

Introduction to Encryption

introduction to encryption

Welcome

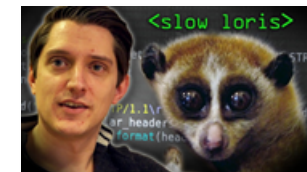
Michael Pound

- Assistant Professor at the University of Nottingham, UK
- Teach the final year Security module 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

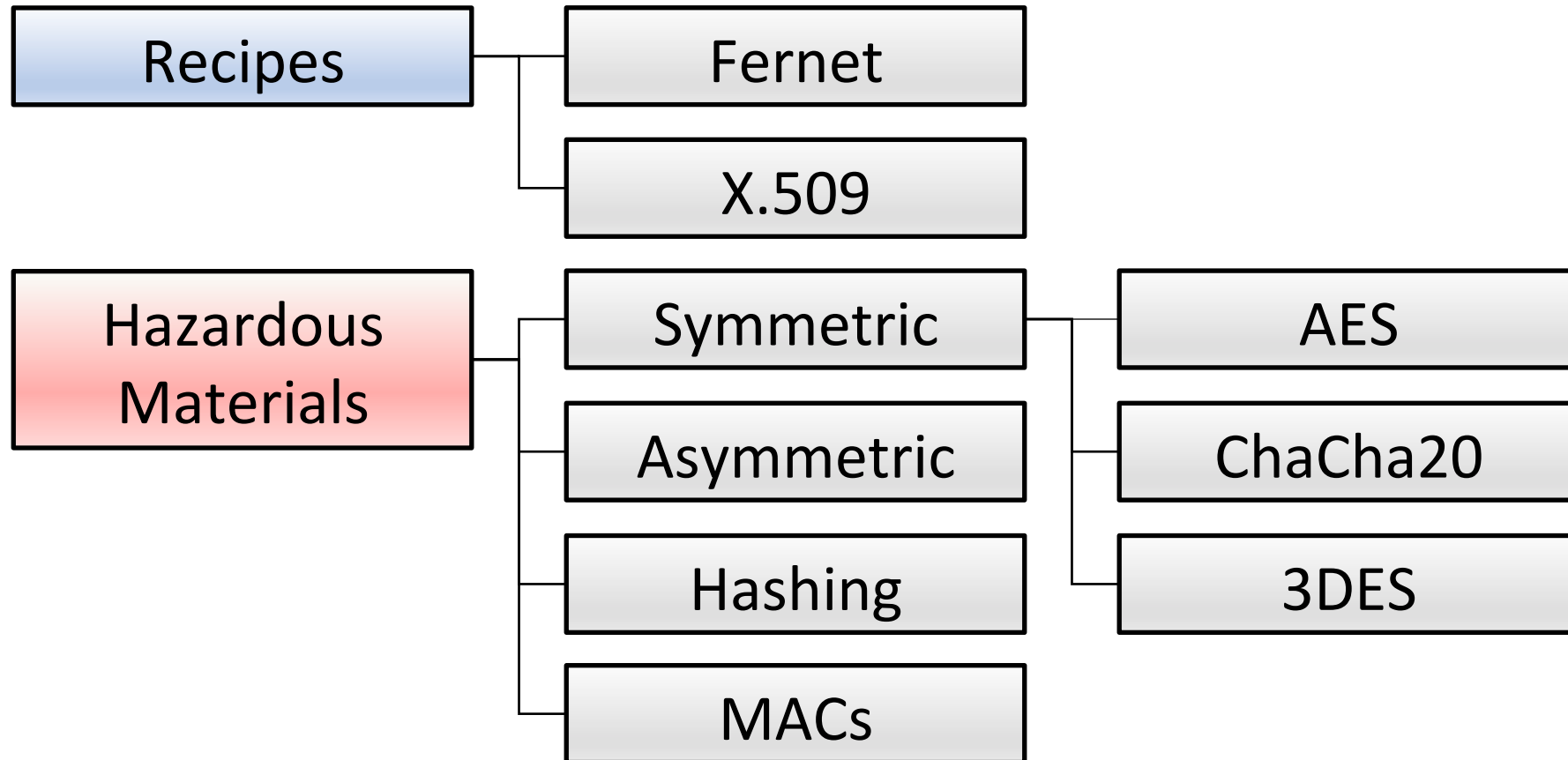
- This course will give you an understanding of the **foundations of cryptography**
- We'll cover **the most common primitives**, and what they're for
- Experience **using these tools** within a programming language
- Insight into development of **secure protocols** and systems
- Knowledge of **common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

The Exercises

- The course materials come with exercises in Python
 - You only need a general grasp of python to have a go
 - Slightly altered exercise are available for python 2 or 3
 - 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>



Python Cryptography

- Most recent versions of cryptography will work during these exercises, but the final exercises on AEAD require 2.0+

```
import cryptography
print (cryptography.__version__)
```

- If you're using pip, upgrade with:
 `pip install cryptography --upgrade`

Python and Unicode

- Python strings have changed a lot between v2 and v3
- Python 2:
 - A string is a sequence of bytes
 - Strings can be passed directly into cryptography
- Python 3:
 - A string is a sequence of Unicode code points
 - bytes / bytearray are sequences of bytes
 - Unicode strings must be converted into bytes for encryption, and back out again

```
>>> s = 'Café'
'Café'
>>> b = s.encode("UTF-8")
b'caf\xc3\xa9'
>>> b.decode("UTF-8")
'Café'
```

Printing strings

- In python when bytes are printed, it assumes we plan to read them,, this doesn't always work!

- Python 2:

```
>>> os.urandom(16)
'ao{\x07o\xb2k/\xf6\xba\n\xae\xef \xd6'
```

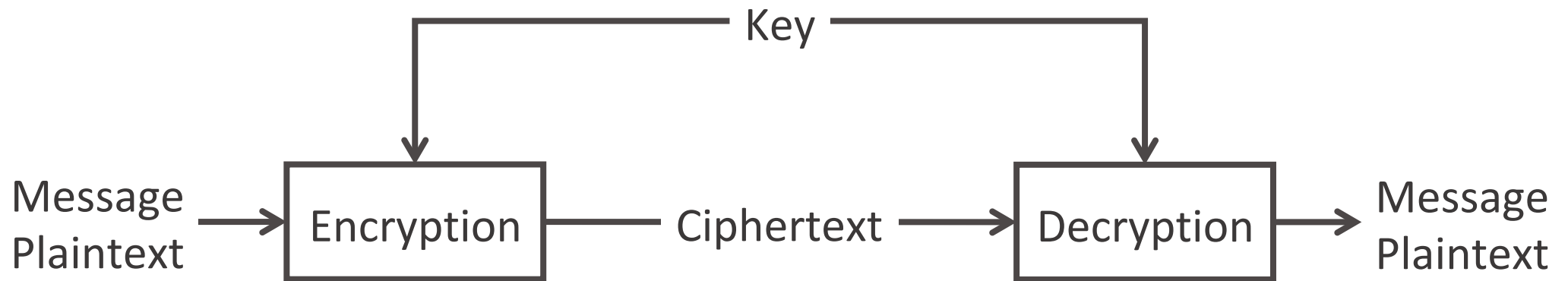
- Python 3:

```
>>> os.urandom(16)
b'!\xcf\xd6.m?\xfd\xde\xd8\xc6\xc4t*\xac\xab\x1a'
```


When Use Encryption?

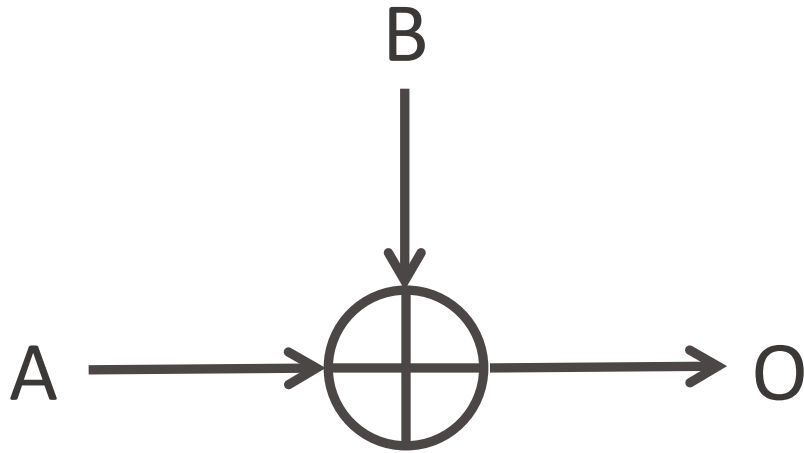
- User / Company Database Records
- Cloud Storage
- Password storage
- Compliance
- In transit:
 - Network communication (e.g. inter-application)
 - Authentication data
 - Payment and money transfers

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:

$$\begin{aligned} A \oplus B \oplus A &= (A \oplus A) \oplus B \\ &= 0 \oplus B \\ &= B \end{aligned}$$

- 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

$$\begin{array}{rcl} \text{M} & 01011010 & 00110101 \\ & & \oplus \\ \text{K} & 01001011 & 10111001 \\ & & = \\ \text{C} & 00010001 & 10001100 \end{array}$$

Perfect Secrecy

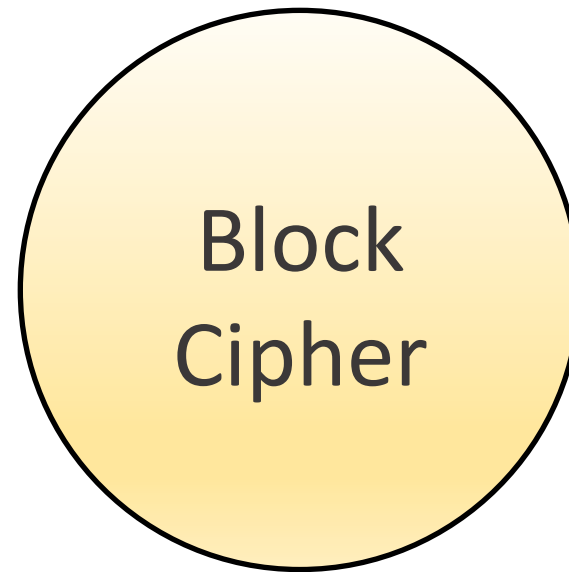
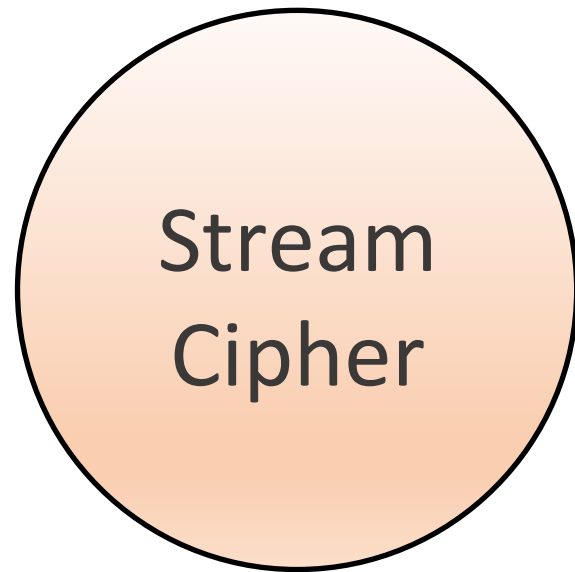
But...

- 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

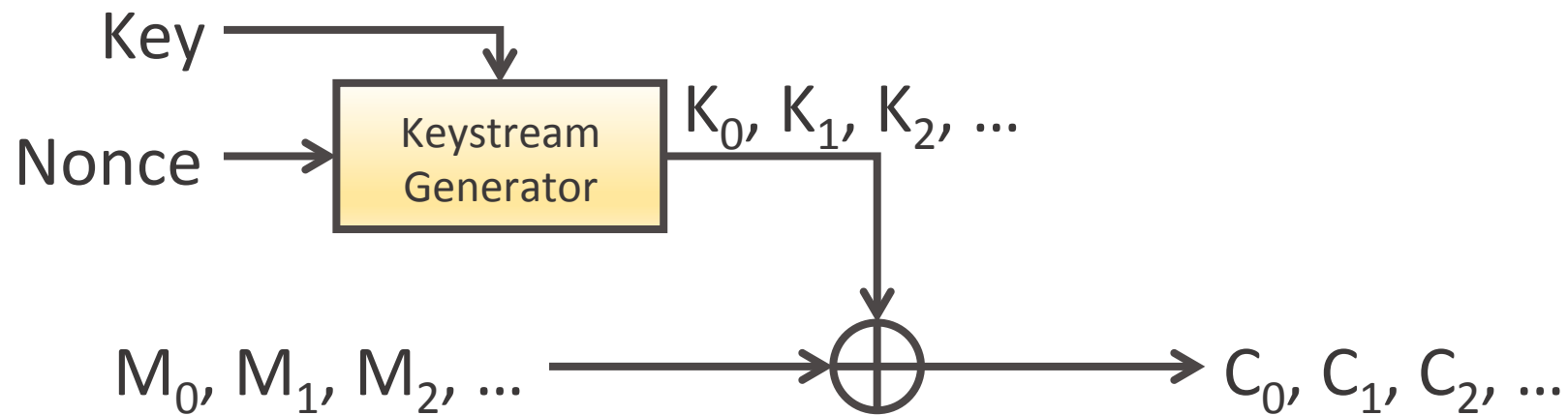
Symmetric Primitives

- 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

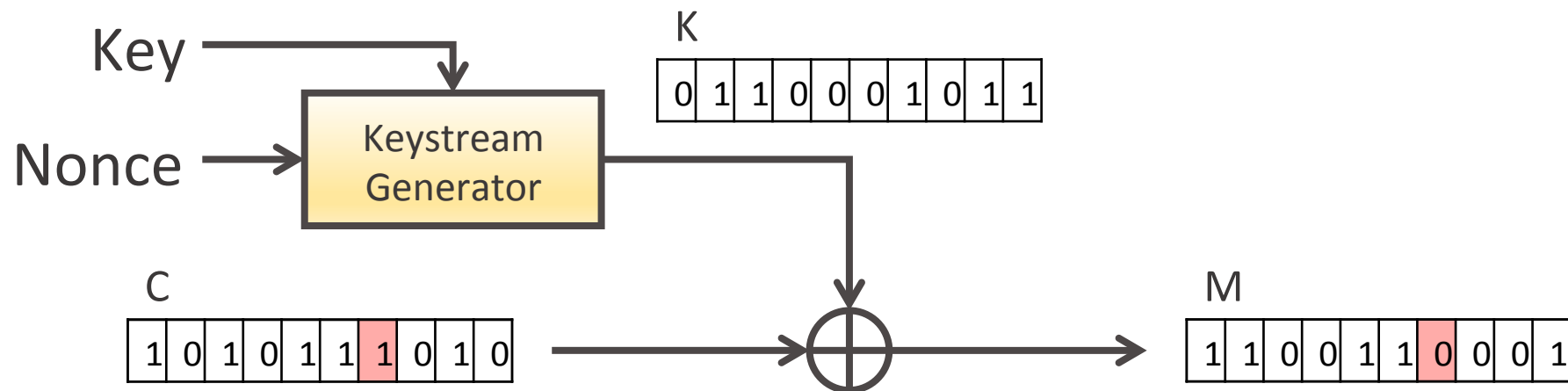


Advantages of Stream Ciphers

- 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

Weaknesses of Stream Ciphers

- Stream ciphers simulate a one-time pad, so the keystream has very strict requirements:
 - The keystream must appear statistically random
 - You must *never* use a key + nonce pair for a second time
- Stream ciphers are vulnerable to attacks on the ciphertext:



Confidentiality vs Integrity

- Preventing unauthorised users from reading our messages is **Confidentiality**
 - This is almost never satisfactory on its own!
- We need to be sure our message hasn't been altered. This is message **Integrity**
 - We generally need a second primitive to add integrity

Block Ciphers

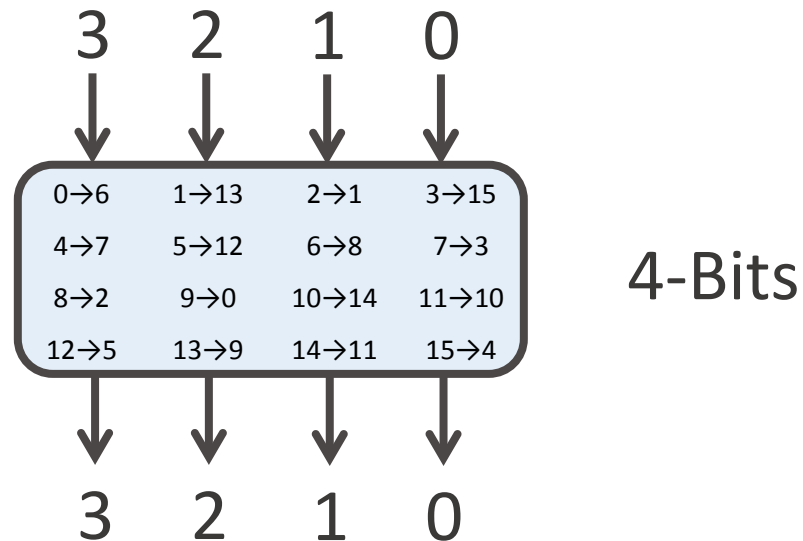
- Block ciphers take an input of a **fixed size**, and return an output of the same size
- Block ciphers attempt to hide the transformation from message to ciphertext through **confusion**, and **diffusion**

SP-Networks

- Logic networks that perform repeated substitution and permutation operations
 - Replacing bytes with others
 - Swapping bytes around
- Some substitution and permutation is combined into a single **round**
- Rounds are then **repeated** enough times to ensure the algorithm is secure

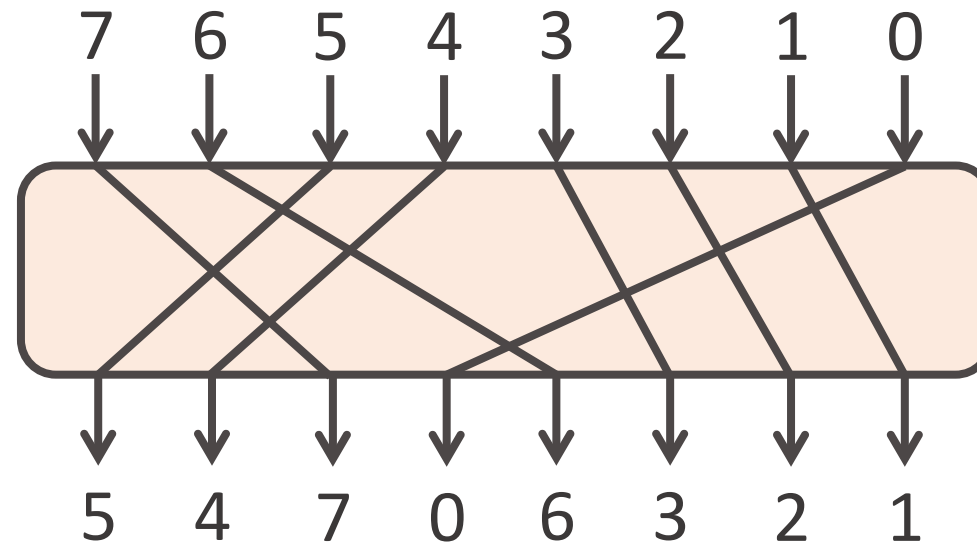
SP-Networks

- Substitution Boxes – Add **confusion** by replacing values with other values using a lookup table



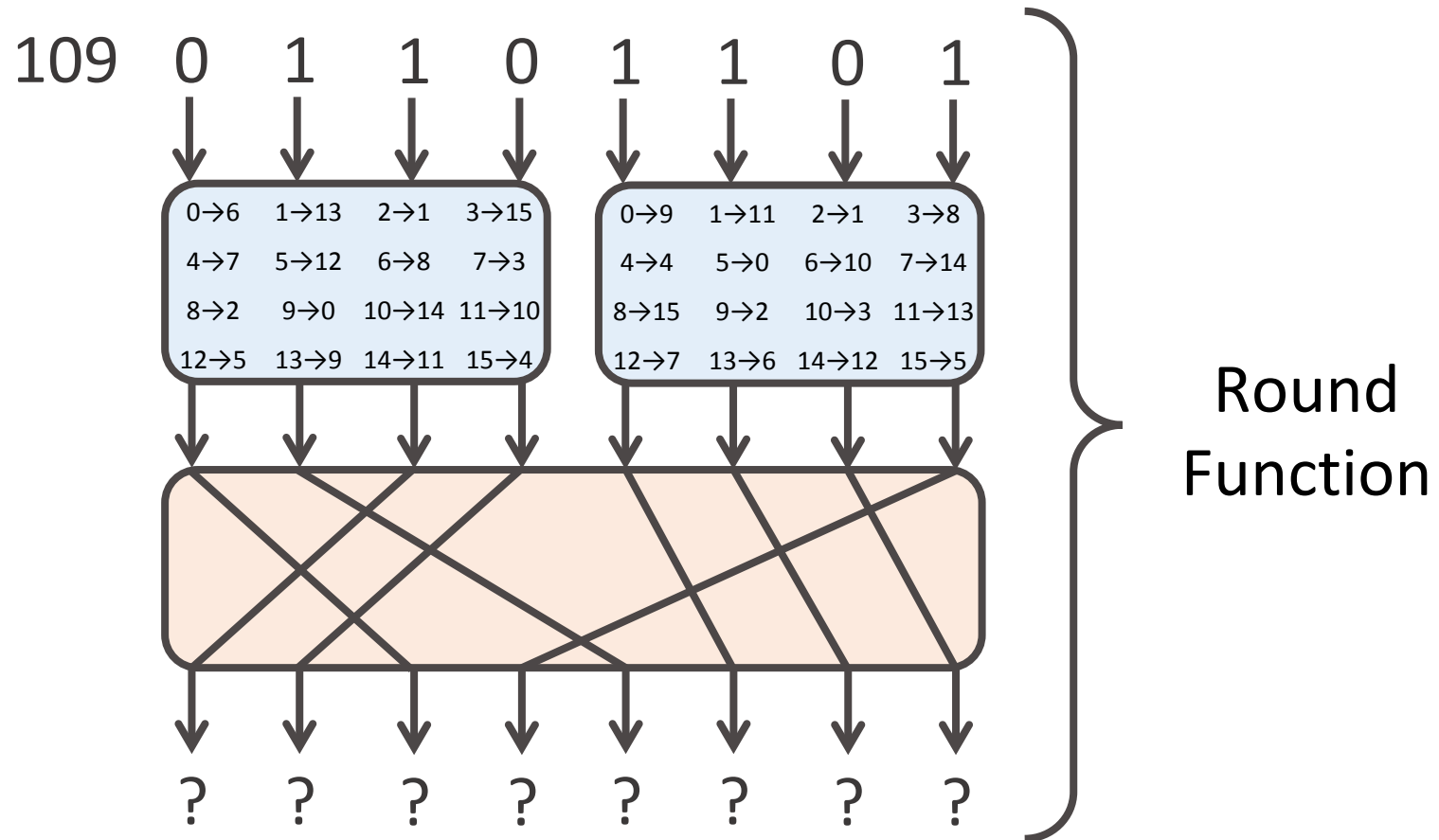
SP-Networks

- Permutation Boxes – Add **diffusion** by moving values around from input to output



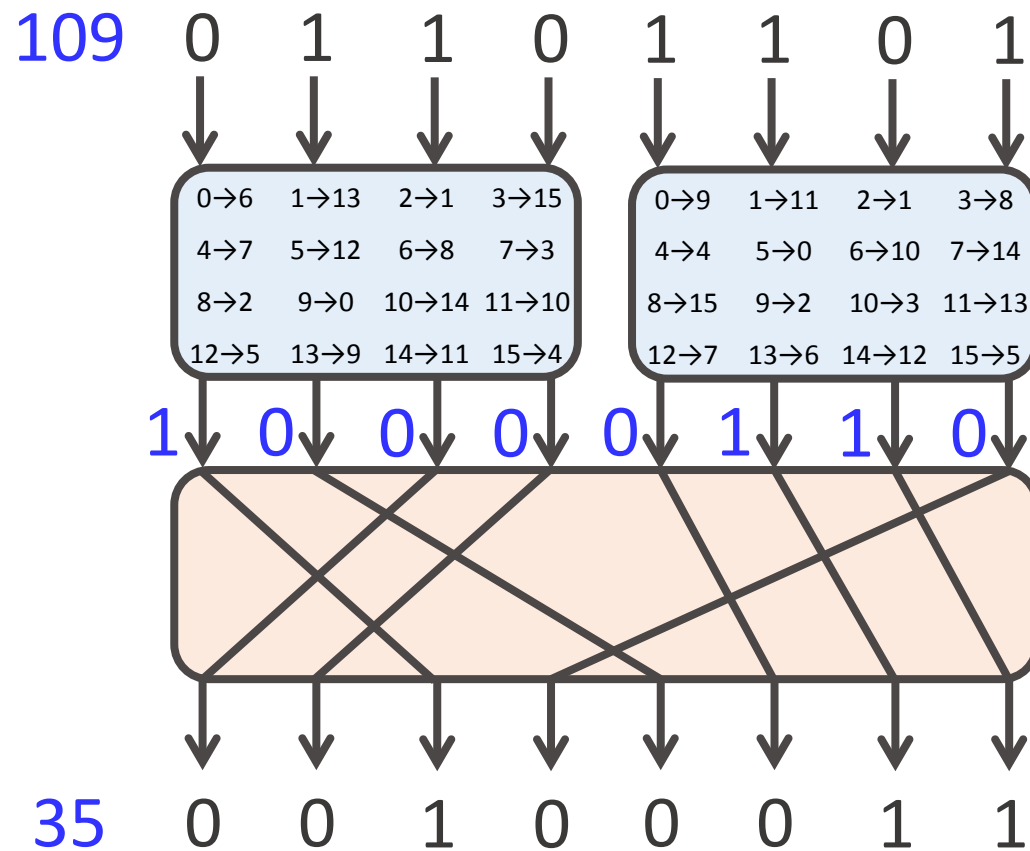
SP-Networks

- A simple 8-bit SP-Network



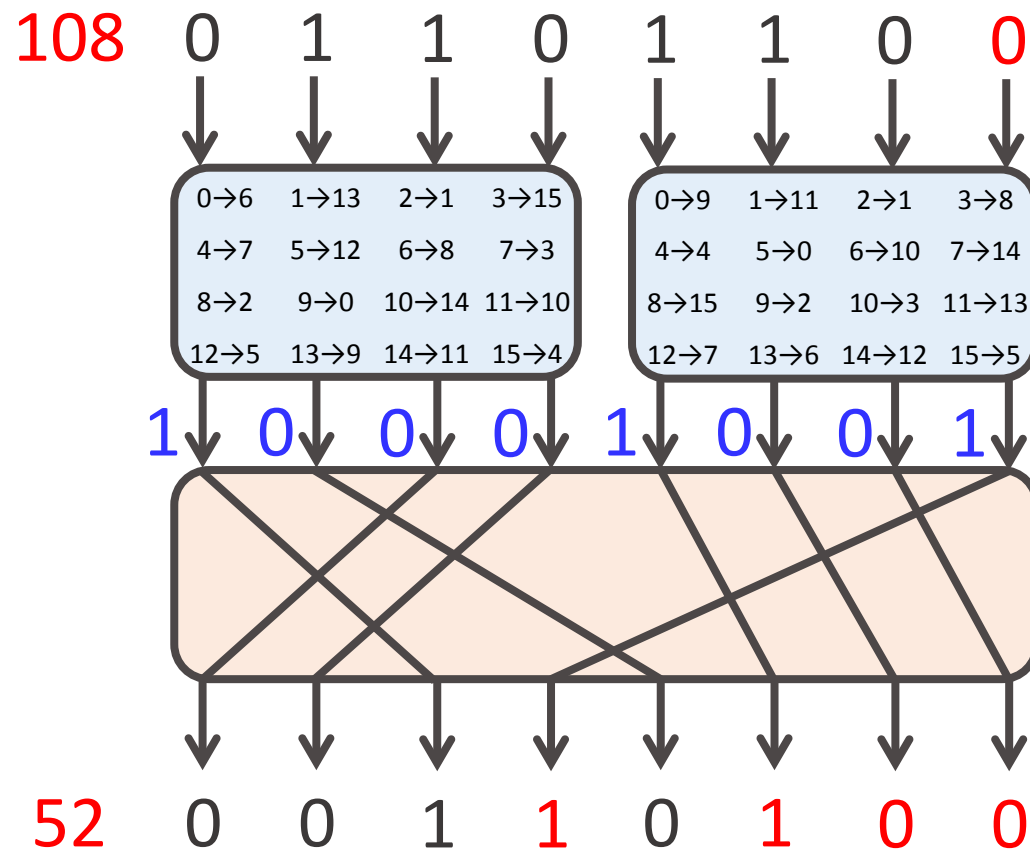
SP-Networks

- A simple 8-bit SP-Network



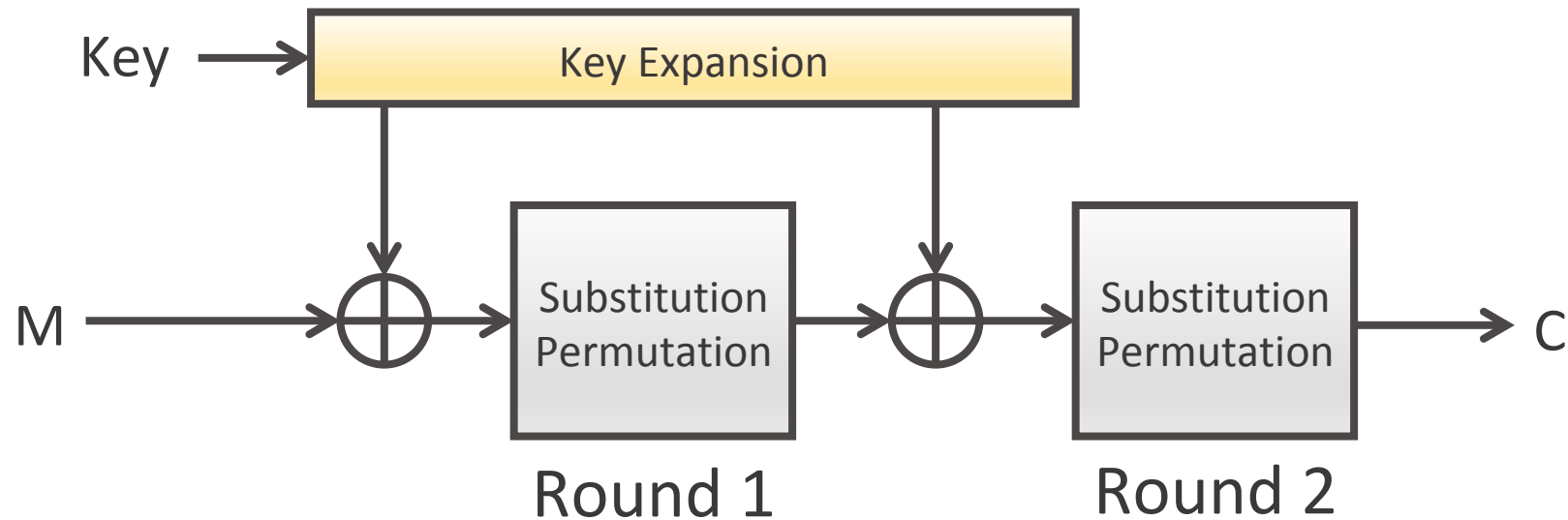
SP-Networks

- Notice a small change **diffuses** through the output



Key Mixing

- We can introduce a key using XOR:



Modern Block Ciphers

- Most block ciphers are SP-Networks
- There are others (e.g. Feistel Ciphers)
- The Advanced Encryption Standard (AES) is an SP-Network

The take home message:

Almost everything uses AES

AES

- Superseded DES as a standard in 2002
- A standard built around the [Rijndael](#) algorithm
- Rijndael is an SP-Network with a 128-bit block size, and a key length of 128, 192 or 256-bits
- Round count depends on key length
 - 10, 12 or 14 cycles
- Each Round:
 - SubBytes, ShiftRows, MixColumns

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

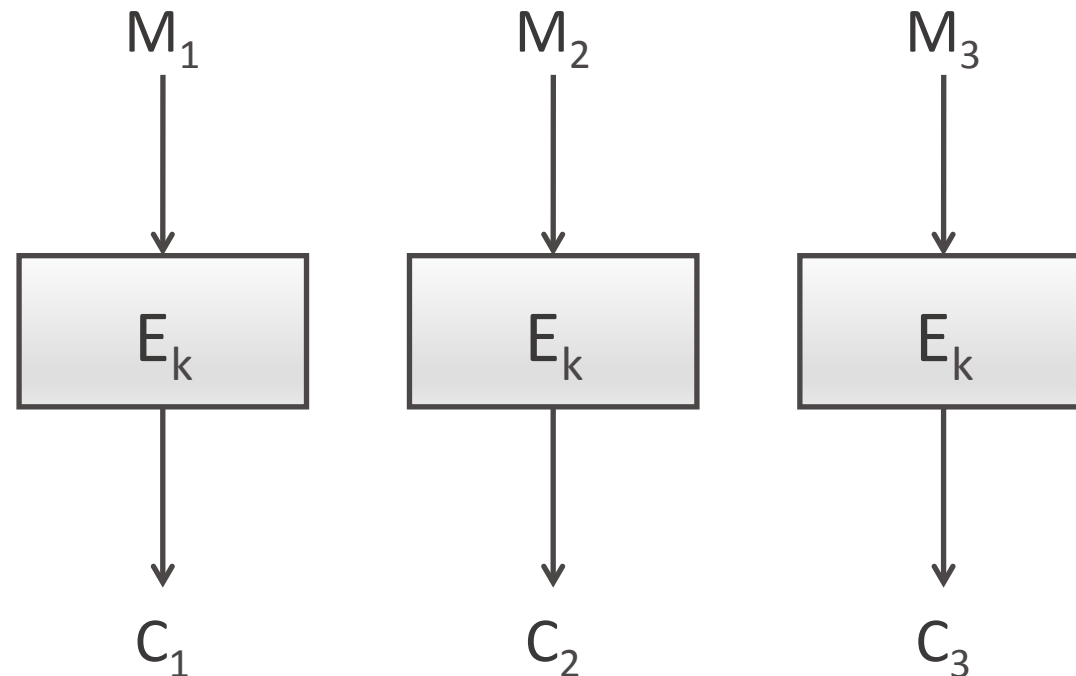
Modes of Operation

- 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

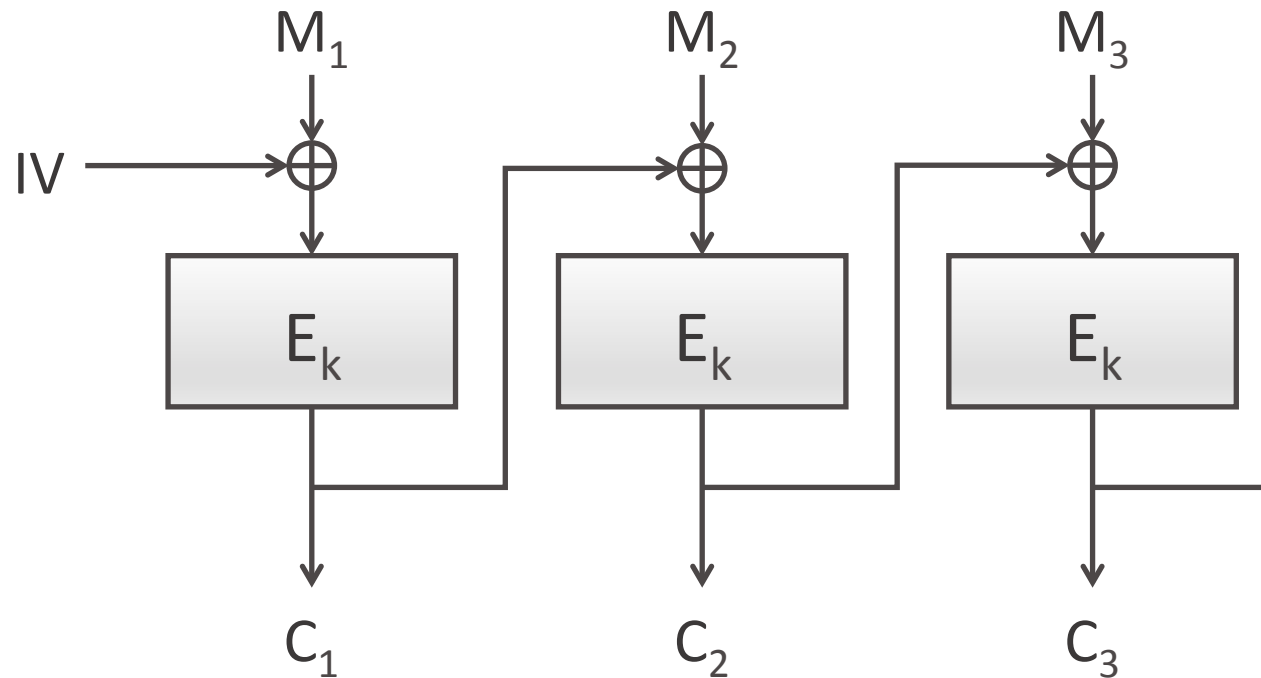
Electronic Code Book (ECB)

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



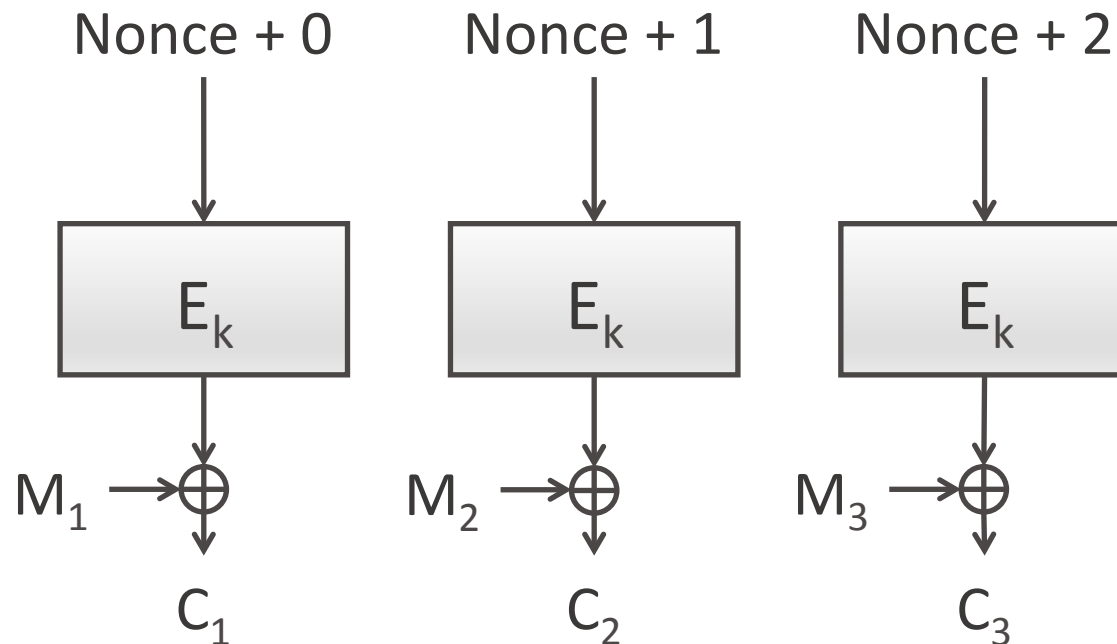
Cipher Block Chaining (CBC)

- 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
from cryptography.hazmat.backends import default_backend
backend = default_backend()

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(), backend=backend)

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

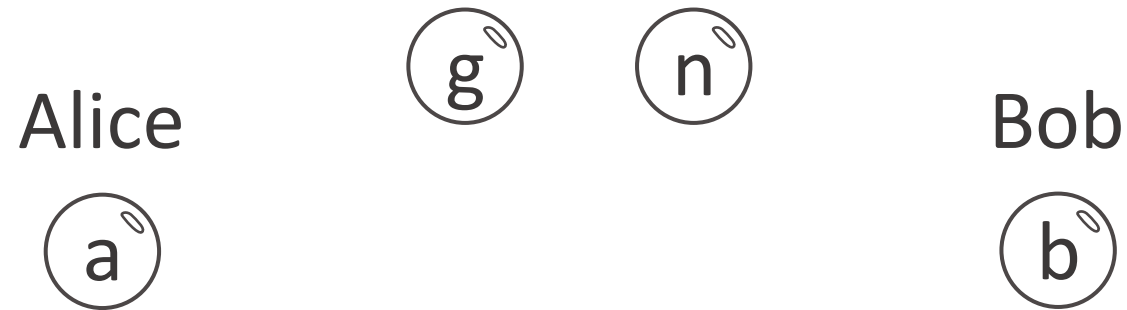
1. Alice and Bob agree on a set of base parameters



a generator g , and a very large prime number n

Diffie-Hellman

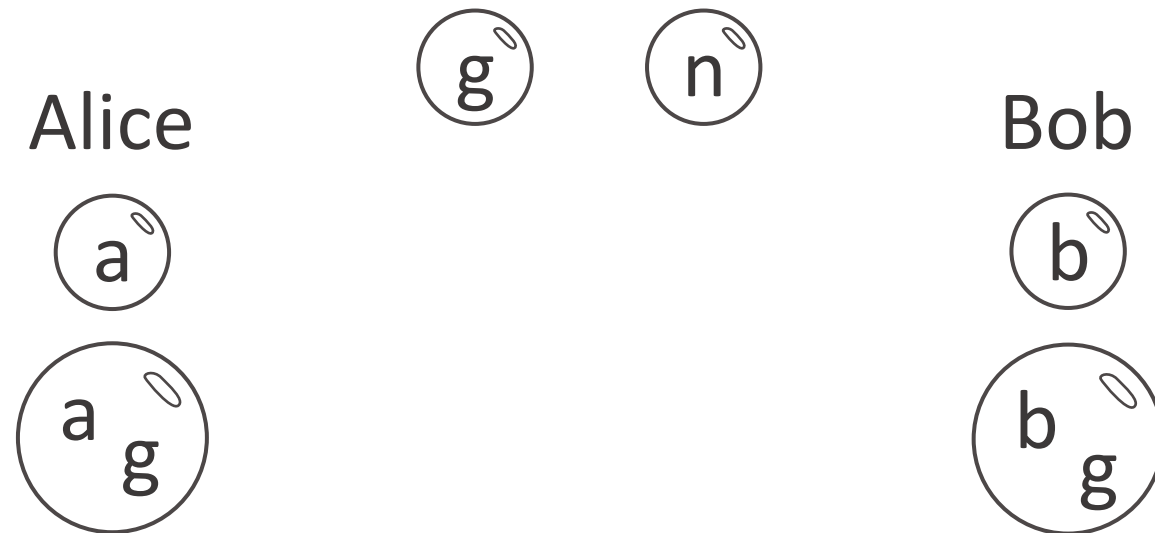
2. Alice and Bob select random numbers as their private keys



The private numbers are just values between 1 and n

Diffie-Hellman

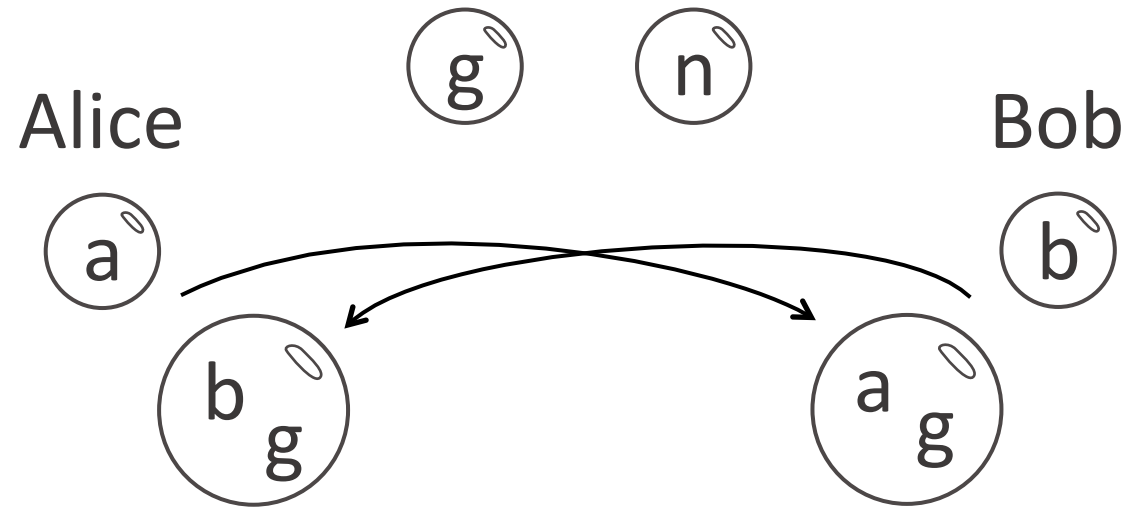
3. Alice and Bob each calculate a public key by mixing their private value with the generator



It's **extremely difficult** to extract the private key back out of these mixes

Diffie-Hellman

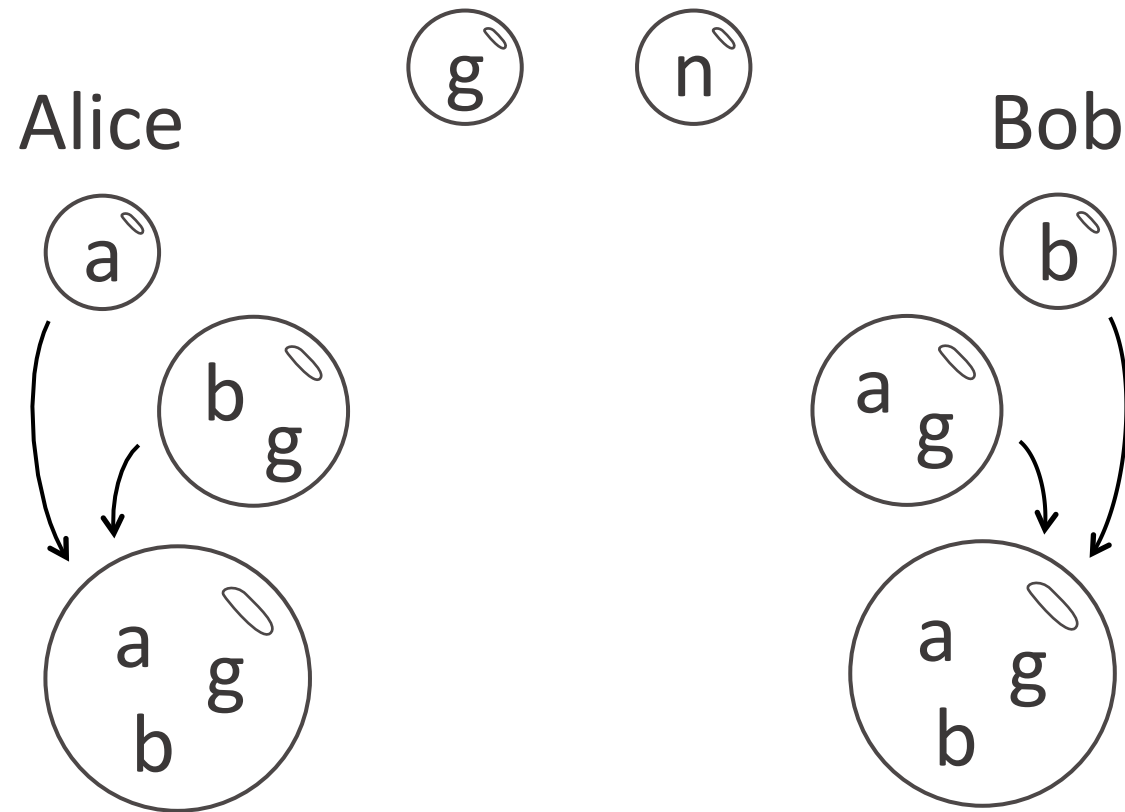
4. They swap these public keys over the wire – everyone sees these



This will usually be done using TCP, e.g. during a TLS handshake

Diffie-Hellman

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



Diffie-Hellman: Practice

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

Alice

Bob

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

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
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)
```



Diffie-Hellman: Practice

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

Alice

Bob

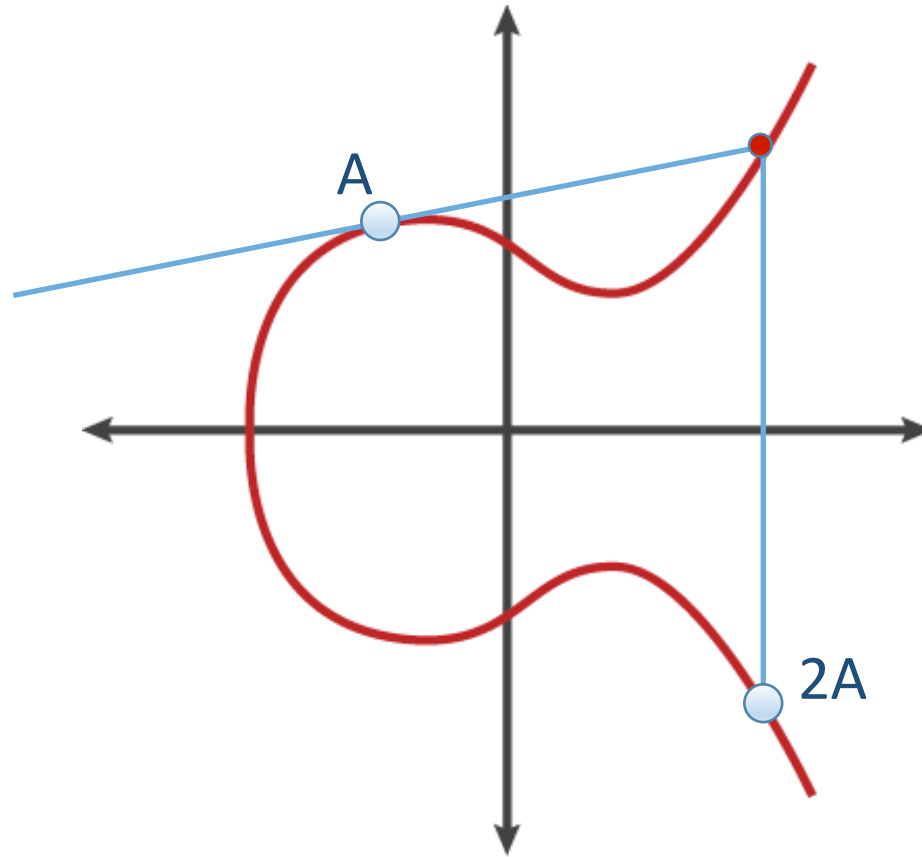
```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
hkdf = HKDF(algorithm=hashes.SHA256(), length=32, salt=None, info=None, backend=default_backend())

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()

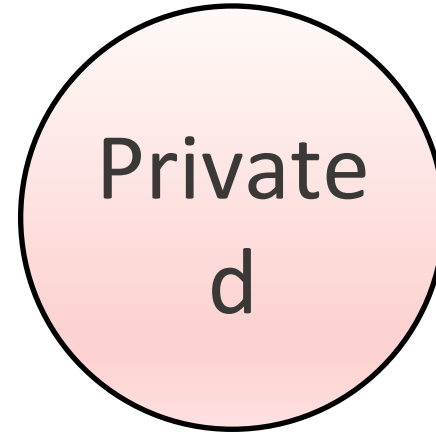
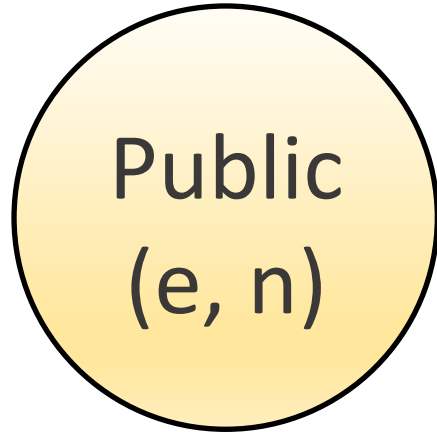
# Obtain servers public DH key - also normally over a network
server_public_key = server.get_public_key()
print ("The server's public key is: ", server_public_key.public_numbers().y)

### 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.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa

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

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.

Using RSA

- The keys (e, n) and (d) are reversible – either can be used for encryption, and the other used for decryption
- Everyone knows the public key, only the owner knows the private key
- This leads us to two very useful use cases for RSA:
 1. Encryption only the owner can read
 2. Signing that must have been performed by the owner

Encryption

Us

```
public_key = #RSA public key of server e.g. from internet

message = b"A secret message for the server."
ciphertext = public_key.encrypt(
    message,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Send ciphertext to server
```

Always use padding with
RSA, OAEP is recommended

Server

```
ciphertext = # Receive from sender

plaintext = private_key.decrypt(
    ciphertext,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
```

Signing

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

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!")
```

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

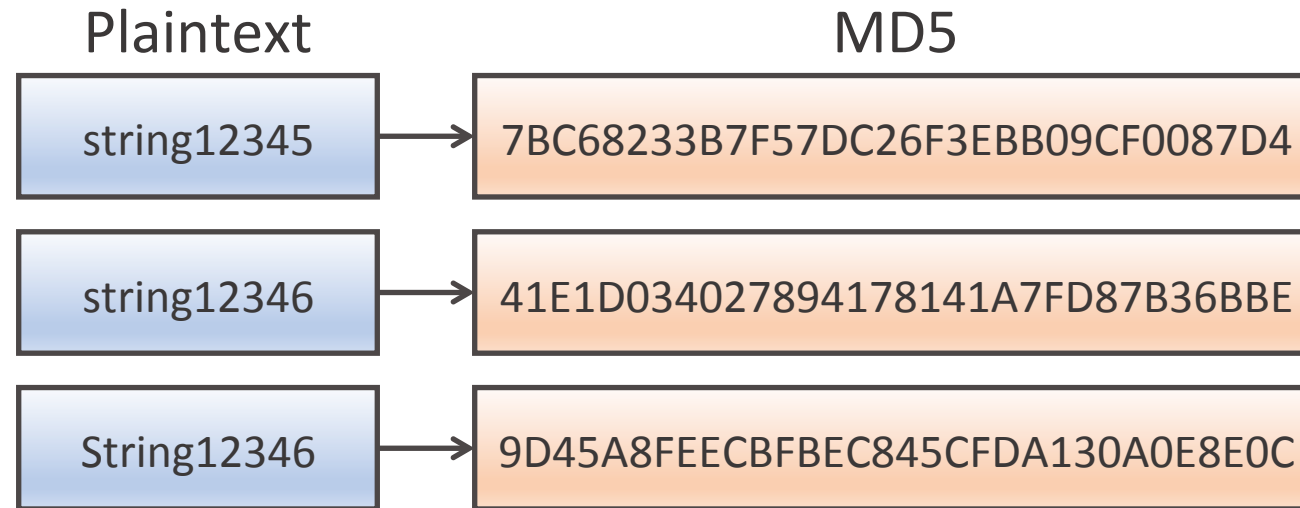
```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
digest.update(long_message)
digest.update(more_long_message)
h = digest.finalize()
print (len(h))
```

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

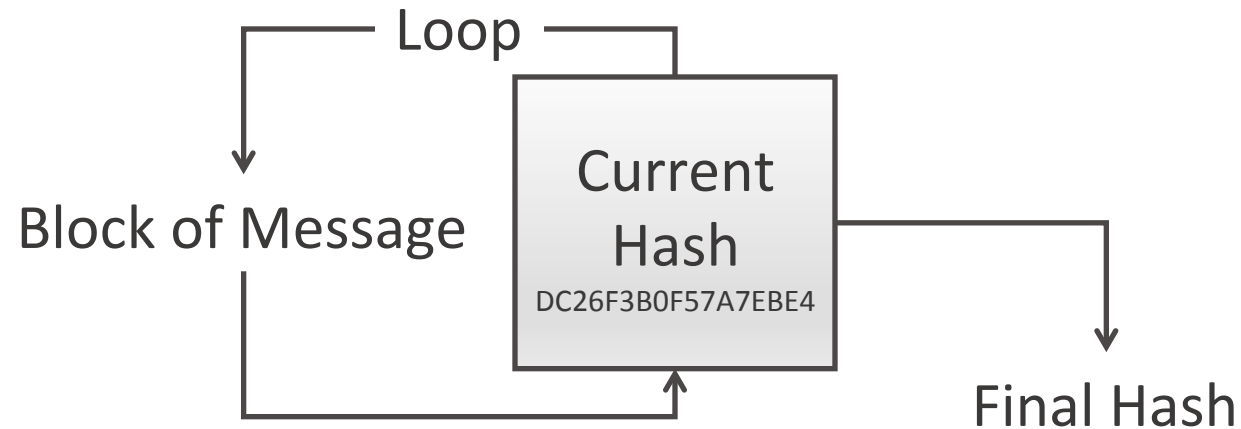


Strong Hash Functions

- 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

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

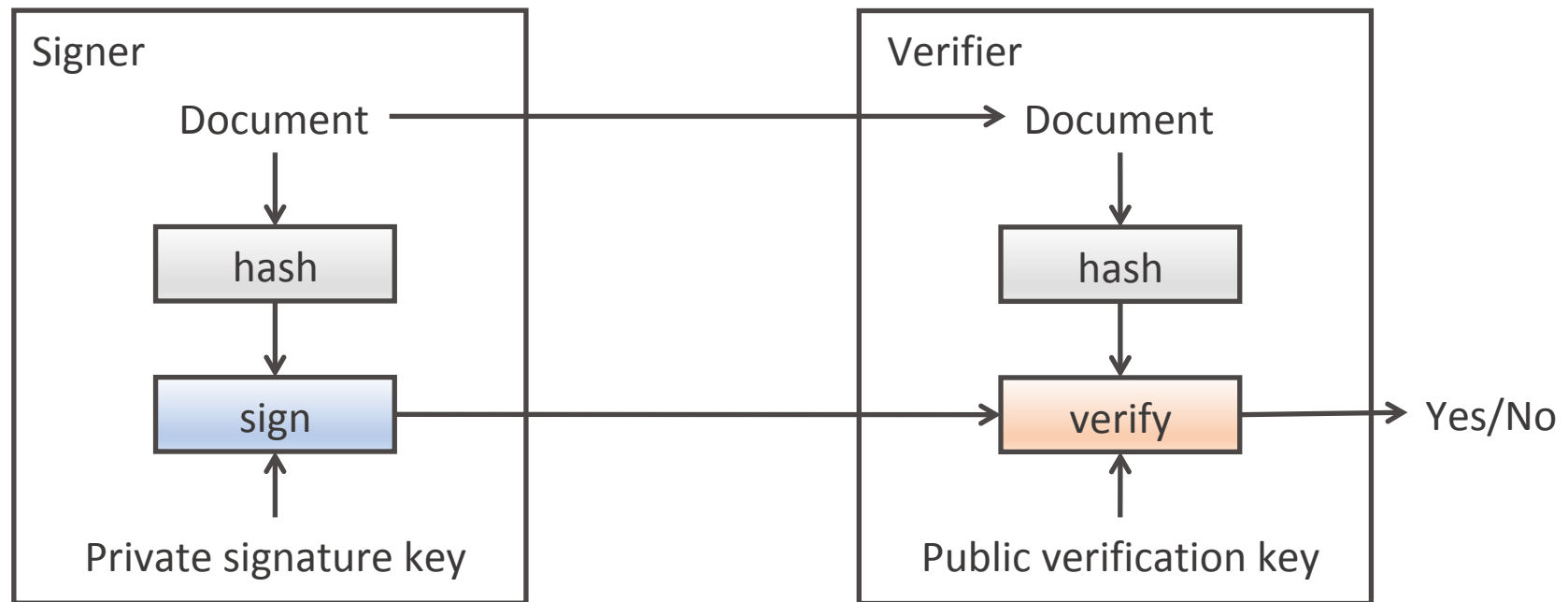
HMAC

- Standard MACs have a few issues due to the structure of common hash functions like SHA-256
- HMAC is the most common approach, it splits a key in two and hashes twice

```
import os
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac
key = os.urandom(32)
h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
h.update(b"message to hash")
h.finalize()
```

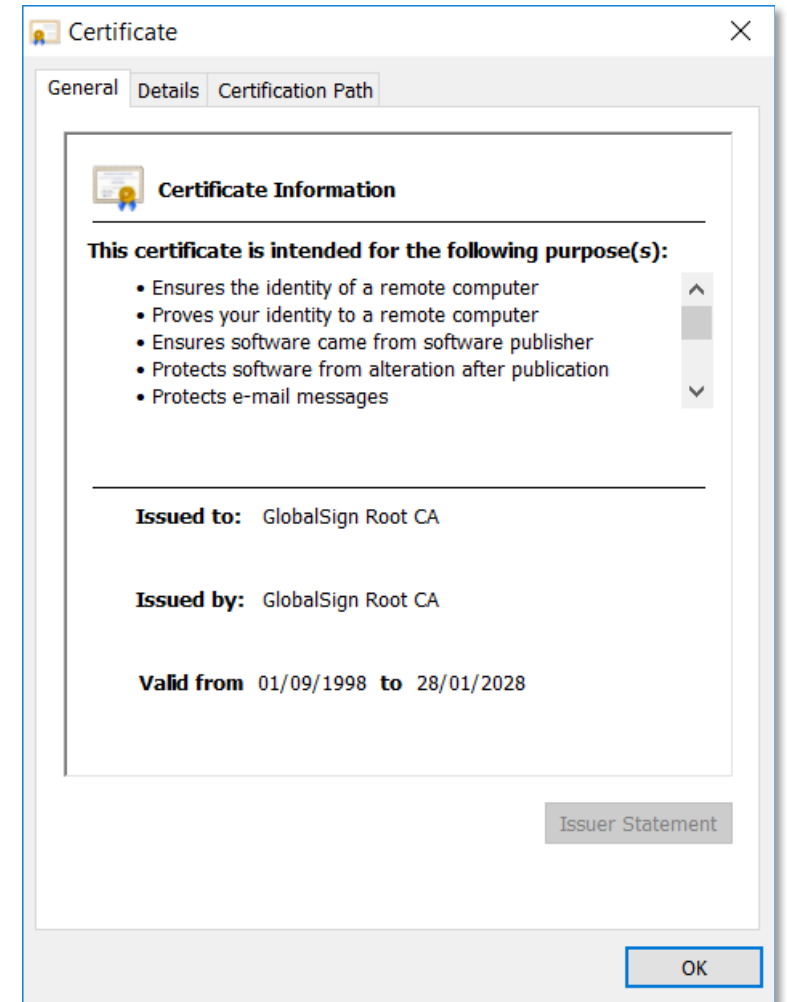

Digital Signatures – Hashing + PK

- 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)



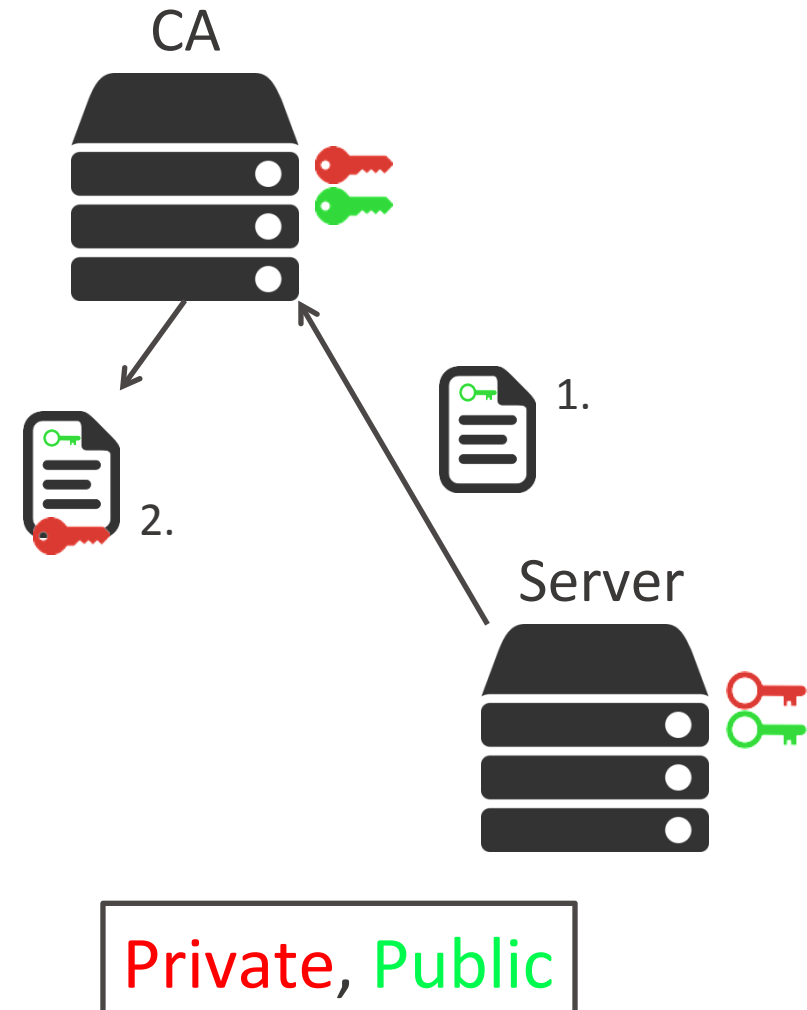
X.509 v3

- Organised by Public Key Infrastructure (PKI)
- The standard for digital certificates holds information on the type, subject and issuer
- The issuer will usually be a trusted third party, like Verisign or Globalsign

Version #
Serial number
Signature algorithm
Issuer name
Validity period
Subject name
Subject PK Info
Issuer ID #
Subject ID #
Extensions
CA Signature

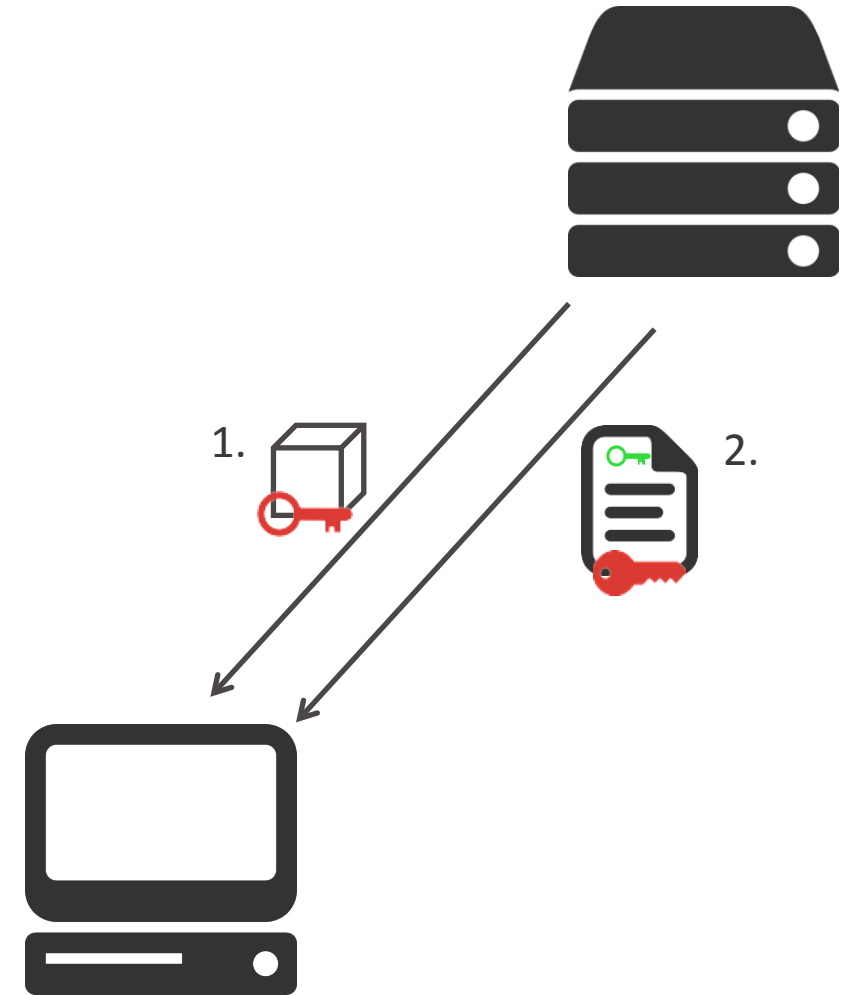
Using Digital Certificates - Issuance

1. Server produces a certificate containing their public key, which they want people to trust
2. They go to a certificate authority (CA), who after doing ID checks, sign the certificate with their private key



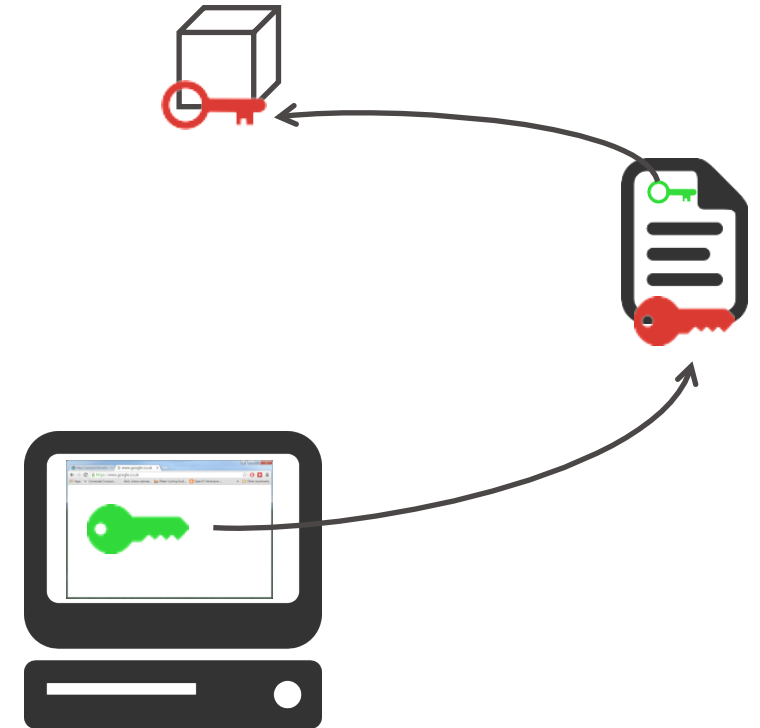
Using Digital Certificates - Authenticity

1. We want to use the server, they send us something (e.g. DHKEX parameters) signed with their private key
2. They send us a certificate, which has been signed by a CA for verification



Using Digital Certificates - Authenticity

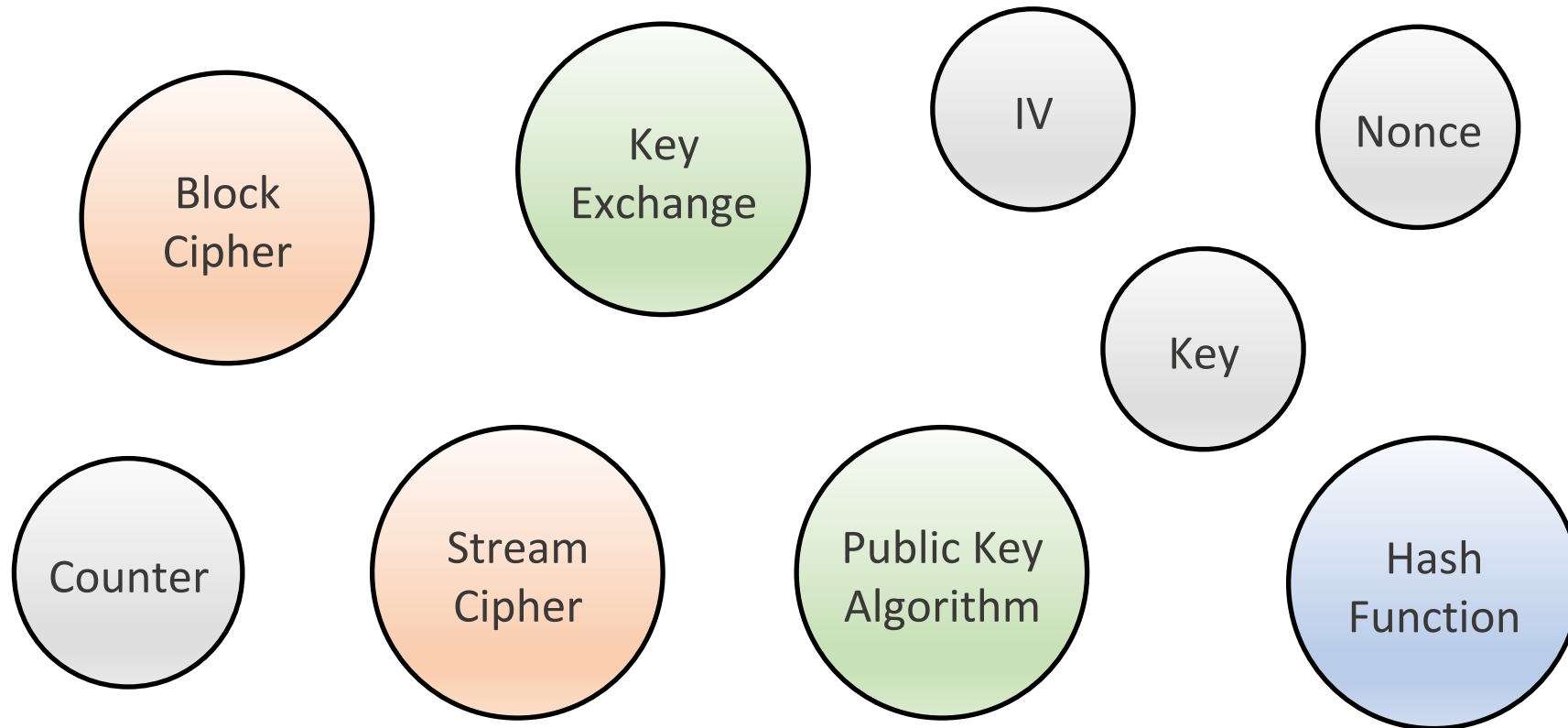
3. We have the CA public key stored in our browser. We trust the browser, so we trust the CA. We use the CA public key to verify the certificate
4. Now the certificate is verified, we trust it, and use the public key it gives us to verify the object



Complete Systems

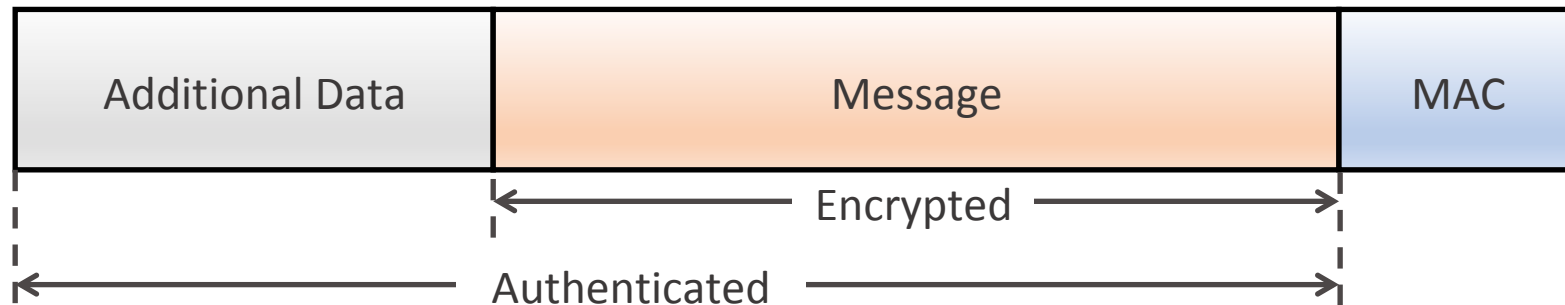
Protocols

- The building blocks of cryptography need to be carefully assembled into **protocols** if we are to use them safely



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, nonce + aad)
chacha.decrypt(nonce, ct, nonce + 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, nonce + aad)
aesgcm.decrypt(nonce, ct, nonce + aad)
```

```
'a secret message'
```

Exercise – AEAD

Encrypt and decrypt a database record using AEAD

aead.py

```
# This is our fictitious database record. For this exercise, this is just a local object, but could just as easily
# be stored in a real database.
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": "QQ123456C",
    "Phone": "01224103232",
    "Data": None,
    "Nonce": None,
}

}

### Task 1 ###
# Implement this function to take a record, and encrypt it using an AEAD cipher. You must adhere to the following:
def encrypt_record(record, key, nonce):
    ...
    plaintext = None
```

Protocol Handshakes

1. The 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

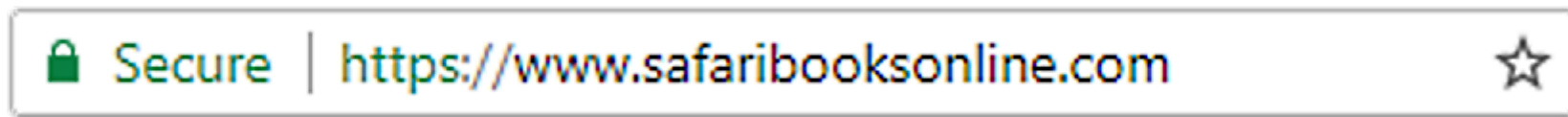
- Packets encrypted using the agreed cipher
- Includes a MAC or similar to verify integrity

Common Protocol Issues

- Ciphertexts that aren't secured with a MAC
- Messages that don't include a time-stamp or counter
- Protocols that don't use PK for authenticity
- Reuse of Nonces or IVs
- There are many more!

What can we do?

- 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 v1.2
- 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!

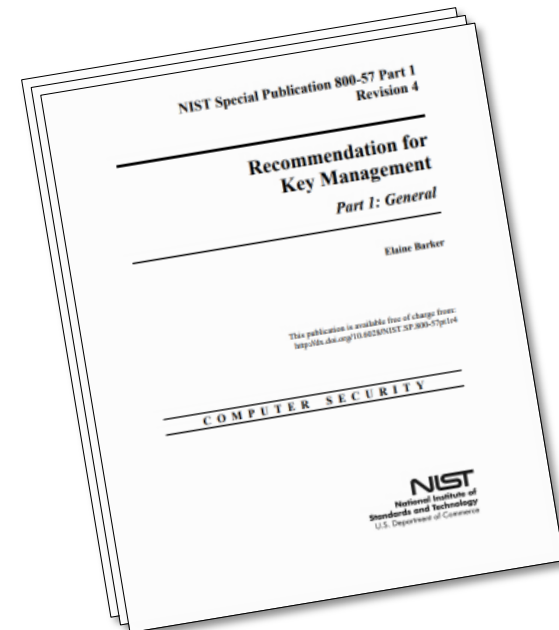
- Implementing your own algorithms is a bad idea!
- Never use hard-coded keys, convenience comes at a cost
- Never reuse IVs or Nonces
- Don't use ECB mode. Prefer CBC, but better yet CTR or GCM
- Don't use small public key sizes. At least 2048 bits! Elliptic curve is preferable: P-256 or X25519

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. 4



What We've Covered

Security Overview



This page is secure (valid HTTPS).

Valid Certificate

The connection to this site is using a valid, trusted server certificate.

[View certificate](#)

Secure Connection

The connection to this site is encrypted and authenticated using a strong protocol (TLS 1.2), a strong key exchange (ECDHE_RSA with P-256), and a strong cipher (AES_128_GCM).

Secure Resources

All resources on this page are served securely.

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
 - Cryptography Engineering: Ferguson, Schneier & Kohno.

