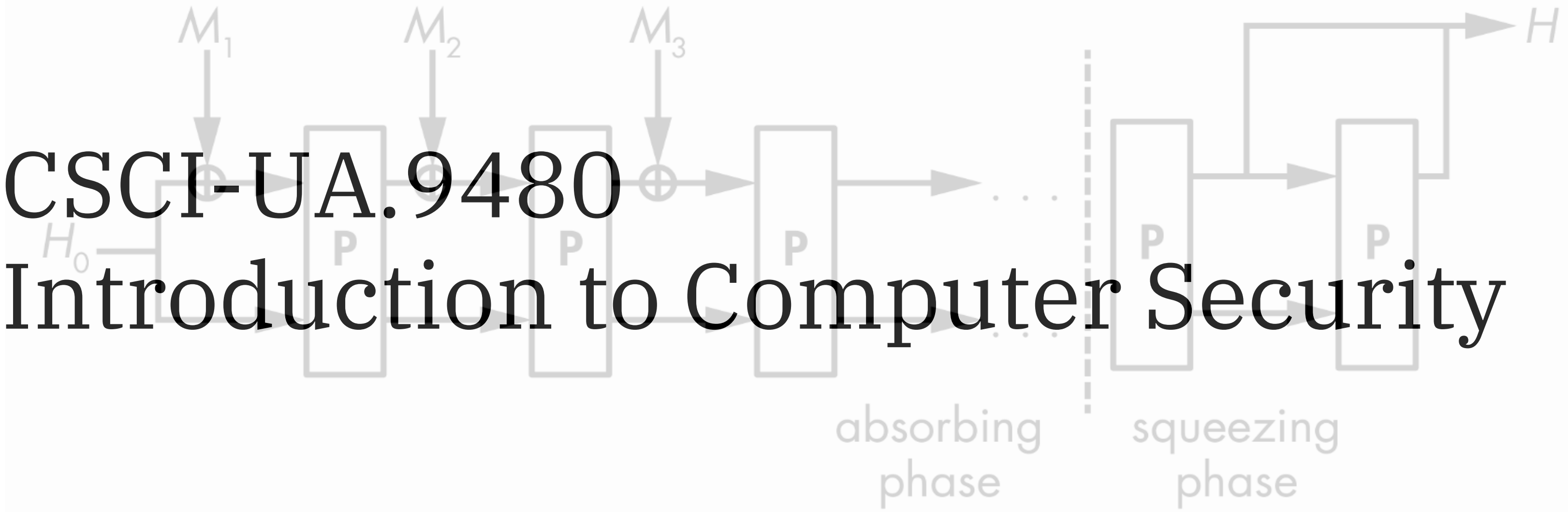


CSCI-UA.9480 Introduction to Computer Security

Session 1.1 One-Way Functions and Hash Functions

Prof. Nadim Kobeissi

Why Hash Functions?

Describing the importance of "the cryptographer's Swiss Army knife."

1.1a

As discussed last time: protocols.

In *protocols*, we reason about:

- Principals: Alice, Bob.
- Security goals: confidentiality, authenticity, forward secrecy…
- Use cases and constraints.
- Attacker model.
- Threat model.

Protocols need to do things.

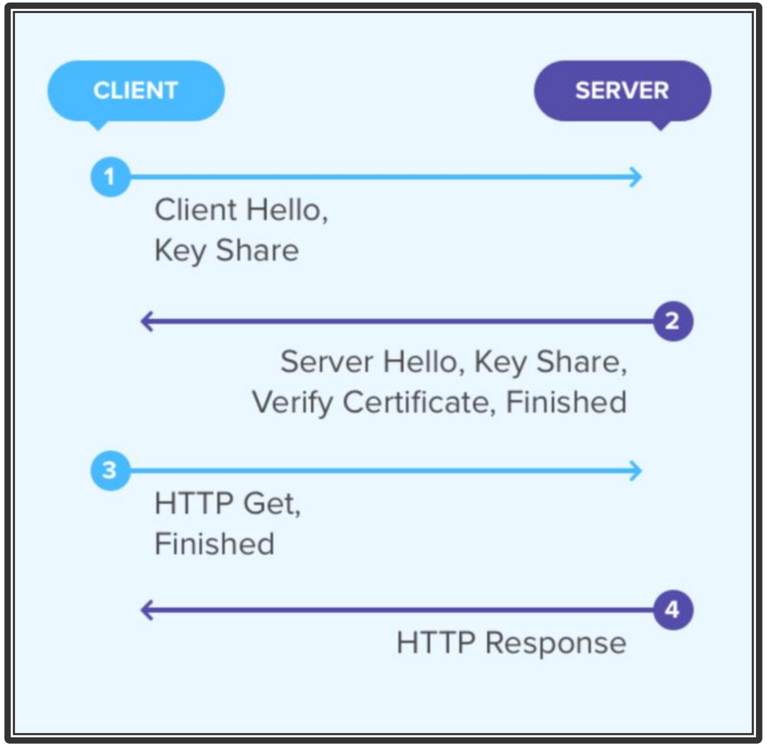
Protocols are frequently entrusted with:

- Communicating secret data without a malicious party being able to read it: *confidentiality*.
- Ensuring that any data Bob receives that appears to be from Alice is indeed from Alice: *authenticity.*
- Limiting the damage that can be caused by device compromise or theft: *post-compromise security.*

Protocols need to do things.

In TLS 1.3 (the latest engine for HTTPS):

- The server *authenticates* itself to the client using signed certificates.
- The client *encrypts* data to the server using ciphers and integrity codes.
- And other things we'll explore later. But for now...



All of these crucial protocols rely on *cryptographic primitives*, which are intricate algorithms that are frequently built from "mathematically hard" foundations or from designs shown to be resistant to cryptanalysis.

“Mathematically hard”: Breaking the security of this cryptographic primitive would be equivalent to solving some math problem that is long-thought to be impossible to solve practically, such as obtaining the discrete logarithm over large prime numbers.

“Resistant to cryptanalysis”: After extensive scrutiny by cryptanalysts, no attack was found to violate the security claims of the design (such as confidentiality, pseudorandomness, etc.)

Protocols need building blocks

Asymmetric primitives.

- *Public key agreement algorithms*: client and server can agree on a secret encryption key over a public channel (wow!)
- *Signature algorithms*: an authority can sign a certificate proving that the server is indeed who it says it is.

Symmetric primitives.

- *Secure hash functions*: the client and the server can generate integrity-preserving codes for encrypted messages.
- *Encryption schemes*: confidential data can be encrypted and exchanged.

What are Hash Functions?

And how are they useful?

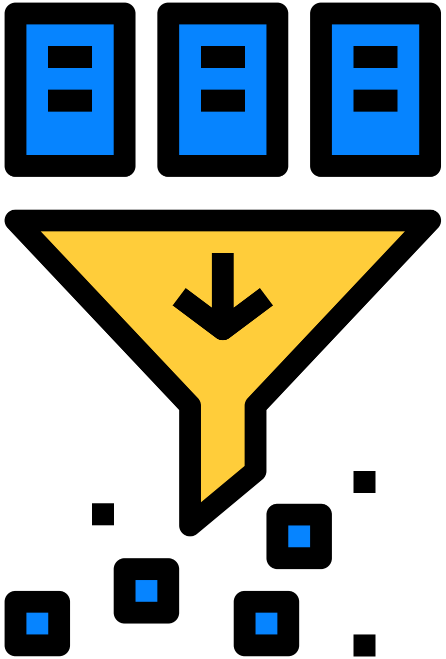
1.1b

OK, so what's a hash function?

Simple!

- A hash function `H(x)` takes some input `x` which can be of any length...
- And produces some value `y` which is of a fixed length (usually 128, 256 , 384 or 512 bits.)

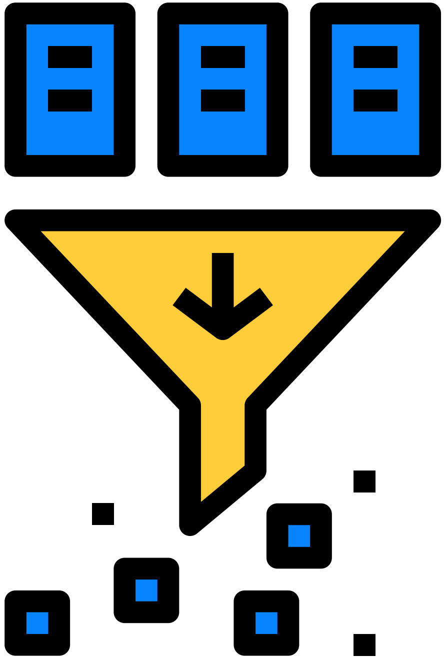
$$H(x) \rightarrow y$$



OK, so what's a *secure* hash function?

A hash function, but…

- Anyone with x can calculate y very easily…
- Going from y back to x is impossible.
- y reveals no information about x (pseudorandom, uniformly chosen.)
- Finding an x′ that also maps to y is extremely improbable.



BLAKE2s(“tomat**o**”) =
5cc655abb6feebac1ba4c24d4b06461a

BLAKE2s(“tomat**e**”) =
75e6179a12dd9303ecdc877aeb6d50ab

Test your knowledge!

Which of the following is an *insecure* hash function?

☐ **A**: MD5.

☐ **B**: BLAKE2.

☐ **C**: SHA2.

Test your knowledge!

Which of the following is an *insecure* hash function?

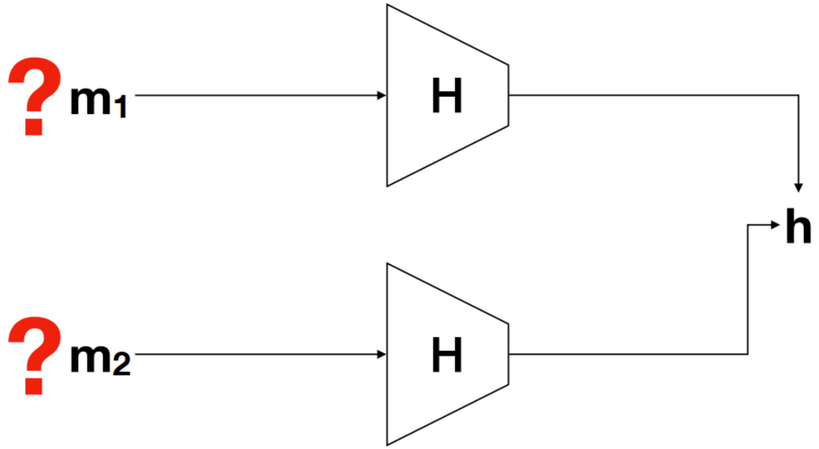
- ☑ **A**: MD5.
- ☐ **B**: BLAKE2.
- ☐ **C**: SHA2.

Which hash functions are safe to use?

Bad (do not use)	Good (do use)
MD4, MD5	BLAKE2 ☺️
SHA1 (shattered.io)	SHA2 (-224, -256, -384, -512)
Non-cryptographic hash functions	SHA3
Cyclic redundancy check (CRC)	
Your own hash function	

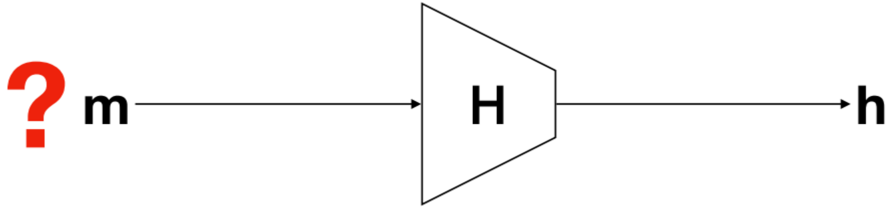
Properties of a secure hash function.

- **Collision resistance.**



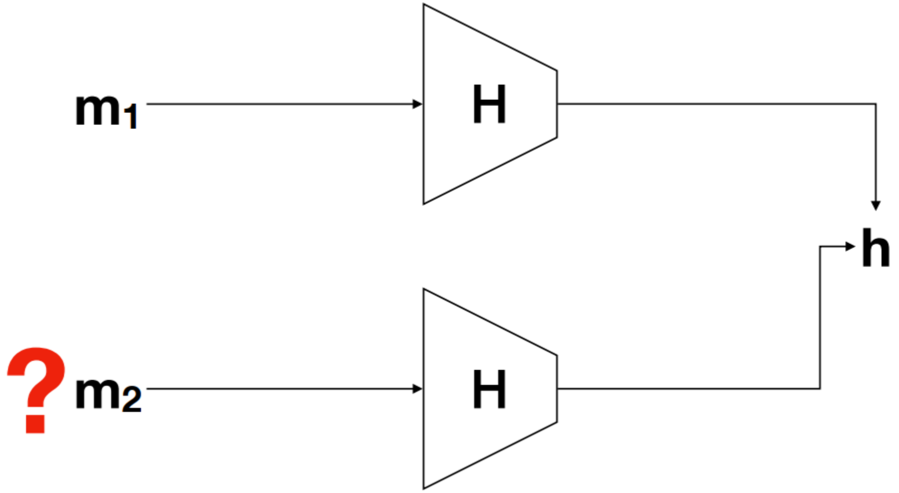
Properties of a secure hash function.

- Collision resistance.
- **Preimage resistance.**



Properties of a secure hash function.

- Collision resistance.
- Preimage resistance.
- **Second preimage resistance.**



Did you know?

Xiaoyun Wang, the Chinese researcher who first broke MD5, had her results initially rejected at USENIX because the translation of the book she was using got the endianness wrong.

How are hash functions useful?

Let's say you want to send a secret message.

- You encrypt a plaintext and get a ciphertext.
- You give your ciphertext to your courier, who is also the Devil (oh, no!)
- The courier switches your ciphertext for another one! What now?!

A wild attacker appears!

How can we use hash functions to prevent the Devil from tampering with our plaintext?

- ☐ **A**: Send `H(plaintext)` along with the encrypted message.
- ☐ **B**: Send `H(ciphertext)` along with the encrypted message.

A wild attacker appears!

How can we use hash functions to prevent the Devil from tampering with our plaintext?

- ☐ **A**: Send `H(plaintext)` along with the encrypted message.
- ☐ **B**: Send `H(ciphertext)` along with the encrypted message.

So unfair! What can we do?!

A wild attacker appears!

How can we use hash functions to prevent the Devil from tampering with our plaintext?

☐ **A**: Send `H(plaintext)` along with the encrypted message.

☐ **B**: Send `H(ciphertext)` along with the encrypted message.

☐ **C**: Send `H(key||ciphertext)` with encrypted message.

A wild attacker appears!

How can we use hash functions to prevent the Devil from tampering with our plaintext?

- ☐ **A**: Send `H(plaintext)` along with the encrypted message.
- ☐ **B**: Send `H(ciphertext)` along with the encrypted message.
- ☐ **C**: Send `H(key||ciphertext)` with encrypted message.

Oh no!!!

A wild attacker appears!

How can we use hash functions to prevent the Devil from tampering with our plaintext?

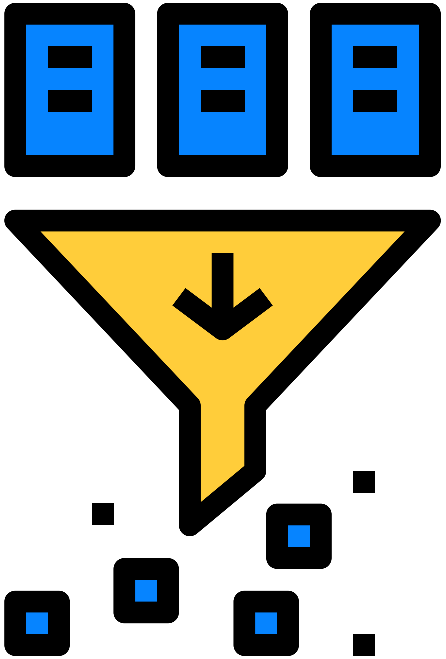
- ☐ **A**: Send `H(plaintext)` along with the encrypted message.
- ☐ **B**: Send `H(ciphertext)` along with the encrypted message.
- ☐ **C**: Send `H(key||ciphertext)` with encrypted message.
- ☑ **D**: Send **HMAC**`(key, ciphertext)` with encrypted message.

Hash functions can preserve integrity.

What we created is a “hash-based message authentication code (HMAC.)

- Options A and B can be created by the Devil.
- Option C is somewhat sensible, but vulnerable to collisions.
- HMACs are a construction that avoid this problem (*opad* and $ipad$ are constants, key size is set):

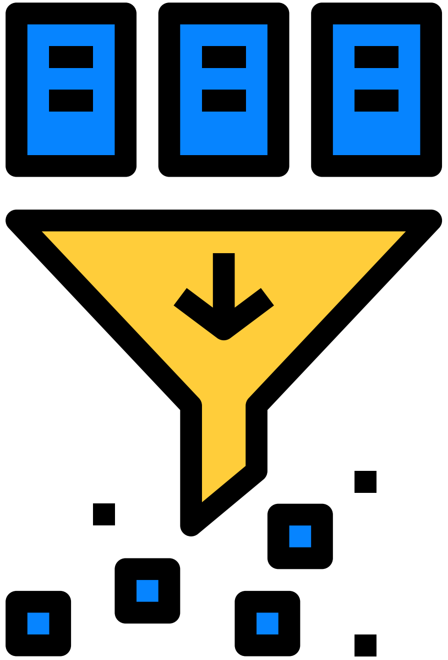
$$\text{HMAC}(K, m) = H\Big((K' \oplus opad) \| H\big((K' \oplus ipad) \| m\big)\Big)$$



Hash functions can preserve integrity.

But what if you send the same message twice?

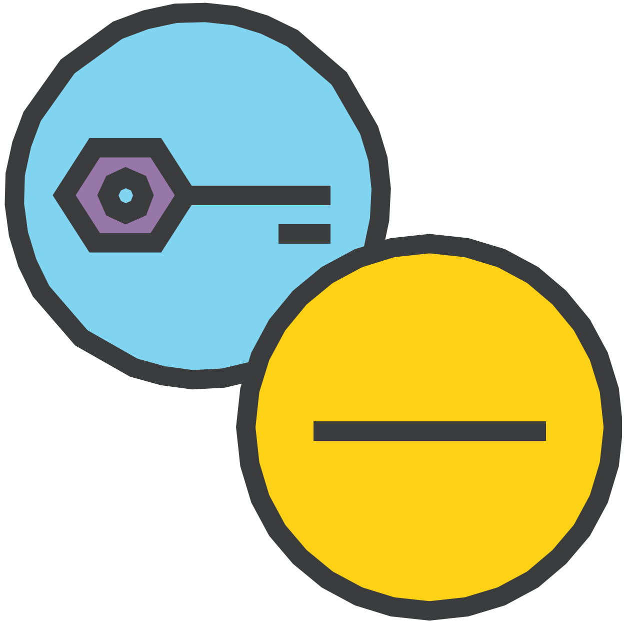
- Same ciphertext. Same HMAC. That’s a distinguisher.
- May also allow for replay attacks.
- That’s why we use *nonces* (***n***umbers used ***once***.)



Hash functions: not just for message integrity.

Another big use case: login authentication.

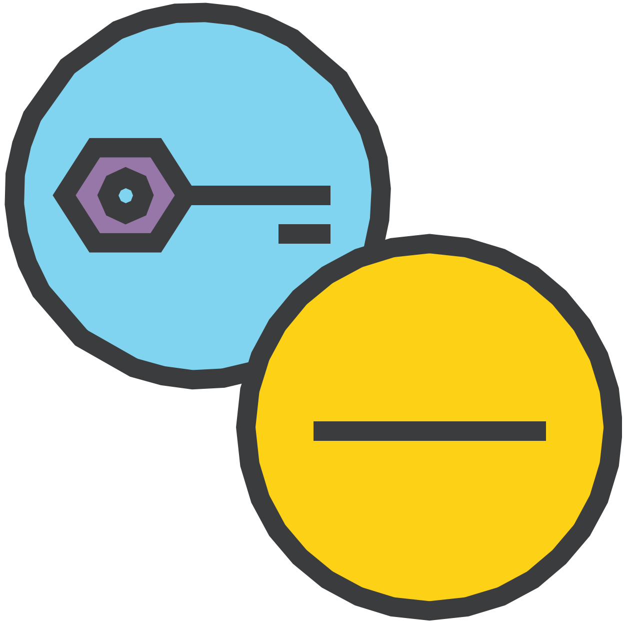
- Storing user passwords on a single server is a bad idea: what if the server gets compromised?
- Storing a hash of the password: better idea.
- Storing a *salted* hash: even better.



Hash functions: not just for message integrity.

Salting and password hashing?

- A salt is a nonce that helps us avoid getting the same hash for the same passwords, and makes hashes less susceptible to lookup-table ("rainbow table")-based attacks.
- A "password hashing" function is an intentionally very slow and expensive hash function that makes brute forcing more expensive. Examples: scrypt, Argon2.

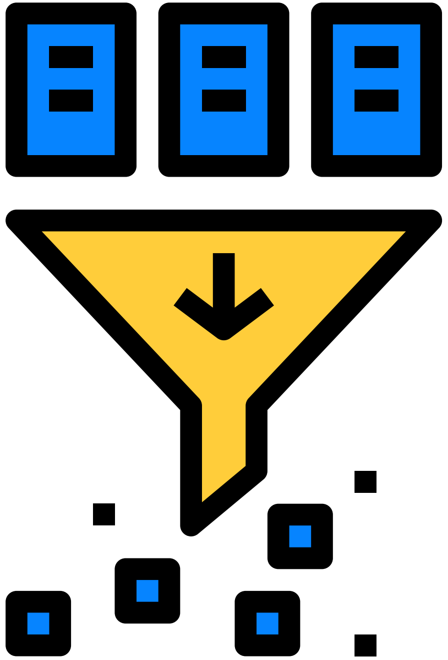


Hash functions: not just for message integrity.

Many other use cases:

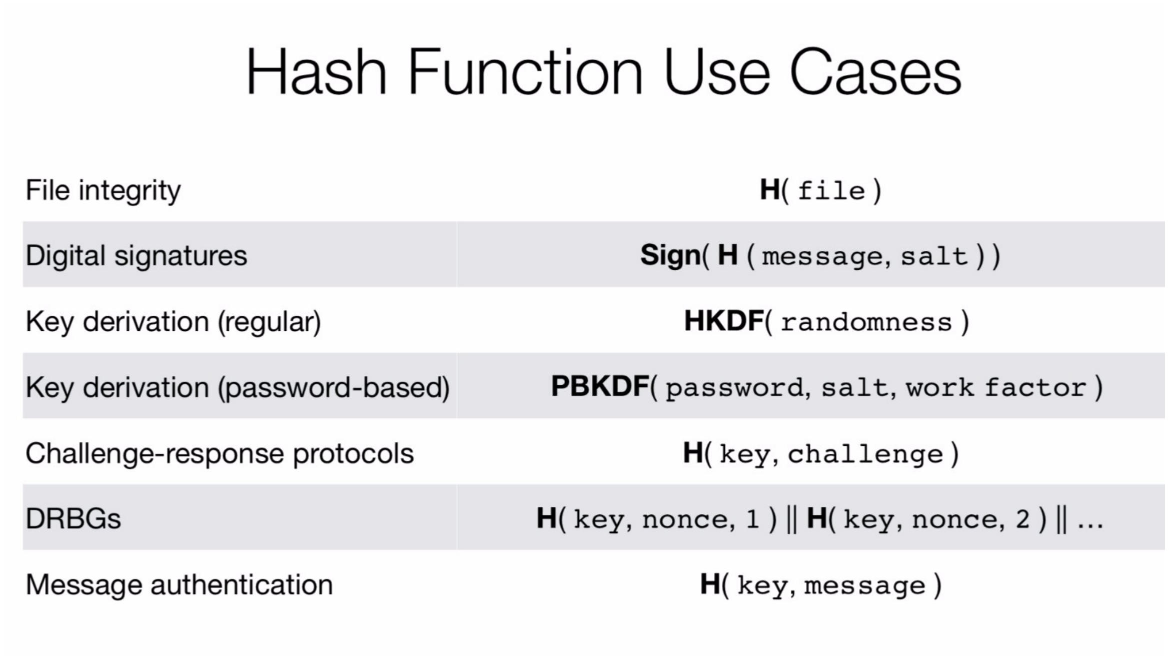
- Quickly scanning for file integrity: generate a hash and match it later.
- Identifying malware samples.
- Proof-of-work.
- Even database sharding!

You can even build encryption schemes and digital signature algorithms out of a hash function!



Hash functions: not just for message integrity.

Git alone uses hash functions in so many different ways:



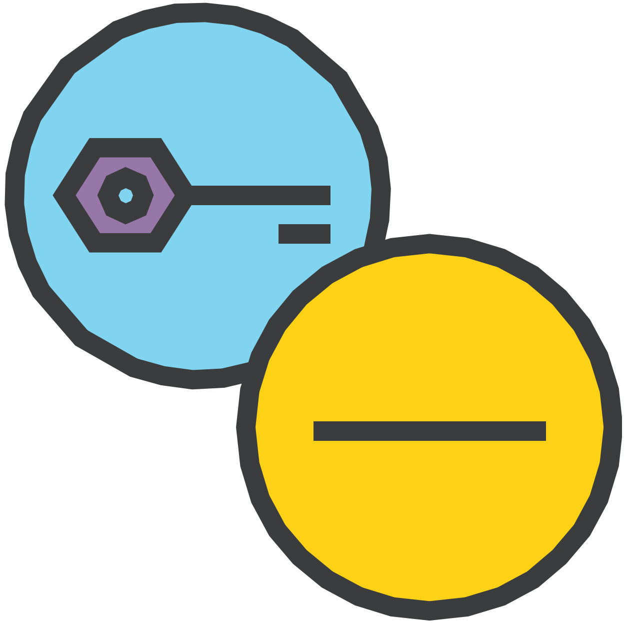
Hash Function Use Cases

File integrity	**H**(file)
Digital signatures	**Sign**(**H** (message, salt))
Key derivation (regular)	**HKDF**(randomness)
Key derivation (password-based)	**PBKDF**(password, salt, work factor)
Challenge-response protocols	**H**(key, challenge)
DRBGs	**H**(key, nonce, 1) || **H**(key, nonce, 2) || …
Message authentication	**H**(key, message)

Password hashing: PBKDF, bcrypt and scrypt

Salting and password hashing?

- *PBKDF2*: Essentially just performs a salted HMAC a certain number of iterations. 10,000+ recommended.
- *Bcrypt*: CPU intensive like PBKDF2, but also RAM intensive.
- *Scrypt*: “Maximally memory hard”; can you think of which attack this can help prevent?



Next time: Symmetric Key Encryption

AES and more.

1.2