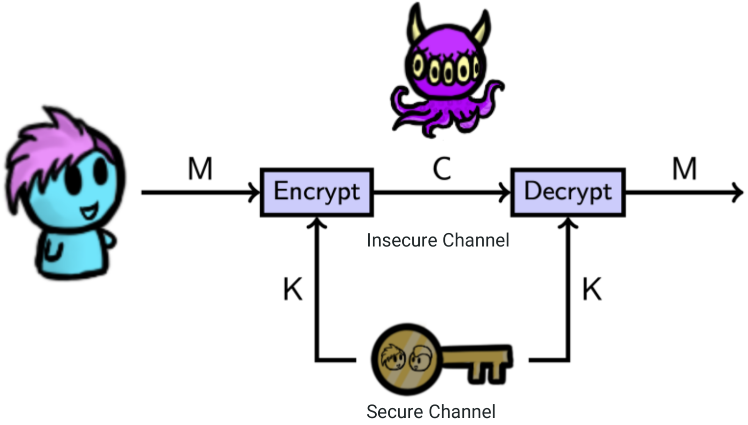
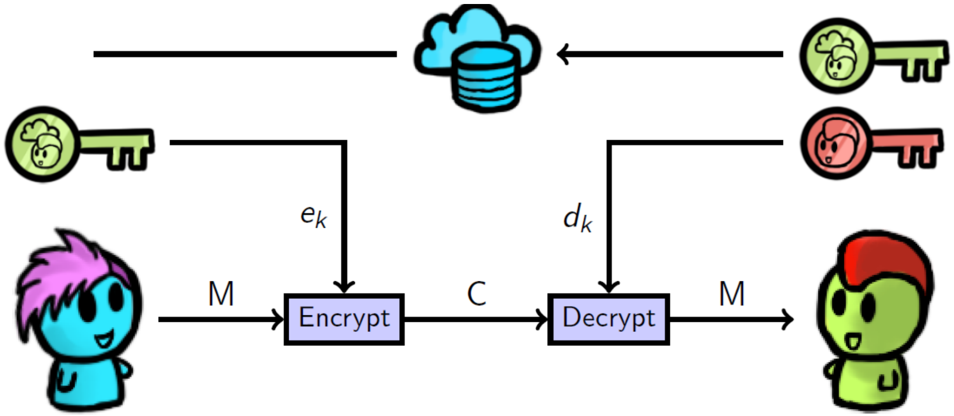
CS459/698 Privacy, Cryptography, Network and Data Security

Integrity and Authenticated Encryption

Fall 2025, Tuesday/Thursday 8:30-9:50am

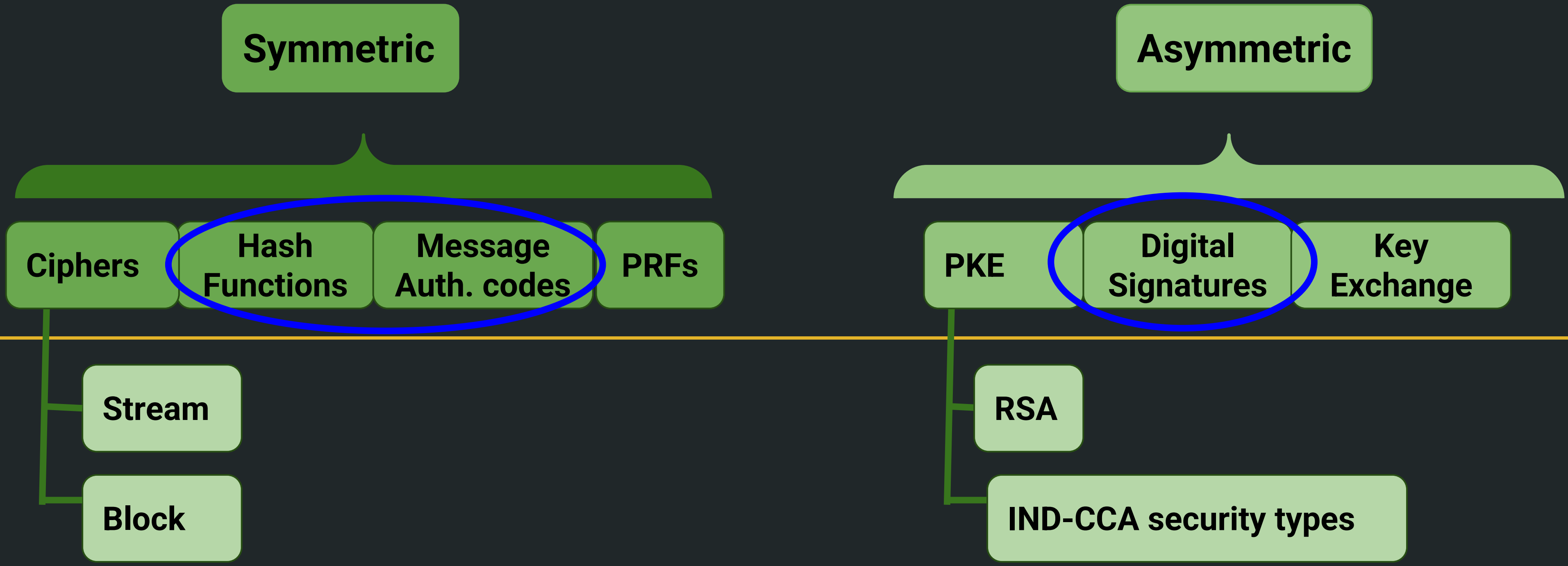
Block/Stream Ciphers, Public Key Cryptography…





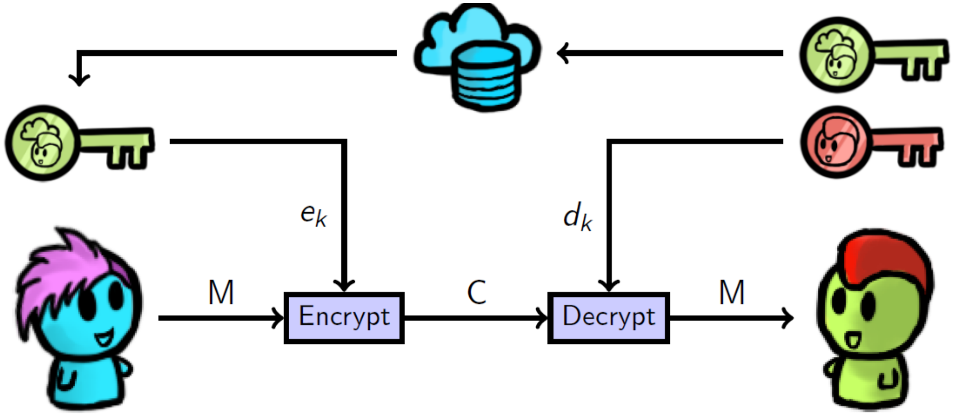
Is that all there is?

Goal: How do we make sure that Bob gets the same message Alice sent?


Symmetric
Asymmetric
Ciphers
Hash Functions
Message Auth. codes
PRFs
PKE
Digital Signatures
Key Exchange
Stream
Block
RSA
IND-CCA security types

Integrity components

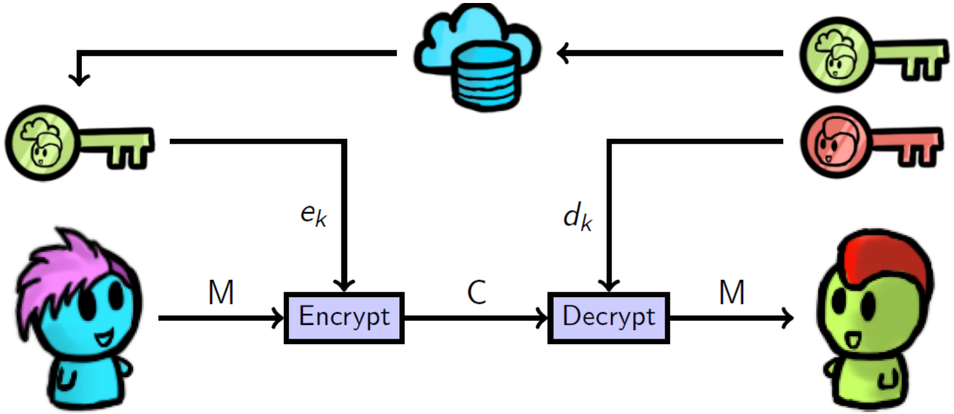
How do we tell if a message has changed in transit?



...wait...is this the message Alice sent?

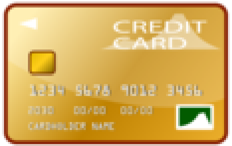
Integrity components

How do we tell if a message has changed in transit?



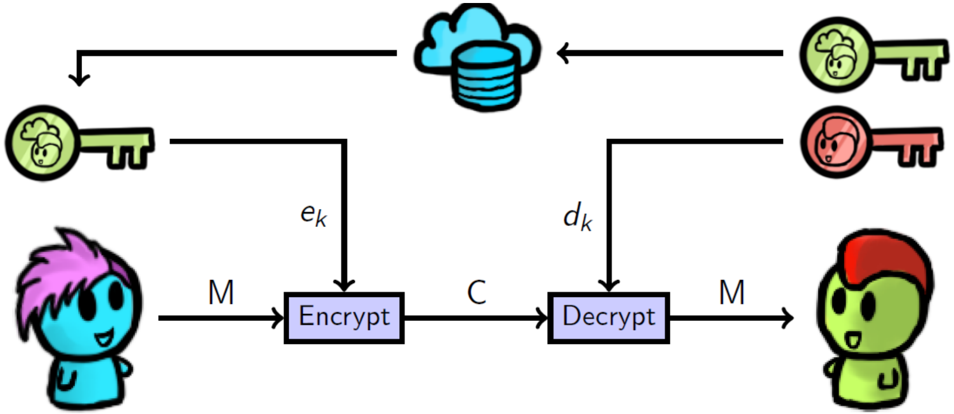
…wait…is this the message Alice sent?

Checksums



Integrity components

How do we tell if a message has changed in transit?



…wait…is this the message Alice sent?

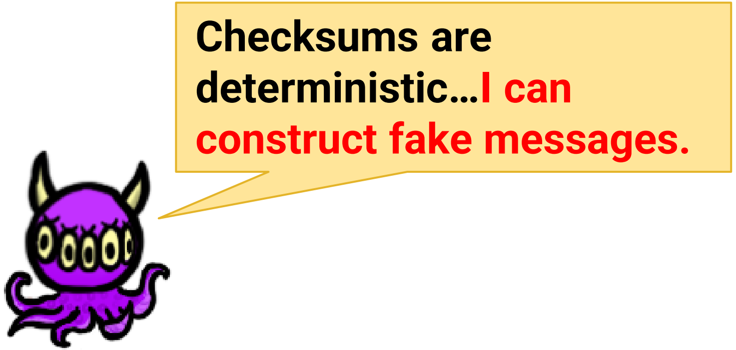
Checksums

Add up all the bytes of M, append the checksum to M so Bob can verify it

Not. Good. Enough.

Checksums are deterministic…

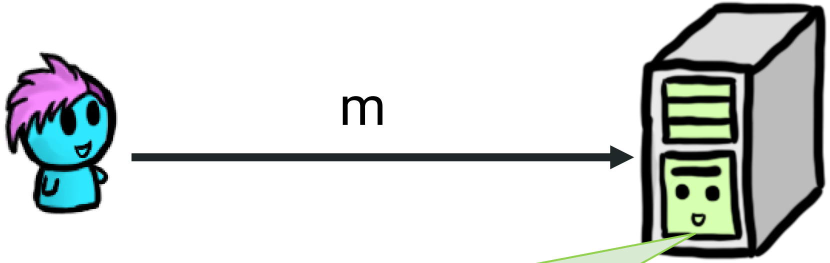
Not. Good. Enough.



Goal: Make it hard for Mallory to find a second message with the same checksum as the "real" message

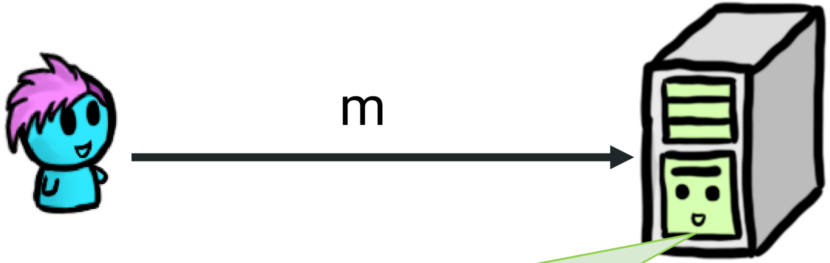
"Cryptographic" checksum

Cryptographic hash functions



I execute a hash function

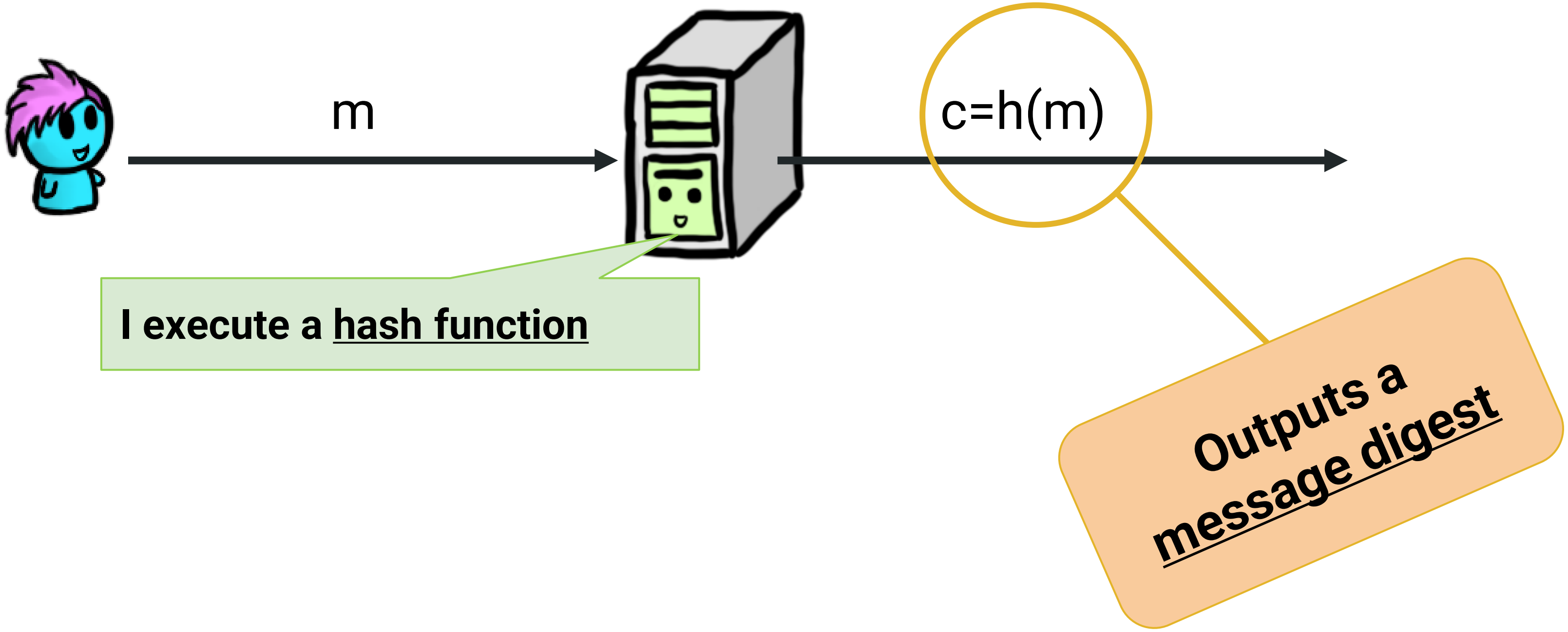
Cryptographic hash functions



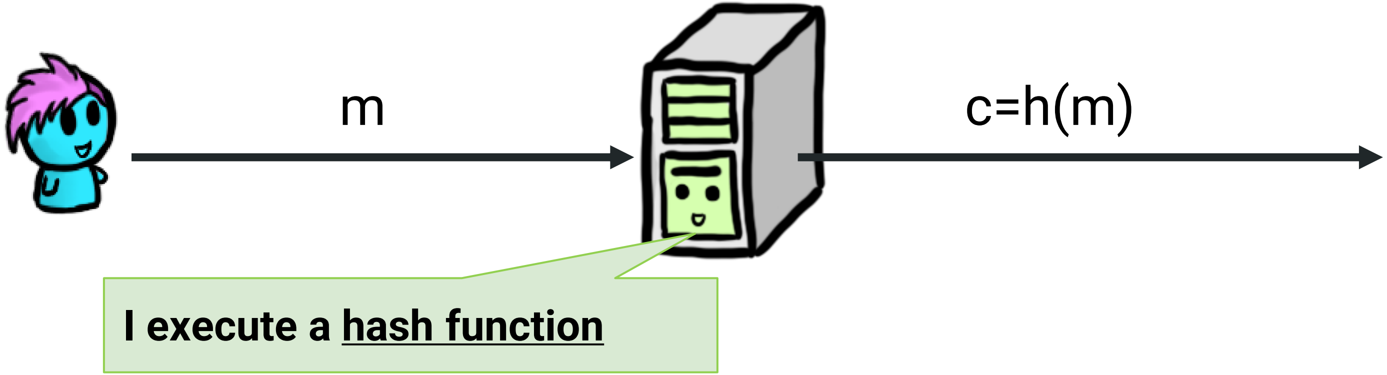
I execute a hash function

Takes an arbitrary length string, and computes a fixed length string.

Cryptographic hash functions



Cryptographic hash functions

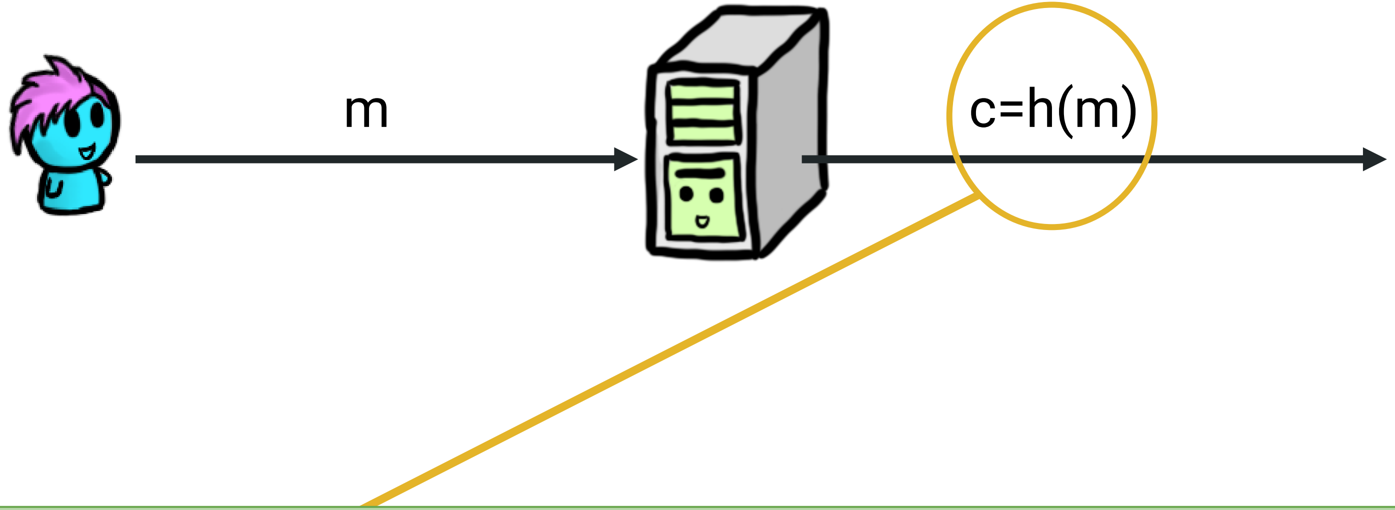


Q: Why is this useful?

Common examples:

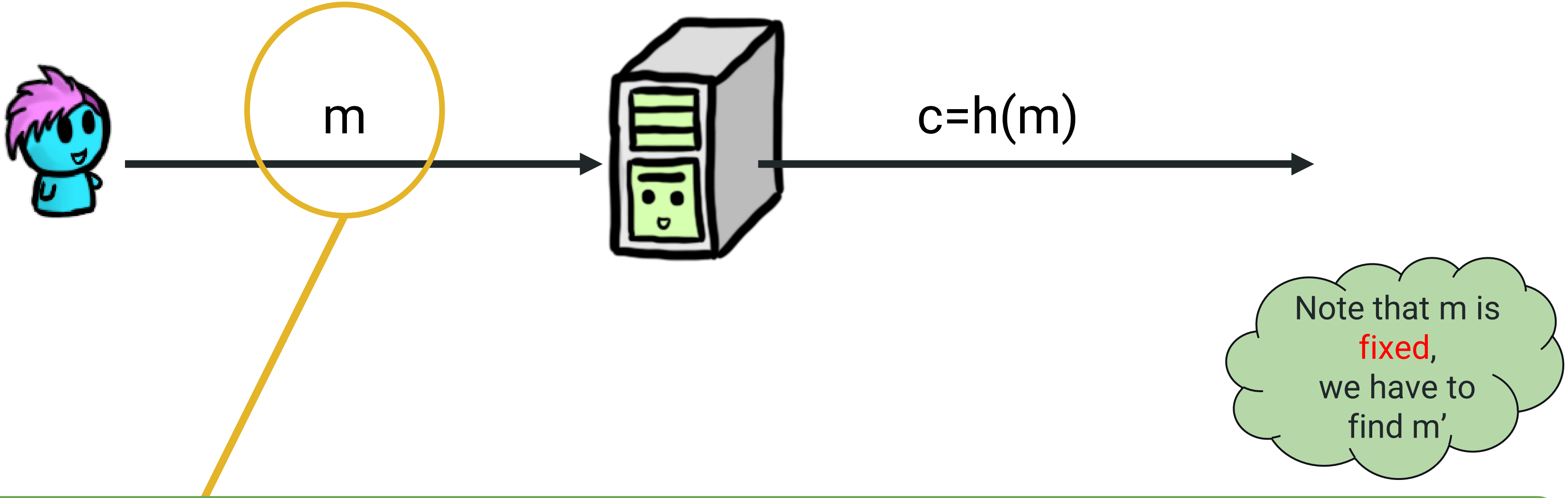
- MD5, SHA-1, SHA-2, SHA-3 (aka Keccak after 2012)

Properties: Preimage-Resistance



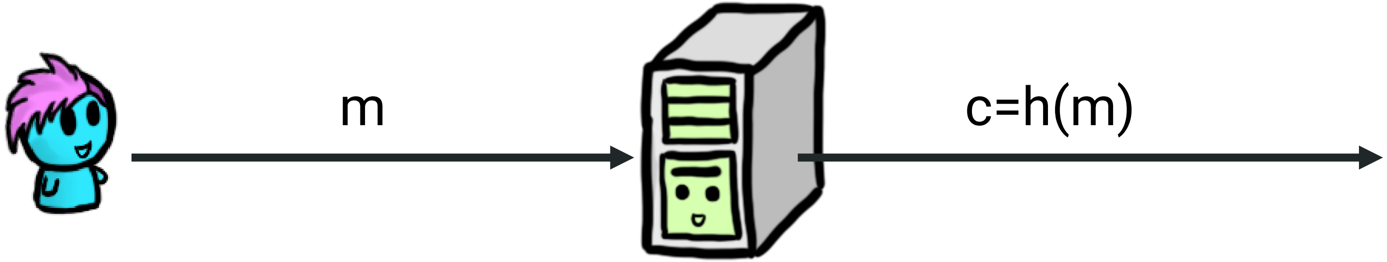
Goal: Given c, it’s “hard” to find m such that h(m) = c
(i.e., a “preimage” of h(m))

Properties: Second Preimage-Resistance



Goal: Given m, it’s “hard” to find m’ ≠ m such that h(m) = h(m’)
(i.e., a “second preimage” of h(m))

Properties: Collision-Resistance



Note that we have **free** choice of m and m'

Goal: It's hard to find any two distinct m, m' such that h(m) = h(m')
(i.e., a "collision")

What do we mean by “hard”?

- SHA-1: takes 2^{160} work to find a preimage or second image

- SHA-1: takes 2^{80} to find a collision using brute-force search
  - For a hash function with an n-bit output, the birthday attack can find collisions in approximately $2^{n/2}$ computations. (2^{80} evaluations)
  - However, there are faster ways than brute-force to find collisions in SHA-1 or MD5

MD5 2008 collision attack against digital certificates.

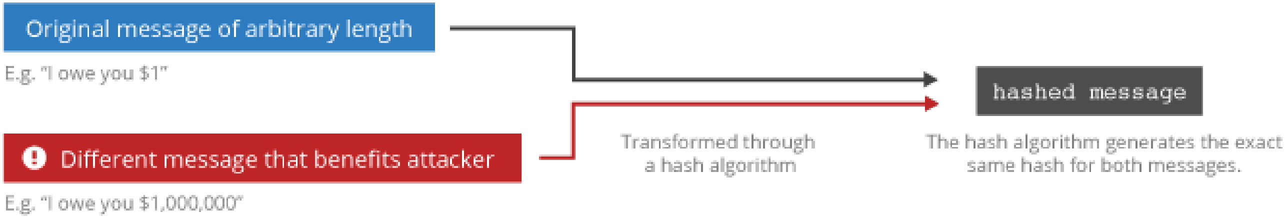
Create 2 cert. with same MD5 hash

SHA-1 2017 collision attack against digital certificates.

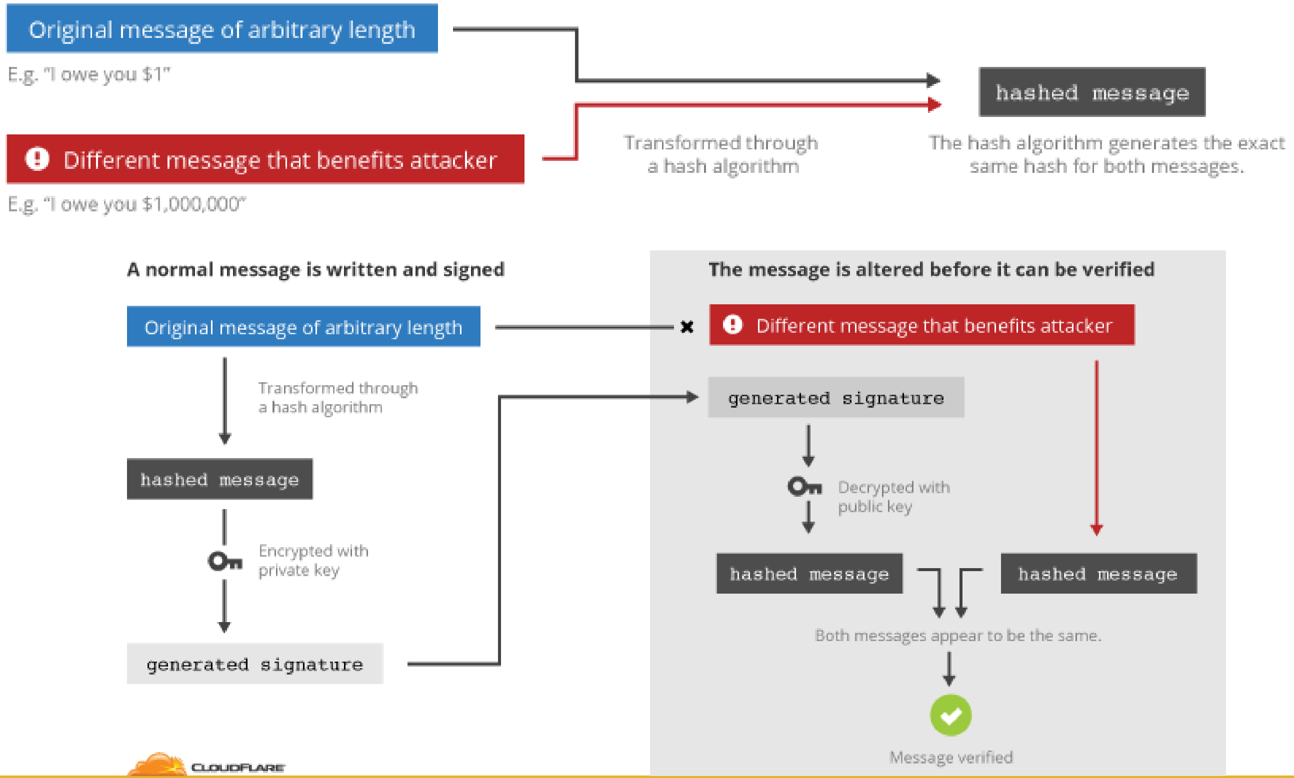
Making it too hard to break these properties?

- SHA-1: takes 2^{160} work to find a preimage or second image
- SHA-1: takes 2^{80} to find a collision using brute-force search
  - However, there are faster ways than brute-force to find collisions in SHA-1 or MD5
- Collisions are always easier to find than preimages or second preimages due to the birthday paradox

How collisions work



How attackers exploit hash collisions



The birthday paradox

- If there are **n** people in a room, what is the probability that at least two people have the same birthday?
- For n = 2: Pr(2) = 1 – $\frac{364}{365}$
- For n = 3: Pr(3) = 1 – $\frac{364}{365}$ x $\frac{363}{365}$
- For n people: Pr(n) = 1 – $\frac{364}{365}$ x $\frac{363}{365}$ x … x $\frac{365-n-1}{365}$

Collisions and the Birthday Paradox

Collisions are easier due to the birthday paradox

What's the probability two of us have the same birthday?

Collisions and the Birthday Paradox

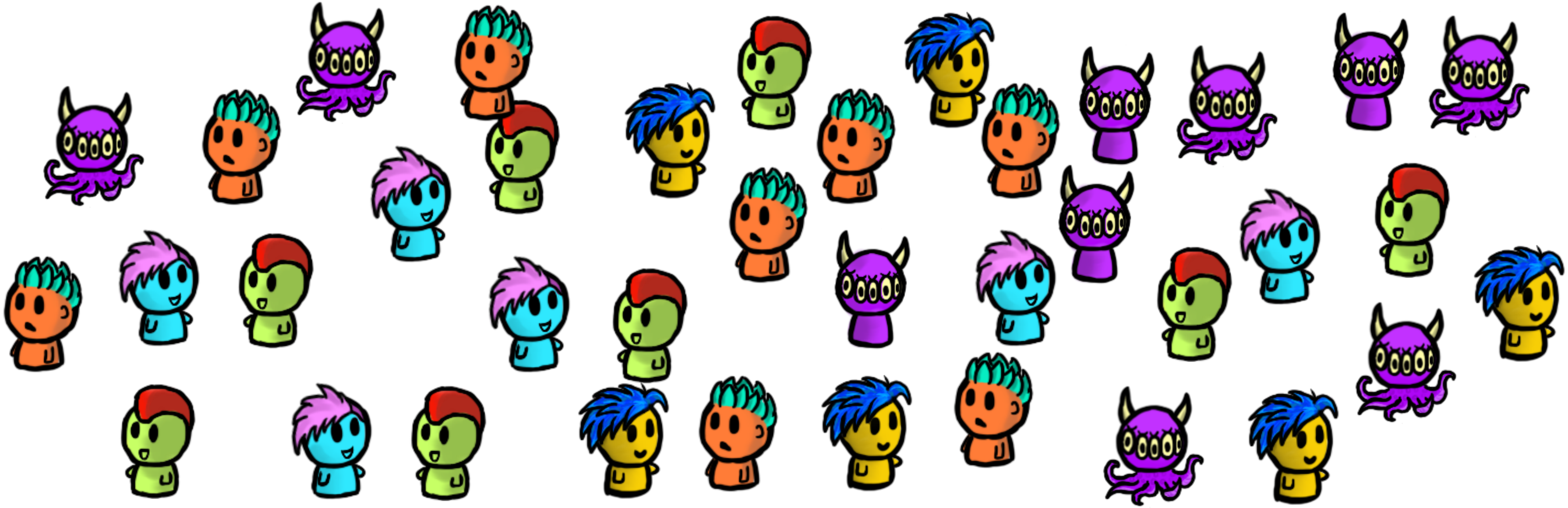
Collisions are easier due to the birthday paradox

What's the probability two of us have the same birthday?

There's 23 of us, so larger than 50%!!

Collisions and the Birthday Paradox

Collisions are easier due to the birthday paradox



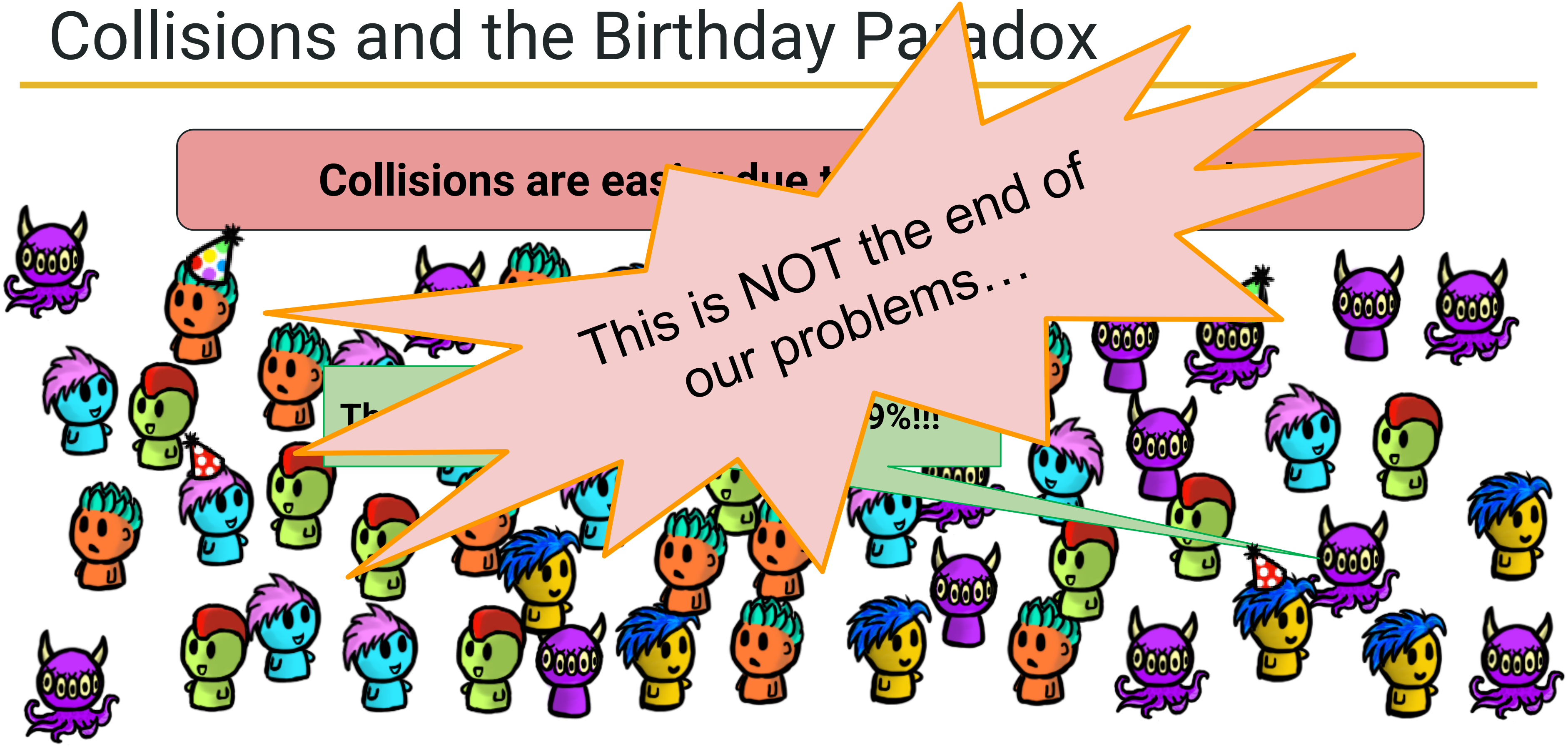
Collisions and the Birthday Paradox

Collisions are easier due to the birthday paradox

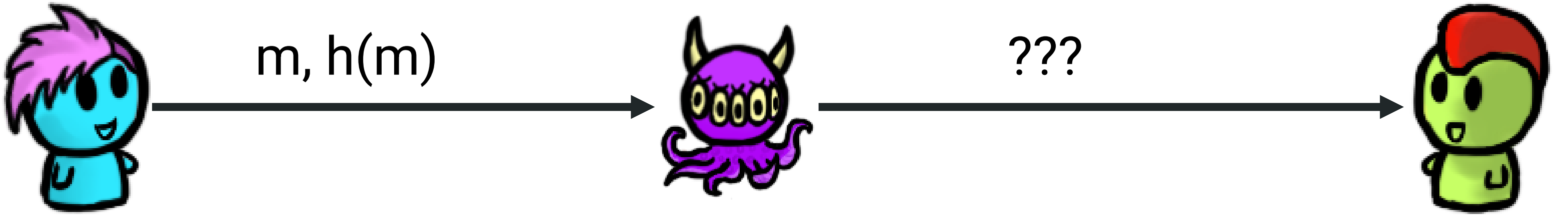
Collisions and the Birthday Paradox

Collisions are easier due to the birthday paradox

Collisions and the Birthday Paradox

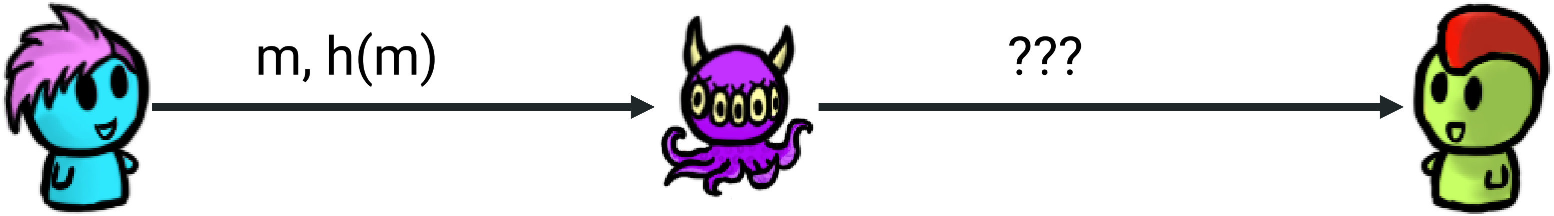


How about a bad example?



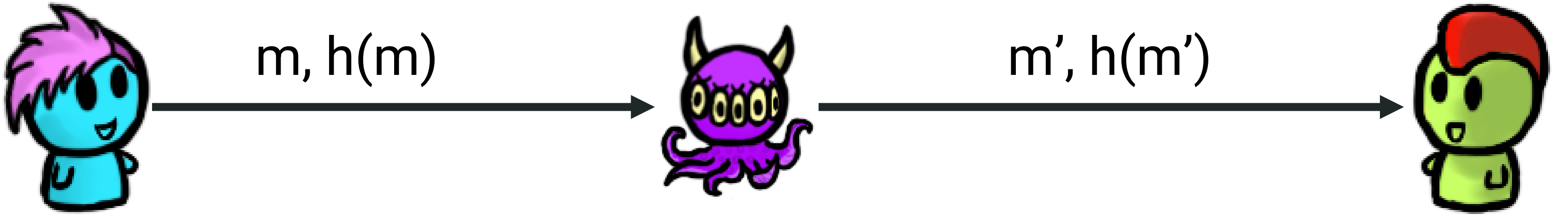
Q: What can Mallory do to send the message she wants (change m)?

How about a bad example?



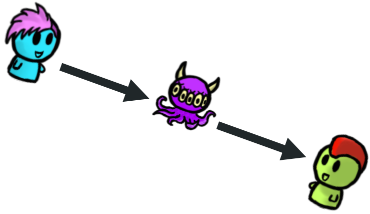
Q: What can Mallory do to send the message she wants (change m)?

A: Just change it…Mallory can compute the new hash herself.



Limitations for Cryptographic Hash Functions

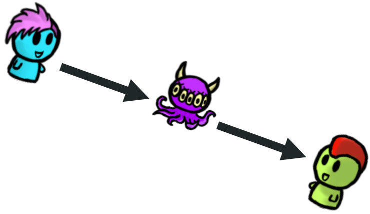
- Integrity guarantees only when there is a <u>**secure**</u> way of sending/storing the message digest



I could publish the hash of my public key on a business card

Limitations for Cryptographic Hash Functions

- Integrity guarantees only when there is a **secure** way of sending/storing the message digest

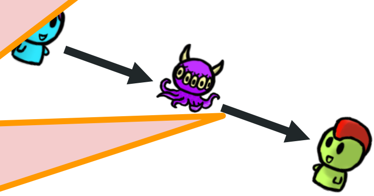


Good idea! Although the key would be too big to place on the card, I could use the hash to... verify it!

I could publish the hash of my public key on a business card

Limitations for Cryptographic Hash Functions

- Integrity guarantees only whe… way of sending/sto… the…



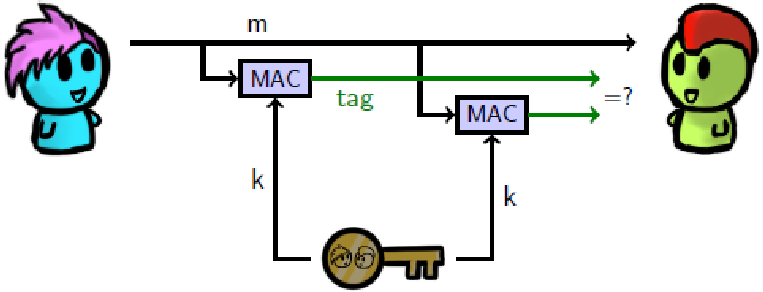
What if…we don’t have an external/physical channel? i.e., using the Internet to communicate

I could publish the hash of my public key on business card

…the key would … card, I … e the hash to… verify it!

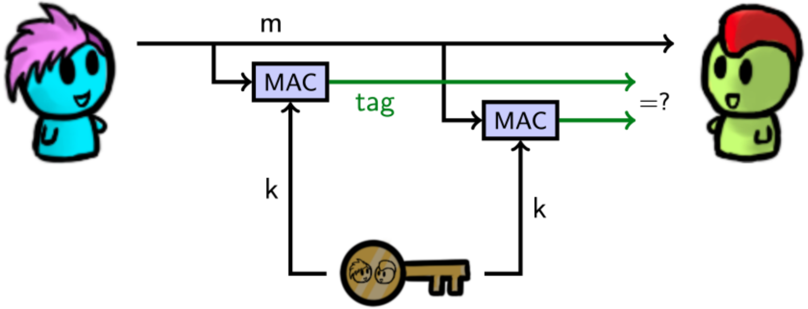
Authentication and Hash Functions

- We can use "keyed hash functions"
- Requires a secrete key to generate, or even check, the computed hash value (sometimes called a tag)



Called: Message authentication codes (MACs)

Message Authentication Codes (MACs)



I don't have the key to generate or check the values…

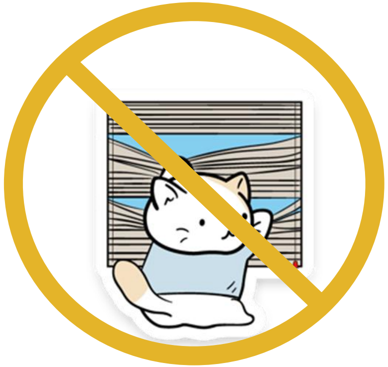
Do the MAC/tag values match?

YES	NO
No one messed with the data	The data has been altered somehow

Common examples:

- SHA-1-HMAC, SHA-256-HMAC, CBC-MAC

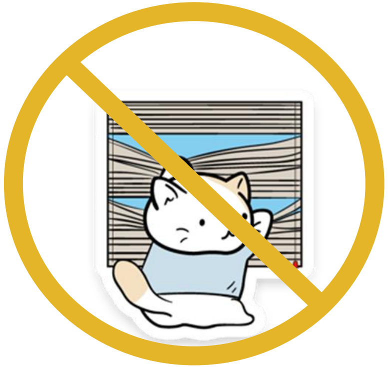
Combine Ciphers and MACs



Confidentiality

Integrity

Combine Ciphers and MACs



Confidentiality

Integrity

In practical we often need both confidentiality and message integrity

But how to combine them? Three possibilities

There are multiple strategies to combine a cipher and a MAC when processing a message

MAC-then-Encrypt, Encrypt-and-MAC, Encrypt-then-MAC

But how to combine them? Three possibilities

There are multiple strategies to combine a cipher and a MAC when processing a message

MAC-then-Encrypt, Encrypt-and-MAC, Encrypt-then-MAC

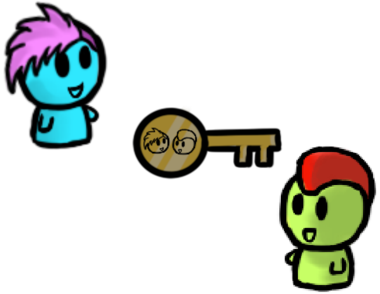
Ideally crypto libraries already provides an authenticated encryption mode that securely combines the two operations, so we don't have to worry about getting it right

- E.g., GCM, CCM (used in WPA2, see later), or OCB mode

Let’s try it!

- Alice and Bob have a secret key **K** for symmetric encryption ($E_k(\cdot)$, $D_k(\cdot)$)

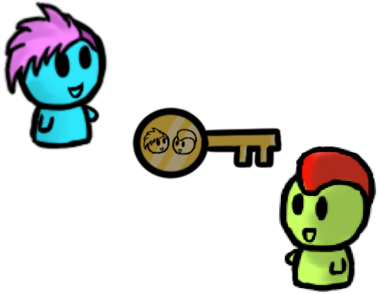
- Also, a secret key **K’** for their $MAC_{K'}(\cdot)$

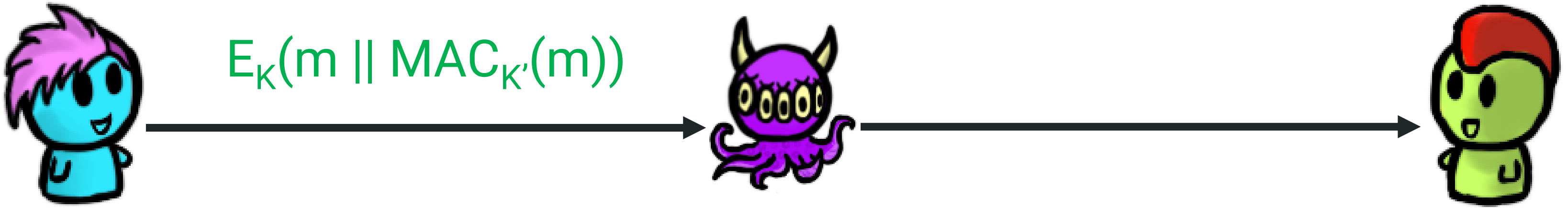


How can Alice build a message for Bob in the following three scenarios?

MAC-then-Encrypt

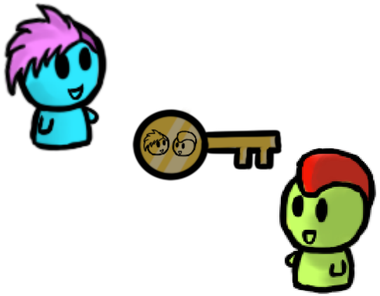
- Compute the MAC on the message, then encrypt the message and MAC together, and send that ciphertext.

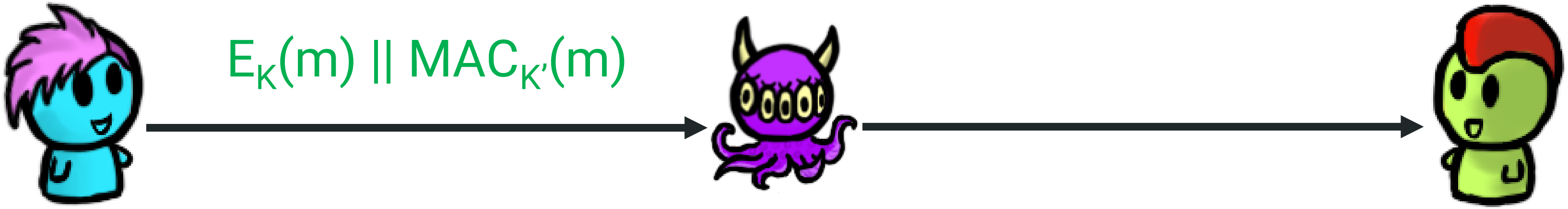




Encrypt-and-MAC

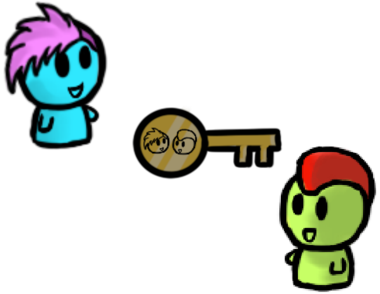
- Compute the MAC on the message, the encryption of the message, and send both.

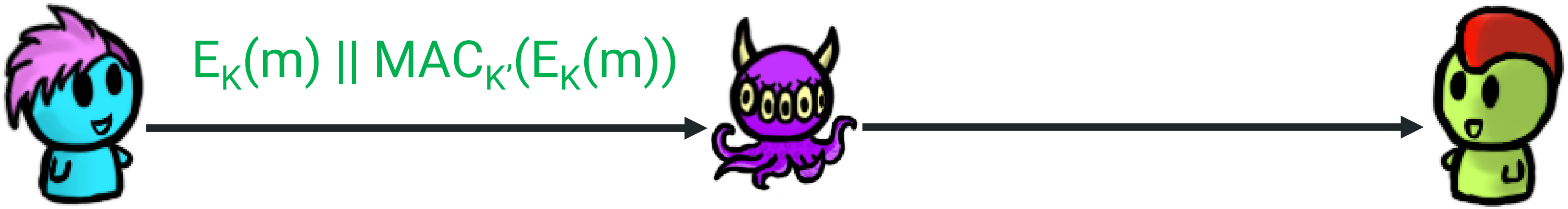




Encrypt-then-MAC

- Encrypt the message, compute the MAC on the encryption, send encrypted message and MAC





Which order is correct?

Q: Which should be recommended then?

$E_K(m \,||\, MAC_{K'}(m))$ **vs.** $E_K(m) \,||\, MAC_{K'}(m)$ **vs.** $E_K(m) \,||\, MAC_{K'}(E_K(m))$

MAC-then-encrypt **Encrypt-and-MAC** **Encrypt-then-MAC**

The Doom Principle

"if you have to perform any cryptographic operation before verifying the MAC on a message you've received, it will somehow inevitably lead to doom."

The Doom Principle

“if you have to perform any cryptographic operation before verifying the MAC on a message you’ve received, it will somehow inevitably lead to doom.”

Q: What are possible problems that can arise from the orderings?

The Doom Principle

Q: What are possible problems that can arise from the orderings?

- **MAC-then-Encrypt:** Allows an adversary to force Bob into decrypting the ciphertext before verifying the MAC. May lead to a padding oracle attack

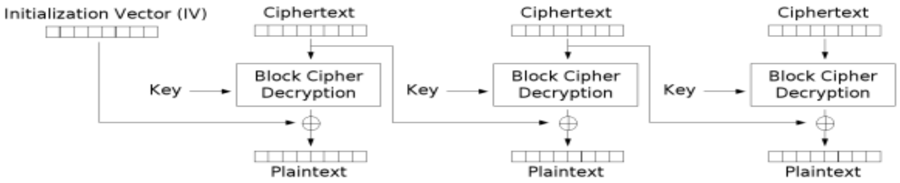
The Doom of MAC-then-Encrypt

Observation: To verify the MAC, Bob first has to decrypt the message, since the MAC is part of the encrypted payload

- **Padding oracle attack:** The idea is for the attacker to send modified ciphertexts to Bob and observe how he responds.
- With CBC, by modifying the last block of the ciphertext in a way that alters the block's padding, the attacker can tell if the padding is valid or not.
- If the padding is invalid, the system might respond differently (e.g., with an error message that is padding-specific). This information leakage allows the attacker to gradually decrypt the ciphertext byte by byte.

The Doom of MAC-then-Encrypt

- **Padding oracle attack:**
  - If a block needs to be padded out by 5 bytes, for instance, Alice appends 5 bytes each with value 0x05 before encryption
  - Mallory tampers with the last byte of the second-to-last ciphertext block
  - Bob decrypts the ciphertext, looks at the value of the last byte (call it N), and ensures that the preceding N-1 bytes also have the value of N.
  - If Bob encounters an incorrect padding → Abort and return padding error to Alice (visible to Mallory).
  - Otherwise, Mallory will not see a padding error and infers that the last byte of the decrypted plaintext is (likely) 0x01, allowing Mallory to compute the last byte of the original plaintext. Repeat for remaining bytes.



Cipher Block Chaining (CBC) mode decryption

The Doom Principle

Q: What are possible problems that can arise from the orderings?

- **Encrypt-and-MAC:** Allows an adversary to force Bob into decrypting the ciphertext to verify the MAC. May lead to a chosen-ciphertext attack

The Doom of Encrypt-and-MAC

Q: What happens if the MAC has no mechanism to provide confidentiality?

- MACs are meant to provide integrity
- MACs are often implemented by a **deterministic** algorithm without an explicit random input (essentially, for a given key and message, the output of the MAC is always the same).
- If a deterministic MAC is used, then there is no guarantee that the tag $E_K(m)$ || $\mathbf{MAC_{K'}(m)}$ will not leak information about the secret message **m**.

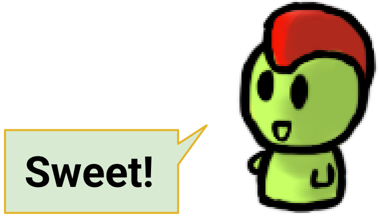
Which order is correct?

We want the receiver to verify the MAC first!

The recommended strategy is Encrypt-then-MAC:

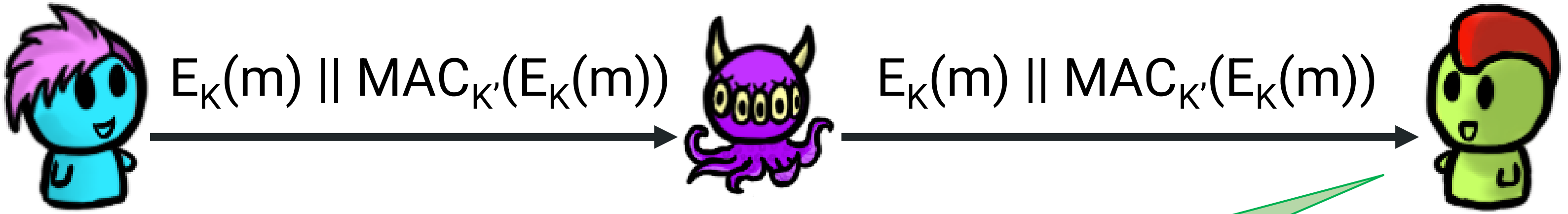
$E_K(m) \,||\, MAC_{K'}(E_K(m))$

- **Encrypt-then-MAC:** Allows Bob to check the MAC of the ciphertext before performing any decryption whatsoever (e.g., prevent attacks by immediately closing a connection if the MAC fails)



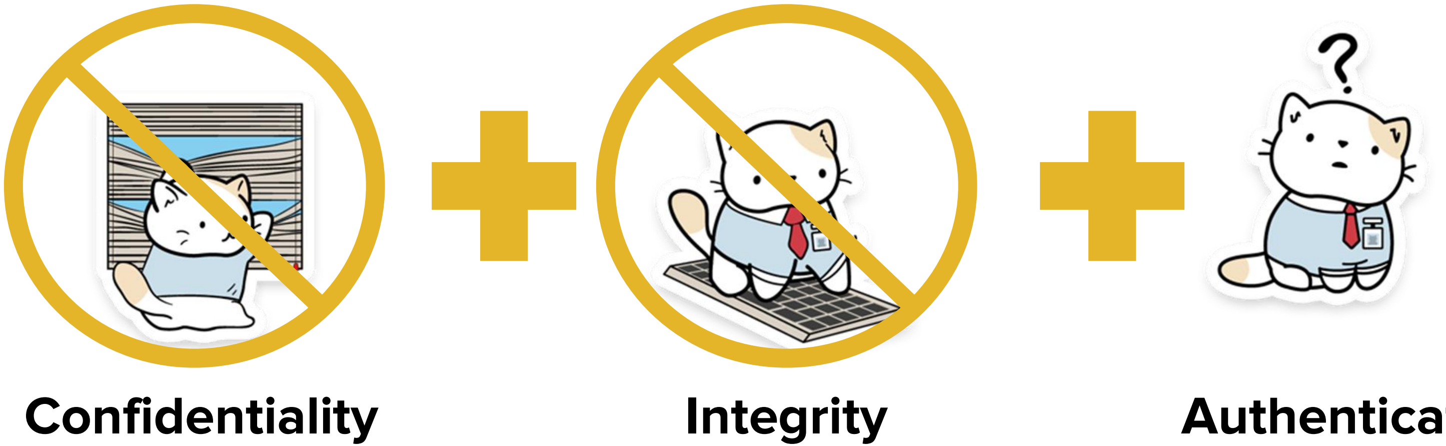
More properties that matter

Repudiation



Alice sent *m*, and I received the same m she sent.

Repudiation

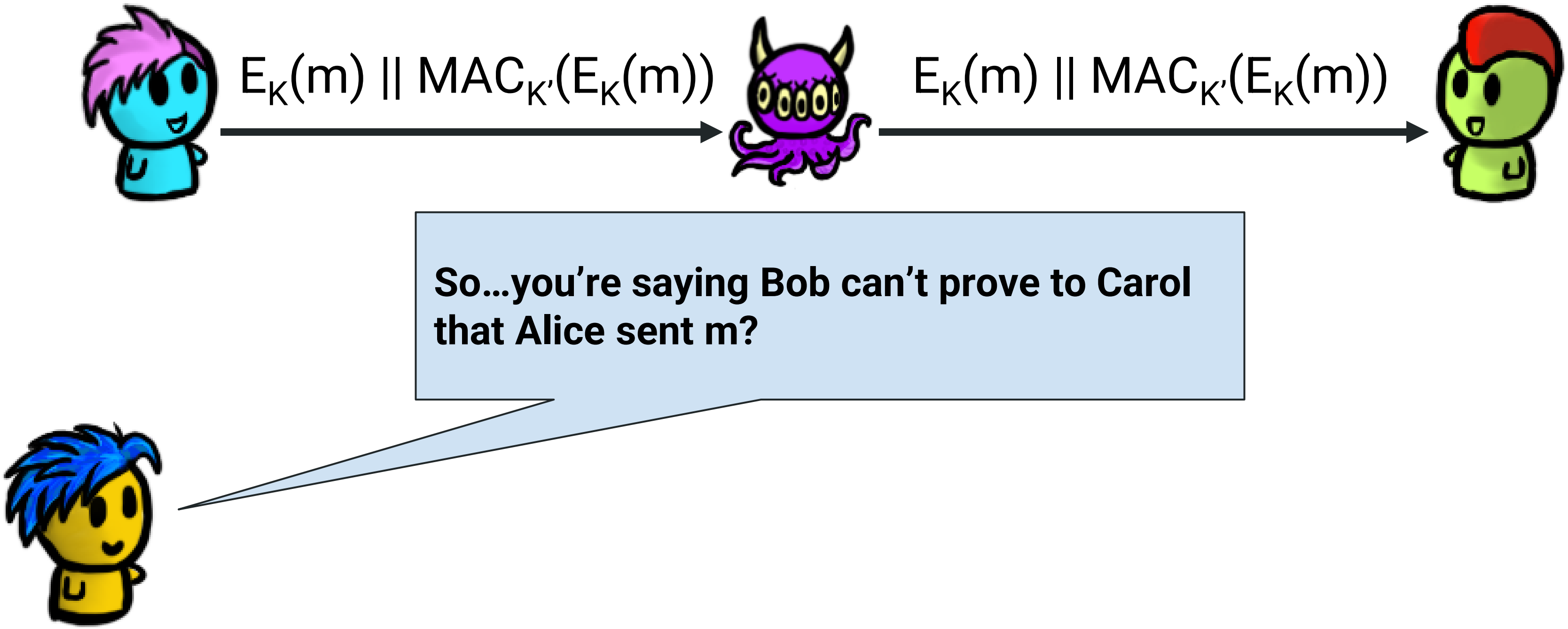


Repudiation

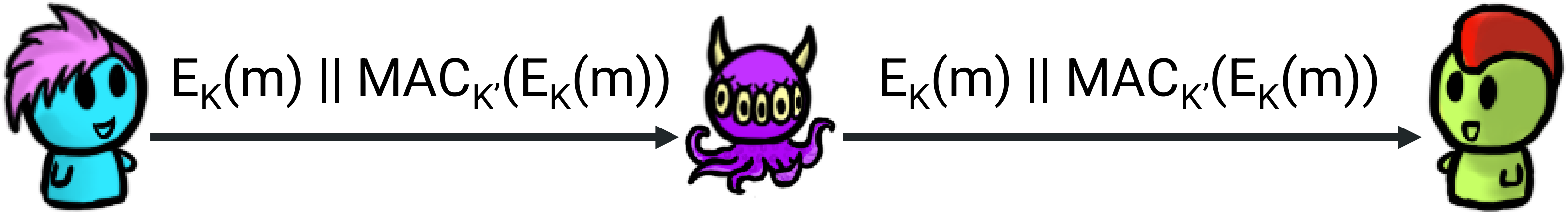
Almost, but not quite a "signature"

Confidentiality + **Integrity** + **Authentication**

Repudiation



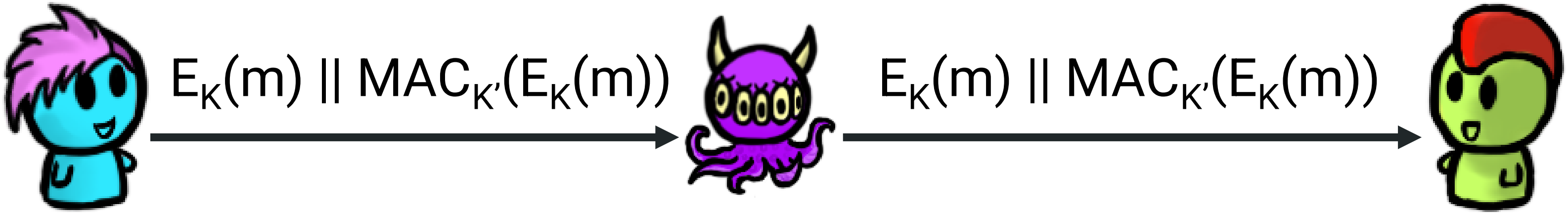
Repudiation



So…you're saying Bob can't prove to Carol that Alice sent m?

Q: Why can't Bob prove it?

Repudiation

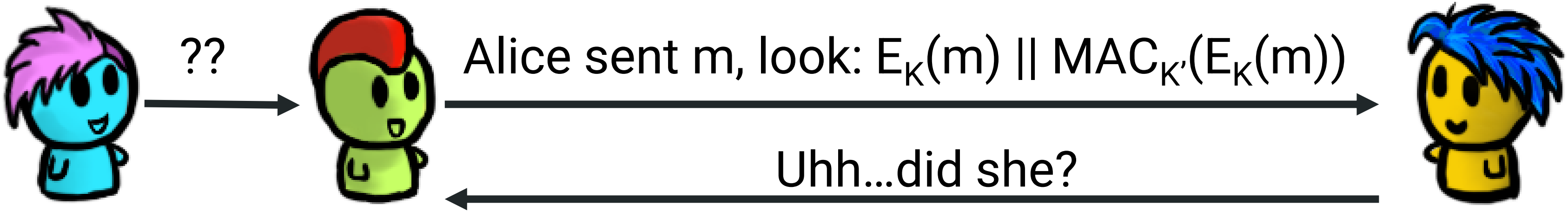


So…you're saying Bob can't prove to Carol that Alice sent m?

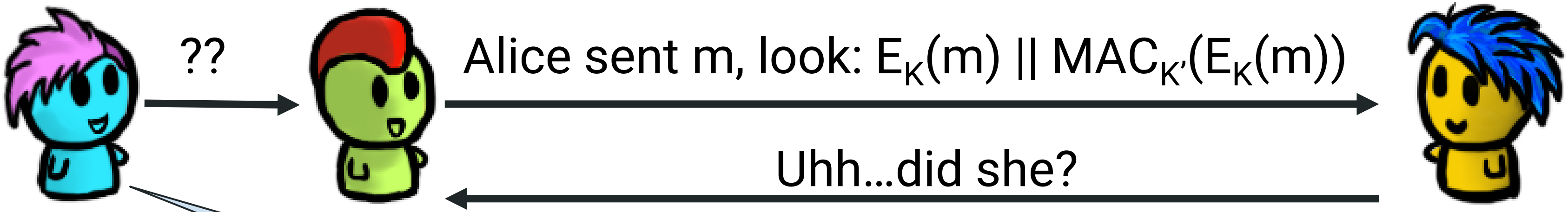
Q: Why can't Bob prove it?

A: Either Alice or Bob could create any message and MAC combination…also Carol doesn't know the secret key.

Implications?



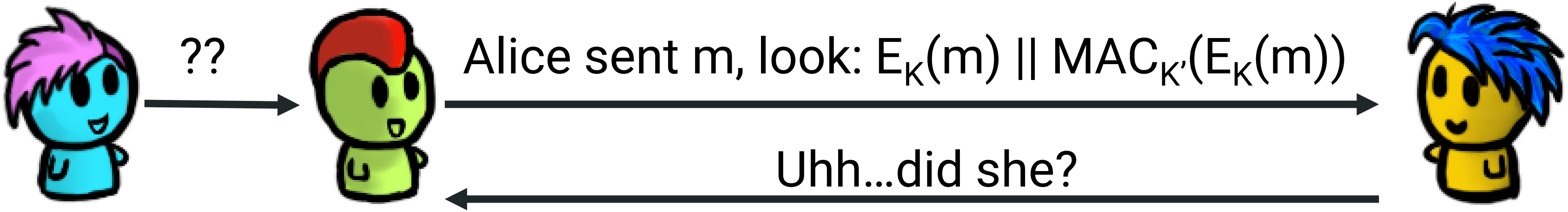
Implications?



Nope! Bob made everything up!
Both the message and the MAC

Bob be like

Implications?

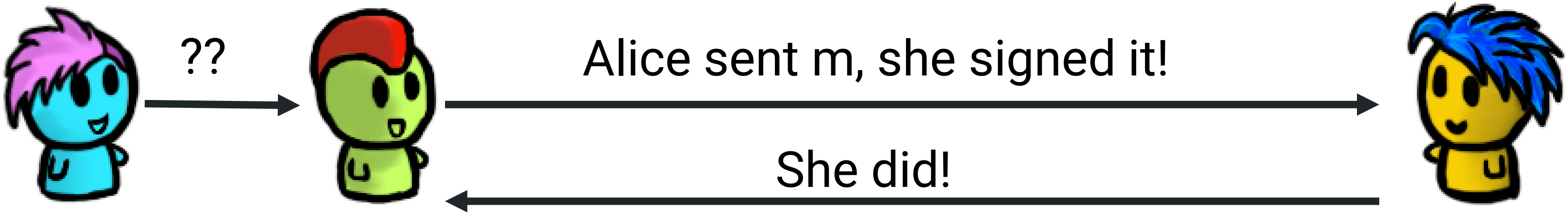


This is called repudiation, and we sometimes want to avoid it

Repudiation Property: For some applications this property is good (e.g., private conversations)…others less good (e.g., e-commerce…).

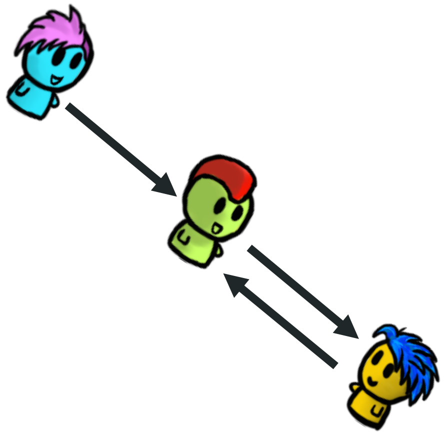
Digital Signatures - For When Repudiation is Bad

For non-repudiation, what we want is a true digital signature, with the following properties:



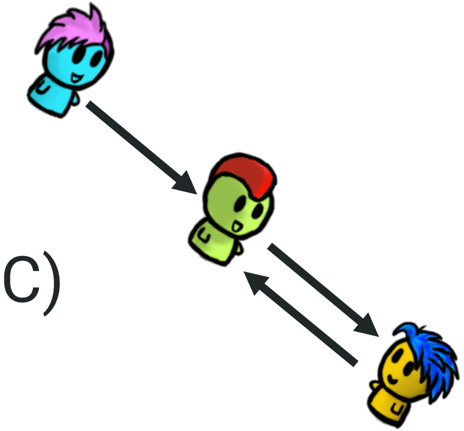
Properties of digital signatures

If Bob receives a message with Alice's digital signature on it, then:



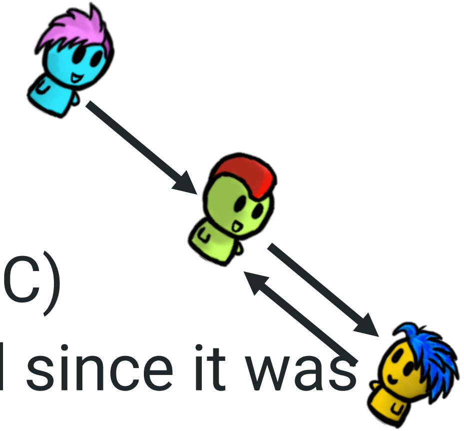
Properties of digital signatures

If Bob receives a message with Alice's digital signature on it, then:



- Bob knows Alice sent it, and not  (like a MAC)

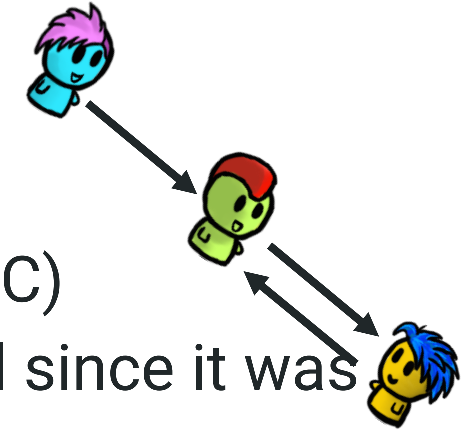
Properties of digital signatures



If Bob receives a message with Alice’s digital signature on it, then:

- Bob knows Alice sent it, and not  (like a MAC)
- Bob knows the message has not been altered since it was sent (like a MAC)

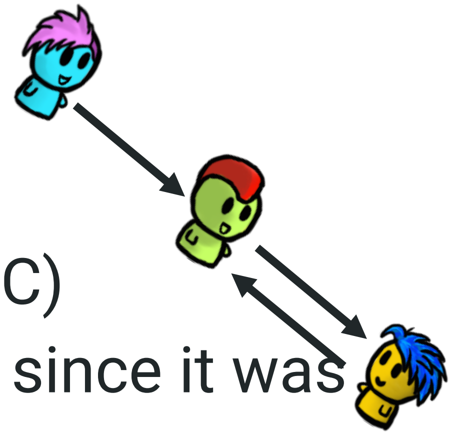
Properties of digital signatures



If Bob receives a message with Alice’s digital signature on it, then:

- Bob knows Alice sent it, and not  (like a MAC)
- Bob knows the message has not been altered since it was sent (like a MAC)
- Bob can prove these properties to a third party (NOT like a MAC)

Properties of digital signatures

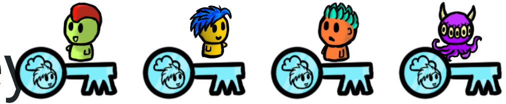


If Bob receives a message with Alice's digital signature on it, then:

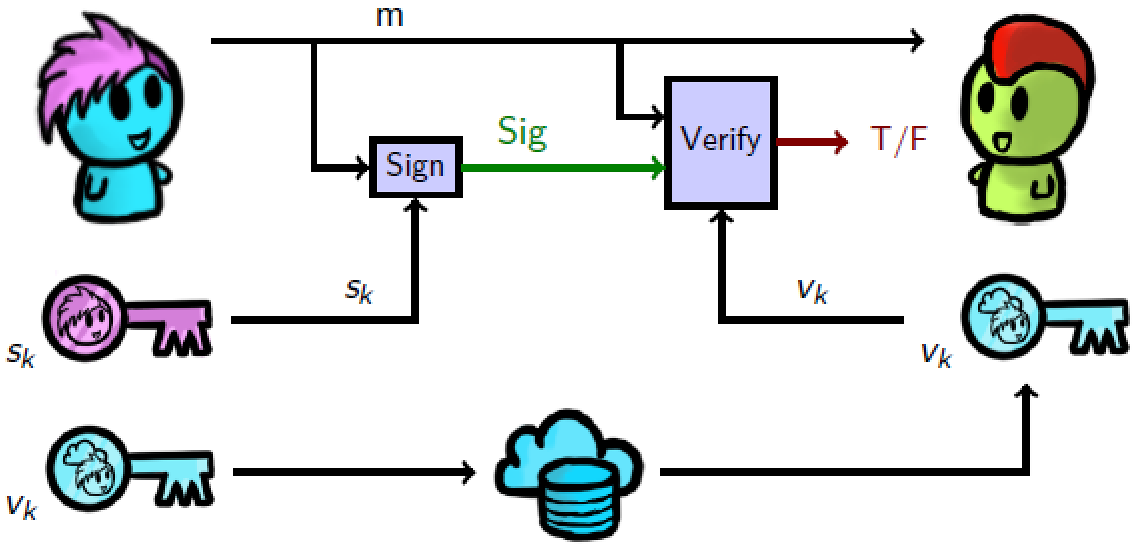
- Bob knows Alice sent it, and not , (like a MAC)
- Bob knows the message has not been altered since it was sent (like a MAC)
- Bob can prove these properties to a third party (NOT like a MAC)

Achievable? Use techniques similar to public-key crypto (last class)

Making Digital Signatures

1. A pair of keys 
2. Everyone gets Alice's public verification key 
3. Alice signs m with her private signature key S_k 
4. Bob verifies m with Alice's public verification key V_k 
5. If it verifies correctly, the signature is valid

Digital Signatures at a Glance

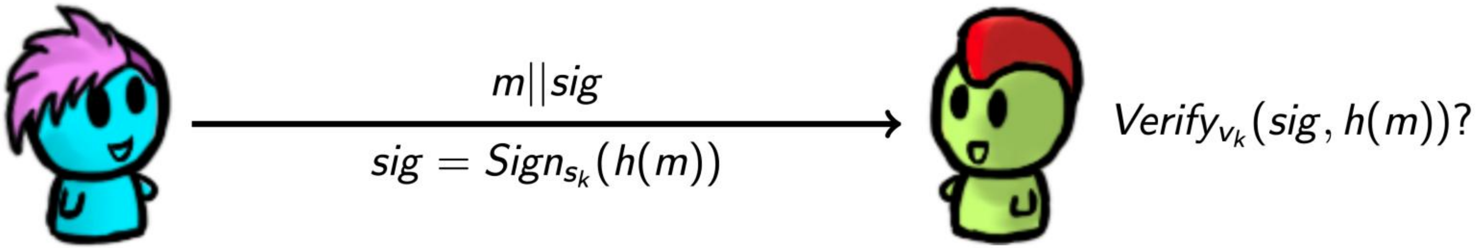


Faster Signatures

- Signing large messages is slow
  → "hybridize" the signatures to make them faster
- A hash is much smaller than the message… faster to sign

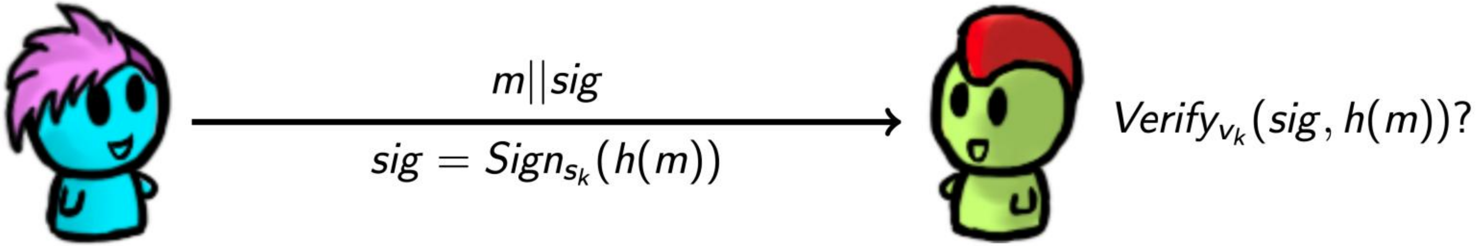
Faster Signatures - aka More Hybrids

- Signing large messages is slow
  → “hybridize” the signatures to make them faster
- A hash is much smaller than the message… faster to sign



Faster Signatures - aka More Hybrids

- Signing large messages is slow
  → "hybridize" the signatures to make them faster
- A hash is much smaller than the message… faster to sign



- Finally, authenticity and confidentiality are separate
  → you need to include both if you want to achieve both

Combining PKE and digital signatures

- Alice has two different key pairs:
  - → an (encryption, decryption) key pair e_k^A , d_k^A
  - → a (signature, verification) key pair s_k^A , v_k^A
- So does Bob : e_k^B , d_k^B and s_k^B , v_k^B
- Alice uses e_k^B to encrypt a message destined for Bob:
  - → $C = E_{e_k^B}(M)$
- She uses s_k^A to sign the ciphertext:
  - → $T = Sign_{s_k^A}(C)$
- Bob uses v_k^A to check the signature:
  - → $Verify_{v_k^A}(C,T)$, if verified, C is authentic
- He uses d_k^B to decrypt the ciphertext:
  - → $M = D_{d_k^B}(C)$

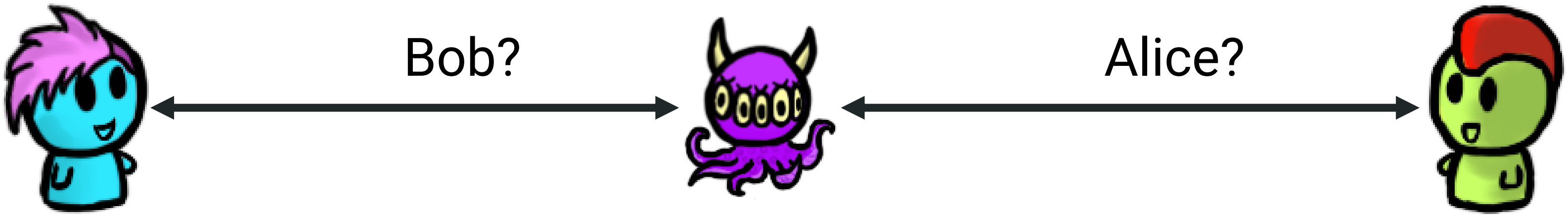
Relationship between key pairs

- Alice’s (signature, verification) key pair is long-lived, whereas her (encryption, decryption) key pair is short-lived

    → Provides forward secrecy

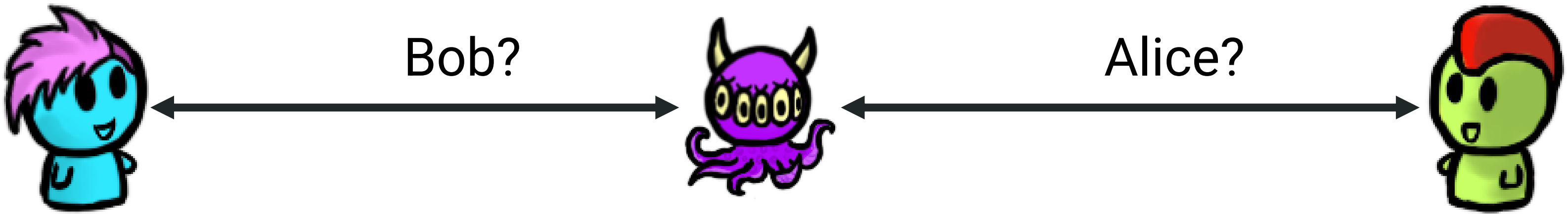
- When creating a new (encryption, decryption) key pair, Alice uses her signing key to sign her new encryption key and Bob uses Alice’s verification key to verify the signature on this new key

The Key Management Problem



Q: How can Alice and Bob be sure they’re talking to each other?

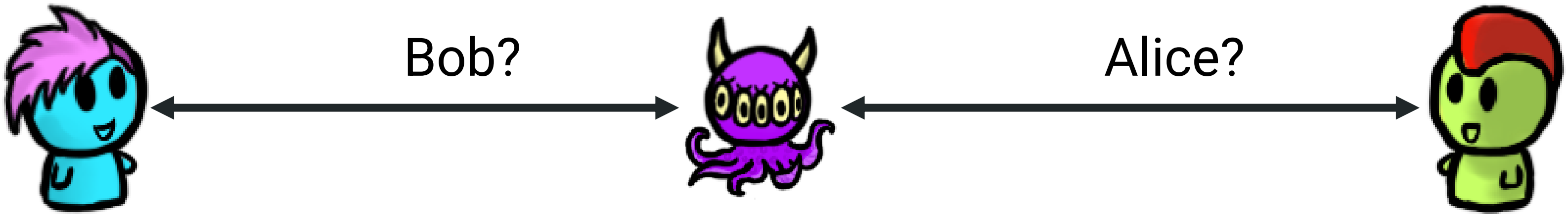
The Key Management Problem



Q: How can Alice and Bob be sure they’re talking to each other?

A: By having each other’s verification key!

The Key Management Problem

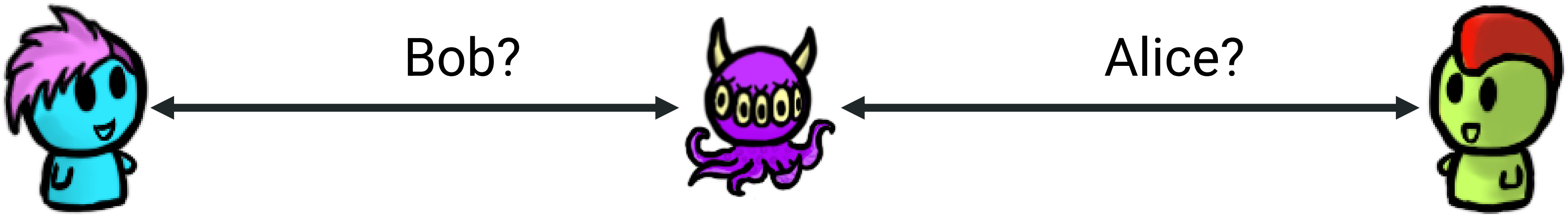


Q: How can Alice and Bob be sure they're talking to each other?

A: By having each other's verification key!

Q: But how do they get these keys?

The Key Management Problem…Solutions?



Q: But how do they get these keys?

A: Know it personally (manual keying e.g., SSH)

A: Trust a friend (web of trust e.g., PGP)

A: Trust some third party to tell them (CAs, e.g., TLS/SSL)

Next up: More Cryptography…