Tiny Encryption Algorithm (TEA)
Cryptography 4005.705.01
Graduate Team ACD - Final Report

Benjamin Andrews
bla7168@rit.edu

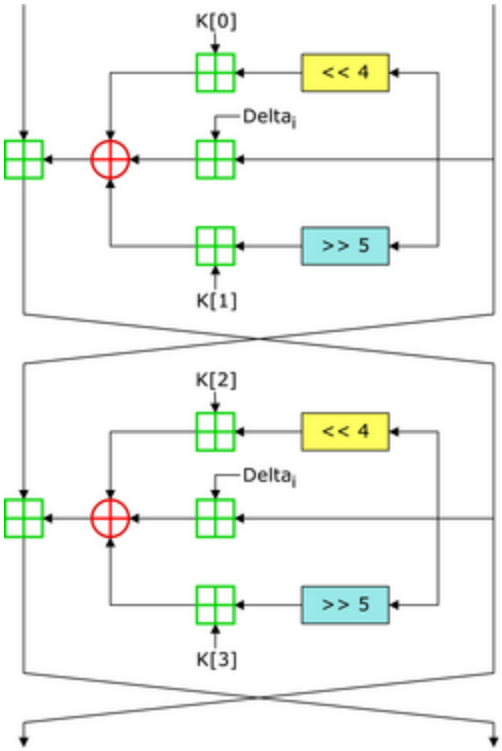
Scott Chapman
sdc3737@rit.edu

Steven Dearstyne
sjd3779@rit.edu

Algorithm Description

**A description of the algorithm(s) of your block cipher, including plenty of diagrams to show how the algorithm(s) work.*

The Tiny Encryption Algorithm (TEA) block cipher was designed with speed and simplicity in mind. It is a variant of the Feistel Cipher. TEA operates on a 64 bit block of data that is then split up into two 32 bit unsigned integers during the encryption process. TEA uses a 128 bit key, and a magic constant is also utilized which is defined as 2^32/(the golden ratio). This quantity looks like 2654435769 when expressed as an integer. Multiples of this constant are used during each round, and its inclusion in the algorithm was to prevent attacks that try to take advantage of symmetry between rounds. The original cipher spec was written by Roger Needham and David Wheeler and was first presented in 1994.[1]



https://upload.wikimedia.org/wikipedia/commons/a/a1/TEA_InfoBox_Diagram.png

The diagram above shows 2 Feistel rounds of TEA. These two Feistal rounds make up one

cycle of TEA. The cipher starts with a 64 bit data block that is split up into two 32 bit blocks which we will call L and R. L is the left side of the block (represented by the arrow in the top left of the diagram), and R is the right side (top right of the diagram). These blocks are swapped per round; this swap can be seen where the two lines intersect in the middle of the diagram.

TEA has a 128 bit key that is split up into four 32 bit subkeys, which can be seen as K[0-3] in the diagram. Delta is defined as a constant, 2^32/(golden ratio), which is 2654435769 as an integer. Multiples of delta are used in each round (mod 2^32).

In the first feistel round R is used as an input to several operations. All addition operations are (mod 2^32).
1. R goes through a left shift of 4 and then is added to K[0]
2. R is added to Delta
3. R goes through a right shift of 5 and then is added to K[1]

An XOR operation is then applied to the result of those three operations and finally, the result of the XOR operation is added to L. This result then becomes R for the next feistel round, because of the swap.

Description of the partial key search attack

**A description of the attack on your block cipher.*

We implemented our attack on 1 Feistal round of TEA. In 1 Feistal round of TEA the key size is effectively reduced to 64 bits, which we call the first 32 bits subkey T and the second 32 bits subkey U. Normally a 128 bit key would be used in 2 or more rounds of TEA. This is because of the Feistal nature of TEA; only half of the 128 bit key is used in the first Feistal round. The remaining 64 bits of the key are used in the 2nd round. Using known plaintext and ciphertext pairs, we will perform a brute force attack on the 64 bit bit key by starting from 0 and incrementing by one until we guess a correct value for subkey T. This means that our brute force search will be of the order 2^32. We can do this because of our ability to calculate subkey U once we guess a value for T.

Given:

$K = (T, U, V, W)$

$K1 = (T, U, Delta1)$

$K2 = (V, W, Delta2)$

Solve for U:

$L1 = R0$

$R1 = L0 + F(R0, K1)$

$R1 = L0 + (((R0 << 4) + T) \text{ xor } ((R0 >> 5) + U) \text{ xor } (R0 + Delta1))$

$R1 - L0 = (((R0 << 4) + T) \text{ xor } ((R0 >> 5) + U) \text{ xor } (R0 + Delta1))$

$(R1 - L0) \text{ xor } ((R0 << 4) + T) = ((R0 >> 5) + U) \text{ xor } (R0 + Delta1))$

$(R1 - L0) \text{ xor } ((R0 << 4) + T) \text{ xor } (R0 + Delta1) = ((R0 >> 5) + U)$

$U = (R1 - L0) \text{ xor } ((R0 << 4) + T) \text{ xor } (R0 + Delta1) - (R0 >> 5)$

Taken from our powerpoint slides from Presentation 1

We are able to calculate U because of our ability to solve the encryption formula for subkey U. In the diagram above you can see the steps required to solve the TEA cipher equation for subkey U. The TEA cipher operations are substituted for F(R0, K1), and after some subtracting and XORing on either side of the equation we are end up with an equation solved for subkey U. Because L0, R0, and R1 are known (from the plaintext/ciphertext file generated using the oracle), and Delta is also known, all we have to do is guess a value for subkey T in order to solve for U. Additionally, if we use a different set of L0, R0, and R1 with the same Delta and the same T, we should get the same U. This “different set” of L0, R0, and R1 is really just a different or 2nd plaintext/ciphertext pair that we need to use. If we test a 2nd plaintext/ciphertext pair and we get a different value for U, that means that our guess for subkey T was incorrect, and we can increment our guess and do the calculation again. Thus, we can continually do this until we find a value for T that gives the same value for U for all of the plaintexts and ciphertexts.

Description of attack program

**A description of your attack program, including the format of the input and output data.*

Our attack program takes one argument which is the name of the file with the plaintext/ciphertext pairs generated by the Oracle program. The plaintext/ciphertext pairs are in the following format:

[64 bit plaintext, 64 bit ciphertext]

```
f4a35085 77366253, 77366253 01930435
49e4ae25 2f8be54c, 2f8be54c a7e02b36
764f51c6 3583a04c, 3583a04c 5b267bff
8af575cd a51a81c8, a51a81c8 f10202be
```

The general idea of the attack program is that we only have to bruteforce 2^32 possible values. This is because given a subkey T, we can calculate a value for subkey U by using known plaintext/ciphertext pairs. Using this same value for T, we can calculate another subkey U by using a different plaintext/ciphertext pair. This is described in more detail in the “Description of the partial key search attack” above.

The following steps/pseudocode outline the general procedure of the attack program.
1. Read in the file with the plaintext/ciphertext pairs into lists. These values are then converted from strings in the file to 32 bit unsigned integers.
2. Guess a value for subkey T, starting at 0.
3. Calculate the value for U using our guess and the first plaintext/ciphertext pair
4. Calculate the value for U using our guess and the second plaintext/ciphertext pair
5. Do those two values of U match?
    6. If not, this is the incorrect guess for T. Increment our guess and go back to Step 3.
    6. If yes, we need to verify that this guess for subkey T is correct
        7. Repeat steps 3 and 4 ten times with different plaintext/ciphertext pairs
        8. If the values of U did not match every time, increment our guess for T and go back to Step 3.
        9. If the values of U matched every time, this is the correct guess for T and we’ve found the key!!

An interesting note about our attack methodology was the number of plaintexts/ciphertexts required for the verifying step (Step 7). We initially had this set at 100 instead of 10, but we decided to reduce it to see if the attack would still succeed. The attack did still succeed, which means that even when two values of U might initially match, they never continued to match after being tested on 20 (10 * 2) additional plaintext/ciphertext pairs.

The output of our attack program can be seen in the Post Attack Analysis section.

The comments in the tea_attack.py file contains more detail about various sections of code and what their purpose is. Please consult this file if you desire more detail.

Post Attack Analysis

** The results of running your attack program to find several different unknown encryption keys.*

Our attack program was successful in finding three different random encryption keys that were generated through the use of our oracle program. Our oracle program selects a random 128 bit

key, and then generates 100 random plaintext/ciphertext pairs using 1 Feistal round of TEA encryption. These plaintext/ciphertext pairs are what the attack program uses to calculate the key. This is what the output of the oracle looks like for one of the keys that we generated.

```
C:\Users\Steven\Documents\My Dropbox\2012-2013\123_cryptography\teamine>python t
ea_oracle.py
Random key: 03b1402c 6adb57ce 7683d59c 31e449d9
Printed 1000 plaintext/ciphertext pairs to plain_cipher_newkey.txt using above k
ey
```

Keep in mind that because we are only attacking 1 Feistal round of TEA, we are only trying to discover the first 64 bits of the key because the other 64 bits of the key are not used in the 1st round of TEA.

Attack #1:
Key: [0xff0f7457, 0x43fd99f7, 0x75f8c48f, 0x2927c18c]

We piped the output of our attack script to a file, “output_equivkeys.txt” (attack1_output.txt in zip file). This is what the contents of the file were after the script completed.

```
ben@mail:~/Dropbox/Crypto/attack$ cat output_equivkeys.txt
Key found! [7f0f7457, c3fd99f7]
Key found! [ff0f7457, 43fd99f7]
7798.93027806 seconds
```

The first thing you will notice is that the script seems to have found two keys. The second key found is the correct 64 bit key that we are looking for, and the first key looks very similar but the first hexadecimal character is different for each of the 32 bits. We were initially confused by this, but then we remembered that the TEA cipher actually has an equivalent key weakness, where each 128 bit key ends up being equivalent to 3 other keys in the keyspace. In our case, we are only brute forcing the 64 bit subkey, so it makes sense that we would find another key in our brute force search that would be equivalent to the key that we initially set.

To prove that these keys are actually equivalent, we can use our TEA cipher implementation.

```
C:\Users\Steven\Documents\My Dropbox\2012-2013\123_cryptography\teamine>python b
en_tea.py 1
Implementation of the Tiny Encryption Algorithm in python
keysize is:             128
blocksize is:           64
number of rounds is:    1
plaintext is:           ['0xffffffffL', '0xffffffffL']
key is:                 ['0x7f0f7457', '0xc3fd99f7L', '0x75f8c48f', '0x2927c18c
']
resulting ciphertext is: ['0x2ac59408', '0xc0d2e6e1']
and decrypted is:       ['0xffffffff', '0xffffffff']

C:\Users\Steven\Documents\My Dropbox\2012-2013\123_cryptography\teamine>python b
en_tea.py 1
Implementation of the Tiny Encryption Algorithm in python
keysize is:             128
blocksize is:           64
number of rounds is:    1
plaintext is:           ['0xffffffffL', '0xffffffffL']
key is:                 ['0xff0f7457L', '0x43fd99f7', '0x75f8c48f', '0x2927c18c
']
resulting ciphertext is: ['0x2ac59408', '0xc0d2e6e1']
and decrypted is:       ['0xffffffff', '0xffffffff']
```

As you can see, the resulting ciphertext is the same using either key that was discovered by our attack program. In this case, we specified a parameter of 1 on the command line for 1 full round of TEA, but the keys are actually equivalent for all rounds, even 32.

```
C:\Users\Steven\Documents\My Dropbox\2012-2013\123_cryptography\teamine>python b
en_tea.py 32
Implementation of the Tiny Encryption Algorithm in python
keysize is:             128
blocksize is:           64
number of rounds is:    32
plaintext is:           ['0xffffffffL', '0xffffffffL']
key is:                 ['0xff0f7457L', '0x43fd99f7', '0x75f8c48f', '0x2927c18c
']
resulting ciphertext is: ['0x90ed725', '0x3931ad9f']
and decrypted is:       ['0xffffffff', '0xffffffff']

C:\Users\Steven\Documents\My Dropbox\2012-2013\123_cryptography\teamine>python b
en_tea.py 32
Implementation of the Tiny Encryption Algorithm in python
keysize is:             128
blocksize is:           64
number of rounds is:    32
plaintext is:           ['0xffffffffL', '0xffffffffL']
key is:                 ['0x7f0f7457', '0xc3fd99f7L', '0x75f8c48f', '0x2927c18c
']
resulting ciphertext is: ['0x90ed725', '0x3931ad9f']
and decrypted is:       ['0xffffffff', '0xffffffff']
```

Given these results for our first run of the attack program, we anticipated that it would find equivalent keys for other keys, and we were correct.

Our attack program also took 7798.93 seconds to run, which is roughly 2.17 hours. Our attack program is brute forcing 2^32 possible values for subkey T, so this is about 550,000 keys per second.

Attack #2:

Key: [0x81592a8f, 0xfcd4e29c, 0x32f0bbf1, 0x28d99069]

Our next attack was also successful, and also found an equivalent key just like in Attack #1. We piped the output of our attack program to “newoutput.txt” (attack2_output.txt in zip file) and these were the contents.

```
ben@mail:~/Dropbox/Crypto/attack$ cat newoutput.txt
Key found! [01592a8f, 7cd4e29c]
Key found! [81592a8f, fcd4e29c]
7663.11188293 seconds
```

This time the attack took 7663 seconds, which is roughly 2.13 hours.

Attack #3:
Key: [0x03b1402c, 0x6adb57ce, 0x7683d59c, 0x31e449d9]

Our third attack was again successful, you can see the output of our attack program below. (attack3_output.txt in zip file)

```
C:\Users\Steven\Documents\My Dropbox\2012-2013\123_cryptography\teamine>python t
ea_attack3.py
Key found! [03b1402c, 6adb57ce]
Key found! [83b1402c, eadb57ce]
20419.2309999 seconds
```

You will notice this time that the attack took significantly longer than Attack #1 and Attack #2. This is because in Attack #1 and Attack #2 we were running the script on Ben’s computer whereas for Attack #3 I decided to see how fast the attack script would run on my laptop. Ben’s computer has a quad-core processor which is clocked at 4.1GHz, whereas my laptop has the old Core 2 Duo processor clocked at 2.4GHz. This difference in CPU speed is the reason for the noticeable speed difference. This time the attack took 20419 seconds which is roughly 5.7 hours. Another factor that could have contributed to the speed difference was that while the attack was running I had my Power plan set to “Balanced” instead of “High performance.”

You can see by the results of the first two attacks, however, that on a faster CPU (~4.1Ghz) our attack program only takes a little over two hours to run. In our opinion, this was a good speed for a brute force of order 2^32.

An analysis of how many encryptions and how many known plaintexts and ciphertexts are required for the attack to succeed.

Technically, our attack could require up to 2^32 guesses in order to succeed. This is because we start our attack from zero, and increment our guess for subkey T after each unsuccessful guess. If the subkey T used to encrypt the plaintext was 0xffffffff, this means that the our program would not test that subkey until 2^32 iterations. So to ENSURE that our attack succeeds, it would require a brute force of magnitude 2^32.

Another important point to note is that although our oracle generates 100 random plaintext/ciphertext pairs, currently our attack is functioning correctly with only 22 random plaintexts/ciphertexts. Most of the heavy lifting is actually done using only 2 plaintext/ciphertext pairs because of the values that we calculate for U don't match, we know that our guess for subkey T is wrong. If we know our guess is wrong we can increment our guess and reuse the same plaintext/ciphertext pairs. If the values of U do match, we test 20 additional plaintext/ciphertext pairs to guarantee that our guess for T is correct. Thus, we really only need 22 plaintext/ciphertext pairs, and our attack program is currently functioning in this manner.

Literature Search Comparison

** An analysis of each item found in your literature search, including a comparison of their results to yours.*

A few different attacks have been published on TEA, including a differential related-key attack [2]. Bruce Schneier and others successfully implemented this differential related-key attack to break TEA with only 2^23 chosen plaintexts. However, TEA has a problem with equivalent keys, which results in the key size being essentially reduced to 126 bits instead of 128 bits. This is due to the fact that every key is actually equal to three other keys [3].

Our method of attack consisted of brute forcing the first round. This is a less complex and less sophisticated method of deriving the keys by just guessing. With each increase in rounds the difficulty and complexity increases significantly.

Our attack program is able to brute force the 64 bit subkey by using only 22 chosen plaintexts. This is significantly less than the 2^23 chosen plaintexts that are needed for the differential related-key attack, but that attack was on full-round TEA instead of just one Feistal round. By doing our attack on only one Feistal round, we only have to worry about 64 bits of the key instead of the full 128 bit key, and the complexity of deducing the key is severely reduced. See the previous analysis section for more detail on why only 22 plaintext/ciphertext pairs are needed.

Developers Manual

**A developer's manual for your software (i.e. exact instructions for how to compile the software).*

All of our scripts do not need to be compiled and can be run simply by typing "python name_of_script.py", where "name_of_script" is replaced with the name of the script you want to run. Additionally, please consult the user manual to see if the scripts require any additional parameters to be run.

User Manual

**A user's manual for your software (i.e. exact instructions for how to run the software, how to use the UI if any, etc.).*

tea_oracle.py
To use our attack program, first you must generate a list of plaintext/ciphertext pairs using the tea_oracle.py script.

Usage: python tea_oracle.py <name_of_output_file>

This script will generate a random 128 bit key, encrypt 100 random plaintexts, and then output the plaintext/ciphertext pairs to a file. You must provide an output filename to the tea_oracle.py script as a parameter.

tea_attack.py
To use our attack program, just supply the filename of the text file that the oracle program created as a parameter to the attack program.

Usage: python tea_attack.py <plaintext/ciphertext_filename>

Cipher Implementation
Our block cipher implementation contains two files. The first file, BlockCipher.py, creates a class BlockCipher and specifies all necessary functions for the block cipher. The second file, tea.py, contains the main function and uses the BlockCipher class to make a new object and perform its accompanying functions.

BlockCipher.py
This file contains the BlockCipher class, and needs to be imported into a file with a main function for use. This can be done by placing the BlockCipher.py file in the same directory as the file with the main function, and by including the following line in the main function file: from BlockCipher import * From this file, you can manipulate the block and key sizes by changing variables blocksize and keysize.

tea.py
This file contains the main function necessary for creating a BlockCipher object and specifying details about the desired operation. The desired number of TEA rounds for the operation is passed as the only parameter to the file. Usage is as follows:
Usage: python tea.py <number of rounds>

The desired plaintext to be used and key are set in this file. These are given as lists with the variable names text and key. The BlockCipher.py file must be included in the same directory as this file for the program to work correctly. This file is included by the line: from BlockCipher import *

Takeaways (All)

**A discussion of what you learned from the project.*

We learned quite a bit from this project. Most importantly, we learned not to look at the output of a program and just because it's not what we expected, assume that it's incorrect. When we initially ran our brute force program against the entire 2^32 keyspace, it returned a key that was not the key we were expecting. However, we came to find out that after a considerable amount of frustration that this was not actually an incorrect key, but an equivalent key. Because the output was not what we expected, we assumed something was wrong instead of assuming that our program was performing correctly.

We also learned that the Tiny Encryption Algorithm is able to perform securely in lightweight environments, especially for low power devices like RFID. Only performing one round leaves TEA open to attack. However the difficulty of breaking TEA's encryption becomes more difficult as the rounds are increased. The multiple rounds performed in the full implementation of TEA leads to its more secure nature.

Futurework (All)
**A discussion of possible future work.*

One thing that could be done in the future would be parallelization or multithreading of the brute force program. This would likely increase the speed of our brute force attack. For example, one thread could start at guess 0 and the other thread at guess 2^32 and eventually meet in the middle. This would likely decrease the time it takes to find the keys.

Another thing we could do is get rid of the getu function entirely in the attack program. Instead of calling the function every time a guess of T needs to be verified, we could put the function code right into our main loop. This may decrease the program execution time by reducing the number of function calls.

Individual Statement
**A statement of what each individual team member did on the project.*

Scott organized group meetings to maintain project deadlines, delegated group tasks, and performed research and writing. Steven worked mainly on the attack program and had some involvement with the block cipher implementation. He also created the Oracle which generates random keys and plaintext/ciphertext pairs. Ben created the implementation of the block cipher, as well as assisted Steven with the attack operations.

References

[1] Wheeler, David, and Roger Needham. "TEA, a tiny encryption algorithm." *Fast Software Encryption*. Springer Berlin/Heidelberg, 1995. http://www.cix.co.uk/~klockstone/tea.pdf
[2] Kelsey, John, Bruce Schneier, and David Wagner. "Related-key cryptanalysis of 3-way,

biham-des, cast, des-x, newdes, rc2, and tea." *Information and Communications Security* (1997): 233-246. https://www.schneier.com/paper-relatedkey.pdf
[3] Kelsey, John, Bruce Schneier, and David Wagner. "Key-schedule cryptanalysis of idea, g-des, gost, safer, and triple-des." Advances in Cryptology—CRYPTO'96. Springer Berlin/Heidelberg, 1996. https://www.schneier.com/paper-key-schedule.pdf

Addendum

Block Cipher Source Code - BlockCipher.py and tea.py

BlockCipher.py

```
#!/usr/bin/python2.7

from ctypes import *
from sys import *

# Class implementation of our TEA block cipher
class BlockCipher():

    # Set block and key sizes according to cipher specification
    blocksize = 64
    keysize = 128

    # Storage for object-specific number of TEA rounds and key
    R = None
    key = None

    # Function to report the block size of the TEA cipher
    def blockSize(self):
            return self.blocksize

    # Function to report the key size of the TEA cipher
    def keySize(self):
            return self.keysize

    # Function to set the number of TEA rounds for the current object
    def setRounds(self, R):
            self.R = R

    # Function to set the key for the current object
```

```
    def setKey(self, key):
        self.key = key

    # Function to encrypt specified plaintext block
    def encrypt(self, text):

        # Split plaintext into halves for input into feistal structure
        sidea = c_uint32(text[0])
        sideb = c_uint32(text[1])

        # Value storage for key schedule
        sum = c_uint32(0)

        # Magic value used for key schedule
        delta = 0x9E3779B9

        # Set the number of TEA rounds for the encryption process
        count = self.R

        # Output storage
        ciphertext = [0,0]

        # Iterate through specified TEA encryption rounds
        while(count > 0):

            # Increment value for key schedule
            sum.value += delta

            # Perform first encryption with a TEA feistal round on one half of the plaintext.
This is the first part of a full TEA cycle, which is two feistal rounds
            sidea.value += ( sideb.value << 4 ) + self.key[0] ^ sideb.value + sum.value ^ (
sideb.value >> 5 ) + self.key[1]
            # Perform second encryption with a TEA feistal round on the other half of the
plaintext.  This completes a full TEA cycle, or two feistal rounds
            sideb.value += ( sidea.value << 4 ) + self.key[2] ^ sidea.value + sum.value ^ (
sidea.value >> 5 ) + self.key[3]

            # Decrement remaining TEA rounds after finishing one
            count -= 1

        # Get the resulting output of the TEA encryption rounds and put it in the output list
        ciphertext[0] = sidea.value
        ciphertext[1] = sideb.value
```

```
        # Return the ciphertext of the encryption process
        return ciphertext

    def decrypt(self, ctext):

        # Split ciphertext into halves for input into feistal structure
        sidea = c_uint32(ctext[0])
        sideb = c_uint32(ctext[1])

        # Value storage for key schedule.  During the decryption process, this is set to the magic value multiplied by the number of TEA decryption rounds
        sum = c_uint32(0x9E3779B9 * self.R)

        # Magic value used for key schedule
        delta = 0x9E3779B9

        # Set the number of TEA rounds for the encryption process
        count = self.R

        # Output storage
        plaintext=[0,0]

        # Iterate through specified TEA decryption rounds
        while (count > 0):

            # Perform first decryption with a TEA feistal round on one half of the ciphertext. This is the first part of a full TEA cycle, which is two feistal rounds
            sideb.value -= ( sidea.value << 4 ) + self.key[2] ^ sidea.value + sum.value ^ ( sidea.value >> 5 ) + self.key[3]
            # Perform second decryption with a TEA feistal round on the other half of the ciphertext.  This completes a full TEA cycle, or two feistal rounds
            sidea.value -= ( sideb.value << 4 ) + self.key[0] ^ sideb.value + sum.value ^ ( sideb.value >> 5 ) + self.key[1]

            # Decrement value for key schedule
            sum.value -= delta

            # Decrement remaining TEA rounds after finishing one
            count -= 1

        # Get the resulting output of the TEA decryption rounds and put it in the output list
        plaintext[0]=sidea.value
```

```
        plaintext[1]=sideb.value

        # Return the plaintext of the decryption process
        return plaintext
```

tea.py

```
#!/usr/bin/python2.7

from ctypes import *
from sys import *
from BlockCipher import *

# Main function to create an instance of the BlockCipher class and utilize provided functions
def main():

   # Check parameters given.  If no rounds given, specify usage and exit
   if (len(argv) != 2):
        print ("Usage: python tea.py <number of rounds>")
        exit(1)

   # Set the plaintext desired for the encryption operation
   text = [0xffffffff, 0xffffffff]

   # Set the key to be used for the encryption and decryption operations
   key = [0xff0f7457, 0x43fd99f7, 0x75f8c48f, 0x2927c18c]

   # Create new BlockCipher object
   test = BlockCipher()

   # Take the specified key and set that as the encryption/decryption key for the current
BlockCipher object
   test.setKey(key)

   # Take the specified number of TEA rounds and set that as the number of TEA rounds for the
current BlockCipher object
   test.setRounds(int(argv[1]))

   # Pass given plaintext block to the BlockCipher encrypt function and store the output
   ciphertext  = test.encrypt(text)
```

```
    # Take ciphertext resulting from encryption operation and perform the decryption function on
the value.  Then store the output
    plaintext = test.decrypt(ciphertext)

    # Print out specifics of the TEA implementation to the user
    print ("Implementation of the Tiny Encryption Algorithm in python")
    print ("keysize is:     %d" % test.keySize())
    print ("blocksize is:           %d" % test.blockSize())

    # Output the specified number of TEA rounds
    print ("number of rounds is:      %d" % test.R)

    # Output the plaintext specified for the encryption process
    print ("plaintext is:           " + str([hex(x) for x in text]))

    # Output the key specified for the encryption and decryption processes
    print ("key is:         " + str([hex(x) for x in test.key]))

    # Output the resulting ciphertext from the encryption function
    print ("resulting ciphertext is: " + str([hex(x).rstrip("L") for x in ciphertext]))

    # Output the resulting plaintext from the decryption function performed on the ciphertext
    print ("and decrypted is:      " + str([hex(x).rstrip("L") for x in plaintext]))

# Run the main function
if __name__ == "__main__":
    main()
```