

Introduction to Encryption

Welcome

Michael Pound

- Associate Professor at the University of Nottingham, UK
- Teach the final year Security and Cryptography modules on our degrees

Email: michael.pound@nottingham.ac.uk

Twitter: [@_mikepound](https://twitter.com/_mikepound)

Videos: [YouTube.com/computerphile](https://www.youtube.com/computerphile)



About This Course

- This course will give you an understanding of the **foundations of cryptography**
- We'll cover **the most common primitives**, and what they're for
- Experience **using these tools** within a programming language
- Insight into development of **secure protocols** and systems
- Knowledge of **common mistakes** and pitfalls

Some Questions Answered

When would I use symmetric or public key cryptography?

How would I store a password?

How do I safely transfer a secret key?

How does HTTPS work?

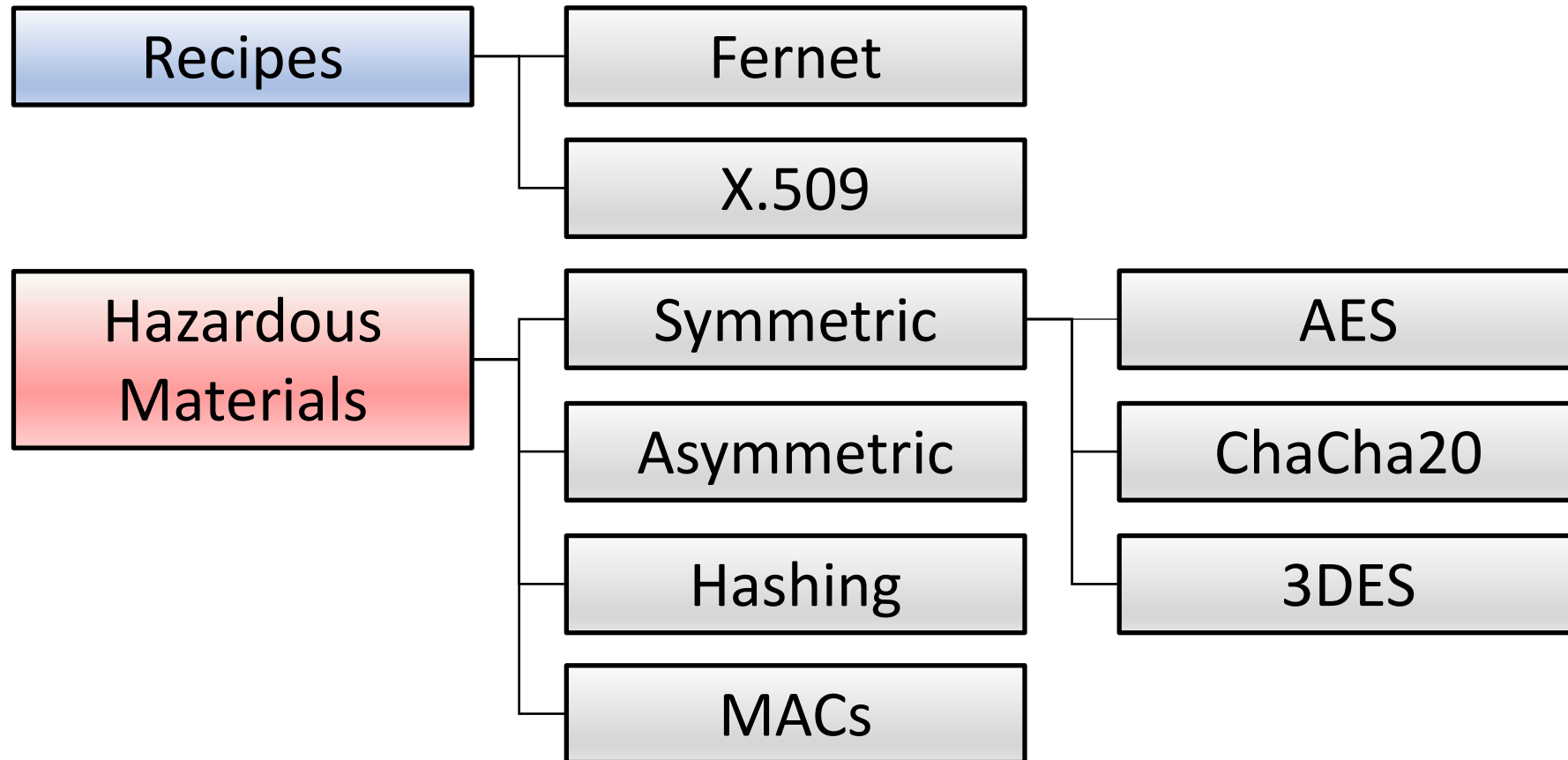
What do we do if our private key is leaked?

The Exercises

- The course materials come with exercises in Python
 - You only need a [general grasp of python](#) to have a go
 - [Answers are included](#) – try to avoid peeking!
- Each exercise is a short python script with code missing, and comments pointing you in the right direction
- Don't worry if you don't finish them in the time, and by all means do more after the course!

Python Cryptography

- The python Cryptography module can be found at <https://cryptography.io>

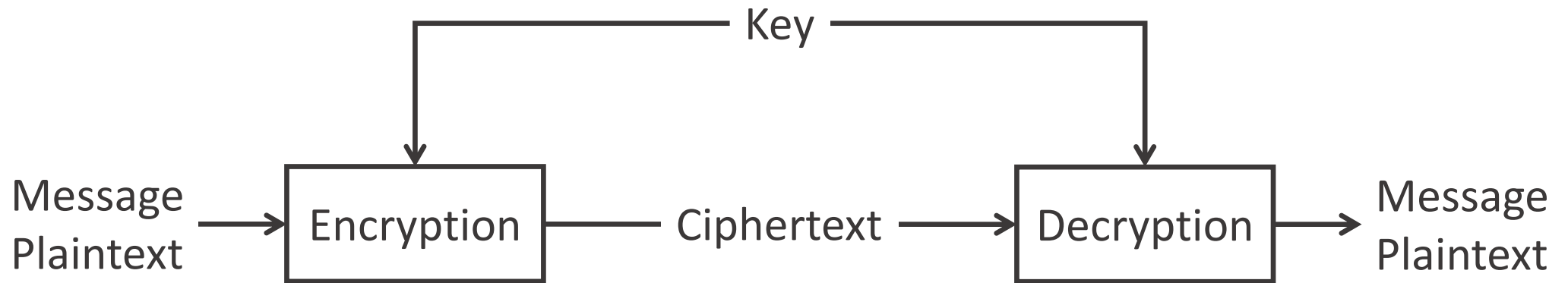


When to use Encryption?

- User / Company Database Records
- Cloud Storage
- Password storage
- Compliance
- In transit:
 - Network communication (e.g. inter-application)
 - Authentication data
 - Payment and money transfers

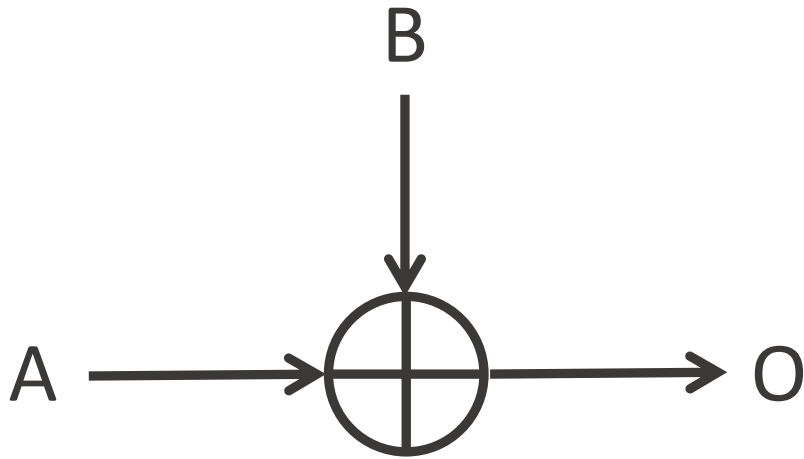
Not an exhaustive list!

Some Terminology



The Power of XOR

“XOR is a binary operator between two values that returns true if either one input or the other is true, but not both”



A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

The Power of XOR

- Applying XOR twice, reverses its effect:

$$A \oplus B \oplus A = B$$

- Think of A as encrypting B, and then decrypting it again

The One Time Pad

Can we design a cipher that uses XOR to encrypt and decrypt a message?

- Use a key that's the same length as the message
- XOR each message bit with each key bit

$$\begin{array}{rcl} M & \text{[noise]} & \\ & \oplus & \\ K & \text{[noise]} & \\ & = & \\ C & 00010001 & 10001100 \end{array}$$

Perfect Secrecy

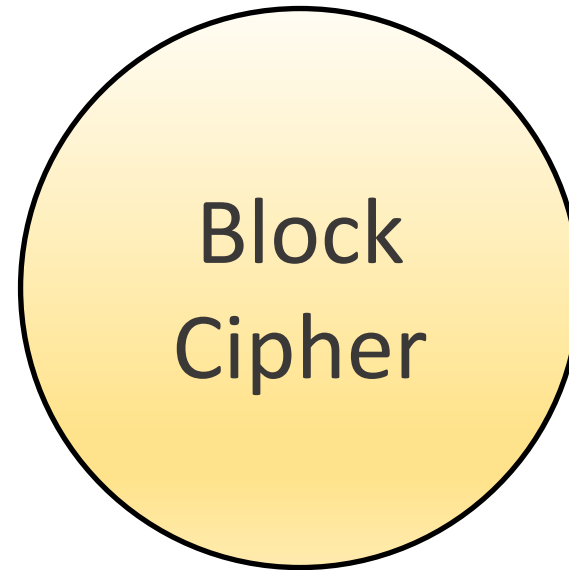
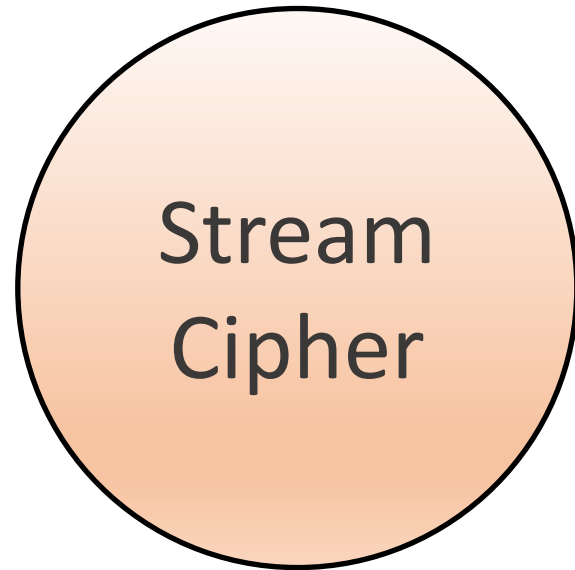
But...

- The one time pad is **not practical**:
 - A 1GB file would need a 1GB key!
 - How are we transporting these keys? Or storing them?
 - If you ever **reuse a key**, the entire cipher is **broken**

Symmetric Cryptography

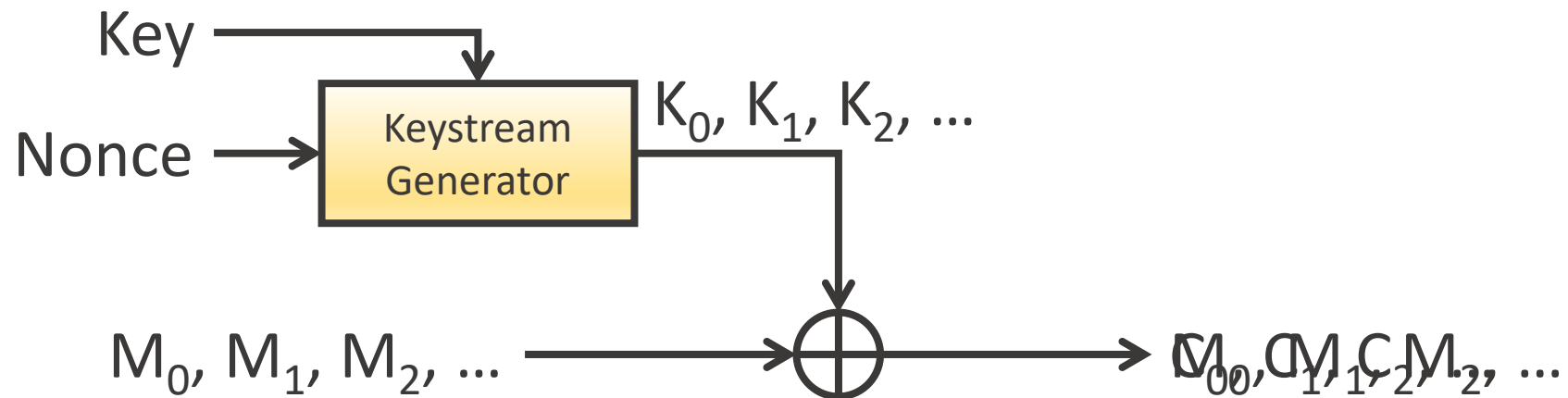
Symmetric Primitives

- Symmetric cryptography broadly splits into two types



Stream Ciphers

- We can approximate a one-time pad by generating an infinite **pseudo-random keystream**
- Stream ciphers work on messages of any length



Pros and Cons of Stream Ciphers

- + Encrypting **long continuous streams**, possibly of unknown length
- + Extremely **fast** with a **low memory** footprint, ideal for low-power devices
- + If designed well, can **seek** to any location **in the stream**
- The keystream **must appear statistically random**
- You must **never** reuse a key + nonce
- Stream ciphers **do not protect the ciphertext**

Modern Block Ciphers

- Block ciphers encrypt a fixed sized block
- Attempt to confuse an attacker with **confusion** and **diffusion**
- The Advanced Encryption Standard (AES) is the most popular

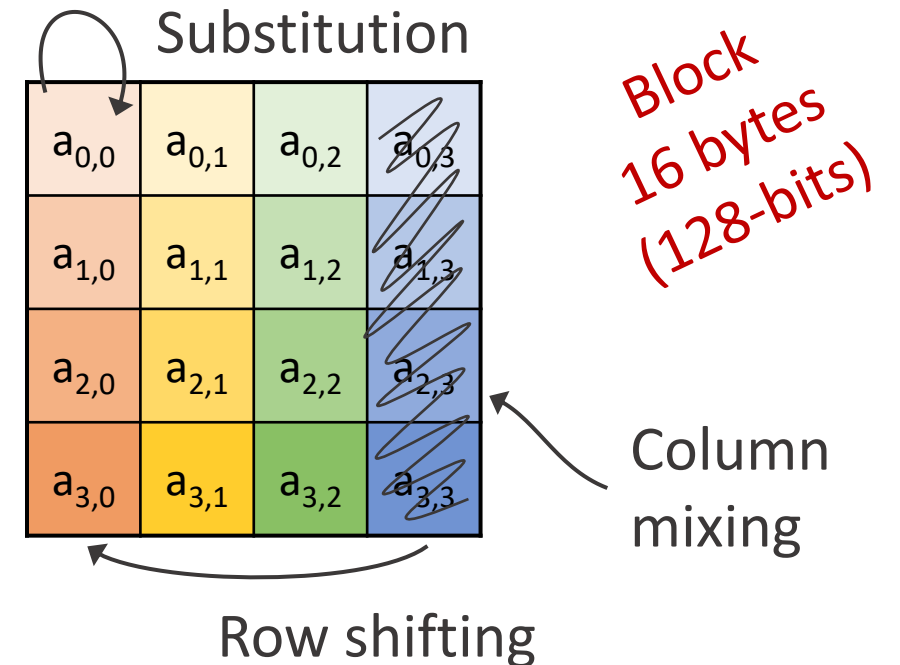
The take home message:

Almost everything uses AES

AES

- Superseded DES as a standard in 2002
- A standard built around the **Rijndael** algorithm

- 128-bit block size
- Key length of 128, 192 or 256-bits
- 10, 12 or 14 rounds



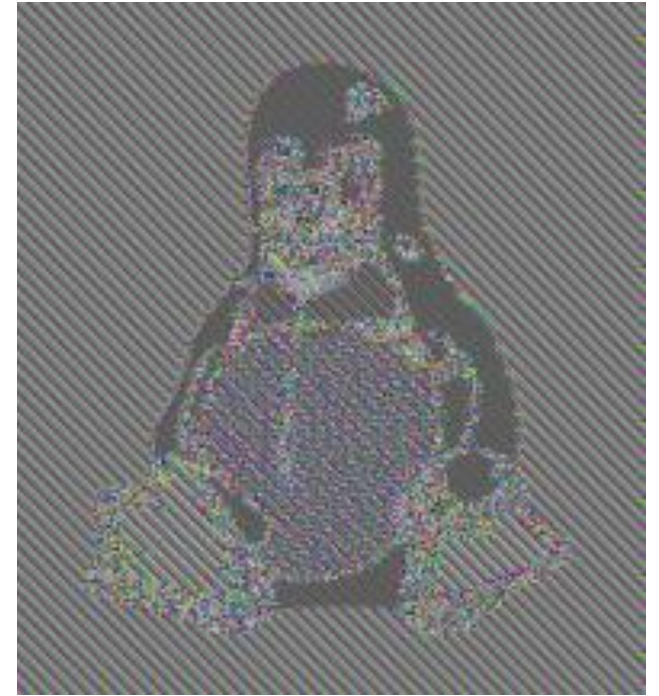
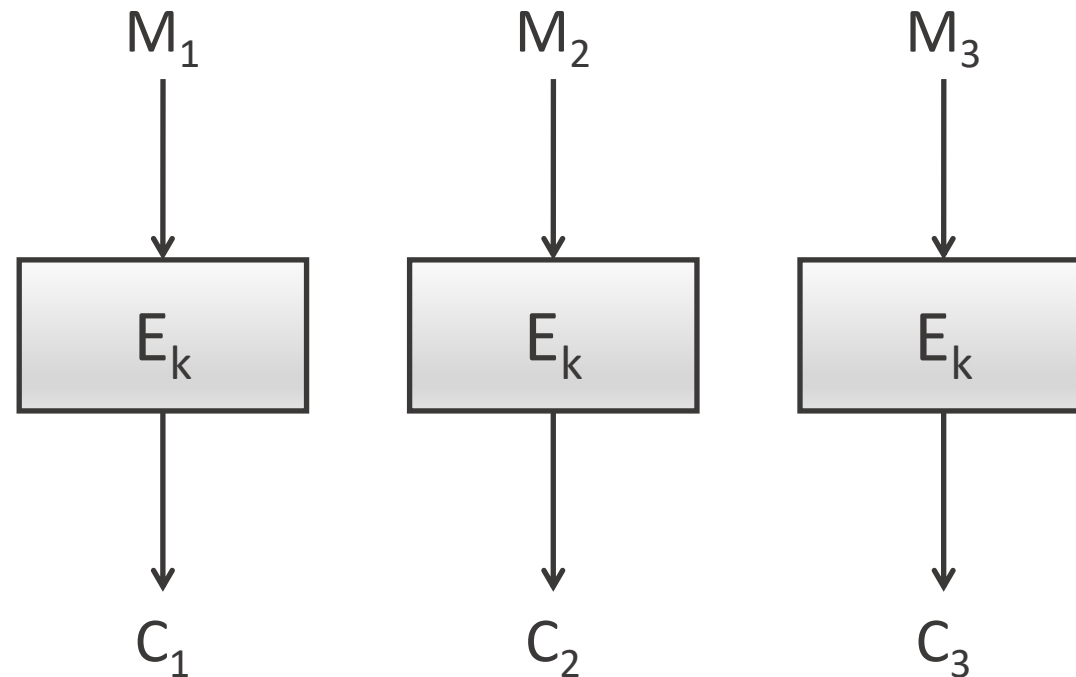
Modes of Operation

- Realistically, messages of exactly 128-bits are pretty unlikely
- We need some mechanism to encrypt messages that are longer or shorter

Modes of operation combine multiple instances of block encryption into a usable protocol

Electronic Code Book (ECB)

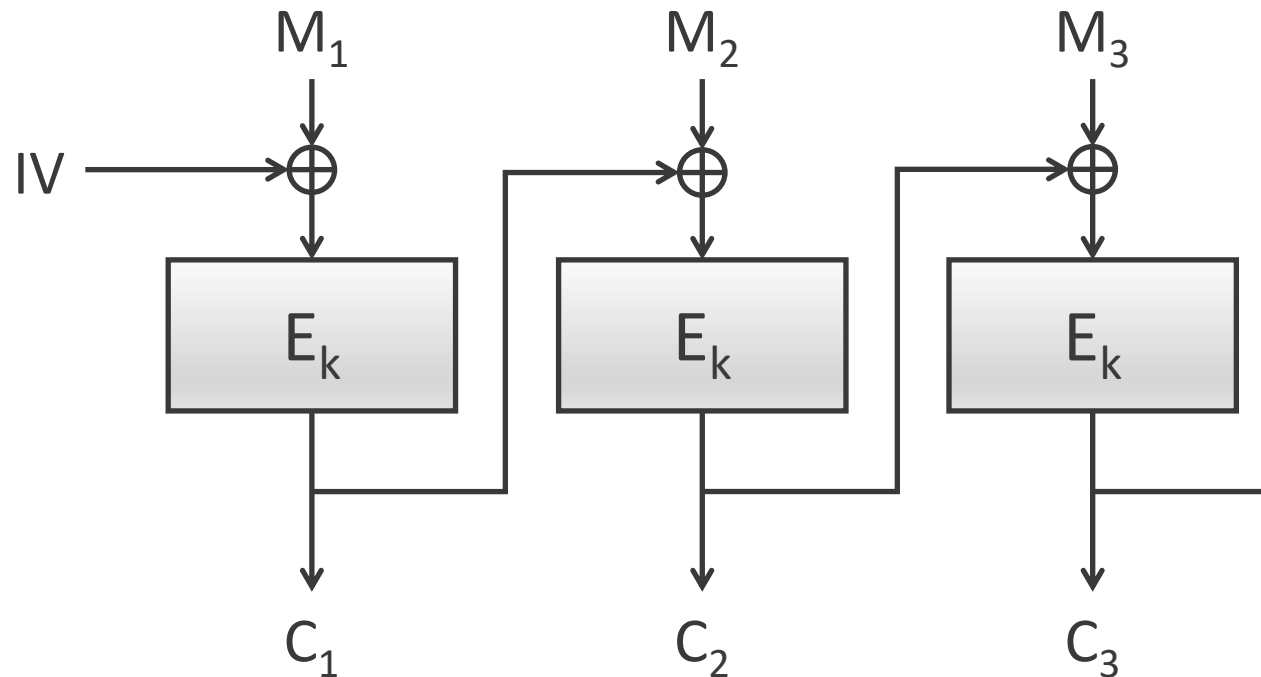
- Just encrypt each block one after another
 - Weak to redundant data divulging patterns



The ECB Penguin

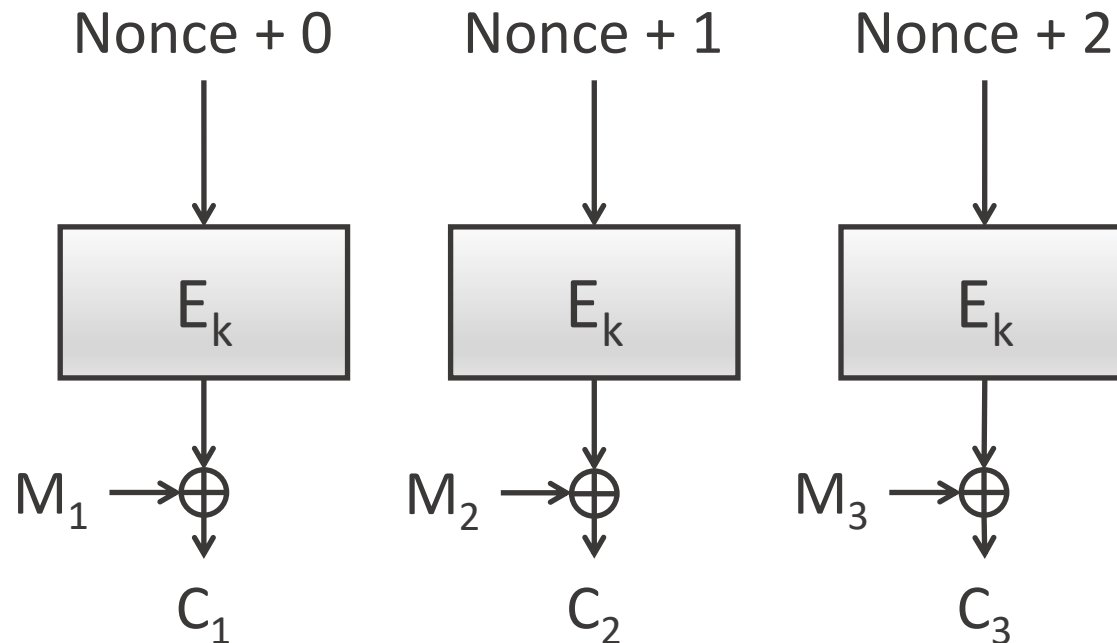
Cipher Block Chaining (CBC)

- XOR the output of each cipher block with the next input
 - Lots of problems with this!



Counter Mode (CTR)

- Encrypting a counter to produce a **stream cipher**
 - Pretty good - can also be parallelized!



Using AES

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

key = os.urandom(32)
iv = os.urandom(16) # Not used in ECB mode

message = b"a secret message" # Exactly 128-bits long luckily!

algorithm = algorithms.AES(key)
cipher = Cipher(algorithm, mode=modes.ECB())

encryptor = cipher.encryptor()
ct = encryptor.update(message) + encryptor.finalize()
decryptor = cipher.decryptor()
pt = decryptor.update(ct) + decryptor.finalize()
print (pt)
```

```
'a secret message'
```

Exercise – Symmetric Encryption

Encrypt and decrypt messages
using AES in CBC and CTR modes

`symmetric.py`

```
### Task 1 ###  
# Now it's your turn! CBC uses a similar interface to ECB, except that it requires both a key,  
# Initialise these randomly now. Make the key 32 bytes and the IV 16 bytes.  
key = None  
iv = None  
  
# Now fill in the code here to encrypt the same message as ECB, remember to use the CBC.  
cipher = None  
encryptor = None  
ciphertext = None  
decryptor = None  
plaintext = None
```


Asymmetric Cryptography

What's Asymmetric Crypto For?

Symmetric cryptography has **some remaining issues**:

1. How do we establish a shared secret key?
2. How do we guarantee who's on the other end of the line?

For later:

3. How do we know someone hasn't tampered with our message?

Public-key Cryptography

- Two keys, a public key and a private key
- Public-key (asymmetric) cryptography hinges upon the premise that:

It is computationally infeasible to calculate a private from a public key

- In practice this is achieved through **intractable mathematical problems**

Key Exchange

Diffie-Hellman

- Two parties can jointly agree a **shared secret** over an **insecure channel**
- DH-KEX underpins almost every aspect of our modern lives – it is incredible!

Your phone is probably negotiating a key exchange right now...

Diffie-Hellman: Practice

1. Alice and Bob agree on a set of base parameters

Alice

Bob

```
from cryptography.hazmat.primitives.asymmetric import dh
parameters = dh.generate_parameters(generator=2, key_size=2048)
```

In fact this produces a **generator g** , and a **very large prime number n**

Diffie-Hellman: Practice

2. Alice and Bob select random numbers as their private keys

Alice

```
alice_private = parameters.generate_private_key()
```

Bob

```
bob_private = parameters.generate_private_key()
```

The private numbers are just values between 1 and n

Diffie-Hellman: Practice

3. Alice and Bob each calculate a public key

Alice

```
alice_private = parameters.generate_private_key()  
alice_public = alice_private.public_key()
```

Bob

```
bob_private = parameters.generate_private_key()  
bob_public = bob_private.public_key()
```

The public key combines the generator and the private key using n
It's **extremely difficult** to extract the private key back out

Diffie-Hellman: Practice

4. They swap these public keys over the wire

Alice

```
alice_private = parameters.generate_private_key()  
alice_public = alice_private.public_key()
```

Bob

```
bob_private = parameters.generate_private_key()  
bob_public = bob_private.public_key()
```



Diffie-Hellman: Practice

5. They combine their private key with the other's public key, creating the shared secret key

Alice

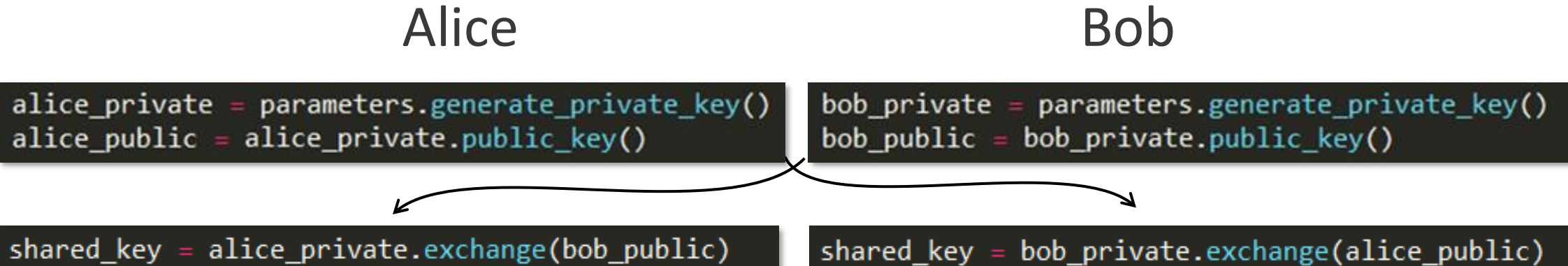
```
alice_private = parameters.generate_private_key()  
alice_public = alice_private.public_key()
```

Bob

```
bob_private = parameters.generate_private_key()  
bob_public = bob_private.public_key()
```

```
shared_key = alice_private.exchange(bob_public)
```

```
shared_key = bob_private.exchange(alice_public)
```



```
graph TD; Alice[Alice] --> Bob[Bob]; Bob --> Alice;
```

Diffie-Hellman: Practice

6. The shared secret is usually called the **pre-master secret**. It's used to derive session keys

Alice

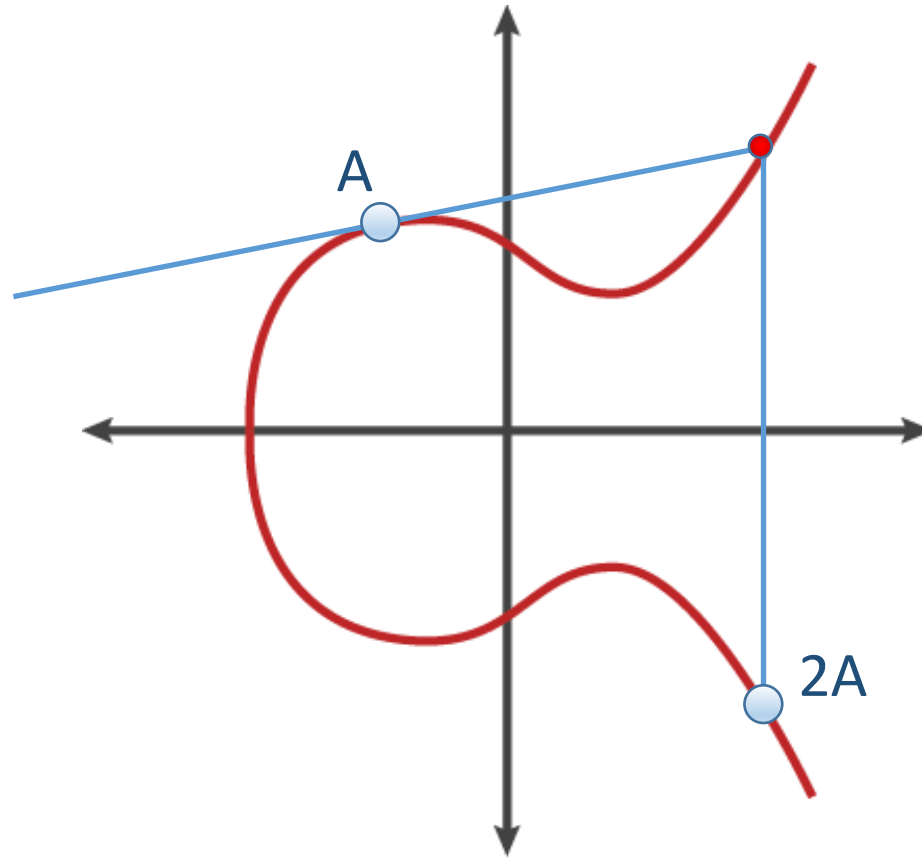
Bob

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
hkdf = HKDF(algorithm=hashes.SHA256(), length=32, salt=None, info=None)
aes_key = hkdf.derive(shared_key)
```

This hashed-key derivation function (HKDF) uses the SHA-256 hash function, which we'll cover later

Elliptic Curve Cryptography

- Elliptic curves are a drop-in replacement for the mathematics underpinning regular Diffie-Hellman



Benefits of Elliptic Curves

- Elliptic curves are **much stronger** than traditional public-key schemes for **the same key length**:

Symmetric	Diffie-Hellman and RSA	Elliptic Curve
56	512	112
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Ephemeral Mode

- In most protocols, running Diffie-Hellman in ephemeral mode forces a new key exchange every time
- This is perfect forward secrecy

Don't perform a handshake and use those keys for months!

Exercise – Diffie-Hellman

Perform a DH key exchange with a server

asymmetric/DHServer.py
dhkeyexchange.py

```
# Obtain parameters from the server - this would normally come over a network
parameters = server.get_parameters()

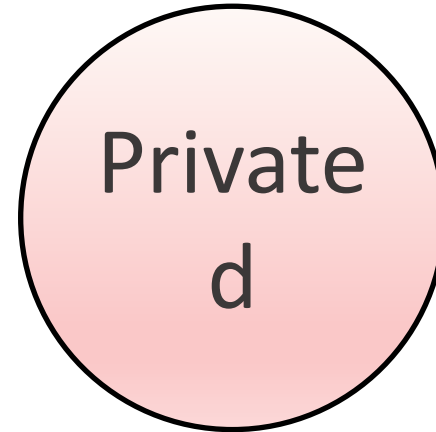
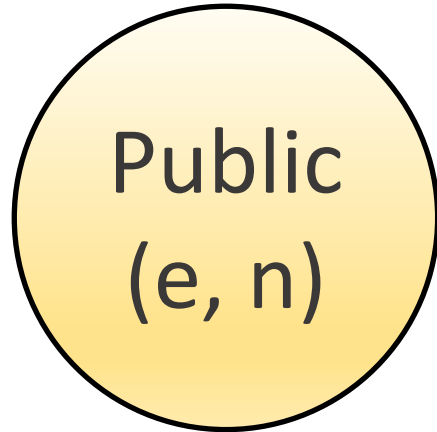
# Obtain servers public DH key - also normally over a network
server_public_key = server.get_public_key()
print ("The server's public key is: ", server_public_key.public_numbers().y)

### Task 1 ###
# Generate your own private key (look at DHServer get_public_key for hints)
private_key = None
if private_key != None:
    print ("Our private key is: ", private_key.private_numbers().x)
```

Public-Key Cryptography

RSA

- RSA is the most common method for general public key cryptography
- It provides both **encryption** and/or **authentication**
- RSA provides us with two keys:



e is usually a small number, d is a much larger number
n is a very large semi-prime number $n=p \cdot q$

Generating Keys

- Generating RSA key pairs is quite time-consuming, and should be done rarely
- The general process is to choose e , find an n at random and then calculate the secret d

```
from cryptography.hazmat.primitives.asymmetric import rsa

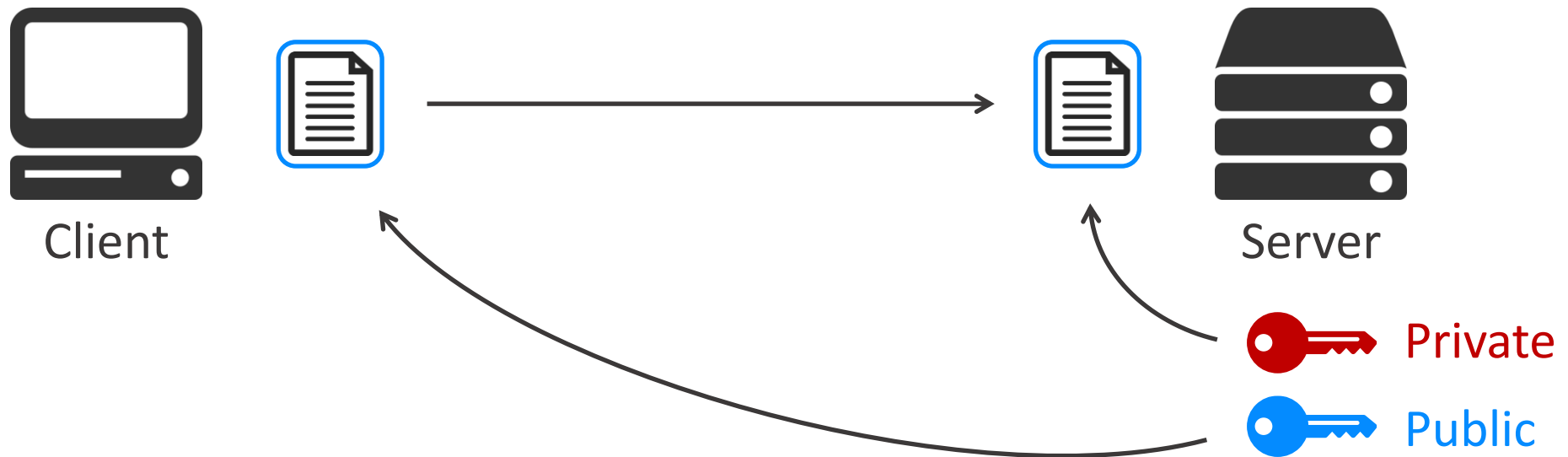
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)

public_key = private_key.public_key()
```

e is almost always 3 or 65537, n is 2048 bits and generated at random. d will be secretly computed during this process.

RSA Encryption

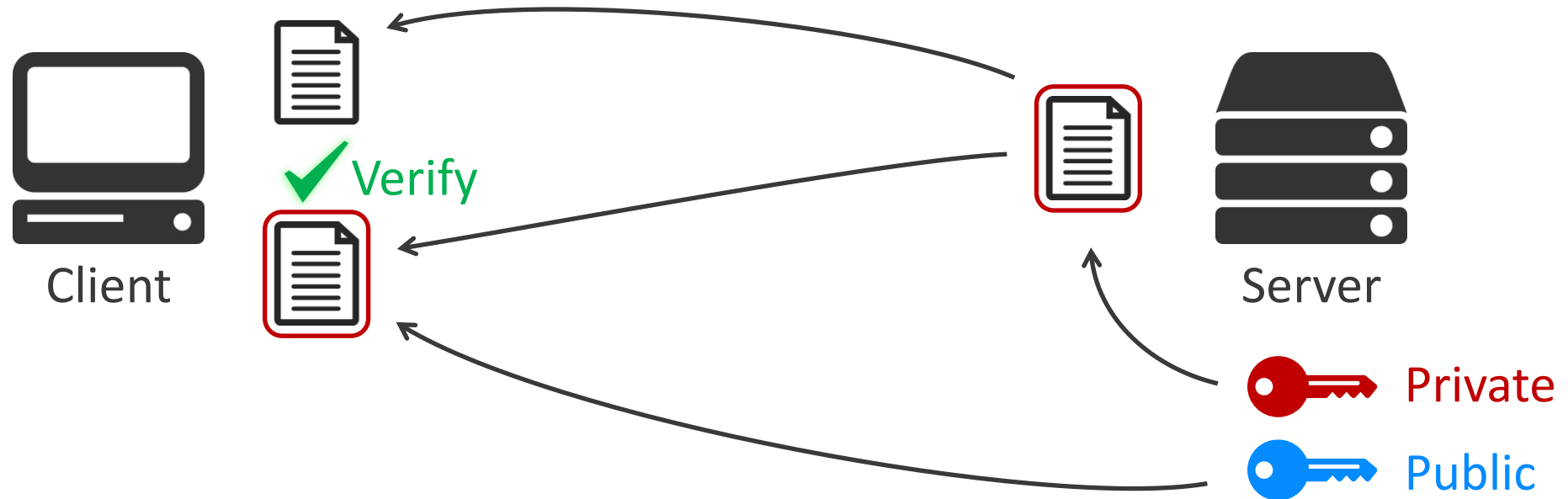
- Encryption performed by the **public key** can only be reversed using the **private key**



Using RSA for encryption is not really encouraged

RSA Signing

- The authenticity of signatures generated by the **private key** can be verified by the **public key**



Signing

① Server

```
signature = private_key.sign(  
    b"This is the message we wanted the server to sign",  
    padding.PSS(  
        mgf=padding.MGF1(hashes.SHA256()),  
        salt_length=padding.PSS.MAX_LENGTH  
    ),  
    hashes.SHA256()  
)
```

Send signature to us

② Us

```
signed_message = # Receive from server  
  
try:  
    public_key.verify(  
        signed_message,  
        b"This is the message we wanted the server to sign",  
        padding.PSS(  
            mgf=padding.MGF1(hashes.SHA256()),  
            salt_length=padding.PSS.MAX_LENGTH  
        ),  
        hashes.SHA256()  
    )  
    print ("The server successfully signed the message.")  
except InvalidSignature:  
    print("The server failed our signature verification!")
```

Probabilistic Signature Scheme (PSS) is the recommended padding scheme for signatures

DSA

- The main alternative to RSA is The Digital Signature Algorithm (DSA)
- It acts a lot like RSA, but uses mathematics similar to Diffie-Hellman
- It doesn't encrypt, but can be used for signing messages
- It can also make use of Elliptic Curves

Exercise – RSA

Use a server's public key to verify its identity

asymmetric/RSAServer.py
rsa.py

```
### Task 1 ###
# If a server signs a message with its private key, we can verify with its public key.
# First choose a message and have the server sign it:
message = b"Insert your verification message here"
signed_message = server.sign_document(message)

# To sign the message, the server will encrypt a hash of it using its private key, we can repeat this
# and verify it with the public key

try:
    # Verify the signed message using public_key.verify()
    # Documentation here: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#verify
    # For padding, use PSS with MGF1-SHA256 as per the documentation

    # Verify will raise an exception if the signature fails. Replace this line with a verification:
    raise InvalidSignature("We've not implemented verification yet!")

    print("The server successfully signed the message.")
except InvalidSignature:
    print("The server failed our signature verification!")
```

Hash Functions

Hash Functions

- Another cryptographic primitive in our toolbox
- Takes a message of any length, and returns a **pseudorandom hash** of fixed length

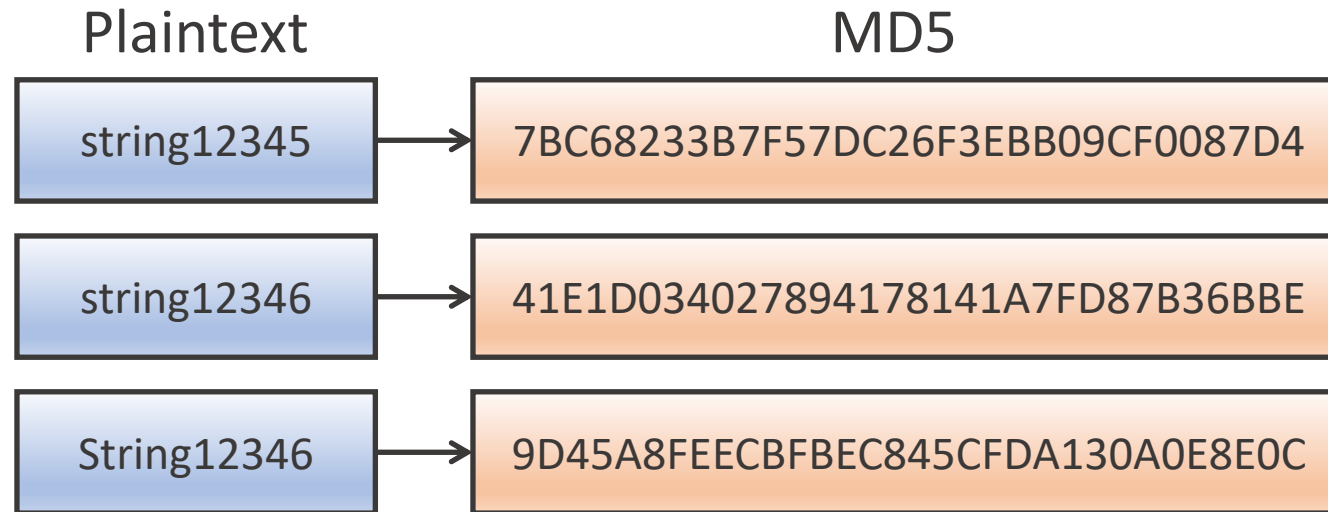
```
from cryptography.hazmat.primitives import hashes
digest = hashes.Hash(hashes.SHA256())
digest.update(long_message)
digest.update(more_long_message)
h = digest.finalize()
print (len(h))
```

32

- Hash functions are used everywhere. Message authentication, integrity, passwords etc.

Strong Hash Functions

- The output must be indistinguishable from **random noise**
- Bit changes must diffused through the entire output

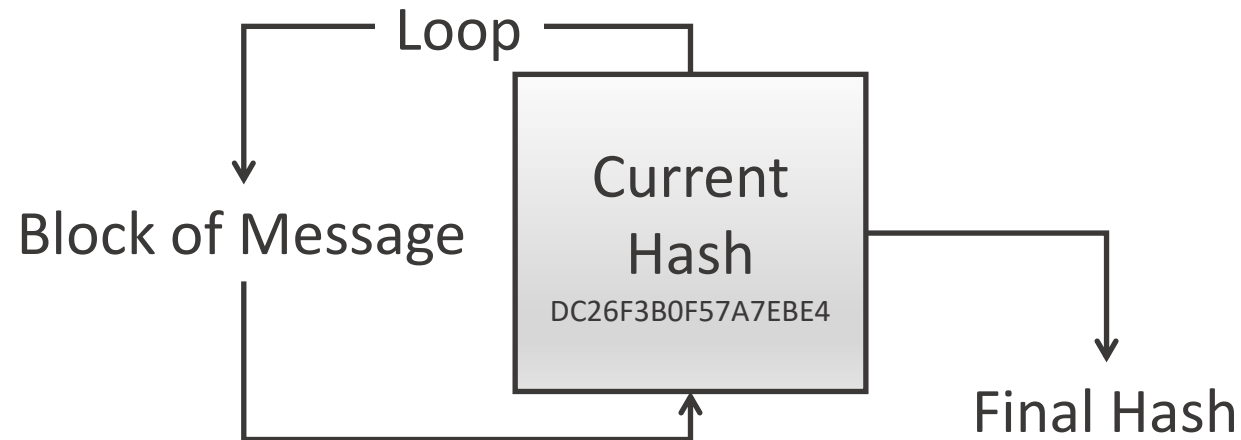


Strong Hash Functions

- For a hash function to be useful, we need it to have some important properties:
 1. Given a hash, we can't reverse it
 2. Given **a message** and its hash, we can't find another message that hashes to the same thing
 3. We can't find **any two** messages that have the same hash

Hash Functions

- Usually hash functions iteratively jumble blocks of a message after another
- Once all the message is gone, we read the current hash



- The initial hash is usually defined in the spec

Notable Examples

Name	Output Length	Rounds	Security
MD5	128-bit	4	Broken
SHA-1	160	80	Recent collision found, not trusted
SHA-2	224, 256, 384, 512	64, 80	Some theories, currently considered safe
SHA-3 (Keccak)	224, 256, 384, 512 SHAKE128, 256	24	Secure, but relatively untested, strength variable
PBKDF2	Varies		Iterates another hash function
bcrypt	184-bit		GPUs struggle to crack it

Cryptography

Password
Storage

Exercise – Hash Functions

Hash various messages including the entire works of Shakespeare!

hashing.py

```
### Task 1 ###
# Hash the following message using SHA-1, SHA-256 and SHA-512.
# To make things a bit easier, SHA-1 has been done for you:
message = b'A message to be hashed'
sha1.update(message)
sha1hash = sha1.finalize()
if sha1hash != None:
    print ("SHA-1:", b16encode(sha1hash))

# Repeat this process for SHA-256
sha256hash = None
if sha256hash != None:
    print ("SHA-256:", b16encode(sha256hash))
```

Where are hashes used?

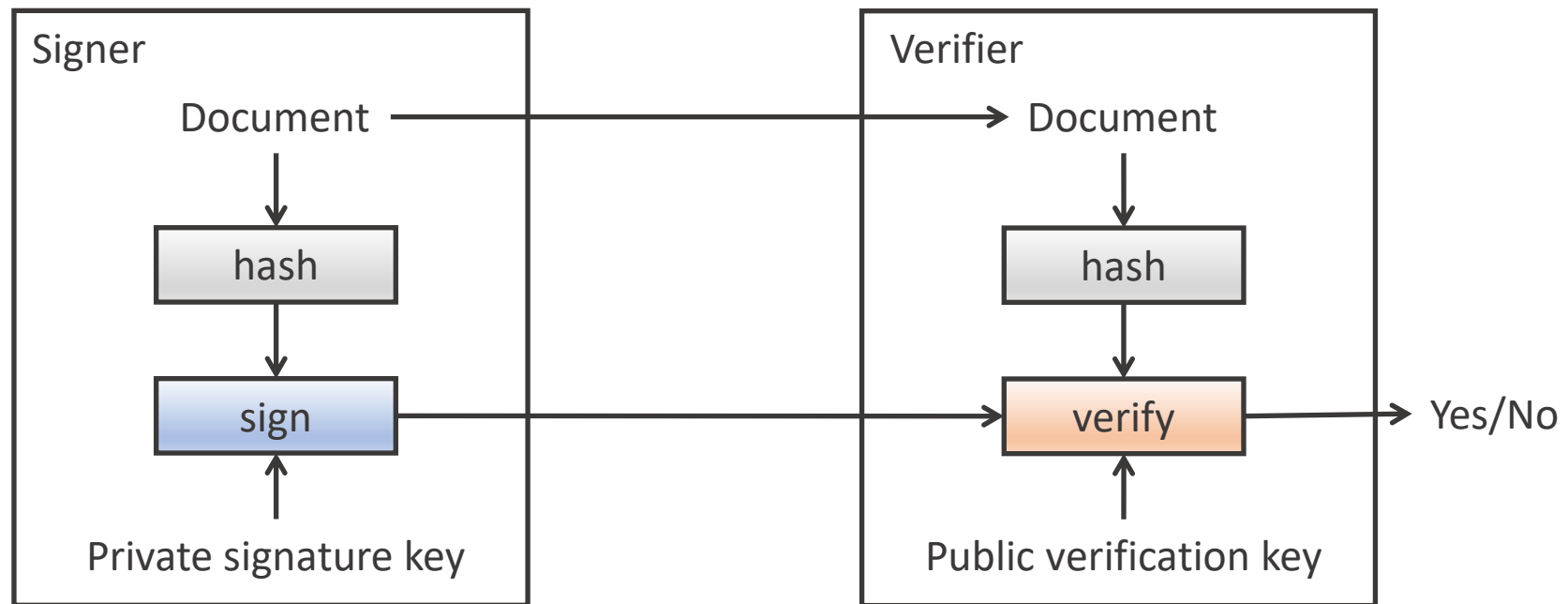
- Recall that symmetric cryptography is often vulnerable to message tampering – **Integrity**
- Hashing lets us ensure that a message hasn't been altered:



This is a **Message Authentication Code**

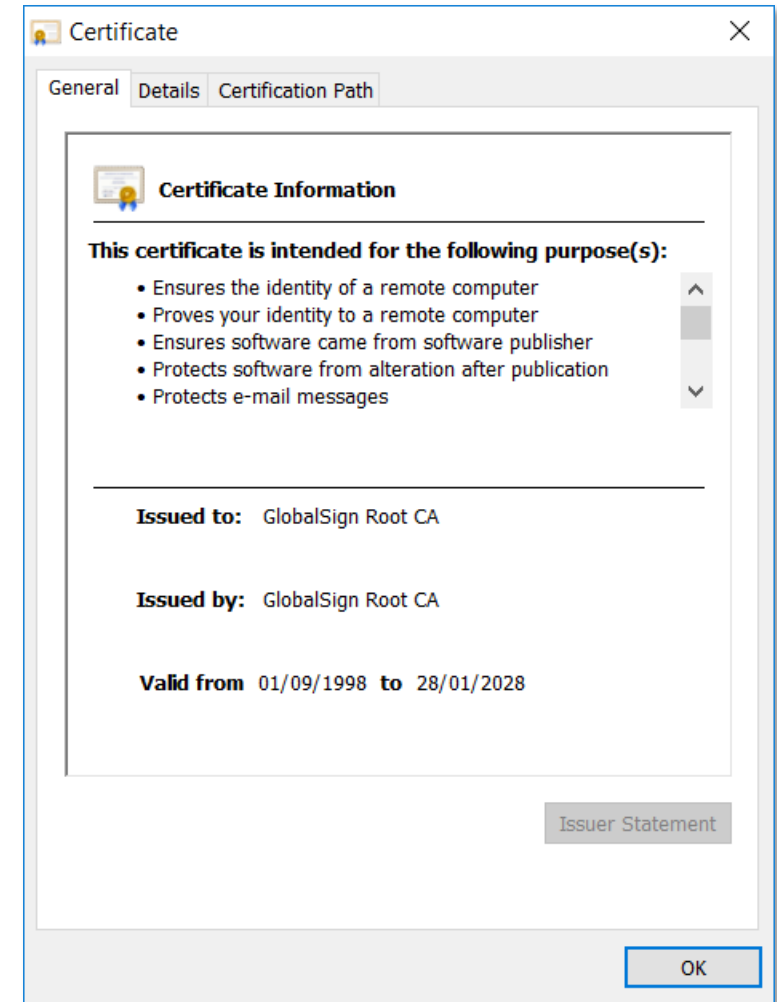
Digital Signatures – Hashing + PK

- Hashes are used with RSA and DSA to create digital signatures
- They prove the authenticity of the sender



Digital Certificates

- We can use a trusted third party in order to **verify the ownership of a public key**
- Bob then knows he has Alice's genuine key, not an imposter
- Can also be 'self signed'
- An important part of Transport Layer security (TLS)



Certificate Issuance

- Server (server.com) has a public key that they want people to trust
- They create a **Certificate Signing Request**

Subject: Server.com

Version: 1

Signature Algorithm: sha256RSA

Public Key: 30 82 01 0a ... 01 0F

Exponent: 65537

Unsigned Signature

Certificate Issuance

- They go to a certificate authority (CA), who after doing ID checks, create and sign the certificate with their private key

Subject: Server.com

Serial: 44c11bf942ad8329

Signature Algorithm: sha256RSA

Issuer: GlobalTrust SSL CA - G3

Valid From: 18 Nov 2018

Valid To: 18 Nov 2020

Subject: Server Inc

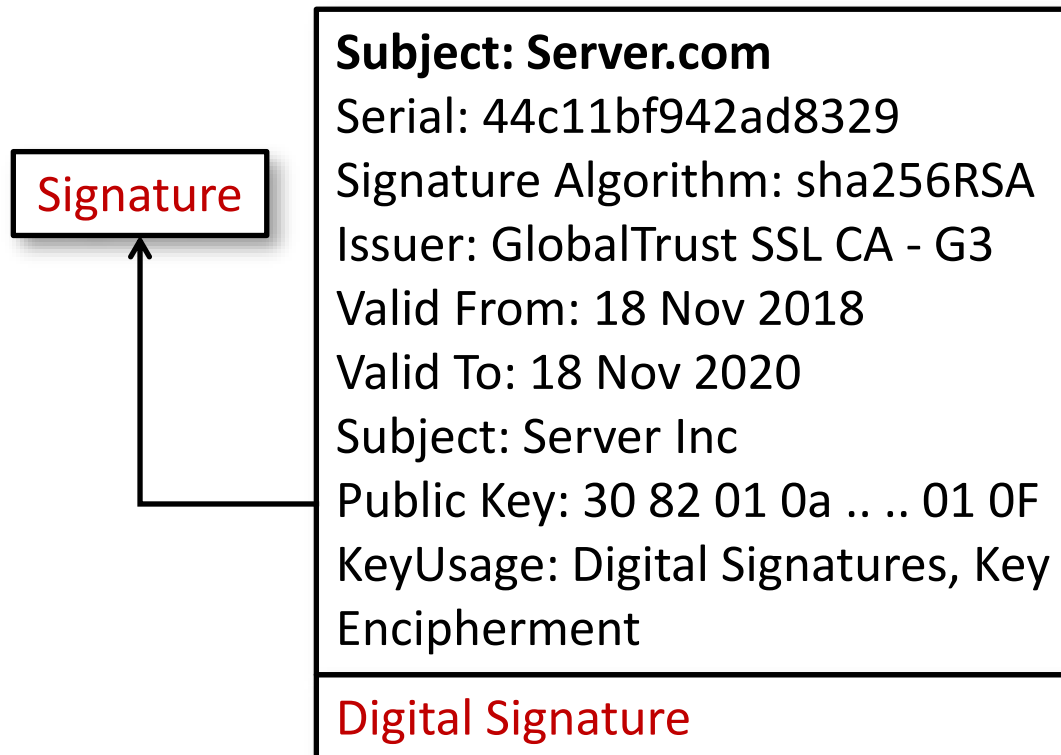
Public Key: 30 82 01 0a 01 0F

KeyUsage: Digital Signatures, Key
Encipherment

Digital Signature

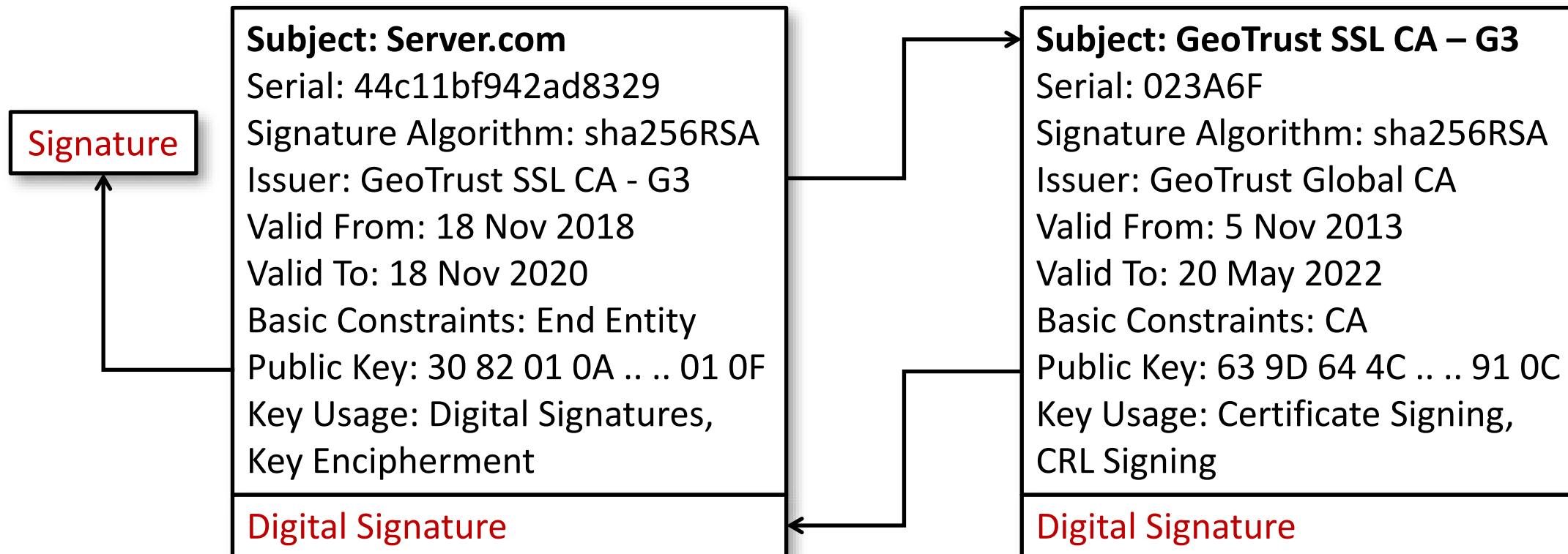
Certificate Use

- The server can supply digital signatures using their key pair, backed by the certificate when requested, e.g. during a TLS handshake



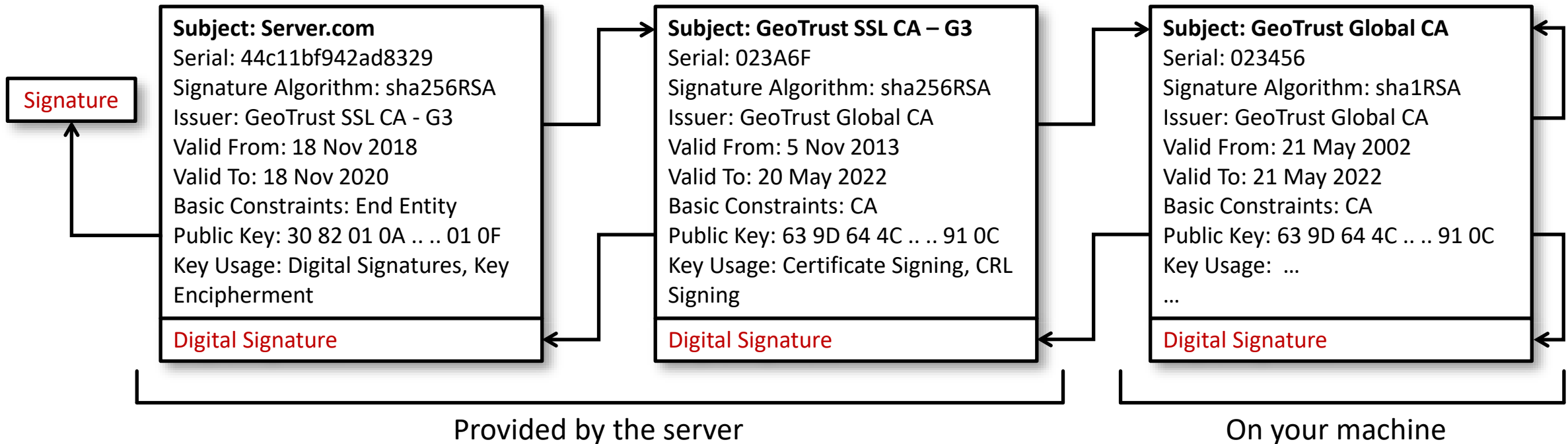
Chains of Trust

- To verify the trust in the Server.com certificate, we need to examine the signing certificate



Chains of Trust

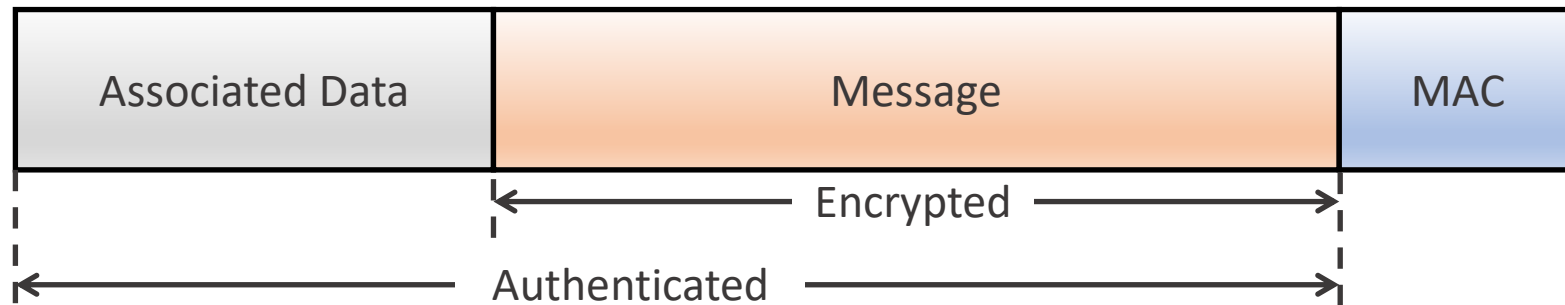
- In many cases, the chain involves multiple certificates
- Chains always end in a root certificate, located on **your machine**



Complete Systems

Authenticated Encryption - AEAD

- It's so common to attach MACs to the end of ciphertext, that this is now usually built into ciphers as part of AEAD mode
- You're often able to authenticate non-encrypted data too



ChaCha20_Poly1305

- A standard AEAD mode of encryption using the ChaCha20 stream cipher, and the Poly1305 MAC

```
import os
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305

data = b"a secret message"
aad = b"some public session data"
key = os.urandom(32)
chacha = ChaCha20Poly1305(key)
nonce = os.urandom(12)

ct = chacha.encrypt(nonce, data, aad)
chacha.decrypt(nonce, ct, aad)

'a secret message'
```

AES Galois Counter Mode (GCM)

- Similar to AES-CTR mode, but computes an **authentication tag** (a GMAC) over the ciphertext and the additional data

```
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

data = b"a secret message"
aad = b"some public session data"
key = os.urandom(16)
aesgcm = AESGCM(key)
nonce = os.urandom(12)

ct = aesgcm.encrypt(nonce, data, aad)
aesgcm.decrypt(nonce, ct, aad)
```

```
'a secret message'
```

Exercise – AEAD

Encrypt and decrypt a database record using AEAD

aead.py

```
record = {
```

```
"ID": "0054",
```

```
"Surname": "Smith",
```

```
"FirstName": "John",
```

```
"JoinDate": "2016-03-12",
```

```
"LastLogin": "2017-05-19",
```

```
"Address": "5 Mornington Crescent, London, WN1 1DA",
```

```
"Nationality": "UK",
```

```
"DOB": "1963-09-14",
```

```
"SSN": "00123456C",
```

```
"Phone": "01224103232",
```

```
"Data": None,
```

```
"Nonce": None,
```

}

Task 1

```
def encrypt_record(record, key, nonce):
```

```
plaintext = None
```

Authenticated

Encrypted

Protocols

Protocols are used to safely combine the various cryptographic primitives

1. Handshake Phase

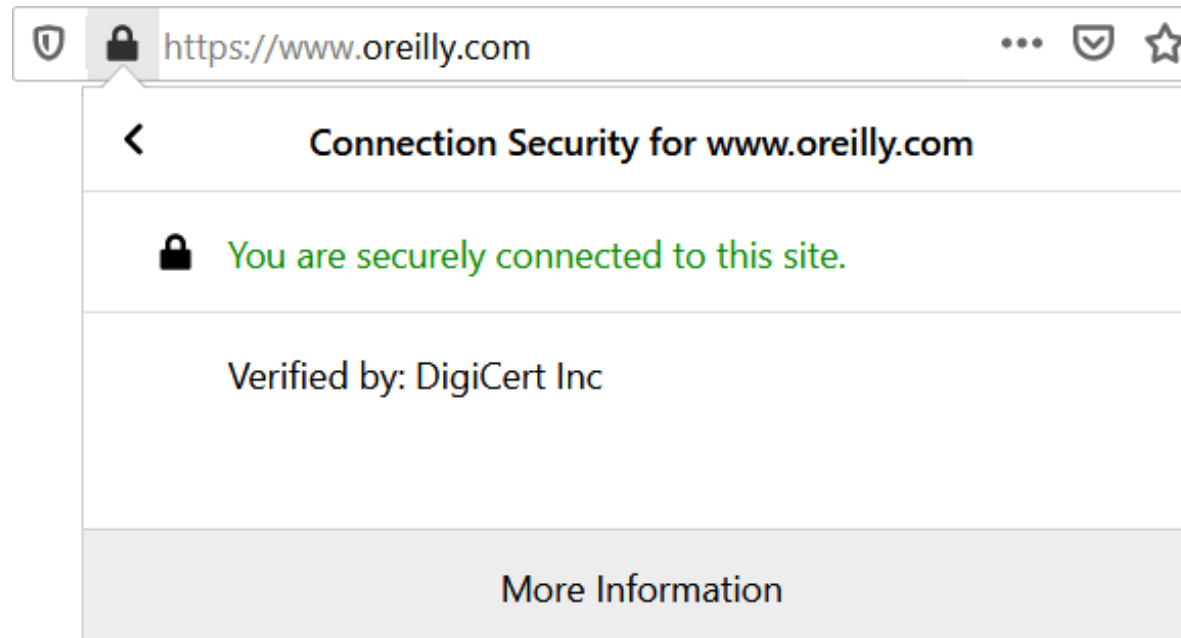
- Agree on a set of **cryptographic protocols**, e.g. AES 128, ECDH, RSA etc.
- Perform a **key exchange** to obtain session keys and other values such as IVs
- Verify any **authenticity using PK**

2. Transport / Record Phase

- Packets encrypted using the agreed cipher
- Includes a MAC or similar to **verify integrity**

Design your own protocol?

- Protocol design is **much harder than you think** to get right
- Avoid it! There are numerous **well-established protocols** out there
- We'll look at TLS (the protocol behind HTTPS) as an example of what's involved



SSL/TLS

- SSL and TLS are protocols for **internet handshakes and encrypted transmission**
- Secure Socket Layer (SSL) came first, then after v3.0 it became Transport Layer Security (TLS), currently v1.3
- People still use SSL as a term, even though technically it's now TLS

We will treat SSL and TLS here interchangeably

TLS Handshake



Historic TLS issues

- Historically TLS has had plenty of problems, **but**, a lot have been cleared up
- Weak ciphers are now phased out with new TLS versions – but protocol downgrade attacks are dangerous
- Incorrect use of IVs lead to breaks
- Various implementation problems like padding issues, use of weak DH parameters etc.

Practical Tips and Tricks

General Tips

- Relax. To an extent it's a cryptographer's job to worry about strange related-key attacks! **Most modern ciphers are unbelievably strong.**
- Careful use of libraries will leave you very secure, just apply some common sense and follow the documentation

Dos

- Use cryptographically strong random numbers. In python this is `os.random` and `os.urandom`, not `math.random`
- Use “recipe” layers where possible. They take the algorithm and protocol decisions out of your hands
- Use ephemeral session keys for communication, and long-term public-keys for authentication
- Be careful to use IVs that are appropriate to the algorithm. If in doubt:
 - Nonces are used `once`
 - IVs must be `unpredictable`

Don'ts!

- Implementing your own algorithms is a bad idea!
- Never use hard-coded keys, convenience comes at a cost
- Never reuse IVs or Nonces
- Don't use ECB mode. Prefer CBC, but better yet CTR or GCM
- Don't use small public key sizes. At least 2048 bits! Elliptic curve is preferable: P-256 or X25519

Network Cryptography Tips

- For general end-to-end communication, consider using TLS
- TLS supports client and server authentication using two certificates – useful for many setups
- Restrict what cipher suites are allowed to avoid protocol downgrades
- If you don't use TLS, always use AEAD or another authenticated encryption mechanism

Storage and DB Tips

- Use password derivation functions not hashing functions for password storage. E.g. PBKDF2
- Encrypt data where possible using keys that you change fairly often
- Encrypt the keys, adding a layer of indirection for an attacker.
- This is a great resource:
 - NIST SP 800-57 Part 1 Rev. 5



Where Now?

- For development, check out the documentation on Cryptography, or the very popular NaCL and libsodium libraries
 - <https://cryptography.io/>
 - <https://nacl.cr.yp.to/>
 - <https://download.libsodium.org/doc/>
- Further reading:
 - Security Engineering: Ross Anderson
<https://www.oreilly.com/library/view/security-engineering-3rd/9781119642787/>
 - Cryptography Engineering: Ferguson, Schneier & Kohno.
 - Understanding Cryptography: Paar & Pelzl

