usyd2

The Data Encryption Standard

MATH2068 Number Theory & Cryptography
Week 7 Lecture 3
(NON-EXAMINABLE MATERIAL)

University of Sydney
NSW 2006
Australia

13th September 2017

Kerckhoff's Principle

The users of a good cipher system should not have to keep anything secret except the key they are using.

In other words, it shouldn't matter if the enemy know or discover the encryption method, so long as they don't discover the key.

Basically, the fewer things you have to keep secret, the better.

Hiding your encryption method from the enemy spies might be difficult, and if it is compromised then changing to a new encryption method is likely to be difficult and expensive.

So you should endeavour to design your system so that the encryption method does not have to be kept secret.

Origin of the DES

In 1972 the US Federal Department of Commerce called for a cryptographic standard for processing and distributing information on computer systems.

They wanted

- a high level of security;
- comprehensive and transparent specifications;
- security not to rely on secrecy of the algorithm;
- easy to implement, cheap, available to all . . . etc..

The system would be available to anyone who wanted to use it. Users would just have to choose their own secret keys.

Without such a publically available system, everyone who needed security would have to develop their own system. And probably lots of people would end up using bad systems.

Origin of the DES (continued)

No proposals were received. When the call was made again in 1974, IBM responded with a system they called "Lucifer".

In cooperation with IBM, the US National Security Agency modified the system, and the result was called the Data Encryption Standard.

The NSA's changes were simplifications, made so that the whole mechanism could be implemented on a single chip.

People with suspicious minds believed (possibly correctly!) that the NSA had built in some secret mechanism that would enable them to decrypt messages without access to the key.

But despite people searching very hard for this secret, noone has ever found it (or at least not disclosed it).

The keyspace used by DES is too small, given the power of modern computers. Exhaustive key searches are now feasible. But triple encryption with the DES algorithm (TDEA) is fine.

Keys

The DES uses 64 bit keys: a key is a sequence of 64 terms, each of which is either 0 or 1. (“Bit” means “binary digit”.)

But each 8th bit is a parity check: it must equal the sum of the previous 7 bits (mod 2). So there are only 56 independent bits.

The same key is used for both enciphering and deciphering.

After choosing a key the user creates 16 “subkeys”. First, the bits of the key are rearranged in the following order:

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

(i.e. the 57th term becomes the 1st term, the 49th becomes the 2nd, . . . , the 4th becomes the 56th.)

Note that the parity check bits (8, 16, 24, . . .) are not used.

First subkey

The rearranged key is split into two halves: C_0 (the first 28 bits) and D_0 (the other 28 bits).

C_0 & D_0 are left-shifted one place, with wrap-around, to produce C_1 and D_1, which are are tacked together to once more form a 56 bit sequence.

From this a 48 bit sequence K_1 is constructed, using the following table.

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

(i.e. 14th term of $C_1 D_1$ is 1st term of K_1 etc..)

Bits 9, 18, 22, 25, 35, 38, 43, 54 of $C_1 D_1$ are not used here.

The other subkeys

K_1 is our first subkey. The others are made similarly.

Split $C_1 D_1$ back into its two halves C_1 and D_1, left shift them both one place to produce C_2 and D_2, tack them together and from this create K_2 using the same table that we used to get K_1.

Split $C_2 D_2$ back into its two halves C_2 and D_2, left shift them both *two* places to produce C_3 and D_3, tack them together and from this create K_3 using the same table we used for K_1 & K_2.

The same process is repeated for K_4, K_5, . . . , K_{16}, obtaining C_{i+1} and D_{i+1} by left shifting C_i and D_i by one place if i is 0, 1, 8 or 15, or by two places for the other values of i.

A total of $4 \times 1 + 12 \times 2$ left shifts are applied altogether; so in fact $C_{16} D_{16} = C_0 D_0$ (since the C_i and D_i have length 28).

Subkeys: final comments

The rules for forming these subkeys may seem rather strange.

Note, however, that they are completely specific: one can say exactly which bits of the original 64 bit key make up the 48 bits of each of the 16 subkeys.

Stating it mathematically, there 16 one-to-one functions $\phi_i \colon \{1, 2, \ldots, 48\} \to \{1, 2, \ldots, 64\}$ such that if the original key is $a_1 a_2 \ldots a_{64}$ then the ith subkey is $a_{\phi_i(1)} a_{\phi_i(2)} \ldots a_{\phi_i(48)}$.

Also, there is negligible computation involved in creating the subkeys.

Enciphering

The plaintext needs to be in the form of a string of 0's and 1's. i.e. it should be a sequence of bits.

Enciphering is done 64 bits at a time.

Given 64 bits of plaintext as input, 64 bits of ciphertext are obtained as follows.

First, a certain transposition cipher is applied. That is, the 64 input bits are put in another order.

Next, 16 so-called *Feistel ciphers* are successively applied. (Feistel ciphers will be described later.) The keys for these 16 Feistel ciphers are the subkeys K_i.

A left shift of 32 places is then applied.

Finally, another transposition cipher is applied, this one in fact being the inverse of the one that was applied first.

The initial permutation and its inverse

The initial transposition cipher T rearranges the 64 input bits in the following order.

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

The inverse of this, applied last, is as follows.

40	8	48	16	56	24	64	32	39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30	37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28	35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26	33	1	41	9	49	17	57	25

Deciphering

It turns out that the deciphering algorithm is exactly the same as the enciphering algorithm, except that the Feistel ciphers are applied in the reverse order.

If we let $\mathscr{P}$ denote the plaintext and $\mathscr{C}$ the ciphertext, the enciphering algorithm is

$$\mathscr{C} = T^{-1}SF_{16}F_{15}\cdots F_3F_2F_1T(\mathscr{P})) \qquad (*)$$

where T is the initial transposition cipher, the F_i are the Feistel ciphers, and S is the left shift through 32 places.

Applying the deciphering algorithm to $\mathscr{C}$ thus produces

$$T^{-1}SF_1F_2\cdots F_{16}T\,T^{-1}SF_{16}\cdots F_2F_1T(\mathscr{P})$$

The $T\,T^{-1}$ cancels out, and because of the way Feistel ciphers are defined it turns out that $F_iSF_i = S$ for all i.

The above expression collapses down to $T^{-1}SST(\mathscr{P})$, which is just $\mathscr{P}$ since S is self inverse.

Feistel Ciphers

This is how the Feistel ciphers work. The 64 bits of text are split into a left half $\mathscr{L}$ and a right half $\mathscr{R}$, each consisting of 32 bits.

The cipher produces a new left half $\mathscr{L}'$ and a new right half $\mathscr{R}'$ by the formulas

$$\mathscr{L}' = \mathscr{R}$$
$$\mathscr{R}' = \mathscr{L} + f(K, \mathscr{R})$$

where K is the key and f is some function that returns a sequence of 32 bits.

The function f depends on the particular Feistel cipher, but could be anything.

Feistel Ciphers (continued)

Note that a left shift of 32 places simply swaps the left half of the text with the right half. This is self-inverse: it is the same as a right shift of 32 places.

So a Feistel cipher F followed by a 32 place shift S corresponds to the following formulas:

$$\mathscr{R}'' = \mathscr{R}$$
$$\mathscr{L}'' = \mathscr{L} + f(K, \mathscr{R})$$

That is, SF leaves $\mathscr{R}$ fixed and adds $f(K, \mathscr{R})$ onto $\mathscr{L}$.

So $SFSF$ leaves $\mathscr{R}$ fixed and adds $2f(K, \mathscr{R})$ onto $\mathscr{L}$.

Twice anything is zero mod 2; so $SFSF$ fixes $\mathscr{L}$ as well as $\mathscr{R}$.

So $SFSF$ is the identity. So FSF is the inverse of S, and this equals S.

The function f

I said that 16 Feistel ciphers are used, but that was a little lie. The same Feistel cipher F is used 16 times, but with 16 different keys.

To describe F fully we just have to describe the function f.

Recall that F is defined by

$$\mathscr{L}' = \mathscr{R}$$
$$\mathscr{R}' = \mathscr{L} + f(K, \mathscr{R})$$

The function takes the 48 bit key K and the 32 bit string $\mathscr{R}$ as input and returns a 32 bit string as output.

Note that $\mathscr{L} + f(K, \mathscr{R})$ means bitwise addition. That is, add the first bit of $\mathscr{L}$ to the first bit of $f(K, \mathscr{R})$, add the 2nd bit to the 2nd bit, etc.. (This is addition mod 2: “xor” in computer science jargon.)

The function f: details

A 48 bit sequence is made from the 32 bits of $\mathscr{R}$, by re-using some of the bits, as follows.

32	1	2	3	4	5	4	5	6	7	8	9	8	9	10	11
12	13	12	13	14	15	16	17	16	17	18	19	20	21	20	21
22	23	24	25	24	25	26	27	28	29	28	29	30	31	32	1

(Bits 4, 5, 8, 9, 12, 13, 16, 17, 20, 21, 24, 25, 28, 29, 32 and 1 get used twice.)

This 48 bit sequence is added to K (row vector addition modulo 2, as above) to give another 48 bit sequence. This is split into 8 pieces of 6 bits each.

These eight 6-bit sequnces are used to make eight 4-bit sequences, by means of eight functions described below.

These eight 4-bit sequences are concatenated to form the 32-bit output of f.

S-boxes

The eight functions from 6 bits to 4 bits are described in terms of eight so-called *S-boxes*, the first of which is as shown.

S_1	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1110	0100	1101	0001	0010	1111	1011	1000	0011	1010	0110	1100	0101	1001	0000	0111
01	0000	1111	0111	0100	1110	0010	1101	0001	1010	0110	1100	1011	1001	0101	0011	1000
10	0100	0001	1110	1000	1101	0110	0010	1011	1111	1100	1001	0111	0011	1010	0101	0000
11	1111	1100	1000	0010	0100	1001	0001	0111	0101	1011	0011	1110	1010	0000	0110	1101

Given a 6 bit sequence as input to the S-box function, the output is obtained as follows.

The first and last bit of the input are stuck together to make a 2 bit number, and this specifies a row of the S-box. The middle four bits specify the column.

Thus, for example, the input 011001 to S_1 will return the entry in row 01 and column 1100 of the table above, namely 1001.

The next three S-boxes

It is not very edifying, but for completeness' sake we list all the S-boxes. Here are S_2, S_3 and S_4.

S_2	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1111	0001	1000	1110	0110	1011	0011	0100	1001	0111	0010	1101	1100	0000	0101	1010
01	0011	1101	0100	0111	1111	0010	1000	1110	1100	0000	0001	1010	0110	1001	1011	0101
10	0000	1110	0111	1011	1010	0100	1101	0001	0101	1000	1100	0110	1001	0011	0010	1111
11	1101	1000	1010	0001	0011	1111	0100	0010	1011	0110	0111	1100	0000	0101	1110	1001

S_3	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1010	0000	0101	1110	0110	0011	1111	0101	0001	1101	1100	0111	1011	0100	0010	1000
01	1101	0111	0000	1001	0011	0100	0110	1010	0010	1000	0101	1110	1100	1011	1111	0001
10	1101	0110	1000	1001	1000	1111	0011	0000	1011	0001	0010	1100	0101	1010	1110	0111
11	0001	1010	1101	0000	0110	1001	1000	0111	0100	1111	1110	0011	1011	0101	0010	1100

S_4	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0111	1101	1110	0011	0000	0110	1001	1010	0001	0010	1000	0101	1011	1100	0100	1111
01	1101	1000	1011	0101	0110	1111	0000	0011	0100	0111	0010	1100	0001	1010	1110	1001
10	1010	0110	1001	0000	1100	1011	0111	1101	1111	0001	0011	1110	0101	0010	1000	0100
11	0011	1111	0000	0110	1010	0001	1101	1000	1001	0100	0101	1011	1100	0111	0010	1110

The rest

S_5	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0010	1100	0100	0001	0111	1010	1011	0110	1000	0101	0011	1111	1101	0000	1110	1001
01	1110	1011	0010	1100	0100	0111	1101	0001	0101	0000	1111	1010	0011	1001	1000	0110
10	0100	0010	0001	1011	1010	1101	0111	1000	1111	1001	1100	0101	0110	0011	0000	1110
11	1011	1000	1100	0111	0001	1110	0010	1101	0110	1111	0000	1001	1010	0100	0101	0011

S_6	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1100	0001	1010	1111	1001	0010	0110	1000	0000	1101	0011	0100	1110	0111	0101	1011
01	1010	1111	0100	0010	0111	1100	1001	0101	0110	0001	1101	1110	0000	1011	0011	1000
10	1001	1110	1111	0101	0010	1000	1100	0011	0111	0000	0100	1010	0001	1101	1011	0110
11	0100	0011	0010	1100	1001	0101	1111	1010	1011	1110	0001	0111	0110	0000	1000	1101

S_7	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	0100	1011	0010	1110	1111	0000	1000	1101	0011	1100	1001	0111	0101	1010	0110	0001
01	1101	0000	1011	0111	0100	1001	0001	1010	1110	0011	0101	1100	0010	1111	1000	0110
10	0001	0100	1011	1101	1100	0011	0111	1110	1010	1111	0110	1000	0000	0101	1001	0010
11	0110	1011	1101	1000	0001	0100	1010	0111	1001	0101	0000	1111	1110	0010	0011	1100

S_8	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	1101	0010	1000	0100	0110	1111	1011	0001	1010	1001	0011	1110	0101	0000	1100	0111
01	0001	1111	1101	1000	1010	0011	0111	0100	1100	0101	0110	1011	0000	1110	1001	0010
10	0111	1011	0100	0001	1001	1100	1110	0010	0000	0110	1010	1101	1111	0011	0101	1000
11	0010	0001	1110	0111	0100	1010	1000	1101	1111	1100	1001	0000	0011	0101	0110	1011

How does it differ from a classical cipher?

The DES is not at all intrinsically different from a classical cipher system.

It is a great big combination of transposition ciphers and polygraph substitution ciphers.

It's more complicated than the classical ciphers, that's all.