



Block Ciphers, TLS/SSL and HTTPS

COSC312 / COSC412

Learning objectives

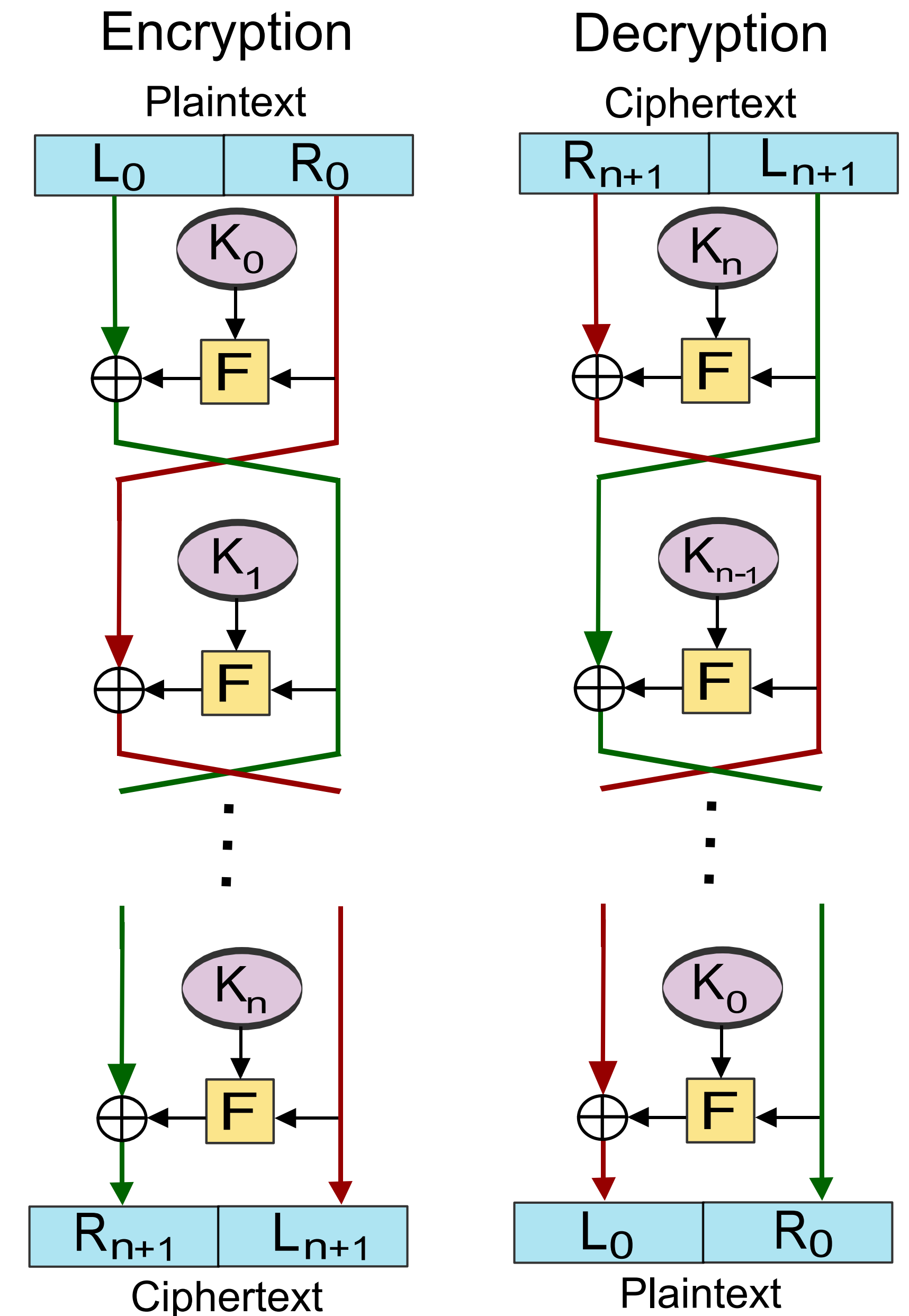
- Explain how **block ciphers** work (but not their crypto.)
 - Contrast block ciphers and stream ciphers
- Understand the **role of asymmetric cryptography** in practical protocols
 - Particularly HTTPS + TLS (*i.e.*, SSL)
- Appreciate structure and function of **digital certificates**

Key properties of block ciphers

- You have **already met stream ciphers**
 - Can think of a key-stream that gets combined with the plaintext to produce cipher text
- Block ciphers instead transform a **block of data at a time**
 - Effect of plain-text bits should be **distributed** throughout the cipher-text bits within block
- Note **block sizes** are often quite small
 - e.g., DES has 64-bit blocks, AES has 128-bit blocks

Example: Feistel ciphers

- Input plaintext is **split in two**: L_0 / R_0
- ‘Round’ **function F** is applied thusly:
 - F takes key K_n
 - Compute T as F transforming R_n
 - Result T is XORed with L_n
 - Input parts are swapped
- Feistel cipher property:
 - **No need to invert** function F to decrypt



Utilising block ciphers

- Block ciphers are used as a device in the construction of secure protocols
 - They are **not a complete means** to effect secure, encrypted communication on their own
- *E.g.*, when there is **more data than the block size**, we need a means for repeated block cipher utilisation
 - how to **generate sub-keys** K_n conveniently?
 - what if plain-text does **not match block size**?
 - these questions are unrelated to specific block ciphers in use

Let's meet aes-128-ecb

```
# (Visit COSC412 resources page for more information.)  
# On your computer run:  
git clone https://altitude.otago.ac.nz/cosc412/demo-vm  
cd demo-vm; vagrant up; vagrant ssh  
# Then, after SSHing to the VM, run:  
. /vagrant/bash-vars.sh
```

- Some simple plaintext:

```
: $; P=$(ruby -e '(0..127).each{|x|print x%8}'); echo -n $P | xxd -g1  
0000000: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37 0123456701234567  
0000010: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37 0123456701234567  
0000020: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37 0123456701234567 ...
```

- Using really terrible parameters, apply AES scheme:

```
: $; B_ENC='echo -n $P | openssl aes-128-ecb -K 61 -e -nopad -nosalt'  
: $; eval $B_ENC | xxd -g1  
0000000: 02 f9 c0 7b 93 46 fc 02 e0 67 48 40 d1 7e 33 19 ...{.F...gH@.~3.  
0000010: 02 f9 c0 7b 93 46 fc 02 e0 67 48 40 d1 7e 33 19 ...{.F...gH@.~3.  
0000020: 02 f9 c0 7b 93 46 fc 02 e0 67 48 40 d1 7e 33 19 ...{.F...gH@.~3. ...
```

- What can you say about this?

What would RC4 have done?

- Encrypting our plaintext with RC4 stream cipher gives:

```
: $; S_ENC='echo -n $P | openssl rc4 -K 61 -e -nopad -nosalt'  
: $; eval $S_ENC | xxd -g1  
0000000: b0 03 63 e6 1b 74 5d 28 a4 d2 bb 87 52 ad 20 42  ..c..t](....R. B  
0000010: d7 89 eb 33 a7 ec 30 19 e4 c3 49 78 14 7d 8d 64  ...3..0...Ix.}.d  
0000020: ee f7 d1 ce 9d 9e 96 ae 2c e4 87 a8 3c d3 16 99  .....<...  
0000030: 3d fa 72 4e 10 82 11 82 5c 08 a0 72 80 17 5c 35  =.rN....\..r..\5  
0000040: d6 0b a4 c9 81 3e bf 14 d5 7e 93 b5 fe 4d 78 80  .....>...~...Mx.  
0000050: f9 e3 7a 10 d2 18 8c 4a fc cd 67 ee 62 b5 f5 94  ..z....J..g.b...  
...
```

- Just FYI: note that we have taken care to handle null
 - Shell variables drop `\0` characters (they end the string), but...
 - Pipes can pass `\0` through streams, which is why we `eval` here

Block cipher 'modes'

- How to use block ciphers on more than a single block of plain-text? Select a block cipher '**mode**'.
- Common block cipher modes:
 - ECB: Electronic codebook
 - CBC: Cipher-block chaining
 - CFB: Cipher feedback
 - OFB: Output feedback
 - CTR: Counter

Explaining our cipher-patterns

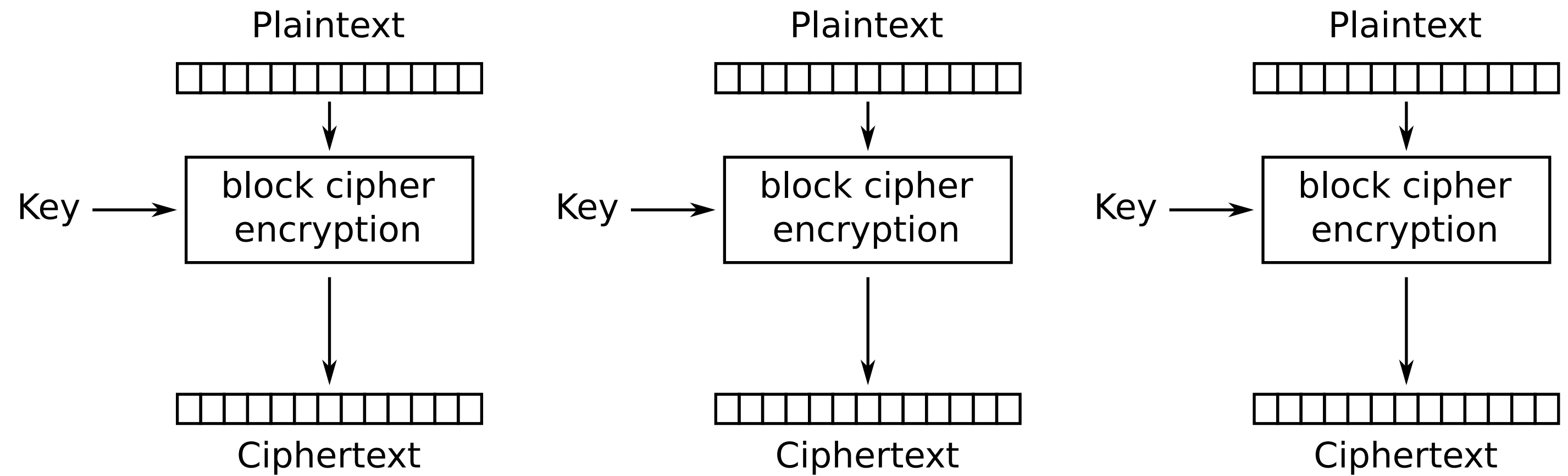
- Recall that we used aes-128-ecb a few slides ago
 - *i.e.*, Electronic Code Book block chaining
- What about aes-128-cbc? Output blocks don't repeat

```
: $; B_ENC='echo -n $P | openssl aes-128-cbc -K 61 -e -iv 61 -nopad -nosalt'  
: $; eval $B_ENC | xxd -g1  
0000000: e8 a0 fd b4 db 5a 0b ce 96 38 15 91 ce 86 82 81 .....Z...8.....  
0000010: 18 d3 cb a2 0b b9 20 45 66 b3 d5 bf 7d 22 32 ba ..... Ef...}"2.  
0000020: c6 36 63 4b 83 c8 aa d9 01 dd 04 81 d3 ad e9 8d .6cK.....  
0000030: 96 f4 73 fb b2 b2 9a b3 79 3e 0f 1a c3 98 5d 59 ..s.....y>....]Y  
0000040: 2c 45 e2 56 2d 6b 4b b6 80 bc ac fd 68 32 32 fb ,E.V-kK.....h22.  
0000050: 92 9f 86 6d 9d 95 7b c6 6f b5 b3 5c fe 7d 5c 29 ...m..{.o..\.})  
0000060: 7d 9e 20 7e d7 f0 30 7c 49 9c 6f db 17 a8 3f 7e }. ~..0|I.o...?~  
0000070: f7 b6 35 aa e0 fc f6 ce 68 39 85 4e e8 e1 68 67 ..5.....h9.N..hg
```

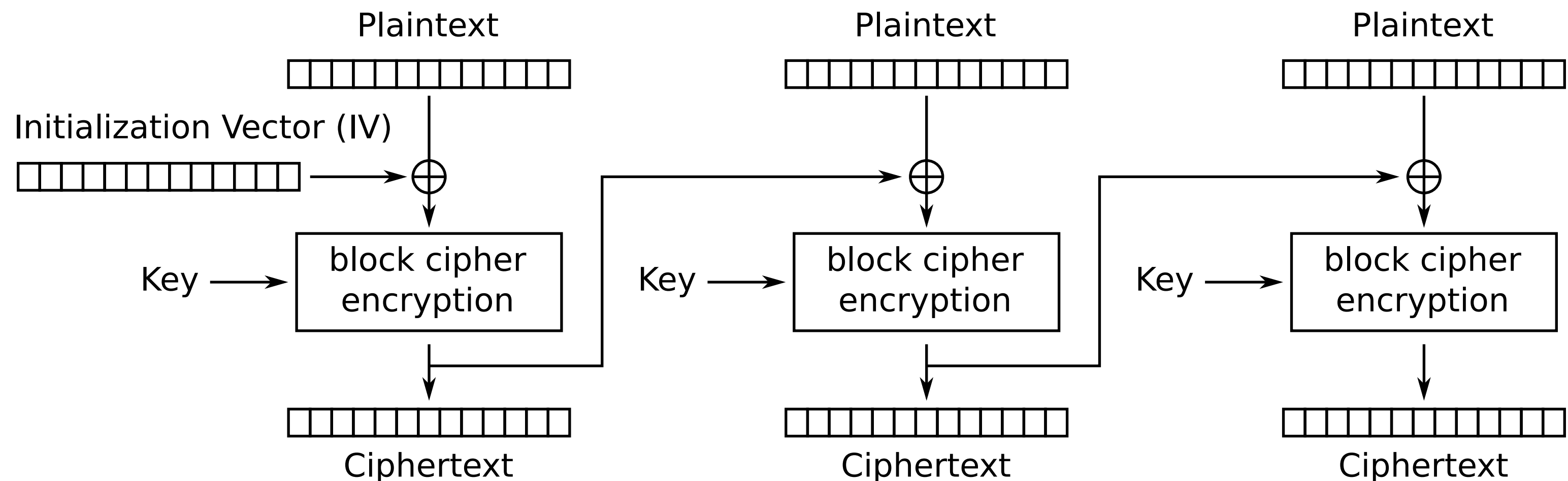
Explaining our cipher-patterns (2)

- The encryption operation of **two different block cipher modes** are shown here:

- ECB (top)
- CBC (bottom)



Electronic Codebook (ECB) mode encryption



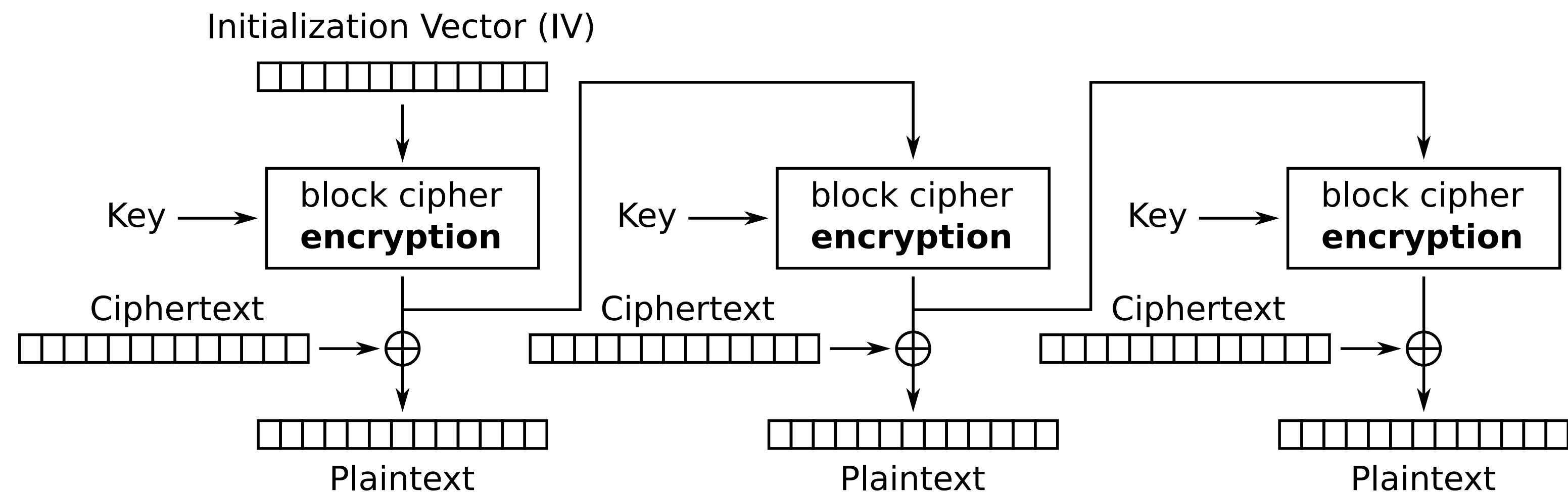
Cipher Block Chaining (CBC) mode encryption

Stream ciphers from block ciphers

- CBC can use the **same key for all blocks**
 - Convenient and straightforward interface
 - Just as for a stream cipher, **repeated data blocks** are not encrypted to matching cipher-text

- Even closer match:

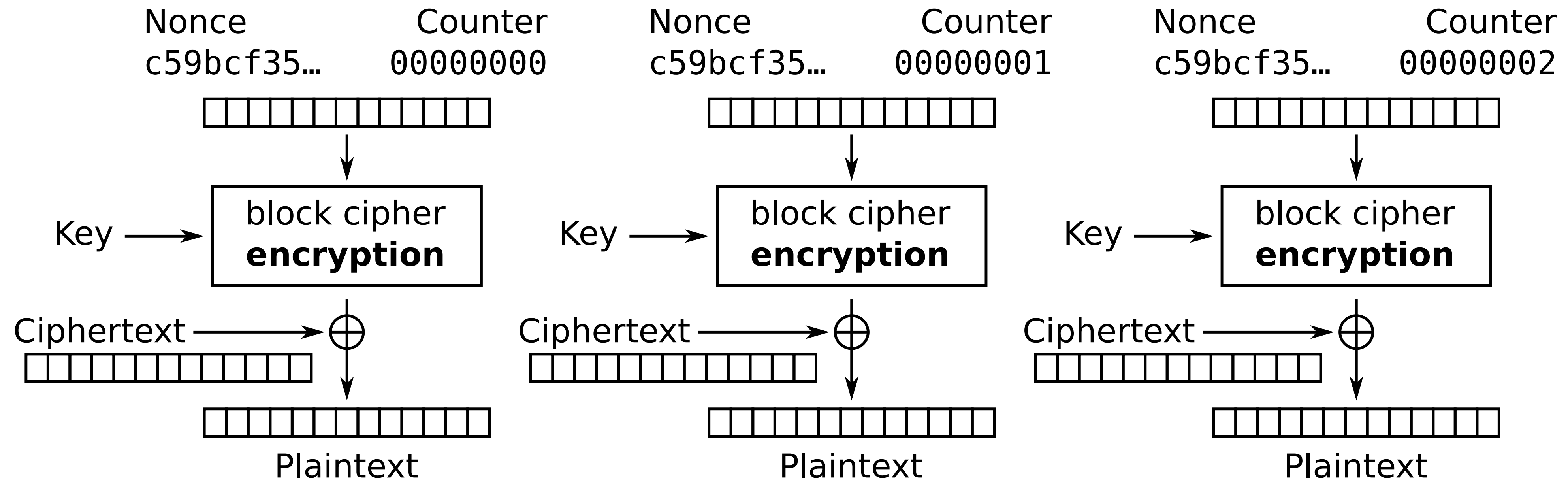
- OFB
- **only needs block encryption**, even for decryption



Output Feedback (OFB) mode decryption

Nice compromise: counter mode

- Similar to OFB, but allows random access



Counter (CTR) mode decryption

- Have we **actually** created a stream cipher?

Block ciphers may need padding

- Plain-text may be of **any length**
 - ECB and CBC (but not OFB) require input to be $n \times \text{block length}$
- How about filling last block with zeros (... ?)
- **Cipher-text stealing**: shuffles last two blocks
 - Allows variable length final block
 - Cost: **some bits are double-encrypted** & extra complexity
 - Also, it changes error propagation effects for last two blocks

Comparing stream and block ciphers

- Some bases for comparison:
 - Effects of different types of **data errors**
 - Different types of **attack susceptibility**
 - Types of **implementation**
 - **Speed** of encoding and decoding
- (... noting some block cipher modes generate a key stream, and so above points may not apply)

Errors: stream versus block ciphers

- Here's a transformer to change a byte in a stream:

```
: $; P=$(ruby -e'(0..127).each{|x|print x%8}') # as before
: $; echo -n "Test string" | LANG=C sed -r '1s/^(.{4})./\1B/' | xxd -g1
00000000: 54 65 73 74 42 73 74 72 69 6e 67          TestBstring ...
```

- Apply above transformer to **change a cipher-text byte** then decrypt the modified data stream:

```
: $; S_ENC='echo -n $P | openssl rc4 -K 61 -e -nosalt -nopad'
: $; eval $S_ENC|LC_ALL=C sed -r '1s/^(.{4})./\1B/'|openssl rc4 -K 61 -d -nosalt -nopad|xxd -g1
00000000: 30 31 32 33 6d 35 36 37 30 31 32 33 34 35 36 37  0123m56701234567
00000010: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37  0123456701234567
00000020: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37  0123456701234567
00000030: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37  0123456701234567
00000040: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37  0123456701234567
00000050: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37  0123456701234567
00000060: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37  0123456701234567
00000070: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37  0123456701234567
```

Errors: stream versus block ciphers

- Now see what happens with block ciphers...

```
: $; P=$(ruby -e '(0..127).each{|x|print x%8}')
: $; B_ENC_ECB='echo -n $P | openssl aes-128-ecb -K 61 -e -iv 61 -nopad -nosalt'
: $; B_ENC_CBC='echo -n $P | openssl aes-128-cbc -K 61 -e -iv 61 -nopad -nosalt'
```

- Output from ECB:

```
: $; eval $B_ENC_ECB | LC_ALL=C sed -r '1s/^(.{4})./\1B/' | \
openssl aes-128-ecb -K 61 -d -iv 61 -nopad -nosalt | xxd -g1
0000000: af 89 3d af 9a 7c cc cd 66 d9 6a a4 84 b1 8e e6  ..=..|..f.j.....
0000010: 30 31 32 33 34 35 36 37 30 31 32 33 34 35 36 37  0123456701234567 ...
```

- Output from CBC:

```
: $; eval $B_ENC_CBC | LC_ALL=C sed -r '1s/^(.{4})./\1B/' | \
openssl aes-128-cbc -K 61 -d -iv 61 -nopad -nosalt | xxd -g1
0000000: 71 d8 89 7d 35 61 b2 39 e4 71 43 f5 96 1c 66 a8  q..}5a.9.qC...f.
0000010: 30 31 32 33 ad 35 36 37 30 31 32 33 34 35 36 37  0123.56701234567 ...
```

Errors: stream versus block ciphers

- Let's change to **removing** a byte...

```
: $; echo -n "Some test string?" | LC_ALL=C sed -r '1s/^(.{4})./\1/' | xxd -g1
00000000: 53 6f 6d 65 74 65 73 74 20 73 74 72 69 6e 67 3f  Sometest string? ...
```

- Stream:

```
: $; eval $S_ENC|LC_ALL=C sed -r '1s/^(.{4})./\1/'|openssl rc4 -K 61 -d -nosalt -nopad| xxd -g1
00000000: 30 31 32 33 5b 1c 43 bb 46 58 0e e6 cb 92 7e a2  0123[.C.FX.....~.
00000010: 6e 53 ea a7 7f e9 1f ca 17 bb 03 5f 5d c5 df bd  nS....._]... ..
```

- Block:

```
: $; eval $B_ENC_ECB|LC_ALL=C sed -r '1s/^(.{4})./\1/'|openssl aes-128-ecb -K 61 -d -iv 61 -nopad -nosalt|xxd -g1
bad decrypt ...
00000000: 6f 5a e3 07 35 fe 24 3d af 71 7e d5 22 ae 1b ea  oZ..5.$=.q~."...
00000010: 1d e3 5b 0a 29 9d 46 fe 8e 35 3e 33 ae 9d 2f a2  ..[.)].F..5>3../.
00000020: 1d e3 5b 0a 29 9d 46 fe 8e 35 3e 33 ae 9d 2f a2  ..[.)].F..5>3../. ...
: $; eval $B_ENC_CBC|LC_ALL=C sed -r '1s/^(.{4})./\1/'|openssl aes-128-cbc -K 61 -d -iv 61 -nopad -nosalt|xxd -g1
bad decrypt ...
00000000: eb 8a 8e ef ed b0 85 8f aa ea fd 95 63 d5 9e 99  .....C...
00000010: 78 73 63 ea 33 e8 b2 5c f1 d3 40 5a 6d 78 94 ed  xsc.3..\...@Zmx..
00000020: 0e 6c 88 02 60 8a 70 80 44 47 4e 1f 7c c8 28 08  .l..`.p.DGN.|.(. ...
```


Attacks: stream versus block ciphers

- For example, stream ciphers' **error-handling poses risk:**
 - Attacker can know where bit flips will affect damaged plain text decoding
 - How can we address this risk?
 - Key stream needs a large period to be (potentially) secure
- Or, block ciphers' final block padding:
 - 1-bit followed by 0s to pad last block is secure
 - Other schemes can allow 'padding oracle' attack

Implementation: stream versus block

- Bulk encryption? **Hardware support**, for example:
 - Stream cipher: linear feedback shift-registers
 - Block cipher: substitution-permutation networks
- All sorts of **subtle risks and concerns**:
 - e.g., timing side-channels; susceptibility to cryptanalysis
- General message remains: don't make assumptions about crypto implementations!

Web security

- Potential usefulness of **authentication** became clear early in WWW development
 - Recall that web systems transfer HTML data over HTTP
- **HTTP authentication** is not an HTML function:
 - Hence it presents a different user interface from HTML forms
 - Can protect non-HTML content
- We will **look at TLS/SSL** and how it supports web security

HTTP basic authentication

- Require users to authenticate before they are allowed to access content
- Client sends HTTP GET / POST / etc.
- Server responds with 401 including header:
 - `WWW-Authenticate: Basic realm="..."`
- Client prompts user for authentication
- Client sends original request, with header:
 - `Authorization: Basic ...`

HTTP basic authentication... TCP/IP

- In your COSC412 VM, set up Apache `: ~$; /vagrant/setup-apache.sh`
 - Use `tcpdump` wrapper on the VM; point your browser to:
<http://localhost:8180/1-basic-auth/> (user/password test/test)

```
: ~$; /vagrant/tcpdump-web.sh
...
10:34:57.560065 IP ubuntu-focal.http > _gateway.62236: Flags [P.], seq 1:736, ack 453, win 63788, length...
HTTP/1.1 401 Unauthorized
Date: Fri, 04 Aug 2023 10:34:57 GMT
Server: Apache/2.4.41 (Ubuntu)
WWW-Authenticate: Basic realm="Authentication Required"
...
10:35:02.162909 IP _gateway.62239 > ubuntu-focal.http: Flags [P.], seq 1:488, ack 1, win 65535, length 487:...
GET /1-basic-auth/ HTTP/1.1
...
Authorization: Basic dGVzdDp0ZXN0
...
```

HTTP digest auth. [RFC2617]

- Improves on HTTP basic (passwords hidden!)
 - Still not an acceptably modern type of security
 - Server sends to client:

```
WWW-Authenticate: Digest realm="Badly protected area",  
nonce="ad30AgQABQA=3a714b04208e85496e47d482a8bac297c78b6887", algorithm=MD5, qop="auth"
```

- Client sends to server:

```
Authorization: Digest username="test", realm="Badly protected area",  
nonce="ad30AgQABQA=3a714b04208e85496e47d482a8bac297c78b6887", uri="/2-digest-auth/",  
response="6b21c1d403b8559fccc860afb2653df0", algorithm="MD5", cnonce="0389790832a7e8db5290d6e9b63276ed",  
nc=00000001, qop="auth"
```

```
: ~$; read HA1 D < <(echo -n 'test:Badly protected area:test' | md5sum)  
: ~$; read HA2 D < <(echo -n 'GET:/2-digest-auth/' | md5sum)  
: ~$; echo -n \  
"$HA1:ad30AgQABQA=3a714b04208e85496e47d482a8bac297c78b6887:00000001:0389790832a7e8db5290d6e9b63276ed:auth:$HA2" |  
md5sum  
6b21c1d403b8559fccc860afb2653df0 -
```


HTML forms-based authentication

- Authentication using HTML forms (above HTTP-level)
 - HTML form submits values to server (e.g., using POST method)
 - Login done in web domain: allows styling, JavaScript, *etc.*
 - But... still no transport-level security:

```
: ~$; /vagrant/tcpdump-web.sh
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
...
09:22:42.398760 IP _gateway.54722 > ubuntu-focal.http: Flags [P.], seq 1:531, ack 1
... Content-Type: application/x-www-form-urlencoded ...
Accept-Encoding: gzip, deflate
...
username=test&password=somepassword
```

A crucial tool: TLS (*i.e.*, SSL)

- Transport Level Security / Secure Sockets Layer
 - Put security at a **lower layer of the network stack** than HTTP
- Establish **secure end-to-end connection**
 - Need a handshaking protocol to establish identities, choose crypto algorithms, *etc.*, below the application layer
- Implementation is **complex!**
 - *c.f.*, OpenSSL ‘heartbleed’ problems

TLS/SSL handshake, *e.g.*, HTTPS (1/3)

- Client opens TCP connection to port 443
- Starts negotiating session details by sending:
 - {Session ID, cipher schemes and their key sizes, compression algorithms}
- Server may resume based on session ID
- Or server creates new session, sending back:
 - {Selected cipher + key size, selected compression algorithm, server certificate, optional client authentication request}

TLS/SSL handshake, *e.g.*, HTTPS (2/3)

- Client: authenticates the server's (S) certificate
- Generates symmetric key PM (pre-master), encrypts with server's public key (S_{pub})
 - $\{E(S_{pub}, PM), \text{client certificate (if requested)}\}$
- Server: authenticates client's certificate, if required
 - If needed uses S_{pri} to decrypt and get PM
- Client and server use PM to generate session key (master key) M

TLS/SSL handshake, *e.g.*, HTTPS (3/3)

- Client says to server it will start using M
- Sends a separate message, encrypted using M , that it has finished handshaking
- Server does the same in response

- Now data can be transferred using $E(M, \text{data})$
- The session key can be re-keyed periodically

HTTP versus HTTPS in action

- Request / response using HTTP

```
: ~$; /vagrant/tcpflow-web.sh
tcpdump: listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
GET /0-no-auth/ HTTP/1.1
Host: localhost:8180
Sec-Fetch-Site: none
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Mode: navigate
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.5.2 Safari/605.1.15
...
HTTP/1.1 200 OK
Date: Fri, 04 Aug 2023 10:42:39 GMT
Server: Apache/2.4.41 (Ubuntu)
...
Content-Length: 270
...
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> ...
```

HTTP versus HTTPS in action

- Request / response using HTTPS

```
: ~$; /vagrant/tcpflow-web.sh
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
.....S.<L0..QIp...J...88.N..z
bw..4:...J...$.#.
.=.<./.....5.....&%.*).....
.g.k.3.9.....4.....localhost.
.....
.....
....._.....N.X. .&o...qW..0b(..".x.^k.m...../...=.....0...0.....0...0.1.0
...U...precise640.."0
.....-p.Y...4.....X../.l.....a.(.....J7u.5t@.'..
scgn.....2..{.....8.....8].....R"s.].....1$.3\...*..0Tt.....a.l..}..rZ|
G.."..'X....._..a..DX.....~/...e.F.(.:.....~..?...
+...3.....y.&W...bk.....o.).....v.;..."!.../...,.3...L ..j.....T.NQ..)..k..U.y?...
-...Ms:...,)SI.r.....w.....P.>.g.{l..^..gf.Y..3.$..3{.....?.....
...
...
```

Very brief history of SSL / TLS

- Netscape released **SSL v2.0** in mid 1990s
 - **Cryptographic weaknesses** in the protocol (handshaking)
- **SSL v3.0** incorporates additional ciphers
 - Key derivation is weaker than in SSL v3.1
- **SSL v3.1 = TLS v1.0** uses both SHA1 and MD5
 - **TLS v1.1** cleans up CBC IV (BEAST attack)
 - **TLS v1.2** adds access to more crypto methods
 - **TLS v1.3** released in 2018: improved protocols; cipher refresh

Protocol use of SSL / TLS

- Probably HTTPS is **most common TLS use**
 - Also email protocols: POP3S, IMAPS
 - File transfer: FTPS (not SFTP!)
- Often TLS use will be **based on port, e.g., 443**
 - Problem: application only gets in after TLS handshake
- TCP/IP connection **need not start encrypted**
 - STARTTLS instruction can facilitate plain-text setup
 - Useful for multiplexing connections

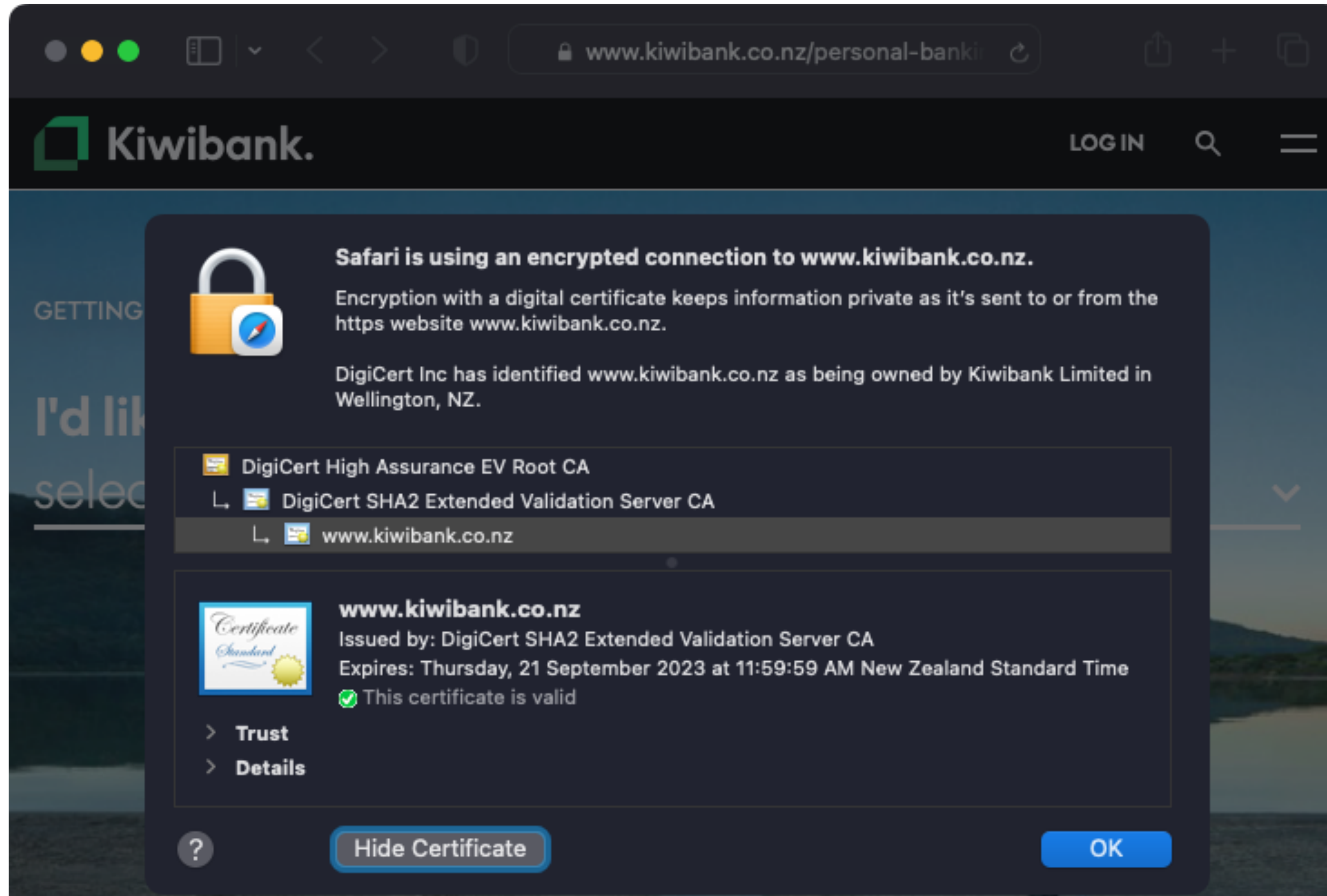
HTTP(S) virtual names

- HTTP/1.1 added `Host:` header
 - Allows one IP address to **host multiple sites**
- ...but HTTPS applied port-based connections
 - TLS handshake finishes **before server sees HTTP**, thus
 - ... server does not know **which certificate** to show
 - Wildcard certificates and `subjectAltName` extension can help
- Now **Server Name Indication (SNI)** common [RFC 6066]
 - Client can add extension with target host name

What SSL/TLS does not achieve

- Transport Layer Security encrypts end-to-end
 - ... but only as a **stream between two hosts**
- Increasingly server-side could benefit from not decrypting **all** content on its arrival
 - Complex caching, load-balancing in the cloud
- Needs lots of **management**:
 - Filter allowed algorithms, check randomness
 - Certificate management: validation; expiry; ...

Digital certificates



Digital certificates have many parts

- Recap: certificate ‘proves’ you own a public key
- Flexible encoding (e.g., ASN.1) of certificates stores:
 - A **public key** (so can send data to the certificate’s owner)
 - **Issue details**: name (X.500), serial number, lifetime
 - Indicate **allowed uses**, and other metadata
 - **Digital signature** (...checked using public key)
 - How do we check this digital signature’s key?
- **X.509 standards** cover web certificates

Our test server has a certificate

```
: ~$; sudo cat /etc/ssl/certs/ssl-cert-snakeoil.pem | openssl x509 -text
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      61:a1:b8:87:fd:c3:95:82:83:f1:30:69:77:5f:49:71:f9:3a:fe:c2
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: CN = ubuntu-focal
    Validity
      Not Before: Aug  4 10:34:12 2023 GMT
      Not After : Aug  1 10:34:12 2033 GMT
    Subject: CN = ubuntu-focal
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public-Key: (2048 bit)
      Modulus:
        00:c5:07:82:40:d8:03:ba:fe:50:6c:b8:82:05:b8:
...
      Exponent: 65537 (0x10001)
...
    Signature Algorithm: sha256WithRSAEncryption
      34:8d:40:2e:2f:7e:b4:3a:34:73:40:fa:67:f7:e4:1b:54:d9:
      7a:c1:cd:3d:e2:fc:0a:a2:5c:30:0e:ac:3a:cf:7d:89:3c:a4:
...
```


We previously discussed trust setup

- Browser has pre-loaded **root certificates**
 - Actually also has intermediate certificates: **Why?**
- You can declare that you trust a certificate
 - Like SSH: you **can approve exception**
 - Also like SSH, be sure that you actually want to!
- Can import **your own certificates** into your browser
 - Necessary for client-side authentication
 - ... which hardly seems to get used these days

Verification of a certificate's quality

- First: certificate **should validate on own terms**
 - *i.e.*, should not be outside valid date range
 - Cryptographic verification of keys is successful
- However that's not all: **beware weak cryptography**
 - Keys will specify algorithms they use: may be old
 - Are the CAs that are used trustworthy?
- A “perfect” certificate may need revocation—**why?**

Revocation of digital certificates

- Private key revealed/stolen? Certificate use must stop!
 - The **compromised certificate is dangerous**:
 - taken on its own, **it's internally consistent & digitally "correct"**
- Owner requests CA add certificate's serial number to **Certificate Revocation List (CRL)**
 - CRL is signed by CA, but clients have to get it to act on it!
- CRLs are likely to be growing over time
 - (Why won't they grow without bound?)

Online Certificate Status Protocol

- OCSP aims to be **more lightweight than CRLs**
 - OCSP responder sends pub-key signed “certificate is OK” messages when asked
 - ... however OCSP **can suffer replay attacks (!)**
 - Not all servers support **use of a nonce (!!)**
- Only in the last few years have web browsers started supporting it by default
 - If in doubt, use Firefox... (in recent history it has **picked secure default choices** even at the potential expense of speed)

HTTPS certificates can be acquired for free

- Let's Encrypt provides **free certificates automatically**
 - Uses a challenge/response method to have the requestor prove that they have access to resources
 - Common approach is to request **demonstration of the ownership of a particular DNS domain**
- No CA attestation of your affiliation... no EV certificates
 - ... but do get end-to-end encryption



Summary

- **Explored block ciphers**, and contrasted their behaviour to stream ciphers
- **Discussed web security**, and how cryptography has become involved in web technology
- **Examined digital certificates**, and how they use cryptography to help security engineering