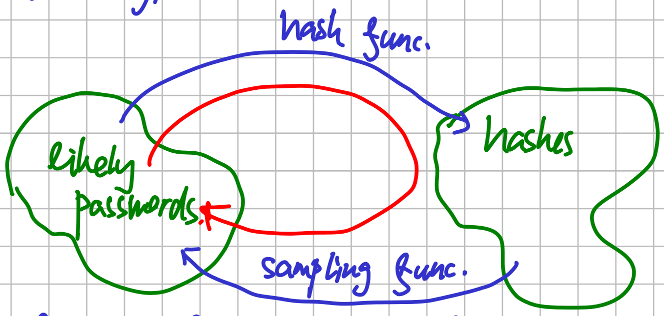
Passwords

- passwd policy
  - length
  - char classes
  - frequent changes
  - similarity to previous passwords
- attacks
  - online
  - offline – e.g., stolen DB of accounts

Solution: hash the password

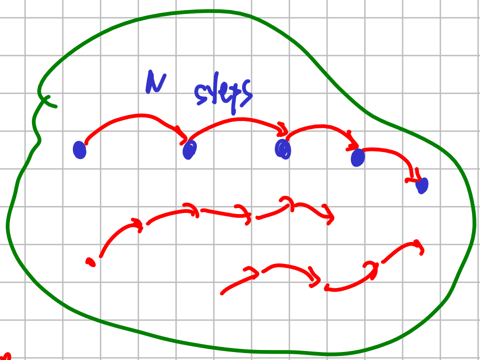
Cracking: ① brute-force (dictionary)

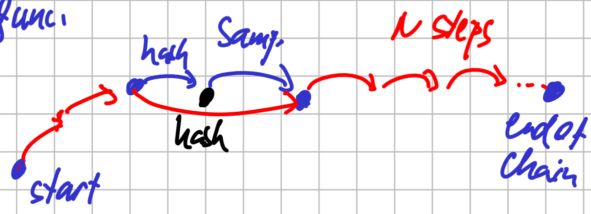
② precomputation



f = hash func. ∘ sampling func.

for each chain: start
we store end



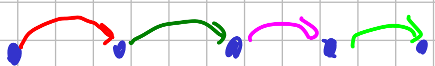


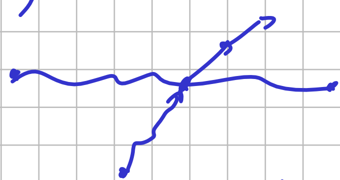
optimistic: just N steps
table of size # passwords / N

↑ for disjoint chains (unlikely)

③ Rainbow table

N colors, i-th edge of chain has color i
N sampling functions (1 per color)





Consequences:
- minimizes common parts of chain
  → almost disjoint chains ... space ≈ #passwds / N
- lookup traverses N steps for each of N possible starting colors
  ↳ we need time N^2 per lookup

Example: project-rainbowcrack.com
for SHA1, ASCII, 1-8-char passwds → 460 GB

<u>Defense</u>: Ⓐ salting of hashes
- for each account, store <u>salt</u> (a nonce)
- hash(salt || passwd)

Ⓑ iterating hashes → key stretching

Ⓒ key strengthening
- discard some bits of the nonce
  → need to brute-force them to validate the password

PBKDF2 Password-Based Key Derivation Function

Inputs: salt
- s := # iterations
- t := # blocks to generate
- keyed PRF (e.g., HMAC)

Output: $B_1, B_2, \dots, B_t$

where: $B_i := B_i^1 \oplus B_i^2 \oplus \dots \oplus B_i^s$

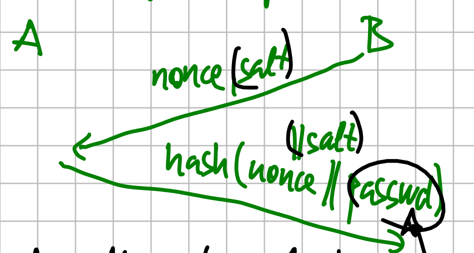
$B_i^1 := PRF(passwd, salt \| i)$

$B_i^{j+1} := PRF(passwd, B_i^j)$

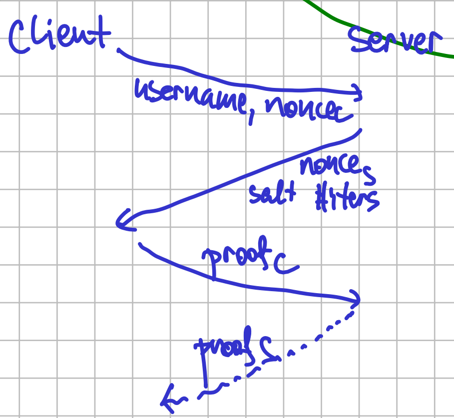
Other choices: functions needing both time & memory
↳ e.g., Argon2.

---

① Challenge-Response auth.



B has the plain-text passwords
→ store the hash? use it passwd



② SCRAM Salted Challenge-Response Authentication Mechanism

- fix salt, # iterations of KDF — KDF of the real passwd
- define: client key K_c := HMAC(passwd, "Client Key")
  server key: K_s := HMAC(passwd, "Server Key")
- client remembers: passwd, username
- server: username, salt, # iterations, K_s, hash(K_c)

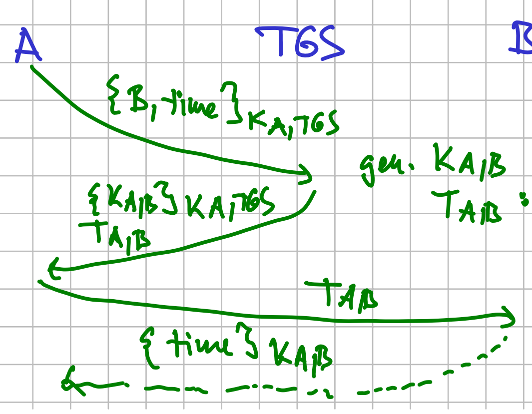
history of relation

$proof_c := K_c \oplus HMAC(H(K_c), Auth)$

$proof_s := HMAC(K_s, Auth)$

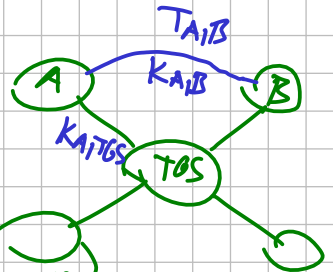
Kerberos distributed key mgmt using symmetric crypto (MIT 198x)

- principals (clients, servers, ...)
- Ticket Granting Service (TGS) – has a shared secret key with each principal ($K_{A,TGS}$)
  - when A wants to talk with B:



gen. $K_{A,B}$

$T_{AB} := (B, \{A, \text{time range}, K_{AB}\}_{K_{B,TGS}})$



Problems: • need clock sync (signed NTP)
  └ little bit off: allow diff. ≤ δ, remember all messages younger than δ

Details (Kerberos v5):
- TGS is also a principal → $\forall A\ \exists T_{A,TGS}$ ← the TGT
- authenticators for establishing session key from $T_{A,B}$:

$$\mathcal{A}_{A,B} := \{A, \text{timestamp}, \text{sess. key}\}_{K_{A,B}}$$

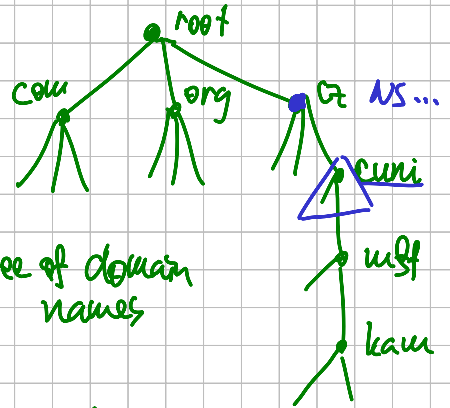
  single-use

- initial auth by password → auth service: produces TGT encrypted by hash(passwd)

  ... but there can be off-line attacks on passwd:
  → pre-authentication: send to auth service: $\{\text{time}\}_{\text{hash(passwd)}}$

DNSSEC protocol | Secure Domain Name System

Review of DNS:



tree of domain names

nodes contain records like:

A	IPv4 address
AAAA	IPv6 address
MX	mail exchanger

- each zone: DNSKEY record ← pub key
- each pair (name, type): RRSIG record
  signing records for name+type using the DNSKEY (private)
  - works offline!
- each NS delegation is accompanied with DS record: hash of DNSKEY in the delegated zone
- root of trust (root, keys of private domains)
- multiple keys per zone — key rotation
  └ zone signing × key signing ← DS
  (key signing signs this → zone signing)

- proving non-existence?
  ↳ sort the names, sign the gaps

  $N_1 < N_2 < N_3 < \ldots < \ldots$

  sign. on the gap (N_1, N_2)

  Actually: N_1 NSEC N_2 (types for N_1)

but... makes it easy to enumerate all names → NSEC3 ... chain of hashes