The Tiny Encryption Algorithm (TEA)

Derek Williams
CPSC 6128 – Network Security
Columbus State University
April 26, 2008

1. ABSTRACT

The Tiny Encryption Algorithm (TEA) was designed by Wheeler and Needham to be indeed "tiny" (small and simple), yet fast and cryptographically strong. In my research and experiments, I sought to gain first-hand experience to assess its relative simplicity, performance, and effectiveness. This paper reports my findings, affirms the inventors' claims, identifies problems with incorrect implementations and cryptanalysis, and recommends some solutions.

2. INTRODUCTION

The Tiny Encryption Algorithm (TEA) is a symmetric (private) key encryption algorithm created by David Wheeler and Roger Needham of Cambridge University and published in 1994. It was designed for simplicity and performance, while seeking an encryption strength on par with more complicated and resource-intensive algorithms such as DES (Data Encryption Standard). Wheeler and Needham summarize this as follows: "it is hoped that it can easily be translated into most languages in a compatible way… it uses little set up time and does… enough rounds to make it secure… it can replace DES in software, and is short enough to write into almost any program on any computer." [6]

From academic research, one can learn of TEA's relative merits. My work sought to go further and gain a first-hand understanding of TEA's simplicity (ease of implementation), performance, and effectiveness (cryptographic strength). Through experiments, I sought to answer the following questions:

- *Simplicity:* Is TEA easy to implement in a variety of languages, including both traditional third-generation languages and newer dynamic scripting languages?
- *Performance:* Is TEA's design efficient? How does its runtime speed compare with other similar block encryption algorithms?
- *Strength:* Is TEA a cryptographically strong algorithm? Are certain published TEA "weaknesses" significant?

Gaining the experience to answer these questions required writing code, running tests and measurements, and doing some very basic cryptanalysis. The sections below describe this work, along with my results and conclusions.

3. DISCUSSION

With published goals, design, and implementation of the TEA algorithm in mind, I wrote code to literally put its simplicity, performance, and effectiveness to the test. I implemented both the original TEA and modified TEA (XTEA) algorithms; see below for further information on XTEA.

3.1 Simplicity

TEA's simplicity is easily demonstrated: the encryption and decryption algorithms are represented in only seven lines of C code each; see *Figure 1* below for the original C encryption function, as published by the authors (the decryption function is given in *Figure 2* of the Appendix). The algorithm requires very little memory and, unlike other block ciphers of similar strength, does not require tables for P-boxes or S-boxes (permutation or substitution lookups). Instead, it uses multiple passes, opting for "a large number of iterations rather than a complicated program."[6] The authors recommend 32 passes (64 rounds), but note that just six cycles will provide excellent diffusion and suggest that 16 sixteen passes may be used where performance is critical.

```
void code(long* v, long* k)  {
      unsigned long y=v[0],z=v[1], sum=0,     /* set up */
                   delta=0x9e3779b9,   /* a key schedule constant */
                   n=32 ;

      while (n-->0) {                         /* basic cycle start */
            sum += delta ;
            y += ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]) ;
            z += ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]) ;
        }                                  /* end cycle */
      v[0]=y ; v[1]=z ; }
```

Figure 1 - TEA Encryption Function

TEA has been implemented in many languages, including C, C++, C#, Forth, Java, JavaScript, Macromedia Flash, Perl, PHP, Python, Ruby, SQL Server, Tcl, and a variety of assembly languages. It has been deployed on many modern server and personal computer platforms, and in embedded systems and small devices, such as Palm Pilots, cell phones, and Microsoft's Xbox.

To assess the simplicity of the algorithm, I wrote my own implementations of TEA (and XTEA) in C, Java, Smalltalk, and Ruby. The choice of these languages was deliberate: to claim that the algorithm is simple and easy to implement, we should attempt it in very different programming languages. Ease of implementation in one language does not always equate to ease in a very different language.

I chose C as the reference implementation, since that is what the authors published and could be used for verification. I chose Java for its popularity and for its access to other industrial strength encryption/decryption algorithms (for performance comparison). I chose Smalltalk and Ruby to assess the relative ease of implementation and performance in dynamic interpreted languages that lack built-in bitwise operators. I wrote accompanying test drivers in each implementation language to verify correctness and measure performance.

I found implementing and measuring TEA to be straightforward, particularly in C and Java. The C implementation was quick: I simply pasted in the published algorithm and added test drivers around it. Java was almost as easy; I was able to use the core of the algorithm directly, with a minor edit: I changed the C right shift operators (>>) to the Java unsigned right shift operators (>>>), to avoid undesired sign extension.

In all implementations, I had to write code to move byte data to and from 32-bit words. This varied more than the TEA algorithm itself across the different languages. I am curious as to the effect of different byte

orders (little endian vs. big endian) on the algorithm's word-wise processing, but I did all my work under an Intel (little endian) architecture. All performance and strength measurements were done with eight-byte plaintexts, to avoid the effects of padding and block chaining.

The Smalltalk implementation required a few iterations to get right. The Smalltalk language does not have built-in bitwise operations (primitives), rather bitXor: and bitShift: messages sent to Integers. Further, Smalltalk provides large numbers that automatically extend ("grow"), so it was necessary to "mask off" the high order bytes of intermediate results. Other minor language differences affected the translation; for example, Smalltalk array indexes are 1-based vs. C's 0-based offsets. Yet I deliberately worked to make the Smalltalk implementation "look" like the C implementation (and a little less object-oriented), so that it was easier to compare to the published C reference implementation.

I did the Ruby implementation last and found it easy to code, for three reasons: 1) I was able to closely match the "look" and syntax of the C reference implementation, 2) I applied the lessons learned from my Smalltalk implementation about masking off high-order overflow bits, and 3) as an interpreted language, Ruby's coding and testing cycle is very quick and iterative. After initial coding, I added a couple of common optimizations, such as pulling key indexing out of the loop, to improve performance.

For each implementation, I wrote test cases (typically with xUnit) to verify that encryption and decryption results were correct.

3.2 Performance

TEA employs a Feistel network [1] (a symmetric block cipher) that uses a combination of bit shifting, XOR, and add operations to create the necessary diffusion and confusion of data. It does these operations on 32 bit words rather than single bytes, a very important optimization that the authors note avoids "wasting the power of a computer." It uses a 128 bit (4 word) key, mixing in its individual word components in an effective key schedule. The original implementation operates on 64 bits (two words) of data at a time, although variants (such as *Block TEA*) allow arbitrary-sized blocks.

To measure performance, I wrote test drivers to run each of the basic encryption and decryption functions in all four language implementations against varying data. I used simple calls to get the current millisecond clock value and subtracted the stop time from the start time to calculate elapsed milliseconds. I deliberately did not measure the time to move byte data in and out of words (blocks). To get accurate and precise measurements against such a fast algorithm (and a PC clock that has such a large millisecond granularity), I measured multiple passes of each function (10,000 to 1,000,000) and divided where necessary to "normalize" the results. The results are reported in *Table 1* below.

3.3 Strength

TEA has weathered years of cryptanalysis quite well. A couple of minor weaknesses were found in the algorithm shortly after publication. These were corrected by the authors in the form of *TEA Extensions*, often referred to as *XTEA* [7]. The corrections did not significantly alter the algorithm and maintained its performance and simplicity; for example, the modified XTEA algorithm can still be represented in only seven lines of C code for each of the encryption and decryption steps. At the same time, the authors published a modified variable-block algorithm (Block TEA), often called *XXTEA*.

I found that a very effective means of learning TEA's cryptographic strengths was to write test drivers and experiment with it. I began by formatting TEA encrypted outputs in binary to watch the confusion and diffusion patterns with different combinations of keys, data, and iteration counts (rounds). Indeed,

the key and data bits "travel quickly" and seemingly randomly across the encrypted results. The "delta" constant is effective in "hiding" weak keys; for example, using all zeros for both key and data still yields well dispersed bits even with very few rounds.

Given that my Java implementations performed well and were easy to work with, I wrote test drivers around them to do "brute force" (exhaustive search) "attacks" to attempt to find three types of weaknesses:

- *Hash collisions.* TEA was not designed to be a hash algorithm, and its potential for hash collisions has been documented. But, since TEA has been repeatedly misused as a hash (for example, in Microsoft's Xbox and ReiserFS), it's useful to understand the likelihood of hash collisions. Doing "brute force" searches for collisions is quite simple: for a given key, try different "plaintext" data and store each encrypted result (along with the input data) in a map for lookup. If two plaintexts produce the same encrypted text, a collision has occurred.
- *Related keys.* A potential related key weakness has been documented for the original TEA implementation [2], and partly addressed in the modified (XTEA) version. While much has been written and misunderstood about the severity of this weakness, the result is that the effective key length is decreased from 128 bits to 126 bits (each key having three other related keys), leaving TEA still quite strong, and with over double the effective key length of DES. With minor adjustments to my hash collision test driver (this time, keep the data constant but permute the key), I was able to easily search for and find related keys.
- *Key cracking.* A brute force key search of the keyspace was easy to code: simply try candidate keys in sequence until finding one that produces a known plaintext-ciphertext pair. This could take as long as 2^{128} iterations to find, but the actual runtime is smaller, for two reasons: 1) with luck, the matching key will be found much sooner than exhausting all combinations, and that will, of course, be "the last place you look," and 2) as described above, related keys reduce the search space to $O(2^{126})$. From performance measurements, I calculated that a full exhaustive search on my computer alone with my Java implementation would require over 800 trillion eons: well past my report deadline. Of course with faster implementations, more and better hardware, weak key choice, and luck, a match could be found significantly sooner. I did crack one key in my test driver.

See section 4.3 below for a summary of results from my cryptanalysis experiments.

4. RESULTS

The following sections summarize the results of my measurements and experiments.

4.1 Simplicity

Table 1 below reports the lines of code counts for just the core algorithms in each language. *Figure 4* through *Figure 6* in the appendix show the core of each of the Java, Smalltalk, and Ruby implementations.

As indicated by the very low line of code counts, the algorithm is indeed simple to implement in these varied languages, particularly when deliberately matching the published reference implementation as closely as possible. No single "port" took more than one hour to complete, including the time to write test drivers and xUnit test cases.

Language	*LOC (encrypt)*	*LOC (decrypt)*	*Encrypt time (1,000,000, millisecs)*	*Decrypt time (1,000,000, millisecs)*
C	7	7	391	390
Java	7	7	313	266
Ruby	12	13	1004	1034
Smalltalk	12	12	120,300	120,800

Table 1 - TEA Implementation Comparison

After completing my implementations and measurements, I researched and reviewed a few other implementations available on the internet and tested a few of them. This raised concerns about the broad availability of some incorrect (erroneous) implementations.

For example, Saurav Chatterjee of Jadavpur University published a Java TEA implementation that confused me. Its rounds used *Math.pow()* to exponentiate results. Obviously, this did not give encrypted results that matched my TEA and XTEA outputs, and it ran significantly slower than all other TEA and XTEA implementations. After studying the implementation, it became clear how the author erred: he read the C XOR operator (caret, ^) as exponentiation, and transliterated it to Java as such. One would expect the scope of this incorrect implementation to be limited, but it is included on several web pages, and is linked to by Wikipedia (and "if it's on Wikipedia, it must be true"). "Encryptor beware," indeed. Further, I found a published Smalltalk XTEA implementation that was equally confusing and also incorrect; for example, it omitted the step of XOR'ing in the sum. It was fast, but had some unique properties such as sometimes giving back the plaintext as the ciphertext when a zero key is used.

These broadly-available incorrect implementations point to the need for published test suites (plaintext/ciphertext pairs for given keys) to verify encrypted results. I created my own test data and results from the C implementation, and found this very helpful for verifying and debugging the other implementations as I wrote them (*Figure 7* of the Appendix shows one of these). Such certification test suites could help ensure that incorrect implementations do not survive and make their way into production software.

4.2 Performance

Table 1 above includes the run times for one million runs in each of the different implementations. These times are for the (original) TEA implementations with 32 iterations; XTEA times (in C, Smalltalk, and Java) were similar. All tests were run on a Dell Latitude D620 laptop with an Intel Core2 Duo running at 2.0 GHz with Windows XP. The particular tools and compilers used were GCC C 3.4.4, Sun Java 1.6.0, VisualAge Smalltalk 7.5.1, and Ruby 1.8.6.

I ran ten passes of each and reported the median times. I varied the inputs (data and keys) but found that this had no direct impact on the run time. TEA's linear performance is a strength, since timing attacks can be effective (for example, to reduce the search key space) against non-linear algorithms.

The actual times themselves are not as significant as the relative comparisons. I was surprised to find that the Java run times were so fast and that the Smalltalk performance was so poor. This likely reflects more on the quality of the virtual machines (and their code caches) rather than an intrinsic problem in the language, although low-level (bitwise) operations in interpreted languages rarely perform as well as

compiled code or assembly code. Ruby and Smalltalk users who must encrypt large amounts of data may be better served by implementing the algorithm in C or assembly (in a DLL) and calling out to it.

I was interested in how TEA performance compared to 56-bit DES. Since DES is readily available in Java (javax.crypto.Cipher.getInstance("DES")), I wrote a test driver there to measure and compare. I found that DES was over 18 times slower for encryption (less for decryption) than TEA. This is not a scientific comparison, since the Java DES implementation includes additional overhead for validation, loading, etc. A PC Magazine study showed TEA (128 bit, 32 iterations) to be over 60% faster than 56-bit DES, and about 4 times faster than 168-bit 3DES.

The Block TEA and XXTEA variants are recommend for encrypting large amounts of data, but I did not implement or measure those algorithms.

4.3 Strength

In a practical sense, modified TEA (XTEA) with proper keys and adequate rounds is quite strong as an encryption algorithm. In an academic sense[1] [3], as noted above, unmodified TEA has a published related key weakness [2] that reduces the effective key length from 2^{128} bits to 2^{126} bits and could result in a partial attack with 2^{34} chosen plaintexts. 2^{34} uncompressed plaintexts would 128 gigabytes of storage, but could be encrypted in just over one hour (it's not known how long the differential attack would take to run). Unfortunately, this often gets misrepresented that TEA is inherently weak and should not be used.

In practice, misinformation about encryption algorithms can be more dangerous than academic weaknesses in the design. Section 4.1 describes two incorrect TEA implementations that are broadly available on the internet, and at least one of these implementations is significantly weaker than the correct implementation. As another example, the Microsoft Xbox misuses TEA as a hash function. The security risk of TEA's related key weaknesses in this context has been overstated.[5] The greater risk for the Xbox is that hashing with any private key algorithm requires storing a key, and the key might be detected, as was quickly done with the Xbox's RC4 key. Choosing a true hash algorithm would have avoided this risk. These examples underscore the importance of truly understanding an algorithm in order to correctly apply it.

As explained above, I wrote and ran simple programs to do very basic cryptanalysis of TEA and XTEA results. This was more an educational exercise than formal research or careful analysis. However, this led to a few items to note:

- I did not find hash collisions or related keys with the full 32-iteration TEA or with XTEA. The single biggest barrier was not processing time, rather storage. Finding collisions or related keys required storing and searching increasingly more encryption results until I exhausted available physical memory. Allowing paging to occur (and writing and searching results to/from disk) significantly slowed the algorithm.
- When I reduced the algorithm from 32 to 16 iterations, it was not difficult to find related keys. A simple variant of my collision searching program (see *Figure 8* in the Appendix) could typically find dozens of related keys (for particular data values) in under a minute.

[1] As Bruce Schneier explains, "in academic cryptography, the rules are relaxed considerably. Breaking a cipher simply means finding a weakness… that can be exploited with a complexity less than brute-force. Never mind that brute-force might require 2^{128} encryptions; an attack requiring 2^{110} encryptions would be considered a break."

- Related keys and hash collisions were not confined to reported bit 16 and 32 permutations [4], although bit 30 [2] did factor in many of them. I would like to see further details on such patterns or, better still, see published algorithms that demonstrate these reported problems.
- Reports of related key and collision discoveries are often overstated; in fact, all such TEA/XTEA reports I found were against weakened (reduced round) implementations, such as this one: http://osdir.com/ml/file-systems.ext2.devel/2002-09/msg00011.html. As stated above, it's quite easy to find related keys with a reduced round TEA algorithm, but the practical significance of this is unclear, apart from leading to "fear, uncertainty, and doubt."
- My exhaustive key search program was simple and purely academic, but it did find one key. I wrote the program to generate a random number key, encrypt a known plaintext against that key, forget the key, and the start a key search from zero to re-encrypt the plaintext and look for a match to the ciphertext. I wrote the program to stop after 1,000,000,000 iterations (about 400 seconds), so it was far from exhaustive. However, in one pass it found a hit simply because the random key selected was low enough. This was not at all surprising, but it was a simple reminder of the importance of good key selection. Humans tend to select low and guessable numbers for keys, and such weak keys can be easily cracked.

5. CONCLUSIONS AND RECOMMENDATIONS

The simple conclusions (and answers to my opening questions) are: yes, TEA/XTEA is easy to implement, fast, efficient, and cryptographically strong. When implemented and used correctly, TEA can be an excellent choice, particularly for encrypting and decrypting small, short-lived data in resource-constrained devices. I have personally seen simple XOR algorithms used for "encryption" in cases where the programmer "doesn't have time, resources, or need for elaborate encryption algorithms." TEA would certainly be a better alternative.

Based on my research, I offer the following recommendations for using TEA and other encryption algorithms:

- Use XTEA whenever memory and processing requirements are limited, or when you want basic fast encryption without requiring pre-requisite code that may be licensed or patented, or carry export restrictions. It should not be used for extremely sensitive or long-lived data.
- Choose algorithms carefully. Understand the purpose and benefits of a particular algorithm before choosing it. TEA/XTEA is a private key encryption algorithm; do not repeat Microsoft's mistake of using it as a hash function. Many security weaknesses have come from the simple fact that an otherwise good algorithm was used for the wrong purpose.
- Choose keys carefully and protect them. TEA/XTEA is a private key algorithm, and its key must be protected. Do not repeat Microsoft's mistake of letting a critical private key be easily detected.
- Seek to understand the true significance of "weakness" claims. A "weakness" may have no real practical impact other than to spread "fear, uncertainty, and doubt." Cryptanalysis algorithms should be published and independently verified.
- Encryption authors should publish test suites to verify implementations, in addition to complete reference implementations.
- Test/verify an encryption algorithm against known expected results before putting it to use.
- Use Block Tea/XXTEA for large amounts of data. Add careful block chaining when encrypting multiple blocks.

6. REFERENCES

[1] Feistel, Horst. *Cryptography and Computer Privacy*. Scientific American. Vol. 228(5). May 1973.

[2] Kelsey, John; Schneier, Bruce; and Wagner, David. *Related-Key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA*. ICICS '97 Proceedings. Springer-Verlag. November 1997.

[3] Scheier, Bruce. *A Self-Study Course in Block-Cipher Cryptanalysis*. Cryptologia, Vol. 24(1). January 2000.

[4] Steil, Michael. 17 *Mistakes Microsoft Made in the Xbox Security System.* October, 2005.

[5] Steil, Michael. *The Hidden Boot Code of the Xbox*. Xbox-Linux. August, 2005.

[6] Wheeler, David J. and Needham, Roger M. *TEA, a Tiny Encryption Algorithm*. Computer Laboratory, Cambridge University, England. November, 1994.

[7] Wheeler, David J. and Needham, Roger M. *TEA Extensions*. Computer Laboratory, Cambridge University, England. October, 1997.

7. APPENDICES

Figure 2 gives the TEA decryption routine as published by Needham and Wheeler. As you can see, it is similar to (the reverse of) the encryption routine shown in *Figure 1* above.

```
void decode(long* v,long* k) {
       unsigned long n=32, sum, y=v[0], z=v[1],
                    delta=0x9e3779b9 ;
                    sum=delta<<5 ;

       while (n-->0) {            /* start cycle */
              z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]) ;
              y-= ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]) ;
              sum-=delta ;
       }      /* end cycle */
       v[0]=y ; v[1]=z ; }
```

Figure 2 –TEA Decryption Routine in C

Figure 3 gives the modified TEA ("XTEA") algorithm as published by Needham and Wheeler.

```
tean( long * v, long * k, long N) {
       unsigned long y=v[0], z=v[1], DELTA=0x9e3779b9 ;
       if (N>0) {
              /* coding */
              unsigned long limit=DELTA*N, sum=0 ;
              while (sum!=limit)
                     y+= (z<<4 ^ z>>5) + z ^ sum + k[sum&3],
                     sum+=DELTA,
                     z+= (y<<4 ^ y>>5) + y ^ sum + k[sum>>11 &3] ;
              }
       else {
              /* decoding */
              unsigned long sum=DELTA*(-N) ;
              while (sum)
                     z-= (y<<4 ^ y>>5) + y ^ sum + k[sum>>11 &3],
                     sum-=DELTA,
                     y-= (z<<4 ^ z>>5) + z ^ sum + k[sum&3] ;
       }
       v[0]=y, v[1]=z ;
       return ; }
```

Figure 3 – Modified TEA (XTEA) Algorithm in C

Figure 4 through *Figure 6* below show the "core" encrypt and decrypt code I wrote in Java, Smalltalk, and Ruby.

```
public class TinyEncryptor {

	private static int delta = 0x9E3779B9; /* a key schedule constant */
	public void code(int[] v, int[] k) {
		int y=v[0], z=v[1], sum=0, n=32;
		int k0=k[0], k1=k[1], k2=k[2], k3=k[3];  /* cache key */
		while (n-- > 0) {
			sum += delta;
			y += ((z << 4) + k0) ^ (z + sum) ^ ((z >>> 5) + k1);
			z += ((y << 4) + k2) ^ (y + sum) ^ ((y >>> 5) + k3);
		}
		v[0]=y;		v[1]=z;
	}

	public void decode (int[] v, int[] k) {
	    int y=v[0], z=v[1], sum=0xC6EF3720, n=32;  /* set up */
	    int k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */

		while (n-- > 0) {
			z -= ((y << 4) + k2) ^ (y + sum) ^ ((y >>> 5) + k3);
			y -= ((z << 4) + k0) ^ (z + sum) ^ ((z >>> 5) + k1);
			sum -= delta;
		}
	    v[0]=y; v[1]=z;
	}
}
```

Figure 4 –TEA Algorithm in Java

```
teaEncrypt: data
	| key y z delta sum |
	key := self encryptionKey.
	y := data at: 1.	z := data at: 2.
	delta := 16r9E3779B9.	sum := 0.

	32 timesRepeat: [
		sum := (sum + delta).
		y := y + ((((z bitShift: 4) + (key at: 1)) bitXor: (z + sum))
bitXor: ((z bitShift: -5) + (key at: 2))).
		y := y bitAnd: 16rFFFFFFFF.
		z := z + ((((y bitShift: 4) + (key at: 3)) bitXor: (y + sum))
bitXor: ((y bitShift: -5) + (key at: 4))).
		z := z bitAnd: 16rFFFFFFFF.
	].
	^Array with: y with: z

teaDecrypt: data
	| key y z delta sum |
	key := self encryptionKey.
	y := data at: 1.	z := data at: 2.
	delta := 16r9E3779B9.	sum := 16rC6EF3720

	32 timesRepeat: [
		n := n - 1.
		z := z - ((((y bitShift: 4) + (key at: 3)) bitXor: (y + sum))
bitXor: ((y bitShift: -5) + (key at: 4))).
		z := z bitAnd: 16rFFFFFFFF.
		y := y - ((((z bitShift: 4) + (key at: 1)) bitXor: (z + sum))
bitXor: ((z bitShift: -5) + (key at: 2))).
		y := y bitAnd: 16rFFFFFFFF.
		sum := (sum - delta).
	].
	^Array with: y with: z
```

Figure 5 –TEA Algorithm in Smalltalk

```ruby
class TEA
  DELTA = 0x9e3779b9

  def self.tea_encrypt(v, k)
    y=v[0]; z=v[1]; sum=0;
    k0=k[0]; k1=k[1]; k2=k[2]; k3=k[3]

    32.times do |i|
      sum += DELTA
      y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1)
      y = y & 0xFFFFFFFF
      z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3)
      z = z & 0xFFFFFFFF
    end
       return [y,z]
  end

  def self.tea_decrypt(v, k)
    y=v[0]; z=v[1]
    k0=k[0]; k1=k[1]; k2=k[2]; k3=k[3]
    sum = DELTA << 5

    32.times do |i|
       z -= ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3)
       z = z & 0xFFFFFFFF
       y -= ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1)
       y = y & 0xFFFFFFFF
       sum -= DELTA
    end
    return [y,z]
  end
end
```

Figure 6 –TEA Algorithm in Ruby

```
testEncrypts
	"Run some basic encryptions and verify the results"
	"self new testEncrypts"


	self encryptionKey: (Array with: 0 with: 0 with: 0 with: 0).
	self
		assert: (self testEncrypt: (Array with: 0 with: 0) compare: (Array with: 16r41EA3A0A with: 16r94BAA940));
		assert: (self testEncrypt: (Array with: 1 with: 1) compare: (Array with: 16rE0050D07 with: 16r4FB50C13));
		assert: (self testEncrypt: (Array with: 16r12345678 with: 16r9ABCDEF0) compare: (Array with: 16r7FE2E480 with: 16r4F66BD75));
		assert: (self testEncrypt: (Array with: 16rFFFFFFFF with: 16rFFFFFFFF) compare: (Array with: 16rF6F4BF6E with: 16r1335B5B8)).


	Transcript cr.
	self encryptionKey: (Array with: 16r12345678 with: 16r9ABCDEF0 with: 16r12345678 with: 16r9ABCDEF0).
	self
		assert: (self testEncrypt: (Array with: 0 with: 0) compare: (Array with: 16rBCDA8737 with: 16r1024D312));
		assert: (self testEncrypt: (Array with: 1 with: 1) compare: (Array with: 16r8AC711A0 with: 16r75CFE57E));
		assert: (self testEncrypt: (Array with: 16r12345678 with: 16r9ABCDEF0) compare: (Array with: 16r3ADDB70 with: 16r5EAEA194));
		assert: (self testEncrypt: (Array with: 16rFFFFFFFF with: 16rFFFFFFFF) compare: (Array with: 16rEEFBE7FB with: 16r70ED4B9D)).
```

Figure 7 –SUnit for Verifying Encryption Results

```
public class TEACollisionSearcher {

	public static void main(String[] args) {
		TinyEncryptor tea = new TinyEncryptor();
		int key[] = {0,0,0x12345678,0x9abcdef0};
		int data[] = {0x12345678,0x9abcdef0};
		HashMap<Integer, Integer[]> map = new HashMap<Integer, Integer[]>();
		for (int i=0; i<=1000000; i++) {
			key[0] = i;  key[1] = i;
			data[0]=0x12345678;  data[1]=0x9abcdef0;
			tea.code(data, key);
			Integer lookup = data[0];
			Integer[] match = map.get(lookup);
			if (match != null && match[0] == data[1]) {
				System.out.println("Collision - Key1[0,1]: "
					+ Integer.toHexString(key[0]) + " "
					+ Integer.toHexString(key[1]) + " "
					+ " Key2[0,1]: "
					+ Integer.toHexString(match[1]) + " "
					+    Integer.toHexString(match[2])    +
":");
			}
			Integer[] put = {data[1], key[0], key[1]};
			map.put(lookup, put);
		}
	}
}
```

Figure 8 –TEA Collision Searcher