Topic List

Encryption (End-2-End)

a. E-2-E Encryption
b. Symmetric encryption
c. Asymmetric / symmetric
d. Diffie-Hellman
e. Signatures
f. HTTPS in depth

TOR and Onion Routing

a. Onion Routing
b. How its used in TOR?
c. TOR in depth (its own key sharing and encryption methods)
d. Potential attacks



LU CompSoc

End-to-End Encryption, TOR & Onion Routing

Tuesday 24 November in SAT B076 at 6pm



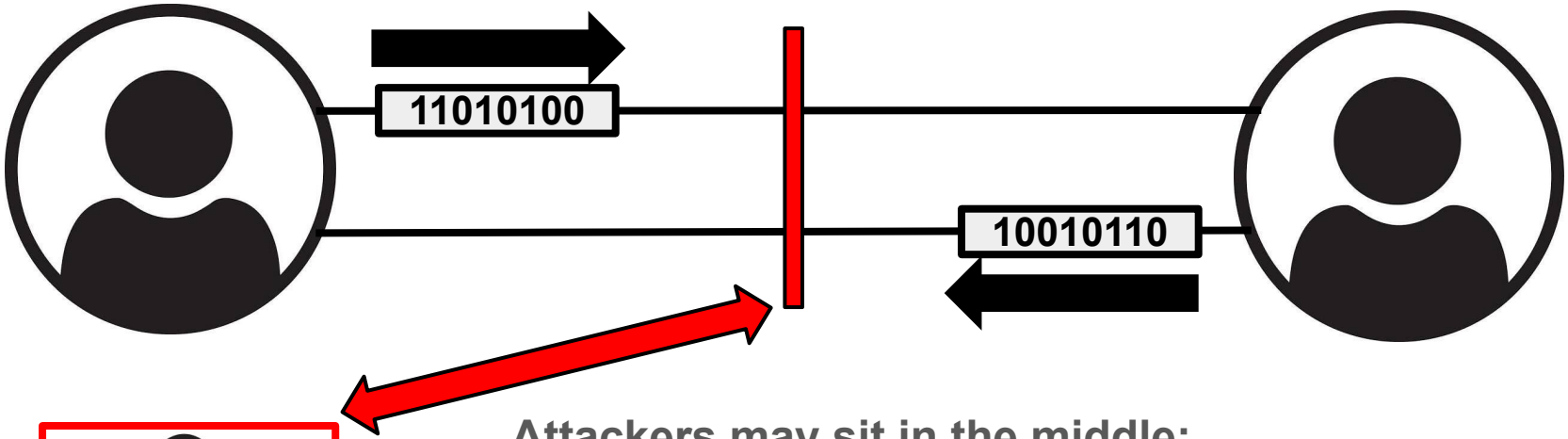






ALL WELCOME

End-to-End Encryption (E2EE)



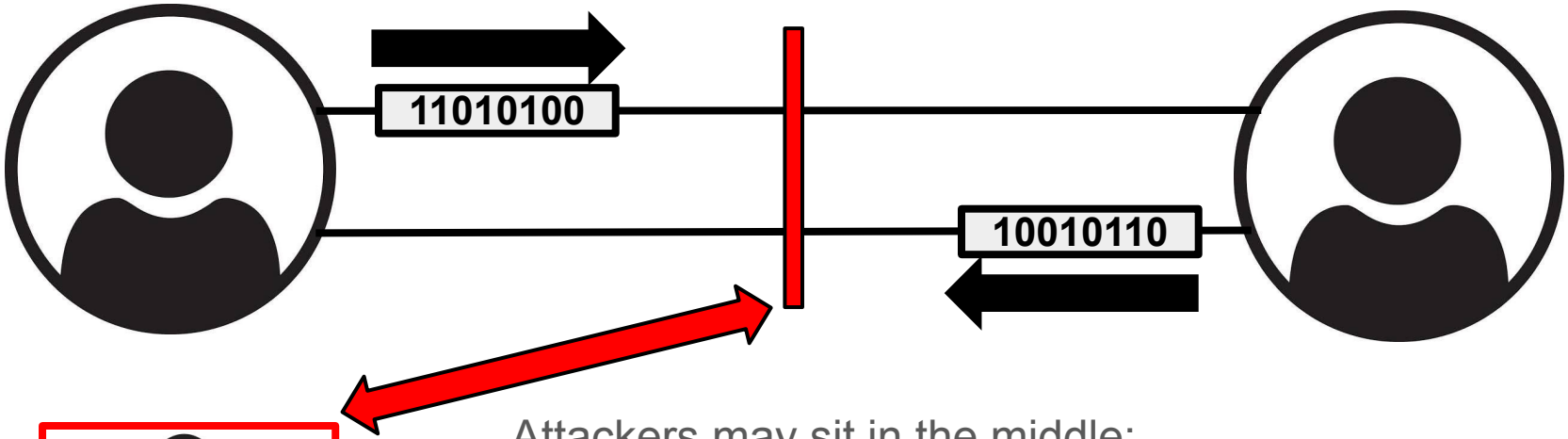


Attackers may sit in the middle:

- Read any messages
- Modify messages
- Pretend to be the recipient (spoofing)
- Delete messages entirely



End-to-End Encryption (E2EE)





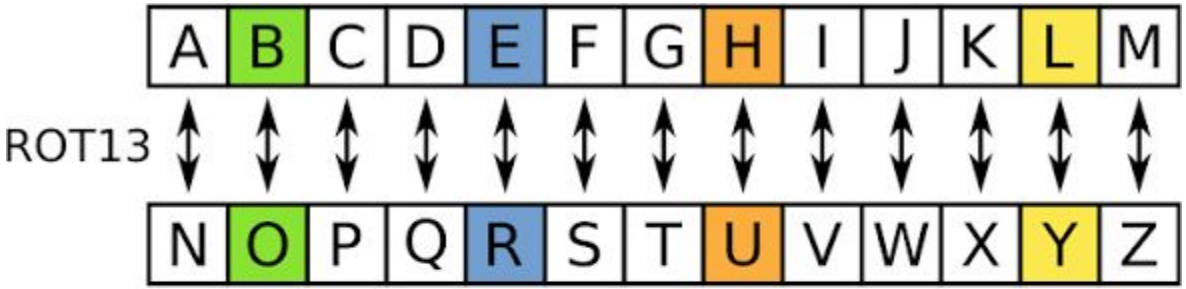
Attackers may sit in the middle:

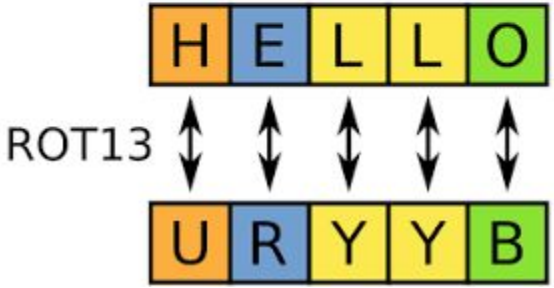
- **Read any messages**
- Modify messages
- Pretend to be the recipient (spoofing)
- Delete messages entirely



Basic Encryption

Caesar Cipher





Issues

- Easy to bruteforce
- Plaintext is too similar to the cipher
  - Frequency analysis
  - Length



A Better Encryption Method

Take a message in its binary form and a key of the same length…

Message: 11010100
Key: 10010110
Ciphertext: 01000010

This process is reversible…

Ciphertext: 01000010
Key: 10010110
Message: 11010100

XOR

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0



Symmetric Encryption

Encrypting by **XOR**ing the key and the message

Decrypting by **XOR**ing the key and the ciphertext

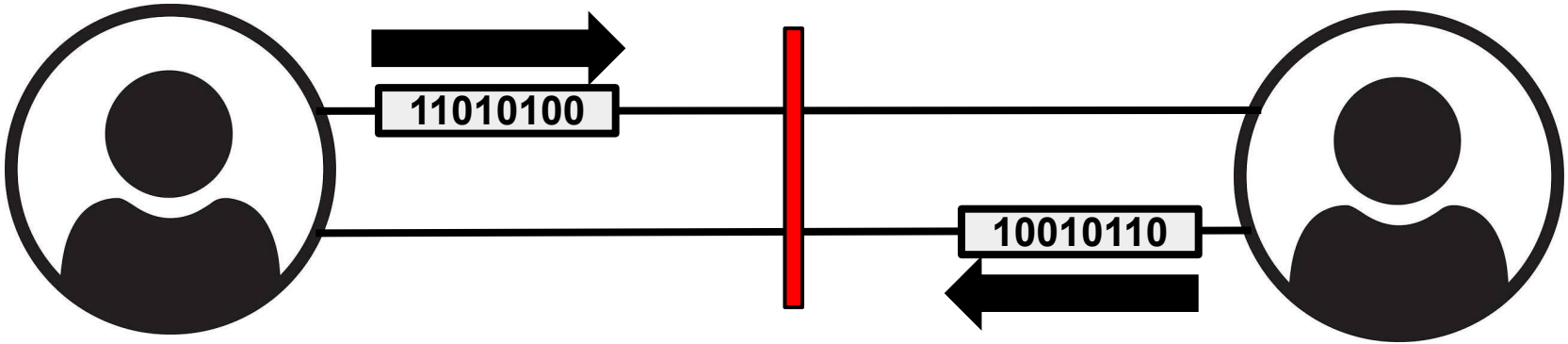
The **key** must be the same size (or bigger) as the message and **randomly generated**

If the key is **private**, **truly random** and only used **once**, this is the most secure encryption method





Issues when using just XOR



1. What if we send the same message twice?

2. How can we hide the length of the message?

3. How can send the recipient the key if we don't already have an encrypted channel?



What if we send the same message twice?

000100010001
000100010001
000100010001
000100010001

=

000100010001
000100010001
000100010001
000100010001

Symmetric encryption using the same key

011101001011
100101101011
010010101100
011010010110

=

011101001011
100101101011
010010101100
011010010110



How might an attacker use this?

Alice

00101010
10010010

Bob

00101010
10010010



Hello

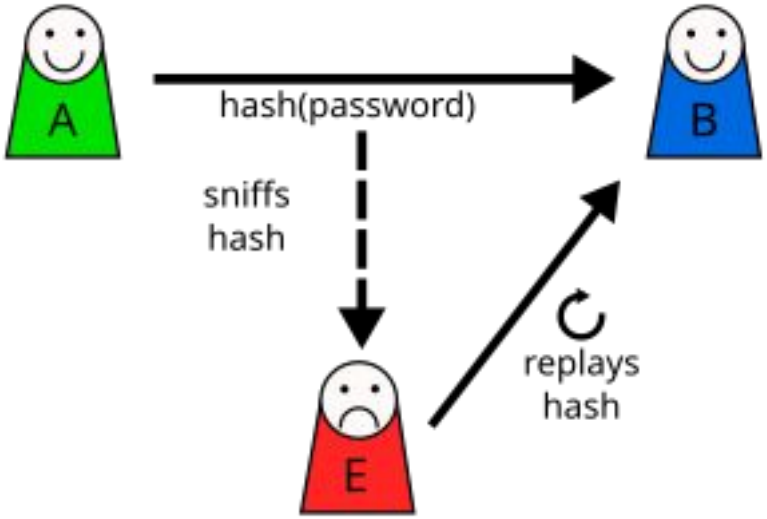
01110100
10111001
01101011
01001010
11000110
10010110

If we are using simple encryption methods, we can easily find a key using the **plaintext** and the **ciphertext**

With XOR, we can just XOR our guess and that ciphertext



Replay Attacks



In a **replay attack,** credentials (passwords, keys, tokens, session IDs) are sniffed by an attacker, then used later to impersonate the sender

It doesn't matter if this is hashed or encrypted in any way!

Alice sends proof of identity (a hashed password) to Bob. Eve (an attacker) later impersonates Alice using the hash she eavesdropped



Encryption with Nonces

A nonce is an arbitrary number, that is used exactly once as part of encryption

They could be:

- Time based
- A sequence number
- Completely random

000100010001
000100010001
000100010001
000100010001

Encryption using symmetric key

011101001011
100101101011
010010101100
011010010110

000100010001
000100010001
000100010001
000100010001

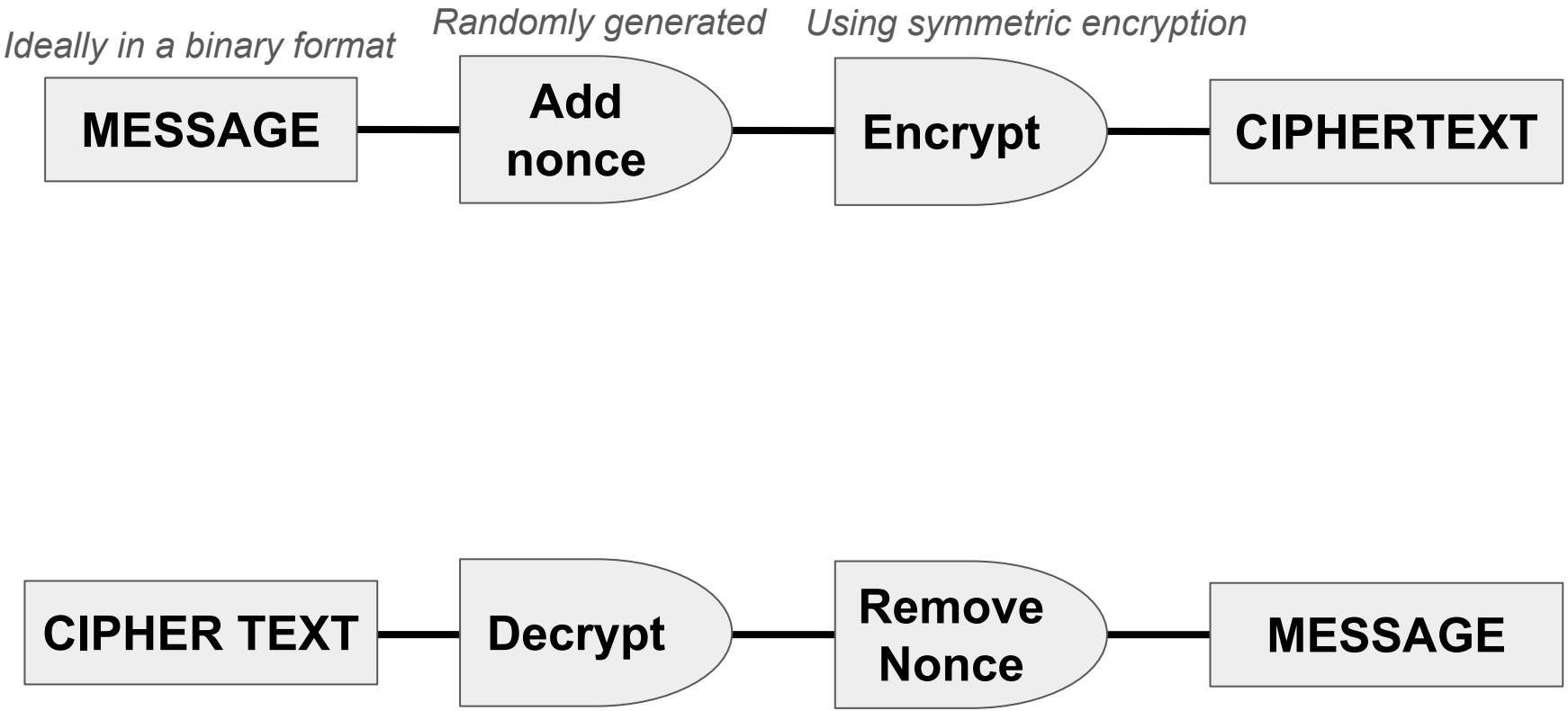
Encryption using symmetric key, and a random nonce

101100101101
011001001110
100101110110
010010110001

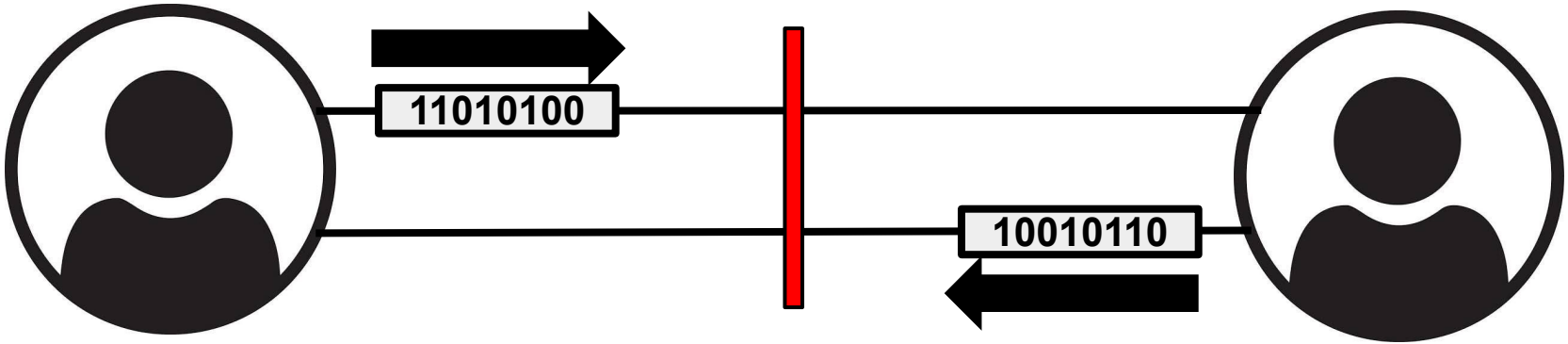
The nonce can be sent **with the message** in **plaintext**!



Our Pipeline



Issues when using just XOR



1. What if we send the same message twice?

2. **How can we hide the length of the message?**

3. How can send the recipient the key if we don't already have an encrypted channel?



How can we hide the length of the message?

Padding!

- Pick a block size (16/32 bytes)
- If a message doesn't fully fit we add extra bytes to the message

PKCS#7

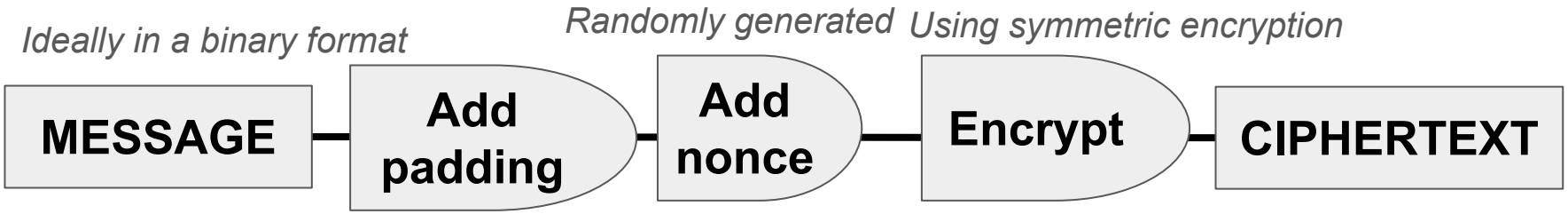
- Add bytes to the end of the message
- Each byte of padding is the total number of bytes added
- If the block size fits exactly, add a block of padding

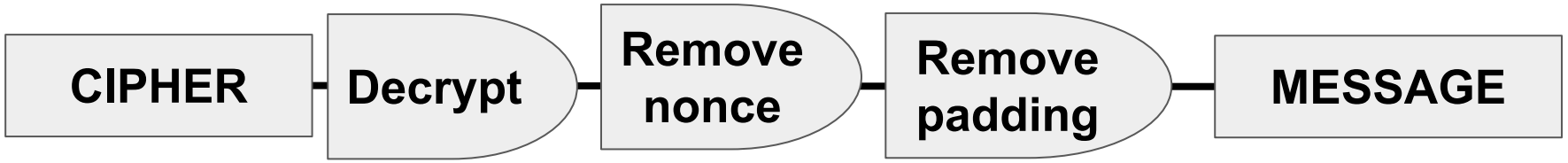
```
01
02 02
03 03 03
04 04 04 04
05 05 05 05 05
06 06 06 06 06 06
etc.
```

01110100
10111001
01101011
01001010
11000110
1001PPP

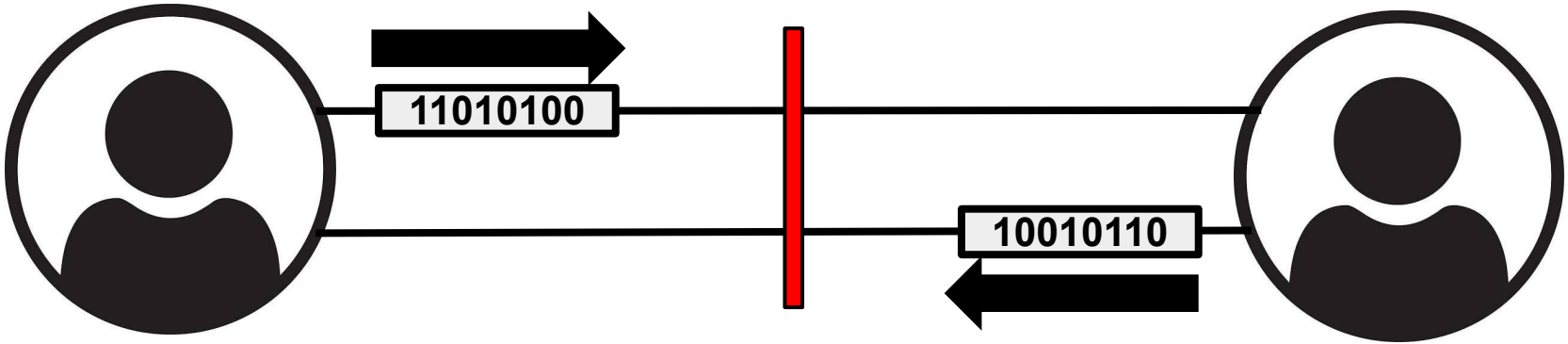


Our Pipeline





Issues when using just XOR



1. What if we send the same message twice?

2. How can we hide the length of the message?

3. **How can send the recipient the key if we don't already have an encrypted channel?**



Asymmetric Encryption

Also called public-key cryptography

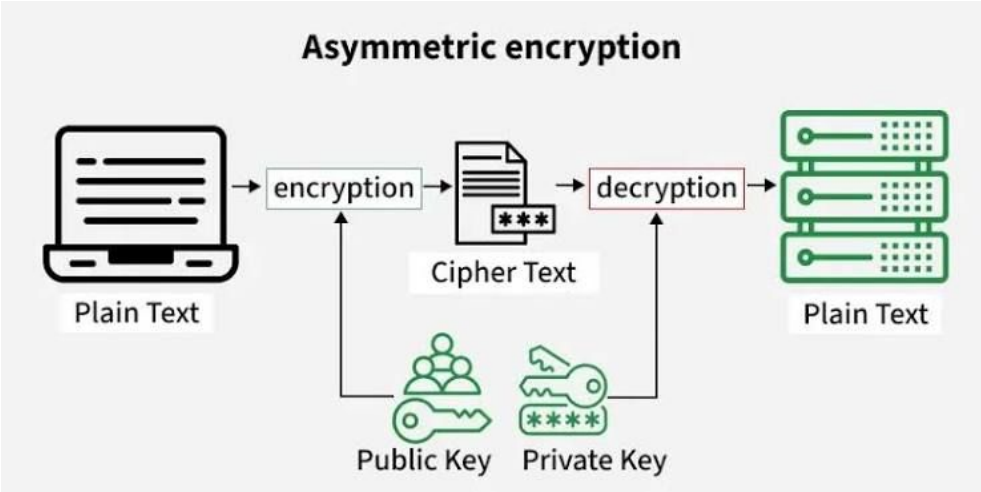
We have two mathematically linked keys:

- **Private key**
- **Public key**

These keys are usually significantly larger

We also need separate keys for sending messages in the other direction

Encryption is significantly slower





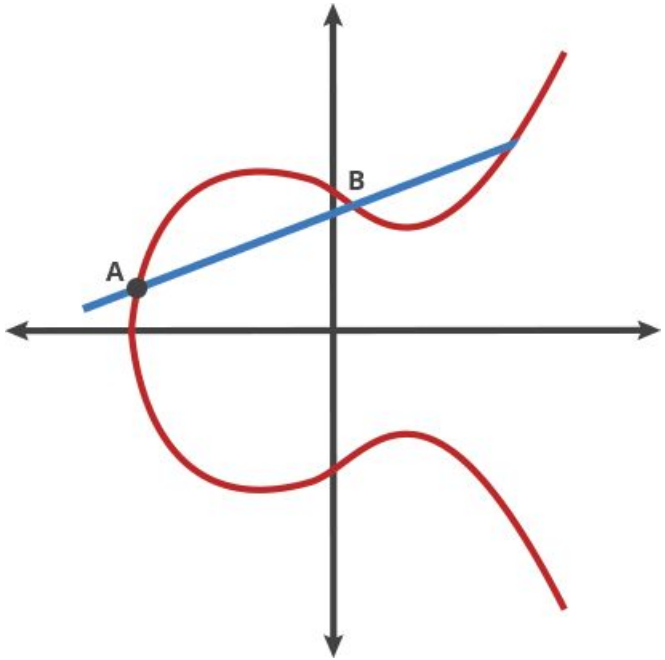
Key Generation

Key pairs are based around problems that are **easy to compute; but hard to reverse**

- Factoring the product of two large primes (RSA)
- **Discrete Logarithm** (Diffie-Hellman)
- Elliptic-Curve

Take a prime *p*, generator *g*, and our private key *a*

Our public key is $\mathbf{g^a \bmod p}$





Key sharing (Diffie-Hellman)

Public-key cryptography still doesn't address establishing a symmetric key over an unencrypted channel

Alice

Pick a private key *a*

Find the public key
$g^a \bmod p$

Combines Bob's public key with her private key

$(g^b \bmod p)^a \bmod p$

= = $g^{ba} \bmod p$

Public

Generator g, prime p

Alice's Public Key
$g^a \bmod p$
Bob's Public Key
$g^b \bmod p$

Bob

Pick a private key *b*

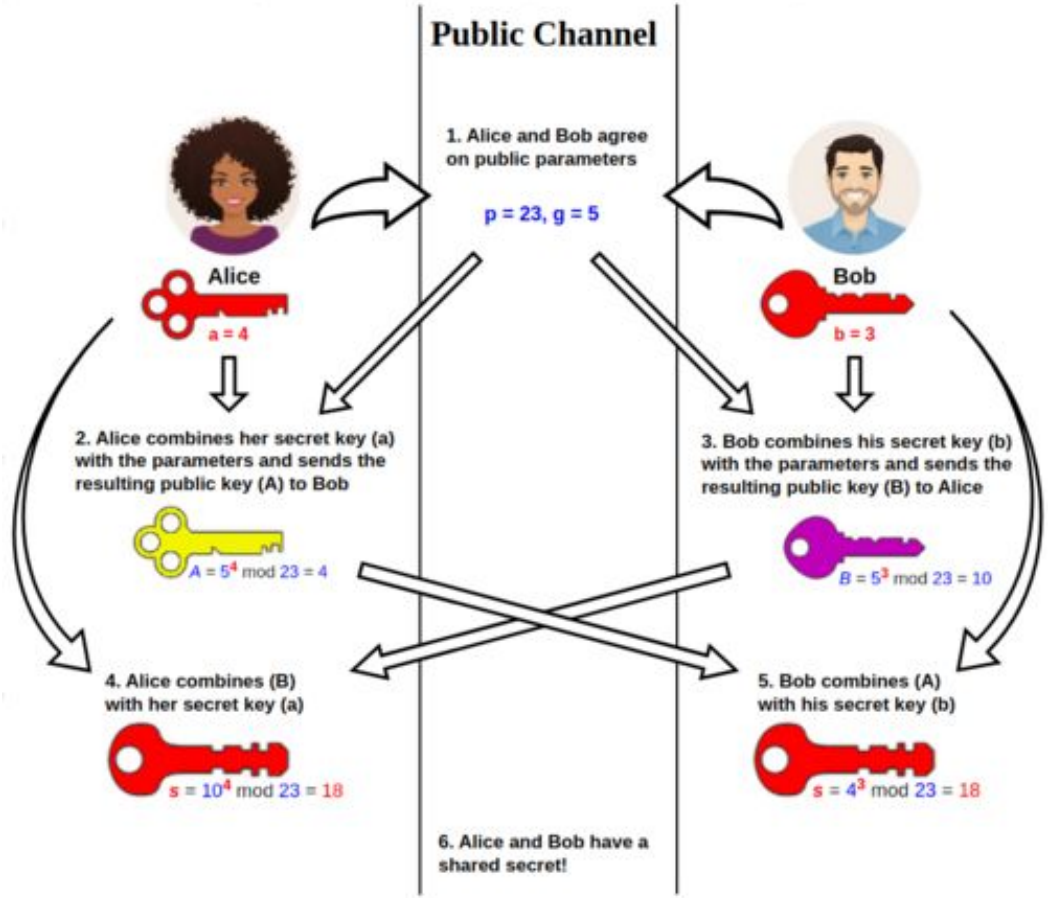
Find the public key
$g^b \bmod p$

Combines Alice's public key with her private key

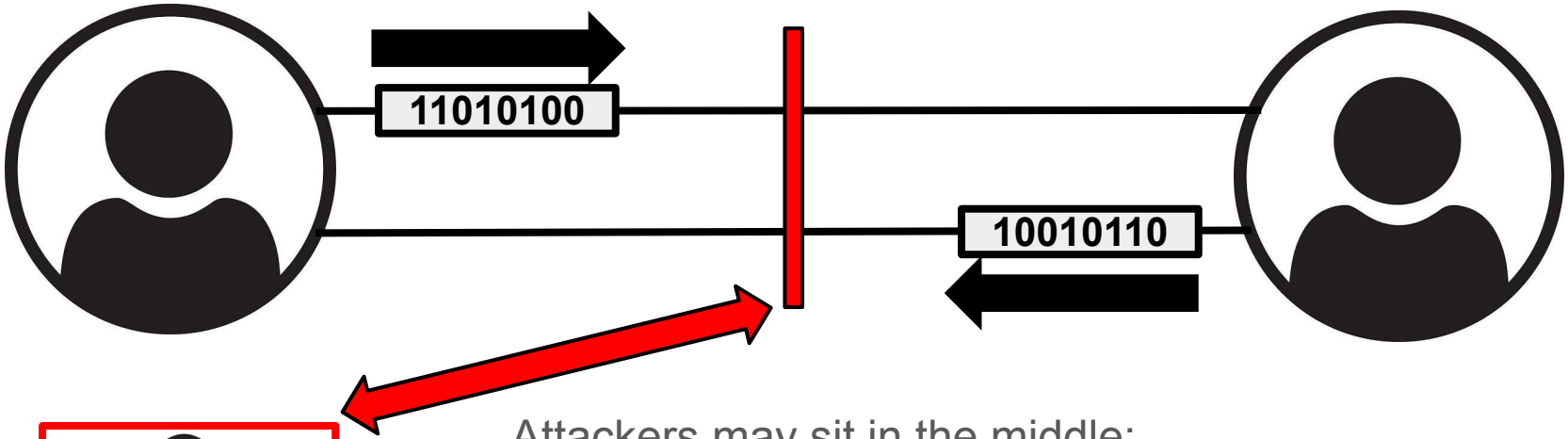
$(g^a \bmod p)^b \bmod p$

= $g^{ab} \bmod p$

Key sharing (Diffie-Hellman)



End-to-End Encryption (E2EE)





Attackers may sit in the middle:

- Read any messages
- **Modify messages**
- Pretend to be the recipient (spoofing)



Signatures

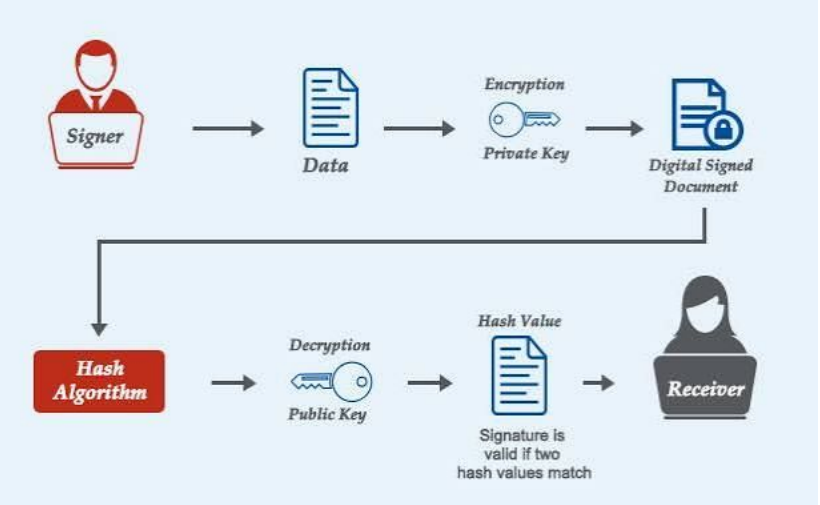
A **signature** can be added to verify its integrity.

Signatures use **public key cryptography**:

- The hash digest of the message is encrypted with the **private key** to create the signature
- The receiver uses the **public-key** to decrypt, then compares the digest with its own computed hash of the message

Why do we use the hash?

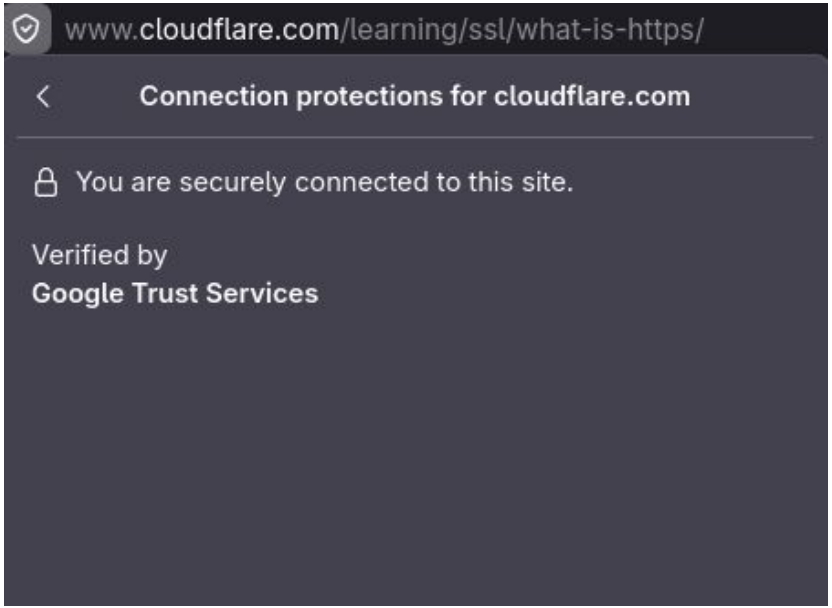
- Shorter (always the same size)
- Faster
- Deterministic



HTTPS

HTTPS is a secure version of the HTTP protocol.

It encrypts all communications end-to-end using the **TLS protocol**





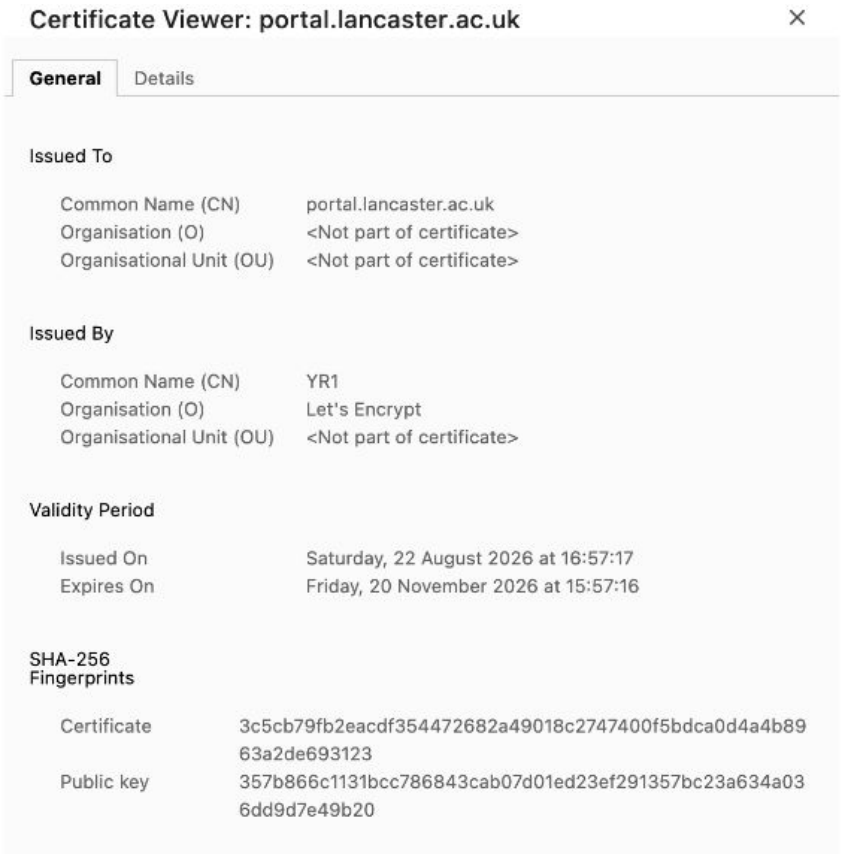
Transport Layer Security (TLS)

TLS Provide

- Confidentiality
- Integrity
- Authentication

A website using TLS must have a **TLS certificate.** This states who owns the domain and the public key.

The certificate is signed by a **certificate authority**



Certificate Authorities (Chain of Trust)



Each certificate contains:

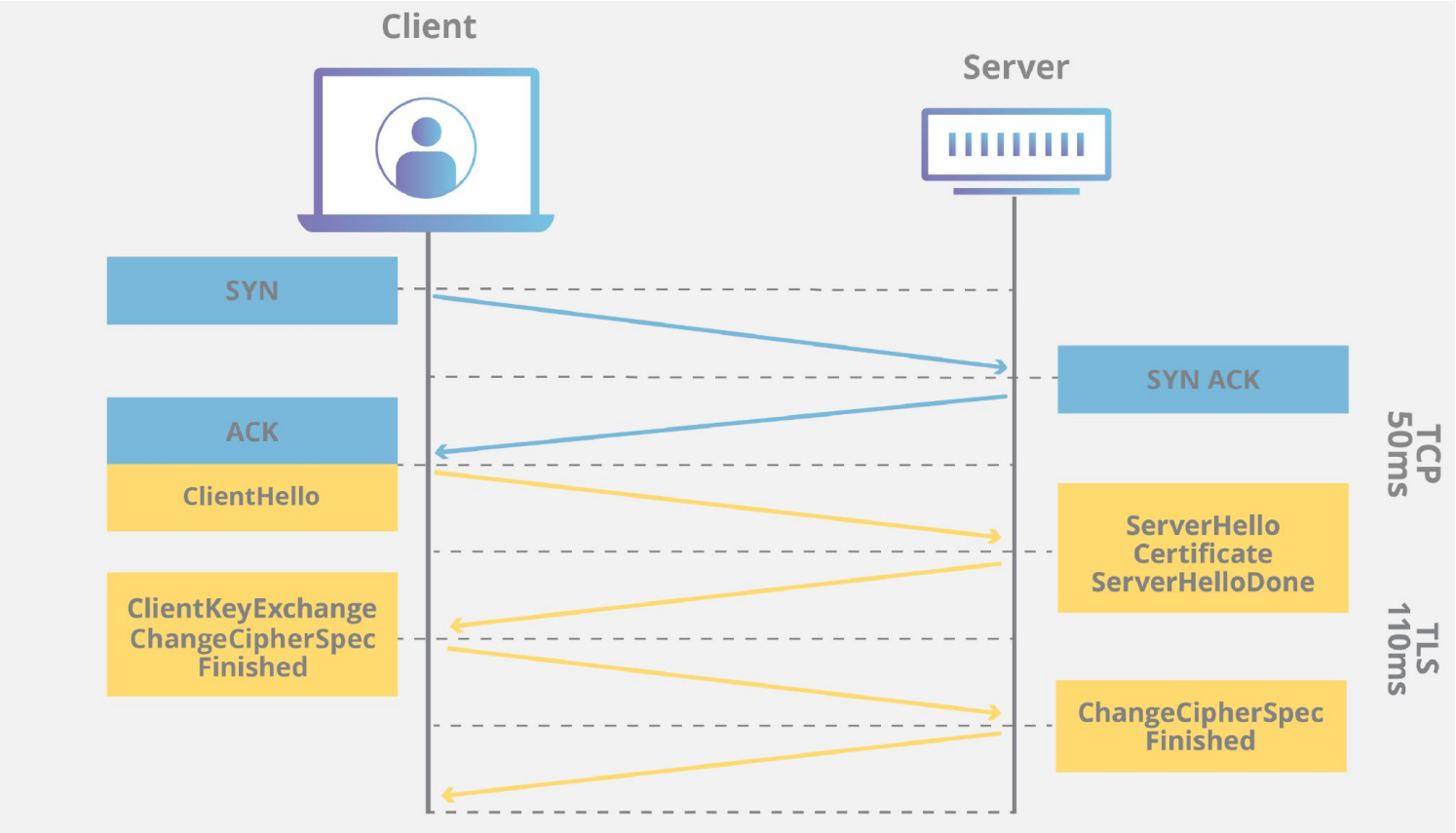
- The owner's domain name
- The owner's public key
- The issuer's domain name
- The issuer's signature

The root certificate signs itself

There are only a handful of root CAs trusted by browsers. Their keys are embedded into the browser so they are verified



TLS Handshake





Client & Server Hello

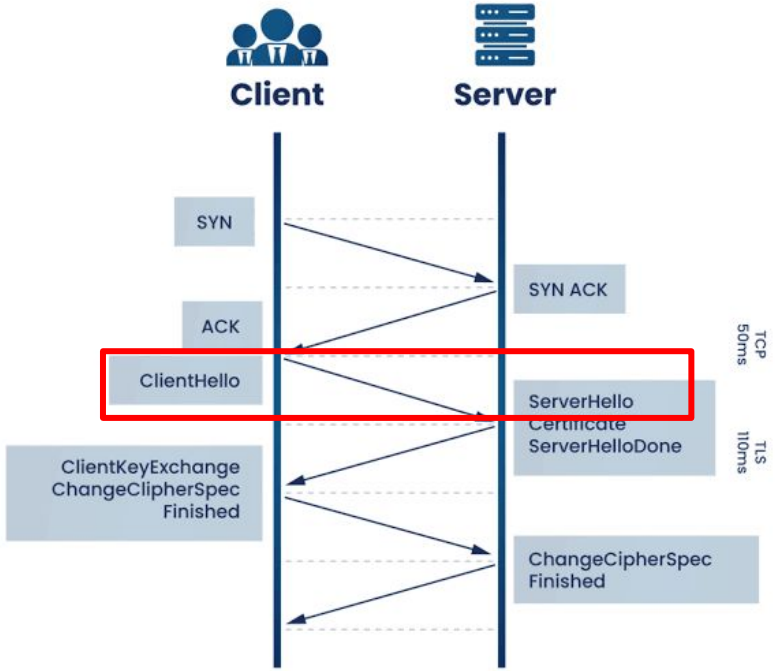
The Client sends:

- A list of the supported ciphersuites
- A 32-byte nonce
- A Diffie-Hellman keyshare

The server sends back

- The chosen ciphersuite
- A 32-byte nonce
- Its Diffie-Hellman

All future communication is encryption



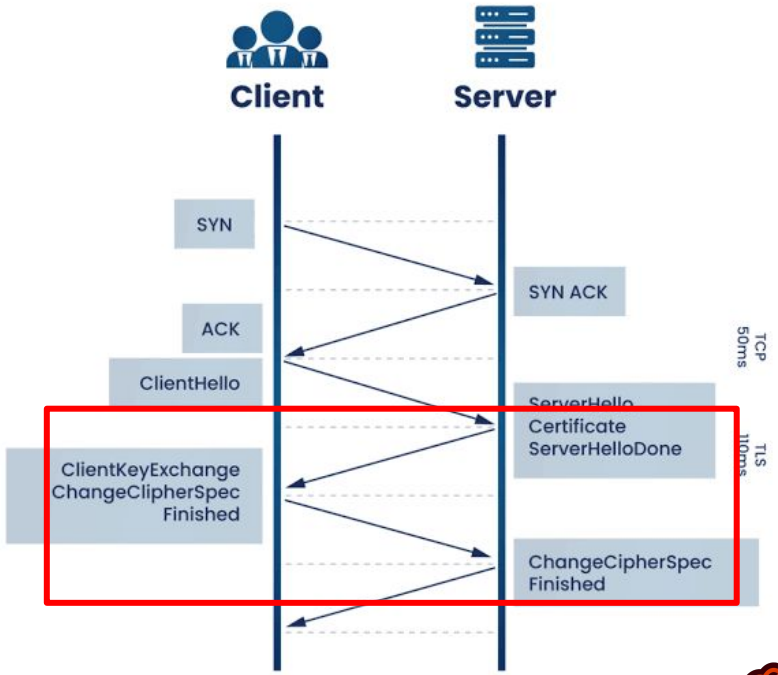


TLS Handshake

The server then sends its certificate; the client:

- Follows the chain of trust to the root
- Checks that the domain name matches where the certificate is coming from

The client and server than exchange the ‘finished’ message, which verifies that everything is set up correctly.



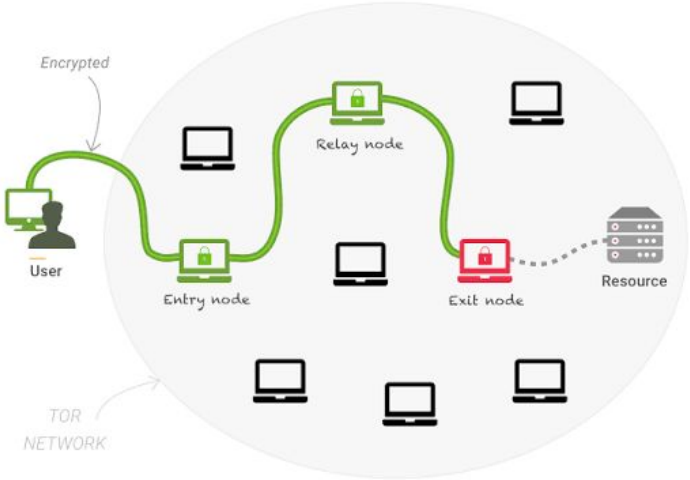


Introducing the Tor Network

Unlike HTTPS, Tor attempts to anonymise internet traffic so that no one can see even what websites you're visiting

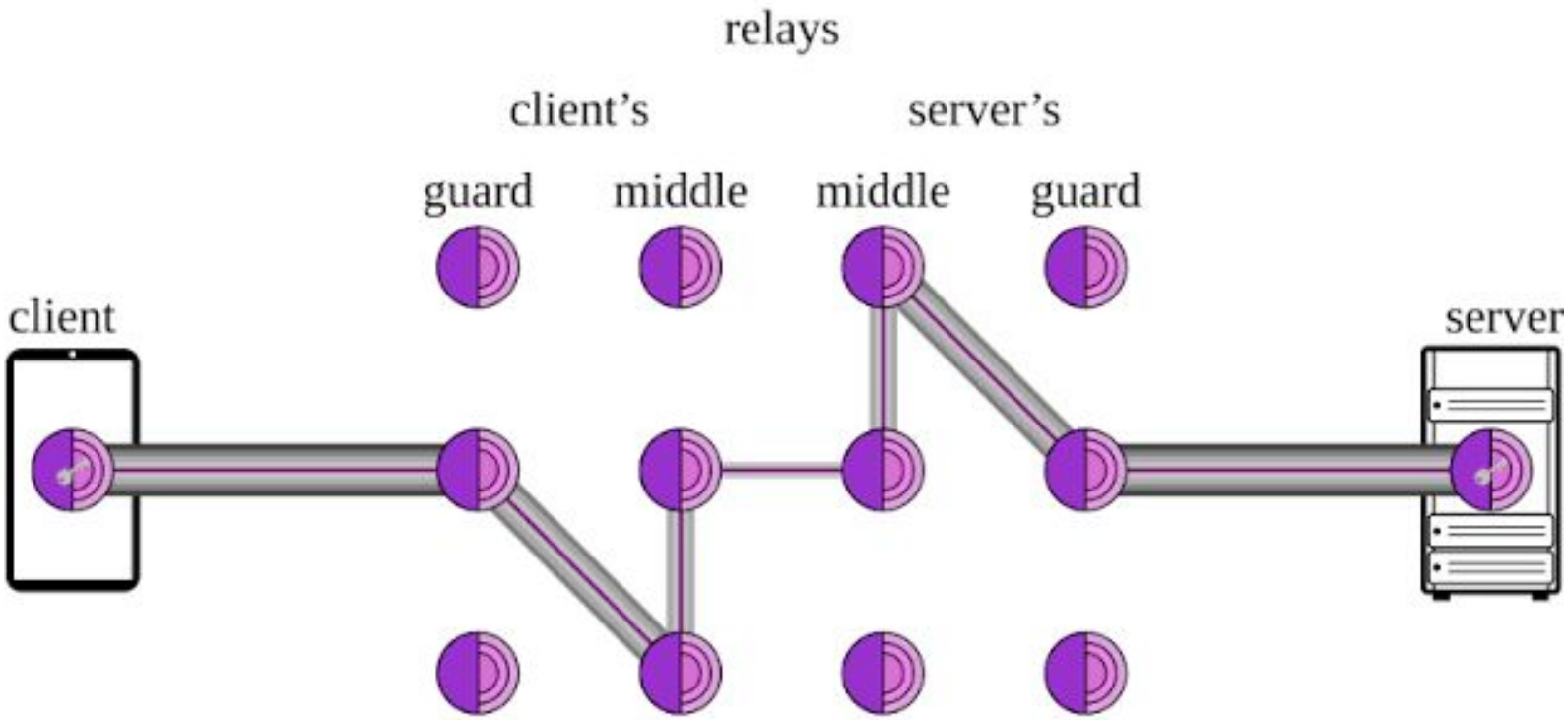
How it works

- Volunteers run relays all around the world
- When you connect to a website, your data runs through these tunnels through various different relays
- It uses onion routing to keep you anonymous: no single relay knows who you are **AND** what you are connecting to



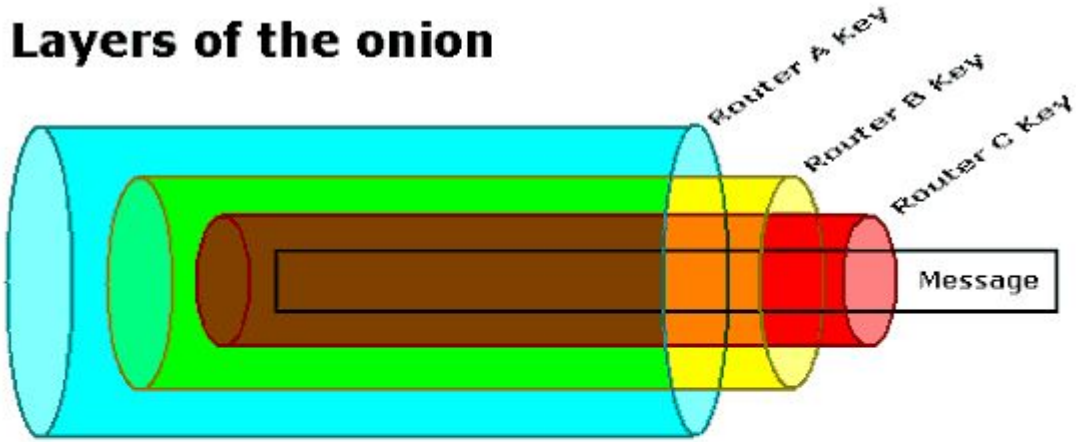


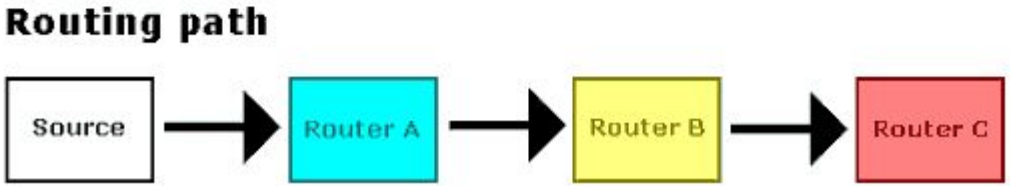
Tor Nodes


relays
client's
server's
guard
middle
middle
guard
client
server

Onion Routing







Onion Routing in Practice

You will receive a **symmetric key** for each relay

Before you send a request over the network: you encrypt it for each relay it will pass through, **adding a layer of encryption**

Each relay will remove the **top layer of encryption using your shared key**

The last node will end with your request. It will make this request





Onion Routing in Practice

As the message goes back through the network, each relay **adds a layer of encryption**

You will need to remove each layer of encryption using all of the shared keys you have



Onion Routing Demo

```
[Ben@laptop-arch CompSoc]$ python3 onion-routing.py
start client sending: b'This is a test message!'
start client padded: b'This is a test message!\t\t\t\t\t\t\t\t\t'
start client: sending request: b'\xa2\xe7\xc2\x03<\x9f\x07\xa0V\xb5\xa1\r\xe8\xbf\xceb\xd0\xf5\x0c/W\xd6\r\x06\x08\xdb\x82X+Mr\xc0'
relay 1: received request: b'\xa2\xe7\xc2\x03<\x9f\x07\xa0V\xb5\xa1\r\xe8\xbf\xceb\xd0\xf5\x0c/W\xd6\r\x06\x08\xdb\x82X+Mr\xc0'
relay 1: sending request: b"\x90\xfc\x0e\xf1Z\xaa\xf3\xf6\x9f\xd5v,1\xd3i\x950'\xfa*\xbd\xbc'\x05\xe7A\xf0\xb4\x91\xbaM\xcb"
relay 2: received request: b"\x90\xfc\x0e\xf1Z\xaa\xf3\xf6\x9f\xd5v,1\xd3i\x950'\xfa*\xbd\xbc'\x05\xe7A\xf0\xb4\x91\xbaM\xcb"
relay 2: sending request: b'\xd5\xe8\xef2\x1f@yh\xb8)\x0e0\xd93\x9e\xa7\xb5\xe1s\x8e\xc6\x17p\x9eM\xc1\x046\xfc\xf6\xe0)'
relay 3: received request: b'\xd5\xe8\xef2\x1f@yh\xb8)\x0e0\xd93\x9e\xa7\xb5\xe1s\x8e\xc6\x17p\x9eM\xc1\x046\xfc\xf6\xe0)'
relay 3: sending request: b'This is a test message!\t\t\t\t\t\t\t\t\t'
end client received: b'This is a test message!\t\t\t\t\t\t\t\t\t'
end client received: This is a test message!
end client responding: b'This is a test message!'
end client padded: b'This is a test response!\x08\x08\x08\x08\x08\x08\x08\x08'
relay 3: received response: b'This is a test response!\x08\x08\x08\x08\x08\x08\x08\x08'
relay 3: sending response: b'\xe4\xf6\xf9\x90\x1e\xd0=\xfdI\x1e\xad~\x07\x8f\xcaG\xce\xe4+\xa1G;\xce\xb2Wl,\xfb\xb4\xb8\xbf\xee'
relay 2: received response: b'\xe4\xf6\xf9\x90\x1e\xd0=\xfdI\x1e\xad~\x07\x8f\xcaG\xce\xe4+\xa1G;\xce\xb2Wl,\xfb\xb4\xb8\xbf\xee'
relay 2: sending response: b'\xc3\xbd\x1b_e\x03\xe3\xbb\x17\x84.[\xdf\x94\xc7\x93\xef\xfb~\x01\xee\x06\xfbr\xbaZu\x10\nq\x138'
relay 1: received response: b'\xc3\xbd\x1b_e\x03\xe3\xbb\x17\x84.[\xdf\x94\xc7\x93\xef\xfb~\x01\xee\x06\xfbr\xbaZu\x10\nq\x138'
relay 1: sending response: b')\xdab\xe9S;\x87\xde\xad\xeb\xb4EN\x11\xc0 \x1f\xf9\x8d_\x1b)\xfb\xd8\x89\xd3<{\x93\xb0\x19^'
start client: received response: b')\xdab\xe9S;\x87\xde\xad\xeb\xb4EN\x11\xc0 \x1f\xf9\x8d_\x1b)\xfb\xd8\x89\xd3<{\x93\xb0\x19^'
start client received response: This is a test response!
```

Vulnerabilities in Tor

You have to be careful before the **guard node** and after the **exit node**.

If someone was monitoring both, then there is essentially no benefit from Tor

Timing attacks can occur, looking at the frequency and gaps between sequential

Tor has its own **chain of trust (CAs)**

- Relays are signed by a **directory authority**
- **Tor browser** has the directory authorities included



Your activity!

Your task

- Clone the repo ‘onion-routing-template’ from our GitHub (linked on the website)
- https://github.com/LUCompSoc/onion-routing-template
- Complete the padding and unpadding functions
- Complete the encryption and decryption functions on the relay

Experiment with adding more relay nodes in the main function. The functionality should not change
(let me know if you need help with this)

https://compsoc.io/w8

If you’ve done that…

- Implement CBC Mode, using an initialisation vector instead of ECB Mode
- Expand the functionality of the end node to respond to certain queries

If you’ve done that…

- Implement AES-GCM