“Blowfish”

An implementation by Edward Wolf and Achilleas Tziaza

Description

The topic we investigated for our team project was Bruce Schneier’s Blowfish cipher. This cipher was proposed in December of 1993, and to this day, no cryptanalysis has been discovered. Even though this algorithm has been replaced with newer ciphers such as Twofish, the Blowfish algorithm still remains a very simple and effective way to encrypt data.

When proposed, the Blowfish algorithm was intended to replace DES because of its vulnerable 56-bit key length. With the processing power of computers increasing, it was possible that DES could become no longer secure. Schneier’s algorithm was aimed at providing a cipher that was capable of encrypting data with a key from 32-bits to 448-bits, with the use of a key-dependent S-boxes and P-box.

The purpose of this project was to demonstrate the practical application of this algorithm. The software we developed demonstrated how a plaintext message can be encrypted using a 32-448-bit key or how an encrypted message can be decrypted.

Software Design

Programming language

When designing the software, many approaches were taken into consideration. Initially we looked into using Microsoft's C# since it handles hex numbers in a much better way. The conversion between integer and hex numbers was the main concern when selecting the programming language since the conversions are a vital part of the algorithm. Converting a hex number to an integer makes the use of exclusive-or (XOR) much easier which is a part of the algorithm.

While deciding which language would fit our project the best, we discovered a wide variety of tools in the RIT Computer Science Cryptography library by Dr. Alan Kaminsky. Using the class Hex.java and its static methods that are described in http://www.cs.rit.edu/~ark/cscl/doc/edu/rit/util/Hex.html , we could easily convert a hex string to an integer and back to hex without any concerns. This helped us conclude that our familiar programming language, Java, was the right approach.

The Algorithm

Using the description of the algorithm found in the creator's (Schneier's) website and the resources found in the *"References"* section of this paper, we developed the algorithm.

As a part of the algorithm are a *P-Box* and four *S-Boxes*. The P-Box is given by the creator whereas the S-Boxes are created using the hexadecimal digits of the constant "π". For this implementation of the Blowfish algorithm, the vectors for the S-Boxes and the P-Box were taken directly from Bruce Schneier's website. This saves time when the algorithm is ran since the P-Box and the S-Boxes do not have to be created, a process that is, comparatively to the algorithm, tedious.

The key that is used for Blowfish has a variable length from 32-bits up to 448-bits in 8-bit increments. This implies flexibility in the strength of the algorithm. The key is used in the following way:

- Take the first 32 bits of the key and XOR them with the first element of the P-Box array. Then take the next 32 bits and XOR them with the second element of the P-Box array and so on and so forth. If there are less than 32-bits remaining for the next item, use the remaining bits and get the needed bits from the beginning of the key as if the key is really a circular string.

- Repeat this process for each S-Box and the P-Box

This clearly implies that the bigger the key, the fewer times the same parts of the key have to be used, which in turn implies that the algorithm is even harder to break. This is also the reason why the creation of the S-Boxes and P-Box were not included in the algorithm and were, instead, taken from the creator's website. Since the key affects the S-

Boxes and P-Box, having them as constants inside the java class does not necessarily imply a security issue since they change according to the key.

Blowfish encrypts and decrypts 64-bit blocks. This means that in terms of ASCII, Blowfish can encrypt/decrypt strings of length 8. Such strings are, for example, "Ed Wolf!" which has a length of 8 characters (ASCII). Each ASCII character is represented as an 8-bit binary code which means that there are 64 bits in an ASCII string of 8 characters. Such characters include exclamation marks and spaces.

To encrypt, the 64-bits that are given as an input to the algorithm are separated to two 32-bit strings. The first 32-bits, starting from the most significant bit (MSB) until the 32nd bit, are called L and the rest are called R, which account as left and right, as shown in *Figure 1* below:

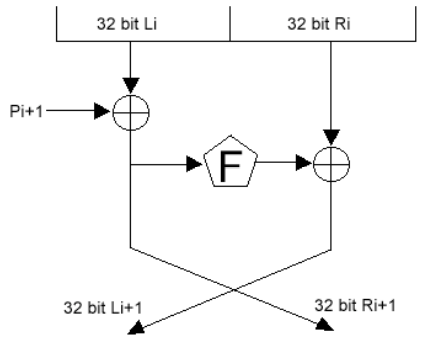


Figure 1: The feistel network (Blowfish, Encryption)

The left side is XORed with an integer from the P-Box that has the same index. It is then passed through the function F (described later on) and XORed with the right side of the 64-bit input. Then the left and right chunks are flipped. This happens 16 times. On the 16th round there is no XOR-ing happening. The L and R are, instead, flipped. Then the final L is XORed with the 17th index of the P-Box and the final R is XORed with the 18th index of the P-Box. This new 64-bit output is the ciphertext and in the algorithm provided for this project, is given in hex. Since each hex number is 4-bits long, the resulting ciphertext has a length of 16 hex characters. The ciphertext along with the key can be later on used to decrypt the message.

The function F of the Blowfish algorithm is given by the following formula:

$$((sBox1[a] + sBox2[b]) \wedge sBox3[c]) + sBox4[d])$$

where sBox1,sBox2,sBox3 and sBox4 are the four S-Boxes and a,b,c,d represent the 32-bit input to the function F, chopped in four consecutive chunks of 8 bits each, starting from the MSB, in the following manner:

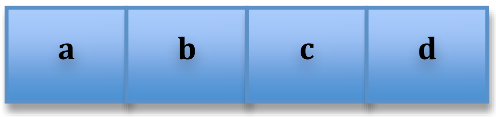


Figure 2: The input of the function F is divided into four 8-bit chunks.

The decryption in Blowfish is the exact same as the encryption with the only difference that the indices of the P-Box are used in reverse order (instead of looping from the first index to the 16th and then using indexes 17 and 18, to decrypt we loop from the 18th index to the 3rd and then use the first and second for the final round).

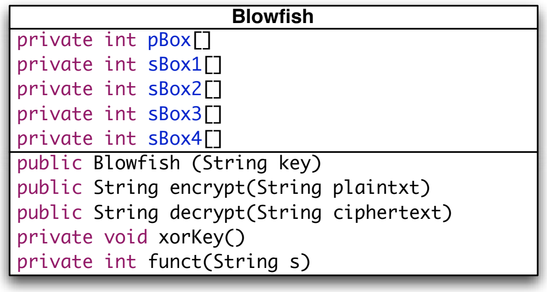
The fact that the decryption is the same as the encryption but in reverse order, makes the algorithm very easy to implement. This sort of "built-in" functionality of the algorithm was something we appreciated when coding the algorithm since we only had to figure out the encryption and then reverse the order of the loop and the indices to get the decryption!

Developer's Manual

Using our implementation of the Blowfish algorithm is very easy. Please follow the steps provided below:

- Once you download the jar file from our website you need to extract it in a folder. The following files should appear in the folder:
    - Blowfish.java – the implementation of the algorithm
    - Hex.java – a class to manipulate hexadecimal numbers taken from the Computer Science Cryptography Library
    - BlowfishProject.java – the main file you can run to test the algorithm.
- Navigate to that folder and use *javac *.java* to recompile and avoid using any previous versions of the .class files that might have accidentally been included in the jar file.

For the developer we have included a UML diagram of the class that was created, with instructions how to integrate this class in your own software development.



```
Blowfish
private int pBox[]
private int sBox1[]
private int sBox2[]
private int sBox3[]
private int sBox4[]
public Blowfish (String key)
public String encrypt(String plaintxt)
public String decrypt(String ciphertext)
private void xorKey()
private int funct(String s)
```

Figure 3: UML diagram of Blowfish.java

A developer could easily use this simple class to incorporate the Blowfish algorithm by including the Blowfish.java file as part of a bigger software platform. The Blowfish constructor takes a key (see key size in the description of the algorithm) as a string of hexadecimal characters and then calls the xorKey() function to XOR the key with the P-Box and S-Boxes. This is done automatically and the developer does not need to worry about this part of the algorithm. Once the key is instantiated, there is no need to pass it when encrypting and decrypting. If the main program remains loaded, the key is stored in a private variable in the Blowfish class and cannot be accessed by anybody besides the Blowfish class itself. A user can then send encrypt and decrypt requests to the software by calling the encrypt() and decrypt() functions.

Since the input is 64-bits, if the input string is larger than that, it can be divided into 64-bit pieces. Each of the 64-bit inputs can be passed through the encryption function and the final ciphertext can be padded together to produce the resulting ciphertext. The same can happen for the decryption.

Once compiled, the program can be executed by following the instructions specified in the *User's Manual* section of this report.

User's Manual

After you compile the software using the instructions provided in the *Developer's Manual* use the following instruction to run the software:

- Type: ***java BlowfishProject [key] [plaintext or ciphertext] [-e encrypt or -d decrypt]*** to encrypt or decrypt a message where:
    - [key] is replaced with a 32-448 bit key given in hex, in increments of one hex character (8-bits)
    - [plaintext or ciphertext] is replaced with the paintext or the ciphertext depending on the operation the user would like to perform
    - [-e or –d] denote encryption or decryption

Since the Blowfish algorithm encrypts only 64-bit blocks, we used the Electronic Codebook mode to encrypt or decrypt the given plaintext or ciphertext. The ECB mode simply divides the input into 64-bit chunks. This way the resulting ciphertext or plaintext are:

- C = E(K,P1) + E(K,P2)
- P = D(K,C1) + D(K,C2)
    - Where E() denotes encryption, P() denotes decryption, K is the key and P1,P2,C1 and C2 are the first two chunks of the input

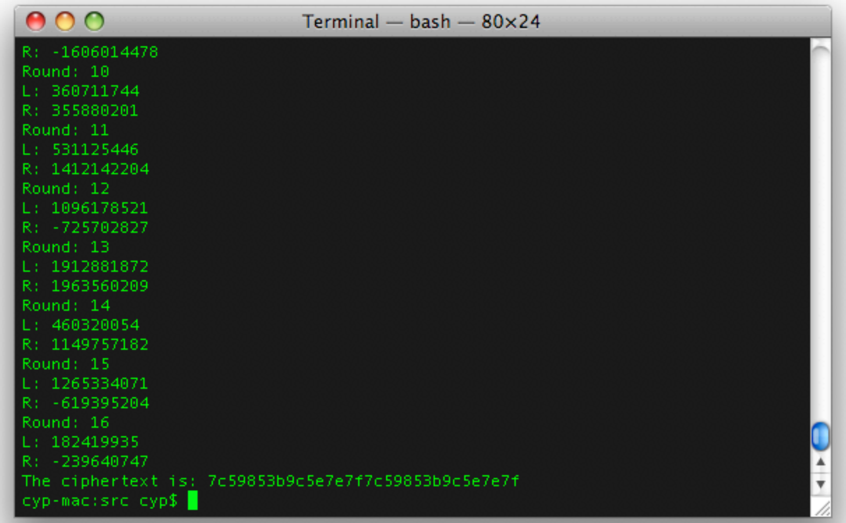
For simplicity reasons and in order to demonstrate that the encryption and decryption can be done in smaller chunks, we only allow for an input of 16 ASCII characters for the plaintext (128-bits) and an input of 32 hex characters for the ciphertext (128-bits). This way two encryptions or two decryptions are happening and the use of the algorithm is demonstrated more than once times.

Also, the interface is currently in a command line mode. Further improvements to the interface could be a graphical user interface.

Below is an example of the use of the software:

java BlowfishProject   f4f3ffffffff   EdWolfEdEdWolfEd   –e

which means: "Encrypt the string EdWolfEdEdWolfEd with the given key"

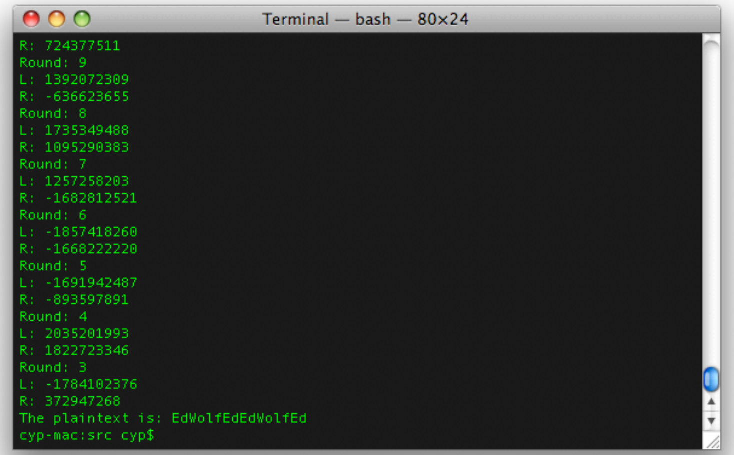


```
R: -1606014478
Round: 10
L: 360711744
R: 355880201
Round: 11
L: 531125446
R: 1412142204
Round: 12
L: 1096178521
R: -725702827
Round: 13
L: 1912881872
R: 1963560209
Round: 14
L: 460320054
R: 1149757182
Round: 15
L: 1265334071
R: -619395204
Round: 16
L: 182419935
R: -239640747
The ciphertext is: 7c59853b9c5e7e7f7c59853b9c5e7e7f
cyp-mac:src cyp$
```

Figure 4: Encryption

As you can see here, each round of the encryption is printed out and the final ciphertext is given at the end. If we want to use the given ciphertext to decrypt it and receive back the plaintext we can use the following command:

java BlowfishProject f4f3ffffffff 7c59853b9c5e7e7f7c59853b9c5e7e7d –d
which means: "Decrypt the given ciphertext 7c59853b9c5e7e7f7c59853b9c5e7e7f with the given key"



```
R: 724377511
Round: 9
L: 1392072309
R: -636623655
Round: 8
L: 1735349488
R: 1095290383
Round: 7
L: 1257258203
R: -1682812521
Round: 6
L: -1857418260
R: -1668222220
Round: 5
L: -1691942487
R: -893597891
Round: 4
L: 2035201993
R: 1822723346
Round: 3
L: -1784102376
R: 372947268
The plaintext is: EdWolfEdEdWolfEd
cyp-mac:src cyp$
```

Figure 5: Decryption

As we can see, the resulting text is our plaintext given back.

Results

As shown in the previous section, the software operates successfully. We also tested the possibility of giving an incorrect key during encryption/decryption, which results to an incorrect output. This means that our code is valid and that the Blowfish algorithm was implemented correctly.

Further tests included using incorrect sizes of the plaintext or ciphertext which yielded the program to not operate successfully and quit, giving the user an error message that was informative.

It is indeed a restriction to the user to have a mandatory length of 16 ASCII characters for the input of the plaintext and 32 hex characters for the input of the ciphertext to the encryption and decryption respectively. This was done though, for display purposes and it does not mean that the algorithm cannot be used for longer inputs. This is the reason why the Blowfish class is separate from the BlowfishProject class. A developer can use the Blowfish class and run it to perform as long of inputs as he or she wants.

All in all, the results were correct and the algorithm works, despite the limitations of the software we created that wraps the class.

What We Learned

First and foremost, through this project we we're afforded the opportunity to explore another algorithm, which was not presented in class. In doing so, we learned how Bruce Schneier was motivated to create a replacement for DES, and also the theory behind creating this algorithm.

Unfortunately, implementing this algorithm in java proved to be challenging. For starters, the java language does not support hex operations in an easy way like C and C++. As such, we were forced to convert all of our plaintext to hexadecimal strings, and then those strings had to be converted to integer in order to be XORed with the P-box and S-boxes using the Hex class we took from our professor's library.

In addition, we learned that no cryptanalysis has been discovered for the Blowfish cipher. While the algorithm is still secure, the developer still recommends that adopters move to his Twofish algorithm. In building off of Blowfish's success, Twofish uses 128-bit block encryption instead of 64-bit. This adds additional security and is therefore more preferable to be implemented.

Lastly, we learned about the applications of this algorithm. Bruce Schneier's website lists over 150 different applications/implementations of the algorithm. After reading through the list, we were fascinated to learn that this algorithm can be use in anything from file/folder encryption to encrypting HTTP traffic.

Possible Future Work

The Blowfish algorithm uses key-dependent S-boxes and P-box to encrypt data. When the key is changed, 521 operations are performed in order to generate the subkeys used in the algorithm. In order to reduce the time of this generation, a file containing all the subkey values could be created for each key. This would reduce the amount of RAM and time needed in order to encrypt data. It is possible that such a file could be loaded when the program parses its command line arguments.

In addition to increasing the algorithm's speed, performing further cryptanalysis on this algorithm could yield additional vulnerabilities. So far, 4 rounds of the Blowfish algorithm are discoverable via a second-order differential attack. Further research would be necessary in order to discover more vulnerability to this algorithm. However, given the fact that the developer of this algorithm is recommending the adoption of Twofish, it may not be worthwhile to spend time analyzing this problem.

One more improvement could happen in the error checking of the algorithm, for example making sure that strings that should contain hexadecimal characters do indeed contain hexadecimal characters and that the input text in ASCII is of the pre-defined size. All the inputs to the Blowfish class are not checked for validity. At least one Exception class that is part of the algorithm needs to be created, to ensure that the proper information is given to the developer when using this implementation of Blowfish as part of a bigger software.

Also, the Electronic Codebook Mode (ECB) is only used for display reasons and it is not supposed to be used in normal software because it leaks too much information about the plaintext. This was, of course, used to display the fact that when an input string is longer than the default input defined by the algorithm, encryption and decryption can still happen but in smaller chunks. It is up to the developer to use other methods such as CBC mode or CFB mode etc. Our purpose was to display the use of the Blowfish class in a wrapper class that processes the input and not to promote the use of ECB mode.

Lastly, the program that we created was done using java. We could implement the program in another language that is friendlier to bitwise operations and unsigned integers (eg. C), which in turn could reduce the code size and complexity. However, given the program's current implementation, we could write a GUI that would show the encryption and decryption of a message using Blowfish fairly easily.

References:

Schneier, Bruce. "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)." Dec. 1993. 22 Mar. 2008 <http://www.schneier.com/paper-blowfish-fse.html>.

Schneier, Bruce. "The Blowfish Encryption Algorithm -- One Year Later." Schneier. Sept. 1995. 22 Mar. 2008 <http://www.schneier.com/paper-blowfish-oneyear.html>.

"Blowfish Encryption." BletchleyPark. 4 May 2008 <http://www.bletchleypark.net/cryptology/blowfish.html>.

"Blowfish (Cipher)." Wikipedia. 3 Mar. 2008. 4 May 2008 <http://en.wikipedia.org/wiki/Blowfish_(cipher)>.