The Diamond2 Block Cipher

by Michael Paul Johnson

Abstract—**The Diamond2 Block Cipher is a royalty-free, symmetric-key encryption algorithm based on a combination of nonlinear functions. This block cipher may be implemented in hardware or software. Diamond uses a block size of 128 bits and a variable length key. A faster variant of Diamond2, called Diamond2 Lite, uses a block size of 64 bits.**

Index Terms—**Diamond2, Diamond, encryption, cryptography, cryptanalysis, cryptology, computer security, communications security, cipher.**

I. INTRODUCTION

General symmetric key block ciphers have numerous applications in computer security, communications security, detection of data tampering, and creation of message digests for authentication purposes. The longer any one such algorithm is used, and the more use it gets, the greater the incentive to break it, and the greater the probability that methods will be devised to break the algorithm. For example Michael J. Wiener has shown that breaking DES is within the capabilities of many nations and corporations [1]. This sort of reduction in the relative security of DES was anticipated several years ago. One proposed solution is the International Data Encryption Algorithm (IDEA™) cipher [2], which was described in [3] and [4] as the Improved Proposed Encryption Standard (IPES). Another one is the MPJ Encryption Algorithm [5], which evolved to the Diamond2 Block Cipher. In the field of cryptography, it is good to have many strong block ciphers available.

II. DESIGN OF DIAMOND2

Diamond2 was designed to be strong enough to provide security for the foreseeable future. It was also designed to be easy to generate keys for, and to be practical to implement in hardware, software, or in a hybrid implementation.

A. Strength

Three major factors influence the strength of a block cipher: (1) key length (and key setup time), (2) block size, and (3) resistance of the algorithm to attacks other than brute force (such as differential cryptanalysis) [3] [6]. The key length is variable to allow you to select your own trade-off between security and volume of keying material needed. The block size is chosen to make brute force attacks using precomputed tables require an obviously intractable amount of data storage.

Diamond2 uses a variable length key. The use of at least a key with at least 128 bits of entropy is recommended for long term protection of very sensitive data, as a hedge against the possibility of computing power increasing by several orders of magnitudes in the coming years.

The block size for the Diamond2 Block Cipher is fixed at 128 bits, because larger block sizes are unlikely to make any practical difference in security, and because this is a convenient binary multiple (16

bytes). Diamond2 Lite has a block size of 64 bits because this is good enough for most applications, and because it allows a much faster total avalanche effect and greater software speed than the 128-bit block size.

The problem of making sure that there is no known attack that is more efficient than brute force is much more difficult than simply selecting sizes for keys and blocks. This is attempted by creating a composite function of simpler nonlinear functions in such a way that the internal intermediate results cannot be solved for and such that there is a strong dependence of every output bit on every input bit and every key bit. Another important consideration is that the author and inventor keep up with significant developments in cryptanalysis. This last requirement is only partially met, in that a large percentage of significant cryptanalysis technology is shrouded in secrecy.

An ideal 128 bit block cipher would use a **z** bit key to select one of $2^{\mathbf{z}}$ functions from the set of all one to one and onto functions that map one input block of 128 bits to one output block of 128 bits. Ideally, these $2^{\mathbf{z}}$ functions would be the most nonlinear and difficult to analyze functions out of the $(2^{128})!$ possible functions. In practice, the key selects one of $2^{\mathbf{z}}$ functions from an arbitrary selection of possible functions.

The use of purely nonlinear functions makes a large portion of mathematical tools ineffective for cryptanalysis. The tools that remain are defeated by ensuring adequate complexity in terms of time and memory requirements that solutions are not practical.

B. Ease of Key Generation

Key generation should be as simple as generating a random number by measuring some random physical process. Since there is no complex or secret strong key selection process, distributed key management protocols are practical. Distributed key management is preferable in many applications to centralized key management because there is no single point of failure at which the whole system could be compromised. (This doesn't preclude centralized key management, of course.)

C. Practical in Hardware or Software

The prototype of the Diamond2 Block Cipher is implemented in a program for a personal computer or workstation. When properly implemented in hardware, Diamond2 should not significantly slow down any practical digital data stream. On the other hand, setting up a new key need not be as fast as the encryption and decryption operations, since (1) key change operations are less frequent than encryption and decryption operations, and (2) a slower key setup operation discourages brute force attacks. The key setup algorithm used by Diamond2 intentionally requires a large number of sequential steps to increase the cost of brute force key searches.

III. BASIS OF DESIGN

The thought process that went into the design of Diamond2 is based on the following ideas:

1. Linear functions and combinations of functions can often be solved analytically in ways that are not obvious to the cipher designer, and should be avoided. This includes standard arithmetic functions, math in finite fields, and Boolean arithmetic.

2. Reversible block ciphers with a block size of **n** bits can be viewed as a simple substitution cipher on an alphabet of $\mathbf{2^n}$ characters, with a key that selects the permutation used.

3. Simple substitution ciphers can be represented with a look-up table or array, but in practice the array required is too big to fit comfortably in a computer's memory.

4. An adequate subset of the oversized look-up table can be simulated by simply interleaving rounds of substitution of sub-blocks with bit permutations that serve to spread functional dependencies across sub-block boundaries.

IV. DESCRIPTION OF ALGORITHM

Although I will attempt an accurate English description of the Diamond2 Block Cipher, a more concise description may be found in the source code of the reference implementation, below. In case of conflict, believe the source code, since that is what I tested and analyzed while validating this cipher.

The Diamond2 Block Cipher consists of three main parts: (1) key scheduling, (2) substitution steps, and (3) permutation steps. Encryption and decryption both consist of **n** rounds of substitution operations, where **n** is at least 10. Each substitution operation takes each of the 16 input bytes of 8 bits each, and substitutes another byte for it. This done with the contents of the substitution array for that byte position and round number. The key scheduling operation fills the internal substitution arrays based on the key. Between each substitution, a fixed permutation step uses a bit selection process to make each output byte a function of eight different input bytes. Unlike DES, every round alters every byte of the input block (instead of just half of the input block). After 5 rounds, every bit of the output block is a nonlinear function of every bit of the input block and every bit of the key. The additional rounds after the fifth round serve to ensure that solving for the contents of the individual substitution arrays is more work than a brute force attack on the cipher. They also serve to increase the number of possible functional relationships that the key selects from, thus making this algorithm closer to the ideal block cipher, and making cryptanalysis more difficult.

A. Key Scheduling

There is one substitution array for each of the 16 bytes of the encryption block for each round. For a ten round implementation of Diamond2, 160 substitution arrays are to be filled. Each of the 160 arrays contains 256 elements of one byte each. It is convenient to look at the set of substitution arrays as one three dimensional array, indexed by round, byte position within the 16 byte encryption block, and input byte value. A similarly indexed inverse substitution array is used during decryption. For the substitution to be reversible, each of the 256 possible values of an 8 bit byte must occur exactly once in the array. The process used to make this happen consists of five processes: (1) array filling, (2) element placement, (3) pseudorandom key expansion, (4) pseudorandom number normalization, and (5) array inversion. Although key scheduling can be done more quickly in a dedicated hardware implementation, a more economical hybrid design would do the key scheduling in firmware and the actual encryption or decryption in hardware.

Array filling is simply a nested loop where all 160 substitution arrays are filled. It is concisely expressed in this pseudo code:

```
For rounds := 1 to n
   For byte position := 1 to 16
      For element value := 255 down to 0
         Place this element.
```

Element placement is done by placing the current element in one of the unfilled positions in the current array. The unfilled positions of the current array are numbered from 0 to the value of the element being placed. A number in this same range is then selected by generating a pseudorandom number normalized to this much smaller range. This offset is used to place the current element and mark that location as having been filled. In the trivial case where there is only one more unfilled element, no pseudorandom number is generated.

Pseudorandom key expansion uses a simple method to provide key dependent bits as needed to place array elements. A pointer is set to the first 8-bit byte of the key. A 32 bit CRC accumulator is set to all ones (FFFFFFFF hexadecimal). This initial value is used rather than all zeros so that an all zero external key would not be weak. Every time a pseudorandom number is requested, the CRC is updated using the CCITT CRC-32 [7] using the byte in the previously filled array indexed by the key byte pointed to by the pointer. In the special case of the first array filled, the CRC is updated directly by the key byte pointed to by the pointer. The pointer is then moved to the next key byte. After the pointer is moved beyond the end of the last key byte, the CRC is updated with the least significant byte of the size of the key (in bytes), then with the next to least significant byte of the size of the key (in bytes), then the pointer is moved back to the first byte of the key. If the actual key size used is not a multiple of 8 bits, then the unused bits of the last key byte are set to 1, with the used bits occupying the least significant bits of the byte.

Although no upper limit is explicitly given for key size, increasing the key size provides no significant increase in security if more than approximately 28 672 · **n** bits are used, where **n** is the number of rounds used. This upper limit is large enough that even fictional computers [8] would have difficulty with a brute force attack.

To normalize the 32 bit accumulator value to the desired number range from 0 to **n**, first perform a logical “and” operation on the accumulator with the value $2^{m}-1$, where **m** is the smallest integer value such that $2^{m}-1 \geq \mathbf{n}$. This will select the minimum number of bits required to cover the range needed. If the resulting value is less than or equal to **n**, use it. If it is not, then repeat the above process with a new pseudorandom number. If, after 97 attempts the value is still not in range (a very low probability condition), simply subtract **n** from the value and use it.

If the decryption mode of Diamond2 is to be used, calculate the inverse substitution arrays directly from the encryption substitution arrays as follows:

```
For rounds := 1 to n
   For byte position := 1 to 16
      For k := 0 to 255 do
         inverse array[array[k]] := k
```

Note that this type of inverse substitution array computation, together with the inverse permutations are what allow the greater effect per round than the traditional involution operation of Fiestel type block ciphers like DES and Blowfish.

B. Substitution

In each substitution round, each byte of the input block is replaced with the contents of the substitution array for that round, byte position, and byte value. For decryption, the same operation is performed with the inverse substitution array. In a hardware implementation, this is can be done quickly by simply addressing static RAM. Note that the substitution arrays used in the Diamond2 Block Cipher are different from the S-Boxes used in ciphers like DES, in that (1) they are much larger, (2) there are more of them, and (3) they are not used in conjunction with a simpler operation with a key that could be solved for with differential cryptanalysis.

C. Permutation

Between each substitution round, a fixed permutation is performed. The purpose of this permutation step is to increase the effective block size of the cipher by making each output byte a function of 8 input bytes by simply selecting one bit from each of 8 input bytes. Every bit of the input block is used exactly once in the output block. In a hardware, this can be done with literal wire crossings. In software, efficiency is gained by ensuring that every bit ends up in the same position relative to a byte boundary as where it started.

The specific permutation used for encryption takes the least significant bit of each byte from the input byte in the same position. The next most significant bit is taken from the input byte indexed as one byte higher (mod 16). The next most significant bit is taken from the input byte indexed as two higher (mod 16), and so on. For decryption, the inverse of this operation is the same, except the byte positions used are one byte lower (mod 16) instead of higher.

After 2 rounds, every output byte is a function of 8 input bytes and all key bytes (if the key is less than 4080 bytes, which is likely). After 3 rounds, every output byte is a function of 15 input bytes and the key. After 4 rounds, every output byte is a function of every input byte and the key. The minimum of 6 additional rounds are intended to make cryptanalysis more difficult.

V. REFERENCE SOURCE CODE

The following ANSI C or C++ source code fragment is a more concise and accurate description of the Diamond2 Block Cipher than the above English description.

A. DIAMOND2.H

```
/* diamond2.h -- program interface to the Diamond2 and Diamond2 Lite Block
   Ciphers.  This file dedicated to the Public Domain by Mike Johnson, the
   author.*/

extern void set_diamond2_key(byte *external_key, /* Variable length key */
                        uint key_size,      /* Length of key in bytes */
                        uint rounds,        /* Number of rounds to use (5 to 15
                                            for Diamond, 4 to 30 for Diamond Lite) */
                        boolean invert,     /* true if mpj_decrypt may be called. */
                        int block_size);    /* 16 for Diamond; 8 for Diamond Lite. */
/* Call before the first call to diamond2_encrypt_block() or diamond2_decrypt_block */

extern void diamond2_encrypt_block(byte *x, byte *y);
/* Call set_diamond2_key() with a block_size of 16 before first calling
```

```
   diamond2_encrypt_block().   x is input, y is output.
*/

extern void diamond2_decrypt_block(byte *x, byte *y);
/* Call set_diamond2_key() with a block_size of 16 before first calling
   diamond2_decrypt_block().   x is input, y is output.
*/

extern void lite2_encrypt_block(byte *x, byte *y);
/* Call set_diamond2_key() with a block_size of 8 before first calling
   lite2_encrypt_block().   x is input, y is output.
*/

void lite2_decrypt_block(byte *x, byte *y);
/* Call set_diamond2_key() with a block_size of 8 before first calling
   lite2_decrypt_block().   x is input, y is output.
*/

extern void diamond2_done(void);
/* Clears internal keys.  Call after the last call to
diamond2_encrypt_block() or diamond2_decrypt_block() with a given key.  */
```

B. DIAMOND2.CPP

```
/* diamond2.c - Encryption designed to exceed DES in security.
   This file and the Diamond2 and Diamond2 Lite Block Ciphers
   described herein are hereby dedicated to the Public Domain by the
   author and inventor, Michael Paul Johnson.  Feel free to use these
   for any purpose that is legally and morally right.  The names
   "Diamond2 Block Cipher" and "Diamond2 Lite Block Cipher" should only
   be used to describe the algorithms described in this file, to avoid
   confusion.

   Disclaimers:  the following comes with no warranty, expressed or
   implied.  You, the user, must determine the suitability of this
   information to your own uses.  You must also find out what legal
   requirements exist with respect to this data and programs using
   it, and comply with whatever valid requirements exist.
*/
#include <stdio.h>
#include <stdlib.h>
#ifdef UNIX
#include <memory.h>
#else
#include <mem.h>
#endif
#include "def.h"
#include "diamond2.h"
#include "crc.h"

static byte *key = NULL;
static uint keysize;
static uint keyindex;
static uint roundsize;          /* Number of bytes in one round of substitution boxes. */
static int blocksize;           /* Number of bytes in a block. */
static unsigned long accum;
static uint numrounds;
static byte *s = NULL;             /* Substitution boxes. */
static byte *si = NULL;            /* Inverse substitution boxes. */

static uint keyrand(uint max_value, byte *sbox)    /* Returns uniformly distributed
pseudorandom */
    {                           /* value based on key[], sized keysize */
    uint prandvalue, i;              /* Change from Diamond to Diamond 2: use of */
    unsigned long mask;              /* sbox (previous 256-byte permutation array)*/

    if (!max_value) return 0;
    mask = 0L;              /* Create a mask to get the minimum */
    for (i=max_value; i > 0; i = i >> 1)    /* number of bits to cover the */
        mask = (mask << 1) | 1L;        /* range 0 to max_value. */
    i=0;
```

```
    do
        {
        if (sbox)
            accum = crc32(accum, sbox[key[keyindex++]]);
        else
            accum = crc32(accum, key[keyindex++]);
        if (keyindex >= keysize)
            {
            keyindex = 0;   /* Recycle thru the key */
            accum = crc32(accum, (keysize & 0xFF));
            accum = crc32(accum, ((keysize >> 8) & 0xFF));
            }
        prandvalue = (uint) (accum & mask);
        if ((++i>97) && (prandvalue > max_value))   /* Don't loop forever. */
            prandvalue -= max_value;                /* Introduce negligible bias. */
        }
    while (prandvalue > max_value); /* Discard out of range values. */
    return prandvalue;
    }

static void makeonebox(uint i, uint j, byte *sbox)
    {  /* Change from Diamond to Diamond 2: use of sbox. */
           /* sbox is either NULL or a pointer to the previously filled array. */
    int n;
    uint pos, m, p;
    boolean filled[256];

    for (m = 0; m < 256; m++)   /* The filled array is used to make sure that */
        filled[m] = false;      /* each byte of the array is filled only once. */
    for (n = 255; n >= 0 ; n--) /* n counts the number of bytes left to fill */
        {
        pos = keyrand(n, sbox); /* pos is the position among the UNFILLED */
                                /* components of the s array that the */
                                /* number n should be placed.  */
        p=0;
        while (filled[p]) p++;
        for (m=0; m<pos; m++)
            {
            p++;
            while (filled[p]) p++;
            }
        *(s + (roundsize*i) + (256*j) + p) = n;
        filled[p] = true;
        }
    }

void set_diamond2_key(byte *external_key, uint key_size, uint rounds,
    boolean invert, int block_size)
/* This procedure generates internal keys by filling the substitution box array
  s based on the external key given as input.  It DOES take a bit of time. */
  {
    uint i, j, k;
    byte *sbox;

    if (s) diamond2_done();
    numrounds = rounds;
    if (block_size == 8)
        {
        blocksize = 8;
        roundsize = 2048U;
        if (numrounds < 3)
            {
            puts("Numrounds out of range in set_diamond2_key()");
            exit(10);
            }
        }
    else if (block_size == 16)
        {
        blocksize = 16;
        roundsize = 4096U;
        if (numrounds < 5)
            {
            puts("Numrounds out of range in set_diamond2_key()");
            exit(10);
            }
```

```
        }
    else
        {
        puts("Unsupported block size in set_diamond2_key()");
        exit(11);
        }

    if ((numrounds * blocksize) > 255)
        {
        puts("Numrounds out of range in set_diamond2_key()");
        exit(10);
        }
    if (BuildCRCTable())
        {
        puts("Not enough memory.");
        exit(5);
        }
    s=(byte *) malloc(numrounds * roundsize);
    if (!s)
        {
        puts("Out of memory.");
        exit(5);
        }
    key = external_key;
    keysize = key_size;
    keyindex = 0;
    accum = 0xFFFFFFFFL;

    sbox = NULL;
    for (i = 0; i < numrounds; i++)
        {
        for (j = 0; j < blocksize; j++)
            {
            makeonebox(i, j, sbox);
            sbox = s + ((roundsize * i) + (256 * j));
            }
        }
    if (invert)
        {   /* Fill the inverse substitution box array si.  It is not
               necessary to do this unless the decryption mode is used.  */
        si=(byte *) malloc(numrounds * roundsize);
        if (!si)
            {
            puts("Out of memory.");
            exit(5);
            }
        for (i = 0; i < numrounds; i++)
            {
            for (j = 0; j < blocksize; j++)
                {
                for (k = 0; k < 256; k++)
                    {
                    *(si + (roundsize * i) + (256 * j) + *(s + (roundsize * i) + (256 * j) +
k)) = k;
                    }
                }
            }
        }
    }

static void permute(byte *x, byte *y)   /* x and y must be different.
  This procedure is designed to make each bit of the output dependent on as
  many bytes of the input as possible, especially after repeated application.
  Each output byte takes its least significant bit from the corresponding
  input byte.  The next higher bit comes from the corresponding bit of the
  next higher input byte.  This is done until all bits of the output byte
  are filled.
*/
    {
    y[0] = (x[0] & 1) | (x[1] & 2) | (x[2] & 4) |
            (x[3] & 8) | (x[4] & 16) | (x[5] & 32) |
            (x[6] & 64) | (x[7] & 128);
    y[1] = (x[1] & 1) | (x[2] & 2) | (x[3] & 4) |
            (x[4] & 8) | (x[5] & 16) | (x[6] & 32) |
            (x[7] & 64) | (x[8] & 128);
```

```
    y[2] = (x[2] & 1) | (x[3] & 2) | (x[4] & 4) |
           (x[5] & 8) | (x[6] & 16) | (x[7] & 32) |
           (x[8] & 64) | (x[9] & 128);
    y[3] = (x[3] & 1) | (x[4] & 2) | (x[5] & 4) |
           (x[6] & 8) | (x[7] & 16) | (x[8] & 32) |
           (x[9] & 64) | (x[10] & 128);
    y[4] = (x[4] & 1) | (x[5] & 2) | (x[6] & 4) |
           (x[7] & 8) | (x[8] & 16) | (x[9] & 32) |
           (x[10] & 64) | (x[11] & 128);
    y[5] = (x[5] & 1) | (x[6] & 2) | (x[7] & 4) |
           (x[8] & 8) | (x[9] & 16) | (x[10] & 32) |
           (x[11] & 64) | (x[12] & 128);
    y[6] = (x[6] & 1) | (x[7] & 2) | (x[8] & 4) |
           (x[9] & 8) | (x[10] & 16) | (x[11] & 32) |
           (x[12] & 64) | (x[13] & 128);
    y[7] = (x[7] & 1) | (x[8] & 2) | (x[9] & 4) |
           (x[10] & 8) | (x[11] & 16) | (x[12] & 32) |
           (x[13] & 64) | (x[14] & 128);
    y[8] = (x[8] & 1) | (x[9] & 2) | (x[10] & 4) |
           (x[11] & 8) | (x[12] & 16) | (x[13] & 32) |
           (x[14] & 64) | (x[15] & 128);
    y[9] = (x[9] & 1) | (x[10] & 2) | (x[11] & 4) |
           (x[12] & 8) | (x[13] & 16) | (x[14] & 32) |
           (x[15] & 64) | (x[0] & 128);
    y[10] = (x[10] & 1) | (x[11] & 2) | (x[12] & 4) |
           (x[13] & 8) | (x[14] & 16) | (x[15] & 32) |
           (x[0] & 64) | (x[1] & 128);
    y[11] = (x[11] & 1) | (x[12] & 2) | (x[13] & 4) |
           (x[14] & 8) | (x[15] & 16) | (x[0] & 32) |
           (x[1] & 64) | (x[2] & 128);
    y[12] = (x[12] & 1) | (x[13] & 2) | (x[14] & 4) |
           (x[15] & 8) | (x[0] & 16) | (x[1] & 32) |
           (x[2] & 64) | (x[3] & 128);
    y[13] = (x[13] & 1) | (x[14] & 2) | (x[15] & 4) |
           (x[0] & 8) | (x[1] & 16) | (x[2] & 32) |
           (x[3] & 64) | (x[4] & 128);
    y[14] = (x[14] & 1) | (x[15] & 2) | (x[0] & 4) |
           (x[1] & 8) | (x[2] & 16) | (x[3] & 32) |
           (x[4] & 64) | (x[5] & 128);
    y[15] = (x[15] & 1) | (x[0] & 2) | (x[1] & 4) |
           (x[2] & 8) | (x[3] & 16) | (x[4] & 32) |
           (x[5] & 64) | (x[6] & 128);
    }

static void ipermute(byte *x, byte *y) /* x!=y */
/* This is the inverse of the procedure permute. */
    {
    y[0] = (x[0] & 1) | (x[15] & 2) | (x[14] & 4) |
           (x[13] & 8) | (x[12] & 16) | (x[11] & 32) |
           (x[10] & 64) | (x[9] & 128);
    y[1] = (x[1] & 1) | (x[0] & 2) | (x[15] & 4) |
           (x[14] & 8) | (x[13] & 16) | (x[12] & 32) |
           (x[11] & 64) | (x[10] & 128);
    y[2] = (x[2] & 1) | (x[1] & 2) | (x[0] & 4) |
           (x[15] & 8) | (x[14] & 16) | (x[13] & 32) |
           (x[12] & 64) | (x[11] & 128);
    y[3] = (x[3] & 1) | (x[2] & 2) | (x[1] & 4) |
           (x[0] & 8) | (x[15] & 16) | (x[14] & 32) |
           (x[13] & 64) | (x[12] & 128);
    y[4] = (x[4] & 1) | (x[3] & 2) | (x[2] & 4) |
           (x[1] & 8) | (x[0] & 16) | (x[15] & 32) |
           (x[14] & 64) | (x[13] & 128);
    y[5] = (x[5] & 1) | (x[4] & 2) | (x[3] & 4) |
           (x[2] & 8) | (x[1] & 16) | (x[0] & 32) |
           (x[15] & 64) | (x[14] & 128);
    y[6] = (x[6] & 1) | (x[5] & 2) | (x[4] & 4) |
           (x[3] & 8) | (x[2] & 16) | (x[1] & 32) |
           (x[0] & 64) | (x[15] & 128);
    y[7] = (x[7] & 1) | (x[6] & 2) | (x[5] & 4) |
           (x[4] & 8) | (x[3] & 16) | (x[2] & 32) |
           (x[1] & 64) | (x[0] & 128);
    y[8] = (x[8] & 1) | (x[7] & 2) | (x[6] & 4) |
           (x[5] & 8) | (x[4] & 16) | (x[3] & 32) |
           (x[2] & 64) | (x[1] & 128);
    y[9] = (x[9] & 1) | (x[8] & 2) | (x[7] & 4) |
```

```
            (x[6] & 8) | (x[5] & 16) | (x[4] & 32) |
            (x[3] & 64) | (x[2] & 128);
    y[10] = (x[10] & 1) | (x[9] & 2) | (x[8] & 4) |
            (x[7] & 8) | (x[6] & 16) | (x[5] & 32) |
            (x[4] & 64) | (x[3] & 128);
    y[11] = (x[11] & 1) | (x[10] & 2) | (x[9] & 4) |
            (x[8] & 8) | (x[7] & 16) | (x[6] & 32) |
            (x[5] & 64) | (x[4] & 128);
    y[12] = (x[12] & 1) | (x[11] & 2) | (x[10] & 4) |
            (x[9] & 8) | (x[8] & 16) | (x[7] & 32) |
            (x[6] & 64) | (x[5] & 128);
    y[13] = (x[13] & 1) | (x[12] & 2) | (x[11] & 4) |
            (x[10] & 8) | (x[9] & 16) | (x[8] & 32) |
            (x[7] & 64) | (x[6] & 128);
    y[14] = (x[14] & 1) | (x[13] & 2) | (x[12] & 4) |
            (x[11] & 8) | (x[10] & 16) | (x[9] & 32) |
            (x[8] & 64) | (x[7] & 128);
    y[15] = (x[15] & 1) | (x[14] & 2) | (x[13] & 4) |
            (x[12] & 8) | (x[11] & 16) | (x[10] & 32) |
            (x[9] & 64) | (x[8] & 128);
    }

static void CALLTYPE permute_lite2(byte *a, byte *b)
  {
/* This procedure is designed to make each bit of the output dependent on as
  many bytes of the input as possible, especially after repeated application.
*/
    b[0] = (a[0] & 1) + (a[1] & 2) + (a[2] & 4) + (a[3] & 8) + (a[4] & 0x10) +
        (a[5] & 0x20) + (a[6] & 0x40) + (a[7] & 0x80);
    b[1] = (a[1] & 1) + (a[2] & 2) + (a[3] & 4) + (a[4] & 8) + (a[5] & 0x10) +
        (a[6] & 0x20) + (a[7] & 0x40) + (a[0] & 0x80);
    b[2] = (a[2] & 1) + (a[3] & 2) + (a[4] & 4) + (a[5] & 8) + (a[6] & 0x10) +
        (a[7] & 0x20) + (a[0] & 0x40) + (a[1] & 0x80);
    b[3] = (a[3] & 1) + (a[4] & 2) + (a[5] & 4) + (a[6] & 8) + (a[7] & 0x10) +
        (a[0] & 0x20) + (a[1] & 0x40) + (a[2] & 0x80);
    b[4] = (a[4] & 1) + (a[5] & 2) + (a[6] & 4) + (a[7] & 8) + (a[0] & 0x10) +
        (a[1] & 0x20) + (a[2] & 0x40) + (a[3] & 0x80);
    b[5] = (a[5] & 1) + (a[6] & 2) + (a[7] & 4) + (a[0] & 8) + (a[1] & 0x10) +
        (a[2] & 0x20) + (a[3] & 0x40) + (a[4] & 0x80);
    b[6] = (a[6] & 1) + (a[7] & 2) + (a[0] & 4) + (a[1] & 8) + (a[2] & 0x10) +
        (a[3] & 0x20) + (a[4] & 0x40) + (a[5] & 0x80);
    b[7] = (a[7] & 1) + (a[0] & 2) + (a[1] & 4) + (a[2] & 8) + (a[3] & 0x10) +
        (a[4] & 0x20) + (a[5] & 0x40) + (a[6] & 0x80);
  }

static void CALLTYPE ipermute_lite2(byte *b, byte *a)
  {
/* This is the inverse of the procedure permute. */
    a[0] = (b[0] & 1) + (b[7] & 2) + (b[6] & 4) + (b[5] & 8) + (b[4] & 0x10) +
        (b[3] & 0x20) + (b[2] & 0x40) + (b[1] & 0x80);
    a[1] = (b[1] & 1) + (b[0] & 2) + (b[7] & 4) + (b[6] & 8) + (b[5] & 0x10) +
        (b[4] & 0x20) + (b[3] & 0x40) + (b[2] & 0x80);
    a[2] = (b[2] & 1) + (b[1] & 2) + (b[0] & 4) + (b[7] & 8) + (b[6] & 0x10) +
        (b[5] & 0x20) + (b[4] & 0x40) + (b[3] & 0x80);
    a[3] = (b[3] & 1) + (b[2] & 2) + (b[1] & 4) + (b[0] & 8) + (b[7] & 0x10) +
        (b[6] & 0x20) + (b[5] & 0x40) + (b[4] & 0x80);
    a[4] = (b[4] & 1) + (b[3] & 2) + (b[2] & 4) + (b[1] & 8) + (b[0] & 0x10) +
        (b[7] & 0x20) + (b[6] & 0x40) + (b[5] & 0x80);
    a[5] = (b[5] & 1) + (b[4] & 2) + (b[3] & 4) + (b[2] & 8) + (b[1] & 0x10) +
        (b[0] & 0x20) + (b[7] & 0x40) + (b[6] & 0x80);
    a[6] = (b[6] & 1) + (b[5] & 2) + (b[4] & 4) + (b[3] & 8) + (b[2] & 0x10) +
        (b[1] & 0x20) + (b[0] & 0x40) + (b[7] & 0x80);
    a[7] = (b[7] & 1) + (b[6] & 2) + (b[5] & 4) + (b[4] & 8) + (b[3] & 0x10) +
        (b[2] & 0x20) + (b[1] & 0x40) + (b[0] & 0x80);
  }

static void substitute(uint round, byte *x, byte *y)
    {
    uint i;

    for (i = 0; i < blocksize; i++)
        y[i] = *(s + (roundsize*round) + (256*i) + x[i]);
    }

static void isubst(uint round, byte *x, byte *y)
```

```
    {
    uint i;

    for (i = 0; i < blocksize; i++)
        y[i] = *(si + (roundsize*round) + (256*i) + x[i]);
    }

void diamond2_encrypt_block(byte *x, byte *y)
/* Encrypt a block of 16 bytes. */
    {
    uint round;
    byte z[16];

    substitute(0, x, y);
    for (round=1; round < numrounds; round++)
        {
        permute(y, z);
        substitute(round, z, y);
        }
    }

void diamond2_decrypt_block(byte *x, byte *y)
/* Decrypt a block of 16 bytes. */
    {
    int round;
    byte z[16];

    isubst(numrounds-1, x, y);
    for (round=numrounds-2; round >= 0; round--)
        {
        ipermute(y, z);
        isubst(round, z, y);
        }
    }

void lite2_encrypt_block(byte *x, byte *y)
/* Encrypt a block of 16 bytes. */
    {
    uint round;
    byte z[16];

    substitute(0, x, y);
    for (round=1; round < numrounds; round++)
        {
        permute_lite2(y, z);
        substitute(round, z, y);
        }
    }

void lite2_decrypt_block(byte *x, byte *y)
/* Decrypt a block of 8 bytes. */
    {
    int round;
    byte z[8];

    isubst(numrounds-1, x, y);
    for (round=numrounds-2; round >= 0; round--)
        {
        ipermute_lite2(y, z);
        isubst(round, z, y);
        }
    }

void diamond2_done(void)
    {
    if (s)
        {
        memset(s, 0, numrounds * roundsize);
        free((char *)s);
        s=NULL;
        }
    if (si)
        {
        memset(si, 0, numrounds * roundsize);
        free((char *)si);
```

```
        si=NULL;
        }
    }
```

VI. CRYPTANALYSIS OF DIAMOND2

The Diamond Encryption Algorithm (the immediate predecessor of this algorithm) suffered from a form of weak keys. It was possible for a key to result in all of the individual 256-byte substitution arrays to have the same contents if the position of the input key pointer and the key scheduling CRC value were the same at the start of the second substitution array as they were at the beginning of the first one. While the probability of this was low (one in about 2^{40}), it is possible that the regular substitution structure that resulted might have allowed a simplified analytical attack. To correct this, Diamond2 uses the contents of the substitution array just completed in filling the current substitution array (except for the first one, of course). This greatly increases the amount of state information in the tiny pseudorandom number generator used in key scheduling, such that this minor weakness is eliminated.

While I considered the above mentioned flaw worth correction, it certainly doesn't point to any practical attacks on the Diamond Encryption Algorithm — especially given the low probability that the keys in actual use are the weak ones. My sincere thanks go to Colin Plumb for pointing this out so that I could fix it.

In the design of Diamond2, several types of cryptanalytic attacks were considered. The reasons why I currently consider each of them to be computationally infeasible are listed below. If anyone finds an attack on Diamond2 that is better than a brute force attack on the key, please let me know. The following consists of rough order of magnitude estimations and hand waving, but they are of value anyway. To construct more exact proofs, actual construction of all of the cryptanalytic attacks that your opponent might try is required. It is conjectured that actual construction of the attacks mentioned would be much more complex than the following estimates indicate.

A. Brute Force

Exhaustive key search can be made intractable (beyond the reach of any likely enemy) by choosing a key length of around 120 bits. A loose lower bound of the cost of exhaustive key search can be placed with the following generous assumptions. Assuming a massively parallel machine can perform a trillion decryptions per second (with different keys) on each of a billion processors, then an exhaustive key search would take an average of about 42 million years. The user may wish to use smaller key sizes in some applications to save in key management costs, while still maintaining adequate protection for the value of the specific data. Larger keys than 128 bits probably do not contribute significantly to the overall security of a system of data protection, because of some other attacks on data security that are possible. If you do want to use a much larger key, increasing the number of rounds to greater than 10 is recommended.

Another form of brute force attack that is available with block ciphers is the precomputed dictionary attack. The idea here is to create a database of one very probable plain text block encrypted under all possible keys. Sort the resulting cipher text/key pairs by cipher text value and store it in a table. Then to attack a piece of cipher text, look up the possible key in the table and try it on the rest of the message. This may take several iterations, but would be likely to succeed if you could store so much data. The problems here are, of course (1) time to generate the table, and (2) sorting and storing the table.

Note that the Diamond2 Block Cipher allows the use of key sizes that are too short for real security, because brute force attacks are practical for 4 or 5 byte keys (for almost any kind of encryption algorithm).

B. Partial Dictionary Attack

This attack is the one that makes (1) larger block sizes better, and (2) chaining modes instead of Electronic Codebook (ECB) mode better.

Given a set of known plain text and ciphertext using the same key, it is possible to create a partial dictionary of block values. When applied to a sufficiently large body of text, the probability of block reuse is high enough that some information from further encrypted messages with the same key "leak." This is kind of like hiding your secret text under a well-worn cloth with many holes in it. A critical part of the plain text may or may not show through. I know of one test that demonstrated 10% to 20% leakage of text using DES in ECB mode. This would be particularly bad in stereotyped communications, like financial transactions and personal letters. Increasing the block size from 8 to 16 bytes helps, but use of a good block chaining mode with random initialization vectors helps even more.

Note that this attack applies to any block cipher, and it explains why I much prefer the 16-byte block size of Diamond2 over the 8-byte block size of the most common block ciphers.

C. Analytical Attack with Chosen Plain Text

An analytical attack involves solving for the contents of at least one of the substitution arrays. If one array could be isolated by selecting carefully chosen inputs and outputs, then its contents could be solved for. Once this was done, this knowledge could be used to solve for additional substitution arrays. The problem with this attack is that because every output byte is a function of every input byte and at least 112 substitution arrays, this decomposition is difficult.

The only way such an attack might be practical is if all the substitution arrays were identical, which was rare (with a probability of about 2^{-40}) for Diamond and nearly impossible for Diamond2.

I conjecture that a loose lower bound on the complexity of this kind of attack is that it would take more operations than $256 \cdot 2^x$, where x is the number of arrays in the dependency chain for each byte. For a 10 round Diamond2 implementation, this would be an approximate total of $256 \cdot 16(2^{112} + 2^{96} + 2^{80} + 2^{64} + 2^{48} + 2^{33} + 2^{18} + 2^{10} + 2^2) \approx 2 \cdot 10^{37}$ operations. This is about as hard as solving for a 124 bit key with a precomputed plain text attack, but even less practical.

D. Differential Cryptanalysis

Attacking Diamond2 with a form of differential cryptanalysis as described in [6] does not work directly. A similar approach using differences in known plain text values would have some value in reduced Diamond2 variants with 3 or fewer rounds, but would be no better than the analytical attack discussed above for 10 or more rounds.

E. Solving for the Key from an Array

If you could solve for one substitution array, and knew (or guessed) the length of the key, a method might be constructed to directly solve for the key from the contents of one substitution array. This reduces the strength lower bound estimate for an analytical attack on a 10 round Diamond2 system to about $256 \cdot 2^{112} \approx 10^{36}$, or about as strong as a 120 bit key.

The design of the Diamond2 Block Cipher is to make isolation of an individual substitution array computationally infeasible, but if someone figured out how to do that anyway, this would be one of the risks. Another concern would be some kind of divide-and-conquer attack on all of the substitution arrays, one at a time. Diamond2 is stronger than Diamond in this respect because of the dependence of each substitution array on the previous substitution array as well as the key.

F. Bypassing the Algorithm

In any security system, care must be taken to avoid the possibility that the hardware and/or software doing the encryption and decryption is not tampered with or replaced. For very high security applications, a hardware device that is implemented on a tamper resistant chip in a tamper resistant enclosure is preferable to a pure software implementation. Care must also be taken to ensure that the sensitive data is physically secure when in plain text form. Key management and security protocols, although they is beyond the scope of this paper, are also critical concerns that can easily make or break any security system.

G. The Blessing and Curse of Novelty

Two things happen as an encryption algorithm ages. The probability that weaknesses in the algorithm will be published increases, and in the absence of such weaknesses, confidence increases. The probability that well-used algorithms are being subverted, *without the knowledge of the algorithm users*, by clever cryptanalysis and/or dedicated software and hardware cracking technology also increases. For example, I would be quite surprised if at least 5 organizations didn't own at least one dedicated DES cracking machine each — including at least one that I don't want reading my mail.

In other words, for optimal security, avoid using encryption algorithms that are both old and widely used. Also avoid using new encryption algorithms that have not been adequately reviewed or that you haven't searched for published weaknesses for.

VII. DIAMOND2 LITE

Where software speed and table space are critical, a variant of Diamond2 that has a block size of 8 bytes (64 bits) and a minimum of four rounds (8 recommended) is a reasonable compromise. This variant has the advantage that every bit of the output is a function of every bit of the input and every bit of the key after only two rounds. At least four rounds are needed, however, to ensure that the algorithm is strong enough to justify keys of about 64 bits. This only requires 8192 bytes of table space and offers faster speed in software than the full Diamond2 with a 16 bit block size.

It is conjectured that Diamond2 Lite with 8 rounds and a key length of 128 bits is at least equivalent in security to the IDEA™ and Blowfish ciphers, and more secure than the aging DES algorithm.

VIII. LEGAL ISSUES

The Diamond2 and Diamond2 Lite Block Ciphers may be used for any legal purpose without payment of royalties to the inventor or his employer, however the names “Diamond2 Block Cipher” and “Diamond2 Lite Block Cipher” are Trade Marks owned by the inventor, and may not be used in connection with any algorithm that does not comply with the reference implementation given herein. The Diamond2 Block Cipher is the same as the Diamond, MPJ and MPJ2 Encryption Algorithms, with the exception of the key expansion algorithm. Some governments may restrict the use, publication, or export of strong encryption technology.

IX. CONCLUSION

Diamond2 and Diamond2 Lite are two of several alternatives to the aging and now relatively insecure DES algorithm. Source code for a software implementation of Diamond2 in C is available in the USA and Canada on the Colorado Catacombs BBS at 303-772-1062 in the file DLOCK2.ZIP or on the Internet as **`ftp://ftp.csn.net/mpj/I_will_not_export/crypto_???????/file/dlock2.zip`**, where the **`???????`** is revealed in **`ftp://ftp.csn.net/mpj/README`**. Comments, questions, and reports of possible weaknesses should be sent to the author at one of the following. I recommend that you ask me if any weaknesses have been found in the Diamond2 Block Cipher before using it in any critical applications.

Michael Paul Johnson
PO BOX 1151
LONGMONT CO 80502-1151
USA

BBS: 303-772-1062
Internet mail: m.p.johnson@ieee.org, mpj@csn.net, or mpj@netcom.com
CompuServe: 71331,2332

X. REFERENCES

[1] Michael J. Wiener, “Efficient DES Key Search,” Bell-Northern Research, PO Box 3511 Station C, Ottawa, Ontario, K1Y 4H7, Canada, 20 August 1993.

[2] Theodor Brüggemann and Hoger Bürk, “Der Verschlüsselungsalgorithmus IDEA™,” Ascom Tech AG, Fachbereich Kryptologie, Ziegelmattstrasse 1, CH - 4503 Solothurn, Switzerland.

[3] Xuejia Lai and James L. Massey, “Markov Ciphers and Differential Cryptanalysis,” in *Advances in Cryptology – EUROCRYPT '91,* Springer-Verlag, pp 17-38.

[4] Xuejia Lai “Detailed Description and a Software Implementation of the IPES Cipher,” Institut für Signal- und Informationsverarbeitung, ETH Zürich.

[5] Michael Paul Johnson, “Beyond DES: Data Compression and the MPJ Encryption Algorithm,” Master’s Thesis at the University of Colorado at Colorado Springs, 1989. Available by anonymous ftp to ftp.csn.net in /mpj or on the author’s BBS at 303-772-1062.

[6] Eli Biham and Adi Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. New York: Springer-Verlag, 1993.

[7] *C Programmers Guide to NetBIOS*, Howard W. Sams & Co., Inc.

[8] Rick Sternbach and Michael Okuda, *Star Trek the Next Generation Technical Manual*, New York: Pocket Books, 1991.

[9] Xuejia Lai and James L. Massey, “A Proposal for a New Block Encryption Standard,” in *Advances in Cryptology – EUROCRYPT '90*, Springer-Verlag, pp 389-404., 1990.