https://brown-csci1660.github.io

CS1660: Intro to Computer Systems Security
Spring 2026

Lecture 7: Addendum

Instructor: **Nikos Triandopoulos**

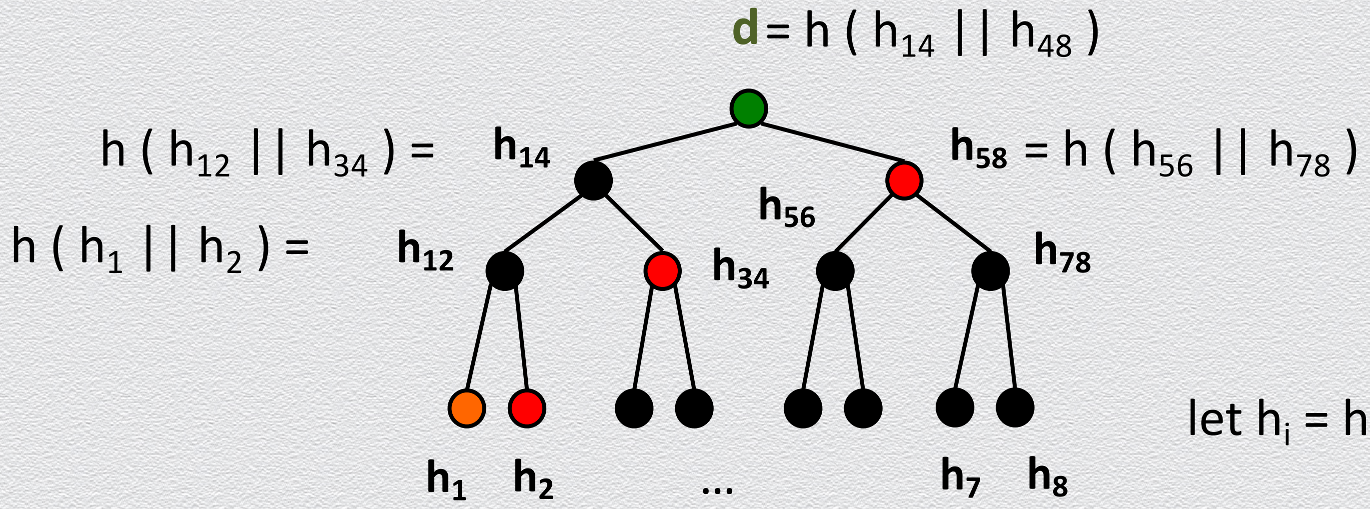
February 12, 2026

BROWN

7.3.2+ The Merkle tree

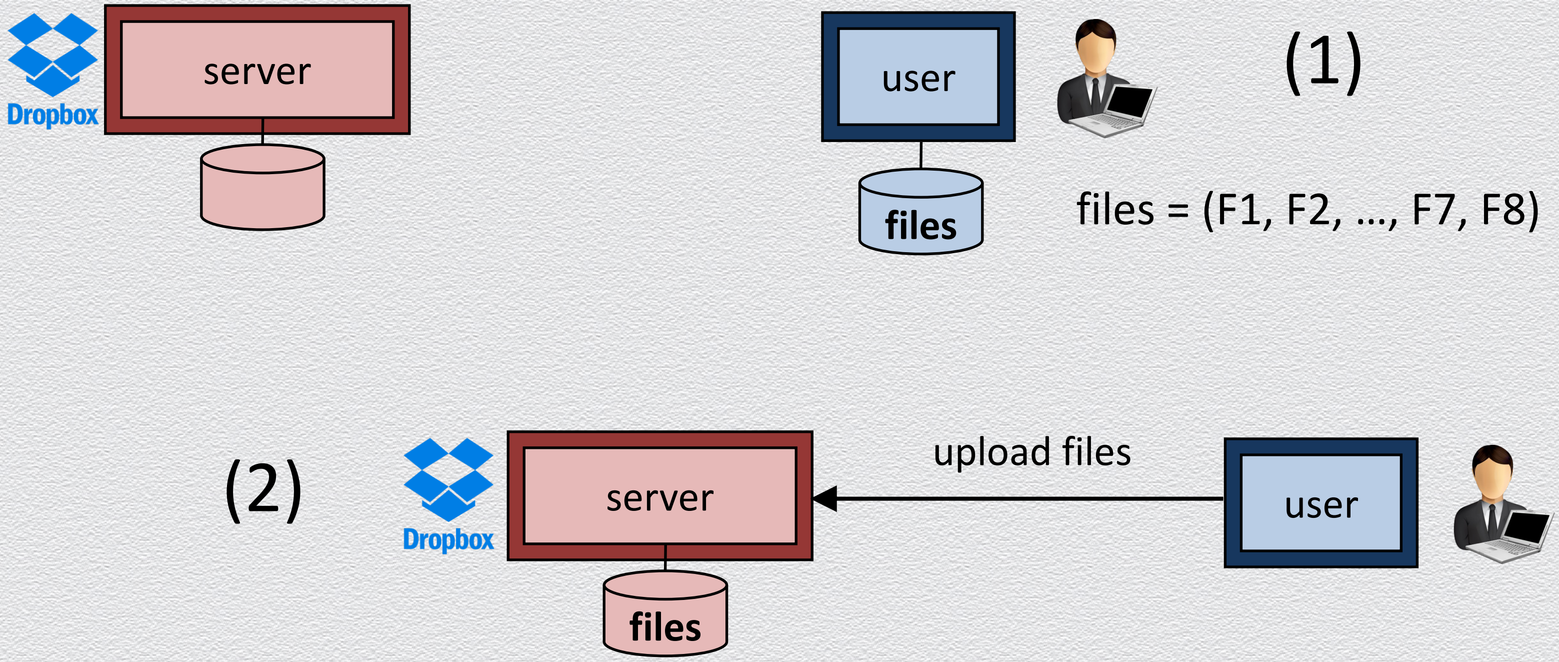
Application 6: The Merkle tree

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

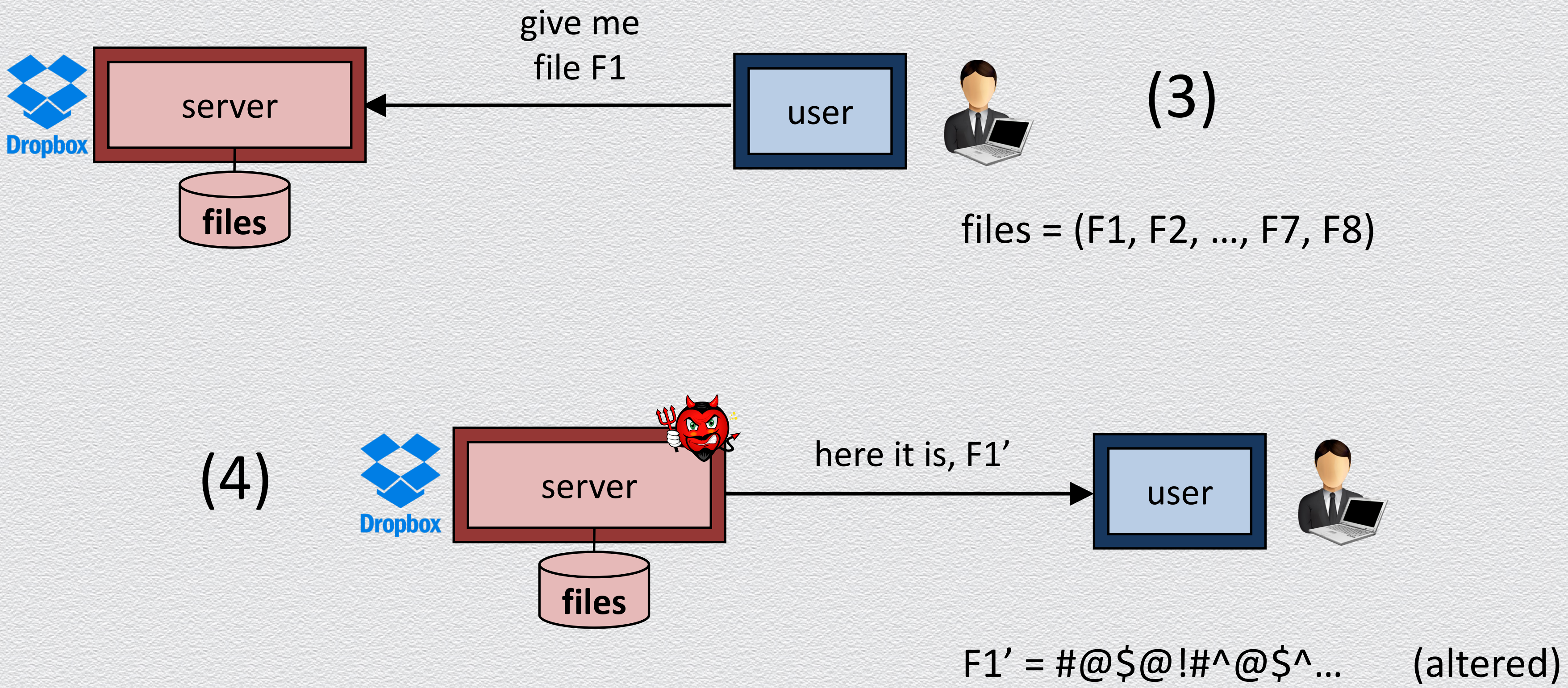


let $h_i = h(F_i)$, $1 \leq i \leq n$

Example 6: Secure cloud storage



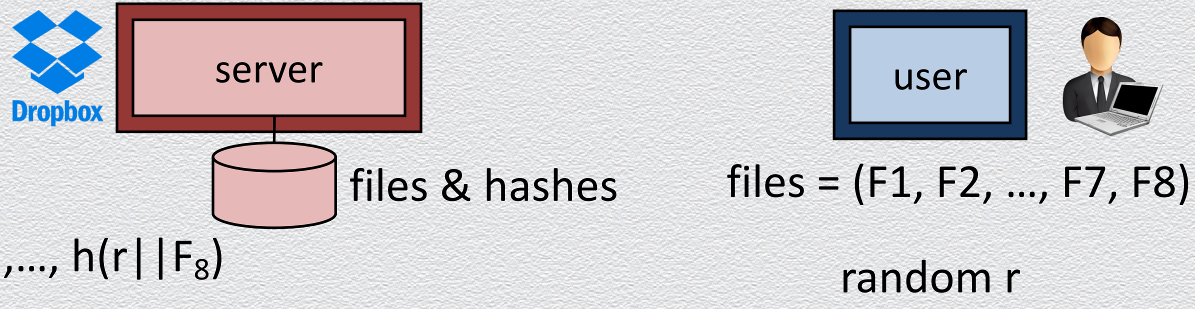
Example 6: Secure cloud storage



Example 6: Secure cloud storage – per-file hashing

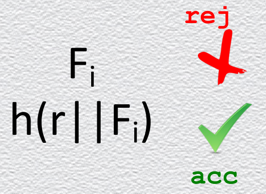
Bob wants to outsource storage of files $F_1, F_2, \ldots, F_8$ to Dropbox & check their integrity

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



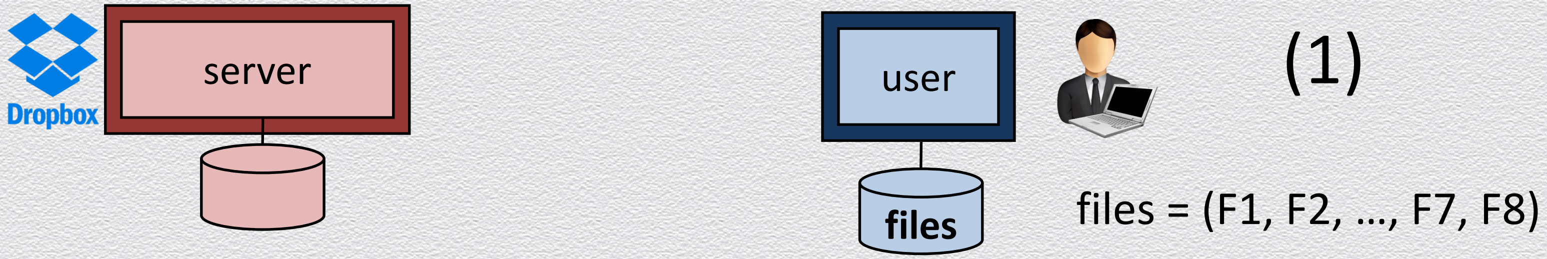
Every time Bob **reads** a file F_i, he also reads $h(r||F_i)$ to verify F_i's integrity

- any problems with **writes**?



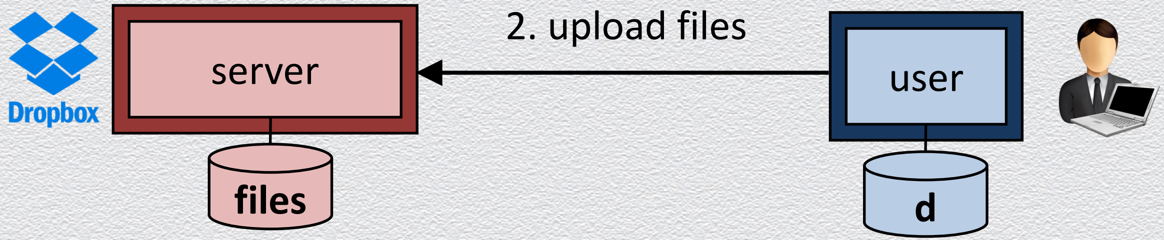
Example 6: Secure cloud storage – per-file-set hashing

(1)

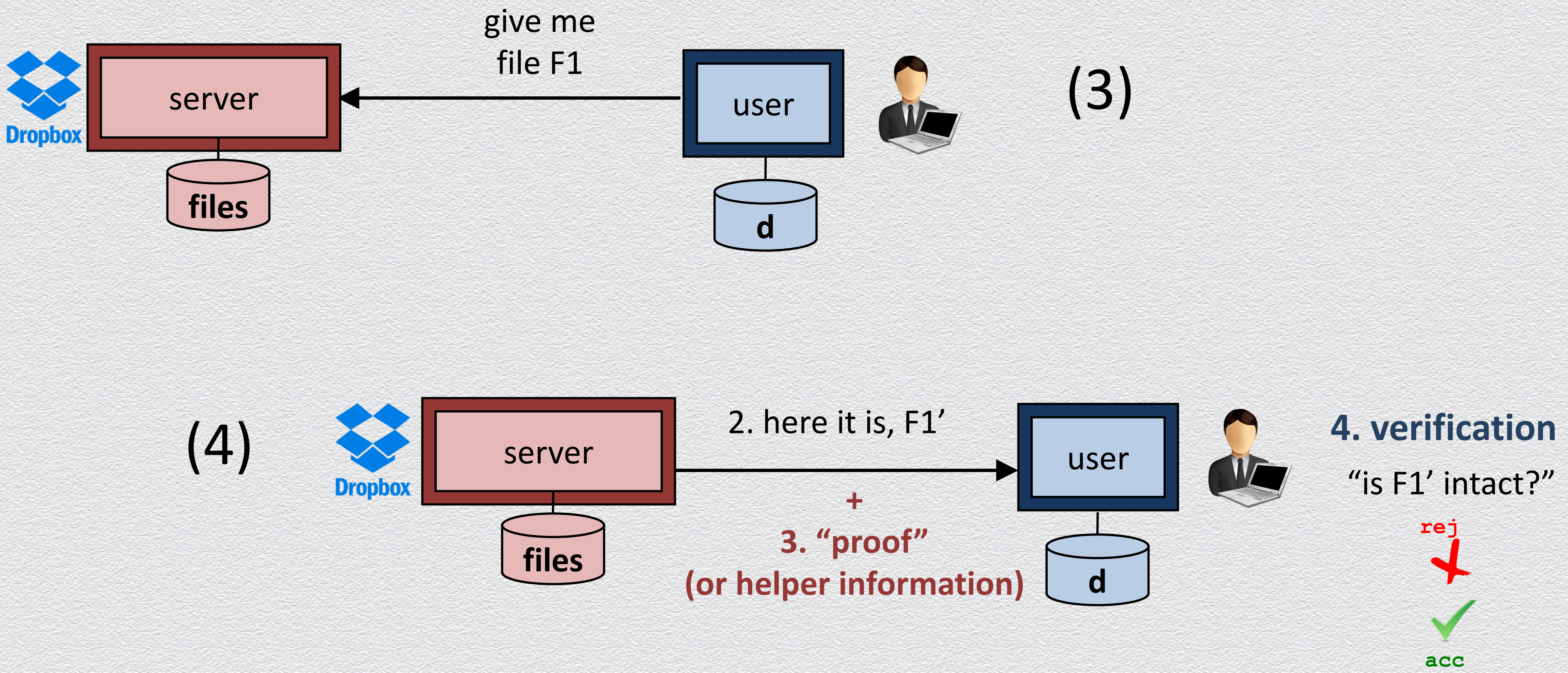


1. use CR hash function h to compute over all files a digest d, |d| << |F|

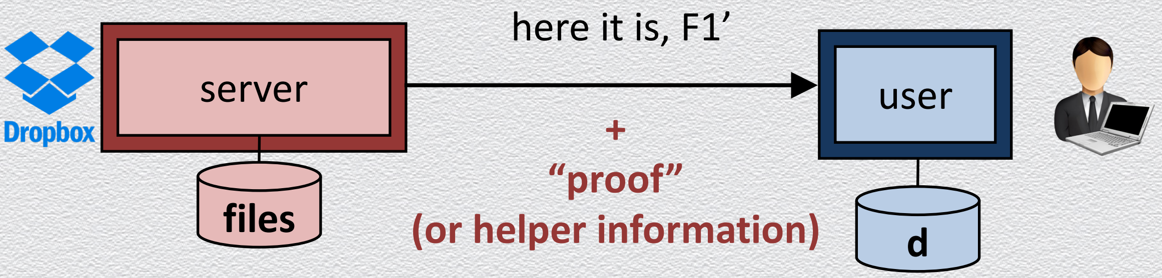
(2)



Example 6: Secure cloud storage – integrity checking



Example 6: Secure cloud storage – verification



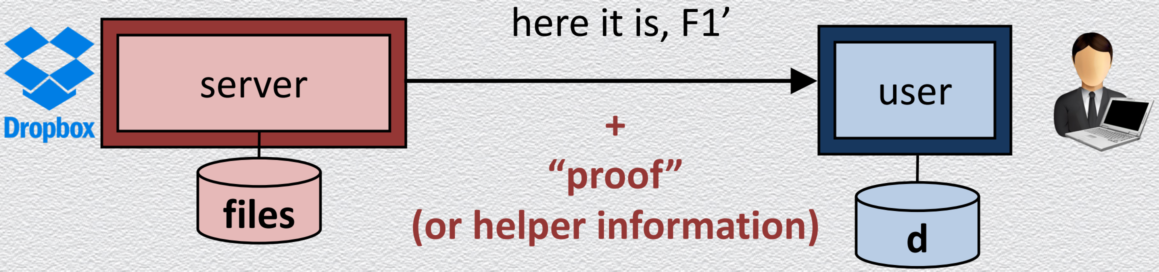
(4)

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



(4)

verification

“is F1’ intact?”

d’ ≠ d rej

d’ = d acc

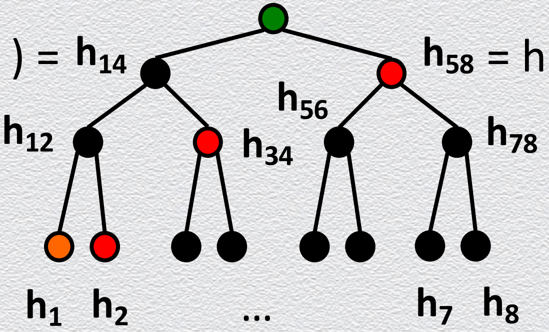
digest is the **green** root hash

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

7.5+ 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 tracking 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"

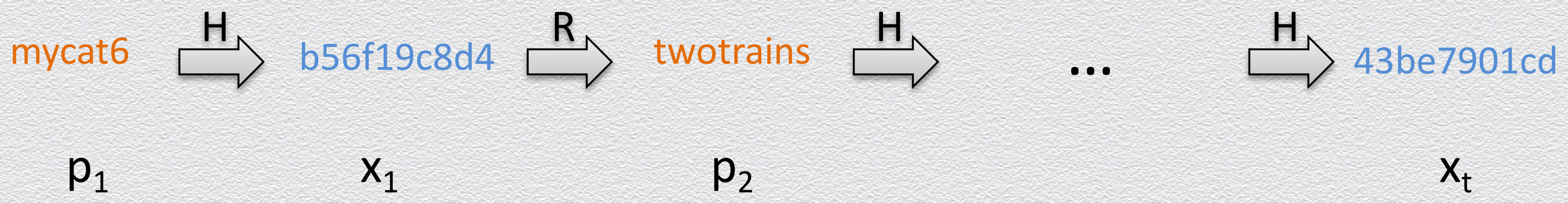
Reduction function

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

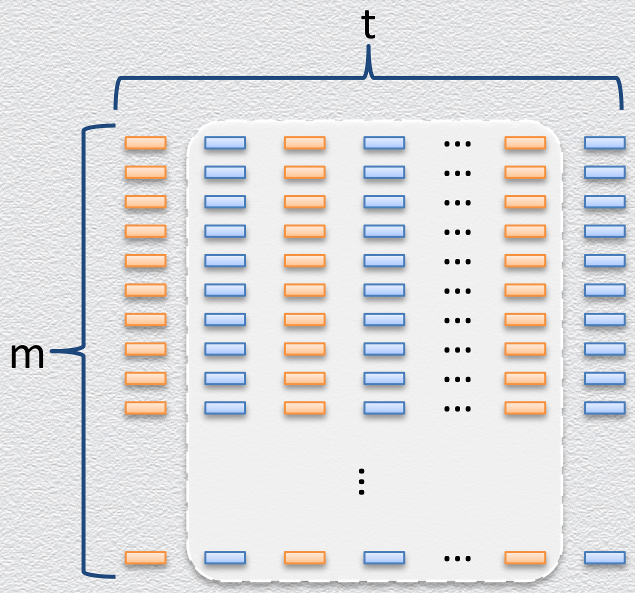
Password chain

- 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



Hellman's method

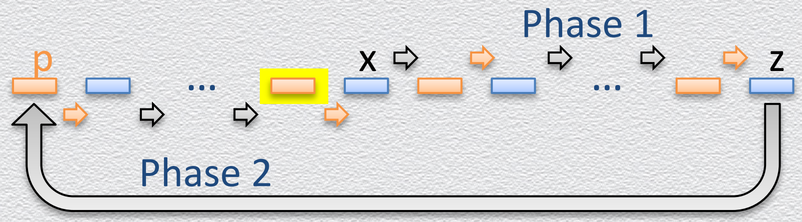
- Starting from m random passwords, build a table of m password chains, each of length t
- The expected number of distinct passwords in a table is $\Omega(mt)$
- Compressed storage:
  - For each chain, keep only the first password, p, and the last hash value, z
  - Store pairs (z, p) in a dictionary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



Classic password recovery

Recovery of password with hash value x

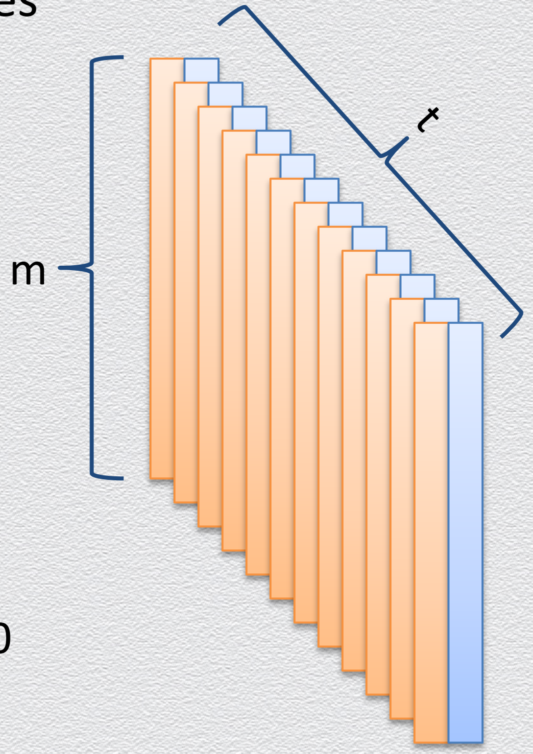
- Step 1: traverse the suffix of the chain starting at x
  - y = x;
  - while p = D.get(y) is null
    - y = H(R(y)) //advance
    - if i++ > t return "failure" //x not in the table
- Step 2: traverse the prefix of the chain ending at x
  - while y = H(p) ≠ x
    - p = R(y) //advance
    - if j++ > t 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., $n = 1{,}000{,}000{,}000$, $m = t = n^{1/3}$, $mt = t^2 = n^{2/3} = 1{,}000{,}000$

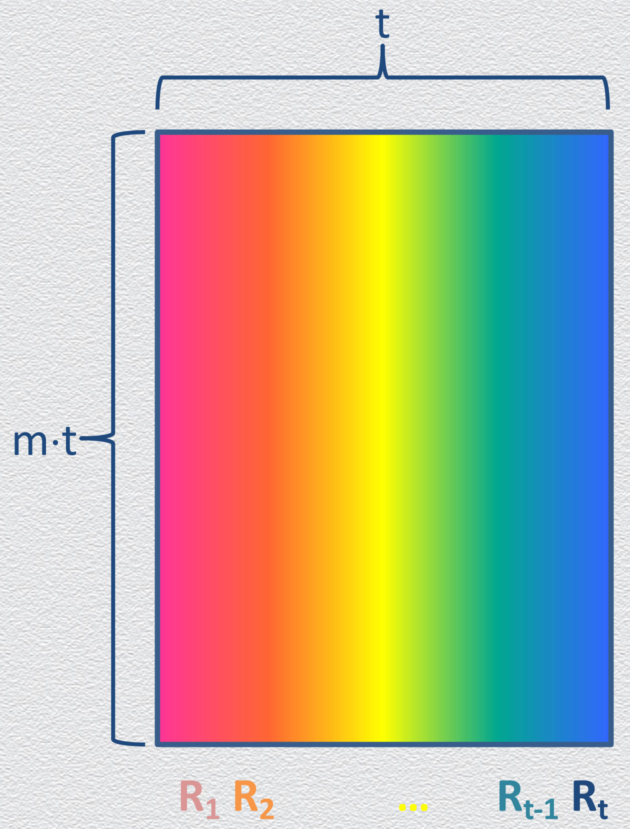
Rainbow table

Instead of t different tables, use a single table with

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

Performance

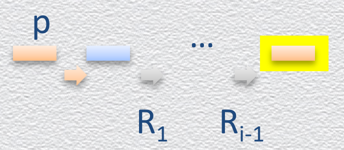
- Storage : mt hash values (as before)
- Recovery : $t^2/2$ hash computations & t dictionary lookups (lower than before)



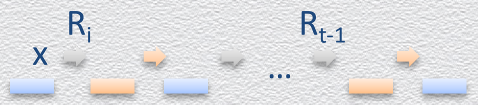
Rainbow-table password recovery

```
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

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