Benjamin Fenster
CS-266 (Ling)
19 December 2005

Notes – Secret Key Cryptography

I. Introduction
  a. Take some fixed-size block (broken off the message, maybe 128 bits worth) and a fixed-size key, and turn it into another block.
  b. A longer key means more secure
  c. Generic Block Encryption
    i. One-to-one, so block sizes are the same (can reliably encrypt / decrypt)
    ii. Keys need to be long enough to discourage a known-plaintext attack (needs to be longer by about 2 bits every year)
    iii. Done with substitutions, permutations
      1. Substitute: Swap any value with any other
      2. Permute: Swap individual bits in the message
      3. Combine
    iv. Want the probability of possible outputs to be distributed randomly: if the opponent doesn't know the key there's no direct way to learn it.
    v. Bit Spreading: A 1 in position 23 may result in a 1 in any position of the output

II. DES: Data Encryption Standard
  a. Used 56-bit keys originally, *perhaps* because the government could break that level of encryption
  b. Keys are just eight ASCII characters
  c. Blocks are 64 bits
  d. Designed to be hard to attack with software, but easier to break with specialized hardware
  e. It's not a big deal if it's slow in software, since legitimate users only need to do one encryption / decryption (instead of 2^{56} of them)
  f. Algorithm
    i. Initial Permutation
      1. Swap the bits in a fixed (publicly known) way
      2. Adds nothing to security
      3. The point: it's easy to do in hardware but harder in software.
    ii. Break the message into two 32-bit pieces
    iii. To each piece, apply a key
    iv. Will do 16 rounds, applying a 48-bit key to each (generated from the original 56-bit key)
    v. Then perform a final permutation that's the inverse of the initial permutation
  g. How to Generate Keys
    i. Keys are originally 64 bits with 1/8 of those for parity
    ii. We just discard the parity bits for the encryption
    iii. Generate C_0, D_0 by permutations of the 56-bit key (each will be 28 bits)
    iv. Will rotate left by 1 bit for round 1, 2, 9, 16 and by 2 bits for all other rounds. That way we end up using all 28 bits of each key
    v. So we have 28 + 28 bits; we only need 24 + 24.
    vi. Take a permutation of C_i for the left half of K_i and a permutation of D_i for the right half of K_i.
    vii. Thus we've constructed a key for round i.
  h. Rounds
    i. Move the right half fo the left half ($L_{n+1} = R_n$)
    ii. $R_{n+1} = L_n \oplus m(R_n, K_n)$ where m() is the "mangler function" (more later)
    iii. $L_{n+1} = R_n$
  i. Decryption
    i. $R_{n+1} = L_n \oplus m(L_{n+1}, K_n)$
    ii. $R_{n+1} \oplus m(L_{n+1}, K_n) = L_n \oplus \underline{m(L_{n+1}, K_n) \oplus m(L_{n+1}, K_n)}$
    iii. The underlined part is all zeros, so $R_{n+1} \oplus m(L_{n+1}, K_n) = L_n$
    iv. $L_n = R_{n+1} \oplus m(L_{n+1}, K_n)$

v. $R_n = L_{n+1}$
vi. Decryption is the same as encryption, but swapped

j. Mangler Function
   i. Takes the 32-bit R_n, 48-bit key
   ii. Doesn't affect the ability to encrypt/decrypt. This function just adds security
   iii. Need to expand R_n to 48 bits by expanding each 4 bits into 6.
   iv. Each 4 bits borrows a bit from either side (shifting around the ends),
      1. Have, say, 0100 1110 1000 1101 … to start
      2. Becomes: 101001 011101 010001 011010
   v. Now we have the input and key the same length (each 48 bits)
   vi. Have 8 chunks of 6 bits in each
   vii. xor together for each chunk, then treat them separately
      1. R_n 101001 011101 010001 011010
      2. K_n 111111 111111 111111 111111
      3. 101001 ⊕ 111111 = 010110
   viii. Turn each six-bit chunk back into 4 using an S-box
      1. Take 010110, output 1100
      2. Basically a conversion "function" from 6-bit inputs into 4-bit outputs
   ix. That yields a 32-bit output (eight chunks of 4 bits)
   x. Then apply a permutation to the result, that's the final mangler function result

k. Weak Keys
   i. Some keys should be avoided (for C_0, D_0)
   ii. 0000…0000
   iii. 1111…1111
   iv. 0101…
   v. 1010…

III. IDEA: International Data Encryption Algorithm
   a. 64-bit blocks, 128-bit keys (more secure)
   b. All primitive operations map two 16-bit things into one 16-bit thing. In examples here we'll use four-bit values to make it easier to write.
   c. Three Operations
      i. a ⊕ b = c (xor)
         1. If you know any two, you can compute the third.
         2. b = c ⊕ a
         3. a = b ⊕ c
         4. 1100 ⊕ 0100 = 1001
         5. 0110 ⊕ 0110 = 0000
         6. With ⊕, any value is its own additive inverse
      ii. a + b = c
         1. But mod 2^{16} at the end (ignore the carrying bit)
         2. (a + b) mod 2^{16} = a + b
         3. If a + b = c, b = c – a (mod 2^{16})
         4. May get a negative difference, but will always have a positive remainder
         5. 10 + 1 = 11
         6. 9 + 8 = 1
      iii. a ⊗ b = a x b mod (2^{16} + 1)
         1. Re-encode using the values 1 to 2^{16} since 0 is boring in multiplication
         2. 9 ⊗ 12 = 6 (multiply 9 and 12, divide by 17, take the remainder)
         3. 7 ⊗ 14 = 13
         4. Zero is boring, so eliminate it, then encode 16 as 0 since it won't fit into the 4 bits available anyway.
         5. a ⊗5 = 11, a ⊗ 5 ⊗ 7 = 11 ⊗ 7, a = 9
         6. Given any number a, there's a unique inverse b such that a ⊗ b = 1
   d. Per-Round Keys
      i. 17 rounds, with nine of them odd and eight even

    ii. Need a total of 52 keys (generated from the 128-bit key)
    iii. Get eight keys just by breaking K into eight 16-bit pieces
    iv. Now shift K by 25 bits (rotating around as you do)
    v. Break that into 8 more 16-bit keys.
    vi. Shift by 25 again and repeat until all 52 keys are generated
    vii. There will be four extra keys on the last set; just ignore them
  e. Odd Rounds
    i. Take four per-round keys K_a, K_b, K_c, K_d
    ii. Split the 64-bit message into X_a, X_b, X_c, X_d
    iii. $X_a' = X_a \otimes K_a$
    iv. $X_b' = X_c + K_c$
    v. $X_c' = X_b + K_b$
    vi. $X_d' = X_d \otimes K_d$
    vii. Can decrypt; just use the multiplicative / additive inverses
  f. Even Rounds
    i. Take two per-round keys K_e, K_f
    ii. Compute $Y_{IN} = X_A \oplus X_B$, $Z_{IN} = X_C \oplus X_D$
    iii. Send Y_{IN}, Z_{IN}, K_e, K_f into the mangler function
    iv. $X_a' = X_a \oplus Y_{OUT}$
    v. $X_b' = X_b \oplus Y_{OUT}$
    vi. $X_c = X_c \oplus Z_{OUT}$
    vii. $X_d = X_d \oplus Z_{OUT}$
  g. Mangler Functions
    i. $Y_{OUT} = ((K_e \otimes Y_{IN}) + Z_{IN}) \otimes K_f$
    ii. $Z_{OUT} = (K_e \otimes Y_{IN}) + Y_{OUT}$

IV. AES
  a. History
    i. Key length of 56 bits in DES doesn't seem secure enough for current technology
    ii. Triple DES
      1. Given a 112-bit key (2 keys, still 56 bits each)
      2. Encrypt with K1, decrypt with K2, encrypt with K1
      3. If you use *double* DES there's a fairly easy attack against it. Details later
      4. Triple DES is much more secure, but is also much too slow.
    iii. IDEA
      1. Good, secure, efficient
      2. Patented! Nobody wants to pay royalties
      3. Want a royalty-free standard
    iv. AES
      1. NISI proposes something to replace DES
      2. Makes a public call on 12 September 1997 to create a publicly-designed cryptosystem
      3. People need to *believe* it's secure, so it should be done in public
      4. Want a block length of 128 bits
      5. Want key length to be variable: 128, 192, 256 bits
      6. Want world-wide availability without royalties
      7. 21 proposals submitted, 15 met the criteria 5 chosen as finalists
      8. MARS, RC6, Rijndael, Serpent, Twofish
      9. Rijndael became AES based on efficiencies, memory use, politics, et cetera
      10. All five finalists were secure
  b. Description
    i. Both block length and key length are variable (128, …, 256)
    ii. More general than the requirements demanded
    iii. The number of rounds N_R depends on the key length
    iv. Given plaintext X (128 bits), create a state (4 x 4 array), put 1 byte in each cell

v. All operations happen on this state. For example: Round Key ⊕ State (ADDROUNDKEY)
vi. For the first $N_R – 1$ rounds, do:
    1. SUBBYTES substitution
    2. SHIFTROWS permutation
    3. MIXCOLUMNS
    4. ADDROUNDKEY
vii. On the last round, don't do MIXCOLUMNS
viii. Then the ciphertext is just what's left in the state.

c. Algorithm
    i. Initialize state
    ii. SUBBYTES: Use an S-box (specifically chosen for security)
    iii. SHIFTROW: Shift the ith row left by i bytes (*not* bits!)
    iv. MIXCOLUMN
        1. Applied independently to each column
        2. Lookup column of four bytes for each element in the original column
        3. $result_1 = a_1 \oplus b_4 \oplus c_3 \oplus d_2$
        4. $result_2 = a_2 \oplus b_1 \oplus c_4 \oplus d_3$
        5. $result_3 = a_3 \oplus b_2 \oplus c_1 \oplus d_4$
        6. $result_4 = a_4 \oplus b_3 \oplus c_2 \oplus d_1$
        7. Now you have a new column

V. Modes of Operation
a. These algorithms only describe how to encrypt really short messages (we've been measuring in bits)
b. We need a way to break a large message up into pieces, encrypt the pieces, and put it back together to get the final ciphertext
c. Electronic Code Book (ECB)
    i. Break the message into 64-bit or 128-bit chunks (depending on which algorithm you're using)
    ii. Encrypt each chunk individually
    iii. Encryption: $c_i = E_k(m_i)$ for all i
    iv. Decryption: $m_i = D_k(c_i)$ for all i
    v. So a given block of ciphertext is obtained by just encrypting the corresponding block of the message
    vi. Very simple!
    vii. The Rub: It's easy to change the message
        1. Could swap two blocks and change the meaning, undetected by the recipient
        2. In salary data, for example, could swap two salaries to your own benefit, and you wouldn't have to know the key to do it.
    viii. Due to this problem, this scheme is rarely used.
    ix. The benefit: Changing one bit only affects one block of the message (each is independent)
d. Cipher Block Chaining
    i. Generate some $r_1 r_2 \ldots r_k$
    ii. Encryption: $c_i = E_k(m_i \oplus r_i)$
    iii. Decryption: $D_k(c_i) \oplus r_i = m_i$
    iv. So we need to have access to these random r_i's in both steps.
    v. This doubles the message length if r_i is generated randomly
    vi. Let's generate r_i from the message:
        1. $r_{i+1} = c_i$
        2. Now there's no need to send r_i AND nobody can rearrange the blocks
        3. Choose r_i= IV (some initialization vector). You'll still need to send that much, but that's tiny compared to a long message.
    vii. Set $c_0 = r_1$ for notation purposes.

viii. Encryption: $c_i = E_k(m_i \oplus c_{i-1})$
ix. Decryption: $m_i = D_k(c_i) \oplus c_{i-1}$
x. If one block is altered, the following block is affected too, but that effect does not propagate further since each block depends only on one other block.
xi. Since the IV is random, the encrypted message will be different each time. One cannot detect if the message has changed if it's sent twice.
xii. A Problem: Say you want to change m_7 from 5 to 7. You can't change c_7 without knowing the key. You CAN change c_6 by $\oplus$ing it with something (000000010) to turn what was 101_2 to 111_2. This would screw up m_6 in an unpredictable way though.

e. One-Time Pad
   i. Genreate r_i randomly
   ii. $c_i = m_i \oplus r_i$
   iii. Note that there's no encryption function here; it's just $\oplus$ing.
   iv. This is completely secure, though it's very hard to generate truly random numbers.
f. Output Feedback Mode
   i. Generate IV = b_0
   ii. Generate $b_1 = E_k(b_0)$ using a shared key
   iii. Then $c_i = m_i \oplus b_i$
   iv. Decryption:
      1. Receive c plus b_0
      2. Generate all b_i using the same encryption
      3. Then $m_i = c_i \oplus b_i$
      4. No decryption function is needed
      5. Since you don't need decryption, you could use a hash function.
   v. A problem: If a bad guy knows both m_i and c_i s/he can compute b_i easily without ever knowing the key (and then can create a new message)
   vi. A benefit: Changing one bit in c_i affects only one bit of m_i
g. Cipher Feedback Mode
   i. Want to generate b_i so we can do $c_i = m_i \oplus b_i$ again
   ii. $b_{i+1} = E_k(C_i)$, with b_1 random
   iii. $c_i = m_i \oplus b_i = m_i \oplus E_k(c_{i-1})$
   iv. $m_i = c_i \oplus E_k(c_{i-1})$
h. Why is this strategy secure?
   i. Assume b_i is generated randomly
   ii. We want to know the probability of m_i being some message given c_i. $P(m_i \mid c_i)$
   iii. Each b_i you choose leads to a different m_i
   iv. $P(m_i \mid c_i) = P(b_i) = (^1/_2)^{64}$
   v. As long as you don't know b_i, this is perfectly secure
   vi. We want $P(m_i = 000) = P(m_i = 0001) = \ldots = P(m_i = 1111)$

VI. Preserving Integrity
   a. Want to compute something like a checksum from a message such that it can't be altered without knowing the key
   b. Send only the last block of CBC (called CBC Residue)
   c. Security and Integrity
      i. Want to use CBC to encrypt and to generate a hash, since it requires a total of one pass
      ii. Compute hash: one pass
      iii. Then compute CBC (message | hash) and transmit it
      iv. Using a hash function is more efficient than encrypting

VII. Multiple Encryption DES
   a. Could be used for other cryptosystems
   b. Have two keys
   c. $c = E_{k1}(D_{k2}(E_{k1}(m)))$

d. Why two keys? If we used the same key you'd just end up encrypting once in the end because of $\oplus$.
e. Why not encrypt with the same key twice? Because finding the key by brute force is just as hard that way as if you'd only done it once.
f. Why not just encrypt with k_1, then with k_2
    i. Suppose the bad guy knows (m_1, c_2), (m_2, c_2), and (m_3, c_3)
    ii. For each possible key, encrypt m_1 and decrypt c_1
    iii. Find cases where the results are the same: those are potential matches.
    iv. There are 2^{48} possible (k_1, k_2) pairs that encrypt this way, and even though it's not really feasible to brute force that it's still less secure than single DES
    v. If there are two intermediate steps (by encrypting, decrypting, and encrypting again), this attack isn't possible
g. CBC
    i. Could treat Triple DES as a single algorithm and do CBC at the end
    ii. Could take the intermediate result and xor, but this means it's no longer protected from transmission errors the way it was before.