, x_5$ and one 16-bit block x_6
  - Compute $y = x_1 \oplus x_2 \ldots \oplus x_5$
  - Split y into 6-bit blocks $y_1, y_2, \ldots, y_8$
  - Let $p = a_{y_1}, a_{y_2}, \ldots, a_{y_8}$
- This method can be generalized to arbitrary password spaces

Password chain

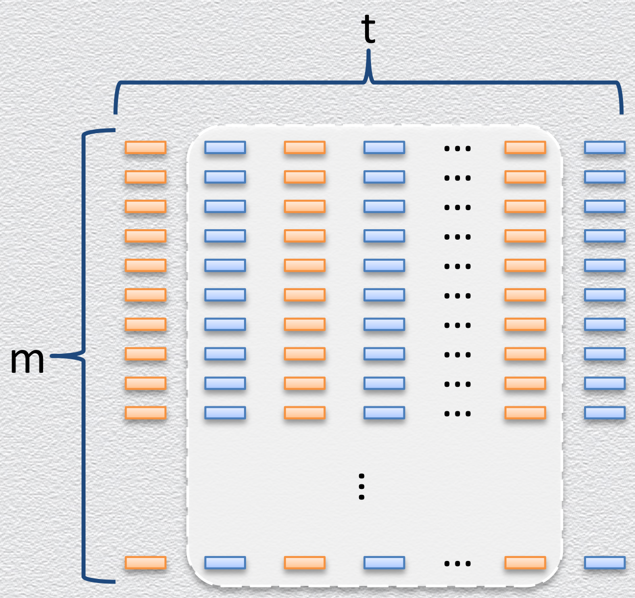
- Sequence (of size t) alternating **passwords** & **hashes**
  - Start with a random password p_1
  - Alternate using cryptographic hash function H & reduction function R
    - $x_i = H(p_i)$, $p_{i+1} = R(x_i)$
  - End with a hash value x_t

mycat6 $\xrightarrow{H}$ b56f19c8d4 $\xrightarrow{R}$ twotrains $\xrightarrow{H}$ ... $\xrightarrow{H}$ 43be7901cd

p_1 x_1 p_2 x_t

Hellman's method

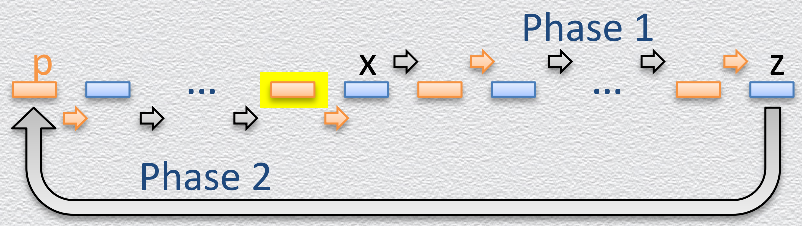
- Starting from m random passwords, build a table of m password chains, each of length t
- The expected number of distinct passwords in a table is $\Omega(mt)$
- Compressed storage:
  - For each chain, keep only the first password, p, and the last hash value, z
  - Store pairs (z, p) in a dictionary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



Classic password recovery

Recovery of password with hash value x

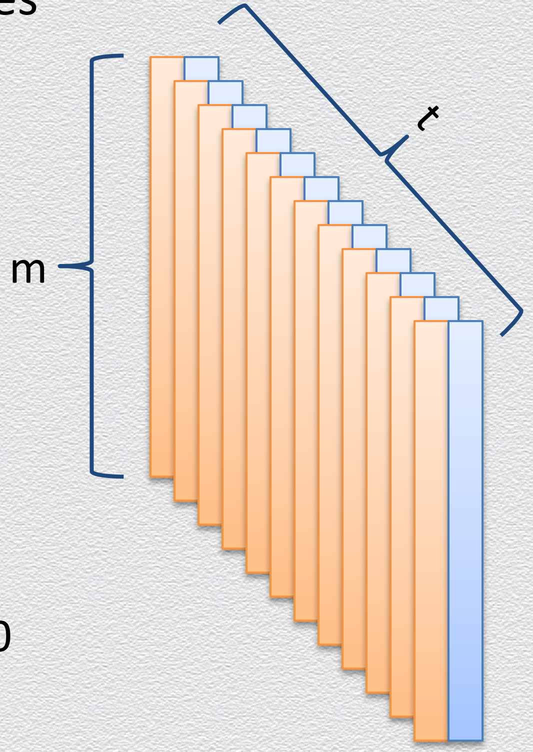
- Step 1: traverse the suffix of the chain starting at x
  - y = x;
  - while p = D.get(y) is null
    - y = H(R(y)) //advance
    - if i++ > t return "failure" //x not in the table
- Step 2: traverse the prefix of the chain ending at x
  - while y = H(p) ≠ x
    - p = R(y) //advance
    - if j++ > t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



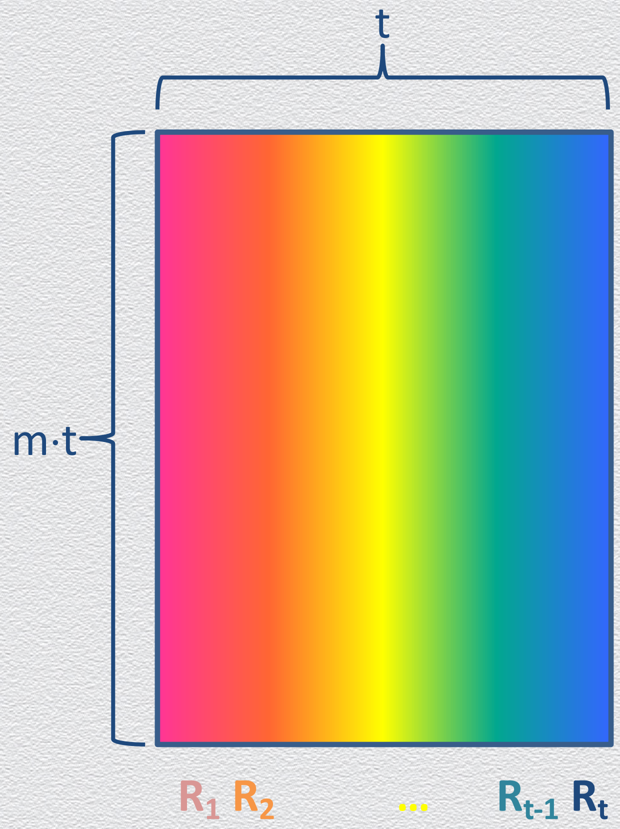
Performance

- 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**



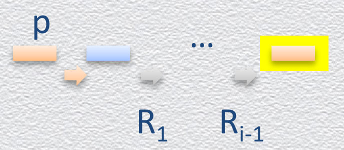
Performance

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

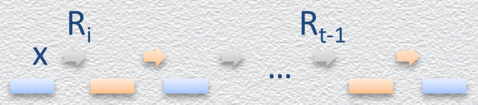
Rainbow-table password recovery

```
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(R_j(y)) //advance
    if p = D.get(y) is not null //candidate position i
        for j = 1 … i - 1 //traverse from 1 to i
            p = R_j(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

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