Cryptography - RC4 Algorithm

Quentin Galvane Baptiste Uzel

February 18, 2012

Contents

Chapter 1

Introduction

The team "batumki" is made up of two exchange students from INSA (engineering school in France) : Baptiste Uzel and Quentin GALVANE.

After some research on the web to find an interesting cryptographic primitive to implement, we decided to implement RC4. We chose this stream cipher for several reasons. First of all, this cipher is one of the most widely used stream cipher. Moreover it is used by really important and famous protocols and standards such as SSL, TSL, WEP, etc. Another reason for this choice is that it is well known for its simplicity and efficiency and we wanted to see if we could really optimize the performance of this cipher after implementing a first basic version.

So after presenting the RC4 stream cipher and explaining the way it works, we will present our work on the subject. Since the algorithm is rather simple, our approach consisted in first trying different languages and compare the simplicity of the implementation and its efficiency.

After the comparison, we will present and explain the design of our soft and the details of the implementation on the two language we selected: Java and C. We will then analize the performance of these first implementations and explain the different improvement we came up with thanks to the previous analysis.

We will then give a developer's manual and a user's manual to ease the use of our software. And finally, we will conclude with what we learned from this project and what could still be done.

Chapter 2

Description of RC4

2.1 History

This stream cipher was invented in 1987 by Ron Rivest, one of the inventors of the RSA public key cryptography algorithm and co-founders of RSA security. Even though the RC4 cipher is officially named "Rivest Cipher 4", it is also known as "Ron's Code 4" (RC2, RC5 and RC6 also exist).

The trade secret behind RC4 was revealed in september 1994 when the description of the cipher was sent to the Cypherpunks mailing list (group of people interested in privacy and cryptography who used this mailing list to communicate). After that, the description was posted on many website and the genuineness of the information was confirmed as the resulting outputs of the described cipher were matching the outputs of licenced RC4.

RC4 had a really large success thanks to its simplicity and efficiency. It was used in many popular standards and protocols such as WEP, WPA, SSL or TLS. Unfortunately, the cipher has some weaknesses and is not used anymore in modern protocols.

2.2 Principle

The RC4 algorithm generates a pseudo-random keystream that is then used to generate the ciphertext (by XORing it with the plaintext). It is called pseudo-random because it generates a sequence of numbers that only approximates the properties of random numbers. The sequence of bytes generated is not random since the output is always the same for a given input but it has to approximate random properties to make it harder to crack. The keystream is generated from a variable length key using an internal state composed of the following elements:

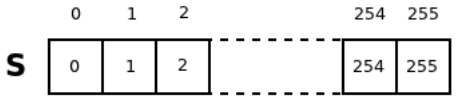
- A 256 bytes array (denoted S) containing a permutation of these 256 bytes
- Two indexes i and j, used to point elements in the S array (only 8 bits are necessary for each index since the array only have 256 elements)

Once the S array has been initialized and "shuffled" with the key-scheduling algorithm (KSA), it is used and modified in the pseudo-random generation algorithm (PRGA) to generate the keystream. A more detail description of these two algorithm is given in sections 2.3 and 2.4.

2.3 The key-scheduling algorithm

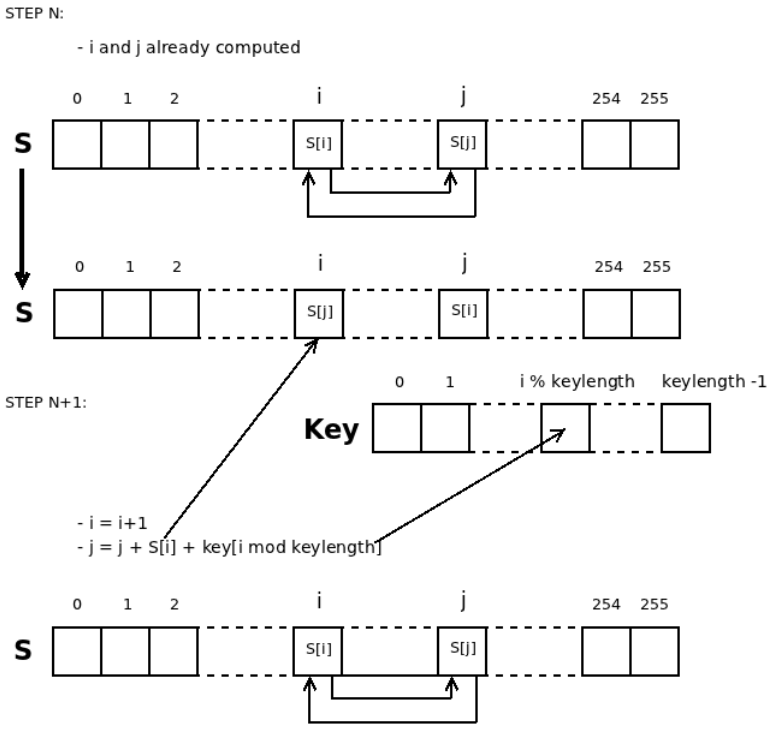
As explained before, the key-scheduling algorithm is used to generate the permutation array.

The first step of this algorithm consist in initializing the S table with the identity permutation: the values in the array are equal to their index.



Once the S array is initialized, the next step consist in shuffling the array using the key to make it a permutation array. To do so, we simply iterate 256 times the following actions after initializing i and j to 0:

- compute j = j + S[i] + key[i mod keylength]
- swap S[i] and S[j]
- increment i



Once i has reached 256 (the 256 iterations were completed), the S array has been properly initialized.

Here is some pseudo-code corresponding to the key-scheduling algorithm:

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

Now that the S array is generated, it is used in the next step of the RC4 algorithm to generate the keystream.

2.4 The pseudo-random generation algorithm

This step of the algorithm consists in generating a keystream of the size of the message to encrypt. This algorithm enables us to generate a keystream of any size.

To do so, we first initialize the two indexes to 0 and we then start the generation of the keystream one byte at a time until we reached the size of the message to encrypt. For each new byte to compute we do the following actions:

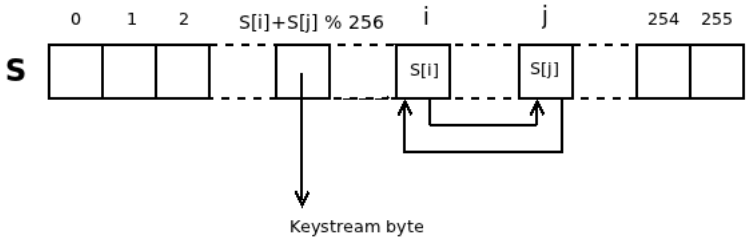
- Compute new value of i and j:

  i := (i + 1) % 256

  j := (j + S[i]) % 256

- Swap S[i] and S[j] to have a dynamic state (it makes it obviously harder to crack than if the state was computed only once and use for the generation of the whole keystream)

- Retrieve the next byte of the keystream from the S array at the index S[i]+S[j]% 256

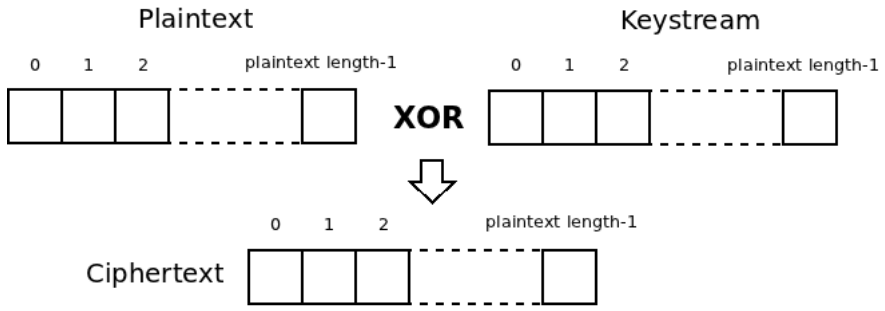


Here is some pseudo-code corresponding to the pseudo-random generation algorithm:

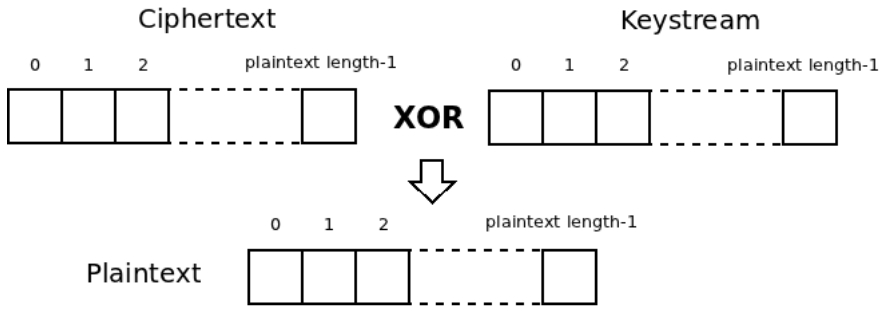
```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    K := S[(S[i] + S[j]) mod 256]
    output K
endwhile
```

2.5 Encryption and decryption

Once the keystream has been generated, the encryption of the plaintext is really simple: it simply consists of a XOR between the plaintext and the keystream. See below an illustration of the encryption:



As for the decryption, it is as simple as the encryption, we only have to do the opposite: XOR the ciphertext with the keystream.



Chapter 3

Implementation language

3.1 The different choices

Since implementing a first basic version of this algorithm appeared relatively simple, we first decided to implement it in several languages in order to see which one would be the fastest and to see if there would be a consequent difference in the performance. Here is the list of language we decided to try:

- Java
- C
- Perl
- Python
- Assembly (x86-64)

Obviously, the C and the Assembly implementation should give better results than the other three, but it still is interesting to find out how better the results are and thus if it is really more interesting to implement it in C/assembly or if in the end the results have the same order of magnitude.

3.2 The tests

To test this, we ran a small script that launched all the different programs (none of them were optimized yet) and printed the duration of the encryption. But we only want to mesure the time of encryption and nothing else. In order to do that the encryption process is repeated several times (this is a parametrizable benchmark option of all of our programs), thus making the initialization time of the program insignificant (start interpreter for python, perl, java, spawn of a new process, ...).

Moreover, considering that for these tests, the input and the output don't matter, and that it might be interesting to test the result for different sizes of plaintext message, we decided to simply pass in parameter the number of bytes to encrypt and then initialize the array with an arbitrary message value.

The script launches the programs implemented in the different language with the same parameters:

- The key (first parameter of the script)
- The number of bytes to encrypt; it is possible to give several number of bytes, the script iterates on the number of arguments to launch the tests with all the parameters.
- The number of times the encryption process should be repeated.

In order to have some results to analyse, we ran the tests for all the languages and for different number of bytes. For this first tests, we computed the durations in the script by running the program with a lots of iterations on the encrypting process (this functionnality is explained later) to make the rest of the time negligeable and compute an average that would be as close as possible of duration from the actual encryption time.

The results below are expressed in seconds:

Input (bytes)	C	Java	Perl	Python	Assembly
1	0.0002128	0.009017	0.00296189	0.0026721	0.0001744
100	0.0001986	0.0070740	0.0013649	0.0032263	0.0001626
10000	0.0003064	0.0085251	0.0154201	0.0092925	0.0002799
1000000	0.0046334	0.0153853	1.3140923	0.6825420	0.0043138
5000000	0.0254401	0.0433961	6.1996585	3.4717734	0.0221296
10000000	0.0472536	0.0828023	12.8377507	6.8467364	0.0425221

Figure 3.1: Encryption time for different input sizes and different languages

3.3 Conclusion

As expected, the results obtained in C and assembly are the bests (see figure 3.1 & figure 3.2). This is really not a surprise since thes two languages are both low-level. We can actually notice that the results in assembly are only slightly better than the results in C.

Python and Perl being interpreted language, we guessed that the results wouldn't be really good and we can see that indeed, the difference is quite important: for large numbers of bytes the results of the Perl implementation are more than 100 times worst than the results of the Java or C implementations.

The interesting part of these results is that even if for small numbers of bytes to encrypt (until about 100,000) the difference between C and Java is important, we can observe that for large numbers of bytes, this difference is really not that important. The results in C are only 2 times better than the results in Java.

From these results, we finally decided to try to optimize both the C and Java versions and try to compare the results at the end.

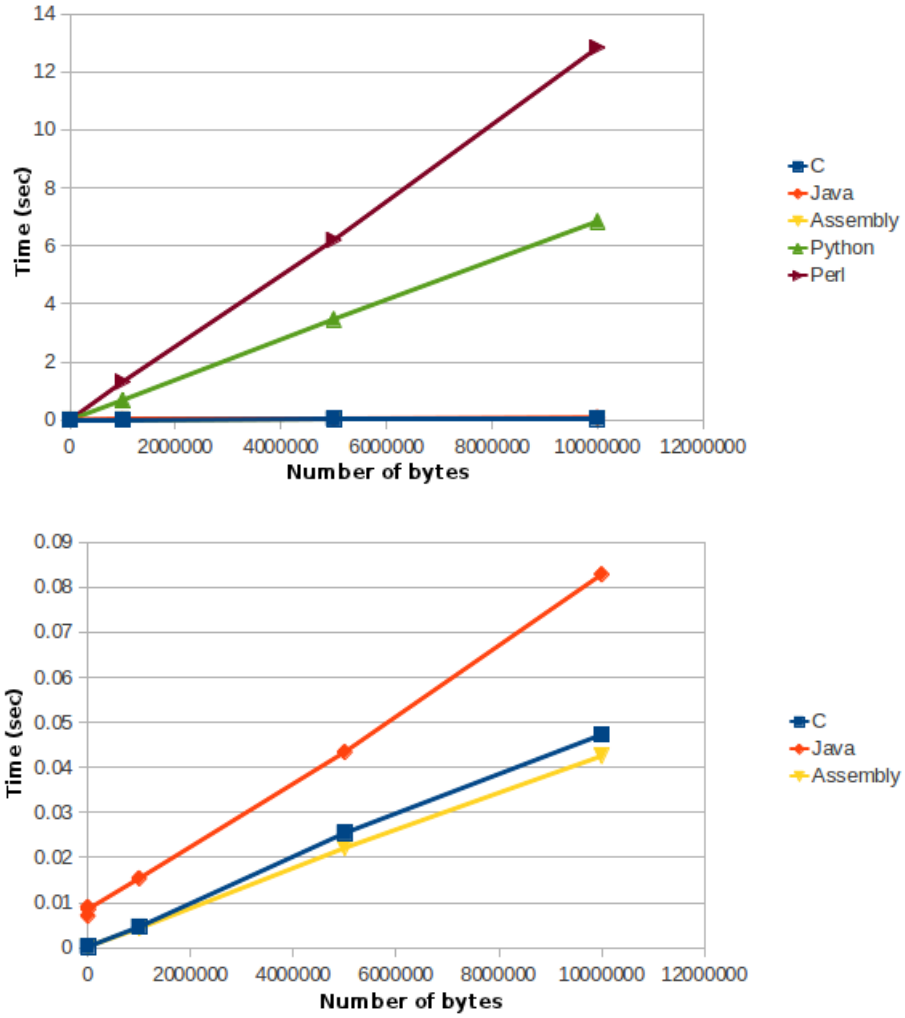


Figure 3.2: Chart showing execution time for different input text size

Chapter 4

Initial design

4.1 Initial design of our software

As explained before, we decided to work on both our C and Java implementations. The code for these two versions can be found in the appendix and a precise explanation of the structure is described below. In order to make it easier for the user and the different tests that we ran, all the programs (in the different languages) have the same calling convention, same inputs, same outputs. Several options can be used and here we will especially make use of these two:

- `--bench=N` : gives the possibility to chose the number of bytes to encrypt. Instead of reading the plaintext message from the standard input, we generate a random plaintext of N bytes.
- `--bench-loop=M` : The encryption is made M times and the average duration is computed. It gives a better precision of the results.

For more detail on the different options available, refer to chapter 7 where the exact specifications are given.

4.1.1 C implementation

The structure of this implementation is made of three majors functions:

- **The main function** is responsible for parsing the arguments (options & encryption key), for calling the encryption function and printing the result if necessary.
- **The RC4 keystream generating function** is responsible for running the KSA and PRGA algorithms in order to generate the requested amount of keystream bytes.
- **The RC4 encrypting function** is responsible for calling the keystream function and using the result to encrypt the input text.

Depending on the arguments passed to the main routine of the program, there are two possible major senarios. In benchmark mode, the input text is a random input of variable size that will be encrypted one or more times. In standard mode, the input is read from stdin and the output is written to stdout.

4.1.2 Java implementation

The structure of this implementation is quite simple and is divided in several parts. First, we have the main function that represent our software. The main program has several objectives:

- Check that the program was properly called with the correct options and display the help if necessary.
- Instanciate the RC4 object using the constructor corresponding to the option and parameters given by the user. Two different constructors are available.
- Launch the encryption of the input text (M times if specified by the user; 1 time by default).

For the constructor, we have two different constructors: one that simply load the input of the standard input and another one that generate a random plaintext of N bytes where N was specified by the user. In this first implementation, we only used bytes for the array (intput, key, S, keystream, output) and the indexes (i and j).

Now, for the encryption function, we simply divided it in three sub functions:

- The key scheduling algorithm that initialize the S array.
- The pseudo random generation algorithm that generates a keystream of the size of the input.
- The generation of the output by XORing the keystream and the input plaintext.

The generation of the output is not really interesting and is directly done in the encryption function, but it might be more interesting to see what we did for the KSA and the PRGA. For this two subfunctions, we actually encountered a slight problem: bytes are not unsigned ; thus the values in the S array that are also used as indexes are in the range [-128 ; 127] instead of [0 ; 255]. To solve this problem, we simply created a static function (posValue) that takes into parameter a byte and returns the same if it is positive and the byte plus 256 if it is negative. This way, the value are in the right range, and we can use this function when we need to use a value of an array as index (and only in that situation because we don't want to modify the bytes in the array).

4.2 Examples of inputs and outputs of our program

The results are obviously the same for all of the different implementations we have written. We have taken care to make them all respect the same calling convention and the same output format. The program takes one or more arguments : zero or more options and the private key, specified as an ASCII string. The input plaintext (resp ciphertext) is to be read from STDIN while the output ciphertext (resp plaintext) will be written to STDOUT. More details about the different options are given in the chapter: User's manual.

Below are a few examples of input/output of our program:

```
$ echo -n "Baptiste" | ./rc4 --out-hexa "Quentin"
b07ce1f8d34cb924
```

Figure 4.1: Encrypting "Baptiste" using the key "Quentin". The result is given in hexadecimal as a raw output would not be well suited for display.

```
$ echo -n "b07ce1f8d34cb924" | ./rc4 --in-hexa "Quentin"
Baptiste
```

Figure 4.2: Decrypting the previous output using the same key ("Quentin").

```
\$ uname -o | ./rc4 "Torvalds" | ./rc4 "Torvalds"
GNU/Linux
```

Figure 4.3: Encrypting & Decrypting the output of the *uname* command. As we can see, the result of this command is preserved through the next encryption and decryption.

4.3 The initial performances of our software

In order to optimize the performances of our software, the first step consisted in evaluating these performances. The interesting part with this algorithm is that we have two distinct elements to study: First we have the KSA for which, the size of the input won't have any effect; actually, only the key and its length slightly affect its performances. And then we have the PRGA that directly depends on the size of the input.

So in order to optimize properly our software, we will have to study the two cases separatly. Therefore, we will study the measurments on encryption for two really different number of bytes (using the benchmark option of our program).

First, we will work with a number of bytes in the same order of magnitude than the size of the S array (256 bytes). This way, we will be able to observe the KSA profile and draw some conclusion. Moreover, since the duration of encryption is very small, in order to have reliable data, we will use the bench-loop option to encrypt the message a large number of times and compute the average that will be way more significative than the time for a single encryption.

Then, to study the performances of the PRGA, we will work with really large number of bytes (size of the plaintext superior to 100 time the size of the S array). In this situation, the time required by the KSA to initialized the S array will be negligeable and we will only pay attention to the performance of the PRGA. Moreover, we won't have to repeat the encryption many times to obtain reliable data since the size of the input will already be important and therefore provide us with the precision that we seek.

4.3.1 C implementation

Before looking at the profiling results of this implementation, let us analyze the total encryption time for different input size. To avoid any approximation induced by the initialization of the program, we will only consider the time required for encryption. To have more precise results, the running times will be averaged over several runs.

To compute the average time of an encryption, We surrounded the encryption loop (successive encryption of the same plaintext with the same key) with timers:

```
long start = utime();
for(i = 0; i < bench_loop; i++)
    rc4_crypt(key, text, text_length);
printf("encryption time = %ld\n", (utime() - start)/bench_loop);
```

First Results

We can now test the program with the following command:

```
./rc4 --bench=N --bench-loop=M Key
```

Where N is the nuber of bytes and M the number of iterations. Here are a few samples of running times for different number of bytes and different number of iterations. The key used was "Batumki"

For small inputs and 100,000 iterations:

Parameters	50 Bytes	100 Bytes	200 Bytes	300 Bytes
Time (msec)	0.0015	0.0023	0.0025	0.0026

We may notice that the running time is not proportional with the size of the input plaintext. This can easily be explained by the fact that there are constant computations involved: the KSA which always requires 256 iterations no matter what the input size is. Testing performances with small input sizes such as those tested in the previous array is important as it will help measure the efficiency of optimizations on the KSA.

For large inputs and 100 iterations:

Parameters	1,000,000 Bytes	2,000,000 Bytes	3,000,000 Bytes	4,000,000 Bytes
Time (msec)	4.301	8.408	12.646	16.758

We may notice that for important input size, the running time is linear with the number of bytes of the plaintext. This is due to the fact that the time spent on the KSA becomes insignificant.

Method level profiling

Now we are going to analyze the profile of our implementation in order to get hints for future optimizations. As the GNU profiler does only provide function level profiling, we have break the keystream generation function down into several sub routines: rc4_ksa, rc4_prga.

This is the results for a small input (100 bytes)

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 72.80      1.44     1.44  1000000     1.44     1.44  rc4_ksa
 17.19      1.78     0.34  1000000     0.34     0.34  rc4_prga
  6.07      1.90     0.12  1000000     0.12     1.90  rc4_keystream
  4.04      1.98     0.08  1000000     0.08     1.98  rc4_crypt
```

We can see that most of the running time is spent on the KSA algorithm, which is not surprising for a small input size (100 bytes). The rest of the time is spent in the PRGA algorithm, in the rc4_keystream (which is responsible for initializing the permutation array and allocating memory for the keystream). The encrypting function (rc4_crypt) is only responsible for 4% of the total running time.

Here are the results for a large input (10000000 bytes)

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 82.13      3.61     3.61      100    36.14    36.14  rc4_prga
 17.97      4.40     0.79      100     7.91    44.04  rc4_crypt
  0.00      4.40     0.00      100     0.00    36.14  rc4_keystream
  0.00      4.40     0.00      100     0.00     0.00  rc4_ksa
  0.00      4.40     0.00        2     0.00     0.00  utime
```

Here we can see that most of the time is spent on the PRGA algorithm and that about 20% of the total running time is spent on the encrypting function. This information tells us what we need to improve to get better results for large inputs (which is what we want).

If we try to look closer and find the bottleneck of our implementation we can see that, in the PRGA algorithm, the lines responsible for most of the running time are :

```
SWAP(unsigned char, S[i], S[j]);   (~ 60 %)
*out = S[ (S[i] + S[j]) & 0xFF ];  (~ 40 %)
```

Namely, a byte swap which yields reading two bytes at two random positions in the permutation array (which is not cache friendly) and writing two bytes at the same (cached) positions. The next line consists in reading two bytes from memory (actually reading these bytes from the memory is not necessary as they are cached in the processor registers from the previous line), then reading a byte at position `S[i] + S[j]` which can be considered random (which is again not cache friendly) and then writting the result to the current keystream position (incremented by 1 each time, so cache friendly).

The bottleneck of the encrypting function `rc4_crypt` is reduced to this line:

```
*text ^= *ptr;
```

It consists in reading two bytes from memory, XORing them, and writing them back to the current location in the text buffer. The pointers (`ptr, text`) that are incremented at each iteration are actually cached in the processor registers so incrementing them is not really time consuming. This can be easily seen when compiling using the `-S` option of GCC.

4.3.2 Java implementation

First of all, before looking at the profile of the software, the first thing to do is to evaluate the time of the encryption. And this time, unlike the approximation that we had for the first tests (where the times were computed in a script, outside the programs), we will only consider the time required for the encryption. The plaintext is randomly generated at the beginning and then used for all the iterations.

To compute the average time of an encryption, I simply surrounded my encryption loop (successive encryption of the same plaintext with the same key) with timers:

```
long t1 = System.nanoTime();
for (int i = 0; i < m; ++ i)
    cipher.encrypt ();
cipher.print();
long t2 = System.nanoTime();
```

The time for one encryption is then easily computed by subtracting t1 to t2 and dividing the results by m (the number of iteration).

Basic Results

We can now test the program with the following command:

```
java RC4 --bench=N --bench-loop=M Key
```

Where N is the nuber of bytes and M the number of iterations. Here are a few examples for different number of bytes and different number of iterations. The key used was "Batumki"

For small inputs and 1,000,000 iterations:

Parameters	50 Bytes	100 Bytes	200 Bytes	300 Bytes
Time (msec)	0.002425	0.002849	0.003335	0.004557

As expected, the results are not linear for small number of bytes because of the KSA that will always do its 256 iteration, for any input size. These few values illustrate the reason why it is important to also optimize the KSA: If we only have small plaintext to encrypt, all the time will be spent in the KSA that will always do its 256 iterations (even if the plaintext is shorter than 256 bytes).

For large inputs and 100 iterations:

Parameters	1,000,000 Bytes	2,000,000 Bytes	3,000,000 Bytes	4,000,000 Bytes
Time (msec)	7.857301	15.239555	22.643495	29.894436

And as we already saw it before, for important number of bytes to encrypt, the time is linear to the number of bytes because the time spent on the KSA become negligeable.

For the rest of the optimization work, I will work with the two following sets of parameters:

- 100 Bytes and 1,000,000 iterations
- 1,000,000 Bytes and 100 iterations

Method level profiling

Now that we have a few basic results, we can start working without the JIT to avoid affecting the profiling and the measurments we will use to optimize the program. Before going to a line level profiler, let's take a look at the results we have with a simple method level compiler.

This is the results for the small input

```
Elapsed time: 0.044581 msec

Flat profile of 44.59 secs (4358 total ticks): main

  Interpreted + native   Method
```

```
 48.5%  2112  +     0    RC4.prga
 48.3%  2104  +     0    RC4.ksa
  1.8%    79  +     0    RC4.posValue
  1.3%    58  +     0    RC4.encrypt
  0.1%     4  +     0    RC4.main
  0.0%     1  +     0    java.util.regex.Pattern.rangeFor
100.0%  4358  +     0    Total interpreted
```

With this input, we can see that the KSA represent almost the half of the time of the encryption. The PRGA also represent almost 50% of the time. This result is really not surprising because since the input is small, the other part of the code such as the generation of the output from the keystream are really fast to compute.

So this first method level profile doesn't give us much information on how to improve our implementation. Let's now take a look at the results with a large input:

```
Elapsed time: 202.801817 msec

Flat profile of 20.29 secs (1994 total ticks): main

  Interpreted + native   Method
 61.3%  1222  +     0    RC4.prga
 27.8%   554  +     0    RC4.posValue
 10.9%   218  +     0    RC4.encrypt
100.0%  1994  +     0    Total interpreted
```

This profile is actually much more interesting than the previous one. First of all, it definitely confirms we conclude previously: the KSA is completely negligeable for large inputs. Then, as expected, most of the time is spend on the pseudo random generation algorithm, and the rest of the function encrypt (the generation of the output from the keystream) now represent more than 10% of the total time. But the most interesting information here is the importance of the function posValue: almost 30% of the total time. This function wasn't a problem for small inputs (because of the small size of the plaintext, there were few access that required the function), but it here appears as a serious loss of efficiency.

Line level profiling

After this preview of the situation with the method level profile, let's dig a little more into the subject with a line level profile.

First, the results with Performances for small inputs. An extract of the result can be found in the appendix REFXXXXX. This profile is really interesting and tells us which lines occupies the most time.

Using this file, we created the following array that litsts the important line and the percentage of the process they represent under the parameters previously stated:

line	83	84	77	68	97	101	102	106
%	23%	21%	10%	10%	9%	5%	4%	4%

With this array, we now know which are the lines we have to try to improve and optimize. So we know that when we want to encrypt 100 byte with our software most of the time is spent in lines 83 and 84. With these informations, it is now really easy to find which line to try to implement, and which lines don't really affect the performance of the program.

We can now do the same with the encryption of 1,000,000 bytes. The resulting file can be find in the appendix and enabled us to do the following table.

line	106	68	101	102	121
%	28%	25%	14%	12%	8%

This array shows us two things:

- The main loop of the PRGA (lines from 101 to 106) occupies more than 60% of the time.
- 25% of the time is lost in the posValue function (line 68) when it could certainly be avoided.

Now, thanks to this profiles, we know where to start the optimization. And at each modification, we will just have to generate the new profile and see if we can still improve the algorithm.

Chapter 5

Revised design

The next step of the project will now consist in using our previous implementation and the different elements that we were able to get from the profiles to optimize our program in both C and Java. At each new version, we will run equivalent tests and try to find new improvement to add to the program

5.1 The evolution of the software

5.1.1 C implementation

The profiling information we saw previously helped us understand what were the most time consuming instructions of our implementation. About 80% of the total running time on the PRGA algorithm and 20% on the encrypting function.

Improving the encrypting procedure

If we first look at the encrypting function `rc4_crypt()`.

```
/* Encrypt/Decrypt a plain/cipher text */
void rc4_crypt(char const* key, char* text, unsigned textlength)
{
    unsigned char *keystream, *ptr;

    /* alloc memory and compute the key stream */
    keystream = malloc(sizeof(unsigned char) * textlength);
    if(keystream == NULL)
        perror("Failed to alloc memory");
    rc4_keystream(key, textlength, keystream);

    /* crypt */
    for(ptr = keystream; textlength > 0; text++, ptr++, textlength--)
        *text ^= *ptr;

    /* free key stream memory */
    free(keystream);
}
```

We can see that the XOR function is applied to each byte of the input text / keystream. That means that when this program is run, the processor does the following for each byte:

- Load current byte of the input text
- Load current byte of the keystream
- XOR them
- Store result in current byte of the input text

The two loading steps implies loading memory from the memory (L1 cache at least) for each byte of the input text. Fortunately, the word size of modern processors is 64 bits (8 bytes). That means we could handle 8 bytes at a time using quads. The PRGA C code becomes:

```
/* Encrypt/Decrypt a plain/cipher text */
void rc4_crypt(char const* key, char* text, unsigned textlength)
{
    unsigned char *keystream, *ptr;

    /* alloc memory and compute the key stream */
    keystream = malloc(sizeof(unsigned char) * textlength);
    if(keystream == NULL)
        perror("Failed to alloc memory");
    rc4_keystream(key, textlength, keystream);

    /* crypt */
    unsigned remainings = textlength & 7;
    textlength >>= 3;
    for(ptr = keystream; textlength > 0; text+=8, ptr+=8, textlength--)
        *((quad_t*)text) ^= *((quad_t*)ptr);

    /* crypt remaining bytes */
    for(; remainings > 0; text++, ptr++, remainings--)
        *text ^= *ptr;

    /* free key stream memory */
    free(keystream);
}
```

It should be stressed that this would only be valid on a 64 bit architecture.

Improving the PRGA procedure

We saw that the bottleneck of our implementation was memory access. Having separated the keystream generator from the encrypting function is not a good idea in that regard. Indeed, if we first generate the whole keystream and then use it to encrypt the whole text, we have to

- Allocate a large memory area (for 10000000 input size, this is about 10MBytes)
- Write to each byte location in the PRGA
- Read from each byte location in the encrypting function

For an input of 10M bytes, that means the program will do 20M memory access. This significantly affects the total running time of our program. And yet, it could be avoided. Indeed, if we included the PRGA operation in the encryption loop, we would not have to store the keystream anymore: the current keystream byte computed by the PRGA would be stored in a register and XORed right away with the current byte of the input text.

In this version, the PRGA procedure disappear and the encryption function becomes:

```
/* Encrypt/Decrypt a plain/cipher text */
void rc4_crypt(char const* key, char* text, unsigned textlength)
{
    unsigned char S[256];
    unsigned i, j, keylength;

    /* compute the size of the key */
    for(keylength = 0; *key; keylength++, key++);
    key -= keylength;

    /* initializes the permutation array */
    for(i = 0; i < 256; i++)
        S[i] = i;

    /* run the ksa */
    rc4_ksa(key, keylength, S);

    /* crypt */
    for(i = 0, j = 0; textlength > 0; text++, textlength--)
    {
        i = (i + 1) & 0xFF;
        j = (j + S[i]) & 0xFF;
        SWAP(unsigned char, S[i], S[j]);
        *text ^= S[ (S[i] + S[j]) & 0xFF ];
    }
}
```

Now we can see there are minor improvements we could do. Indeed, the modulus 256, here written as `& 0xFF` could easily be avoided if we were working with byte index. The encrypting function becomes:

```
/* Encrypt/Decrypt a plain/cipher text */
void rc4_crypt(char const* key, char* text, unsigned textlength)
```

```
{
    unsigned char S[256];
    unsigned keylength;
    unsigned char i=0, j;

    /* compute the size of the key */
    for(keylength = 0; *key; keylength++, key++);
    key -= keylength;

    /* initializes the permutation array */
    do { S[i] = i; i++; } while(i != 0);

    /* run the ksa */
    rc4_ksa(key, keylength, S);

    /* crypt */
    for(i = 0, j = 0; textlength > 0; text++, textlength--)
    {
        i++;
        j += S[i];
        SWAP(unsigned char, S[i], S[j]);
        *text ^= S[ (unsigned char)(S[i] + S[j]) ];
    }
}
```

Now would be a good idea to merge the previous improvement made on the encrypting function (when we XORed 8 bytes at a time). It unfortunately does not make any sense as the keystream bytes are generated on the fly. We have tried generating 8 bytes of the keystream at a time and then XOR these 8 bytes as a quad with the current 8 bytes of the input text, however this implies memory access, and as the benchmarking of this solution showed, it actually makes the running time worse.

5.1.2 Java implementation

We previously saw that one of the things to change was the posValue function. Indeed, we saw with the method level profile and then with the line level profile that this function occupies almost 30% of the time for encryption of large inputs. The best solution that we came up with was finally to use shorts instead of bytes. In java, short are the equivalent to unsigned byte and therefore are perfectly adapted for what we are doing.

The modification were not event really important, but the results were quite impressive. For the tests with the small inputs, the results didn't really changed much, but for the tests with the large inputs, the time went from 7.53 msec with the JIT on the first version to 5.23 for the second version. This single modification improved the result of 30.5%. The results without the JIT were actually event more impressive since we went from 203.5 msec to 127.5 msec, which correspond to an improvement of about 35%.

Considering how good the last results were for large inputs, we decided to keep going in this direction and improve the results even more. So from the profile, we made the same analysis as before and ended up with this really interesting array:

line	108	103	123	104	106
%	46%	21%	17%	9%	4%

So as we can see, almost 50% of the time is occupied by the line 108:

```
keystream[nb] = s[(s[i] + s[j]) % 256];
```

And by studying this line and the possible modifications we could apply to it, we came up with a modification that can actually be applied to several lines. Indeed, several times in the code, we need to compute the values modulo 256. And since 256 is a power of two, the modulo can simply be computed with a simple AND. For example here, we can replace all the %256 in the code by a simple binary operator: &255.

This modification was really easy to implement, and the results were actually surprising. We actually observes a real improvement when we ran the program without the JIT on large inputs (we went from 127.45 msec to 82.69 msec; or an improvement of another 35%). But no concrete improvement were observed when we ran the program with the JIT (we only went from). After giving it some thought, we realized that it might simply be because this modification might already be done by the JIT. This is typically the kind of optimisation a compiler could easily use. We then updated the profile and here is the new array of the most important lines:

line	108	123	103	104	106
%	46%	26%	21%	4%	1%

We can see that the lines from 103 to 108 (constitue the PGRA) represent more than 70% of the time spent.

```
103         i = (i + 1) & 255;
104         j = (j + s[i]) & 255;
105         short tmp = s[i];
106         s[i] = s[j];
107         s[j] = tmp;
108         keystream[nb] = s[(s[i] + s[j]) & 255];
```

Unfortunately it seems there is not much left to do on these lines, so we only have one line left: line 123.

```
123             output[nb] = (short)(plaintext[nb] ^ keystream[nb]);
```

And for this line there might actually somthing really interesting to do. Instead of generating the keystream in the PGRA and then XORing the keystream with the plaintext in another loop, we could generate the the output text directly in the PGRA. It would also save us the trouble of generating the array containing the keystream.

We did implement this modifications and the result are quite satisfying. With the JIT, the total time goes from 5.06 msec to 3.94msec (22% improvement), and without the JIT, the time goes from 81.89 msec to 68.67 msec (16% of improvement).

After all these modifications, more than 96% of the work for is concentrated on few lines we already talked about (See the appendix REF XXX). So for the PRGA, there is not much we can do now. But we can now try to improve the KSA by studying the profile generated with a small input. The three lines that are interesting are the line 79 (20%), 85 (22%), 86 (24%):

```
79      short[] s = new short[256];
...
85      for (int i= 0 ; i<256 ; i++){
86          j = (j + s[i] + key[i % keylength]) & 255;
```

Obviously, we can't do anything for the lines 79 and 85, but there might be something interesting to do with the line 86: we use a counter to get the index i%keylength. This way, we won't have to compute a modulo at each iteration.

After implementing it, we realized that it didn't change anything for a simple reason: the size of the key. Since the keylength can't exceed 256, computing the modulo or using a counter doesn't make a difference.

5.2 The final performances and analysis of our software

5.2.1 C implementation

First, here is a recap of the different versions of our C implementation:

- V0: Original design
- V1: Use of 64-bit words (quad) to XOR 8 bytes at a time.
- V2: Generation of the output directly in the PRGA.
- V3: V2 + Use of 8-bit indexes to avoid modulus.
- V4: V1 + V3

Here are the running times for these different versions for an input of 10000000 bytes (average over 100 samples):

Versions	V0	V1	V2	V3	V4
Time (msec)	45.942	39.715	34.384	30.430	34.551

We can see that both the use of 64-bit words to XOR 8 bytes at a time (V1) and the generation of the output directly in the PRGA (V2) significantly improve the program. The use of 8-bit indexes instead of 32/64-bit indexes to avoid modulus operations (V3) improve the program even more. However if we use all of these at the same time (V4), the running time of the program gets worse.

5.2.2 Java implementation

Through all the previous modifications, we managed to really improve the initial program and have results much closer to the C or assembly implementation. Here is a recap of the evolution of the soft. We have4 version of the soft:

- V1: Initial implementation.
- V2: Suppression of the posValue function.
- V3: Use of the AND binary operator instead of modulos.
- V4: Generation of the output directly in the PRGA.

We only used for this array the results we had with the large inputs since we couldn't really find any efficient way to improve the KSA (thus the conclusions should).

Version	V1	V2	V3	V4
Without the JIT	203.52	127.45	81.89	68.67
Improvement	0%	37.4%	36.22%	16.1%
Total Improvement	0%	37.4%	60.0%	66.3%
With the JIT	7.53	5.23	5.06	3.95
Improvement	0%	30.5%	3.3%	21.9%
Total Improvement	0%	30.5%	32.8%	47.5%
Without/with ratio	27.07	24.3	16.2	17.3

The important discontinuity in the without/with JIT ratio highlight a point that we already mentioned earlier: the improvement that we added in the version 3 might already be handled by the JIT compiler. This would explain why the ratio is close for the first two elements that don't have the improvement and close for the two elements that have it. For the two versions where the optimization was added (V3 and V4), the results will proportionnally improve more without the JIT than with it (because the optimization was already done by the JIT compiler) which explain why the ratio decreases. But the important part of this table is that in the end, with the JIT compiler on, we managed to decrease by 50% the time necessary for the encryption. And when it is turned off, the optimization is over 65%.

5.2.3 Conclusion

After improving our C and Java implementations, it is now time to compare the results. To compare the results, we launched the same command with the two implementations.

First, let's compare the results for a small number of bytes to encrypt (200 bytes). The values given below are averages over 10,000,000 iterations.

Language	Initial version time (msec)	Optimized version time (msec)	Improvement
C	0.0022	0.0020	9.1%
Java	0.003232	0.001570	51.4%

We can see that the results here are really interesting. Even though the implementation in C was already good, we still managed to improve the result. Moreover, for the Java implementation, we divided the encrytption time by two and we now (for this specific input and key) have a faster encryption in java (only for a realy small number of bytes).

For large numbers of bytes (5,000,000 bytes with a result averaged over 100 iterations), we have the following results.

Language	Initial version time (msec)	Optimized version time (msec)	Improvement
C	22.361	15.940	28.7%
Java	36.738581	19.390350	47.2%

These results are also really interesting. We can see that with this optimization of almost 50% the java implementation almost caught up with the C implementation (it actually did better than the initial one). But the optimization of the C program is also really impressing with a improvement of almost 30%.

Considering the fact that the RC4 algorithm is a really simple algorithm that doesn't let much place for improvement (the operations applied are really simple).

Chapter 6

Developer's manual

Here we provide instructions for compiling the different versions of our program.

6.0.4 C implementation

You just have to run `make`. This will generate the `rc4` binary.

6.0.5 Assembly x86-64 implementation

You just have to run `make`. This will generate the `rc4` binary. This will only work on a 64-bit architecture on a 64-bit Linux Kernel >= 2.6.

6.0.6 Python implementation

There is no need to compile anything. You just have to make sure you have a python interpreter >= 2.7. You also have to make sure the python interpreter binary (or at least a link to the binary) is in the directory `/usr/bin/` of your system.

6.0.7 Perl implementation

There is no need to compile anything. You just have to make sure you have a perl interpreter. You also have to make sure the perl interpreter binary (or at least a link to the binary) is in the directory `/usr/bin/` of your system.

6.0.8 Java implementation

The program should be compiled with the following command: `javac RC4.java` and it should be run using the command `java RC4 [OPTIONS]... KEY`

Chapter 7

User's manual

As it has been said before, all of our 5 implementations respect the same calling convention, input and output format. Our program can be used in two modes:

- **Standard mode** : The plain/cipher text will be read from the standard input (STDIN) and the resulting cipher/plain text will be written to the standard output (STDOUT)
- **Benchmark mode** : No plain/cipher text needs to be provided on STDIN and no output will be written to STDOUT. The program will encrypt a number of random bytes (number given as a parameter) and this encryption will be repeated a given number of times (this is a parameter as well).

7.1 Standard mode

The plain/cipher text should be provided on STDIN. It can be specified either as a raw text (this is the default behavior) or as hexadecimal (in which case the option `--in-hexa` should be specified in the arguments). So for example the two followings commands are equivalent:

```
$ echo -n "43727970746F"|./rc4 --in-hexa Batumki
$ echo -n "Crypto"|./rc4 Batumki
```

In a very similar way, we can ask the program to output the cipher/plain text either as a raw text (this is the default behavior) or in hexadecimal (in which case the option `--out-hexa` should be specified in the arguments).

It should be stressed that the STDIN stream must be closed before the encryption starts. On Unix you may send the End-of-Transmission character using the keys `<CTRL-D>`.

7.2 Benchmark mode

In benchmark mode, there are only two important options:

- `--bench=N` where N is the number of bytes to encrypt, it should be greater than 1 for the benchmark mode to be activated.
- `--bench-loop=M` where M is the number of times to repeat the encryption. If it is not specified, the encryption will be run only once.

7.3 Argument format

When invoking our program, the user should specify first all the options that he wants to activate and then specify the key as the last argument (see figure 7.1).

```
Usage: rc4 [OPTION]... KEY
Encrypt or decrypt a plain/cipher text using the RC4 algorithm
The input text will be read from stdin and en/decrypted using the key KEY

  --bench=N         benchmark mode, encrypts N random bytes
  --bench-loop=M    the encryption will be run M times, default is 1
  --in-hexa         input text will be interpreted as hexa, default is raw
  --out-hexa        output text will be written in hexa, default is raw
```

Figure 7.1: Help output provided as an error when the user does not properly uses the command.

Chapter 8

What we learned from the project

This project was actually really interesting in many ways.

First of all, this project gave us the opportunity to implement a concrete cryptographic primitive and also see and understand its whole mechanism on our own. Even if the RC4 is not the most complicated stream cipher, it was interesting to try to figure out how it works and what made it a such popular stream cipher.

Another element that we really appreciated with this project was that it required us to work with profilers. Since we are not really used to this kind of tool, it was the perfect occasion to learn how it works and how to use it to improve program performances. This kind of tools will always be used and knowing how to use it will always be useful.

Finally, one of the last thing that we learned for this project was LaTeX. It is not directly related with the subject, but it really is one of the things we learned for this project and more precisely for this report.

Chapter 9

Possible future work

After finishing all the optimizations we could possibly find on this stream cipher, we decided to take a look at RC4's variants and see if it would give us some new ideas. Unfortunately, none of the variance really deals with performances.

The main objective of these variants is not to help with the performances but to try to make the stream cipher more secure with simple ideas. The key scheduling being the main weakness of the algorithm, the RC4A variant for example deals with the problem by using two different S array. The VMPC is another variant that is similar in every way to the RC4 algorithm except for the KSA that will iterate 768 times instead of only 256.

So an idea of future work would be to consider this time the security aspect of the cipher more than the performance and try to come up with another, more secure, variant of RC4. Many people already worked on the subject, so it might not be easy to get new ideas, but it should be really interesting to try to make this really popular stream cipher more secure.

Chapter 10

Workload sharing

In order to be as productive as possible, as soon as the project started, we decided to organize and divide the work. Here is the workload that we managed to keep for this project:

Implementation in several languages

Tasks	GALVANE Quentin	UZEL Baptiste
Python implementation		X
C implementation		X
Perl implementation	X	
Java implementation	X	
Assembly implementation		X
Tests on implementations	X	X

Finding and implementing optimizations

Tasks	GALVANE Quentin	UZEL Baptiste
Java improvements	X	
C improvements		X
Developer's manual	X	
User's manual		X

Chapter 11

Appendix

```
Elapsed time: 0.050771 msec
Dumping CPU usage by sampling running threads ... done.

...
TRACE 300053:
RC4.ksa(RC4.java:83)
RC4.encrypt(RC4.java:116)
RC4.main(RC4.java:196)
TRACE 300059:
RC4.ksa(RC4.java:84)
RC4.encrypt(RC4.java:116)
RC4.main(RC4.java:196)
TRACE 300061:
RC4.ksa(RC4.java:77)
RC4.encrypt(RC4.java:116)
RC4.main(RC4.java:196)
TRACE 300052:
RC4.ksa(RC4.java:84)
RC4.encrypt(RC4.java:116)
RC4.main(RC4.java:196)
TRACE 300055:
RC4.prga(RC4.java:97)
RC4.encrypt(RC4.java:118)
RC4.main(RC4.java:196)
TRACE 300060:
RC4.prga(RC4.java:101)
RC4.encrypt(RC4.java:118)
RC4.main(RC4.java:196)
...
CPU SAMPLES BEGIN (total = 4978) Sat Feb 18 10:06:24 2012
rank   self  accum   count trace method
   1 23.20% 23.20%    1155 300053 RC4.ksa
   2 11.09% 34.29%     552 300059 RC4.ksa
   3 10.91% 45.20%     543 300061 RC4.ksa
   4 10.14% 55.34%     505 300052 RC4.ksa
   5  8.72% 64.06%     434 300055 RC4.prga
   6  5.04% 69.10%     251 300060 RC4.prga
   7  4.34% 73.44%     216 300064 RC4.prga
   8  3.90% 77.34%     194 300056 RC4.prga
   9  3.56% 80.90%     177 300054 RC4.posValue
  10  3.29% 84.19%     164 300058 RC4.encrypt
  11  3.21% 87.40%     160 300057 RC4.prga
  12  2.23% 89.63%     111 300068 RC4.posValue
  13  2.19% 91.82%     109 300066 RC4.posValue
  14  2.17% 93.99%     108 300065 RC4.posValue
  15  1.95% 95.94%      97 300072 RC4.prga
  16  1.33% 97.27%      66 300067 RC4.ksa
  ...
```

Figure 11.1: Line level profile of the version 1 for 100 Bytes and 1,000,000 iterations

```
Elapsed time: 223.317847 msec
Dumping CPU usage by sampling running threads ... done.

...
TRACE 300056:
RC4.prga(RC4.java:101)
RC4.encrypt(RC4.java:118)
RC4.main(RC4.java:196)
TRACE 300059:
RC4.prga(RC4.java:106)
RC4.encrypt(RC4.java:118)
RC4.main(RC4.java:196)
TRACE 300061:
RC4.prga(RC4.java:102)
RC4.encrypt(RC4.java:118)
RC4.main(RC4.java:196)
TRACE 300058:
RC4.prga(RC4.java:106)
RC4.encrypt(RC4.java:118)
RC4.main(RC4.java:196)
TRACE 300053:
RC4.posValue(RC4.java:68)
RC4.prga(RC4.java:106)
RC4.encrypt(RC4.java:118)
RC4.main(RC4.java:196)
TRACE 300067:
RC4.posValue(RC4.java:68)
RC4.prga(RC4.java:106)
RC4.encrypt(RC4.java:118)
RC4.main(RC4.java:196)

...
CPU SAMPLES BEGIN (total = 2188) Sat Feb 18 10:05:16 2012
rank   self  accum   count trace method
   1 14.35% 14.35%     314 300056 RC4.prga
   2 11.79% 26.14%     258 300059 RC4.prga
   3 11.61% 37.75%     254 300061 RC4.prga
   4 10.83% 48.58%     237 300058 RC4.prga
   5  9.69% 58.27%     212 300053 RC4.posValue
   6  8.09% 66.36%     177 300067 RC4.posValue
   7  7.91% 74.27%     173 300062 RC4.encrypt
   8  6.90% 81.17%     151 300060 RC4.posValue
   9  5.67% 86.84%     124 300054 RC4.prga
```

Figure 11.2: Line level profile of the version 1 for 1,000,000 Bytes and 100 iterations

```
Elapsed time: 127.451420 msec
Dumping CPU usage by sampling running threads ... done.

...
TRACE 300056:
RC4.prga(RC4.java:108)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300054:
RC4.prga(RC4.java:103)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300057:
RC4.prga(RC4.java:108)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300058:
RC4.encrypt(RC4.java:123)
RC4.main(RC4.java:199)
TRACE 300059:
RC4.prga(RC4.java:104)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300062:
RC4.encrypt(RC4.java:123)
RC4.main(RC4.java:199)
TRACE 300053:
RC4.prga(RC4.java:108)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300055:
RC4.prga(RC4.java:106)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
...
CPU SAMPLES BEGIN (total = 1251) Sat Feb 18 13:17:15 2012
rank   self  accum   count trace method
   1 22.46% 22.46%     281 300056 RC4.prga
   2 20.86% 43.33%     261 300054 RC4.prga
   3 19.10% 62.43%     239 300057 RC4.prga
   4 10.39% 72.82%     130 300058 RC4.encrypt
   5  8.71% 81.53%     109 300059 RC4.prga
   6  6.71% 88.25%      84 300062 RC4.encrypt
   7  4.64% 92.89%      58 300053 RC4.prga
   8  3.52% 96.40%      44 300055 RC4.prga
```

Figure 11.3: Line level profile of the version 2 for 1,000,000 Bytes and 100 iterations

```
Elapsed time: 82.693405 msec
Dumping CPU usage by sampling running threads ... done.

...
TRACE 300054:
RC4.prga(RC4.java:103)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300053:
RC4.prga(RC4.java:108)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300058:
RC4.encrypt(RC4.java:123)
RC4.main(RC4.java:199)
TRACE 300056:
RC4.encrypt(RC4.java:123)
RC4.main(RC4.java:199)
TRACE 300057:
RC4.prga(RC4.java:108)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300059:
RC4.prga(RC4.java:104)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300055:
RC4.prga(RC4.java:108)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
TRACE 300062:
RC4.prga(RC4.java:106)
RC4.encrypt(RC4.java:120)
RC4.main(RC4.java:199)
...
CPU SAMPLES BEGIN (total = 808) Sat Feb 18 14:01:27 2012
rank   self  accum   count trace method
   1 24.88% 24.88%     201 300054 RC4.prga
   2 23.89% 48.76%     193 300053 RC4.prga
   3 14.98% 63.74%     121 300058 RC4.encrypt
   4 11.01% 74.75%      89 300056 RC4.encrypt
   5 10.40% 85.15%      84 300057 RC4.prga
   6  7.18% 92.33%      58 300059 RC4.prga
   7  4.08% 96.41%      33 300055 RC4.prga
   8  1.24% 97.65%      10 300062 RC4.prga
```

Figure 11.4: Line level profile of the version 3 for 1,000,000 Bytes and 100 iterations

```
Elapsed time: 68.669057 msec
Dumping CPU usage by sampling running threads ... done.

THREAD END (id = 200004)
TRACE 300053:
RC4.prga(RC4.java:102)
RC4.encrypt(RC4.java:119)
RC4.main(RC4.java:195)
TRACE 300052:
RC4.prga(RC4.java:107)
RC4.encrypt(RC4.java:119)
RC4.main(RC4.java:195)
TRACE 300054:
RC4.prga(RC4.java:107)
RC4.encrypt(RC4.java:119)
RC4.main(RC4.java:195)
TRACE 300055:
RC4.prga(RC4.java:103)
RC4.encrypt(RC4.java:119)
RC4.main(RC4.java:195)
TRACE 300057:
RC4.prga(RC4.java:107)
RC4.encrypt(RC4.java:119)
RC4.main(RC4.java:195)
...
CPU SAMPLES BEGIN (total = 675) Sat Feb 18 14:47:37 2012
rank   self  accum   count trace method
   1 42.22% 42.22%     285 300053 RC4.prga
   2 30.67% 72.89%     207 300052 RC4.prga
   3 10.67% 83.56%      72 300054 RC4.prga
   4 10.37% 93.93%      70 300055 RC4.prga
   5  2.96% 96.89%      20 300057 RC4.prga
```

Figure 11.5: Line level profile of the version 4 for 1,000,000 Bytes and 100 iterations

```
Elapsed time: 0.022470 msec
Dumping CPU usage by sampling running threads ... done.

...
TRACE 300057:
RC4.ksa(RC4.java:86)
RC4.encrypt(RC4.java:117)
RC4.main(RC4.java:195)
TRACE 300053:
RC4.ksa(RC4.java:79)
RC4.encrypt(RC4.java:117)
RC4.main(RC4.java:195)
TRACE 300055:
RC4.ksa(RC4.java:85)
RC4.encrypt(RC4.java:117)
RC4.main(RC4.java:195)
...
CPU SAMPLES BEGIN (total = 2200) Sat Feb 18 14:58:32 2012
rank   self  accum   count trace method
   1 24.50% 24.50%     539 300057 RC4.ksa
   2 21.73% 46.23%     478 300053 RC4.ksa
   3 20.05% 66.27%     441 300055 RC4.ksa
```

Figure 11.6: Line level profile of the version 4 for 100 Bytes and 1,000,000 iterations