

A decorative graphic on the left side of the slide. It consists of a blue parallelogram and a light green parallelogram, both tilted at an angle. The blue shape is in the foreground, and the green shape is partially behind it. The background of the entire slide is a dark navy blue with subtle, lighter blue diagonal stripes.

Rainbow Tables

What is a rainbow table?



?



????



A brief overview of passwords

Server has information, client wants information

Client sends password - this is the *prover* (because it's proving it's real)

Server checks password - this is the *verifier* (because it's verifying)

Basic Password Protocol - it's a basic protocol for passwords

- Secret Key : sk = Password: pw (stored/memorized somehow by the prover)
- Verifier key : vk = sk (just compare them.)
- Verifier authenticates the password

No encryption, people are nice.



Password security

Plaintext password storage is dumb, stupid, and insecure.

- Vulnerable to all sorts of stuff; if the logon server is compromised at all, the attacker can log into everyone's accounts
- Makes the magnitude of a data breach *much* greater

We should probably encrypt it somehow.

Solution: Don't be dumb

- Store the password as something that isn't a plaintext.

Password security

Password Protocol (version 1):

Basic idea: Instead of storing the password as a plaintext, store the hash of the password.

- If the hash function is secure, then theoretically even a leak of the password database shouldn't immediately compromise accounts.

Prover's secret key = $sk = pw$ = The password

Verifier key = $vk = H(sk)$ = Not the password

Formal version

Password ID protocol $I_{\text{PWD}} = (G, P, V)$:

$G = \text{set of } pw \xleftarrow{R} P$, output $sk := pw$ and $vk = H(pw)$

Algorithm P on input $sk = pw$, and algorithm V on input $vk = H(pw)$ interact as such:

- P sends pw to V
- V outputs `accept` if it receives pw such that $H(pw) = vk$, or outputs `reject` otherwise.

Attacks → *Direct attacks*

Formal-ish version

Protocol $I = (G, P, V)$, as defined in the last slide, and adversary A

Key generation: Challenger runs $(vk, sk) \leftarrow G()$, and sends vk to A .

Impersonation: Challenger acts as the verifier using protocol I while A acts as the prover.

Adversary A wins if V outputs `accept` at any point.

A 's advantage relative to I is the probability that A wins the game.

If this advantage is negligible for all efficient adversaries A , I can be said to be secure against direct attacks.

In other words, security against direct attacks is how likely an adversary is to obtain the secret key (password) if they already have the verifier key (hashed password.)



Attacks → *Dictionary attacks*

Fact: Humans are fallible.

→ password, 123456, 123456789, guest, qwerty, 12345678, 111111, 12345, 123456, and 123123 are consistently the most-used passwords

If 5% of people use one of these passwords, getting into *someone's* account isn't hard.

- From now on, call such a commonly-used password a *weak password*, as opposed to a *strong password*.

Dictionary attack: if you think someone's password is weak, just guess from a list of common passwords until you get in. Much more efficient than brute force.

Fun thought experiment

Think of how dumb the average human is, then realize that half of humanity is statistically dumber.



Attacks → *Dictionary attacks*

Online dictionary attack: an adversary suspects that someone's password is weak. An online dictionary attack is just trying to log in by trying a list of commonly used passwords one at a time until one of them works.

Another approach is to try many *different* accounts with the same commonly used password. An attacker can get around IP filters with botnets and the like.

Attacks like these can be mitigated somewhat by putting in safeguards to the login site.



Attacks → *Dictionary attacks*

Offline dictionary attack: Similar to an online dictionary attack, except that the login server has been compromised to the point where the attacker has access to vk , and by extension, to $H(pw)$.

This allows the attacker to make guesses at the password *without sending them to the login server*, and upon finding the correct password, get in first try.

Since login attempts are not being made to the login server, these cannot be mitigated with the same techniques

About 50% of passwords can be cracked after about 1.5 billion offline hashes. With GPU acceleration, this is trivial; the AMD Radeon RX 6700 is a video card that costs less than \$300 and can perform around 45 million hashes every second, or just over 30 seconds for 1.5 billion hashes.



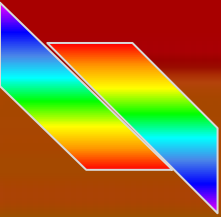
Attacks → *Dictionary attacks*

Preprocessing: Accelerates obtaining passwords for multiple accounts.

Instead of hashing a guess and comparing it, *hash all of the guesses you're going to make beforehand and store them in a list*, then compare the processed ciphertexts. You can now check multiple accounts without re-doing the hashes every time. It thus becomes practical to guess far more passwords if you are trying to crack many accounts

The drawback is the memory requirement; you need to store the hashes somewhere. If you want to store hashes for all 128^8 8-character ASCII passwords, that's around 4 terabytes of data.

The tradeoff is space for time. Attacking after preprocessing takes no hashes (or $O(1)$), but requires $O(n)$ storage. Without preprocessing, no additional storage is needed, but it requires $O(n)$ hashes for each attack.



Rainbow tables

Rainbow tables are a modification to the preprocessing method.

Middle ground of the space-time tradeoff.

- No preprocessing = $O(1)$ preprocessing, $O(n)$ attack time, $O(1)$ space
- Preprocessing = $O(n)$ preprocessing, $O(1)$ attack time, $O(n)$ space
- Rainbow tables \approx $O(n)$ preprocessing, $O(n^{2/3})$ attack time, $O(n^{2/3})$ space.

Ex: 1 terabyte of memory for preprocessing table (~\$4000 for 8x 128GB DDR4-2400 modules, requires an expensive server/HEDT platform supporting ECC LRDIMMs) becomes 1 gigabyte of memory (4GB DDR4-3200 DIMMs are <\$10 on ebay, and your computer already has this.)

On moderately-sized data sets, this can be the difference between storing hashes on memory and storage.

On large data sets, this can be the difference between allowing you to store the data *at all*.

Ex: 1PB of HDD = 64x 16TB HDDs @ \$250 each = \$16000, 2TB HDD = \$40.



Rainbow tables

The goal: we have a hash function $H: p \rightarrow v$. We want to reverse it.

- p = the set of passwords
- v = possible ciphertexts (for example, $\{0,1\}^{256}$ for 256-bit keys)

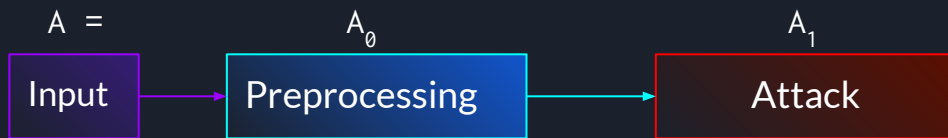
The problem: H is a hash function and doesn't like being reversed

The idea: You can't reverse a hash function and keep the simplicity, but *maybe you can allow yourself to skip a few steps*.

Call the reverse algorithm $A = (A_0, A_1)$, which will once again consist of a preprocessing and attack phase.

$pw \xleftarrow{R} p, y \leftarrow H(pw)$. Let $N := |p|$

A search of p will yield the preimage of y after at most N calls to H .





Rainbow tables \rightarrow *Preprocessing phase*

```
pw  $\xleftarrow{R}$  p,  
y  $\leftarrow H(\text{pw})$   
N  $:= |p|$ 
```

Input

$h: p \rightarrow v$
(random function)

p = passwords

v = possible
ciphertexts

N $:= |p|$

Preprocessing

Algorithm A_0 :

Interrogates h with a list of guessed passwords, output a table L containing $|p|$ pairs in P^2

This is our rainbow table; store it in memory/storage for use in the attack phase

Attack

A_1



Rainbow tables → *Attack phase*

```
pw  $\xleftarrow{R}$  p,  
y  $\leftarrow H(pw)$   
N := |p|
```

Input

$h: p \rightarrow v$
(random function)

p = passwords

v = possible
ciphertexts

$N := |p|$

Preprocessing

Table L

Attack

Algorithm A_1 :

A_1 is called as $A_1(L, y)$, using L to find an inverse of y .

Outputs a preimage, pw' , in $h^{-1}(y)$ with a nearly 100% success rate.

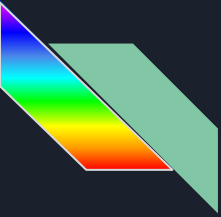


Runtime relations

The approximate amount of time an algorithm takes to invert h can be expressed with this algorithm:

$$t(A_1) \times l^2 \approx N^2$$

In english: the more data you store in table L , the faster the attack goes. If you store a table of size $l=n^{2/3}$, you will need *about* $n^{2/3}$ time to reverse h .



Rainbow table algorithm → *Hellman's basic time-space tradeoff*

A relatively early approach made to crack DES. It employs a reduction function $g: y \rightarrow p$ to “reduce” an element of y to an element in p . If g is also the random function, then $f(pw) := g(h(pw))$ maps p to itself; this function f is used in the preprocessing (A_0) phase.

The A_0 phase takes two parameters, τ and l .



Hellman's Basic Time-Space Tradeoff → *Pseudocode*

Input

$h: p \rightarrow v$
(random function)

p = passwords

v = possible
ciphertexts

$N := |p|$

Parameters τ, l

Preprocessing

Algorithm A_0 :

for $i = 1, \dots, l$:

$pw_i \xleftarrow{R} p$

$z_i \leftarrow f^{(\tau)}(pw_i) \in p$

Return $L := \{(pw_1, z_1), \dots, (pw_l, z_l)\} \subseteq p^2$

Generate chains of length τ .

Attack

Algorithm $A_1(L, y)$:

$z \leftarrow g(y) \in p$

for $i = 1, \dots, \tau$:

if there exists pw such that $(pw, z) \in L$:

$pw \leftarrow f^{(\tau-i)}(pw)$

If $h(pw) = y$:

return pw

$z \leftarrow f(z) \in p$

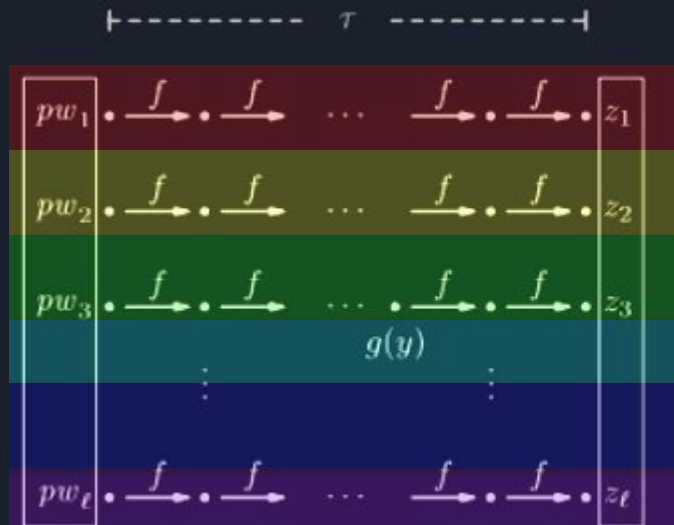
output "fail"

If z is a chain endpoint, traverse the chain from the beginning; if the inverse is found, return it, otherwise move down the chain.

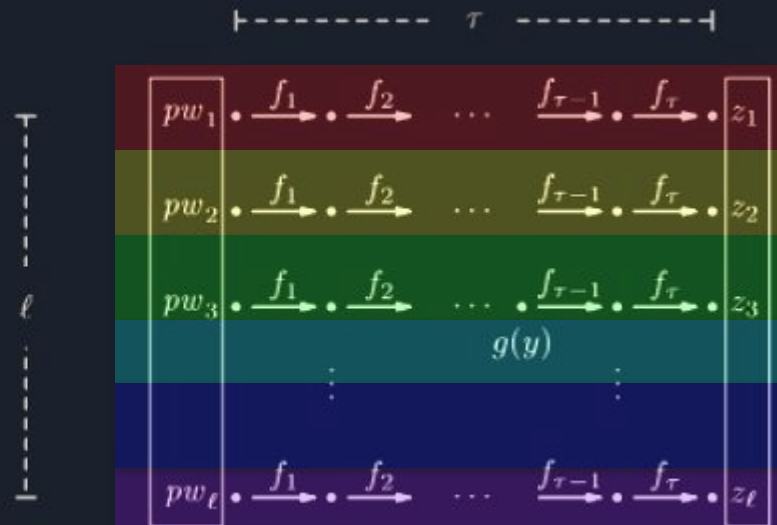
If $g(y)$ is not on any chain, it's a failure.

Hellman's Time-Space Tradeoff \rightarrow

Graphical representation



(a) Hellman's basic time-space tradeoff

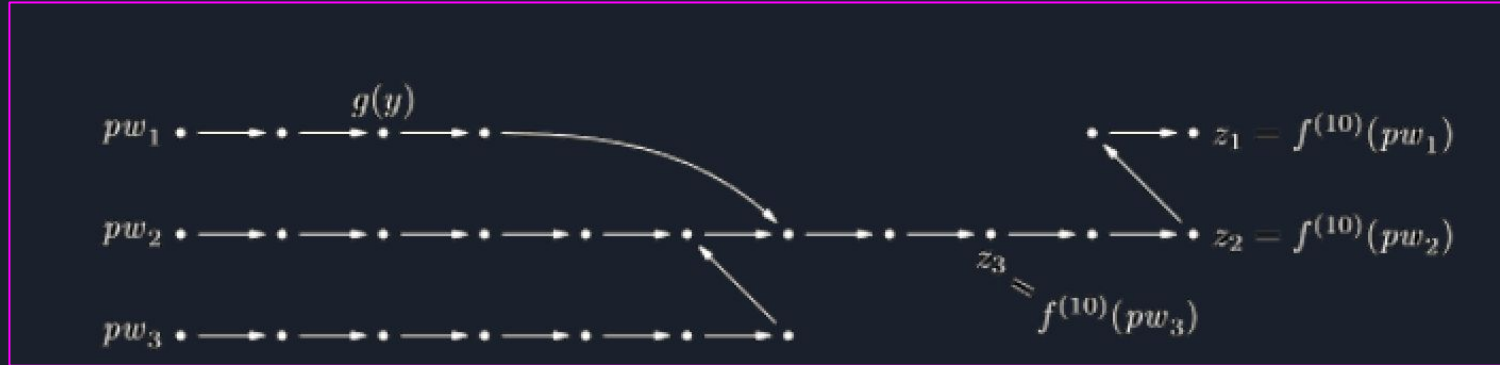


(b) rainbow tables

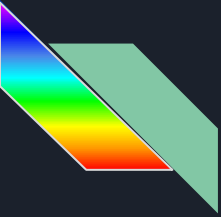
Hellman's Time-Space tradeoff →

Why it kind of sucks

Chain collisions!



This is the *chain merge problem*.



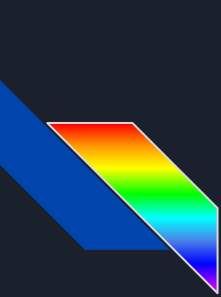
Rainbow table algorithm → *Merge Problem Bypass*

A solution is to use an *independent* reduction function for every column:

$g_i: y \rightarrow p$ for all columns $i=1, \dots, \tau$

The preprocessing phase is otherwise identical, producing the same output L :

$$f_i(pw) = g_i(h(pw))$$



Revised Rainbow table algorithm → *Pseudocode*

Input

$h: p \rightarrow v$
(random function)

p = passwords

v = possible
ciphertexts

$N := |p|$

Parameters τ, l

Preprocessing

Algorithm A_0 :

for $i = 1, \dots, l$:

$pw_i \xleftarrow{R} p$

$z_i \leftarrow f^{(\tau)}(pw_i) \in p$

Return $L := \{(pw_1, z_1), \dots, (pw_l, z_l)\} \subseteq p^2$

Generate chains of length τ .

Attack

Algorithm $A_1(L, y)$:

for $i = \tau, \tau-1, \dots, 1$:

$z \leftarrow f_{\tau}(f_{\tau-1}(\dots f_{i+1}(g_i(y)) \dots)) \in p$

if there exists pw such that $(pw, z) \in L$:


$pw \leftarrow f_{i-1}(\dots f_2(f_1(pw)) \dots)$

If $h(pw) = y$:

return pw

output “fail”

If z is a chain endpoint, traverse the chain from the beginning; if the inverse is found, return it, otherwise move up the chain to the previous one.
If $g(y)$ is not on any chain, it's a failure.



Revised Rainbow Table Algorithm →

Running time

Maximum running time of A_1 :

$$t(A_1) = \tau^2/2$$

This satisfies the original $t(A_1) \times l^2 \approx N^2$ supposition:

$$l^2 \times t(A_1) \geq N^2/2$$

Rainbow tables → *Real-world examples*

You can download preprocessed rainbow tables for several common hash functions; they are meant to be used with a program called *RainbowCrack*.

- Typically available for passwords of length 1-8 characters on old-ish algorithms
 - MD5, SHA1, NTLM

Pictured: download page for a bunch of rainbow tables

| Algorithm | Table ID | Charset | Plaintext Length | Key Space | Success Rate | Table Size | Files |
|-----------|--|------------------------------------|------------------|--|--------------|------------|-----------------------|
| LM | lm_ascii-32-65-123-4#1-7 | ascii-32-65-123-4 | 1 to 7 | 7,555,858,447,479 ≈ 2 ^{42.8} | 99.9 % | 27 GB | Files |
| NTLM | ntlm_ascii-32-95#1-7 | ascii-32-95 | 1 to 7 | 70,576,641,626,495 ≈ 2 ^{46.8} | 99.9 % | 52 GB | Files |
| NTLM | ntlm_ascii-32-95#1-8 | ascii-32-95 | 1 to 8 | 6,704,780,954,517,120 ≈ 2 ^{52.6} | 96.8 % | 460 GB | Files |
| NTLM | ntlm_mixedalpha-numeric#1-8 | mixedalpha-numeric | 1 to 8 | 221,919,451,578,090 ≈ 2 ^{47.7} | 99.9 % | 127 GB | Files |
| NTLM | ntlm_mixedalpha-numeric#1-9 | mixedalpha-numeric | 1 to 9 | 13,759,005,997,841,642 ≈ 2 ^{53.6} | 96.8 % | 690 GB | Files |
| NTLM | ntlm_loweralpha-numeric#1-9 | loweralpha-numeric | 1 to 9 | 104,461,669,716,084 ≈ 2 ^{46.6} | 99.9 % | 65 GB | Files |
| NTLM | ntlm_loweralpha-numeric#1-10 | loweralpha-numeric | 1 to 10 | 3,760,620,109,779,060 ≈ 2 ^{51.7} | 96.8 % | 316 GB | Files |
| MD5 | md5_ascii-32-95#1-7 | ascii-32-95 | 1 to 7 | 70,576,641,626,495 ≈ 2 ^{46.8} | 99.9 % | 52 GB | Files |
| MD5 | md5_ascii-32-95#1-8 | ascii-32-95 | 1 to 8 | 6,704,780,954,517,120 ≈ 2 ^{52.6} | 96.8 % | 460 GB | Files |
| MD5 | md5_mixedalpha-numeric#1-8 | mixedalpha-numeric | 1 to 8 | 221,919,451,578,090 ≈ 2 ^{47.7} | 99.9 % | 127 GB | Files |
| MD5 | md5_mixedalpha-numeric#1-9 | mixedalpha-numeric | 1 to 9 | 13,759,005,997,841,642 ≈ 2 ^{53.6} | 96.8 % | 690 GB | Files |
| MD5 | md5_loweralpha-numeric#1-9 | loweralpha-numeric | 1 to 9 | 104,461,669,716,084 ≈ 2 ^{46.6} | 99.9 % | 65 GB | Files |
| MD5 | md5_loweralpha-numeric#1-10 | loweralpha-numeric | 1 to 10 | 3,760,620,109,779,060 ≈ 2 ^{51.7} | 96.8 % | 316 GB | Files |
| SHA1 | sha1_ascii-32-95#1-7 | ascii-32-95 | 1 to 7 | 70,576,641,626,495 ≈ 2 ^{46.8} | 99.9 % | 52 GB | Files |
| SHA1 | sha1_ascii-32-95#1-8 | ascii-32-95 | 1 to 8 | 6,704,780,954,517,120 ≈ 2 ^{52.6} | 96.8 % | 460 GB | Files |
| SHA1 | sha1_mixedalpha-numeric#1-8 | mixedalpha-numeric | 1 to 8 | 221,919,451,578,090 ≈ 2 ^{47.7} | 99.9 % | 127 GB | Files |

Rainbow Tables → Time to crack

The 5700XT with a relatively low-end NVMe SSD was able to crack 9-character SHA1 hashes in about 20 minutes.

Takeaway: Technology advances quickly; what was secure a decade ago can be bypassed in half an hour by someone with a \$700 desktop computer nowadays.

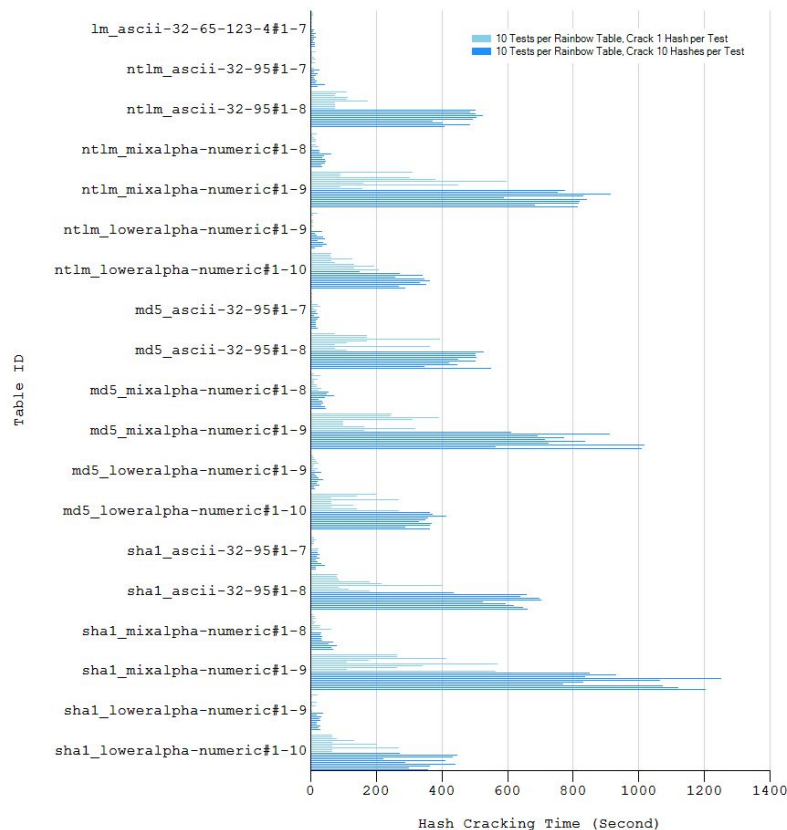
Hash Cracking with Rainbow Tables

Software: RainbowCrack 1.8

GPU: AMD Radeon RX 5700 XT

Memory: 16 GB DDR4

Disk: SSD with Sequential File Read Performance 1600 MB/s





Sources

Everyone's favorite slide

<https://toc.cryptobook.us/book.pdf> - The textbook for this class

<https://www.geeksforgeeks.org/understanding-rainbow-table-attack/> - helped with translating PhD to English

<http://project-rainbowcrack.com/> - the RainbowCrack homepage

<https://pcpartpicker.com/products/> - pricing on various computer parts

Disclaimer: I'm not responsible for what you decide to do with this information.