Introduction to Crypto

Jacob Stolker
University of Illinois
CS 461

MP3 - Cryptography

- Checkpoint 1 (20 points)
- Checkpoint 2 (80 points)
- Important notes:
  - Almost no partial credit, either it works or it does not.
  - We recommend to always test your solutions.
  - Run on Python 3.10 or below if outside VM (VM should be 3.6 and should have no problems)

Secure Communication Puzzle

Secure Communication Puzzle

- *Riccardo has found a dangerous vulnerability on his CPU and wants to responsibly disclose it to Intel by sending them a letter. However, he knows from experience that any time he sends or receives something that is not in a locked box, the postal service steals it. Luckily, Riccardo has a box, a padlock and a key that he can use to lock his letter in a box. Intel also has their own box, padlock and key. How can Riccardo arrange, with these resources, to communicate his finding securely to Intel?*

Goals of MP3

- Become familiar with existing cryptographic libraries and how to utilize them.
- Understand pitfalls in cryptography and appreciate why you should not write your own cryptographic libraries.
- Execute a classic cryptographic attack on MD5 and other broken cryptographic algorithms.
- Execute a length-extension attack similar to a historical attack executed successfully against Flickr.

3.1.1 – I/O & Mechanics

- Don’t go online and convert it there
  - Write a python I/O Script

```
import sys
if __name__ == "__main__":
      if len(sys.argv) != ___:    #number of expected arguments
            exit()
      first_arg = sys.argv[1]     #argv[0] is usually the script name
      …
      # strip() remove any leading or trailing whitespace characters
      with open('file_name') as f:
            file_content = f.read().strip()  # or .readlines()
      …
      with open(usually_arg_3, 'w') as f:
            output_file.write(something)
```

- 8 bits = 1 byte = 2 hex digits = 1 char

3.1.1 – I/O & Mechanics

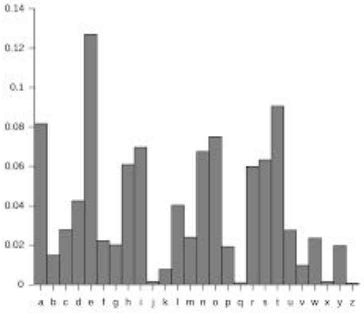
- 8 bits = 1 byte = 2 hex digits = 1 char
- Check the docs for other tips, try some of this in a python environment!

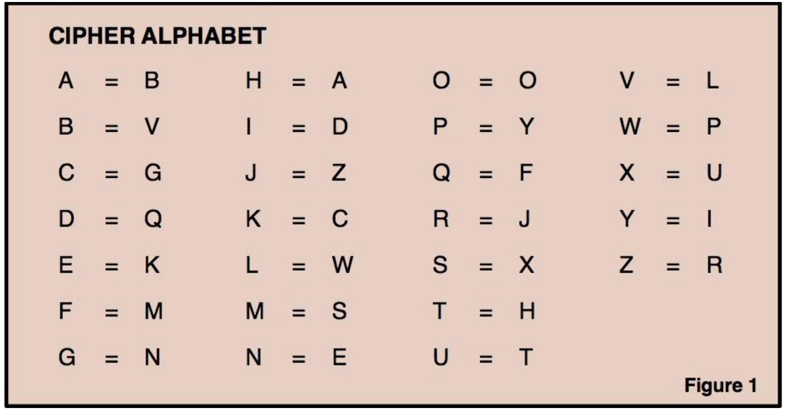
1. `a = "Hello"`
2. `a_byte = a_string.encode() # b'Hello’`
3. `a_hex = a_byte.hex() # 48-65-6c-6c-6f`
4. `a_int = int(a_hex, 16) # 310939249775`
5. `a_bin = bin(a_int) # 0b1001000-01100101-01101100-01101100-01101111`
6. `a_b = a_byte.decode() # “Hello”`
7. `a_byte_b = bytes.fromhex(a_hex) # b'Hello’`
8. `a_int_b = int(a_bin, 2) # 310939249775`
9. `a_hex_b = hex(a_int) # 0x48656c6c6f`

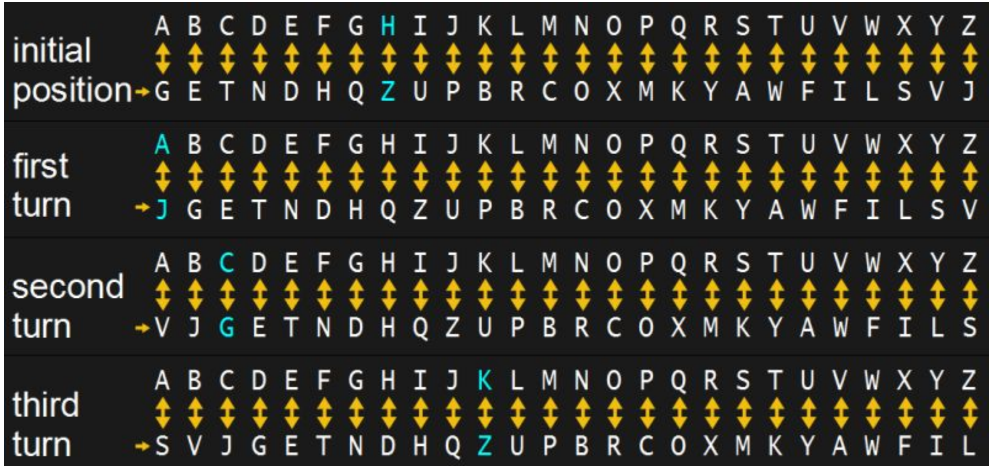
```
decode(somefile.read().strip(), ‘hex’) # from codecs import decode
decode(a_hex, ‘hex’) # this will be b’Hello’
decode(decode(a_hex, ‘hex’)) # ‘Hello’
```

3.1.2 – Substitution Ciphers

- One of the most basic cipher algorithms
- Was popular in the past
  - Enigma Machine
- Considered Broken
  - Statistical Frequency Analysis







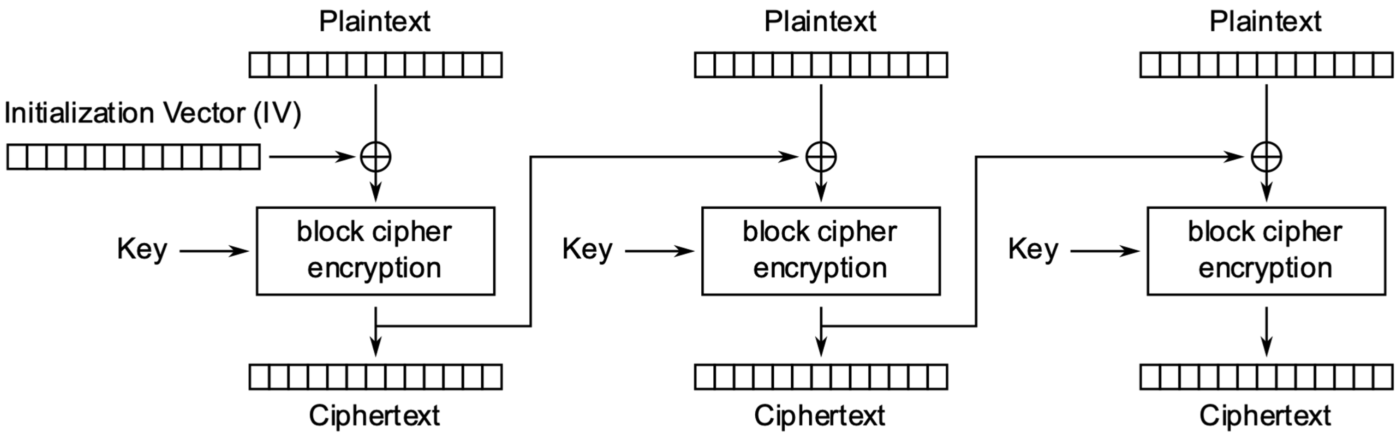
https://hackaday.com/wp-content/uploads/2017/08/rotated_substitution_cipher.png?w=800

3.1.3 – Advanced Encryption Standard (AES)

- AES was a Encryption Competition to find a new algorithm (since DES)
- Symmetric encryption and decryption
- Alice and Bob both know and share some secret key
- Works on fixed sized blocks (128 bit, 16 bytes, 32 hex digits), one at a time
- Many variations
    - Block chaining modes (we'll use CBC for checkpoint 1)
    - 3 key sizes (we'll use 256 bits)

3.1.3 – Advanced Encryption Standard

- 128 bit (16 bytes, 32 hex digits) block size, 3 key sizes (we'll use 256 bits)



Cipher Block Chaining (CBC) mode encryption

3.1.3 – Advanced Encryption Standard

```
from Crypto.Cipher import AES

ciphertext = load_ciphertext_we_gave_you_as_bytes()
key = load_key_we_gave_you_as_bytes()
iv = load_iv_we_gave_you_as_bytes()

cipher = AES.new(key, AES.MODE_CBC, iv=iv)
plaintext = cipher.decrypt(ciphertext)
# ciphertext must be multiple of 16 bytes
```

3.1.4 – Importance of a good key

- We use AES with 256-bit keys, in CBC Mode
- What if you knew that the first 251 bits were all 0s?
- You'd have to guess only the last 5 bits.
- How many guesses will it take?

3.1.4 – Importance of a good key

- We use AES with 256-bit keys, in CBC Mode
- What if you knew that the first 251 bits were all 0s?
- You'd have to guess only the last 5 bits.
- How many guesses will it take?
  - 2^5=32 Bruteforce-able!!

3.1.4 – Importance of a good key

- We use AES with 256-bit keys, in CBC Mode
- What if you knew that the first 251 bits were all 0s?
- You'd have to guess only the last 5 bits.
- How many guesses will it take?
  - 2^5=32 Bruteforce-able!!
- How many guesses would it take if the key was random?

3.1.4 – Importance of a good key

- We use AES with 256-bit keys, in CBC Mode
- What if you knew that the first 251 bits were all 0s?
- You’d have to guess only the last 5 bits.
- How many guesses will it take?
    - 2^5=32 Bruteforce-able!!
- How many guesses would it take if the key was random?
    - $2^{256} = 1.158 \times 10^{77}$
    - There are about ~10^{80} Atoms in the Universe

3.1.5 – RSA Decryption (Rivest–Shamir–Adleman)

- Asymmetric encryption and decryption.
- Each party has 2 keys: private and public key.
- Alice and Bob know the other party's public key.
- The private key is used to decrypt ciphertexts, and never shared.

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

R.L. Rivest, A. Shamir, and L. Adleman*

Original paper from 1977

3.1.5 – RSA Decryption

- e - public prime (commonly 3 or 65537)
- n - public modulus
- d - secret
- m - plaintext
- c - ciphertext

- Encryption: $c = m^e \bmod(n)$
- Decryption: $m = c^d \bmod(n)$

3.1.5 – RSA Decryption

- e - public prime (commonly 3 or 65537)
- n - public modulus
- d - secret
- m - plaintext
- c - ciphertext

Modular exponentiation

- Encryption: $c = m^e \bmod(n)$
- Decryption: $m = c^d \bmod(n)$

3.1.5 – RSA Decryption

- e - public prime (commonly 3 or 65537)
- n - public modulus
- d - secret
- m - plaintext
- c - ciphertext

Modular exponentiation

- Encryption: $c = m^e \bmod(n)$
- Decryption: $m = c^d \bmod(n)$

We'll Discuss how to generate n and d next discussion!

3.1.5 – RSA Decryption

- Sometimes useful to strip the 0x after you convert to hex!!

```
hex(RSA_decryption)[2:]
```

- The pow() function in Python implements fast modular exponentiation

```
pow(base, exp, modulus)
```

3.1.6 – Hash Functions

- Map arbitrarily long input strings to a fixed-length output
- Not all hash functions are cryptographically useful!
- A cryptographic hash should be:
  - Preimage Resistant (One-Way): Given H(a) it's hard to find a
  - Collision Resistant: It's hard to find (a,b) s.t. H(a) == H(b)
  - Second Preimage Resistant: Given a, it's hard to find b s.t. H(a) == H(b)

3.1.6 – A "Weak Hashing Algorithm"

- WHA is not Second Preimage Resistant:
  - Given a, it's not hard to find b: H(a) == H(b).
- Analyze WHA for weaknesses - under what circumstances will you get two strings that hash to the same value?
- Generate a string that hashes to the same value as the one we gave you.

Hint 1: Write out the hex values as binary and observe what the algorithm does

Hint 2: on a piece of paper, write down the expanded outHash value for input string "abc". Do you notice any interesting property?

3.2.1 – MD5 Length Extension

Flickr's API Signature Forgery Vulnerability

Thai Duong and Juliano Rizzo

Date Published: Sep. 28, 2009

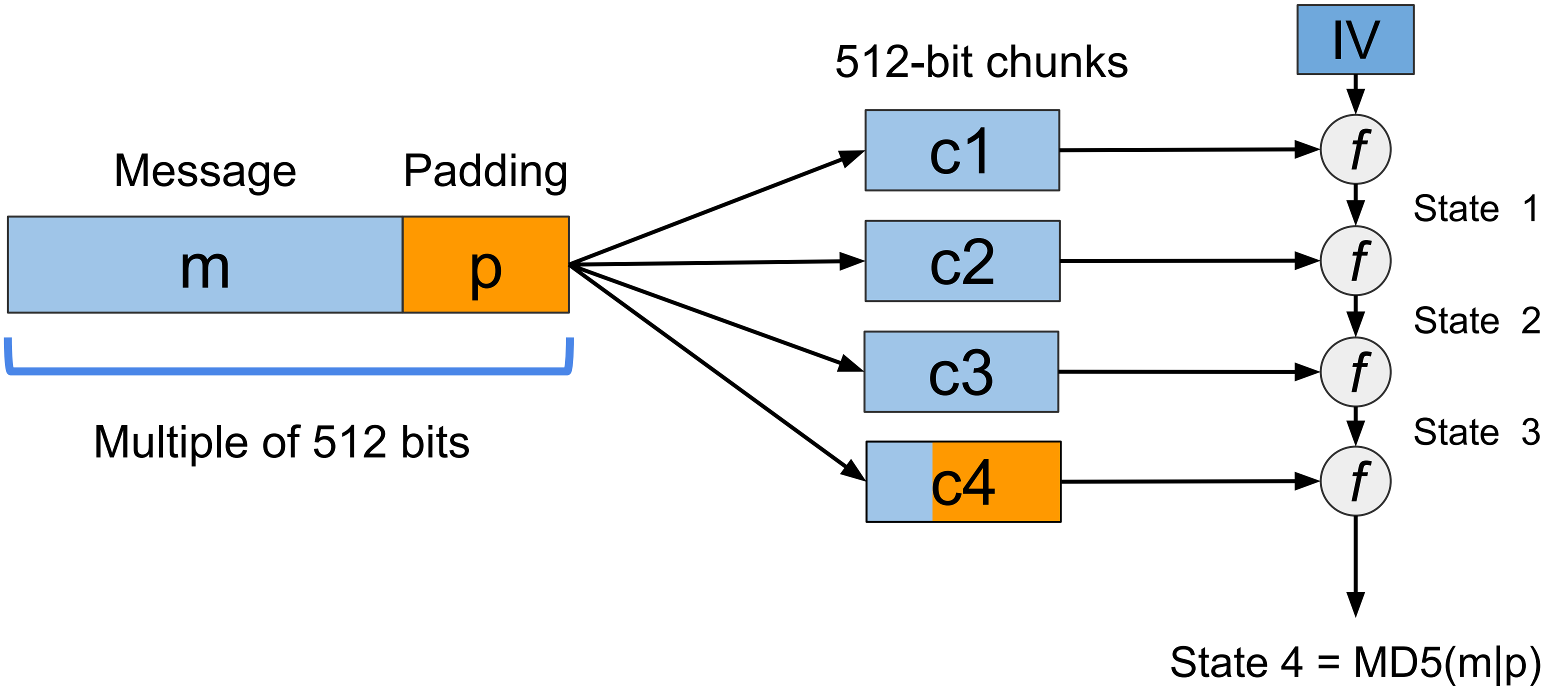
Advisory ID: MOCB-01

Advisory URL: http://netifera.com/research/flickr_api_signature_forgery.pdf

Title: Flickr's API Signature Forgery Vulnerability

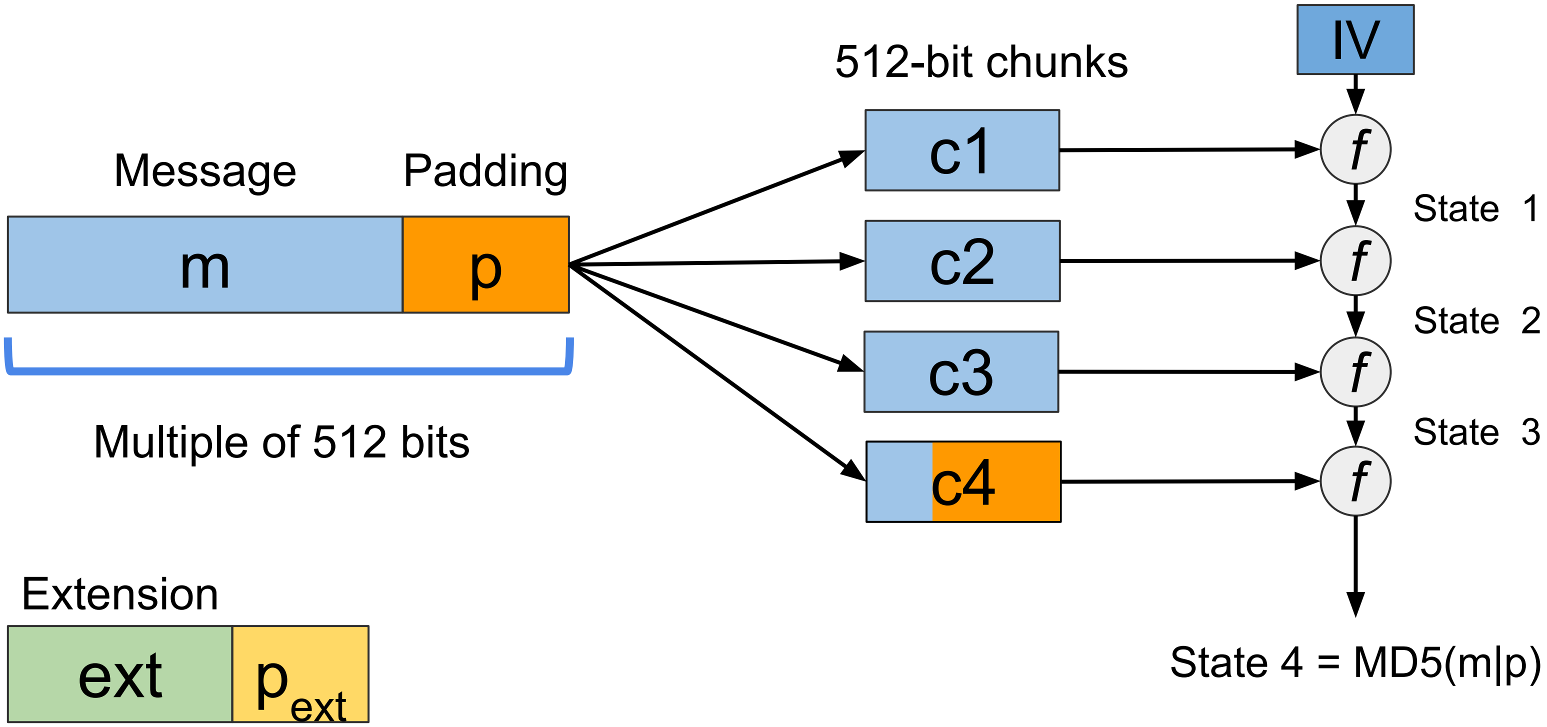
Remotely Exploitable: Yes

3.2.1 – MD5 Length Extension

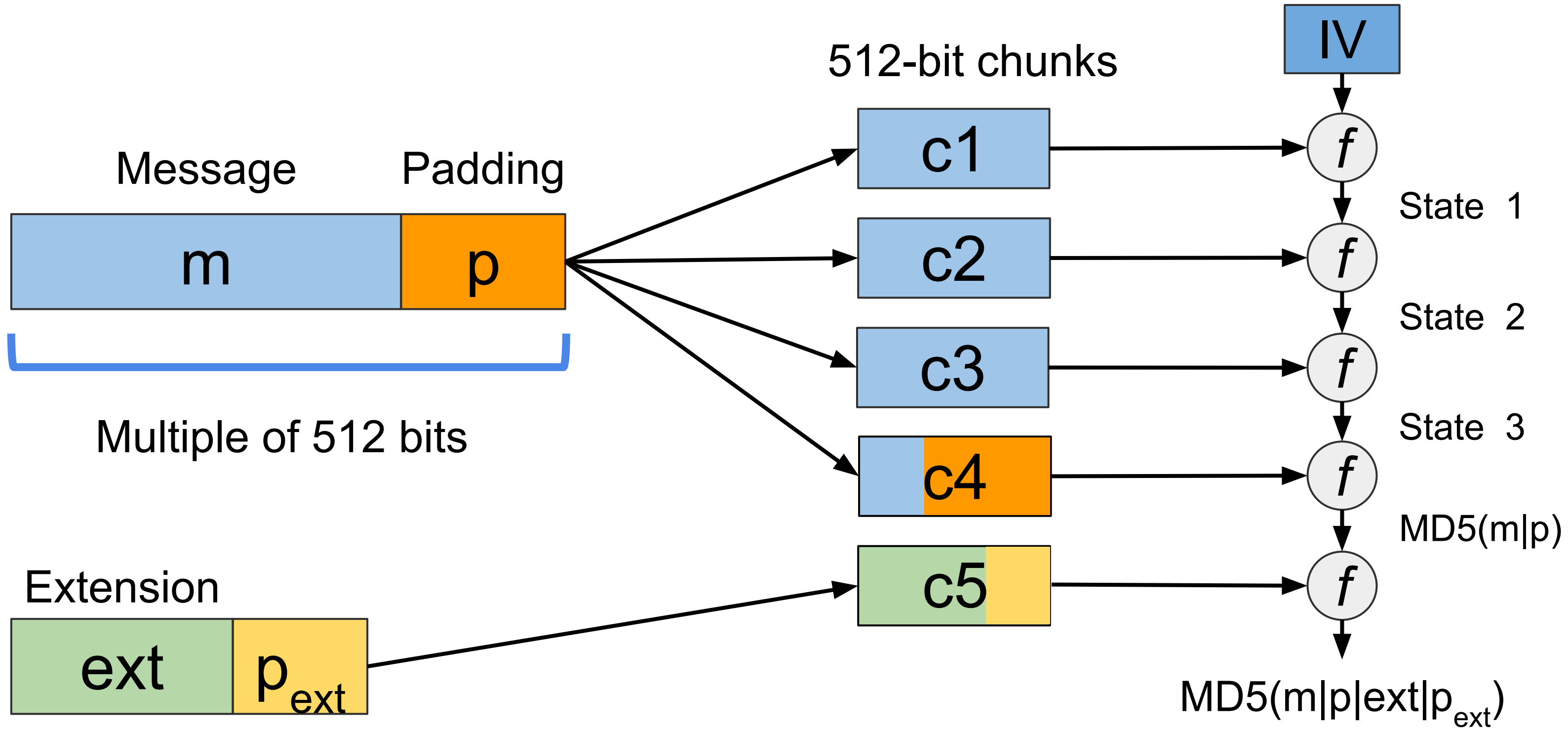


f = one-way compression function

3.2.1 – MD5 Length Extension



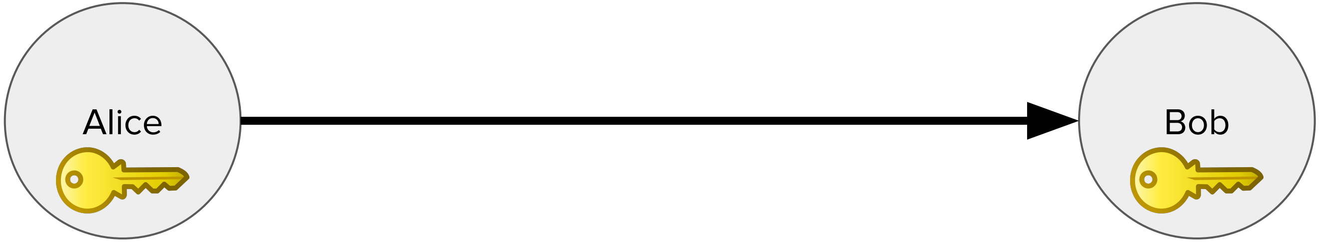
3.2.1 – MD5 Length Extension



3.2.1 – MD5 Length Extension

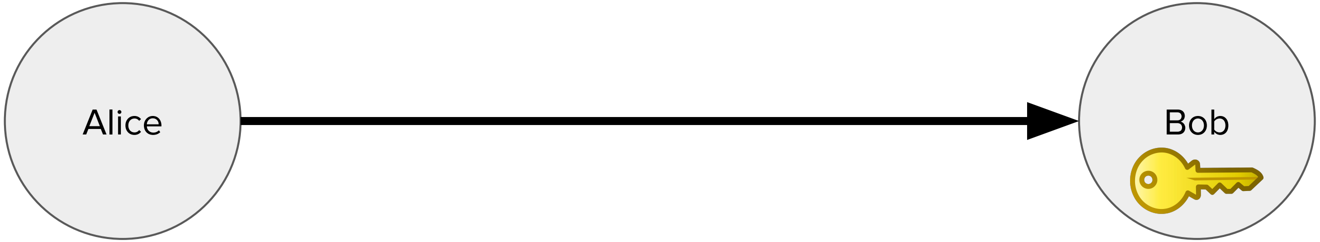
- MAC: Message Authentication Code
  - Goal: Verify message integrity
  - Not: Secrecy
- HMAC: Hash-based Message Authentication Code
  - We can build a MAC using hash functions!
  - Intuition: add a secret key k to the message before hashing!
  - HMAC(m,k) = H(k | m | p)
- Only someone with possession of k can generate/verify a message m that matches HMAC(m,k)

3.2.1 – MD5 Length Extension



"CS461 is hard"

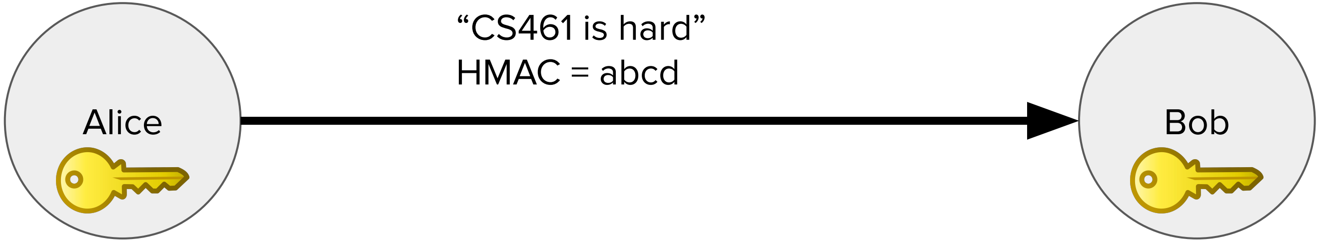
3.2.1 – MD5 Length Extension



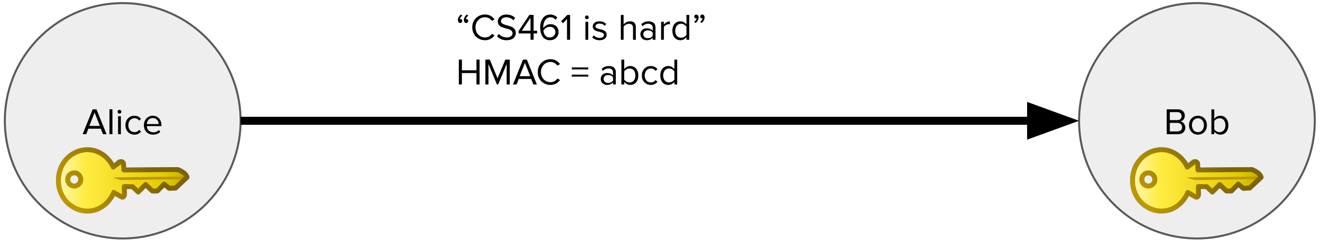
“CS461 is hard”

H( || “CS461 is hard” || padding) = abcd

3.2.1 – MD5 Length Extension



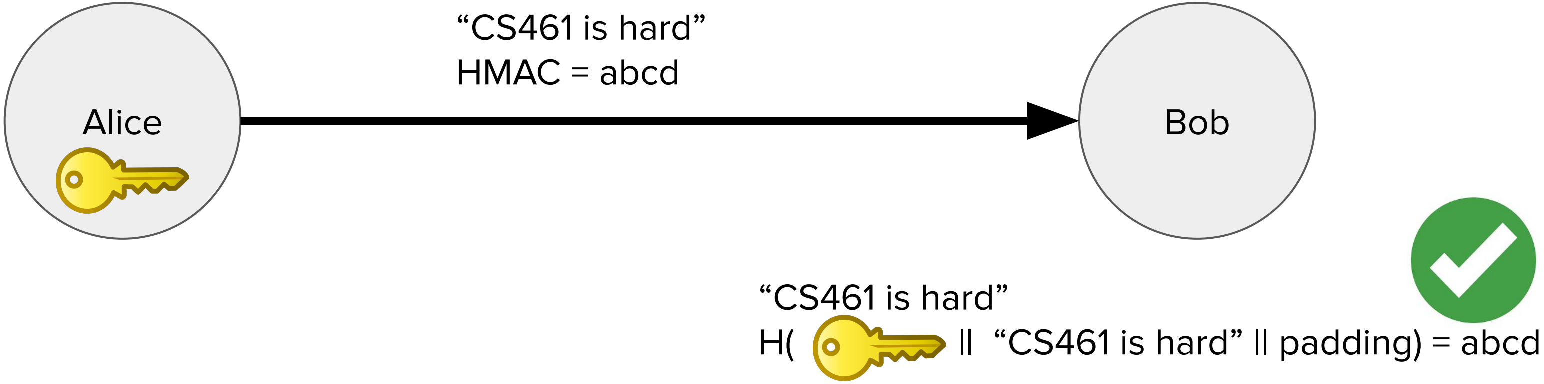
3.2.1 – MD5 Length Extension



"CS461 is hard"

H(|| "CS461 is hard" || padding) =? abcd

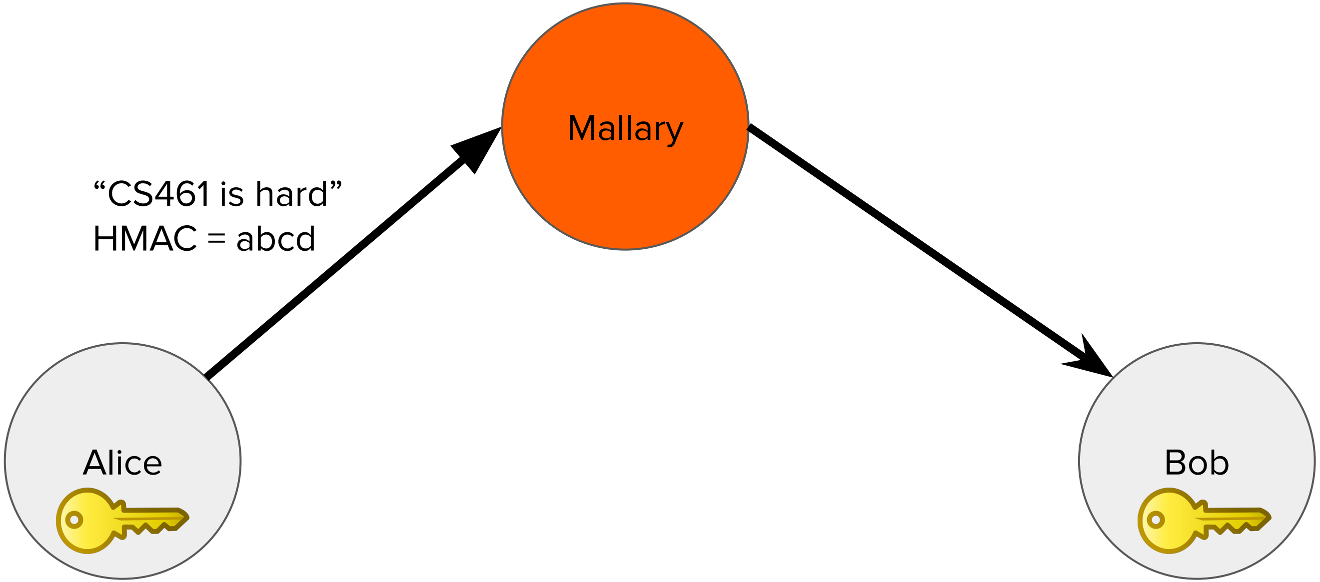
3.2.1 – MD5 Length Extension



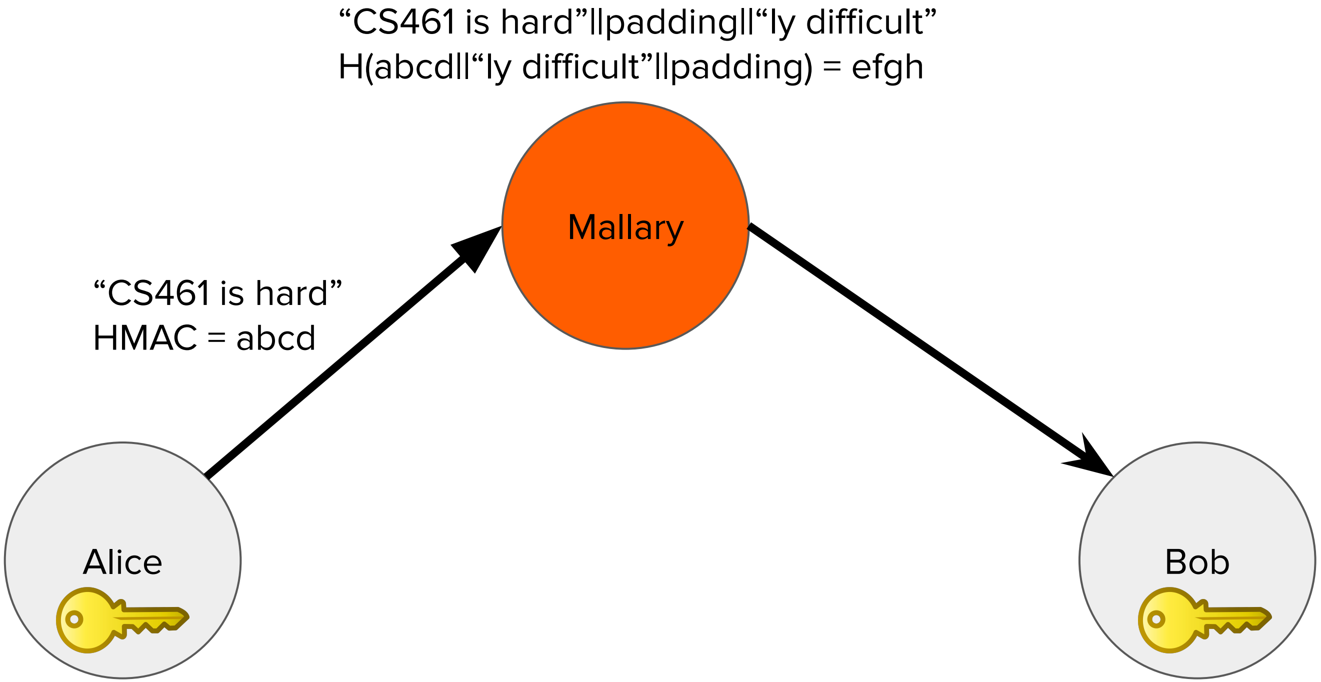
3.2.1 – MD5 Length Extension

- We (attacker) know m and HMAC(m,k) = H(k | m | p)
- Goal is to produce:
  - m′ = m | p | ext and a valid HMAC(m′,k) = H(k | m | p | ext | p′)
- How? Use length extension!
- Attacker can send m′ and HMAC(m′,k) ← without knowing k!
- You have to: 1) craft malicious m′ and 2) calculate HMAC(m′,k)

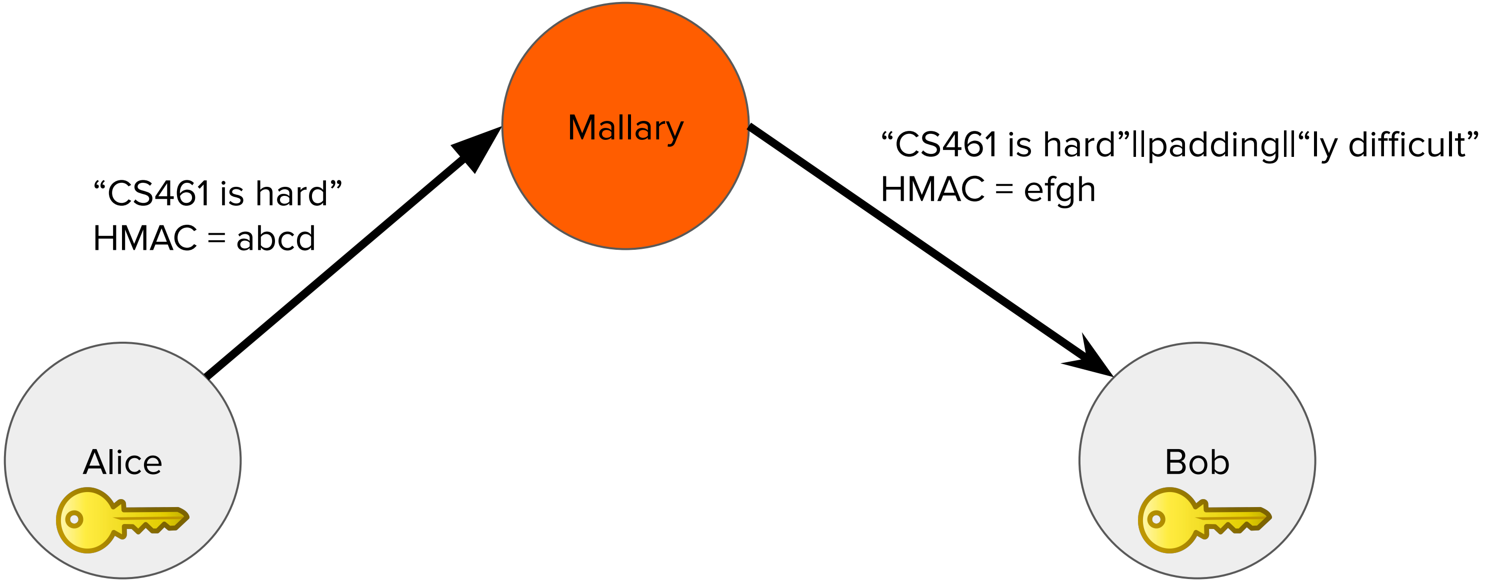
3.2.1 – MD5 Length Extension



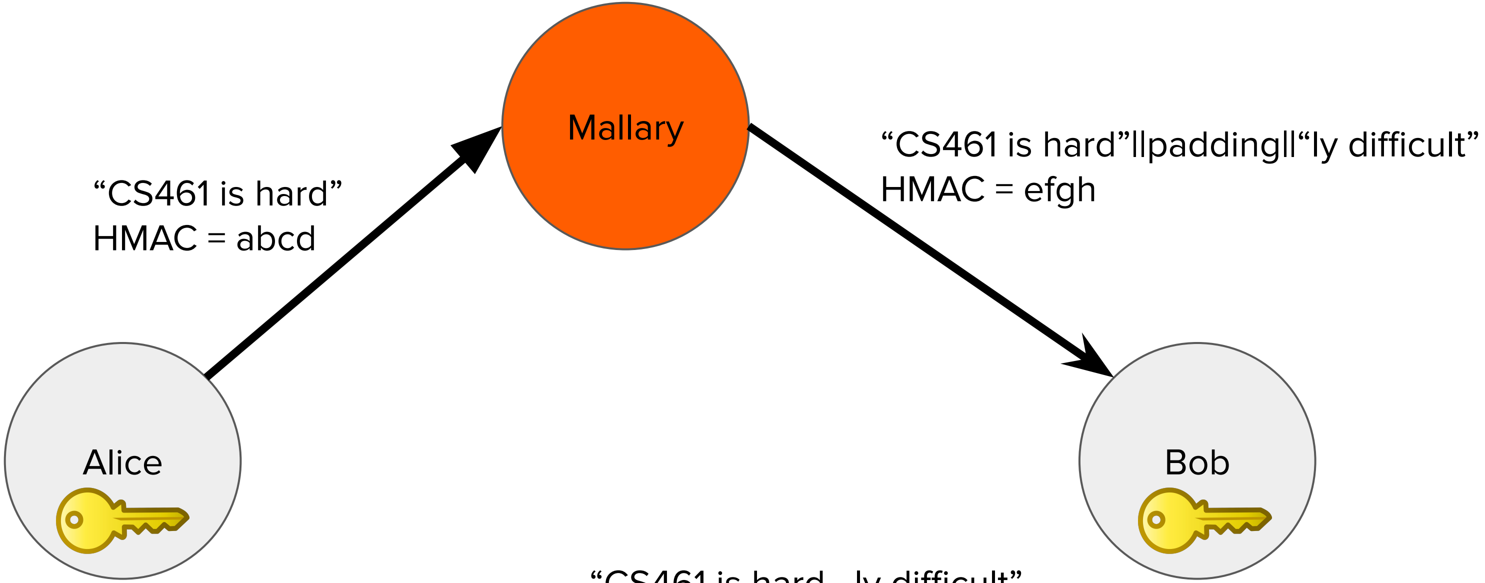
3.2.1 – MD5 Length Extension



3.2.1 – MD5 Length Extension



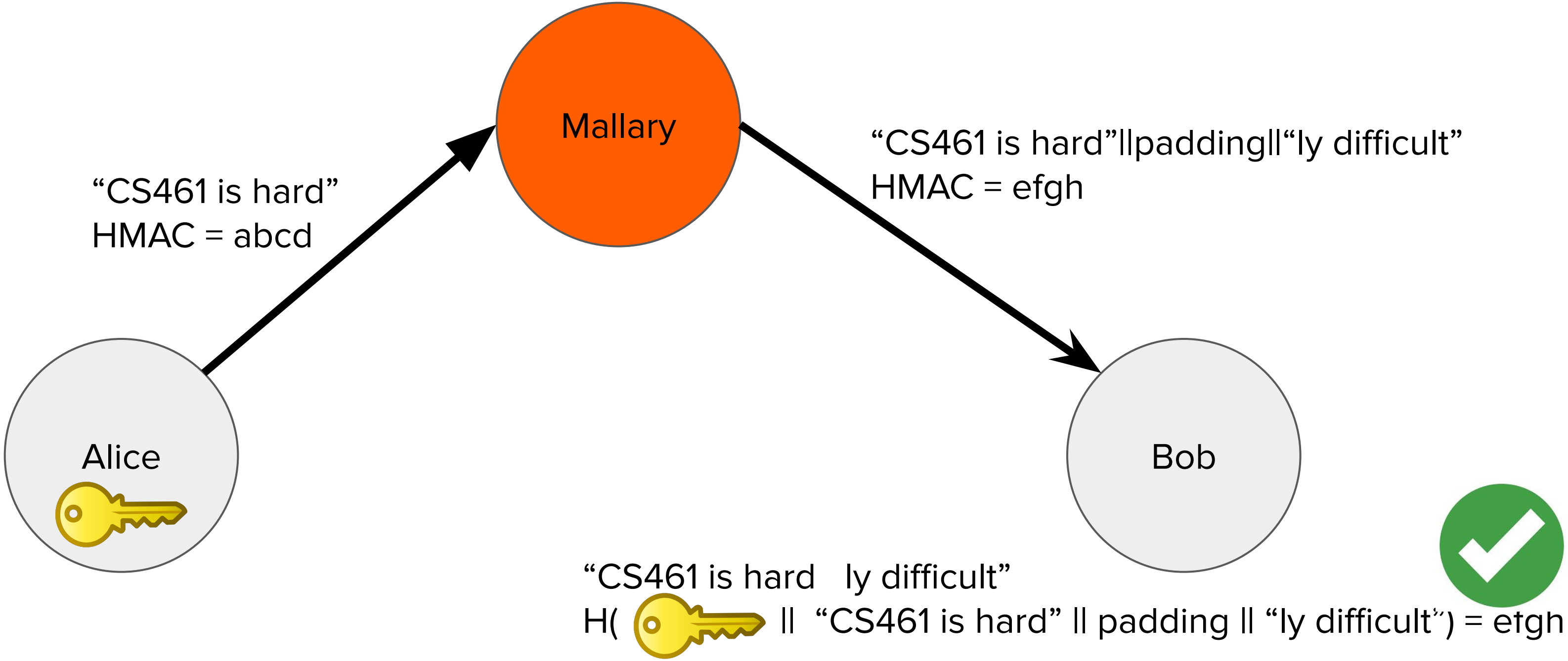
3.2.1 – MD5 Length Extension



“CS461 is hard ly difficult”

H(|| “CS461 is hard” || padding || “ly difficult”) =? efgh

3.2.1 – MD5 Length Extension



Next week we'll learn about

- 3.2.2 – generating two files with the same MD5 hash (collisions)
- 3.2.3 – decrypt an AES ciphertext w/o the key (padding oracle)
- 3.2.4 – create a pair of distinct (but valid) certificates, which both share the same signature.
- 3.3.1 – decrypt RSA ciphertext by factoring weak moduli.