# We're Getting Cryptic

# Cryptography

Cryptography is the process of scrambling information into an unreadable form, in order to keep it secret!

Anytime we send information over the internet, all kinds of people have the opportunity to see it and try to read it - things like our credit card numbers, health records, passwords, bank accounts, or even just private messages to other people.

# Cryptography

In Cryptography, when we scramble information, it's called **encryption**. When we unscramble it, it's called **decryption**!

The goal of Cryptography is to encode information in such a way that only someone with verified authority such as a password or secret key is able to **decrypt** the data.

# General Encryption

## Data

SECRET SECRETS! Do NOT tell ANYONE what this secret message says secretly!

## Key

kxzhfoil'k/j4n;lakejboilkhsuf
kljejndo;isl4h29lhl.q,n3lieifvj
3il;jkfnu4io;l8d9po3i2hnblikj
hnos;ifua;ls83ij2o;o22ilkhsp
ailk3hb

# General Encryption

## Data

101000110101001001011
010010010110100010010
101101010101011101010
100001010111101010111
111001000100101010101
010101010001101010010
0101100010

## Key

110010100011010010010
100000100101111010010
111010010101010010100
010101100101000110100
100101000001001011110
100101110100101010100
1010001010

# General Encryption

Data

101000110101001001011
010010010110100010010
101101010101011101010
100001010111101010111
111001000100101010101
010101010001101010010
0101100010

Key

110010100011010010010
100000100101111010010
111010010101010010100
010101100101000110100
100101000001001011110
100101110100101010100
1010001010

encrypt (data, key)

Encrypted Message

001010011101001010010
101110101001001011000
100100101110101000101
011111010101001000101
011110101000101011101
010010010010111101101
0101110101

# General Encryption

## Data

1010001101010010010111
0100100101101000100101
1011010101010111010101
0000101011110101011111
1100100010010101010101
0101010100011010100100
101100010

## Key

1100101000110100100101
0000010010111101001011
1010010101010010100010
1010110010100011010010
0101000000100101111010
0101110100101010100101
0001010

encrypt (data, key)

## Encrypted Message

H{{"#as;o3n2 2l;ilnaw
lak3j2n
'L2n

l23;l;jlk32i 0p8934m
lknils8os;;w; &@LK asldf
l2k{[]3 23]@}@ :"

# General Encryption

Encrypted Message

00101001110100101001 0
10111010100100101100 0
10010010111010100010 1
01111101010100100010 1
01111010100010101110 1
01001001001011110110 1
0101110101

Key

11001010001101001001 0
10000010010111101001 0
11101001010101001010 0
01010110010100011010 0
10010100000100101111 0
10010111010010101010 0
1010001010

decrypt (data, key)

Decrypted Message

SECRET SECRETS! Do NOT tell ANYONE what this secret message says secretly!

# General Encryption

Encrypted Message

0010100111010010100 10
1011101010010010110 00

Key

1100101000110100100 10
1000001001011110100 10
1110100101010100101 00
0101011001010001101 00
1001010000010010111 10
1001011101001010101 00
1010001010

decrypt (data, key)

Decrypted Message

SECRET SECRETS! Do NOT tell ANYONE what this secret message says secretly!

The **ONLY** way to decrypt the message is with the correct **key** - otherwise, you'll just get more gobbledygook!

# A Historical Example

One of the oldest (and simplest) examples of Cryptography is the **Caesar Cipher**, which was used by Julius Caesar of Roman fame to send secret messages to his army!

Because the messages were encrypted, if they were ever intercepted by the enemy, they would be unreadable!

# Caesar Cipher

In Caesar Cipher, each letter in the message is *shifted* by a certain amount, called the **key**.

With a key of 1,

A becomes B
B becomes C
C becomes D
....
Z becomes A

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

K

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

KS

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

KS S

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

KS SZ

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

**KS SZIV XLIVI**

# Caesar Cipher

Example Encoding

GO OVER THERE with a key of 4 would result in:

**KS SZIV XLIVI**

GO OVER THERE

We can decrypt the message if we know what the key is! We just have to shift the other direction by that many values!

# Caesar Cipher

If we wanted to write some code to accomplish this task, it might look like this:

```
message = input("Gimme a secret message! ")
key = 3
secret = encrypt(message, key)

send(secret)
```

# Caesar Cipher

If we wanted to write some code to accomplish this task, it might look like this:

```
message = input("Gimme a secret message! ")
key = 3
secret = encrypt(message, key)

send(secret)
```

We can use the Caesar Cipher encryption algorithm if we want to, but if we want to use something a little more secure we totally can!

# Caesar Cipher

Caesar Cipher is one of the easiest encryption methods to understand, which also means that it's one of the easiest methods to crack. The biggest detriment is that there are only 26 possible keys to try before the code is cracked!

Another issue with Caesar Cipher is that it can only encode alphabetical data - if we want to encode something like an image or some audio, we're out of luck!

# Today's Cryptography

- Similarities to Caesar's Cipher
  - Still encrypt data according to some key
  - Works via a mathematical basis

- Differences from Caesar's Cipher
  - The math for encryption is much more complicated than "shift down the alphabet"
  - The key is significantly larger

# Cracking Encryption

If an attacker tries every single possible key, they will eventually be able to decrypt the message. They don't need to know the key beforehand!

In Caesar Cipher, there are only 26 possible keys, which will take modern computers less than a second!

# Cracking Encryption

In the past, we've used **40-bit keys** for normal encryption. Since keys are stored in binary, that means that there are $2^{40}$ possible keys!

That's 1,099,511,627,776 total possible keys!

As computers continued to improve in speed, though, even **40-bit keys** weren't enough to stop hackers!

# Cracking Encryption

Today's encryption algorithms use **256-bit keys**!

This means that there are $2^{256}$ possible keys - that's 115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936 total!

Even with today's computing power, it would take trillions of years to try every single possible key!

# Computational Problems

Different problems can be classified by how long it takes for computers to solve them.

We need our encryption to take computers a long time to break - we need it to be a computationally hard problem.

Trying every single key should take a very long time - it should be a hard problem!

# Computational Problems

In easy computational problems, increasing the size of the input causes the time it takes to solve to increase at a reasonable rate.

# Computational Problems

In easy computational problems, increasing the size of the input causes the time it takes to solve to increase at a reasonable rate.

Finding the smallest element in a list of numbers is an easy problem, because adding 1 element means that you only need to look at 1 more value to determine if it's smaller.

`[10, 4, 3, 9, 7, 2]` vs `[10, 4, 3, 9, 7, 2, 5]`

# Computational Problems

In computationally hard problems, solve time increases at an **exponential** rate, rather than a linear one.

# Computational Problems

In computationally hard problems, solve time increases at an **exponential** rate, rather than a linear one.

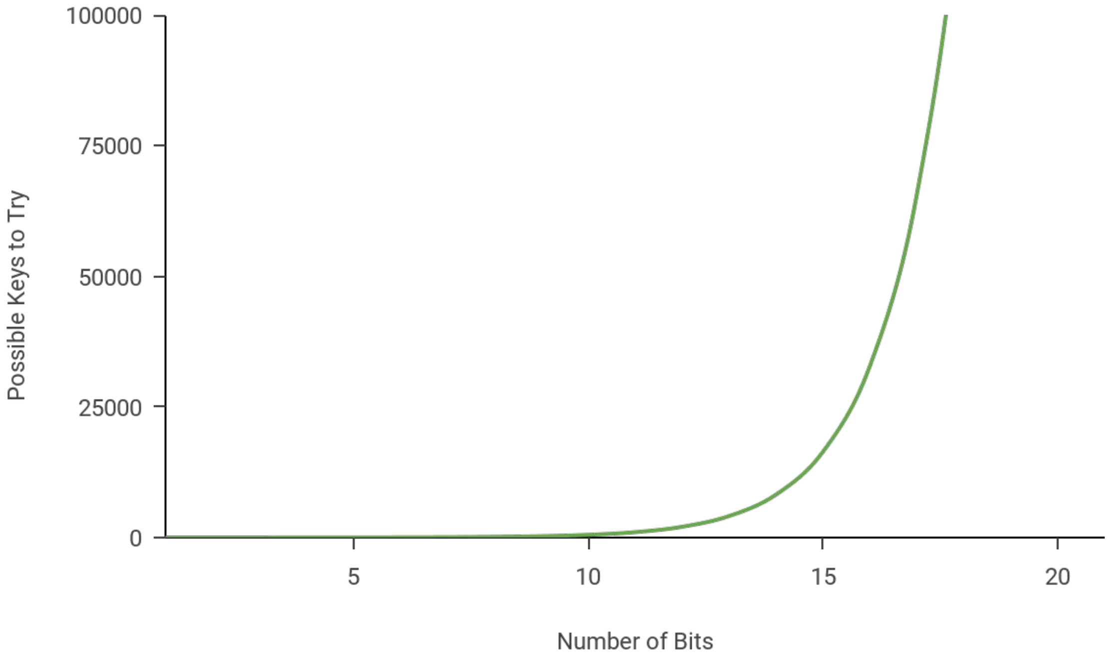
Cracking a large key is a computationally hard problem, because each time an additional bit is added, the number of possible keys doubles!

2 bits -> 4 possible keys

3 bits -> 8 possible keys

4 bits -> 16 possible keys

# Computational Problems



# Computational Problems

Let's discuss some differences between **40-bit** vs **256-bit** key size.

256 bits is roughly 6 times larger than 40 bits, so it takes up a decent amount more storage.

The number of keys is where the biggest difference comes in though - **256-bit** keys do not have 6 times more possible values than **40-bit** keys, it has $2^{216}$ more possible values!

# Computational Problems

Cracking an encryption is a **hard** problem for computers to solve, because they are not able to try every possible key in a reasonable amount of time. Even if computers get faster, all we need to do is increase the number of bits being used to store keys and the time will increase exponentially!

# Encryption

There are 2 main types of encryption used today:

- Symmetric Encryption
  - The same key is used to both encrypt and decrypt the data
- Asymmetric Encryption
  - One key is used to encrypt the data, but a different key is used to decrypt it

# Symmetric Encryption

Alph and Jeph have met in private, and decided on a private key that they'll share with one another.

# Symmetric Encryption

Alph and Jeph each have a copy of the same exact key.

# Symmetric Encryption

Alph wants to send a message to Jeph.

# Symmetric Encryption

Alph first encrypts the message using the key, then sends it over to Jeph.

# Symmetric Encryption

Alph first encrypts the message using the key, then sends it over to Jeph.

# Symmetric Encryption

Alph first encrypts the message using the key, then sends it over to Jeph.

# Symmetric Encryption

Jeph can then use the key to decrypt the message and read it!

# Symmetric Encryption

Jeph can then use the key to decrypt the message and read it!

# Symmetric Encryption

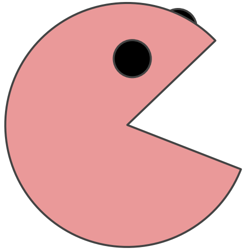
Jeph can then use the key to decrypt the message and read it!

# Symmetric Encryption

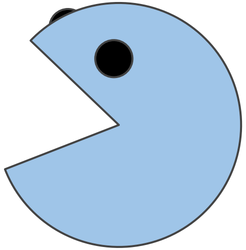
The problem with Symmetric Encryption is that there's billions of devices on the internet, so every single computer can't meet with every single other computer to agree on a private key!

# Symmetric Encryption

The solution to this problem is called **public key encryption**, and it's an asymmetric encryption system! One key is used to encrypt, and another is used to decrypt!



Alph



Jeph

# Symmetric Encryption

Alph has both a **public** key and a **private** key.
Similarly, Jeph has both a **public** key and a **private** key.

# Symmetric Encryption

Each person's **public** key is used for **encryption**, while their **private** key is used for **decryption**!

# Symmetric Encryption

As you might expect, your **public** key is available for everyone in the world to see, but your **private** key is kept secret.

# Symmetric Encryption

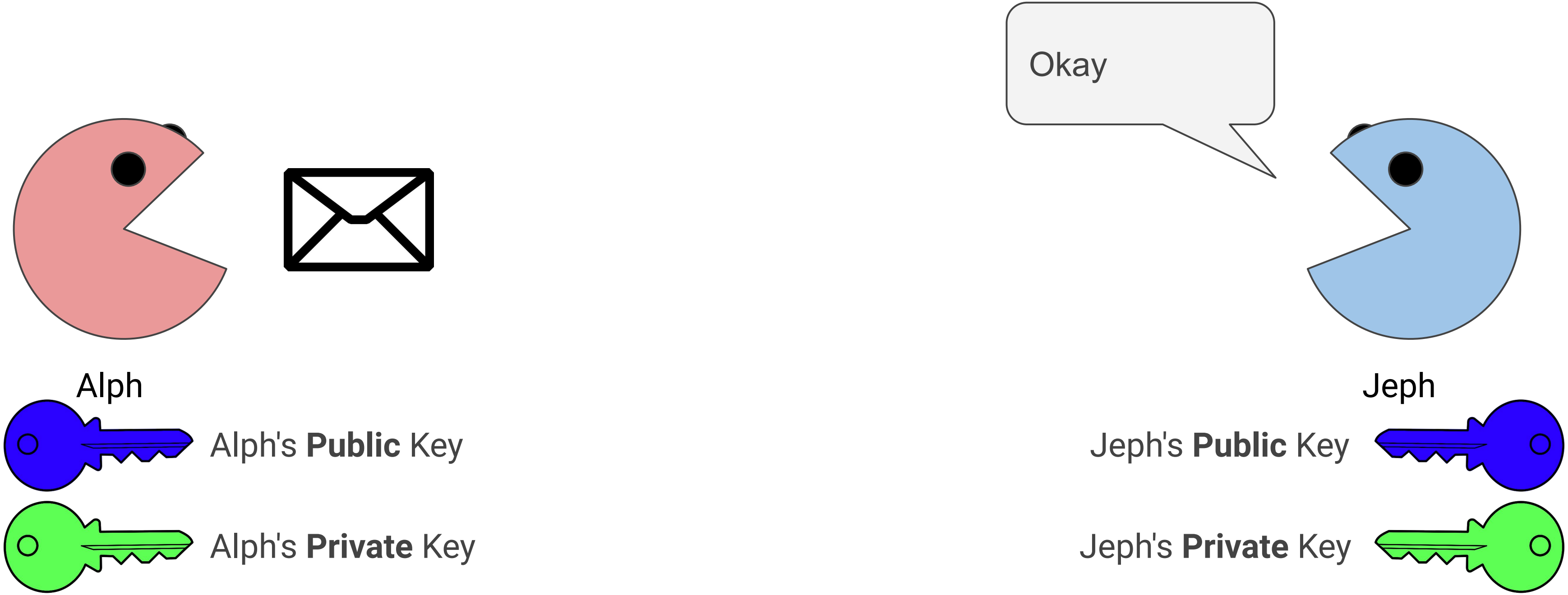
Alph wants to send a message to Jeph.

# Symmetric Encryption

Alph asks for Jeph's **public** key, and is obliged.

# Symmetric Encryption

Alph asks for Jeph's **public** key, and is obliged.



# Symmetric Encryption

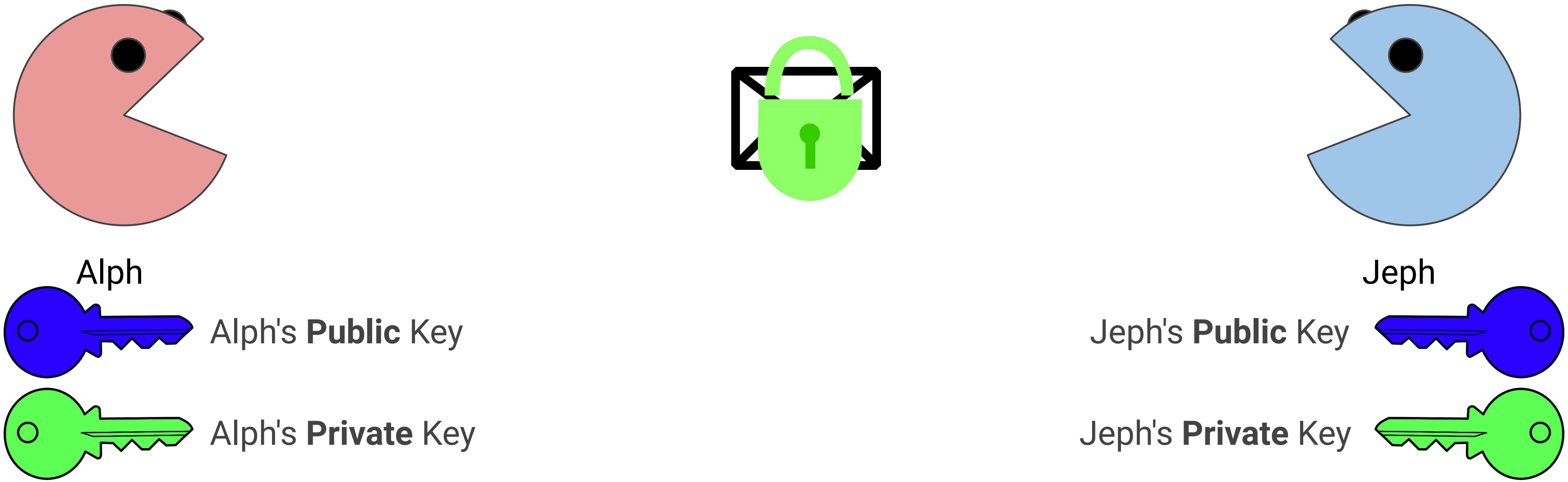
Alph asks for Jeph's **public** key, and is obliged.

# Symmetric Encryption

Alph encrypts the message using **Jeph's private key**.

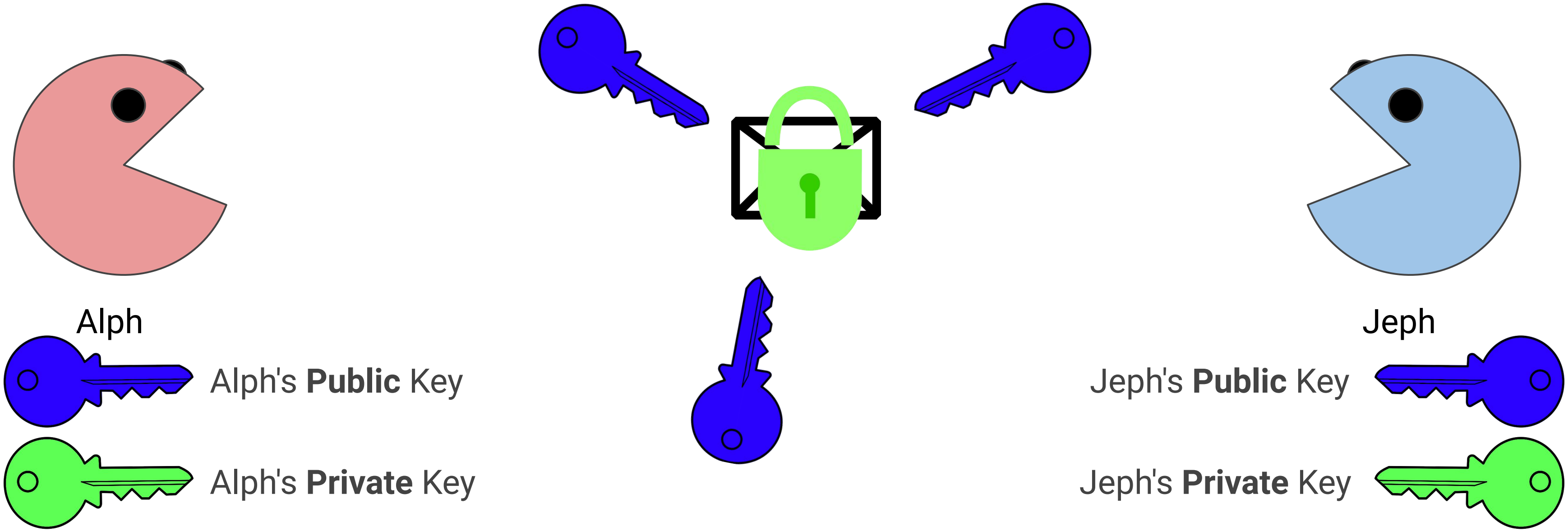
# Symmetric Encryption

Alph sends the encrypted message to Jeph.



# Symmetric Encryption

Even if attackers have Jeph's **public** key, they won't be able to decrypt the message, because only **private** keys decrypt!



# Symmetric Encryption

Jeph receives the message, then **decrypts** it using the **private** key.

# Symmetric Encryption

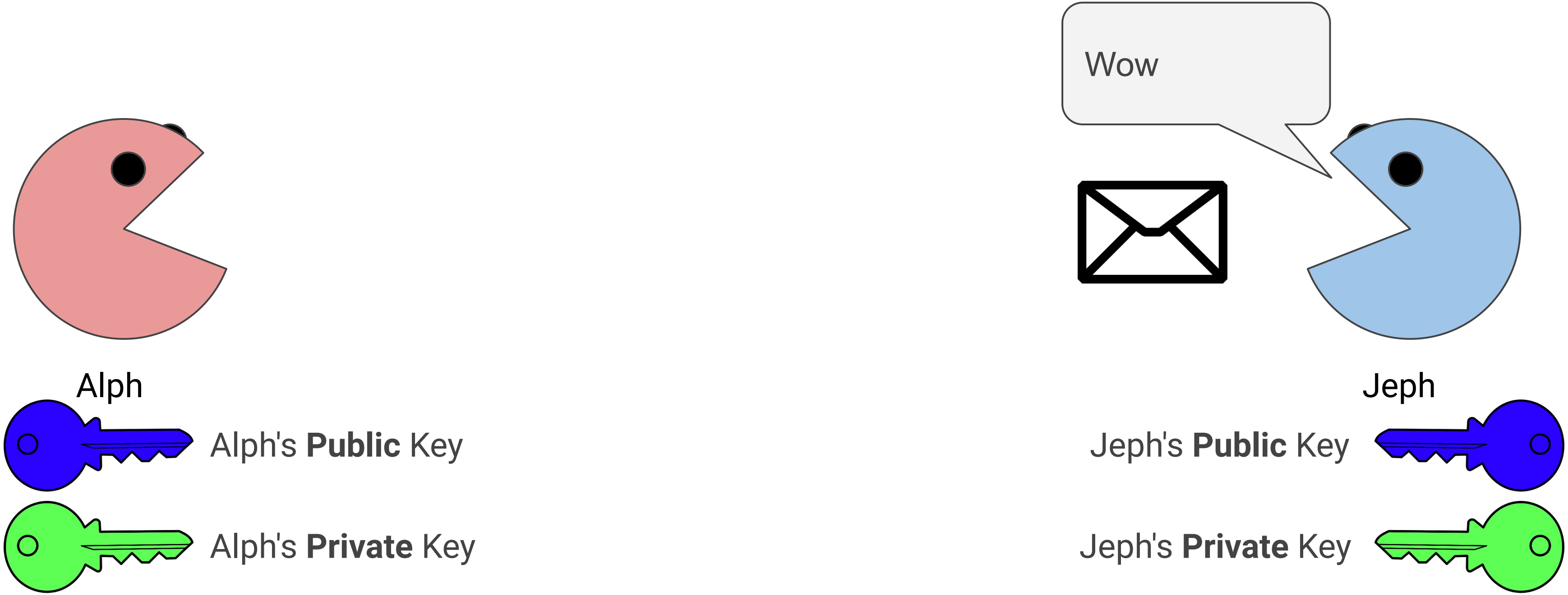
Jeph receives the message, then **decrypts** it using the **private** key.

# Symmetric Encryption

Jeph receives the message, then **decrypts** it using the **private** key.

# Symmetric Encryption

Jeph receives the message, then **decrypts** it using the **private** key.



# Password Management

## TIME IT TAKES A HACKER TO BRUTE FORCE YOUR PASSWORD

| Number of Characters | Numbers Only | Lowercase Letters | Upper and Lowercase Letters | Numbers, Upper and Lowercase Letters | Numbers, Upper and Lowercase Letters, Symbols |
|---|---|---|---|---|---|
| 4 | Instantly | Instantly | Instantly | Instantly | Instantly |
| 5 | Instantly | Instantly | Instantly | Instantly | Instantly |
| 6 | Instantly | Instantly | Instantly | 1 sec | 5 secs |
| 7 | Instantly | Instantly | 25 secs | 1 min | 6 mins |
| 8 | Instantly | 5 secs | 22 mins | 1 hour | 8 hours |
| 9 | Instantly | 2 mins | 19 hours | 3 days | 3 weeks |
| 10 | Instantly | 58 mins | 1 month | 7 months | 5 years |
| 11 | 2 secs | 1 day | 5 years | 41 years | 400 years |
| 12 | 25 secs | 3 weeks | 300 years | 2k years | 34k years |
| 13 | 4 mins | 1 year | 16k years | 100k years | 2m years |
| 14 | 41 mins | 51 years | 800k years | 9m years | 200m years |
| 15 | 6 hours | 1k years | 43m years | 600m years | 15 bn years |
| 16 | 2 days | 34k years | 2bn years | 37bn years | 1tn years |
| 17 | 4 weeks | 800k years | 100bn years | 2tn years | 93tn years |
| 18 | 9 months | 23m years | 6tn years | 100 tn years | 7qd years |

HIVE SYSTEMS

-Data sourced from HowSecureismyPassword.net