Team RZC: Fast Data Encipherment Algorithm (FEAL)

Zachary Miller (zrm6085@rit.edu)
Carlos Leonardo (cal3678@rit.edu)

FEAL Algorithm

FEAL [1] is a Block Cipher that normally includes eight Feistel Rounds during its execution and works with 64-bit long plaintext and ciphertext blocks, however, in our implementation, the number of rounds is variable. The algorithm begins by taking the initial 64-bit key specified by the user and immediately executes the Key Schedule process. The key schedule process generates a total of 16 subkeys (for an 8 rounds FEAL implementation), each one of 16-bit long. In total, these subkeys conform a 256-bit key.

For the eight round implementation, there is a total of 16 subkeys; one subkey for each round (from 0 to 7), as well as eight additional subkeys (from 8 to 15). Four of these additional subkeys (from 8 to 11) are used in the beginning and the last four (from 12 to 15) in the ending of the encipherment process. This is better understood looking at these step by step indications:

1. The 64-bit long plaintext and User Key are accepted as input.
2. During the key schedule process FEAL takes the 64-bit user key and generates 8 subkeys plus one subkey for each round (each subkey is 16-bit long).
3. First phase of the encipherment algorithm uses 4 subkeys (from 8 to 11).
4. Each Feistel round uses one subkey. Normally, the implementation consists of eight rounds and it uses the subkeys from 0 to 7. Our Implementation has a variable number of rounds.
5. Last phase of the encipherment algorithm uses 4 more subkeys (from 12 to 15).
6. At the end, a random 64-bit ciphertext is returned.

To generate the subkeys, the Key Schedule algorithm uses a special function defined by the authors and an S-function which adds the inputs and rotates the bits. The FEAL encipherment algorithm will output a supposedly random 64-bit ciphertext from the original 64-bit plaintext. This means FEAL needs two 64-bit sequences one for the plaintext and one for the key.

S-Function

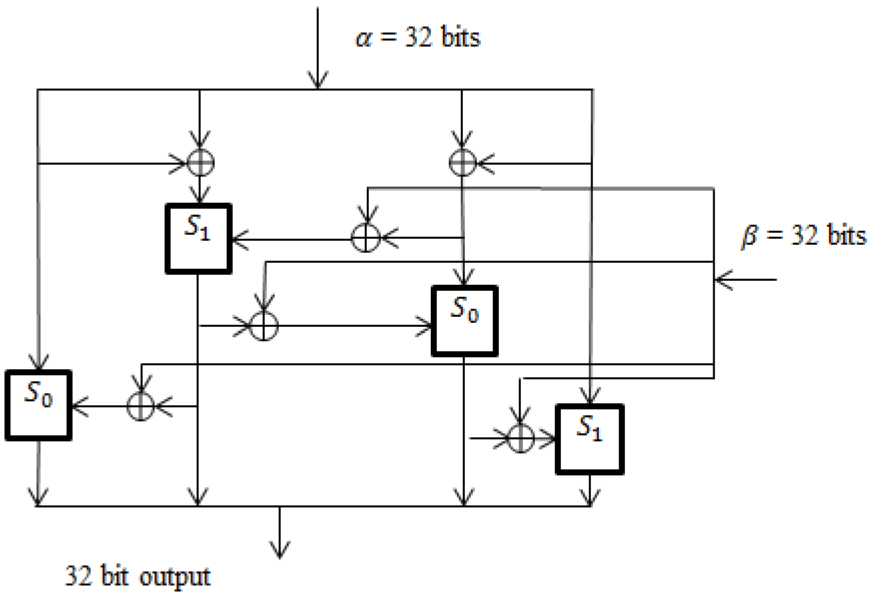
The S-Function is used within the Fk-Function and F-Function. Fk-Function is used during the Key Schedule process and the F-Function is used during the encryption and decryption process. Basically, S-Function takes care of implementing the S-Boxes of FEAL. It accepts three byte parameters and returns a single byte value. The S-Function can be explained following these steps:

```
byte A
byte B
byte δ = 0 or 1
S(A,B,δ) = RotateL2(T)
T = A + B + δ mod 256
```

Fk-Function

The Fk-Function is used during the Key Schedule process to generate the 16-bit subkeys. It is important to mention the Fk-Function internally uses the S-Function. It accepts two 32-bit parameters and returns a 32-bit value containing two 16-bit subkeys each time is executed. The following diagram explains how the Fk-Function works:

```
α and β: 32 bit halves of initial key
```

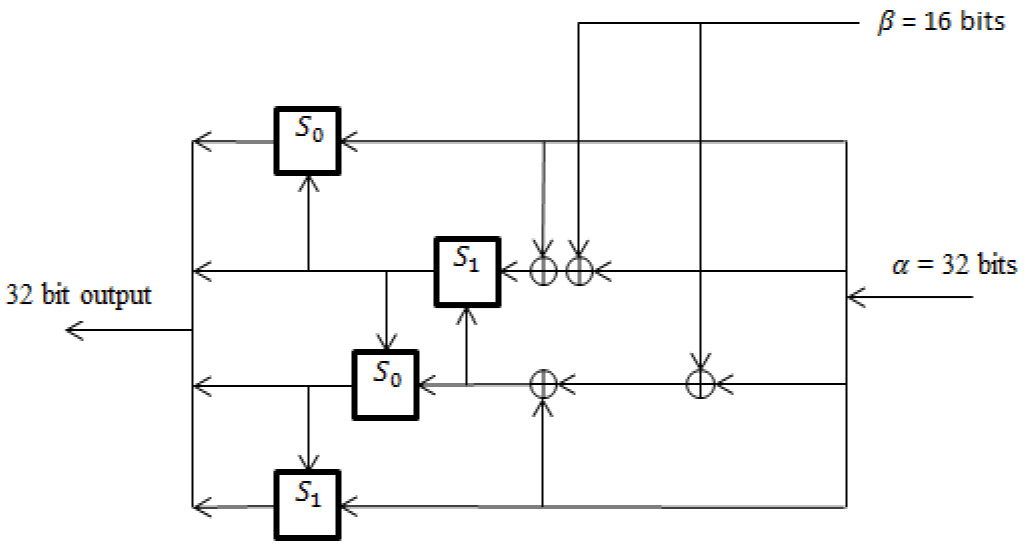


Function F_k

The F-Function

The F-Function is used during the encryption and decryption processes and accepts two parameters. One of the parameters is a 16-bit subkey and the other is a 32-bit text message from the Feistel round. The following diagram explains the F-Function:

```
α: 32 bit from Feistel Round
β: 16 bit sub-key
```



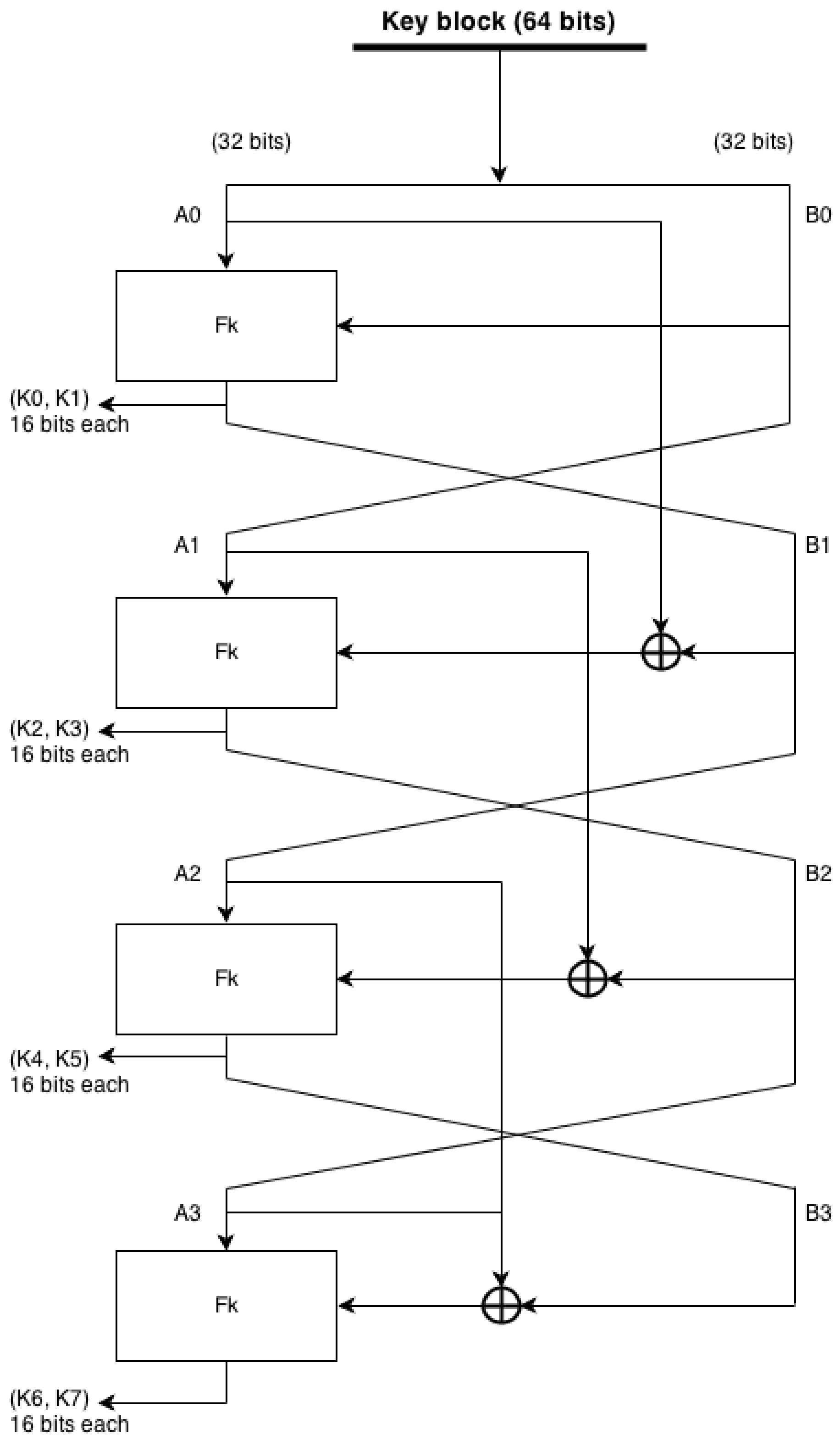
Function F

Key Generation

The key generation function, also called Key Schedule, takes the original 64-bit User Key as input and uses the Fk function to generate the 16-bit subkeys needed by the encryption process. The numbers of keys it generates depends on the number of rounds that will be used. The number of subkeys follows this formula:

```
nsK = nR+8
```

Where `nsK` is the numbers of subkeys needed and `nR` is the number of rounds. The 8 constant correspond to the last additional eight subkeys. Four of these subkeys are used in the beginning and the last four subkeys are used at the end of the encipherment process. This Key Generation Process in explained in the following Key Schedule diagram:



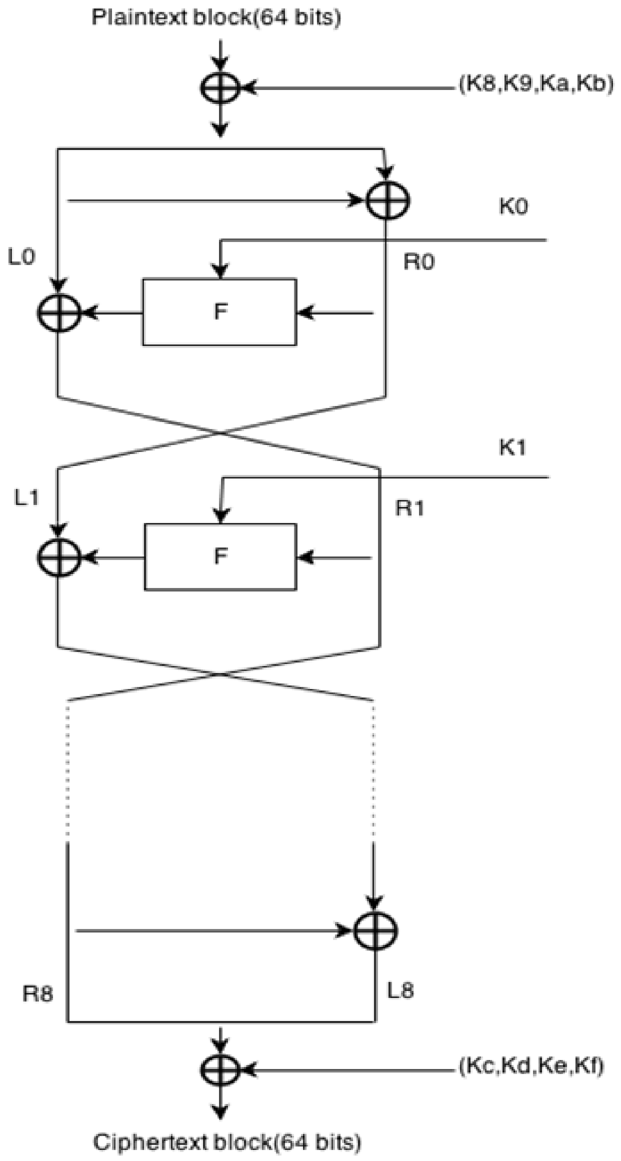
Key Schedule diagram

Encryption

The FEAL encryption process consists of mixing the 64-bit plaintext and key provided by the user and apply the F-Function to each one of the Feistel rounds. In the beginning, four 16-bit subkeys from the key schedule are used. In an eight-rounds implementation, these first four subkeys are from subkey 8 to 11. Then, one 16-bit subkey is used for each Feistel round (from subkey 0 to 7). At the end, the last four 16-bit subkeys (from 12 to 15) are used. The Encryption / Decryption diagram below shows the entire process.

Decryption

The FEAL decryption process uses the same approach as the encryption. The only difference in this case is the process runs backwards. First, it uses the last four 16-bit subkeys (from 12 to 15), then one subkey for each round and then the other four 16-bit subkeys (from 8 to 11) to recover the plaintext. The Encryption / Decryption diagram below shows the entire process.



Encryption / Decryption diagram.

Source Code of our implementation of FEAL

Our source code for our FEAL implementation is displayed below:

```
import java.util.Arrays;
import edu.rit.util.Hex;
import edu.rit.util.Packing;

public class FEALCipher implements BlockCipher
{
	//Set up parameters and default values
	public int R = 8;
	byte deltaParam = 0;
	int numOfSubKeys = (this.R+8);
	short[] subKey = new short[numOfSubKeys];

	//Returns the block size of FEAL in bytes
	public int blockSize()
	{
		return 8;
	}

	//Returns the key size in Bytes
	public int keySize()
	{
		return 8;
	}

	//Function to set the number of rounds.  This is used in our attack
	//It also sets up the appropriate number of subkeys which need to be generated
	public void setRounds(int R)
	{
		this.R = R;
		this.numOfSubKeys = (this.R+8);
		this.subKey = new short[numOfSubKeys];
	}

	//Method to access the subkeys used in the encryption process
	public short[] getSubKeys()
	{
		return subKey;
	}
```

```
//Method to set the key and extend keys to get the subkeys
//Takes in a 64-bit key
//Extends it based on how many rounds are needed
//Makes use of the FK function to generate the key
public void setKey(byte[] key)
{
        //Set up the necessary parameters to extend the key using the Fiestel rounds
        int roundsLimit = ((this.R+8)/2)+1;
        int[] A = new int[roundsLimit+1];  // Left  part of the 64 bits Key (MSB)
        int[] B = new int[roundsLimit+1];  // Right part of the 64 bits Key (LSB)
        int[] D = new int[roundsLimit+1];  // Result from functionFK
        int fkOutput = 0;

        // Spliting the 64 bits key into two 32 bits ints (A and B) to pass those to functionFK
        A[0] = A[0] |= ((key[0] & 255) << 24);
        A[0] = A[0] |= ((key[1] & 255) << 16);
        A[0] = A[0] |= ((key[2] & 255) << 8);
        A[0] = A[0] |=  (key[3] & 255);

        B[0] = B[0] |= ((key[4] & 255) << 24);
        B[0] = B[0] |= ((key[5] & 255) << 16);
        B[0] = B[0] |= ((key[6] & 255) << 8);
        B[0] = B[0] |=  (key[7] & 255);

        // Getting the subkeys: Using the key schedule
        for(int r=1; r < (roundsLimit); r++)
        {
                D[r] = A[r-1];
                A[r] = B[r-1];
                B[r] = functionFK(A[r-1], (B[r-1] ^ D[r-1]));

                subKey[2*(r-1)] = (short) (B[r] >> 16);
                subKey[(2*(r-1)+1)] = (short) B[r];
        }

        //Checks to see final case if our R is odd.  We need an extra key than what is normally used
        if(this.R%2 == 1)
        {
                D[roundsLimit] = A[roundsLimit-1];
                A[roundsLimit] = B[roundsLimit-1];
                B[roundsLimit] = functionFK(A[roundsLimit-1], (B[roundsLimit-1] ^ D[roundsLimit-1]));
                subKey[2*(roundsLimit-1)] = (short) (B[roundsLimit] >> 16);
        }
}
```

```
//Method for encrypting a single plaintext block of 64 bits
public void encrypt(byte[] text)
{
        //Set up side Arrays
        int[] RightArray = new int[R+2];
        int[] LeftArray = new int[R+2];

        //Pack text into Right and Left 32-bit integers
        LeftArray[0] |= ((text[0] & 255) << 24);
        LeftArray[0] |= ((text[1] & 255) << 16);
        LeftArray[0] |= ((text[2] & 255) << 8);
        LeftArray[0] |=  (text[3] & 255);

        RightArray[0] |= ((text[4] & 255) << 24);
        RightArray[0] |= ((text[5] & 255) << 16);
        RightArray[0] |= ((text[6] & 255) << 8);
        RightArray[0] |=  (text[7] & 255);

        //Set up the initial keys for use
        int leftInitKey = 0;
        int rightInitKey = 0;
        leftInitKey |= ((subKey[R]&65535) << 16);
        leftInitKey |= ((subKey[R+1]&65535));

        rightInitKey |= ((subKey[R+2] & 65535) << 16);
        rightInitKey |= ((subKey[R+3] & 65535));

        LeftArray[0] = LeftArray[0] ^ leftInitKey;
        RightArray[0] = RightArray[0] ^ rightInitKey;

        //Initial xor
        RightArray[0] = RightArray[0] ^ LeftArray[0];

        //Feistel rounds
        for(int i = 1; i<= R;i++)
        {
                RightArray[i] = LeftArray[i-1] ^ functionF(RightArray[i-1],subKey[i-1]);
                LeftArray[i] = RightArray[i-1];
        }

        //Final xor of he left and right sides
        LeftArray[R+1] = LeftArray[R] ^ RightArray[R];
        RightArray[R+1] = RightArray[R];
```

```
		//Set up final keys for xor
		int leftFinalKey = 0;
		int rightFinalKey = 0;
		rightFinalKey |= ((subKey[R+4]&65535) << 16);
		rightFinalKey |= ((subKey[R+5]&65535));

		leftFinalKey |= ((subKey[R+6] & 65535) << 16);
		leftFinalKey |= ((subKey[R+7] & 65535));
		//Xor the last subkeys
		LeftArray[R+1] = LeftArray[R+1] ^ leftFinalKey;
		RightArray[R+1] = RightArray[R+1] ^ rightFinalKey;

		//pack the results back into the encryption array
		for(int i = 3;i>=0;i--)
		{
			text[i] = (byte)RightArray[R+1];
			RightArray[R+1] = RightArray[R+1]>>>8;
		}
		for(int i = 7;i>=4;i--)
		{
			text[i] = (byte)LeftArray[R+1];
			LeftArray[R+1] = LeftArray[R+1]>>>8;
		}
	}

	//Function F used for encryption
	//Inputs: A, B, where A is the plaintext and B is the subkey
	private int functionF(int A, short B)
	{
		//set up and pack int and short inputs into byte arrays for convinence and set up a output array
		byte[] subA = new byte[4];
		byte[] subB = new byte[2];

		subA[0] = (byte)(A>>>24);
		subA[1] = (byte)(A>>>16);
		subA[2] = (byte)(A>>> 8);
		subA[3] = (byte)(A>>> 0);

		subB[0] = (byte)(B>>>8);
		subB[1] = (byte)(B>>>0);
```

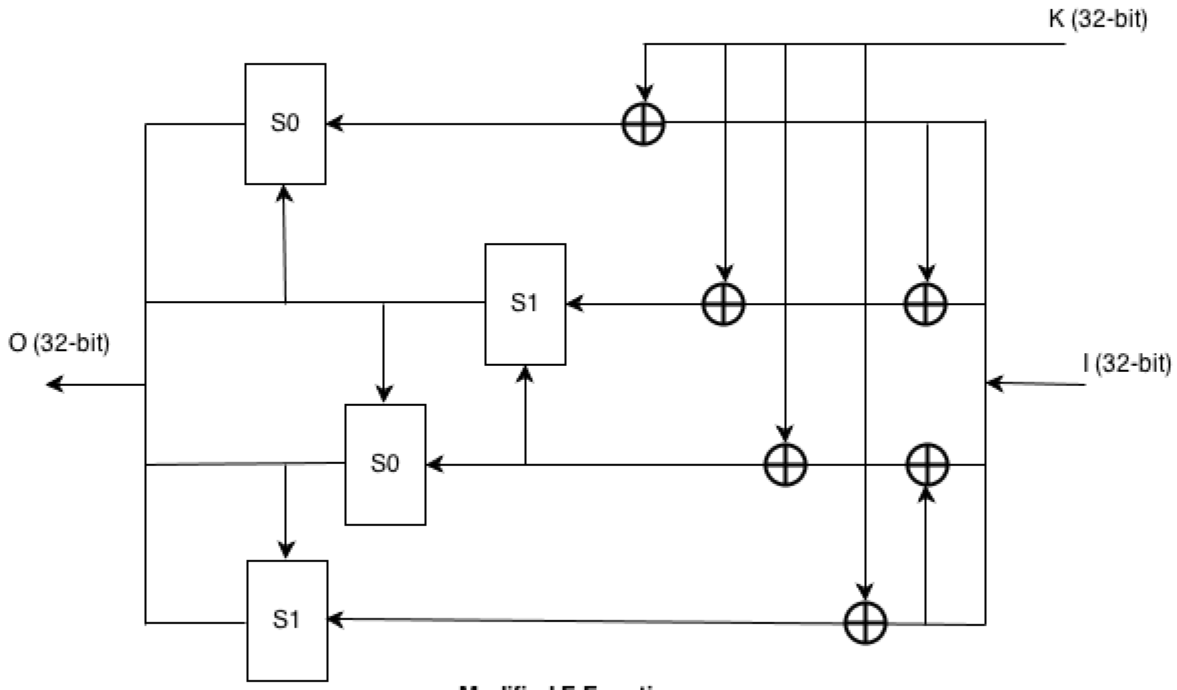
```
        byte[] fOut = new byte[4];

        //Do some initial xoring
        fOut[1] = (byte)((subA[1]&255) ^ (subB[0]&255) ^ (subA[0]&255));
        fOut[2] = (byte)((subA[2]&255) ^ (subB[1]&255) ^ (subA[3]&255));

        //Call the S function given a particular path through the Function
        fOut[1] = functionS(fOut[1],fOut[2],(byte)1);
        fOut[2] = functionS(fOut[2],fOut[1],(byte)0);
        fOut[0] = functionS(subA[0],fOut[1],(byte)0);
        fOut[3] = functionS(subA[3],fOut[2],(byte)1);

        //Set up and pack the result into an output integer
        int output = 0;
        output |= (fOut[0]&255);
        output = output<<8;
        output |= (fOut[1]&255);
        output = output<<8;
        output |= (fOut[2]&255);
        output = output<<8;
        output |= (fOut[3]&255);

        return output;
    }

    //Function FK which is used in the key generation phase
    //Inputs: A and B which are both 32-bit integers
    private int functionFK(int A, int B)
    {
        //Set up substitute bytes and pack in the two inputs
        byte[] subA = new byte[4];
        byte[] subB = new byte[4];

        subA[0] = (byte)(A>>>24);
        subA[1] = (byte)(A>>>16);
        subA[2] = (byte)(A>>> 8);
        subA[3] = (byte)(A>>> 0);

        subB[0] = (byte)(B>>>24);
        subB[1] = (byte)(B>>>16);
        subB[2] = (byte)(B>>> 8);
        subB[3] = (byte)(B>>> 0);
```

```
                //Set up the output byte array for convinence
                byte[] fKOut = new byte[4];
                fKOut[1] = (byte)((subA[1]&255) ^ (subA[0]&255));
                fKOut[2] = (byte)((subA[2]&255) ^ (subA[3]&255));

                //Do the appropriate xor functions
                fKOut[1] = functionS(fKOut[1], (byte)((fKOut[2]&255) ^ (subB[0]&255)),(byte)1);
                fKOut[2] = functionS(fKOut[2], (byte)((fKOut[1]&255) ^ (subB[1]&255)),(byte)0);
                fKOut[0] = functionS(subA[0], (byte)((fKOut[1]&255) ^ (subB[2]&255)),(byte)0);
                fKOut[3] = functionS(subA[3], (byte)((fKOut[2]&255) ^ (subB[3]&255)),(byte)1);

                //Pack the output into an integer
                int output = 0;
                output |= (fKOut[0]&255);
                output = output<<8;
                output |= (fKOut[1]&255);
                output = output<<8;
                output |= (fKOut[2]&255);
                output = output<<8;
                output |= (fKOut[3]&255);

                return output;
        }

        //S Function used in both F and FK
        //Takes in 2 bytes and a 1 or zero for delta.
        //Sums them then mods by 256 and left rotates by 2
        private byte functionS(byte A, byte B, byte delta)
        {
                //Sum inputs
                byte T = (byte)(((A&255) + (B&255) + (delta&255))%256);

                //Left Rotate by 2
                return (byte)(((T&255)<<(byte)2)|((T&255)>>>(byte)6));
        }
}
```

We apologize for the organization of this previous section as we needed to put all of the source code in even though we have over 200 lines of code. The contents can also be viewed in the FEALCipher.java class

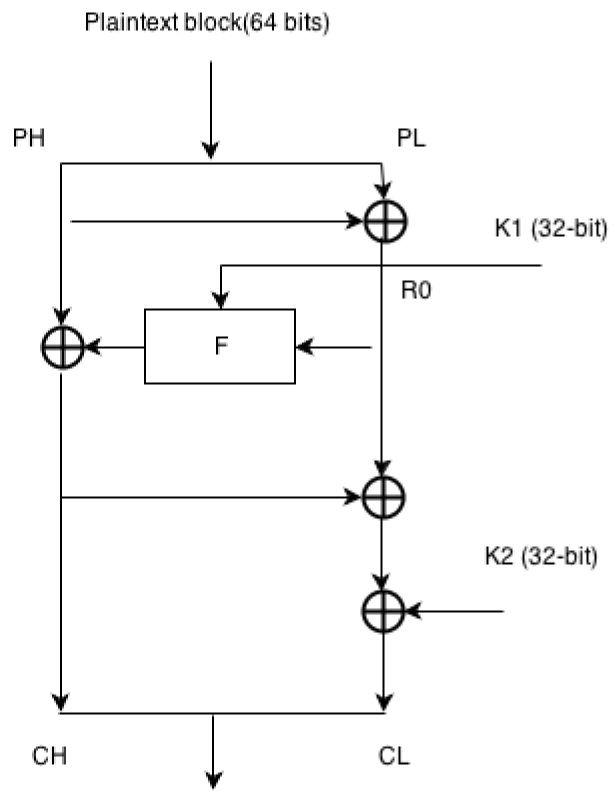
Ad-Hoc Attack on FEAL Cipher

To implement a structural attack on the FEAL block cipher, we first looked into the Linear Attack described in [2]. In this attack, the authors attempt to rewrite the FEAL F-Function into an equivalent form where a 32-bit key is used within the modified F-Function much like the original form with a 16-bit key. It should be important to know here that these 32-bit keys are not part of either the original 64-bit user key used for the Key Schedule input of FEAL, nor are they in any way combinations of the 16-bit subkeys output from the Key Schedule. If we need to refer to these 16-bit subkeys or the original 64-bit key, we will use the keyword “original” to denote that they are from the Encryption part. The only difference between this modified F-Function and the original F-Function, is that instead of only xoring the key with the middle two bytes of the input message, the attack xors the 32-bit key with all of the bytes of the input message. The modified F-Function is displayed below:



Modified F-Function

Taking this idea of the modified F-Function, we then reduced the number of rounds to just one (1). It became apparent that because we have reduced this problem to one (1) round, we no longer could implement the Linear Attack from [2]. However, we can take some ideas from this attack to hopefully break the cipher. We also realized quite quickly that we will need to have two (2) 32-bit keys in order to make the plaintexts equal the outputs when the keys are input. With this 2nd 32-bit key, we will have the following one (1) round algorithm:



One Round FEAL

Now that we have the modified algorithm, we can start to develop linear equations to try to find the keys. First we will take care of the 32-bit key k2. Because we have the plaintexts and the ciphertexts, we can derive the value for k2 with the following equation.

$$k_2 = P_L \oplus P_H \oplus C_H \oplus C_L$$

Even though this value does not seem to be true for all plaintexts and ciphertexts given a particular normal secret key, it will always be the same for a key regardless of the plaintexts and ciphertexts. Because of this, we have a value for k2. Now we need to derive the bits for k1. Because this is only a 32-bit number, we theoretically could brute force it until it gave us the correct output, but this will take far too long and it really would not show any structural attack on the cipher. Instead, we will systematically go through each byte of the Key and figure out a possible key which will be able to be used on the rest of the plaintext and ciphertext pairs. To do this, we start with the second most least significant byte of k1 which we will define as k1[8~15] where 8~15 denotes the 8 through 15th least significant bits of k1. By tracing the path of the modified F-Function while tracing the path for the one (1) round cipher, we see that there must exist a key where the following equation must hold true:

$$\begin{aligned} P_H[8{\sim}15] \oplus C_H[8{\sim}15] \\ = S(C_H[8{\sim}15], P_H[8{\sim}15] \oplus P_L[8{\sim}15] \oplus P_L[0{\sim}7] \oplus P_H[0{\sim}7] \oplus K_1[8{\sim}15], 0) \end{aligned}$$

Using this equation, we can brute force all possible combinations for k1[8~15] which is 256 possibilities and then record which byte will give us the valid key byte. We then can do similar processes for k1[24~31] and k1[0~7]. Neither of these are dependent on any of the other bits within the key so for each key byte, we just add 256 operations. The equations needed for these keys are viewed below:

$$P_H[24{\sim}31] \oplus C_H[24{\sim}31] = S(P_H[16{\sim}23] \oplus C_H[16{\sim}23], P_H[24{\sim}31] \oplus P_L[24{\sim}31] \oplus K_1[24{\sim}31], 0),$$

$$P_H[0{\sim}7] \oplus C_H[0{\sim}7] = S(P_H[8{\sim}15] \oplus C_H[8{\sim}15], P_H[0{\sim}7] \oplus P_L[0{\sim}7] \oplus K_1[0{\sim}7], 1),$$

Now the only final key we need to derive is k1[16~23]. The reason why we cannot derive this before is that it relies on what k1[8~15] is. But now that we have an appropriate value for this, we can use it to find a candidate key for k1[16~23]. Again this is only a 256 guess operation so it will bring our total number of operations to 2^10 or 1024. The equation for this derivation is below:

$$P_H[16{\sim}23] \oplus C_H[16{\sim}23] = S(P_H[8{\sim}15] \oplus P_L[8{\sim}15] \oplus P_L[0{\sim}7] \oplus P_H[0{\sim}7] \oplus K_1[8{\sim}15], P_H[16{\sim}23] \oplus P_L[16{\sim}23] \oplus P_H[24{\sim}31] \oplus P_L[24{\sim}31] \oplus K_1[16{\sim}23])$$

After this we have a valid key for a given plaintext and ciphertext pair which means we have effectively broken the FEAL cipher by doing roughly 1024 operations. The only problem with this is that unfortunately we cannot derive a generic k1 for all plaintext and ciphertext pairs as it will not be the same in all cases. We believe that the Linear Attack Described in [2] was able to do this because they only attacked FEAL-4 through FEAL-8. These extra rounds cause for more mixing of the data and as such the allow for generic keys which will work for all cases. It is important to know that the reference paper was very vague as to how to accurately modify the algorithm to be able to perform an attack on the cipher.

To remedy this, we do have some suggestions. First when we tried multiple plaintext and ciphertext pairs, we noticed that certain key bytes occurred much more frequently than others. Because of this, we can figure out a relatively small number of candidate keys (between 20 and 50 for each byte of the key for a given original key for the FEAL cipher), our search to find a key which will give the correct output of the original FEAL cipher is still much smaller than 2^32 or even the 2^64 to derive the original key, as it will be in our observed worse case of 50^4. We should also note that here we still are unsure if this will work with 100% accuracy, but based on our observations, this seems to work pretty well. We should also note that the number of candidate keys slightly depends on the number of plaintext and ciphertext pairs. As this number increases, there is a higher chance for there to be more candidate keys(which could cause a problem if it gets too large). The good thing about this is that while the number does go up, certain byte values for keys will also increase at the same time. This means that even though we may have more potential keys, we should also see an increase in the frequency of certain keys occurring.

Description of our implementation of an Ad-Hoc Attack on FEAL

Our implementation of our Ad-Hoc attack acts much like we described above. It will take in a number for how many plaintext and ciphertext pairs are requested and a 64-bit key in hexadecimal. Once it has these values, it will set up a FEAL cipher instance with one (1) round and set the key to be the input key. The attack program named FEALAttack.java will then randomly create n plaintexts and encrypt them using the sub-keys generated in the setKey method of FEALCipher.java. Once these pairs are found, the program will first derive the value for k2. Because we know that this value will be constant regardless of our input plaintext and output ciphertext, we can easily find this. Once we have k2 generated, we proceed to try to find valid key bytes for k1 using the described attack above. Once we have candidate keys for this given plaintext and ciphertext pair, we check to see if we have already seen a given byte value from another plaintext-ciphertext pair and if we have not we add it to an ArrayList. We then check to see if we have seen the entire k1 value or not and add this into an ArrayList as well if unseen. We then proceed on to test the rest of the generated plaintext-ciphertext pairs and record any new unseen candidate keys. Once we are done with all of this, we sort the ArrayLists for each candidate key list and then output the size and the candidate keys. There is also some other information printed out before all of this regarding the input key and its generated 16-bit subkeys as well as a couple plaintext ciphertext pairs generated by the function.

Results from our implementation of an Ad-Hoc Attack on FEAL

We ran a couple of tests on our Ad-Hoc implementation of FEAL to test to see if this is a valid way to attack the 1-round cipher. These tests gave us some very interesting results which we did not expect. Originally we thought that there would be 1 value for k1 which would work in all cases. This was quickly disproved as we could easily find two different sets of plaintexts and ciphertexts where our k1 was not the same. Even though we could not find a key which would work for all cases, the results of our implementation show that there are a common set of byte values which occur quite frequently. However the number of these values is dependent on which original key is used for the encryption as certain original keys will have more or less candidates for each byte of k1. Although this could be an issue if the number of candidate keys gets too large (as we need to multiply the counts to see how many possibilities must be tried for new plaintexts), after a certain number of plaintext and ciphertext pairs we do not add any new candidate keys. The only difference is that we could add new full 32-bit numbers for k1 which are made up of combinations of candidate keys for each byte position. Below is a set of sample original keys with the number of candidate keys derived for each byte of k1 as well as the number of total operations to search for a usable key for any plaintext:

Original Key	#K1_1	#K1_2	#k1_3	#k1_4	Total Operations
0123456789ABCDEF	54	45	27	60	3936600
AAAAAAAAAAAAAAAA	80	79	24	72	10920960
FEDCBA9876543210	14	46	40	4	103040
5738290BC37D3FEA	80	84	48	56	18063360

Analysis of our implementation of an Ad-Hoc Attack on FEAL

Our implementations seemed to do pretty well in finding a fairly small number of potential candidate keys for each byte of k1. In all our experiments, the highest number of potential keys for a given original key was around 80. In most cases if this high of a number was experienced, the other three bytes of the k1 generally had less candidates. Even so to find a valid k1 which would work for a specific pair of plaintext and ciphertext messages in a worse case scenario would be around 80^4 which is still substantially less than brute forcing the search of k1 or even brute forcing the search of the original 64-bit key. We also checked to see how fast the attack will converge onto a set of valid candidate keys given a particular original key. We noticed that somewhere between 1000 and 10,000 plaintext-ciphertext pairs, the attack would no longer find any new candidate keys for the bytes of k1. It is also interesting to note that even with 1,000 plaintext-ciphertext pairs, the attack will find the majority of the candidate bytes(usually only 1 or 2 less) which means that if an attacker cannot afford the time for using 10,000 pairs, they may be able to break the code within 1,000 pairs even though it is not guaranteed to work. To figure out the worst case number of encryptions we will need , lets assume that it will take 10,000 plaintext and ciphertext pairs to break the modified FEAL. Because the attack takes 2^10 operations to derive a specific candidate key for all bytes of the k1, the complexity will become 10,000(2^10) because we must do this each time. Then to deduce which key to use given a new plaintext and ciphertext pair, we need to add in another 80^4 operations to figure out which permutation of candidate keys will work. This means that the entire process is roughly 10,000(2^10) + 80^4 which is about 10240000 + 40960000 = 51,200,000 in the worst observed case. We should also note that this could be a little bit higher or substantially lower depending on how many candidate keys are derived. It should be noted that this is less than 2^26 operations total so it is a significant improvement over a brute force of 32 or 64 bit keys(k1 and original key respectively).

Analysis of Other Attacks on FEAL

In our study of different attacks on the FEAL cipher, we came across two different techniques which crack the FEAL cipher in different ways.

The first technique is the Linear attack described in the paper *A New Method for Known Plaintext Attack of FEAL Cipher* [2]. This was the original attack we wanted to try to implement. We were unable to figure this one out for the reduced rounds(they started with FEAL 4) and as such we decided to borrow some ideas and see if we can apply them to the 1 round FEAL. This paper was characterized by being one of the first Linear Attacks on any cipher(The authors would later go on to use this to break DES). The main idea for this paper was to trace certain pathways through the modified F Function to try to find linear combinations of particular bits from the plaintext and the ciphertext. By doing this they could reduce the number of searches through the key space as they now knew what some bits were. Unfortunately they do not go into much detail as to how many operations are needed to guess bits of each key, but they do say that FEAL 4 could be broken within 6 minutes by using only 5 plaintexts. They also could break FEAL 6 with 100 texts and FEAL 8 with 2^15 plaintexts. These results are much better than ours as we could only break FEAL 1 and we needed around 10,000 plaintexts to do it. What is unclear is that we do not know exactly how many operations needed to be done to derive each key. Because they know that you need at least 4(2^12) to derive K1, they do not go into detail as to how to get the rest of the keys. We do not know if they can find these keys within the same amount of complexity or not. Despite this, the Linear Attack of FEAL 4 was still substantially better than our attack as we do not know exactly how large the candidate key space can grow depending on which original key is used.

The second attack we studied was a differential attack proposed in the paper *Differential Cryptanalysis of Feal and N-Hash* by Eli Biham and Adi Shamir[4]. In this paper, the authors describe an attack where they try to deduce the inputs and outputs of the middle two S Functions in the modified F Function. By computing all possible inputs and outputs, the authors look to find similarities between different pairs by looking at their particular differentials. Once they find a differential which will either always have similar inputs and outputs, they proceed to attack each key systematically to deduce the subkeys. This approach differs from the Linear Attack and ours because it actually finds the subkeys generated from the key schedule of FEAL and not keys which represent the original subkeys and the various operations performed on them. This study was able to calculate the key for FEAL 8 with 1,000 pairs with over a 95% success rate. If the authors doubled the number of plaintext-ciphertext pairs, they improved the accuracy to almost 100%. It should also be noted that this attack is very fast as well as they report the differential based attack can crack it in less than 2 minutes. This attack is also much better than ours. This one is able to crack a much more complicated problem(8 rounds instead of 1) in substantially less time with a smaller number of needed pairs. It is interesting that this approach is not guaranteed to work. Because this attack is extremely likely to break the cipher, we realized that we do not need to have something that works exactly 100% of the time. We believe that we have come very close as after 10,000 plaintext-ciphertext pairs we do not add any new candidate keys(we even tried up to a million and still the same candidates were retrieved).

Developer's Manual

The code deliverables for this project must be compiled in order to be used. To do this, it is necessary to have Java Developer Kit (JDK) installed on the system as well as the Parallel Java Library from the RIT CS department [3]. It is also necessary to modify the classpath variable to include the Parallel Java Jar file. Because we used the RIT CS lab computers to develop and run our software, the easiest way to run these programs is to use these systems. If you are not using these computers, the following commands will not execute properly.

First, we need to set up the classpath to include the parallel java library[3]. To do this on a lab computer just type in the following command:

```
export CLASSPATH=.:/home/fac/ark/public_html/pj.jar
```

Once this is set up, we can compile all of the programs with the following commands:

```
javac BlockCipher.java
javac FEALCipher.java
javac FEALAttack.java
javac FEALTest.java.
```

The BlockCipher.java class is the required interface from the project webpage. FEALCipher.java is the class which does the key generation and encryption process which implements the BlockCipher interface. FEALAttack.java is our implementation of our Ad-hoc attack on the one (1) round FEAL cipher. FEALTest is a test class which will allow for easy encryption by allowing the user to input a number of rounds, a message and a key and it will print out a ciphertext.

To develop any other software using our implementation of the FEAL cipher, one must follow these steps or else it will not work properly. First a FEALCipher object must be made. Then using this object, the setRounds(int R) method needs to be called to set the appropriate number of rounds(It defaults to 8). Once the number of rounds is set, you must set the key by calling the setKey(byte[] key) method. This will generate the appropriate number of subkeys needed for the encryption process. Once the key is set, you can encrypt the message by calling the encrypt(byte[] text) method which will overwrite the text array with the encrypted message. This is all that is needed to do to implement the FEAL Cipher into your own code.

User's Manual

To use this set of programs, you must first compile the programs as described above in the Developers manual. Once the programs are compiled, they can be run easily with the following commands. Because BlockCipher.java is an interface, you cannot run this at all. Also, the FEALCipher.java is not executable as it does not have a main method. To run an instance of the FEALCipher, please use the FEALTest.java class. This class can be executed using the following command:

```
java FEALTest <Plaintext> <key> <#Rounds>
```

where <Plaintext> is the 64-bit input plaintext in hexadecimal, <key> is the original 64-bit key in hexadecimal used for subkey generation and <#Rounds> is the number of Feistel rounds requested by the user. This program will output the encrypted message as well as remind the user what the input parameters were.

To implement the attack FEALAttack.java, one must use this command:

```
java FEALAttack <number of plaintexts> <Key In Hex>
```

Where <number of plaintexts> is the requested number of randomized plaintext and ciphertext pairs used for the key derivation. <Key In Hex> is the original input 64-bit key used to encrypt the random plaintext messages. This program will output the derived value for k2 as well as the candidate keys for each byte of k1 as well as the number of bytes needed to cover each input case. The outputs are further described in the implementation part of this paper above.

Lessons learned from the project

After this project we learned a couple of things about Cryptography. First, algorithms can be relatively easy to implement and get working properly, but can be fairly difficult to break given certain circumstances. Doing cryptanalysis is quite difficult for a lot of cases and even cryptanalysis on relatively unsecure ciphers can be quite difficult if you do not completely understand all that is going on in the cipher. Second, we learned that even though a certain technique might not be able to be applied very well to a concrete circumstance, one can often try some ideas inspired by other work to figure out a way to break the code differently. Because one approach will not solve all of the problems, cryptographers carrying out cryptanalysis will often need to adapt to their current situation. Third, we learned that even though a solution might not be a complete solve-all solution, sometimes a set of solutions can be just as valuable. For instance we cannot generate a k1 value which will apply for all possible plaintext and ciphertext pairs. We can however find a list of candidate keys which occur relatively frequently and try each one of these. Because this list is substantially smaller than the full list of numbers, we can quickly find a valid key for a given plaintext and ciphertext pair. Fourth, we learned that no matter how difficult a problem can seem at times, one should never give up. There were times with this project where we just could not think of any other way to try to fix our issues. But we kept going and we tried different approaches until we found one that worked. We believe that this project allows us to see some of the issues facing cryptanalysis in general and how difficult this can be.

Discussion of Future Work

There are a couple of things we could do in the future. Although this cipher is proven to be broken with as little as 5 known plaintext-ciphertext pairs(for FEAL-4)[1], our approach could be valid if we were able to figure out exactly what the accuracy is for a given key. Because we are currently unsure about how well this will work overall, future work should be devoted to running some statistical measurements to gauge how effective this method is. Because we can only generate a list of keys which should work, we do not know how common is it for a key to appear on the list in comparison to not appearing. If any further work is to be done, this is likely the place to do it.

Another idea is to find another place to add a 3rd key for the attack which could increase the likelihood that a given key would work for more plaintext ciphertext pairs. In other words we would try to efficiently derive a third key which could be put at a new location in the 1-round algorithm which would hopefully stay relatively constant given any plaintext or ciphertext pair. The main thing to be careful of here is to make sure that deriving the third key takes negligible time. This is probably not possible, but it could add some more insight to get more concrete results.

Statement of Group activity

Each group member worked an equal amount on this project. The majority of the coding was done in a group session with Zack coding out the S, F and Fk Functions while Carlos coded out the Key generation methods. Once these steps were done, we both worked on the encryption method together. Once our implementation of the cipher was finished, we both looked into figuring out the attack. Once the idea of our attack was formed, Zack began to code out the derivation functions while Carlos began working on the write up of the paper. Once Zack finished commenting and implementing the code, he began helping Carlos finish up the writing of the documentation. Because of this, we feel as though we both contributed evenly towards the completion of this project. We will also be evenly contributing to the completion of the presentation as well.

References:

[1] Akihiro Shimizu and Shoji Miyaguchi, *Fast Data Encipherment Algorithm FEAL.* Advances in Cryptography *EUROCRYPT* '87. Amsterdam, 1987. pp. 267-278.

[2] Mitsuru Matsui and Atsuhiro Yamagishi. *A New Method for Known Plaintext Attack of FEAL Cipher*. Advances in Cryptography EUROCRYPT '92. Balatonfured, Hungary, 1992. pp. 81-91.

[3] Parallel Java Library: http://www.cs.rit.edu/~ark/pj.shtml

[4] Eli Biham and Adi Shamir *Differential Cryptanalysis of Feal and N-Hash*. Advances in Cryptography *EUROCRYPT '91*. Brighton UK, 1991. pp.1-16.