Mert D. Pesé

Clemson University

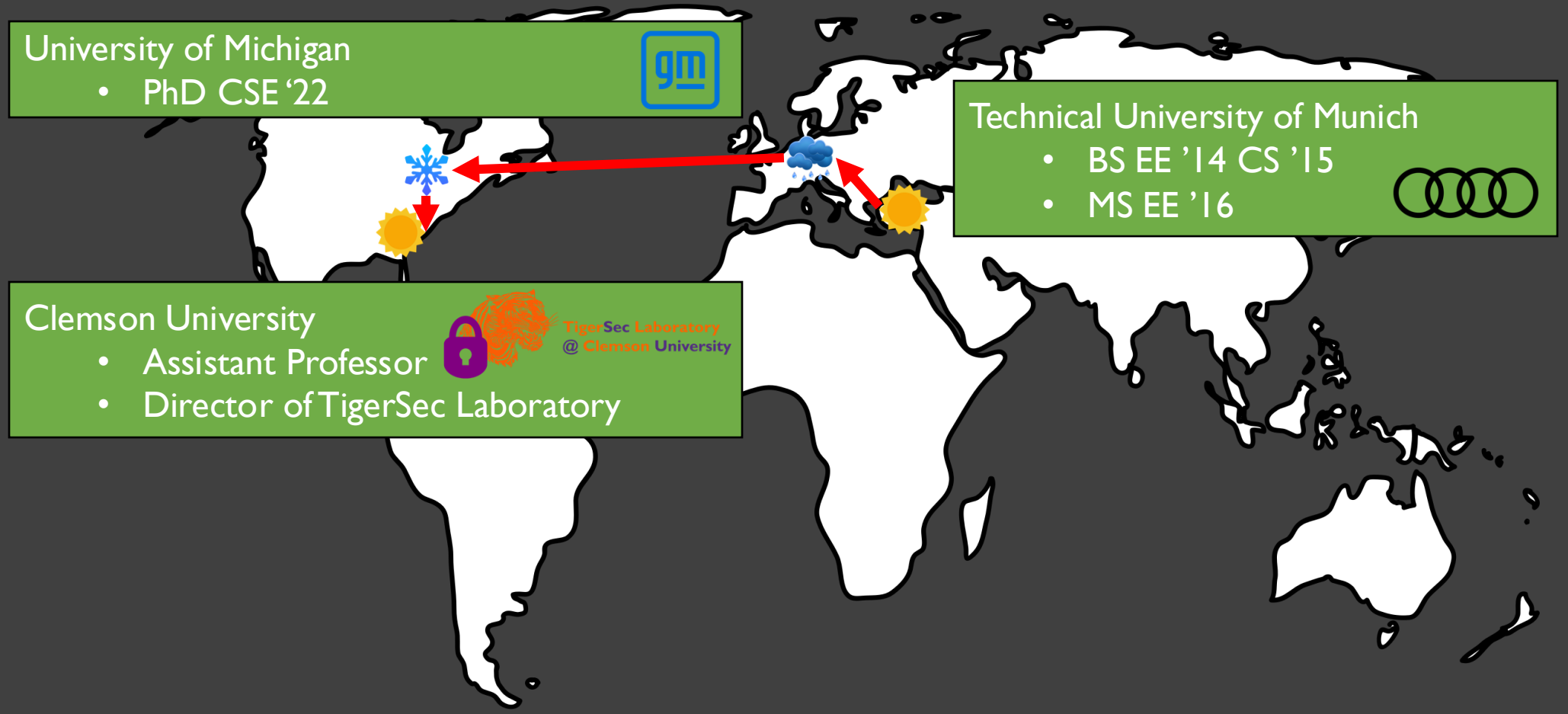
Applied Crypto

CyberBoat Challenge 2024

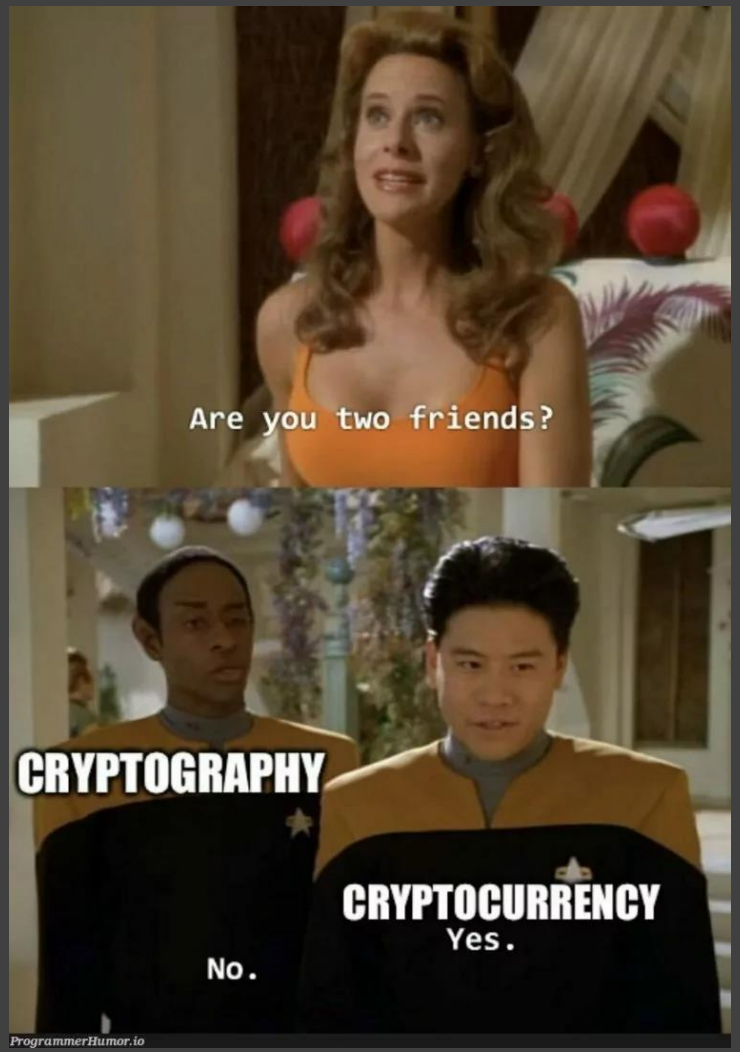
CLEMSON UNIVERSITY

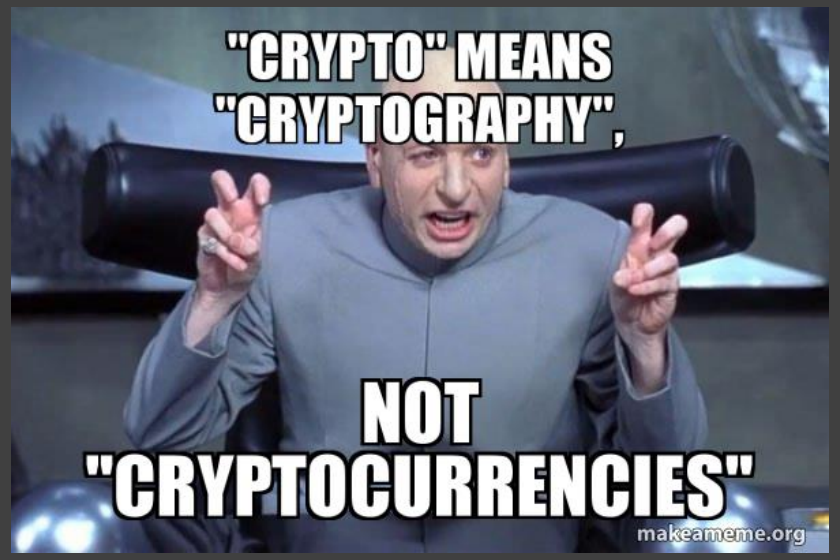
About Me

Mert D. Pesé, Ph.D. (mpese@clemson.edu)
Assistant Professor, Clemson University
Area of Research: Automotive Security and Privacy



Today's lecture: Applied Crypto





Agenda

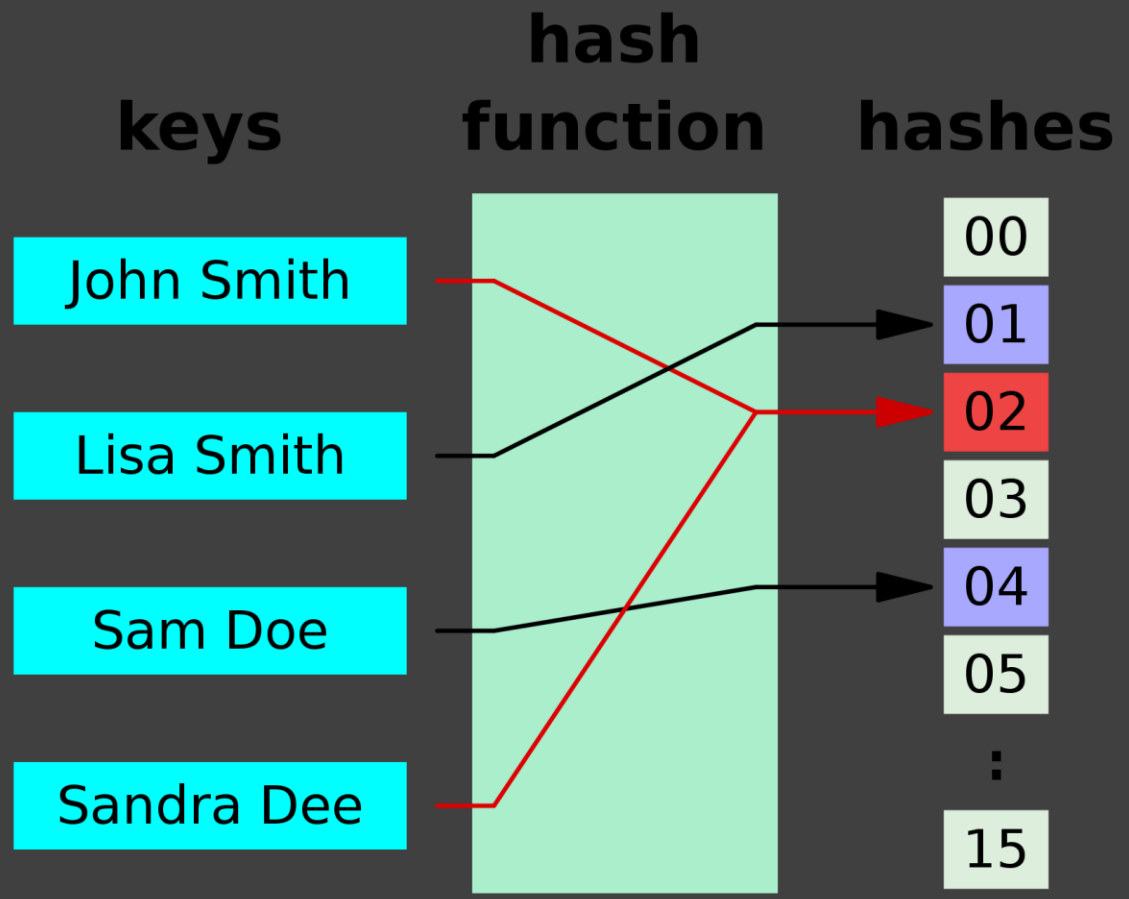
- **Part I: Hashing**
- Part II: Encryption
- Part III: Authentication
- Part IV: Protocols

Hash functions

- Cryptographic Hash Function **h**
  - Take message **m** of arbitrary length and produces a smaller, fixed-size number **h(m)**
  - Output of hash function is called hash/digest
- Desired properties of h:
  - Collision resistance: Hard to find $x \neq y$ such that $h(x) = h(y)$
  - Preimage resistance: Given y, hard to find x such that $f(x) = y$
  - Second preimage resistance: Given fixed x, should be hard to find $x' \neq x$ such that $h(x) = h(x')$
- Examples: MD5, SHA-1, SHA-256, SHA-512, CRC32, bcrypt, scrypt

Hash Function Properties: Example

- Collision resistance: Hard to find $x \neq y$ such that $h(x) = h(y)$
- Preimage resistance: Given y, hard to find x such that $f(x) = y$
- Second preimage resistance: Given fixed x, should be hard to find $x' \neq x$ such that $h(x) = h(x')$



Bad Hash Functions

- **MD5**
  - Once ubiquitous, broken in **2004**
  - Now it's easy to find collisions (pairs of messages with same MD5 hash)
  - Exploited to attack real systems
- **SHA1**
  - Fairly widely used, deprecated by NIST in 2011
  - Started to be unsupported in HTTPS in **2016**
  - First collision announced in **2017**
  - Getting the Internet to change is hard, phased out by **2030**
  - Don't use in new applications!
- Rule of thumb: Hash output should be at least 256 bits long

Google Security Blog

The latest news and insights from Google on security and safety on the Internet

Announcing the first SHA1 collision

February 23, 2017

Posted by Marc Stevens (CWI Amsterdam), Elie Bursztein (Google), Pierre Karpman (CWI Amsterdam), Ange Albertini (Google), Yarik Markov (Google), Alex Petit Bianco (Google), Clement Baisse (Google)

Hash Functions: Examples

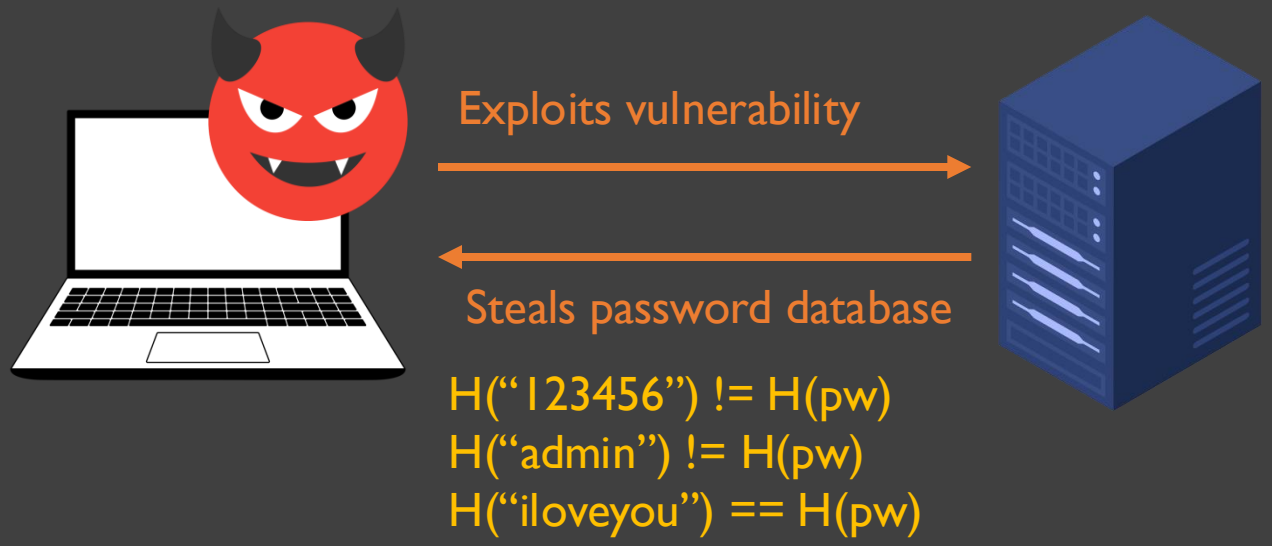
- Downloading software online
  - Large download files contain separate text file with MD5 hashsum
  - Run ***md5sum [FILE]*** on Linux, compare with hashsum in text file
- De-duplication
  - Search for duplicate files on your hard drive
- Passwords on backend database
  - Never store entire password in plaintext
  - Better: Store hashed password. **PROBLEMS?**
  - Even better: Add salt to password, then hash!

Defending against Password Breaches

- How should site store passwords to reduce risk?
- **Bad**: Plaintext passwords
  - **Pro**: Easy.
  - **Con**: If leaked, company goes bust.
- **Bad**: Encrypted passwords
- **Better**? Password hashes
  - Store H(password) in database.
  - Compare H(submitted_pw) to H(password) to authenticate.
  - **Pro**: Site doesn't learn password. Leaked database doesn't immediately reveal passwords.
  - **Con**: Identical passwords have identical hashes.

Attack: Offline Password Guessing

- Attacker computes hashes of possible passwords and searches for them in stolen password hash database.
- **Brute force search** of all passwords of length **n** takes $O(c^n)$ time.
- **Dictionary attack**: Search corpus of previously leaked passwords and variants.
  - Can do massively parallel hashing on EC2, GPUs, or custom ASICs.
  - When searching huge dictionary against many hashes, vast speedups by using a precomputed data structure called a Rainbow Table.



Defending against Offline Guessing

- How should site store passwords to reduce risk?
- Best: **Salted password hashes**
  - Randomly generate long **salt** when password is set. Never reuse salt!
  - Store `<salt, H(salt || password)>` in database.
  - Compare H(salt || submitted_pw) to H(salt || password).
  - **Q**: Adversary can compromise the salt too! Is this a problem?
  - **A**: Identical passwords have different salted hashes. Attacker has to restart offline guessing for each stored password. Rainbow tables are useless!
- **Q**: Which hash function to use?
- **A**: Something really slow. Good Password Hash Functions: PBKDF2, bcrypt, scrypt, argon2

Part I: Hands-On Challenge (10 minutes)

- Tools
  - hashcat
  - hashid
- Useful links
  - https://hashcat.net/wiki/doku.php?id=hashcat
  - RockYou dictionary https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt
- Reverse these hashes (PartI/hashX.txt):
  - 9fde43b502745f5832dd908b040b35d0
  - 9f0c989dd3e4c7a6ef5512d6347a9af24b95ee886e9874edbebf2055166c5ed6
  - Ed81de1d5860c891f157ed20de6ff1558ae153f0 (hint: 2 character salt)

Agenda

- Part I: Hashing
- **Part II: Encryption**
- Part III: Authentication
- Part IV: Protocols

Confidentiality

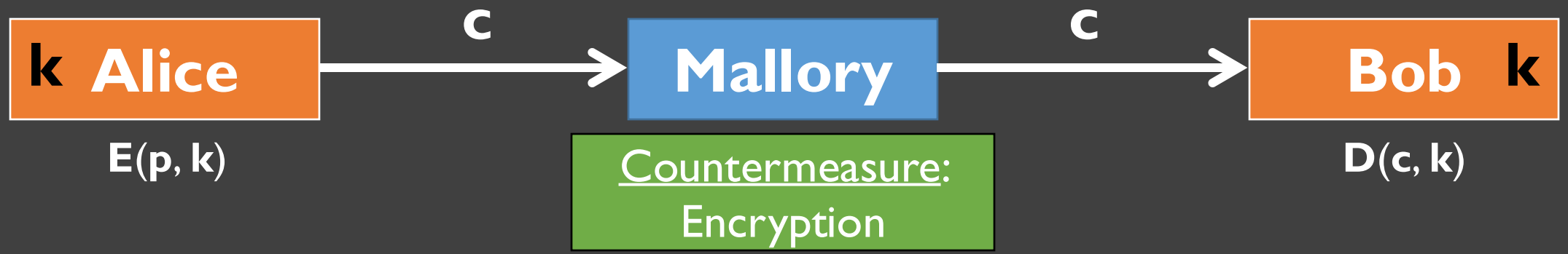
- Two parties want to communicate across an untrusted intermediary

- **Problem**: Ensure that only trusted parties (Bob) can read the message

Attack:
Eavesdropping

Confidentiality

- **Confidentiality**: Information has not been disclosed in an unauthorized way
- **Terminology**
  - **p** plaintext – original, readable message
  - **c** ciphertext – transmitted, unreadable message
  - **k** secret key – know only to Alice and Bob, used to go between p and c
  - **E** encryption function – E(p, k) → c
  - **D** decryption function – D(c, k) → p



Old School Crypto: Caesar Cipher

- First recorded use: Julius Caesar (100-44 BC)
- Replaces each plaintext letter with one a fixed number of places down the alphabet

  Encryption: $c_i := (p_i + k) \bmod 26$

  Decryption: $p_i := (c_i - k) \bmod 26$

e.g., (k=3):

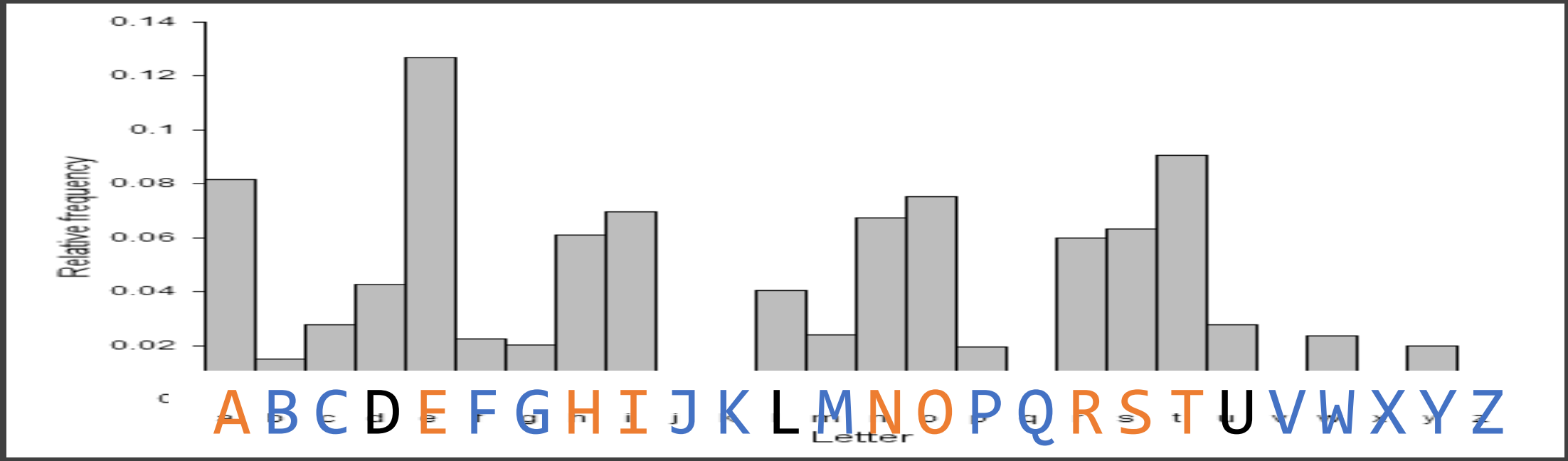
```
Plain:     ABCDEFGHIJKLMNOPQRSTUVWXYZ
+Shift:    33333333333333333333333333
=Cipher:   DEFGHIJKLMNOPQRSTUVWXYZABC

Plain:     fox   go wolverines
+Key:      333   33 3333333333
=Cipher:   ira   jr zroyhulqhv
```

How can we break it?

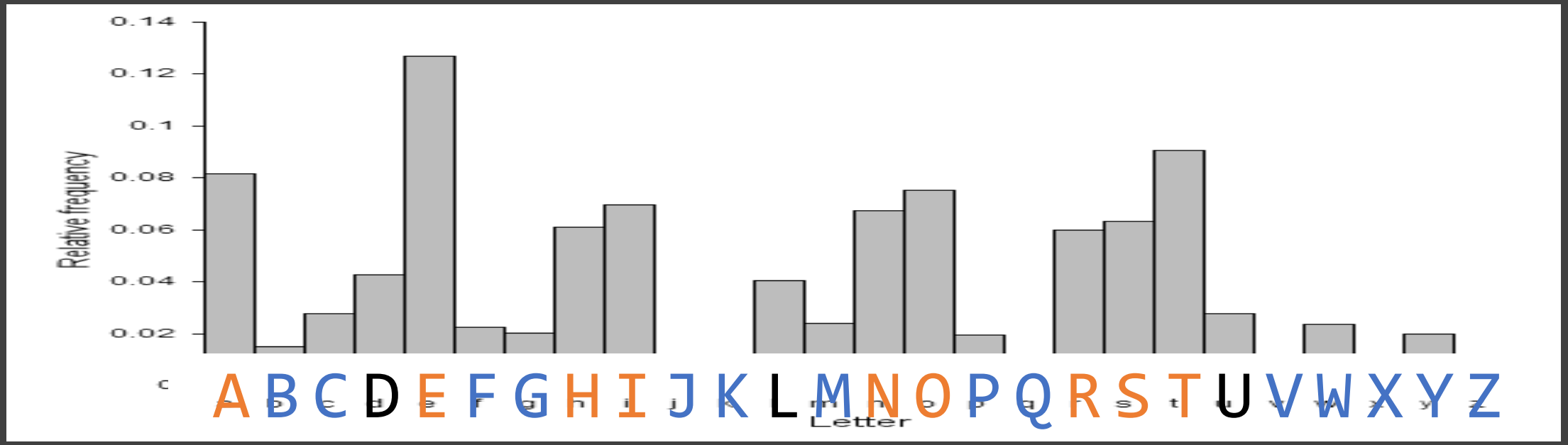
Caesar Cipher Cryptanalysis

- Only 26 possible keys: Try every possible **k** by "brute force"
- How can a computer recognize the right one?
  - Solution: English has distinctive letter frequency distribution (use χ^2 test)



Part II: Hands-On Challenge (5 minutes)

- Tools
  - https://gchq.github.io/CyberChef/
- Decrypt this Caesar cipher (PartII/caesar.txt)
  - V cyrqtr nyyrtvnapr gb gur synt bs gur Havgrq Fgngrf bs Nzrevpn naq gb gur Erchoyvp sbe juvpu vg fgnaqf, bar Angvba haqre Tbq, vaqvivfvoyr, jvgu yvoregl naq whfgvpr sbe nyy.



Back to the Present: One-Time Pad (OTP)

- Alice encrypts, Bob decrypts using same secret key *k*
- Intuition: One-Time Pad (OTP) for E and D
  - Alice and Bob jointly generate a secret, very long, string of truly random bits (the OTP **k**)
  - To encrypt: $c_i = p_i$ xor k_i
  - To decrypt: $p_i = c_i$ xor k_i
- Theoretically secure because the cipher text doesn't reveal any information (aside from length) about plain text

a	**b**	**a** xor **b**
0	0	0
0	1	1
1	0	1
1	1	0

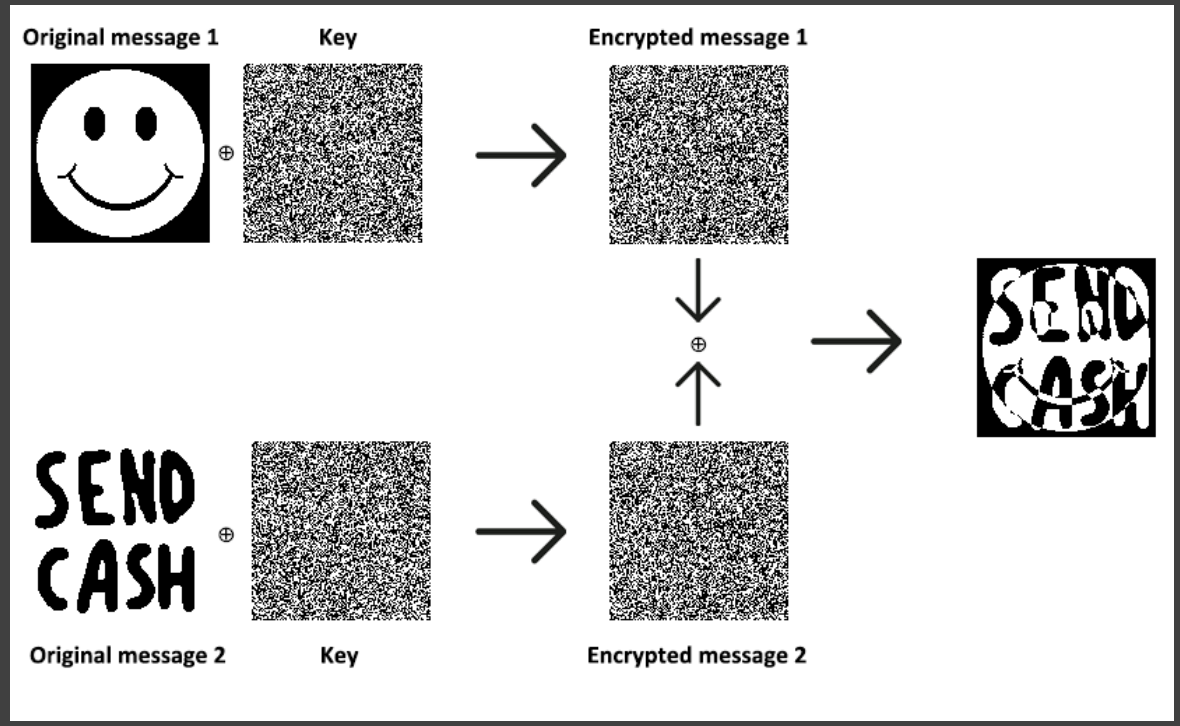
a xor **b** xor **b** = **a**
a xor **b** xor **a** = **b**

Should we trust our intuition?

- **No.**
- "One-time" means you should never reuse any part of the pad. If you do:
  - Adversary learns (a xor k) and (b xor k)
  - Adversary xors those to get (a xor b) which is useful to them

a	**b**	**a** xor **b**
0	0	0
0	1	1
1	0	1
1	1	0

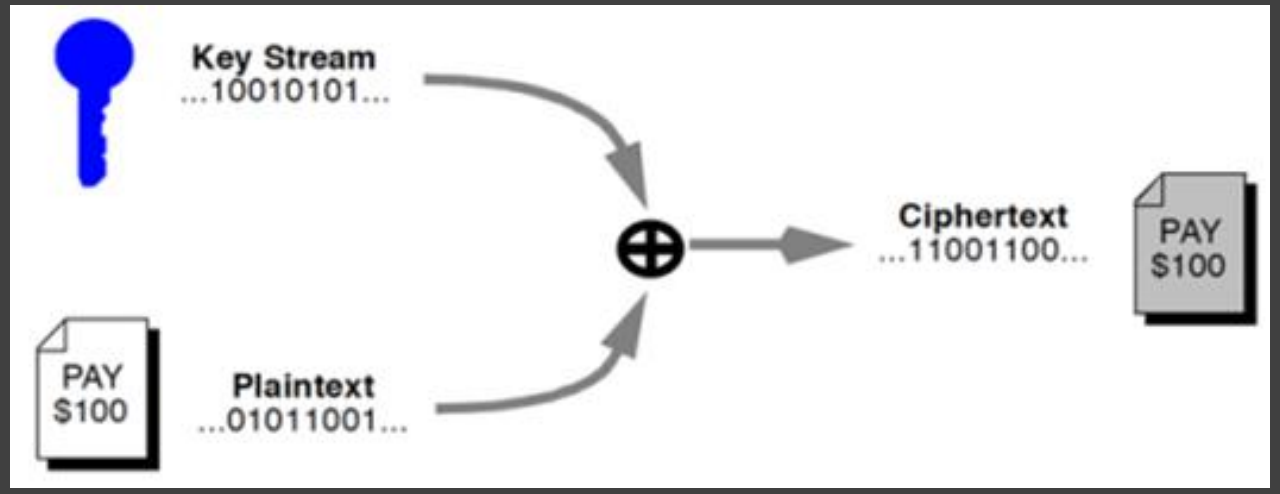
a xor **b** xor **b** = **a**
a xor **b** xor **a** = **b**



Stream Cipher

- Similar to OTP, but use secret k as seed to generate pseudorandom bit stream
- Encrypt data one bit at a time
- Used if data is a constant stream of information
- Example: RC4, Salsa20

Vulnerable, DO NOT USE!!!

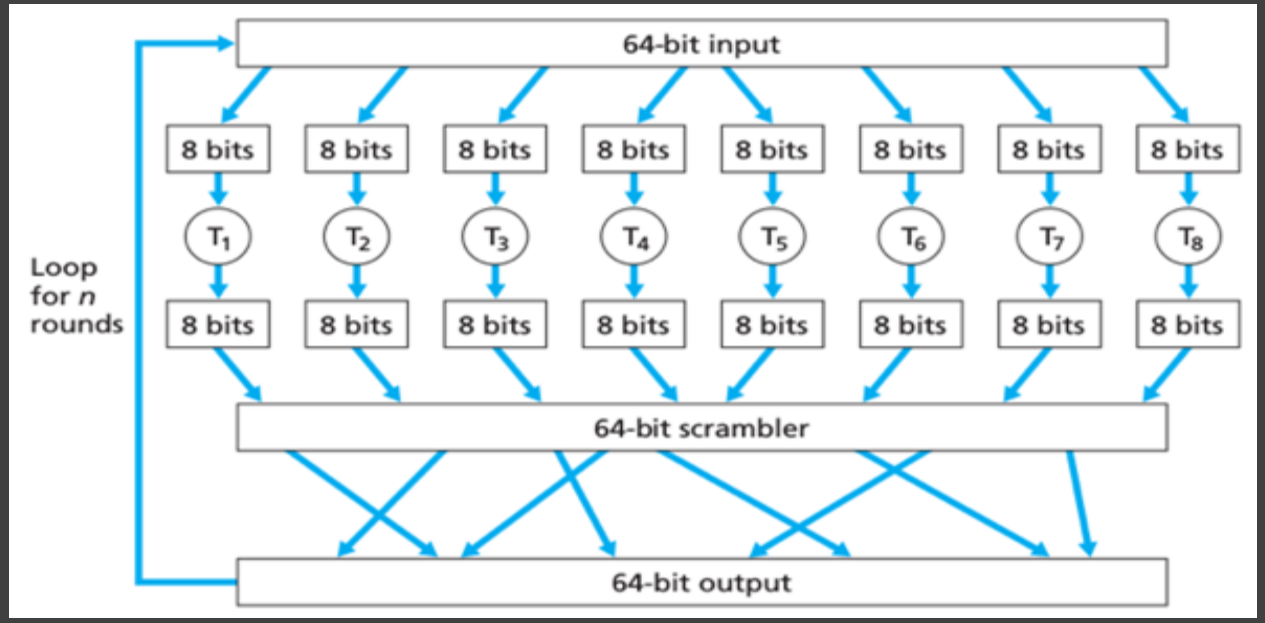


Part II: Hands-On Challenge (10 minutes)

- Tools
  - https://www.rapidtables.com/convert/number/ascii-to-hex.html
  - https://xor.pw/
- Ciphertext 1 (PartII/ciphertext1.txt): 0x547d1a3da0
- Ciphertext 2 (PartII/ciphertext2.txt): 0x465c3f0bbad5611437 58eed9614078
- Ciphertext 3 (PartII/ciphertext3.txt): 0x51601058f9d473583a1df4db7747760af5df7915
- Hints
  - Keystream of the stream cipher is 6 bytes long and repeats throughout the encryption process.
  - First ciphertext (Ciphertext 1) corresponds to the plaintext "FILE:".
- Your tasks:
  - Recover the Keystream: Use the known plaintext hint and Ciphertext 1 to deduce the 6-byte repeating keystream.
  - Decrypt the Ciphertexts: Use the recovered keystream to decrypt Ciphertext 2 and Ciphertext 3 and reveal their plaintexts.

Block Ciphers

- Functions that encrypts fixed-size blocks with a reusable key
- Different ciphers use different block sizes and key lengths
- Inverse function decrypts when used with same key
- Most commonly used approach for encryption



How should a good block cipher look like?

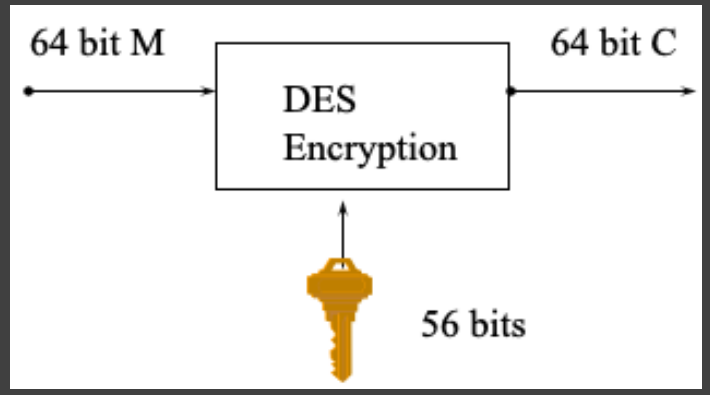
- Confusion
  - each bit of the ciphertext should depend on several parts of the plaintext
  - destroy features of the plaintext
- Diffusion
  - changing a single bit of plaintext, statistically changes 50% of the ciphertext bits, and similarly, changing one bit of the ciphertext, causes 50% of the plaintext bits to change
  - hides features of the plaintext

Example: Data Encryption Standard (DES)

- Published in 1977, standardized in 1979
- Key: 64 bit quantity = 8-bit parity + 56-bit key

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

- 64 bit input, 64 bit output
- Widespread use until 1997
  - Why?



Based on the Feistel cipher structure with 16 rounds of processing

Example: Data Encryption Standard (DES)

- Brute-force key search
  - Trying 1 key/microsecond would take 1000+ years on average, due to the large key space size, $2^{56} \approx 7.2 \times 10^{16}$
  - 1997: on Internet in a few months
  - 1998: on dedicated hardware (EFF) in a few days
  - 1999: above combined in 22hrs!
- Alternatives to DES
  - Triple DES (3DES)
    - 168-bit key, no brute force attacks (longer than existence of galaxy)
    - Encryption is slower than DES
  - Advanced Encryption Standard (AES)
    - US NIST issued call for ciphers in 1997
    - Daemen & Rijmen's cipher was selected as the AES in 2000

No longer considered secure: 56 bit keys are vulnerable to exhaustive search

Today: Use at least 80 bits, preferably 128 bits

	DES	AES
Developed	1977	2000
Key Length	56 bits	128, 192, or 256 bits
Cipher Type	Symmetric block cipher	Symmetric block cipher
Block Size	64 bits	128 bits
Security	Proven inadequate	Considered secure

How to encrypt longer messages?

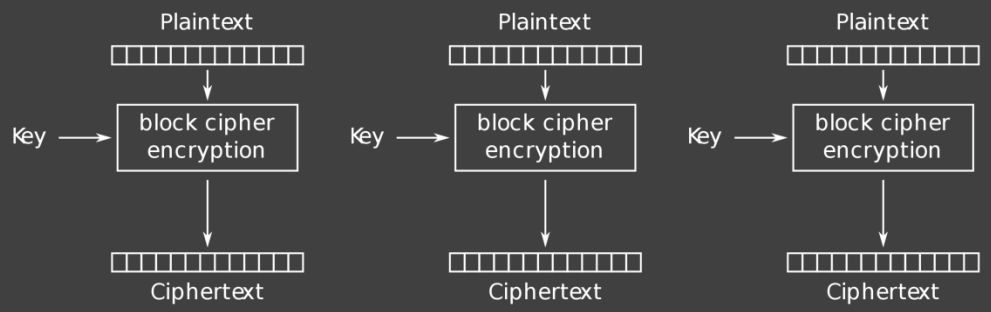
- Can only encrypt in units of cipher blocksize
  - Cut message into multiple chunks of blocksize
- Message might not be multiples of blocksize
  - Add padding to end of message
- How to properly encrypt multi-block messages?
  - Electronic Code Book (ECB)
  - Cipher Block Chaining (CBC)
  - Cipher Feedback (CFB)
  - Output Feedback(OFB)
  - Counter (CTR)

Cipher Modes

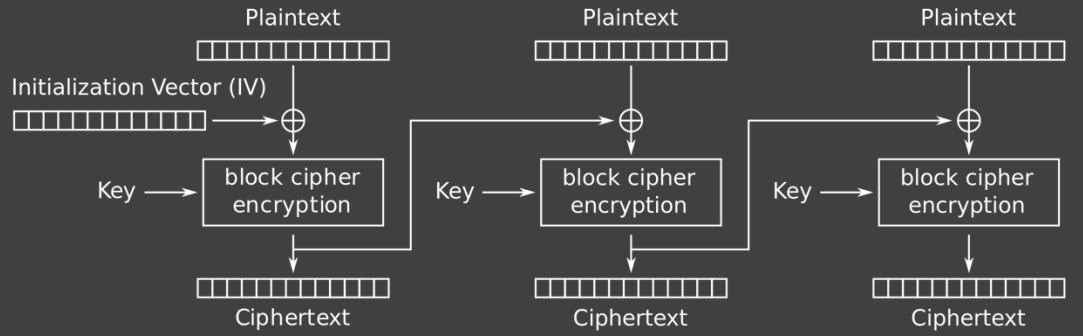
Plaintext Pseudorandom ECB mode

Don't use ECB!

CBC (Cipher-Block Chaining) Mode



Electronic Codebook (ECB) mode encryption



Cipher Block Chaining (CBC) mode encryption

- Initialization: $C_0 \leftarrow \mathrm{IV}$
- Encryption: $C_i \leftarrow E_K(B_i \oplus C_{i-1})$
- Decryption: $B_i \leftarrow D_K(C_i) \oplus C_{i-1}$

Agenda

- Part I: Hashing
- Part II: Encryption
- **Part III: Authentication**
- Part IV: Protocols

Integrity

- Two parties want to communicate across an untrusted intermediary

- **Problem:** Ensure that a message received by Bob is a message sent by Alice

Attack:
Spoofing

Integrity

- **Integrity**: Information has not been altered in an unauthorized way
- Threat model
  - Mallory can see, modify, forge messages
  - Mallory wants to trick Bob into accepting a message Alice didn‘t send
- Risk assessment
  - Very likely that Mallory will distort all communication between Bob and Alice

Countermeasure:
Message Authentication Code (MAC)

How to enforce integrity?

- Two parties want to communicate across an untrusted intermediary

- **Problem:** Ensure that a message received by Bob is a message sent by Alice

- Let v = h(m)
- Bob checks that h(m') == v', otherwise m' untrusted

Function h should:

- Consistent output
- 1-to-1 mapping between m and h(m)
- Unknown to Mallory

Is it really a good idea to use hash functions?

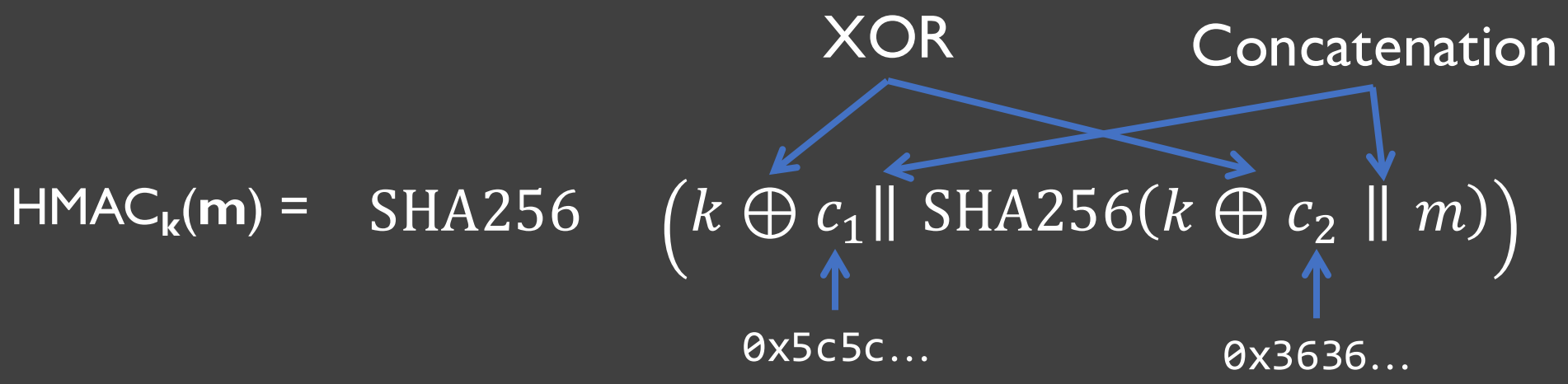
- **No.**
- If hash function h is known to Mallory, game over.
  - Mallory can calculate correct v = h(m) all the time
  - If v depends on another parameter **k**, Mallory has to know both function and **k**
  - **k** is called secret key, $v = h_k(m)$
  - Mallory can know which hash function **h** is used (public knowledge), but cannot know secret key **k** (can be changed at certain intervals)
  - Kerckhoff's principle: Security of a cryptosystem should only depend on the secrecy of the key, not of the algorithms involved.

"Security by obscurity"

"Kerckhoff's Principle"

So, what shall we do?

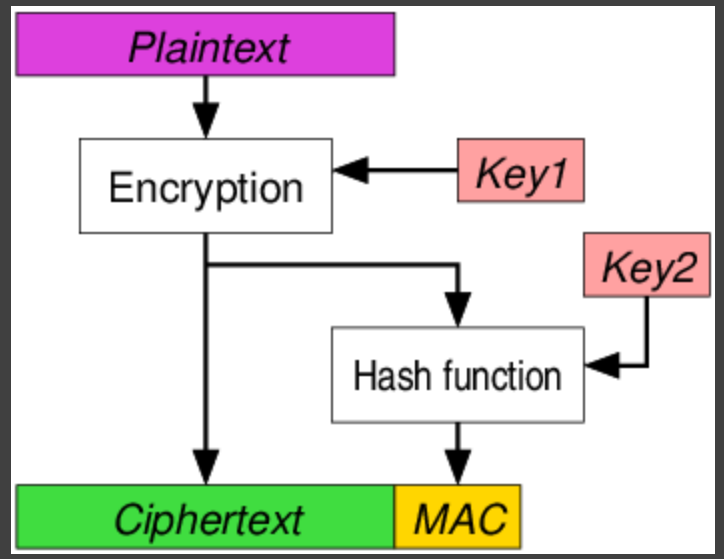
- Use Hash-Based Message Authentication Code (HMAC)
  - Example: HMAC-SHA256

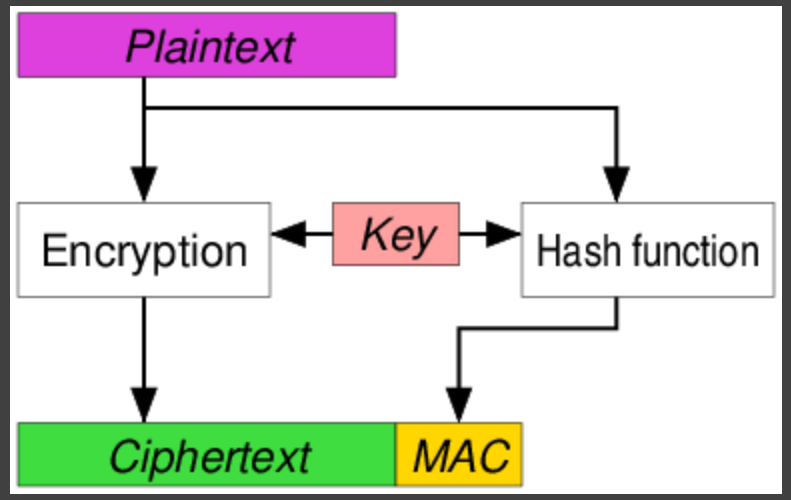


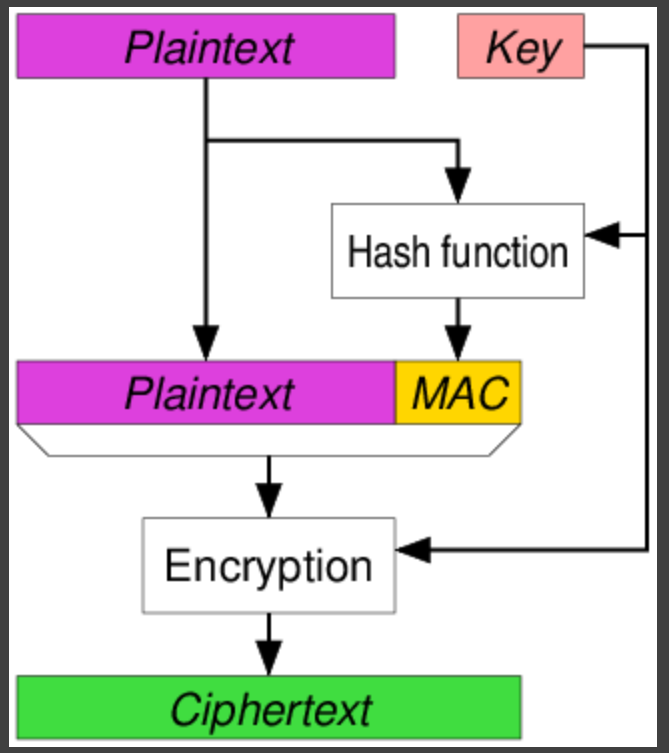
- Alice (sender) generates HMAC, Bob (receiver) verifies HMAC
  - Both Alice and Bob must share the same secret key *k*
  - This is called **symmetric cryptography**

How to enforce confidentiality and integrity?

- Which one to choose?







(1) Encrypt-then-MAC

(2) Encrypt-and-MAC

(3) MAC-then-Encrypt

- Use separate keys for confidentiality and integrity

Secret Key (Symmetric) Cryptography

- Assumption we've been making so far:
  Alice and Bob shared a secret key in advance
- Limitation: Sender and receiver must share the same key
  - Needs secure channel for key distribution
  - Impossible for two parties having no prior relationship
  - Use Diffie-Hellman (DH) to determine shared secret/key
  - Needs many ($\frac{N(N-1)}{2}$) keys for **N** parties to communicate
  - Example: If everybody in this class (**N**=40) wanted to chat over a secure channel, everyone had to store 780 keys!
  - 1560 keys if you use a key for integrity and confidentiality each!
  - 3120 keys if you consider the back-channel!

Problem: Scaling!

Problem: Scaling

- Suppose Alice publishes data to lots of people, and they all want to verify integrity…
  - Can't share an integrity key with everybody, or else anybody could forge messages!

- Suppose Bob wants to receive data from lots of people, confidentially…
  - Schemes we've discussed would require a separate key shared with each person

Public-Key (Asymmetric) Cryptography

- So far: Encryption key E_k = Decryption key D_k
  - **Symmetric-Key** Cryptography
  - Requires separate key shared with each person

- New idea: Each party has a pair (K, K^{-1}) of keys:
  - K is the **public key**, can be publicly shared with any other party
  - K^{-1} is the **private key**, needs to be kept secret

- Knowing the public-key K, it is computationally infeasible to compute the private key K^{-1}. Other way possible.

Asymmetric Encryption

Solves message confidentiality!

- Alice wants so send encrypted message to Bob
  - Alice knows Bob‘s public key, Bob knows Alice‘s public key
- Idea: Alice encrypts with K_B, only Bob can decrypt with K_B^{-1}
  - $D(E(p,K_B), K_B^{-1}) = p$
- Many can encrypt a message for Bob, only Bob can decrypt
  - How can Bob make sure that Alice encrypted the message for him?

Digital Signature

Solves message integrity & sender authenticity!

- Bob needs to know if message is coming from Alice
  - Alice knows Bob's public key, Bob knows Alice's public key
- Idea: Alice encrypts (**signs**) with K_A^{-1}, Bob decrypts (**verifies**) with K_A
  - Only Alice has her own private key, so message must be coming from her
  - Bob receives Alice's **digital signature d**

RSA (Rivest-Shamir-Adleman)

- RSA is the best know public-key cryptosystem. Security is based on the (believed) difficulty of factoring large numbers.
- Most famous implementation of asymmetric cryptography
- Can be used for encryption and digital signature
- Factor of 1000 or more slower than AES
- Key size is 2048 or 4096 bits!

Part III: Hands-On Challenge (10 minutes)

- Tools
  - openssl
- Generate RSA private key
  - openssl genrsa -out key.pem 2048
- Compute public key from private key
  - openssl rsa -in key.pem -outform PEM -pubout -out public.pem
- Calculate digital signature over ”Hello Cyberboat!” (PartIII/message.txt)
  - openssl dgst -sha256 -sign key.pem -out message.sig message.txt
- Verify digital signature
  - openssl dgst -sha256 -verify public.pem -signature message.sig message.txt

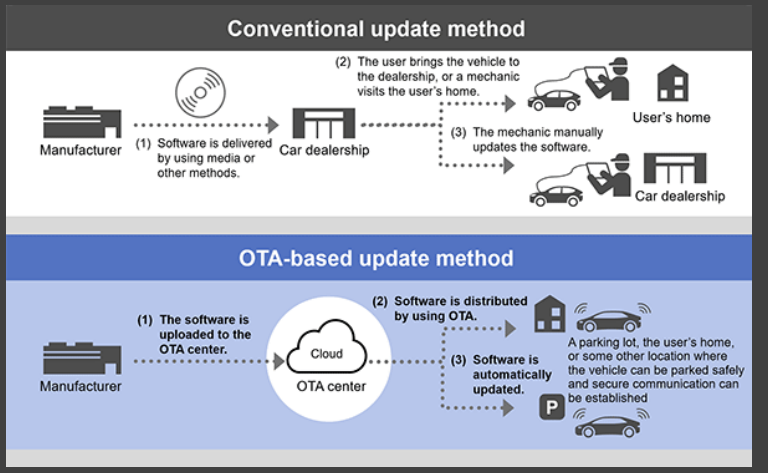
Part III: Hands-On Challenge (10 minutes)

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAngdZeS823Hd6o9JgGe7C
SxNJEQ2l3IJfbgQWf4GoIhKBtmf+b92CLlL6FKNOgK4OtJlc96wOB2cs2KXKKPpL
nRm18ezTqg5VcCkGwuqzejWSExtyY/5IB7OzRJgLcB7mNtGpY607oAN6Oq6WjXnY
lH9lwJR3E/fzdNIj85WZLmbQfI/YvV5CZZ5sjBP0sjtd6ztw2QrN9Q2U7OFyVTSF
Bd+RkFlsataKNBe+2P5cATVsXg1L003eGqFSBhzMrWJ8oRBKNWxe97IXntDi1ZSb
4h6gVvvi5DHUai3b8bDy1vgIlBTBWKTdDNmug69fBvyIbCia9E/PycOadiT1RH1y
IwIDAQAB
-----END PUBLIC KEY-----
```

- Tools
  - openssl
- Useful links
  - https://opensource.com/article/21/4/encryption-decryption-openssl
- Use pkeyutl instead of rsautl (deprecated)
- Here is my public key (PartIII/mert.pem)
- Send me encrypted messages: https://docs.google.com/document/d/1fZLo_7arghP01m098c9C83StQXwQtaMAz1H6VvYw_EQ/edit?usp=sharing

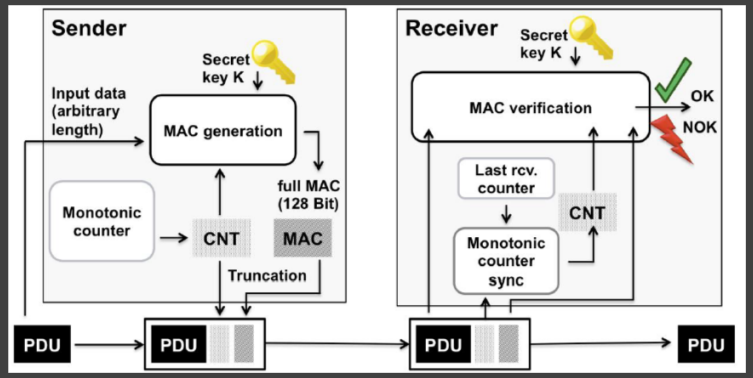
Digital Signatures in Vehicles?

- Secure Over-the-Air Updates
  - Firmware signature needs to be verified before reflashing ECU(s)
- Component Authentication and Secure Boot
  - Digital signature validates that ECU and firmware are genuine and untampered before operation
  - During boot, firmware signature is verified against OEM's public key to ensure it has not been modified
  - Prevents counterfeit "aftermarket" parts
- Secure In-Vehicle Communication
  - Spoofing protection (ensures message integrity)
  - Usually done with symmetric crypto/MAC (e.g., AUTOSAR SecOC)



Source: https://www.hitachi.com/products/it/lumada/global/en/spcon/uc_00866s/index.html

Source: https://www.keihin-na.com/automotive/electronic-control-units/

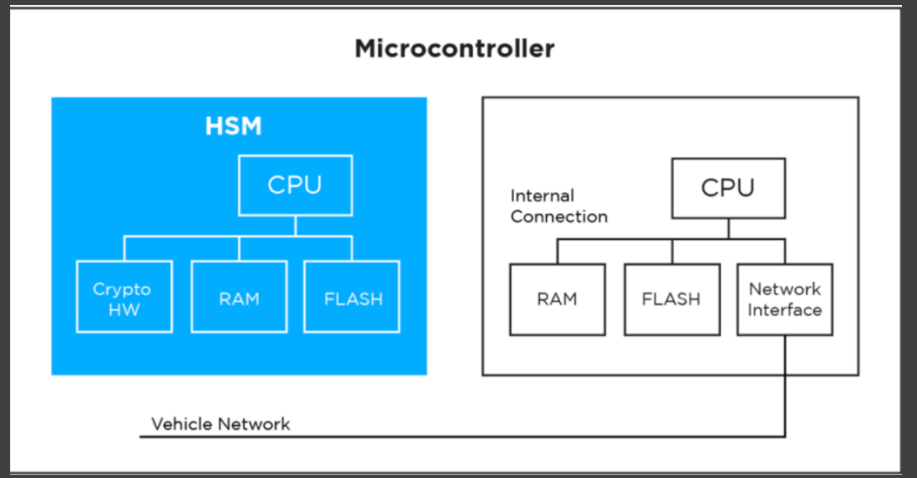


Source: https://www.renesas.com/en/blogs/art-networking-series-8-cansec-can-xl-layer-2-security-protocol

Keys must be stored securely and may not be compromised!

Hardware Security Module (HSM)

- Physical device that provides secure generation, storage, and management of cryptographic keys
  - Protect sensitive data
  - Perform cryptographic operations
  - Secure key lifecycle management
- Functions of an HSM
  - Key management (secure generation, storage, distribution of crypto keys)
  - Hardware-accelerated cryptographic operations (encryption, signing, hashing)
  - Secure storage of credentials and certificates
  - Authentication/Secure Boot (only legitimate firmware can boot)
- Standard APIs to interface HSM (e.g., SHE+)



Source: https://autocrypt.io/hsm-tee-closer-look-at-vehicle-cybersecurity/

Agenda

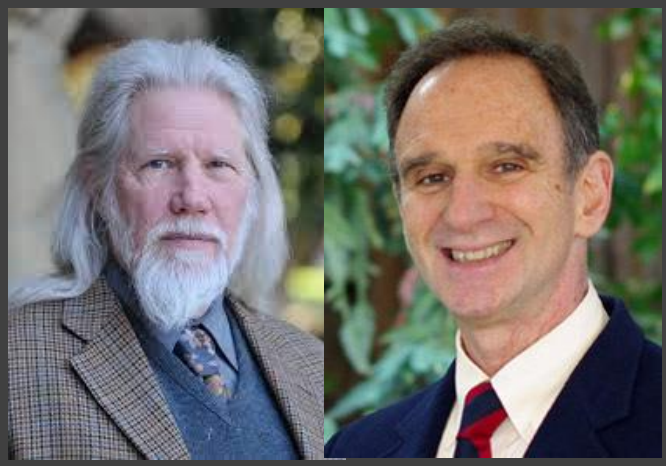
- Part I: Hashing
- Part II: Encryption
- Part III: Authentication
- **Part IV: Protocols**

Secret Key (Symmetric) Cryptography

- Assumption we've been making so far:
  Alice and Bob shared a secret key in advance
- Limitation: Sender and receiver must share the same key
  - Needs secure channel for key distribution
  - Impossible for two parties having no prior relationship
  - Needs many keys for **N** parties to communicate

Amazing fact:
Alice and Bob can have a public conversation to derive a shared key!

How to derive a shared secret key?



- Diffie-Hellman (DH) Key Exchange (1976)
  1. Alice and Bob agree on public parameters
     - **p**: a large "safe prime" s.t. (**p**-1)/2 is also prime
     - **g**: "generator", a small integer (e.g., 2)
  2. **Alice**
     Generates random secret value **a**.
     (0 < **a** < **p**)

     Bob
     Generates random secret value **b**.
     (0 < **b** < **p**)

     a Alice → $g^a \bmod p$ → **b** Bob

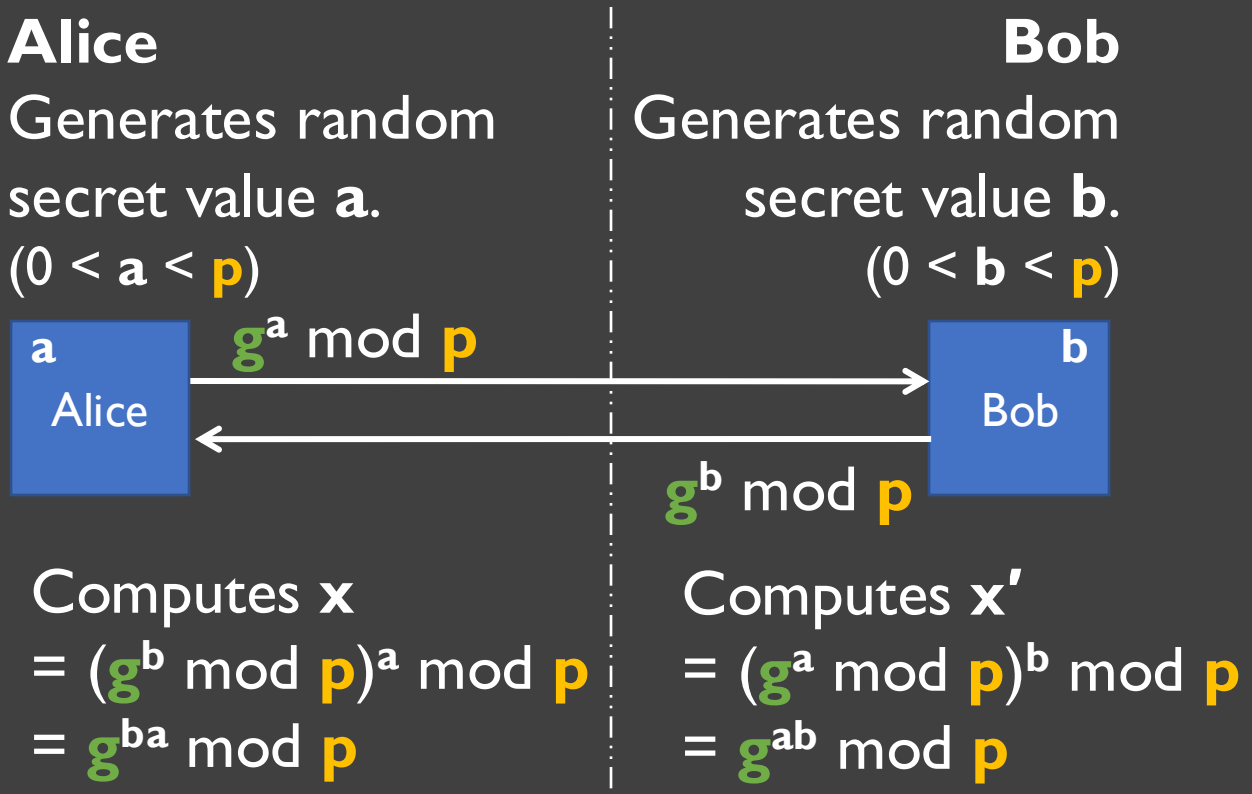
     b Bob → $g^b \bmod p$ → **a** Alice
  3. Computes **x**
     $= (g^b \bmod p)^a \bmod p$
     $= g^{ba} \bmod p$

     Computes **x′**
     $= (g^a \bmod p)^b \bmod p$
     $= g^{ab} \bmod p$

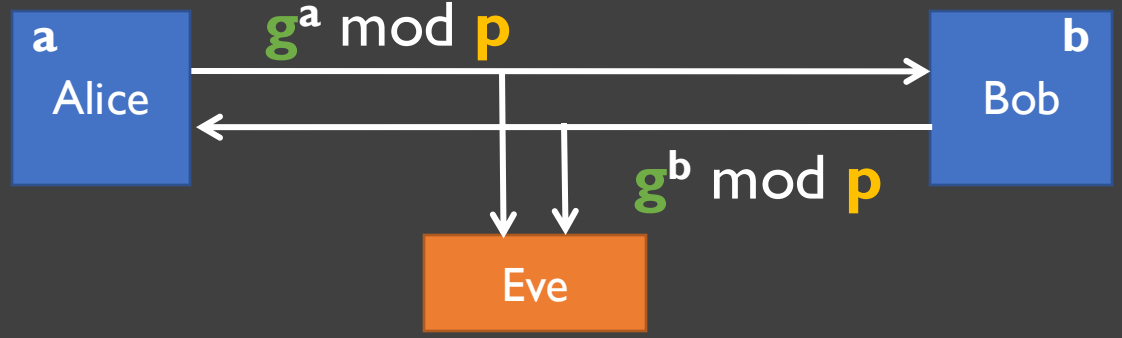
(Notice that **x** == **x′**)
Can use **k** := **h**(**x**) as a shared key!

DH: Example

- Non-secret values in **blue**, and secret values in **orange**:
- Alice and Bob agree to use a modulus $p = 23$ and base $g = 5$.
- Alice chooses a secret integer $a = 6$, then sends Bob $A = g^a \bmod p$
  - $A = 5^6 \bmod 23 = 8$
- Bob chooses a secret integer $b = 15$, then sends Alice $B = g^b \bmod p$
  - $B = 5^{15} \bmod 23 = 19$
- Alice computes $x = B^a \bmod p$
  - $x = 19^6 \bmod 23 = 2$
- Bob computes $x = A^b \bmod p$
  - $x = 8^{15} \bmod 23 = 2$
- Alice and Bob now share a secret (the number 2).



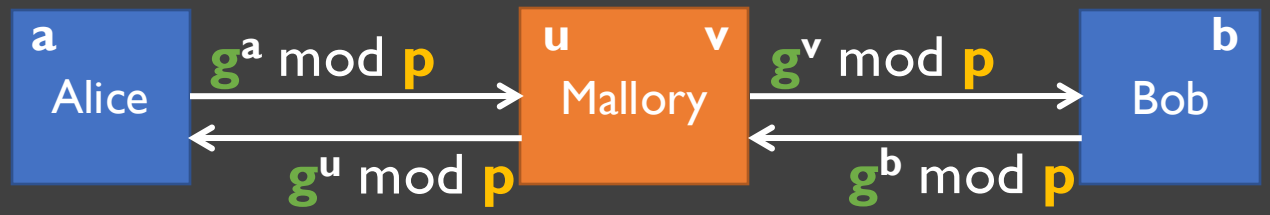
Passive eavesdropping attack against DH



Eve knows: p, g, g^a mod p, g^b mod p

- Eve wants to compute **x** = g^{ab} mod **p**
- Best known approach: Find **a** or **b**, then compute **x**
- Finding **y** given g^y mod **p** is an instance of the discrete log problem: No known efficient algorithm.

Man-in-the-middle (MITM) attack

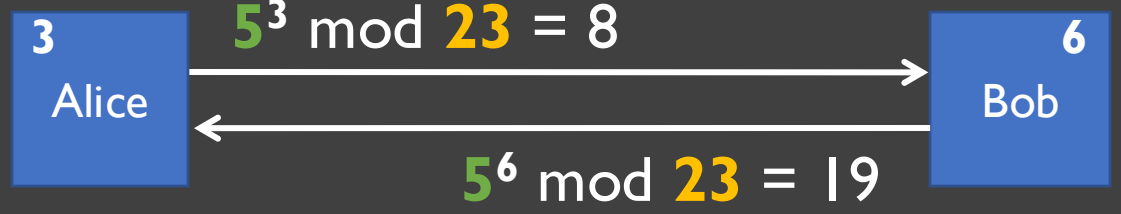


- Alice does DH exchange, really with Mallory, ends up with g^{au} mod p
- Bob does DH exchange, really with Mallory, ends up with g^{bv} mod p
- Alice and Bob each think they're talking with the other, but really Mallory is between them and knows both secrets!

DH gives you a shared secret, but you don't know who's on the other end ☹

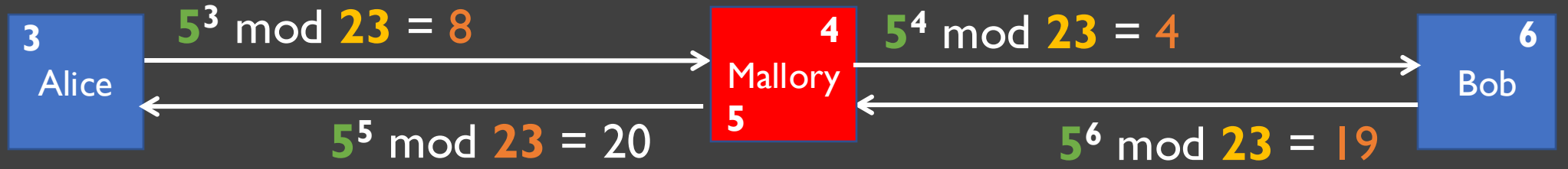
Part IV: Hands-On Challenge (15 minutes)

- Tools
  - mitmproxy
  - curl
  - netcat
- Alice (port 8081) and Bob (port 8082) want to perform (slightly modified) DH
- MITM attacker Mallory sits on port 8080 (running mitmproxy)
- Terminal 1 (Mallory)
  - mitmproxy --listen-port 8080
- Terminal 2 (Alice)
  - nc –l 8081
- Terminal 3 (Bob)
  - nc –l 8082



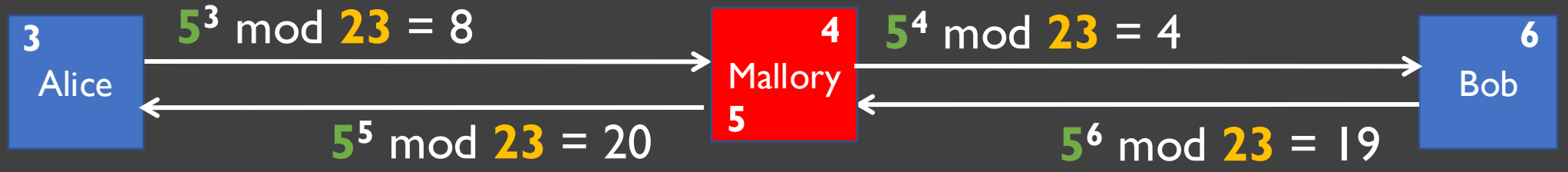
Part IV: Hands-On Challenge (15 minutes)

- Terminal 4 (Alice)
  - export http_proxy=http://localhost:8080
  - curl -X POST http://localhost:8081 -d "public_key=8" -H "Content-Type: application/x-www-form-urlencoded"
- Terminal 5 (Alice)
  - export http_proxy=http://localhost:8080
  - curl -X POST http://localhost:8082 -d "public_key=19&encrypted_flag=81,70,84" -H "Content-Type: application/x-www-form-urlencoded"
- **Task 1**: In mitmproxy, edit the keyshares
  - Mallory intercepts DH and spoofs protocol



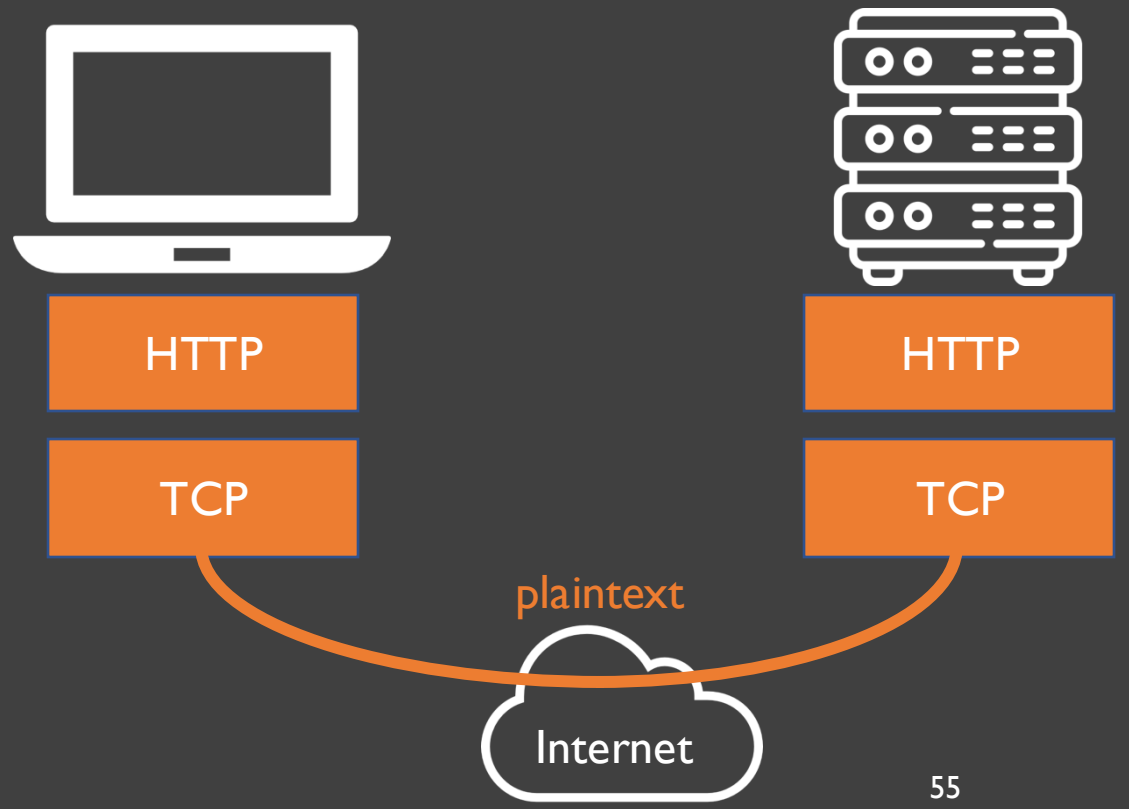
Part IV: Hands-On Challenge (15 minutes)

- Mallory has now access to encrypted flag
  - "(…) &encrypted_flag=81,70,84 (…)"
  - Can only be decrypted with shared secret
- **Task 2**: Calculated shared secret (see slide 47/48)
- **Task 3**: Use shared secret as password to byte-wise XOR with encrypted flag to decrypt
  - i.e., 81 ^ SECRET, 70 ^ SECRET, 84 ^ SECRET yields 3 numbers for the password
  - Use https://www.rapidtables.com/convert/number/hex-to-ascii.html to convert to ASCII



Why do we need TLS?

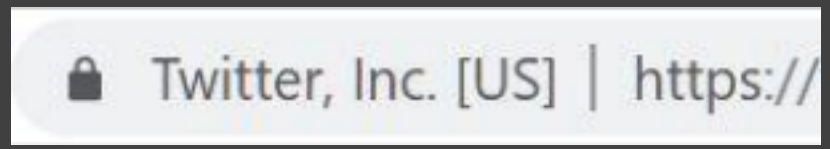
- Traditionally, HTTP (web), SMTP (email), and other applications were carried over the Internet using **TCP**, a **plaintext transport protocol**.
- TCP provides:
  - "Phone call"-like semantics:
    dial, send/receive data stream, hang up
- TCP doesn't provide:
  - Confidentiality
  - Integrity
  - Authentication



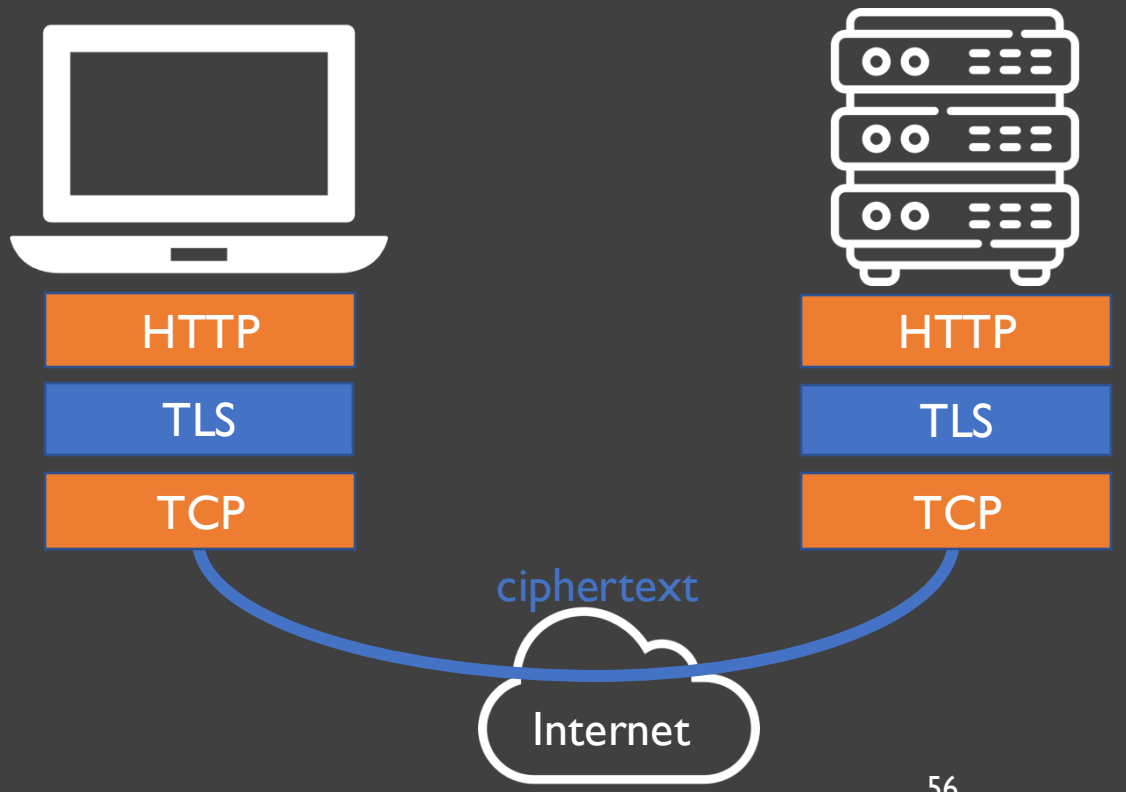
Transport Layer Security (TLS)

- **TLS** (Transport Layer Security) is a cryptographic protocol that is layered above TCP to provide a secure channel.
- Commonly used with many application protocols:

  HTTP over TLS ⇨ HTTPS

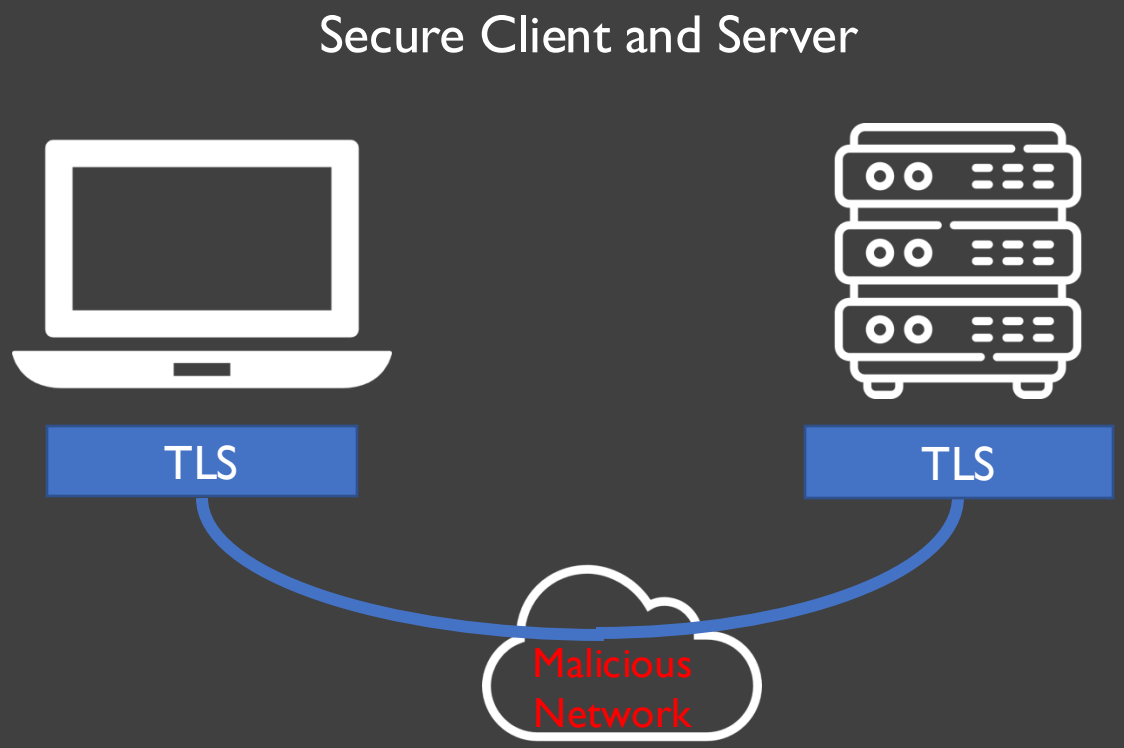
  

  SMTP, FTP, XMPP, OpenVPN, and others



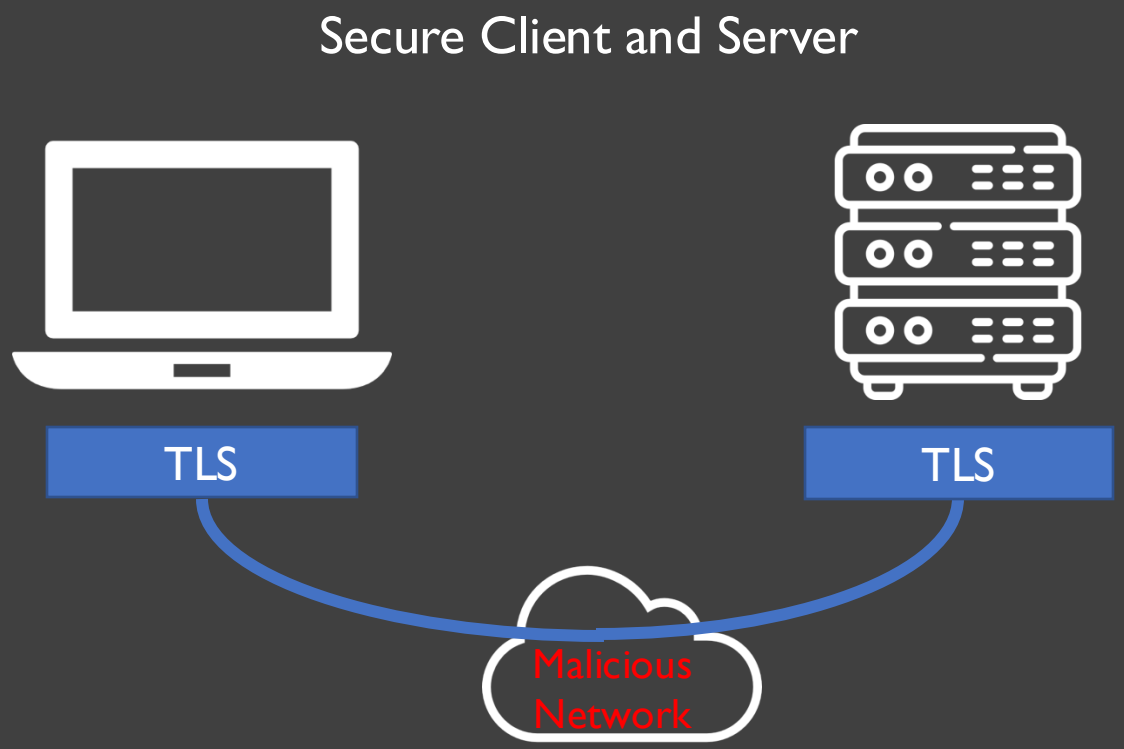
Don't roll your own crypto. If your program sends data over the Internet, use TLS!

TLS Threat Model: Malicious Networks



- TLS assumes client and server are secure but talking over a malicious network with active and passive attackers.
- **All Internet traffic faces these threats.**
  Examples:
  - Government surveillance and censorship
  - ISPs harvesting data for tracking, injecting ads
  - Compromised routers, Wi-Fi APs, DNS servers
  - ARP spoofing, BGP route hijacking

TLS Benefits and Limitations



- TLS provides:
  - **Confidentiality** and **integrity** protection for application data while in transit using symmetric cryptography. **How do we obtain secret key? TLS Handshake!**
  - Client can **authenticate** server's identity. **Why is this important?**
- TLS does not protect against:
  - Malware/intruder on the client/server
  - Phishing, social engineering
  - Tracking by the sites you visit
  - Denial-of-service

Certificates

- **How does a client obtain server's public key?**
- Server presents a **certificate** that includes that public key belongs to the site (e.g., amazon.com)
  - Signed by a **certificate authority (CA)** using a digital signature
- CA is an entity trusted by clients to verify server identities and issue certificates
- Your browsers (e.g., Firefox, Chrome) trust a specific set of CAs as **root CAs**
  - Shipped with the public keys of the root CAs
  - Why do we need more than one?

X.509 Certificates

- A trusted authority vouches that certain public key belongs to a particular site
  - Format called X.509
- Browsers ship with CA public keys for large number of trusted CAs (accreditation process)
- Important fields:
  - Common Name (CN)
  - Expiration Date
  - Subject's Public Key
  - Issuer -- e.g., Verisign
  - Issuer's signature
- Common Name field
  - Explicit name, e.g. computing.clemson.edu
  - Or wildcard, e.g. *.clemson.edu

Subject: C=US/O=Google Inc/CN=www.google.com
Issuer: C=US/O=Google Inc/CN=Google Internet Authority
Serial Number:
01:b1:04:17:be:22:48:b4:8e:1e:8b:a0:73:c9:ac:83
Expiration Period: Jul 12 2010 - Jul 19 2012
Public Key Algorithm: rsaEncryption
Public Key:
43:1d:53:2e:09:ef:dc:50:54:0a:fb:9a:f0:fa:14:58:ad:a0:81:b0:3d
7c:be:b1:82:19:b9:7c3:8:04:e9:1e5d:b5:80:af:d4:a0:81:b0:b0:6
8:5b:a4:a4
:ff:b5:8a:3a:a2:29:e2:6c:7c3:8:04:e9:1e5d:b5:7c3:8:04:e9:39:23
:46

Signature Algorithm: sha1WithRSAEncryption

Signature:
39:10:83:2e:09:ef:ac:50:04:0a:fb:9a:f0:fa:14:58:ad:a0:81:b0:3d
7c:be:b1:82:19:b9:7c3:8:04:e9:1e5d:b5:80:af:d4:a0:81:b0:b0:68
:5b:a4:a4
:ff:b5:8a:3a:a2:29:e2:6c:7c3:8:04:e9:1e5d:b5:7c3:8:04:e9:1e:5d:
b5

Obtaining a Certificate

How does Alice (web browser) obtain K_{Bob}?

Browser (Alice)

(Knows K_{CA})

Server (Bob)

1. Choose (K_{Bob},K_{Bob}^{-1})

--- K_{Bob} and proof he is "Bob" -->

<-- Signs certificate with K_{CA}^{-1} ----
"Bob's key is K_{Bob} -- Signed, CA"

3. Keeps cert on file

Certificate Authority (CA)
[Think of like a notary]

(Keeps K_{CA}^{-1} Secret)

2. Checks proof (cert verification)

Certificate Issuance (before serving users)

Certificate Use (in TLS)

<-- Sends cert to Alice ----
"Bob's key is K_{Bob} -- Signed, CA"

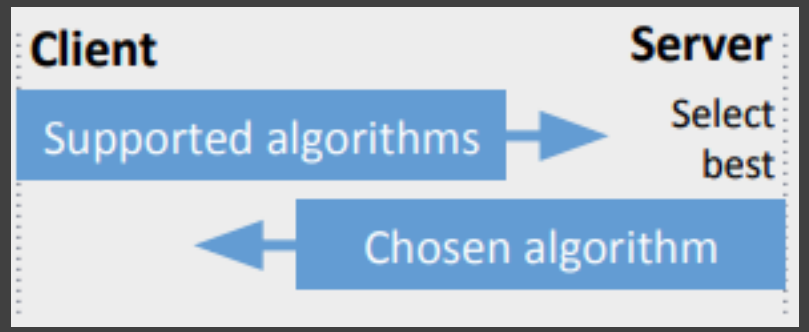
4. Verifies signature on cert using K_{CA}, learns correct K_{Bob}

Certificate Verification

- To obtain a cert, server must **prove its identity** to the CA.
- Common verification methods:
  - **Email**: Receive confirmation code at an email address specified when the domain was registered.
  - **HTTP**: Place file with confirmation code at specified URL on domain.
  - **DNS**: Add record containing confirmation code to domain's DNS zone
- Cert has expiration date (e.g., one year ahead)
  - After expiration: Certificate invalidation

TLS 1.3 Handshake

Negotiate Crypto Algorithms



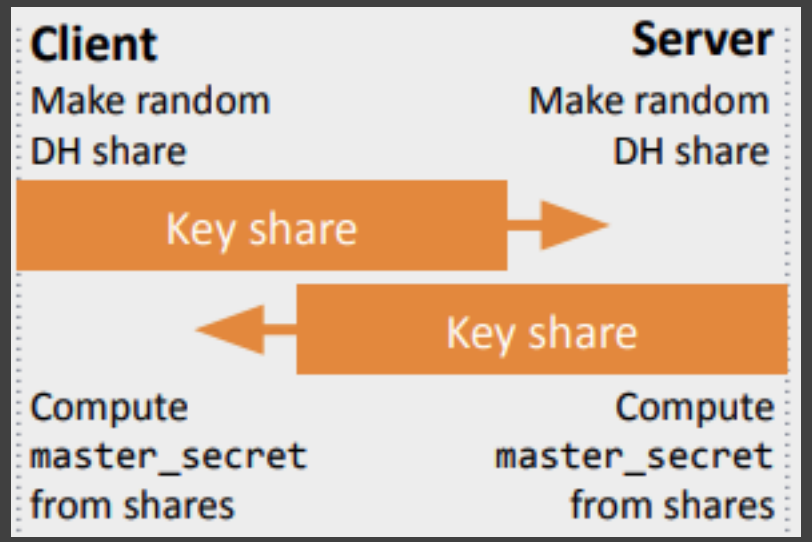
Find best mutually supported:

Key exchange algorithm
e.g., EC DHE w/ Curve25519

Signature algorithm
e.g., RSA w/ SHA-256, PKCS #1 pad

Symmetric crypto algorithm
e.g., AES-128 GCM

Establish Shared Secret

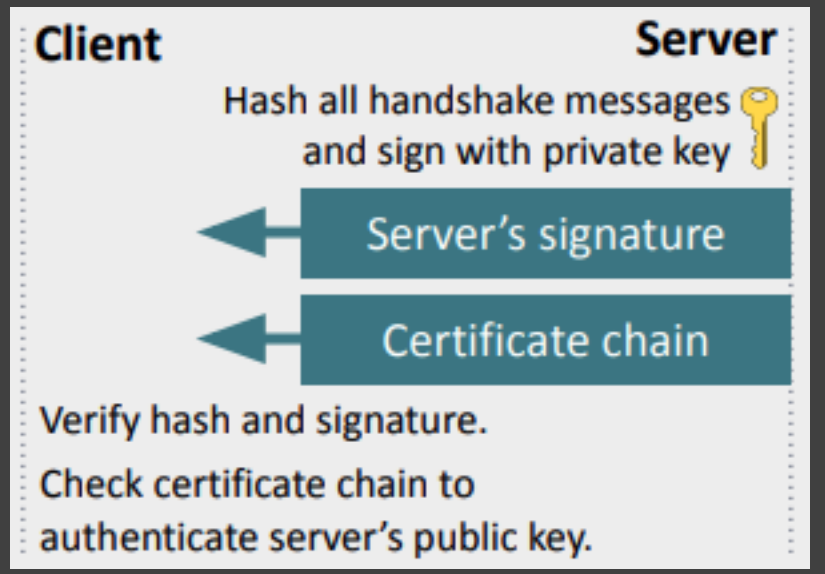


Mutually compute a **shared secret**.

Diffie-Hellman for **forward secrecy**.

Derive symmetric keys from shared secret and encrypt and integrity check all further data.

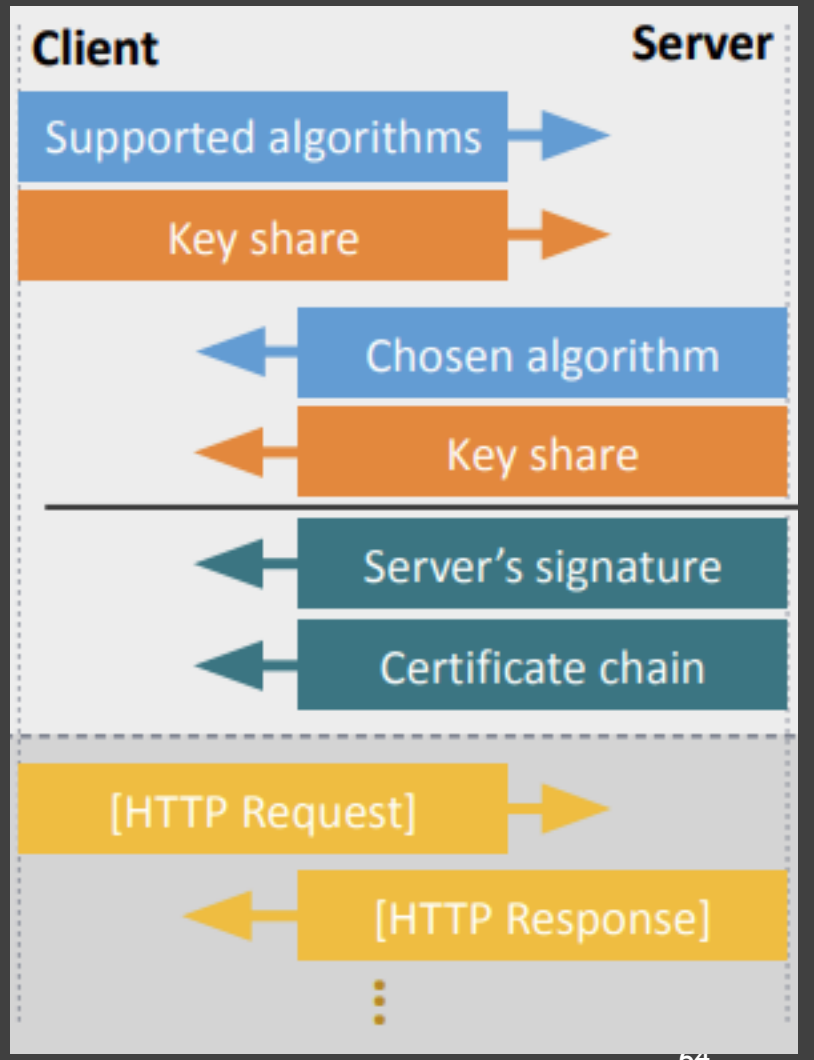
Authenticate Server



Server signs, and client verifies (using server's public key from cert), a hash of the entire **handshake transcript** to this point.

TLS 1.3

- TLS handshake is NOT vulnerable to MITM attacks
  - Server authentication
  - Key share exchange messages are signed
  - Does not mean TLS cannot be attacked ☹
- After handshake is completed, all HTTP traffic becomes HTTPS traffic
  - Encrypted and authenticated
- Also used for in-vehicle traffic (Automotive Ethernet)
  - Becoming increasingly common in new vehicles



Summary

- Cryptography is used in all aspects of security (networks, systems, etc.)
- Three major security principles:
  - Integrity
  - Confidentiality
  - Availability
- Hashing is a building block for integrity and confidentiality algorithms
- Symmetric vs asymmetric crypto
- Diffie Hellman Key Exchange algorithm allows agreeing on a shared secret key on insecure channel
  - Essential in TLS

Thank You!

Mert D. Pesé, Ph.D.

TigerSec Laboratory
@ Clemson University

mpese@clemson.edu
www.linkedin.com/in/mertpese

Further Reading

Security Engineering by Ross Anderson
https://www.cl.cam.ac.uk/archive/rja14/Papers/SEv3.pdf