Tiny Encryption Algorithm

Team Garrett - Cryptography Submission

Jessie Grabowski, Jeff Keurian

5/9/2010

I. High Level Description of TEA

The Tiny Encryption Algorithm (TEA) was first published in 1994 by Roger Needham, and David Wheeler from Cambridge University of the United Kingdom. TEA was initially designed to be an extremely small algorithm when implemented in terms of the memory foot print required to store the algorithm. This was accomplished by making the basic operations very simple and weak; security is achieved by repeating these simple operations many times. As the basic operations are very simple TEA is also regarded as a very high speed encryption algorithm. These properties have made TEA a choice for both weak hardware or software encryption implementations in the past as TEA can be operated in all modes as specified by DES as outlined in the specification.

There are many notable fallbacks of TEA and it is considered broken. The first issue of note is that TEA uses “equivalent keys” thus weakening the effectiveness of its key length and requires only complexity O(2^32) using a related key attack to break. This is much less than the intended key brute force strength of 2^128. Two revisions of TEA have since been published including XTEA and XXTEA which boast enhanced security and the ability to support arbitrary block sizes making TEA obsolete as a secure cryptographic method.

The specification for TEA states a 128-bit key is to be divided into four 32-bit key words and the block size of each encryption is 64 bits, of which is to be divided into two 32-bit words. TEA utilizes a Feistel scheme for its encryption rounds in which 1 round of TEA includes 2 Feistel operations and a number of additions and bitwise XOR operations as shown below in figure 1.

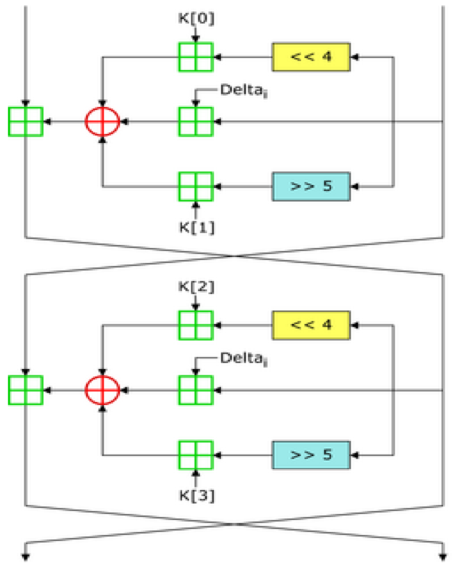

Figure 1 – Basic TEA Round

The specification simply “suggests” that 32 TEA rounds be completed for each 64-bit block encrypted, all online resources appear to follow this suggestion. This means a full encryption of a block is simply 32 TEA rounds which involves 64 Fiestel rounds.

TEA utilizes a value denoted as DELTA in the specification which is defined as $(\sqrt{5} - 1) * 2^{31}$ which is "derived from the golden ratio" is used in multiples for each round to prevent symmetry based exploits on the Feistel operations as shown in figure 1.

The key schedule simply exists as exclusive OR'ing the key words with a shifted value of the last state of each of the block words, this operation "causes all bits of the key and data to be mixed repeatedly".

A Decryption simply involves the inverse operations in reserve order; that is 32 rounds of subtract and XOR operations in the opposite order of the encryption. TEA is considered a high speed algorithm as there is virtually no set up or complex key schedule for the encryption and decryption algorithm.

A second serious shortcoming of the TEA algorithm is the lack of official test vectors in the specification and proof as to why the operations were chosen and considered secure other than stating a number of other algorithms were tested. The specification also does not explicitly state bit ordering.

II. Input and Output Examples of the Primitive

As previously stated the TEA block size is 64 bits the following values were used as test vectors with a key of 0 for all encryptions.

Encryption:

Input	Output
Plaintext (0x)	Ciphertext (0x)
00000000 00000000	4e4e642a 43de3739
41ea3a0a 94baa940	b9354a86 1ea75492
b9354a86 1ea75492	1dbae8aa ae2bba9a
1dbae8aa ae2bba9a	0eb60bc9 0296522d

Decryption:

Input	Output
Ciphertext (0x)	Plaintext (0x)
0eb60bc9 0296522d	1dbae8aa ae2bba9a
1dbae8aa ae2bba9a	b9354a86 1ea75492
b9354a86 1ea75492	41ea3a0a 94baa940
41ea3a0a 94baa940	00000000 00000000

As the Decryption Values matched the encryption values after multiple encryptions it was deemed that the implementation worked. Additional test cases are provided with this report submission.

III. Original Software Design

The original design of the software involved implementing TEA directly from the specification while including potential “naïve mistakes” which would cause performance degradation such as leaving the key values in an array. The algorithm was chosen to be implemented in Java and the program was designed to simply implement the primitive and not one of its modes of operations. The program uses static methods to execute encryption and decryption operations so no object creation would be necessary if a different main routine was desired by a developer solely using the functions.

The encryption and decryption operations were performed on the provided ciphertext or plain text arrays that were provided as method arguments; that is to say each round of the encryption accessed ciphertext[0] and ciphertext[1]. The value delta outline in the specification as $\left(\sqrt{5}-1\right) * 2^{31}$ was calculated each time the static encryption or decryption methods were called and the 128 bit key was stored as two 64 bit words in an array. This array was also accessed each time the key was needed; this introduced the need to shift and mask the lower bits of the key when the upper 32 bits were needed and mask the upper bits when the lower 32 bits were needed.

The inputs to the primitive’s program were designed to execute one encryption, or repeat the same encryption for timing analysis multiple times these inputs include:

Mode – Encrypt or decrypt the data
Key0 - First (Most significant) 64 bits of the key in hex
Key1 - Second (least significant) 64 bits of the key in hex
Block0 – First 32 bits of the block to encrypt or decrypt in hex
Block1 – Second 32 bits of the block to encrypt or decrypt in hex
Iterations – The number of times to repeat the encryption

The output is simply the 64 bits on enciphered or deciphered data based on the mode of operation.

See TEA.java for the original source code.

IV. Time Measurements and Profiling information

Timing and profiling analysis was completed using the following input parameters:

java -Xint -Xprof -agentlib:hprof=cpu=samples,depth=10 TEA e 0 0 0 0 5000000

meaning the input test vector included a key which was all zeros and a block which was all zeros. The mode of operation was encryption and 5 million iterations were performed. The high number of iterations were required as TEA is innately a fast encryption algorithm and in order to eliminate the affect of overhead from program startup and non encryption related instructions on execution time.

As shown by the line used to execute the analysis the JIT complier was disabled to ensure the byte code was interpreted precisely as it was written.

Only the encryption method was used to analyze timing as the decryption method was designed and implemented using the same software techniques and style; thus any insight gained on program timing would apply to both modes of operation.

The following is the timing profile for TEA.java The final running time was found as 237.29 seconds on a modern laptop with JRE 1.6.0_16

```
---------------------------------------------------------------------

Flat profile of 237.29 secs (11391 total ticks): main

  Interpreted + native   Method
 20.1%  2288  +     0    java.lang.String.toCharArray
 15.0%  1706  +     0    java.math.BigInteger.<init>
  9.5%  1078  +     0    TEA.doEncrypt
  8.6%   980  +     0    java.util.Arrays.copyOfRange
  8.3%   944  +     0    java.math.MutableBigInteger.divideMagnitude
  5.5%   625  +     0    java.math.BigDecimal.<init>
  4.6%   519  +     0    java.math.MutableBigInteger.<init>
  3.1%     0  +   353    java.lang.System.arraycopy
  2.8%   321  +     0    java.lang.CharacterDataLatin1.digit
  2.2%   253  +     0    java.math.BigInteger.multiply
  1.9%   220  +     0    java.lang.Character.digit
  1.5%   173  +     0    java.math.MutableBigInteger.divide
  1.2%   138  +     0    java.math.MutableBigInteger.primitiveLeftShift
  1.2%   134  +     0    java.lang.CharacterDataLatin1.getProperties
  1.1%   129  +     0    java.math.BigDecimal.divideAndRound
  1.1%   121  +     0    java.math.BigDecimal.multiply
  1.0%   119  +     0    java.lang.Character.digit
  1.0%   115  +     0    java.math.MutableBigInteger.normalize
  0.8%    88  +     0    java.math.MutableBigInteger.mulsub
  0.7%    79  +     0    java.math.BigDecimal.setScale
  0.6%    68  +     0    java.lang.Integer.numberOfLeadingZeros
  0.6%    67  +     0    java.math.BigInteger.primitiveLeftShift
  0.5%    61  +     0    java.math.MutableBigInteger.rightShift
  0.5%    57  +     0    java.math.BigInteger.valueOf
  0.5%    54  +     0    java.math.MutableBigInteger.toBigDecimal
 99.9% 11027  +   354    Total interpreted (including elided)

  Thread-local ticks:
  0.1%    10             Class loader


Flat profile of 0.00 secs (1 total ticks): DestroyJavaVM

  Thread-local ticks:
100.0%     1             Blocked (of total)


Global summary of 237.29 seconds:
100.0% 12841             Received ticks
  0.0%     6             Received GC ticks
 11.8%  1516             Other VM operations
  0.1%    10             Class loader
Dumping CPU usage by sampling running threads ... done.
```

V. Timing Analysis

From reviewing the above profiling information it is immediately evident that the program is spending the most time in the java library functions for toCharArray() and BigInteger functions. Upon review of the code it can be seen that this is due to the calculation of DELTA upon each encryption cycle. As there is only one function used for encryption due to the algorithms small footprint it is also apparent that only 10% of the ticks are actually within the encryption function; thus finding a method to remove the calculations of delta all together will produce an expected speedup of approximately 10 due to Amdhal's law where speedup = 1/[(1-P)+P/S] where P is the percent of code that is altered and S is the speedup to that code. Assuming P = .9 and S = infinity due to completely removing the code, Speedup = 1 / .1 = 10 from just removing this delta operation and its subsequent java library calls.

It is noted that to complete 5 million encryptions at a total of 237.29 seconds a single TEA encryption with this implementation takes about 47.46us.

Additional changes that would reduce running time include unrolling the loop of the round operations and caching the values used in each round operation including the key and current cihpertext state. This is expected to greatly improve performance as memory accesses to arrays are slow if they are required to access main memory or the hard disk due to paging; secondly Java's native exception checking on array accesses increases overhead even further. Also by caching the key values multiple redundant shifts and masks are removed from each round operation.

VI. Revised Software Design

The first change made to the software design was changing the methods from a static implementation to member methods of an object. Additionally with this change the value of delta was pre-computed and stored as a private constant for the class and upon object construction each of the key words was stored as state variables.

The encryption and decryption methods themselves were changed by manually unrolling the round loop four times. This meant the instructions inside the loop were repeated four times and the bounds of the loop reduced by a factor of four.

This operation is an effort to both increase the basic block size of the code, should compiling optimizations be used giving the complier more low level instructions to re order (if supported by Java) to reduce pipeline stalls, and also reduce the number of branches that may be predicted incorrectly (if supported by Java) or cause pipeline stalls due to looping.

Local variables were also created in the methods in order to prevent accesses to the ciphertext or plain array with the exception of reading the initial data and writing the final data.

After a number of experiments with small program changes, the while loops in the encryption and decryption methods were replaced with a for loop which appeared to have minor, but consistent, improvements on execution time.

See TEA_optimized.java for the revised source code.

VII. Revised Timing Measurements and Analysis

The same parameters used for testing the original program were used for timing analysis of the revised algorithm as shown below again with the JIT complier disabled.

```
----------------------------------------------------------------------
Flat profile of 21.44 secs (1734 total ticks): main

  Interpreted + native   Method
 98.7%  1712  +     0    TEA_optimized.doEncryptRound
  1.2%    21  +     0    TEA_optimized.main
  0.1%     0  +     1    java.io.WinNTFileSystem.getBooleanAttributes
100.0%  1733  +     1    Total interpreted


Flat profile of 0.00 secs (1 total ticks): DestroyJavaVM

  Thread-local ticks:
100.0%     1             Blocked (of total)
```

```
Global summary of 21.44 seconds:
100.0%  1747             Received ticks
  0.7%    12             Other VM operations
Dumping CPU usage by sampling running threads ... done.
```

It can be seen immediately that there no longer exists any calls to a Java library functions thus the removal of delta calculations provided a significant and expected improvement. The overall speedup compared to the original software is found as 11.067 and a single encryption now takes approximately 4.28us.

Various degrees of loop unrolling were experimented with and it was concluded that, with the above profiling information, four iterations of unrolling was ideal and unrolling provided only a slight benefit in execution time. This may be due to the fact that all instructions in the loop create data dependencies on the previous instruction and all iterations on the loop are iteration dependent thus unrolling the loop and eliminating branching affects is minimal in comparison to the delay required to finish the previous instructions.

Also once the loop is unrolled by a high amount the TEA loses one of its main benefits of a small memory footprint.

It was also determined that local variables improved performance over state variables in terms of the round iteration storage space y and z. When these variables were made state variables of the object the software ran consistently slower when compared to the variables being initialized each time in the encryption and decryption functions.

Finally, by removing the accesses to arrays for each of the 32 bits of key data a significant speedup was noticed.

VIII. Developer’s manual

The provided program utilizes the CS department library to parse hex values from the command line arguments thus the library should be added to the classpath before compilation by running the following bash command:

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

To compile the program simply navigate to the directory the source code is located and run the command:

```
 javac TEA_optimized.java
```

or for the original version:

```
javac TEA.java
```

IX. User's manual

To use the program to encrypt or decrypt a block of 64 bits:

Determine the 128 bit key in hex.
Determine 64 bit block to encode or decode in hex.

Run the program by using the command with parameters:

```
java TEA_optimized  [mode] [key0] [key1] [data1] [data0] <iterations>
```

where the parameters used are:

REQUIRED:
mode – "e" to encrypt "d" to decrypt
key0 – first 64 bits of the key
key1 – second 64 bits of the key
data1 – first 32 bits of the data block
data2 – second 32 bits of the block

OPTIONAL:
Iterations – number of times to repeat the encryption (does not alter output)

X. Lessons Learned

One of the most valuable concepts learned through this project is that profiling should be done without complier optimizations enabled otherwise the true underlying cause of a slowdown in the algorithm or implementation technique may be difficult or impossible to find. Also, attempting to optimize a fairly simple and small block of code is a difficult task as the smaller the operation grain size gets the more difficult it is to improve execution time; with an algorithm that consists of just small operations optimization is a difficult task.

Additionally, loop unrolling is an invaluable tool, while compliers will often perform this operation on loops with static bounds transparently, understanding why and how this optimization is critical for modern architectures is important and can allow for software to be designed more effectively with performance in mind. However there can be a tradeoff as unrolling a loop with a large number of instructions has the potential to make code hard to read for programmers as well as increase the memory footprint as more instructions are generated for the program. This may be counterproductive to a main goal of the program to begin with as is the case with TEA.

Finally with respect to Java implementations and objects, state variables can potentially be slower to access in comparison to local variables when many repetitive reads and writes are being performed on them by multiple instructions.

XI. Possible Future Work

File I/O options to the program in order to increase ease of use would be a useful feature to add to the program in the future such that multiple blocks of data could be encrypted in ECB mode. Extending upon that concept as TEA can be operated in any mode DES supports thus adding the multiple modes of operation, specifically CFB, would make the program much more useful. Alternatively as TEA is obsolete and XXTEA supports variable length blocks an implementation of the up-to-date version of the specification would be better worth the effort.

TEA would be interesting to implement using a hardware description language and placed on an FPGA platform; or at the very least simulated in an HDL in order to determine the maximum speed of the algorithm as it was originally intended.

XII. Team Contributions

Both team members Jessie and Jeff collaborated on the original design architecture as well as the improvements to the implementations in Java as well as testing the software.

Jessie is mostly responsible for the report while Jeff was responsible for revising the report. Finally Jeff was mostly responsible for the presentation while Jessie revised it.

XIII. References

TEA Specification - http://www.cix.co.uk/~klockstone/tea.pdf
TEA Wikipedia Entry - http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm
TEA and XTEA Test Vectors - http://www.cix.co.uk/~klockstone/teavect.htm
Amdhal's Law - http://en.wikipedia.org/wiki/Amdahl%27s_law