

## 1. Some sample cryptographic hash functions

- a. Hash functions are deterministic (the same input provides the same output → not randomized)

```

15  /***** HEADER FILES *****/
16  #include <stdlib.h>
17  #include <memory.h>
18  #include "sha256.h"
19
20  /***** MACROS *****/
21  #define ROTLEFT(a,b) (((a) << (b)) | ((a) >> (32-(b))))
22  #define ROTRIGHT(a,b) (((a) >> (b)) | ((a) << (32-(b))))
23
24  #define CH(x,y,z) (((x) & (y)) ^ ~(x) & (z))
25  #define MAJ(x,y,z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
26  #define EP0(x) (ROTRIGHT(x,2) ^ ROTRIGHT(x,13) ^ ROTRIGHT(x,22))
27  #define EP1(x) (ROTRIGHT(x,6) ^ ROTRIGHT(x,11) ^ ROTRIGHT(x,25))
28  #define SIG0(x) (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))
29  #define SIG1(x) (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))
30
31  /***** VARIABLES *****/
32  static const WORD k[64] = {
33      0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1,0x923f82a4,0xab1c5ed5,
34      0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,0x72be5d74,0x80deb1fe,0x9bdc06a7,0xc19bf174,
35      0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0x76f988da,
36      0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147,0x06ca6351,0x14292967,

```

- b.
- c. SHA256, SHA512, SHA3

## 2. Properties of cryptographic hash functions

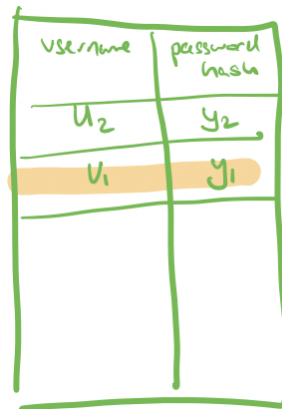
- a. One-wayness: Given a target  $y$ , it is hard to find a “preimage”  $x$ , where  $H(x) = y$
- b. Collision resistance: it is hard to find two “preimages”  $x, x'$  where  $H(x) = H(x')$

## 3. Password hashing

- |         |              |               |
|---------|--------------|---------------|
| a. User | Login Server | Password File |
|---------|--------------|---------------|

Password → HTTPS →  $H(\text{password}) = y$  → find the match of username  $u$  with the value  $y$

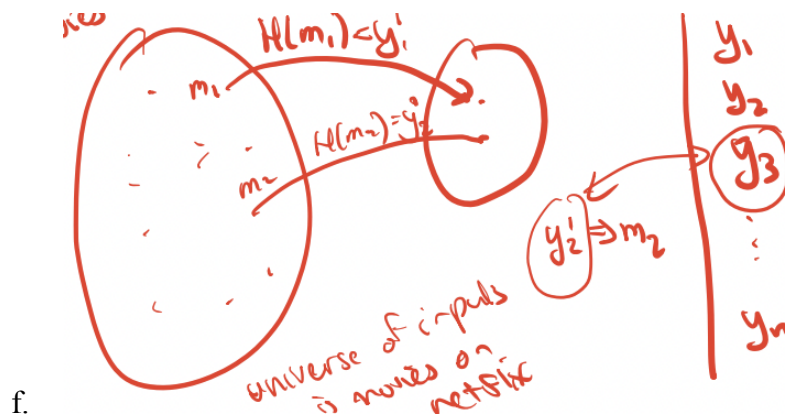
If there exists a match of  $u$  and  $y$  that matches, it validates the user and logs the user in



username	password hash
$u_2$	$y_2$
$u_1$	$y_1$

- b. Password File:
- c. User passes in a strong password over HTTPS and sends it to the login server, which uses the hash function to produce a value of  $y$ . If the match of  $u$  and  $y$  exist in the password file, users will be logged into the website
4. Why don't we store passwords encrypted?
- If we have the passwords encrypted ( $\text{loginServer}_k()$ ), the login server would have to run the encryption algorithm every time user passes in a password as an input
  - If the login server was compromised, the adversary learns the key and can decrypt all passwords
  - However by using the hash function, if the adversary compromises the login server, it should be "hard" to recover all the passwords
  - If the space of inputs is too small, there can be a dictionary attack (method of breaking into password-protected device by systematically entering every word in a dictionary as a password)

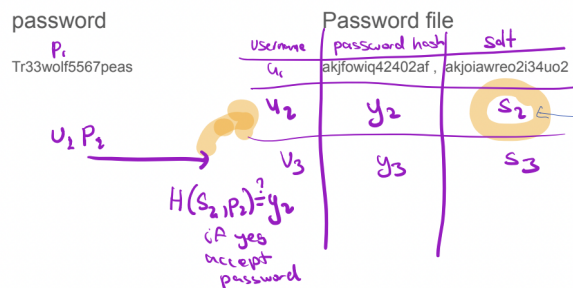
- e. For example, if the inputs = movies on netflix (assume there are about 10,000 movies in netflix) → adversary can try all the 10,000 inputs and learn its outputs



- g. Therefore, the adversary can be given a target  $y$  to find the input  $x$  (since the adversary knows which  $y$  corresponds to  $x$  after systematically doing all the entering) due to the small number of inputs
- h. Notice that “choosing a strong password” makes it more unlikely that your password will be part of a dictionary → do not make stupid and easy passwords

## 5. Salting your password file

- a. If the hash function is used straight where you simply hash the password, the adversary can take this file and use to attack numerous password files



$$y_i = H(s_i, p_i)$$

$$y_2 = H(s_2, p_2)$$

concatenate

→ password hash = hash function of salt and password

- b.
  - c. Therefore, you “salt the password” (salt value is chosen randomly)
  - d. It is a technique to protect passwords stored in databases by adding a string of 32 or more characters and then hashing them
  - e. Note: the salt is stored in the clear! (adversary also gets the salt)
  - f. Salting → raises the level of difficulty → it causes a precomputed dictionary to become useless → the dictionaries that use the same hash function all become different due to the random salt value tied to each user
  - g. Even though they can crack smaller number of passwords (can find out the passwords for a small number of people or specific people by trying all the different combinations of password with the salt value), they need to put in lots of resources to find the passwords for every user
6. Public key
- a. Even if the public key PK is known to everyone, it does not reveal anything about the secret key
  - b. Digital key - code signing, bitcoin mining
7. What’s the problem with symmetric crypto?
- a. It requires pairwise key distribution (aka how do we get shared keys to Alice and Bob)
8. Asymmetric crypto aka public key crypto
- a. Alice has key pair (secret key, public key)

- b. Pk is Alice's public key; everyone (including the adversary) knows the public key
- c. Sk is Alice's secret key. Only Alice knows the secret key
- d. Important – knowledge of the public key does not reveal anything about the secret key

9. Defining a digital signature scheme

- a. Plaintext  $m$                       signature  $o = \text{Sign}(m)$                        $(sk, pk)$
- b. Correctness:  $\text{Ver}(m, \text{Sign}(m)) = 1$  [Validly signed message verifies correctly]
- c. Security: The adversary cannot forge the signature without Sk
- d. Can adversary verify the signature? Yes (anyone can verify the message)