CS 114: Network Security

Lecture 5 - Public Key Cryptography

Prof. Daniel Votipka

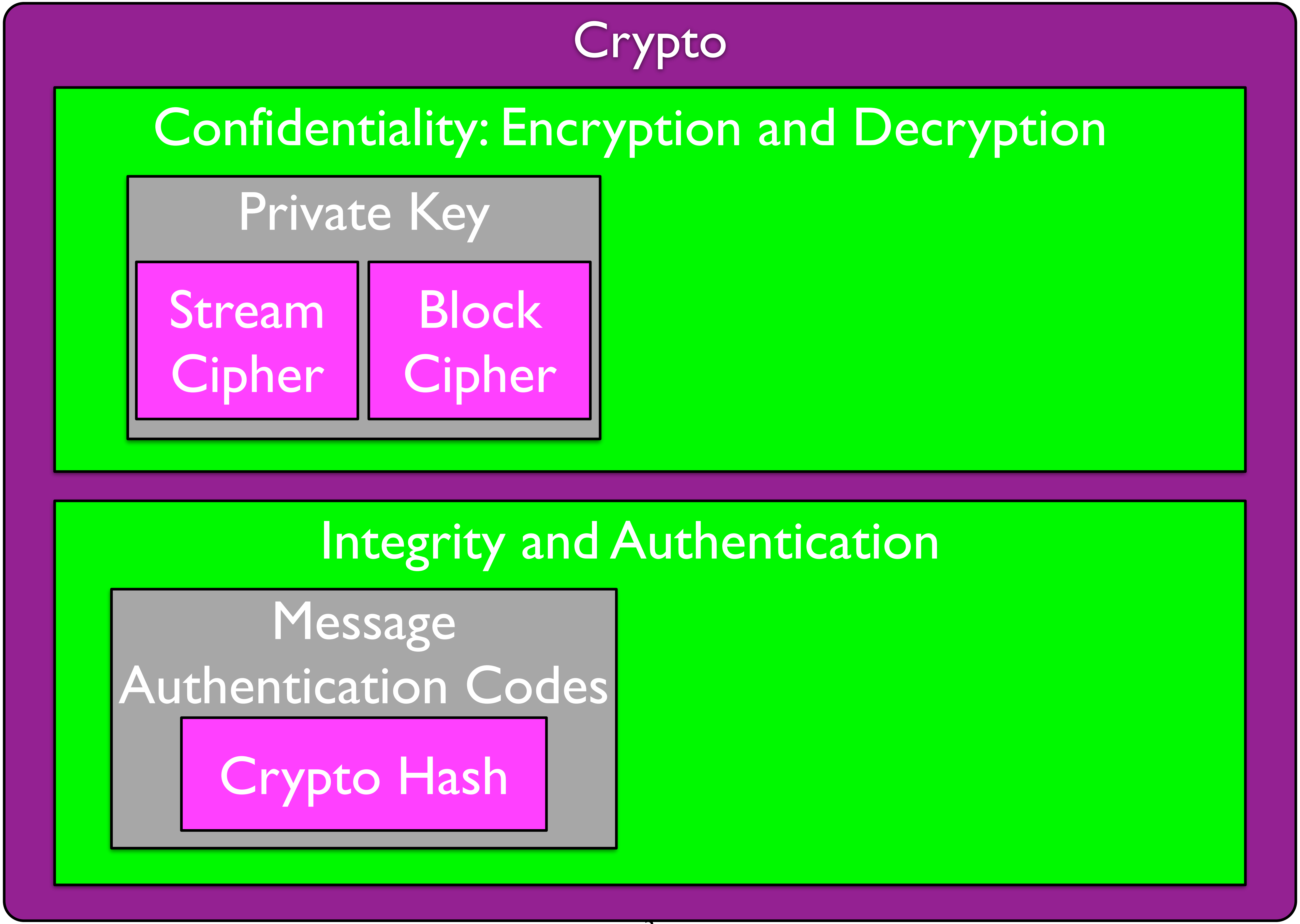
Spring 2023

(some slides courtesy of Prof. Micah Sherr and Prof. Patrick McDaniel)

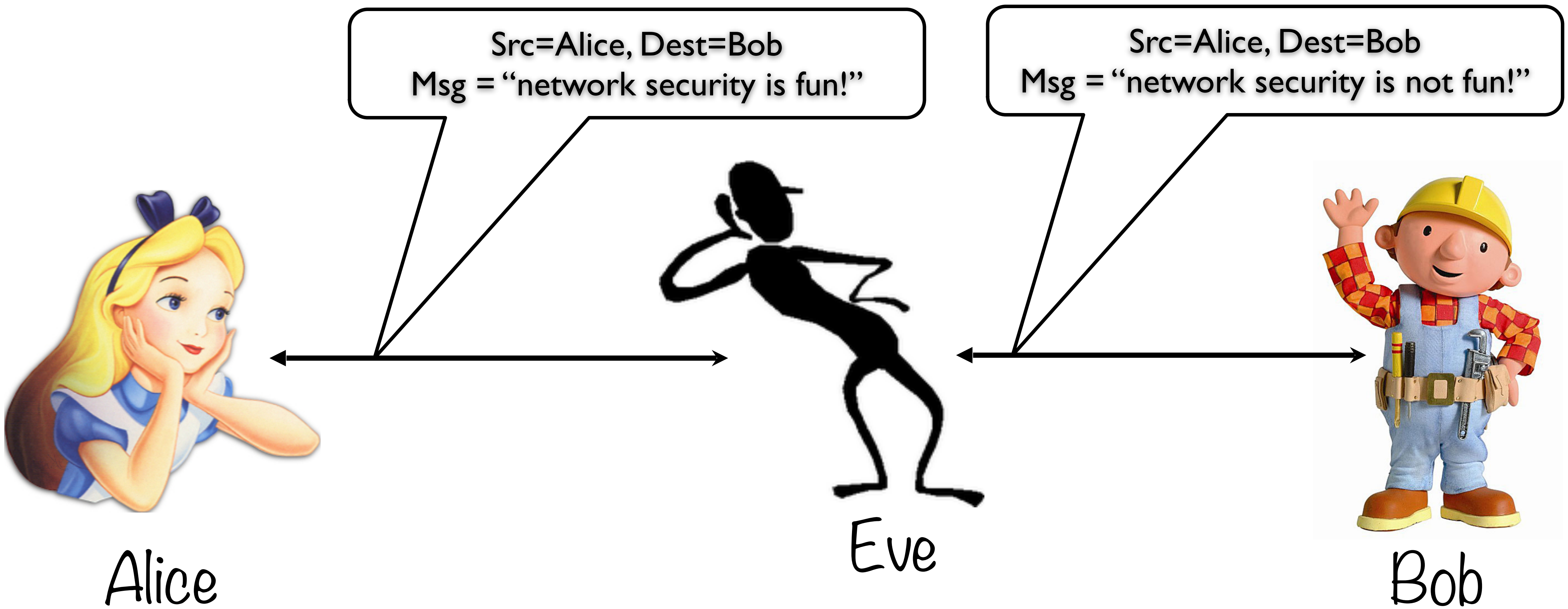
Tufts UNIVERSITY

Administrivia

- Homework 1, part 1 due **Tonight** at 11:59pm
- Homework 1, part 2 due Feb. 28th at 11:59pm


Crypto
Confidentiality: Encryption and Decryption
Private Key
Stream Cipher
Block Cipher
Integrity and Authentication
Message Authentication Codes
Crypto Hash

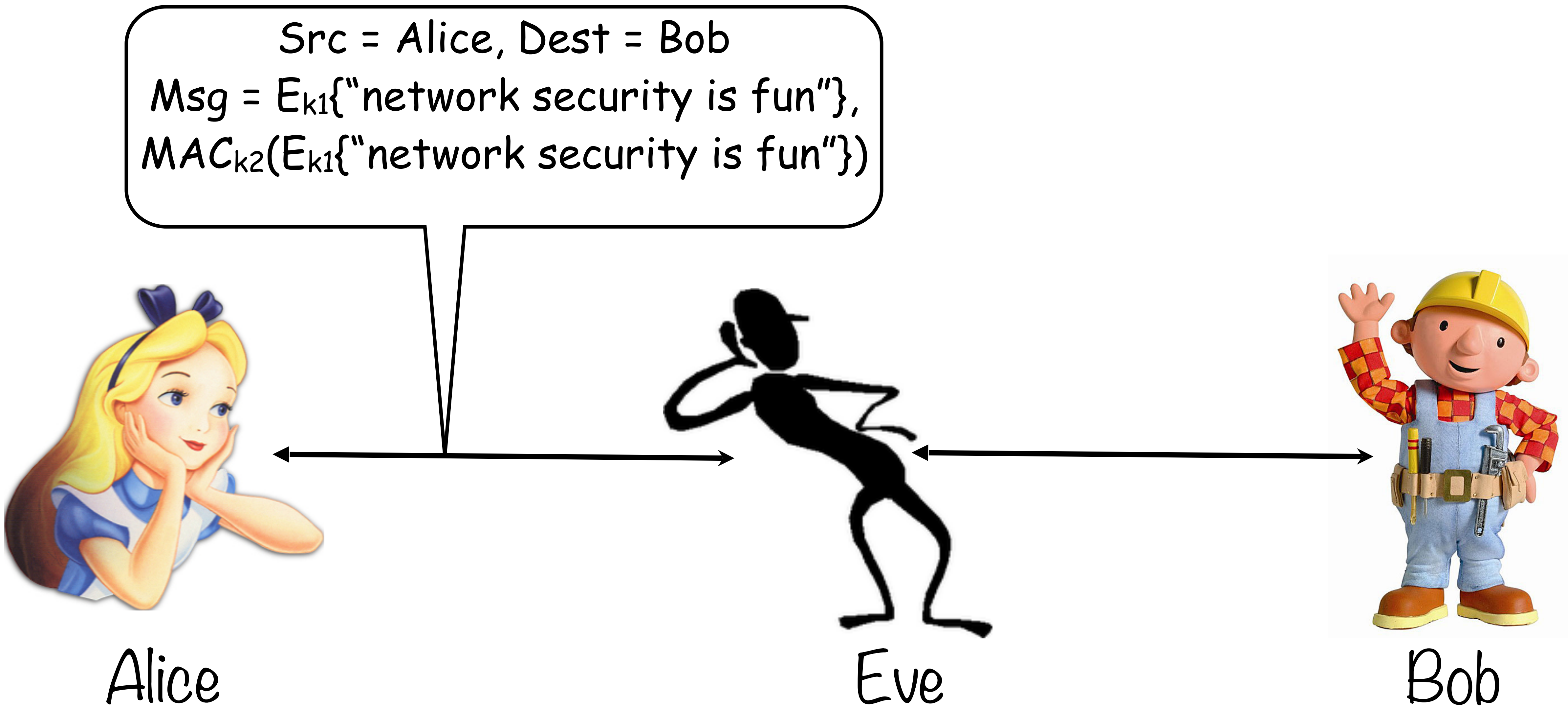
Man-in-the-Middle (MitM) attack



Cryptographic Hash Functions

- Properties of good <u>cryptographic</u> hash functions:
  - **preimage resistance**
  - **2nd-preimage resistance**
  - **collision resistance**

Encryption and Message Authenticity



Without knowing *k1*,
Eve can’t read Alice’s message.

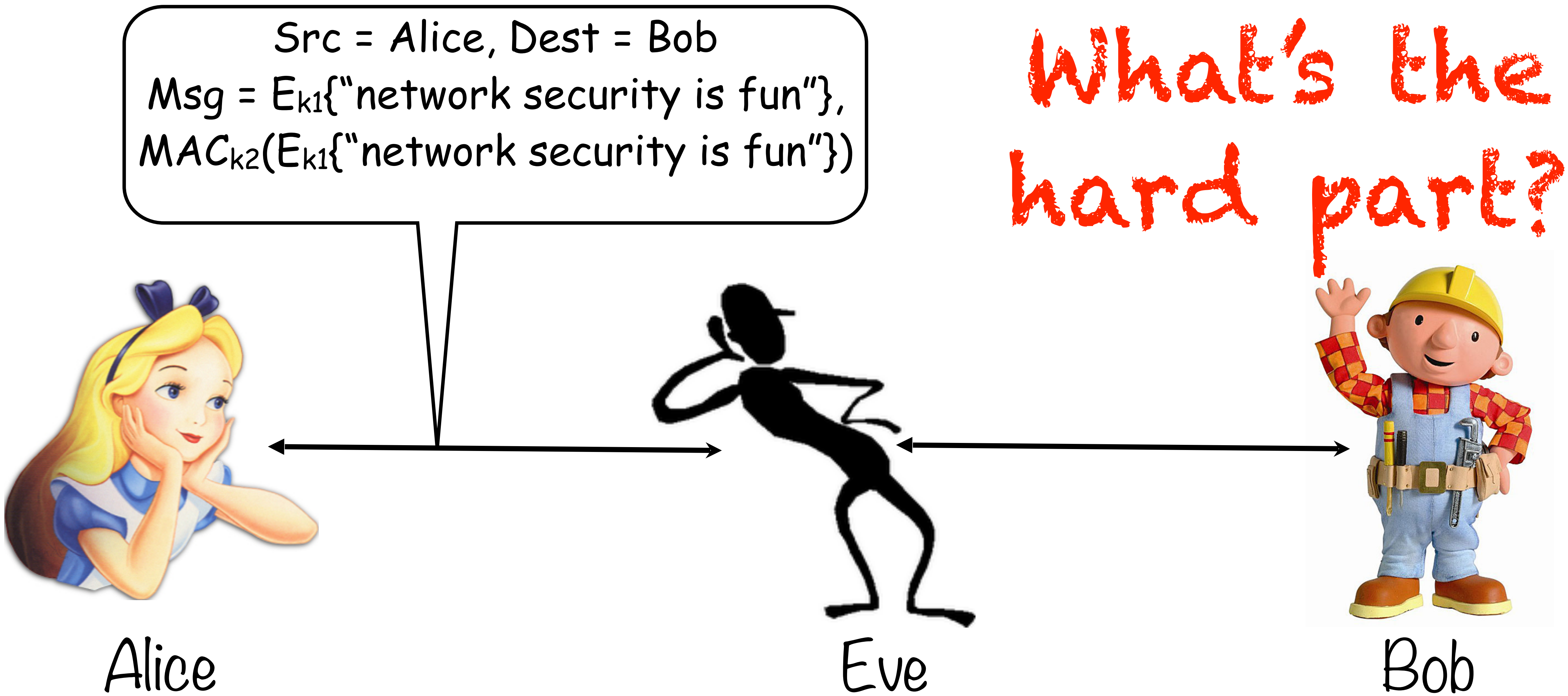
Without knowing *k2*, Eve can’t compute a valid MAC for her forged message!

Message Authentication Codes (MACs)

- MACs provide message **integrity** and **authenticity**
- $MAC_K(M)$ – use symmetric encryption to produce short sequence of bits that depends on both the message (M) and the key (K)
- MACs should be resistant to **existential forgery**: Eve should not be able to produce a valid MAC for a message M' without knowing K
- To provide confidentiality, authenticity, and integrity of a message, Alice sends
  - MAC-then-Encrypt: $E_K(M, MAC_K(M))$ where $E_K(X)$ is the encryption of X using key K; or
  - Encrypt-then-MAC: $E_K(M), MAC_K(E_K(M))$ ← Best option

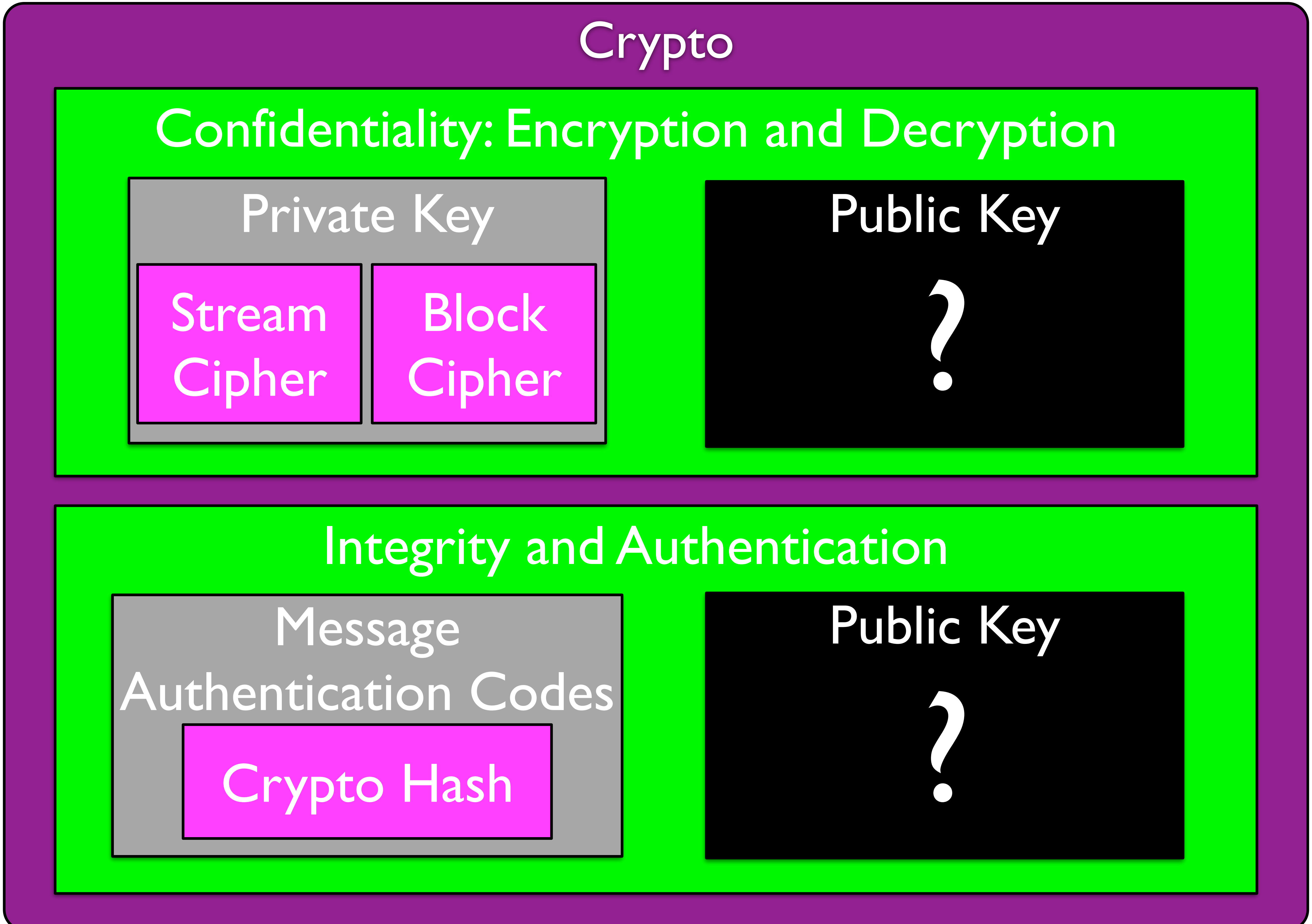
  or
  - Encrypt-and-MAC: $E_K(M), MAC_K(M)$
  - Proves that M was encrypted (confidentiality) by someone who knew K (authenticity) and hasn't been changed (integrity)

Encryption and Message Authenticity



Without knowing *k1*, Eve can't read Alice's message.

Without knowing *k2*, Eve can't compute a valid MAC for her forged message!


Crypto
Confidentiality: Encryption and Decryption
Private Key
Stream Cipher
Block Cipher
Public Key
?
Integrity and Authentication
Message Authentication Codes
Crypto Hash
Public Key
?

Private Key Crypto

Public Key Crypto
(10,000 ft view)

- Separate keys for encryption and decryption
  - Public key: anyone can know this
  - Private key: kept confidential
- Anyone can encrypt a message to you using your public key
- The private key (kept confidential) is required to decrypt the communication
- Alice and Bob no longer have to have *a priori* shared a secret key

Public Key Cryptography

- Each key pair consists of a public and private component: k^+ (public key), k^- (private key)

$$D_{k^-}(E_{k^+}(m)) = m$$

- Public keys are distributed (typically) through public key certificates
  - Anyone can communicate secretly with you ***if they have your certificate***

Public Key Cryptography

E_{B+}(“cs114 is cool” D_{B-}( ))

Alice

(A^+, A^-)

Bob

(B^+, B^-)

RSA
(Rivest, Shamir, Adelman)

- The dominant public key algorithm
  - The algorithm itself is conceptually simple
  - Why it is secure is very deep (number theory)
  - Uses properties of exponentiation modulo a product of large primes

"A method for obtaining Digital Signatures and Public Key Cryptosystems", Communications of the ACM, Feb. 1978.

RSA Key Generation

- Choose distinct primes p and q
- Compute $n = pq$
- Compute $\Phi(n) = \Phi(pq) = (p-1)(q-1)$
- Randomly choose $1<e< \Phi(pq)$ such that e and $\Phi(pq)$ are coprime. e is the **public key exponent**
- Compute $d=e^{-1} \bmod(\Phi(pq))$. d is the **private key exponent**

Why does this work?

Euler's Totient Function

- **coprime**: having no common positive factors other than 1 (also called **relatively prime**)
  - 16 and 25 are coprime
  - 6 and 27 are not coprime
- **Euler's Totient Function**: Φ(n) = number of integers less than or equal to n that are coprime with n

$$\Phi(n) = n \cdot \prod_{p|n} (1 - \frac{1}{p})$$

where product ranges over distinct primes dividing n

Euler's Totient Function

$$\Phi(n) = n \cdot \prod_{p|n} (1 - \frac{1}{p})$$

$\Phi(18) = 18(1\text{-}1/3)(1\text{-}1/2) = 6$

1,5,7,11,13,17

Euler's Totient Function

- **coprime**: having no common positive factors other than 1 (also called **relatively prime**)
  - 16 and 25 are coprime
  - 6 and 27 are not coprime
- **Euler's Totient Function**: Φ(n) = number of integers less than or equal to n that are coprime with n

$$\Phi(n) = n \cdot \prod_{p|n} (1 - \frac{1}{p})$$

where product ranges over distinct primes dividing n

- If m and n are coprime, then Φ(mn) = Φ(m)Φ(n)
- If m is prime, then Φ(m) = m - 1

RSA Key Generation

- Choose distinct primes p and q
- Compute $n = pq$
- Compute $\Phi(n) = \Phi(pq)$
$= (p-1)(q-1)$
- Randomly choose $1<e< \Phi(pq)$ such that e and $\Phi(pq)$ are coprime. e is the **public key exponent**
- Compute $d=e^{-1} \bmod(\Phi(pq))$. d is the **private key exponent**

Example:

let $p=3, q=11$

$n=33$

$\Phi(pq)=(3-1)(11-1)=20$

let $e=7$

Modular Arithmetic

- Integers $Z_n = \{0, 1, 2, ..., n-1\}$
- x mod n = remainder of x divided by n
  - 5 mod 13 = 5
  - 13 mod 5 = 3
- y is **modular inverse** of x iff xy mod n = 1
  - 4 is inverse of 3 in Z_{11}
- Z_n has modular inverses for all integers n is co-prime with except 0

RSA Key Generation

- Choose distinct primes p and q
- Compute n = pq
- Compute $\Phi(n) = \Phi(pq)$
  $= (p-1)(q-1)$
- Randomly choose $1<e< \Phi(pq)$ such that e and $\Phi(pq)$ are coprime. e is the **public key exponent**
- Compute $d=e^{-1} \bmod(\Phi(pq))$. d is the **private key exponent**

Example:

let p=3, q=11

n=33

$\Phi(pq)=(3-1)(11-1)=20$

let e=7

$ed \bmod \Phi(pq) = 1$

$7d \bmod 20 = 1$

d = 3

RSA Encryption/ Decryption

- Public key k^+ is {e,n} and private key k^- is {d,n}
- Encryption and Decryption

$$E_{k+}(M) : \text{ciphertext} = \text{plaintext}^e \bmod n$$

$$D_{k-}(\text{ciphertext}) : \text{plaintext} = \text{ciphertext}^d \bmod n$$

- Example
  - Public key (7,33), Private Key (3,33)
  - Plaintext: 4
  - $E_{\{7,33\}}(4) = 4^7 \bmod 33 = 16384 \bmod 33 = 16$
  - $D_{\{3,33\}}(16) = 16^3 \bmod 33 = 4096 \bmod 33 = 4$

Why does it work?

- Difficult to find $\Phi(n)$ or d using only e and n.
- Finding d is equivalent in difficulty to factoring n as p*q
  - No efficient integer factorization algorithm is known
  - Example: Took 18 months to factor a 200 digit number into its 2 prime factors
- It is feasible to encrypt and decrypt because:
  - It is possible to find large primes.
  - It is possible to find coprimes and their inverses.
  - Modular exponentiation is feasible.

Modular exponentiation is easy!

$4^{10} \bmod 497$

- $e' = 1$. $c = (1 \cdot 4) \bmod 497 = 4 \bmod 497 = \mathbf{4}$.
- $e' = 2$. $c = (4 \cdot 4) \bmod 497 = 16 \bmod 497 = \mathbf{16}$.
- $e' = 3$. $c = (16 \cdot 4) \bmod 497 = 64 \bmod 497 = \mathbf{64}$.
- $e' = 4$. $c = (64 \cdot 4) \bmod 497 = 256 \bmod 497 = \mathbf{256}$.
- $e' = 5$. $c = (256 \cdot 4) \bmod 497 = 1024 \bmod 497 = \mathbf{30}$.
- $e' = 6$. $c = (30 \cdot 4) \bmod 497 = 120 \bmod 497 = \mathbf{120}$.
- $e' = 7$. $c = (120 \cdot 4) \bmod 497 = 480 \bmod 497 = \mathbf{480}$.
- $e' = 8$. $c = (480 \cdot 4) \bmod 497 = 1920 \bmod 497 = \mathbf{429}$.
- $e' = 9$. $c = (429 \cdot 4) \bmod 497 = 1716 \bmod 497 = \mathbf{225}$.
- $e' = 10$. $c = (225 \cdot 4) \bmod 497 = 900 \bmod 497 = \mathbf{403}$.

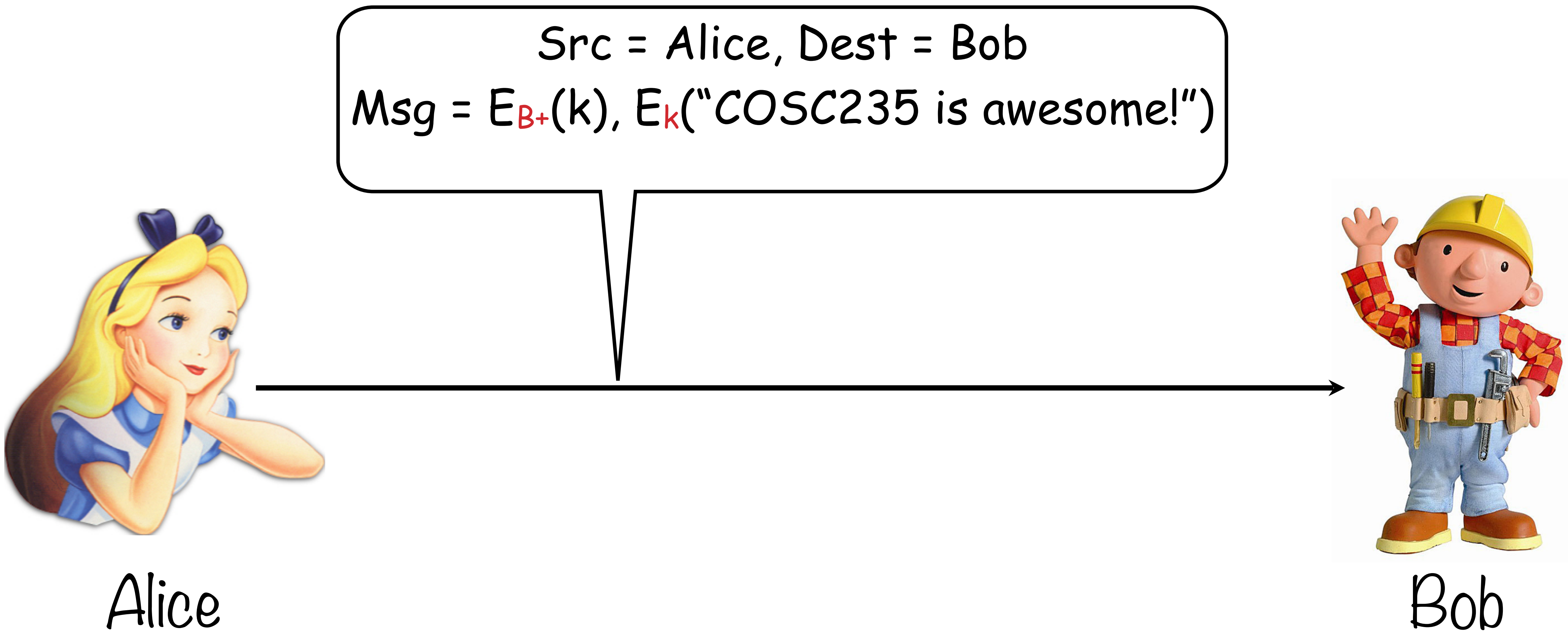
Why do we care about private key crypto?

- Most public key systems use at least 1,024-bit keys
  - Key size not comparable to symmetric key algorithms
- RSA is much slower than most symmetric crypto algorithms
  - AES: ~161 MB/s
  - RSA: ~82 KB/s
  - This is **too** slow to use for modern network communication!
  - Solution: Use hybrid encryption

Hybrid Cryptosystems

- In practice, public-key cryptography is used to secure and distribute session keys.
- These keys are used with symmetric algorithms for communication.
- Sender generates a random session key, encrypts it using receiver's public key and sends it.
- Receiver decrypts the message to recover the session key.
- Both encrypt/decrypt their communications using the same key.
- Key is destroyed in the end.

Hybrid Cryptosystems



(B^+,B^-) is Bob's long-term public-private key pair.

k is the session key; sometimes called the **ephemeral key**.

Crypto

Confidentiality: Encryption and Decryption

Private Key

- Stream Cipher
- Block Cipher

Public Key

- RSA!

Integrity and Authentication

Message Authentication Codes

- Crypto Hash

Public Key

?

Digital Signatures

Digital Signatures

- A digital signature serves the same purpose as a real signature.
  - It is a mark that only sender can make
  - Other people can easily recognize it as belonging to the sender
- Digital signatures must be:
  - **Unforgeable**: If Alice signs message M with signature S, it is impossible for someone else to produce the pair (M, S).
  - **Authentic**: If Bob receives the pair (M, S) and knows Alice's public key, he can check ("verify") that the signature is really from Alice

Encryption using private key

$E_{k-}(M)$: ciphertext = plaintext^{d} mod n

D_{k+}(ciphertext) : plaintext = ciphertext^{e} mod n

How can Alice *sign* a digital document?

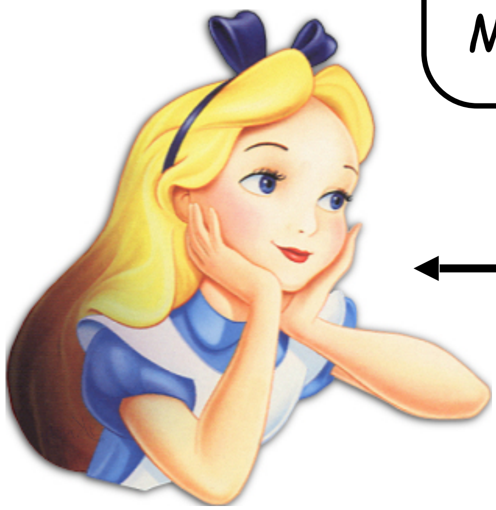
- Digital document: M
- Since RSA is slow, hash M to compute digest: $m = h(M)$
- Signature: $Sig(M) = E_{k-}(m) = m^d \bmod n$
  - Since only Alice knows k^-, only she can create the signature
- To verify: Verify(M,Sig(M))
  - Bob computes h(m) and compares it with $D_{k+}(Sig(M))$
  - Bob can compute $D_{k+}(Sig(M))$ since he knows k^+ (Alice's public key)
  - If and only if they match, the signature is verified (otherwise, verification fails)

Properties of a Digital Signature

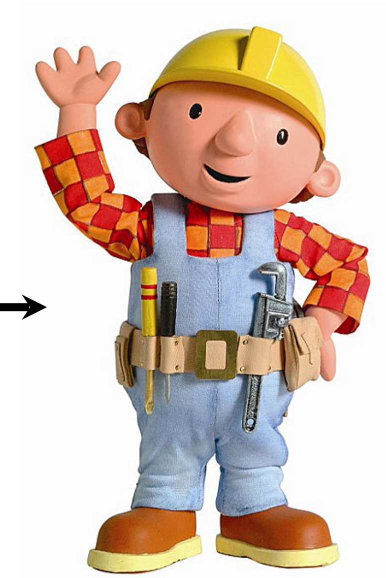
- **No forgery possible:** No one can forge a message that is purportedly from Alice. If you get a signed message you should be able to verify that it's really from Alice
- **No alteration/Integrity:** No party can undetectably alter a signed message
- Provides authentication, integrity, and **non-repudiation** (cannot deny having signed a signed message)

Non-Repudiation

Src = Alice, Dest = Bob
Msg = {“network security is fun!”, MAC_k(“network security is fun!”)}

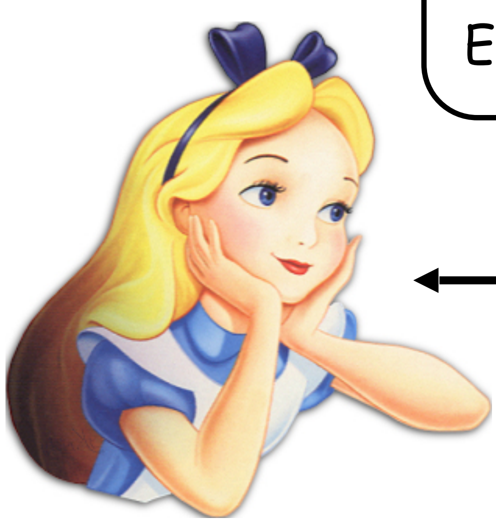


Alice

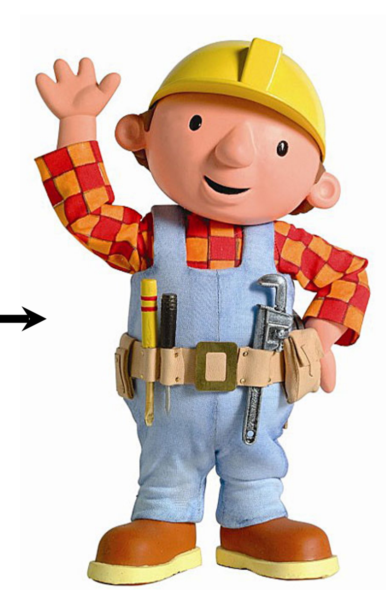


Bob

Src = Alice, Dest = Bob
Msg = {“network security is fun!”, E_{A-}(h(“network security is fun!”))}



Alice



Bob

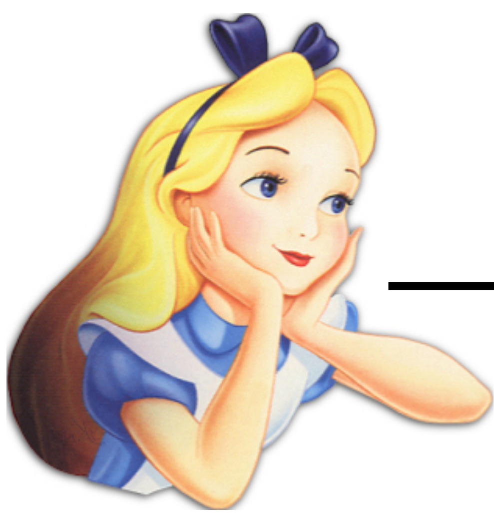
Which of these offer non-repudiation?

Putting it all together

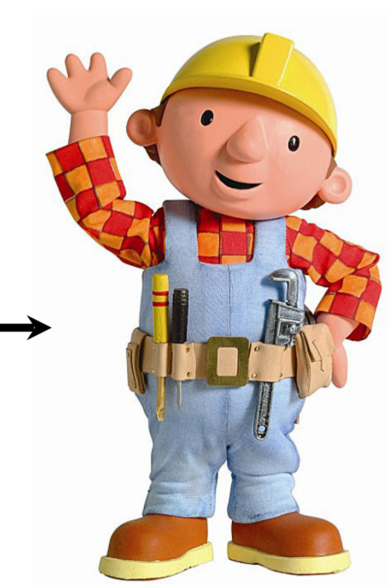
Define m = "cs114 is awesome"

Src = Alice, Dest = Bob

Msg = $E_{B+}(k)$, $E_k($ m, $E_{A-}(h(m))$)



Alice



Bob

(A^+,A^-) is Alice's long-term public-private key pair.

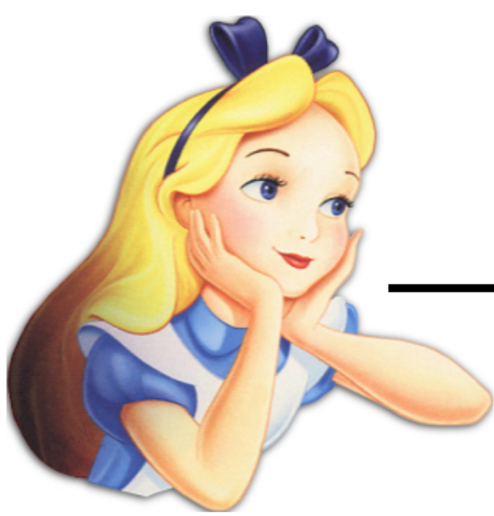
(B^+,B^-) is Bob's long-term public-private key pair.

k is the session key; sometimes called the **ephemeral key**.

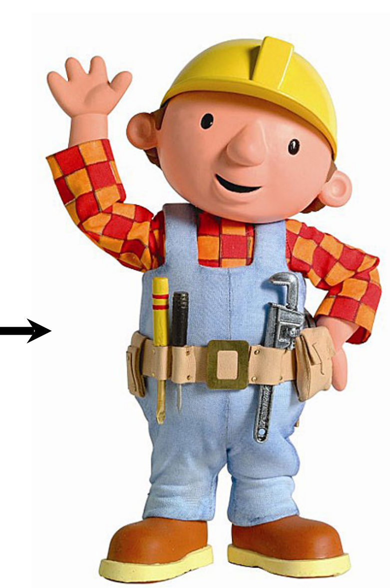
Putting it all together

Define m = "cs114 is awesome"

Src = Alice, Dest = Bob
Msg = $\mathbf{E_{B+}(k)}$, $E_k($ m, $E_{A-}(h(m))$ $)$



Alice

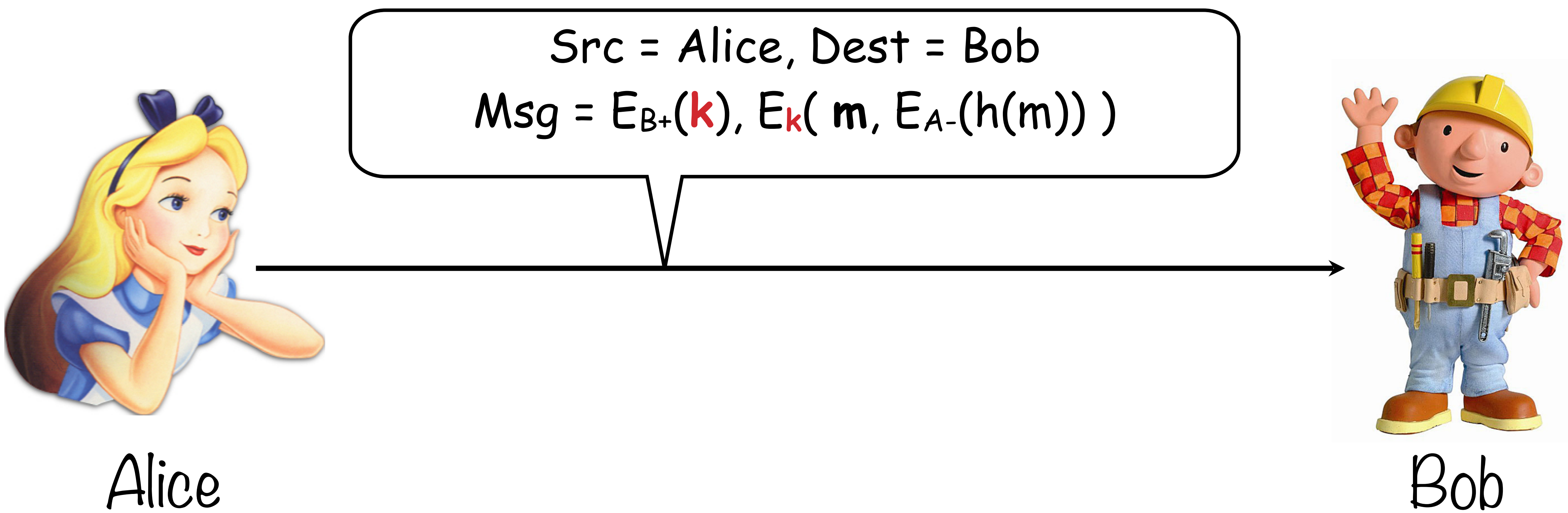


Bob

(A^+,A^-) is Alice's long-term public-private key pair.
(B^+,B^-) is Bob's long-term public-private key pair.
k is the session key; sometimes called the **ephemeral key**.

Putting it all together

Define m = “cs114 is awesome”



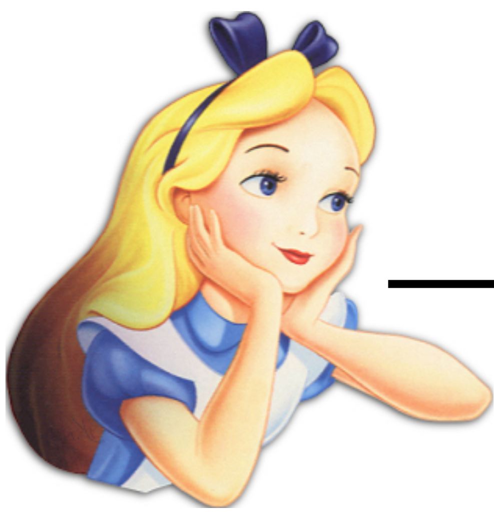
(A^+, A^-) is Alice’s long-term public-private key pair.
(B^+, B^-) is Bob’s long-term public-private key pair.
k is the session key; sometimes called the **ephemeral key**.

Putting it all together

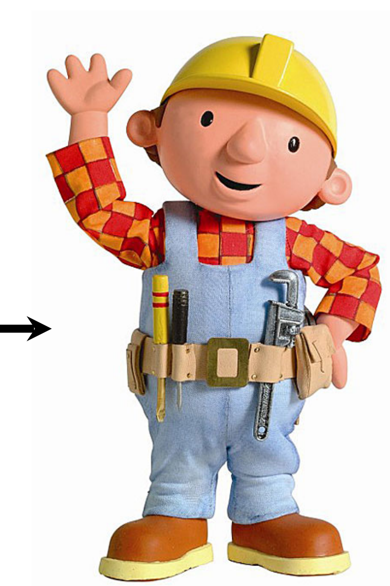
Define m = “cs2114 is awesome”

Src = Alice, Dest = Bob

Msg = $E_{B+}(k)$, $E_k($ m, $\mathbf{E_{A-}(h(m))}$ $)$



Alice



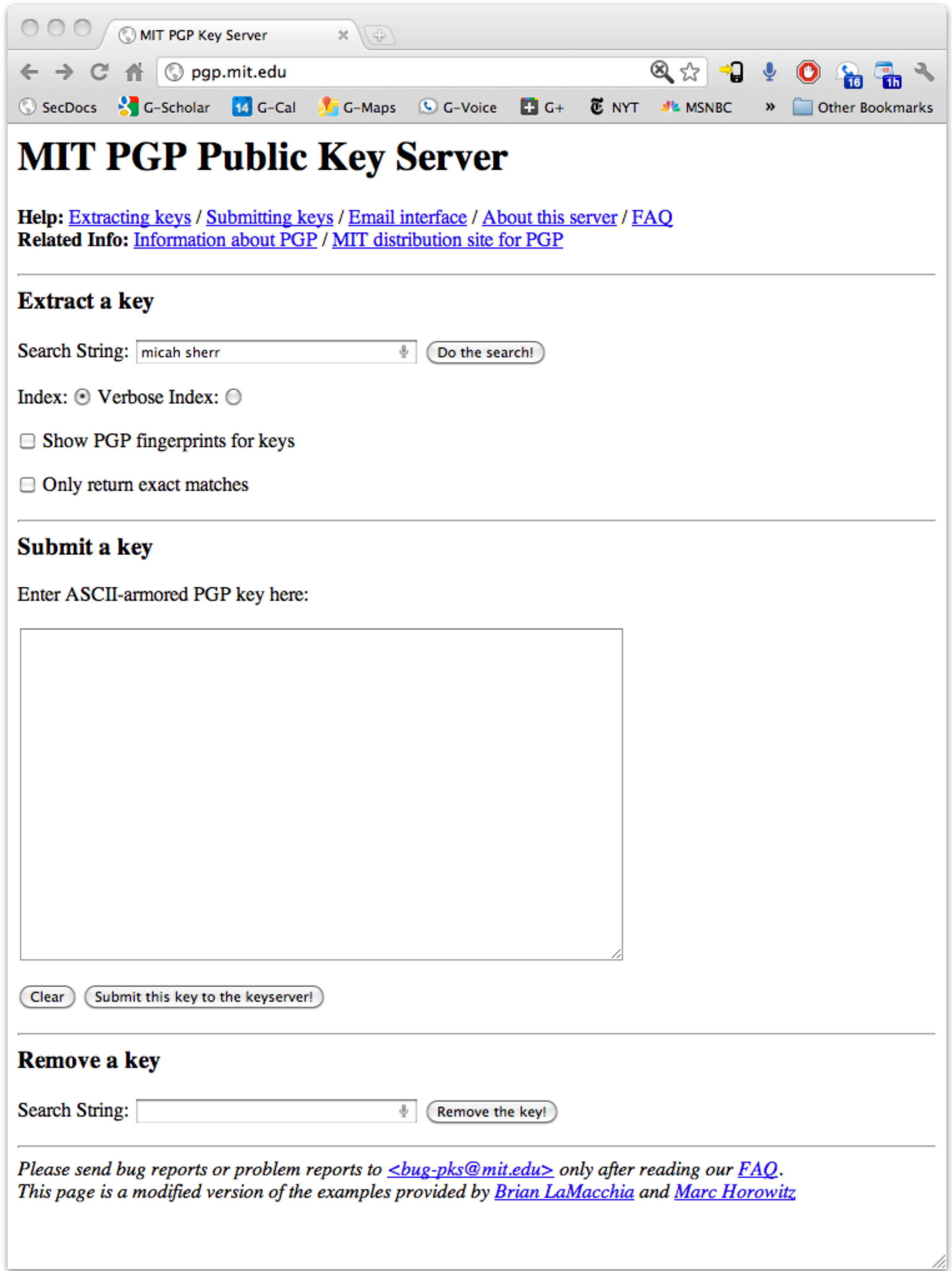
Bob

(A^+,A^-) is Alice’s long-term public-private key pair.

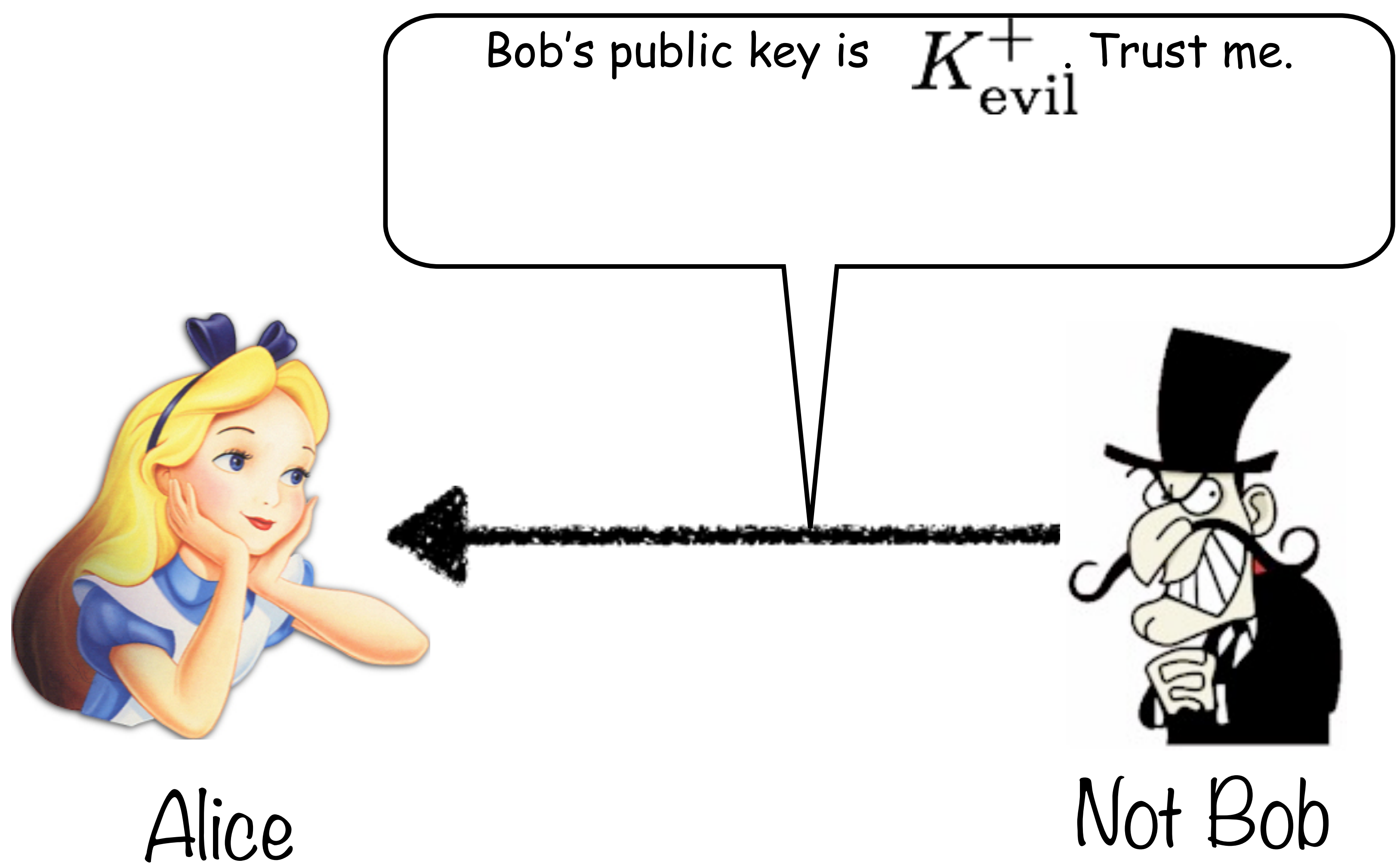
(B^+,B^-) is Bob’s long-term public-private key pair.

k is the session key; sometimes called the **ephemeral key**.

Key Management



But how do we *verify* we're using the correct public key?



Short answer:
We can't.

It's turtles all the way down.

(more on this next week)