Network Security

Introduction

Chapter 1

Learning Objective

- Introduce the security requirements
  - confidentiality
  - integrity
  - availability
- Describe the X.800 security architecture for OSI

Network Security Requirements

Computer Network Security

- Definition: The protection afforded to an automated information system in order to attain the application objectives to preserving the integrity, availability, and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications).

  - NIST Computer Security Handbook

Confidentiality

- **Data confidentiality:** Assures that private or confidential information is not made available or disclosed to unauthorized individuals;
- **Privacy:** Assures that individual's control or influence what information related to them may be collected and stored and by whom and to whom that information may be disclosed
- i.e., student grade information

Integrity

- **Data integrity:** Assures that data (both stored and in transmitted packets) and programs are changed only in a specified and authorized manner;
- **System integrity:** Assures that a system performs its intended function in an unimpaired manner, free from deliberate or inadvertent unauthorized manipulation of the system
- i.e., a hospital patient's allergy information

Availability

- **Availability:** Assures that systems work promptly, and service is not denied to authorized users, ensuring timely and reliable access to and use of information
- i.e., denial of service attack

Other security requirements

- **Authenticity**
- **Accountability**
  - tracible data source,
  - fault isolation
  - intrusion detection and prevention,
  - recovery and legal action
  - system must keep records of their activities to permit later forensic analysis to trace security breaches or to aid in transaction disputes

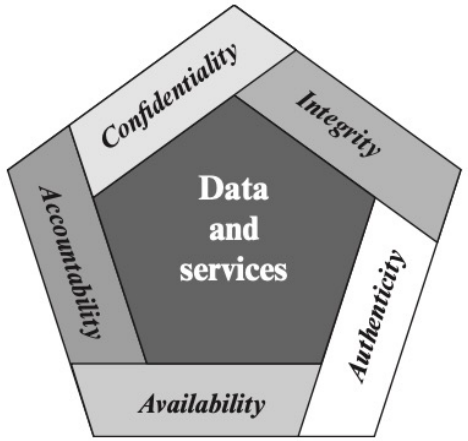
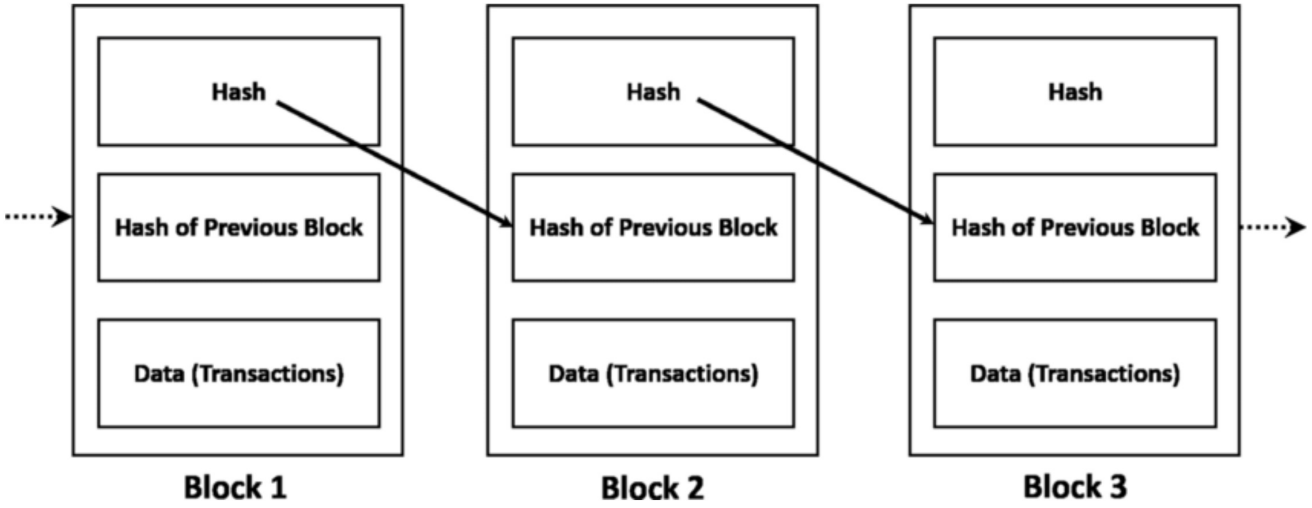


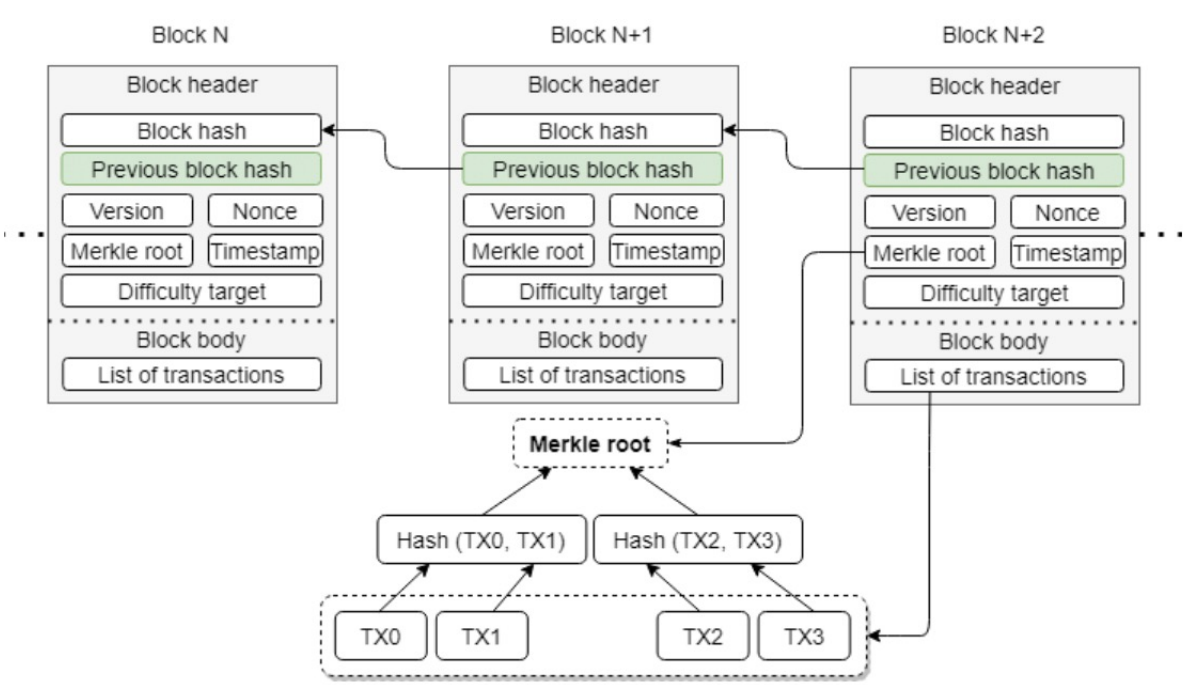
Figure 1.1 Essential Network and Computer Security Requirements

Question

- What security requirements does a blockchain system have achieved?



A Hyperledger



Project

- **Task1: OnDemand Professor Q&A Bot**
  - Your task is to build a Q&A Bot over private data that answers questions about the network security course using the open-source alternatives to ChatGPT that can be run on your local machine. Data privacy can be compromised when sending data over the internet, so it is mandatory to keep it on your local system.
  - Your Q&A Bot should be able to understand user questions and provide appropriate answers from the local database, then the citations should be added **(must be accomplished)** if the response is from the internet, then the web references should be added.
  - Train your bot using network security lecture slides, network security textbook, and the Internet.
  - By using Wireshark capture data for Step 4's of the LLM workflow shown in Figure 1. Provide detailed explanations of the trace data. Also, Maintain a record of Step 1's prompt and its mapping to the trace data in Step 4's.
- **Task2: Quiz Bot**
  - Your task is to build a quiz bot based on a network security course using the open-source alternatives to ChatGPT that can be run on your local machine. Data privacy can be compromised when sending data over the internet, so it is mandatory to keep it on your local system.
  - Two types of questions should be offered by the bot: randomly generated questions and specific topic questions and the answers should be pulled from the network security database. Train your bot using network security quizzes, lecture slides, network security textbook, and the Internet.
  - The quiz must include multiple-choice questions, true/false questions, and open-ended questions.
  - Finally, the bot should be able to provide feedback on the user's answers.

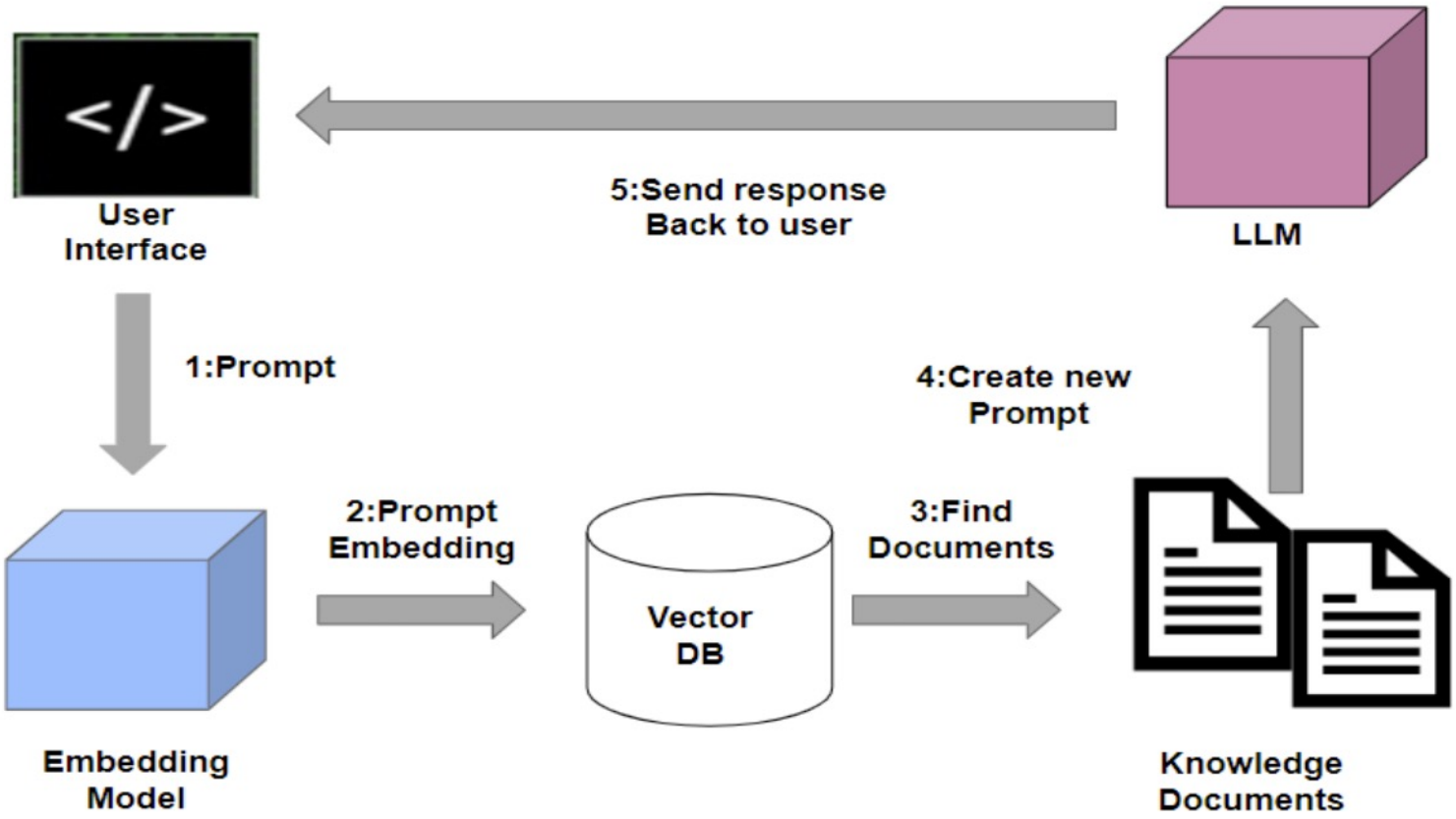


Figure 1: LLM workflow

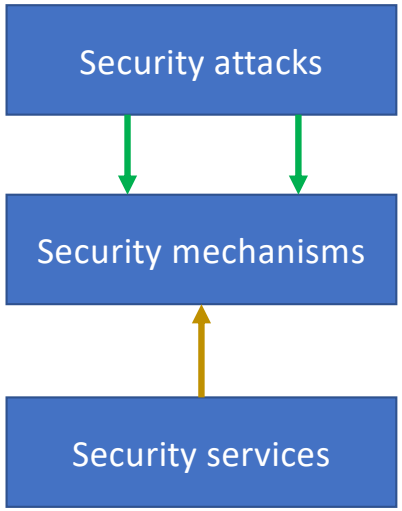
Outline

- Review
- OSI Security Architecture
  - Attack model

OSI Security Architecture

OSI Security Architecture

- International Telecommunication Union – Telecommunication (ITU-T) recommends X.800
- Security Architecture for Open Systems Interconnection (OSI)
  - Defines a systematic way of defining and providing security requirements
  - Used by IT managers and vendors in their products



a process (or a device incorporating such a process) to detect, prevent, or recover from an attack

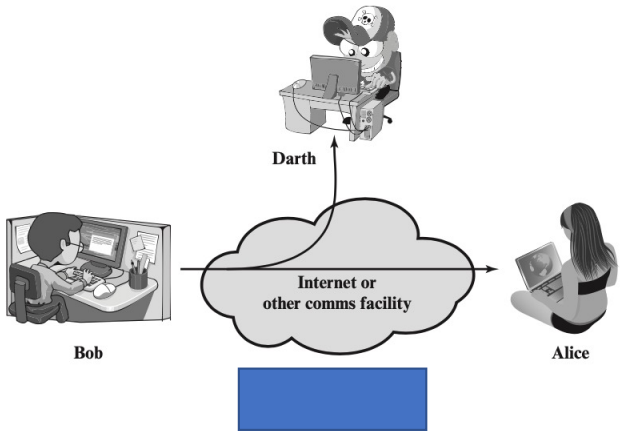
enhances the security of the data processing systems and the information transfers, such as policies

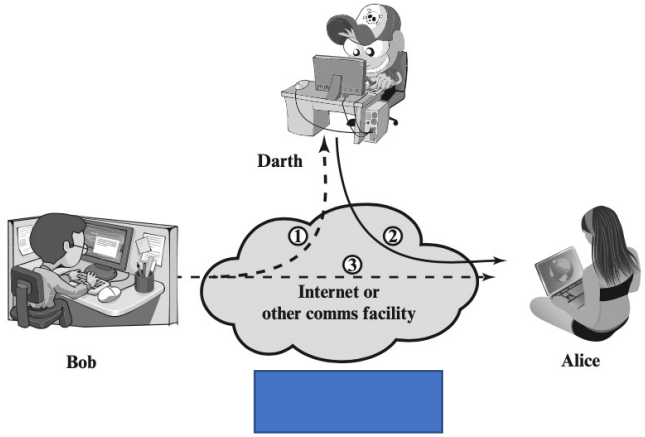
Other Security Architectures

- NIST, Cybersecurity Framework (CSF)
  - https://www.nist.gov/cyberframework
  - VIRTUAL WORKSHOP #2 | February 15, 2023 (9:00 AM – 5:30 PM EST). Discuss potential significant updates to the CSF
  - https://www.nist.gov/news-events/events/2023/02/journey-nist-cybersecurity-framework-csf-20-workshop-2
- OWASP - Open Web Application Security Project
  - Web application security
  - OWASP Application Security Verification Standard (ASVS) - https://owasp.org/www-project-application-security-verification-standard/
  - OWASP Web Security Testing - https://owasp.org/www-project-web-security-testing-guide/
  - OWASP foundation

Security attack

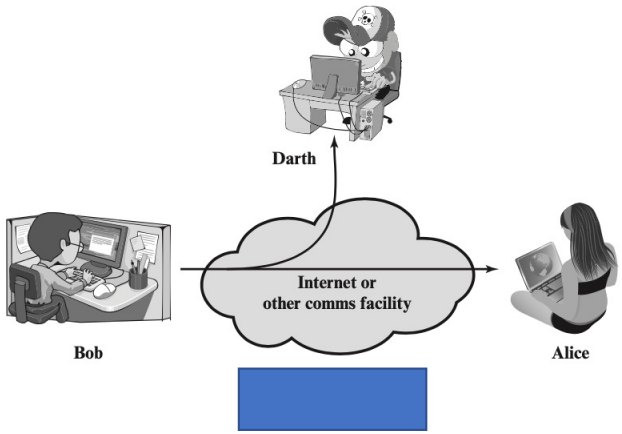
- **Definition**: any action that compromises the security of information owned by an organization
- Two types of security attacks
  - Passive attack
  - Active attack

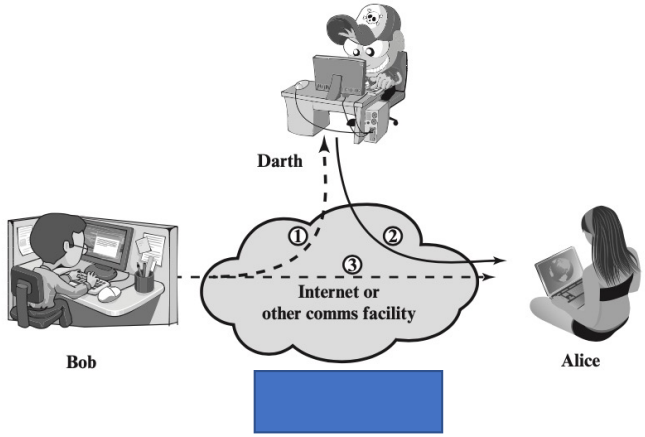




Security attack

- **Definition**: any action that compromises the security of information owned by an organization
- Two types of security attacks
  - Passive attack
  - Active attack





Passive attack

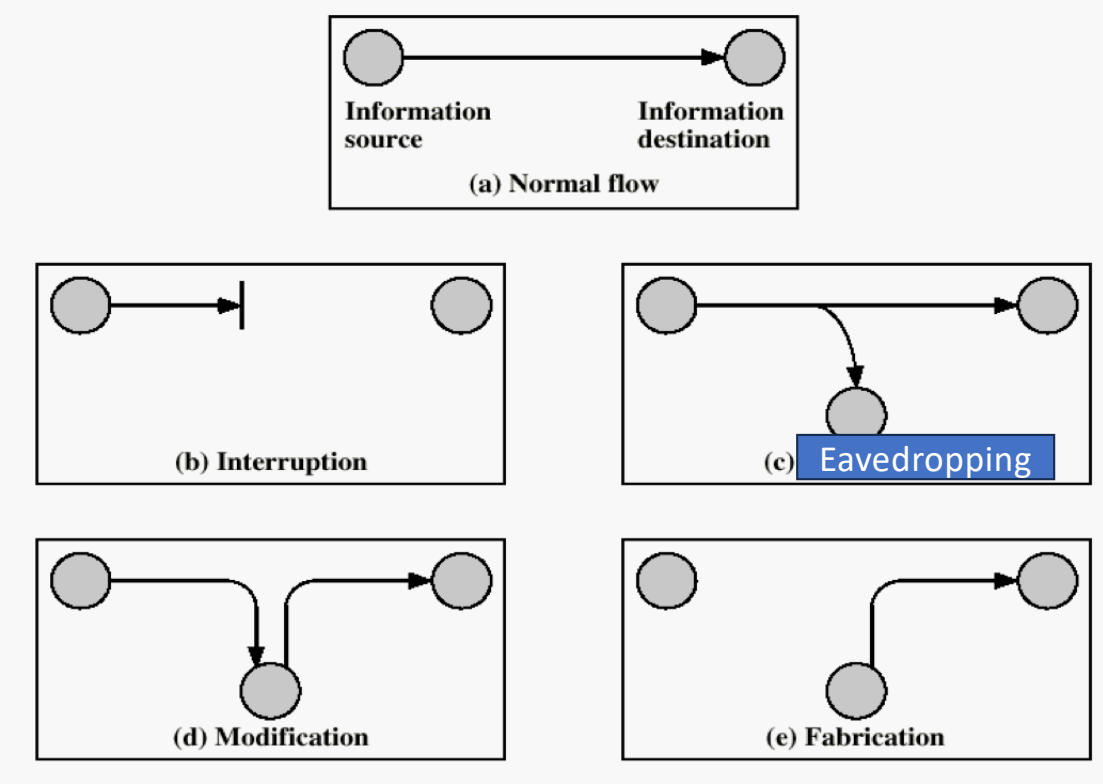
- i.e. eavesdropping on or monitoring of transmissions
- Goal: obtain information being transmitted
  - release of message contents
  - traffic analysis – a promiscuous sniffer
- Very difficult to detect – no alteration of the data
- But easy to prevent, why?

Active attack

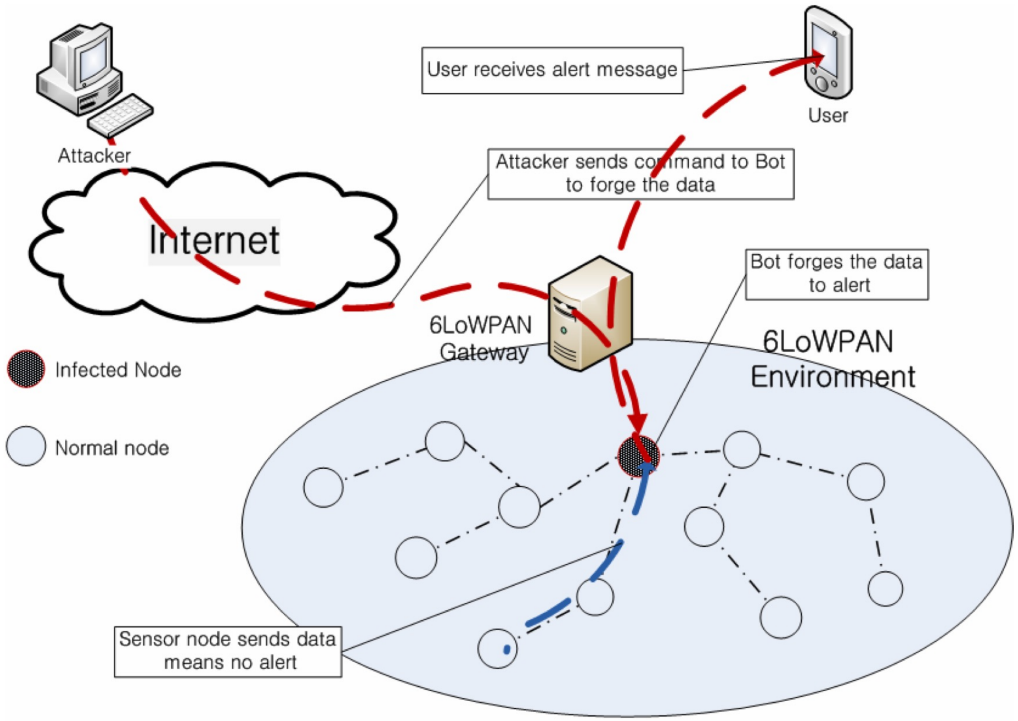
- active attack includes:
  - replay
  - Modification of messages
  - Denial of service
  - Masquerade

Example: two points communication

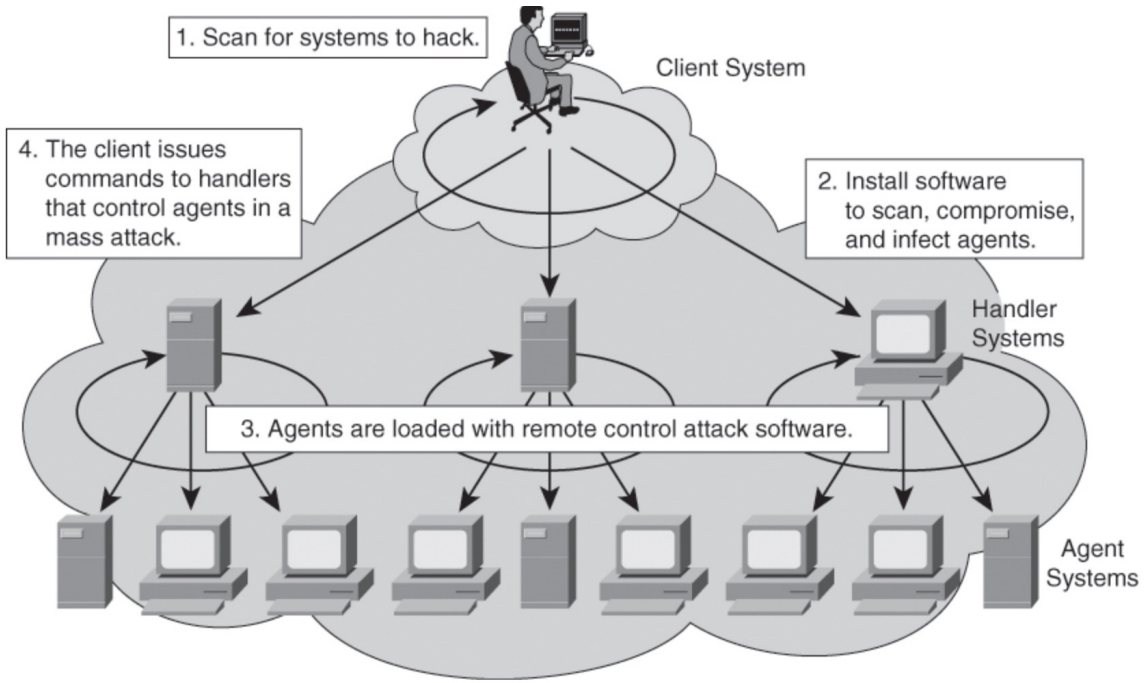
- Generic types of attacks



Example of modification attack in 6LoWPAN



Example: a group of attackers

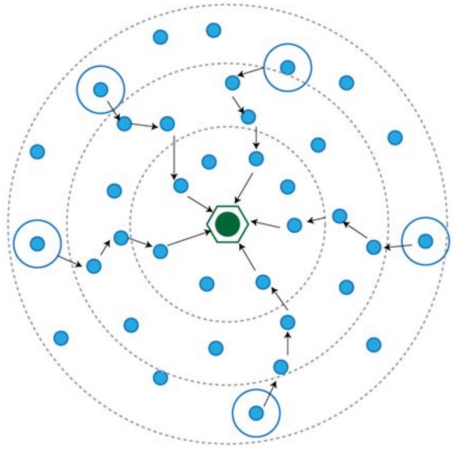


Know Your Threat Model

- **Threat model:** A model of who your attacker is and what resources they have
- One of the best ways to counter an attacker is to attack their reasons

Example: adversary model

- “The adversary is assumed to be intelligent and has limited number of resources. Before capturing the nodes, it exploits the various vulnerabilities of the networks. It knows the topology of the network, routing information. It aims to capture the sink node so as to disrupt the whole traffic. If it is not able to capture the sink node, it will capture the nearby nodes of the sink. It tries to disrupt the whole traffic of the network with minimum number of captured nodes. It is also assumed that the adversary tends to attack more on the nodes closer to the data sink than nodes that are far away”



No class on Wednesday

- No class on Wednesday (Sept 18, 2024) due to the Job fair. Wish you good luck!
- Reminder to form a project group by Sept. 9th, 2024

Know Your Threat Model

- **Threat model:** A model of who your attacker is and what resources they have
- One of the best ways to counter an attacker is to attack their reasons

Story…

- The bear race
- **Takeaway:** Even if a defense is not perfect, it is important to always stay on top of best security measures

I don’t have to outrun the bear. I just have to outrun you

Human Factors

- The users
  - Users like convenience (ease of use)
  - If a security system is unusable, it will be unused
  - Users will find way to subvert security systems if it makes their lives easier
- The programmers
  - Programmers make mistakes
  - Programmers use tools that allow them to make mistakes (e.g. C and C++)
- Everyone else
  - Social engineering attacks exploit other people's trust and access for personal gain

Design in security from the start

- When building a new system, include security as part of the design considerations rather than patching it after the fact
  - A lot of systems today were not designed with security from the start, resulting in patches that don’t fully fix the problem!
- Keep these security principles in mind whenever you write code!

Security Services and Mechanisms

TABLE 1/X.800

Illustration of relationship of security services and mechanisms

Service \ Mechanism	Encipherment	Digital signature	Acces control	Data integrity	Authenti-cation exchange	Traffic padding	Routing control	Notari-zation
Peer entity authentication	Y	Y	·	·	Y	·	·	·
Data origin authentication	Y	Y	·	·	·	·	·	·
Access control service	·	·	Y	·	·	·	·	·
Connection confidentiality	Y	·	·	·	·	·	Y	·
Connectionless confidentiality	Y	·	·	·	·	·	Y	·
Selective field confidentiality	Y	·	·	·	·	·	·	·
Traffic flow confidentiality	Y	·	·	·	·	Y	Y	·
Connection Integrity with recovery	Y	·	·	Y	·	·	·	·
Connection integrity without recovery	Y	·	·	Y	·	·	·	·
Selective field connection integrity	Y	·	·	Y	·	·	·	·
Connectionless integrity	Y	Y	·	Y	·	·	·	·
Selective field connectionless integrity	Y	Y	·	Y	·	·	·	·
Non-repudiation. Origin	·	Y	·	Y	·	·	·	Y
Non-repudiation. Delivery	·	Y	·	Y	·	·	·	Y

· The mechanism is considered not to be appropriate.

Y Yes: the mechanism is considered to be appropiate, either on its own or in combination with other mechanisms.

Note – In some instances, the mechanism provides more than is necessary for the relevant service but could nevertheless be used.

Supplementary materials

- Internet Security Glossary, v2 – produced by Internet Society (ISOC) https://datatracker.ietf.org/doc/html/rfc4949
- X.800 – OSI network security

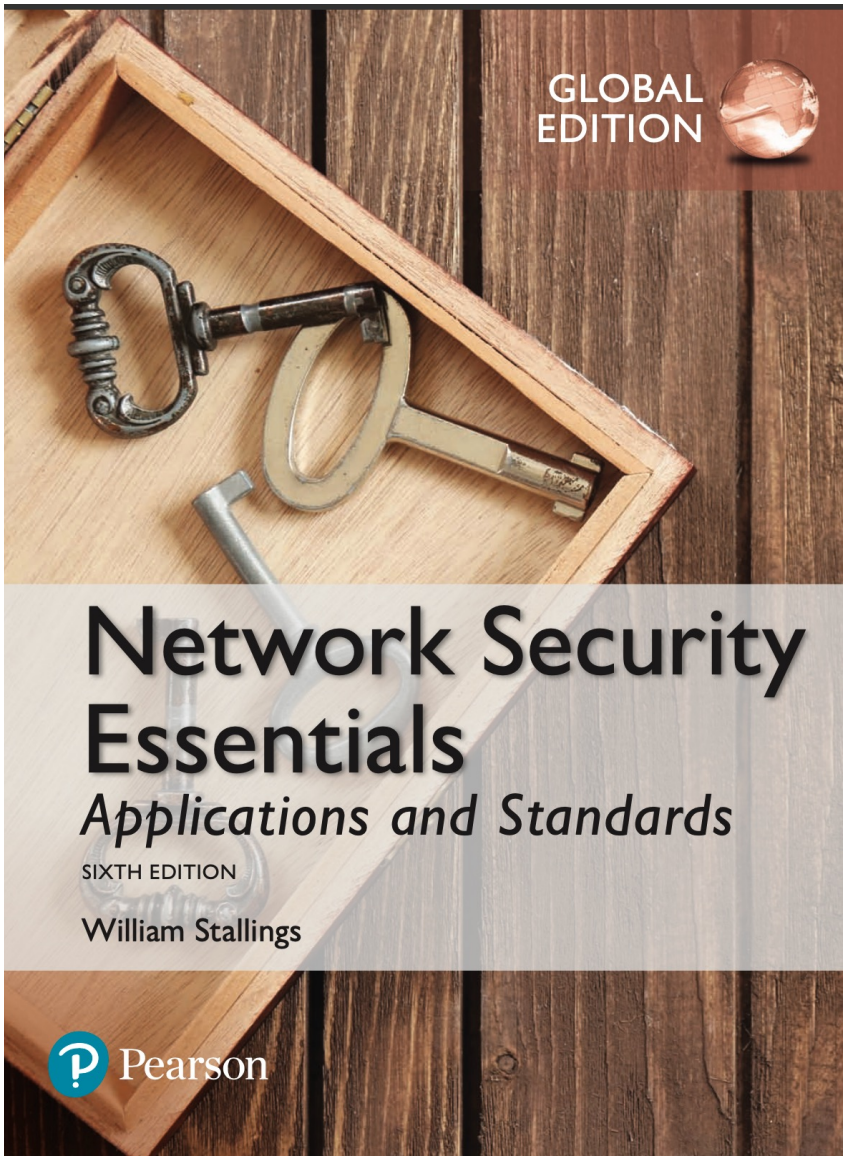
  https://www.itu.int/rec/dologin_pub.asp?lang=f&id=T-REC-X.800-199103-I!!PDF-E&type=items

Summary for Chapter 1

- Have learned:
  - Security requirements
  - Attack models
  - X.800 secure architecture, security services, mechanisms

Review Questions

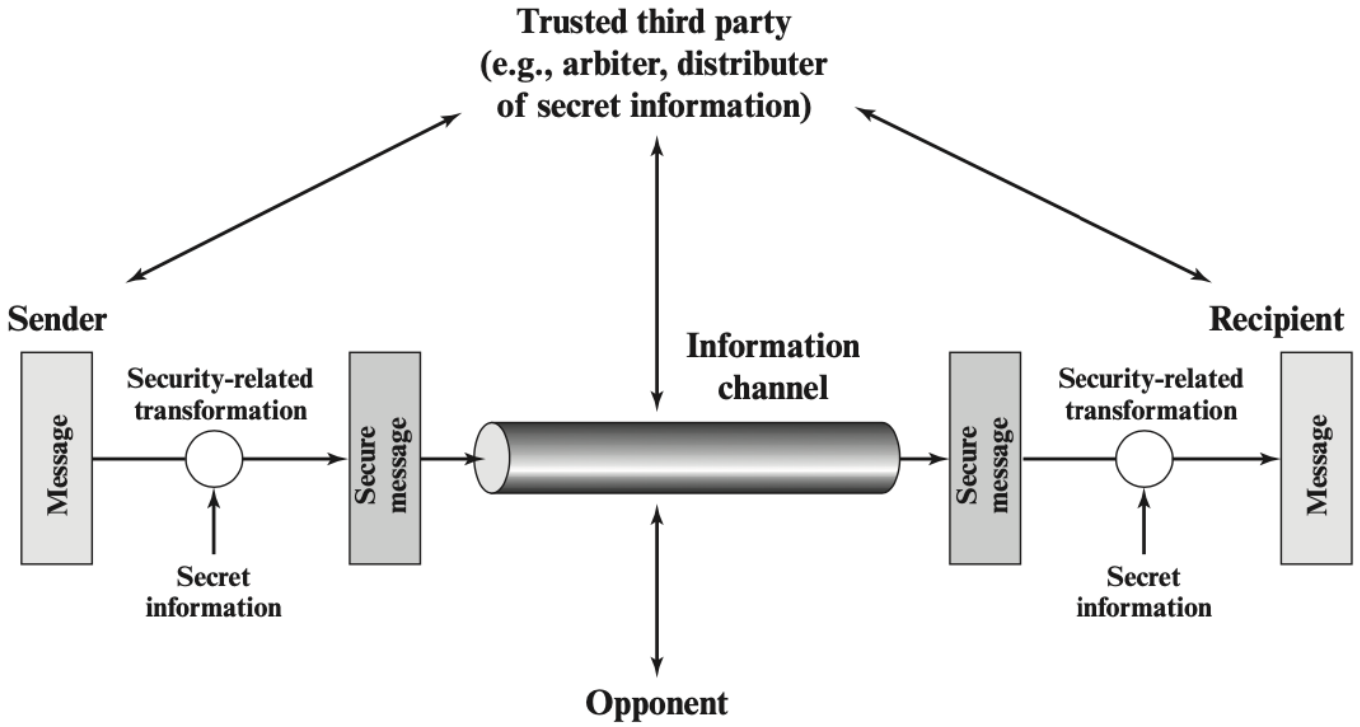
- William Stallings (WS), “Network Security Essentials”, 6th Global Edition
- RQ 1.1 - 1.3
- Prob 1.5



Symmetric Encryption and Message Confidentiality

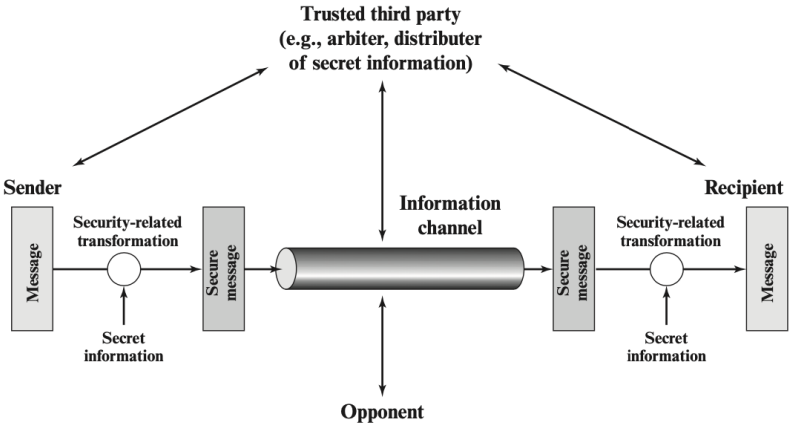
Chapter 2

Model for network security



Model for network security

- Using this model requires us to:
  - design a suitable algorithm for the security transformation
  - generate the secret information (keys) used by the algorithm
  - develop methods to distribute and share the secret information
  - specify a protocol enabling the principals to use the transformation and secret information for a security service

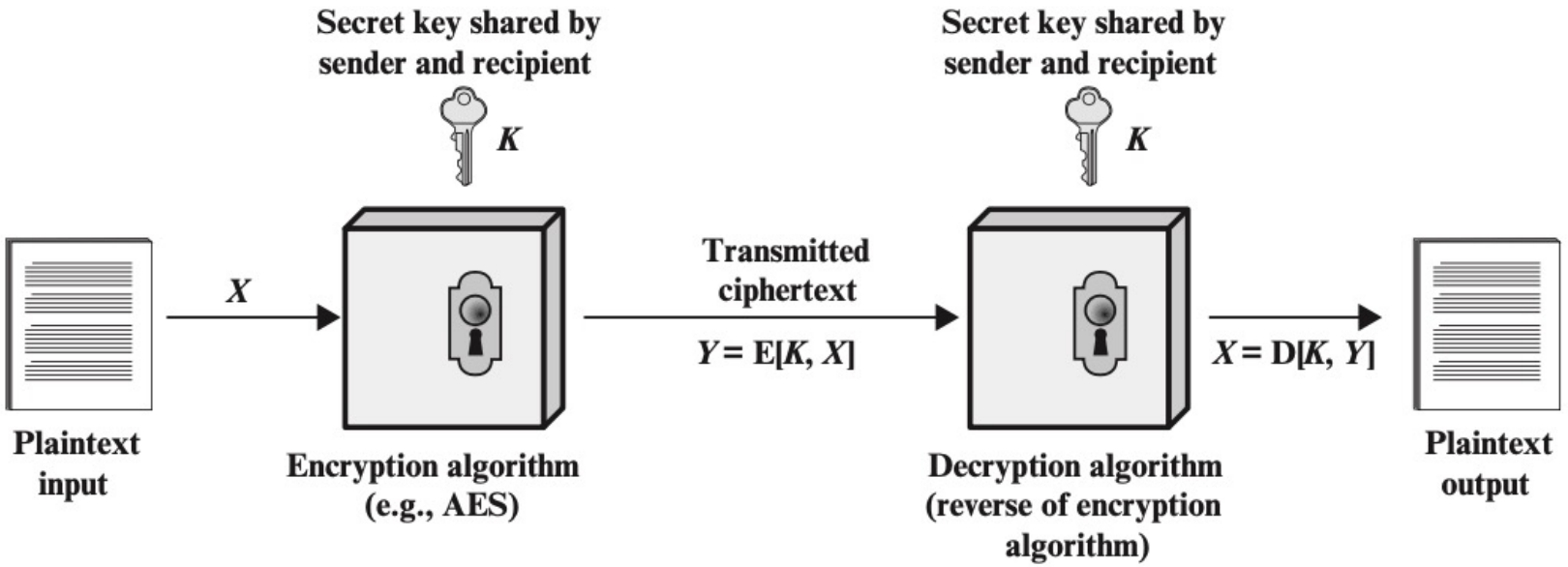


Symmetric Encryption Principles

Symmetric encryption

- Sender and recipient share a common/same key
- Was the only type of cryptography, prior to invention of public-key in 1970’s

Simplified model of symmetric encryption



Symmetric encryption

- Has five ingredients
  - **Plaintext**: the original message or data
  - **Encryption algorithm**: performs various substitutions and transformations on the plaintext
  - **Secret key**
  - **Ciphertext**: the coded message
  - **Decryption algorithm**: takes the ciphertext and the same secret key and produces the original plaintext

Other basic terminology

- **cipher** - algorithm for transforming plaintext to ciphertext
- **encipher (encrypt)** - converting plaintext to ciphertext
- **decipher (decrypt)** - recovering plaintext from ciphertext
- **cryptography** - study of encryption principles/methods
- **cryptanalysis (codebreaking)** - the study of principles/ methods of deciphering ciphertext *without* knowing key

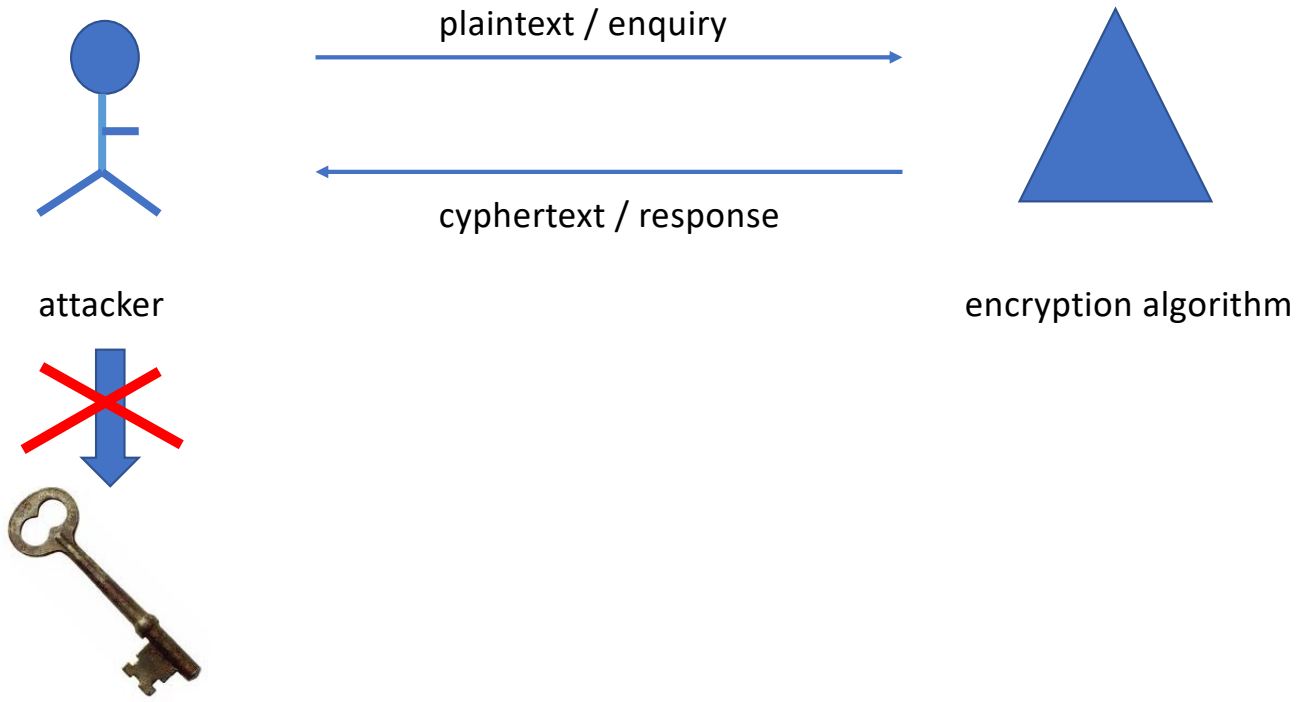
Requirements

- Two requirements for secure use of symmetric encryption:
  - a strong encryption algorithm
  - a secret key known only to sender / receiver

  $Y = E_K(X)$

  $X = D_K(Y)$
- assume encryption algorithm is known
- the security of symmetric encryption depends on the secrecy of the key
- implies a secure channel to distribute key

A strong encryption algorithm



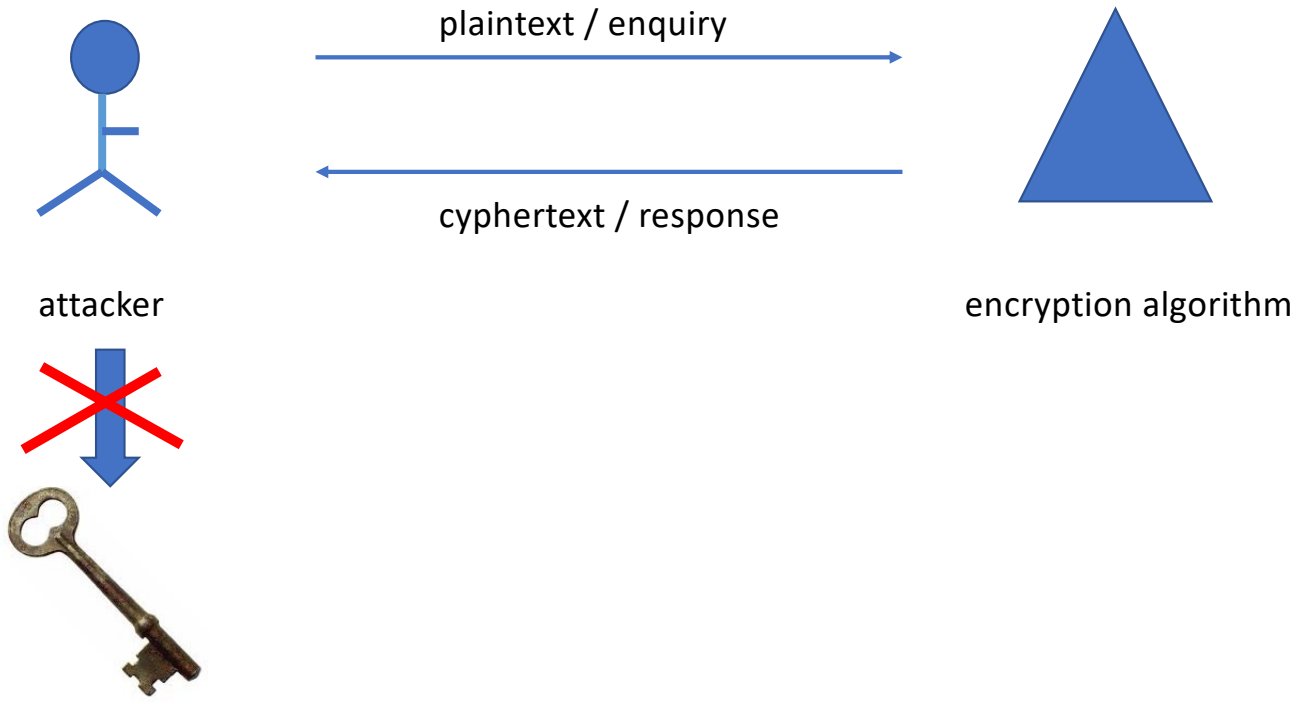
TA & Grader

- TA Name: Faiyaz, Amir (Project, Review & Quiz)
- Email: afaiyaz@ttu.edu
- Reminder: Submit the names and emails of your group members to

  FALL 2024 CS5342 PROJECT GROUP NAMES.xlsx

- Grader Name: Han, Namgyu (Homework, Quiz, Exam grading)
- Email: Namgyu.Han@ttu.edu

A strong encryption algorithm



Secure Encryption Scheme

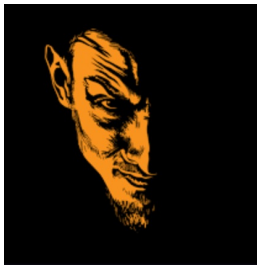
- **Unconditional security**
  - no matter how much computer power is available, the cipher cannot be broken since the ciphertext provides insufficient information to uniquely determine the corresponding plaintext
- **Computational security**
  - the cost of breaking the cipher exceeds the value of the encrypted information;
  - or the time required to break the cipher exceeds the useful lifetime of the information

Desired characteristics

- Cipher needs to completely obscure statistical properties of original message
- more practically Shannon suggested combining elements to obtain:
  - Confusion – how does changing a bit of the key affect the ciphertext?
  - Diffusion – how does changing one bit of the plaintext affect the ciphertext?

confusion

ciphertext



diffusion

plaintext

Ways to achieve

- Symmetric Encryption:
  - substitution / transposition / hybrid
- Asymmetric Encryption:
  - Mathematical hardness - problems that are efficient to compute in one direction, but inefficient to reverse by the attacker
    - Examples: Modular arithmetic, factoring, discrete logarithm problem, Elliptic Logs over Elliptic Curves

Two basic types

- Block Ciphers
  - Typically 64, 128 bit blocks
  - A k-bit plaintext block maps to a k-bit ciphertext block
  - Usually employ Feistel structure
- Stream Ciphers
  - A key is used to generate a stream of pseudo-random bits – key stream
  - Just XOR plaintext bits with the key stream for encryption
  - For decryption generate the key stream and XOR with the ciphertext!

Symmetric Block Encryption

Block cipher

- the most commonly used symmetric encryption algorithms
- input: fixed-size blocks (Typically 64, 128 bit blocks), output: equal size blocks
- provide secrecy and/or authentication services
- Data Encryption Standard (DES), triple DES (3DES), and the Advanced Encryption Standard (AES)s
- Usually employ Feistel structure

Feistel Cipher Structure

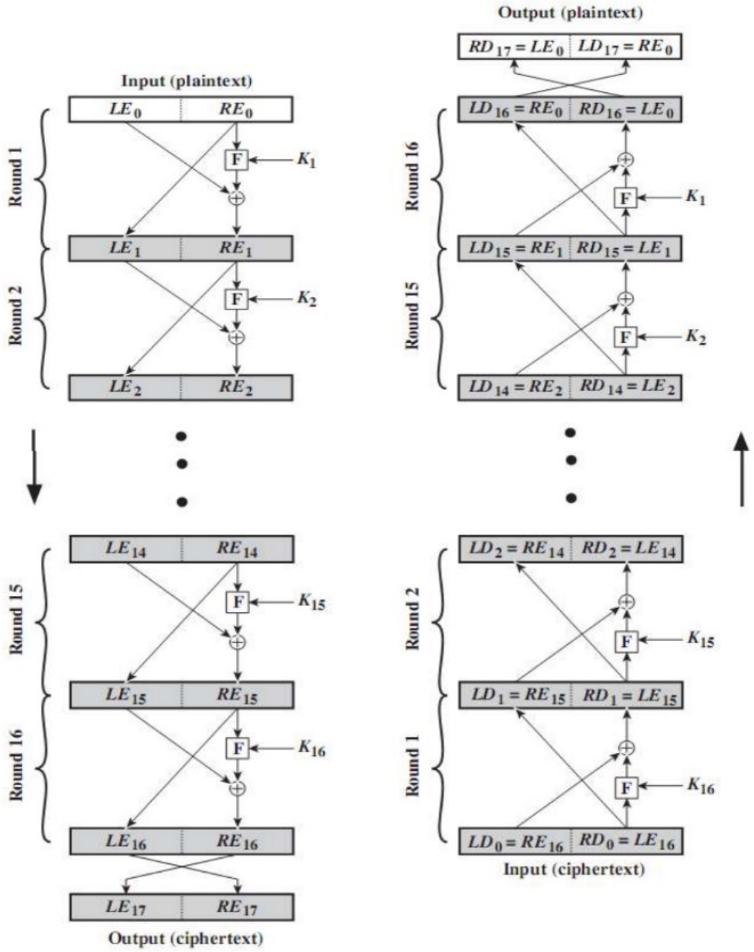
Feistel Cipher Structure

- most symmetric block ciphers are based on a **Feistel Cipher Structure**
- based on the two primitive cryptographic operations
  - *substitution* (S-box)
  - *permutation* (P-box)
- provide *confusion* and *diffusion* of message

Feistel Cipher Structure

- Horst Feistel devised the **feistel cipher** in the 1973
  - based on concept of invertible product cipher
- partitions input block into two halves
  - process through multiple rounds which
    - perform a substitution on left data half
    - based on round function of right half & subkey
    - then have permutation swapping halves
- implements Shannon's substitution-permutation network concept

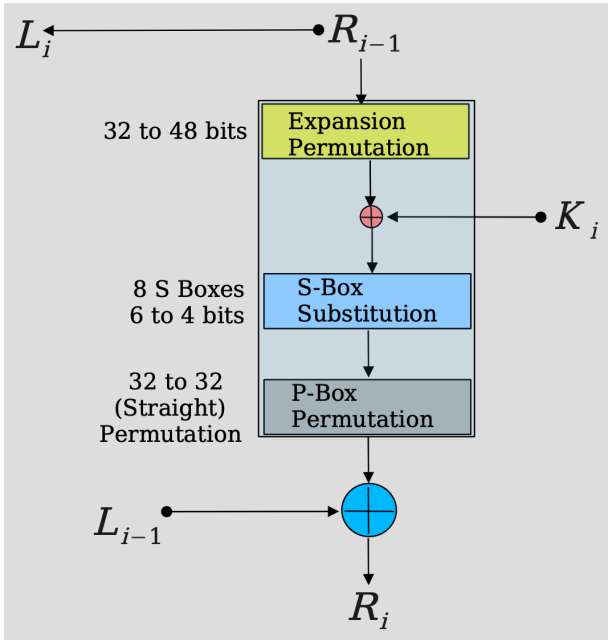
Feistel Encryption and Decryption



Encryption

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$



Symmetric Block Encryption

Block cipher

- the most commonly used symmetric encryption algorithms
- input: fixed-size blocks (Typically 64, 128 bit blocks), output: equal size blocks
- provide secrecy and/or authentication services
- Data Encryption Standard (DES), triple DES (3DES), and the Advanced Encryption Standard (AES)s
- Usually employ Feistel structure

Feistel Cipher Structure

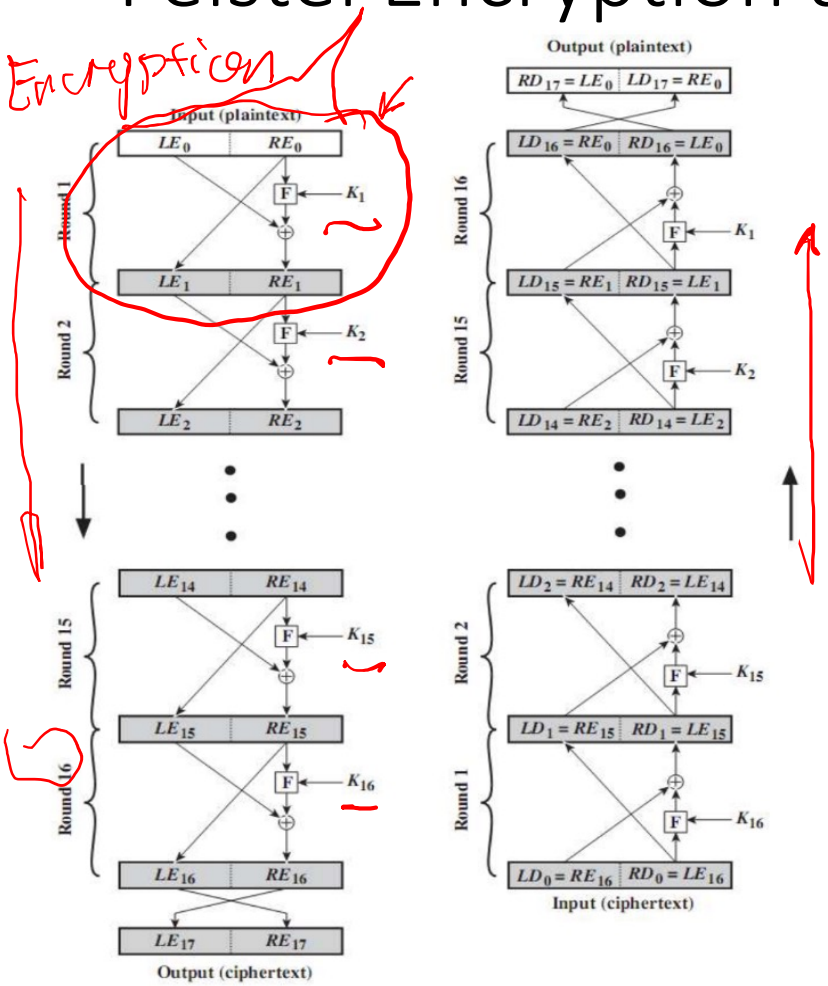
Feistel Cipher Structure

- most symmetric block ciphers are based on a **Feistel Cipher Structure**
- based on the two primitive cryptographic operations
  - *substitution* (S-box)
  - *permutation* (P-box)
- provide *confusion* and *diffusion* of message

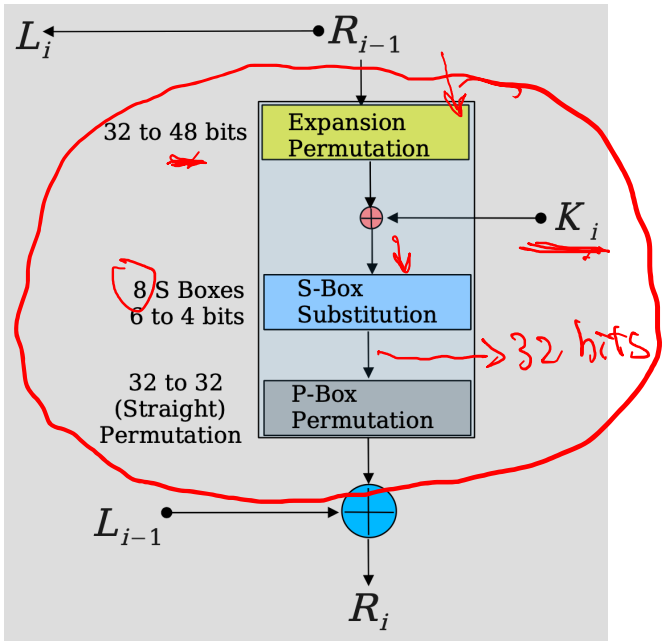
Feistel Cipher Structure

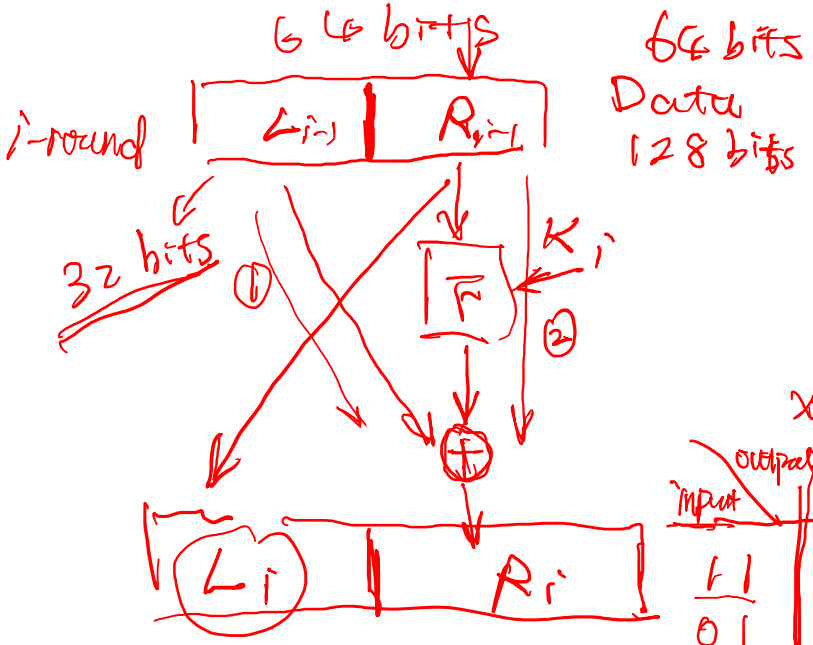
- Horst Feistel devised the **feistel cipher** in the 1973
  - based on concept of invertible product cipher
- partitions input block into two halves
  - process through multiple rounds which
    - perform a substitution on left data half
    - based on round function of right half & subkey
    - then have permutation swapping halves
- implements Shannon's substitution-permutation network concept

Feistel Encryption and Decryption



$$\textit{Encryption}$$
$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$



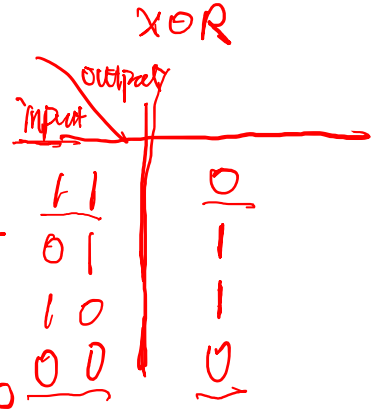


64 bits
Data
128 bits

$L_i = R_{i-1}$

$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$

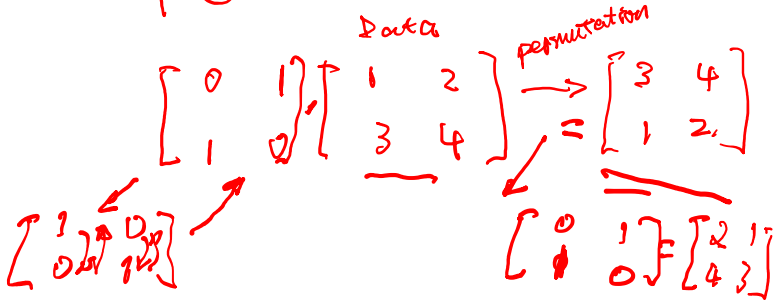
Feistel structure.

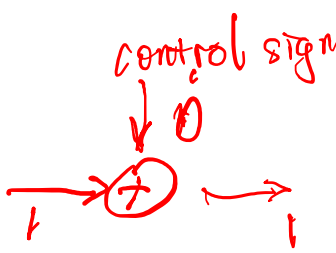


use cases

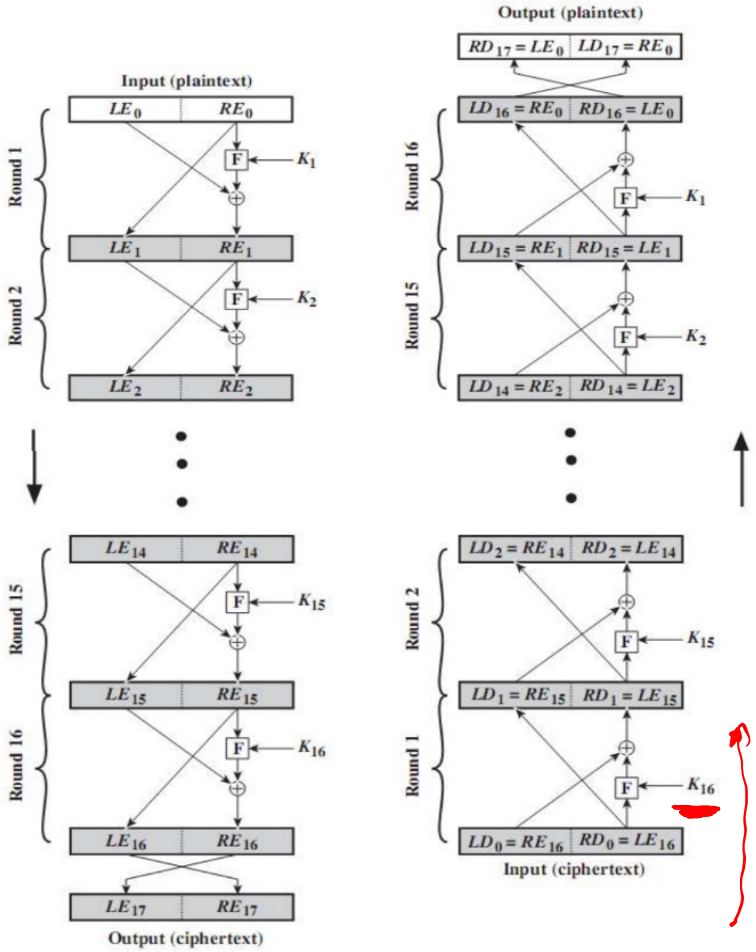
① detect differences of input.

② Flip bits





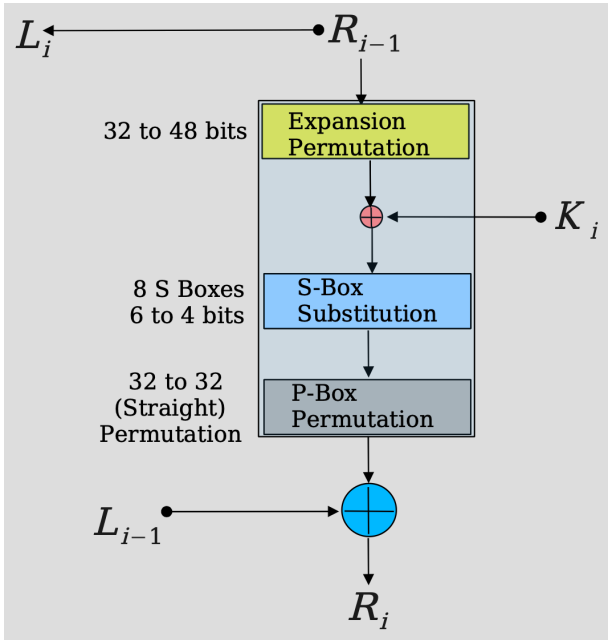
Feistel Encryption and Decryption



Encryption

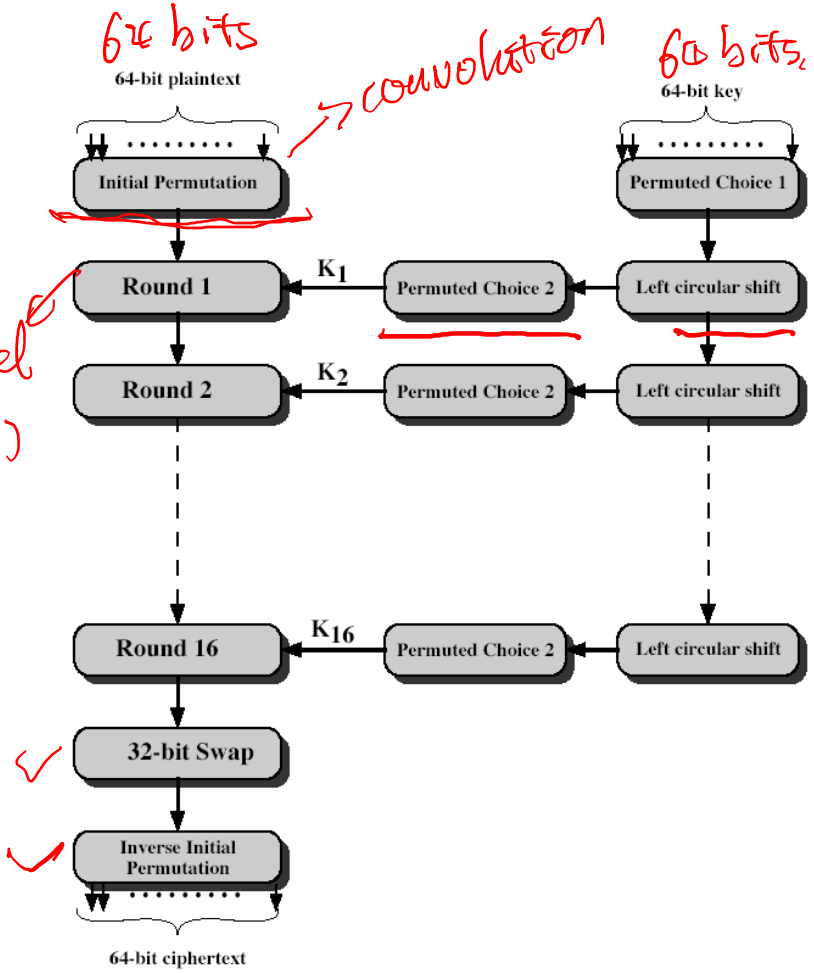
$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$



DES encryption

- 64 bits plaintext
- 56 bits effective key length

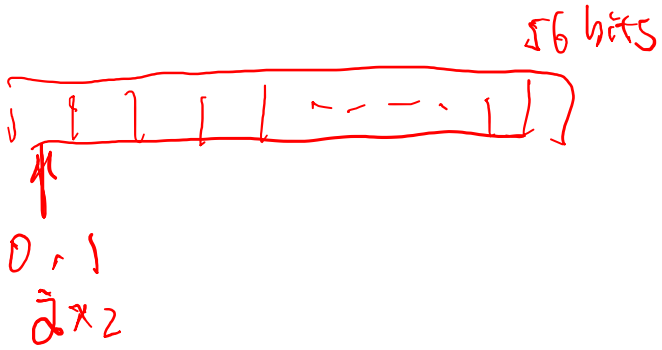


DES Weakness

try every possibity of keys

- short length key (56 bits) is not secure enough. Brutal force search takes short time.

2^{56}

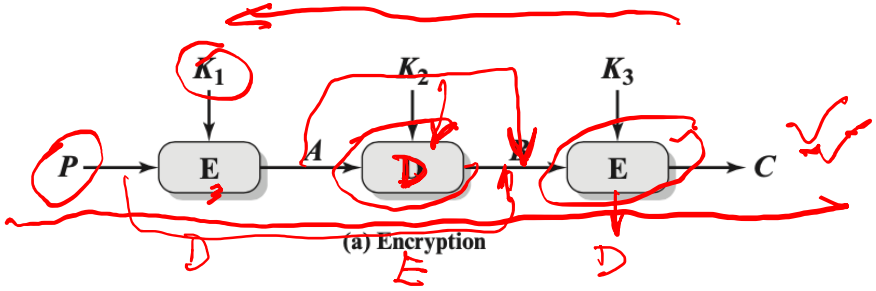


Triple DES (3DES)

$$C = E(K_3, D(K_2, E(K_1, P)))$$

where

C = ciphertext
P = plaintext
$E[K, X]$ = encryption of X using key K
$D[K, Y]$ = decryption of Y using key K



(b) Decryption

Decrypting with the wrong key will further convolute the output

3DES

- Triple DES with three different keys – brute-force complexity 2^{168}
- 3DES is the FIPS-approved symmetric encryption algorithm
- Weakness: slow speed for encryption

FIPS – Federal Information Processing Standards. The United States' Federal Information Processing Standards are publicly announced standards developed by the National Institute of Standards and Technology for use in computer systems by non-military American government agencies and government contractors

AES

- clearly a replacement for DES was needed
  - have theoretical attacks that can break it
  - have demonstrated exhaustive key search attacks
- can use Triple-DES – but slow with small blocks
- US NIST issued call for ciphers in 1997
- 15 candidates accepted in Jun 98
- 5 were short-listed in Aug-99
- Rijndael was selected as the AES in Oct-2000
- issued as FIPS PUB 197 standard in Nov-2001

NIST issued calls for post quantum

review AES

key L

PKI: unsafe

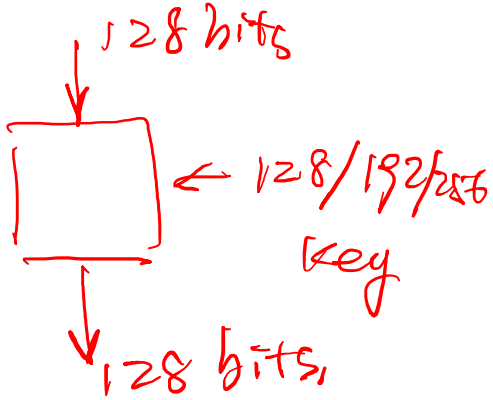
RSA & Shor

Criteria to evaluate AES

- General security
- Software implementations
- Restricted-space environments
- Hardware implementations
- Attacks on implementations
- Encryption versus decryption
- Key agility
- Other versatility and flexibility
- Potential for instruction-level parallelism

Cryptographic Standards and Guidelines | CSRC (nist.gov)

AES Specification



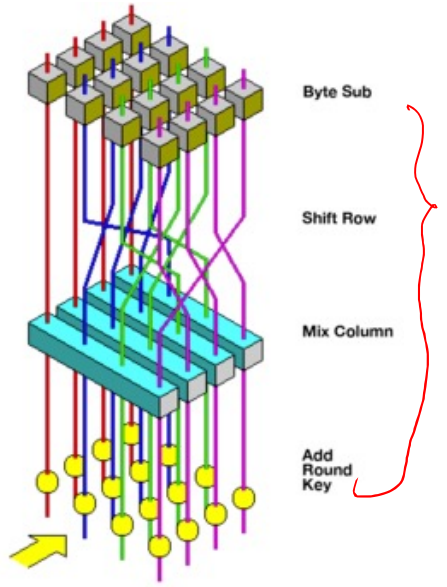
- symmetric block cipher
- 128-bit data, 128/192/256-bit keys
- stronger & faster than Triple-DES
- provide full specification & design details
- both C & Java implementations
- NIST have released all submissions & unclassified analyses

https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/aes-development/Rijndael-ammended.pdf

The AES Cipher - Rijndael

- an **iterative** rather than **feistel** cipher
  - treats data in 4 groups of 4 bytes
  - operates an entire block in every round
- designed to be:
  - resistant against known-plaintext attacks
  - speed and code compactness on many CPUs
  - design simplicity

Rijndael



- processes data as 4 groups of 4 bytes (state) = 128 bits
- has 10/12/14 rounds in which state undergoes:
  - byte substitution (1 S-box used on every byte)
  - shift rows (permute bytes row by row)
  - mix columns (alter each byte in a column as a function of all of the bytes in the column)
  - add round key (XOR state with key material)
- 128-bit keys – 10 rounds, 192-bit keys – 12 rounds, 256-bit keys – 14 rounds

AES Specification

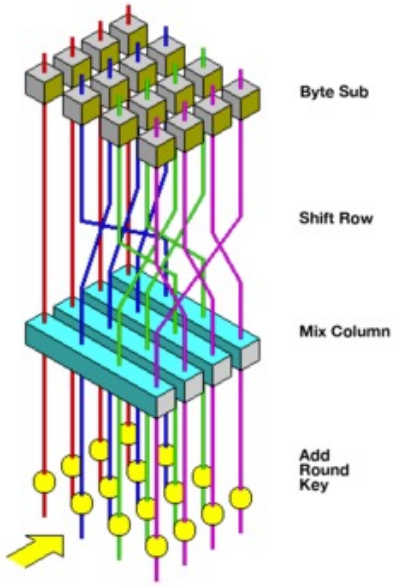
- symmetric block cipher
- 128-bit data, 128/192/256-bit keys
- stronger & faster than Triple-DES
- provide full specification & design details
- both C & Java implementations
- NIST have released all submissions & unclassified analyses

https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/aes-development/Rijndael-ammended.pdf

The AES Cipher - Rijndael

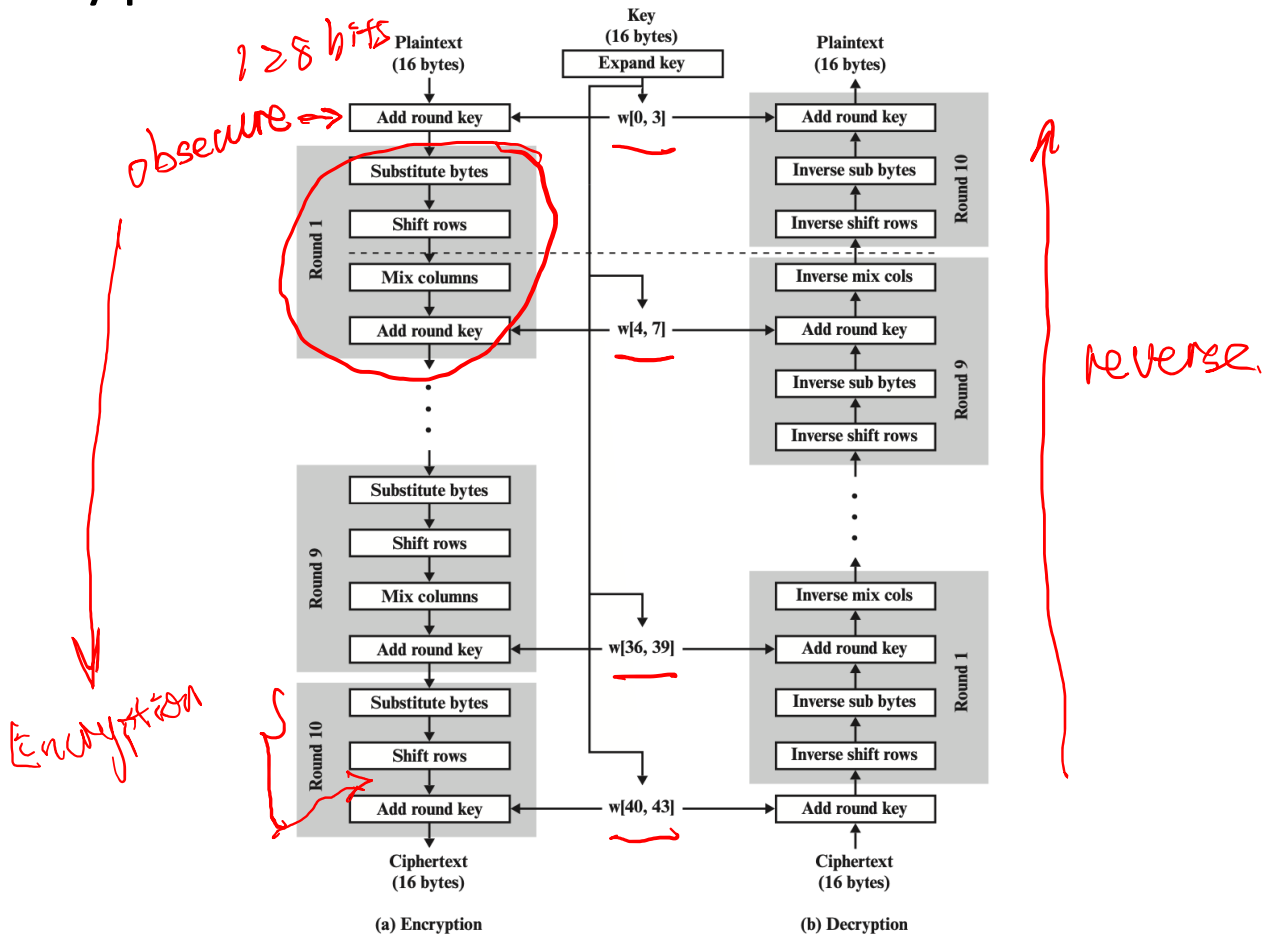
- an **iterative** rather than **feistel** cipher
  - treats data in 4 groups of 4 bytes
  - operates an entire block in every round
- designed to be:
  - resistant against known-plaintext attacks
  - speed and code compactness on many CPUs
  - design simplicity

Rijndael

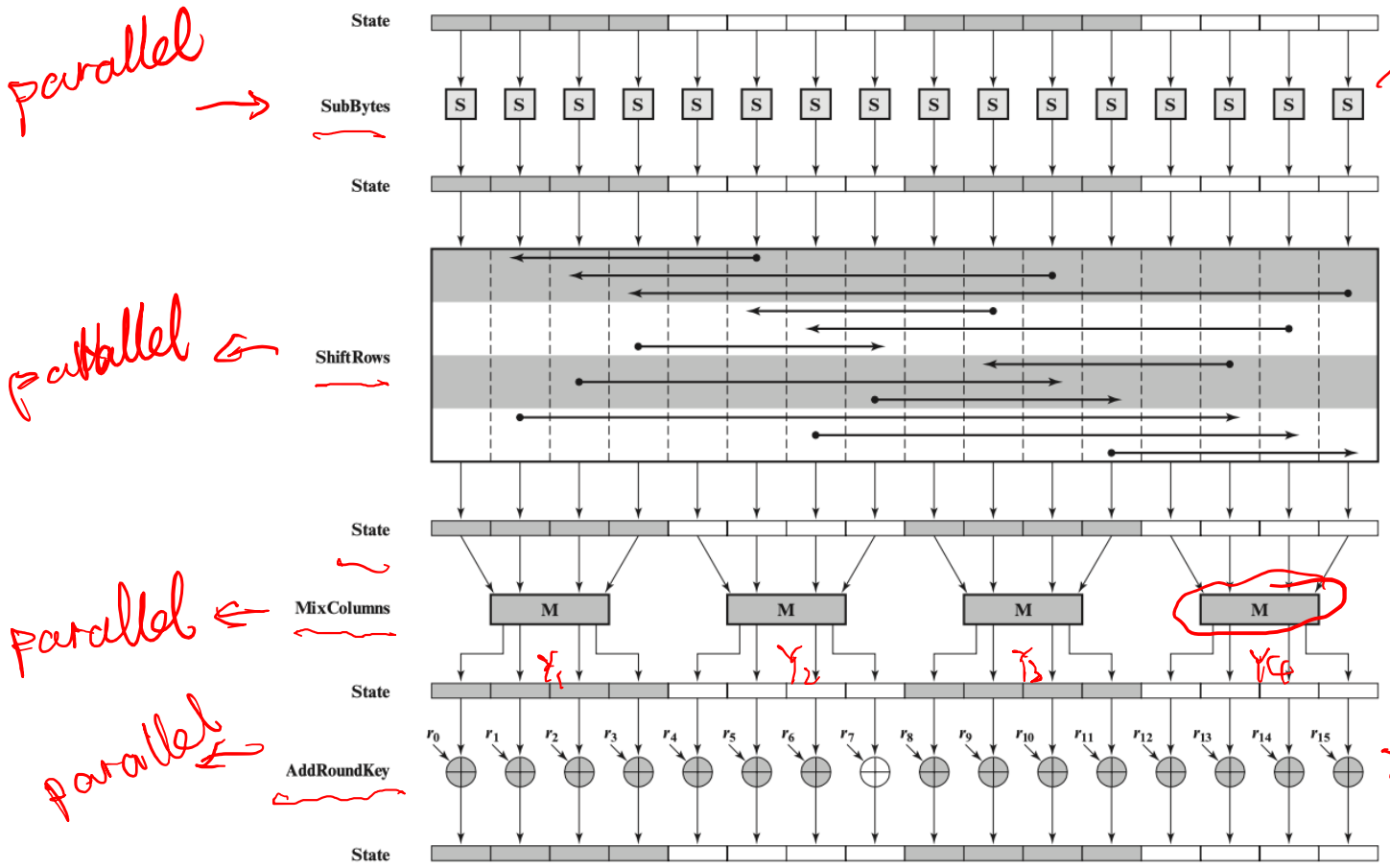


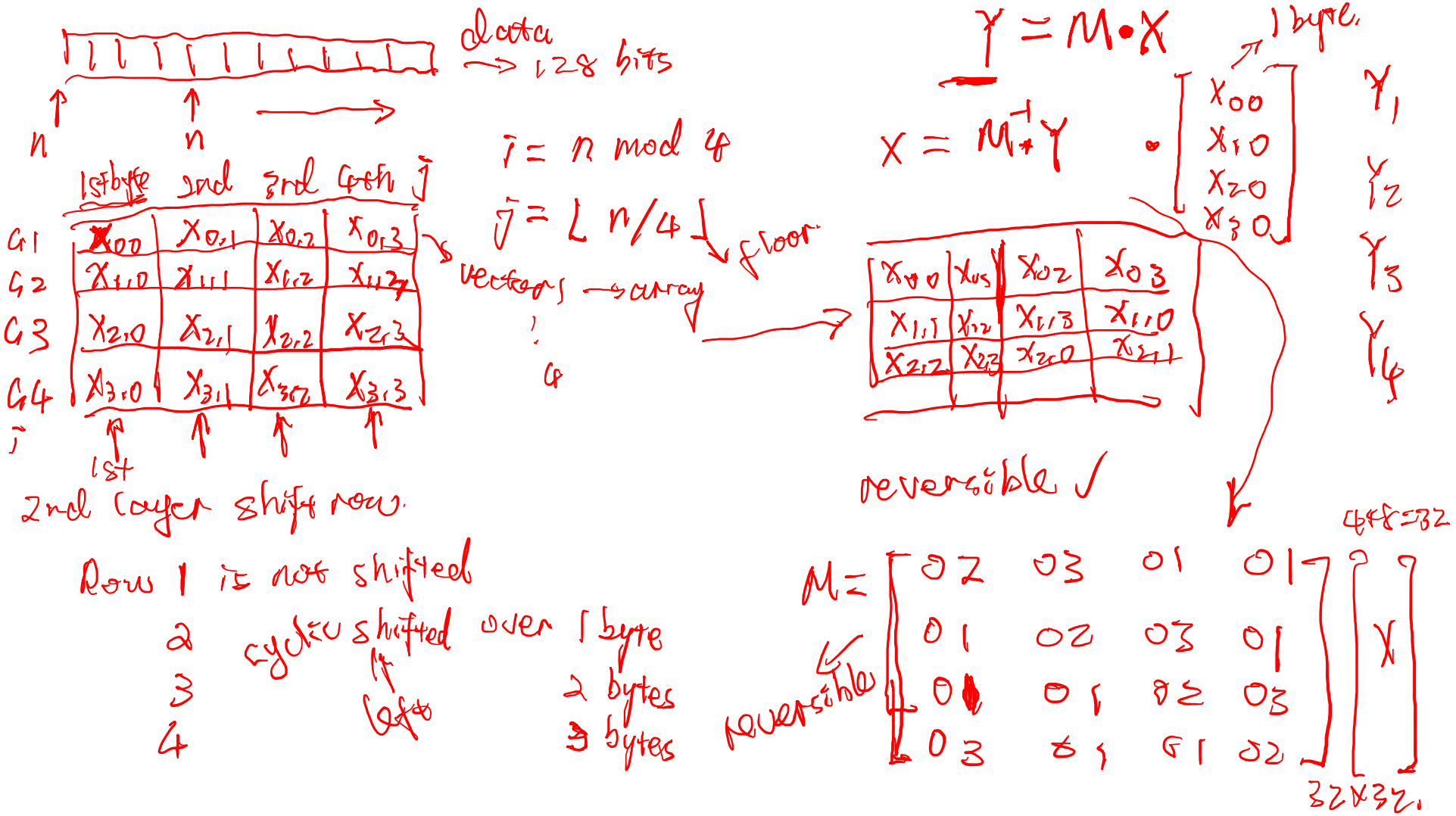
- processes data as 4 groups of 4 bytes (state) = 128 bits
- has 10/12/14 rounds in which state undergoes:
  - byte substitution (1 S-box used on every byte)
  - shift rows (permute bytes row by row)
  - mix columns (alter each byte in a column as a function of all of the bytes in the column)
  - add round key (XOR state with key material)
- 128-bit keys – 10 rounds, 192-bit keys – 12 rounds, 256-bit keys – 14 rounds

AES Encryption and Decryption



AES encryption round




data
→ 128 bits
n
n
i = n mod 4
j = ⌊ n/4 ⌋
floor
1st byte
2nd
3rd
4th
j
G1
G2
G3
G4
i
1st
vectors → array
4
Y = M • X
X = M⁻¹ • Y
1 byte.
reversible ✓
2nd layer shift row.
Row 1 is not shifted
2 cyclic shifted over 1 byte
3 2 bytes
4 3 bytes
left
reversible
M =
4*8=32
32x32.

AES pros

- Most operations can be combined into XOR and table lookups - hence very fast & efficient

Take-home Exercises

- Find an AES API to encrypt a text (A), then decrypt it and check whether the original text (A) equals the decrypted text (B). Whether A = B?
- Compare the decryption time with different key lengths, and with DES and 3DES.
  - Suggestions: find a large A file. Run decryption a couple of times and take the average.

Reading materials

- FIPS 197, Advanced Encryption Standard (AES) (nist.gov)

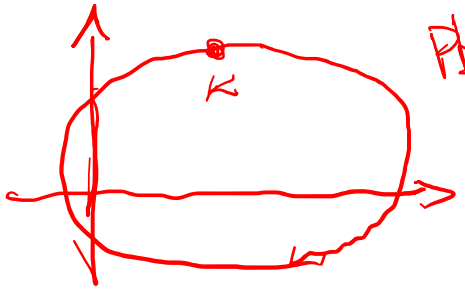
WPEC 2024: NIST Workshop on Privacy-Enhancing Cryptography

- Time: September 24–26, 2024
- Free to register
- Virtual conference via Zoom
- https://csrc.nist.gov/events/2024/wpec2024

Random and Pseudorandom Numbers

When to use random numbers?

- Generation of a stream key for symmetric stream cipher
- Generation of keys for public-key algorithms
  - RSA public-key encryption algorithm (described in Chapter 3)
- Generation of a symmetric key for use as a temporary **session key**
  - used in a number of networking applications, such as Transport Layer Security (Chapter 5), Wi-Fi (Chapter 6), e-mail security (Chapter 7), and IP security (Chapter 8)
- In a number of key distribution scenarios
  - Kerberos (Chapter 4)

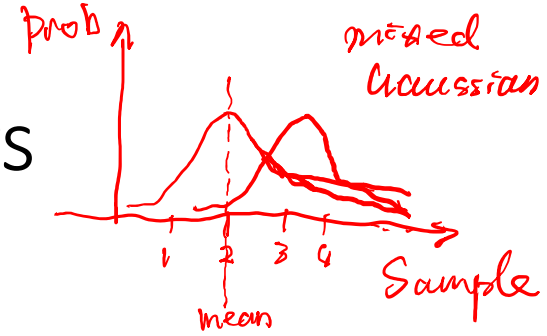


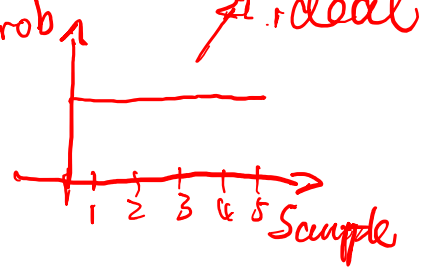
Two types of random numbers

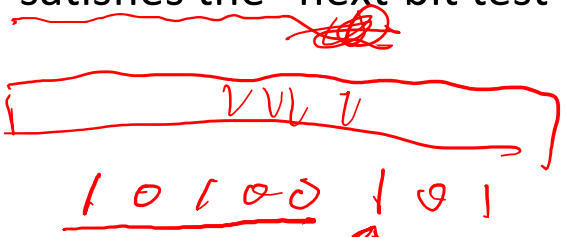
- True random numbers:
  - generated in non-deterministic ways. They are not predictable and repeatable
- Pseudorandom numbers:
  - appear random, but are obtained in a deterministic, repeatable, and predictable manner

Properties of Random Numbers

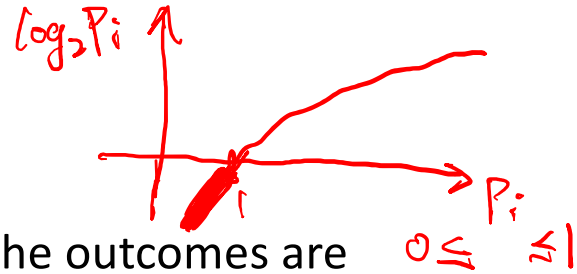
- Randomness
  - Uniformity
    - distribution of bits in the sequence should be uniform
  - Independence
    - no one subsequence in the sequence can be inferred from the others
- Unpredictable
  - satisfies the "next-bit test"







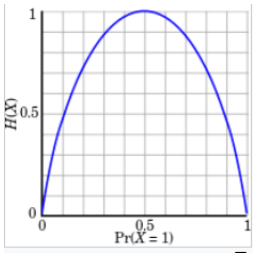
Entropy



- A measure of uncertainty
  - In other words, a measure of how unpredictable the outcomes are
  - High entropy = unpredictable outcomes = desirable in cryptography
  - The uniform distribution has the highest entropy (every outcome equally likely, e.g. fair coin toss)
  - Usually measured in bits (so 3 bits of entropy = uniform, random distribution over 8 values)

$$H = -\sum_i p_i \log_2 (p_i)$$

Entropy of an information source

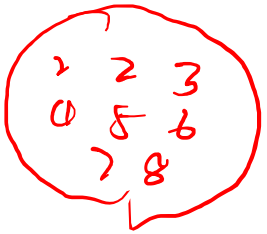


$$H = -\sum_{i=1}^{N=8} P_i \log_2(P_i)$$

$$= -\left[\frac{1}{8} \log_2 \frac{1}{8} + 0 \cdot \log_2 \frac{1}{16} + \frac{1}{16} \log_2 \frac{1}{16} + \cdots \frac{3}{16} \log_2 \frac{3}{16} \right]$$

1 value. 0 values. 3 value. 8 values

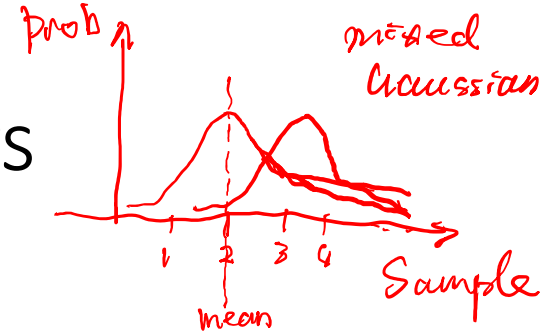
data source random

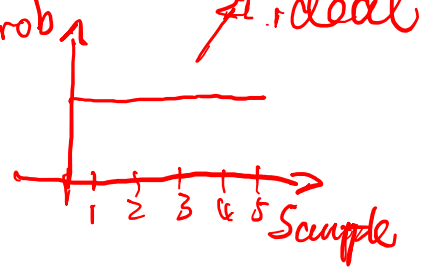


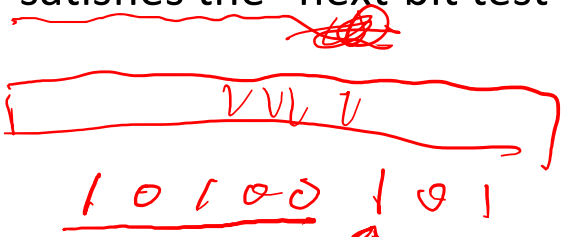
value	Prob
1	1/8
2	0
3	1/16
4	1/4
5	1/8
6	3/16
7	1/16
8	3/16

Properties of Random Numbers

- Randomness
  - Uniformity
    - distribution of bits in the sequence should be uniform
  - Independence
    - no one subsequence in the sequence can be inferred from the others
- Unpredictable
  - satisfies the "next-bit test"





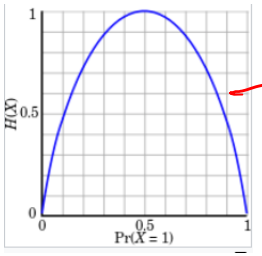


Entropy

- A measure of uncertainty
  - In other words, a measure of how unpredictable the outcomes are
  - High entropy = unpredictable outcomes = desirable in cryptography
  - The uniform distribution has the highest entropy (every outcome equally likely, e.g. fair coin toss)
  - Usually measured in bits (so 3 bits of entropy = uniform, random distribution over 8 values)

$$H = -\sum_i p_i \log_2 (p_i)$$

Entropy of an information source



$$H = -\sum_{i=1}^{n=8} P_i \log_2(P_i) \quad \checkmark$$

$$\frac{dH}{dP_i} = 0 \Rightarrow P_i^* = \frac{1}{n}$$

$$\sum_{i=1} P_i = 1$$

$$= -\left[\frac{1}{8}\log_2\frac{1}{8} + 0 + \frac{1}{16}\log_2\frac{1}{16} + \cdots \frac{3}{16}\log_2\frac{3}{16}\right]$$

1 value, 2 values, 3 value, 8 value

$$= -\left[-\frac{3}{8} - \frac{4}{16} - \frac{2}{4} - \frac{3}{8} - 0.405 - \frac{4}{16} - 0.405\right]$$

$$= 2.234$$

$$8 = 2^3$$

$$H_2 = 1.24$$

ideal $H = -\sum_{i=1}^{8} \frac{1}{8}\log_2\frac{1}{8}$

$= -8 \cdot \frac{1}{8}\log_2\frac{1}{8}$

$= -8 \cdot \frac{1}{8}(-3) = 3$

prob. $\int$ empirical (true)

$P_i = \frac{C_i}{\sum_i C_i} < 1$

$\sum_i P_i = 1$

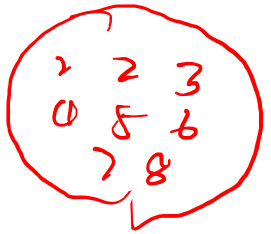
$t \to \infty$

t period

C_i

$0 \le C_i$

data source random



value		prob
1	C_1	1/8
2	C_2	0
3	C_3	1/16
4		1/4
5		1/8
6		3/16
7		1/16
8		3/16

$\partial \rightarrow$ partial derivative

$$\frac{dH}{dp_i} = -\sum_{i=1}^{n}\left[\log_2(p_i) + p_i \cdot \frac{d\log_2 p_i}{dp_i}\right] \qquad \frac{d[\log_a x]}{dx} = \frac{1}{x} \rightarrow \text{search.}$$

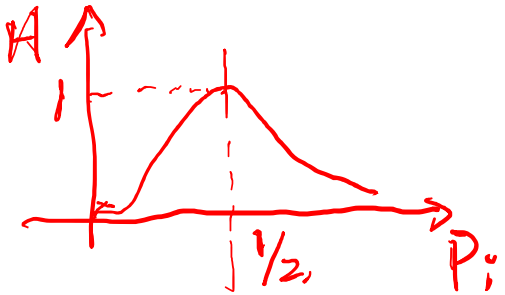
$$= 0$$

$$p_i = \frac{1}{n}$$

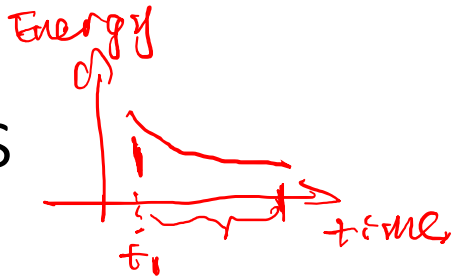
$$\sum_{i=1}^{2} \frac{1}{2} \cdot \log_2 \frac{1}{2}$$

$$2^1 = 2$$

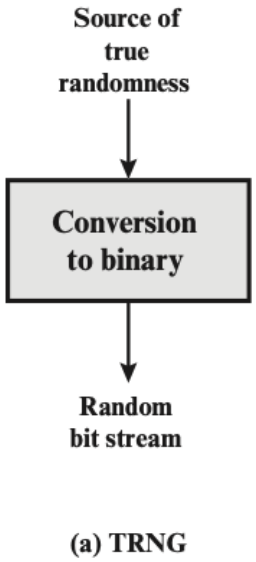
$n = 2$



True random numbers generators



- Several sources of randomness – natural sources of randomness
  - decay times of radioactive materials
  - electrical noise from a resistor or semiconductor
  - radio channel or audible noise
  - keyboard timings
  - disk electrical activity
  - mouse movements
  - Physical unclonable function (PUF)
- Some are better than others



(a) TRNG

Combining sources of randomness

- Suppose r1, r2, …, rk are random numbers from different sources. E.g.,

  r1 = electrical noise from a resistor or semiconductor

  r2 = sample of hip-hop music on radio

  r3 = clock on computer

  b = r1⊕r2⊕…⊕rk

  If any one of r1, r2, …, rk is truly random, then so is b

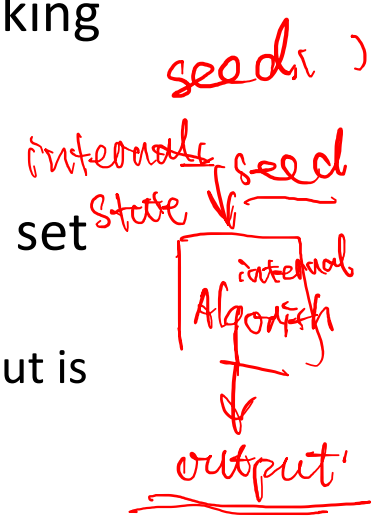
  Many poor sources + 1 good source = good entropy

Pseudorandom Number Generators (PRNGs)

- True randomness is expensive
- **Pseudorandom number generator** (**PRNGs**): An algorithm that uses a little bit of true randomness to generate a lot of random-looking output
  - Also called **deterministic random bit generators** (**DRBGs**)
- PRNGs are deterministic: Output is generated according to a set algorithm
  - However, for an attacker who can't see the internal state, the output is *computationally indistinguishable* from true randomness

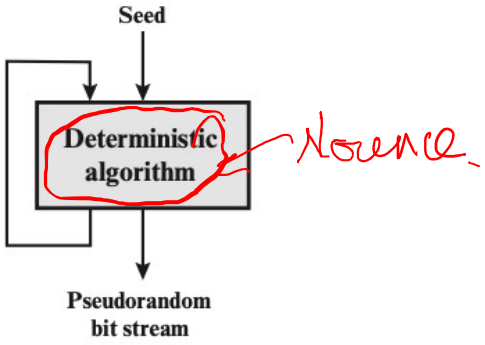
Pseudorandom Number Generators (PRNGs)

- True randomness is expensive
- **Pseudorandom number generator** (**PRNGs**): An algorithm that uses a little bit of true randomness to generate a lot of random-looking output
  - Also called **deterministic random bit generators** (**DRBGs**)
- PRNGs are deterministic: Output is generated according to a set algorithm
  - However, for an attacker who can't see the internal state, the output is *computationally indistinguishable* from true randomness



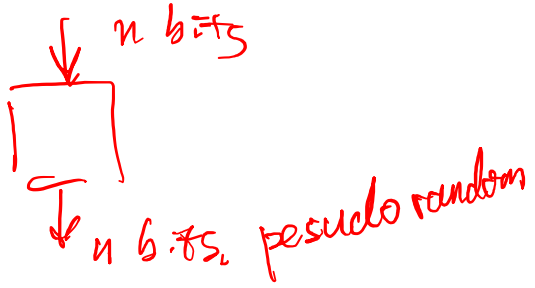
PRNG: Definition

- A PRNG has two functions:
  - PRNG.Seed(randomness): Initializes the internal state using the entropy
    - Input: Some truly random bits
  - PRNG.Generate(*n*): Generate *n* pseudorandom bits
    - Input: A number *n*
    - Output: *n* pseudorandom bits
    - Updates the internal state as needed

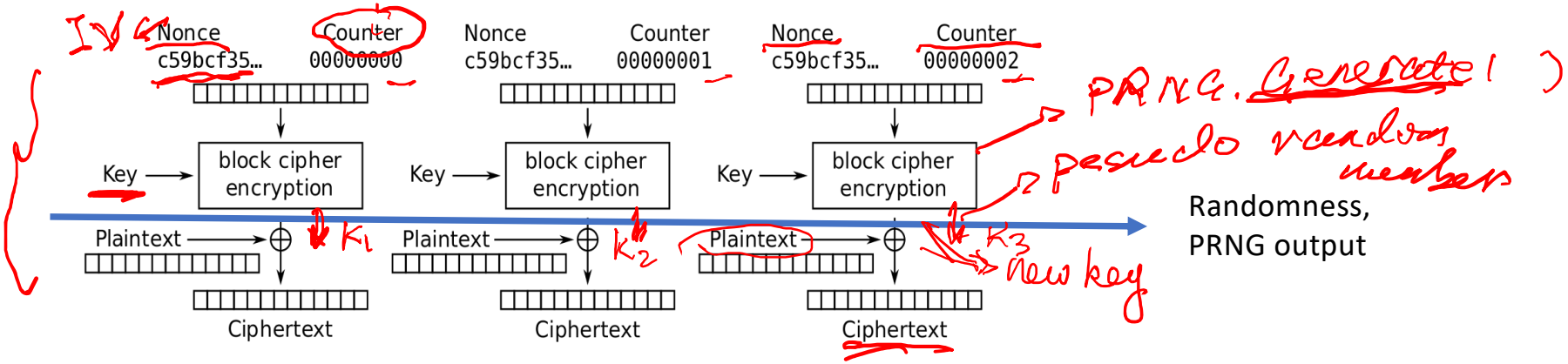


- Properties
  - **Correctness**: Deterministic
  - **Efficiency**: Efficient to generate pseudorandom bits
  - **Security**: Indistinguishability from random
  - **Rollback resistance**: cannot deduce anything about any previously-generated bit

Example construction of PRNG



- Using block cipher in Counter (CTR) mode:
- If you want m random bits, and a block cipher with E_k has n bits, apply the block cipher m/n times and concatenate the result:
- PRNG.Seed(K | IV) = E_k(IV, 1) | E_k(IV, 2) | E_k(IV, 3) … E_k(IV, ceil(m/n)),
  - | is concatenation
  - Initialization vector (IV) / Nonce – typically is random or pseudorandom



Counter (CTR) mode encryption

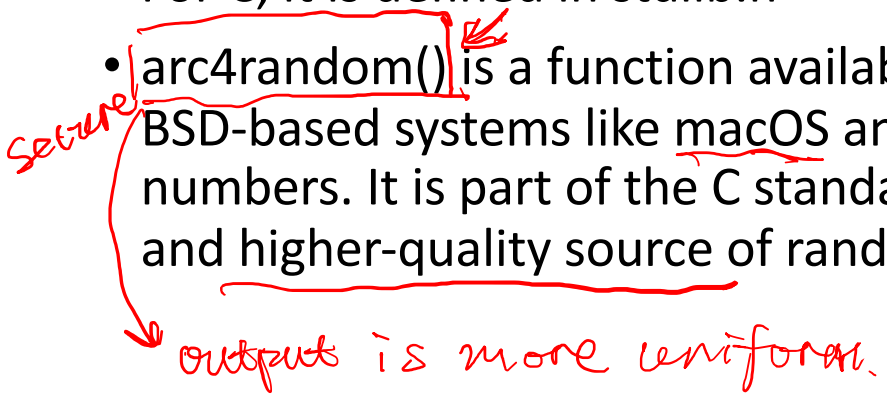
PRNG: Security

- Can we design a PRNG that is truly random?
- A PRNG cannot be truly random
  - The output is deterministic given the initial seed
- A secure PRNG is computationally indistinguishable from random to an attacker
  - Game: Present an attacker with a truly random sequence and a sequence outputted from a secure PRNG
  - An attacker should be able to determine which is which with probability ≈ 0
- Equivalence: An attacker cannot predict future output of the PRNG

P[attacker correctly identifies target] ≤ 0

Create pseudorandom numbers

- Truly random numbers are impossible with any program!
- However, we can generate seemingly random numbers, called pseudorandom numbers
- The function rand() returns a non-negative number between 0 and RAND_MAX
- For C, it is defined in stdlib.h
- arc4random() is a function available in some operating systems (primarily BSD-based systems like macOS and FreeBSD) that generates random numbers. It is part of the C standard library and provides a more secure and higher-quality source of random numbers compared to rand()

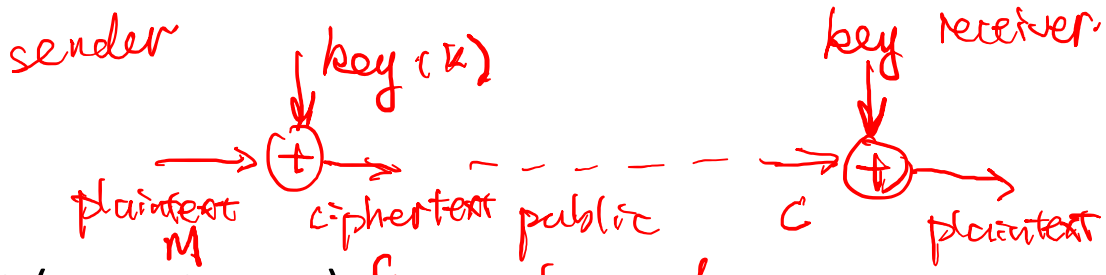


PRNGs: Summary

- True randomness requires sampling a physical process
- PRNG: An algorithm that uses a little bit of true randomness to generate a lot of random-looking output
  - Seed(entropy): Initialize internal state
  - Generate(n): Generate n bits of pseudorandom output
- Security: computationally indistinguishable from truly random bits

Stream Ciphers

Stream Ciphers



- process the message bit by bit (as a stream)
- typically have a (pseudo) random **stream key**
- combined (XOR) with plaintext bit by bit
- randomness of **stream key** completely destroys any statistically properties in the message
  - $C_i = M_i$ XOR $StreamKey_i$
- what could be simpler!!!!
- but must never reuse stream key
  - otherwise, can remove effect and recover messages, M⊕K⊕K = M

$M=1 \quad 1 \oplus 0 = 1$

$M=0 \quad 0 \oplus 0 = 0$

$C_i = M_i \oplus K_i$ ← as random as possible

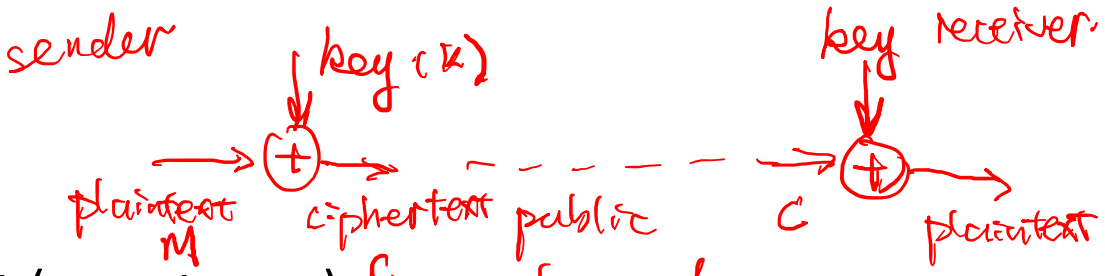
security

ideal case: truely random

Associativity $M \oplus K \oplus K = M \oplus (K \oplus K)$

Decrypt $= M \oplus 0 = M$

Stream Ciphers



- process the message bit by bit (as a stream)
- typically have a (pseudo) random **stream key**
- combined (XOR) with plaintext bit by bit
- randomness of **stream key** completely destroys any statistically properties in the message
  - $C_i = M_i$ XOR $StreamKey_i$
- what could be simpler!!!!
- but must never reuse stream key
  - otherwise, can remove effect and recover messages, M⊕K⊕K = M

$M=1 \quad 1 \oplus 0 = 1$

$M=0 \quad 0 \oplus 0 = 0$

$C_i = M_i \oplus K_i$ ← as random as possible

security

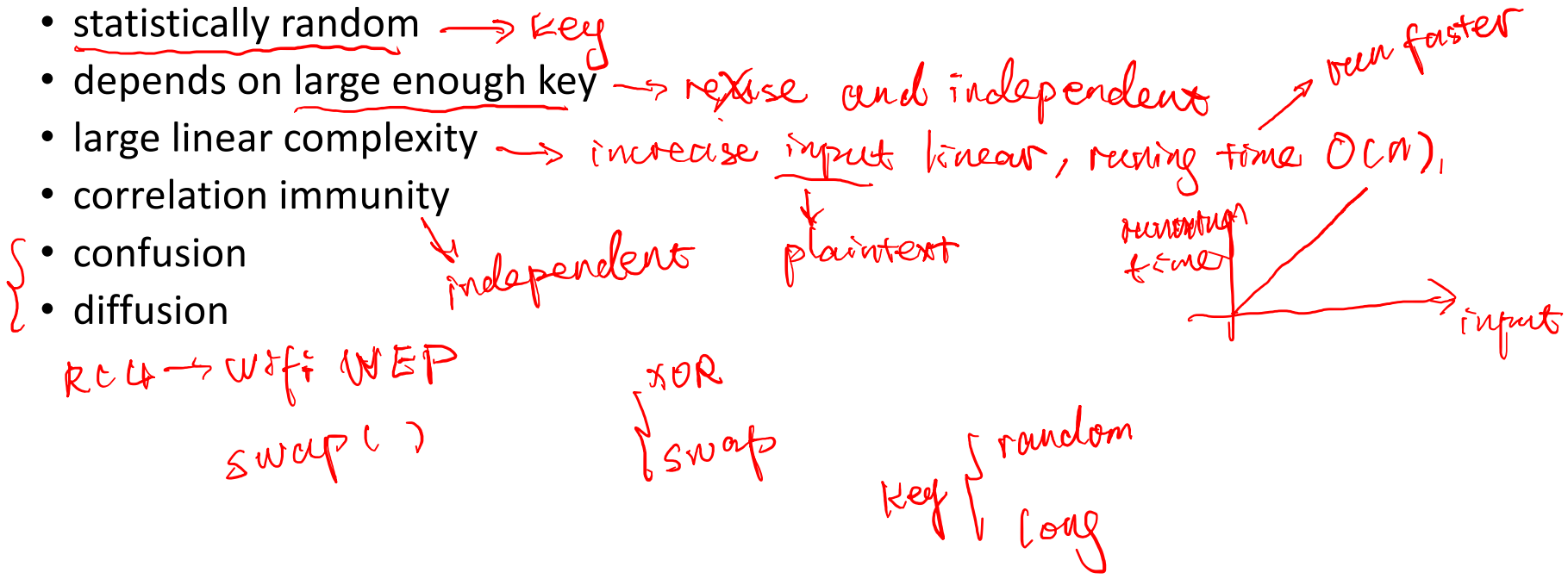
ideal case: truely random

Associativity $M \oplus K \oplus K = M \oplus (K \oplus K)$

Decrypt $= M \oplus 0 = M$

Stream Cipher Properties

- some design considerations are:
  - statistically random
  - depends on large enough key
  - large linear complexity
  - correlation immunity
  - confusion
  - diffusion



How to generate Stream Key?

- How to generate Stream Key?

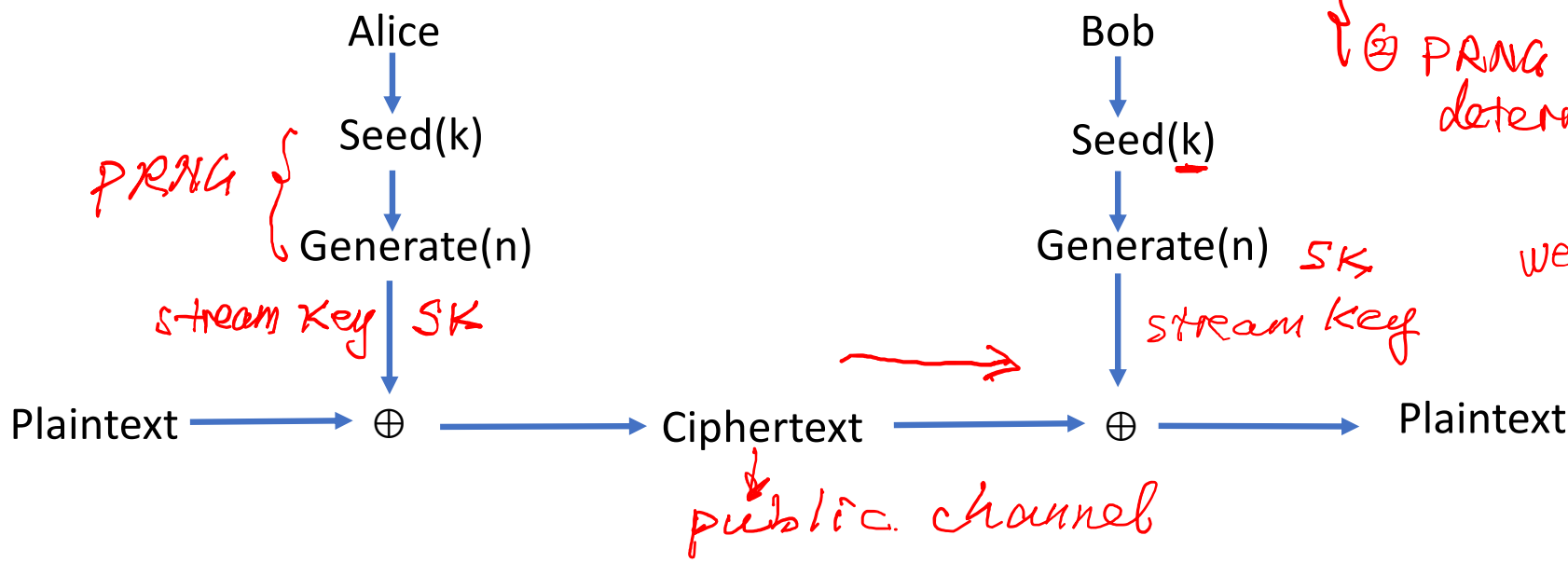
PRNG

Stream Ciphers

- Idea: replace "rand" by "pseudo rand"
- Use Pseudo Random Number Generator
  - A secure PRNG produces output that looks indistinguishable from random
  - An attacker who can't see the internal PRNG state can't learn any output
- PRNG: $\{0,1\}^s \rightarrow \{0,1\}^n$
  - expand a short (e.g., 128-bit) random seed into a long (typically unbounded) string that "looks random"
- Secret key is the seed
  - Basic encryption method: $E_{key}[M] = M \oplus PRNG(key)$

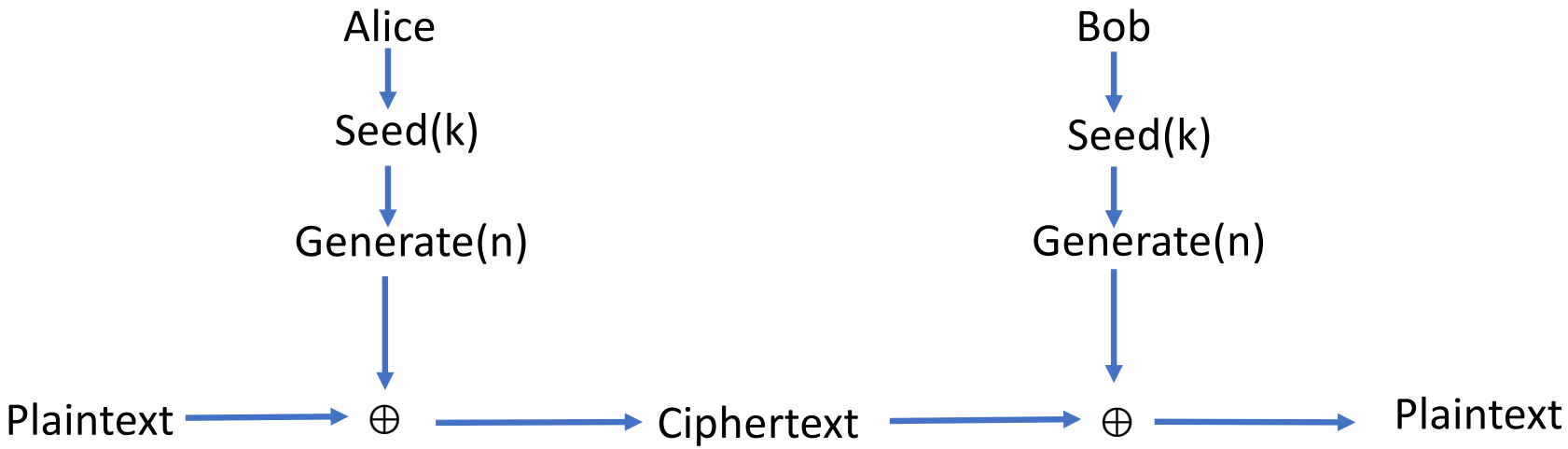
Stream Ciphers

- Protocol: Alice and Bob both seed a secure PRNG with their symmetric secret key, and then use the output as the key for stream key



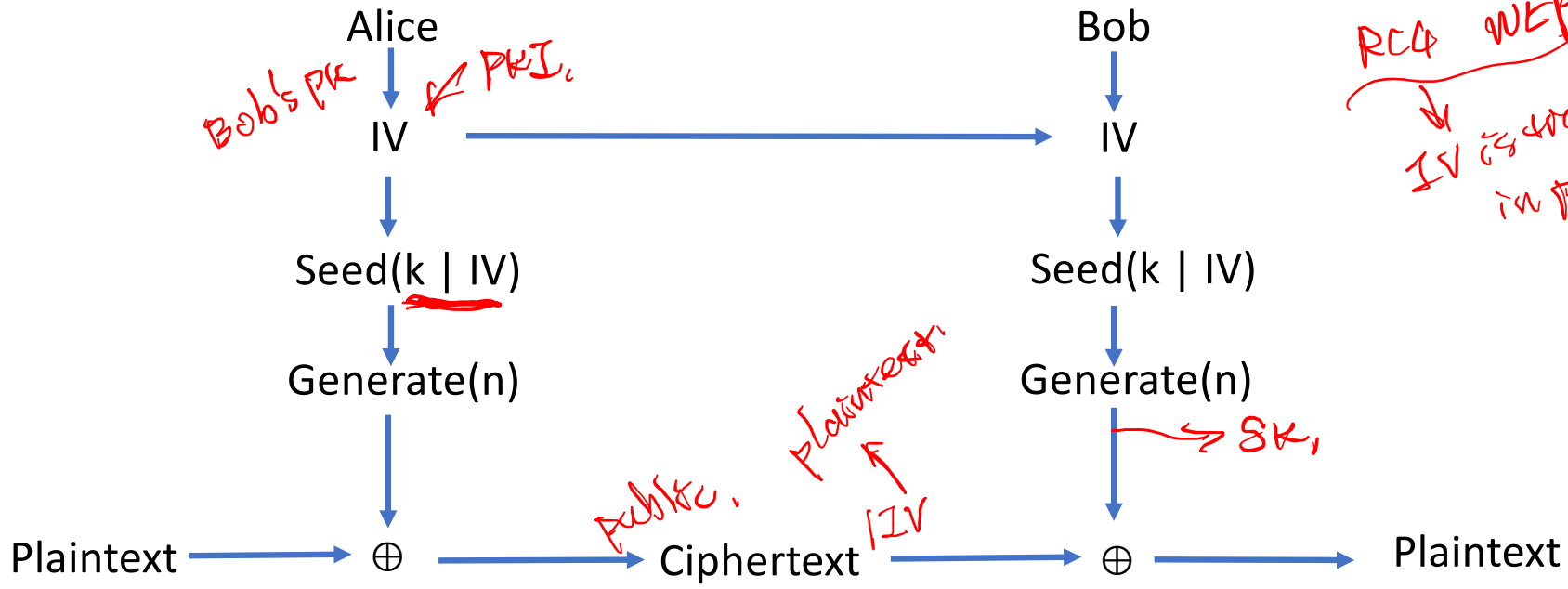
Stream Ciphers: Encrypting Multiple Messages

- How do we encrypt multiple messages without key reuses?



Stream Ciphers: Encrypting Multiple Messages

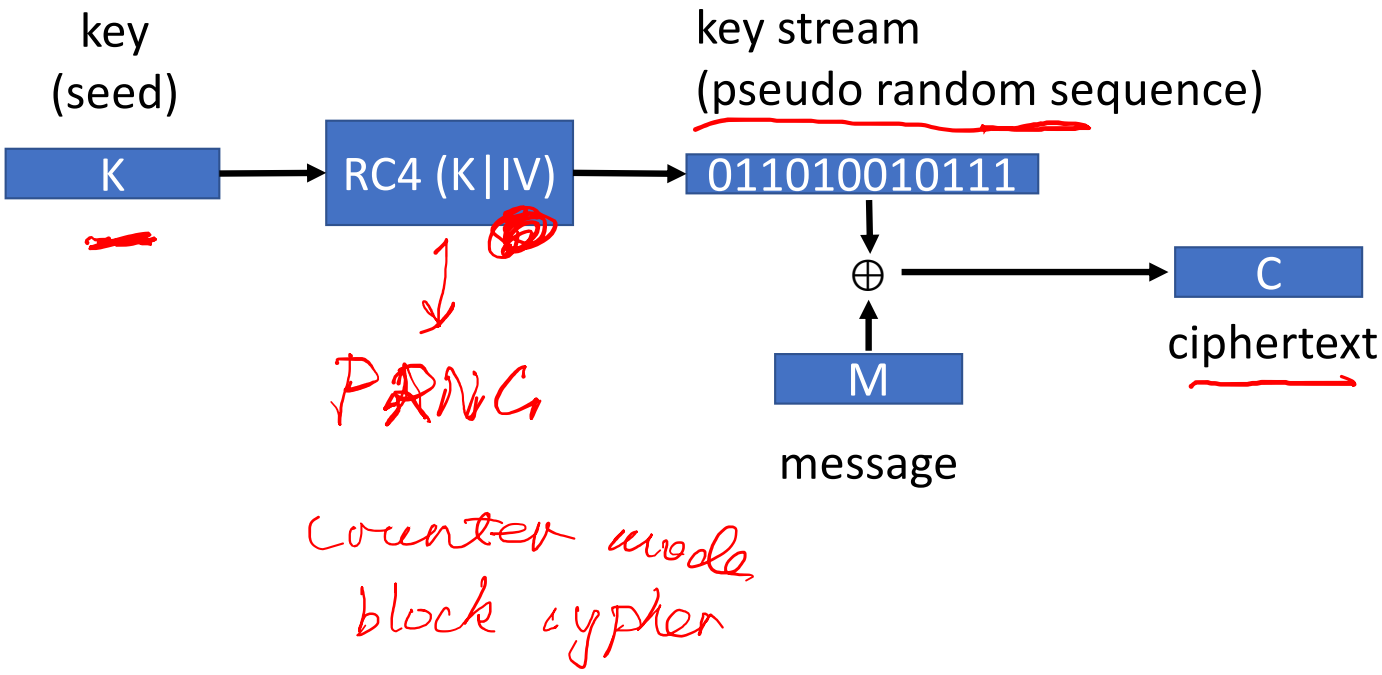
- Solution: For each message, seed the PRNG with the key and a random IV, concatenated("|"). Send the IV with the ciphertext



Real-world example: RC4

- a proprietary cipher designed in 1987
- Extremely simple but effective!
- Very fast - especially in software
- Easily adapts to any key length, byte-oriented stream cipher
- widely used (web SSL/TLS, wireless WEP, WAP)
- A trade secret by RSA Security
- uses that permutation to scramble input info processed a byte at a time

RC4 Stream Cipher



RC4 Key Schedule

- starts with an array S of numbers: 0…255
- use key to well and truly shuffle
- S forms internal state of the cipher
- given a key k of length l bytes

```
/* Initialization */
for i = 0 to 255 do
S[i] = i;
T[i] = K[i mod keylen];

/* Initial Permutation of S */
j = 0;
for i = 0 to 255 do
  j = (j + S[i] + T[i]) mod 256;
  Swap (S[i], S[j]);
```

Throw away T & K, retain S

$T[i] = K[i \bmod keylen]$ $K = [1, 2, 3, 4]$ $keylen = 4$

If $keylen > i$; $T[i] = [1, 2, 3, 4]$ $0 \bmod 4 = 0 \Rightarrow K[0]$

$1 \bmod 4 = 1 \Rightarrow K[1]$

$T[i] = K[i]$

If $keylen \leq i$ $T[4] = K[4 \bmod 4] = K[0] = 1$

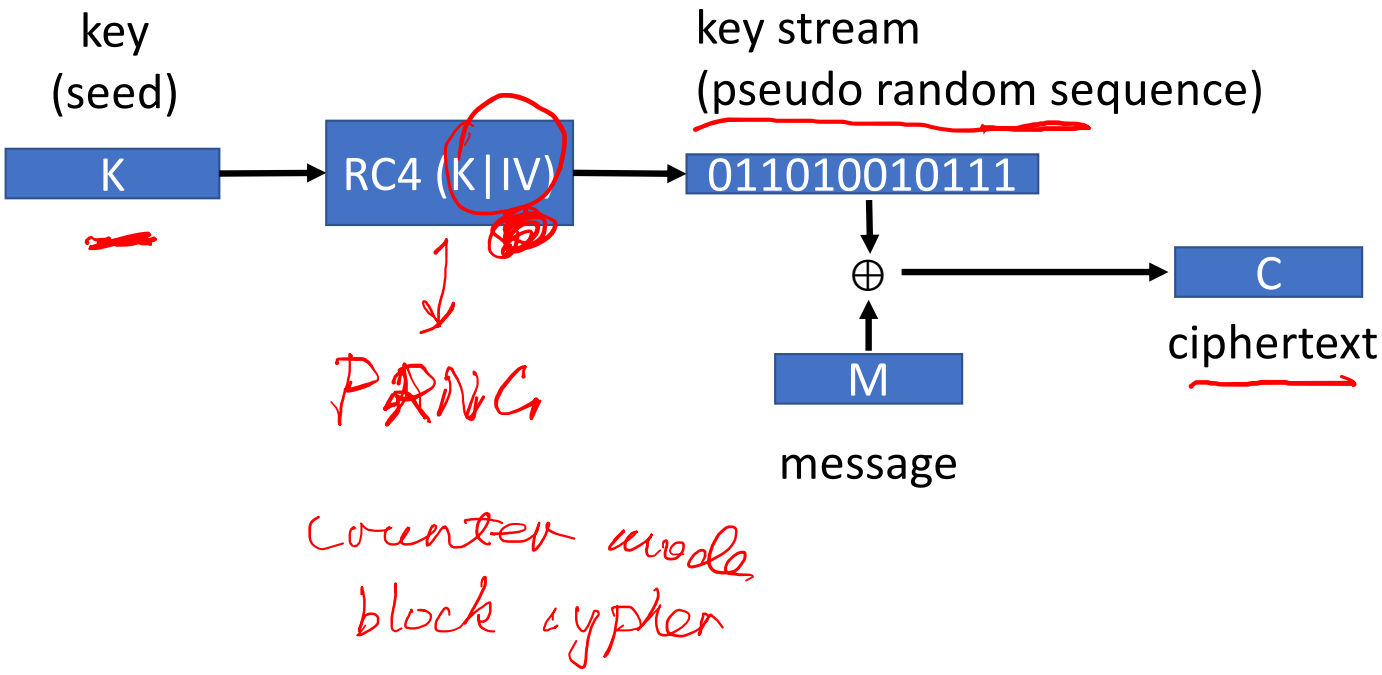
$T[5] = K[5 \bmod 4] = K[1] = 2$

$T = [1, 2, 3, 4, 1, 2]$ 3 4 , 1 2 > 4

repeat pattern, ⇒ reuse key

∴ add IV to be large enough

RC4 Stream Cipher



RC4 Key Schedule

- starts with an array S of numbers: 0…255
- use key to well and truly shuffle
- S forms internal state of the cipher
- given a key k of length l bytes

```
/* Initialization */
for i = 0 to 255 do
S[i] = i;
T[i] = K[i mod keylen];

/* Initial Permutation of S */
j = 0;
for i = 0 to 255 do
  j = (j + S[i] + T[i]) mod 256;
  Swap (S[i], S[j]);
```

Throw away T & K, retain S

$T[i] = K[i \bmod keylen]$ $K = [1, 2, 3, 4]$ $keylen = 4$

If $keylen > i$; $T[i] = [1, 2, 3, 4]$ $0 \bmod 4 = 0 \Rightarrow K[0]$

$1 \bmod 4 = 1 \Rightarrow K[1]$

$T[i] = K[i]$

If $keylen \leq i$ $T[4] = K[4 \bmod 4] = K[0] = 1$

$T[5] = K[5 \bmod 4] = K[1] = 2$

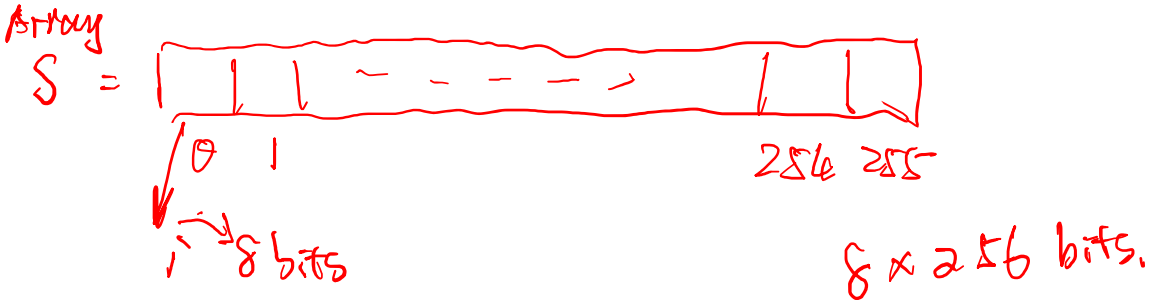
interesting $[0, 255]$ $T = [1, 2, 3, 4, 1, 2]$ 3 4, 12 > 4

repeat pattern, ⇒ reuse key

∴ add IV to be large enough

RC4 Encryption

- encryption continues shuffling array values
- sum of shuffled pair selects "stream key" value
- XOR with next byte of message to en/decrypt




```
i = j = 0
for each message byte Mi
    i = (i + 1) (mod 256)
    j = (j + S[i]) (mod 256)
    swap(S[i], S[j])
    t = (S[i] + S[j]) (mod 256)
    Ci = Mi' XOR S[t]
```

relates

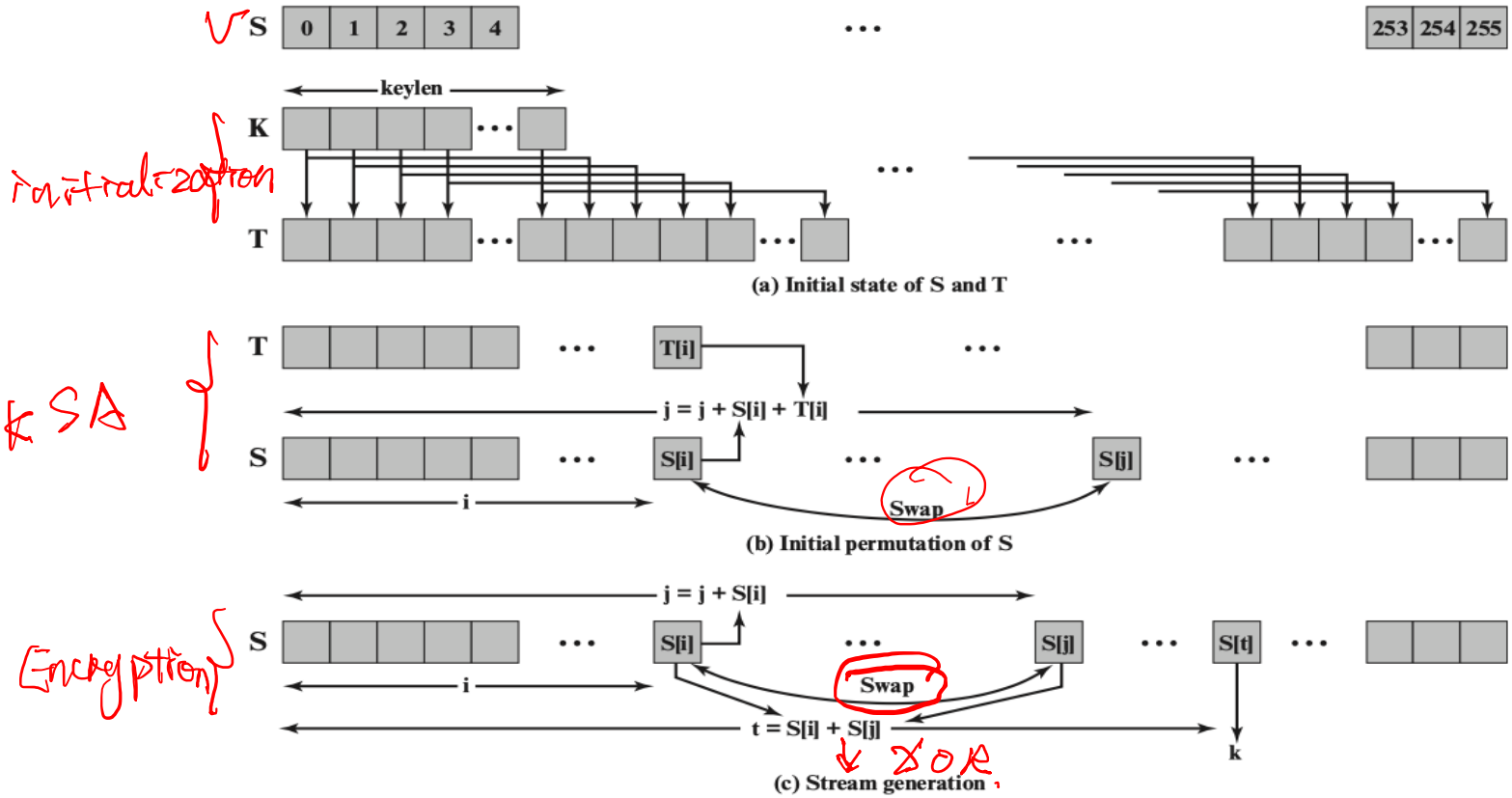
diffusion

8 bit byte

new key byte

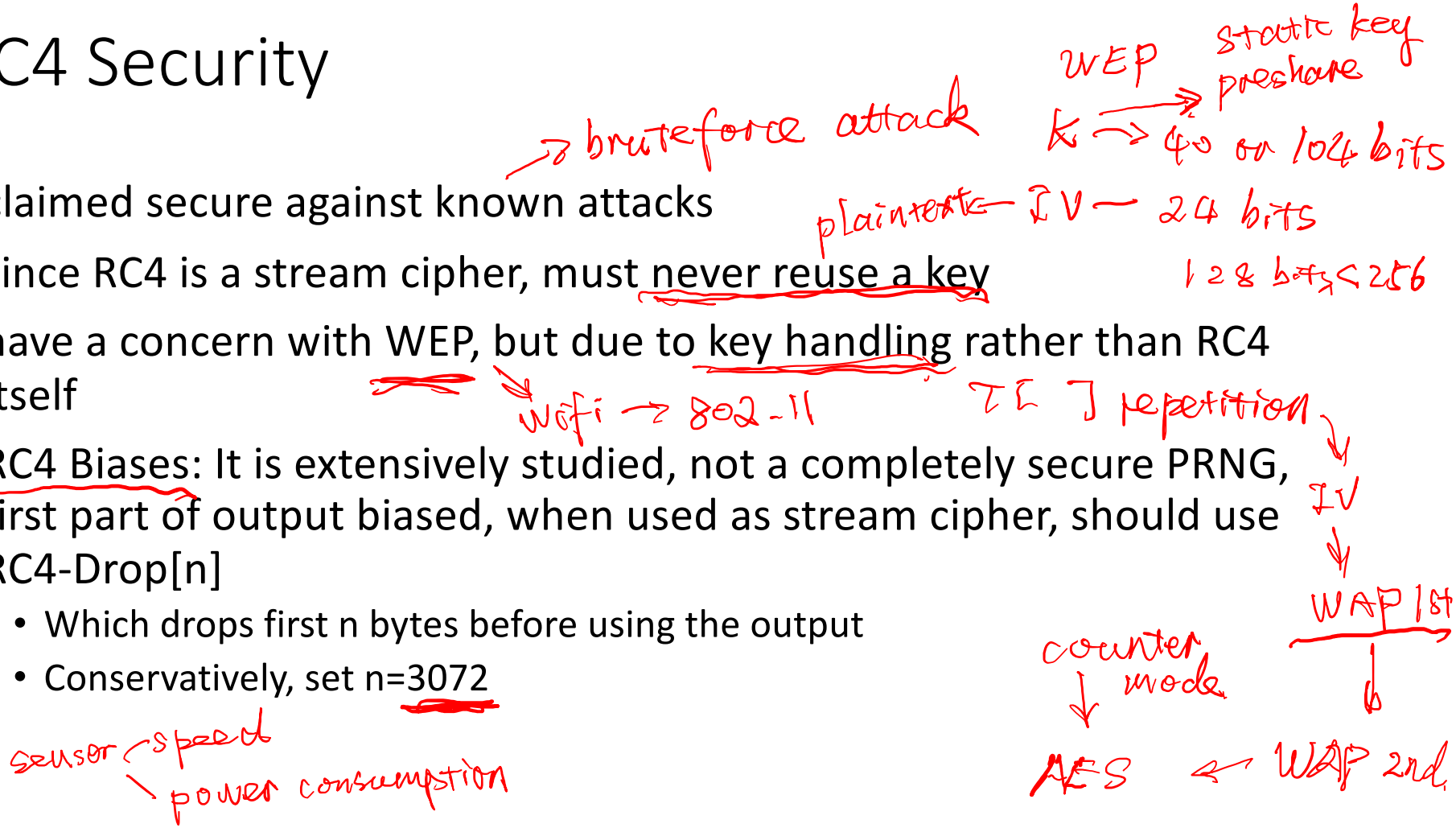
XOR

RC4


S
0
1
2
3
4
253
254
255
keylen
K
T
(a) Initial state of S and T
T
T[i]
j = j + S[i] + T[i]
S
S[i]
S[j]
i
Swap
(b) Initial permutation of S
j = j + S[i]
S
S[i]
S[j]
S[t]
i
Swap
t = S[i] + S[j]
k
(c) Stream generation

RC4 Security

- claimed secure against known attacks
- since RC4 is a stream cipher, must never reuse a key
- have a concern with WEP, but due to key handling rather than RC4 itself
- RC4 Biases: It is extensively studied, not a completely secure PRNG, first part of output biased, when used as stream cipher, should use RC4-Drop[n]
  - Which drops first n bytes before using the output
  - Conservatively, set n=3072



Summary – Chapter 2

- Symmetric block cipher
  - DES, 3DES
  - AES
- Random number
  - true random number
  - pseudorandom number
- Stream cipher
- The security of symmetric encryption depends on the secrecy of the key
- Symmetric encryption: pros and cons

Modular Arithmetic

- Definition (congruent modulo):
  - given b – a = km for some k ϵ Z, then a $\equiv b$ (mod m)
- Given a $\equiv b$ (mod m) and c $\equiv d$ (mod m), then
  - a + b ≡ c + d (mod m)
  - a - b ≡ c - d (mod m)
  - a + c ≡ b + d (mod m)
  - a × c ≡ b × d (mod m)
  - $a^k \equiv b^k$ (mod m)
  - ka = kb (mod m)
  - p(a) ≡ p(b) (mod m), any polynomial *p*(*x*) with integer coefficients
- $A \oplus B \oplus B$ = A

Thank you!

Network Security

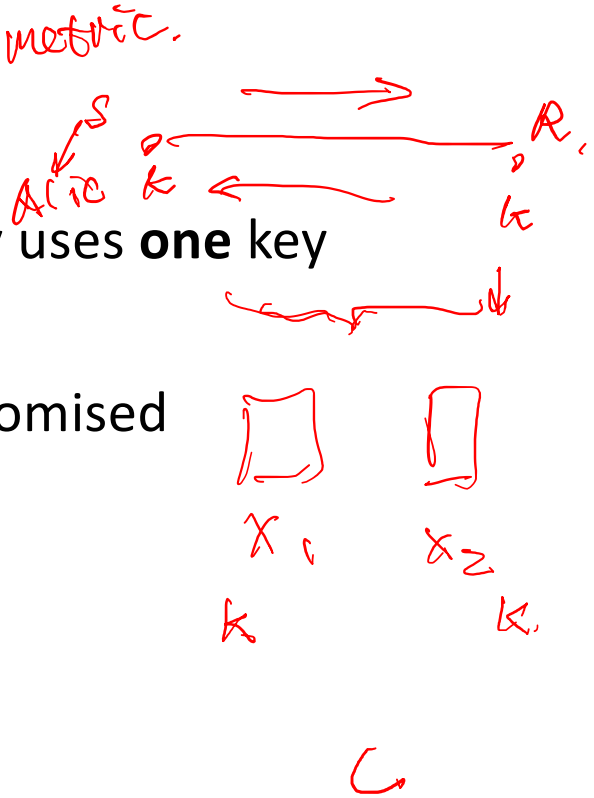
Chapter 3

Public-Key Cryptography and Message Authentication

Public-Key Cryptography

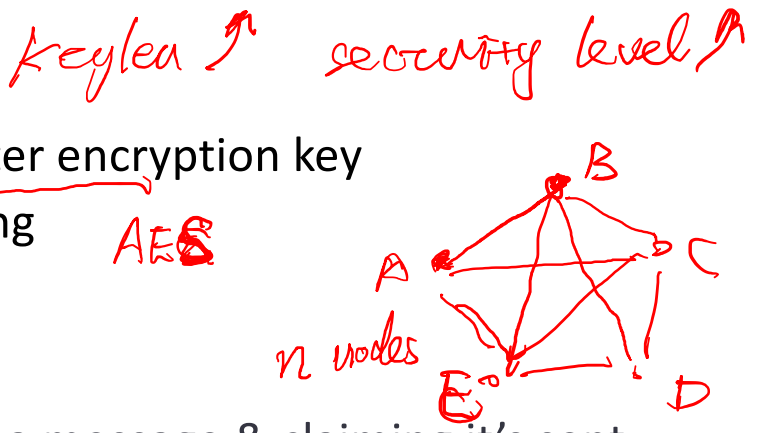
Conventional cryptography

- traditional **private/secret/single-key** cryptography uses **one** key
- shared by both sender and receiver
- if this key is disclosed, communications are compromised
- also is **symmetric**, parties are equal



Pros and cons

- Pros:
  - Encryption is fast for large amounts of data
  - Provide the same level of security with a shorter encryption key
  - By now, it's unbreakable to quantum computing
- Cons
  - Key distribution assumes a secure channel
  - Does not protect sender from receiver forging a message & claiming it's sent by sender
  - It does not scale well for large networks. It requires a separate key for each pair of communicating parties, which can result in a large number of keys to manage and protect.



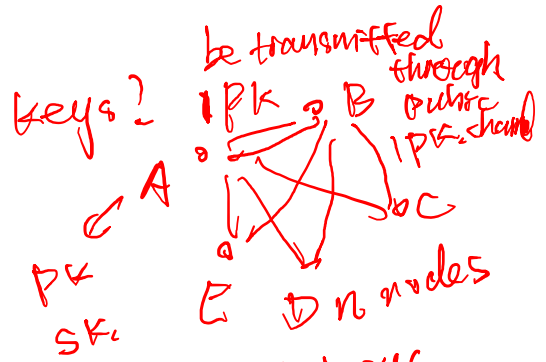
Homework 1 - individual

- Chapter 1 & 2
- **Deadline**: Tuesday, October 8, 11:59 PM
- Submit your homework via the provided link.
- The Google submission timestamp will be considered final.
- A 10% penalty will be applied for each day of late submission.

Review & Quiz I

- Chapter 1 & 2
- Wednesday (Oct. 9, 2024), in class
- Please ensure your participation
- No make-up quiz

Public-Key Cryptography



- In public-key schemes, each person has two keys
  - **Public key**: Known to everybody
  - **Private key**: Only known by that person
  - Keys come in pairs: every public key corresponds to one private key
- Uses number theory
  - Examples: Modular arithmetic, factoring, discrete logarithm problem, Elliptic logs over Elliptic Curves
  - Contrast with symmetric-key cryptography (uses XORs and bit-shifts)
- Messages are numbers
  - Contrast with symmetric-key cryptography (messages are bit strings)

Public-key Cryptography

- **Benefit:** No longer need to assume that Alice and Bob already share a secret → pk can be sent through public channel.
- **Drawback:** Much slower than symmetric-key cryptography
  - Number theory calculations are much slower than XORs and bit-shifts

· speed.

64 bits AES ⇒ 2¹⁵ RSA. ✗ quantum computer

Reading materials

- Encryption: Strengths and Weaknesses of Public-key Cryptography
- Public-key cryptography is a public invention due to Whitfield Diffie & Martin Hellman at Stanford Uni in 1976

Public-key cryptography

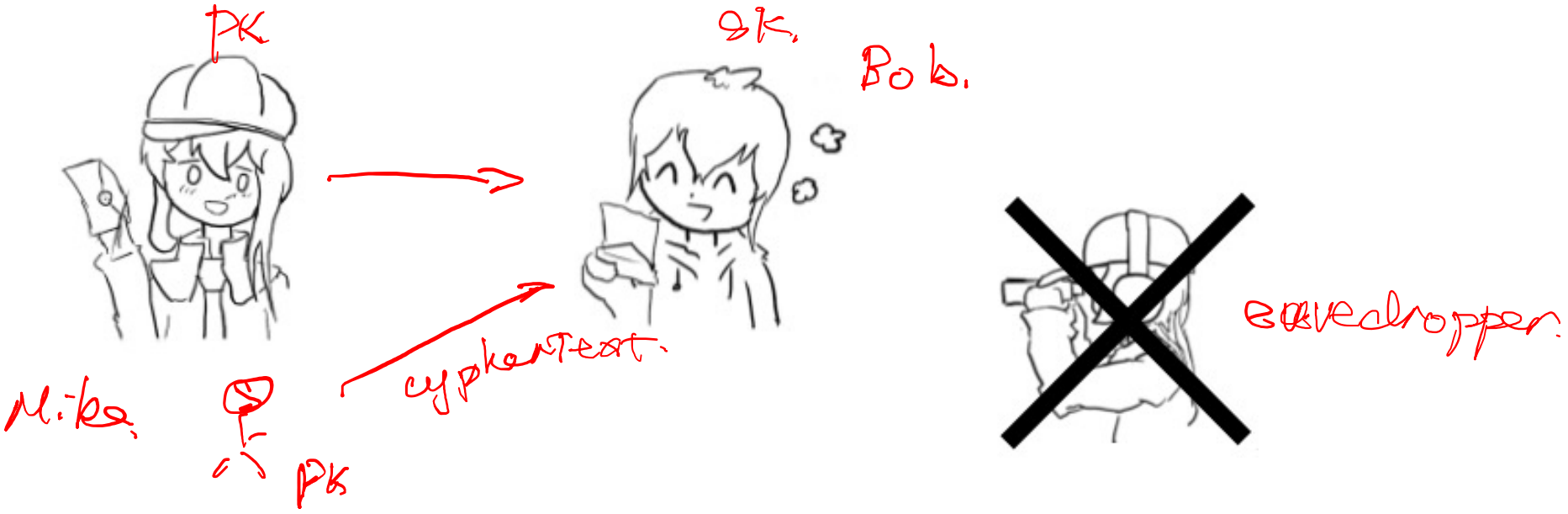
- **public-key/two-key/asymmetric** cryptography involves the use of **two** keys:
  - a **public-key**, which may be known by anybody, and can be used to **encrypt messages**, and **verify signatures**
  - a **private-key**, known only to the recipient, used to **decrypt messages**, and **sign** (create) **signatures**
- is **asymmetric** because
  - Not the same key
  - those who encrypt messages or verify signatures **cannot** decrypt messages or create signatures

PK

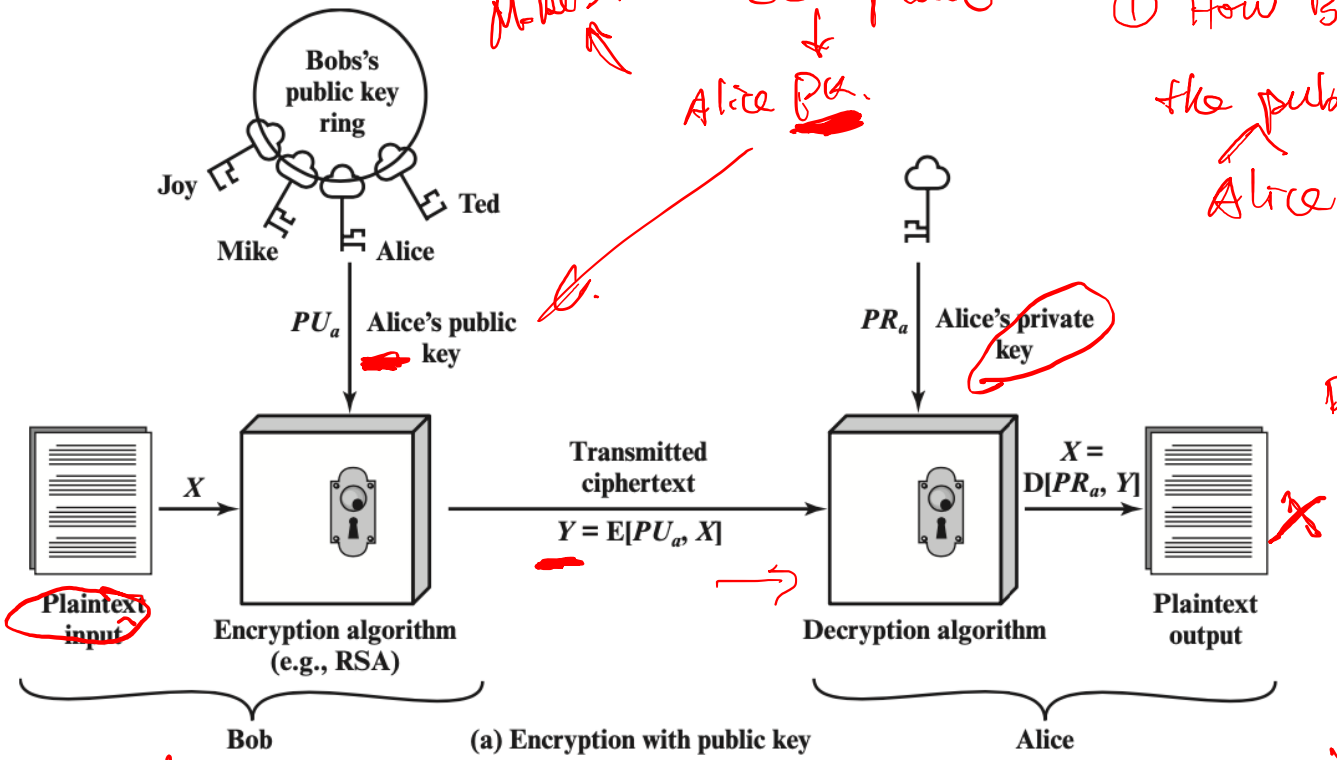
SK ↛ encrypt & verify

Public-Key Encryption

- Everybody can encrypt with the public key
- Only the recipient can decrypt with the private key



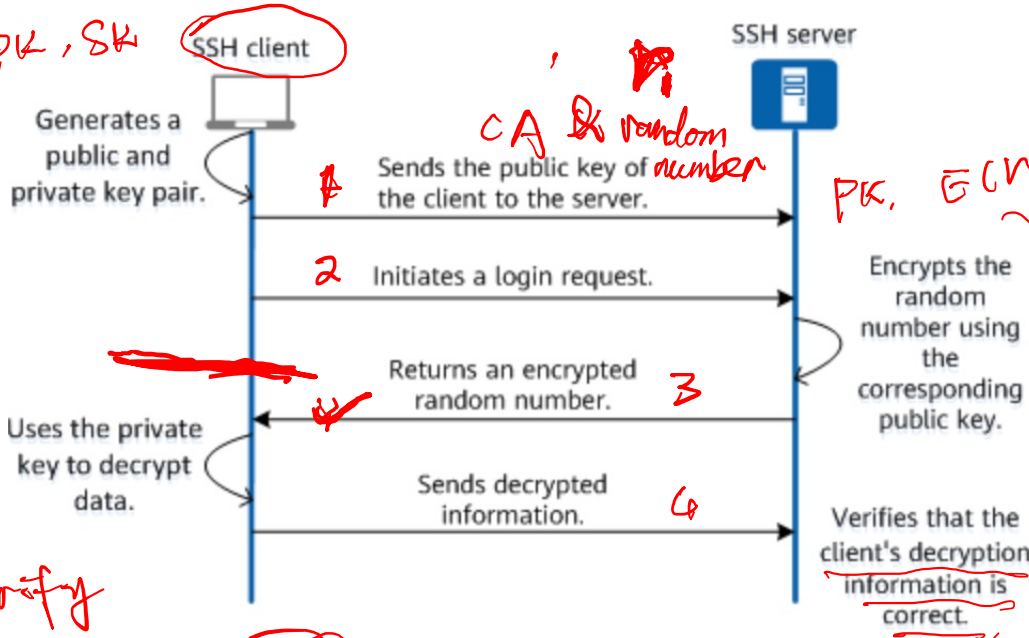
Public-Key Cryptography - Encryption



Encryption steps

- step1: generate a pair of keys
- step2: keep the private key / secret key (SK) and distribute the public key (PK) – place PK in a public register or other accessible file
- step3: Bob encrypts the message with Alice's PK
- step4: upon receiving the ciphertext (CT), Alice decrypt CT with SK

An example of key distribution



SSH key-based client authentication. Source: Peiyue and Yuanyuan 2021.

1.Peiyue, G. and F. Yuanyuan. 2021. "What Is SSH?" Info-Finder, Huawei, July 22. Updated 2021-12-14. Accessed 2023-04-18.

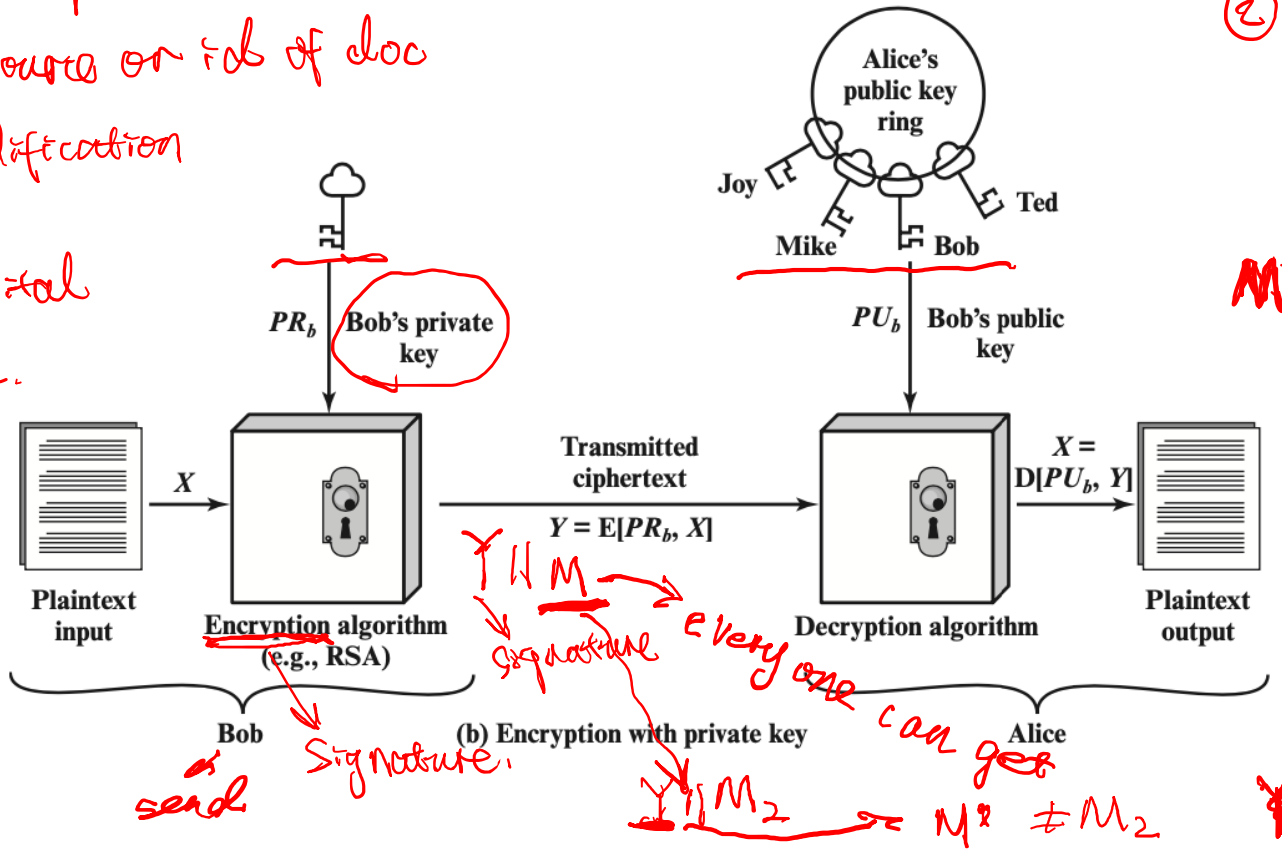
Encryption steps

- step1: generate a pair of keys
- step2: keep the private key / secret key (SK) and distribute the public key (PK) – place PK in a public register or other accessible file
- step3: Bob encrypts the message with Alice's PK
- step4: upon receiving the ciphertext (CT), Alice decrypt CT with SK

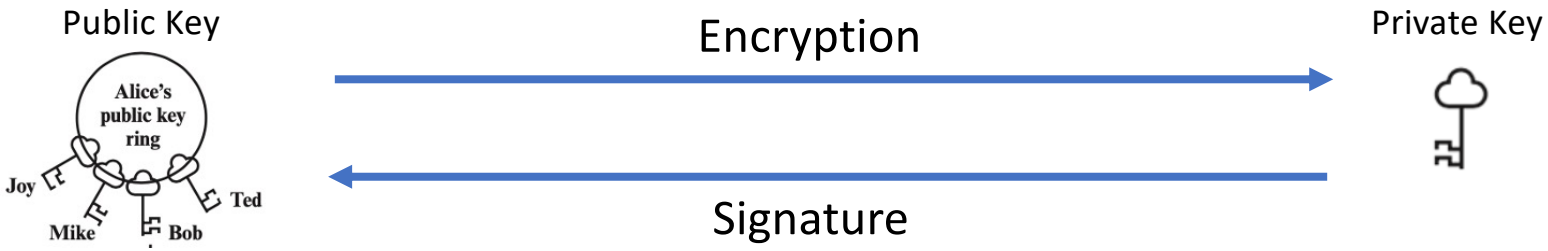
Public-Key Encryption: Definition

- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the public key, and *SK* is the private (secret) key
  - Enc(*PK*, *M*) → *C*: Encrypt a plaintext *M* using public key *PK* to produce ciphertext *C*
  - Dec(*SK*, *C*) → *M*: Decrypt a ciphertext *C* using secret key *SK*
- Properties
  - **Correctness**: Decrypting a ciphertext should result in the message that was originally encrypted
    - Dec(*SK*, Enc(*PK*, *M*)) = *M* for all *PK*, *SK* ← KeyGen() and *M*
  - **Efficiency**: Encryption/decryption should be fast
  - **Security**: 1. Alice (the challenger) just gives Eve (the adversary) the public key, and Eve doesn't request encryptions. Eve cannot guess out anything; 2. computationally infeasible to recover M with PK and ciphertext

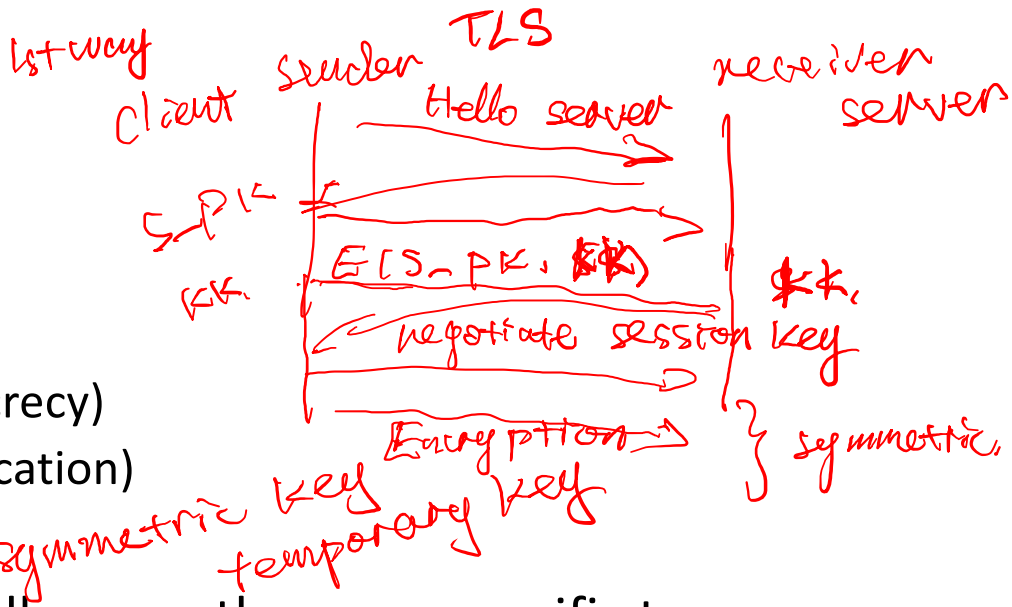
Public-Key Cryptography - Signature



Review


Public Key
Alice's public key ring
Joy
Ted
Mike
Bob
Encryption
Private Key
Signature

Public-Key application



- can classify uses into 3 categories:
  - **encryption/decryption** (provide secrecy)
  - **digital signatures** (provide authentication)
  - **key exchange** (of session keys)
- some algorithms are suitable for all uses; others are specific to one
- Either of the two related keys can be used for encryption, with the other used for decryption

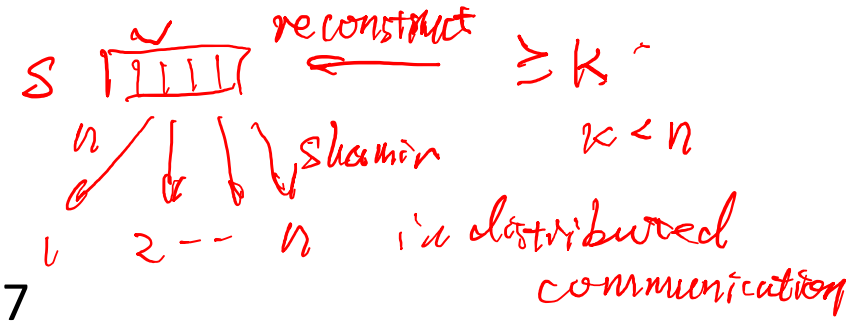
Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Diffie–Hellman	No	No	Yes
DSS	No	No	Yes
Elliptic curve	Yes	Yes	Yes

Security of Public Key Schemes

- Keys used are very large (>512bits)
  - like private key schemes brute force **exhaustive search** attack is always theoretically possible
- Security relies on a large enough difference in difficulty between easy (en/decrypt) and hard (cryptanalyze) problems
  - more generally the hard problem is known, it's just made too hard to do in practice
- Requires the use of very large numbers, hence is slow compared to private/symmetric key schemes

Public-Key Cryptography Algorithm
(RSA)

RSA Public-key encryption



- by Rivest, Shamir & Adleman of MIT in 1977
- currently the "work horse" of Internet security
  - most public key infrastructure (PKI) products
  - SSL/TLS: certificates and key-exchange
  - secure e-mail: PGP, Outlook, ….
- based on exponentiation in a finite (Galois) field over integers modulo a prime
  - exponentiation takes $O((\log n)^3)$ operations (easy)
- security due to cost of factoring large integer numbers
  - factorization takes $O(e^{\log n \log \log n})$ operations (hard)
- uses large integers (eg. 1024 bits)

RSA key setup

- each user generates a public/private key pair by:
  - selecting two large primes at random - p, q
  - computing their system modulus n=p·q
    - note ø(n)=(p-1)(q-1)
  - selecting at random the encryption key e
    - where 1<e<ø(n), gcd(e,ø(n))=1
  - solve following equation to find decryption key d
    - ed=1 mod ø(n)
  - publish their public encryption key: pk={e,n}
  - keep secret private decryption key: sk={d,p,q}

gcd → greatest common divisor

gcd(8, 12) = 4

```
int gcd(int a, int b) {
    if (a == 0)
        return b;
    return gcd(b % a, a)
}
```

remainder

integer

Euler's Totient function

pk.

coprime

relative prime.

Key Generation	
Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	gcd $(\phi(n), e) = 1; 1 < e < \phi(n)$
Calculate d	$de \bmod \phi(n) = 1$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

d exist?

if yes, how to get d?

gcd(9, 2) = 1

prime

gcd(24, 54)

```
24 | 54 | 2
   48
   ---
    6
```

gcd(6, 24)

```
6 | 24 | 4
  24
  --
   0
```

return 6

$gcd(e, \phi(n)) = 1 \mod \phi(n)$.

To prove, use contradiction

Assume $gcd(e, \phi(n)) = h > 1$ $\quad h \in \mathbb{Z}$

By definition

$e = k_1 \cdot h \quad k_1 \in \mathbb{Z}$ ①

$\phi(n) = k_2 \cdot h \quad k_2 \in \mathbb{Z}$ ②

$\because \quad ed = 1 \mod \phi(n)$

$\Rightarrow \quad ed = 1 + k \cdot \phi(n) \quad k \in \mathbb{Z}$

By definition of modular

substitute e and $\phi(n)$ with ①, ②

$k_1 \cdot h d = 1 + k \cdot k_2 \cdot h$

A multiple of h — A multiple of h — A multiple of h

$\Rightarrow$ contradiction

Inverse algorithm

1. Extended Euclidean Algorithm

$ed + k \cdot \phi(n) = 1 \mod \phi(n)$

2. $1 \le d < \phi(n)$ & $d \in \mathbb{Z}$

```
for(int d=1; d<φ(n); d++)
    ed=1 mod φ(n)
```

RSA example

Key Generation	
Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n), e) = 1; 1 < e < \phi(n)$
Calculate d	$de \bmod \phi(n) = 1$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

1. Select primes: $p=17$ & $q=11$
2. Compute $n = pq = 17 \times 11 = 187$
3. Compute $\phi(n) = (p-1)(q-1) = 16 \times 10 = 160$
4. Select e: $\gcd(e,160)=1$; choose $e=7$
5. Determine d: $de=1 \bmod 160$ and $d < 160$ Value is $d=23$ since $23 \times 7 = 161 = 10 \times 160 + 1$
6. Publish public key $pk=\{7,187\}$
7. Keep secret private key $sk=\{23,17,11\}$

RSA example

Key Generation	
Select p, q	p and q both prime, $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n), e) = 1; 1 < e < \phi(n)$
Calculate d	$de \bmod \phi(n) = 1$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

1. Select primes: $p=17$ & $q=11$
2. Compute $n = pq = 17 \times 11 = 187$
3. Compute $\phi(n) = (p-1)(q-1) = 16 \times 10 = 160$
4. Select e : $\gcd(e, 160) = 1$; choose $e=7$
5. Determine d: $de = 1 \bmod 160$ and $d < 160$ Value is $d=23$ since $23 \times 7 = 161 = 10 \times 160 + 1$
6. Publish public key $pk = \{7, 187\}$
7. Keep secret private key $sk = \{23, 17, 11\}$

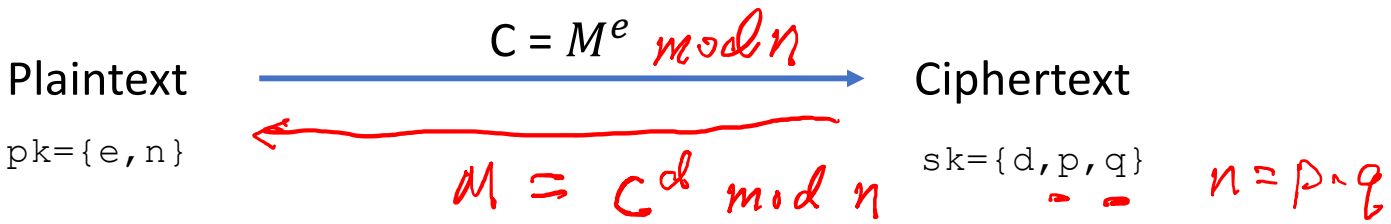
RSA use

modular exponentiation

- to encrypt a message M the sender:
  - obtains **public key** of recipient `pk={e,n}`
  - computes: $C=M^e \bmod n$, where $0 \le M < n$
- to decrypt the ciphertext C the owner:
  - uses their private key `sk={d,p,q}`
  - computes: $M=C^d \bmod n$
- note that the message M must be smaller than the modulus n (block if needed)

Encryption	
Plaintext:	$M < n$
Ciphertext:	$C = M^e \pmod n$

Decryption	
Ciphertext:	C
Plaintext:	$M = C^d \pmod n$



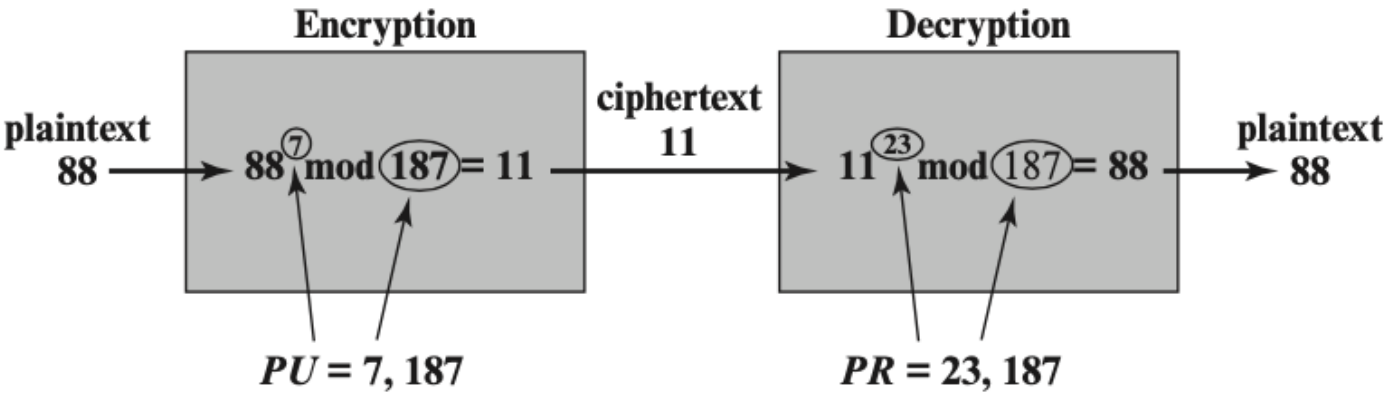
RSA example continue

- sample RSA encryption/decryption is:
- given message $M = 88$ ($88<187$)
- encryption:

  $C = 88^7 \bmod 187 = 11$
- decryption:

  $M = 11^{23} \bmod 187 = 88$

Example of RSA algorithm



RSA key generation

- users of RSA must:
  - determine two primes at random - p, q
  - select either e or d and compute the other
- primes p, q must not be easily derived from modulus n=p.q
  - means must be sufficiently large
  - typically guess and use probabilistic test
- exponents e, d are inverses, so use Inverse algorithm to compute the other

Correctness of RSA

- Euler's theorem: if gcd (M, n) = 1, then $M^{\phi(n)} = 1 \bmod n$. Here $\phi(n)$ is Euler's totient function: the number of integers in {1, 2, . . ., n-1} which are relatively prime to n. When n is a prime, this theorem is just Fermat's little theorem

$$\begin{aligned} M' = C^d \bmod n &= M^{ed} \bmod n \\ &= M^{k\phi(n)+1} \bmod n \\ &= [M^{\phi(n)}]^k \cdot M \bmod n \\ &= M \bmod n \end{aligned}$$

Encryption	
Plaintext:	$M < n$
Ciphertext:	$C = M^e \pmod n$

$M \xrightarrow[e]{C = M^e \bmod n} C \xrightarrow{d} M'$

To prove $M' = M$

$$M' = C^d \bmod n$$
$$= (M^e)^d \bmod n$$
$$= M^{ed} \bmod n \quad (1)$$

$$e \cdot d = 1 \bmod \phi(n). \quad (2)$$

By definition of modular arithmetic.

$(2) \Rightarrow \quad ed = 1 + k \cdot \phi(n) \quad k \in \mathbb{Z} \quad (3)$

$$(1) = M^{k \cdot \phi(n) + 1} \bmod n$$
$$= M^{k \cdot \phi(n)} \cdot M \bmod n$$
$$= \left(M^{\phi(n)}\right)^k \cdot M \bmod n \quad (4)$$

By Euler's theorem

$\because$ if $\gcd(M, \phi(n)) = 1$

$$M^{\phi(n)} \bmod n = 1$$

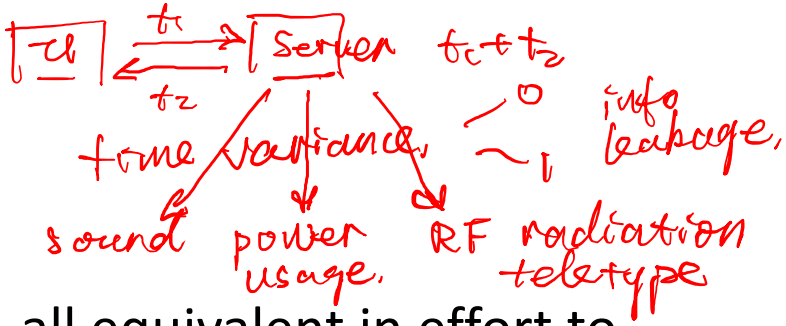
if $M << n$, n is large number

$$\Rightarrow M^{\phi(n)} \bmod n = 1 \quad (5)$$

$(4) = (1)^k \cdot M \bmod n$

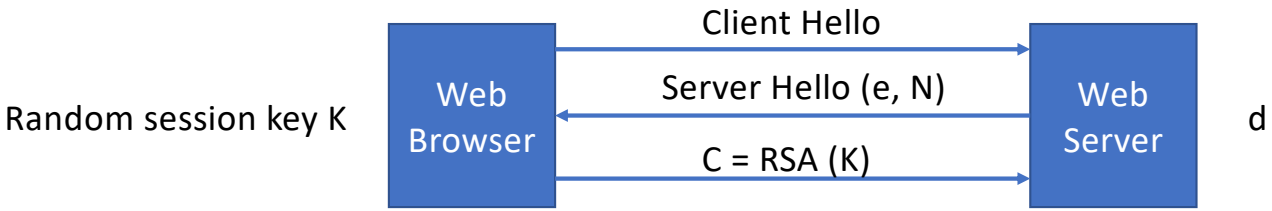
$M' = M$ ∎

Attack approaches



- **Mathematical attacks**: several approaches, all equivalent in effort to factoring the product of two primes. The defense against mathematical attacks is to use a large key size.
- **Timing attacks**: These depend on the running time of the decryption algorithm
- **Chosen ciphertext attacks**: this type of attacks exploits properties of the RSA algorithm by selecting blocks of data. These attacks can be thwarted by suitable padding of the plaintext, such as PKCS1 V1.5 in SSL

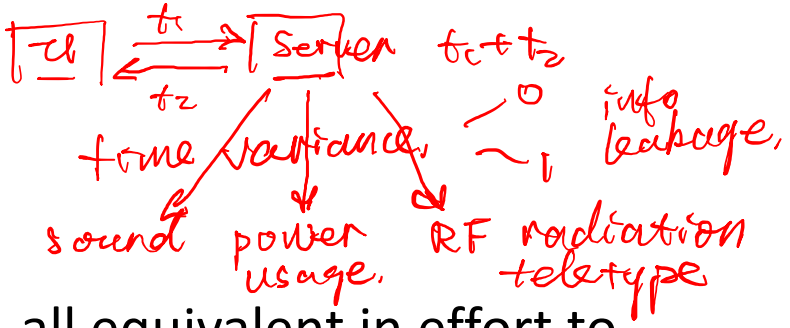
A simple attack on textbook RSA



- Session-key K is 64 bits. View $K \in \{0,\ldots,2^{64}\}$
  - Eavesdropper sees: $C = K^e \pmod N$.
- Suppose $\mathbf{K = K_1 \cdot K_2}$ where $K_1, K_2 < 2^{34}$.
  - Then: $\mathbf{C/K_1^e = K_2^e} \pmod N$
- Build table: $C/1^e, C/2^e, C/3^e, \ldots, C/2^{34e}$. time: 2^{34}

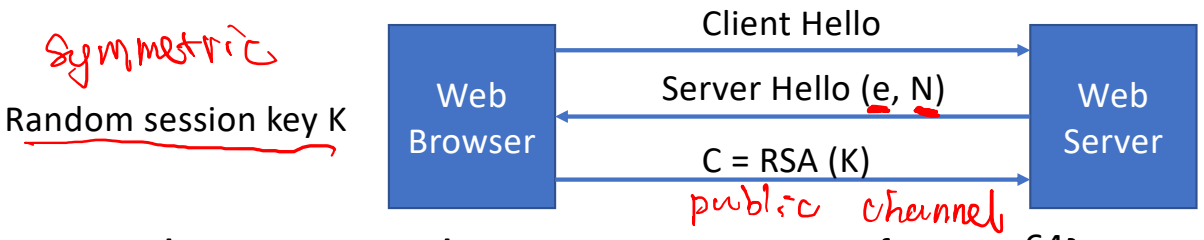
  For $K_2 = 0,\ldots, 2^{34}$ test if K_2^e is in table. time: $2^{34} \cdot 34$
- Attack time: $\approx 2^{40} << 2^{64}$

Attack approaches



- **Mathematical attacks**: several approaches, all equivalent in effort to factoring the product of two primes. The defense against mathematical attacks is to use a large key size.
- **Timing attacks**: These depend on the running time of the decryption algorithm
- **Chosen ciphertext attacks**: this type of attacks exploits properties of the RSA algorithm by selecting blocks of data. These attacks can be thwarted by suitable padding of the plaintext, such as PKCS1 V1.5 in SSL

A simple attack on textbook RSA



- Session-key K is 64 bits. View $K \in \{0,\ldots,2^{64}\}$
  - Eavesdropper sees: $C = K^e \pmod N$.
- Suppose $\mathbf{K = K_1 \cdot K_2}$ where $K_1, K_2 < 2^{34}$.
  - Then: $\mathbf{C/K_1^e = K_2^e} \pmod N$
- Build table: $C/1^e, C/2^e, C/3^e, \ldots, C/2^{34e}$. time: 2^{34}

  For $K_2 = 0,\ldots, 2^{34}$ test if K_2^e is in table. time: $2^{34} \cdot 34$
- Attack time: $\approx 2^{40} << 2^{64}$

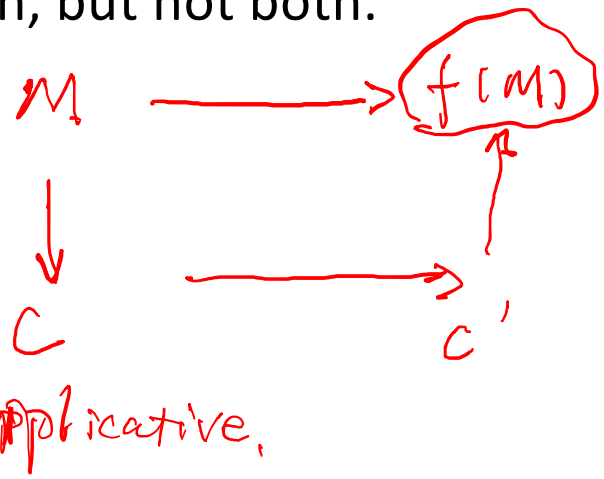
Take-home exercise – no need to submit

- SW textbook (6th edition) problems: 3.14 & 3.15

Homomorphic encryption

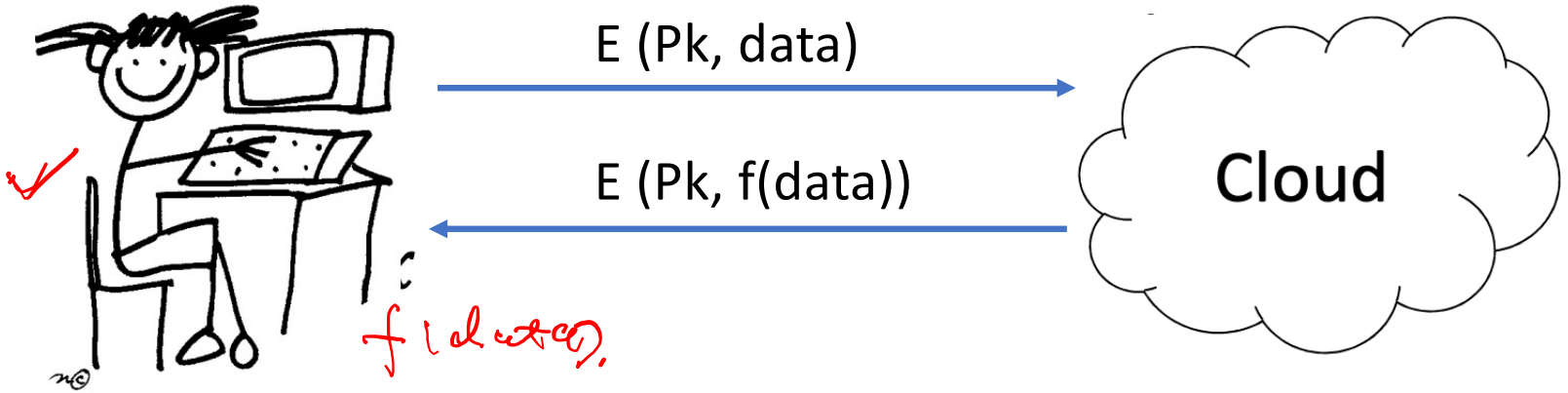
- Encryption scheme that allows computation on ciphertexts
  - an extension of public-key encryption scheme that allows anyone in possession of the public key to perform operations on encrypted data without access to the decryption key
- Partially Homomorphic Encryption: Initial public-key systems that allow this for either addition or multiplication, but not both.
  - i.e. RSA
- Fully homomorphic encryption (FHE)

$$E(m_1) \cdot E(m_2) = m_1^e \cdot m_2^e \bmod n$$
$$= (m_1 \cdot m_2)^e \bmod n$$
$$= E(m_1 \cdot m_2)$$



Application of homomorphic encryption

- One Use case: cloud computing
  - A weak computational device Alice (e.g., a mobile phone or a laptop) wishes to perform a computationally heavy task, beyond her computational means. She can delegate it to a much stronger (but still feasible) machine Bob (the cloud, or a supercomputer) who offers the service of doing so. The problem is that Alice does not trust Bob, who may give the wrong answer due to laziness, fault, or malice.



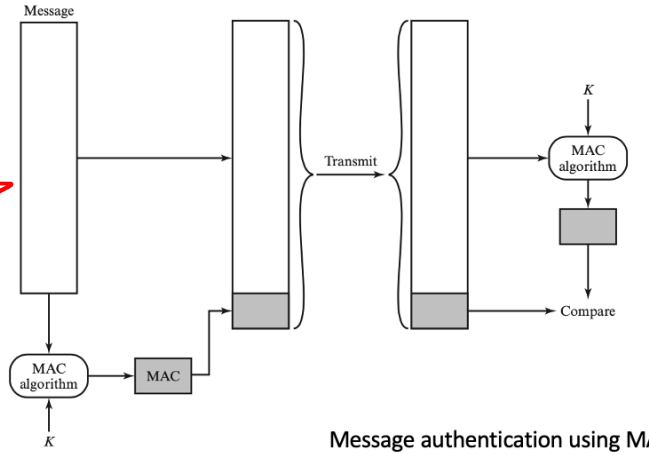
RSA reading materials

- A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

Message Authentication

Message authentication

- message authentication is concerned with:
  - protecting the integrity of a message
  - validating identity of originator
  - non-repudiation of origin (dispute resolution)
- then three alternative functions used:
  - message encryption – symmetric
  - message authentication code (MAC)
  - digital signature



Message authentication using MAC

Message encryption

- Symmetric message encryption by itself also provides a measure of authentication
- if symmetric encryption is used then:
  - receiver knows sender must have created it
  - since only sender and receiver know key used
  - know content cannot be altered

Yes
the sender encrypted msg.
& use the same key
only belongs to the sender

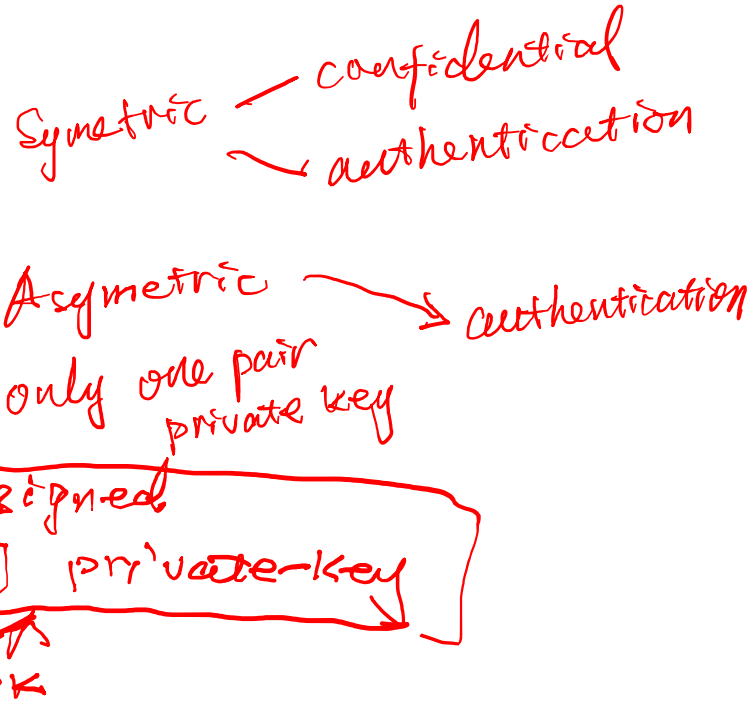
group key

Homework 1 questions

- Symmetric Block Cypher provides authentication and confidentiality
  - Ans: True

Message encryption

- if public-key encryption is used:
  - encryption provides no confidence of sender
  - since anyone potentially knows public-key
  - so, need to recognize corrupted messages
  - however, if
    - sender **signs** message using their private-key
    - then encrypts with recipients' public key
    - have both secrecy and authentication
  - but at cost of two public-key uses on message



Reasons to avoid encryption authentication

- Encryption software is quite slow
- Encryption hardware costs are nonnegligible
- Encryption hardware is optimized toward large data sizes
- An encryption algorithm may be protected by a patent

RC4

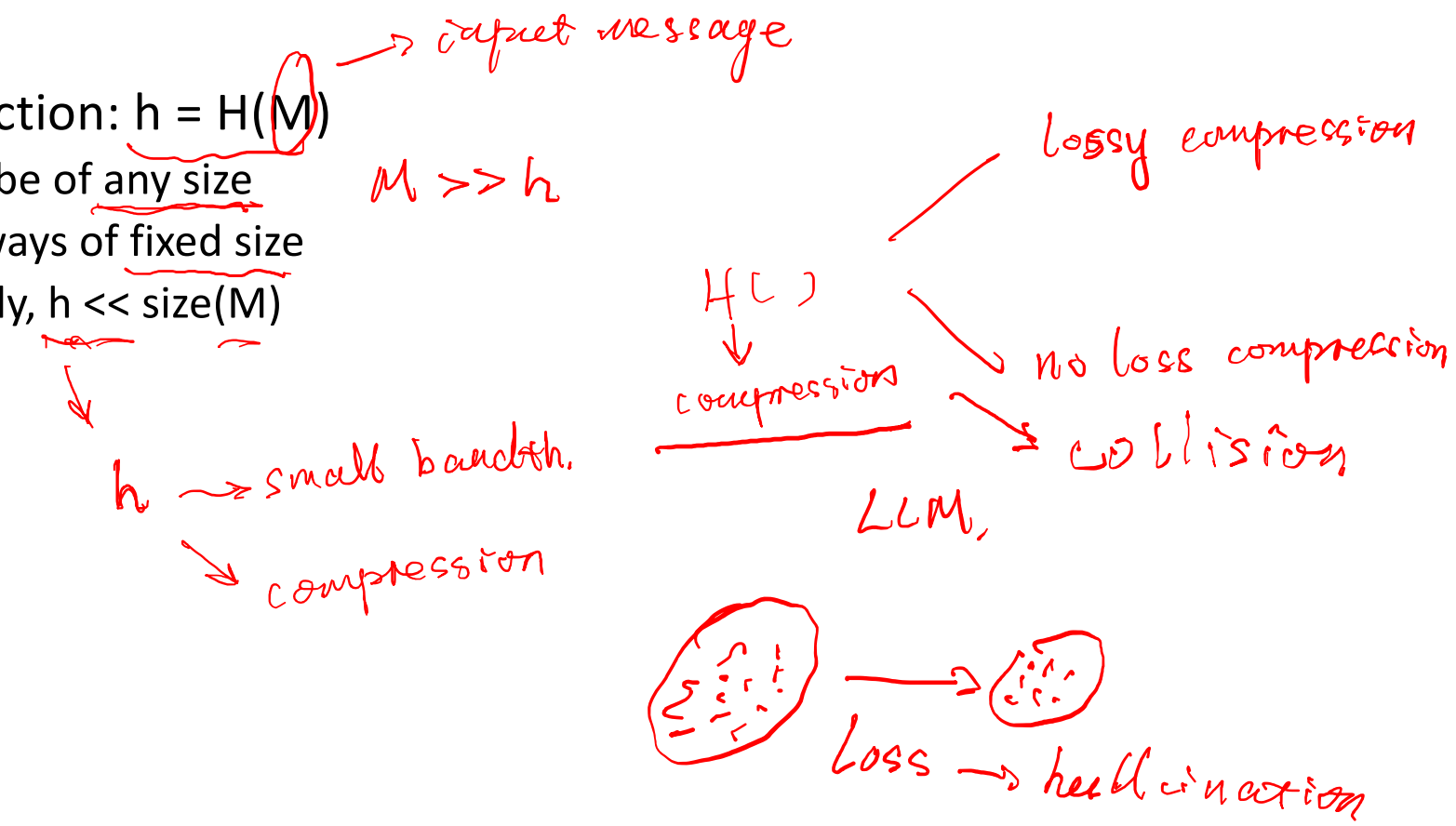
① MAC (Hash function)

② Digital Signature

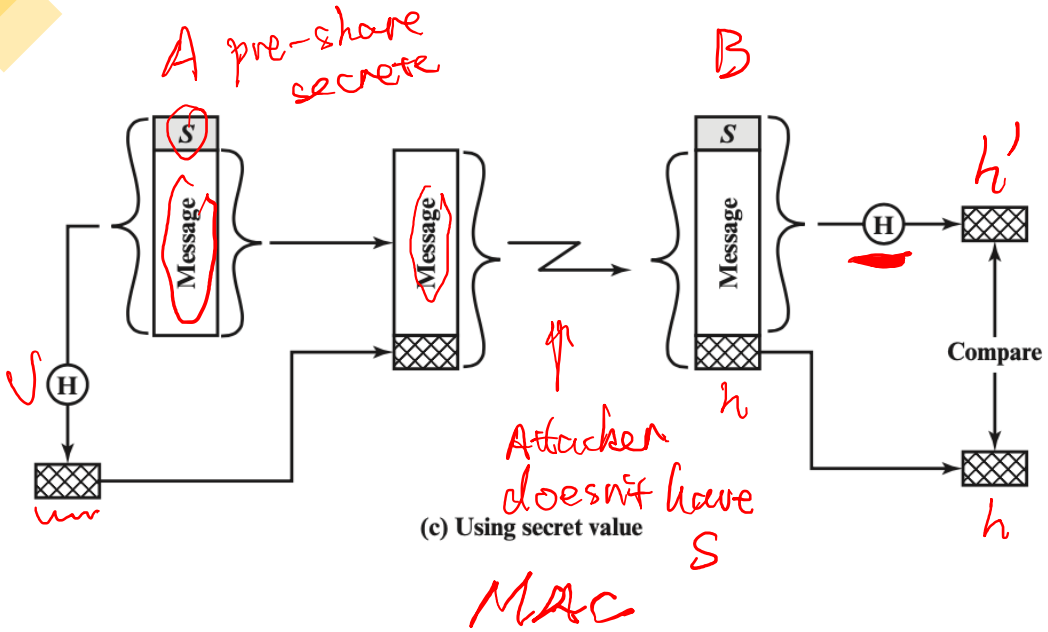
Hash Function

Hash functions

- Hash function: h = H(M)
  - M can be of any size
  - h is always of fixed size
  - Typically, h << size(M)



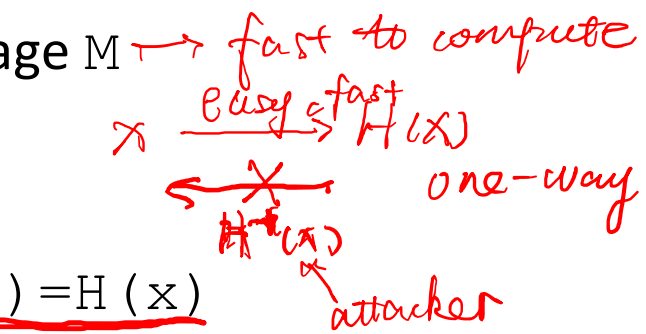
One use case - using hash function



- Initialization: A and B share a common secret, S_{AB}
- Message, M
- A calculates $MD_M = H(S_{AB} || M)$
- B recalculates MD'_M, and check
- $MD'_M = MD_M$

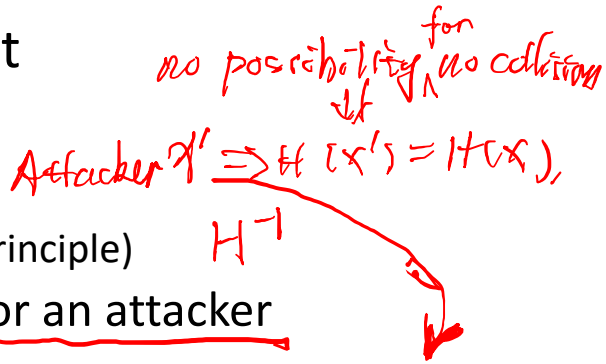
Requirements for secure hash functions

- 1. can be applied to any sized message `M`
- 2. produces fixed-length output `h`
- 3. is easy to compute `h=H(M)` for any message `M`
- 4. given `h` is infeasible to find `x` s.t. `H(x)=h`
  - one-way property or preimage resistance
- 5. given `x` is infeasible to find `x'` s.t. `H(x')=H(x)`
  - weak collision resistance or second pre-image resistant
- 6. infeasible to find any pair of `x, x'` s.t. `H(x')=H(x)`
  - strong collision resistance



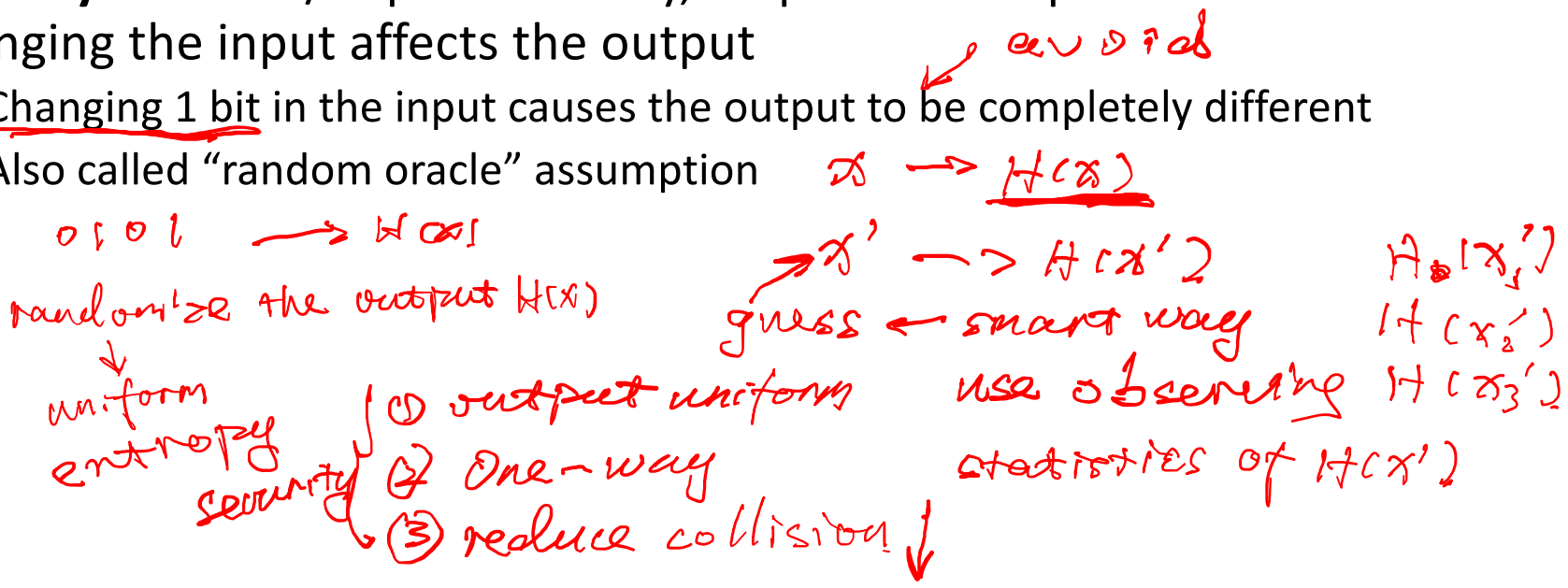
Hash Function: Collision Resistance

- **Collision**: Two different inputs with the same output
  - $x \neq x'$ and $H(x) = H(x')$
  - Can we design a hash function with no collisions?
    - No, because there are more inputs than outputs (pigeonhole principle)
  - However, we want to make finding collisions *infeasible* for an attacker
- **Collision resistance**: It is infeasible to (i.e. no polynomial time attacker can) find any pair of inputs $x' \neq x$ such that $H(x) = H(x')$



Secure hash function

- A hash function that satisfies the first five properties is referred to as a weak hash function
- **Security:** random/unpredictability, no predictable patterns for how changing the input affects the output
  - Changing 1 bit in the input causes the output to be completely different
  - Also called "random oracle" assumption



Secure hash function

- A hash function that satisfies the first five properties is referred to as a weak hash function
- **Security:** random/unpredictability, no predictable patterns for how changing the input affects the output
  - Changing 1 bit in the input causes the output to be completely different
  - Also called “random oracle” assumption
- A message digest
  - a cryptographic hash function containing a string of digits created by a one-way hashing formula
  - provides data integrity
- Examples: SHA-1 (Secure Hash Algorithm 1), SHA-2, SHA-3, MD5

Hash Function: Examples

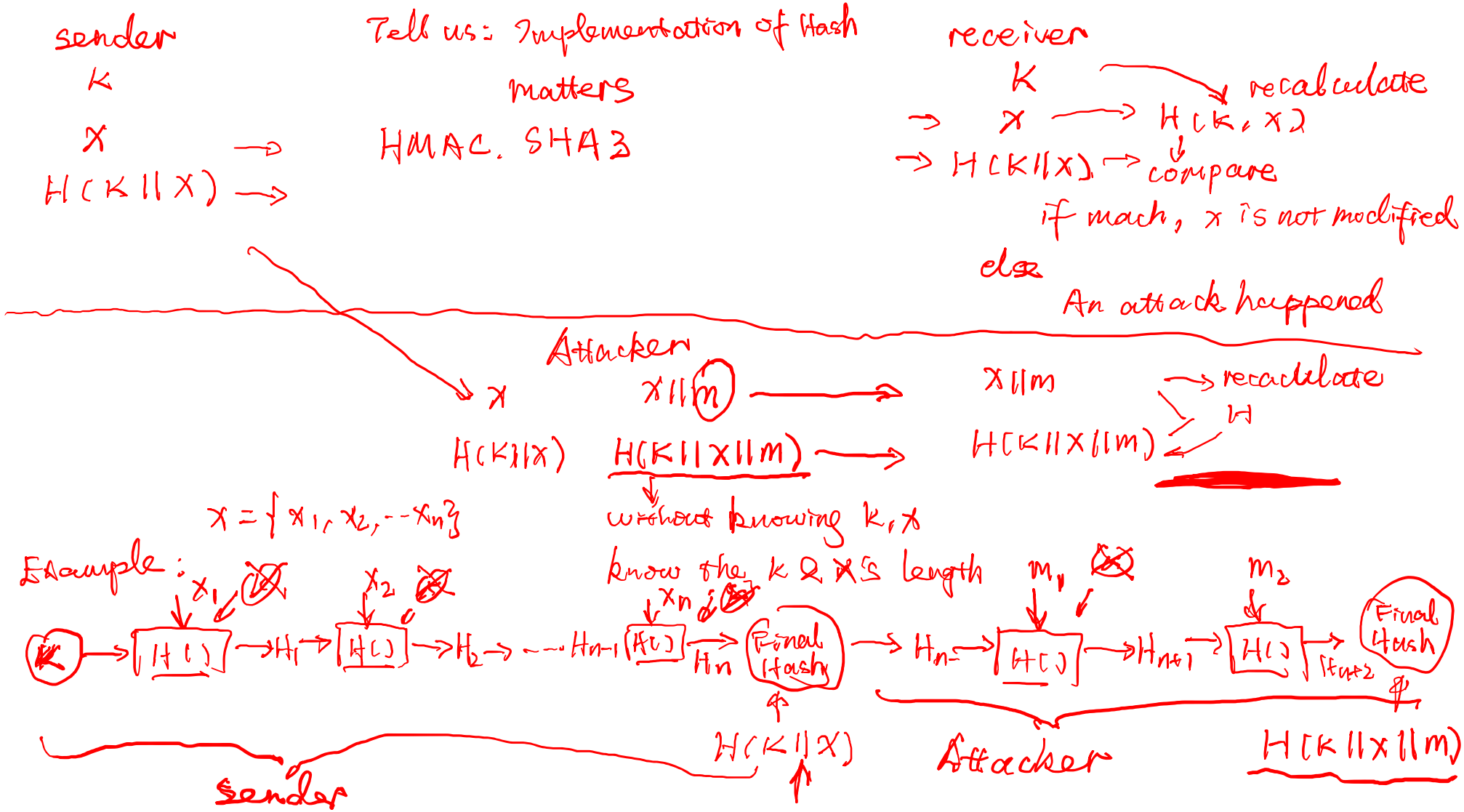
- MD5
  - Output: 128 bits
  - Security: Completely broken
- SHA-1
  - Output: 160 bits
  - Security: Completely broken in 2017
  - Was known to be weak before 2017, but still used sometimes
- SHA-2
  - Output: 256, 384, or 512 bits (sometimes labeled SHA-256, SHA-384, SHA-512)
  - Not currently broken, but some variants are vulnerable to a length extension attack
  - Current standard
- SHA-3 (Keccak)
  - Output: 256, 384, or 512 bits
  - Current standard (not meant to replace SHA-2, just a different construction)

Length Extension Attacks

- **Length extension attack**: Given $H(x)$ and the length of x, but not x, an attacker can create $H(x \,||\, m)$ for any m of the attacker's choosing
  - Length extension attack - Wikipedia
- SHA-256 (256-bit version of SHA-2) is vulnerable
- SHA-3 is not vulnerable

Length Extension Attacks

- **Length extension attack**: Given $H(x)$ and the length of x, but not x, an attacker can create $H(x \,||\, m)$ for any m of the attacker's choosing
  - Length extension attack - Wikipedia
- SHA-256 (256-bit version of SHA-2) is vulnerable
- SHA-3 is not vulnerable


sender
K
X
H(K||X)
Tell us: Implementation of Hash matters
HMAC. SHA3
receiver
K
X
H(K||X)
recalculate
H(K, X)
compare
if mach, x is not modified
else
An attack happened
Attacker
X
H(K||X)
X||m
H(K||X||m)
X||m
recalculate H
H(K||X||m)
X = {x1, x2, --xn}
without knowing K, X
know the K & X's length
Example:
x1
x2
xn
m1
m2
K
H()
H1
H2
Hn-1
Hn
Final Hash
Hn+1
Hn+2
Final Hash
H(K||X)
Attacker
H(K||X||m)
sender

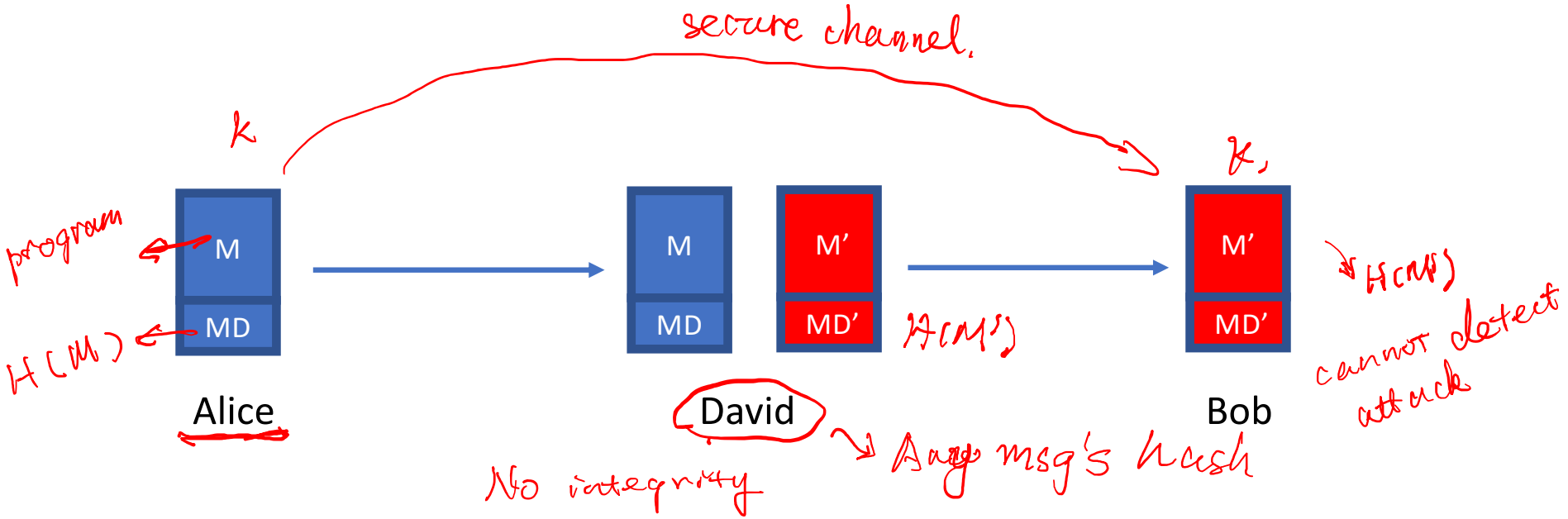
Does hashes provide integrity?

- It depends on your threat model
- Scenario
  - Mozilla publishes a new version of Firefox on some download servers
  - Alice downloads the program binary
  - How can she be sure that nobody tampered with the program?
- Idea: use cryptographic hashes
  - Mozilla hashes the program binary and publishes the hash on its website
  - Alice hashes the binary she downloaded and checks that it matches the hash on the website
  - If Alice downloaded a malicious program, the hash would not match (tampering detected!)
  - An attacker can't create a malicious program with the same hash (collision resistance)
- Threat model: We assume the attacker cannot modify the hash on the website
  - We have integrity, as long as we can communicate the hash securely

Do hashes provide integrity?

- It depends on your threat model
- Scenario
  - Alice and Bob want to communicate over an insecure channel
  - David might tamper with messages
- Idea: Use cryptographic hashes
  - Alice sends her message with a cryptographic hash over the channel
  - Bob receives the message and computes a hash on the message
  - Bob checks that the hash he computed matches the hash sent by Alice
- Threat model: David can modify the message *and the hash*
  - No integrity!

Man-in-the-middle attack



Do hashes provide integrity?

- It depends on your threat model
- If the attacker can modify the hash, hashes don't provide integrity
- Main issue: Hashes are *unkeyed* functions
  - There is no secret key being used as input, so any attacker can compute a hash on any value

Solutions

- A message digest created using a secret symmetric key is known as a Message Authentication Code (MAC), because it can provide assurance that the message has not been modified
- The sender can also generate a message digest and then encrypt the digest using the private key of an asymmetric key pair, forming a digital signature. The signature must then be verified by the receiver through comparing it with a locally generated digest

Hashes: Summary

- Map arbitrary-length input to fixed-length output
- Output is deterministic
- Security properties
  - One way: Given an output y, it is infeasible to find any input x such that $H(x) = y$.
  - Second preimage resistant: Given an input x, it is infeasible to find another input $x' \neq x$ such that $H(x) = H(x')$.
  - Collision resistant: It is infeasible to find any pair of inputs $x' \neq x$ such that $H(x) = H(x')$.
  - Randomized output
- Some hashes are vulnerable to length extension attacks
- Hashes don't provide integrity (unless you can publish the hash securely)

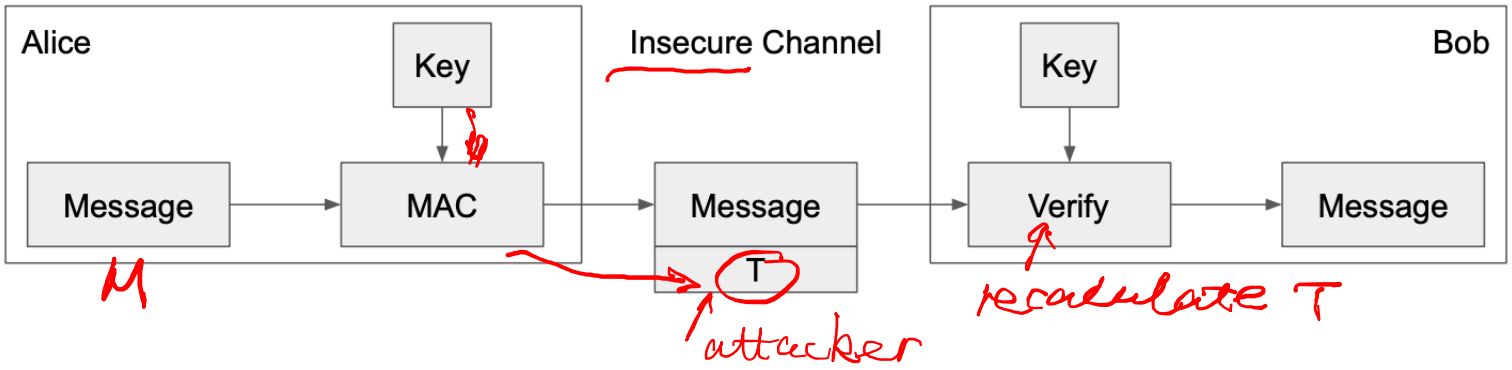
Message Authentication Code

Message authentication code (MAC)

- generated by an algorithm that creates a small fixed-sized block
  - depending on both message and some key
  - not be reversible
  - $MAC_M = F(K_{AB}, M)$
- appended to message as a signature
- receiver performs same computation on message and checks it matches the MAC
- provides assurance that message is unaltered and comes from sender

MACs: Usage

- Alice wants to send *M* to Bob, but doesn't want David to tamper with it
- Alice sends *M* and *T* = MAC(*K*, *M*) to Bob
- Bob receives *M* and *T*
- Bob computes MAC(*K*, *M*) and checks that it matches *T*
- If the MACs match, Bob is confident the message has not been tampered with (integrity)



MACs: Definition

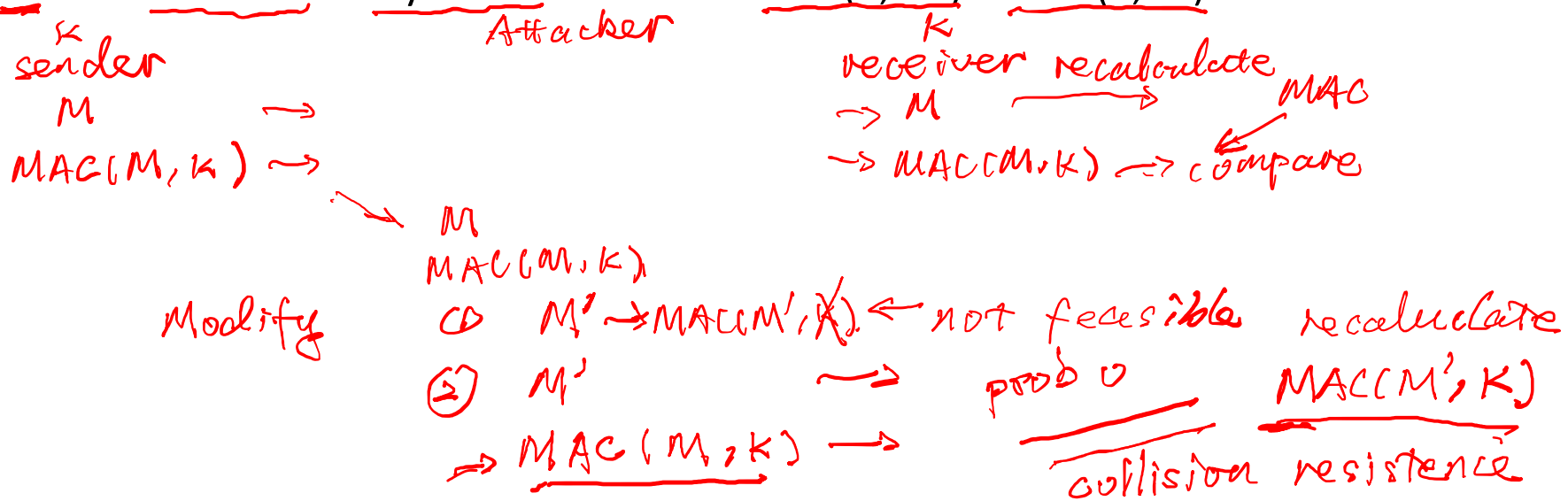
- Two parts:
  - KeyGen() → *K*: Generate a key *K*
  - MAC(*K*, *M*) → *T*: Generate a tag *T* for the message *M* using key *K*
    - Inputs: A secret key and an arbitrary-length message
    - Output: A fixed-length **tag** on the message
- Properties
  - **Correctness**: Determinism
    - Note: Some more complicated MAC schemes have an additional Verify(*K*, *M*, *T*) function that don't require determinism, but this is out of scope
  - **Efficiency**: Computing a MAC should be efficient
  - **Security**: existentially unforgeable under chosen plaintext attack

Mid-term Exam

- Nov. 6, 2024 (Wednesday), 12:00 pm – 12:50 pm, in class
- Closed book, but you're allowed to bring one cheat sheet (1 A4-sized paper)
- Chapter 1 – 3
- Will have a review class on Nov. 1st , during class

Existentially unforgeable

- A secure MAC is **existentially unforgeable**: without the key, an attacker cannot create a valid tag on a message
  - David cannot generate MAC(*K*, *M*') without *K*
  - David cannot find any *M*' ≠ *M* such that MAC(*K*, *M*') = MAC(*K*, *M*)



Example: HMAC

- issued as RFC 2104 [1]
- has been chosen as the mandatory-to-implement MAC for IP Security
- Used in Transport Layer Security (TLS) and Secure Electronic Transaction (SET)

[1] “HMAC: Keyed-Hashing for Message Authentication”, RFC 2104, https://datatracker.ietf.org/doc/html/rfc2104

HMAC(K, M)

- will produce two keys to increase security
- If key is longer than the desired size, we can hash it first, but be careful with using keys that are too much smaller, they have to have enough randomness in them
- Output $H[(K^+ \oplus opad) \,||\, H[(K^+ \oplus ipad) \,||\, M]]$

Example: HMAC

- HMAC(K, M):
  - Output $H[(K^+ \oplus opad) \,||\, H[(K^+ \oplus ipad) \,||\, M]]$
- Use K to derive two different keys
  - *opad* (outer pad) is the hard-coded byte **0x5c** repeated until it's the same length as K^+
  - *ipad* (inner pad) is the hard-coded byte **0x36** repeated until it's the same length as K^+
  - As long as *opad* and *ipad* are different, you'll get two different keys
  - For paranoia, the designers chose two very different bit patterns, even though they theoretically need only differ in one bit

HMAC

$H[(K^+ \oplus opad) \,||\, H[(K^+ \oplus ipad) \,||\, M]]$

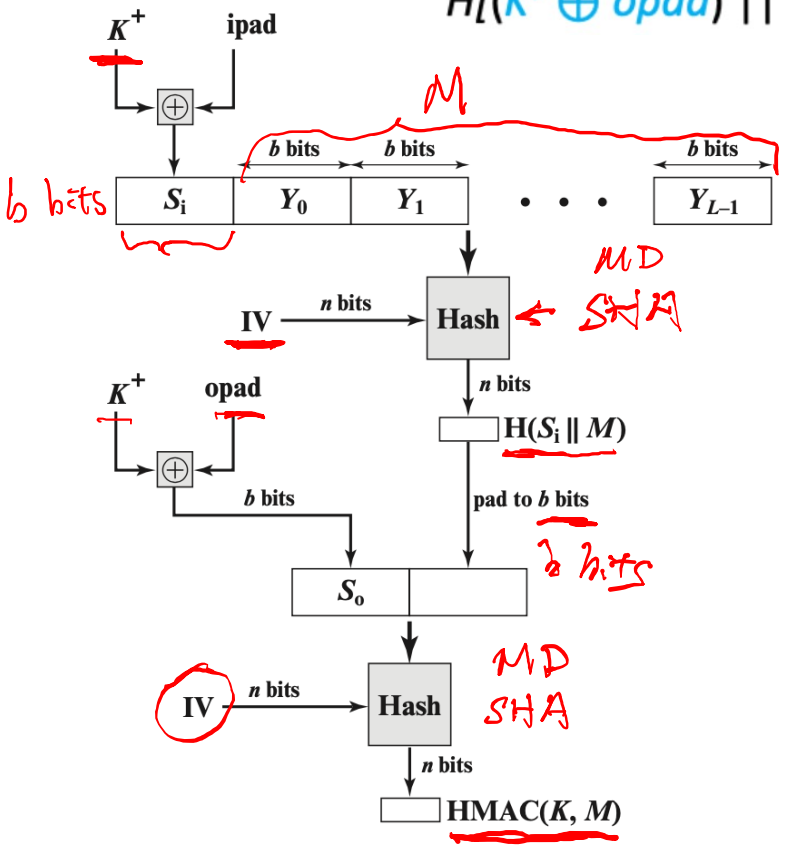


Figure 3.6 HMAC Structure

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

$$K^+ = \begin{cases} H(K) & K \text{ is larger than block size} \\ K & otherwise \end{cases}$$

ipad = 00110110 , repeat b/8 times

opad = 01011100, repeat b/8 times

HMAC procedure

$$H[(K^+ \oplus opad) \,||\, H[(K^+ \oplus ipad) \,||\, M]]$$

- Step 1: Append zeros to the left end of K to create a b-bit string K^+ (e.g., if K is of length 160 bits and b = 512, then K will be appended with 44 zero bytes);
- Step 2: XOR (bitwise exclusive-OR) K^+ with ipad to produce the b-bit block S_i;
- Step 3: Append M to S_i;
- Step 4: Apply H to the stream generated in step 3;
- Step 5: XOR K^+ with opad to produce the b-bit block S_o;
- Step 6: Append the hash result from step 4 to S_o;
- Step 7: Apply H to the stream generated in step 6 and output the result.

HMAC Properties

- HMAC(K, M) = $H[(K^+ \oplus opad) \,||\, H((K^+ \oplus ipad) \,||\, M]]$
- HMAC is a hash function, so it has the properties of the underlying hash too
  - It is collision resistant
  - Given HMAC(K, M), an attacker can't learn M – *one way*
  - If the underlying hash is secure, HMAC doesn't reveal M, but it is still deterministic
- You can't verify a tag T if you don't have K
- This means that an attacker can't brute-force the message M without knowing K

MACs: Summary

- Inputs: a secret key and a message
- Output: a tag on the message
- A secure MAC is unforgeable: Even if David can trick Alice into creating MACs for messages that David chooses, David cannot create a valid MAC on a message that she hasn't seen before
  - Example: HMAC(K, M) = $H((K^+ \oplus opad) \,||\, H((K^+ \oplus ipad) \,||\, M))$
- MACs do not provide confidentiality

Do MACs provide integrity?

- Do MACs provide integrity?
  - Yes. An attacker cannot tamper with the message without being detected
- Do MACs provide authenticity?
  - It depends on your threat model
  - If only two people have the secret key, MACs provide authenticity: it has a valid MAC, and it's not from me, so it must be from the other person
  - More than one secret key, If a message has a valid MAC, you can be sure it came from *someone with the secret key*, but you can't narrow it down to one person
- Do MACs provide confidentiality?

Authenticated Encryption

Authenticated Encryption: Definition

- **Authenticated encryption** (**AE**): A scheme that simultaneously guarantees confidentiality and integrity (and authenticity, depending on your threat model) on a message
- Two ways of achieving authenticated encryption:
  - Combine schemes that provide confidentiality with schemes that provide integrity
  - Use a scheme that is designed to provide confidentiality and integrity

Authenticated Encryption: Definition

- **Authenticated encryption** (**AE**): A scheme that simultaneously guarantees confidentiality and integrity (and authenticity, depending on your threat model) on a message
- Two ways of achieving authenticated encryption:
  - Combine schemes that provide confidentiality with schemes that provide integrity
  - Use a scheme that is designed to provide confidentiality and integrity

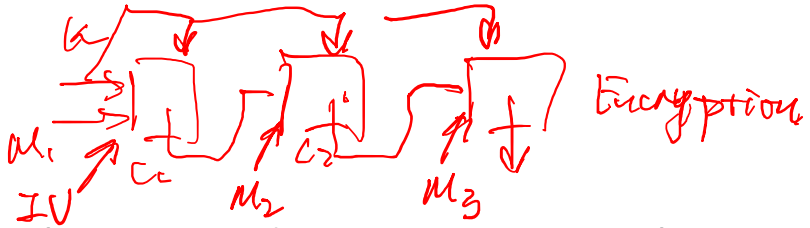
Scratchpad: Let's design it together

- You can use:
  - An encryption scheme: Enc(K, M) and Dec(K, M)
  - An unforgeable MAC scheme (e.g. HMAC): MAC(K, M)
- First attempt: Alice sends Enc(K_1, M) and MAC(K_2, M)
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? No, the MAC is not secure
- Idea 1: Let's compute the MAC on the *ciphertext* instead of the plaintext: Enc(K_1, M) and MAC(k_2, Enc(K_1, M))
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? Yes, the MAC might leak info about the ciphertext, but that's okay
- Idea 2: Let's encrypt the MAC too: Enc(K_1, M || MAC(K_2, M))
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? Yes, everything is encrypted

MAC-then-Encrypt or Encrypt-then-MAC?

- Method 1: Encrypt-then-MAC
  - First compute Enc(K_1, M)
  - Then MAC the ciphertext: MAC(K_2, Enc(K_1, M))
- Method 2: MAC-then-encrypt
  - First compute MAC(K_2, M)
  - Then encrypt the message and the MAC together: Enc(k_1, M || MAC(K_2, M))
- Which is better?
  - In theory, both are secure if applied properly
  - MAC-then-encrypt has a flaw: You don't know if tampering has occurred until after decrypting
    - Attacker can supply arbitrary tampered input, and you always have to decrypt it
    - Passing attacker-chosen input through the decryption function can cause side-channel leaks
- **Always use encrypt-then-MAC** because it's more robust to mistakes

TLS 1.0 "Lucky 13" Attack



- TLS: A protocol for sending encrypted and authenticated messages over the Internet
- TLS 1.0 uses MAC-then-encrypt: $Enc(k_1, M \,||\, MAC(k_2, M))$
  - The encryption algorithm is AES-CBC
- The Lucky 13 attack abuses MAC-then-encrypt to read encrypted messages
  - Guess a byte of plaintext and change the ciphertext accordingly
  - The MAC will error, but the time it takes to error is different depending on if the guess is correct
  - Attacker measures how long it takes to error in order to learn information about plaintext
  - TLS will send the message again if the MAC errors, so the attacker can guess repeatedly
- Takeaways
  - Side channel attack: The algorithm is proved secure, but poor implementation made it vulnerable
  - Always encrypt-then-MAC

Authenticated Encryption: Summary

- Authenticated encryption: A scheme that simultaneously guarantees confidentiality and integrity (and authenticity) on a message
- First approach: Combine schemes that provide confidentiality with schemes that provide integrity and authenticity
  - MAC-then-encrypt: Enc(K_1, M || MAC(K_2, M))
  - Encrypt-then-MAC: MAC(K_2, Enc(K_1, M))
  - Always use Encrypt-then-MAC because it's more robust to mistakes

Digital Signature

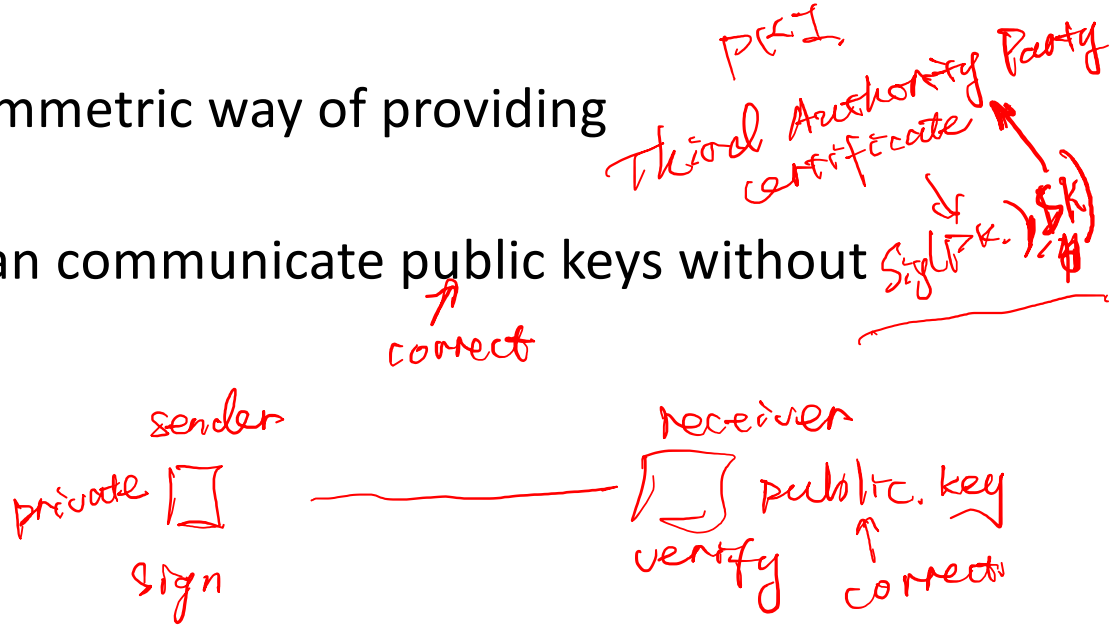
Digital Signatures

- NIST FIPS PUB 186-4 - the result of a cryptographic transformation of data that, when properly implemented, provides a mechanism for verifying origin authentication, data integrity, and signatory non-repudiation
- Based on asymmetric keys

no confidentiality

Digital Signatures

- Asymmetric cryptography is good because we don't need to share a secret key
- Digital signatures are the asymmetric way of providing integrity/authenticity to data
- Assume that Alice and Bob can communicate public keys without David interfering



Digital Signatures: Definition

- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the verify (public) key, and *SK* is the signing (secret) key
  - Sign(*SK*, *M*) → *sig*: Sign the message *M* using the signing key *SK* to produce the signature *sig*
  - Verify(*PK*, *M*, *sig*) → {0, 1}: Verify the signature *sig* on message *M* using the verify key *PK* and output 1 if valid and 0 if invalid
- Properties:
  - **Correctness**: Verification should be successful for a signature generated over any message
    - Verify(*PK*, *M*, Sign(*SK*, *M*)) = 1 for all *PK*, *SK* ← KeyGen() and *M*
  - **Efficiency**: Signing/verifying should be fast
  - **Security**: Same as for MACs except that the attacker also receives PK
    - Namely, no attacker can forge a signature for a message

without private key

RSA Signature

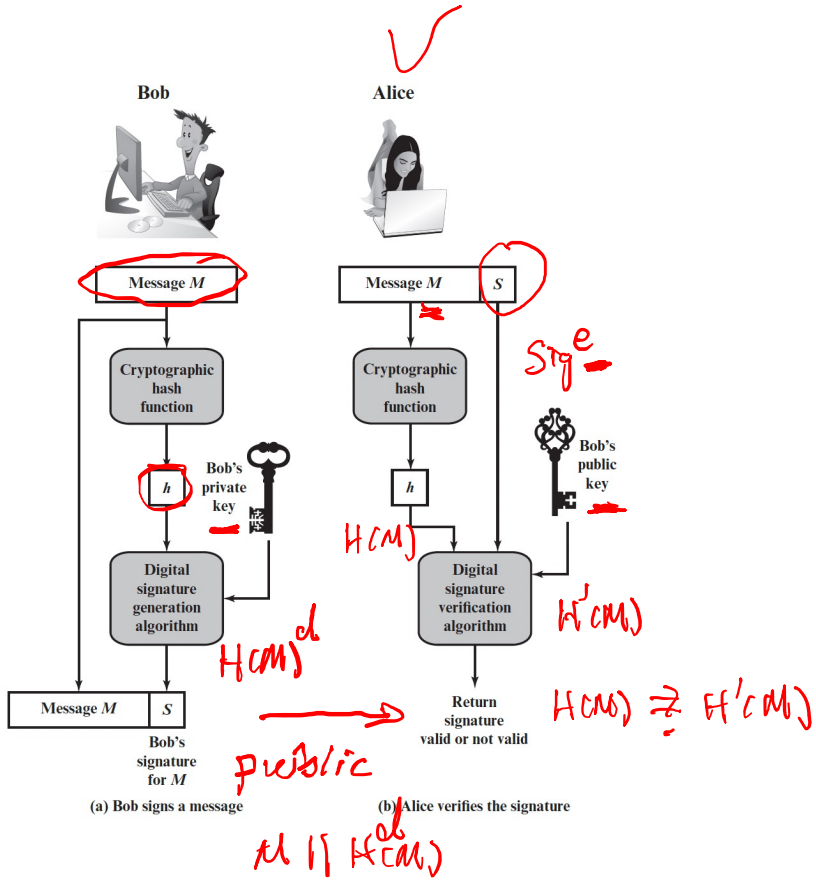
- KeyGen(): Same as RSA Encryption
  - Randomly pick two large primes, p and q
  - Compute $n = pq$ → large number, public.
    - n is usually between 2048 bits and 4096 bits long
  - Choose e
    - Requirement: e is relatively prime to $(p - 1)(q - 1)$ $= \phi(n)$ $\gcd(e, \phi(n)) = 1$
    - Requirement: $2 < e < (p - 1)(q - 1) = \phi(n)$
  - Compute $d = e^{-1} \bmod (p - 1)(q - 1)$
  - **Public key**: n and e → $\phi(n)$, public
  - **Private key:** d

A Short Quiz

- We will have a short quiz on Wednesday, Oct. 30, in class
- A short quiz will cover the materials taught that day.

RSA Signatures

- Sign(d, M):
  - Compute $H(M)^d \bmod n$
- Verify(e, n, M, sig)
  - Verify that $H(M) \equiv sig^e \bmod n$



RSA Digital Signature Algo

Step1: Generate a hash value, or message digest, mHash from the message *M* to be signed

Step2: Pad mHash with a constant value padding1 and pseudorandom value salt to form *M'*

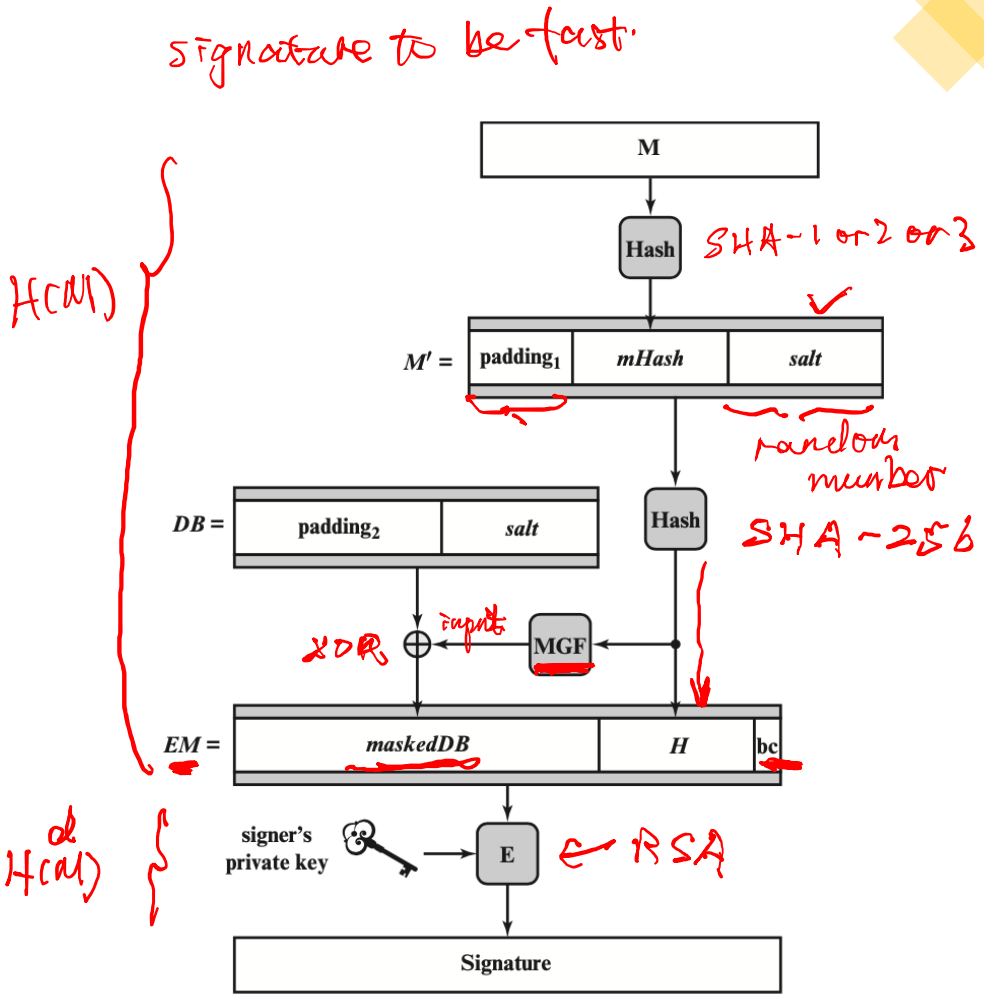
Step3: Generate hash value *H* from *M'*

Step4: Generate a block DB consisting of a constant value padding 2 and salt

Step5: Use the mask generating function MGF, which produces a randomized out-put from input *H* of the same length as DB

Step 6: Create the encoded message (EM) block by padding *H* with the hexadecimal constant bc and the XOR of DB and output of MGF

Step 7: Encrypt EM with RSA using the signer's private key



RSA Signatures: Correctness

Same prove process as RSA decryption

H(M): fast, less bandwidth

Theorem: $sig^e \equiv H(M) \bmod N$

Proof:

Because $Sig = H(M)^d \bmod N$

$$sig^e = [H(M)^d]^e \bmod N = H(M)^{ed} \bmod N$$

$$= H(M)^{de} \bmod N \quad (1)$$

Because $d \cdot e = 1 \bmod \phi(n)$ by definition of modular area

$d \cdot e = 1 + k \cdot \phi(n)$ $k \in \mathbb{Z}$

$$= H(M)^{k\phi(N)+1} \bmod N$$

$$= [H(M)^{\phi(N)}]^k \cdot H(M) \bmod N$$

Euler's theorem: if $\gcd(M, N) = 1$ then $M^{\phi(N)} = 1 \bmod N$

$$= 1^k \cdot H(M) \bmod N$$

$$= H(M) \bmod N$$

■

RSA Signatures: Correctness

Theorem: $sig^e \equiv H(M) \bmod N$

Proof:

$$\text{sig}^e = [H(M)^d\ ]^e\ mod\ N \quad = H(M)^{ed} \bmod N$$

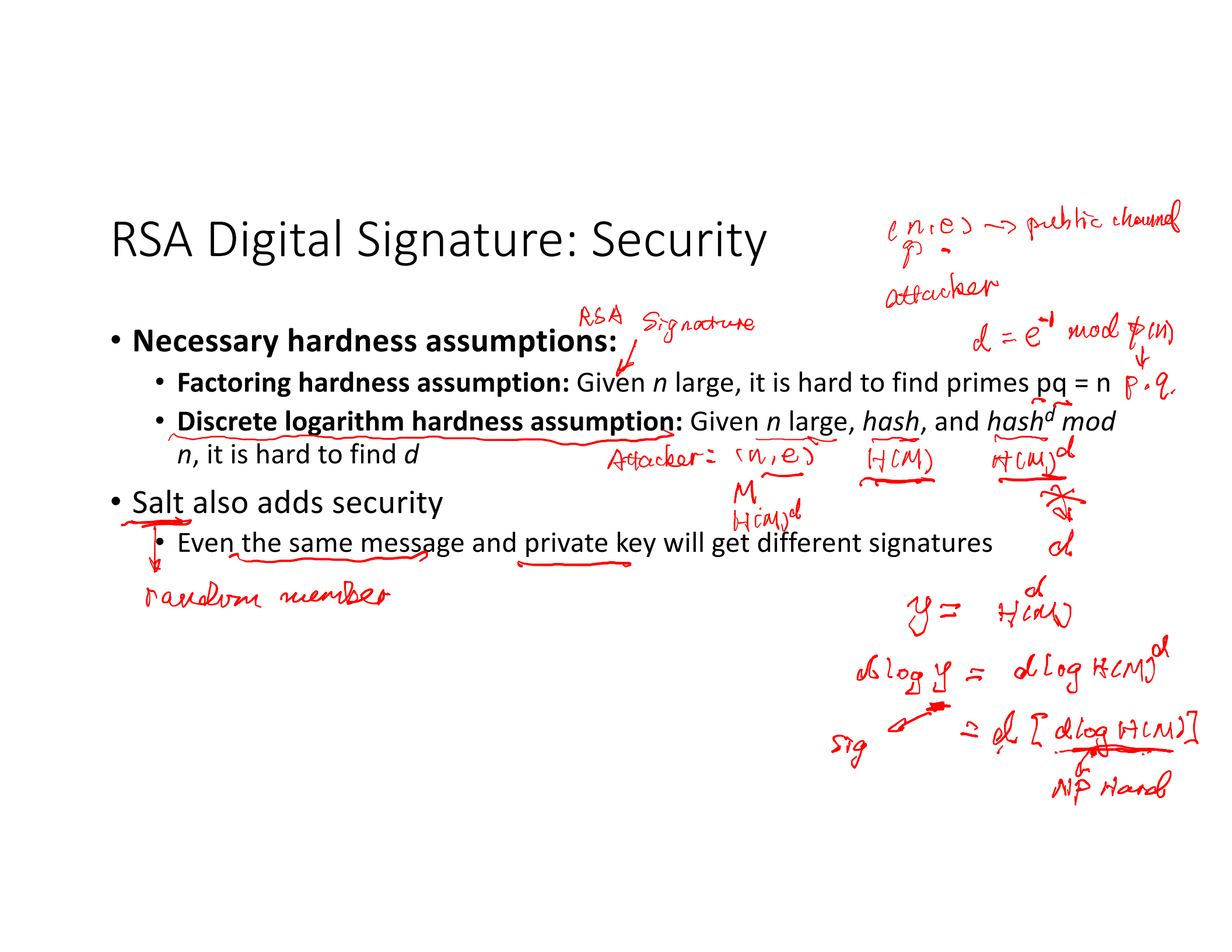
$$= H(M)^{k\phi(n)+1} \bmod N$$

$$= [H(M)^{\phi(n)}]^k \cdot H(M) \bmod N$$

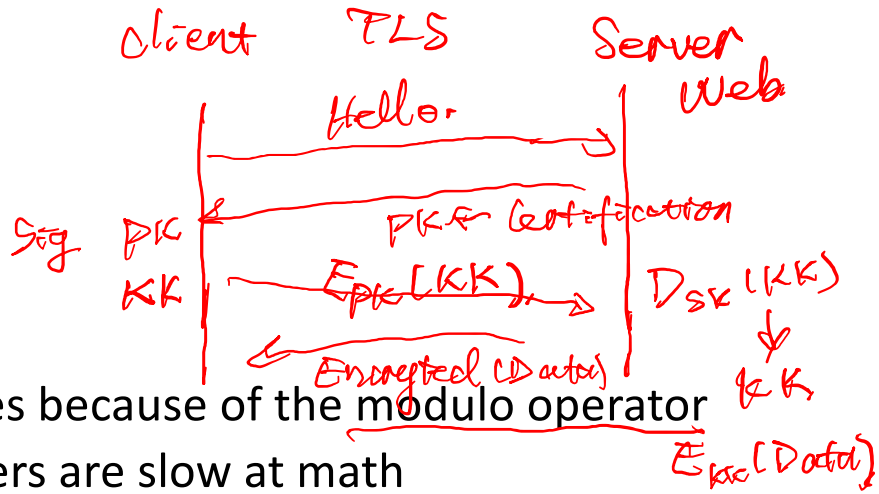
$$= H(M) \bmod N$$

RSA Digital Signature: Security

- **Necessary hardness assumptions:**
  - **Factoring hardness assumption:** Given *n* large, it is hard to find primes pq = n
  - **Discrete logarithm hardness assumption:** Given *n* large, *hash*, and $hash^d$ *mod n*, it is hard to find *d*
- Salt also adds security
  - Even the same message and private key will get different signatures



Hybrid Encryption



- Issues with public-key encryption
  - Notice: We can only encrypt small messages because of the modulo operator
  - Notice: There is a lot of math, and computers are slow at math
  - Result: We don't use asymmetric for large messages
- **Hybrid encryption**: Encrypt data under a randomly generated key *K* using symmetric encryption, and encrypt *K* using asymmetric encryption
  - $Enc_{Asym}(PK, K)$; $Enc_{Sym}(K, \text{large message})$
  - Benefit: Now we can encrypt large amounts of data quickly using symmetric encryption, and we still have the security of asymmetric encryption

Homework (Textbook) – no submission

- Review Question: 3.1, 3.2, 3.3, 3.4, 3.5, 3.6
- Problems:
  - prove correctness of RSA digital signature
  - 3.14 & 3.15

Homework 2 - individual

- For Chapter 3
- Deadline Friday, Nov. 1 before class
- 10% penalty per day for late submission

Thank you!

Network Security

Chapter 4

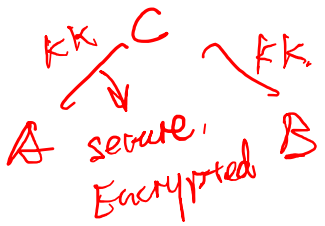
Key Distribution

Symmetric Key Distribution and User Authentication

4.2

Ways to achieve symmetric key distribution

- A key could be selected by A and physically delivered to B
- A third party could select the key and physically deliver it to A and B
- If A and B have previously and recently used a key, one party could transmit the new key to the other, using the old key to encrypt the new key
- If A and B each have an encrypted connection to a third-party C, C could deliver a key on the encrypted links to A and B



Terminologies

- Session key
- Permanent key
- key distribution center (KDC)
  - third party authority, centralized infrastructure
  - give permissions for two parties to communicate

Diffie-Hellman Key Exchange

Section 3.5

In class quiz on Wednesday

- We will have a short quiz on Wednesday, Oct. 30, in class
- A short quiz will cover the materials taught that day.
- Please be on time for class to avoid missing the quiz questions.

Diffie-Hellman Key Exchange

Section 3.5

Outline

- Diffie-Hellman Key Exchange (DHKE) Algorithm
- Analysis of DHKE

Use case

Attack { Passive : Eve. no modification of msg
Active : modify msg

① sniffer
② ISP
③ Government

Alice

msg

Encryption
① Symmetric
② Asymmetric

unsecure →

pre-shared secret key

Bob

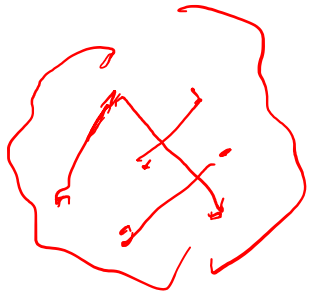
How to distribute the symmetric key?

4 ways : 1. physical deliver
2. third authority physical deliver
3. old key encrypts new
4. Encrypted by A Encrypted by B

Recall: ways to achieve symmetric key distribution

- A key could be selected by A and physically delivered to B
- A third party could select the key and physically deliver it to A and B
- If A and B have previously and recently used a key, one party could transmit the new key to the other, using the old key to encrypt the new key
- If A and B each have an encrypted connection to a third-party C, C could deliver a key on the encrypted links to A and B

Diffie-Hellman Key Exchange



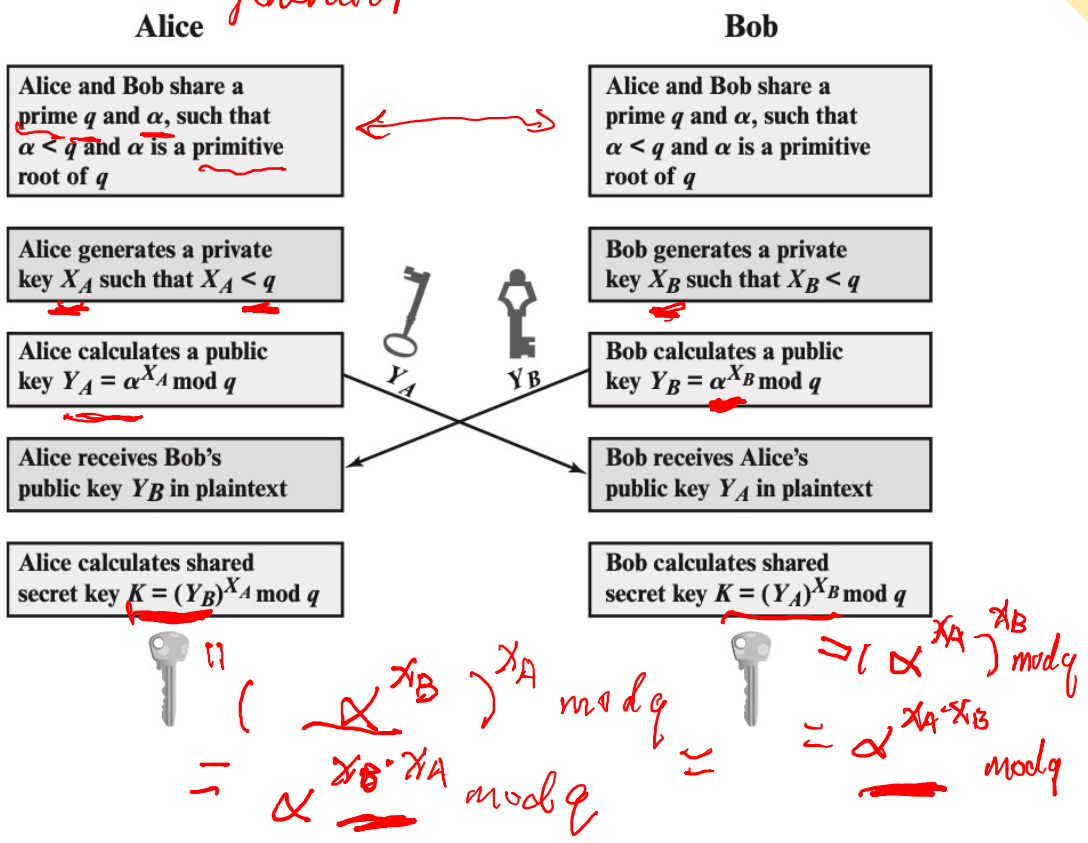
- Invented by Whitfield Diffie and Martin Hellman in 1976
- Allows Alice and Bob to exchange a key without Eve learning it
- No third party involved
- After DHKE, a common shared key, $\alpha^{X_A X_B}$ is established, it can be used to encrypt message
- A common shared key is symmetric

The Diffie-Hellman Key Exchange

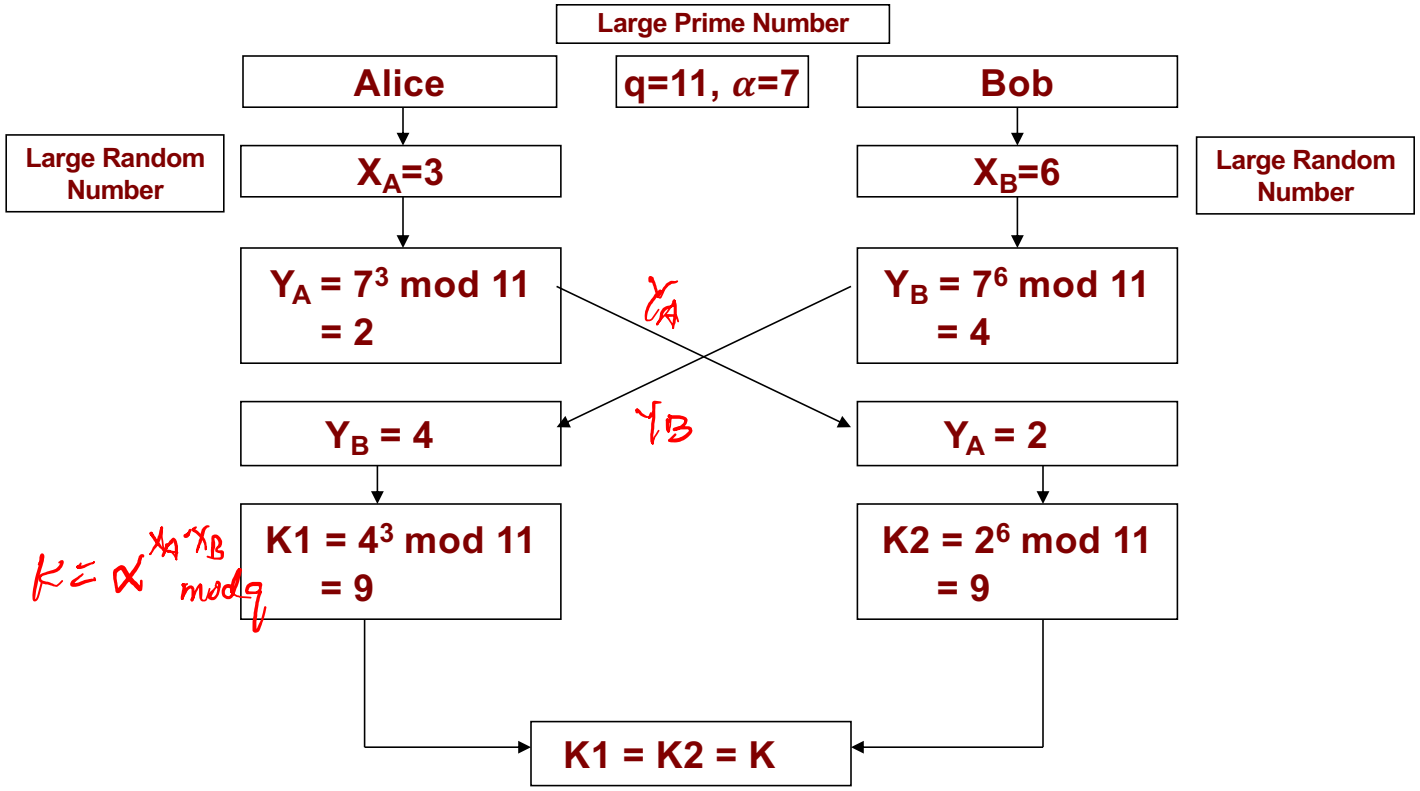
- From B's view
- $K = {Y_B}^{X_A} \ mod \ q$

  $= (\alpha^{X_B} \ mod \ q)^{X_A} \ mod \ q$

  $= \alpha^{X_B X_A} \ mod \ q$



Example of the Diffie -Hellman algorithm



Note: X_A, X_B, K1, K2 are Private to others

Analysis of DHKE - Attack

$Y_A = \alpha^{X_A}$ $\quad X_A = dlog_{\alpha,q} Y_A \mod q$

- Adversary gets q, α, Y_A, Y_B.
- She needs to compute either X_A or $X_B = dlog_{\alpha,q} Y_B$
- Secure?

Discrete Log Problem

Cryptographic assumptions:

- **Discrete logarithm problem** (**discrete log problem**): Given $\alpha, q, \alpha^{X_A} \ mod \ q$ for random X_A, it is computationally hard to find X_A
- **Diffie-Hellman assumption**: Given $\alpha, q, \alpha^{X_A} \ mod \ q$, and $\alpha^{X_B} \ mod \ q$ for random X_A, X_B, no polynomial time attacker can distinguish between a random value R and $\alpha^{X_A X_B} \ mod \ q$.
  - Intuition: The best known algorithm is to first calculate X_A and then compute $(\alpha^{X_B})^{X_A} \ mod \ q$, but this requires solving the discrete log problem, which is hard!
- Note: Multiplying the values doesn't work, since you get $\alpha^{X_A + X_B} \ mod \ p \ \neq \ \alpha^{X_A X_B} \ mod \ p$

DHKE in Python Cryptography Library

- https://cryptography.io/en/latest/hazmat/primitives/asymmetric/

Summing Up

- Symmetric Key crypto has a major problem:
  - How do two people who don't know each other share a key?
- A Diffie-Hellman key exchange lets them compute a shared key even in the presence of an eavesdropper, Eve.
- However, if attack is active, instead of passive, this wouldn't work …
- Diffie-Hellman suffers man-in-the-middle attack (next class)

Take home exercises

- SW, "Network Security Essentials", 6th Edition, 2017
  - Problems – 3.21

  Consider a Diffie-Hellman scheme with a common prime q = 11 and a primitive root α = 2.

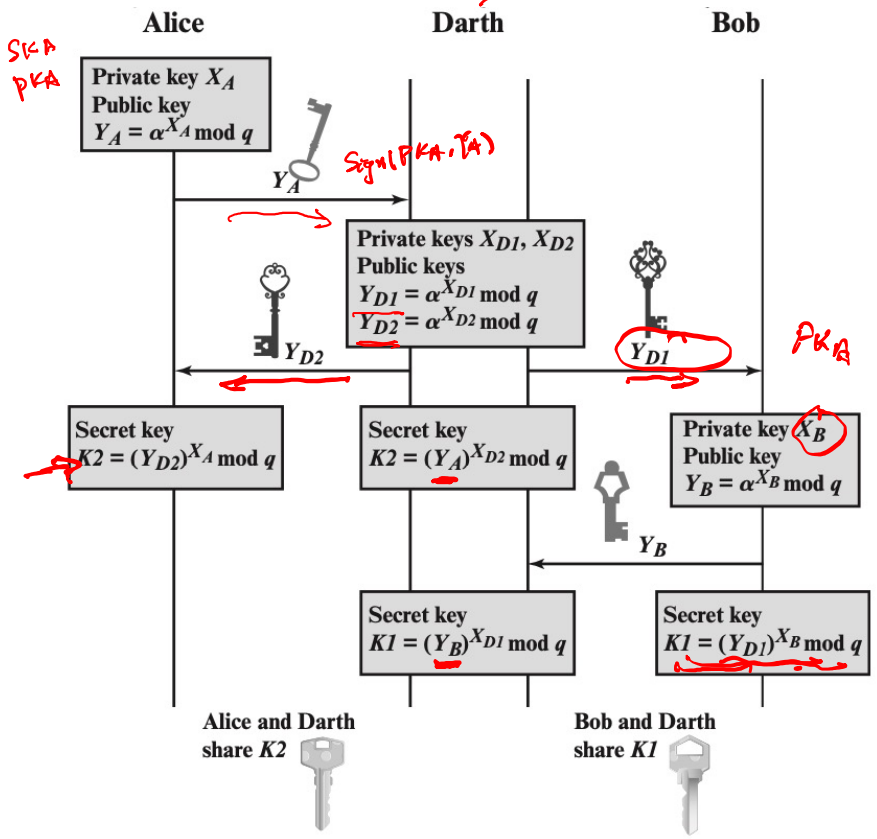
  a. if user A has public key Y_A = 9, what is A's private key X_A?

  b. If user B has public key Y_B = 3, what is the shared secret key K?

Diffie-Hellman is susceptible to man-in-the-middle attacks

- David can alter messages, block messages, and send her own messages
- **DH is not** secure against a MITM attacker: David can just do a DH with both sides!

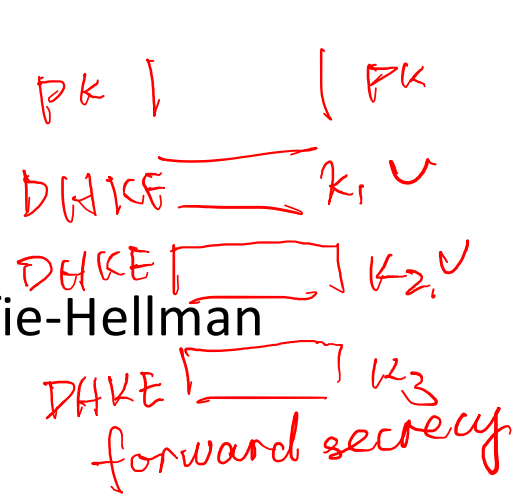
Diffie-Hellman: Security



Diffie-Hellman: issues

- Diffie-Hellman is not secure against a MITM adversary
- DHE is an *active protocol*: Alice and Bob need to be online at the same time to exchange keys
  - What if Bob wants to encrypt something and send it to Alice for her to read later?
- Diffie-Hellman does not provide *authentication*
  - You exchanged keys with someone, but Diffie-Hellman makes no guarantees about who you exchanged keys with; it could be David!

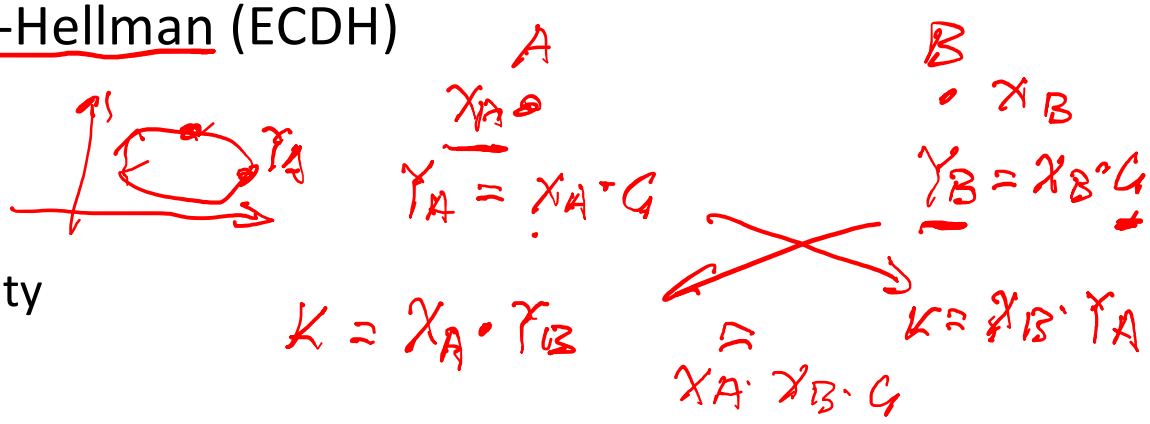
Ephemerality of Diffie-Hellman



- Diffie-Hellman can be used ephemerally (called Diffie-Hellman ephemeral, or DHE)
  - **Ephemeral**: Short-term and temporary, not permanent
  - Alice and Bob discard X_A, X_B and $K = \alpha^{X_A X_B} \ mod \ q$ when they're done
  - Because you need X_A and X_B to derive K, you can never derive K again!
  - Sometimes K is called a **session key**, because it's only used for an ephemeral session
- Eve can't decrypt any messages she recorded: Nobody saved X_A, X_B or K, and her recording only has $\alpha^{X_A} \ mod \ q$ and $\alpha^{X_B} \ mod \ q$!

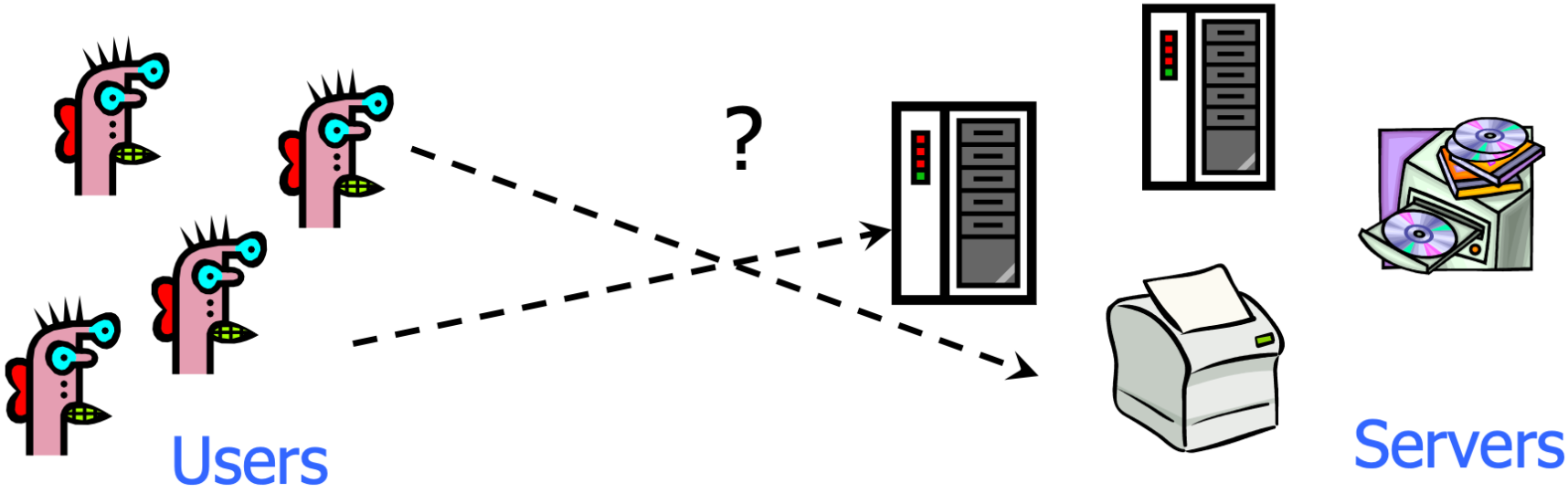
Diffie-Hellman Key Exchange: Summary

- Algorithm:
  - Alice chooses X_A and sends $\alpha^{X_A} \; mod \; q$ to Bob
  - Bob chooses X_B and sends $\alpha^{X_B} \; mod \; q$ to Alice
  - Their shared secret is $(\alpha^{X_A})^{X_B} = (\alpha^{X_B})^{X_A} = \; \alpha^{X_A X_B} \; mod \; q$
- Diffie-Hellman provides forwards secrecy: Nothing is saved or can be recorded that can ever recover the key
- Diffie-Hellman can be performed over other mathematical groups, such as elliptic-curve Diffie-Hellman (ECDH)
- Issues
  - **Not** secure against MITM
  - Both parties must be online
  - Does not provide authenticity



Kerberos
4.3

Many-to Many Authentication



How do users prove their identities when requesting services from machines on the network?

Threats

- User impersonation
  - Malicious user with access to a workstation pretends to be another user from the same workstation
- Network address impersonation
  - Malicious user changes network address of his workstation to impersonate another workstation
- Eavesdropping, tampering, replay
  - Malicious user eavesdrops, tampers, or replays other users' conversations to gain unauthorized access

Requirements

- Security
  - against attacks by eavesdroppers and malicious users
- Transparency
  - users shouldn’t notice authentication taking place
  - entering password is ok, if done rarely
- Scalability
  - Large number of users and servers

Kerberos

- scenario: users at workstations wish to access services on servers distributed throughout the network – many to many authentication

Kerberos

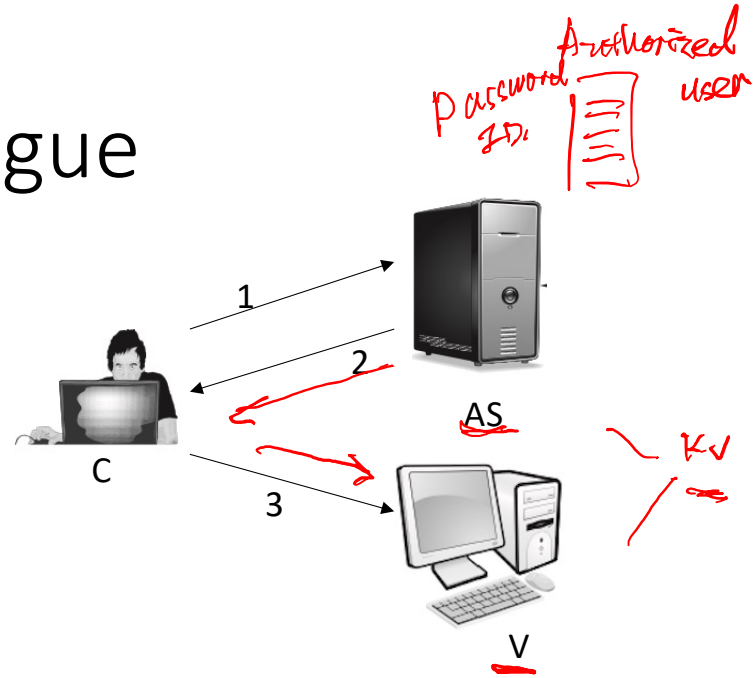
- a centralized authentication server provides mutual authentication between users and servers
  - a key distribution and user authentication service developed at MIT
  - works in an open distributed environment
- client-service model
- Kerberos protocol messages are protected against eavesdropping and replay attacks
- Kerberos v4 and v5 [RFC 4120]

Kerberos

- a centralized authentication server provides mutual authentication between users and servers
  - a key distribution and user authentication service developed at MIT
  - works in an open distributed environment
- client-service model
- Kerberos protocol messages are protected against eavesdropping and replay attacks
- Kerberos v4 and v5 [RFC 4120]

A Simple Authentication Dialogue

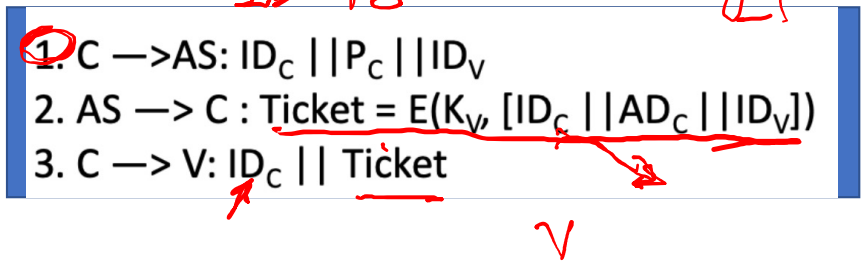
- 1. C —>AS: ID_C ||P_C ||ID_V
- 2. AS —> C : Ticket = $E(K_V, [ID_C ||AD_C ||ID_V])$
- 3. C —> V: ID_C || Ticket



- AS – authentication server
- ID_* - identifier
- P_C - password of user
- AD_C - network address of C
- K_V - secret encryption key shared by AS and V

Advantage

- Client and malicious attacker cannot alter ID_C (impersonate), AD_C(change of address) , ID_V
- server V can verify the user is authenticated through ID_C , and grants service to C
- guarantee the ticket is valid only if it is transmitted from the same client that initially requested the ticket



Secure?

- Insecure: password is transmitted openly and frequently
- Solution: no password transmitted by involving ticket-granting server (TGS)

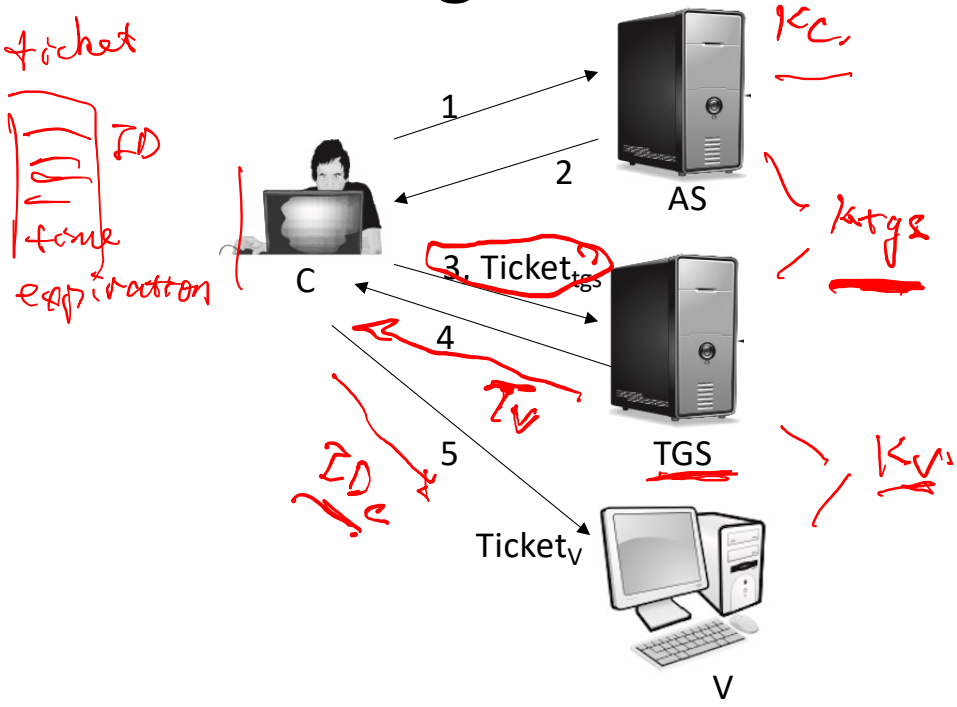
1. C —>AS: $ID_C || P_C || ID_V$
2. AS —> C : Ticket = $E(K_V, [ID_C || AD_C || ID_V])$
3. C —> V: ID_C || Ticket

A More Secure Authentication Dialogue

- Once per user logon session
  - (1) C —>AS: $ID_C || ID_{tgs}$
  - (2) AS —> C: $E(K_C, Ticket_{tgs})$
- Once per type of service:
  - (3) C —>TGS: $ID_C || ID_V || Ticket_{tgs}$
  - (4) TGS —> C: $Ticket_V$
- Once per service session:
  - (5) C —> V: $ID_C || Ticket_V$

$$Ticket_{tgs} = E(K_{tgs}, [ID_C \| AD_C \| ID_{tgs} \| TS_1 \| Lifetime_1])$$

$$Ticket_v = E(K_v, [ID_C \| AD_C \| ID_v \| TS_2 \| Lifetime_2])$$



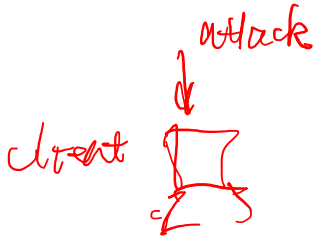
1. C —>AS: $ID_C || P_C || ID_V$
2. AS —> C : Ticket = $E(K_V, [ID_C || AD_C || ID_V])$
3. C —> V: $ID_C || Ticket$

Advantage

- No password transmitted in plaintext
- Timestamp is added to prevent reuse of ticket by an attacker

Secure?

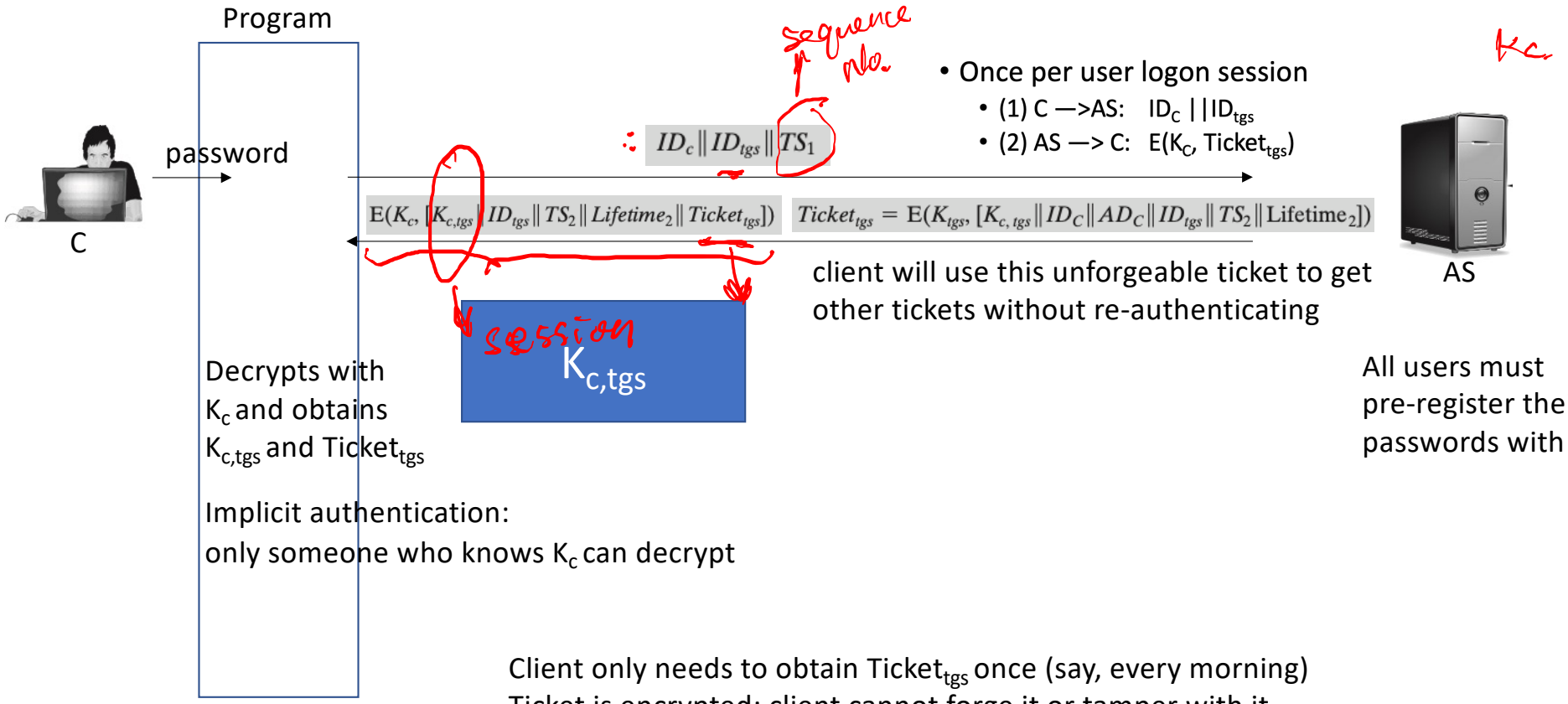
no user authentication



- Ticket hijacking
  - Malicious user may steal the service ticket of another user on the same workstation and try to use it
    - Network address verification does not help
  - Servers must verify that the user who is presenting the ticket is the same user to whom the ticket was issued
- No server authentication
  - Attacker may misconfigure the network so that he receives messages addressed to a legitimate server – man in the middle attack
    - Capture private information from users and/or deny service
  - Servers must prove their identity to users
- Solution: session key

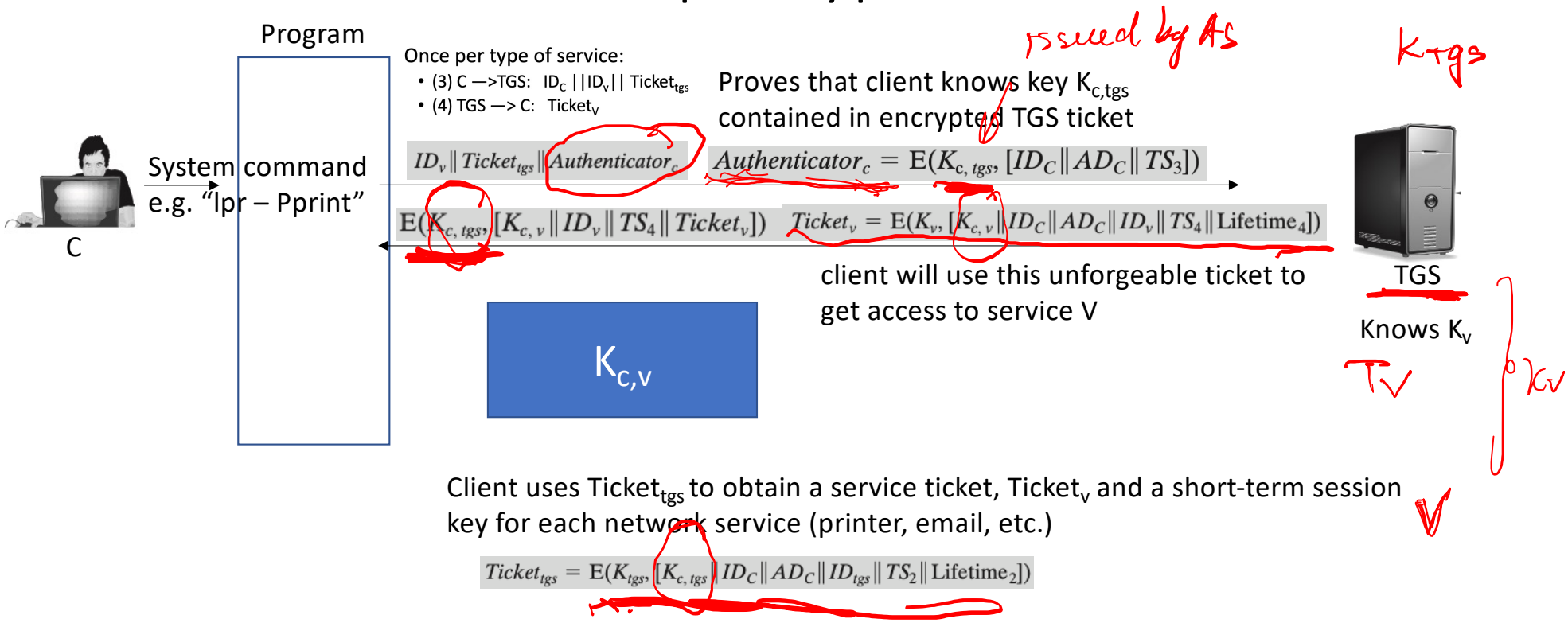
- Once per user logon session
  - (1) C —>AS: $ID_C || ID_{tgs}$
  - (2) AS —> C: $E(K_C, Ticket_{tgs})$
- Once per type of service:
  - (3) C —>TGS: $ID_C || ID_V || Ticket_{tgs}$
  - (4) TGS —> C: $Ticket_V$
- Once per service session:
  - (5) C —> V: $ID_C || Ticket_V$

Kerberos v4. - once per user logon session



Client only needs to obtain $Ticket_{tgs}$ once (say, every morning)
Ticket is encrypted; client cannot forge it or tamper with it

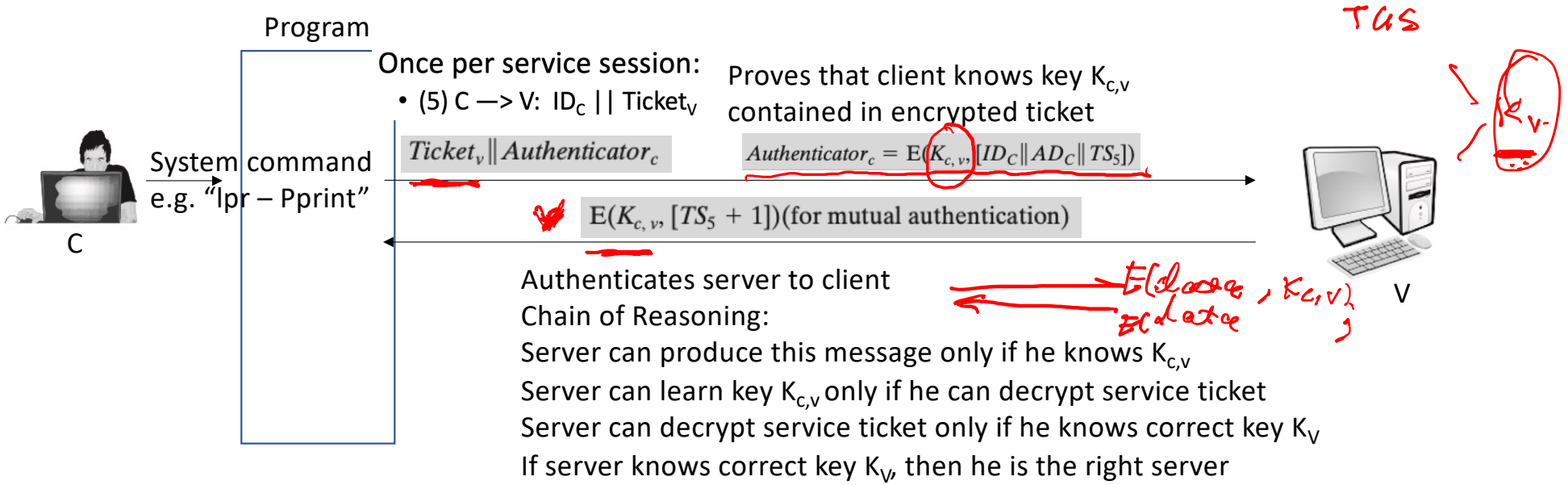
Kerberos v4. - once per type of service



Client uses $Ticket_{tgs}$ to obtain a service ticket, $Ticket_v$ and a short-term session key for each network service (printer, email, etc.)

$$Ticket_{tgs} = E(K_{tgs}, [K_{c,tgs} \| ID_C \| AD_C \| ID_{tgs} \| TS_2 \| Lifetime_2])$$

Kerberos v4. - once per service session



Authenticates server to client

Chain of Reasoning:

Server can produce this message only if he knows $K_{c,v}$

Server can learn key $K_{c,v}$ only if he can decrypt service ticket

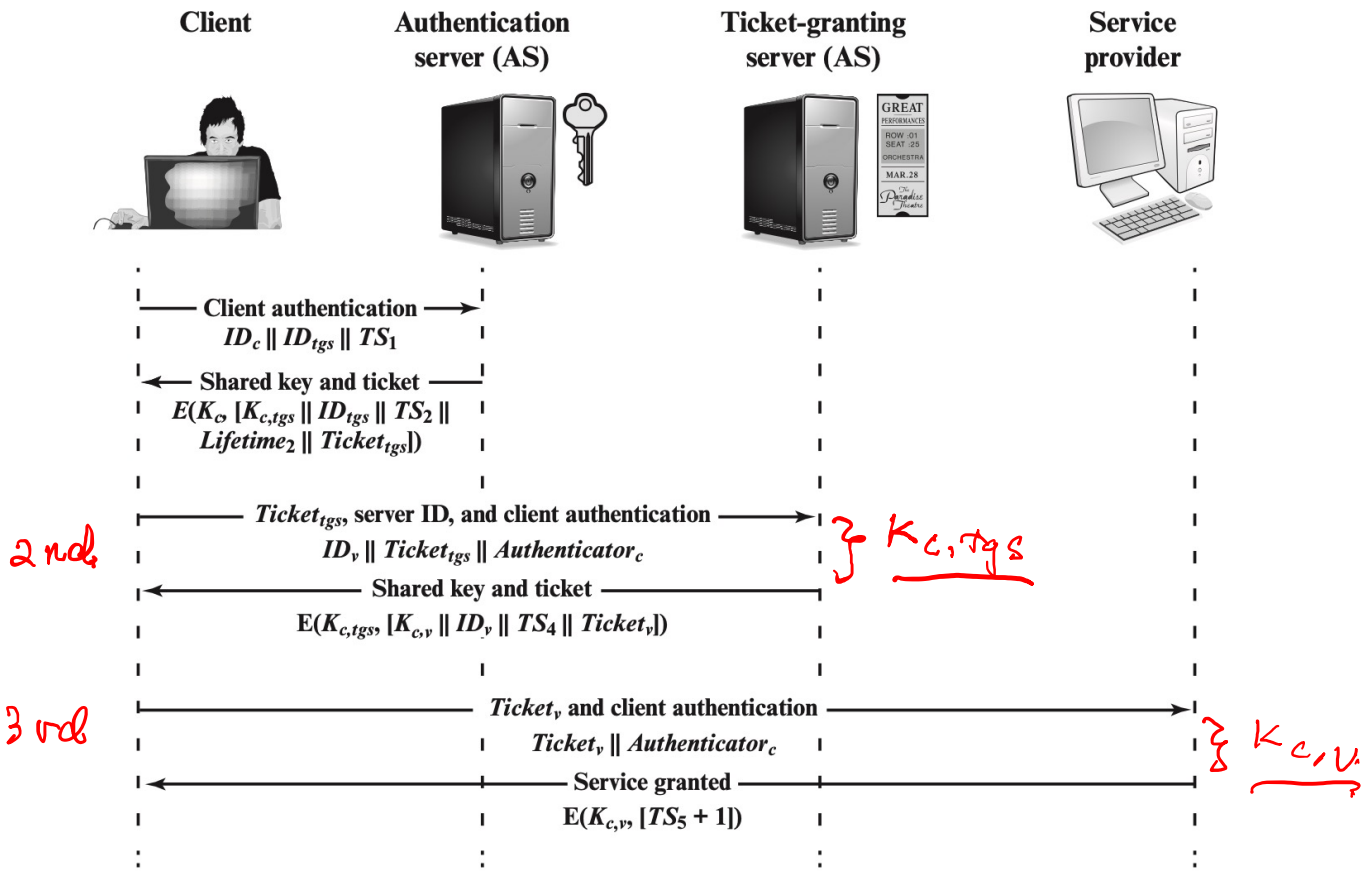
Server can decrypt service ticket only if he knows correct key K_V

If server knows correct key K_V, then he is the right server

For each service request, client uses the short-term key, $K_{c,v}$, for that service and the ticket he received from TGS

$$Ticket_v = E(K_v, [K_{c,v} \| ID_C \| AD_C \| ID_v \| TS_4 \| Lifetime_4])$$

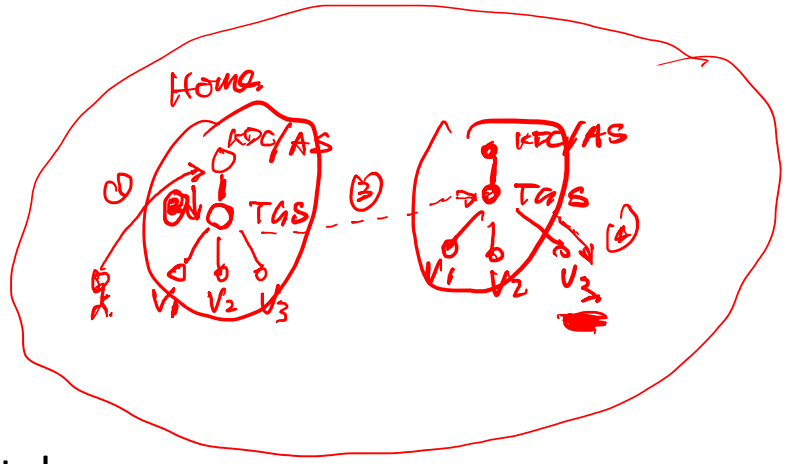
Overview of Kerberos



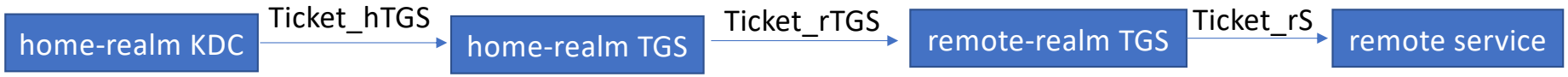
Important Ideas in Kerberos

- Short-term session keys
  - Long-term secrets used only to derive short-term keys
  - Separate session key for each user-server pair
    - Re-used by multiple sessions between same user and server
- Proofs of identity based on authenticators
  - Client encrypts his identity, addr, time with session key; knowledge of key proves client has authenticated to KDC/AS
    - Also prevents replays (if clocks are globally synchronized)
  - Server learns this key separately (via encrypted ticket that client can't decrypt), then verifies client's authenticator
- Symmetric cryptography only

Kerberos in Large Networks



- One KDC isn't enough for large networks
- Network is divided into realms
  - KDCs in different realms have different key databases
- To access a service in another realm, users must...
  - Get ticket for home-realm TGS from home-realm KDC
  - Get ticket for remote-realm TGS from home-realm TGS
    - As if remote-realm TGS were just another network service
  - Get ticket for remote service from that realm's TGS
  - Use remote-realm ticket to access service



Practical Uses of Kerberos

- Microsoft Windows – Active Directory
- Email, FTP, network file systems, many other applications have been kerberized
  - Use of Kerberos is transparent for the end user
  - Transparency is important for usability!
- Local authentication
  - login and su in OpenBSD
- Authentication for network protocols
  - rsh
- Secure windowing systems

Readings

- Kerberos: The Network Authentication Protocol
  https://web.mit.edu/kerberos/

Practice – no submission

- William Stallings, “Network Security Essentials”, 6 Edition, 2017
  - Chapter 4’s problems: 4.8, 4.9, 4.10

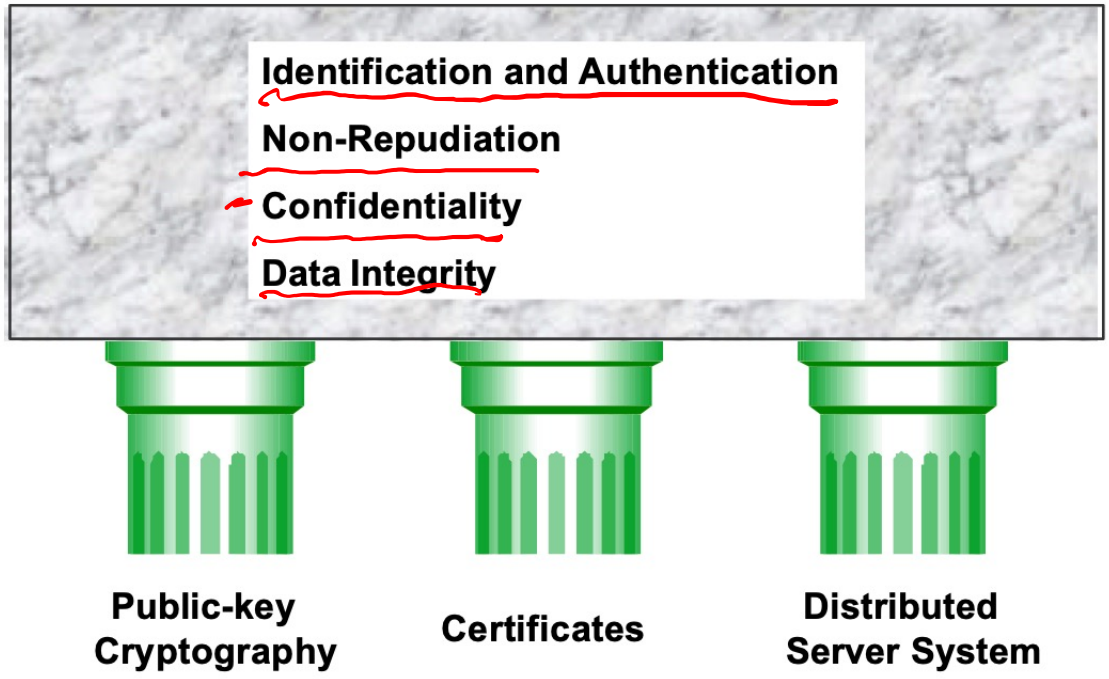
Key Distribution Using Asymmetric Encryption

PKI and Certificates

(Section 4.5)

What is PKI?

- Use of public-key cryptography and X.509 certificates in a distributed server system to establish secure domains and trusted relationships



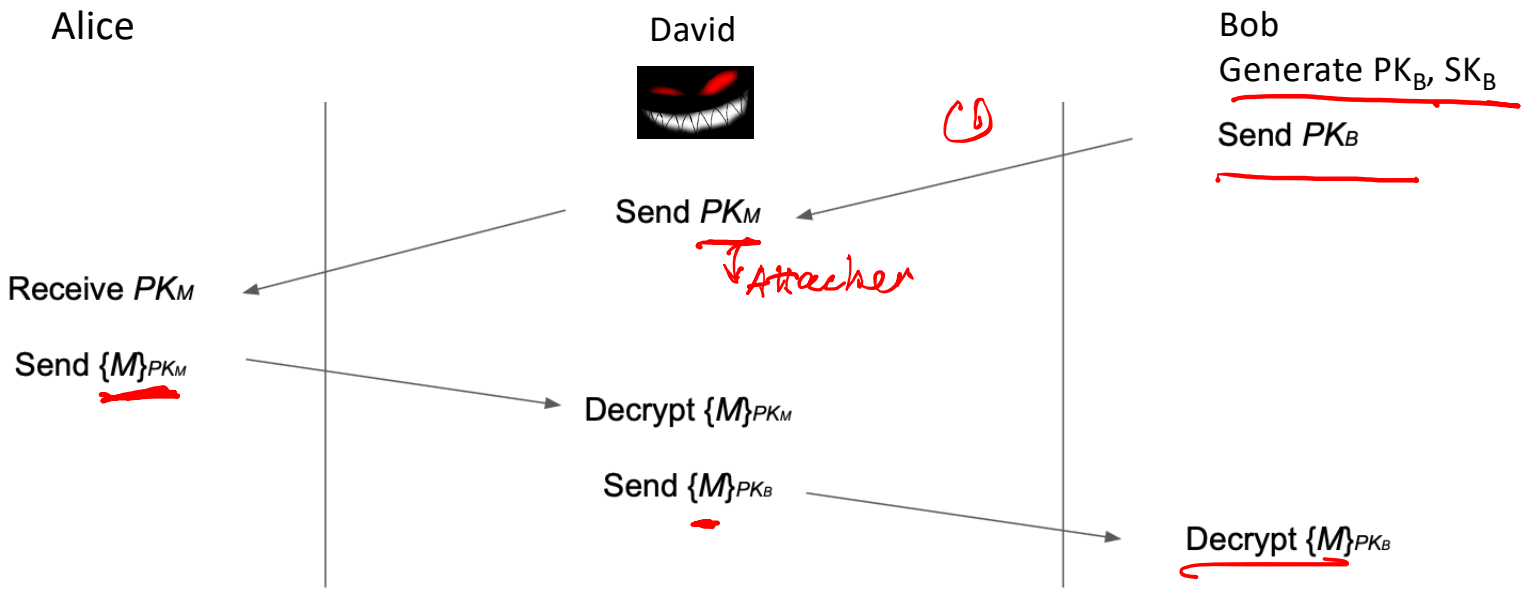
Why use public-key cryptography?

- Review: Public-key cryptography is great! We can communicate securely without a shared secret
  - Public-key encryption: Everybody encrypts with the public key, but only the owner of the private key can decrypt
  - Digital signatures: Only the owner of the private key can sign, but everybody can verify with the public key

Problem: Distributing Public Keys

- Public-key cryptography alone is not secure against man-in-the-middle attacks
- Scenario
  - Alice wants to send a message to Bob
  - Alice asks Bob for his public key
  - Bob sends his public key to Alice
  - Alice encrypts her message with Bob’s public key and sends it to Bob
- What can David do?
  - Replace Bob's public key with David’s public key
  - Now Alice has encrypted the message with David’s public key, and David can read it!

Problem: Distributing Public Keys



Man-in-the-Middle Attack

Solution: Distributing Public Keys

- Idea: Sign Bob's public key to prevent tampering
- Problem
  - If Bob signs his public key, we need his public key to verify the signature
  - But Bob's public key is what we were trying to verify in the first place!
  - Circular problem: Alice can never trust any public key she receives
- You cannot gain trust if you trust nothing. You need a root of trust!
  - **Trust anchor**: Someone that we implicitly trust
  - From our trust anchor, we can begin to trust others

Trust-on-First-Use

- **Trust-on-first-use**: The first time you communicate, trust the public key that is used and warn the user if it changes in the future
  - Used in SSH and a couple other protocols
  - Idea: Attacks aren't frequent, so assume that you aren't being attacked the first time communicate

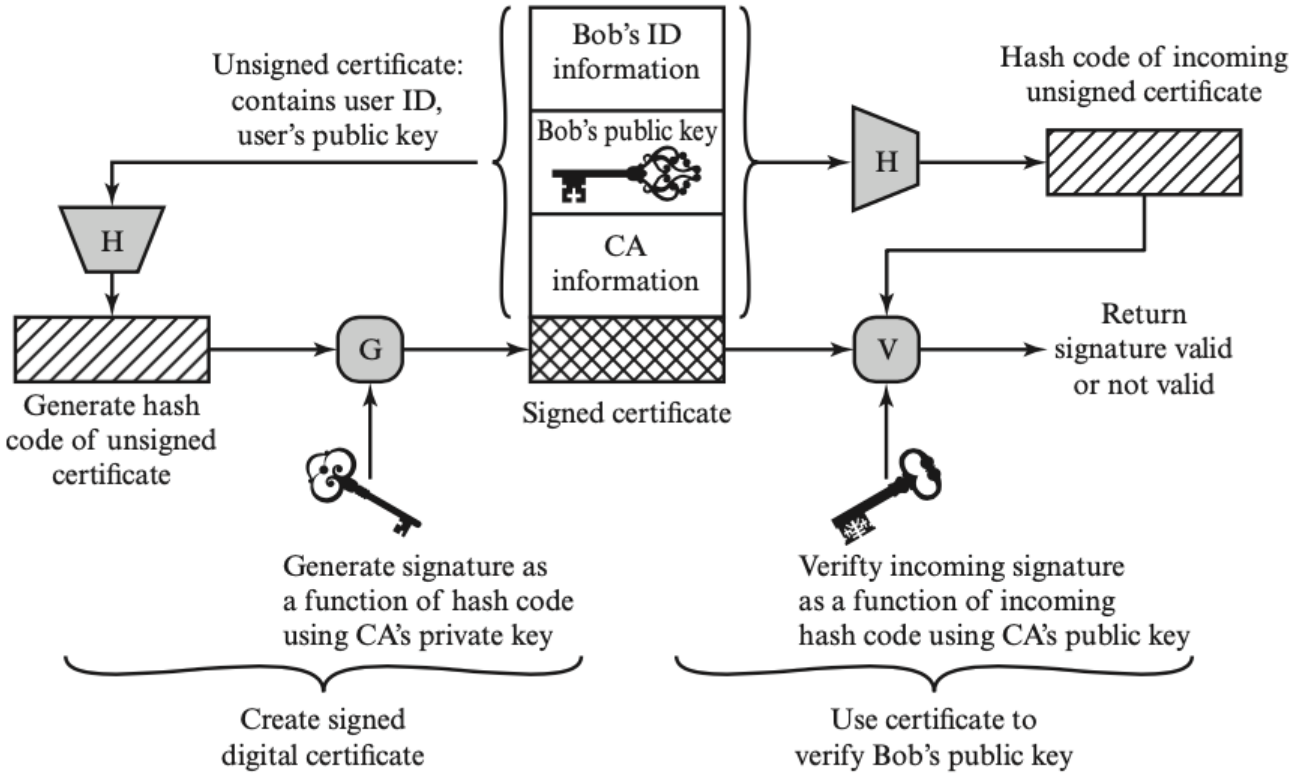
Certificates

Certificates

Certificates

- **Certificate**: A signed endorsement of someone's public key
  - A certificate contains at least two things: The **identity** of the person, and the **key**
- Abbreviated notation
  - Signing with a private key *SK*: {"Message"}$_{SK^{-1}}$
    - Recall: A signed message must contain the message along with the signature; you can't send the signature by itself!
- Scenario: Alice wants Bob's public key. Alice trusts Charlie (PK_C, SK_C)
  - Charlie is our trust anchor
- If we trust PK_C, a certificate we would trust is {"Bob's public key is PK_B"}$_{SK_C^{-1}}$

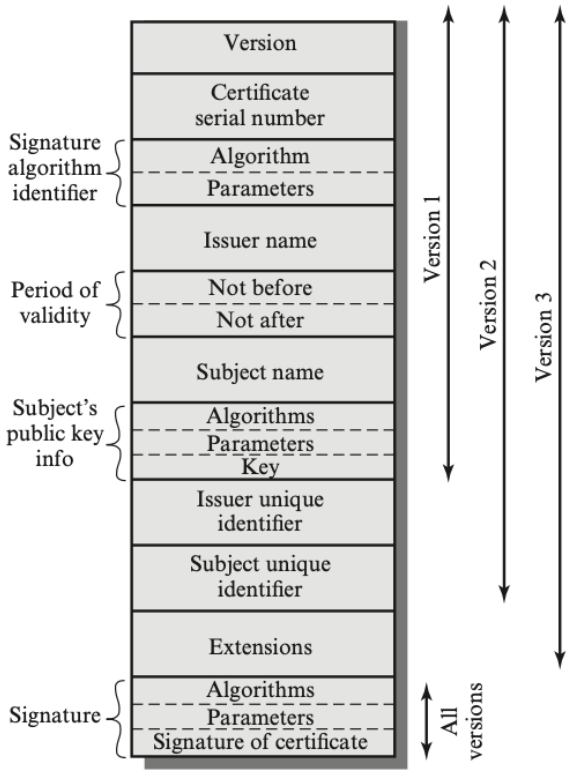
How do we use public-key certificate?



X.509 Certificates

- Certificate serial # - SN
- Period validity - T^A
- Subject's public key info - Ap
- Signature signed by CA's private key
- Math notation:

$$CA \ll A \gg = CA\{V, SN, AI, CA, UCA, A, UA, Ap, T^A\}$$



readings

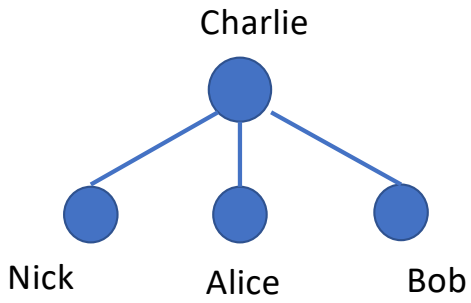
- Barnes, R.; Hoffman-Andrews, J.; McCarney, D.; Kasten, J. (March 2019). *Automatic Certificate Management Environment (ACME) RFC 8555*. IETF
- Internet Security Research Group, “Annual Report”, https://www.abetterinternet.org/annual-reports/
- Minkyu Kim, “A Survey of Kerberos V and Public-Key Kerberos Security”, https://www.cse.wustl.edu/~jain/cse571-09/ftp/kerb5/index.html

Obtaining a User Certificate

The Trusted Directory

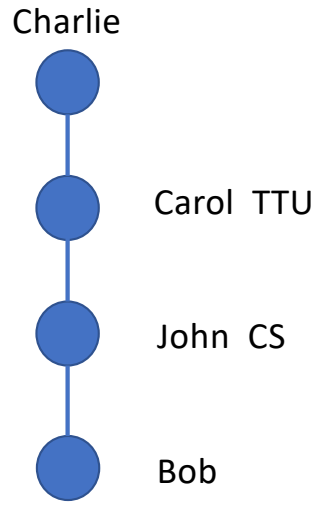
- Idea: Make a central, trusted directory (TD) from where you can fetch anybody's public key
  - The TD has a public/private keypair PK_{TD}, SK_{TD}
  - The directory publishes PK_{TD} so that everyone knows it (baked into computers, phones, OS, etc.)
  - When you request Bob's public key, the directory sends a certificate for Bob's public key
    - {"Bob's public key is PK_B"}$_{SK_{TD}^{-1}}$
  - If you trust the directory, then now you trust every public key from the directory
- What do we have to trust? (assumption)
  - We have received TD's key correctly
  - TD won't sign a key without verifying the identity of the owner

The Trusted Directory



- Let's say that Charlie (an authority) runs the TD
  - We want Nick's public key: Ask Charlie
  - We want Alice's public key: Ask Charlie
  - We want Bob's public key: Ask Charlie
  - Charlie has better things to do (like making sure his private key isn't stolen)!
- Problems?
  - Scalability: One directory won't have enough compute power to serve the entire world
  - Single point of failure:
    - If the directory fails, your application might become unavailable
    - If the directory is compromised, you can't trust anyone
    - If the directory is compromised, it is difficult to recover

Certificate Authorities



- Addressing scalability: Hierarchical trust
  - The roots of trust may **delegate** trust and signing power to other authorities
    - {"Carol Christ's public key is PK_{CC}, and I trust her to sign for TTU"}$_{SKc^{-1}}$
    - {"John Canny's public key is PK_{JC}, and I trust him to sign for the CS department"}$_{SKCC^{-1}}$
    - {"Bob's public key is PK_{Bob} (but I don't trust him to sign for anyone else)"}$_{SKJC^{-1}}$
  - Charlie is still the root of trust (**root certificate authority**, or **root CA**)
  - CC and JC receive delegated trust (**intermediate CAs**)
  - Bob's identity can be trusted
- When Alice wants to request Bob's PK, she can obtain this list of certificates (certificate chain) from any untrusted service, and does not need to contact the respective parties because the signatures are unforgeable
- Addressing scalability: Multiple trust anchors
  - There are ~150 root CAs who are implicitly trusted by most devices
  - Public keys are hard-coded into operating systems and devices

Review class & quiz 4

- Friday (Nov. 15) in class
- DHKE & Kerberos

- Thank you!