Implementing Cryptography #3

Ed Zaluska

COMP6230:

Implementing Cybersecurity

DES details

- DES uses a combination of **permutation** (scramble bits – easy in hardware) and **substitution** (table look-up)
- the permutation uses a 'P box'

  (32 inputs, 32 outputs)
- the substitution uses an 'S box'

  (6-bit input, 4-bit output)

  (in fact 8 S-boxes are used in parallel)
- the algorithm is a *Feistel cipher*

  (after Horst Feistel), used in many block ciphers (but not AES)

Feistel cipher algorithm

- split the block into two : for DES, we have a 64-bit ‘P’ block which gives a 32-bit ‘L’ block and a 32-bit ‘R’ block
- use R and the key K to produce a 32-bit ‘M’ block (details later)
- generate L* from L XOR M
- note R is unchanged at this point:
- swap the two blocks: set L = R, set R = L*
- repeat for a number of iterations (or *rounds*) (16 for DES)
- decryption simply works backwards: the final R is unchanged, hence the final M block can be re-created and XOR’d with the final L to give the previous R.

DES details (2)

- the 56-bit key K is expanded into 16 48-bit subkeys (one per round) (key bits are simply re-used with an (over-)complex scheme)
- the 32-bit R block is expanded into 48-bits and XOR’d with the subkey
- the 48-bit result is split into 8 sets of 6-bits, these are the inputs to the S-boxes
- the 8*4 = 32-bit outputs go through the P-box
- the output of the P-box is the M-block

DES details (3)

- the S and P boxes are *unchanged* between rounds
- the S-box provides a ‘one-way’ function: the 4-bit output could have come from *any* of four *different* 6-bit inputs
- compare the S-box function with *hashing* (N:1 mapping)
- decryption simply goes backwards
- (N.B. XOR reverses the original changes)

DES details (4)

- S1 contents:

  14 4 13 1 2 15 11 8 3 10 6 12 5 9 0 7

  0 15 7 4 14 2 13 1 10 6 12 11 9 5 3 8

  4 1 14 8 13 6 2 11 15 12 9 7 3 10 5 0

  15 12 8 2 4 9 1 7 5 11 3 14 10 0 6 13
- 64 entries, each number 0..15 appears exactly four times
- S2 to S8: used in parallel to give 32 bits

Advanced Encryption Standard (AES)

- 1997 open call for proposals for a DES replacement: world-wide response
- five finalists: exhaustive public scrutiny and evaluation
- winner of the competition was Rijndael
- (other finalists also had good security, but Rijndael was faster: Rijndael 86 votes, Serpent 59 votes, Twofish 31 votes, RC6 23 votes, MARS 13 votes)
- Belgian cryptographers: Vincent Rijmen and Joan Daemen
- 26 May 2002 FIPS 197

AES details

- symmetric block cipher
- design principle is *substitution-permutation*
- 128 bit block (internally 4x4 byte array, known as the *state*)
- key length 128, 192 or 256 bits
- iterations (*rounds*) 10 (12/14 longer keys)
- the longer key lengths are approved by NSA for **top secret** US Government documents!
- very fast execution (hardware and software)

AES details - steps 1 & 2

- each round has four *steps*
- step one: byte substitution – look-up each byte in the state in an S-box

  (N.B. 256 entries, completely different to DES S-box)

  1:1 mapping hence *invertible*

  (just a *monoalphabetic substitution cipher*)
- step two: shift rows – rotate bytes in each row to the left: row 0 (top) – no shift, row 1- shift 1,

  row 2- shift 2, row 3 (bottom) – shift 3

AES details – step 3

- step 3: mix columns – new column is the product of the old column with a constant 4x4 matrix [M] (this is the main source of diffusion in Rijndael)
  i.e. [new_column] = [M] [old_column]
- Galois Field arithmetic is used – GF(256)
  additions just XOR, multiplication more complex
- essential that the constant matrix selected has an *inverse* i.e. [M] [M’] = [I]
- decryption uses the *inverse matrix* for this step

AES details – step 4

- step four: XOR state with round key

  (the round keys are derived from the main key, just as with DES)
- note that each step is reversible
- (i.e. 1:1 mapping)
- hence decrypt by running algorithm backwards

Block Cipher Modes - 1

- *Electronic Code Book* (ECB) mode
- do **NOT** use!
- split P/T into (e.g.) 128-bit blocks, each enciphered independently
- Ci = E(K, Pi)
- just a monoalphabetic substitution cipher (albeit with a very large alphabet)
- **cannot** detect missing/duplicated/re-ordered C/T blocks

Block Cipher Modes - 2

- *Cipher Block Chaining* (CBC) mode (IBM 1976)
- XOR each P/T block with *previous C/T block* before encrypting

  $C_i = E(K, P_i \text{ XOR } C_{i-1})$
- (obviously reverse when decrypting)
- what about the first P/T block? precede C/T with an *initialization vector* (iv), use this as the *previous C/T block* for first encryption

  C_0 = iv = initialization vector

Initialisation Vector (iv)

- iv is at start of message/document (unencrypted)
- option 1 – fixed iv? (no - not random)

  (often messages have a predictable start)
- option 2 – use a counter for the iv? (no)
- option 3 – random iv – not just *random* but *different* for every message
- objective is that identical P/T and identical key produce different C/T

Block Cipher Modes - 3

- *Output Feedback* (OFB) mode
- block cipher used to generate a key stream (KS) which can be XOR'd with the P/T (cf. one-time pad) – 'stream cipher'

  KS_0 = iv (essential this is unique)

  $K_i = E(K, KS_{i-1})$

  $C_i = P_i$ XOR KS_i
- key stream can be precomputed
- P/T can be sent in smaller blocks (even bytes)

Message Authentication Code (MAC)

- not used to encipher data, rather to protect the integrity and authenticity of a message
- simplest system is to encrypt using CBC, then throw away everything except the final block – this is the MAC

  (usually iv = 0 when computing a MAC)
- recipient repeats calculation on message: if the MAC computed is identical to the one sent, then the message is confirmed and also the identity of the sender

Key distribution (continued)

- using symmetric encryption, if *n* parties need to communicate each pair of users needs their own separate key
- total of n(n-1)/2 keys: significant key management/distribution problem
- one potential solution is a *key distribution centre* – another is asymmetric encryption (*public key*)

Asymmetric Encryption

- each user has two keys, public and private
- public key widely distributed, private key *secret*
- can encrypt with public/decrypt private

  (*or encrypt with private/decrypt public*)
- somebody sends you a message encrypted with your public key: only *you* have the private key to decrypt
- or you encrypt something with your private key and send it: recipient decrypts with your public key – only *you* could have sent it
- (basis of authentication and non-repudiation)

Rivest-Shamir-Adelman (RSA)

- introduced in 1978, no flaws yet known
- based on the difficulty of factoring very large numbers (hundreds of digits)
- typically 10,000 times slower than symmetric enciphering
- N.B. key lengths for equivalent security of symmetric/asymmetric encryption cannot be directly compared! Asymmetric length needs to be about ten times (?) longer

RSA details…

- *key pair* (private + public key)
  (always created together)
- private key (d, N)
- public key (e, N)
- (N is a very large number – 100’s of digits)
- encrypt: C/T = ((P/T**e) mod N)
- decrypt: P/T = ((C/T**d) mod N)
  n.b. ((x**d)**e) = ((x**e)**d) = x**(e*d)
  (in practice, special software is used to perform exponentiation modulo N efficiently)

RSA implementation

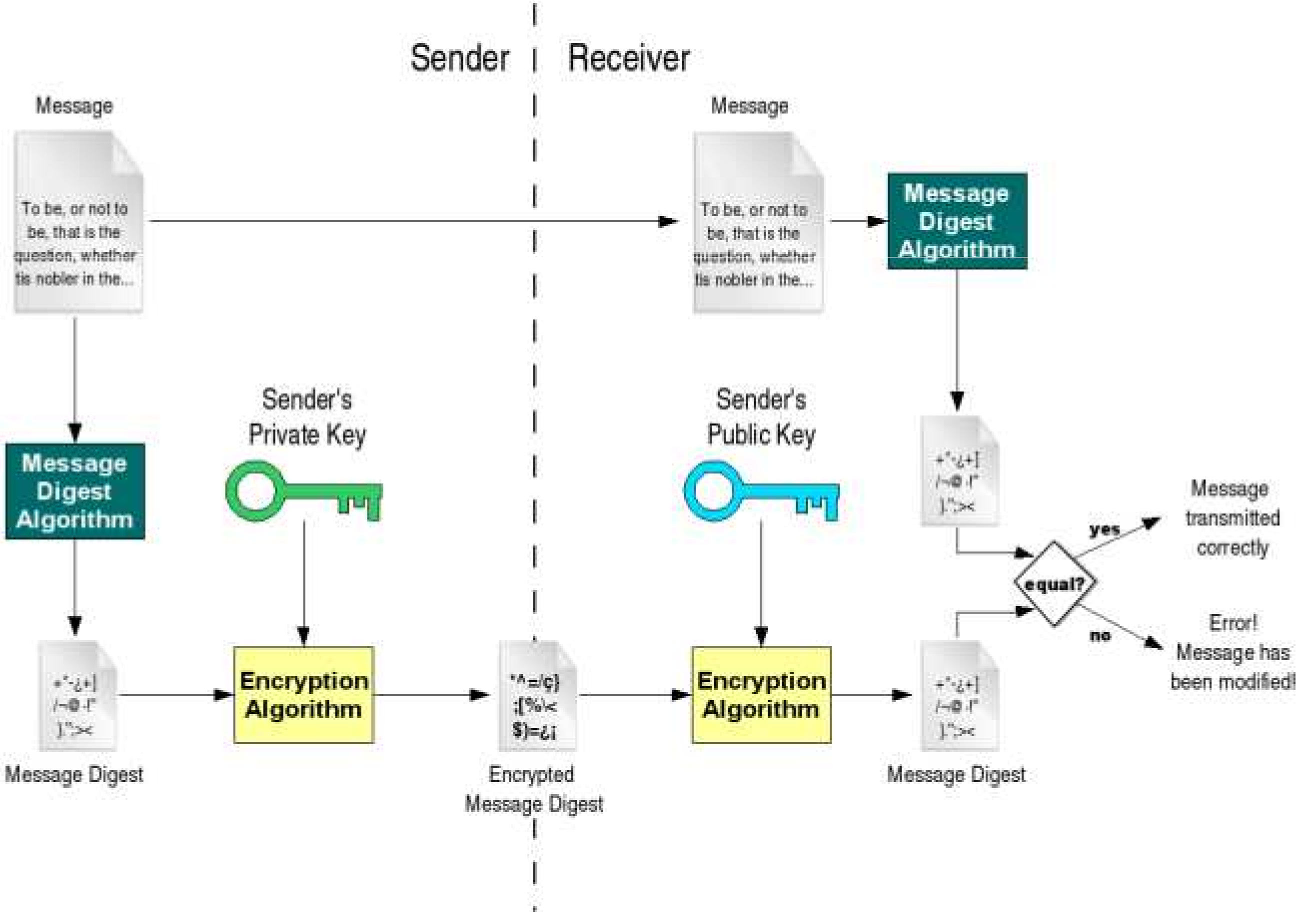
- first choose p, q - both large primes
  (larger than 10**100)
  (this effectively defines the key pair)
- n = p * q
- z = (p-1) * (q-1)
- e = any number relatively prime z
  (no common factor apart from 1)
- Select d such that (e*d) mod z = 1

RSA simple example (python)

```
>>> def en(x): return x**e%n    # encipher x function – public key (e,n)
>>> def dc(x): return x**d%n    # decipher x function – private key (d,n)

>>> p=137; q=131; n=p*q; z=(p-1)*(q-1); e=7; d=10103
>>> z%e, (e*d)%z      # test e and d meet requirements
(5, 1)  #z%e must be non-zero, (e*d)%z must be 1
>>> # print plaintext, ciphertext, deciphered C/T
>>> for i in range (11,20):  j=en(i); k=dc(j); print i,j,k
11 14676 11
12 9596 12
13 5805 13
14 10773 14
15 3935 15
16 2177 16     # etc.
```

Message authentication



Cryptographic Hash Functions

- also known as *message digests*
- generates a (small) *digest* or *fingerprint*
- *any* input change should produce a different digest
- one-way function: extremely difficult to go backwards (*"mashed potato"* analogy)
- must provide good *collision resistance*

  i.e. must be extremely difficult to find another message that produces the same digest, even though there are an infinite number of messages that all produce the same digest
- compresses information in arbitrarily-long message into (typically) 160 bits (SHA-1) 'Secure Hash Algorithm'

Cryptographic Hash Functions(2)

- basis of ‘digital signature (used for *non-repudiation*)

  e.g. electronic contract that has the same legal validity as a signed written contract (can be enforced by the courts)
- encrypt the message digest with your asymmetric private key and publish with the document: anybody can check the validity of the document by decrypting using your (published) public key and checking against the digest produced from the document itself: if they are identical, then the document is authentic.
- (also works with symmetric key, but then both sides need to possess the secret key)
- MD5 (128 bits) widely used in past, now deprecated

  (as is SHA-1, replaced by SHA-2)

SHA-1

- function operates on any length of P/T
- produces a 160-bit digest
- Developed by the NSA, published by NIST
  (FIPS 180-1 1995 and IETF RFC 3174)
  (internal block size 32 bits, 80 rounds)
- very hard to find another P/T that produces the same 160-bit hash
- (collision attacks should require 2**80 hashes, but 2005 research suggested that only 2**69 required, hence the replacement by SHA-2)

SHA-1 operation

- algorithm relies on ‘scrambling’ the input bits
- step 1: preprocess into 512-bit chunks,
  set the final digest H0 to H5 (32-bit words) to preset vales

- step 2: for each chunk, expand the 16 32-bit words into an 80-word array W (just re-use bits),
  set-up 5 32-bit words A to E as working storage
- step 3: for (i=0; i<80; i++)
- { temp = (A leftrotate 5) + fi(B,C,D) + E + Ki + W[i];
  E=D; D=C; C=(B leftrotate 30); B = A; A= temp}
  where “fi” is a different combination of the arguments and “ki” is a different constant: these change every 20 words
- step 4. H0 = H0+A; H1=H1+B; H2=H2+C; H3=H3+D; H4=H4+E
  (repeat steps 2,3 and 4 until all chunks processed)

SHA-1 example (python)

```
>>> from hashlib import *

>>> print sha1("The quick brown fox jumps over the lazy
    dog").hexdigest()

2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

>>> print sha1("the quick brown fox jumps over the lazy
    dog").hexdigest()

16312751ef9307c3fd1afbcb993cdc80464ba0f1
```

SHA-2

- SHA-2 recommended for US government use after 2010 (but still not widely used, no SHA-1 collisions have yet been discovered)
- FIPS PUB 180-4 (2001)
- six hash functions with digests that are 224, 256, 384 or 512 bits
- SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256

  (SHA-3 defined in 2012 – “just in case”)

Key exchange (1)

- suppose you are S and want to send a symmetric key K-s to R?
- S asymmetric keys: K-pub-s and K-priv-s
- R asymmetric keys: K-pub-r and K-priv-r
- send E(K-pub-r, E(K-priv-s, K-s)) to R
- R first decrypts with K-priv-r (only R can do this), then decrypts with K-pub-s (only S could have sent message)

Key exchange (2)

- Diffie-Hellman key exchange protocol
- first published public key algorithm (1976)
  (earlier GCHQ work kept secret)
- Alice and Bob wish to agree a shared secret key
  - first they agree two values:
  - modulus m, a large prime number [(m-1)/2 also prime]
  - integer g
- Alice generates a large random number “a”
  (kept secret)
- computes x = g**a mod m and sends to Bob

Diffie-Hellman key exchange

- Bob generates a large random number “b” (kept secret)
- computes y = g**b mod m and sends to Alice
- Alice computes K1 = y**a mod m
- Bob computes K2 = x**b mod m
- n.b. K1 = K2 = g**(a*b) mod m
- this is the **shared secret key**
- (secure because it is very hard to compute discrete logarithms modulo a large prime number)
- vulnerable to a ‘man in the middle’ attack

‘man in the middle’ attack

- (now sometimes ‘middleperson’ attacks)
- do **not** confuse with the ‘meet-in-the-middle’ attack!
- Alice ------ Trudy ------ Bob
- Alice *thinks* she is talking to Bob (actually talking to Trudy) and negotiates a key K1
- Bob *thinks* he is talking to Alice (actually talking to Trudy) and negotiates a key K2
- Trudy ends up knowing both K1 and K2 and can relay messages between Alice and Bob without them being aware if the interception

Authentication Protocols

- need to ensure that the other party in an exchange is who they claim to be
- challenge-response protocols widely used
- example: symmetric key
- Alice (A) & Bob (B) share a secret key Kab
- A sends E(Kab, Ra) to B, B replies with E(Kab, Ra+1)

  ("Ra+1" – any prearranged modification is acceptable)
- Ra is a *nonce* ("number used once") – a one-time random number generated by A just before it is sent (to avoid replay attacks)
- For this to be secure, Ra needs to be truly random (not pseudo-random) and cannot have been previously used (search against a stored list or incorporate a timestamp)
- Repeat B-A-B for B to have confidence in A

Authentication Protocols (2)

- example: asymmetric keys
- A sends E(K-pub-b, Ra)
- B replies with Ra
- only B can decrypt this message – so A has confidence in B
- repeat B-A-B for B to have confidence in A (if required)

Key Distribution Centre (KDC)

- If N parties need to communicate, need N**2 passwords or a KDC with only N passwords
- the KDC is **trusted**: each user has a single secret key shared with the KDC
- simple example: A generates Ks (a session key) and sends A, E(Ka, (B,Ks)) to the KDC
- The KDC sends a message to B: E(Kb,(A, Ks)
- This simple protocol is vunerable to replay attacks: counter with timestamps in the message or nonce
- much more complexity in practice

Digital signatures

- must be *unforgeable*
  (i.e. nobody else can sign)
- must be *authentic*
  (check signature correct)
- no alterations possible to document
- no re-use possible
- asymmetric encryption is one solution
  (see ‘key exchange (1)’ earlier)

Electronic Contracts

- A and B first agree the terms of a contract and agree on a hash function “H” – they both already have asymmetric key pairs
- They both calculate a message digest:
- MD = H(contract) - hence ‘MDa’ and ‘MDb’
- each now signs the digest with their own private key
- SMDa = E(K-priv-a, MDa) (+ ‘SMDb’)
- exchange these signed digests: each now signs the *other* signed digest with their own private key:
- DSMDa = E(K-priv-a, SMDb) (+ ‘DSMDb’)
- now append these two double-signed digests to the contract

Electronic Contracts (2)

- if a third party (e.g. a court of law) wishes to verify the contract they compute
  - TPMD1 = H(contract)
  - TPMD2 = D(K-pub-b, D(K-pub-a, DSMDa))
  - TPMD3 = D(K-pub-a, D(K-pub-b, DSMDb))

- If TPMD1 = TPMD2 = TPMD3
  then the contract is verified

X.509 Certificates

- who can you trust?
- a respected individual that you know?
- that individual might write you a ‘letter of introduction’ – this could allow anybody who knows that ‘respected individual’ to trust you?
- same idea to validate a public key: the key and the identity are bound together in a certificate, signed by the ‘respected individual’
- X.509 certificates are a widely-used mechanism

Certificate Authorities

- for a X.509 certificate, the ‘respected individual’ is replaced by a *certification authority* (CA)
- in return for a fee, the CA will verify your identity and issue you an X.509 certificate – this ties together (in the certificate) your identity with the public key matching your private key
- the hash of the certificate is signed by the CA (using the CA private key)

How can you trust the CA?

- ‘chain of trust’ possible (i.e. a certificate signed by a number of issuers)
- eventually there must be a root CA that you *have* to trust
- all browsers have ‘self-signed’ certificates issued by the major CA’s built into them
- potential for serious problems if one of these CA’s is compromised (e.g. Comodo)

Summary

- very short introduction to a complex area
- non-mathematical introduction: detailed discussion of cryptography algorithms would be highly mathematical
- the important techniques have been discussed in sufficient detail to explain their use in practice
- use a *published* implementation of a *published* strong encryption system!