Lecture 7: Hash functions, MACs and authenticated encryption

TTM4135

Relates to Stallings Chapters 11 and 12

Spring Semester, 2025

Motivation

- Hash functions are versatile cryptographic functions used as a building block for authentication
- Message authentication codes (MACs) are a symmetric key cryptographic mechanisms providing authentication and integrity services
- The standardised MAC, HMAC, can be based on many different hash functions and is often used in the TLS protocol
- Authenticated encryption combines confidentiality and authentication in one mechanism
- GCM is a standardised authenticated encryption algorithm also often used in TLS

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

Passwords and hashing

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

Passwords and hashing

Hash functions

A *hash function* H is a public function such that:

- H is simple and fast to compute
- H takes as input a message M of arbitrary length and outputs a message digest $H(M)$ of fixed length

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

Passwords and hashing

Security properties of hash functions

Collision resistant:

- It should be infeasible to find any two different values x_1 and x_2 with $H(x_1) = H(x_2)$.

One-way (or preimage resistant):

- Given a value y it should be infeasible to find any input x with $H(x) = y$.

Second-preimage resistant:

- Given a value x_1 it should be infeasible to find a different value x_2 with $H(x_1) = H(x_2)$.

An attacker who can break second-preimage resistance can also break collision resistance.

The birthday paradox

- In a group of 23 randomly chosen people, the probability that at least two have the same birthday is over 0.5.
- In general, if we choose around $\sqrt{M}$ values from a set of size M, the probability of getting two values the same is around 0.5
- Suppose a hash function H has an output size of k bits. If H is regarded as a random function then $2^{k/2}$ trials are enough to find a collision with probability around 0.5.
- Today 2^{128} trials would be considered infeasible. Therefore, in order to satisfy collision resistance, hash functions should have output of at least 256 bits.

Birthday paradox example, $M = 100$

# trials	Collision prob.	# trials	Collision prob.
1	0	13	.55727
2	.01000	14	.61483
3	.02980	15	.66876
4	.05891	16	.71845
5	.09656	17	.76350
6	.14174	18	.80371
7	.19324	19	.83905
8	.24972	20	.86964
9	.30975	21	.89572
10	.37188	22	.91762
11	.43470	23	.93575
12	.49689	24	.95053

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

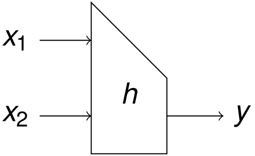
Passwords and hashing

Iterated hash functions

- Cryptographic hash functions need to take arbitrary-sized input and produce a fixed size output.
- As we saw from block ciphers, we can process arbitrary-sized data by having a function that processes fixed-sized data and use it repeatedly.
- An *iterated hash function* splits the input into blocks of fixed size and operates on each block sequentially using the same function with fixed size inputs.
- Merkle–Damgård construction: use a fixed-size *compression function* applied to multiple blocks of the message.

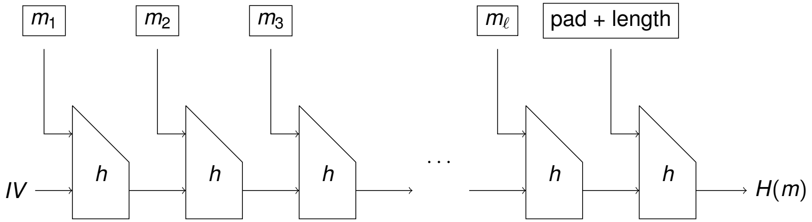
Compression function h

h takes two n-bit input strings x_1 and x_2 and produces an n bit output string y.



Merkle–Damgård construction

1. Break message m into n-bit blocks $m_1 \| m_2 \| \ldots \| m_\ell$.
2. Add padding and an encoding of the length of m. This process may, or may not, add one block.
3. Input each block into compression function h along with chained output; use IV to get started.



Use of Merkle–Damgård construction

Security: If compression function h is collision-resistant, then hash function H is collision-resistant. Proof on blackboard.

But also some security weaknesses:

- Length extension attack: once you have one collision, easy to find more
- Second-preimage attacks not as hard as they should be

Many standard, and former standard, hash functions are Merkle–Damgård constructions: MD5, SHA-1, SHA-2 family

Outline

- Hash functions
  - Security properties
  - Iterated hash functions
  - Standardized hash functions
  - Using hash functions
- Message Authentication Codes (MACs)
  - MACs from hash functions – HMAC
- Authenticated encryption
  - Combining encryption and MAC
  - Galois Counter Mode (GCM)
- Passwords and hashing

MDx family

- Proposed by Rivest and widely used throughout 1990s
- Deployed family members were MD2, MD4 and MD5
- All have 128-bit output
- All of them are broken (concrete collisions have been found)
- In 2006, MD5 collisions could be found in one minute on a PC

SHA-0 and SHA-1

- Based on MDx family design but larger output and more complex
- Originally Secure Hash Algorithm published by US standard agency NBS (now called NIST) in 1993 and later given name SHA-0
- Replaced by SHA-1 in 1995 with very small change to algorithm
- Both SHA-0 and SHA-1 have 160 bit output.
- SHA-0 has been broken (collisions found in 2004)
- First SHA-1 collision found in 2017 - attack is 100 000 times faster than brute force search

SHA-2 family

Developed in response to (real or theoretical) attacks on MD5 and SHA-1.

	Hash size	Block size	Security match
SHA-224	224 bits	512 bits	2 key 3DES
SHA-512/224	224 bits	1024 bits	2 key 3DES
SHA-256	256 bits	512 bits	AES-128
SHA-512/256	256 bits	1024 bits	AES-128
SHA-384	384 bits	1024 bits	AES-192
SHA-512	512 bits	1024 bits	AES-256

- Standardized in FIPS PUB 180-4 (August 2015)
- Known collectively as SHA-2

Padding in the SHA-2 family

- The message length field is:
  - 64 bits when the block length is 512 bits
  - 128 bits when the block length is 1024 bits
- There is always at least one bit of padding[1]. After the first ‘1’ in the padding, enough ‘0’ bits are added so that after the length field is added there is an exact number of complete blocks.
- Adding the padding and length field sometimes adds an extra block and sometimes does not.

[1] To avoid trivial second preimage attacks.

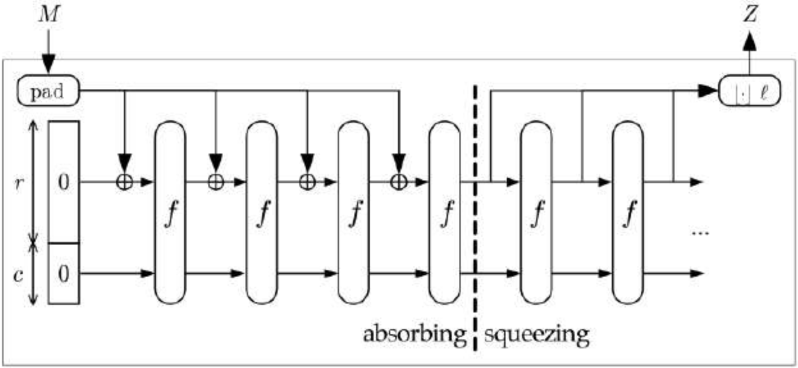
SHA-3

- Late 2000s seen to be a crisis in hash function design
- MDx and SHA family are all based on the same basic design and there have been several unexpected attacks on these in recent years
- In November 2007, NIST announced a competition for a new hash standard called SHA-3
  - Entries closed October 2008; 64 original candidates
  - 14 went through to Round 2, with 5 finalists announced in December 2010
  - Keccak selected as winner in October 2012.
  - Keccak doesn't use compression function as in Merkle–Damgård construction. Instead it uses a *sponge* construction
  - Standardized in FIPS PUB 202, August 2015

The sponge construction

- Input is padded and broken down into blocks of r bits
- The b bits of the state are initialized to zero, and the sponge function proceeds:
  - Absorbing phase: the input blocks are XOR'ed into the r first bits of the state, and the function f is applied iteratively
  - Squeezing phase: the first r bits of the state are returned as output blocks, interleaved with applying the function f.
    - The number of output blocks is chosen by the user.
    - The last c bits of the state are never directly affected by the input blocks and are never output during the squeezing phase.
  - Since the input/ output sizes can be arbitrarily long, the sponge construction can be used to build various primitives (hash functions, stream ciphers, MAC etc).
    - Long input, short output $\rightarrow$ Hash function
    - Short input, large output $\rightarrow$ Key stream

The Keccak function



Source: https://keccak.team/sponge_duplex.html

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

Passwords and hashing

Uses of hash functions

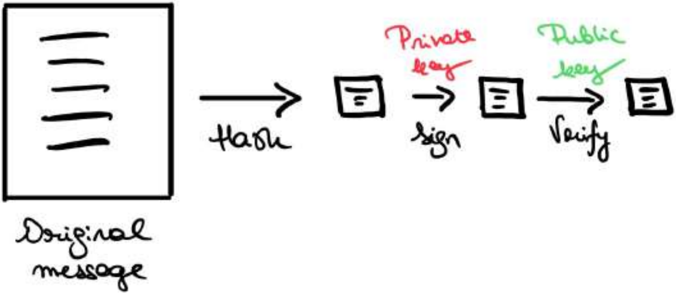
Hash functions have many uses. Note that applying a hash function is *not* encryption:

- Hash computation does not depend on a key*.
- It is not not designed to go backwards to find the input (generally not possible).

While hash functions alone do not provide data authentication, they often help in achieving it:

- Authenticate the hash of a message to authenticate the message.
- Building block for Message Authentication Codes (see HMAC below).
- Building block for signatures (later lecture).

Reminder: signatures



Hash functions and keys

- Sometimes we write hash functions as taking a key s as an input
- $H^s(x) = H(s, x)$
- It must be hard to find a collision for a randomly generated key s
- The key is *not* kept secret; collision resistance must hold even if the adversary has the key s
- This is why we[2] write H^s and not H_s

[2] Katz-Lindell book

Storing passwords for login

Usual to store user passwords on servers using hash functions

- Store salted hashes of passwords: pick random *salt*, compute $h = H(pw, salt)$, store $(salt, h)$
  - easy to check entered password pw': compare stored h and computed $H(pw', salt)$
  - hard to recover pw from $h = H(pw)$ assuming H is preimage resistant
  - attacker needs to store a different dictionary for each *salt*

Note that using a *slower* hash function slows down password guessing

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

Passwords and hashing

Message Authentication Codes (MAC)s

- Recall one of the goals of cryptography is to enable *secure communications*
  - But what does this mean?
- We have covered *secrecy* so far (i.e. *hiding* the message)
- But *integrity can be even more important*
- For example, a bank receives a transfer request from user Alice to user Bob in the amount of $ 10,000
  - Did it really come from Alice?
  - Is the amount correct? Was it modified during transmission?

Encryption vs Message Authentication

- Encryption *does not* guarantee message integrity
- These are *different* notions
- Recall that we saw that flipping certain bits in the ciphertext results in flipping certain bits in the plaintext
- If the adversary also has partial information about the plaintext (e.g. an estimate of the amount that is being sent), it can predict with some accuracy what the changes are
  - Even the OTP is vulnerable to this, so this does not mean that the encryption scheme is not secure!
- An adversary can also randomly change the ciphertext, to ruin the underlying message!

Message Authentication Code (MAC)s

- A *message authentication code (MAC)* is a cryptographic mechanism used for message integrity and authentication
- On input a secret key K and an arbitrary length message M, a MAC algorithm outputs a fixed-length tag:

$$T = \mathrm{MAC}(M, K),$$

- A MAC is a symmetric key algorithm: sender and receiver both have the secret key K
- The sender sends the pair (M, T) but M may or may not be encrypted
- The recipient recomputes the tag $T' = \mathrm{MAC}(M', K)$ on the received message M' and checks that $T' = T$

MAC properties

The basic security property of a MAC is called *unforgeability*:

- It is not feasible to produce a message M and a tag T such that $T = \mathrm{MAC}(M, K)$ without knowledge of the key K

The more complete notion of security is *unforgeability under chosen message attack*:

- The attacker is given access to a *forging oracle*: on input any message M of the attacker's choice the MAC tag $T = \mathrm{MAC}(M, K)$ is returned
- It is not feasible for the attacker to produce a valid (M, T) pair that was not already asked to the oracle

Not a signature scheme, but can be thought of as the symmetric version of a signature scheme. Here the point is that only *authorised* entities can authenticate messages.

MAC properties

The basic security property of a MAC is called *unforgeability*:

- It is not feasible to produce a message M and a tag T such that $T = \mathrm{MAC}(M, K)$ without knowledge of the key K

The more complete notion of security is *unforgeability under chosen message attack*:

- The attacker is given access to a *forging oracle*: on input any message M of the attacker's choice the MAC tag $T = \mathrm{MAC}(M, K)$ is returned
- It is not feasible for the attacker to produce a valid (M, T) pair that was not already asked to the oracle

Not a signature scheme, but can be thought of as the symmetric version of a signature scheme. Here the point is that only *authorised* entities can authenticate messages.

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

Passwords and hashing

HMAC

- Proposed by Bellare, Canetti, Krawczyk in 1996
- Built from any iterated cryptographic hash function H, e.g., MD5, SHA-1, SHA-256, ...
- Standardized and used in many applications including TLS and IPsec
- Details in FIPS-PUB 198-1 (July 2008)

HMAC construction

Let H be any iterated cryptographic hash function. Then define:

$$\mathrm{HMAC}(M, K) = H(\ (K \oplus \mathrm{opad}) \parallel H((K \oplus \mathrm{ipad}) \parallel M)\ )$$

where

- M: message to be authenticated
- K: key padded with zeros to be the block size of H
- opad: fixed string `0x5c5c5c...5c`
- ipad: fixed string `0x363636...36`
- $\parallel$ denotes concatenation of bit strings.
- HMAC is secure (unforgeable) if H is collision resistant or if H is a pseudorandom function.

HMAC

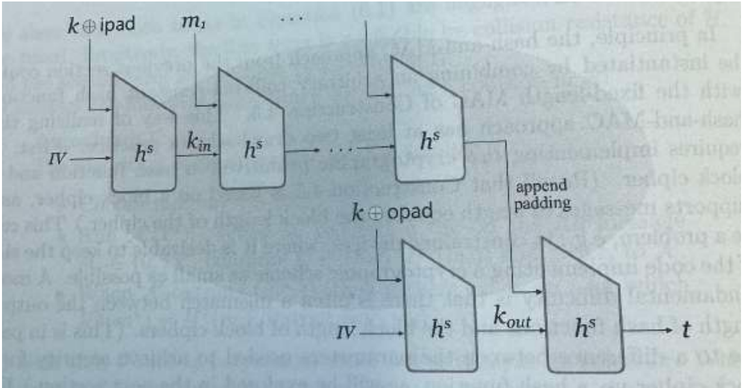


FIGURE 6.2: HMAC, pictorially.

Source: Katz-Lindell book, third edition.

Security of HMAC

- *Security:* HMAC is secure if H is collision resistant or if H is a pseudorandom function.
- HMAC is designed to resist length extension attacks (even if H is a Merkle–Damgård hash function).
- HMAC is often used as a *pseudorandom function* for deriving keys in cryptographic protocols.

H is h^s in the previous slide.

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

Passwords and hashing

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

Passwords and hashing

Authenticated encryption

Suppose Alice and Bob have a shared key K.

Suppose Alice has a message M that she wants to send to Bob with *confidentiality* and *authenticity/integrity*.

How should she do this? Two options:

1. Split the key K into two parts (K_1 and K_2), encrypt with K_1 to provide confidentiality and use K_2 with a MAC to provide authenticity and integrity.
2. Use a dedicated algorithm which provides both properties – this is called *authenticated encryption*.

Combining encryption and message authentication

Three possible ways to combine encryption and MAC are:

Encrypt and MAC: encrypt M, apply MAC to M, and send the two results

MAC then encrypt: apply MAC to M to get tag T, then encrypt $M \| T$, and send the ciphertext

Encrypt then MAC: encrypt M to get ciphertext C, then MAC C, and send the two results

It turns out that *encrypt-then-MAC* is the safest approach.

- $C = \mathrm{Enc}(M, K_1)$
- $T = \mathrm{MAC}(C, K_2)$
- send $C \| T$

Encrypt and MAC

- No integrity on the ciphertext! Only on the plaintext, which can be problematic.
- This may not achieve the most basic level of secrecy.
- Even a strongly secure MAC does not guarantee *anything* about secrecy.
- A MAC may leak information about m in the tag t.
  - Think of a MAC who always outputs the first bits of m in the tag.

MAC then Encrypt

- Plaintext integrity only, but this time the MAC is encrypted.
- Because we pad the message with the tag, we have two sources of decryption error:
  - Padding may be incorrect.
  - Tag may not verify.
- An attacker can distinguish between the failures and exlpoit this.
- This type of attack has been carried out against some TLS configurations.

Authenticated encryption with associated data (AEAD)

- An AEAD algorithm is a symmetric key cryptosystem
- Inputs to the AEAD encryption process are:
  - a message M
  - associated data A
  - the shared key K
- The AEAD output O may contain different elements such as a ciphertext and tag
- The sender sends O and A to the recipient
- The receiver either accepts the message M and data A, or reports *failure*
- Any AEAD algorithm should provide
  - confidentiality for M
  - authentication for both M and A

Outline

Hash functions
- Security properties
- Iterated hash functions
- Standardized hash functions
- Using hash functions

Message Authentication Codes (MACs)
- MACs from hash functions – HMAC

Authenticated encryption
- Combining encryption and MAC
- Galois Counter Mode (GCM)

Passwords and hashing

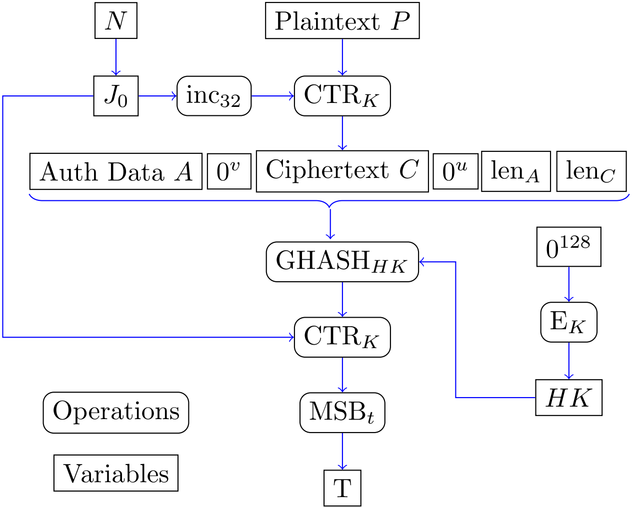
Galois Counter Mode (GCM)

- A block cipher mode providing AEAD
- Most commonly used mode in web-based TLS
- Combines CTR mode on a block cipher (typically AES) with a special keyed hash function called GHASH.
- Standard definition in NIST SP-800 38D
- Due to hardware support of AES and carry-less addition in modern Intel chips, GCM using AES can be faster than using AES with HMAC.

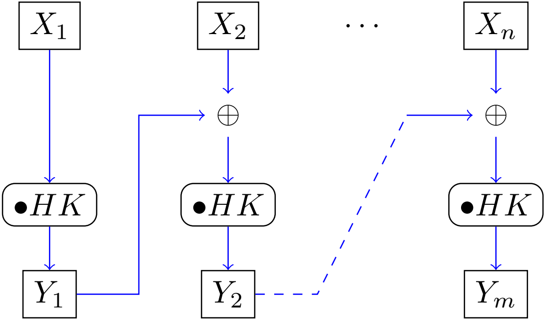
GCM algorithm

- GHASH uses multiplication in the finite field $GF(2^{128})$
  - Generated by the polynomial $x^{128} + x^7 + x^2 + 1$.
- Inputs are plaintext P, authenticated data A, and nonce N
- Values u and v are the minimum number of 0s required to expand A and C to complete blocks
- Outputs are ciphertext C and tag T. The length of A, len_A, and the length of C, len_C, are 64-bit values
- In TLS the length of T is $t = 128$ bits and the nonce N is 96 bits. The initial block input to CTR mode of E (denoted CTR in diagram) is $J_0 = N \parallel 0^{31} \parallel 1$.
- The function inc_{32} increments the right-most 32 bits of the input string by 1 modulo 2^{32}

GCM algorithm



GHASH



- Output is $Y_m = GHASH_{HK}(X_1, \ldots, X_m)$
- Operation $\bullet$ is multiplication in the finite field $GF(2^{128})$
- $HK = E(0^{128}, K)$ is the hash subkey.

GCM decryption

- The elements transmitted to the receiver are the ciphertext C, the nonce N, the tag T and the authenticated data A.
- All elements required to recompute the tag T are available to the receiver who shares key K. The tag is recomputed and checked with received tag. If tags do not match then output is declared invalid.
- If the tag is correct then the plaintext can be recomputed by generating the same key stream, from CTR mode, as is used for encryption.

Outline

- Hash functions
  - Security properties
  - Iterated hash functions
  - Standardized hash functions
  - Using hash functions
- Message Authentication Codes (MACs)
  - MACs from hash functions – HMAC
- Authenticated encryption
  - Combining encryption and MAC
  - Galois Counter Mode (GCM)
- **Passwords and hashing**

Cryptography and passwords

Cryptography needs high-entropy secrets:

- RC4, AES-128: 128 bit secret key
- HMAC-SHA1: 160-bit secret key
- AES-256, HMAC-SHA256: 256-bit secret key

128 bits = about 23 character alphanumeric secret

Humans can only memorize low entropy passwords:

- RockYou.com password database compromised in 2009; *password entropy 21.1 bits*

Uses of passwords

Applications of passwords:

- *Login*: system stores password to compare with the value the user types at login to decide whether to allow access. Obviously don't want to store passwords in plaintext on disk.
- *Key derivation*: user remembers a password that will be used to derive a key for encryption, e.g., disk encryption.

Dictionary attacks against passwords

- Attacker obtains a dictionary of passwords sorted by approximate frequency of use.
- Attacker iterates through dictionary from most frequent to least frequent passwords.

Storing passwords for login

How can we store user passwords on hard disks for checking at login?

- Store passwords in plaintext: Bad idea; anyone who gets hard disk can learn password.
- Store passwords encrypted using a secret key: Where do you store the secret key? Becomes a chicken-and-egg problem.
- Store hashes of passwords: $h = H(pw)$
  - Pro: easy to check entered password pw': compare stored h and computed $H(pw')$
  - Pro: hard to recover pw from $h = H(pw)$ assuming H is preimage resistant
  - Con: attacker could store a dictionary of $pw, H(pw)$ and compare with stolen h

Storing passwords for login

How can we store user passwords on hard disks for checking at login?

- Store salted hashes of passwords: pick random *salt*, compute $h = H(pw, salt)$, store $(salt, h)$
  - Pro: easy to check entered password pw': compare stored h and computed $H(pw', salt)$
  - Pro: hard to recover pw from $h = H(pw)$ assuming H is preimage resistant
  - Pro: attacker needs to store a different dictionary for each *salt*
  - Con: doesn't slow down per-password attacks