Intro to crypto

PQC Spring School 2024

Kathrin Hövelmanns

March 12th, 2024

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Brief history of communicating secrets

Scytale

Bodymod steganography
(Histiaeus,
acc. to Herodotus)

Caesar cipher

700 BC

440 BC

50 BC

Brief history of communicating secrets

Scytale

Bodymod steganography
(Histiaeus,
acc. to Herodotus)

Caesar cipher

700 BC

440 BC

50 BC

‚Shift by 3‘ rule:

A	→	D
T		W
T		W
A		D
C		F
K		N

Brief history of communicating secrets

Scytale

Bodymod steganography
(Histiaeus,
acc. to Herodotus)

Caesar cipher

‚Shift by 3' rule:

A	→	D	D	→	A
T		W	W		T
T		W	W		T
A		D	D		A
C		F	F		C
K		N	N		K

700 BC

440 BC

50 BC

Brief history of communicating secrets

Scytale

Bodymod steganography
(Histiaeus,
acc. to Herodotus)

Caesar cipher

700 BC

440 BC

50 BC

Problem:

Techniques will never remain secret.

Brief history of communicating secrets

Scytale

Bodymod steganography
(Histiaeus,
acc. to Herodotus)

Caesar cipher

'It should not be a problem if [the system] falls into enemy hands.'

Kerckhoffs, 1883

700 BC

440 BC

50 BC

Brief history of communicating secrets

‚Caesar with codeword‘:

Codeword:
CRYPTO

A	A -> C	C
T	A -> R	K
T	A -> Y	R
A	A -> P	P
C	A -> T	V
K	A -> O	Y

Vigenère

… not too much going on here …

700 BC

440 BC

50 BC

1500 AD

Brief history of communicating secrets

‚Caesar with codeword‘:

Codeword:
CRYPTO

A	→	A -> C	→	C
T		A -> R		K
T		A -> Y		R
A		A -> P		P
C		A -> T		V
K		A -> O		Y

No codeword ->
no info

Vigenère

… not too much going on here …

700 BC

440 BC

50 BC

1500 AD

Brief history of communicating secrets

‚Caesar with codeword‘:

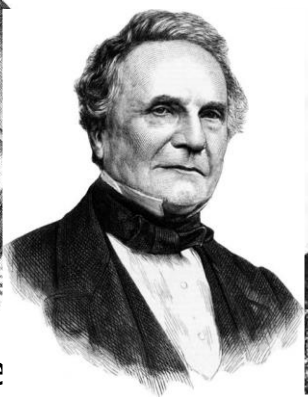
Codeword:
CRYPTO

A	A -> C	C
T	A -> R	K
T	A -> Y	R
A	A -> P	P
C	A -> T	V
K	A -> O	Y

No codeword -> no info

Tried Statistics?

Vigenère



Babbage
1854

Kasiski
1863

… not too much going on here …

700 BC

440 BC

50 BC

1500 AD

Brief history of communicating secrets

Scytale

Bodymod st
(Histi
acc. to H

Beware hubris.

You don’t find attacks on how you communicate?

Doesn’t mean no one else does!

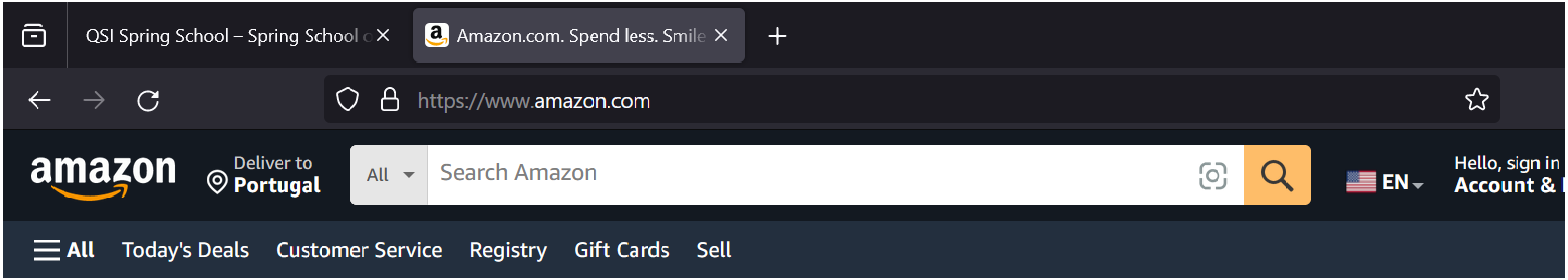
Caesar cipher

Vigenère

… not too much going on here …

700 BC   440 BC   50 BC   1500 AD   1930 AD

Did you use any cryptography today?



Amazon uses https, https invokes the TLS protocol

TLS uses cryptography

TLS is actually quite ubiquitous:

shopping, banking, Netflix, gmail, Facebook (yes, I'm old), ...

Did you use any cryptography today?

Secure instant messaging:

How many apps do you use?

What do we want from cryptography?

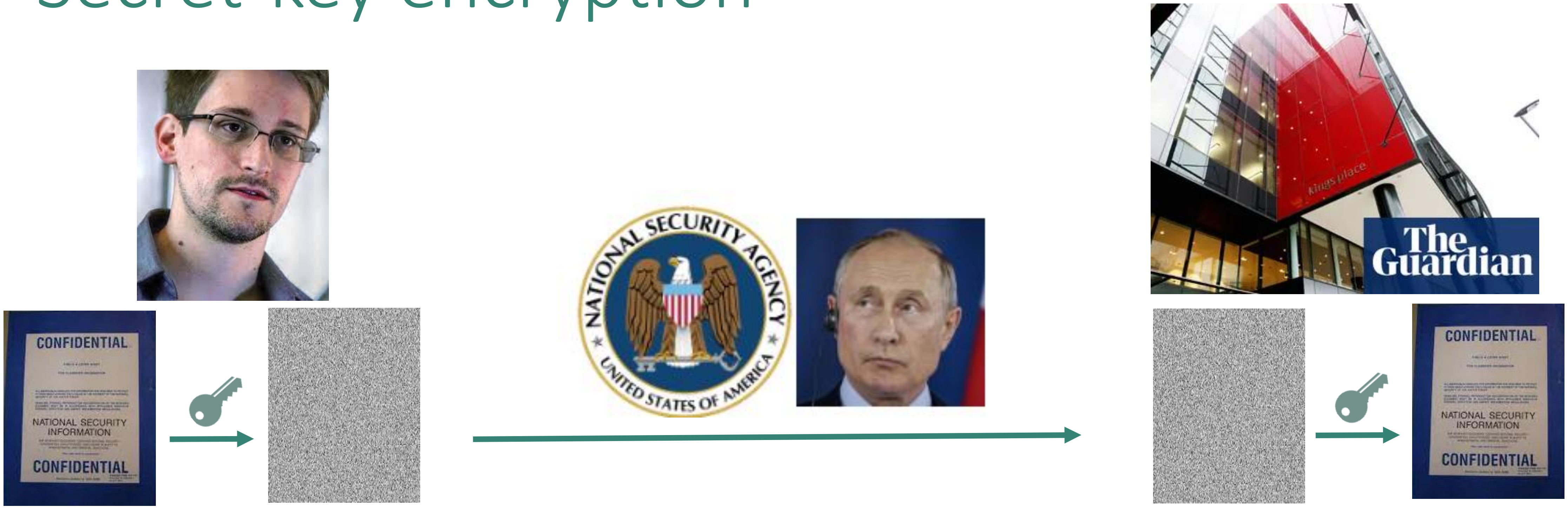
Confidentiality:

Keeping secrets secret.

Integrity + authenticity:

Ensure that message really came from declared sender + arrived unaltered

Secret-key encryption



$Encrypt$ takes plaintext and key, and produces ciphertext

$Decrypt$ takes ciphertext and key, and produces plaintext

Goal #1: Confidentiality despite espionage (prerequisite: adversary does not know key)

One-time pad

Key K is picked uniformly random from ℓ -bit strings: $K \leftarrow \{0,1\}^{\ell}$

Plain- and ciphertexts are also ℓ -bit strings: $m, \ c \in \{0,1\}^{\ell}$

$Encrypt_K(m) = K \oplus m$: add K and m, modulo 2 in each position

mod 2 = divide by 2, take remainder

e.g., $01 \oplus 11 = (0 + 1\ mod\ 2)(1 + 1\ mod\ 2) = 10$

$Decrypt_K(c) = K \oplus c$

This works: $Decrypt_K\big(Encrypt_K(m)\big) = K \oplus Encrypt_K(m) = K \oplus K \oplus m = m$

Perfect security

Formally: (*KeyGen*, *Encrypt*, *Decrypt*) **perfectly secure** iff for all plaintexts m_1, m_2 and all ciphertexts c:

$$\Pr[Encrypt_K(m_1) = c] = \Pr[Encrypt_K(m_2) = c]$$

Probability taken over the choice of key K

Important fact (Shannon): only possible if there are as many keys as there are potential messages

One-time pad is perfectly secure

One-time pad: $Encrypt_K(m) = K \oplus m$, K chosen randomly

Suppose adversary

- gets c = 01
- knows: m is either m_1= 11 or m_2= 01
- but doesn't know K

Can it tell which message m was?

No: could be m_1= 11 (if K= 10) or m_2= 01 (if K= 00)

both equally likely!

One-time pad is perfectly secure… <u>if used **once**</u>

One-time pad: $Encrypt_K(m) = K \oplus m$, K chosen randomly

Suppose

- adversary sees first encryption: c_1 = 01
- **but now also c_2 = c_1 = 01**

→ Adversary learns that same message was sent twice

Computational security

We want to encrypt

- arbitrary amounts of data
- with a single, short key

→ perfectly secure symmetric-key encryption infeasible in practice

Computational security ('IND-CPA') as relaxation of security goal:

Telling $Encrypt_K(m_1)$ from $Encrypt_K(m_2)$ should be

- computationally infeasible (**IND**istinguishability),
- even if you chose m_1 and m_2 yourself (**C**hosen **P**laintext **A**ttack).

Permutations

A permutation is a mapping $\Pi: S \to S$ from some set S to itself that is one-to-one.

In other words: Π has an inverse $\Pi^{-1}: S \to S$.

Example: $S = \{A, B, C\}$

A permutation and its inverse:

x	A	B	C
$\pi(x)$	C	A	B

y	A	B	C
$\pi^{-1}(x)$	B	C	A

Not a permutation:

x	A	B	C
$\pi(x)$	C	B	B

Block ciphers are families of permutations

Block ciphers = two-input functions

$$\mathrm{E}\colon\ Keys \times \{0,1\}^\ell \to \{0,1\}^\ell$$

so such each key K gives us a permutation

$$\begin{aligned} E_K\colon \{0,1\}^\ell &\to \{0,1\}^\ell \\ x &\mapsto E(K,x) \end{aligned}$$

(so for each key K, E_K has an inverse E_K^{-1})

(For practice: all functions E_K, E_K^{-1} should be efficiently computable)

Using block ciphers to encrypt

Encrypting $m = m_1 \cdots m_\ell$:

$c = \mathrm{E_k}(m_1) \cdots \mathrm{E_k}(m_\ell)$

$\xrightarrow{\quad c \quad}$

Decrypting $c = c_1 \cdots c_\ell$:

$m = \mathrm{E}_\mathrm{k}^{-1}(c_1) \cdots \mathrm{E}_\mathrm{k}^{-1}(c_\ell)$

Security requirement:

c should leak neither m nor k!

Data Encryption Standard (DES)

1972: NBS (now NIST) aims to standardise a block cipher

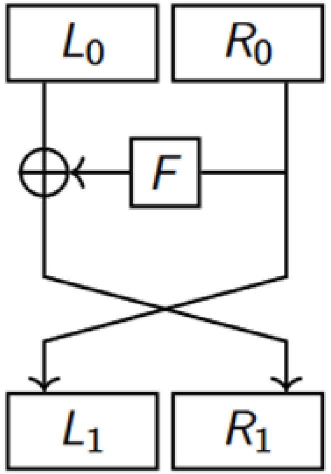
1974: IBM designs Lucifer, which evolves into DES

Widely adopted (e.g., used in ATMs)

High-level design:

- Feistel network, made of successive rounds
- Each round = simple operation, using a bit of the secret key

Data Encryption Standard (DES): Feistel round



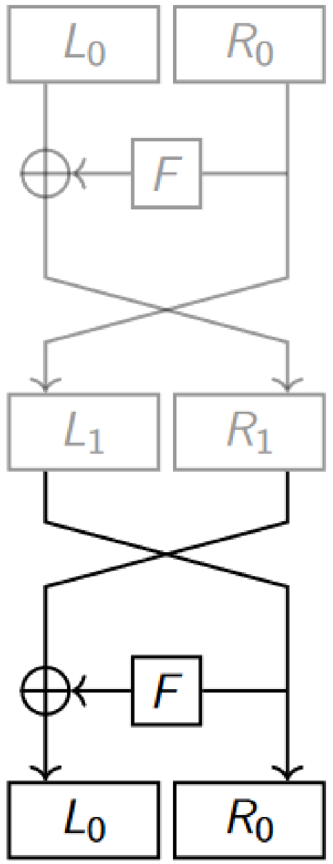
Split message into left half (L_0) and right half (R_0)

Apply some nonlinear (key-dependent) function F to R_0 to get OTP key for L_0

Swap sides

Image credit: E. Thome

Data Encryption Standard (DES): Feistel round



Split message into left half (L_0) and right half (R_0)

Apply some nonlinear (key-dependent) function F to R_0 to get OTP key for L_0

Swap sides

We can invert easily → this is a permutation!

Image credit: E. Thome

Data Encryption Standard (DES): round chaining

One round looks simple enough

→ in practice DES chains as many as 16 rounds

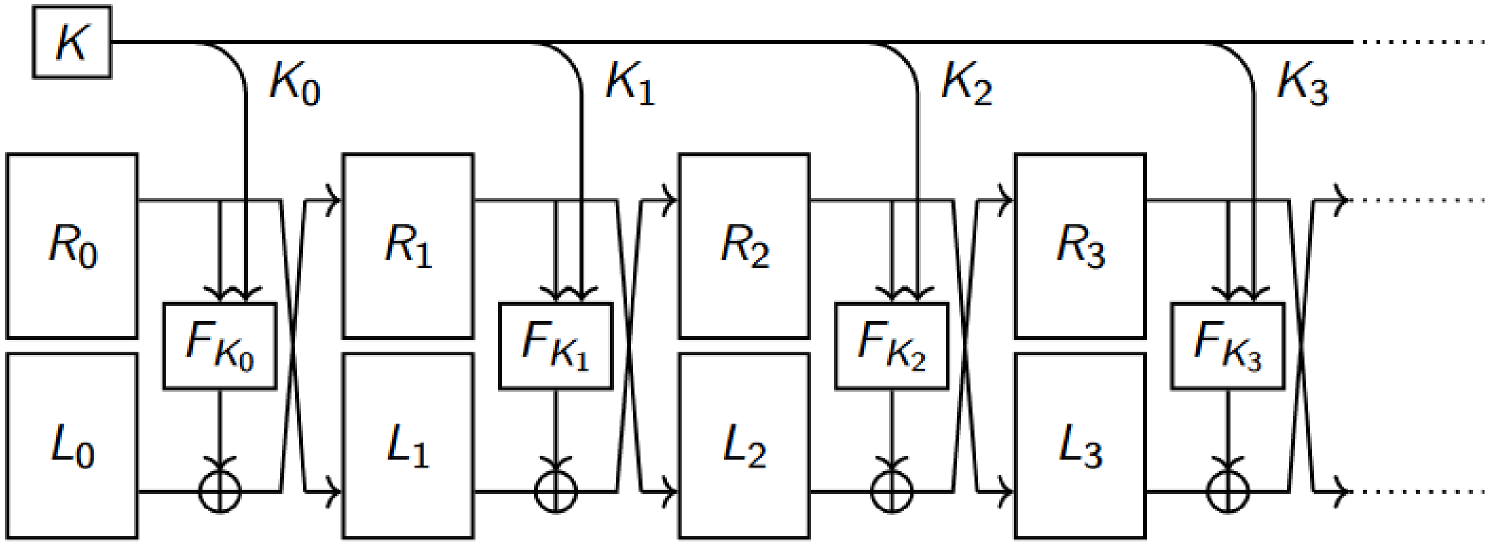


Image credit: E. Thome

Block cipher evolution

DES key length: 56 bits → brute-force vulnerability:

- DES cracker (1998, Electronic Frontier Foundation, US$ 250,000)
- COPACOBANA (2006, U Bochum + Kiel, US$ 10,000)

<u>If</u> DES is still used, then as Triple-DES, using three keys k_1, k_2 and k_3:

$$c = Encrypt_{k_3}\left(Decrypt_{k_2}\left(Encrypt_{k_1}(m)\right)\right)$$

AES: new standard, established in 2001

- chosen during 'competition' established by National Institute for Standardisation (NIST)
- not Feistel-based: based on Rijndael cipher, designed by Daemen and Rijmen

Modes of operation

So far: block cipher encrypt ℓ bits of message

What if messages are longer than ℓ bits?

Just split + encrypt block-wise? ('Electronic codebook')

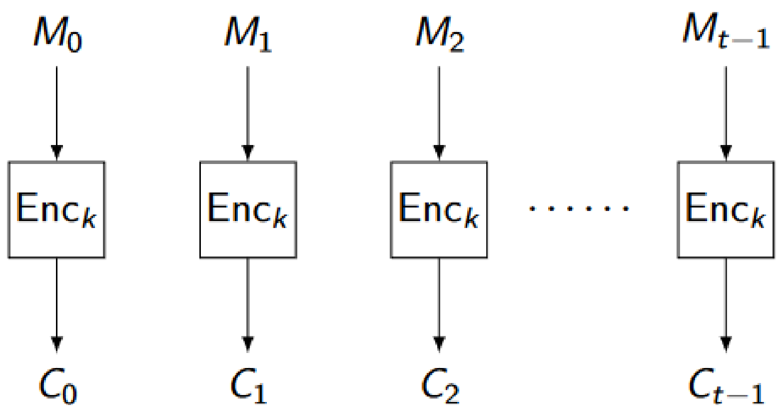


Image credit: T. Lange + J. Jean

Modes of operation

Other approaches
+ quantum impact: Andre’s talk

So far: block cipher encrypt ℓ bits of message

What if messages are longer than ℓ bits?

Just split + encrypt block-wise? (‘Electronic codebook’)

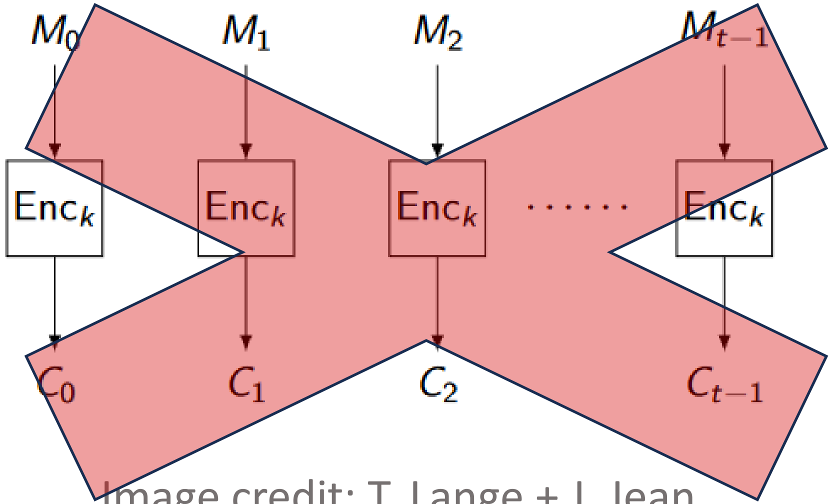
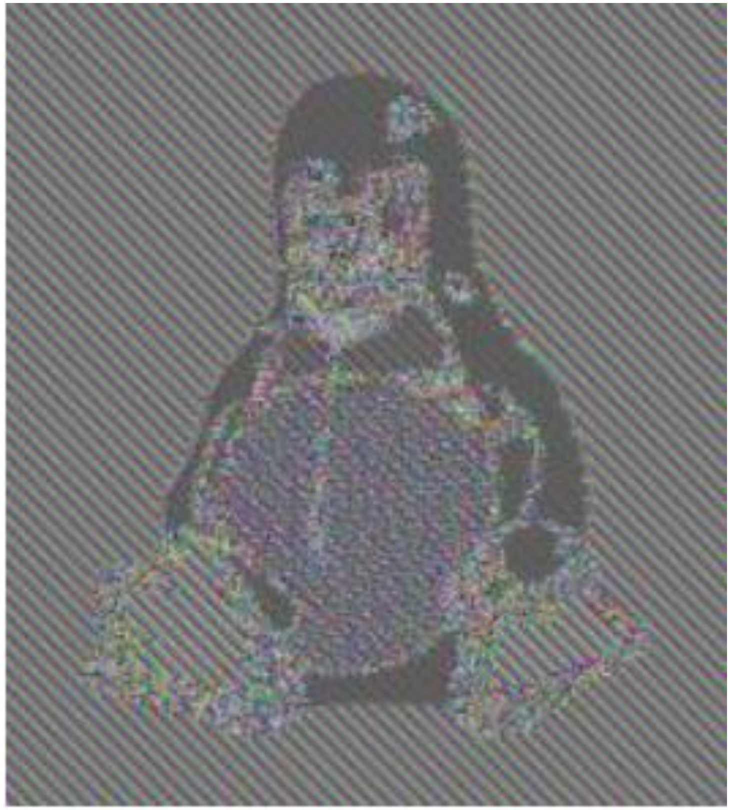


Image credit: T. Lange + J. Jean



ECB penguin by en:User:Lunkwill

Secret-key encryption: wrap-up

Perfect secrecy is expensive (**large** keys)

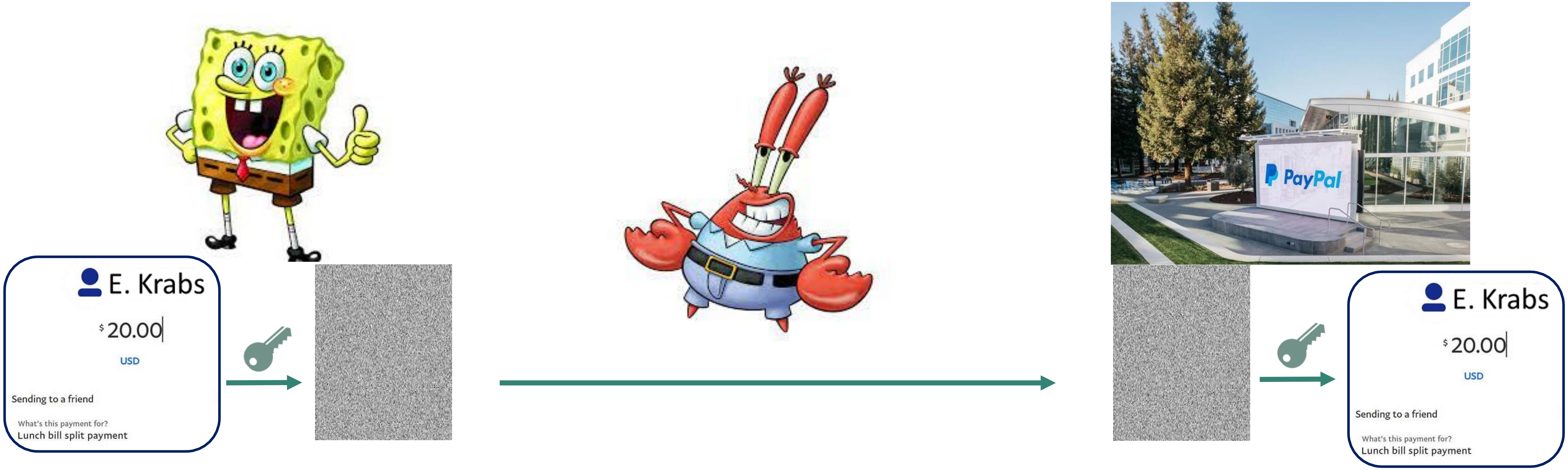
One-time pad only is perfectly secure if we switch the key each time

In practice, we use a

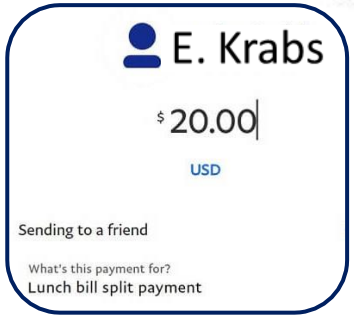
- block cipher to encrypt blocks
- secure mode of operation (not ECB!) to encrypt messages longer than a single block

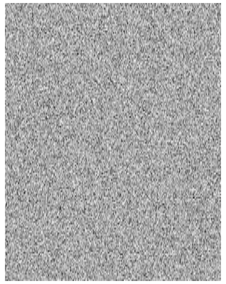
So far: discussed confidentiality, but not authenticity and/or integrity

Does secret-key encryption provide integrity?



Does secret-key encryption provide integrity?





Mr. Krabs knows his block ciphers → tweaks ciphertext so it decrypts to 'pay 99000' instead of 'pay 20'.

Hash functions

Function generating short handle (‘fingerprint’) for larger pieces of data:

$$\mathrm{Hash}\colon \{0,1\}^* \to \{0,1\}^n$$

Quite **ubiquitous in crypto**:

- message authentication codes (in a few slides: HMAC), e.g. in TLS
- digital certificates for public-key infrastructures
- public-key encryption, digital signatures (in second half of talk)
- secure password storage

Hash functions

Function generating short handle (‘fingerprint’) for larger pieces of data:

$$\text{Hash}: \{0,1\}^* \rightarrow \{0,1\}^n$$

Security goals: e.g. we could want that the fingerprints

- are hard to compute without knowing the data
- change a lot even when the data is changed only a tiny bit (e.g., bit flip)
- uniquely identify the data (PGP fingerprints)
- do not give enough information to reconstruct the data

Hash functions: security definitions

Function generating short handle ('fingerprint') for larger pieces of data:

$$\text{Hash}: \{0,1\}^* \to \{0,1\}^n$$

- **Preimage resistance**:

  Given output $y \in \{0,1\}^n$, it's hard to find $x \in \{0,1\}^*$ with $\text{Hash}\,(x) = y$ ('preimage').

  typically many!

- **Second preimage resistance**:

  Given random input $x \in \{0,1\}^*$, it's hard to find $x' \neq x$ with $\text{Hash}\,(x) = \text{Hash}\,(x')$.

- **Collision resistance**:

  It's hard to find x and $x' \neq x$ with $\text{Hash}\,(x) = \text{Hash}\,(x')$.

Increasingly harder task for adversary

Hash functions: SHA-2 (‘Secure hash algorithm’)

Designed by the National Security Agency (NSA), first published in 2001.

Built using the **Merkle–Damgård** construction (next slide), from a **compression function**.

Main idea:

- easier to build fixed-size **compression**
- If you have secure **compression** function, **MD** gives you a hash function for free

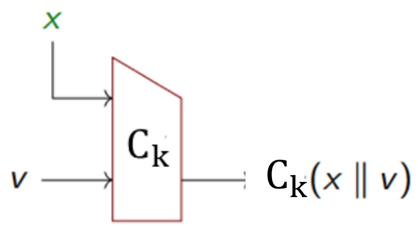
Compression in SHA-2:

Davies-Meyer construction, using specialized block cipher

Family of keyed functions

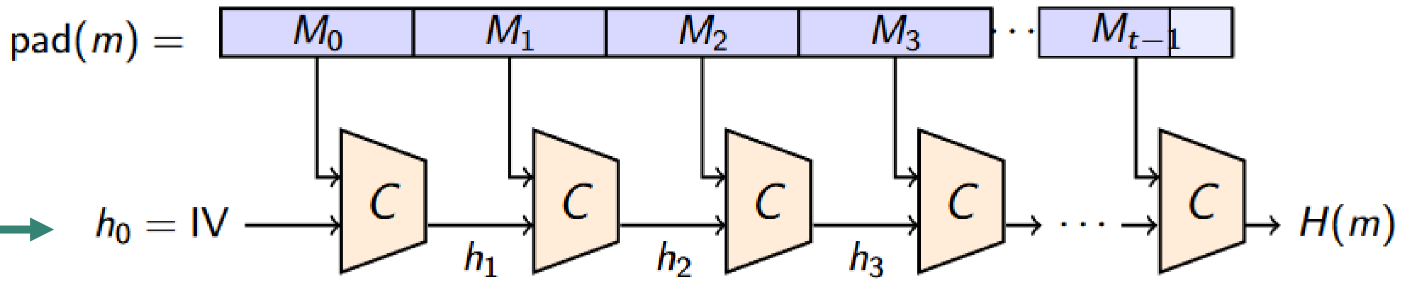
$$\mathrm{C}: \{0,1\}^k \times \{0,1\}^{2n} \to \{0,1\}^n$$

with inputs of fixed size $2n$ that get ‘compressed’ to half their size.



Box based on slide by E. Thome

Hash functions: Merkle-Damgård construction



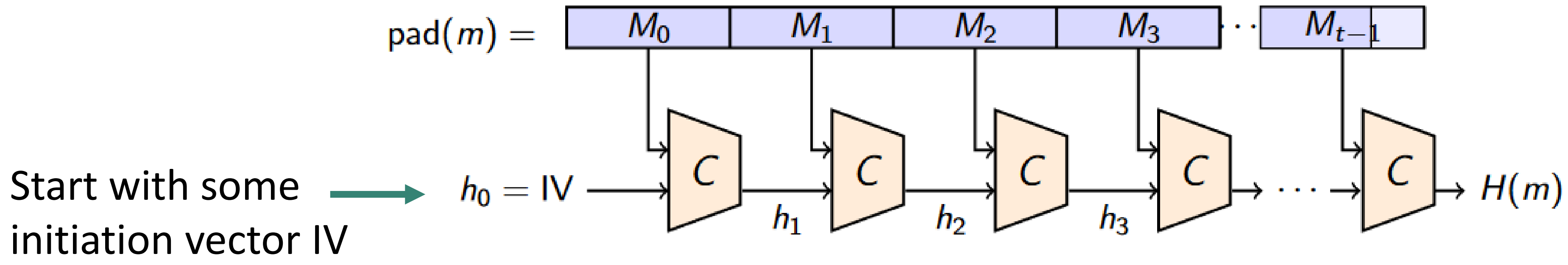
Start with some initiation vector IV

$pad(m)$:

- Dissect full message m into size-n blocks $M_1, \cdots M_t$ (to fit into compression function C)
- Use padding in the last block M_t to fill it up to size n

Each step takes n message bits as input, together with previous n-bit output h_{i-1}, and compresses these to n-bit block: $h_i = C(M_{i-1}, h_{i-1})$.

Hash functions: Merkle-Damgård construction



$pad(m)$:

- Dissect full message m into size-n blocks $M_1, \cdots M_t$ (to fit into compression function C)
- Use padding in the last block M_t to fill it up to size n

Pros of this iterative design:

- Simplifies security reasoning: if compression function C is collision-resistant, then so is H.
- Incremental computation nice for small devices (stream data one block at a time)

Hash functions evolution

SHA-1 (predecessor of SHA-2):

- flaws known since 2005, attacks public since 2017 (https://shattered.io/), 2020 (https://sha-mbles.github.io/)
- still used for fingerprints (e.g., git) ☹

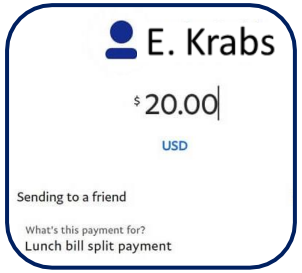
SHA-2:

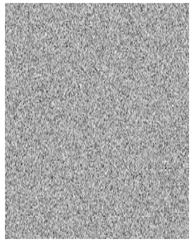
- currently deemed secure
- widely used in various security applications and protocols

SHA-3: Latest addition to SHA family

- established during NIST standardization effort for hash functions
- not based on Merkle-Damgård, but on ‘sponges’
- currently deemed secure

Hash functions good integrity checks?





send $\left(M, tag = \mathrm{Hash}(M)\right)$

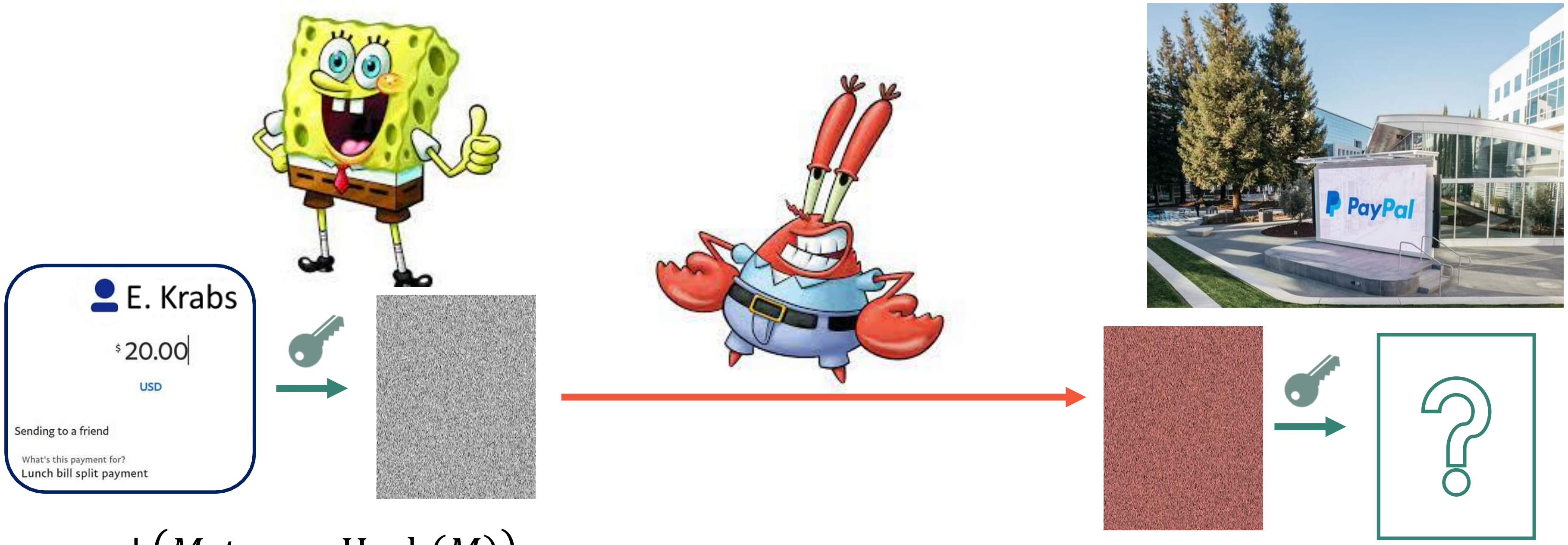
e.g., $\mathrm{Hash} = \mathrm{SHA3}$

receive (M', tag')

check that $tag' = \mathrm{Hash}(M')$

Q: Does this ensure the integrity of M'?

Hash functions good integrity checks?



send $\left(M, tag = \mathrm{Hash}(M)\right)$

e.g., $\mathrm{Hash} =$ SHA3

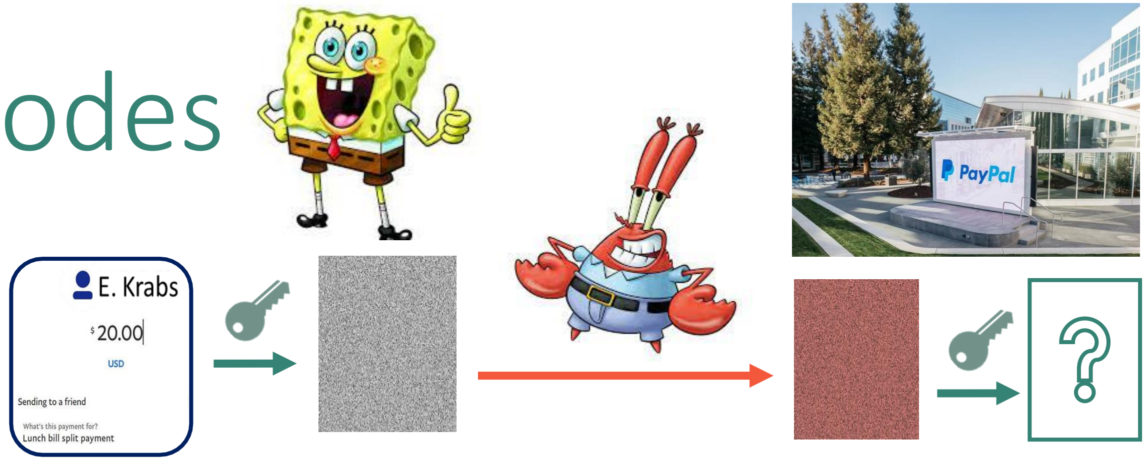
receive (M', tag')

check that $tag' = \mathrm{Hash}(M')$

Q: Does this ensure the integrity of M'?

No: Mr. Krabs can pick his own c' and compute tag' for c' → **keyless integrity checks won't work**!

Message authentication codes



MAC = 'checksum', taking key k and message M (plaintext or ciphertext) to produce authentication tag:

$$\mathrm{MAC}\colon Keys \times \{0,1\}^m \rightarrow \{0,1\}^t$$

$\rightarrow$ MAC can convince Paypal that M really comes from Spongebob

Security goal = UnForgeability: Computing a valid MAC without knowing k is hard.

- **UF against Chosen Message Attacks (UF-CMA):**

  even when given the power to request $\mathrm{MAC}(k, M_i)$ on chosen messages M_i,
  computing a valid $\mathrm{MAC}(k, M')$ for a new a new $M' \neq M_i$ is hard.

Hash-based MACs

Proposal: Take hash function $\mathrm{Hash}\colon \{0,1\}^* \to \{0,1\}^n$ and set

$$\mathrm{MAC}_k(M) = \mathrm{Hash}(k, M)$$

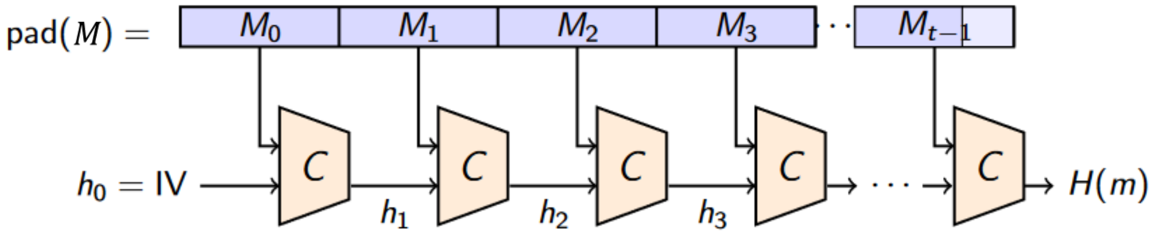
Q: Hard to produce a valid $\mathrm{MAC}_k(M')$ if we can request $\mathrm{MAC}_k(M_i)$ for any M_i we like?

Hash-based MACs

Proposal: Take hash function Hash: $\{0,1\}^* \to \{0,1\}^n$ and set

$$\mathrm{MAC}_k(M) = \mathrm{Hash}(k, M)$$

Length extension attack :



Exploit 'chaining' structure of Hash: pick message $M = hello$, request $tag = \mathrm{Hash}(k, hello)$.

- View $hello$ in padded block structure + add something: $M' = \boxed{hell|oXXX|\boldsymbol{dork}}$
- Tag for $helloXXXdork$:

$$\mathrm{Hash}(k, helloXXXdork) = \mathrm{Hash}(\mathrm{Hash}(k, hello), dork) = \mathrm{Hash}(tag, dork)$$

Without knowing $\boldsymbol{k}$, we can forge a tag for the message $\boldsymbol{helloXXXdork}$!

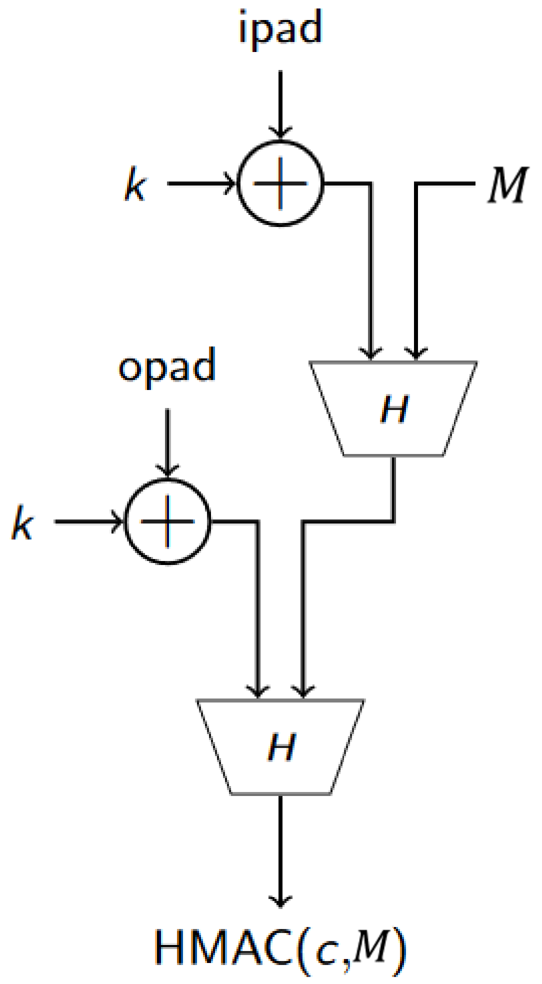
Hash-based MACs: HMAC

Puts the key k

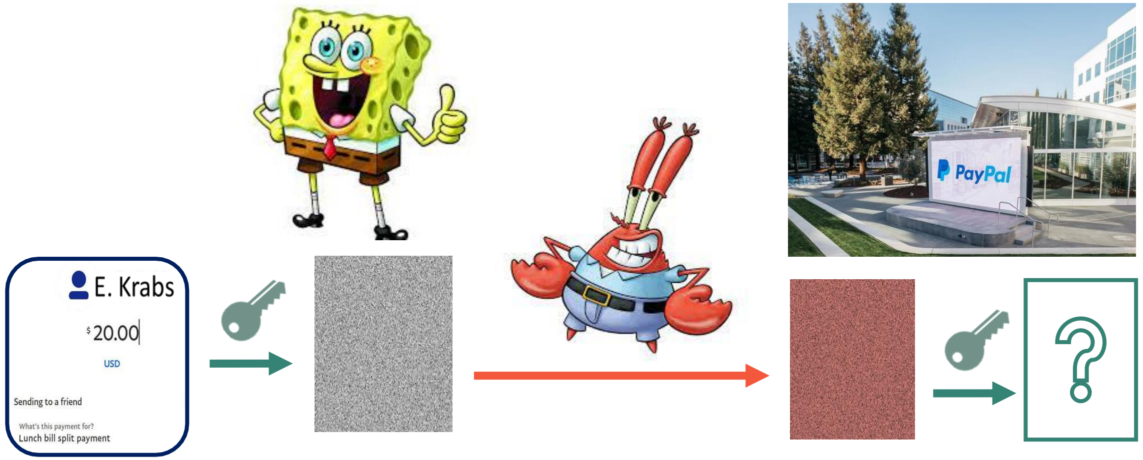
- at the end to prevent length-extension attacks (you'd need to know $dork|\boldsymbol{k}$),
- but also at the beginning (to deal with collisions).

Mixes up k via two different padding strings ($ipad$, $opad$), so that the MAC doesn't use the same key twice

$$\mathrm{HMAC}_k(M) = \mathrm{Hash}(k \oplus opad, \mathrm{Hash}(k \oplus ipad, M))$$



Authenticated encryption



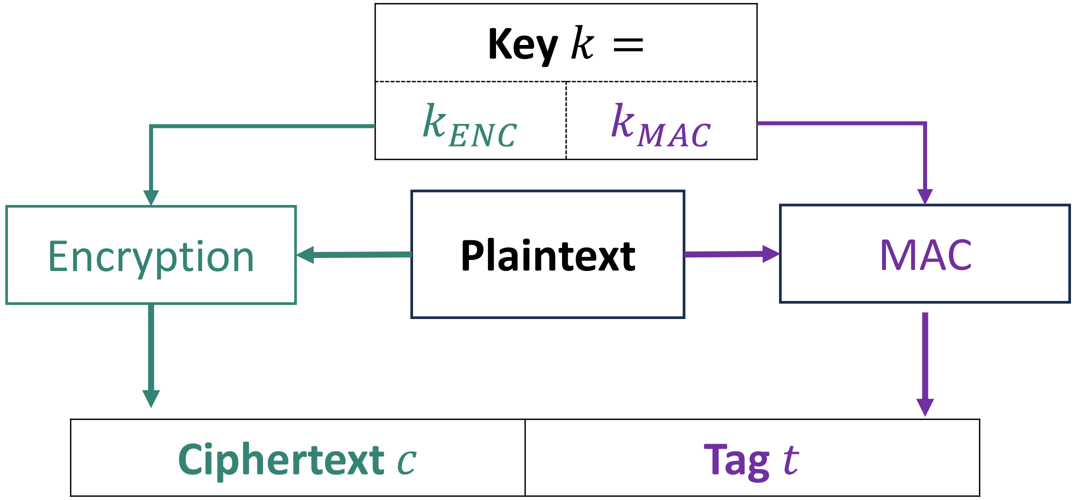
We looked at confidentiality and authenticity separately:

Goal	Primitive	Security notion
Data confidentiality	Secret-key encryption	**IND-CPA**: Hard to tell $Encrypt_K(m_1)$ from $Encrypt_K(m_2)$
Data authenticity / integrity	Message authentication code	**UF-CMA**: Hard to forge $MAC(k, M')$, even when seeing $MAC(k, M_1)$, $MAC(k, M_2), \cdots$

Q: How to achieve both goals at once?

Three common combination approaches

- **Encrypt-and-MAC**
  - used in SSH

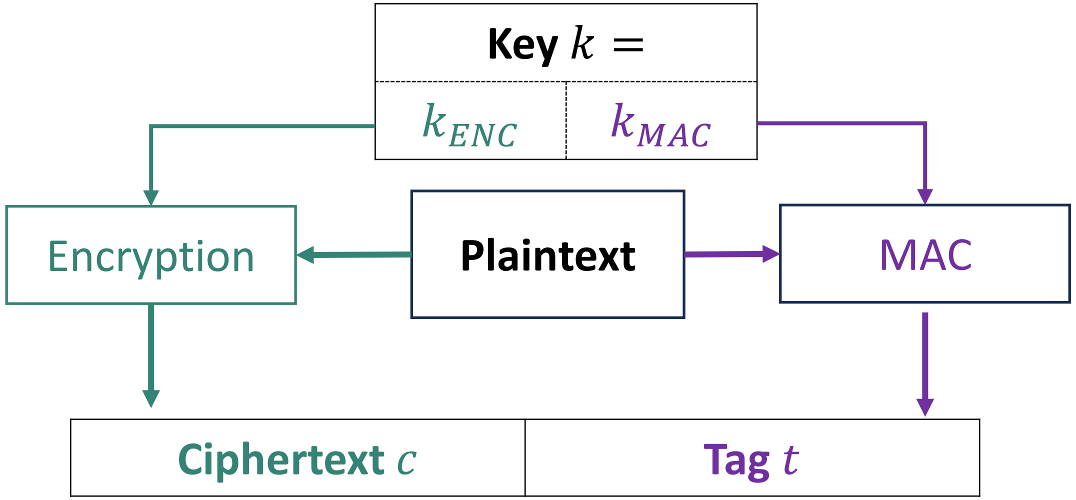


Confidentiality?

Adversaries can detect resent messages because MAC is deterministic

Three common combination approaches

- **Encrypt-and-MAC**
  - used in SSH
  - not secure per se (SSH uses modifications)

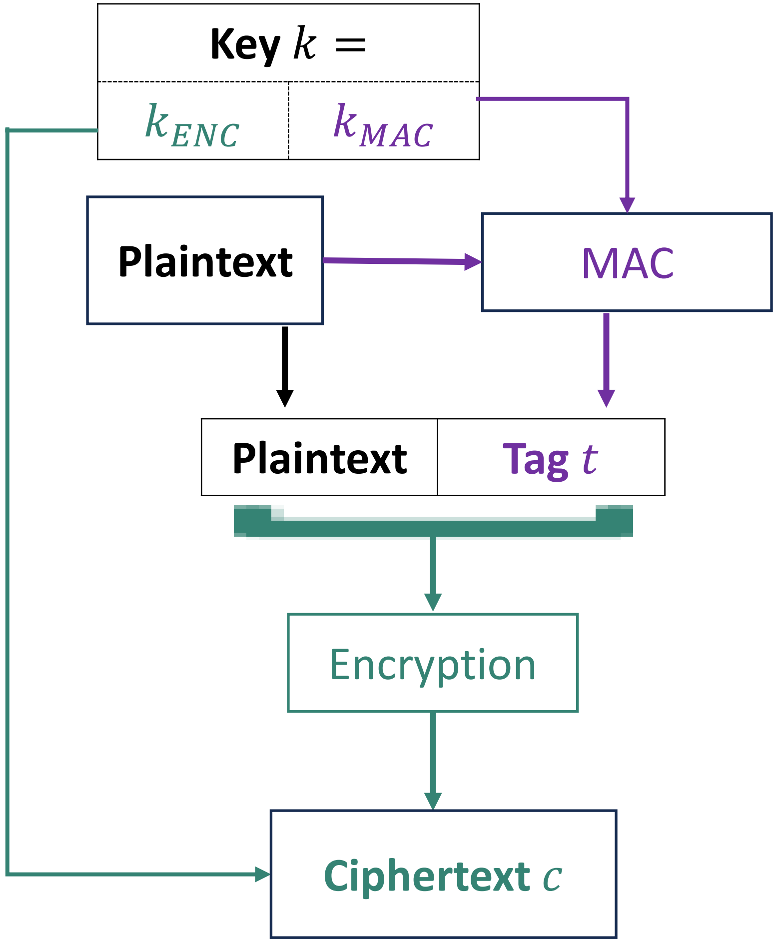


Integrity?

Not necessarily: may be able to tweak c into c' in a way that its decryption is still the same. Then t is still valid!

Three common combination approaches

- **Encrypt-and-MAC**
  - used in SSH
  - not secure per se (SSH uses modifications)
- **MAC-then-Encrypt**
  - used in TLS 1.2



Confidentiality?

If encryption is IND-CPA secure,

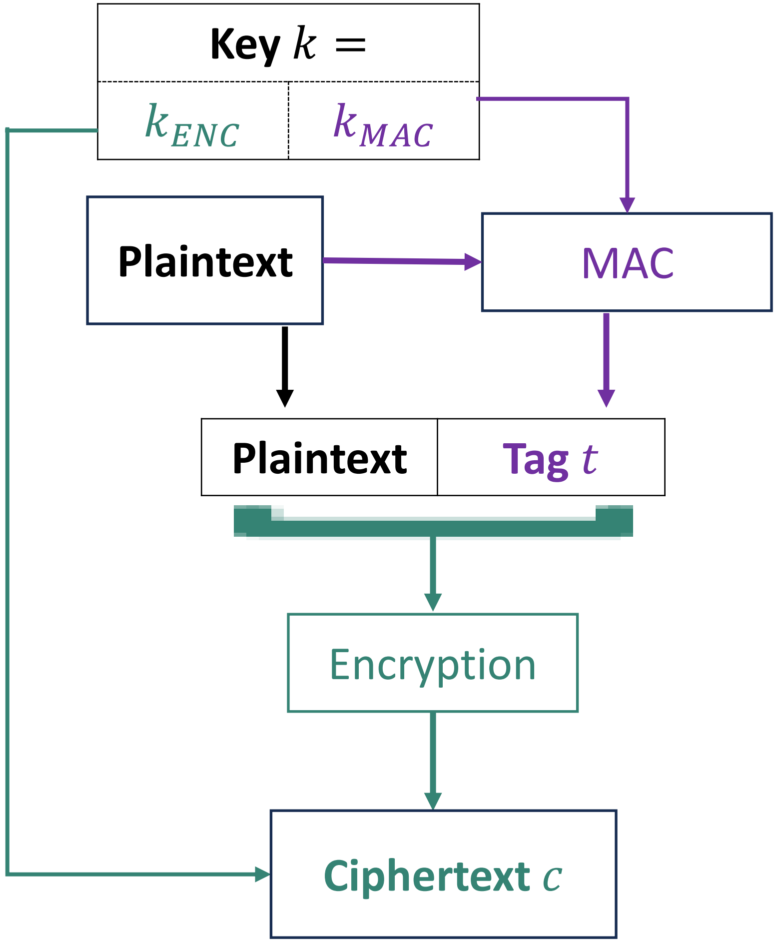
- resent messages are unnoticeable (despite MAC)
- the MAC-then-encrypt construction is also IND-CPA secure

Three common combination approaches

- **Encrypt-and-MAC**
  - used in SSH
  - not secure per se (SSH uses modifications)
- **MAC-then-Encrypt**
  - used in TLS 1.2
  - not secure per se, but can be if done right

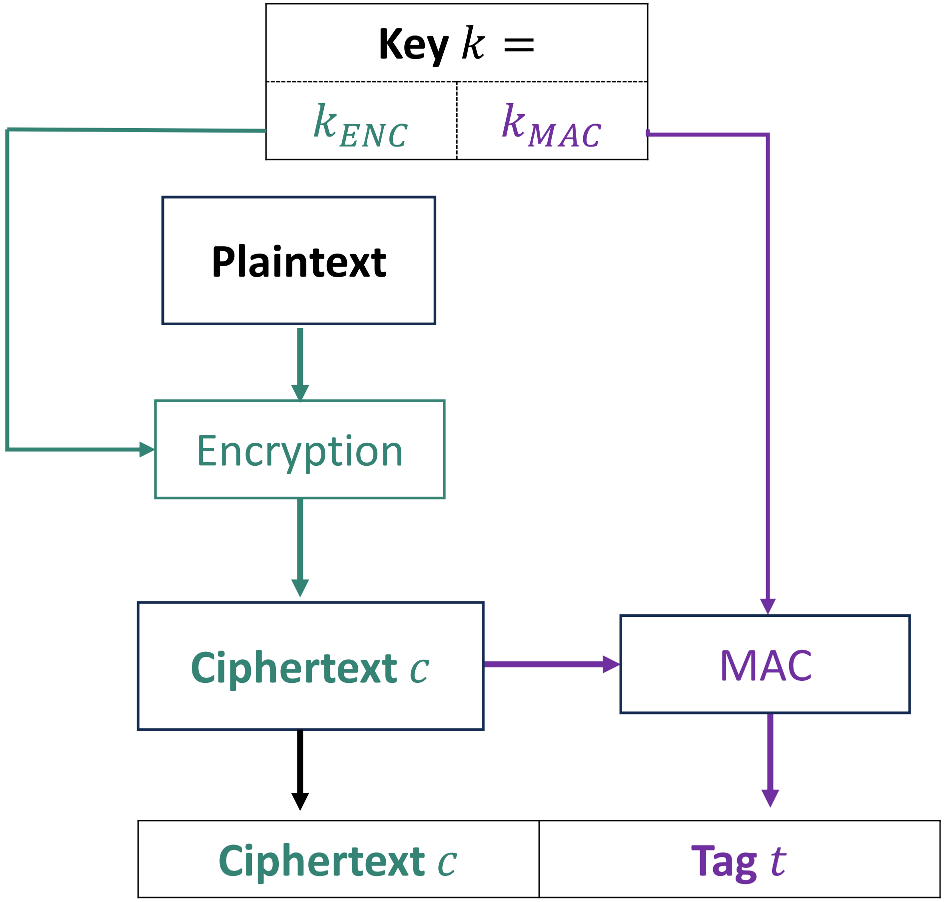
Integrity?

Same problem as before!



Three common combination approaches

- **Encrypt-and-MAC**
  - used in SSH
  - not secure per se (SSH uses modifications)
- **MAC-then-Encrypt**
  - used in TLS 1.2
  - not secure per se, but can be if done right
- **Encrypt-then-MAC**
  - used in IPSec
  - Confidentiality: IND-CPA if Encryption is IND-CPA
  - Integrity: no computing right t' for c' without k_{MAC}



Proof sketch: Encrypt-then-MAC is IND-CPA

Want to show: if $Encrypt$ is IND-CPA secure, then so is Encrypt-then-MAC.

Encrypt-then-MAC$(k_{ENC}, k_{MAC}, m) = (c, t)$ with $c = Encrypt(k_{ENC}, m)$ and $t = MAC(k_{MAC}, c)$

Tool: Turn attack on Encrypt-then-MAC into attack on $Encrypt$ (‘**security reduction**’):

- Show: Successful attack on Encrypt-then-MAC gives successful attack on $Encrypt$
- But $Encrypt$ is secure. So no successful attack on Encrypt-then-MAC can exist!

Encrypt-then-MAC functionality

c = encryption of either m_1 or m_2
$t = MAC(k_{MAC}, c)$

m_1, m_2

c, t

IND-CPA attack on Encrypt-then-MAC

(c, t) belongs to m_1/ m_2!

Proof sketch: Encrypt-then-MAC is IND-CPA

Want to show: if $Encrypt$ is IND-CPA secure, then so is Encrypt-then-MAC.

Encrypt-then-MAC$(k_{ENC}, k_{MAC}, m) = (c, t)$ with $c = Encrypt(k_{ENC}, m)$ and $t = MAC(k_{MAC}, c)$

Tool: Turn attack on Encrypt-then-MAC into attack on $Encrypt$ (‘**security reduction**’):

- Show: Successful attack on Encrypt-then-MAC gives successful attack on $Encrypt$
- But $Encrypt$ is secure. So no successful attack on Encrypt-then-MAC can exist!

$Encrypt$ functionality

c = encryption of either m_1 or m_2

m_1, m_2

c

IND-CPA attack on $Encrypt$

c belongs to m_1/ m_2! (copy Vlad)

Pick your own k_{MAC}

c = encryption of either m_1 or m_2

$t = MAC(k_{MAC}, c)$

m_1, m_2

c, t

IND-CPA attack on Encrypt-then-MAC

(c, t) belongs to m_1/ m_2!

How to share a secret key?

Public-key encryption (PKE)

Alice

Bob

Image source: xkcd.com

Public-key encryption (PKE)

Alice

Bob

Image source: xkcd.com

Public-key encryption (PKE)

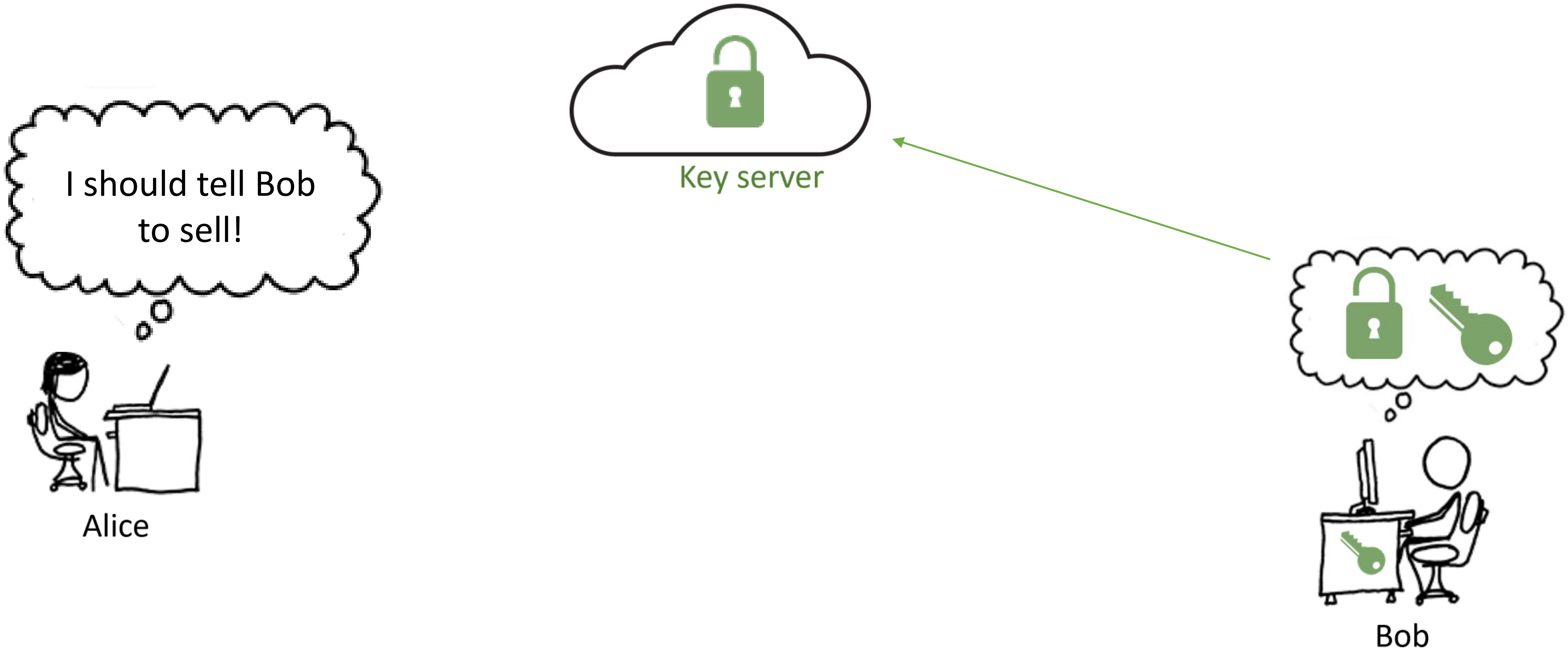


Image source: xkcd.com

Public-key encryption (PKE)

Key server

Alice

Bob

Image source: xkcd.com

Public-key encryption (PKE)

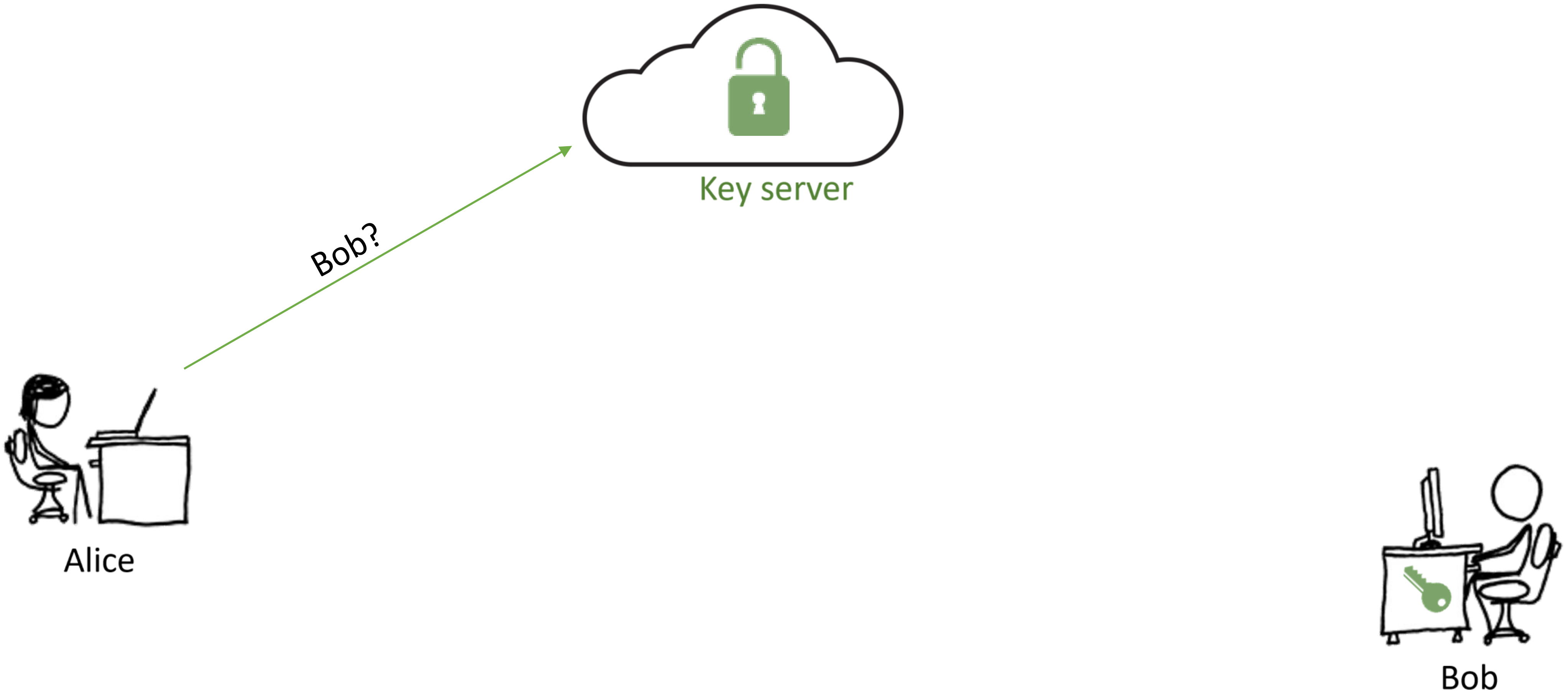


Image source: xkcd.com

Public-key encryption (PKE)

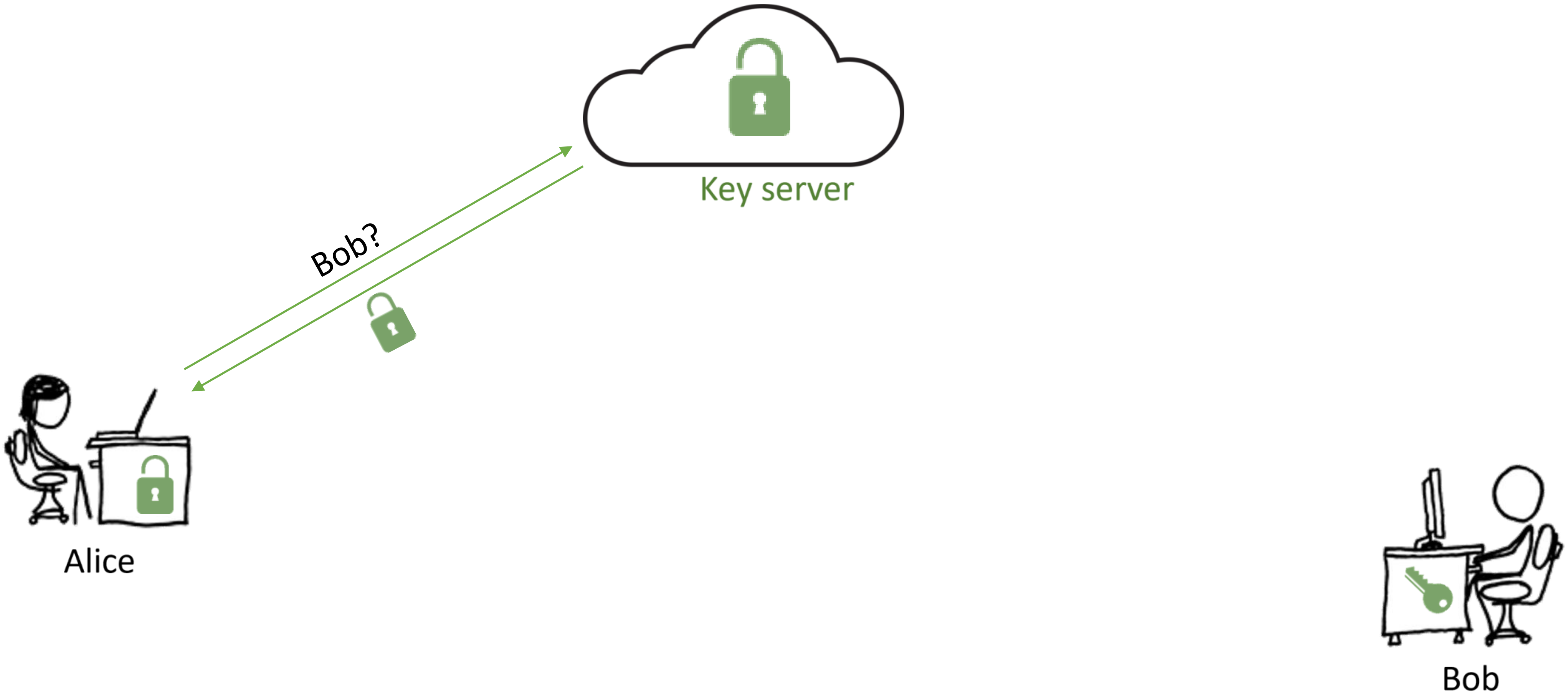


Image source: xkcd.com

Public-key encryption (PKE)

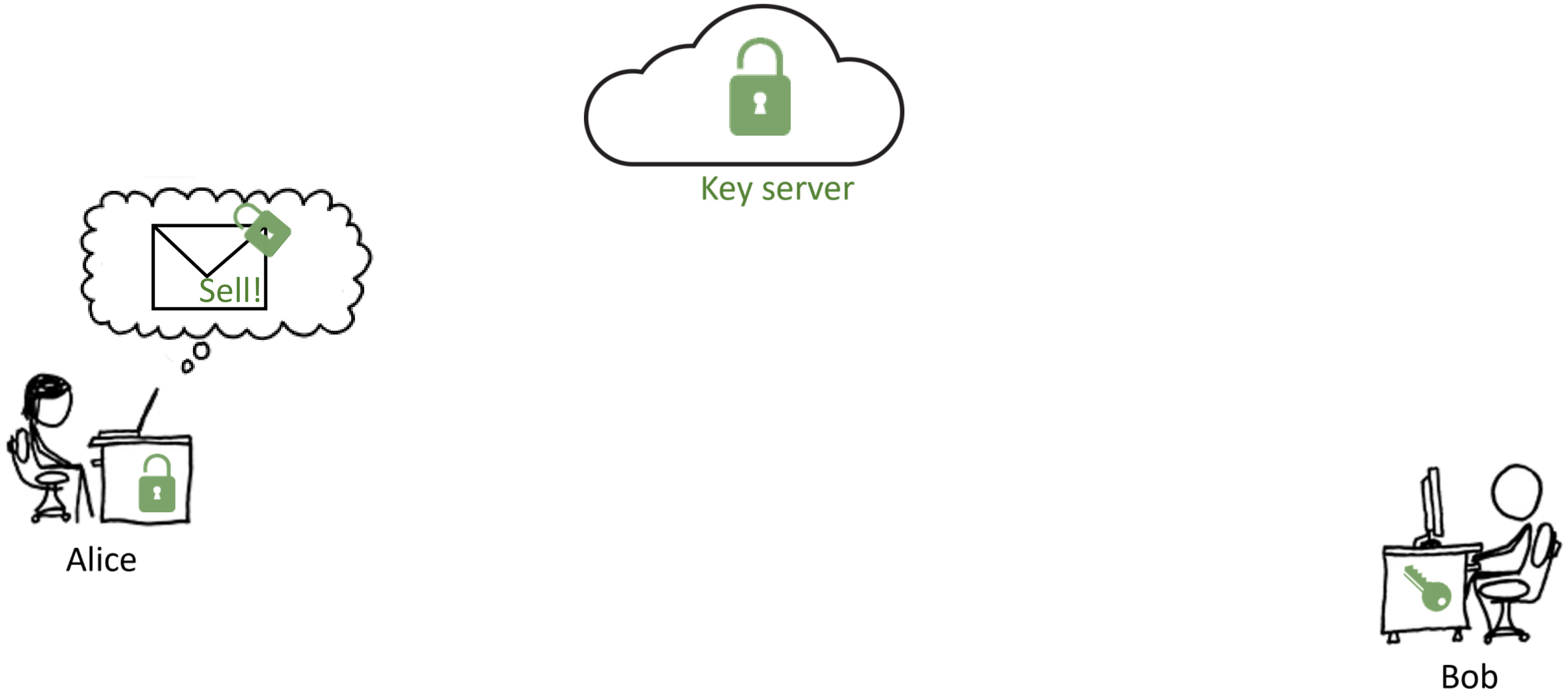


Image source: xkcd.com

Public-key encryption (PKE)

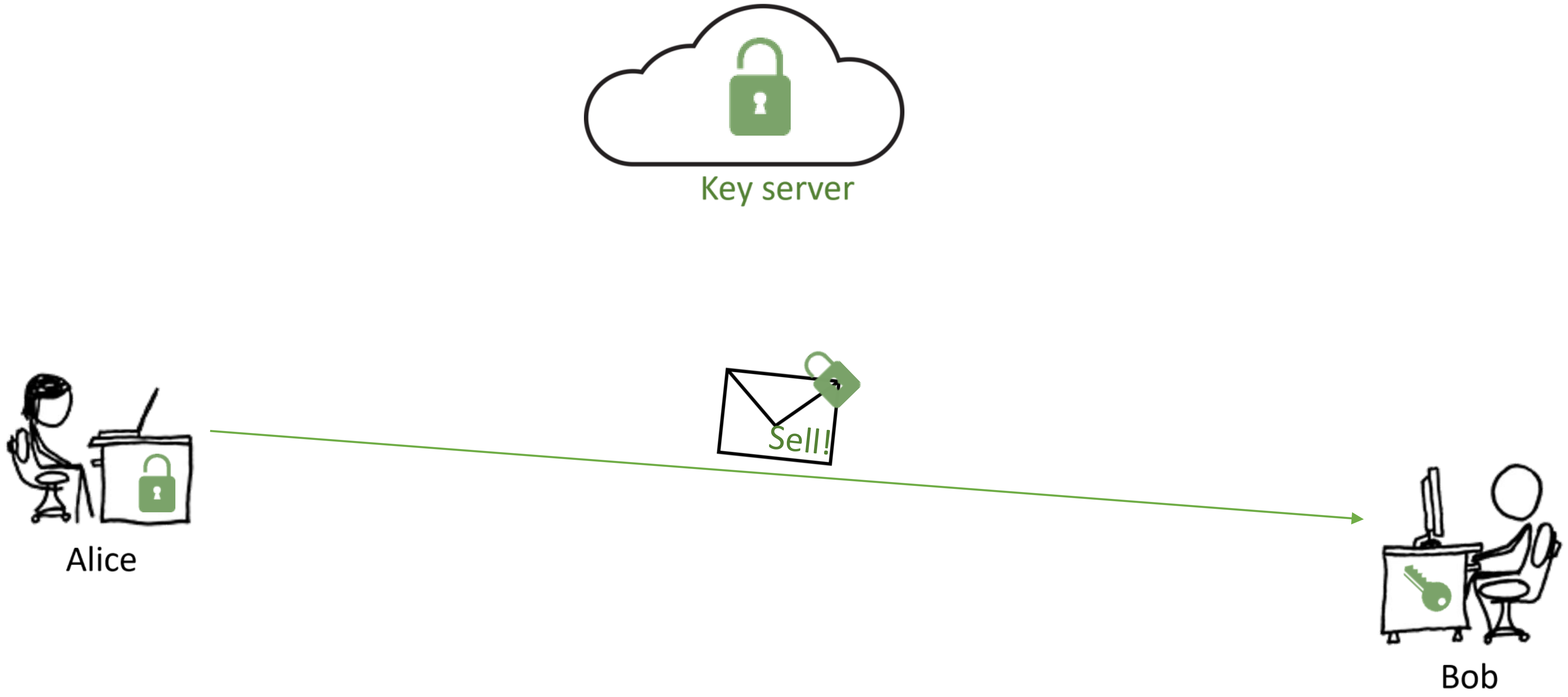


Image source: xkcd.com

Public-key encryption (PKE)

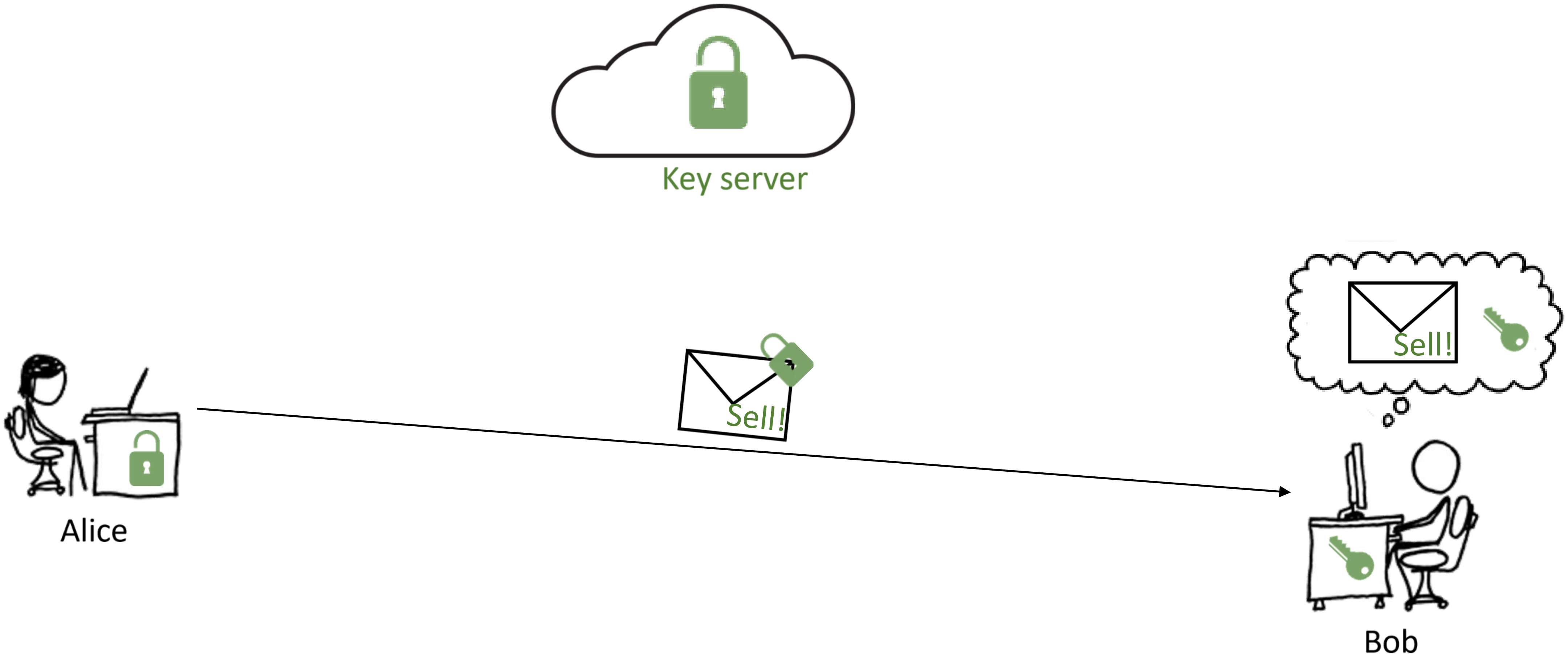


Image source: xkcd.com

Security definitions

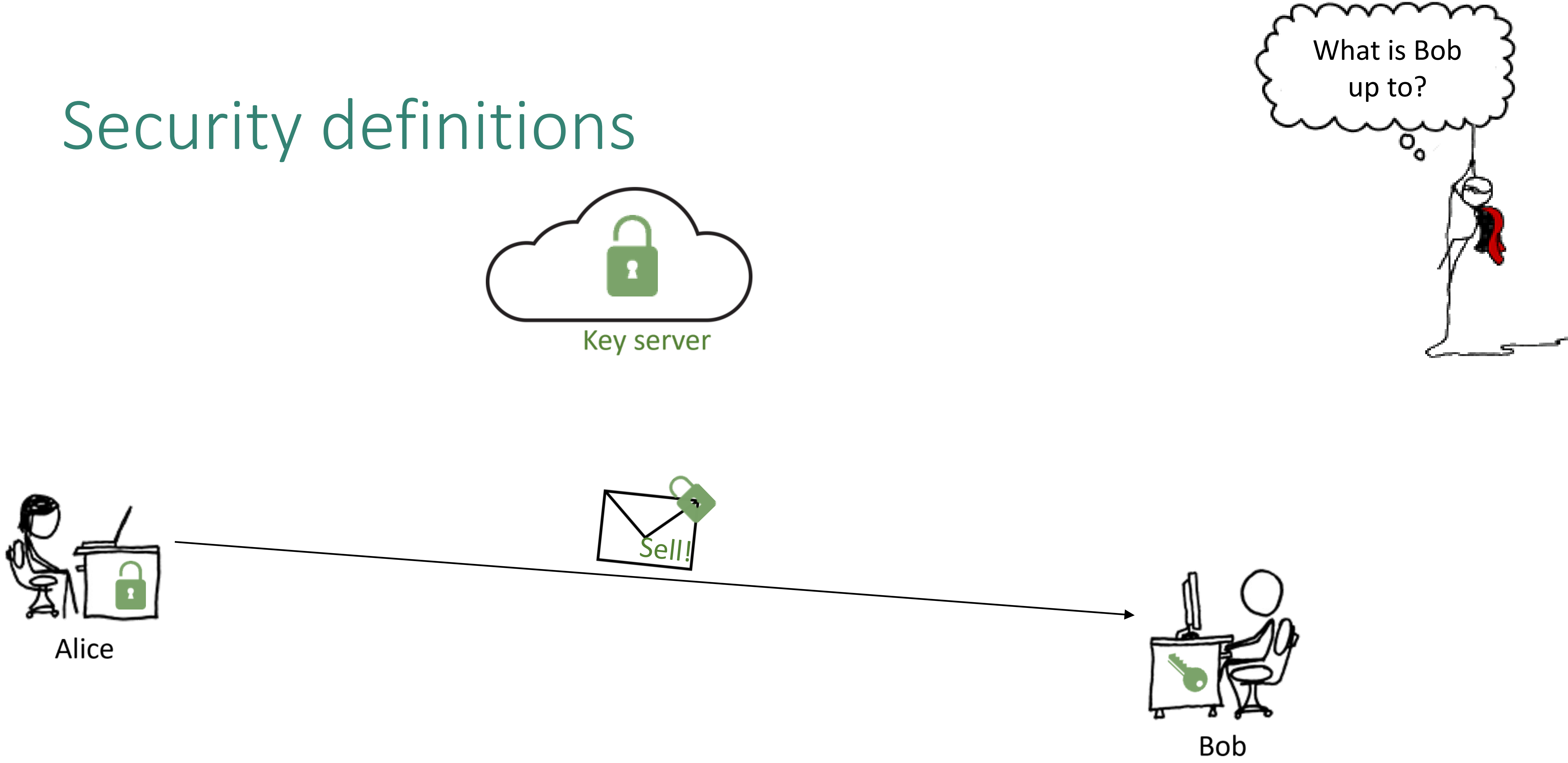


Image source: xkcd.com

Security definitions

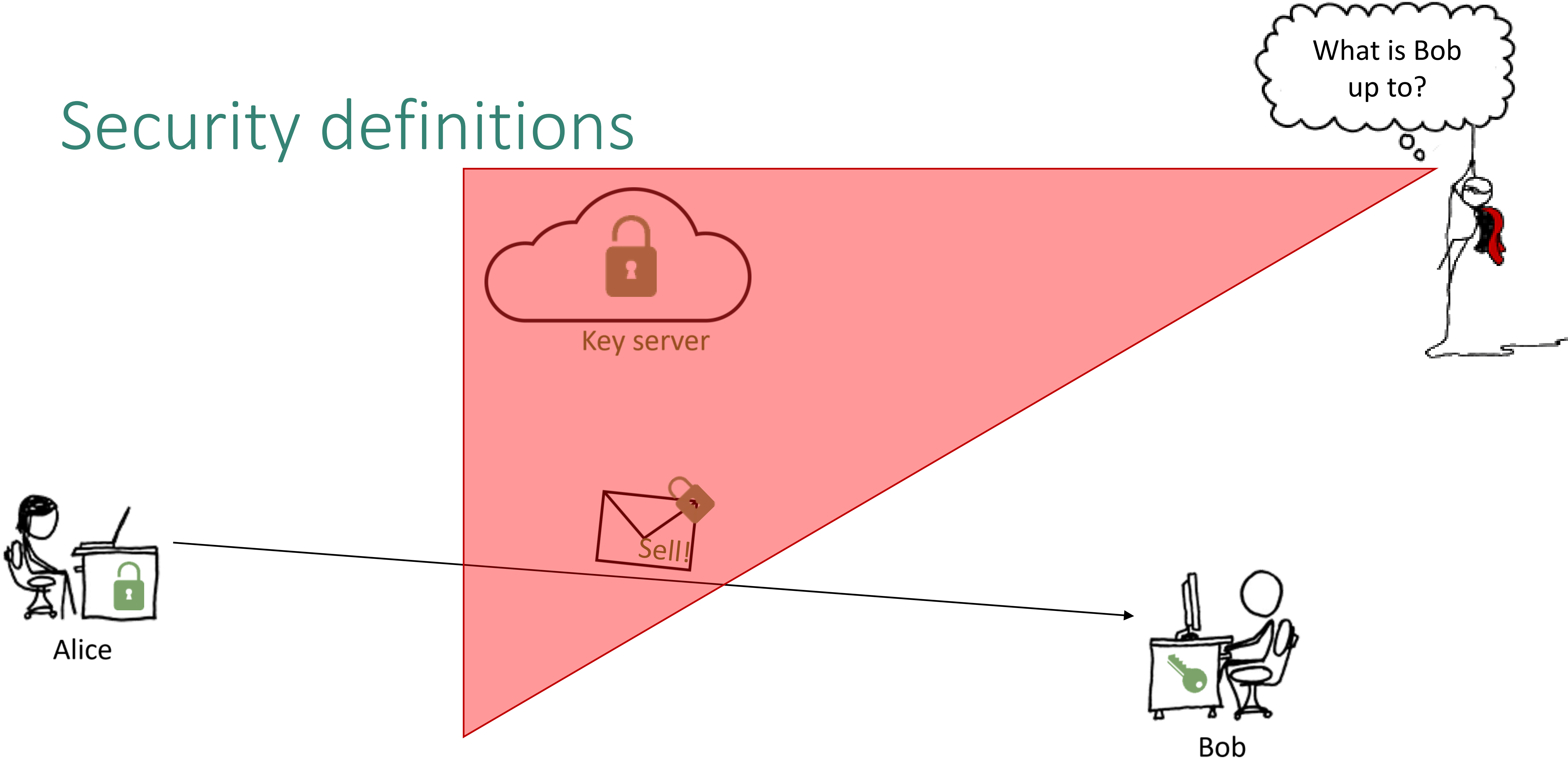


Image source: xkcd.com

Ciphertext indistinguishability

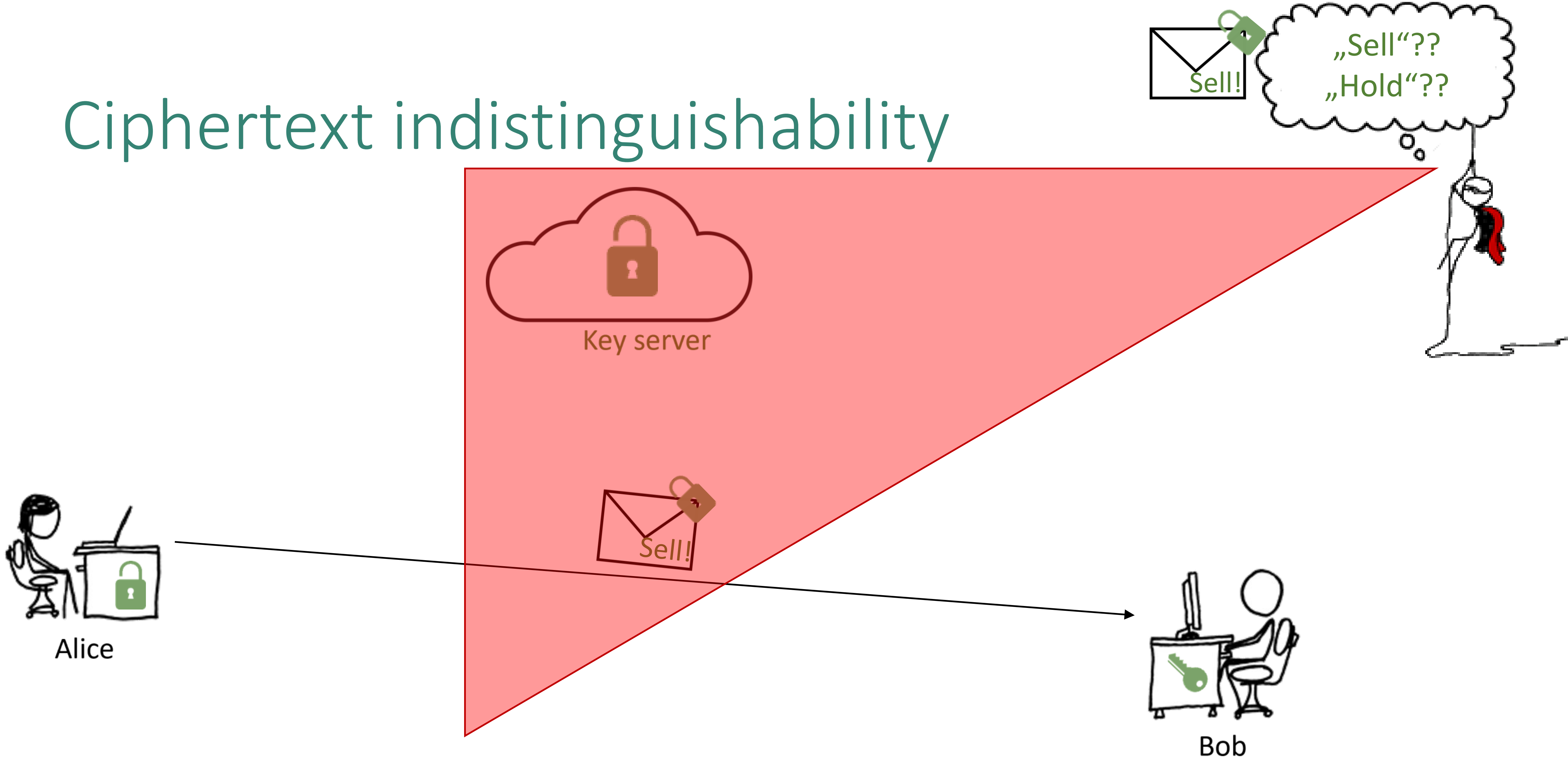
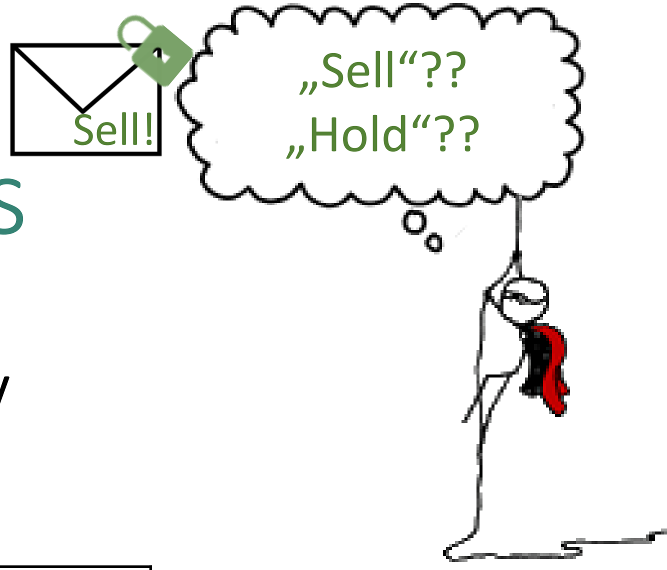


Image source: xkcd.com

Ciphertext indistinguishability games



Indistinguishability under **c**hosen-**p**laintext **a**ttacks = public key version of symmetric-key IND-CPA:

Left game	**Right game**
Adversary gets public key Adversary picks two messages m1 and m2 Adversary gets encryption of:	
m1	m2
Adversary guesses which game it’s playing	

Question: Can we have IND-CPA security if encryption is deterministic*?

* = encrypting a message m always gives the same result

Image source: xkcd.com

Ciphertext indistinguishability games

Indistinguishability under **c**hosen-**p**laintext **a**ttacks = public key version of symmetric-key IND-CPA:

<table>
<tr><th>Left game</th><th>Right game</th></tr>
<tr><td colspan="2">Adversary gets public key
Adversary picks two messages m1 and m2
Adversary gets encryption of:</td></tr>
<tr><td>m1</td><td>m2</td></tr>
<tr><td colspan="2">Adversary guesses which game it's playing</td></tr>
</table>

Question: Can we have IND-CPA security if encryption is deterministic*?

No, but encryption could still be hard to invert.

* = encrypting a message m always gives the same result

Image source: xkcd.com

PKE example: Schoolbook RSA

Alice

How could Alice encrypt ‚sell‘?

RSA: computations with primes!

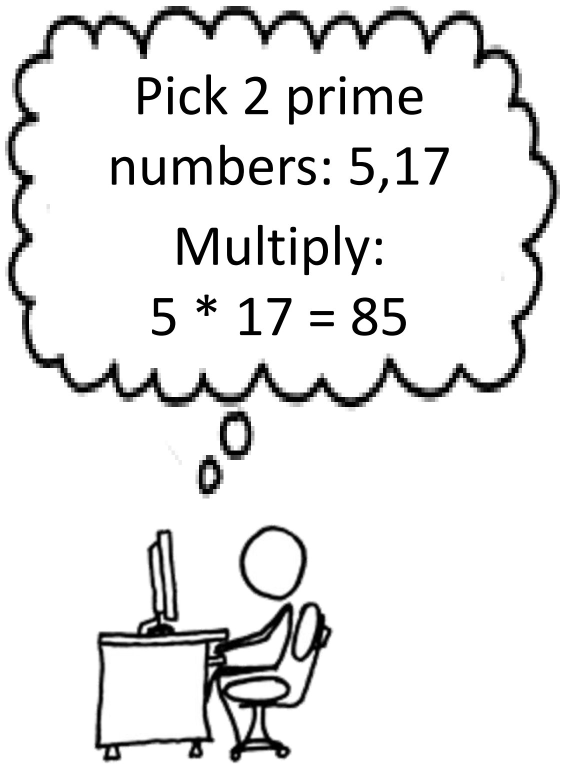
Schoolbook RSA = simplification of PKCS#1, the PKE scheme used in TLS’s predecessor.

Bob

Image source: xkcd.com

PKE example: Schoolbook RSA

Alice

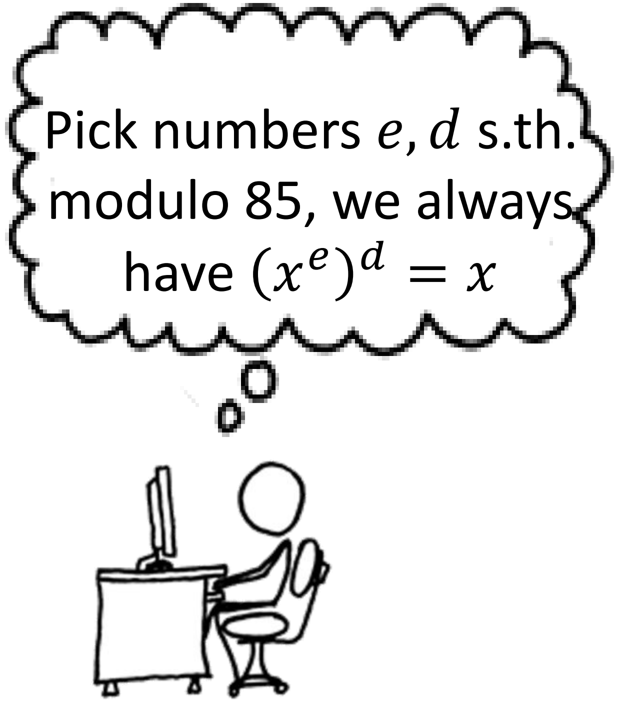


Bob

Image source: xkcd.com

PKE example: Schoolbook RSA

Alice



Bob

Image source: xkcd.com

PKE example: Schoolbook RSA

Alice

Example:

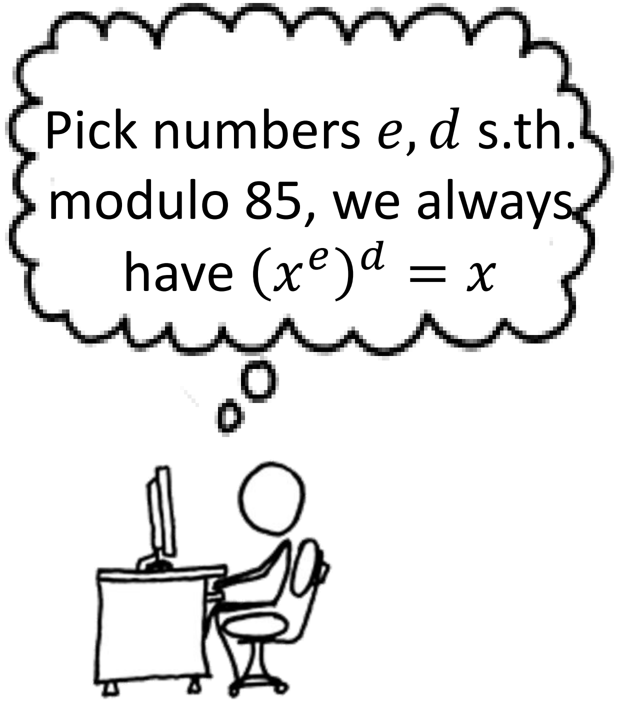
$e = 5, d = 13$

$x = 2$

$x^e = 2^5 = 32$

$(2^e)^d = 32^{13}$ (large, but has remainder 2!)

Also works for $x = 3, x = 4, x = 5, \ldots$



Bob

PKE example: Schoolbook RSA

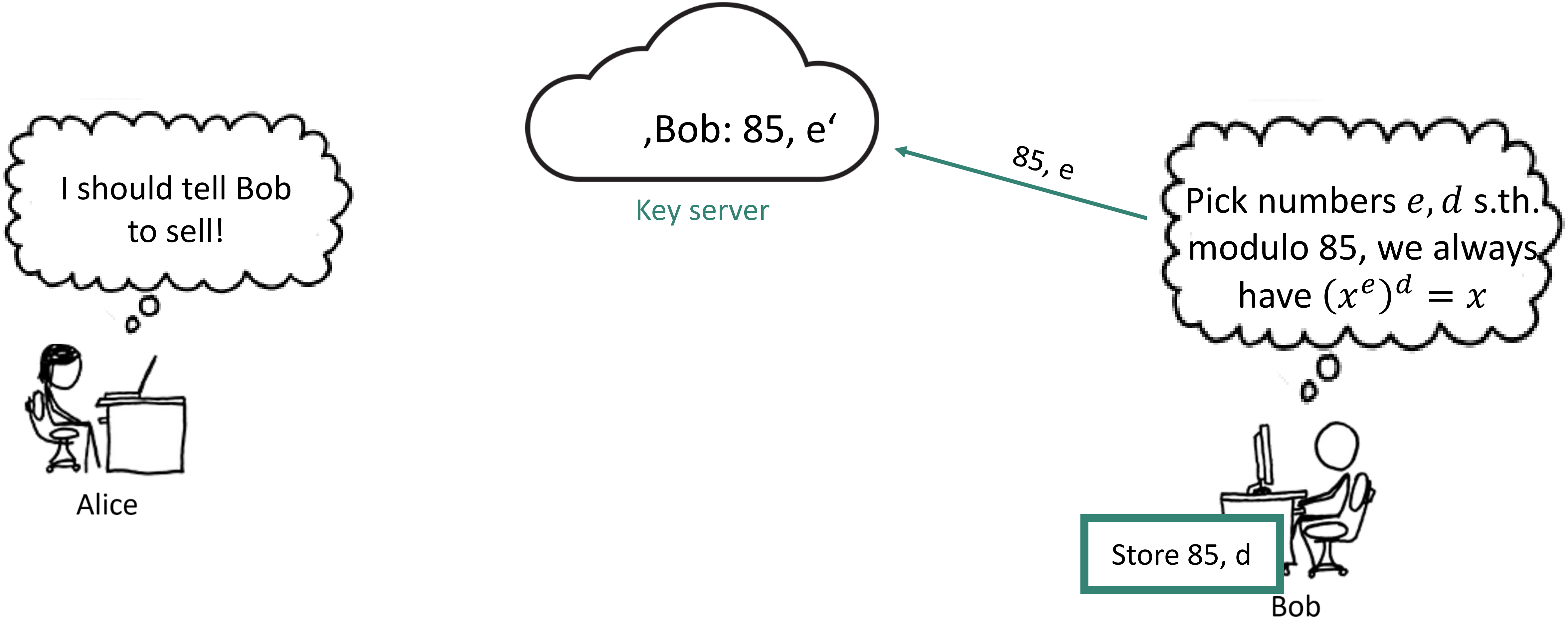


Image source: xkcd.com

PKE example: Schoolbook RSA

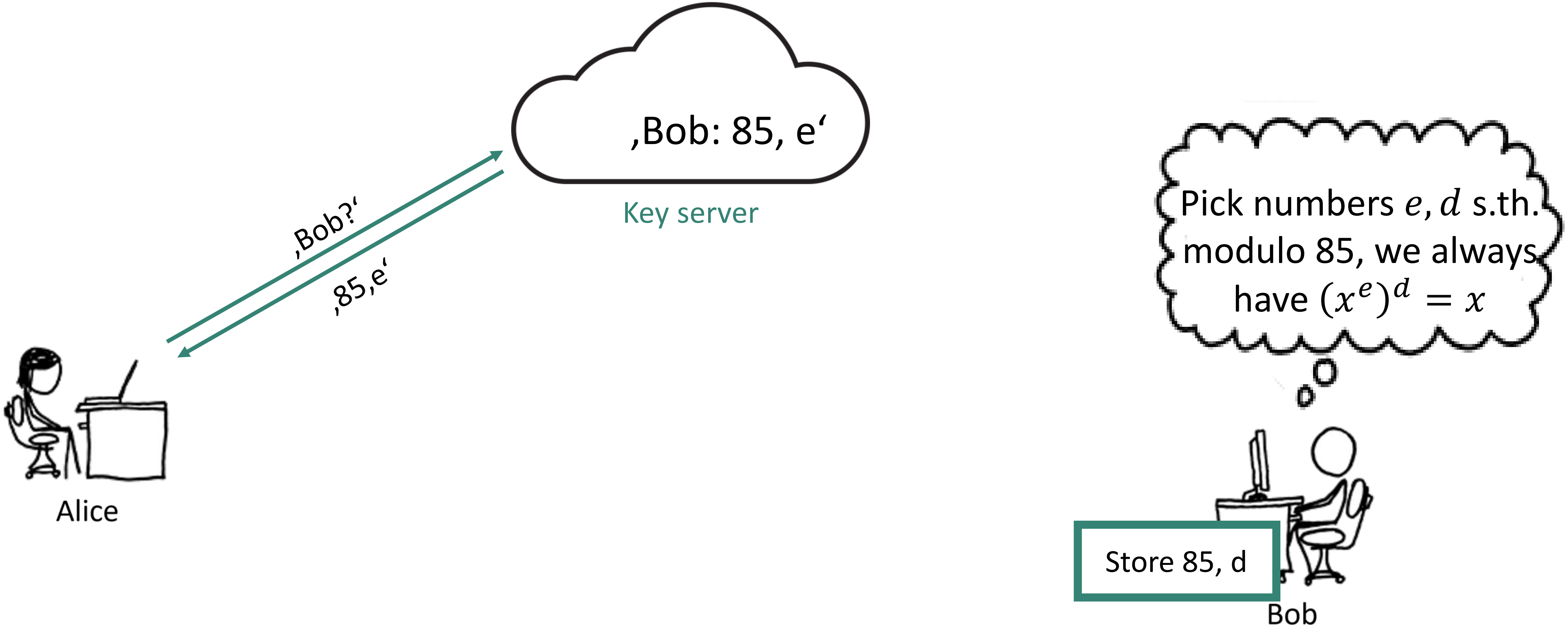


Image source: xkcd.com

PKE example: Schoolbook RSA

Use math:

Lock ‚Sell!‘ with 85, e.

Alice

The math:

Convert ‚Sell‘ into integer $m < 85$

Compute m^e

Divide by 85, keep the remainder

Use the remainder as ‚Sell!‘

Pick numbers e, d s.th. modulo 85, we always have $(x^e)^d = x$

Store 85, d

Bob

Image source: xkcd.com

PKE example: Schoolbook RSA

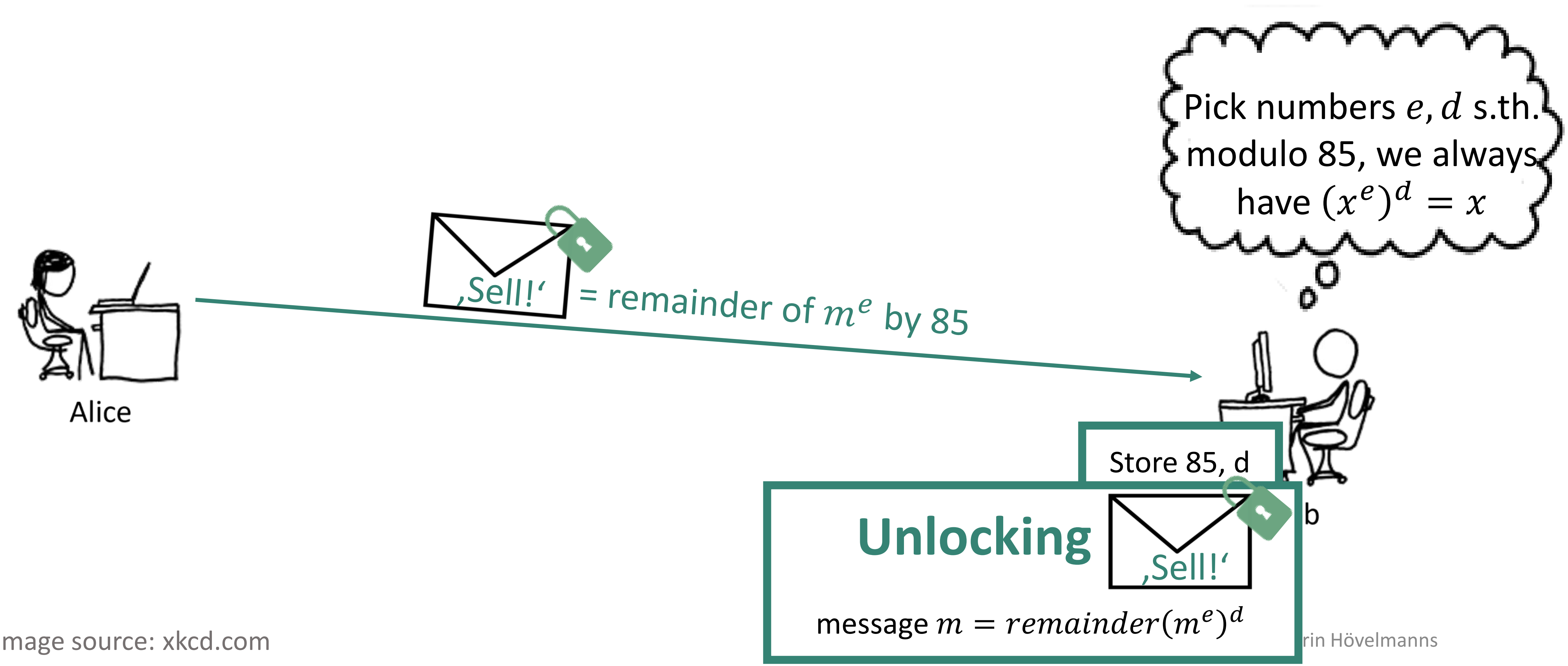


Image source: xkcd.com

Security intuition: RSA = trapdoor permutation

Like on the previous slides, we take

- as modulus N a prime product.
- e, d s.th. dividing $(x^e)^d$ by N always has remainder x.

$$\mathrm{RSA}_e\colon \{1,2,3,\cdots,N-1\} \to \{1,2,3,\cdots,N-1\}$$
$$x \mapsto x^e \bmod N$$

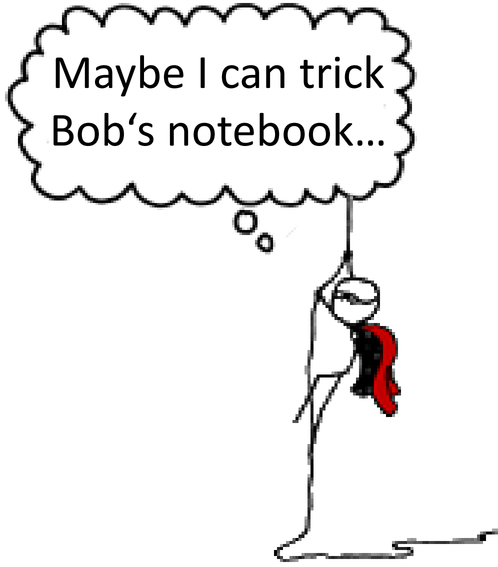
By choice of e and d, RSA_e is a permutation

So-called **trapdoor one-way** permutation: Computing x^e is easy, inverting is

- believed to be hard given only N and e (public key) $\leftarrow$ **if we chose the parameters appropriately (!)**
- easy given trapdoor d (the secret key)

⚠ RSA_e **may be hard to invert, but is deterministic** $\rightarrow$ **no IND-CPA security!**

⚠ **In practice, we need appropriate padding.**



Chosen-ciphertext attacks

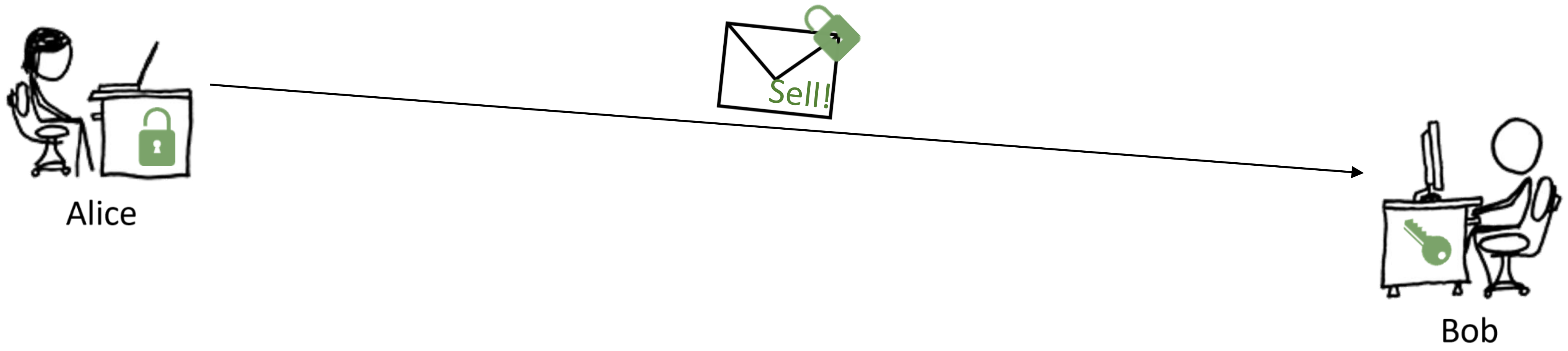
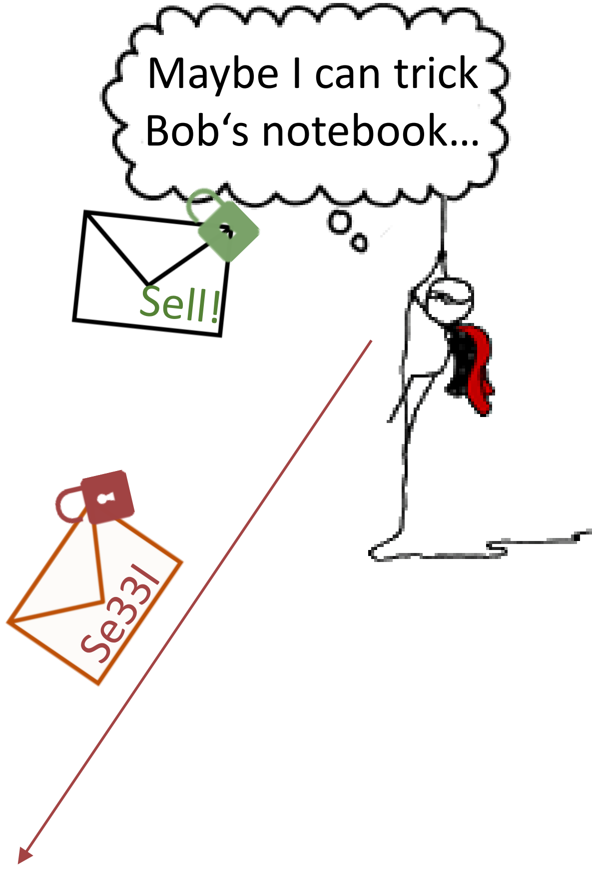


Image source: xkcd.com

Chosen-ciphertext attacks



Alice

Bob

Image source: xkcd.com

Chosen-ciphertext attacks

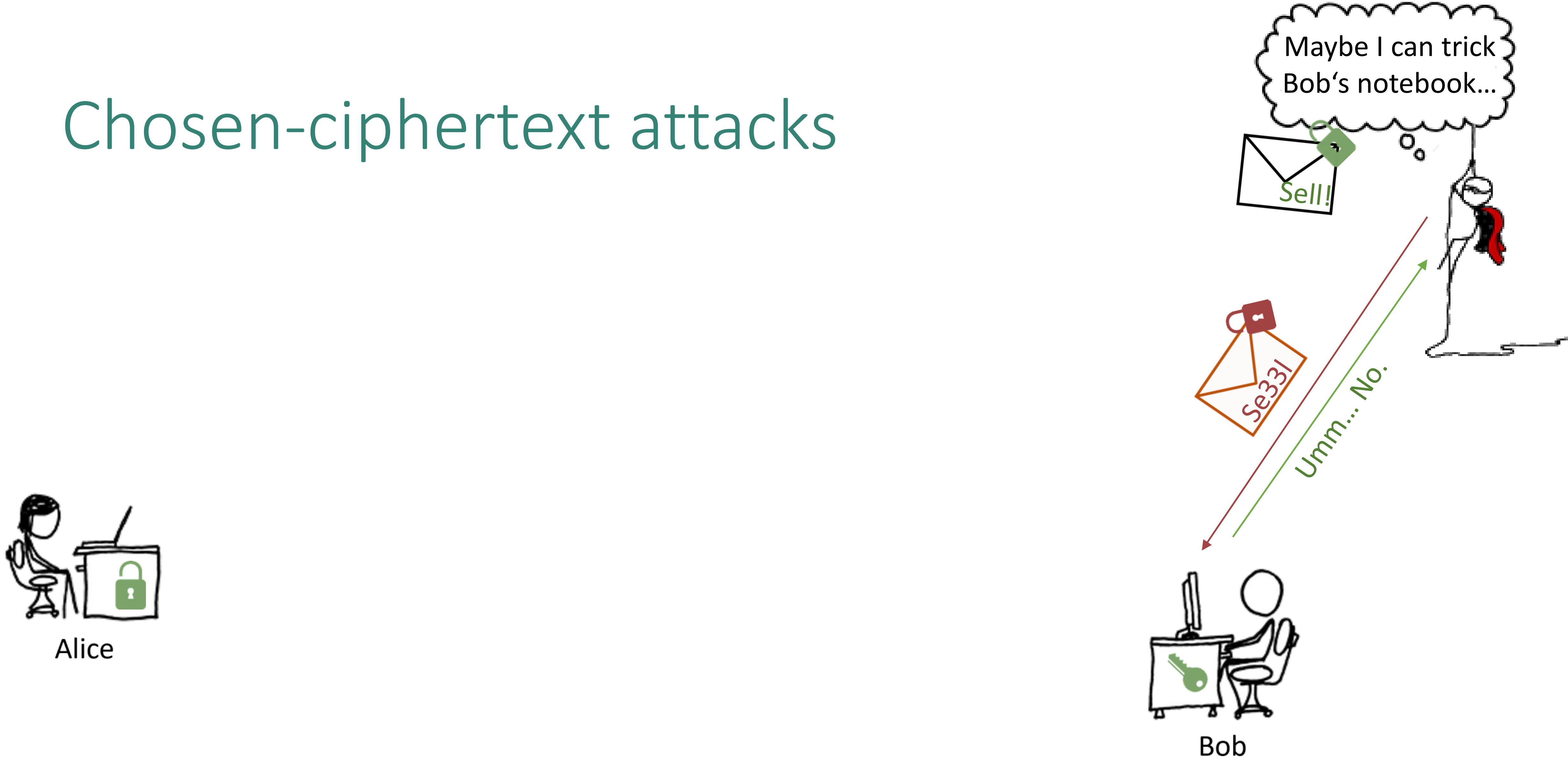


Image source: xkcd.com

Chosen-ciphertext attacks

Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1

Daniel Bleichenbacher

Bell Laboratories
700 Mountain Ave.
Murray Hill, NJ 07974
E-mail: bleichen@research.bell-labs.com

[Bleichenbacher 98]

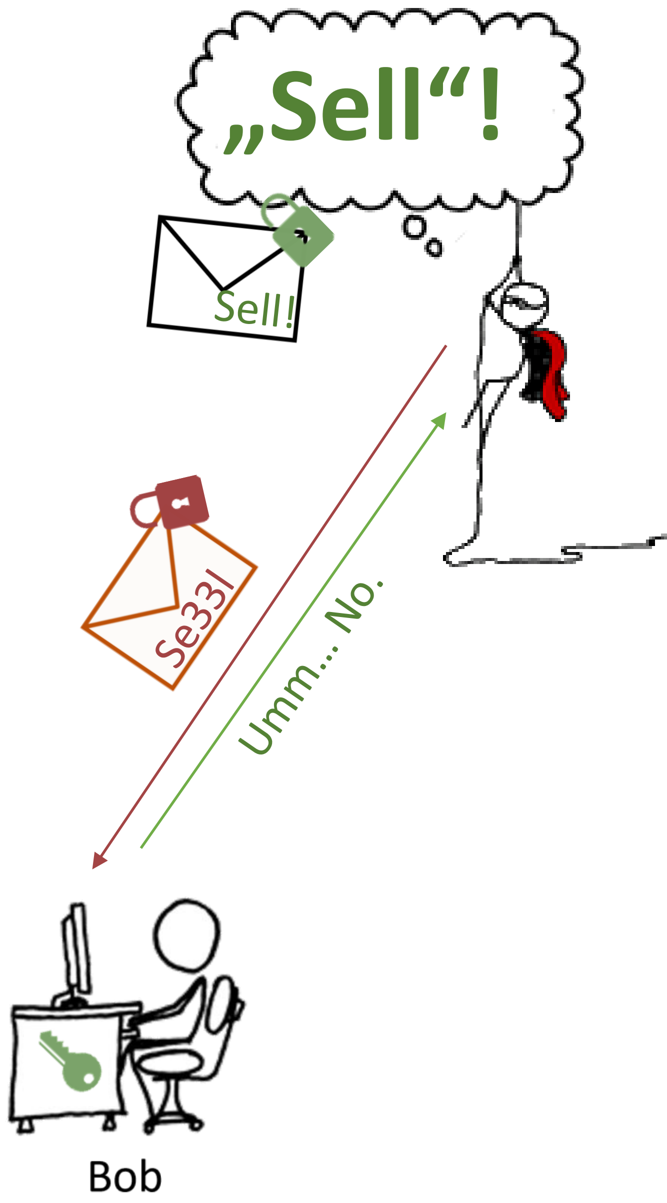
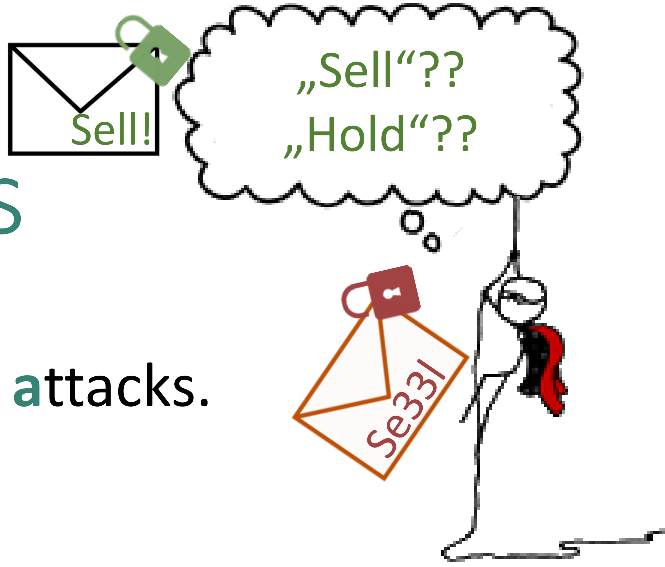


Image source: xkcd.com

Ciphertext indistinguishability games



IND-CCA security: **Ind**istinguishability under **c**hosen-**c**iphertext **a**ttacks.

Like IND-CPA:

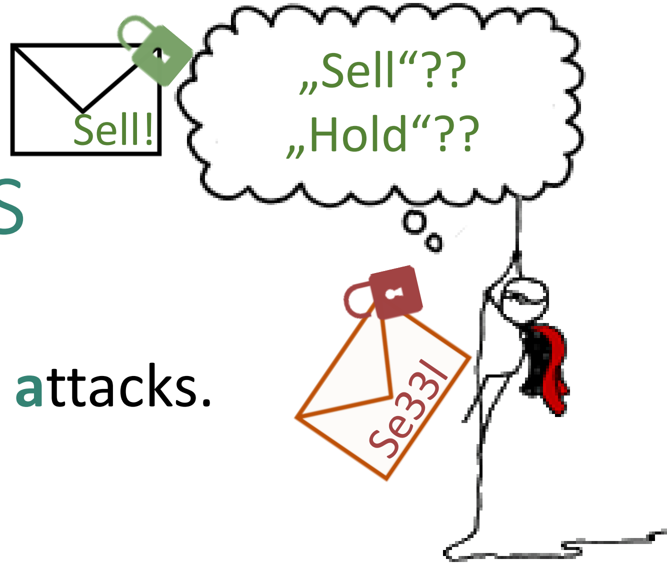
Left game	**Right game**
Adversary gets public key Adversary picks two messages m1 and m2 Adversary gets encryption of:	
m1	m2
Adversary guesses which game it's playing	

Difference to IND-CPA: Adversary can additionally request decryptions for any ciphertext is chooses…

Wait, can't this always be won?

Image source: xkcd.com

Ciphertext indistinguishability games



IND-CCA security: **Ind**istinguishability under **c**hosen-**c**iphertext **a**ttacks.

Like IND-CPA:

<table>
<tr><th>Left game</th><th>Right game</th></tr>
<tr><td colspan="2">Adversary gets public key
Adversary picks two messages m1 and m2
Adversary gets encryption of:</td></tr>
<tr><td>m1</td><td>m2</td></tr>
<tr><td colspan="2">Adversary guesses which game it’s playing</td></tr>
</table>

Difference to IND-CPA: Adversary can additionally request decryptions for any ciphertext is chooses… except the provided encryption of m1/m2

Image source: xkcd.com

Back to what we wanted

Goal: Find a public-key method to securely establish symmetric keys K_{sym}.

(Why not just use PKE to send encrypted messages? Efficiency.)

This is called a **K**ey **E**ncapsulation **M**echanism (KEM).

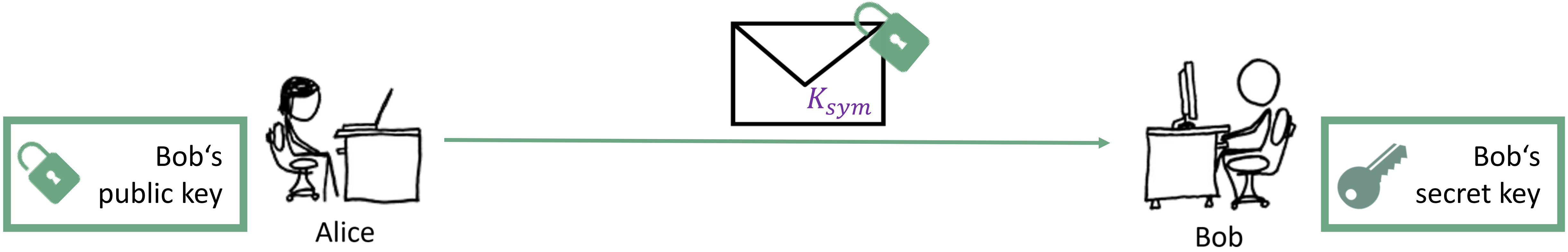


Image source: xkcd.com

Key Encapsulation Mechanisms (KEMs)

A KEM consists of 3 Algorithms:

1. **KeyGen**: Outputs a public/secret key pair (pk, sk)
2. **Encapsulate**(pk): Uses pk to create K_{sym} and a ciphertext c
3. **Decapsulate**(sk, c): Uses sk to recreate K_{sym} from c

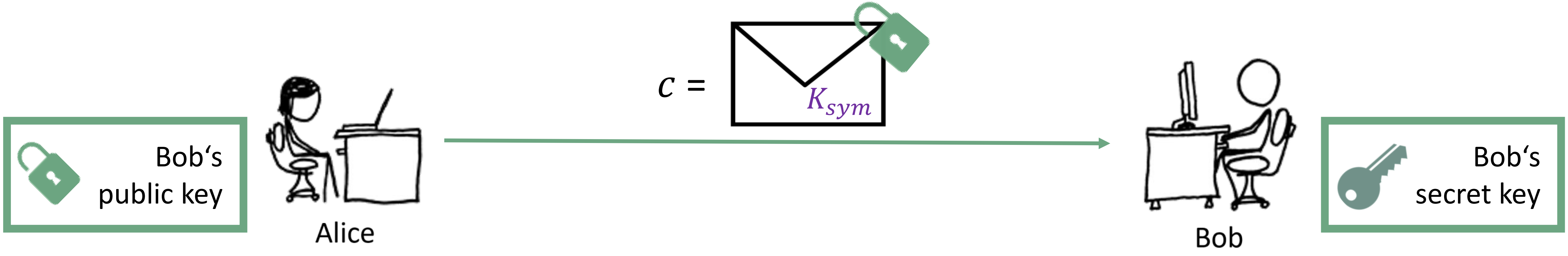
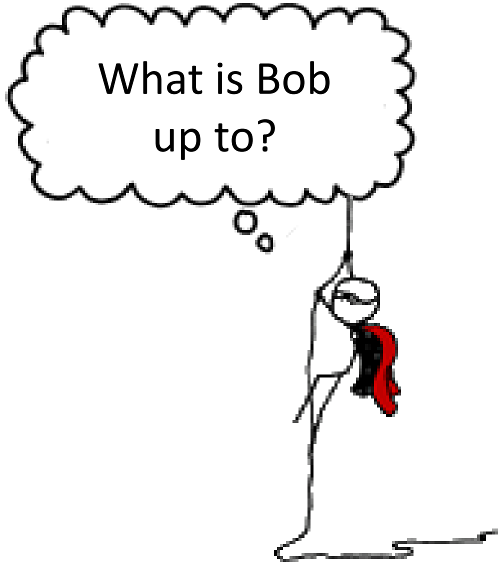


Image source: xkcd.com

KEMs: Security definition



A ciphertext c shouldn't leak substantial information about K_{sym}.

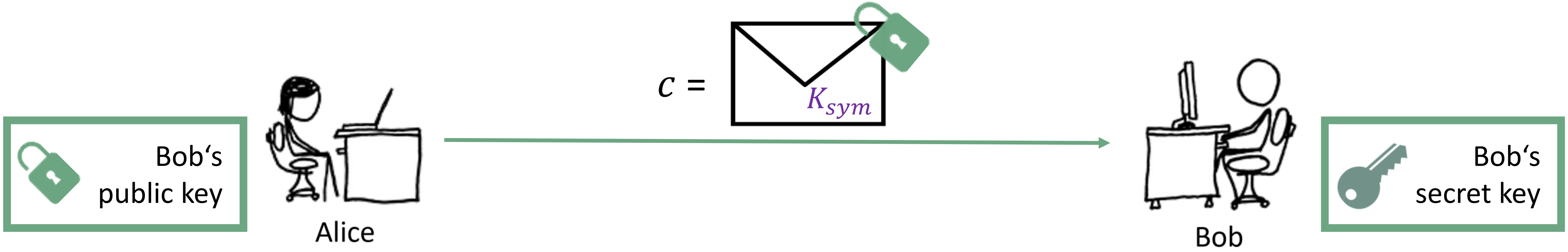
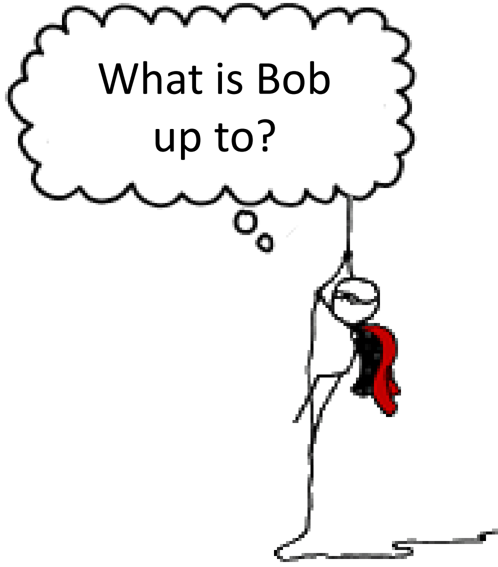


Image source: xkcd.com

Indistinguishability game for KEMs



IND-CPA-KEM security: **Ind**istinguishability for KEMs.

Left game	**Right game**
Adversary gets public key Adversary gets ciphertext c computed via Encapsulate, together with	
The K_{sym} that accompanied c	A uniformly random K_{sym}
Adversary guesses which game it's playing	

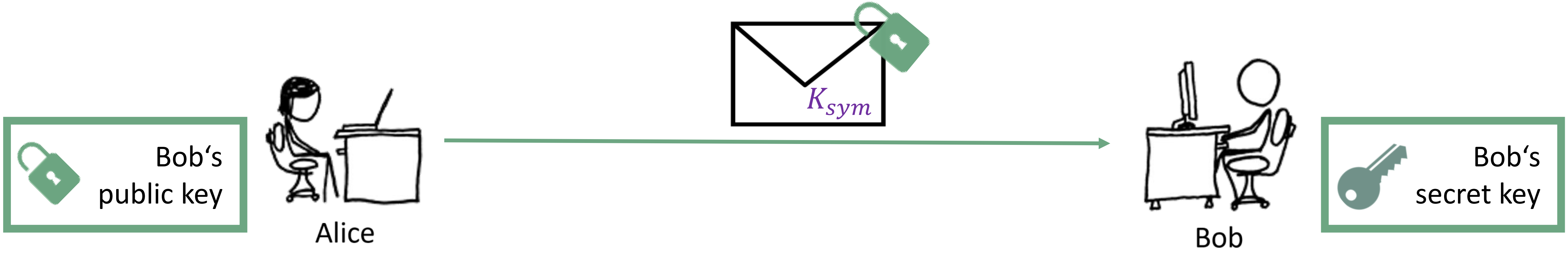
Image source: xkcd.com

KEMs in practice: NIST ‘competition’

Shared approach: PKE from hardness assumption + Fujisaki-Okamoto ‘recipe’

Fujisaki-Okamoto (FO) :

- ‘generic’ encryption-to-key-encapsulation recipe
- KYBER = moduleLWE encryption, plugged into FO recipe



Fujisaki-Okamoto KEMs: initial idea

Goal: Find a way to establish symmetric keys K_{sym}, securely.

You may use a public-key encryption scheme.

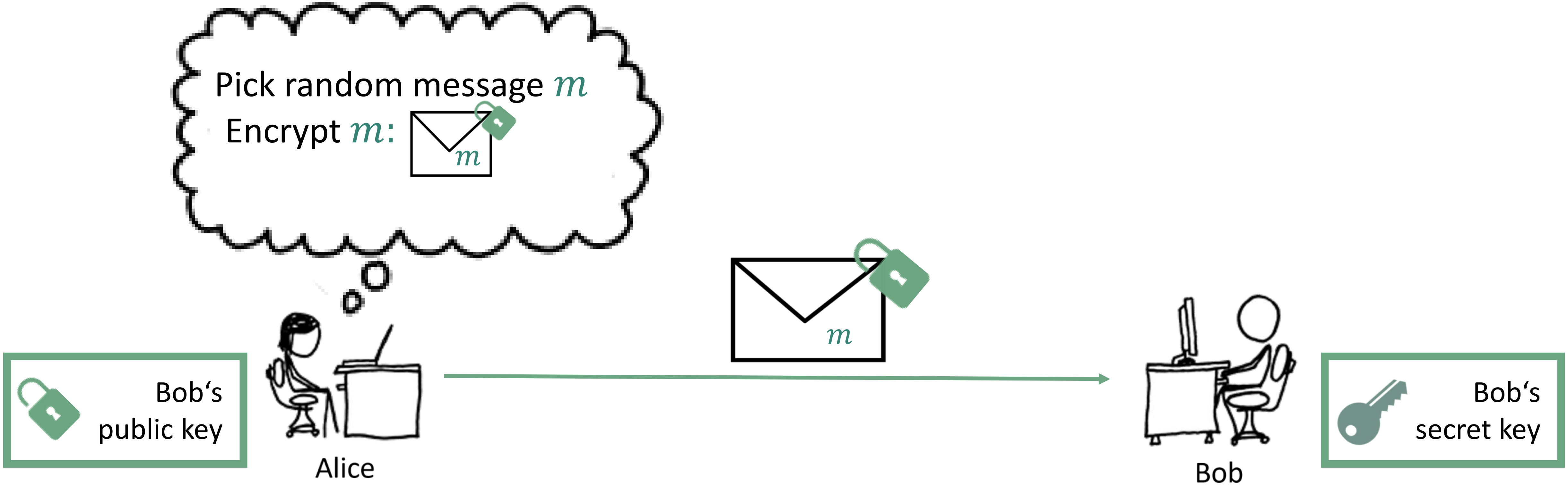


Image source: xkcd.com

Fujisaki-Okamoto KEMs: initial idea

Goal: Find a way to establish symmetric keys K_{sym}, se

You may use a public-key encryption scheme.

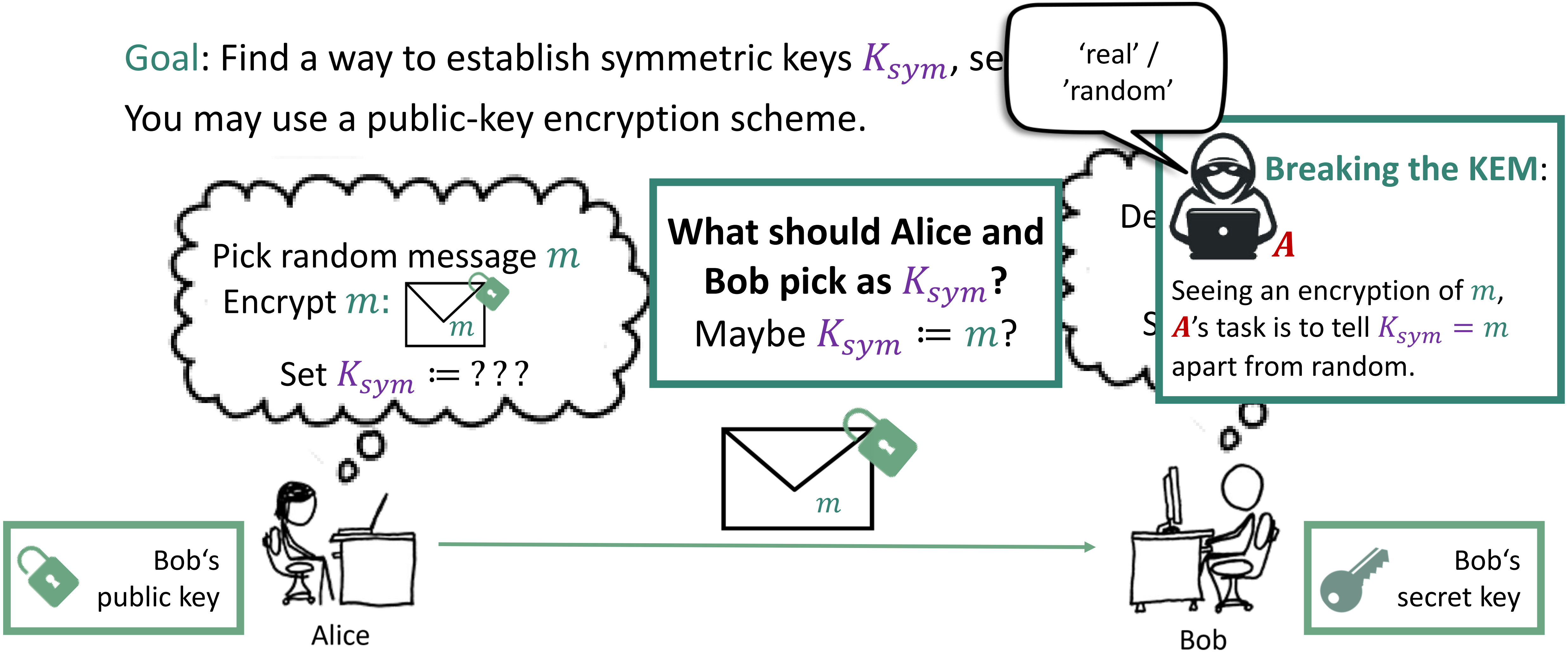


Image source: xkcd.com

Fujisaki-Okamoto KEMs: initial idea

Goal: Find a way to establish symmetric keys K_{sym}, securely.

You may use a public-key encryption scheme and a hash function.

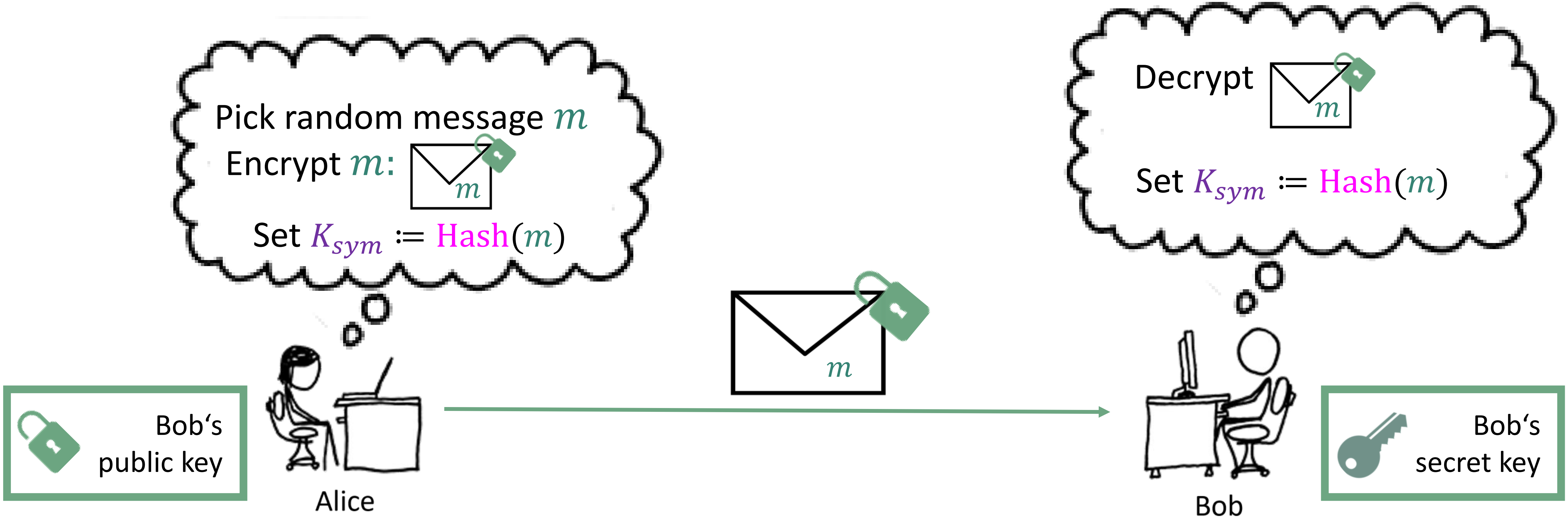


Image source: xkcd.com

Fujisaki-Okamoto K

Goal: Find a way to establish sym

You may use a public-key encryp

Q: Is this secure?

Proof heuristic:

<u>Assume</u> (!) Hash outputs are unpredictable + unrelated

→ A has 0 chance distinguishing without computing $\mathrm{Hash}(m)$ itself

… for which it needs to know m

… meaning it inverted encryption!

'real' / 'random'

Breaking the KEM:

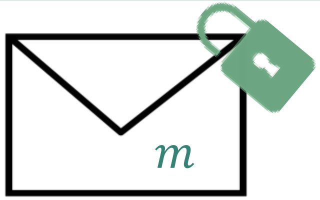
A

Seeing an encryption of m, A's task is to tell $K_{sym} = \mathrm{Hash}(m)$ apart from random.

Pick random message m

Encrypt m:

Set $K_{sym} := \mathrm{Hash}(m)$



Bob's public key

Alice

Bob

Bob's secret key

Image source: xkcd.com

Security against chosen-ciphertext attacks

Goal: Find a way to establish symmetric keys K_{sym} with **chosen-ciphertext** security.

→ attacker allowed to **request decapsulation for any ciphertext.**

Only high-level: slightly alter how the KEM en-/decapsulates:

Altered decapsulation will

- detect malicious ciphertexts
- punish those by rejecting to return a meaningful key.

→ hard for attacker to request useful decapsulations

It is still being researched today which altering strategy works best!

Take-aways

PKEs give us confidentiality (without secret meetings), KEMs make this more efficient.

We have a ‘cooking recipe’ for turning PKE into a KEM (called Fujisaki-Okamoto).

We used a ‚lego‘ approach very common in crypto:

Simple primitive (e.g., PKE) → Transformer (e.g., FO) → More complex primitive (e.g., KEM)

Q: how can we guarantee data authenticity/integrity?

Image source: xkcd.com

Digital signatures – a bit like MACs:

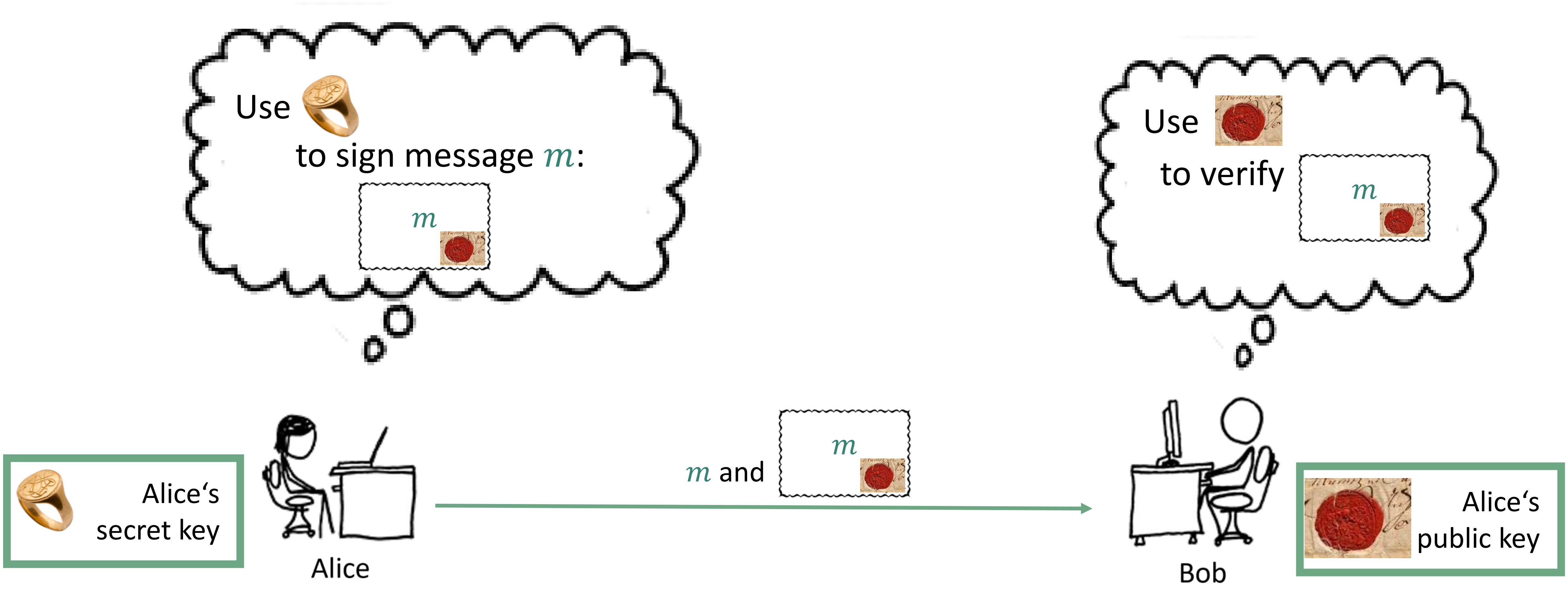


Image source: xkcd.com

Digital signatures: security goals

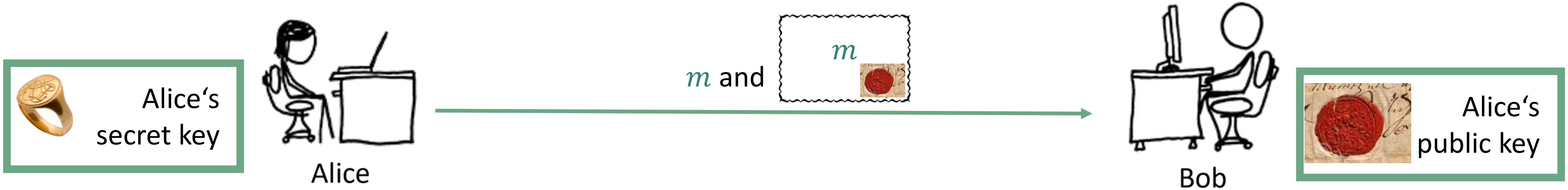
Security goal = UnForgeability: Computing a valid signature without knowing secret key sk is hard.

(Attackers will know the public key, though.)

- **UF against Chosen Message Attacks (UF-CMA):**

  even given the power to request signatures on chosen messages m_i,

  a valid signature for a new message $m' \neq m_i$ is hard to produce.



Digital signatures – a bit like MACs, but not fully:

Signatures:

MACs:

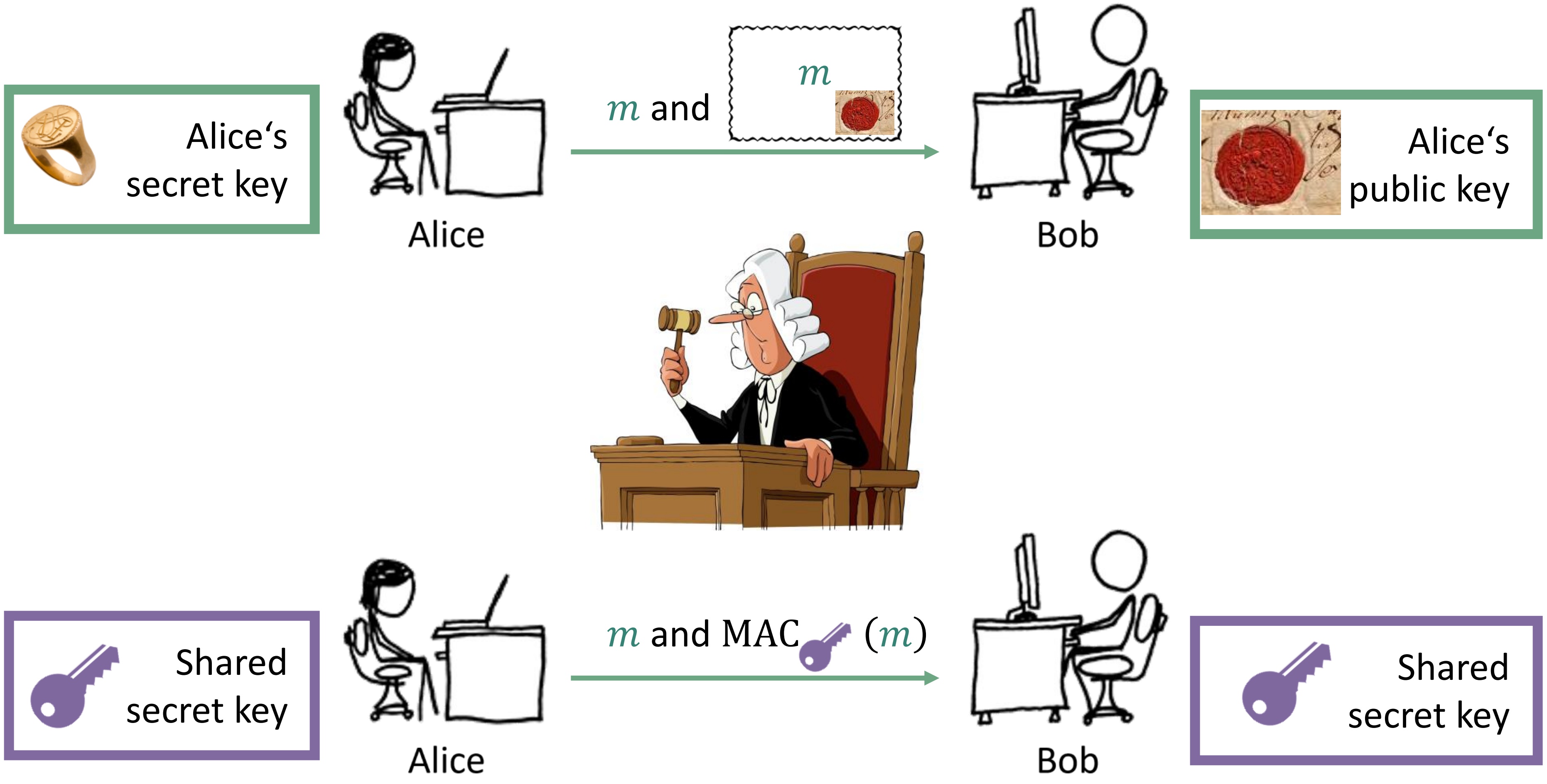


Image source: xkcd.com

Schoolbook RSA signatures

Remember RSA function: We take

- as modulus N a prime product.
- e, d s.th. dividing $(x^e)^d$ by N always has remainder x → RSA_e is a permutation:

$$\mathrm{RSA}_e\colon \{1, 2, 3, \cdots, N-1\} \to \{1, 2, 3, \cdots, N-1\}$$
$$x \mapsto x^e \bmod N$$

Like before, we set: public key  $=(N, e)$, secret key  $=\ d$:

Image source: xkcd.com

Schoolbook RSA signatures

Remember RSA function: We take

- as modulus N a prime product.
- e, d s.th. dividing $(x^e)^d$ by N always has remainder x → RSA_e is a permutation:

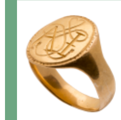
$$\mathrm{RSA}_e\colon \{1,2,3,\cdots,N-1\} \to \{1,2,3,\cdots,N-1\}$$
$$x \mapsto x^e \bmod N$$

Verify  :

Check $s^e = m \bmod N$

Sign message $m < N$:

$= m^d \bmod N$

 d

Alice

m and $s =$ 

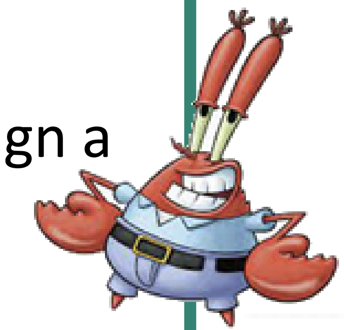
Bob

 (N, e)

Image source: xkcd.com

Q: Is this secure?

Can Mr. Krabs - only knowing the public key N, e, but not d – sign a message such that Bob accepts the signature?)



$$\mathrm{RSA}_e : \{1, 2, 3, \cdots, N-1\} \to \{1, 2, 3, \cdots, N-1\}$$
$$x \mapsto x^e \bmod N$$

Sign message $m < N$:

$\boxed{m} = m^d \bmod N$

Verify $\boxed{m}$:

Check $s^e = m \bmod N$

d

Alice

m and $s = \boxed{m}$

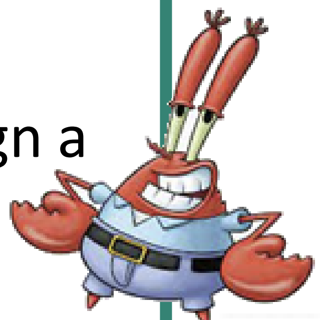
Bob

(N, e)

Image source: xkcd.com

Q: Is this secure?

Can Mr. Krabs - only knowing the public key N, e, but not d – sign a message such that Bob accepts the signature?)



Key-only forgery: Pick arbitrary ‘signature’ s, set $m = s^e \bmod N$

$\rightarrow$ s is a valid signature for m that will be accepted by Bob!

In practice, however, m might look unconvincing to the recipient.

Sign message $m < N$:

m $= m^d \bmod N$

$$\mathrm{RSA}_e\colon \{1,2,3,\cdots,N-1\} \rightarrow \{1,2,3,\cdots,N-1\}$$
$$x \mapsto x^e \bmod N$$

Verify m :

Check $s^e = m \bmod N$

d

Alice

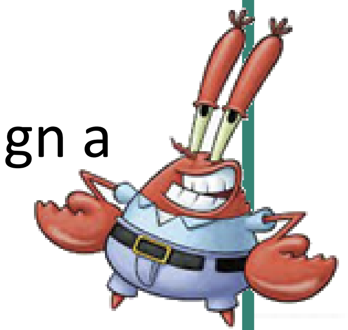
m and $s =$ m

Bob

(N, e)

Image source: xkcd.com

Q: Is this secure?



Can Mr. Krabs - only knowing the public key N, e, but not d – sign a message such that Bob accepts the signature?)

Targetted forgery via signature requests: Choose target message m^*.

We'll exploit the multiplicative property of the RSA function ('verification preserves multiplication'):

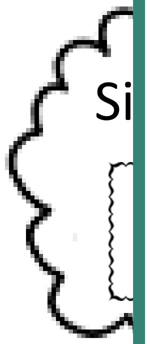
$$(s_1 \cdot s_2)^e = s_1^e \cdot s_2^e \bmod N$$

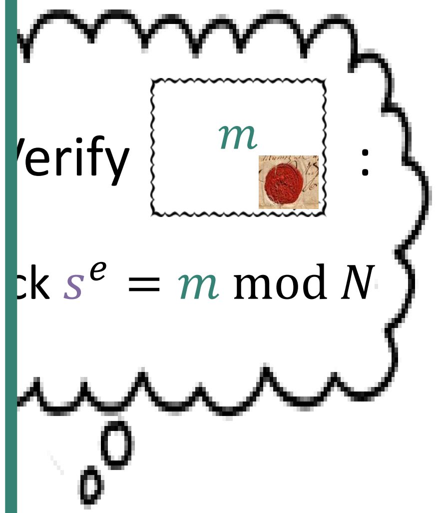
Attack:

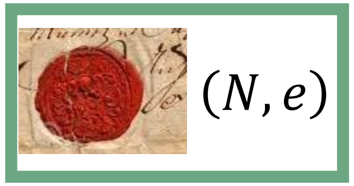
- Pick arbitrary message m_1, and m_1^{-1} such that $m_1 m_1^{-1} \bmod N = 1$.
- Request signature s_1 for m_1: you get $s_1 = m_1^d$
  and signature s_2 for $m_2 = m_1^{-1} \cdot m^*$: you get $s_2 = m_2^d$

Sign m^* with $\mathrm{s}^* = s_1 \cdot s_2 \rightarrow$ Bob accepts since $(s^*)^e = m^* \bmod N$:

$$(s^*)^e = s_1^e \cdot s_2^e = m_1 \cdot m_2 = m_1 \cdot m_1^{-1} \cdot m^* = m^* \bmod N$$







Alice

Bob

Image source: xkcd.com

Q: Can we tweak this so it becomes secure?

Idea: Pick hash function Hash: $\{0,1\}^* \to \{1,2,3,\cdots,N-1\}$, sign messages $m \in \{0,1\}^*$ by applying RSA signature approach to $\mathrm{Hash}(m)$.

Advantage 1: We can now sign arbitrary-length messages.

Advantage 2: Targetted attack a lot harder: need to find m, m_1, m_2 such that

$$\mathrm{Hash}(m) = \mathrm{Hash}(m_1) \cdot \mathrm{Hash}(m_2) \bmod N$$

Verify m :

$s^e = \mathrm{Hash}(m) \bmod N$?

$$\mathrm{RSA}_e\colon \{1,2,3,\cdots,N-1\} \to \{1,2,3,\cdots,N-1\}$$

$$x \mapsto x^e \bmod N$$

Sign message $m < N$:

m $= \mathrm{Hash}(m)^d \bmod N$

d

Alice

m and $s =$ m

Bob

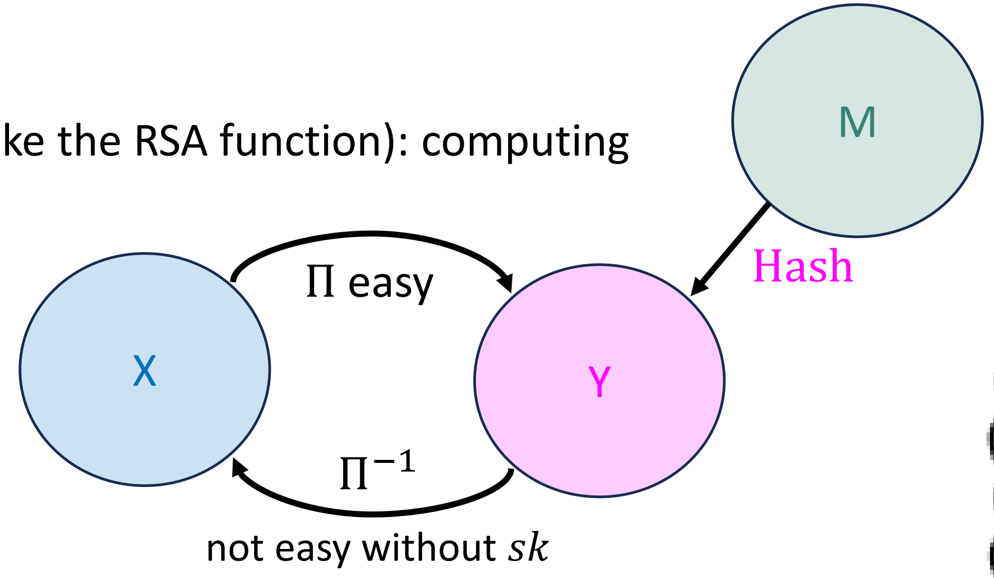
(N, e)

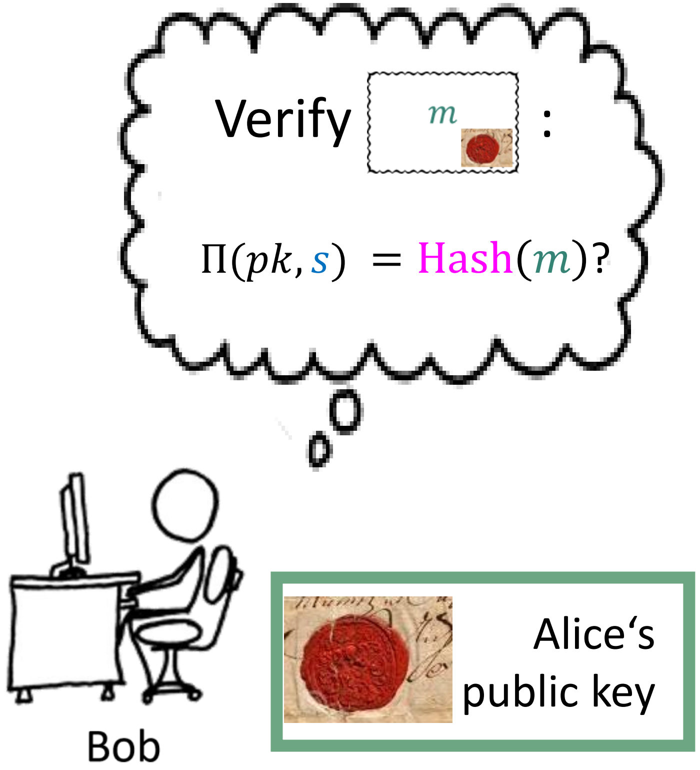
Image source: xkcd.com

Abstraction of tweak : full domain hash (FDH)

Take trapdoor one-way permutation Π (like the RSA function): computing

- $\Pi(pk, x)$ is easy (e.g., x^e)
- $\Pi^{-1}(sk, y)$ is (e.g., y^d)
  - hard when not knowing sk
  - easy when knowing sk





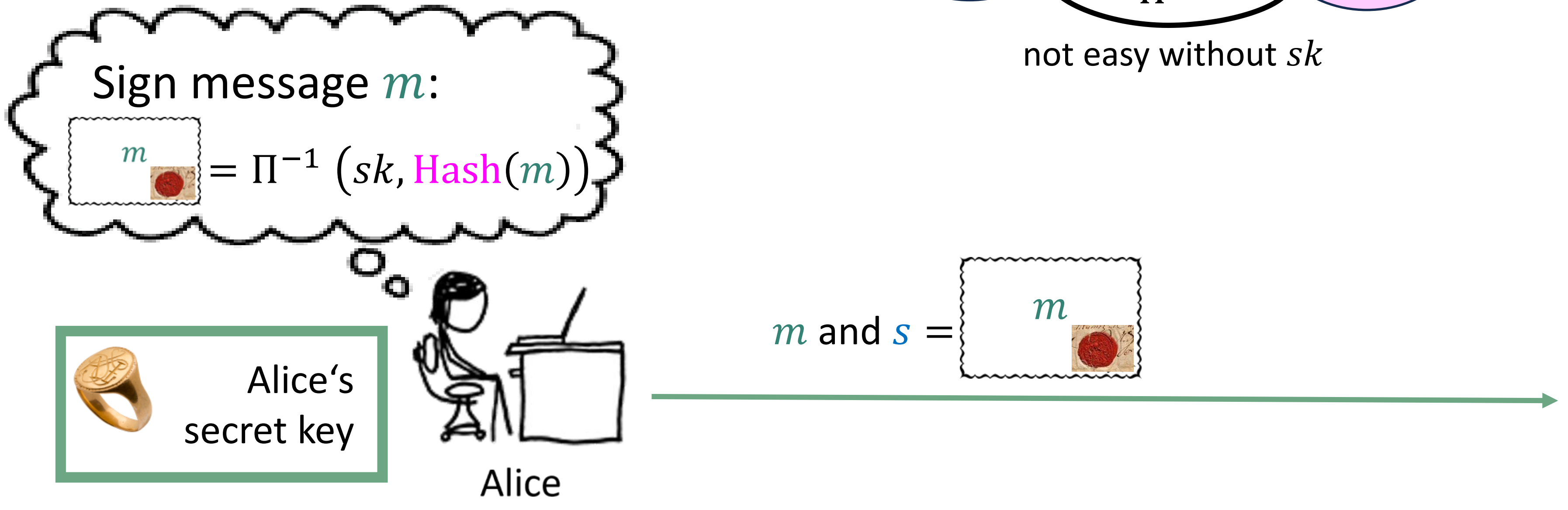


Image source: xkcd.com

Approach based on identification schemes

Hey Alice, is
this really you?

Alice's
secret key

Alice

Bob

Alice's
public key

Image source: xkcd.com

Approach based on identification schemes

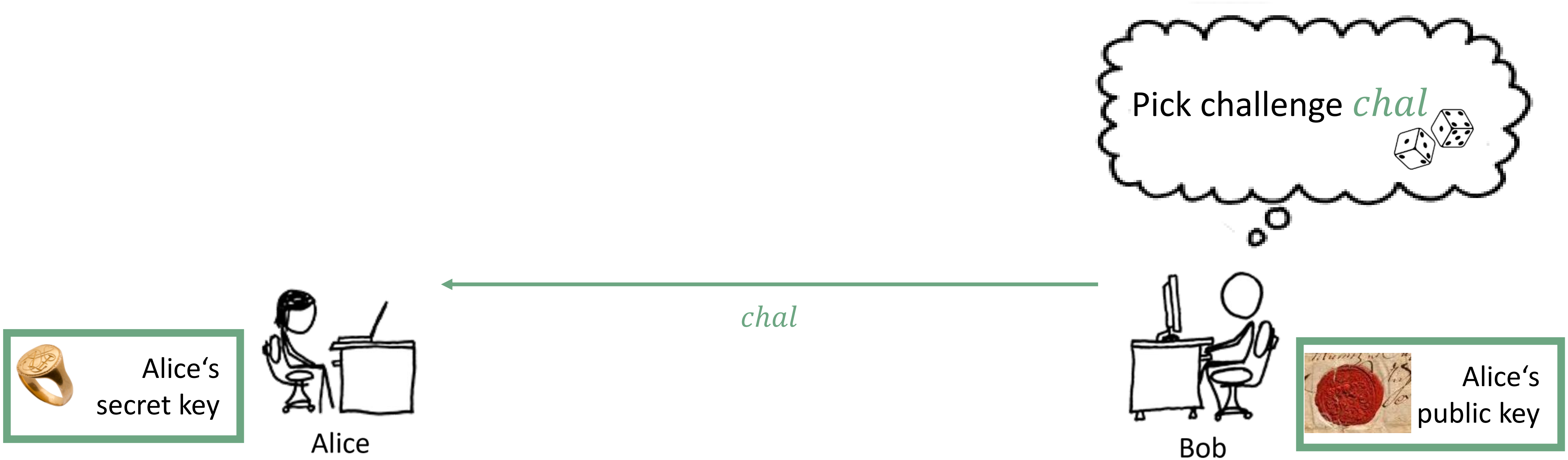


Image source: xkcd.com

Approach based on identification schemes

Image source: xkcd.com

Approach based on identification schemes

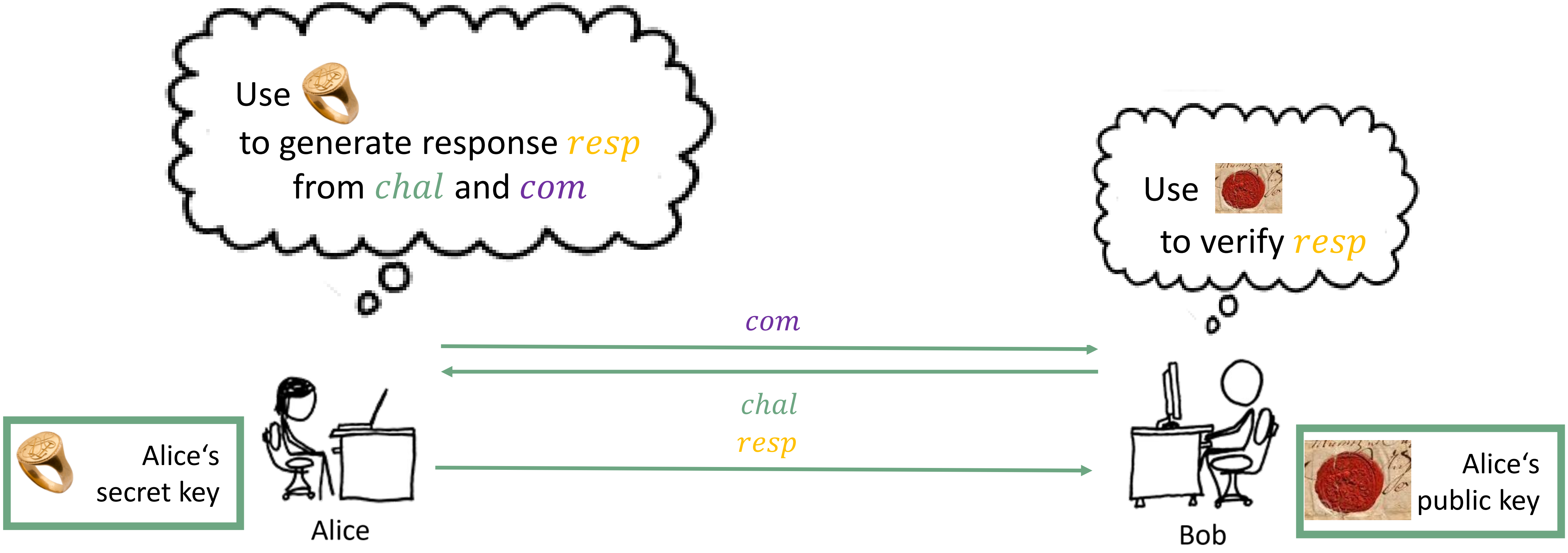


Image source: xkcd.com

Approach based on identification schemes

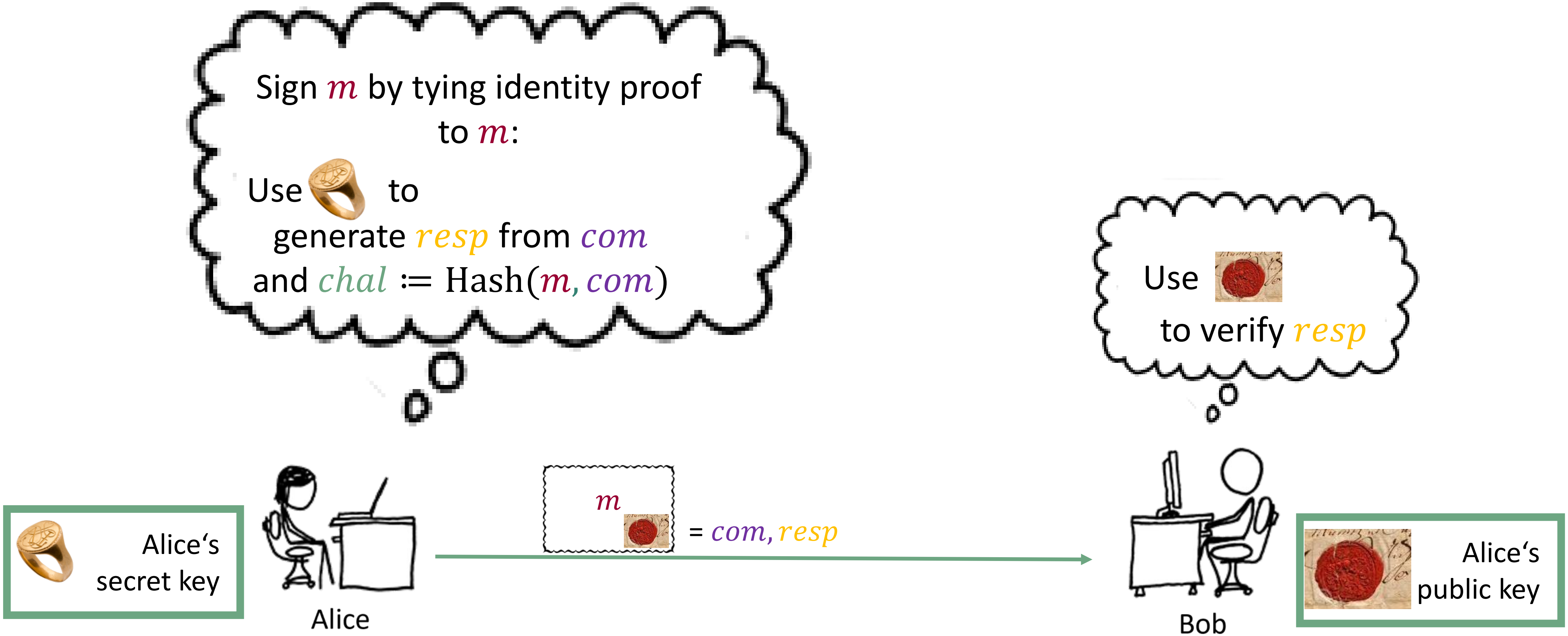


Image source: xkcd.com

Take-aways

We have a 'cooking recipe' for building signatures from a one-way trapdoor function

We again used the 'lego' approach:

Simpler primitive (RSA function) → Transformer (FDH) → More complex primitive (signing algorithm)

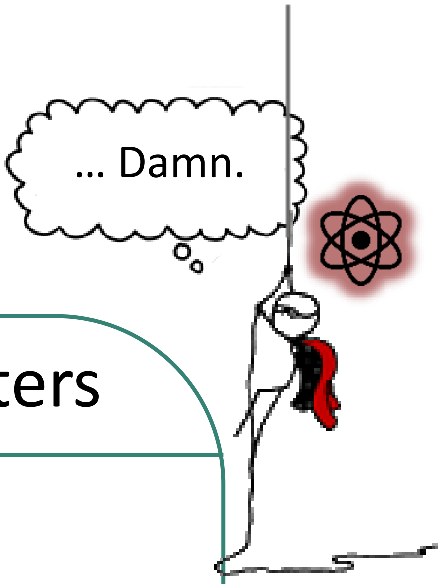
There are also other 'recipes' you will probably encounter during this week

All known recipes require some hardness assumption (e.g., 'inverting x^e is hard')

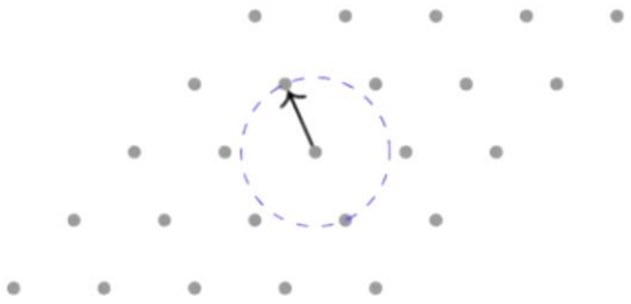
Q: how would we prove security against quantum attackers? (next talk)

Image source: xkcd.com

Post-quantum crypto



~~RSA~~ Problem that (hopefully) is hard even for quantum computers

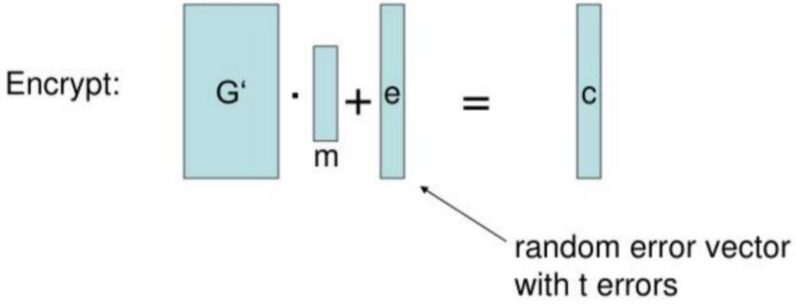


Finding a shortest vector in a lattice (Thu)

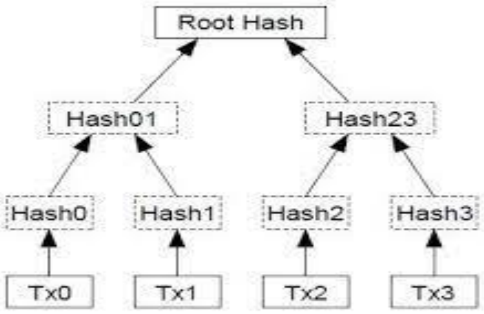
$$1000\,x + x^2 + 423\,y^2 z = 1$$
$$655\,y + 53\,yz = 13$$
$$29\,x + 3\,y^2 + 53\,xz^2 = 4$$

Solving multi-variable polynomial equations (Fri)

Decoding error-correcting codes (Wed)



Attacking hash functions (Wed)



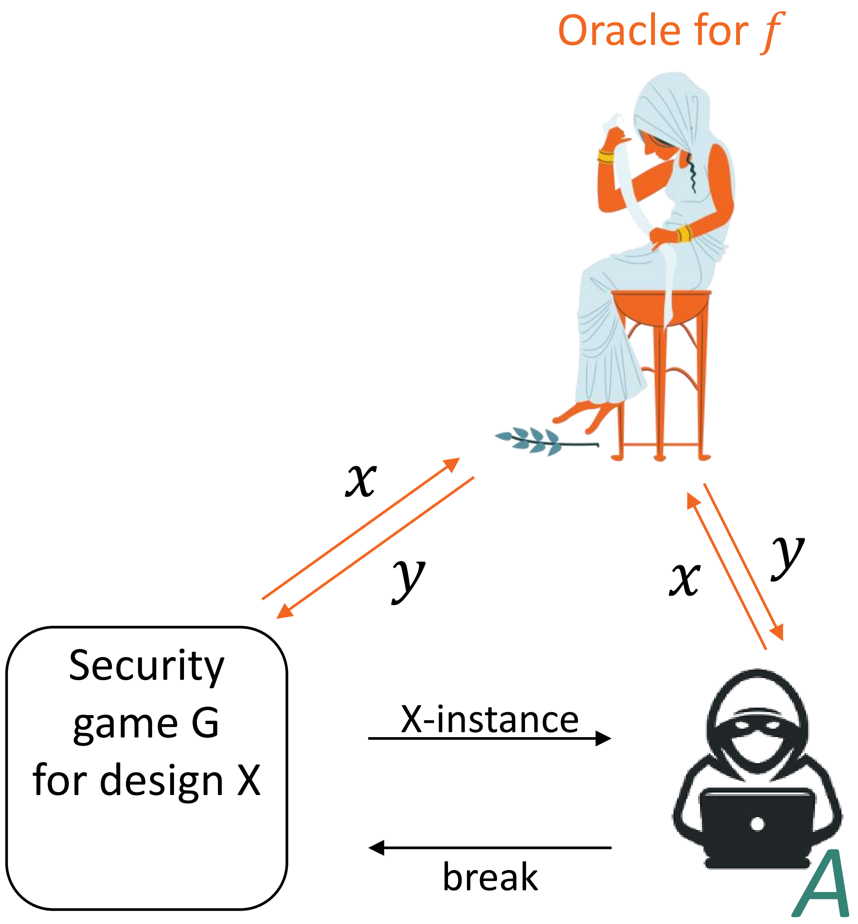
If time permits: random oracle model (ROM)

Heuristic: Replace hash function

$$\text{Hash}: \{0,1\}^n \to \{0,1\}^m$$

with ‘oracle box’ for truly random

$$f: \{0,1\}^n \to \{0,1\}^m$$



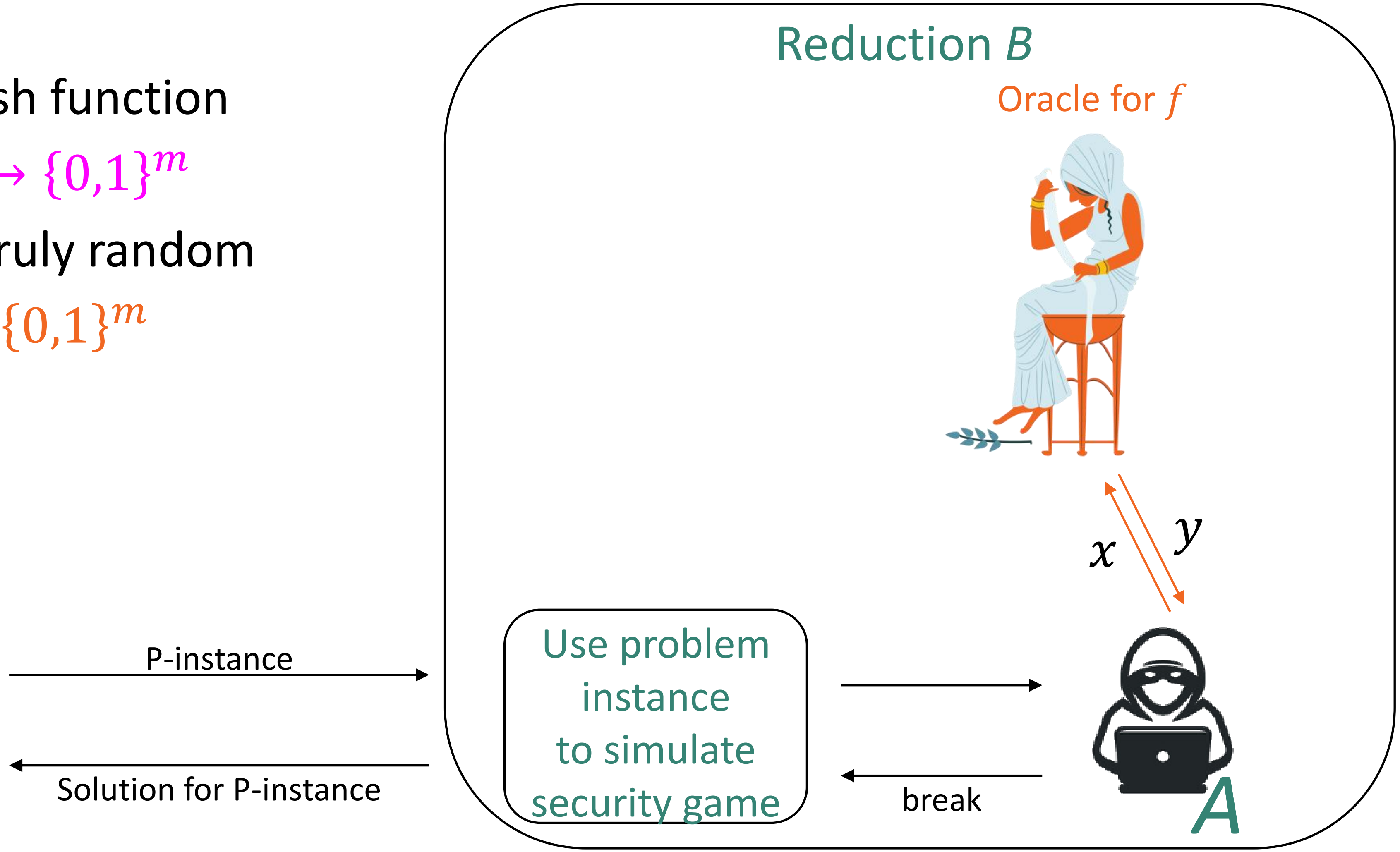
If time permits: random oracle model (ROM)

Heuristic: Replace hash function

$$\mathrm{Hash}: \{0,1\}^n \to \{0,1\}^m$$

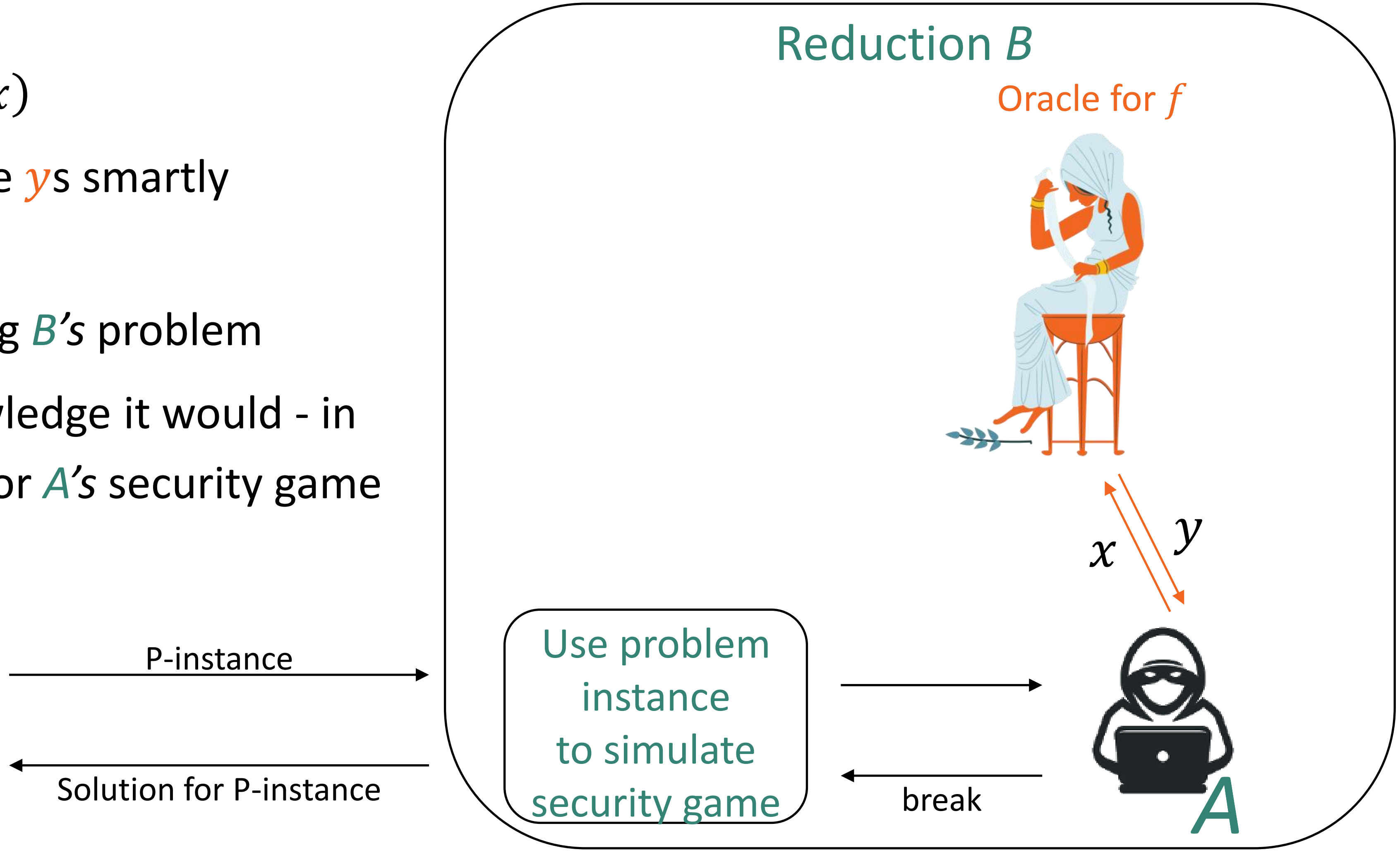
with 'oracle box' for truly random

$$f: \{0,1\}^n \to \{0,1\}^m$$



Perks of the random oracle model

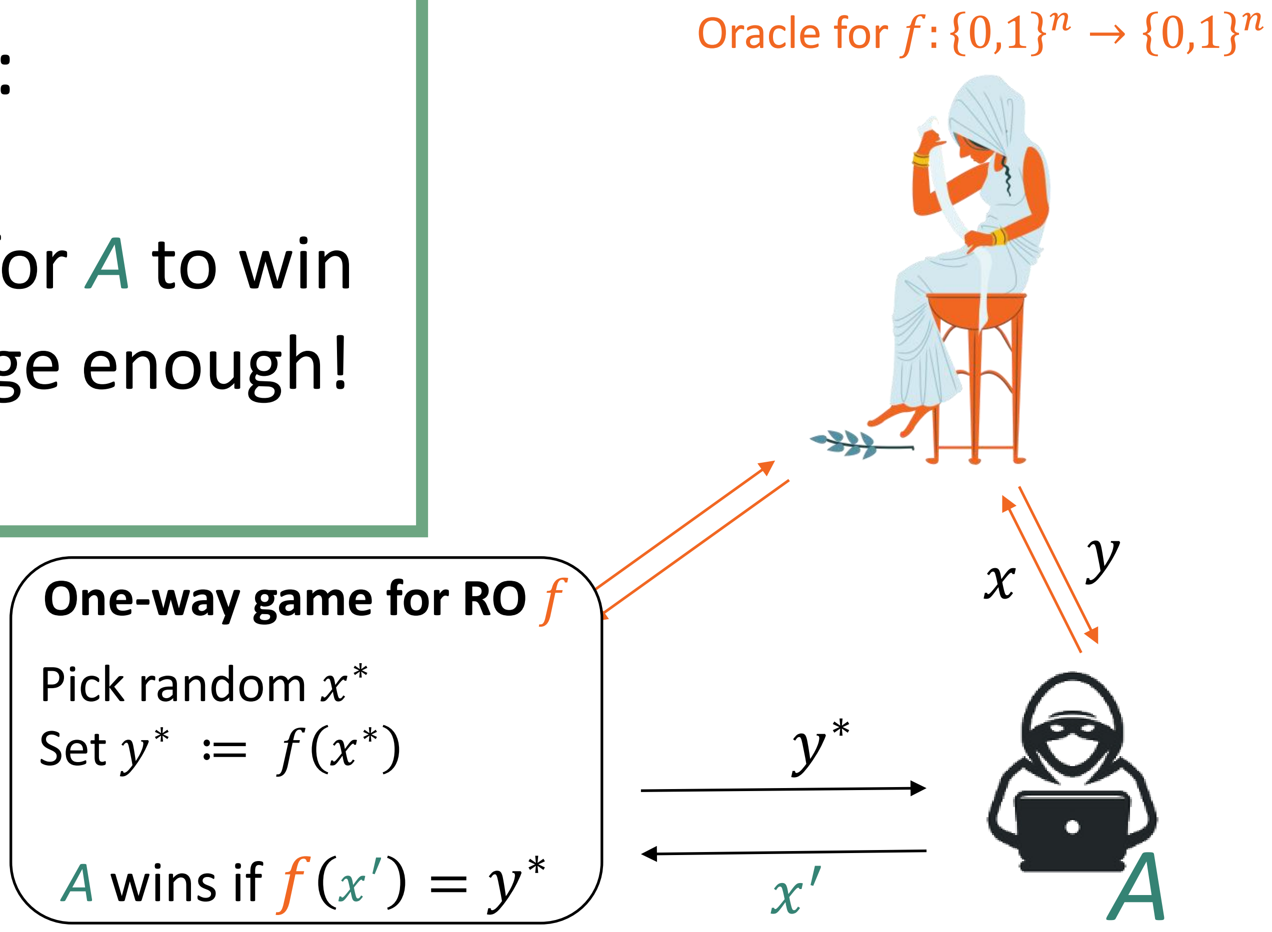
- Unpredictability of $f(x)$
- 'Tricking A': Picking the ys smartly enough, B can
  a) trick A into solving B's problem
  b) feign secret knowledge it would - in principle - need for A's security game



Practice example: ROs as one-way functions

Short DIY break:

Try to reason why it is hard for A to win the one-way game if n is large enough!



Practice example: ROs as one-way functions

Say A makes q many queries to f

- Per query $x \neq x^*$: f returns y^* with probability $\frac{1}{2^n}$
- A queries f on x^* with probability $\lessapprox \frac{q}{2^n}$
- If no query yields y^*: $f(x')= y^*$ with probability $\frac{1}{2^n}$

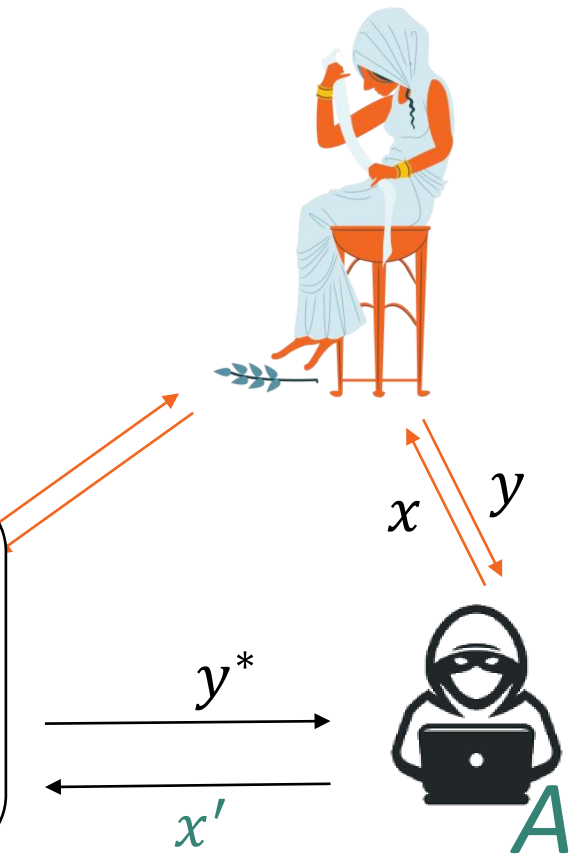
$$\Pr[A\ wins] \lessapprox \frac{q}{2^n} + \frac{q}{2^n} + \frac{1}{2^n}$$

One-way game for RO f

Pick random x^*
Set $y^* := f(x^*)$

A wins if $f(x') = y^*$

Oracle for $f: \{0,1\}^n \rightarrow \{0,1\}^n$



This heuristic seems weird.

☹ No theoretical justification

Counterexamples: designs that are

- secure in the ROM, but
- insecure when instantiating RO with any hash function

☺ So far: good track record for ‘natural’ schemes

Helps identify design bugs