CHAPTER 5

Massively Parallel Problems

in which we encounter a cryptographic problem requiring an enormous amount of computation; we build a sequential program to solve the problem; we reflect on how a parallel program could solve the problem; and we learn why such problems are called massively parallel

CHAPTER 5 Massively Parallel Problems

5.1 Breaking the Cipher

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

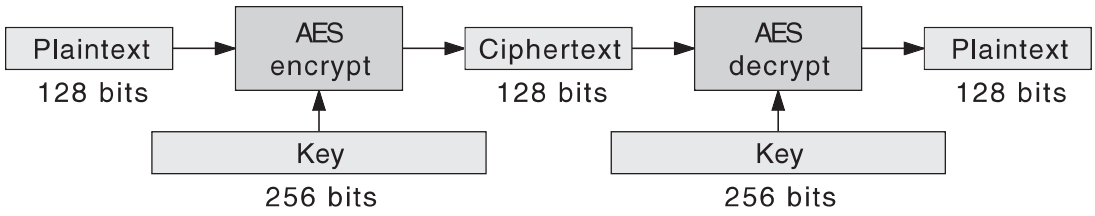

Figure 5.1 Encryption and decryption using AES

Here's how one person named Alice uses a block cipher to send an encrypted message to another person named Bob. (In cryptographic lore, the two parties in a secure communication are always named Alice and Bob.) Alice feeds the original message, called the **plaintext**, into the AES **encryption function**. For AES, the plaintext is a block of 128 bits. The encryption function's output is another block of 128 bits, called the **ciphertext**. (AES is called a "block" cipher because it converts a *block* of plaintext into an equal-sized *block* of ciphertext.) The ciphertext is random-seeming gibberish that reveals nothing about the plaintext. 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. Upon receiving the ciphertext, Bob feeds the ciphertext into the AES **decryption function**, which converts the ciphertext back into the original plaintext.

The security of block cipher encryption rests in the **key**, which is an input parameter to the encryption function and the decryption function. For AES, the key is a 256-bit value. The same plaintext, encrypted with a different key, will yield a different ciphertext. To recover the original plaintext from the ciphertext, the decryption function must use the same key as the encryption function. Because knowledge of the key would let Eve decrypt ciphertext messages, the key must be a **secret key** known only to Alice and Bob.

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

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.

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.

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, and 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.

5.2 Preparing the Input

Let's do our AES partial key search on a realistic example. First, we need a random 256-bit key. However, it's a bad idea to use a **pseudorandom number generator (PRNG)** such as class java.util.Random to generate the key. The problem is that class java.util.Random has only 48 bits of internal state, from which it generates random values. If the secret key came from class java.util.Random, Eve would have to search only the 2^{48} possible internal state values to find the key, not the 2^{256} possible key values.

Rather than use a PRNG, 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.

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.

```
package edu.rit.smp.keysearch;
import edu.rit.util.Hex;
import java.security.SecureRandom;
public class MakeKey
   {
   public static void main
      (String[] args)
      throws Exception
      {
      System.out.println (Hex.toString (SecureRandom.getSeed (32)));
      }
   }
```

And here is an example of what the MakeKey program prints.

```
$ java edu.rit.smp.keysearch.MakeKey
26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf44719520b
```

Next, we need a plaintext-ciphertext pair to use for our known plaintext attack. Here is a program that creates just such a pair. The program takes three command-line arguments: a message string to encrypt, the encryption key (generated by the MakeKey program), and *n*, the number of key bits for which to search. The program prints the plaintext block (a 128-bit hexadecimal number), the ciphertext block (a 128-bit hexadecimal number), the partial key with the *n* least-significant bits set to 0 (a 256-bit hexadecimal number), and *n*, the number of key bits for which to search. To do the encryption, the program uses an instance of class AES256Cipher from the Parallel Java Library.

```
package edu.rit.smp.keysearch;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.util.Hex;
public class Encrypt
   {
   public static void main
      (String[] args)
      throws Exception
      {
      // Parse command line arguments.
      if (args.length != 3) usage();
      String message = args[0];
      byte[] key = Hex.toByteArray (args[1]);
      int n = Integer.parseInt (args[2]);

      // Set up plaintext block.
      byte[] msg = message.getBytes();
      byte[] block = new byte [16];
      System.arraycopy
         (msg, 0, block, 0, Math.min (msg.length, 16));
      System.out.println (Hex.toString (block));

      // Encrypt plaintext.
      AES256Cipher cipher = new AES256Cipher (key);
      cipher.encrypt (block);
      System.out.println (Hex.toString (block));

      // 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 (Hex.toString (key));
         System.out.println (n);
         }

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

And here is an example of what the Encrypt program prints. (The Java command stretches across two lines.) The Encrypt program's output will become the actual key-searching program's input.

```
$ java edu.rit.smp.keysearch.Encrypt "Hello, world!" \
  26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf44719520b 20
48656c6c6f2c20776f726c6421000000
af3afe16ce815ad209f34b009da37e58
26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf447100000
20
```

5.3 Sequential Key Search Program

Here is the FindKeySeq program, which takes the Encrypt program's outputs as command-line arguments—the plaintext, the ciphertext, the partial key, and 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. It is a sequential program (no parallelism); later, we will modify it to make a parallel program.

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.

```
package edu.rit.smp.keysearch;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.util.Hex;
public class FindKeySeq
   {
   // Command line arguments.
   static byte[] plaintext;
   static byte[] ciphertext;
```

```
static byte[] partialkey;
static int n;

// Variables for doing trial encryptions.
static int keylsbs;
static int maxcounter;
static byte[] foundkey;
static byte[] trialkey;
static byte[] trialciphertext;
static AES256Cipher cipher;

/**
 * AES partial key search main program.
 */
public static void main
   (String[] args)
   throws Exception
   {
   // Start timing.
   long t1 = System.currentTimeMillis();

   // Parse command line arguments.
   if (args.length != 4) usage();
   plaintext = Hex.toByteArray (args[0]);
   ciphertext = Hex.toByteArray (args[1]);
   partialkey = Hex.toByteArray (args[2]);
   n = Integer.parseInt (args[3]);

   // 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);
      }
```

The variable `keylsbs` holds the least-significant 32 bits of the partial key from the command line; this value has *n* low-order zero bits.

```
// Set up variables for doing trial encryptions.
keylsbs =
   ((partialkey[28] & 0xFF) << 24) |
   ((partialkey[29] & 0xFF) << 16) |
   ((partialkey[30] & 0xFF) <<  8) |
   ((partialkey[31] & 0xFF)      );
```

To search over the missing key bits, we run an integer counter from 0 to $2^n - 1$. The variable `maxcounter` holds the upper bound for the counter. To be certain that this upper bound fits in a variable of type `int`, we ensure that n lies in the range 0 through 30. The variable `trialkey` is for holding the encryption key we are currently trying; this starts out the same as the partial key from the command line. The variable `trialciphertext` is for holding the output ciphertext from the current encryption. The cipher object performs the actual encryptions.

```
maxcounter = (1 << n) - 1;
trialkey = new byte [32];
System.arraycopy (partialkey, 0, trialkey, 0, 32);
trialciphertext = new byte [16];
cipher = new AES256Cipher (trialkey);
```

Now we can run the search loop, with the counter going from 0 to $2^n - 1$. To set up the trial encryption key, we combine the counter value with the least significant bits of the partial key (`keylsbs`) using bitwise-or (the | operator). The counter fills in the missing low-order bits of the partial key, thus creating the complete trial key. We use the cipher object to encrypt the plaintext with this trial key.

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

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

If the resulting ciphertext equals the input ciphertext, we have found the correct key, and we store a copy of it in the variable `foundkey`. At this point, we could exit the loop, because further trials are pointless. However, for now, we will stay in the loop; the program will always try all 2^n key values. (We will need this behavior later, when we study the parallel version's running time.)

```
            // 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");
         }

      /**
       * 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;
         }
      }
```

Here is an example of the FindKeySeq program's output. (The Java command stretches across five lines.) The inputs (command-line arguments) are what the Encrypt program generated—the plaintext

block, the ciphertext block, the partial key with the low-order bits missing, and the number of missing key bits. The FindKeySeq program prints the complete key that it found, as well as the running time.

```
$ java edu.rit.smp.keysearch.FindKeySeq \
  48656c6c6f2c20776f726c6421000000 \
  af3afe16ce815ad209f34b009da37e58 \
  26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf447100000 \
  20
26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf44719520b
2936 msec
```

5.4 Transitioning to a Parallel Program

The AES partial key search problem is 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 $k = 2^n - 1$." We are not interested in all the tasks' results, but only in the key for the one task that succeeded. 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.

The FindKeySeq program performs the agenda items by doing a loop over all the missing key bits from 0 to $2^n - 1$. In the sequential program, the loop iterations are performed one at a time, in order. However, because there are no sequential dependencies between tasks, the loop iterations do not have to be performed in sequence—as we saw with the introductory program in Chapter 4. The loop iterations can be performed all at once, in parallel. In fact, 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. For this reason, 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**; there's so much parallelism, it's embarrassing!

In Chapter 4, we saw how to execute a parallel program with each computation in a separate thread and each thread running on its own processor. If n is a small number, such as 2 or 4, we'd have no trouble finding a parallel computer with 2^n processors. But this approach breaks down if n is a more interesting number such as 24 or 28. What we need is an approach somewhere in the middle of the two extremes of doing all the computations in sequence in a single thread and doing each computation in parallel in its own thread.

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—$2^n/K$ of them, to be exact. The set of computations will be *partitioned* among the K threads; multiple tasks will be *clumped* together and executed by one of the K threads.

As a specific example, 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 per-thread iteration counts differ because the total number of iterations is not evenly divisible by *K*.) 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

The Parallel Java Library has classes that let you program **parallel loops**, where the total set of loop iterations is divided among a group of threads in the preceding manner. Chapter 6 will introduce these features of Parallel Java. In Chapter 7, we will use Parallel Java to convert the FindKeySmp program to a parallel program.

5.5 For Further Information

On cryptography, block ciphers, and ways of attacking them:

- Douglas R. Stinson. *Cryptography: Theory and Practice, 3rd Edition*. Chapman & Hall, 2005.
- Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley Publishing, 2003.
- Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World, 2nd Edition*. Prentice Hall PTR, 2002.
- Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
- Bruce Schneier. *Applied Cryptography, Second Edition*. John Wiley & Sons, 1996.

On the Advanced Encryption Standard in particular:

- *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. November 26, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf