The Tiny Encryption Algorithm

SIMON J. SHEPHERD

Abstract  The Tiny Encryption Algorithm (TEA) has been around for just over ten years. It is probably the most "minimal"—and hence fastest—block cipher ever devised and yet appears resistant to most attacks. In this article, we describe the algorithm, its simplicity of design and ease of use, its cryptographic strength, and the wide range of implementations and applications of the cipher.

Keywords  block cipher, cryptanalysis, cryptography, encryption, fast software encryption, feistel cipher, group theory

Introduction

Traditionally, when cryptography was the province of the military and, more recently, of the banking community, ciphers were generally implemented in hardware. Everyone is familiar with the beautifully crafted Enigma machines used by the Germans in World War II, and a number of excellent articles on them have appeared in these pages. Anyone who has worked in the back office of a bank has seen the SWIFT terminals used for inter-bank funds transfer. These are hardware devices and are typical of the way in which cryptography was implemented from the earliest civilisations through to the advent of the electronic computer.

Nowadays, nearly all cipher algorithms are required in software for use with applications running on PCs. In this article, we describe the Tiny Encryption Algorithm (known by its convenient acronym TEA), which is probably the most efficient—and hence fastest—software encryption algorithm ever devised. The code is so minimal it can be memorised, and it is easily programmed in a few minutes in almost any computer language the user prefers. TEA was developed by David Wheeler and Roger Needham at the Computer Laboratory of Cambridge University and was first presented at the Fast Software Encryption workshop at Cambridge in 1994. Their original paper is available online at [20].

Before we look at the detailed operation of TEA, we need to introduce some related cipher algorithms and some key theoretical ideas. We will compare and contrast TEA with the DES which will be familiar to all readers. We will summarise the principles of differential cryptanalysis, the most powerful general attack on Feistel ciphers, and introduce some elementary group theory.

Address correspondence to Dr. Simon Shepherd, Advanced Signals Laboratory, SEDT, University of Bradford, Bradford BD7 1DP, United Kingdom. E-mail: s.j.shepherd@bradford.ac.uk
Feistel Block Ciphers

For more than 30 years, the world standard block cipher was the Data Encryption Standard (DES). Although it has now been replaced with the Advanced Encryption Standard (AES), the DES is an excellent cipher and will be very familiar to readers of this journal. Designed by IBM in the 1950s, it withstood every attempt to break it, including very modern attacks such as differential and linear cryptanalysis. DES is the best-known example of a classical Feistel cipher in that it uses a number of “rounds” to scramble the data. Each round implements Shannon’s [16] classical twin requirements of *diffusion* and *confusion*. These two ideas are absolutely critical to the security of all Feistel ciphers and it is important to distinguish carefully between their contributions to the overall security:

- The purpose of *diffusion* is to disguise the statistics of the underlying plaintext language so that frequency attacks are not possible. This is traditionally achieved by *permutation*.
- The purpose of *confusion* is to make the ciphertext depend on both the plaintext and the key in such a complex manner as to make it as difficult as possible for an attacker to narrow down the keyspace, and hence possibly isolate the key in use. This is traditionally achieved by *substitution*.

Shannon asserted that any algorithm that properly implements these two ideas should be secure. The DES uses 16 rounds of permutations and substitutions to achieve a very secure algorithm. The fact that DES proved resistant to differential and linear cryptanalysis is not just luck. As described in [4], IBM and NSA knew all about differential and linear cryptanalysis half a century before academia caught up and designed the DES from the ground up to be resistant to it.¹ Coppersmith states that “…the entire algorithm was published in the Federal Register, but the design considerations, which we present here, were not published at that time. The design took advantage of certain cryptanalytic techniques, most prominently the technique of differential cryptanalysis, which were not known in the published literature. After discussions with NSA, it was decided that disclosure of the design considerations would reveal the technique of differential cryptanalysis, a powerful technique that can be used against many ciphers. This in turn, would weaken the competitive advantage the United States enjoyed over other countries in the field of cryptography”[4].

We will now define what we mean by a block cipher in the most general sense in terms of group theory, as we will need these ideas later.

**Definition Block cipher:** Let $E$ be a transformation that maps $N$-bit blocks to $N$-bit blocks. We call $E$ a *block cipher*. Let $E_K(X)$ be the encryption of $X$ under key $K$. Then for any fixed $K$, the map sending $X \to E_K(X)$ is a permutation of the set of $N$-bit blocks. Denote this permutation by $P_K$. The set of all $N$-bit permutations is called the symmetric group written $S(2^N)$. The order of the symmetric group is $(2^N)!$. The subset of the total permutations $P_K$, where $K$ ranges over all possible keys, is denoted $E(S(2^N))$ where $E(S(2^N)) \subseteq S(2^N)$.

¹This was not the first or last time an academic “discovery” in cryptography has been announced, only to find later that the military have known about it for years. Readers may recall that it was announced a few years ago that James Wilkinson at GCHQ Cheltenham had discovered what he called “non-secret encryption” years before the functionally identical RSA public key cipher was announced.
In Practice, of Course, \( E(S(2^N)) \) is a vanishing small subset of \( S(2^N) \). This is not only important from a practical point of view in keeping the key size manageable small, but also from a security point of view. For example, DES operates on 64-bit blocks under the control of a 56-bit key. Therefore, the order of \( S(2^N) \) is \( (2^{64})! \), whereas the order of \( E(S(2^N)) \) is only \( 2^{56} \). Thus, the number of mappings permitted by the keyspace is only a tiny fraction (4.853 \( \times 10^{-20} \)) of those available from the complete symmetric group. If the key were to reach the entire set of mappings available, we would require 1.154 \( \times 10^{21} \) keys (or roughly the same number of keys as there are sub-atomic particles in the Universe!) The point is that we must make \( E \) an apparently random mapping from plaintexts to ciphertexts. For further details, the reader is referred to §4 of [18].

**Differential Cryptanalysis**

For the benefit of readers who are not familiar with the technique, we give here an overview of the technique of differential cryptanalysis. If we consider the 16-round DES, let us imagine a cryptanalyst who starts with a known plaintext \( m \) and an unknown key \( k \) and tries to follow the encipherment process through the 16 rounds. The hapless cryptanalyst will quickly become hopelessly entangled since different bits of \( k \) are combined with the message at the input of every S-box. In differential cryptanalysis, however, we start with two messages \( m \) and \( m' \) differing (in terms of Hamming distance) by a known, chosen difference \( \Delta m \), that is:

\[
\Delta m = m \oplus m'.
\]

For example, say the input to box \( SI \) at round \( i \) of the encipherment of \( m \) is:

\[
m_i[32, 1, 2, 3, 4, 5] \oplus k_i[1, 2, 3, 4, 5, 6],
\]

and likewise, the input to box \( SI \) at round \( i \) of the encipherment of \( m' \) is:

\[
m'_i[32, 1, 2, 3, 4, 5] \oplus k_i[1, 2, 3, 4, 5, 6].
\]

From the identity \((a \oplus c) \oplus (b \oplus c) = a \oplus b\) we have that the XOR of the two message inputs is

\[
(m_i[32, 1, 2, 3, 4, 5] \oplus k_i[1, 2, 3, 4, 5, 6]) \oplus (m'_i[32, 1, 2, 3, 4, 5] \oplus k_i[1, 2, 3, 4, 5, 6])
\]

\[
= m_i[32, 1, 2, 3, 4, 5] \oplus m'_i[32, 1, 2, 3, 4, 5]
\]

\[
= \Delta m_i[32, 1, 2, 3, 4, 5].
\]

The dependance on \( k \) has vanished! Now suppose that there is some relationship between the input differences and output differences of an S-box. Differential cryptanalysis uses the possibility that many different input pairs with a given difference \( \Delta I \) give rise to the same output difference \( \Delta O \). Thus, we begin with two plaintext messages who difference is known with certainty and trace through a probable pattern of round-by-round differences. If the ciphertexts exhibit the difference of our probable pattern we can make deductions about the key bits on the basis of this message pair. By trying other pairs, we can gain some knowledge of the probable bits of the key.

Differential cryptanalysis together with its close relation linear cryptanalysis, is far and away the most powerful attack on block ciphers and it is important to bear this attack in mind when designing a strong block cipher. DES was designed *ab initio*
to resist this kind of attack. We shall see that TEA, in spite of its extreme simplicity, also appears to be highly resistant to this analysis.

From Hardware to Software

In terms of implementation, DES was made for hardware. When it was developed, there were no PCs (in fact, very few computers at all). A software version would have not only been pointless, it would have been very difficult. When the banking community were looking for a secure cipher for inter-bank funds transfers (known as the SWIFT system), DES was chosen. The SWIFT terminals used in banks are just hardware embodiments of the DES cipher.

With the advent of the PC and the increasingly widespread use of embedded, ubiquitous computing, the situation changed dramatically. Nowadays, everyone has a PC and nearly all cipher algorithms are required in software. Users just want to load up a program—they don’t want to take the lid off the computer and install a piece of hardware. However, designing ciphers that are efficient in software is quite a different challenge to designing for hardware. For example, the operation of permutation is quite tedious in software, whereas in hardware it can be implemented essentially “for free” just by wiring.

In the 1990s, a series of workshops on Fast Software Encryption were held at Cambridge University in the UK. These workshops were specifically aimed at developing strong, secure ciphers that ran quickly in software. TEA was first published at the 1994 workshop and in terms of meeting the eponymous “Fast Software” epithet, has yet to be beaten!

Avoiding Permutation and Substitution

Before we discuss the details of TEA as an algorithm, we need to introduce another idea concerning an alternative way in which we can achieve Shannon’s requirements of diffusion and confusion but in a manner that is efficient in software. We have already noted that permutation is tedious in software as it requires a lot of registering, while substitution is equally problematic as it requires look-up tables, implying the use of memory. Interestingly, there are other ways around the problem.

One of the best block ciphers ever developed is IDEA (International Data Encryption Algorithm) by Xuejia Lai and Jim Massey at ETH Zurich [12]. In this article, we are not concerned with IDEA per se, but this cipher is worthy of note as it was the first algorithm to use the somewhat different concept that TEA also uses for achieving diffusion and confusion.

IBM’s Don Coppersmith and Jim Massey independently showed that mixing operations from orthogonal algebraic groups—that is, operations that are in some sense “algebraically incompatible”—is functionally equivalent to the diffusion and confusion functions that a classical Feistel block cipher would implement with P-boxes and S-boxes. In terms of a fundamental CPU instruction set, if common operations such as add-mod-2, add-with-carry, shift-left, shift-right, XOR, and so on are combined in the right way, a strong Feistel-type block cipher will result.

2Anyone who has tried to implement DES in software even using a modern, powerful language like C knows how painful it is to code all the bit swapping, permutations and substitutions. And the resulting executable code is relatively slow. However, a number of software versions were coded and Unix used the DES in its standard crypt() algorithm.
The design of a secure algorithm based on this approach is not trivial and requires a deep knowledge of group theory, in particular, Wreath products. The interested reader should consult [21] and the references therein, as well as [19].

The basic idea is this: We have to identify a set of simple instructions (i.e., core CPU instructions that execute in a single cycle) that are bijective functions on the set of $n$-bit binary vectors. All of the basic operations can be regarded as *permutations* which allows us to speak about their cycle structure, element order and so on. Perhaps surprisingly, *every* permutation can be obtained from products of simple operations even if we restrict ourselves to just two operations [21]. Every permutation constitutes the *symmetric group*. Even permutations constitute the *alternating group*, whose order is exactly half that of the symmetric group.

If we recall our group theoretic definition of a block cipher, we can see how these ideas can be applied in practice to the design of secure algorithms. If, for example, we XOR an input value $x$ with an arbitrary (non-zero) constant $c$, we get an *involution*. Any transposition in the cycle decomposition of $x \oplus c$ can be constructed simply by choosing an appropriate $c$. So if we have $n$ bits in the input vectors, we have $2^n-1$ choices for $c$ yielding $2^n-1$ different fixed-point-free involutions. Note that XOR is always an *even* permutation since the number of transpositions in its cycle decomposition is equal to $2^n-1$. Thus, these permutations give rise to the alternating group.

If we now take another operation such as $ROR(k)$ (rotate right by $k$ bits), we have a permutation order of $n = \gcd (k, n)$. This permutation is *not* fixed-point-free and contains cycles of different lengths dividing $n$. Shifting by different $k$ gives non-trivial permutations of this type.

The problem is now how to combine these simple operations to yield fast, secure encryption schemes, that is, permutations of $n$-bit vectors with a high structural complexity. As pointed out in [21], slightly different combinations can give rise to very large differences in the results. For example, if we just combine SHIFT and XOR we can produce $2^n$ different "encryptions." However, if we choose SHIFT and ADD$^3$ we can generate $(2^n)!$ possible encryption functions, a vastly larger number! These differences are surprising in view of the fact that, if we take two arbitrary permutations from the symmetric group of degree $n$, with high probability$^4$ these two elements generate the entire alternating (or symmetric) group. In [21], the authors go on to devise a very simple cipher they call MIX-2 which uses only two operations (ADD and SHIFT) in a classic Feistel round scheme. The register length $n$ can be any power of two, depending on the computer, and the cipher generates the whole symmetric group of order $2^n$. If $M$ is the message, $K = \{b_1, b_2, \ldots, b_k\}$ is a $k$-bit key where bits $\{b_1, b_2, \ldots, b_r\}$ are the first $r$ bits of the key and $r$ is the number of rounds, the cipher code is shown in Code Block 1.

```
Code Block 1. MIX-2 cipher pseudocode.

for (i=0; i<r; i++) {
    if (b_i) ADD_b (M);
    else SHIFT_a (M);
}
```

$^3$Here we mean addition modulo $2^n$, ignoring carry.

$^4$Probability $\to 1$ as $n \to \infty$. 

Here, $a$ and $b$ are odd and the bit sequence $\{b_{r+1}, b_{r+2}, \ldots, b_{r+n-1}, 1\}$ is the binary representation of $b$ and similarly $\{b_{r+n}, b_{r+n+1}, \ldots, b_{k}, 1\}$ is the binary representation of $a$. The number of rounds required to ensure generation of the entire symmetric group is $r = k - n - \log_2(n) + 2$. For example, if $n = 64$ and we choose a 256-bit key $K$, we need $r = 256 - 64 - 6 + 2 = 188$ rounds.

The reader should keep this scheme in mind when looking at the code of TEA. The similarities are clear and it is an enormously insightful exercise to analyse TEA in the same way as MIX-2. We now have all the theory and tools at our disposal to look critically at the design and strength of TEA as a cipher algorithm.

The Tiny Encryption Algorithm

TEA uses the same mixed algebraic groups technique as MIX-2 and IDEA. While it is a little more complex than MIX-2, it is very much simpler than IDEA and hence much faster. (Also it is public domain, whereas many ciphers, such as IDEA, are patented.) The TEA encryption and decryption routines have the general form (in C) as shown in Code Block 2. TEA takes 64 bits of data in $v[0]$ and $v[1]$, and 128 bits of key in $k[0]$ - $k[3]$. The result is returned in $w[0]$ and $w[1]$. Returning the result separately makes implementation of cipher modes other than Electronic Code Book a little bit easier.

The quantity $\delta$ is a “magic number”, chosen to be the Golden ratio \(\left(\frac{\sqrt{5}}{2} - 1/2 \sim 0.618034\right)\) multiplied by $2^{32}$. The actual value is not that important—it simply serves to ensure that encryption/decryption in each round is different. On entry to decipher(), sum is set to be $\delta n$. Which way round you call the functions is arbitrary since the operations are commutative: $D_{K}(E_{K}(P)) = E_{K}(D_{K}(P))$ where $E_{K}$ and $D_{K}$ are encryption and decryption under key $K$, respectively.

TEA can be operated in any of the modes of DES like electronic code book, cipher block chaining, output feedback more, etc. Figure 1 shows two Feistel rounds which is equivalent to one “cycle” of TEA [15].

In terms of CPU instructions, TEA uses four fast (usually single machine cycle) operations that can be found on almost any processor: SHL, SHR, ADD and XOR. The group theoretic combination of these four operations results in a highly random sub-space mapping of the symmetric group after as few as 8 cycles, equivalent to 16 Feistel rounds as used by DES.

Shortly after TEA was published, a minor weakness was discovered (discussed later). In response, the authors responded [14] with a slightly amended version of the algorithm known as XTEA (eXtended TEA). The changes are very minor and an XTEA cycle is shown in Figure 2 for comparison [15] and the ANSI C code in Code Block 3.

There is also a further variant known as Block TEA (and its strengthened variant known as XXTEA) which operates on variable-length blocks which are a selectable multiple of 32-bits in length. The algorithm applies the XTEA round sequentially to each word in the block and adds it to the adjacent result. For details, see [14]. The advantage of this approach is that the need for modes such as cipher block chaining (CBC), output feedback more (OFB), and so on is obviated and efficiency is improved on longer messages.

Cryptanalysis

The general topic of paranoia concerning block ciphers is the number of rounds $n$ required. The authors of TEA indicate that 16 rounds is sufficient but suggest 32
for safety. In fact, we suggest that as few as eight should be quite enough for most applications, especially ones where the data age quickly (real-time video, for example). Our experiments have shown that the algorithm achieves good dispersion after just six iterations. (Dispersion is the property of a cipher that attempts to ensure in the ideal case that half the bits of the ciphertext should change at random when one bit of either the plaintext or the key is flipped.) The iteration count can be made variable if required.

Code Block 2. TEA in ANSI C.

```c
void encipher(unsigned long *const v, unsigned long *const w,
              const unsigned long *const k)
{
    register unsigned long y = v[0], z = v[1], sum = 0, delta = 0x9E3779B9,
                              a = k[0], b = k[1], c = k[2], d = k[3], n = 32;

    while (n-- > 0)
    {
        sum += delta;
        y += (z << 4) + a ^ z + sum ^ (z >> 5) + b;
        z += (y << 4) + c ^ y + sum ^ (y >> 5) + d;
    }

    w[0] = y; w[1] = z;
}

void decipher(unsigned long *const v, unsigned long *const w,
               const unsigned long *const k)
{
    register unsigned long y = v[0], z = v[1], sum = 0x6EF3720,
                                delta = 0x9E3779B9, a = k[0], b = k[1],
                                           c = k[2], d = k[3], n = 32;

    /* sum = delta<<5, in general sum = delta * n */

    while (n-- > 0)
    {
        z -= (y << 4) + c ^ y + sum ^ (y >> 5) + d;
        y -= (z << 4) + a ^ z + sum ^ (z >> 5) + b;
        sum -= delta;
    }

    w[0] = y; w[1] = z;
}
Kelsey et al. reported a minor weakness in the original TEA algorithm [9, 10] which was quickly addressed by the original authors in by a very minor change to the original code [14], the variant being known as XTEA. The “weakness” was very minor and has no significance for real-world applications. For a comprehensive cryptanalysis of TEA and its variants, the reader is encouraged to consult Andem’s thesis [1], which is available for download online.

The main problem with the original version of TEA was the very simple key schedule. The 128-bit key is divided into 4 32-bit words $K_0$, $K_1$, $K_2$, and $K_3$. TEA uses $K_0$ and $K_1$ in the odd-numbered rounds and $K_2$ and $K_3$ in the even-numbered rounds. If it were not for the way in which $delta$ changes the encryption on each round, the cipher would be vulnerable to a slide attack [2], as pointed out by Fleming [5]. The simple way in which the key is introduced into each round in the original algorithm means that inverting simultaneously the most significant bits of $K_0$ and $K_1$ does not change the result of the encryption. (The same is true of $K_2$ and $K_3$). Therefore, TEA has 3 “equivalent keys” for each key, reducing the effective keyspace from 128 bits to 126 bits. Although this is insignificant in terms of a brute-force attack, a curious incident highlights nicely the point that weak implementations of strong algorithms can be very vulnerable!

Figure 1. Two Feistel rounds = 1 “cycle” of TEA.
The Microsoft X-Box gaming console is nothing more than a rather capable PC with the usual motherboard, sound card, graphics card, hard disc, CD drive, and so on. Microsoft sells the X-Box at a considerable discount to the cost of an equivalent PC—the revenue stream being in the somewhat inflated prices of the games. The only limitation is that the X-Box is booted from ROM into a very restricted “operating system” that just runs the games. There is nothing in principle why it should not run a fully-fledged OS like Linux. Naturally, hackers had the idea of breaking this protection so that they could have a capable PC for little cost. To guard against this possibility, Microsoft had “protected” the X-Box by using TEA as a hash function (contained in ROM) during the boot sequence to perform a checksum of the flash memory to see if any unauthorised changes had been made. The Linux X-Box group exploited the weakness in the TEA key schedule to gain control of the machine [3]. Inverting the 32nd and 64th bits of the first 64-bit block of the protected area caused the X-Box to begin executing from RAM, not from ROM. It was then just a matter of placing a boot loader in RAM to boot the desired OS from the disc.

Figure 2. Two Feistel rounds = 1 “cycle” of XTEA.

5Used here in the most complementary meaning of the word.
TEA has also been subjected to various other cryptanalytic studies. The present author worked with Hernandez, Sierra, and Mex-Pereira when they were at Bradford on machine learning algorithms for distinguishing statistical deviations from randomness in ciphers. Later, these authors published some further work on distinguishers for TEA based on genetic algorithms [6–8]. The idea here is to identify a sub-space of the plaintext, which maps to a sub-space of the ciphertext in a statistically anomalous way—the unevenness being detectable by a chi-squared test. The technique is capable of finding distinguishers for up to 4 cycles (8 Feistel rounds) of TEA but no more. Hence, TEA seems to be resistant to this kind of analysis.

Code Block 3. XTEA in ANSI C.

```c
void encipher(const unsigned long *const v, unsigned long *const w,
    const unsigned long * const k)
{
    register unsigned long y=v[0],z=v[1],sum=0,delta=0x9E3779B9,n=32;

    while(n-->)
    {
        y += (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
        sum += delta;
        z += (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
    }

    w[0]=y; w[1]=z;
}

void decipher(const unsigned long *const v, unsigned long *const w,
    const unsigned long * const k)
{
    register unsigned long y=v[0],z=v[1],sum=0xC6EF3720,
        delta=0x9E3779B9,n=32;

    /* sum = delta<5, in general sum = delta * n */

    while(n-->)
    {
        z -= (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
        sum -= delta;
        y -= (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
    }

    w[0]=y; w[1]=z;
}
```
With the publication of differential and linear cryptanalysis in the public domain, several authors have examined TEA in this regard. Perhaps surprisingly, TEA seems more resistant to differential attack than XTEA. Moon et al. [13] attacked 14-round TEA but needed $2^{52.5}$ chosen plaintexts and a computational effort of $2^{85}$ encipherments. Attacks on XTEA by Ko et al. [11] require $2^{20.5}$ chosen plaintexts under related key-pairs and a computational effort of $2^{115.15}$ 27-round encipherments.

Many algorithms are reported as “broken” in the literature, only to find on closer inspection that the attack uses such ludicrous resources as to be of (limited) academic interest only. The attacks on TEA are absolutely impossible in practice, even if one has the reputed resources of the NSA. In summary, TEA has proved to be a strong little algorithm that has withstood all the attacks upon it as a “black box.” However, as the incident with the X-Box shows, good algorithms can be poorly implemented, weakening them to the point of uselessness.

**Conclusion**

TEA has been around for over 10 years and no serious (i.e., practical) weaknesses have been found. The cipher is extremely compact and can (almost) be programmed from memory in any computer language or CPU instruction set for use in a wide range of security applications including confidentiality, hashing, random number generation, and so on. Unlike many other ciphers such as IDEA or those from RSA Labs, TEA is not encumbered by patents or any other commercial claim. It is totally public domain and can be used freely. Its main advantage is very high speed, which is ideally suited to modern applications such as distribution of pay-as-you-go real-time video over broadband Internet.

**Website**

The author has become the unofficial “keeper” of TEA on the Web. The TEA page hosted at [17] contains contributions from a number of authors who have coded TEA in a wide range of computer languages. Due to its simplicity, TEA can be implemented quite easily in almost any high-level language such as C (or even SQL!). It can also very easily be ported to assembly language on most CPUs for almost any platform or processor. Note that TEA is optimised for 32-bit CPUs with fast shift capabilities such as the Pentium or PowerPC where there are sufficient spare registers to hold all the variables. The long register length avoids the need to use stack or temporary memories, again speeding the algorithm. Full details and tested code samples are in the archive. The contributions are quickly checked by other users and bugs rapidly found. The author is reasonably confident that the code in the archive is reliable. The TEA website currently has source code for:

- TEA ~ ANSI C
- TEA ~ Motorola PowerPC
- TEA ~ Motorola 680 × 0
- TEA ~ Java
- TEA ~ Javascript
- TEA ~ SQL Server
- TEA ~ Macromedia Flash
XTEA ~ ANSI C
XTEA in Intel 16-bit assembler (8086, 80186, 80286)
XTEA in Intel 32-bit assembler (386, 486, Pentium)
XTEA ~ Java

Readers are cordially invited to visit the page and make use of the information there. The page is updated regularly with new contributions, so please check the page from time to time. If you use the resources of the page, please reference it for the benefit of others.

Acknowledgments

The author would like to thank Prof. William Stallings of the Board of Cryptologia, for his kind invitation to write this review paper.

About the Author

Simon Shepherd is Professor of Computational Mathematics in the University of Bradford. His research interests include cryptography, computer security, signal processing, and information management.

References


