Serpent Cipher

Christopher Johnson
Kevin Lakotko

Description

Serpent is a 32-round SP-network. The cipher operates on blocks of 128-bits. It is important to note that the cipher does its computations in little-endian. The key is variable but internally is required to be 256-bits therefore they pad any supplied key that is less than 256-bits with a 1 followed by the required number of 0s to make the key 256bits. At a very high level the cipher is an initial permutation followed by a round function executed 32 times and then a final permutation. The output from the final permutation is the ciphertext.

The initial and final permutations are used to help optimize implementation. They are not required when implementing the cipher using bit-slicing. When using bit-slicing it puts the bits into the correct place required for the other functions of the algorithm. The permutations are inverses of each other.

After completion of the initial permutation the output is used as input to the round function. The round function consists of key mixing, a pass through the s-boxes, and in all but the last round a linear transformation. Key mixing consists of xoring the round subkey with input to the round. After key mixing the result is then passed through the s-boxes. There are 32 rounds and 8 s-boxes where a single s-box is used for each round so each s-box is used four times. If the round requires a linear transformation that is applied now. If not there is an additional key mixing operation used.

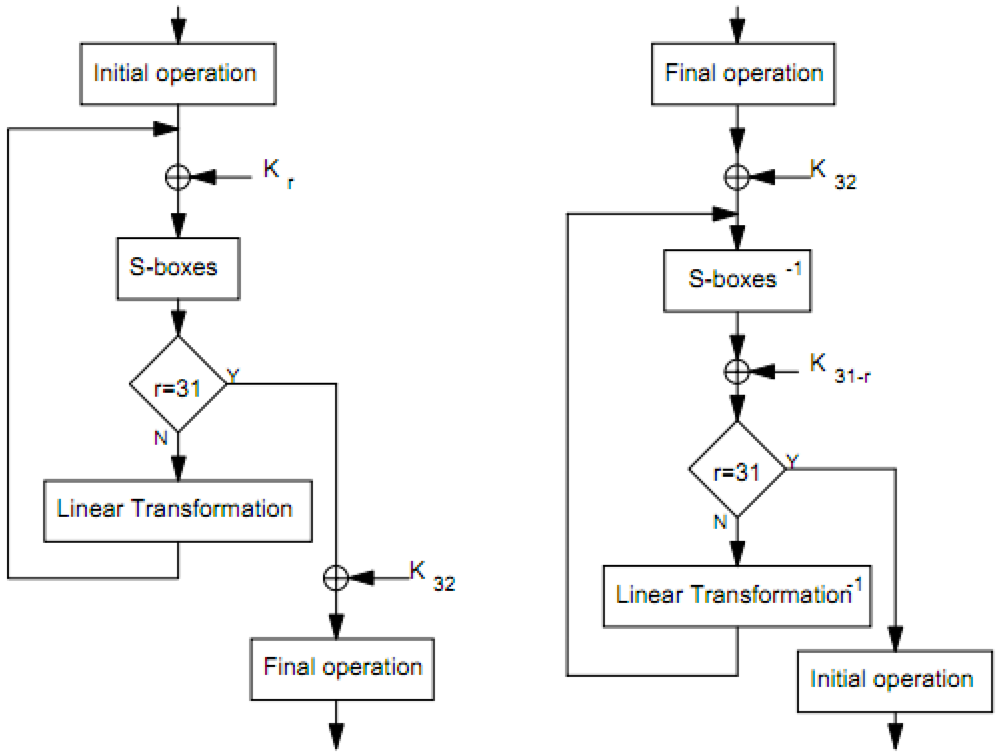

Figure 1: The round function for encryption and decryption
http://csrc.nist.gov/archive/aes/round2/comments/20000513-pbora.pdf

The linear transformation is a linear mixing. When using bit-slicing it is not needed to use the permutations and can use the algorthm specified in the figure. Otherwise a table look up is used. The table lookup takes certain bits and xors them together to get the output bit. So output bit one would be the xor of input bits 16, 52, 56, 70, 83, 94, and 10.

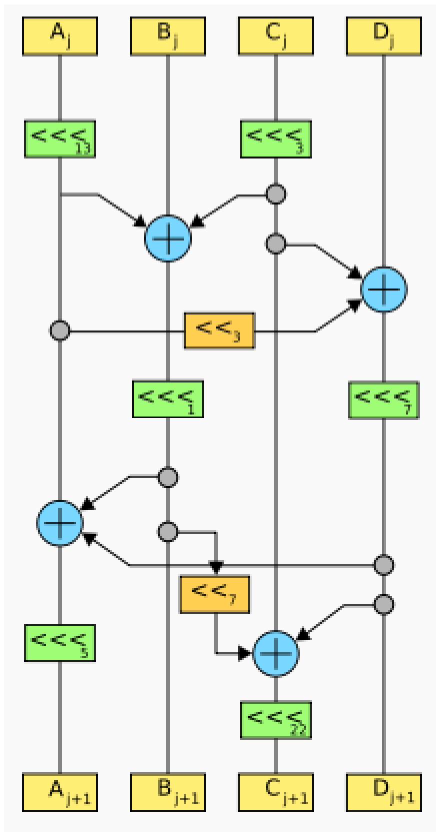

Figure 2: The linear mixing stage
http://en.wikipedia.org/wiki/Serpent_(cipher)

To generate the round subkeys the initial 256 bit key is broken down into 8 32-bit words. These are then expanded to an intermediate key. This means that w_0 depends on the original key and then w_1 depends on w_0. After the 131 prekeys are generated they are passed into the s-boxes starting at S3 and going through them backwards. These outputs are 131 subkeys and make up the round subkeys by concatenating k_{4i},k_{4i+1},k_{4i+2},k_{4i+3}. If not using bit-slicing then the initial permutation must be run on the round subkey to get the bits in the proper order.

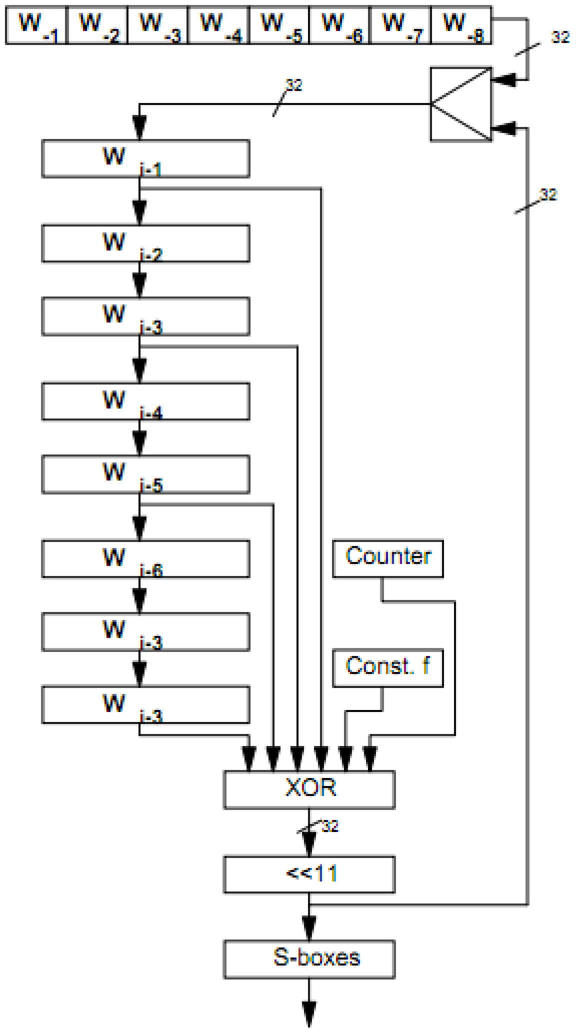

Figure 3: The round key generation
http://csrc.nist.gov/archive/aes/round2/comments/20000513-pbora.pdf

Since this is a symmetric cipher decrypting is trivial. This is accomplished from doing everything in reverse and using the inverse lookup tables and apply the s-boxes in reverse order.

I/O Examples

Plaintext	Key	Ciphertext
0000000000000000 0000000000000000	0000000000000000 0000000000000000 0000000000000000 0000000000000000	8910494504181950 F98DD998A82B6749
0000000000000000 0000000000000000	1111111111111111 1111111111111111 1111111111111111 1111111111111111	2B49414C9E0F97A7 1C1EC5A0055F871A
1111111111111111 1111111111111111	0000000000000000 0000000000000000 0000000000000000 0000000000000000	147C0F0B6B28B7F2 451F9AEB6B616898
1111111111111111 1111111111111111	1111111111111111 1111111111111111 1111111111111111 1111111111111111	E2B941F1A5A12EDB 2F1F77D5A5EA82A4
0101010101010101 0101010101010101	0101010101010101 0101010101010101 0101010101010101 0101010101010101	4827FCFF24454CF8 89642A5BB12397EC
1010101010101010 1010101010101010	1010101010101010 1010101010101010 1010101010101010 1010101010101010	AF62A5F479242C71 2F22AA77E3F85800

Ciphertext	Key	Plaintext
0000000000000000 0000000000000000	0000000000000000 0000000000000000 0000000000000000 0000000000000000	7DAB898203AC242C 29327FD80A241BE8
0000000000000000 0000000000000000	1111111111111111 1111111111111111 1111111111111111 1111111111111111	C202B3846084956C 7A7C3860DAA5078A
1111111111111111 1111111111111111	0000000000000000 0000000000000000 0000000000000000 0000000000000000	BF639CB44A5B3481 2111A675AD5B0524
1111111111111111 1111111111111111	1111111111111111 1111111111111111 1111111111111111 1111111111111111	1C8F51EEE2E35DEE C16D832281D879CB
0101010101010101 0101010101010101	0101010101010101 0101010101010101 0101010101010101 0101010101010101	CEE2590558F9F190 E3818E15E025FA38
1010101010101010 1010101010101010	1010101010101010 1010101010101010 1010101010101010 1010101010101010	CCDA2A7A822DB0DA 8B1777BF55D6FE08

Original Implementation

- The original running time measurements and profile of your software
- An analysis of the original measurements, identifying changes that would reduce the running time

The original implementation of the software separated the area of concerns into different functions. This was to allow multiple people to work on the same file without having to worry about stepping on each other's toes. It also allowed for the testing and improvement of the different functions on an individualized basis.

Three main functions were created and then duplicated/reversed. The first three functions created were the encrypt function, the permutation function, and the key generation function. Important information was also stored in arrays to allow for quick access (e.g. S-Boxes, Initial and Final Permutations, and Linear Transformations). This allowed for functions like the permutation function to be re-used for both the initial and final permutation. The encrypt function was copied and reversed to create the decrypt function.

To the outside developer, the only important functions are the encrypt and the decrypt function. These two functions are in charge of calling all other functions and returning the desired output. This design allows a main function to be created in the same file, while also simplifying the process of encryption and decryption from a second file (i.e. test.c).

The timing of the system was performed by running the encrypt and decrypt functions 100000 times in a row each. This gave an average execution time for both functions. Then a program gprof was used to perform a profiling of the execution.

```
Running 100000 times each...
Name            Total Time(s)           Ave. Time(s)
ENCRYPT                 203.130005              0.002031
DECRYPT                 192.130005              0.001921
```

```
 %      cumulative   self        self         total
 time   seconds      seconds     calls        ms/call      ms/call     name
 36.03  80.17        80.17       3583600000   0.00         0.00        getbitblock
 14.52  112.47       32.30       3100000      0.01         0.03        lintrans
 13.75  143.07       30.60       3100000      0.01         0.03        invlintrans
  8.73  162.49       19.41       7000000      0.00         0.00        permutation
  8.14  180.59       18.10       793600000    0.00         0.00        setbitblock
  7.38  197.01       16.43       200000       0.08         0.27        generatekeys
  3.47  204.74       7.72        844800000    0.00         0.00        getbitint
  3.22   211.90      7.17        844800000    0.00         0.00        getbitfour
  1.06   214.26      2.37        211200000    0.00         0.00         makechar
  1.00   216.49      2.23        313600000    0.00         0.00        sboxchar
  0.73   218.12      1.64        204800000    0.00         0.00        getfourbits
  0.71   219.71      1.58        12800000     0.00         0.00        sboxint
  0.58   221.00      1.29        12800000     0.00         0.00        invsboxint
  0.37   221.81      0.81        102400000    0.00         0.00        invsboxchar
  0.10   222.04      0.23        100000       0.00         1.14        encrypt
  0.10   222.26      0.22        100000       0.00         1.08        decrypt
  0.09   222.47      0.20        26400000     0.00         0.00        rotateleft
  0.00   222.47      0.01        200000       0.00         0.00      finalpermutation
  0.00   222.48      0.01        6800000      0.00         0.00    initialpermutation
  0.00   222.48      0.00        2            0.00         0.00        printClock
```

This chart makes it appear that the getbitblock function was taking up the most amount of time.

This is because it gets called more than any other function in the program. The translation functions are next in line, but those functions rely heavily on getbitblock. The permutation function also takes up some time, as it was never refactored to use bit manipulation functions.

Revised Implementation

In order to improve the speed of our implementation, different steps were performed. It was decided to delete some of the most popularly called methods and move their code to locations where they were called. The permutation function was implemented early on in the system design and was never refactored to utilize some bit access/setting functions that were created later on so a refactoring was performed. This lead to some new times and a new profile being recorded.

```
Running 100000 times...
Name                Total Time(s)        Ave. Time(s)
ENCRYPT             94.339996            0.000943
DECRYPT             87.360001            0.000874

Each sample counts as 0.01 seconds.
  %    cumulative    self           self          total
 time   seconds      seconds        calls         us/call       us/call        name
 32.10   60.85       60.85          3100000       19.63         19.63          lintrans
 28.51  114.90       54.05          3100000       17.44         17.44          invlintrans
 17.36  147.82       32.92          7000000       4.70          4.70           permutation
  8.72  164.36       16.54          200000        82.70         330.74         generatekeys
  3.92  171.79       7.43           844800000     0.01          0.01           getbitint
  3.76  178.92       7.13           844800000     0.01          0.01           getbitfour
  1.23  181.25       2.33           211200000     0.01          0.01           makechar
  1.17  183.47       2.22           313600000     0.01          0.01           sboxchar
  0.89  185.15       1.68           204800000     0.01          0.01           getfourbits
  0.82  186.71       1.55           12800000      0.12          0.24           sboxint
  0.69  188.01       1.31           12800000      0.10          0.23           invsboxint
  0.44  188.84       0.83           102400000     0.01          0.01           invsboxchar
  0.15  189.12       0.28           100000        2.80          982.70         encrypt
  0.14  189.38       0.26           100000        2.60          913.10         decrypt
  0.08  189.54       0.15           26400000      0.01          0.01           rotateleft
  0.02  189.57       0.03           6800000       0.00          4.71         initialpermutation
  0.01  189.58       0.01           200000        0.05          4.75          finalpermutation
  0.00  189.58       0.00           2             0.00          0.00           printClock
```

The execution time of the cipher was nearly cut in half. Most of this efficiency can be linked to the removal of certain bit manipulation functions and placing the code into the functions that used to call it. This reduced the number of stack frames that had to be created and destroyed during the encryption decryption process. The attempts at increasing the permutation efficiency did not work, but the time saved from other improvements made up for it.

How to build

Building the software can be done on any machine that can execute make.

- To build the original implementation run ‘make main-orig’. This generates executable called serpent-orig.
- To build the revised implementation run ‘make main-opt’. This generates executable called serpent-opt.
- To build test programs run ‘make test-opt’, and ‘make test-orig’.

How to use

To encrypt or decrypt run the executable and provide the argument -e to encrypt and -d to decrypt. The key must be 256-bits or 64 hex digits. The plaintext or ciphertext can be fed in by piping a file to standard input.

An example of this is:

```
./main-opt -e 1111111111111111111111111111111111111111111111111111111111111111
<clear >cipher
```

Executing the test programs will perform the timing tests and output similar to the following:

```
Running 100000 times...
Name                Total Time(s)        Ave. Time(s)
ENCRYPT             94.339996            0.000943
DECRYPT             87.360001            0.000874
```

Reflection

We learned how to implement a block cipher and everything that entails. This meant dealing with a large amount of bits and manipulating them with shifts and rotations in a systematic manner. The bits were too large for one data type to hold. It would have been much easier if there was a data type that could have held all these bits instead of splitting everything up into 32 and 64 bit data types.

We learned the importance of using a language as a tool to implement. We initially started in Java but Java is big-endian and also does not have unsigned data types. This does not mean it is impossible to implement just that it would be a harder. We then chose C as our implementation language due to the x86 architecture being little-endian and C including unsigned types.

In terms of future work the implementation could become even more efficient using bit-slicing instead of their standard method. With bit-slicing the permutations would no longer be needed and as we saw this is one of our areas that could use improvement.

We completed the project as a pair using pair programming. Since this was all in the same file and just dealing with such a small implementation it was just easier to pair program most of this.

Chris Johnson:

- Created initial repository
- Implemented permutation function
- Implemented the sub key generation

Kevin Lakotko:

- Initial function skeletons
- Implemented linear transformation function
- Implemented the s-boxes and encrypt/decrypt skeletons using other unimplemented functions.

Both:

- Corrected bugs in initial implementation
- Revised/Optimized implementation

References

- List of references

- http://en.wikipedia.org/wiki/Serpent_(cipher)
  This was used to get general information regarding the cipher and diagrams illustrating the algorithms structure.
- http://www.cl.cam.ac.uk/~rja14/serpent.html
  This is the authors' home page for their submission to AES. This includes the specification and some reference implementations used to help test our implementation.
- http://www.cs.technion.ac.il/~biham/Reports/Serpent/
  This was used to get some test data but was in NESSIE format which switches some bytes.
- http://csrc.nist.gov/archive/aes/round2/comments/20000513-pbora.pdf
  This was used for a couple diagrams showing how the cipher works and the key generating phase.