Efficient Symmetric-Key Ciphers Based on an NP-Complete Subproblem (Extended Abstract)

Matt Blaze
AT&T Laboratories
Murray Hill, NJ 07974
mab@research.att.com

PRELIMINARY DRAFT - October 3, 1996 - DO NOT DISTRIBUTE OR CITE

Abstract

We describe a simple, efficient, security primitive based on the Luby-Rackoff (Feistel) construction with small random functions. We show that recovering the internal state of the primitive is NP-Complete. We describe two simple symmetric-key cryptosystems based on the primitive. *Turtle* is a recursively-structured variable-length block cipher. *Hare* is a simple keystream generator. The ciphers are efficient on modern computers, and appear to resist various known attacks.

1 Introduction

Most modern public-key ciphers are designed to reduce or at least relate closely to some long-studied problem that is believed to be difficult, such as factoring or finding discrete logarithms of large integers. Modern symmetric-key ciphers, on the other hand, can rarely be reduced to widely-studied hard problems and so are frequently designed specifically to resist known cryptanalytic attacks. As a consequence, the design, analysis and even implementation of symmetric-key ciphers is regarded as exceptionally difficult:

1. The algorithms tend to be conceptually complex, characterized by magic constants, irregular structure and awkward bit-level operations. It is virtually impossible, given a description of most ciphers, to comprehend fully the justification for the various parameters, or even implement the algorithm correctly without resorting to direct comparison with another implementation.

2. Most ciphers are sensitive to even trivial, seemingly inconsequential changes to the details of their construction. A single bit change to a DES S-box, for example, usually renders the cipher less secure.

3. It is not considered especially surprising when an attack against some symmetric-key cipher is discovered.

It is possible to separate cipher design into at least three subproblems: the design of the cipher's *structure*, the choice of *security primitives* used within the structure, and the *keying scheme* that converts key material into parameters for the security primitives. It is worth noting that a weakness in any aspect of a cipher's design can weaken the entire cipher. For example, the two-round Luby-Rackoff[5] block cipher structure is vulnerable to "adaptive" attacks that avoid the need to attack the underlying security primitives. Similarly, even ciphers with a strong structure can be vulnerable to attacks, such as differential cryptanalysis[2], that exploit weaknesses in or unexpected interactions among the security primitives.

It seems worthwhile, then, to find efficient and simple symmetric-key cipher structures that use as a security primitive some function that is already known or suspected to be hard. NP-complete problems seem like good candidates for such ciphers, but previous symmetric-key ciphers do not appear to employ obviously NP-complete subproblems. In this paper, we propose a simple cipher primitive, based on Feistel networks, for which recovery of its internal state given its inputs and outputs is NP-complete. We propose a recursively-structured, variable-length block cipher, called *Turtle*, and a simple keystream generator, called *Hare*, that use the primitive.

2 Random Feistel Networks

Suppose we want to construct a block cipher with a $2n$-bit block size. The four-round Luby-Rackoff construction[5], also called a *Feistel network*[3], suggests a simple structure:

$$\begin{aligned} R &= R + f_1(L) \\ L &= L + f_2(R) \\ R &= R + f_3(L) \\ L &= L + f_4(R) \end{aligned}$$

R and L are the n-bit right and left halves of the cipher block, respectively, and f_1, f_2, f_3 and f_4 are (secret) pseudorandom functions that comprise the security primitives. $+$ usually represents modulo 2^n or bitwise modulo-2 addition (exclusive-or). See Figure 1. Decryption reverses the order of function application (using subtraction instead of addition). A number of modern ciphers, including DES, follow this basic structure, but usually with more than four rounds. Many designs depart from the standard Luby-Rackoff construction by using the same pseudorandom function in all rounds but add additional rounds to compensate for potential weaknesses or interactions in the underlying functions.

Note that if n is small (8 or 16 bits), it is entirely feasible on modern computers for f_1, f_2, f_3 and f_4 to be *random*, as opposed to pseudorandom, functions, defined by a complete table of 2^n inputs and outputs, perhaps with outputs selected from some distribution.

The security of this construction with four rounds is known to be linear in the security factors of the underlying pseudorandom functions [5][7]. A natural question to ask, then, is whether the construction with random functions is any *stronger* than the individual functions themselves. For the purposes of discussion, we define "security" as the complexity of recovering the internal state of the four functions given up to the complete collection of 2^{2n} plaintext/ciphertext pairs.

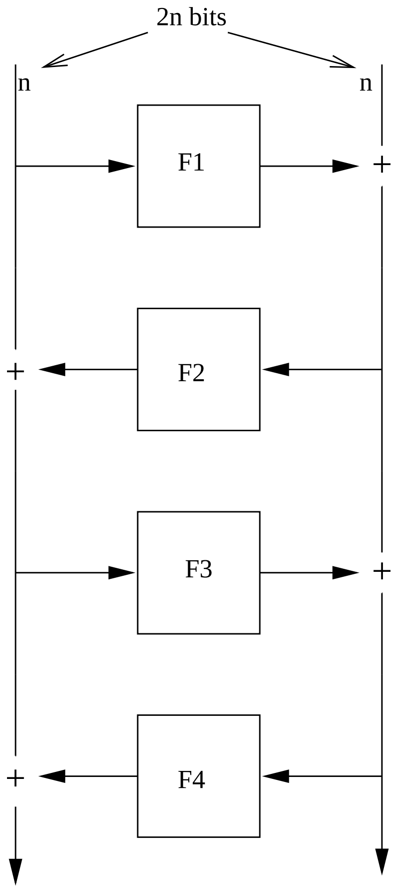

Figure 1: Feistel encryption of $2n$-bits with four n-bit functions

2.1 Cryptanalysis of 3-round Random Feistel Networks is NP-Complete

We define, more formally, the r-round random Feistel network cryptanalysis (r-RFN) problem.

Input: Integer r, A permutation V of the 2^{2n} $2n$-bit integers (derived from the "ciphertext" values for each "plaintext" $P, 0 \leq P < 2^{2n}$) and a collection O of r sets $O_1, O_2, ...O_r$ each consisting of 2^n values $v, 0 \leq v < 2^n$.

Problem: Find r permutations $f_1, f_2, ..., f_r$ on each set in O that act as functions for an r-round Feistel network that implements V, if one exists. The equivalent decision problem asks whether there exists at least one collection of functions that implement V.

(Observe that for fixed r, most permutations on 2^{2n} elements cannot be implemented by any r-round Feistel network with n-bit functions.)

Clearly, r-RFN is in NP, since a guessed solution can be verified by running the Feistel network with each input $0 \leq i < 2^{2n}$ and verifying that each output matches each successive element of V.

We transform Numerical Matching with Target Sums (NMTS) [4] to 3-RFN. We use the notation of Garey and Johnson to denote the parameters of the problem. Note that since NMTS is strongly

NP-complete, B can be bounded to fit some value of n-bits for an instance of 3-RFN. We convert an instance of NMTS to 3-RFN by setting $O_1 = X$ and $O_3 = Y$, padding with extra zero-value elements as needed to make $|B| = |O_1| = |O_3| = 2^n$. We fix all elements of O_2 to zero. We assume in our Feistel construction that f_1 and f_3 are added to the right word of the cleartext and f_2 is added to the left word. Now we create V from the B vector in the NMTS problem by setting V with ciphertext values $< C_l, C_r >$ for each plaintext $< P_l, P_r >$ such that $C_l = P_l$ and $C_r = P_r + B_{P_l}$. The 3-RFN oracle finds a solution iff there is a solution to the corresponding NMTS problem.

It is trivial to see that r-RFN remains NP-complete for all fixed $r \geq 3$, by fixing the values in each $O_4 ... O_r$ to zero. (4-RFN is preferable to 3-RFN as a cipher because 3-round Luby-Rackoff is known to be weak in other respects.)

2.2 RFNs and Security

NP-completeness of a primitive is not sufficient to prove security of a cipher on which it is based. Our formulation of the cryptanalysis problem might not capture all the circumstances under which the cipher might be considered "broken". A more difficult, and open, problem, is that the NP-completeness of RFN says nothing about its *average*-case hardness or the suitability of the hard instances for use as a cipher primitive.

Intuitively, the hard instances for 4-RFNs appear to be those for which each target sum has many pairs of values that sum to it. Functions that are themselves permutations of the values $0...2^n$ therefore, may be good candidates for use as strong cryptographic keys.

Note that with 3-RFNs, however, the internal state of all three functions is easy to recover when f_2 is a permutation[9] (but not if f_2 is constant for all inputs, as it was in our reduction). While the 4-RFN structure with permutations does not appear to have this vulnerability, identifying a distribution of strong keys that is guaranteed to generate hard instances remains an open problem.

3 The Turtle Block Cipher

We propose a simple block cipher based on Feistel networks of random permutations.

The basic 4-RFN structure, while possibly quite secure against recovery of its internal state, is not suitable in practice for use as a block cipher by itself. The requirement for $4n(2^n)$ bits of private memory to yield a block length of $2n$ bits imposes a practical block length limit of 16- or perhaps at most 32-bits on real computers. Of course, block ciphers with such small block sizes have security problems not related to their internal "strength," *e.g.,* it may be feasible for an attacker to collect a complete or partial "codebook" of ciphertext/plaintext pairs.

Turtle is a variable-block size cipher that uses 4-RFNs (with modulo-2 addition) as its security primitive. It employs a recursive structure to expand the block size to any 2^n words. We define Turtle block encryption by the following pseudocode:

```
Turtle(X) = {
        if length(X) = w
        {
                n = n + 1
                return f_n(X)
        }
```

```
        else {
                R = X_right
                L = X_left
                R = R ⊕ Turtle(L)
                L = L ⊕ Turtle(R)
                R = R ⊕ Turtle(L)
                L = L ⊕ Turtle(R)
                return R|L
        }
}
```

w is the word size in bits. X_{left} denotes the left half of bitstring X; X_{right} denotes the right half. $\oplus$ indicates bitwise XOR; $|$ denotes bitstring concatenation. f_i indicates the ith function in a table of random functions on w-bit words; n is a global variable initialized to zero and incremented for each block. k^2 functions are required for a block length of kw bits. Note that the functions serve as the "key." The structure is shown graphically in Figure 2. The decryption function's structure follows directly from the encryption function, with the order of the outermost recursive calls reversed.

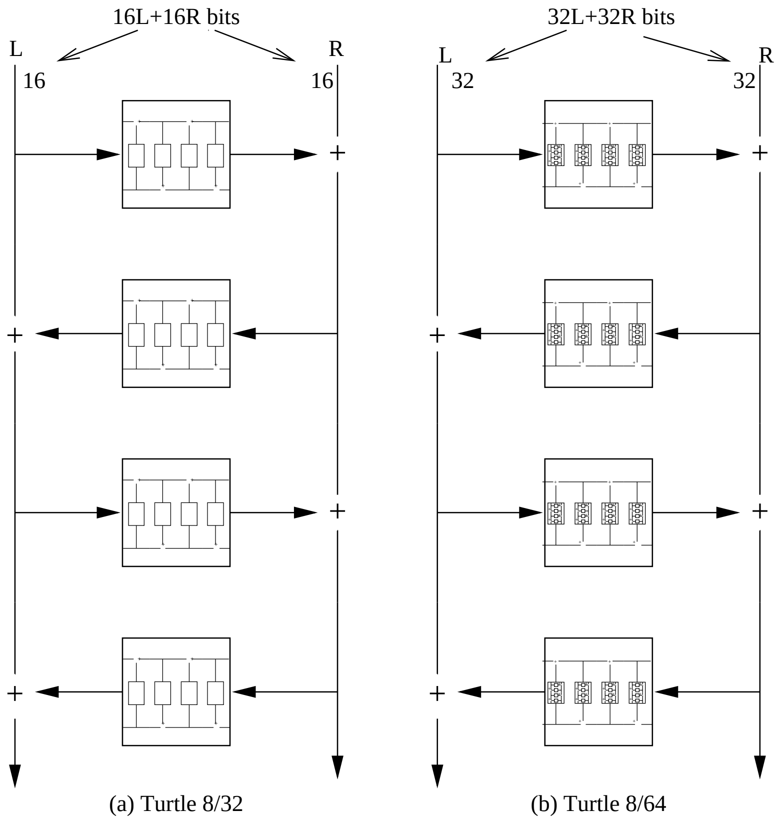

Figure 2: Turtle Structure

By convention, we distinguish between various block lengths and random function word sizes by denoting Turtle on an n bit block with w-bit random functions as "Turtle w/n". Turtle w/n requires $w2^w(n/w)^2$ bits of key material. Turtle 8/64 uses 64 random functions and a total of 131,072 (16K bytes) bits of key material. Turtle 8/128 uses 64K bytes of key material. Turtle 16/x keys are larger; Turtle 16/64 keys are 4,194,240 bits (512K bytes) long.

Obviously, Turtle's security parameter (key) is quite long, which implies that most applications will use smaller "seed keys" in practice. The reference implementation gives a suggested key expansion function based on the *Hare* stream cipher (described in the next section). The (short) key parameter is used to create $(n/w)^2$ permutations on 2^w elements, which are used as the key. The key expansion function is similar in some respects to the (alleged) RC-4 keying algorithm[1], in that it uses the key parameter to swap the elements of the tables. We caution that this function is rather *ad hoc* in structure and as such could be the cipher's weakest aspect.

The complete C language Turtle "reference implementation" is given in the Appendix.

The speed of Turtle in software is roughly comparable to that of other block ciphers, provided enough memory is available to hold the key without paging. For example, the reference implementation of Turtle 8/64 encrypts approximately 60,000 blocks per second on a 100MHz Pentium.

Some improvements on the 4-RFN-based Turtle may be possible. In particular, it may be possible to reduce the size of the key parameter and the number of function calls by using a 3-RFN in all but the outermost recursive layer.

4 The Hare Keystream Generator

The 4-RFN construction can also be used as a simple keystream generator, *e.g.,* by encryption of an increasing counter. Again, the small block sizes that can be used in practice limit the usefulness of this construction by itself. The cycle length is quite short (2^{2n}), and the output becomes increasingly biased as the counter approaches its maximum value. An obvious approach is to use the recursive Turtle construction to construct a more reasonable block length, but we usually expect stream generators to have better performance than block ciphers.

Hare is a keystream generator based on encryption of a counter with a single pass through a 4-RFN. The key consists of two sets of 4-RFNs, one *active* and the other *inactive*. The active RFN is used to encrypt the counter, returning the "left" word of ciphertext as the keystream. The "right" word is used to permute successive elements of the inactive RFN. When the last element of the inactive RFN has been permuted (after 2^n4 words of keystream have been generated), the active and inactive RFNs are swapped. Assuming the 4-RFN primitive is secure, each newly created 4-RFN should be polynomially secure with respect to the 4-RFN from which it was generated.

Hare is defined by the following pseudocode:

HareKeyStream = {
$R = Rcounter$
$L = Lcounter$
$R = R \oplus f_{1,active}(L)$
$L = L \oplus f_{2,active}(R)$
$R = R \oplus f_{3,active}(L)$
$L = L \oplus f_{4,active}(R)$
swap $f_{Rcounter,inactive}(Lcounter), f_{Rcounter,inactive}(R)$

increment *Rcounter* mod $MAXWORD$
if (*Rcounter* $= 0$) increment *Lcounter* mod $MAXWORD$
if (*Lcounter* > 3) {
  Lcounter=0
  Rcounter=0
  swap $active, inactive$
}
$keystream = L$
}

A reference implementation is included in the Appendix.

5 Conclusions

Unlike the vast majority of modern symmetric-key ciphers (with the notable exceptions of RC4 [1] and RC5 [8]), Turtle and Hare are conceptually very simple. There are no constants, bit permutations or hard-coded tables. Turtle's recursive structure provides a natural mechanism for extending its block length to whatever is required. (The recent MISTY cipher [6], while not based on RFNs or a variable block length, also employs a recursive structure). The simple structure and close relationship to an NP-complete problem in Turtle and Hare should facilitate their analysis. They appear to resist known attacks such as differential cryptanalysis.

Perhaps the most important open problem is that of finding keys that generate hard instances of the underlying problem. Permutations (with 4-RFNs) are intuitively attractive in this regard, but we have no proof of this. (A forthcoming paper will examine the use of recursive Feistel-like constructions in which the inner functions are not reversible permutations).

6 Acknowledgements

The author is indebted to Ross Anderson, Whit Diffie, Bruce Schneier, and David Wagner for their helpful comments. Serge Vaudenay pointed out the weakness of 3-RFNs when f_2 is a permutation.

References

[1] Anonymous. (Algorithm attributed to R. Rivest). *The RC-4 algorithm.* Posting to `sci.crypt` Usenet group. September 1994.

[2] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag. 1993.

[3] H. Feistel. "Cryptography and Computer Privacy." *Scientific American.* Vol. 228, 1973.

[4] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman. 1979.

[5] M. Luby and C. Rackoff. "How to Construct Pseudorandom Permutations from Pseudorandom Functions." *SIAM J. Comput.,* 17 (1988), 373–386.

[6] Mitsuru Matsui. "Block encryption algorithm MISTY." Technical report of *IEICE ISEC96-11* (1996-07).

[7] Moni Naor and Omer Reingold. "On the Construction of Pseudorandom Permutations: Luby-Rackoff Revisited." *To appear.* 1996.

[8] Ron Rivest. "The RC-5 Encryption Algorithm." *Proc. Leuven Workshop on Cryptographic Algorithms.* December 1994.

[9] Serge Vaudenay. *Private communication.* October 1996.

Appendix: Turtle Reference Implementation

```
------turtle.h
/*
 * turtle/hare reference implementation
 *    Matt Blaze
 *    September, 1996
 */
#define TURTLEMAXN 16 /* max blocksize in TURTLEWORDs */

/* these next three must be changed together */
#define TURTLEBITS 8 /* word size in bits */
#define NTURTLEWORDS 256 /* 2^TURTLEBITS */
#define TURTLEWORD unsigned char /* unsigned data type with TURTLEBITS bits */

/*
 * basic turtle key data structure
 */
typedef struct {
        int n; /* the blocksize (must be power of 2 and at least 2) */
        int rr[4]; /* the outer round subkey indexes */
        TURTLEWORD sbox[TURTLEMAXN*TURTLEMAXN][NTURTLEWORDS]; /* key tables */
} TK;

/*
 * basic hare key data structure
 */
typedef struct {
        TURTLEWORD r, l;/* current r and l counter state */
        int table;      /* current table (0 or 1) */
        TURTLEWORD sbox[2][4][NTURTLEWORDS];
} HK;

int turtle_key();
```

```
int turtle_encrypt();
int turtle_decrypt();
int hare_key();
TURTLEWORD hare_stream();

------turtle.c
/*
 * turtle.c
 *    Matt Blaze, September, 1996
 *
 * This is the basic turtle/hare cipher reference implementation
 * with a simple key schedule.
 *
 * Turtle blocksize can be any power of two words >= 4;
 * the wordsize is hardcoded in turtle.h.  8 bits is the default.
 *
 * This code turtle encrypts 8-word blocks on a p100 at about 3Mbps.  It
 * can probably be made a couple times faster by unrolling the recursive
 * calls.  Hare runs at about 9Mbps.
 */

#include <stdio.h>
#include "turtle.h"

static int r;
static keyperm();
static int r_turtle_encrypt();

/*
 * Basic turtle encrypt
 *  (encrypts blk in place)
 */
int turtle_encrypt(blk,key)
     TURTLEWORD *blk;
     TK *key;
{
        int nn, i;
        TURTLEWORD buf[TURTLEMAXN];

        if ((key==NULL) || (blk==NULL))
                return -1;
        r=0;
        nn=key->n/2;
        r_turtle_encrypt(&(blk[0]),buf,nn,key);
```

```
        for (i=0; i<nn; i++)
                blk[i+nn] ^= buf[i];
        r_turtle_encrypt(&(blk[nn]),buf,nn,key);
        for (i=0; i<nn; i++)
                blk[i] ^= buf[i];
        r_turtle_encrypt(&(blk[0]),buf,nn,key);
        for (i=0; i<nn; i++)
                blk[i+nn] ^= buf[i];
        r_turtle_encrypt(&(blk[nn]),buf,nn,key);
        for (i=0; i<nn; i++)
                blk[i] ^= buf[i];
        return 0;
}

/*
 * Basic turtle decrypt
 *  (decrypts blk in place)
 */
int turtle_decrypt(blk,key)
     TURTLEWORD *blk;
     TK *key;
{
        int nn, i, rr;
        unsigned char buf[TURTLEMAXN];

        if ((key==NULL) || (blk==NULL))
                return -1;
        nn=key->n/2;
        r=key->rr[3];  /* we have to use the key schedule backwards */
                    /* but only for the OUTERMOST recursive shell */
        r_turtle_encrypt(&(blk[nn]),buf,nn,key);
        for (i=0; i<nn; i++)
                blk[i] ^= buf[i];
        r=key->rr[2];
        r_turtle_encrypt(&(blk[0]),buf,nn,key);
        for (i=0; i<nn; i++)
                blk[i+nn] ^= buf[i];
        r=key->rr[1];
        r_turtle_encrypt(&(blk[nn]),buf,nn,key);
        for (i=0; i<nn; i++)
                blk[i] ^= buf[i];
        r=0;
        r_turtle_encrypt(&(blk[0]),buf,nn,key);
        for (i=0; i<nn; i++)
```

```
                blk[i+nn] ^= buf[i];
        return 0;
}

/*
 * create turtle key from short key.
 * n must be a power of 2 and >= 4.
 */
int turtle_key(shortkey,len,key,n)
     TURTLEWORD *shortkey;
     int len;
     TK *key;
     int n;
{
        int i, j, nn;
        TURTLEWORD other, t;
        HK harekey;

        if ((n<4) || (n>TURTLEMAXN) || (key == NULL) || (shortkey == NULL))
                return -1;
        nn=n*n;
        /* first create a hare key from the shortkey */
        hare_key(shortkey,len,&harekey);
        /* use hare to permute the real sboxes */
        for (j=0; j<nn; j++) {
                for (i=0; i<NTURTLEWORDS; i++) {
                        key->sbox[j][i] = i;
                }
                for (i=0; i<NTURTLEWORDS; i++) {
                        other = hare_stream(&harekey);
                        t = key->sbox[j][i];
                        key->sbox[j][i] = key->sbox[j][other];
                        key->sbox[j][other] = t;
                }
        }
        key->n=n;
        key->rr[3] = nn/4*3;
        key->rr[2] = nn/4*2;
        key->rr[1] = nn/4;
        key->rr[0] = 0;
        return 0;
}

/*
```

```
 * Basic hare stream generator
 *  (returns one TURTLEWORD)
 */
TURTLEWORD hare_stream(key)
     HK *key;
{
        TURTLEWORD r, l, t;
        int table, otable;

        r = key->r;
        l = key->l;
        table = key->table;
        otable = 1-table;
        r^=key->sbox[table][0][l];
        l^=key->sbox[table][1][r];
        r^=key->sbox[table][2][l];
        l^=key->sbox[table][3][r];
        t = key->sbox[otable][key->r][key->l];
        key->sbox[otable][key->r][key->l] = key->sbox[otable][key->r][r];
        key->sbox[otable][key->r][r] = t;
        key->l = (key->l + 1) % NTURTLEWORDS;
        if (key->l == 0) {
                key->r = (key->r + 1) % NTURTLEWORDS;
        }
        if (key->r > 3) {
                key->r = 0;
                key->l = 0;
                key->table = otable;
        }
        return l;
}

/*
 * create hare key from short key
 */
int hare_key(shortkey,len,key)
     TURTLEWORD *shortkey;
     int len;
     HK *key;
{
        if ((key == NULL) || (shortkey == NULL))
                return -1;
        /* first create the tables from the shortkey */
        keyperm(key->sbox[0],shortkey,len,4);
```

```
        /* do it again for the other set */
        keyperm(key->sbox[1],shortkey,len,4);
        key->table = 0;
        key->r = 0;
        key->l = 0;
        return 0;
}

/*********************************************
 * support functions - not part of interface *
 *********************************************/

/*
 * recursive turtle function
 */
static int r_turtle_encrypt(in,out,n,key)
     TURTLEWORD *in;
     TURTLEWORD *out;
     int n;
     TK *key;
{
        int nn, i;
        TURTLEWORD buf[TURTLEMAXN];

        if (n==2) { /* this is the basic lookup */
                out[1] = in[1] ^ key->sbox[r++][in[0]];
                out[0] = in[0] ^ key->sbox[r++][out[1]];
                out[1] ^= key->sbox[r++][out[0]];
                out[0] ^= key->sbox[r++][out[1]];
        } else {  /* recurse */
                nn=n/2;
                r_turtle_encrypt(&(in[0]),buf,nn,key);
                for (i=0; i<nn; i++)
                        out[i+nn] = in[i+nn] ^ buf[i];
                r_turtle_encrypt(&(out[nn]),buf,nn,key);
                for (i=0; i<nn; i++)
                        out[i] = in[i] ^ buf[i];
                r_turtle_encrypt(&(out[0]),buf,nn,key);
                for (i=0; i<nn; i++)
                        out[i+nn] ^= buf[i];
                r_turtle_encrypt(&(out[nn]),buf,nn,key);
                for (i=0; i<nn; i++)
                        out[i] ^= buf[i];
        }
```

```
        return 0;
}

/* Simple key permutation expand function.
 * This is really more of an example than anything else.
 * Generate n permutations on 2^WORDBITS elements from the cryptovariable.
 * This is approximately similar to the RC-4 permutation generator, but
 * we go through each table WORDBITS times, which smoothes out the early
 * swaps and does a total of x log x swaps in each permutation.
 */
static keyperm(sbox,key,len,n)
     TURTLEWORD sbox[][NTURTLEWORDS];
     TURTLEWORD *key;
     int len;
     int n;
{
        int a, b, i, j, k;
        TURTLEWORD t;

        for (b=0; b<n; b++) {
                for (i=0; i<NTURTLEWORDS; i++) {
                        sbox[b][i]=i;
                }
        }
        j=len;
        k=0;
        for (b=0; b<n; b++) {  /* for each keygen sbox */
                for (a=0; a<TURTLEBITS; a++) {  /* n times around for each */
                        for (i=0; i<NTURTLEWORDS; i++) { /* swap w/ element */
                                j = (j + key[k] +
                                     sbox[b][(a*b+i)%NTURTLEWORDS])
                                        %NTURTLEWORDS;
                                t = sbox[b][i];
                                sbox[b][i] = sbox[b][j];
                                sbox[b][j] = t;
                                k=(k+1)%len;
                        }
                }
        }
}
```