Computer Security MT14: Practical 2

Differential cryptanalysis of 2-round modified DES

In this practical you will perform a CPA against a reduced version of DES. You will be given a function which encrypts 64-bit plaintext blocks, and your aim is to determine the 56-bit encryption key in time considerably faster than exhaustive search. You will be using a simplified version of the *differential cryptanalysis* attack, which requires some delicate analysis.

Instructions

To complete this practical, you should hand in a completed and well-commented Java program `DESPrac.java`, plus a selection of its output. To qualify for grade `S`, you should make good progress on tasks 1–4. To qualify for an `S+` you should successfully recover the encryption key completing, or coming close to completing, tasks 1–5.

We will be working with **2-round modified DES**. Like DES, it uses a 56-bit key and works on 64-bit plaintext and ciphertext blocks. It uses just two rounds of a Feistel network, and the mangler function is almost identical to that of DES (it uses an identical expansion "E-box", performs an XOR with a 48-bit subkey, has eight identical "S-boxes", and an identical "P-box" permutation) but with the addition of a further XOR, of the bottom 32 bits of the round's subkey, at the end. A diagram of its structure is displayed in Figure 1, and you can find the details of the DES E-box, S-boxes[1], and P-box in §3.3 of the course textbook, or in many places online.

Because there are only two rounds, there are only two subkeys. The first, K_1, is the bottom (last) 48 bits of the full key K. The second, K_2 is the top (first) 48 bits of the key K.

This cipher differs from 'normal' DES in the following ways:

(1) It omits the initial permutation of the plaintext, and has no swap of left/right, or permutation, of the final output. This has no effect on its security.

(2) It has a simpler key schedule: each round of standard DES uses a selection of 48 non-consecutive bits from the 56-bit key. This has little effect on security but the modified cipher is easier to reason about.

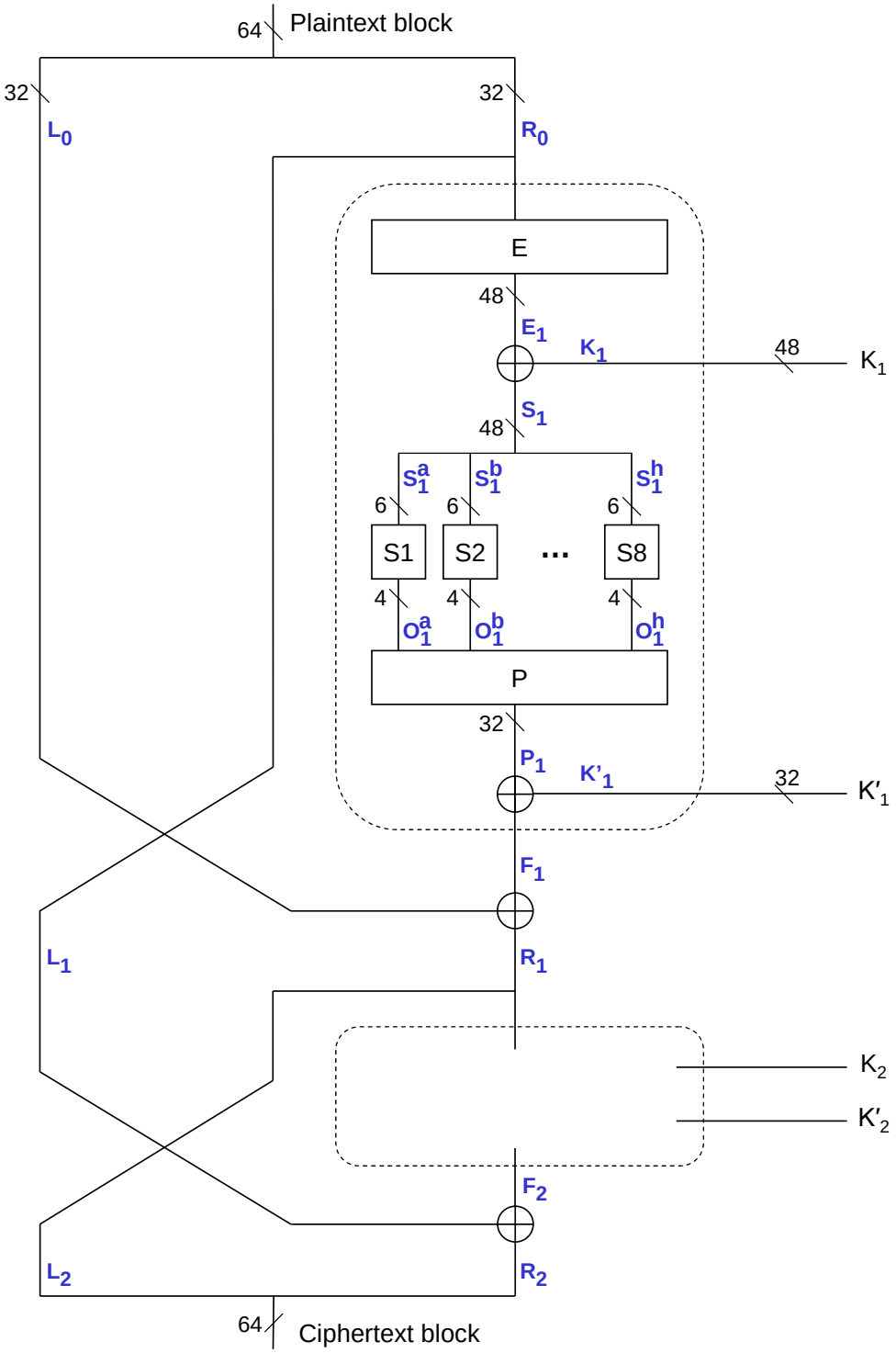
(3) It only has two rounds, rather than sixteen. This makes a big difference. Differential cryptanalysis is a lot more difficult to design when there are more rounds, though the idea is the same.

(4) The "modification" is that we added an extra function to end of the Feistel round: XOR with 32 bits of the subkey. This is simply to prevent a known plaintext attack which is essentially the one from problem sheet 1. It has no effect on the differential cryptanalysis.

You are given an incomplete Java program which has done almost all the work of implementing this cipher, including all the fiddly things like the E-box and P-box and all the S-boxes (which

[1]The S-boxes have been changed so that this year's practical is not identical to last year's.

Figure 1: Structure of 2-round modified DES. K_1 is the bottom 48 bits of the 56-bit key K, and K_2 the top 48 bits. K_1' is the bottom 32 bits of K_1, and K_2' the bottom 32 bits of K_2.



are just lookup tables). What is missing is the function which puts together one round of the Feistel network. You can find the incomplete program `DESPrac.java` in the directory `/usr/local/practicals/security/` on the practical lab machines.

General implementation tips

We will usually write numbers in hexadecimal form. In Java, a hexadecimal constant is prefixed by `0x`*, and we use the same convention in this document. To output a hexadecimal number, use* `Long.toHexString()` *or* `String.format("%#x",...)`*: the latter prints a leading* `0x` *as well.* `System.out.printf()` *is another alternative.*

Throughout the practical, we will use Java's `long` *primitive datatype, which stores a 64-bit integer. Annoyingly, all of Java's integer data types are signed integers in two's complement. This doesn't have much effect when it comes to storing message and cipher blocks in a* `long`*, but it does cause sign extension when we right shift (the 1st bit in the word is copied along). This is taken care of for you in the practical sample code and it should not affect you. It is, however, a pain when you want to store a value in* $\{0,\ldots,255\}$ *as a* `Byte` *and it might be easiest to avoid this by using a bigger datatype instead.*

Use the suffix `L` *for a* `long` *literal, thus* `long P=0x1234567887654321L` *is a 64-bit literal.*

Task 1

(i) Review the code supplied, and complete the implementation of 2-round modified DES.

(ii) Check your implementation by testing that $P =$ `0x1234567887654321` **encrypts, under key** $K =$ `0x33333333333333`**, to** `0xC844E31B90953751`**.**

(iii) Time how long it takes to perform 1000 encryptions, and estimate how long it would take (on average) to recover an unknown 56-bit key using the exhaustion attack. Give your answer in an appropriate time unit (not seconds!).

Tips for task 1

For (i), you only need to complete the function `Feistel`*. In the sample code, all 32-to-64-bit integer quantities are stored in* `long`*s. In case you are not familiar with Java's bitwise operators on primitive datatypes, the binary operator* `|` *performs bitwise OR,* `&` *bitwise AND, and* `^` *bitwise XOR. Binary right shift is performed using* `>>` *and left shift by* `<<`*. So* `1L<<32` *is the* `long` *constant* 2^{32}*.*

For (iii), measure "wallclock time", rather than CPU time. I suggest using `System.nanoTime()`*, which returns a* `long` *timestamp.* **Be careful not to overflow a 64-bit long** *in your calculations for the exhaustion attack time. Use a* `BigInteger`*, or floating-point arithmetic, if you want to be safe.*

The attack on DES is based on **differentials**, which simply measure how differences in input lead to differences in output. So imagine a function f which takes an input A to an output

$B = f(A)$. We can take two different inputs, A and A', and define their **differential**

$$\Delta A = A \oplus A'.$$

(Throughout this practical, ΔX means that there are two values for X, probably picked at random, and we are measuring their XOR difference). We can also write f's output differential

$$\Delta B = f(A) \oplus f(A').$$

In Figure 1 we have labelled some intermediate bit blocks in the 2-round modified DES cipher: L_0 and R_0 are the first 32-bit Feistel halves, E_1 the 48-bit output of the first E-box, and S_1 the result of XORing with the first subkey K_1. We have broken down S_1 into eight 6-bit blocks S_1^a to S_1^h which feed into S-boxes #1–8, resulting in 4-bit blocks O_1^a to O_1^h, which concatenate to O_1. The output of the first P-box is P_1 and the result of the first round's final XOR is F_1. This is the output of the first Feistel mangler, which XORs with L_0 to give R_1. And so on. We have used the same letters for variable names in the code. When we want to implement differentials, we use an underscore instead of a prime, thus `DeltaA = A^A_`.

The reason for studying differentials is that most parts of DES are **linear**. For example, the E-box satisfies

$$\Delta E_1 = \Delta E(R_0) = E(\Delta R_0)$$

and the P-box

$$\Delta P(O_1) = P(\Delta O_1).$$

For XORs it is even simpler, because $\Delta(A \oplus K) = (A \oplus K) \oplus (A' \oplus K) = A \oplus A' = \Delta A$. So no matter what the key K,

$$\Delta S_1 = \Delta E_1, \text{ and} \tag{1}$$
$$\Delta F_1 = \Delta P_1. \tag{2}$$

The nonlinear parts of DES are the all-important S-boxes. Let us take S-box #1: call the input S and output O (these are S_1^a and O_1^a on the diagram).

Table 1: Part of the differential distribution table for S-box #1.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
01	0	0	0	4	8	4	6	2	2	14	8	6	0	2	4	4
02	0	0	0	2	10	10	12	6	0	2	4	6	0	4	6	2
03	4	8	4	8	6	2	0	4	4	2	2	2	4	8	4	2
...																
0C	0	2	0	4	4	8	6	4	0	6	2	4	0	14	0	10
...																

Given a uniformly distributed input (over the 64 possibilities) S, the output O is also uniformly distributed (over the 16 possibilities). But given a uniformly distributed input S and another with a fixed difference $S' = S \oplus \Delta S$, the differential output

$$\Delta O = O \oplus O'$$

is **not uniform**. For example, if $\Delta S = 0$ then, because equal inputs must always give equal outputs, $\Delta O = 0$ always. And it turns out that, if $\Delta S =$ `0x01` (the 6-bit block representing 1 in binary), then ΔO can never be 0 and is must likely to equal `0x9` (the 4-bit block representing 9 in binary): it does so for 14 out of the possible 64 values of S. We can easily calculate this, by writing a program to go over all 64 possible values for ΔS and then, of the 64 possible values for S, work out the frequency of the 16 possible differential outputs ΔO. An excerpt from the output of such a program is displayed in Table 1: the possible differential inputs ΔS are on the left, in hexadecimal, and the possible differential outputs ΔO are along the top, also in hexadecimal. The numbers refer to the probability, out of 64, of observing the output differential given the input differential.

Task 2

(i) Explain why the output of every S-box is uniformly distributed, if the input is uniformly distributed. (This does not require any code, just a sentence of explanation).

(ii) Write the method to display the complete distribution of the output differentials of S-box #1, and include its output in your answer. (The method should work for any S-box, as you will need to look at all the tables for the final task.) If correct, the first few lines of your output will match the excerpt in Table 1.

The rest of this practical can only be performed on the lab machines; you will not be able to do this on your own computer. In `/usr/local/practicals/security/` is the executable file `desencrypt`[2]. Take a copy into your own directory. `encrypt` takes one argument, a single 64-bit integer as 16 digits of hexadecimal, which it encrypts under a key unique to you. It returns the corresponding ciphertext as a 16 digits of hexadecimal. Your aim is to work out what your unique key is by exploiting the differential nonuniformities you just discovered.

We will divide the 56-bit unknown key K into 9 subkeys, $K = K^x \,\|\, K^a \,\|\, \cdots \,\|\, K^h$, K^x is the top 8 bits of K, K^a the next 6 bits, K^b the next 6 bits, ..., K^h the last 6 bits. The reason that the first subkey is a different size is because of the key schedule of modified 2-round DES: K^a corresponds to the part of the key XORed to create S_1^a, the input to S-box #1 in the first Feistel round, K^b feeds into S-box #2, and so on. Your attack is going to reduce each subkey (except K^x, which is only used in the 2nd Feistel round) down to 2 or 4 possibilities, leaving you with only a few possible 56-bit keys to test.

[2]Credit for crypto code: includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit `http://www.openssl.org/`.

Let us take a random plaintext P and another P' with fixed difference

$$\Delta P = \texttt{0x 00 80 80 02 60 00 00 00}^3. \tag{3}$$

The reason for this particular difference will become apparent. We can trace the differentials right through the cipher:

$$\begin{aligned}
\Delta R_0 &= \texttt{0x 60 00 00 00}^3 \\
\text{applying the E-box, } \Delta E_1 &= \texttt{0x 0C 00 00 00 00 00 00 00}^4 \\
\text{using (1) and splitting, } \Delta S_1^a &= \texttt{0x0C}^4, \quad \Delta S_1^b = \ldots = \Delta S_1^h = \texttt{0x00}^4 \\
\text{from Table 1, } \Delta O_1^a &= \texttt{0xD} \text{ with probability } 14/64, \quad \Delta O_2^b = \cdots = \Delta O_2^h = \texttt{0x0} \\
\text{applying the P-box and (2), } \Delta F_1 &= \texttt{0x 00 80 80 02}^3 \text{ with probability } 14/64 \\
\text{using the Feistel XOR, } \Delta R_1 &= \Delta F_1 \oplus \Delta L_0 = \texttt{0x 00 00 00 00} \text{ with probability } 14/64 \\
\text{and } \Delta L_1 &= \Delta R_0 = \texttt{0x 60 00 00 00}
\end{aligned}$$

Now if the input differential to the second Feistel round is zero, the output must be zero too (equal inputs give equal outputs) so we can compute the output differential of the entire cipher. With probability at least 14/64,

$$\begin{aligned}
\Delta L_2 &= \Delta R_1 = \texttt{0x 00 00 00 00} \\
\Delta R_2 &= \Delta F_2 \oplus \Delta L_1 = \texttt{0x 60 00 00 00}
\end{aligned}$$

You may wonder why this helps. To answer, we will pick a random P, a P' with differential (3), and then compute the corresponding ciphertext differential. At least 14/64 of the time it will be as above, in which case **we know the differentials at every point along the cipher**. We call this **hitting the characteristic** (the characteristic is the input-output differential). If we don't hit, we try again with a another random P.

You may wonder why it helps to hit the characteristic. Well, recall that S-box #1 only gives the differential input-output `0x0C` to `0xD` for 14/64 of the possible inputs S_1^a. Call this set of possible 6-bit integers PI. For any random plaintexts P, P' which hit the characteristic, we know E_1 because we can compute it from R_0, and we know that

$$E_1^a \oplus K^a \in PI.$$

This means that K^a has to come from one of 14/64 possibilities. And we can repeat this process over and over, with different random plaintexts P, to reduce the possibilities for K^a still further. You will find you cannot reduce below 2 or 4 options, but this is a lot better than the 64 options when we had no information about K^a.

Task 3

Implement this procedure and reduce the possibilities for your K^a to 2 or 4 options.

[3] Each pair of hexadecimal characters representing 8 bits.

[4] Each pair of hexadecimal characters representing **6 bits**, to match the split into 6-bit chunks going into the S-boxes.

Tips for task 3

The supplied program `DESPrac.java` *already contains a method* `callEncrypt()` *which makes the relevant system call to execute the program* `desencrypt`*, supplied with a given plaintext block, and parses the resulting ciphertext block. It might throw an* `IOException` *if the* `desencrypt` *program cannot be found or executed.*

To generate random plaintexts, set up a `Random` *using* `Random prng=new Random();` *and then call* `long P=prng.nextLong();` *to generate pseudorandom* `long`*s. If you set* `long P_=P^DeltaP;` *then* `P` *and* `P_` *will be uniformly random with the specified difference. Keep generating random plaintexts until you hit the characteristic, then check which values of K^a made that possible.*

For reducing the possible values of K^a you could set up a `Hashset<Byte>`*, initialise it to hold all values $0, \ldots, 63$, and use* `remove()` *to remove impossible subkey values each time you hit the characteristic. When the* `Hashset`*'s* `size()` *reaches 2, or you have tried at least 100 random plaintext blocks, you can stop.*

What makes the attack work is that the input differential **isolated** S-box #1, in the sense that it guaranteed a) that the differential input to all other S-boxes was zero, and b) that a specified differential output happened fairly often (14/64 of the time). We can do the same to S-box #2:

Task 4

Explain briefly why $\langle \Delta L_0, \Delta R_0 \rangle = \langle$ 0x 40 00 40 10, 0x 02 00 00 00$\rangle$ is an input differential which isolates K^b using S-box #2 nonuniformity. Use the same procedure (hopefully parameterized so that you do not need to copy any code!) to whittle down K^b to 2 or 4 possibilities.

And now it is up to you to complete the attack.

Task 5

(i) Find differentials which isolate K^c,...,K^h. Reduce them all to a few options. It is more difficult to find differentials to isolate K^x, because it is not used in the first Feistel round, so it is best to leave it completely unknown.

(ii) You have reduced the set of all possible keys to 2 or 4 for each of K^a to K^h, and 256 for the 8-bit subkey K^x. You have about 20 bits of keyspace left to search through. Perform a KPA which exhausts through all the possible combinations, to determine your complete key K. Note: because K^x only affects two S-boxes in the second round, it will not be uniquely identified from a single plaintext-ciphertext combination (can you see why?). You will need to check at least a couple of different encryptions to determine K exactly.

Tips for task 5

How to find the differentials to isolate S-boxes 3 to 8: the key to making it work is $\Delta R_1 =$ `0x 00 00 00 00`*, which means that you want* $\Delta L_0 = \Delta F_1$*. And you want* ΔR_0 *to expand to something* ΔE_1 *which has zero bits in all but one S-box. Good luck!*

I don't think Java provides an easy `Iterable` *cartesian product, but if you stored the possibilities for each subkey as* `Iterables` *then you can do a smart recursion or just search through all options with a rather ugly nested loop:*

```
long P1=0x01L;               // get plaintext-ciphertext pair for the KPA
long C1=callEncrypt(P1);
long P2=0xFFEEDDCCBBAA9988L; // and another
long C2=callEncrypt(P2);
bigloop:
for(Byte Ka: possibleKa)
 for(Byte Kb: possibleKb)
  for(Byte Kc: possibleKc)
   for(Byte Kd: possibleKd)
    for(Byte Ke: possibleKe)
     for(Byte Kf: possibleKf)
      for(Byte Kg: possibleKg)
       for(Byte Kh: possibleKh)
        for(Integer Kx: possibleKx)
        {
          // get the value from each Byte or Integer as a primitive long
          // then stick the bits together:
          possibleK=Ka.longValue()<<42 | Kb.longValue()<<36 | Kc.longValue()<<30
                  | Kd.longValue()<<24 | Ke.longValue()<<18 | Kf.longValue()<<12
                  | Kg.longValue()<<6 | Kh.longValue() | Kx.longValue()<<48;

          if(TwoRoundModifiedDES(possibleK, P1)==C1)
          {
            if(TwoRoundModifiedDES(possibleK, P2)==C2)
            {
              // Well done! Print the output.
              break bigloop;
            }
          }
        }
```

Differential cryptanalysis was a breakthrough technique in the late 1980s and early 1990s, but full 16-round DES is surprisingly (and not coincidentally) resistant to it. If you are interested in how a full attack would be constructed you can read one of Biham and Shamir's original papers, available from `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.2000`.

`adk@cs.ox.ac.uk`, MT 2014