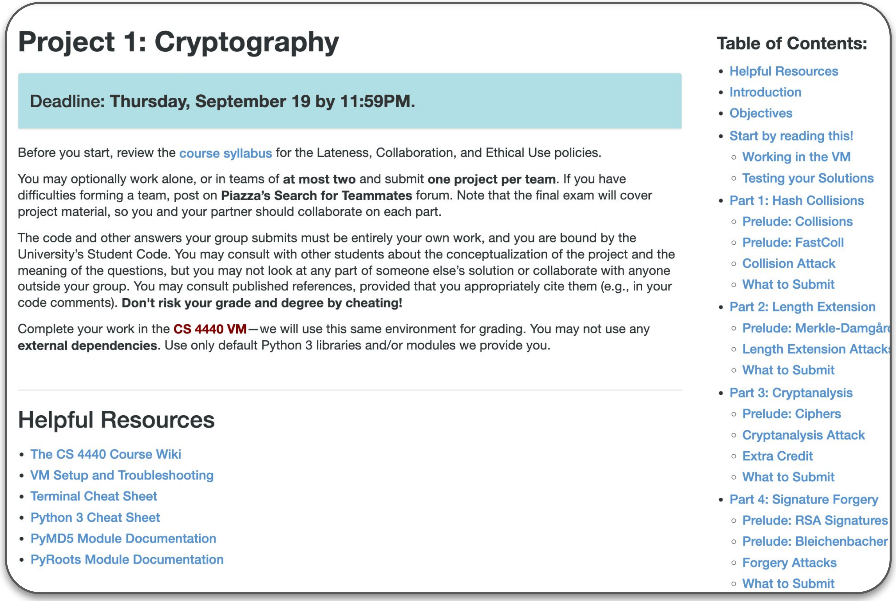
Week 4: Lecture A

Public Key Cryptography

Tuesday, September 10, 2024

Announcements

- **Project 1: Crypto** released (see **Assignments** page on course website)
  - **Deadline:** Thursday, September 19th by 11:59 PM



Project 1: Cryptography

Deadline: **Thursday, September 19 by 11:59PM.**

Before you start, review the course syllabus for the Lateness, Collaboration, and Ethical Use policies.

You may optionally work alone, or in teams of **at most two** and submit **one project per team**. If you have difficulties forming a team, post on **Piazza's Search for Teammates** forum. Note that the final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely your own work, and you are bound by the University's Student Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., in your code comments). **Don't risk your grade and degree by cheating!**

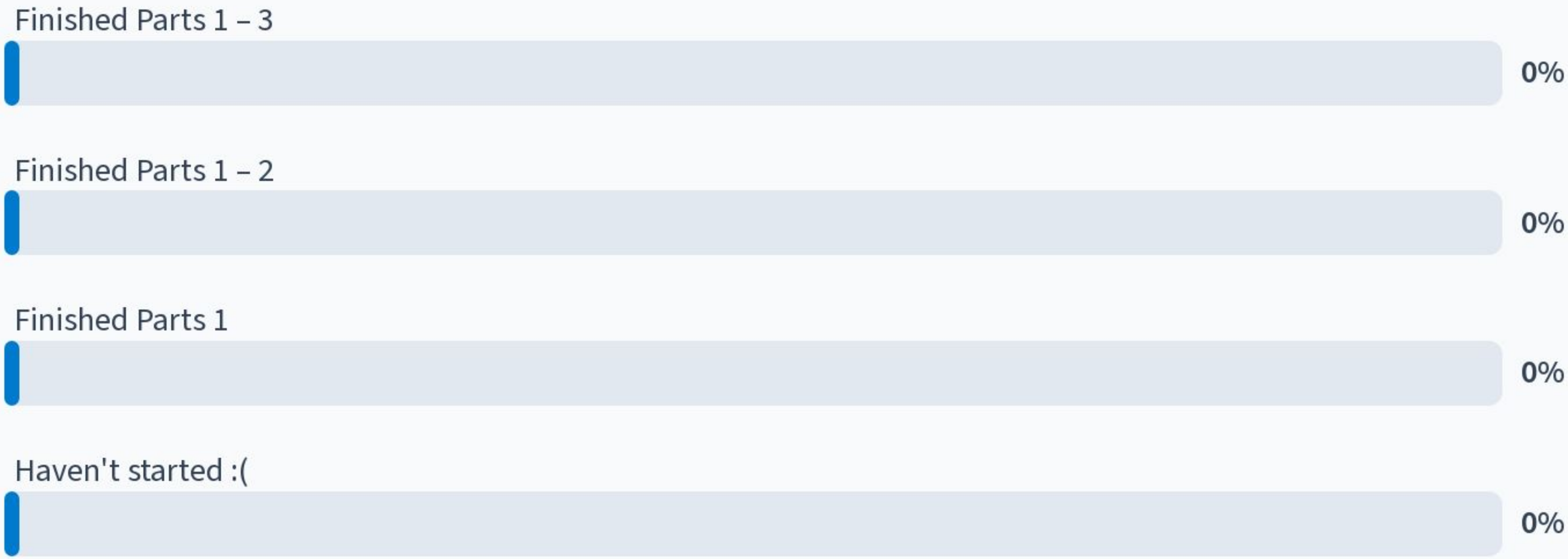
Complete your work in the **CS 4440 VM**—we will use this same environment for grading. You may not use any **external dependencies**. Use only default Python 3 libraries and/or modules we provide you.

Helpful Resources

- The CS 4440 Course Wiki
- VM Setup and Troubleshooting
- Terminal Cheat Sheet
- Python 3 Cheat Sheet
- PyMD5 Module Documentation
- PyRoots Module Documentation

Table of Contents:

Progress on Project 1



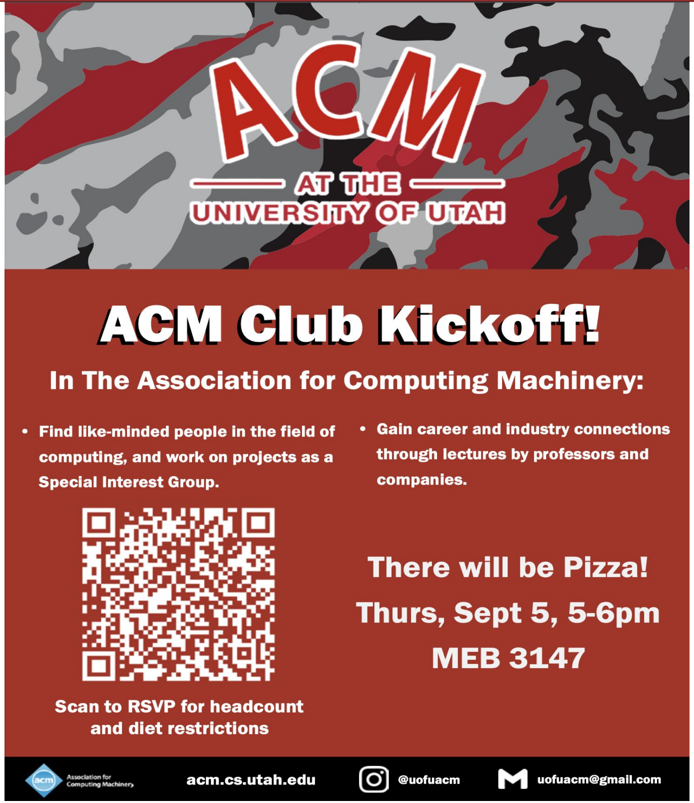
Announcements

utahsec

See Discord for meeting info!

utahsec.cs.utah.edu

Announcements



ACM

AT THE

UNIVERSITY OF UTAH

ACM Club Kickoff!

In The Association for Computing Machinery:

- Find like-minded people in the field of computing, and work on projects as a Special Interest Group.
- Gain career and industry connections through lectures by professors and companies.

There will be Pizza!

Thurs, Sept 5, 5-6pm

MEB 3147

Scan to RSVP for headcount and diet restrictions

acm.cs.utah.edu @uofuacm uofuacm@gmail.com

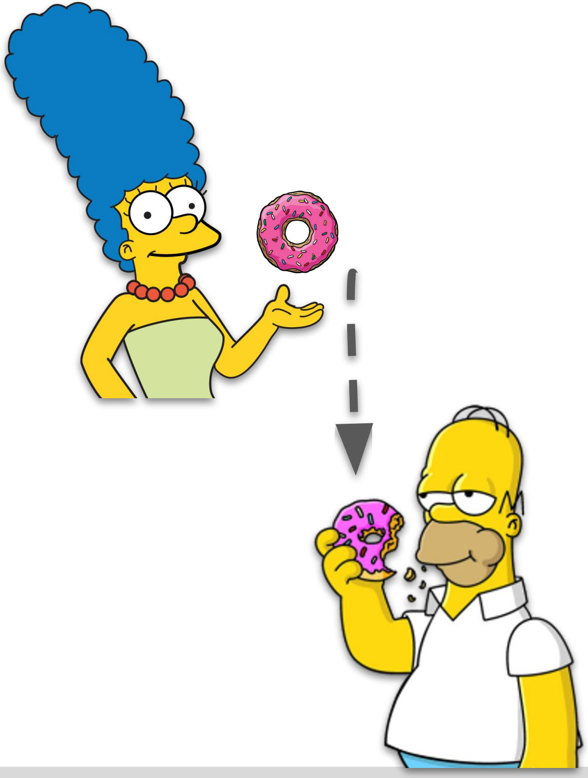
Questions?

Last time on CS 4440…

Symmetric Key Encryption
DES and AES
Block Cipher Modes
Building a Secure Channel

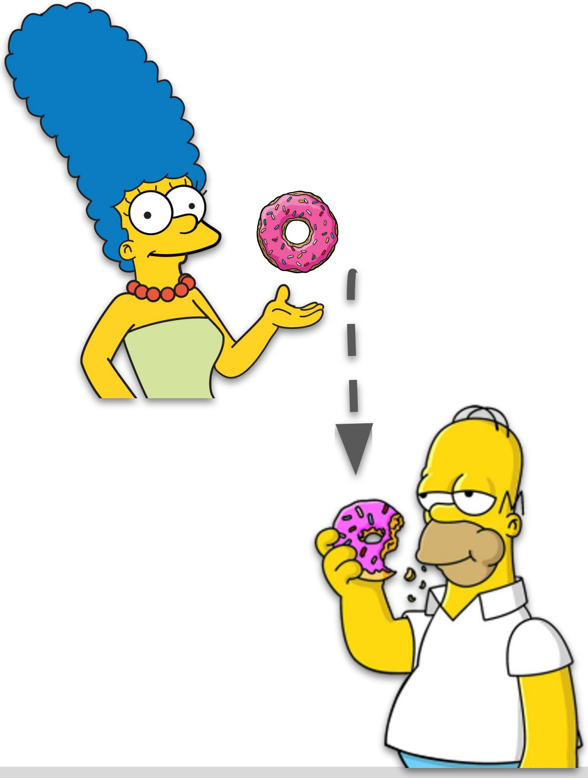
Key-based Encryption Schemes

- **"Symmetric" Key**
  - Encryption and decryption relies on **???**



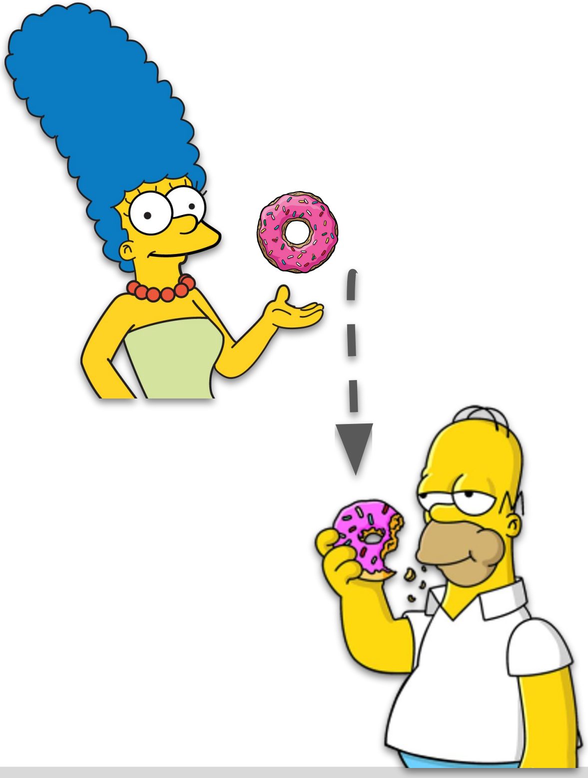
Key-based Encryption Schemes

- **"Symmetric" Key**
  - Encryption and decryption relies on **the same key**
  - Communicating parties must **???**



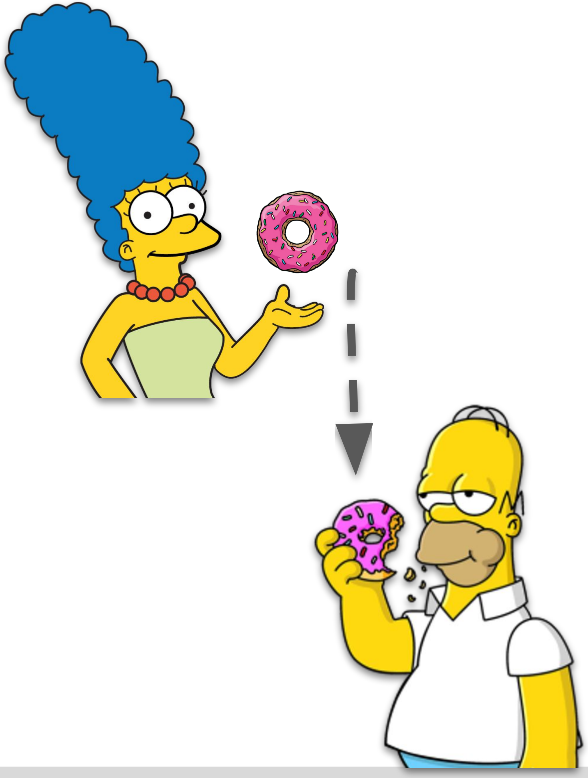
Key-based Encryption Schemes

- **"Symmetric" Key**
  - Encryption and decryption relies on **the same key**
  - Communicating parties must **share key in advance**
  - **Examples: ???**



Key-based Encryption Schemes

- **“Symmetric” Key**
  - Encryption and decryption relies on **the same key**
  - Communicating parties must **share key in advance**
  - **Examples:**
    - Caesar, Vigènere
    - One-time Pad, Stream
    - Transposition ciphers



Symmetric Key Encryption

- Categories of SKE
  - **Stream cipher:** operates on **???**

Symmetric Key Encryption

- Categories of SKE
  - **Stream cipher:** operates on **individual bits** (or bytes); **one at a time**
    - Generates pseudo-random key bits that are **XOR'd** to plaintext bits

Encryption

plaintext = H E L L O

⊕ ⊕ ⊕ ⊕ ⊕

key = A X H J B

↓ ↓ ↓ ↓ ↓

ciphertext = K M I V E

Decryption

ciphertext = K M I V E

⊕ ⊕ ⊕ ⊕ ⊕

key = A X H J B

↓ ↓ ↓ ↓ ↓

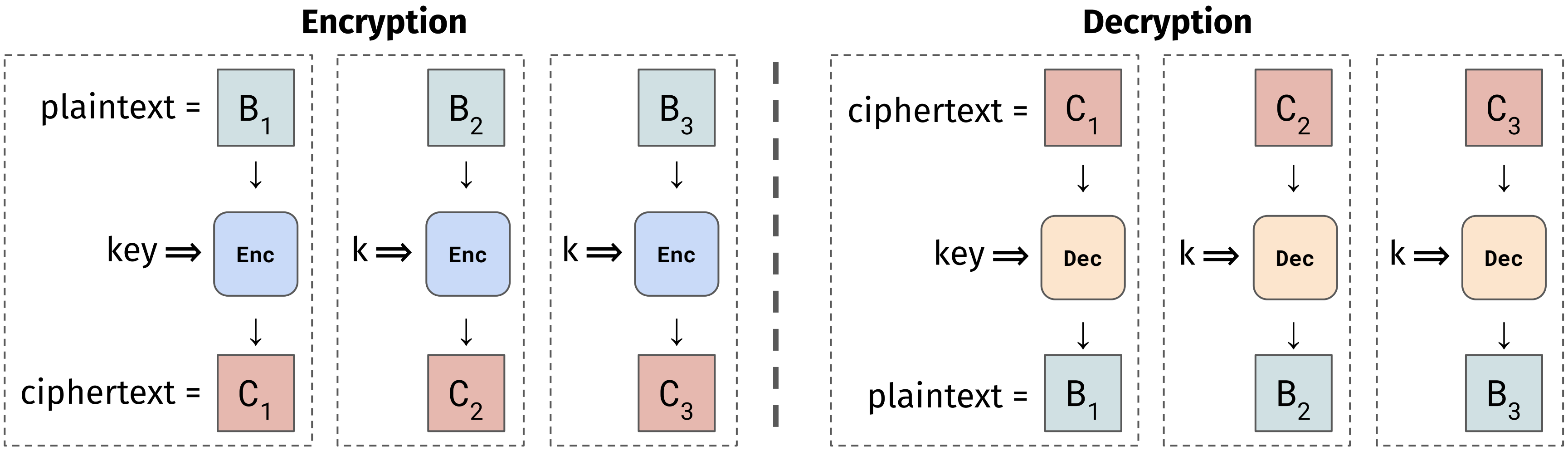
plaintext = H E L L O

Symmetric Key Encryption

- Categories of SKE
    - **Block cipher:** operates on **???**

Symmetric Key Encryption

- Categories of SKE
  - **Block cipher:** operates on **fixed-length groups** of bits called **blocks**
    - Processes blocks using a **???**



Encryption

plaintext = B_1 B_2 B_3

key ⟹ Enc, k ⟹ Enc, k ⟹ Enc

ciphertext = C_1 C_2 C_3

Decryption

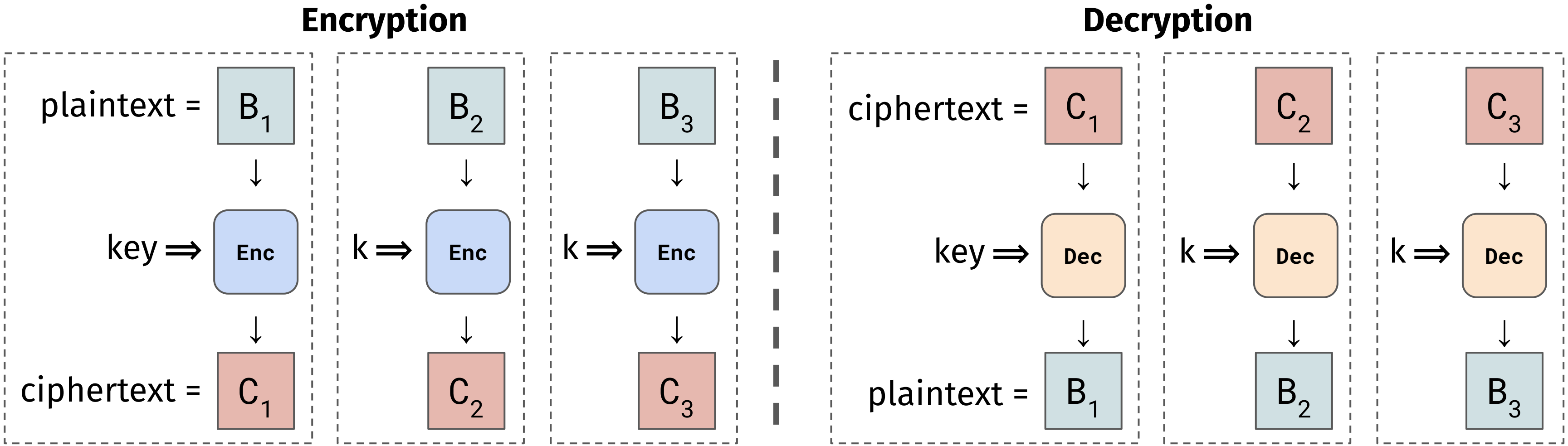
ciphertext = C_1 C_2 C_3

key ⟹ Dec, k ⟹ Dec, k ⟹ Dec

plaintext = B_1 B_2 B_3

Symmetric Key Encryption

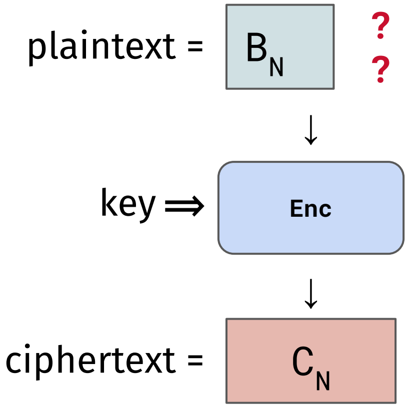
- Categories of SKE
  - **Block cipher:** operates on **fixed-length groups** of bits called **blocks**
    - Processes blocks using a **reversible**, **non-colliding** function



Handling Long Messages

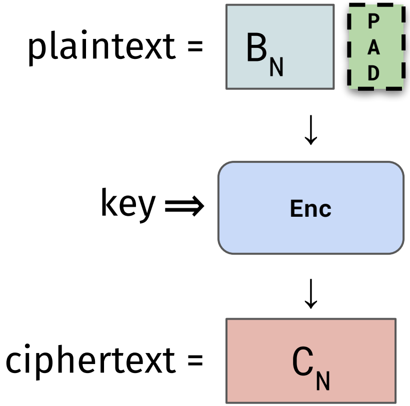
- **Challenge:** How to encrypt longer messages?
  - Can only encrypt in units of cipher block size…
  - But message might not be **multiples** of block size

- **Solution:** ???



Handling Long Messages

- **Challenge:** How to encrypt longer messages?
  - Can only encrypt in units of cipher block size...
  - But message might not be **multiples** of block size
- **Solution:** Append padding to end of message
  - Must be able to recognize and remove padding afterward
  - Common approach: add **n** bytes that have value **n**

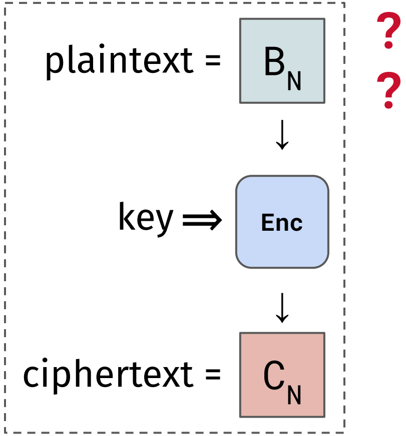

plaintext = B_N PAD

key ⟹ **Enc**

ciphertext = C_N

Handling Long Messages

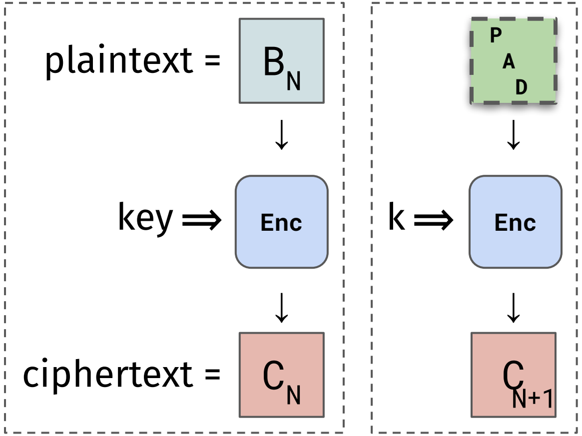
- **Challenge:** What if message terminates a block?
  - End of message might be misread as padding!
- **Solution:** ???



Handling Long Messages

- **Challenge:** What if message terminates a block?
  - End of message might be misread as padding!

- **Solution:** Append an entire new block of padding
  - Padding is necessary to know we're at message end



DES Modes: Electronic Codebook (ECB)

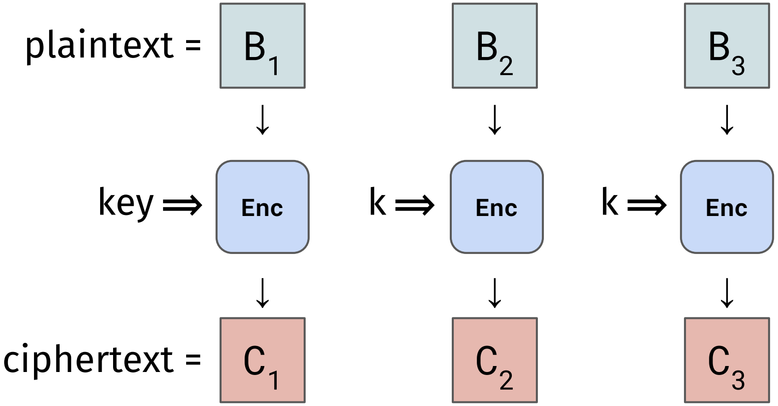
- Electronic Codebook (**ECB**)
  - Message divided into code blocks
  - Each block encrypted/decrypted **???**

plaintext = B_1 B_2 B_3

↓ ↓ ↓

DES Modes: Electronic Codebook (ECB)

- Electronic Codebook (**ECB**)
  - Message divided into code blocks
  - Each block encrypted/decrypted **separately**



plaintext = B_1 B_2 B_3

key ⇒ **Enc** k ⇒ **Enc** k ⇒ **Enc**

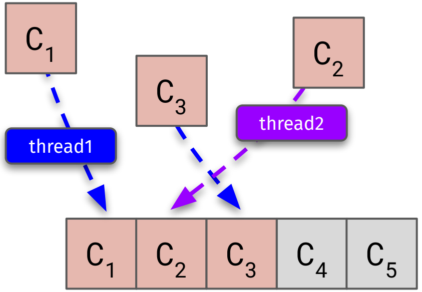
ciphertext = C_1 C_2 C_3

DES Modes: Electronic Codebook (ECB)

- **ECB Strengths: ???**

DES Modes: Electronic Codebook (ECB)

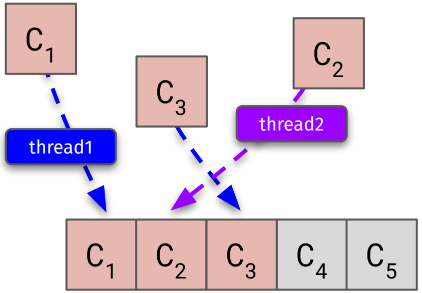
- **ECB Strengths:**
    - Construction is **un-chained**
        - Message can be **processed in parallel**—fast!
        - No wait on previous block's encryption



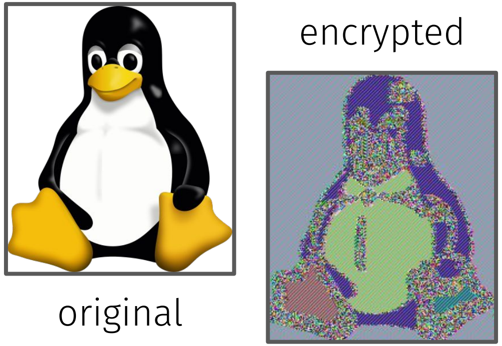
- **ECB Drawbacks: ???**

DES Modes: Electronic Codebook (ECB)

- **ECB Strengths:**
  - Construction is **un-chained**
    - Message can be **processed in parallel**—fast!
    - No wait on previous block's encryption

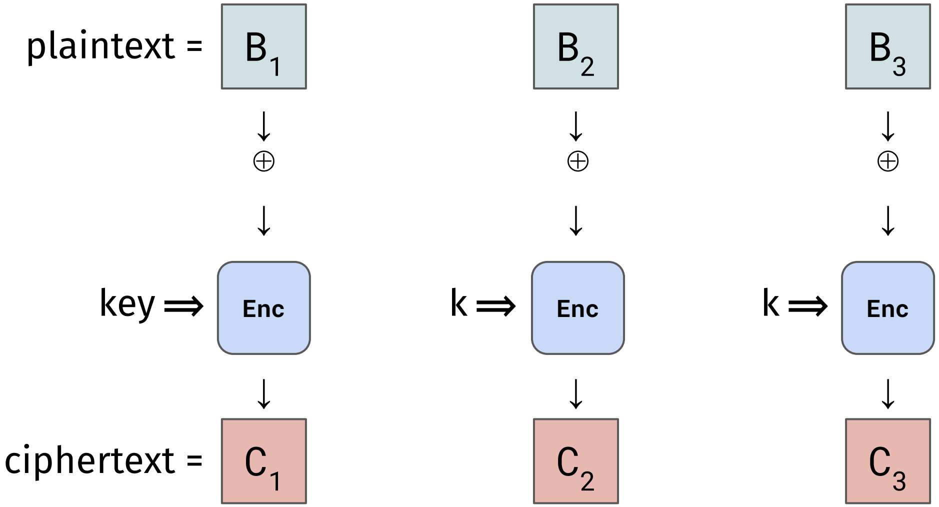


- **ECB Drawbacks:**
  - Identical plaintext blocks produce same ciphertext
    - This results in **low diffusion**
  - Do **larger block sizes** increase diffusion?
    - Yes—but at cost of **higher memory footprint**



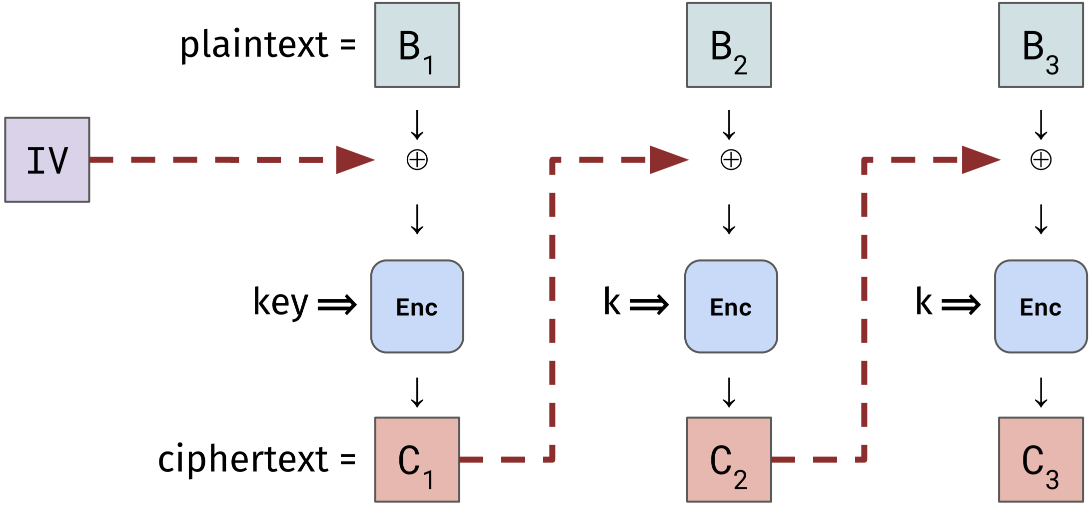
DES Modes: Cipher Block Chaining (CBC)

- Cipher Block Chaining (**CBC**):
  - Construction is **???**



DES Modes: Cipher Block Chaining (CBC)

- Cipher Block Chaining (**CBC**):
  - Construction is **chained using previous cipher block** (initialization vector for first block)



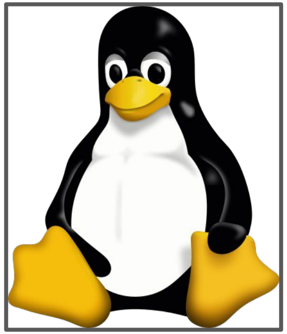
DES Modes: Cipher Block Chaining (CBC)

- **CBC Strengths: ???**

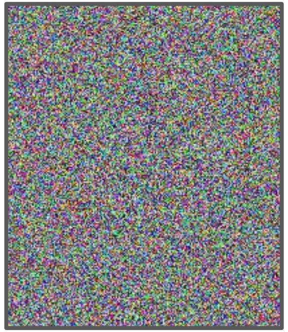
DES Modes: Cipher Block Chaining (CBC)

- **CBC Strengths:**
  - Chained construction far stronger than ECB
    - **More diffusion!**
    - Negates ECB’s need for super-large blocks

- **CBC Drawbacks: ???**

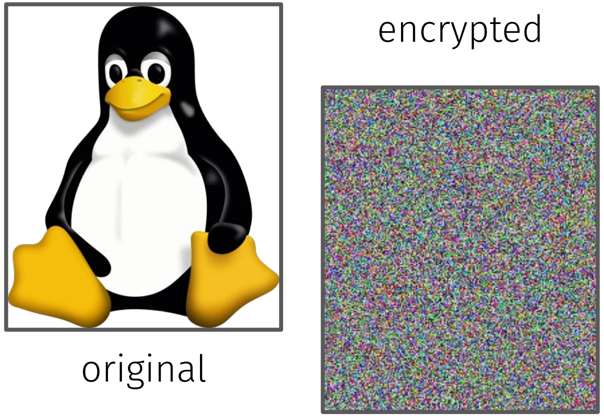

original

encrypted



DES Modes: Cipher Block Chaining (CBC)

- **CBC Strengths:**
  - Chained construction far stronger than ECB
    - **More diffusion!**
    - Negates ECB's need for super-large blocks
- **CBC Drawbacks:**
  - Completely sequential
    - **Cannot be parallelized—slower to process!**
    - No leveraging advances in multi-threading etc.


encrypted

original

Exercise: Stream vs. Block Ciphers

Cipher	**Must wait for data?**	**Parallel processing?**	**Confusion?**	**Diffusion?**
Stream Ciphers				
Block Ciphers				

Exercise: Stream vs. Block Ciphers

Cipher	**Must wait for data?**	**Parallel processing?**	**Confusion?**	**Diffusion?**
Stream Ciphers	**No**	**No**	**Yes**	**No**
Block Ciphers	**Yes**	**Yes**	**Yes**	**Yes**

Questions?

This time on CS 4440…

Key Exchange
Diffie Hellman
RSA
Attacking RSA
Key Management

Key Exchange

Recap: Integrity

- **Problem:** Send message via untrusted channel without being changed

- **Provably-secure solution:** truly random function (e.g., LavaRand)

- **Practical solution:** Pseudo-random Function Family (**PRF**)
  - **Input:** arbitrary-length key **k**
  - **Output:** fixed-length **message digest**
  - Secure if practically indistinguishable from a random function (**unless Mallory knows k**)

- **Real-world:** message authentication codes built with cryptographic hashes
  - E.g., **HMAC-SHA256$_k$(m)**

Recap: Confidentiality

- **Problem:** Send message with secrecy in presence of an eavesdropper

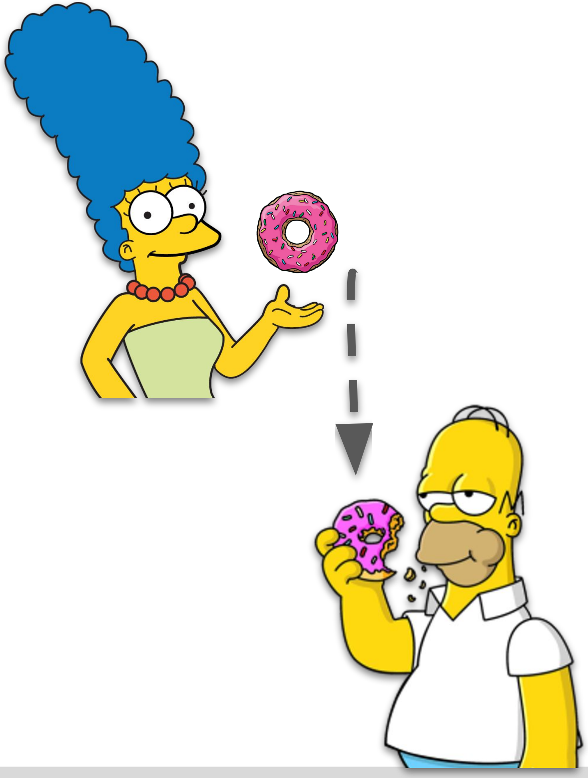
- **Provably-secure solution:** one-time pad with a key as long as **m**

- **Practical solution:** Pseudo-random Generator (**PRG**)
  - **Input:** a small, truly random seed
  - **Output:** arbitrary-length **key stream**
  - Secure if practically indistinguishable from a random stream (**unless Mallory knows k**)

- **Real-world:** steam ciphers, block ciphers
  - E.g., **AES-128 + CBC mode**

Integrity and Confidentiality

- Common theme: **???**

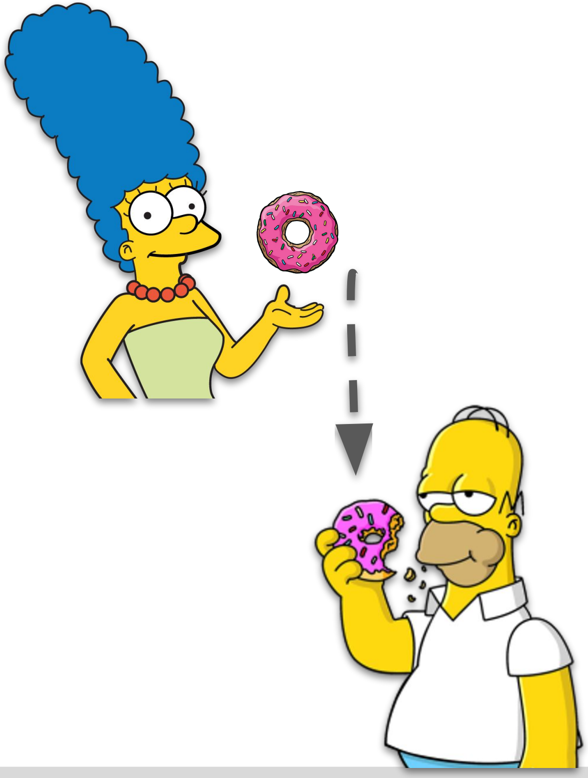


Integrity and Confidentiality

- Common theme: **the key**

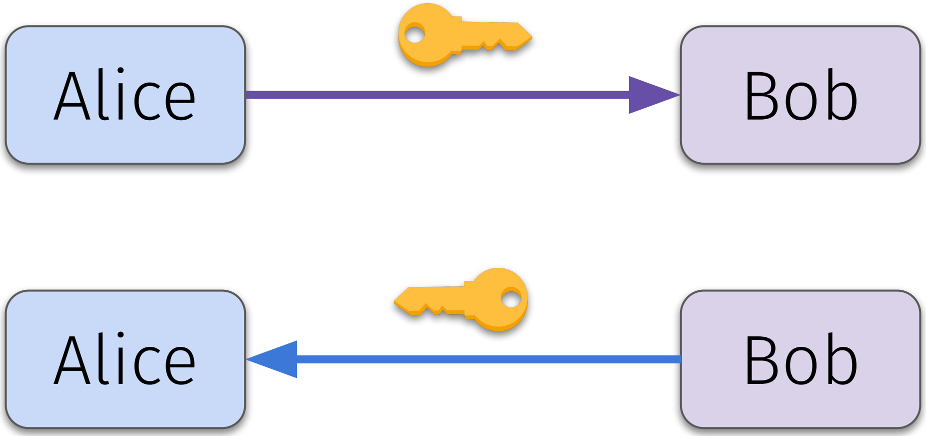
- Key requirements
  - Must be known by both Alice and Bob
  - Must be unknown by anyone else
  - Must be infeasible to guess

- We'd like Alice and Bob to agree on a key that satisfies those properties by sending **public messages** to each other



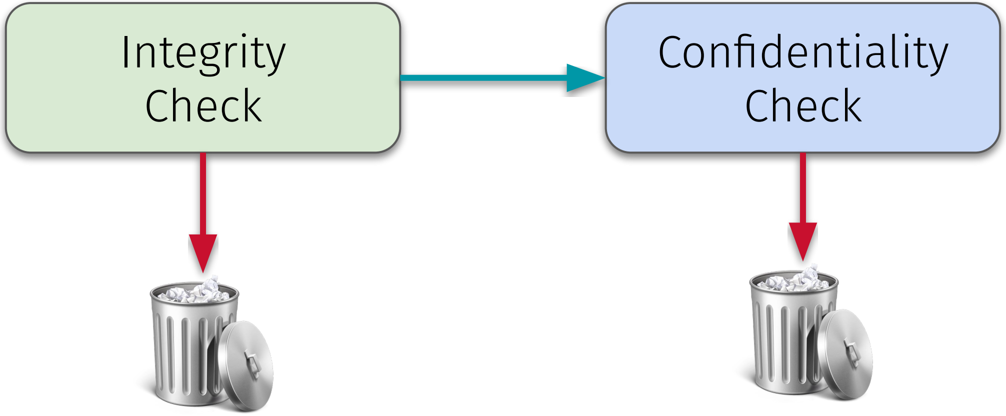
Multi-party Secure Communication

- **Required initialization:** pre-sharing the key
    - Total keys to be shared: **at most <u>two</u>**



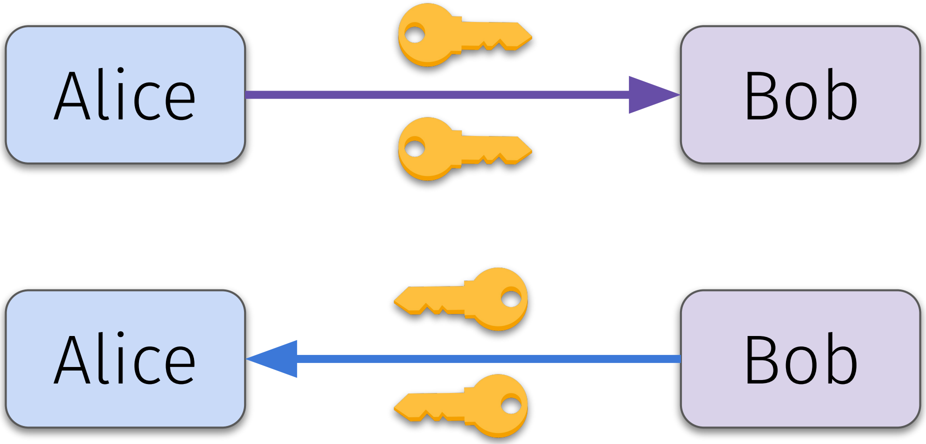
Recap: Secure Channels

- What if you want **confidentiality** and **integrity** at **the same time**?
    - Which would you perform **first**: encrypting or hashing? And why?



Multi-party Secure Communication

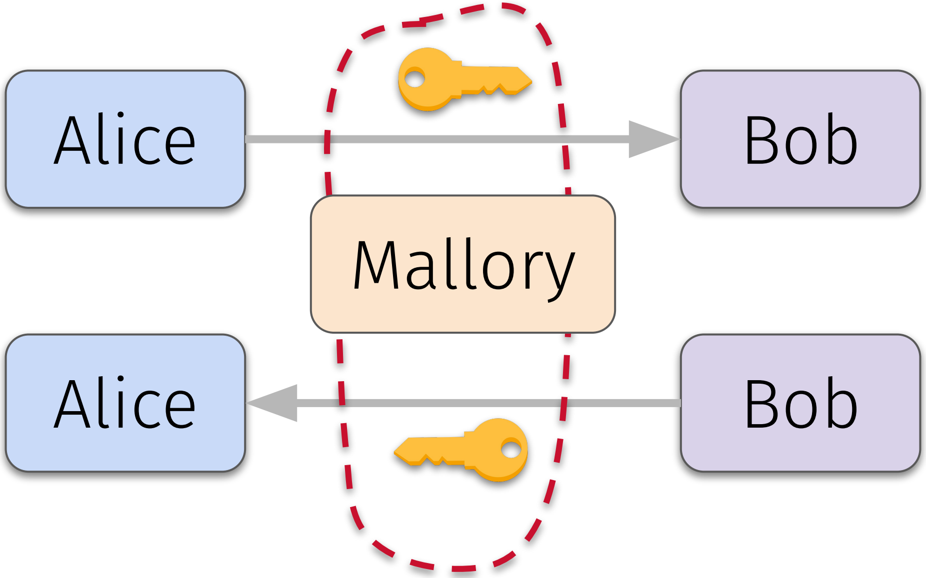
- **Required initialization:** pre-sharing the key
  - Total keys to be shared: **at most <u>two</u>**
  - **<u>Four</u>** if you want **confidentiality** *and* **integrity**



One set of keys for **integrity**, another for **confidentiality**

Multi-party Secure Communication

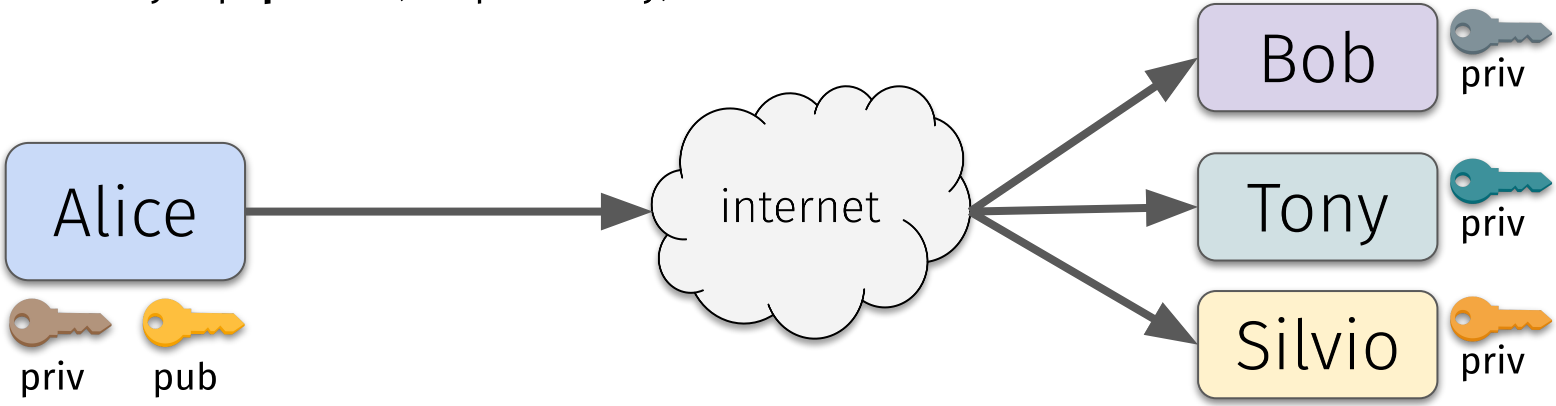
- **Problem:** all keys must be shared **securely**
  - What if Mallory intercept our key?
  - **Man in the Middle attack** (MITM)



Asymmetric Encryption (aka "Public Key")

- **Key idea:** want a **asymmetric** approach to find a **symmetric** key
  - Don't want to have to pre-share keys in advance

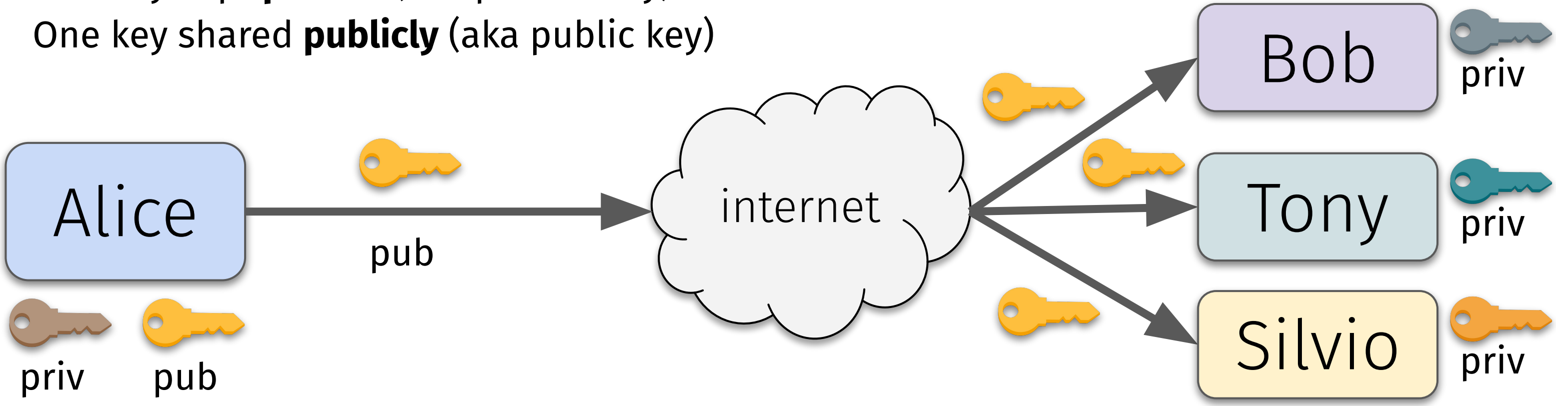
- Suppose users can have **two keys:** encryption and decryption
  - Keys generated in pairs using well-understood mathematical relationship
  - One key kept **private** (aka private key)



Asymmetric Encryption (aka “Public Key”)

- **Key idea:** want a **asymmetric** approach to find a **symmetric** key
  - Don’t want to have to pre-share keys in advance

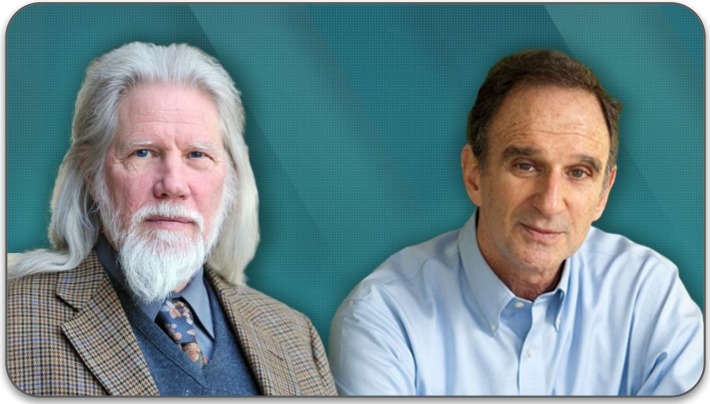
- Suppose users can have **two keys:** encryption and decryption
  - Keys generated in pairs using well-understood mathematical relationship
  - One key kept **private** (aka private key)
  - One key shared **publicly** (aka public key)

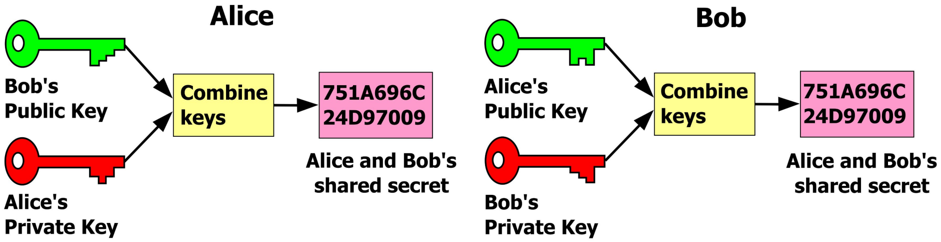


Diffie-Hellman Key Exchange

Diffie Hellman

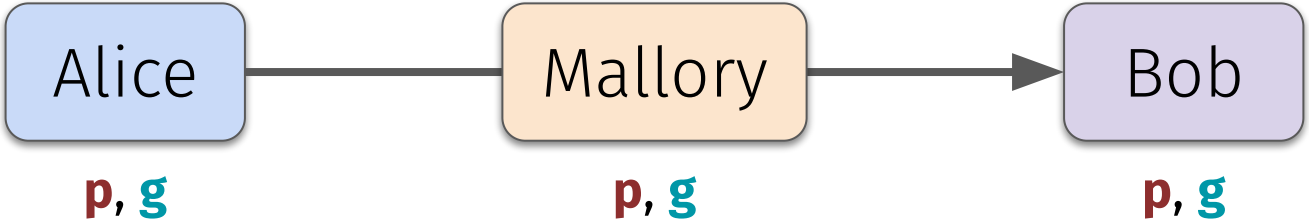
- Protocol for **public key exchange**
  - Forward secrecy via a public conversation **without** any pre-shared information
- Relies on a mathematical hardness assumption called **discrete log problem** (a problem believed to be **NP-hard**)





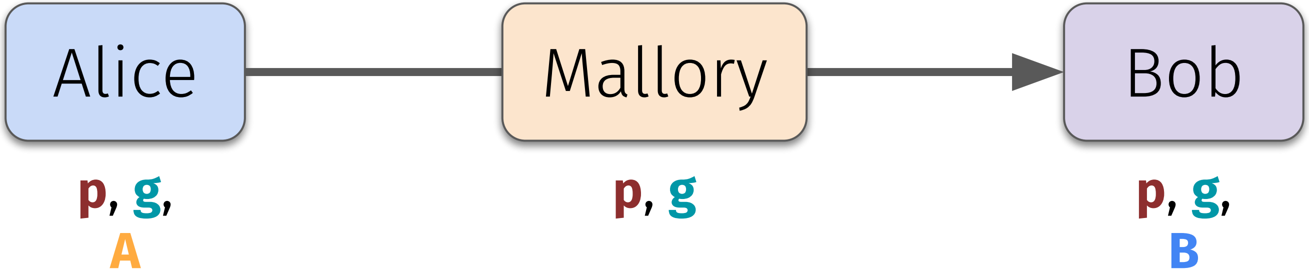
Diffie Hellman

1. **Initialization:** Alice and Bob agree on protocol parameters
   - **p** : a large prime such that **(p-1) / 2** is **also prime**
   - **g** : a small integer called the generator (e.g., **2**)
   - This is likely in a standard



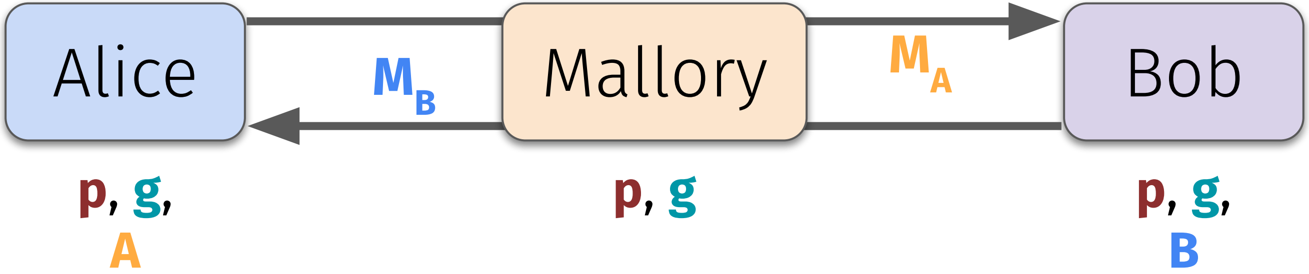
Diffie Hellman

2. **Secret Generation:** Alice and Bob independently generate secret values
   - … such that: `0 < secret_value < p`
   - A : **Alice's** secret value
   - B : **Bob's** secret value



Diffie Hellman

3. **Transmit Secret:** Alice and Bob <u>independently</u> create, exchange a message
    - $M_A = g^{A} \bmod p$
    - $M_B = g^{B} \bmod p$

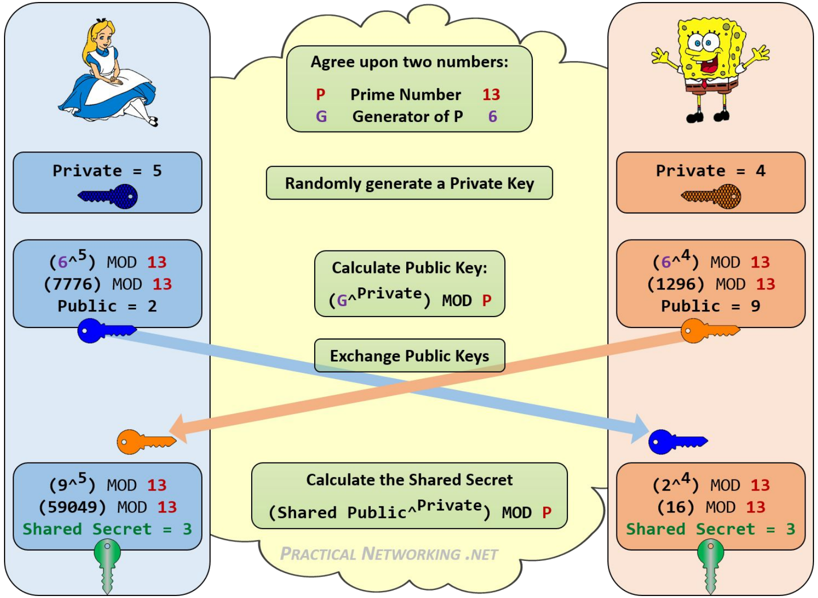


Diffie Hellman

4. Circular Mixing:

- Alice computes: X_A
$$\begin{aligned} X_A &= (M_B)^A \bmod p \\ &= (g^B \bmod p)^A \bmod p \\ &= g^{BA} \bmod p \end{aligned}$$

- Bob computes: X_B
$$\begin{aligned} X_B &= (M_A)^B \bmod p \\ &= (g^A \bmod p)^B \bmod p \\ &= g^{AB} \bmod p \end{aligned}$$



Agree upon two numbers:
P Prime Number 13
G Generator of P 6

Randomly generate a Private Key

Calculate Public Key:
$(G^{Private})$ MOD P

Exchange Public Keys

Calculate the Shared Secret
$(Shared\ Public^{Private})$ MOD P

Private = 5

(6^5) MOD 13
(7776) MOD 13
Public = 2

(9^5) MOD 13
(59049) MOD 13
Shared Secret = 3

Private = 4

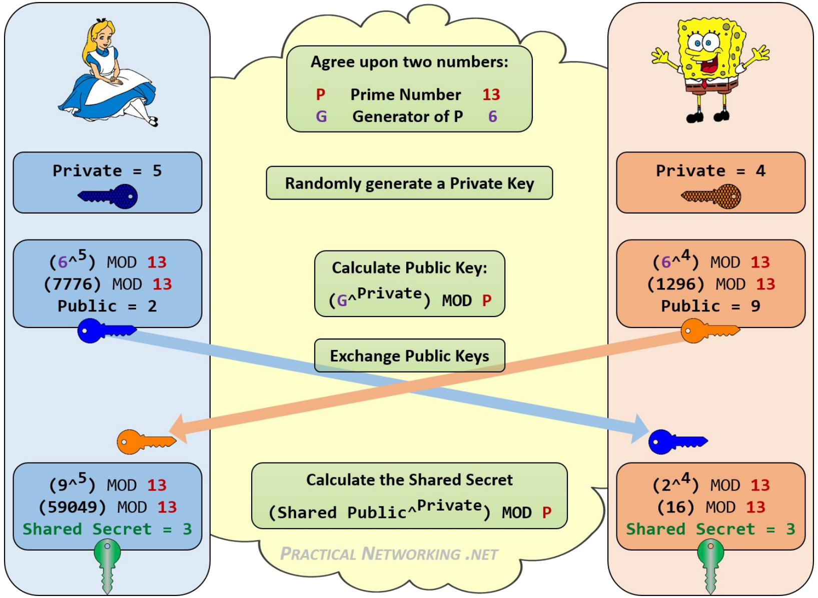
(6^4) MOD 13
(1296) MOD 13
Public = 9

(2^4) MOD 13
(16) MOD 13
Shared Secret = 3

PRACTICAL NETWORKING .NET

Diffie Hellman

4. **Circular Mixing:**
   - Alice computes: X_A $= (M_B)^A \bmod p$
     $= (g^B \bmod p)^A \bmod p$
     $= g^{BA} \bmod p$
   - Bob computes: X_B $= (M_A)^B \bmod p$
     $= (g^A \bmod p)^B \bmod p$
     $= g^{AB} \bmod p$
   - Observe that $X_A == X_B = X$



Agree upon two numbers:

P Prime Number 13

G Generator of P 6

Randomly generate a Private Key

Calculate Public Key:

(G^Private) MOD P

Exchange Public Keys

Calculate the Shared Secret

(Shared Public^Private) MOD P

Private = 5

(6^5) MOD 13

(7776) MOD 13

Public = 2

(9^5) MOD 13

(59049) MOD 13

Shared Secret = 3

Private = 4

(6^4) MOD 13

(1296) MOD 13

Public = 9

(2^4) MOD 13

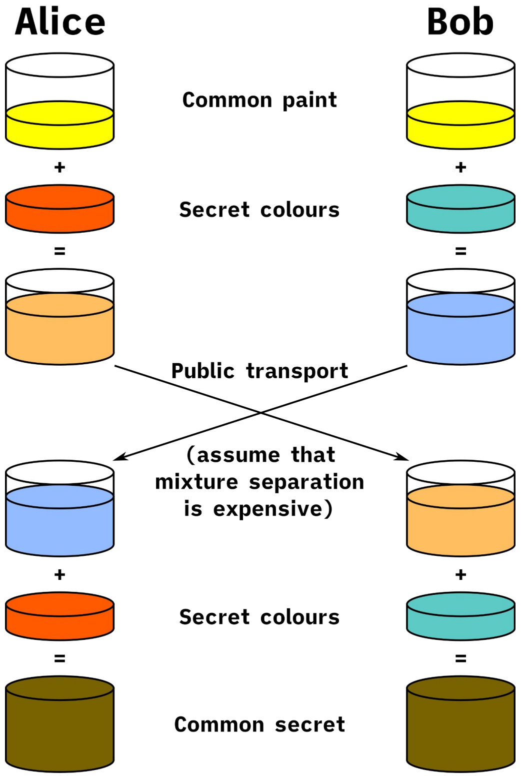
(16) MOD 13

Shared Secret = 3

5. Alice and Bob derive **k := HMAC$_0$(X)** as their **shared key**

A visual analogy of Diffie-Hellman

- Mixing in a new color is a little bit like Diffie-Hellman's exponentiation
- Hard to **invert** to original colors? **Yes!**
- Two different ways of arriving to the same final result (i.e., the shared key)



Questions?

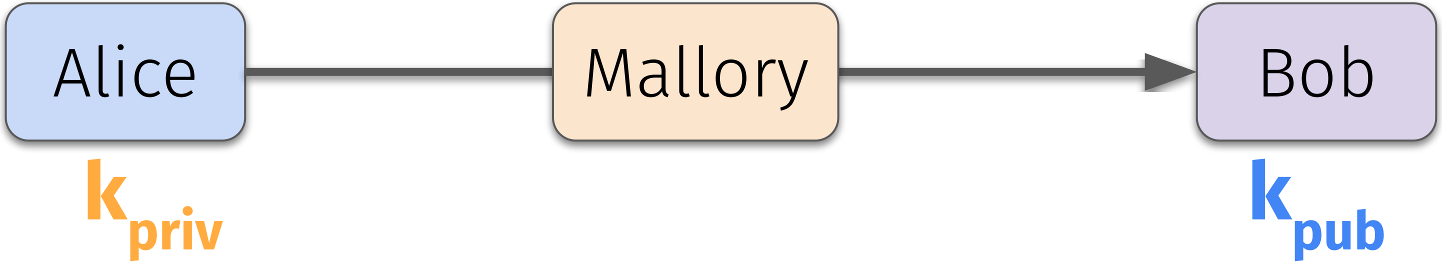
RSA

Authenticity

- So far we've talked about **confidentiality** via public-key **encryption**
- Suppose Alice messages many people that all want to verify **authenticity**
  - They want to know a message came **from Alice**—not someone else!
- Alice can't share an **authenticity key** with everybody…
  - Or else anybody—like Mallory—could **forge** messages!
- **Real-world example:** administrator of a source code repository
  - If anyone had Alice's authenticity key, they could submit **fraudulent code patches**!

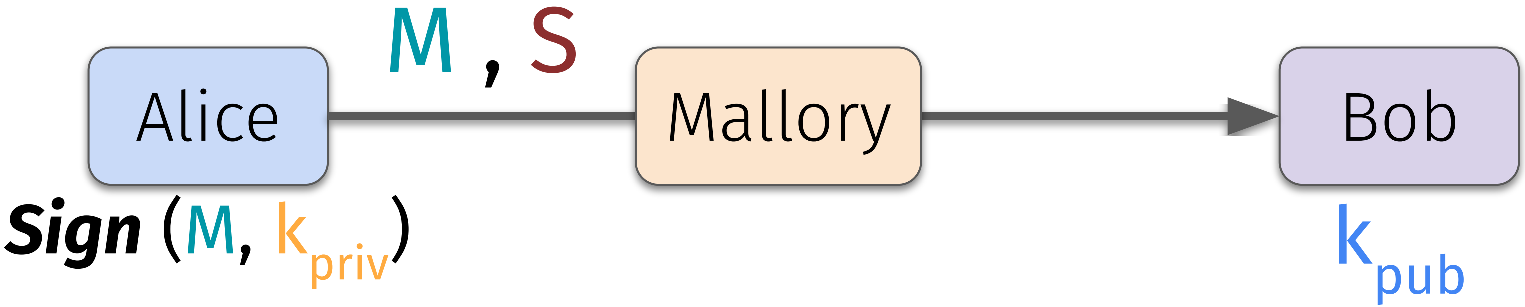
Authenticity via Digital Signatures

- **Key generation:** Alice generates key pair: k_{pub} (public) and k_{priv} (private)



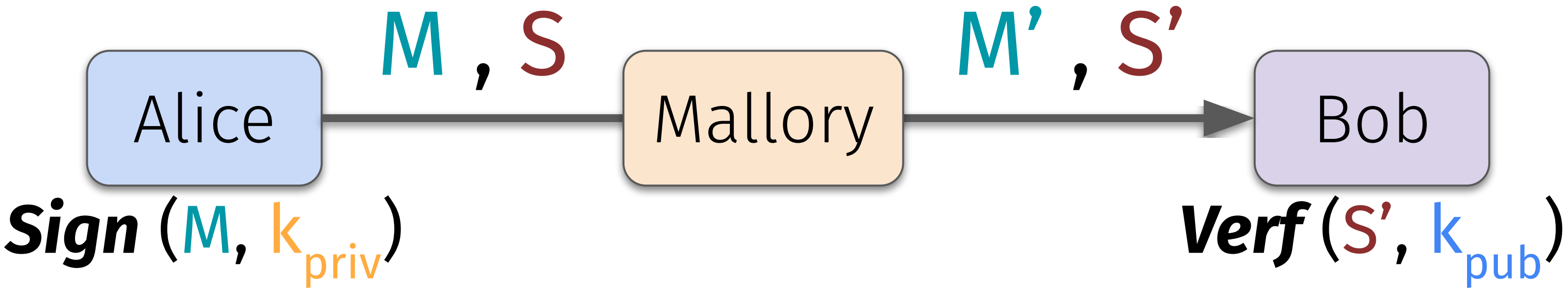
Authenticity via Digital Signatures

- **Key generation:** Alice generates key pair: k_{pub} (public) and k_{priv} (private)
- Alice **signs** message **M** with k_{priv} resulting in signature **S** = ***Sign*** (**M**, k_{priv})



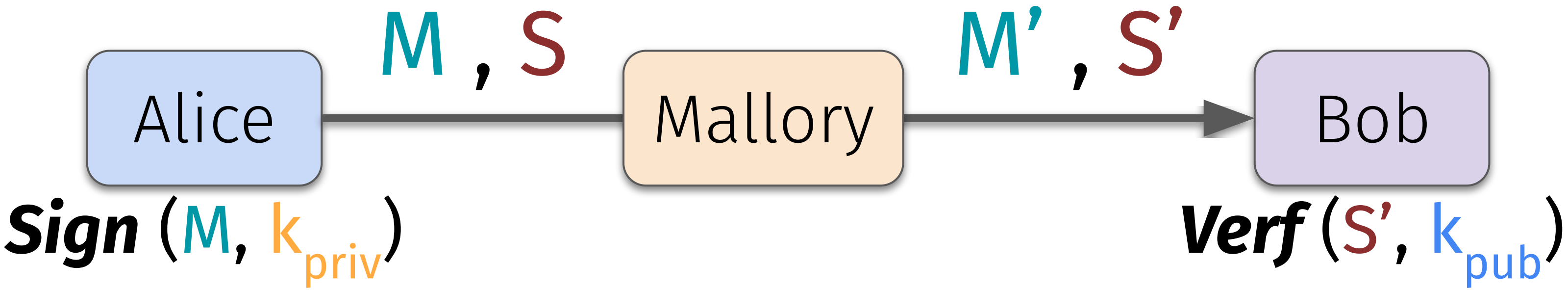
Authenticity via Digital Signatures

- **Key generation:** Alice generates key pair: k_{pub} (public) and k_{priv} (private)
- Alice **signs** message **M** with k_{priv} resulting in signature **S** = ***Sign*** (**M**, k_{priv})
- **Anyone** possessing Alice's k_{pub} can **check** signature via ***Verf*** (**S'**, k_{pub})



Authenticity via Digital Signatures

- **Key generation:** Alice generates key pair: k_{pub} (public) and k_{priv} (private)
- Alice **signs** message **M** with k_{priv} resulting in signature **S** = ***Sign*** (**M**, k_{priv})
- **Anyone** possessing Alice's k_{pub} can **check** signature via ***Verf*** (**S'**, k_{pub})
  - If received message and signature **verified**, then message is **authentic**—from Alice!



Authenticity via Digital Signatures

- **Key generation:** Alice generates key pair: k_{pub} (public) and k_{priv} (private)
- Alice **sig** … M, k_{priv})
- **Anyone** … k_{pub})
  - If received message and signature **verified**, then message is **authentic**—from Alice!

Unforgeability: computationally **infeasible** for **Mallory** to guess **S** or Alice's k_{priv}

… even if **Mallory knows** Alice's k_{pub} or other signatures from other messages!

Ali … ob

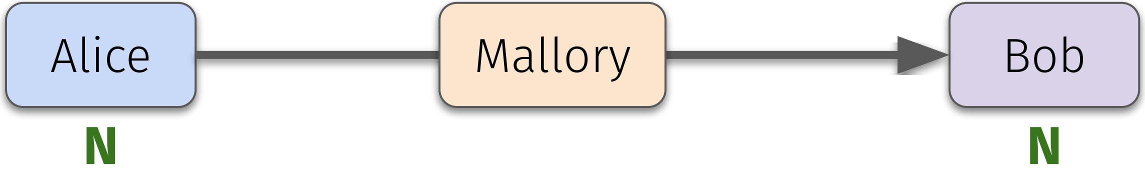
Sign (M, k_{priv}) … ***Verify*** (S', k_{pub})

RSA

- A scheme for **public-key encryption**
  - We'll use it primarily for **digital signatures**
- Best know and most common algorithm for public-key message encryption
- Relies on **integer factorization problem** (maybe believed to be **NP-hard**?)
- Inspired by Diffie-Hellman!

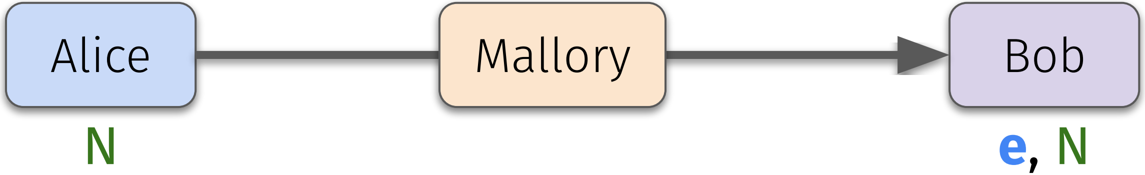
RSA Digital Signatures

1. Pick **large** (e.g., 1024 bits), and **random**, and **prime** numbers **p** and **q**
    - **N** = **p** * **q**
    - **N** serves as the **modulus** for exponentiation



RSA Digital Signatures

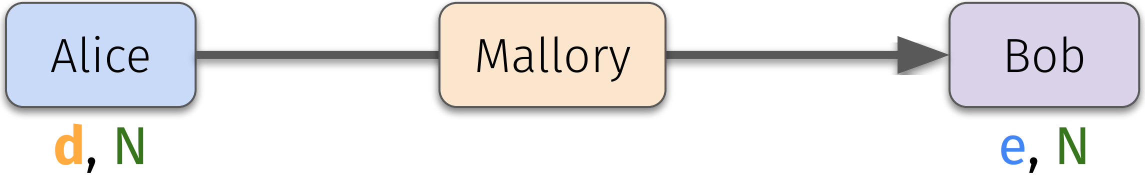
2. **Public key** = (e, N) where e is **relatively prime** to $(p-1)(q-1)$



RSA Digital Signatures

2. **Public key** = `(e,N)` where `e` is **relatively prime** to `(p-1)(q-1)`

3. **Private key** = `(d,N)` where `(e*d)` **`mod`** `((p-1)(q-1)) = 1`

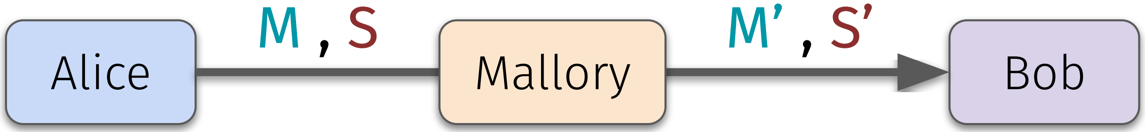


RSA Digital Signatures

2. **Public key** = (e, N) where e is **relatively prime** to $(p-1)(q-1)$
3. **Private key** = (d, N) where $(e*d)$ **mod** $((p-1)(q-1)) = 1$
4. Alice signs: S = ***Sign***$(M, d, N) = (M)^d \bmod N$

RSA Digital Signatures

2. **Public key** = (e , N) where e is **relatively prime** to (p −1) (q −1)
3. **Private key** = (d , N) where (e*d) **mod** ((p −1) (q −1)) = 1
4. Alice signs: S = ***Sign*** (M, d, N) = $(M)^d$ mod N
5. Bob verifies: ***Verif*** (S’, e, N) = $(S')^e$ mod N == M’



Sign (M, d, N)

Verif (S’, e, N) == M’

Messages as Integers

- Here, message **M** really means a really-large **integer**
  - Both Alice and Bob generate these from the plaintext message
- Transmitted/received **alongside the plaintext** message
  - Used by both Alice/Bob in signature generation/verification
- Example based on PKCS #1 v1.5 standard:

$$00\ 01\ \underbrace{FF\ FF\ FF\ \cdots\ FF}_{k/8-38\text{ bytes}}\ 00\ \underbrace{30\ 21\ 30\ 09\ 06\ 05\ 2B\ 0E\ 03\ 02\ 1A\ 05\ 00\ 04\ 14}_{\text{ASN.1 “magic” bytes denoting type of hash algorithm}}\ \underbrace{\underbrace{XX\ XX\ XX\ XX\ \cdots\ XX}_{\text{SHA-1 digest (20 bytes)}}}_{\texttt{SHA1(“Go Chiefs!”)}}$$

RSA for Confidentiality and Integrity

- Subtle fact: RSA can also be used for **integrity** and **confidentiality**

- RSA for **integrity**:
  - **Goal:** Prove that message wasn't tampered
  - Encrypt ("**sign**") with sender's **private key**
  - Decrypt ("**verify**") with sender's **public key**

RSA for Confidentiality and Integrity

- Subtle fact: RSA can also be used for **integrity** and **confidentiality**

- RSA for **integrity**:
  - **Goal:** Prove that message wasn't tampered
  - Encrypt ("**sign**") with sender's **private key**
  - Decrypt ("**verify**") with sender's **public key**

- RSA for **confidentiality**:
  - **Goal:** Allow only intended recipient to read
  - **Encrypt** with recipient's **public key**
  - **Decrypt** with recipient's **private key**

Using RSA

- To generate an RSA **key-pair**:

```
$ openssl genrsa -out private.pem 1024
$ openssl rsa -pubout -in private.pem > public.pem
```

- To **sign** a message with RSA:

```
$ openssl rsautl -sign -inkey private.pem -in a.txt > sig
```

- To **verify** a signed message with RSA:

```
$ openssl rsautl -verify -pubin -inkey public.pem -in sig
```

Recap: Advanced Encryption Standard (AES)

- Today's most common block cipher
  - Designed by NIST competition, with a very long public discussion
  - Widely believed to be secure… but we don't know how to prove it

- Variable **key size**:
  - 128-bit fairly common; also 192-bit and 256-bit versions

- Input message is split into **128-bit blocks**

- Ten **rounds**:
  - Split **k** into ten subkeys (key scheduling)
  - Performs set of identical operations ten times (each with different subkey)

RSA vs. AES

- RSA is **1000x slower** than AES

- RSA is **more complex** than AES

- RSA has **10x larger keys** than AES (e.g., 2048 bits vs. 192 bits)

RSA vs. AES

- RSA is **1000x slower** than AES
- RSA is **more complex** than AES
- RSA has **10x larger keys** than AES (e.g., 2048 bits vs. 192 bits)
- **So why prefer RSA instead of AES?**

RSA > AES because...

RSA is faster than AES	0%
RSA is less complex than AES	0%
RSA requires shared secrets	0%
RSA does not require shared secrets	0%

RSA vs. AES

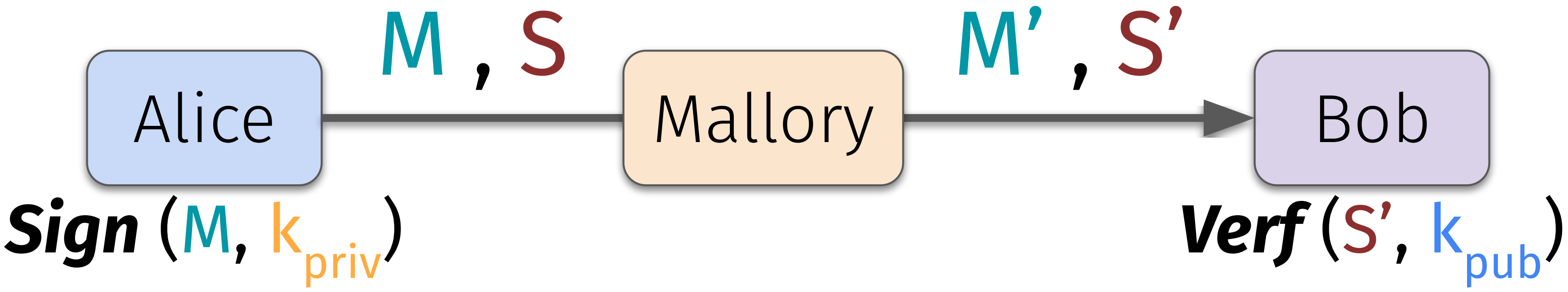
- RSA is **1000x slower** than AES
- RSA is **more complex** than AES
- RSA has **10x larger keys** than AES (e.g., 2048 bits vs. 192 bits)
- **So why prefer RSA instead of AES?** RSA requires **no shared secrets**

Questions?

Attacking RSA Digital Signatures: Bleichenbacher's Attack

Recap: Authenticity via Digital Signatures

- **Key generation:** Alice generates key pair: k_{pub} (public) and k_{priv} (private)
- Alice **signs** message **M** with k_{priv} resulting in signature **S** = ***Sign*** (**M**, k_{priv})
- **Anyone** possessing Alice's k_{pub} can **check** signature via ***Verf*** (**S'**, k_{pub})
  - If received message and signature **verified**, then message is **authentic**—from Alice!



Recap: Authenticity via Digital Signatures

- **Key generation:** Alice generates key pair: k_{pub} (public) and k_{priv} (private)

Unforgeability: computationally **infeasible** for **Mallory** to guess **S** or Alice's k_{priv}

  - If received message and signature verified, then message is **authentic**—from Alice!

… even if **Mallory knows** Alice's k_{pub} or other signatures from other messages!

Sign (M, k_{priv})

Recap: Authenticity via Digital Signatures

- **Key generation:** Alice generates key pair: k_{pub} (public) and k_{priv} (private)

RSA's Verification: $(S')^e \bmod N == M'$

- Alice **signs** message M with k_{priv} resulting in signature S = ***Sign*** (M, k_{priv})

Mallory wants to **forge signatures** to impersonate Alice, but she ***doesn't*** have Alice's **private key**—it's private!

- **Anyone** p k_{pub})
  - If rece Alice!

Ali ob

Sign (M, k_{priv}) ***Verf*** (S', k_{pub})

Bleichenbacher's Signature Forgery Attack

- Pencil-and-paper attack by Daniel Bleichenbacher at CRYPTO 2006
- Exploits **signature verification** in **insecure** RSA implementations
    - Specifically the RSA PKCS #1 standard
- Wreaked havoc on OpenSSL, Firefox

Can we exploit signature verification?

- Bob checks if **message** == (**signature**)$^{\text{exponent}}$ modulo (**N**)
  - In this problem, we know **message** and want to find **signature**

Can we exploit signature verification?

- Bob checks if **message** == (**signature**)$^{\text{exponent}}$ modulo (**N**)
  - In this problem, we know **message** and want to find **signature**

- Recall **N** computed by multiplying two huge prime numbers
  - **Mallory has zero hope of figuring these factors out** (integer factorization problem)

Can we exploit signature verification?

- Bob checks if **message** == (**signature**)$^{\textbf{exponent}}$ modulo (**N**)
  - In this problem, we know **message** and want to find **signature**
- Recall **N** computed by multiplying two huge prime numbers
  - **Mallory has zero hope of figuring these factors out** (integer factorization problem)
- Bob checks if **message** == (**signature**)$^{\textbf{exponent}}$ modulo (**HugeUnfactorableNum**)

Can we exploit signature verification?

- Bob checks if **message** == (**signature**)$^{\text{exponent}}$ modulo (**N**)
  - In this problem, we know **message** and want to find **signature**
- Recall **N** computed by multiplying two huge prime numbers
  - **Mallory has zero hope of figuring these factors out** (integer factorization problem)
- Bob checks if **message** == (**signature**)$^{\text{exponent}}$ modulo (**HugeUnfactorableNum**)
- **Question:** What if the **exponent** is a really **small** integer?

Detour: Modulo Mania!

- What does (**A mod B**) equal if…

- **A is greater than B**
  - 10 mod 7 = **?**
  - 10 mod 8 = **?**
  - 8 mod 3 = **?**

Detour: Modulo Mania!

- What does (A **mod** B) equal if…

- **A is greater than B**
  - 10 mod 7 = **3**
  - 10 mod 8 = **2**
  - 8 mod 3 = **2**

- **A is less than B**
  - 7 mod 10 = **?**
  - 8 mod 10 = **?**
  - 3 mod 8 = **?**

Detour: Modulo Mania!

- What does (A **mod** B) equal if…

- **A is greater than B**
  - 10 mod 7 = **3**
  - 10 mod 8 = **2**
  - 8 mod 3 = **2**

- **A is less than B**
  - 7 mod 10 = **7**
  - 8 mod 10 = **8**
  - 3 mod 8 = **3**

Detour: Modulo Mania!

- What does (**A mod B**) equal if…

- **A is greater than B**
  - 10 mod 7 = **3**
  - 10 mod 8 = **2**
  - 8 mod 3 = **2**

- **A is less than B**
  - 7 mod 10 = **7**
  - 8 mod 10 = **8**
  - 3 mod 8 = **3**

Observation:
If **A** is **less** than **B**…
Then (**A** mod **B**) = **A**

Exploiting Small Exponents

- if **message** == (**signature**)$^{\text{exponent}}$ modulo (**HugeUnfactorableNumber**)
    - But, we know that (**signature**)$^{\text{exponent}}$ << modulo (**HugeUnfactorableNumber**)

Exploiting Small Exponents

- if **message** == (**signature**)$^{\text{exponent}}$ modulo (**HugeUnfactorableNumber**)
  - But, we know that (**signature**)$^{\text{exponent}}$ << modulo (**HugeUnfactorableNumber**)

- if **message** == (**signature**)$^{\text{exponent}}$ ~~modulo (**HugeUnfactorableNumber**)~~

Exploiting Small Exponents

- if **message** == (**signature**)$^{\text{exponent}}$ modulo (**HugeUnfactorableNumber**)
  - But, we know that (**signature**)$^{\text{exponent}}$ << modulo (**HugeUnfactorableNumber**)

- if **message** == (**signature**)$^{\text{exponent}}$ ~~modulo (**HugeUnfactorableNumber**)~~

- if **message** == (**signature**)$^{\text{exponent}}$ → if **message** == (**signature**)^{3}

Exploiting Small Exponents

- if **message** == (**signature**)$^{\text{exponent}}$ modulo (**HugeUnfactorableNumber**)
  - But, we know that (**signature**)$^{\text{exponent}}$ << modulo (**HugeUnfactorableNumber**)
- if **message** == (**signature**)$^{\text{exponent}}$ ~~modulo (**HugeUnfactorableNumber**)~~
- if **message** == (**signature**)$^{\text{exponent}}$ → if **message** == (**signature**)^{3}
- **Problem:** With only the **message**, how can Mallory **forge** Alice's **signature**?

Exploiting Small Exponents

- if **message** == (**signature**)$^{\text{exponent}}$ modulo (**HugeUnfactorableNumber**)
  - But, we know that (**signature**)$^{\text{exponent}}$ << modulo (**HugeUnfactorableNumber**)
- if **message** = ~~leNumber)~~
- if **message** = ture)^{3}

Taking the RSA message's ***N*th root** will reveal the **signature**!

… where *N* = our tiny **exponent**!

- **Problem:** With only the **message**, how can Mallory **forge** Alice's **signature**?

A Correct Message Construction

$$\text{00 01}\ \underbrace{\text{FF FF FF}\ \cdots\ \text{FF}}_{k/8-38\text{ bytes}}\ \text{00}\ \underbrace{\text{30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14}}_{\text{ASN.1 ``magic'' bytes denoting type of hash algorithm}}\ \underbrace{\text{XX XX XX XX}\ \cdots\ \text{XX}}_{\text{SHA-1 digest (20 bytes)}}$$

- Assume key is **2048 bits long**
- Prefix FF's must be **((2048/8)—38)** bytes
  - = **218** total FF's
- Where does **38** come from?

A Correct Message Construction

$$00\ 01\ \underbrace{\mathrm{FF\ FF\ FF}\ \cdots\ \mathrm{FF}}_{k/8-38\text{ bytes}}\ 00\ \underbrace{30\ 21\ 30\ 09\ 06\ 05\ 2\mathrm{B}\ 0\mathrm{E}\ 03\ 02\ 1\mathrm{A}\ 05\ 00\ 04\ 14}_{\text{ASN.1 ``magic'' bytes denoting type of hash algorithm}}\ \underbrace{\mathrm{XX\ XX\ XX\ XX}\ \cdots\ \mathrm{XX}}_{\text{SHA-1 digest (20 bytes)}}$$

`SHA1("Go Chiefs!")`

- Assume key is 2048 **bits long**
- Prefix FF's must be **((2048/8)—38)** bytes
  - = **218** total FF's
- Where does 38 come from?
  - **20-byte** SHA-1 digest
  - **15-byte** ASN.1 hash specifier
  - **3 more bytes** (`00`, `01`, `00`)

If number of FF's don't match **218**, reject message!

Can we take its *N*th root?

- ***N*th-rooting** the **correct** message construction likely **won’t** work—**why?**

$$\text{00 01}\ \underbrace{\text{FF FF FF} \ \cdots\ \text{FF}}_{k/8-38\text{ bytes}}\ \text{00}\ \underbrace{\text{30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14}}_{\text{ASN.1 “magic” bytes denoting type of hash algorithm}}\ \underbrace{\text{XX XX XX XX}\ \cdots\ \text{XX}}_{\text{SHA-1 digest (20 bytes)}}$$

Can we take its *N*th root?

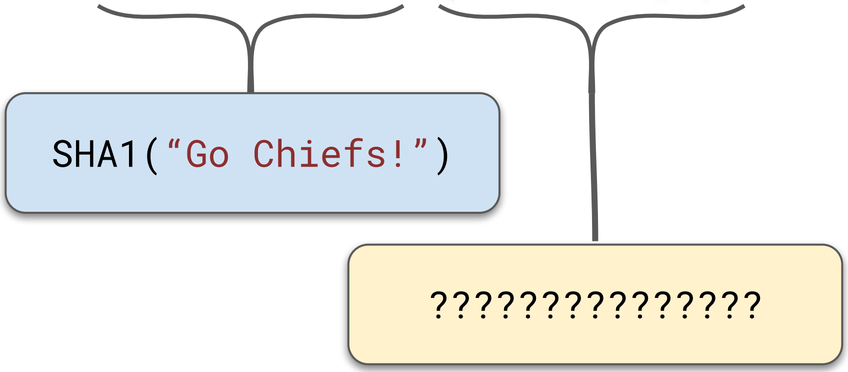
- ***N*th-rooting** the **correct** message construction likely **won't** work—**why?**

$$\text{00 01}\ \underbrace{\text{FF FF FF}\ \cdots\ \text{FF}}_{k/8-38\text{ bytes}}\ \text{00}\ \underbrace{\text{30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14}}_{\text{ASN.1 ``magic'' bytes denoting type of hash algorithm}}\ \underbrace{\text{XX XX XX XX}\ \cdots\ \text{XX}}_{\text{SHA-1 digest (20 bytes)}}$$

  - It is highly unlikely that you get a **perfect root!**
  - Your signature has to be an integer—no decimal remainder!
    - Thus, **message** **will not equal** $(\textbf{signature})^{\textbf{exponent}}$
    - **Attack fails!**

An *Insecure* Message Construction

$$00\ 01\ \mathbf{FF}\ 00\ \underbrace{30\ 21\ 30\ 09\ 06\ 05\ 2B\ 0E\ 03\ 02\ 1A\ 05\ 00\ 04\ 14}_{\text{ASN.1 ``magic'' bytes denoting type of hash algorithm}}\ \underbrace{XX\ XX\ XX\ \cdots\ XX}_{\text{SHA-1 digest (20 bytes)}}\ \underbrace{YY\ YY\ YY\ YY\ \cdots\ YY}_{k/8-39\text{ arbitrary bytes}}$$



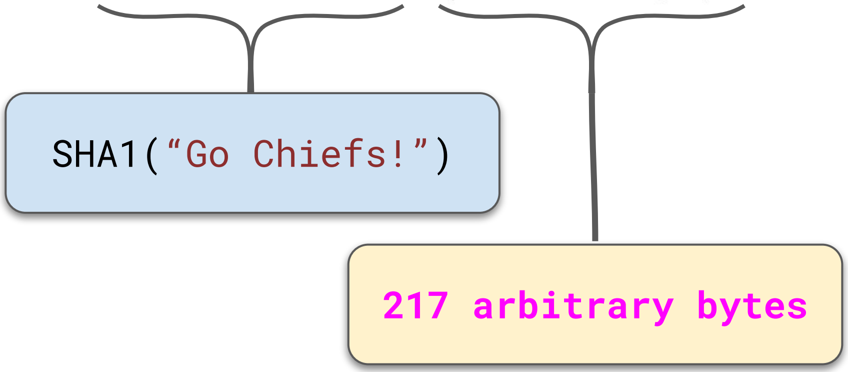
`SHA1("Go Chiefs!")`

`???????????????`

- Assume key is **2048 bits long**
- What if server doesn't count FF's?
  - We could **use just *one*** FF
  - And **???** at the end

An *Insecure* Message Construction

$$00\,01\,\mathbf{FF}\,00\;\underbrace{30\,21\,30\,09\,06\,05\,2B\,0E\,03\,02\,1A\,05\,00\,04\,14}_{\text{ASN.1 “magic” bytes denoting type of hash algorithm}}\;\underbrace{XX\,XX\,XX\;\cdots\;XX}_{\text{SHA-1 digest (20 bytes)}}\;\underbrace{YY\,YY\,YY\,YY\;\cdots\;YY}_{k/8-39\text{ arbitrary bytes}}$$



`SHA1("Go Chiefs!")`

`217 arbitrary bytes`

- Assume key is **2048 bits long**
- What if server doesn't count FF's?
  - We could **use just *one*** FF
  - And **217 arbitrary bytes** at the end
    - These end up **not being checked!**

Can we take its *N*th root?

- How about ***N*th-rooting** the **insecure** message construction?

$$\texttt{00 01 }\mathbf{FF}\texttt{ 00 }\underbrace{\texttt{30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14}}_{\text{ASN.1 ``magic'' bytes denoting type of hash algorithm}}\ \underbrace{\texttt{XX XX XX} \cdots \texttt{XX}}_{\text{SHA-1 digest (20 bytes)}}\ \underbrace{\texttt{YY YY YY YY} \cdots \texttt{YY}}_{k/8-39 \text{ arbitrary bytes}}$$

Can we take its *N*th root?

- How about ***N*th-rooting** the **insecure** message construction?

$$\texttt{00 01 }\mathbf{FF}\texttt{ 00 }\underbrace{\texttt{30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14}}_{\text{ASN.1 “magic” bytes denoting type of hash algorithm}}\ \underbrace{\texttt{XX XX XX}\ \cdots\ \texttt{XX}}_{\text{SHA-1 digest (20 bytes)}}\ \underbrace{\mathbf{YY\ YY\ YY\ YY}\ \cdots\ \mathbf{YY}}_{k/8-39\text{ arbitrary bytes}}$$

  - It is highly unlikely that you get a **perfect root!**
  - Your signature has to be an integer—no decimal remainder!
    - Thus, **message** **will not equal** (**signature**)$^{\textbf{exponent}}$
    - **Attack fails!**

Can we take its *N*th root?

- How about ***N*th-rooting** the **insecure** message construction?

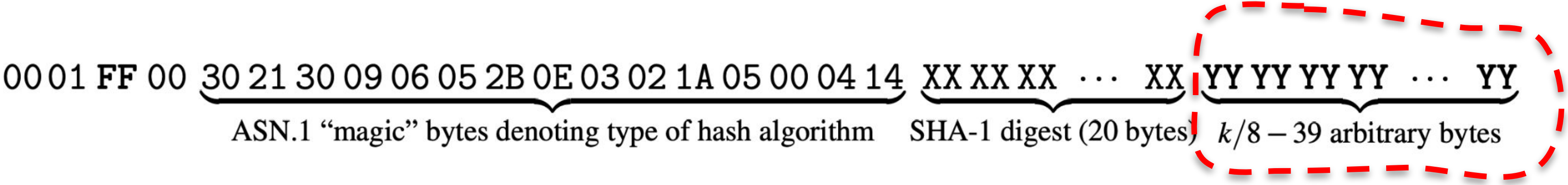
$$\texttt{00 01 FF 00}\ \underbrace{\texttt{30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14}}_{\text{ASN.1 “magic” bytes denoting type of hash algorithm}}\ \underbrace{\texttt{XX XX XX}\ \cdots\ \texttt{XX}}_{\text{SHA-1 digest (20 bytes)}}\ \underbrace{\texttt{YY YY YY YY}\ \cdots\ \texttt{YY}}_{k/8-39\text{ arbitrary bytes}}$$

  - It is highly unlikely that you get a **perfect root!**
  - Your signature has to be an integer—no decimal remainder!
    - Thus, **message will not equal** (**signature**)$^{\text{exponent}}$
    - **Attack fails!**

  - But… we know that the **last 217 bytes of the message aren’t checked** by the server!

Can we take its *N*th root?

- How about ***N*th-rooting** the **insecure** message construction?



$$00\,01\,\mathbf{FF}\,00\,\underbrace{30\,21\,30\,09\,06\,05\,2B\,0E\,03\,02\,1A\,05\,00\,04\,14}_{\text{ASN.1 ``magic'' bytes denoting type of hash algorithm}}\,\underbrace{XX\,XX\,XX\,\cdots\,XX}_{\text{SHA-1 digest (20 bytes)}}\,\underbrace{YY\,YY\,YY\,YY\,\cdots\,YY}_{k/8-39\text{ arbitrary bytes}}$$

  - It is highly unlikely that you get a **perfect root!**
  - Your signature has to be an integer—no decimal remainder!
    - Thus, **message will not equal** (**signature**)$^{\text{exponent}}$
    - **Attack fails!**

  - But… we know that the **last 217 bytes of the message aren't checked** by the server!
    - Thus, **we can "tweak" our signature** such that **message == (signature)**$^{\text{exponent}}$

Can we take its *N*th root?

- How about ***N*th-rooting** the **insecure** message construction?

$$\texttt{00 01 FF 00}\ \underbrace{\texttt{30 21 30 09 06 05 2B 0E 03 02 1A 05 00 04 14}}_{\text{ASN.1 “magic” bytes denoting type of hash algorithm}}\ \underbrace{\texttt{XX XX XX} \cdots \texttt{XX}}_{\text{SHA-1 digest (20 bytes)}}\ \underbrace{\texttt{YY YY YY YY} \cdots \texttt{YY}}_{k/8-39 \text{ arbitrary bytes}}$$

  - It is highly unlikely that you get a **perfect root!**
  - Your signature has to be an integer—no decimal remainder!
    - Thus, **message will not equal** (**signature**)$^{\text{exponent}}$
    - **Attack fails!**

  - But… we know that the **last 217 bytes of the message aren’t checked** by the server!
    - Thus, **we can “tweak” our signature** such that **message ==** (**signature**)$^{\text{exponent}}$
    - When server computes (**signature**)$^{\text{exponent}}$, will get slightly different **message—that’s ok!**

Exploiting Weak Padding Checking

- Write the number `300` in binary:

`1 0 0 1 0 1 1 0 0`

Exploiting Weak Padding Checking

- Write the number 300 in binary:

`1 0 0 1 0 1 1 0 0`

- Take its cube root:

`300^(⅓) = 6.6943` **(not an integer!)**

Exploiting Weak Padding Checking

- Write the number 300 in binary:

`1 0 0 1 0 1 1 0 0`

- Take its cube root:

`300^(⅓) = 6.6943` **(not an integer!)**

- Round up to the **nearest** integer, **cube that**, and write in binary form:

`7^(3) = 1 0 1 0 1 0 1 1 1`

Exploiting Weak Padding Checking

- Compare 300 and 343 side-by-side:

```
1 0 0 1 0 1 1 0 0
1 0 1 0 1 0 1 1 1
```

(**bytes 3–9 don’t match!**)

Exploiting Weak Padding Checking

- Compare 300 and 343 side-by-side:

```
1 0 0 1 0 1 1 0 0      (bytes 3–9 don't match!)
1 0 1 0 1 0 1 1 1
```

- Pretend that **everything after the first two bytes is ignored** by the server

```
1 0 0 1 0 1 1 0 0      (only care about bytes 1–2)
1 0 1 0 1 0 1 1 1
```

Exploiting Weak Padding Checking

- Compare 300 and 343 side-by-side:

```
1 0 0 1 0 1 1 0 0
1 0 1 0 1 0 1 1 1
```

(**bytes 3–9 don't match!**)

- Pretend that **everything after the first two bytes is ignored** by the server

```
1 0 0 1 0 1 1 0 0
1 0 1 0 1 0 1 1 1
```

(**only care about bytes 1–2**)

- **Success! Check passes**

Exploiting Weak Padding Checking

- Compare 300 and 343 side-by-side:

`1 0 0 1 0 1 1 0 0` (**bytes 3–9 don't match!**)

- Pretend t he server

ytes 1–2)

- **Success! Check passes**

Small exponent + **insecure padding**
enables Mallory to **forge signatures**…
without knowing Alice's **private key**!

Questions?

Key Management Rules

Rule #1

- Each key should have **only one purpose**
  - Different RSA keys for signing and encrypting
  - Different symmetric keys for encrypting and MACing
  - Different keys for **Alice** → **Bob** and **Bob** → **Alice**
  - Different keys for different protocols

- **Reason:** prevent attacker from "**repurposing**" content
  - Example: reflection attack

Rule #2

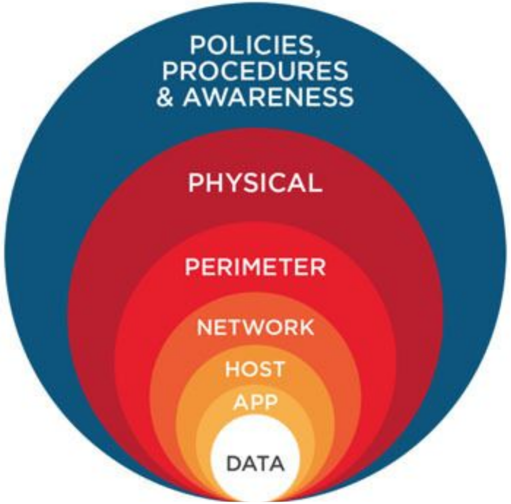
- **Vulnerability** of a key increases with time and use

- Change your keys **periodically**!
  - Use session keys
  - Encrypt your keys
  - **Erase keys from memory** when you're done with them
  - Don't let your keys get swapped out to disk

Rule #3

- Keep your keys **far from the attacker!**
  - Memory of networked and unguarded PC = **bad**
  - Memory of non-networked, guarded PC = **not as bad**
  - Stored in tamper-resistant device: **better**
    - Hardware Security Module (HSM)
    - See FIPS 140-2: “Requirements for Crypto Modules”

- Stored HSM in locked safe: **best**
  - Layered defenses / defense-in-depth



Rule #4

- Protect yourself against compromise of old keys
  - **Bad practice:** Alice tells Bob, “Here’s the **new** key: ...” encrypted under the **old** key
  - Adversary can record this, then **attack old key**
  - **Old key** then used to uncover **new key**

- **Worse yet:**
  - If long chain of keys, he can attack anyone—chain unravels!
  - **Chain only as strong as its weakest link!**

- **Forward secrecy:** learning old key **shouldn’t** help adversary learn new key

Next time on CS 4440…

Security in Practice: Cryptocurrency