Ruby Monstas

Session 17: Interlude: Encryption

Encryption

What comes to mind if you think about encryption?

Encryption

Certificates

AES

Public Key Encryption

Crypto Currencies

Privacy

SSH

VPN

HTTPS

TLS

Quantum Cryptography

Digital Signatures

Encryption Keys

Elliptic curves

PGP/GPG

NSA

SHA-1

Enigma

Caesar Cipher

Symmetric Encryption

Passwords

End-to-end

Encryption

MAGIC!

Encryption

~~MAGIC!~~

MATH!

Mathematical Ingredients

- Long integers
- Multiplication
- Exponentiation
- Division
- Modulo
- Prime numbers

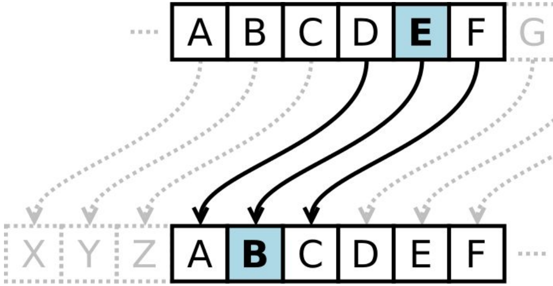
No math details in this talk though!

Topics

- Symmetric Encryption
- Random numbers
- Asymmetric (public key) Encryption
- Cryptographic Hash Functions

A bit of history

Caesar cipher



Enigma

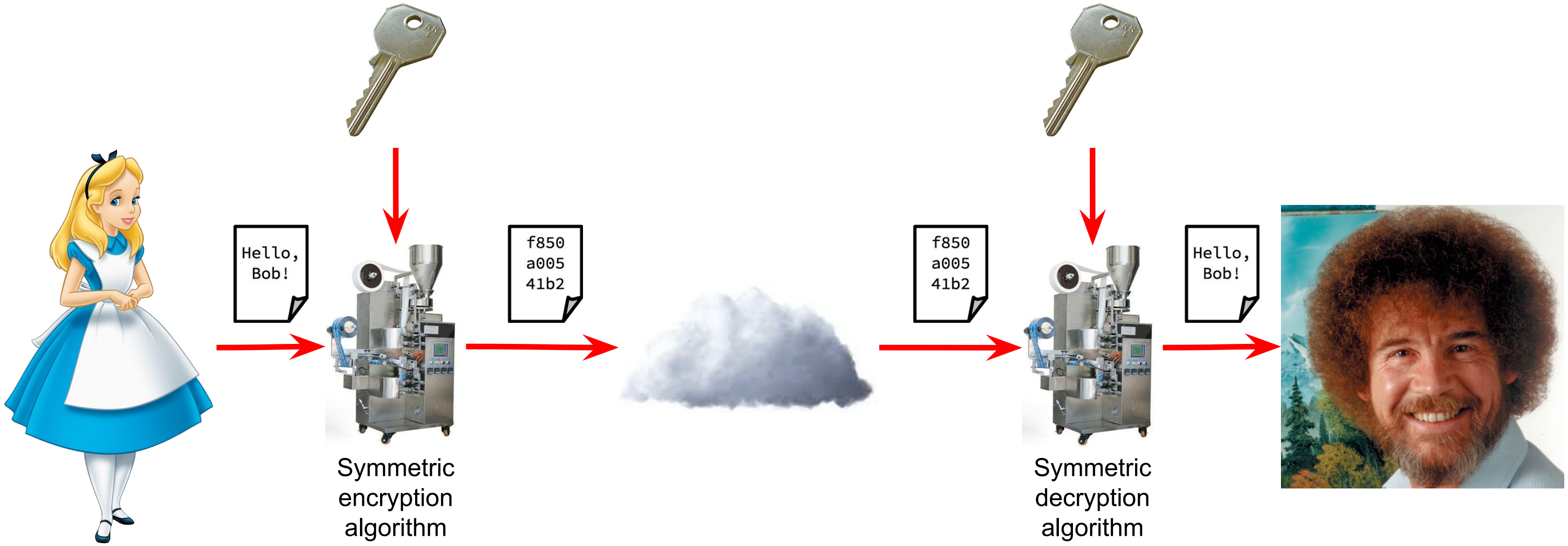
Scytale

Source: https://en.wikipedia.org/wiki/Cryptography

Symmetric encryption

Symmetric encryption

Symmetric encryption



Symmetric encryption

Examples:

- AES (Rijndael)
- DES, 3DES
- Blowfish

Advantage: Generally good performance

Disadvantage: Both parties need to know the key

Symmetric encryption

Problem: People's brains are terrible at generating keys!

If the key or even only part of it can be guessed, it makes an attack easier (brute force).

Random numbers

Why random numbers?

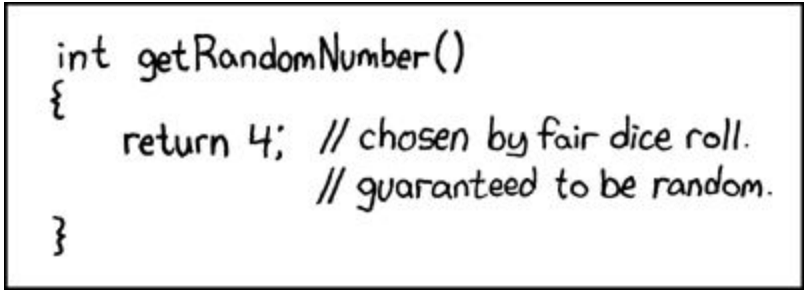
Keys (e.g. to encrypt things with) are generated from random numbers.

Caveat: It’s hard to generate truly random numbers!

Computers are deterministic machines by definition. Where can the randomness come from?

Random numbers

How not to do it:



https://xkcd.com/221/

Random numbers

What to do instead:

Collect truly random data (so-called entropy) and generate random numbers from it!

```
% xxd -l 16 -p /dev/random
03515dce8971a29f6764c0c275784ec0
```

Random numbers

What can happen?

[Wikipedia: Prominent random number generator attacks]

When part of the key is predictable it can take attackers orders of magnitude less time to guess the key!

Symmetric encryption

```ruby
require 'openssl'

ALGORITHM = 'AES-256-CBC'

puts 'Enter message to encrypt:'
message = gets.chomp

cipher = OpenSSL::Cipher.new(ALGORITHM)

key = cipher.random_key
hex_key = key.unpack('H*').first

puts "Randomly generated key in hexadecimal: #{hex_key}"

cipher.encrypt
cipher.key = key

encrypted_message = cipher.update(message)
encrypted_message << cipher.final

hex_encrypted_message = encrypted_message.unpack('H*').first

puts "Encrypted message in hexadecimal: #{hex_encrypted_message}"
```

```
% ruby aes_encrypt.rb
Enter message to encrypt:
Hello, Bob!
Randomly generated key in
hexadecimal:
52b0278e72ef57afdfae73baf1145d4309
4c8ba071e8c5dd7449c99dfa0fe146
Encrypted message in hexadecimal:
d789d4b1d816d150e146d857e927ac8b
```

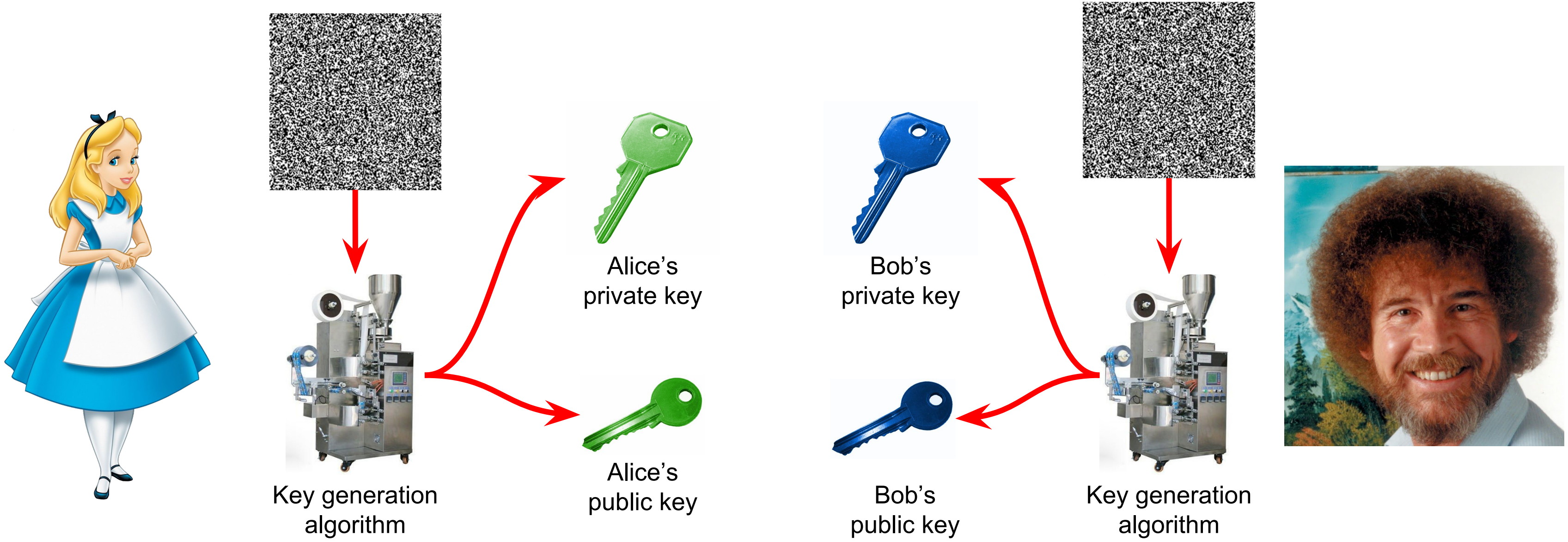
Symmetric encryption

```ruby
require 'openssl'

ALGORITHM = 'AES-256-CBC'

puts 'Enter key to decrypt with (in hexadecimal):'
hex_key = gets.chomp

puts 'Enter message to decrypt (in hexadecimal):'
hex_message = gets.chomp

cipher = OpenSSL::Cipher.new(ALGORITHM)

key = [hex_key].pack('H*')
message = [hex_message].pack('H*')

cipher.decrypt
cipher.key = key

message = cipher.update(message)
message << cipher.final

puts "Decrypted message: #{message}"
```

```
% ruby aes_decrypt.rb
Enter key to decrypt with (in
hexadecimal):
52b0278e72ef57afdfae73baf1145d4309
4c8ba071e8c5dd7449c99dfa0fe146
Enter message to decrypt (in
hexadecimal):
d789d4b1d816d150e146d857e927ac8b
Decrypted message: Hello, Bob!
```

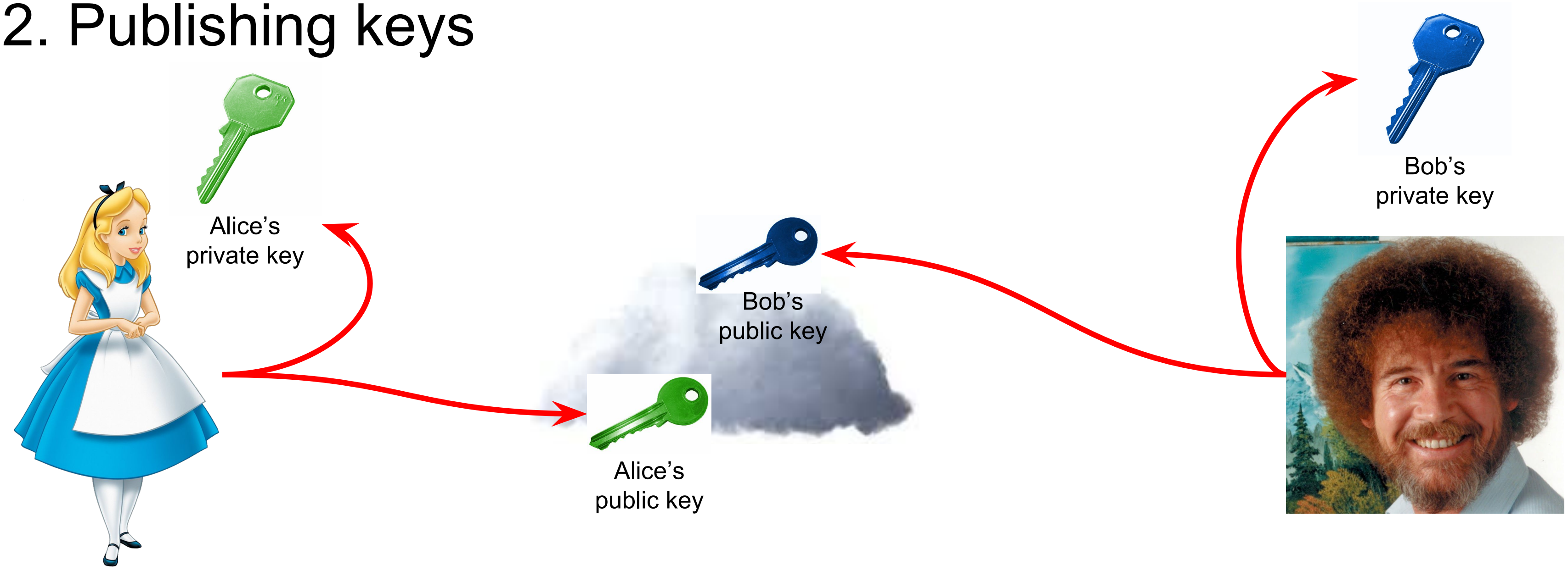
Asymmetric (public key) encryption

1. Generating a key pair



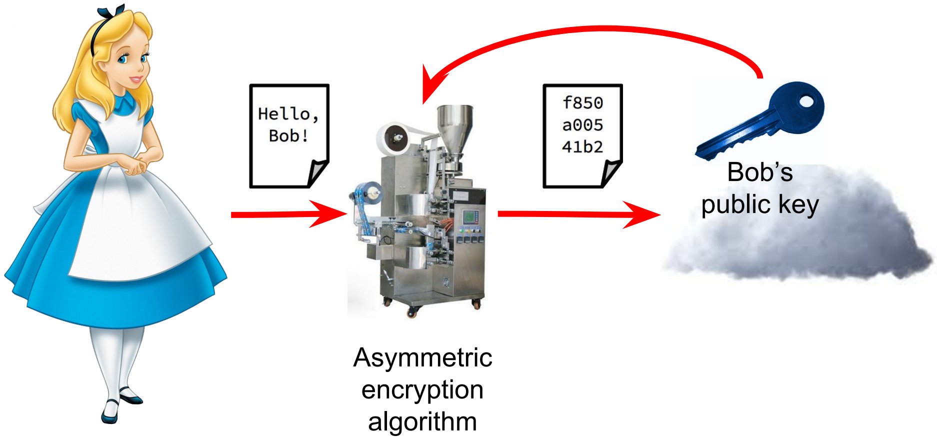
Asymmetric (public key) encryption

2. Publishing keys



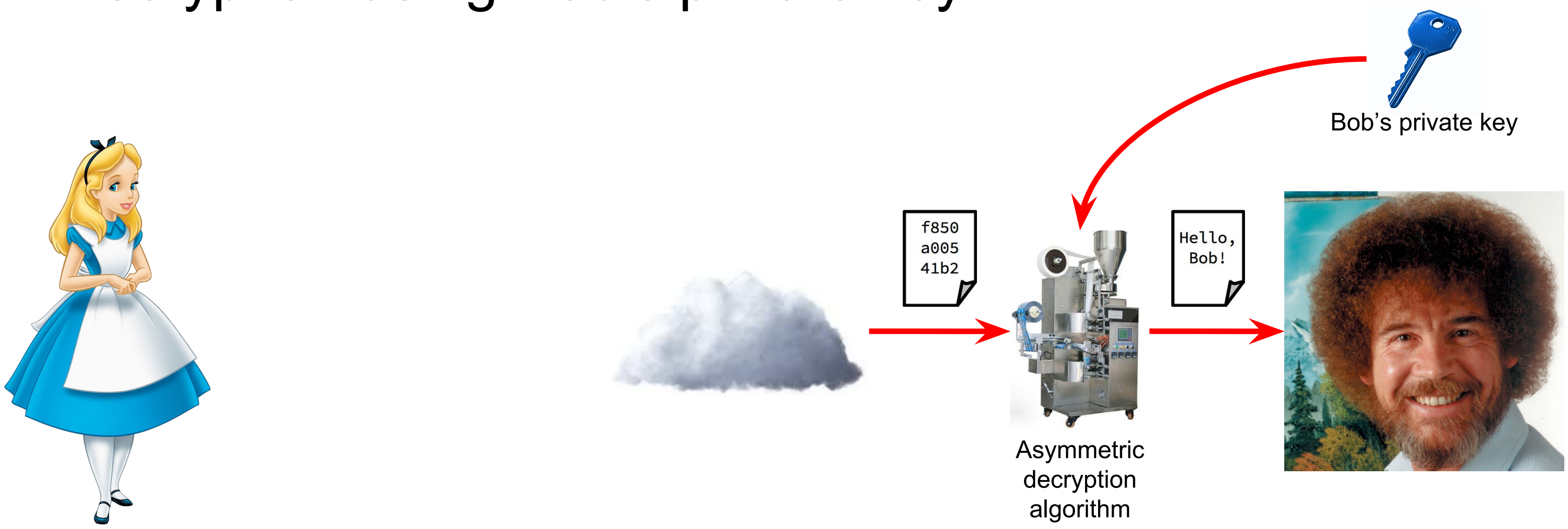
Asymmetric (public key) encryption

3. Encryption using Bob’s public key



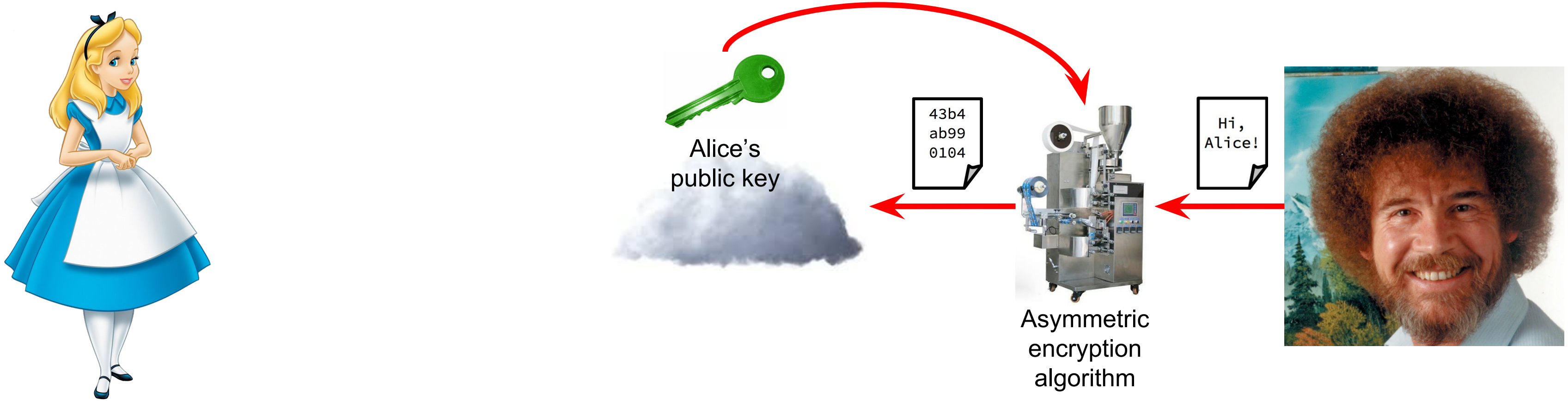
Asymmetric (public key) encryption

4. Decryption using Bob's private key



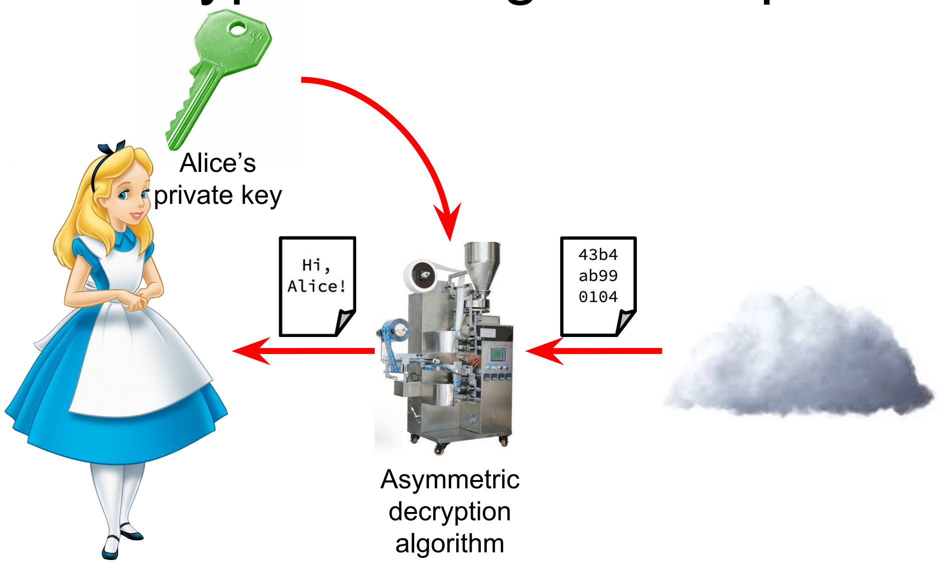
Asymmetric (public key) encryption

5. Encryption using Alice's public key



Asymmetric (public key) encryption

5. Decryption using Alice's private key



Asymmetric (public key) encryption

Examples:

- RSA
- ElGamal
- PGP

Advantage: Public keys can be exchanged in the open

Disadvantage: Generally slower than symmetric crypto

Asymmetric (public key) encryption

Public keys are public. Anyone can use them. How does Bob know the message is from Alice and vice versa?

Enter: Cryptographic Hash Functions!

Cryptographic Hash Functions

Use: “Digesting” an arbitrary length text into a value of fixed length:

```
% echo 'Hello, Bob!' | shasum -a 256
c4aaca0f9c0d691671659dfbcdf030d6009c2551fb53e4761a30cb29fc5f9ffb  -
```

Cryptographic Hash Functions

The ideal cryptographic hash function has five main properties:

- it is deterministic so the same message always results in the same hash
- it is quick to compute the hash value for any given message
- it is infeasible to generate a message from its hash value except by trying all possible messages
- a small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value
- it is infeasible to find two different messages with the same hash value

Source: [Wikipedia: Cryptographic hash function]

Cryptographic Hash Functions

How are passwords stored, e.g. for your Gmail account?

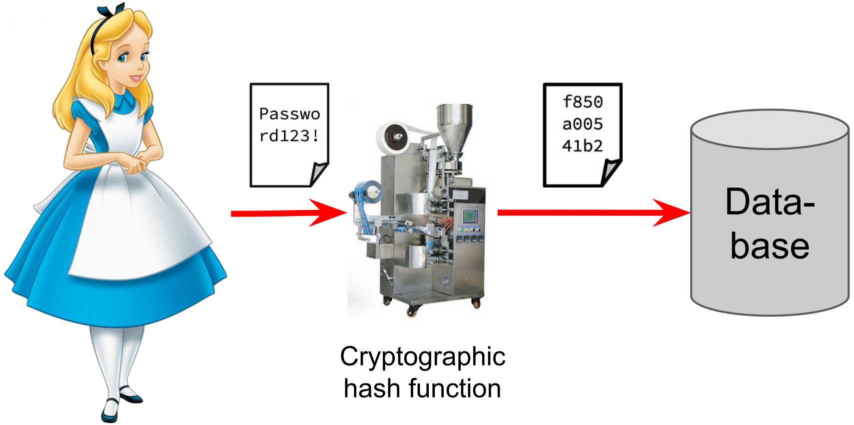
Possibility: In plain text

Disadvantage: If your database gets stolen, all your users' passwords are compromised!

Cryptographic Hash Functions

Better idea: Use a cryptographic hash function!

Sign up:

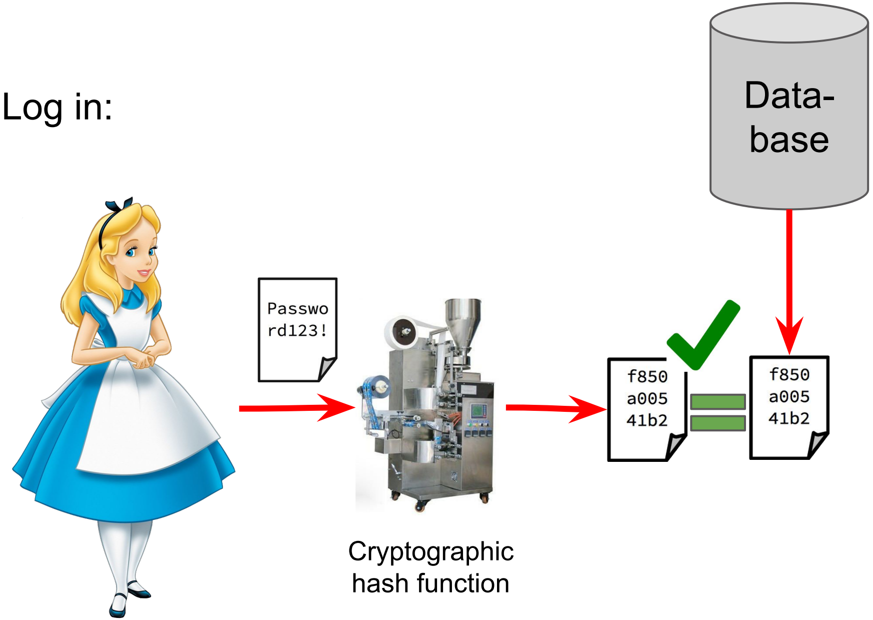


Additional benefit: All the stored, hashed passwords have the same length!

Cryptographic Hash Functions

Better idea: Use a cryptographic hash function!

Log in:



Cryptographic
hash function

Cryptographic Hash Functions

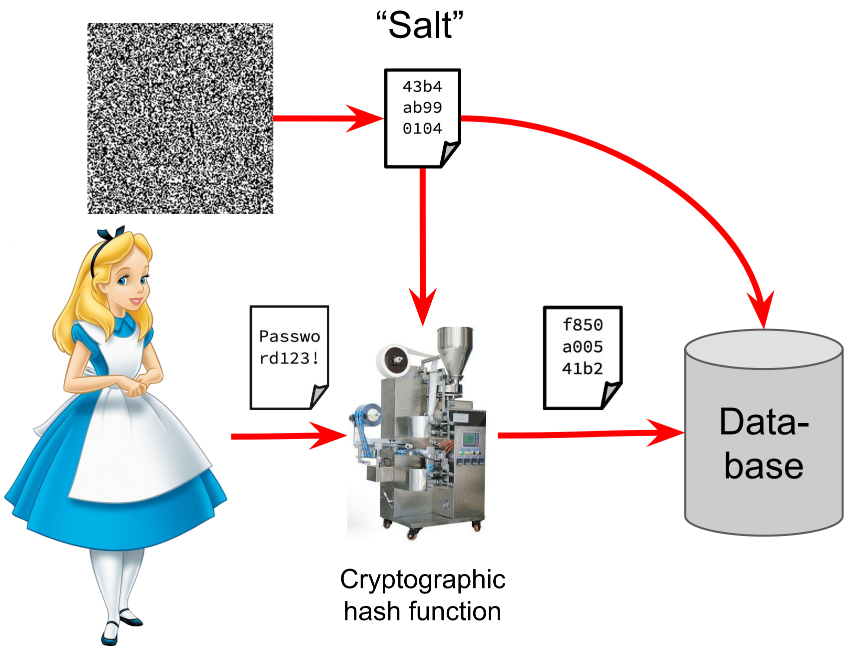
What if two users choose the same password by chance?

An attacker could use that information if the database gets compromised!

Solution: Salt your password!

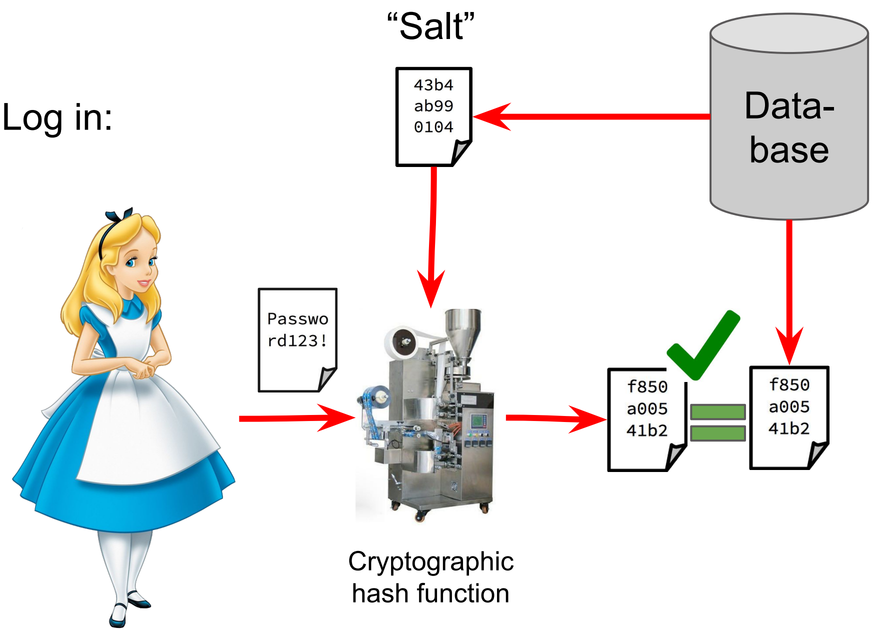
Cryptographic Hash Functions

Sign up:



Cryptographic Hash Functions

Log in:



Cryptographic Hash Functions

Password hashing and salting in Ruby using bcrypt gem:

```
irb(main):001:0> require 'bcrypt'
=> true
irb(main):005:0> password_hash = BCrypt::Password.create("Password123!")
=> "$2a$10$yxazpyL1iZ7lpLr/c8w4l.Eyii7oI3pRwmyw1gS/euLF4CJEtz6RK"
irb(main):006:0> password_object = BCrypt::Password.new(password_hash)
=> "$2a$10$yxazpyL1iZ7lpLr/c8w4l.Eyii7oI3pRwmyw1gS/euLF4CJEtz6RK"
irb(main):007:0> password_object == 'wrong password'
=> false
irb(main):008:0> password_object == 'Password123!'
=> true
```

Handy: bcrypt puts the password hash and the salt in the same String!

Caveat: Bcrypt doesn’t actually use a cryptographic hash function, but the Blowfish symmetric cipher. The principle stays the same though!

Cryptographic Hash Functions

Security as of mid 2018:

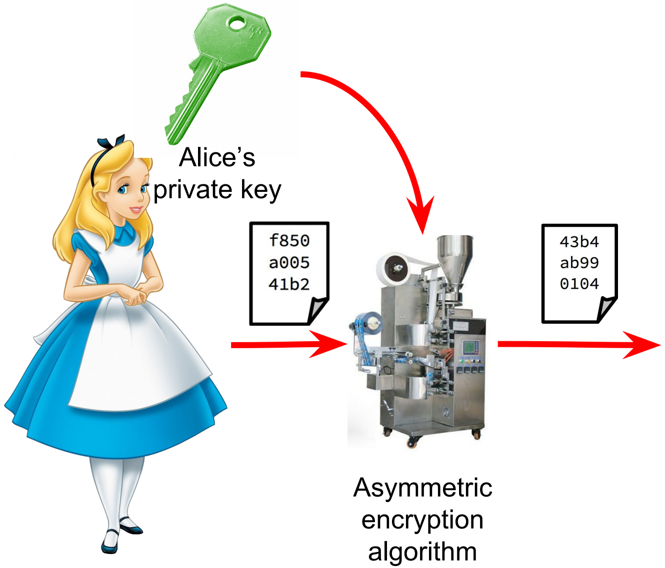
- MD5 is considered broken
- SHA-1 is considered broken
- SHA256 or other SHA variants with longer bit lengths should be used

Putting it all together

1. Calculating a cryptographic hash over the message

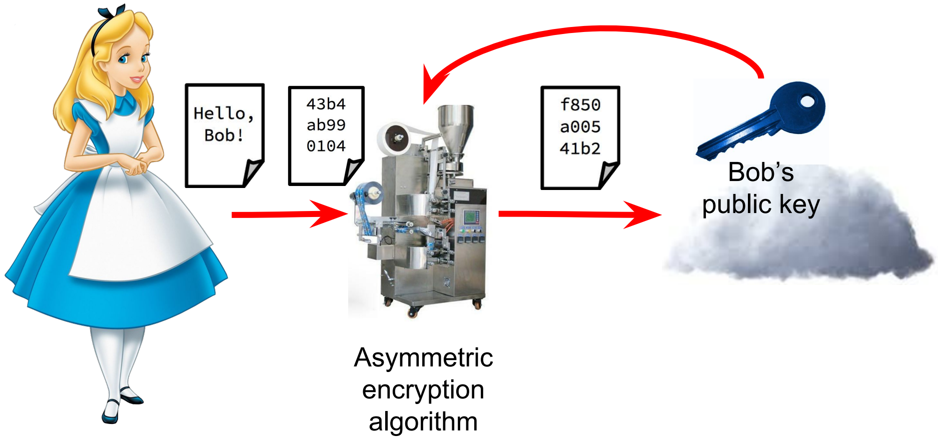
Putting it all together

2. Encrypting the hash using Alice’s private key



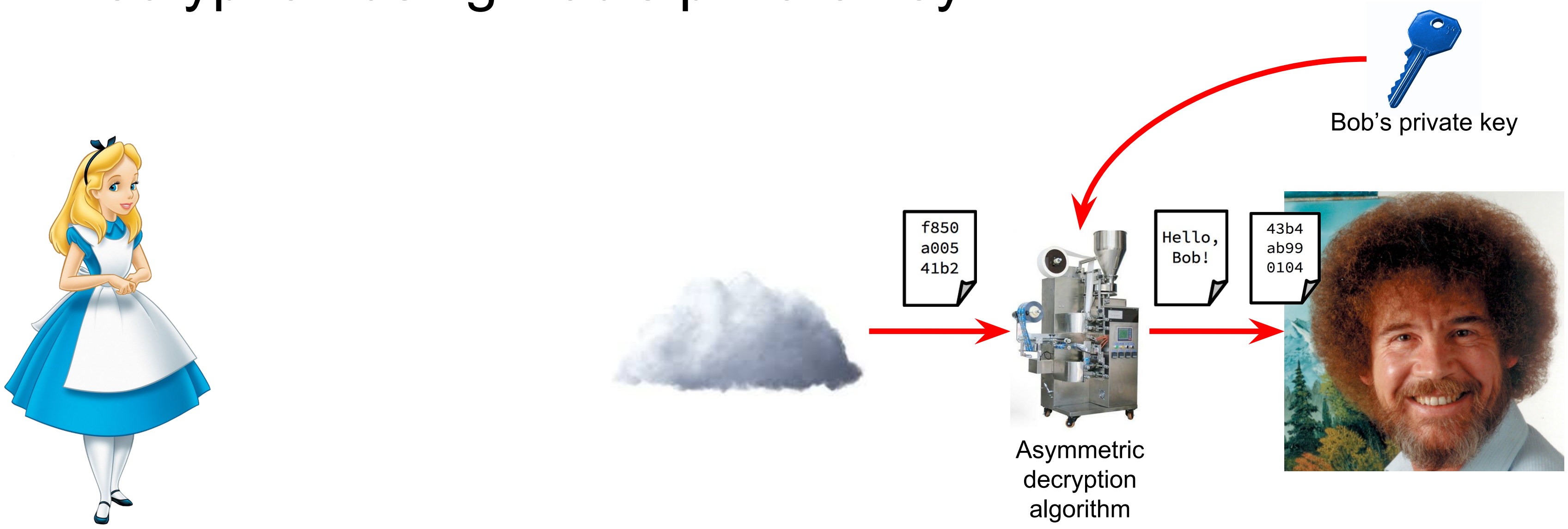
Putting it all together

3. Encrypting message + signature using Bob's public key



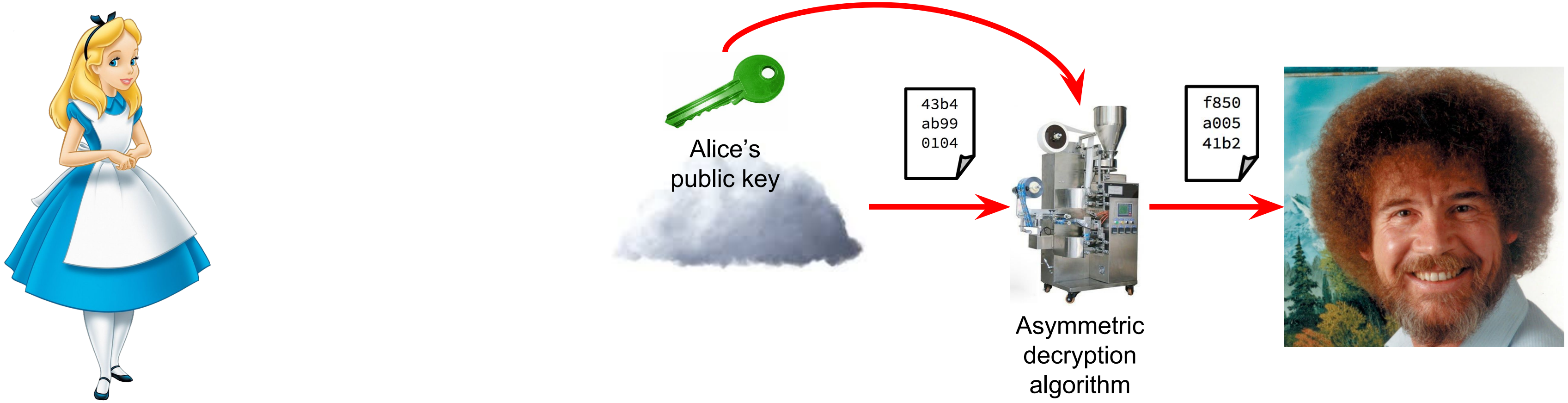
Putting it all together

4. Decryption using Bob's private key



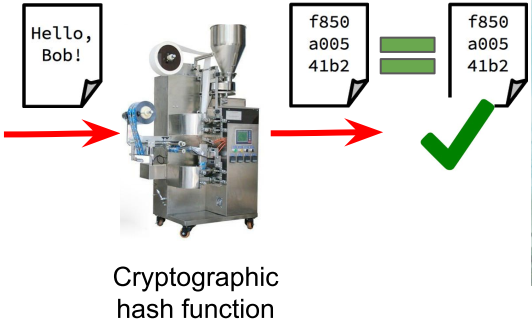
Putting it all together

5. Decryption of signature using Alice's public key



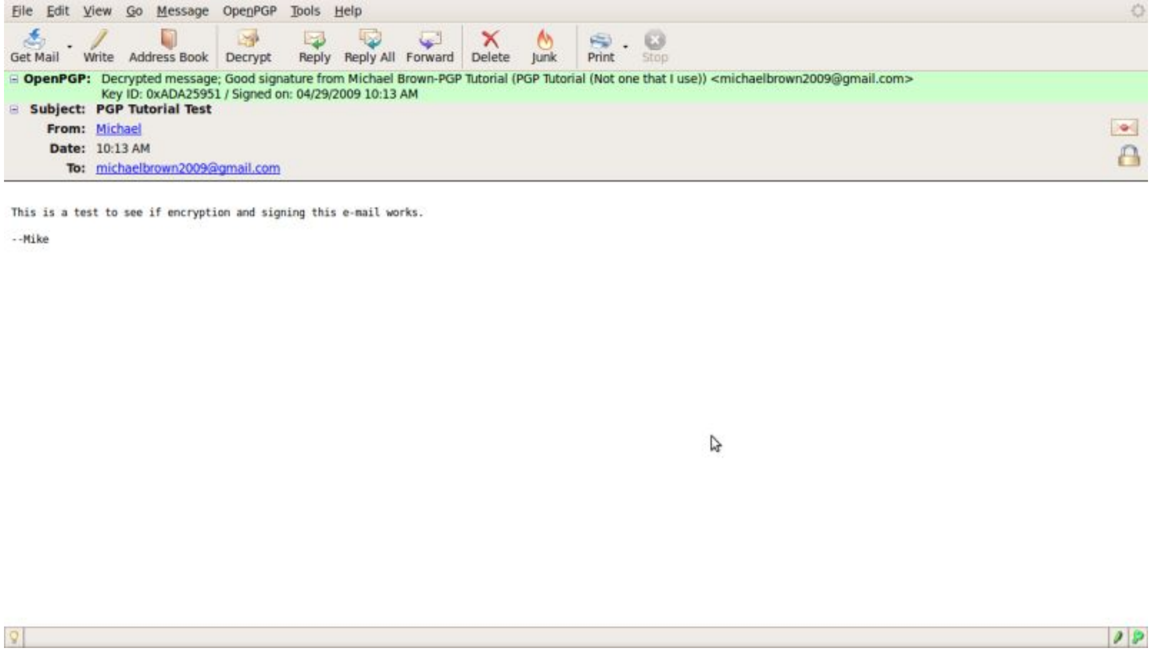
Putting it all together

6. Calculating a cryptographic hash over the message and comparing to Alice's decrypted signature

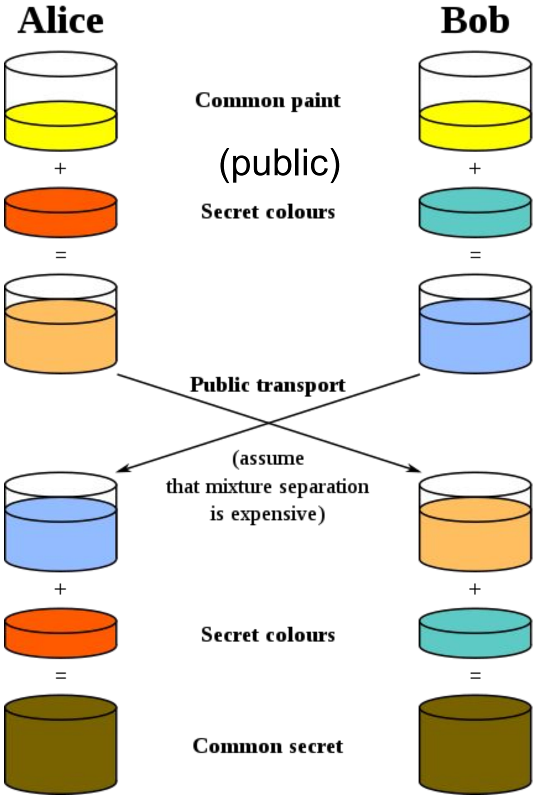


PGP/GPG

This is how PGP/GPG works!



Bonus: Diffie-Hellman Key Exchange



Merkle Hellman Diffie

Turing Award 2015:
Whitfield Diffie, Martin E. Hellman

Source: [Wikipedia: Diffie-Hellman Key Exchange]

Take-home messages

Use well-researched, public algorithms!

Don’t implement your own crypto algorithms!

Use secure sources of randomness!

Keep your private keys private!