Encryption Key Search

Chapter 5

Breaking the Cipher

Encryption: To conceal passwords, credit card numbers, and other sensitive information from prying eyes while e-mail messages and Web pages traverse the public Internet, the information is encrypted.

A block cipher: Nowadays encryption is done using a block cipher, such as the U.S. Government's Advanced Encryption Standard (AES) (Figure 5.1).

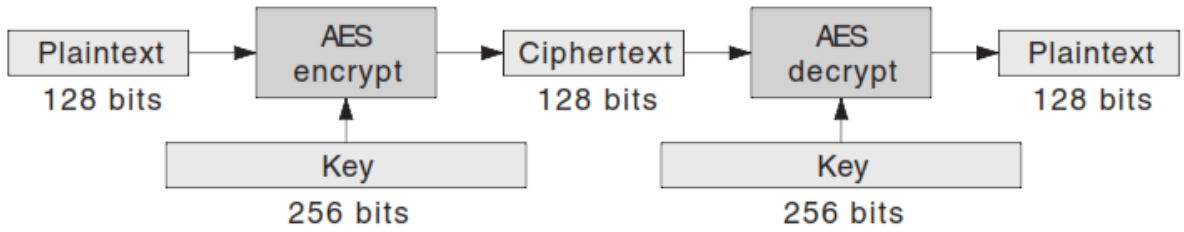

Figure 5.1 Encryption and decryption using AES

Here how Alice send an encrypted message Bob

Feeding the blocks: Alice feeds one block of the original message, called the plaintext, into the AES encryption function. For AES, the plaintext is a block of 128 bits (16bytes).

The ciphertext (Encrypted Blocks): The encryption function's output is another block of 128 bits, called the ciphertext. AES converts a block of plaintext into an equal-sized block of ciphertext.

The ciphertext: The ciphertext is random-seeming gibberish that reveals nothing about the plaintext.

Safe to send the ciphertext: Alice can safely send the ciphertext over a public network without fear that Eve, who is eavesdropping on the network traffic, will discover the sensitive information in the plaintext.

Decryption: Upon receiving the ciphertext, Bob feeds the ciphertext into the AES decryption function, which converts the ciphertext back into the original plaintext. The same key must be used for decryption.

Encryption Key: Both the encryption function and the decryption function use the same key. For AES, the key is a 256-bit value.

Encrypting with a different key: The same plaintext, encrypted with a different key, will yield a different ciphertext.

Only Alice and Bob must know the key: Because knowledge of the key would let Eve decrypt ciphertext messages, the key must be a secret key known only to Alice and Bob.

Finding the Key: One way that Eve could breach the secure communication is to find the key Alice and Bob are using.

Known plaintext attack

One way that Eve could find the key is a known plaintext attack. Eve somehow manages to obtain **both a plaintext block p and the ciphertext block c** that is the result of encrypting p with some secret key k.

Eve then uses her knowledge of these corresponding p and c values to deduce the value of k.

Exhaustive search: One way to find k is an exhaustive search. Eve starts with k = 0, feeds p and k into the encryption function, and checks whether the ciphertext that comes out is equal to c.

If so, Eve has found the correct value for k.

Otherwise, Eve repeats the process with k = 1, k = 2, and so on until she is successful.

Key sizes are large: Block cipher key sizes are chosen to make exhaustive key searches impractical. To find an AES key this way, Eve has to perform on the order of 2^{256} , or 10^{77}, encryptions.

Long before Eve has found the key, the universe will have come to an end.

If part of the key is known: However, suppose Eve knows some of the key. Perhaps Alice and Bob were careless and revealed the values of 232 bits of the 256-bit key.

Then Eve only has to do 2^{24}, or 16 million, encryptions to find the complete key. That's doable.

AES partial key search

This, then, is our first problem: Write a program for an AES partial key search.

The program's inputs are:

- a plaintext block p,
- a ciphertext block c,
- a portion of the key k that was used to produce c from p.

The values of the known key bits are given, along with the number of missing bits.

The program's output is the complete key. The program does an exhaustive search over all possible values for the missing key bits.

Generating random Key

256-bit key: we need a random 256-bit key.

java.util.Random Not Useful: However, it's a bad idea to use a pseudorandom number generator (PRNG) such as class java.util.Random to generate the key.

48 bits of internal state: The problem is that class java.util.Random has only 48 bits of internal state, from which it generates random values.

Entropy source: we should use an entropy source to generate a random key. Most Unix and Linux kernels have a special device file, /dev/random, that provides an entropy source.

The kernel accumulates "randomness," or entropy, into this file from the random times at which certain events occur, such as keystrokes, mouse movements, disk block accesses, and network packet receptions.

Then, as a program reads this file, the kernel uses the accumulated entropy to return truly random bytes.

In contrast, a PRNG generates only pseudo-random values using a deterministic formula.

Random Key Generation Program

Here is a program that prints a random 256-bit key in hexadecimal. To access the platform dependent entropy source in a portable manner, we use the getSeed() method of class java.security.SecureRandom.

```
import edu.rit.util.Hex;

import java.security.SecureRandom;

public class MakeKey {

    public static void main(String[] args) throws Exception {
        //32x8 = 256bit random number
        System.out.println(Hex.toString(SecureRandom.getSeed(32)));
    }
}
```

Encrypting A Block of a mesasge

Plaintext-ciphertext pair: we need a plaintext-ciphertext pair to use for our known plaintext attack.

Below is a program that:

- encrypts a message and keep it for testing.
- deletes the last n bits of the key for testing.

The program takes three inputs:

- a message string to encrypt,
- the encryption key (generated by the MakeKey program),
- n, the number of key bits for which to search.

Encryption Library

To do the encryption, the program uses an instance of class AES256Cipher from the Parallel Java Library.

```
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.util.Hex;

public class Encrypt2 {

// Prevent construction.
    private Encrypt2() {}
```

```
    private static final int[] mask = new int[]{0xff, 0xfe, 0xfc, 0xf8,
0xf0, 0xe0, 0xc0, 0x80};

    public static void main(String[] args) throws Exception {
        // Parse command line arguments.
        //if (args.length != 3) {
        //    usage();
        // }

        //String message = args[0];
        String message = "testing message";

        //String keyStr = args[1];
        String keyStr =
"b661ca5d5df7e4e66944751923247a91c1632bf1dc5821a5cd8d83fd4d8d439f";
        byte[] key = Hex.toByteArray(keyStr);

        // number of key bits to search for
        //int n = Integer.parseInt (args[2]);
        int n = 20;

        // Set up plaintext block.
        byte[] messageBytes = message.getBytes();
        System.out.println("mesasge length: "+messageBytes.length+"
bytes");

        byte[] encryptedMessage = new byte[16];
        System.arraycopy(messageBytes, 0, encryptedMessage, 0,
Math.min(messageBytes.length, 16));
        System.out.println("message to be encrypted: " +
Hex.toString(encryptedMessage));

        // Encrypt2 plaintext.
        AES256Cipher cipher = new AES256Cipher(key);
        cipher.encrypt(encryptedMessage);
        System.out.println("encrypted message:       "+
Hex.toString(encryptedMessage));

        // Wipe out n least significant bits of the key.
        int off = 31;
        int len = n;
        while (len >= 8) {
            key[off] = (byte) 0;
            --off;
            len -= 8;
        }
        key[off] &= mask[len];
        System.out.println("original key:             "+keyStr);
        System.out.println("last "+n+" bits deleted key:
"+Hex.toString(key));
    }


// Hidden operations.
    /**
     * Print a usage message and exit.
     */
    private static void usage() {
        System.err.println("Usage: java edu.rit.smp.keysearch.Encrypt
<message> <key> <n>");
```

```
        System.err.println("<message> = Message string to encrypt");
        System.err.println("<key> = Encryption key (256-bit hexadecimal
number)");
        System.err.println("<n> = Number of key bits to search for");
        System.exit(0);
    }
}
```

Sequentail Key Search

Below is the FindKeySeq program, which takes the Encrypt program's outputs as command-line arguments—

- The plaintext,
- the ciphertext,
- the partial key,
- n, the number of key bits for which to search.

The FindKeySeq program performs an exhaustive search over all the missing key bits and prints the complete key.

static variables

We follow the convention that variables used throughout the main program are declared as static fields of the main program class, rather than local variables of the main() method.

The reason for this will become clear when we write the parallel program.

Limitations: only last 32 bits

This program only works if less than last 32 bits of the key is unknown. It can not discover the original key, if more than 32 bits of the key is unknown or other than the last parts of the key is unknown.

Last Four Bytes of the Partial Key is Assigned to an int variable

0xFF = 11111111 = 255

<<24 means left shifting 24 bits. That byte is shifted 24 bits. That byte become the first byte of the int variable. Most significant byte.

Similarly other bytes are shifted 16, 8 or 0 bits.

```
keyLast4Bytes =
        ((partialkey[28] & 0xFF) << 24)
        | ((partialkey[29] & 0xFF) << 16)
        | ((partialkey[30] & 0xFF) << 8)
        | ((partialkey[31] & 0xFF));
```

Total number of keys to test

İf there is only one unknown bit: There are only two keys to test. That bit can be either zero or one.

İf there are only two unknown bits: There are only four keys to test. Each bit can be either zero or one.

İf there are n unknown bits. There are 2^n different keys to test.

maxcounter has this value: maxcounter = (1 << n);

this is a faster way of calculating 2^n

Constructing the keys for every combination for the last n bits

We can do this by using a counter.

By incrementing a counter from 0 to maxcounter, we can construct every possible key.

Since n may be diffenet than 32, we need to do a bitwise or with keyLast4Bytes:

```
int last4Bytes = keyLast4Bytes | counter;
```

this is the last four bytes to test.

Why not just increment trialkey array byte values

We can do the same thing by incrementing the byte values of the last four bytes of the trialkey array.

In that case, programming gets much more complicated. Most probably It will be much slower too.

Matching method

match method compares two 16 byte arrays. It does not stop when it finds a non-matching byte. It goes all the way to 16 bytes every time.

This is because the array size is small and there is no need to complicate the method with an extra if and break.

```
    private static boolean match(byte[] a, byte[] b) {
        boolean matchsofar = true;
        int n = a.length;
        for (int i = 0; i < n; ++i) {
            matchsofar = matchsofar && a[i] == b[i];
        }
        return matchsofar;
    }
```

Alternative match method

```
    private static boolean match(byte[] a, byte[] b) {
        int n = a.length;
        for (int i = 0; i < n; ++i) {
            if(a[i] != b[i])
                return false;
        }
        return true;
    }
```

Trying all Possible Keys

This program does not stop when it finds the searched key. It goes all the way to try all possible keys.

It could have easily stopped when it found the key in the if section of the for loop with a break.

```
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.util.Hex;

public class FindKeySeq {

// Prevent construction.
    private FindKeySeq() {
    }
// Shared variables.
    // Command line arguments.
    static byte[] plaintext;
    static byte[] ciphertext;
```

```
    static byte[] partialkey;
    static int n;
    // Variables for doing trial encryptions.
    static int keyLast4Bytes;
    static int maxcounter;
    static byte[] foundkey;
    static byte[] trialkey;
    static byte[] trialCipherText;
    static AES256Cipher cipher;

// Main program.
    /**
     * AES partial key search main program.
     */
    public static void main(String[] args)
            throws Exception {
        //Comm.init (args);

        // Start timing.
        long t1 = System.currentTimeMillis();

        // Parse command line arguments.
//        if (args.length != 4) {
//            usage();
//        }
//        plaintext = Hex.toByteArray(args[0]);
        plaintext = Hex.toByteArray("44656e656d65206d6573616ac4b10000");
//        ciphertext = Hex.toByteArray(args[1]);
        ciphertext = Hex.toByteArray("4d168e6cd45dcc264a3330e63fa6a52a");
//        partialkey = Hex.toByteArray(args[2]);
        partialkey =
Hex.toByteArray("b661ca5d5df7e4e66944751923247a91c1632bf1dc5821a5cd8d83fd4d
800000");
//        n = Integer.parseInt(args[3]);
        n = 20;

        // Make sure n is not too small or too large.
        if (n < 0) {
            System.err.println("n = " + n + " is too small");
            System.exit(1);
        }
        if (n > 30) {
            System.err.println("n = " + n + " is too large");
            System.exit(1);
        }

        // Set up variables for doing trial encryptions.
        // transfer the value of last four bytes to an int variable:
keylsbs
        keyLast4Bytes =
                ((partialkey[28] & 0xFF) << 24)
                | ((partialkey[29] & 0xFF) << 16)
                | ((partialkey[30] & 0xFF) << 8)
                | ((partialkey[31] & 0xFF));

        // calculate the number of keys to test 2^n
        maxcounter = (1 << n);

        // construct a trial key. this will get a new value in every
iteration
        trialkey = new byte[32];
```

```
        System.arraycopy(partialkey, 0, trialkey, 0, 32);

        // trialCipherText will get the encrypted message in every
iteration
        trialCipherText = new byte[16];
        cipher = new AES256Cipher(trialkey);

        // Try every possible combination of low-order key bits.
        for (int counter = 0; counter < maxcounter; ++counter) {
            // Fill in low-order key bits.
            int last4Bytes = keyLast4Bytes | counter;
            trialkey[28] = (byte) (last4Bytes >>> 24);
            trialkey[29] = (byte) (last4Bytes >>> 16);
            trialkey[30] = (byte) (last4Bytes >>> 8);
            trialkey[31] = (byte) (last4Bytes);

            // Try the key.
            cipher.setKey(trialkey);
            cipher.encrypt(plaintext, trialCipherText);

            // If the result equals the ciphertext, we found the key.
            if (match(ciphertext, trialCipherText)) {
                foundkey = new byte[32];
                System.arraycopy(trialkey, 0, foundkey, 0, 32);
            }
        }

        // Stop timing.
        long t2 = System.currentTimeMillis();

        // Print the key we found.
        System.out.println(Hex.toString(foundkey));
        System.out.println((t2 - t1) + " msec");
    }

// Hidden operations.
    /**
     * Returns true if the two byte arrays match.
     */
    private static boolean match(byte[] a,
            byte[] b) {
        boolean matchsofar = true;
        int n = a.length;
        for (int i = 0; i < n; ++i) {
            matchsofar = matchsofar && a[i] == b[i];
        }
        return matchsofar;
    }

    /**
     * Print a usage message and exit.
     */
    private static void usage() {
        System.err.println("Usage: java edu.rit.smp.keysearch.FindKeySeq
<plaintext> <ciphertext> <partialkey> <n>");
        System.err.println("<plaintext> = Plaintext (128-bit hexadecimal
number)");
        System.err.println("<ciphertext> = Ciphertext (128-bit hexadecimal
number)");
        System.err.println("<partialkey> = Partial key (256-bit hexadecimal
number)");
```

```
            System.err.println("<n> = Number of key bits to search for");
            System.exit(1);
        }
}
```

Transitioning to a Parallel Program

An agenda parallel problem, where the agenda items are to try all possible values of the missing key bits:

"Try k = 0," "Try k = 1," . . . , "Try k = 2^n – 1."

Only one task has the desired result

All Tasks are independent

In this problem, the results of one task do not in any way affect the results of the other tasks; there are no sequential dependencies between tasks.

Putting it an other way, none of the tasks produces any result needed by any other task.

Sequential Looping

The FindKeySeq sequential program performs each task by doing a loop over all the missing key bits from 0 to 2^n – 1. loop iterations are performed one at a time, in order.

All tasks can be calculated in parallel

However, because there are no sequential dependencies between tasks, the loop iterations can be performed all at once, in parallel.

İf we had 2^n processors

if we had a parallel computer with 2^n processors, we could find the answer in the same amount of time as the sequential program would take to try just one key.

Massively (embarrassingly) parallel problems

A problem such as AES partial key search—one where we can do all the computations en masse, with no dependencies between the computations—is called a massively parallel problem.

It is also sometimes called an embarrassingly parallel problem.

Parallel Program with K threads

Suppose we are solving the AES partial key search program on a parallel computer with K processors. Then we will set up a parallel program with K threads.

Like the sequential program, each thread will execute a loop to do its computations. However, each thread's loop will go through only a subset of the total set of computations— $\mathbf{2^n/K}$ of them, to be exact.

For Example: With 10 Threads

Suppose n is 20; then there are 1,048,576 keys to search.

Suppose K is 10; then six of the threads will do 104,858 iterations each and the other four threads will do 104,857 iterations each.

The range of counter values in each thread will be as follows:

Thread	*Lower Bound*	*Upper Bound*	*Thread*	*Lower Bound*	*Upper Bound*
0	0	104857	5	524290	629147
1	104858	209715	6	629148	734004
2	209716	314573	7	734005	838861
3	314574	419431	8	838862	943718
4	419432	524289	9	943719	1048575