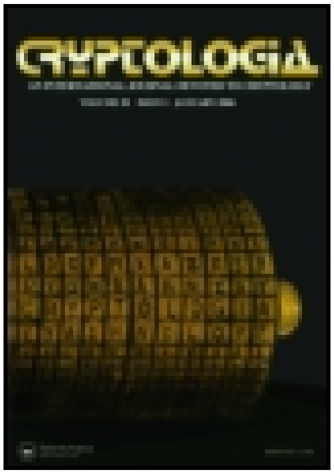

Cryptologia
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/ucry20

WIDE-OPEN ENCRYPTION DESIGN OFFERS FLEXIBLE IMPLEMENTATIONS

ROBERT SCOTT

Published online: 04 Jun 2010.

WIDE-OPEN ENCRYPTION DESIGN OFFERS FLEXIBLE IMPLEMENTATIONS

ROBERT SCOTT

ABSTRACT: Using principles from the Data Encryption Standard (DES). a more software-compatible encryption algorithm is developed that addresses the major problems of DES: key size and secret design.

In 1973 and 1974. the National Bureau of Standards (NBS) called for proposals for a data encryption standard for commercial and non-military governmental use. IBM's proposal. an outgrowth of their "Lucifer" cipher development. was adopted by NBS in 1977. Despite this stamp of approval, DES was. and still is. controversial. The controversy centers around two main points. One is the method used to pick the pseudo-random tables defined by the standard (the so-called "S-boxes"). The other is the key size of 56 bits.

SECRET DESIGN IN DES

Although a lot has been written about DES [1]. [2]. very little public information is available on how DES was designed. IBM's development notes have been classified. The detailed results of the certificational work have also been classified. Nevertheless. certain properties of DES have emerged. Most. if not all. of these properties do appear as an honest attempt to strengthen the algorithm. For example, each S-box is a function from six bits to four bits. If a single bit is changed at the input to any one of the eight S-boxes. at least two output bits will change. Maintaining this one to two "change expansion". as it is called. is not an easy task. In addition. when restricted to the middle four input bits. each S-box function is actually one-to-one onto the output range. Change expansion is a desirable feature. since it hastens diffusion of information. but to achieve it the entries in the S-box tables need to be picked. at least partially. according to strict patterns. Some of these structures have been found [3] in terms of redundant columns when the S-boxes are written out in binary form. In this respect S-box number four is particularly notable. One of the contentions of this paper is that the change propagation afforded by more random tables is more than adequate.

and such design would reduce the suspicion of secret structure that weakens the algorithm.

EXHAUSTIVE SEARCH CONTROVERSY

Perhaps the most serious threat to DES is its keysize. The algorithm uses a 56-bit key. giving about 7.2×10^{16} different keys. However, if a special LSI chip could try a million keys per second. and if a million of these chips were operating in parallel, then all possible keys could be checked in about 20 hours. Professor Martin Hellman of Stanford proposed the feasibility of just such a machine in 1977 [4]. Using technology of that year, he estimated that the search machine would cost 20 million dollars, which. if amortized over five years. gives a cost per solution of less than $5,000. As technology advances, this cost can only come down.

It has been suggested that double encryption using DES would answer the exhaustive search threat. Double encryption would certainly help. but it is still open to a "meet in the middle" attack[5]. So now even triple encryption is suggested. Such measures certainly can address the problem. but they also double or triple the cost of encryption. It would be preferable if a sufficiently secure algorithm could be used right from the start.

SOFTWARE VS. HARDWARE

DES is more than just an algorithm. It is also a set of implementation guidelines meant to ensure overall system security. One such guideline states that any device which is certified as meeting the standard shall implement the encryption algorithm in hardware (or firmware) and not in software. This is an attempt to make it difficult to subvert the encryption system through "bugged" software.

In many applications NBS device certification is not required. and security is handled totally in software. But lacking anything better. designers still use DES. DES can be implemented in software, but not easily or efficiently. The bit permutations called for are difficult to program on a general-purpose computer. particularly if they are to be at all run-time efficient. It takes a sharp programmer with a good knowledge of machine code to do such a job efficiently. There are even initial and final fixed permutations in DES that do not add anything to the strength of the algorithm. since they are completely known. but they add considerably to both the time and program size in a software implementation. The difficulty of microprocessor implementation is illustrated by examining the 6502-based implementation in [6].

A table of execution speeds for several DES chips [7] shows large variations between the slowest (80 bytes/second) and the fastest (1,770,000 bytes/second). The slow chips are fairly general purpose single-chip micro-computers specially programmed to implement DES. In particular, there seems to be a gap of several orders of magnitude between the slowest special LSI designs and the fastest general purpose micros running a software implementation. This gap forces the systems designer into a hardware implementation sooner than would be necessary if the algorithm were more of a compromise between hardware and software.

With the algorithm that I am about to propose (NEWDES), a purely software implementation on a 4 MHz Z-80 microprocessor can do a block encryption in 1,700 microseconds, and the program takes 221 bytes of code and 256 bytes of table space for a total of 477 bytes. On a 10MHz 68000 microprocessor, the encryption can be done in 770 microseconds[8], again with less than half a K of memory. And the algorithm is so easy to describe that even an average programmer can adapt it to his system in a small fraction of the time it would take to correctly implement DES.

STRUCTURE PATTERNED AFTER DES

DES is a block cipher operating on eight bytes at a time. The algorithm I am proposing is also a block cipher with much of the same general structure. One reason to keep this structure is that in the wake of DES, a set of auxiliary standards [9] was developed to specify modes of operation. The standards describe how to use a 64-bit block cipher to prevent falsifying messages, to detect fraudulent insertion of messages, to detect fraudulent modification of messages, and to detect the replay of previously valid messages. Detection is achieved by such mechanisms as cipher feedback (CFB mode), cipher block chaining (CBC mode), and message identifier and authentication codes. These modes of operation are well accepted, common sense methods. They are, however, entirely external to the DES algorithm, and are just as applicable to any 8-byte block cipher.

Since the main feature of NEWDES is its wide-open design, I will try to describe the evolution of its development, citing what I believe to be natural decisions at each stage. Algorithms and formulas will be described in the "C" programming language.

The DES algorithm is composed of 16 successive rounds. In each round, a very complicated function of half of the 8-byte block and some key bits is "added on" (using exclusive-or) to the other half of the block. Decryption is possible in reverse order because in each round, the half-block that generated the function is preserved. This part of the structure I decided to keep because it looked natural to me, however I wanted to avoid any bit permutations because of the awkwardness of software implementations. A most natural byte-oriented first round would be to have the first four bytes generate changes to the last four bytes in order. Thus, if we start with a block composed of bytes B0...B7, we would have

```
B4 = B4 ^ f[B0];    /*  "^" means exclusive-or  */
B5 = B5 ^ f[B1];
B6 = B6 ^ f[B2];
B7 = B7 ^ f[B3];
```

leaving the matter of inserting the key until later. Now if we take the most natural second round, bytes B4...B7 would be used to change B0...B3. But then there would never be any interaction between bytes B0 and B1, so the second round must be different. We can modify this operation with a shift.

```
B1 = B1 ^ f[B4];
B2 = B2 ^ f[B5];
B3 = B3 ^ f[B6];
B0 = B0 ^ f[B7];
```

Now at least we have good diffusion. After seven rounds, every byte has affected every other byte.

Now let us add in the key. An easy way to add the key in is to exclusive-or it with the arguments to the f function, similar to what is done in DES. If the basic key is long enough, the complexity of the expansion is not as important, so let us take a basic key and expand it by replication. Each key byte should be used enough times and there should be enough rounds so that there is complete diffusion of each bit of the key long before the last round. The most straightforward inclusion of an expanded key would be

```
i = 0;

B4 = B4 ^ f[B0 ^ Key[i++]];
B5 = B5 ^ f[B1 ^ Key[i++]];
B6 = B6 ^ f[B2 ^ Key[i++]];
B7 = B7 ^ f[B3 ^ Key[i++]];    /* end of round 1 */

B1 = B1 ^ f[B4 ^ Key[i++]];
B2 = B2 ^ f[B5 ^ Key[i++]];
B3 = B3 ^ f[B6 ^ Key[i++]];
B0 = B0 ^ f[B7 ^ Key[i++]];    /* end of round 2 */

/* etc. */
```

where the "i++" in C means increment i after using it. But if we follow this very symmetric pattern, a curious failure of diffusion occurs. What if the key bytes are all the same and the plaintext input bytes are all the same? Then each two rounds would give eight identical bytes. Identical plaintext bytes are very likely, and if the key is left up to the user, identical key bytes are also quite possible. I do not want such a case to produce such highly structured results, so some asymmetrical modification is necessary. It occurred to me that if the key were not used in one formula, then the symmetric structure could be broken. Since every other formula has an exclusive-or before the table look-up, I chose to replace the use of the key byte with the use of one of the eight data bytes as follows:

```
B4 = B4 ^ f[B0 ^ Key[i++]];
B5 = B5 ^ f[B1 ^ Key[i++]];
B6 = B6 ^ f[B2 ^ Key[i++]];
B7 = B7 ^ f[B3 ^ Key[i++]];      /* end of round 1 */

B1 = B1 ^ f[B4 ^ Key[i++]];
B2 = B2 ^ f[B4 ^ B5];
B3 = B3 ^ f[B6 ^ Key[i++]];
B0 = B0 ^ f[B7 ^ Key[i++]];      /* end of round 2 */

/* etc. */
```

Thus we only need three key bytes in the second round.

In picking the number of rounds, I wanted to keep it near the same number as with DES so that a hardware implementation would incur the same sequential delay, and I wanted the number of key bytes to be about double that of DES. It turned out that those requirements could be met very neatly with 17 rounds and 15 key bytes. Then each key byte is used exactly four times. Also, I wanted an odd number of rounds so that the decryption algorithm would be the same as the encryption algorithm, except for a different key schedule. Now we can write the entire encryption/decryption algorithm as illustrated.

The checks for Key index wrap-around are not included in the last four formulas because wrap-around never occurs there. For those readers with a more graphical orientation, the round-by-round data flow is illustrated in Figure 1. It should also be pointed out that execution time can be improved by pre-expanding the basic key into a 60-byte array. Then the index checks are vastly simplified.

```
        char    B0,B1,B2,B3,B4,B5,B6,B7;
        char    Key[15];
        int     initi,delta,fini;

/* for encryption, set initi=0, delta=1, fini=0
   for decryption, set initi=11, delta=9, fini=12 */

code()
  {
        int     i;
        i = initi;
        while(1)
          {
                B4 = B4 ^ f[B0 ^ Key[i++]];
                if( i == 15 )
                        i = 0;
                B5 = B5 ^ f[B1 ^ Key[i++]];
                if( i == 15 )
                        i = 0;
                B6 = B6 ^ f[B2 ^ Key[i++]];
                if( i == 15 )
                        i = 0;
                B7 = B7 ^ f[B3 ^ Key[i]];

                i = i + delta;
                if( i > 14 )
                        i = i - 15;
                if( i == fini )
                        return;

                B1 = B1 ^ f[B4 ^ Key[i++]];
                B2 = B2 ^ f[B4 ^ B5];
                B3 = B3 ^ f[B6 ^ Key[i++]];
                B0 = B0 ^ f[B7 ^ Key[i]];

                i = i + delta;
                if( i > 14 )
                        i = i - 15;
          }
  }
```

Encryption/Decryption Algorithm

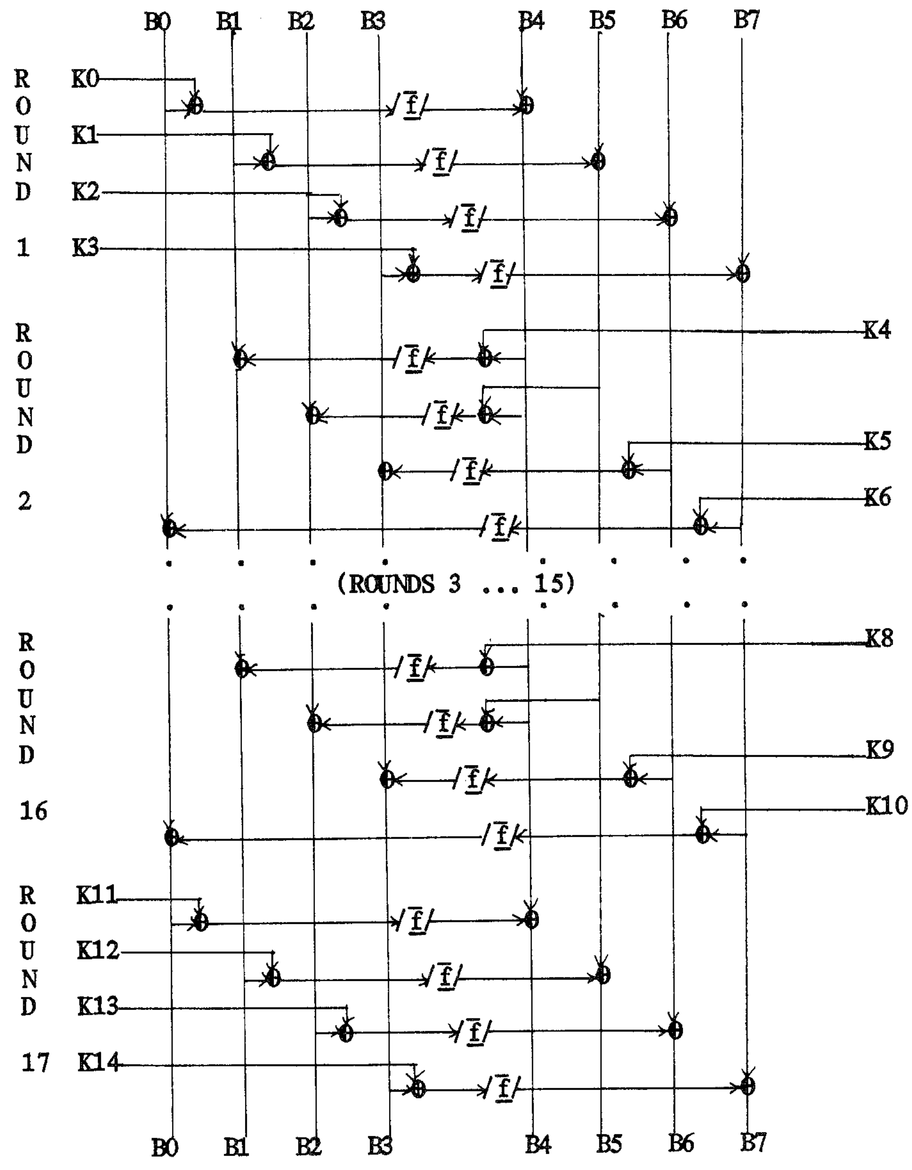

Figure 1. The NEWDES encryption algorithm is shown in terms of its data flow. The rounds not shown are just like rounds 1 and 2, except that the key bytes are K0...K14 repeated four times. The symbol, ⊕, means exclusive-or.

PICKING THE f FUNCTION

In keeping with the concept of a wide-open design, we need to be very careful about how f is derived. f could be a totally random function, but then it would not be one-to-one and onto. I was concerned that the non-uniform coverage of the output range would result in statistical biases that could weaken the algorithm. I decided f should be a random-looking one-to-one onto function (a permutation). Ideally, f could be generated from an apparently random physical process, such as radioactive decay. But the practical problems of certifying the honesty of a piece of hardware to the public precluded that option. I decided that f would have to be generated by a pseudo-random process that could be duplicated by anyone. Of course, there are many equally valid pseudo-random processes. In choosing one over the others, I can only say that the number of "natural-looking" choices is small enough that the number of degrees of freedom in choosing f is too small to permit the selection of f in order to incorporate "significant" structure. Suppose that I were able to think up and check out one choice per minute, and that I had been doing so continuously for the last 30 years. Then I would have checked out less than 16 million choices, which corresponds to 24 bits of information. For example, that would give me the ability to fix three of the 256 bytes in the f table. Three "fixed" bytes hardly seems like enough structure in which to hide a trap-door.

The pseudo-random process I chose was to generate f from a well-known document - the Declaration of Independence. The algorithm was chosen so that f would be a permutation. An easy way to ensure f is a permutation is to start with the identity function f[i] = i, and interchange table entries f[i] and f[j] for a variety of (i,j). I decided to make an interchange for each of the 6534 letters of the document. One of the elements to be interchanged would be f[i] for i equal to 0,1,... with wrap-around after 255. The other element to be interchanged (f[j]) would be determined by the next letter from the Declaration. Since the distribution of letters in English is not uniform, the element to participate in the interchange should not depend solely on one letter from the Declaration. Although the distribution of letters themselves may be non-uniform, the cumulative sum of the letters (in ASCII) in the long run is uniform. So if we start out with j equal to zero, and if we add the ASCII value of the next letter from the Declaration (modulo 256) then j will take on pseudo-random values evenly distributed in the range 0...255. Yet the individual letters and the order in which they appear in the Declaration have a cascading non-linear effect on the f table. Since there are 6534 letters in the Declaration, the interchanging will occur at least 25 times for each table entry, and probably more like 50 times. Thus the final position of each table entry is determined by at least 25 to 50 different interchanges. That appeared to me to be enough interchanges so that the statistical biases in English would play no observable role in the final condition of the f table. The resulting f table is shown in Figure 2. The generation algorithm can be written as:

```
for( i=0; i<256; i++)
        f[i] = i;

i=0;  j=0;

while( (c=getchar()) != EOF )
  {
        if( c > 'Z' )
                c = c - 32;
                  /* force upper case */
        if( c >= 'A'  && c  <=  'Z' )
          {
                i = (i+1) & 255;
                j = (j+c) & 255;
                k = f[i];
                f[i] = f[j];
                f[j] = k;
          }
  }
```

NEWDES can be implemented by just copying the table in Figure 2. But if anyone wants to verify that the f table was generated as described, it is important to note that some reference books may have slightly altered spelling and punctuation from Jefferson's original document. For example, some editors replace Jefferson's phrase "Cruelty & Perfidy" by "Cruely and Perfidy". I did all my work from a photocopy of the original handwritten document, using only the body of the text, ignoring the title and signatures. In order to resolve all ambiguities of case and punctuation, all puncuation and spacing (and special characters, such as the "&" were thrown out, leaving only the letters (A...Z). All letters were converted to upper case ASCII ('A' = 65 and 'Z' = 90) before they were used. The f table shown in Figure 2 has been verified by transcribing the Declaration of Independence seven times into three different computers (IBM 370, DEC 10, and a Z-80 CP/M system) by two different typists, and the generation algorithm was implemented three times in two different languages (PASCAL and C). Typographical errors were found and eventually all seven versions were reconciled and produced the same f table. Nevertheless, I invite independent verification by anyone with the patience to get at least two separate transcriptions to agree with each other.

f[0...255] =

32	137	239	188	102	125	221	72	212	68	81	37	86	237	147	149
70	229	17	124	115	207	33	20	122	143	25	215	51	183	138	142
146	211	110	173	1	228	189	14	103	78	162	36	253	167	116	255
158	45	185	50	98	168	250	235	54	141	195	247	240	63	148	2
224	169	214	180	62	22	117	108	19	172	161	159	160	47	43	171
194	175	178	56	196	112	23	220	89	21	164	130	157	8	85	251
216	44	94	179	226	38	90	119	40	202	34	206	35	69	231	246
29	109	74	71	176	6	60	145	65	13	77	151	12	127	95	199
57	101	5	232	150	210	129	24	181	10	121	187	48	193	139	252
219	64	88	233	96	128	80	53	191	144	218	11	106	132	155	104
91	136	31	42	243	66	126	135	30	26	87	186	182	154	242	123
82	166	208	39	152	190	113	205	114	105	225	84	73	163	99	111
204	61	200	217	170	15	198	28	192	254	134	234	222	7	236	248
201	41	177	156	92	131	67	249	245	184	203	9	241	0	27	46
133	174	75	18	93	209	100	120	76	213	16	83	4	107	140	52
58	55	3	244	97	197	238	227	118	49	79	230	223	165	153	59

Figure 2. The f-function generated from the Declaration of Independence.

DIFFUSION

NEWDES has complete diffusion of plaintext after seven rounds. Complete diffusion means every bit of the initial block has the potential for changing any bit of the block after seven rounds, as illustrated in Figure 3. This is admittedly not as fast as DES, which has bit permutations to spread effects faster; but since there are still 9 rounds left to go, the diffusion is probably sufficent at the end to render any statistical cryptanalysis useless. Key diffusion for any one key byte is complete seven rounds after the key is used. Since every byte of the key is used by the fifth round, complete key diffusion occurs after round 12. But in some sense, it is better diffusion than that of plaintext because while the first occurence of a key byte is busy diffusing through the block, three additional occurences are added in to make the final block dependent on the key in a more complicated way than it is dependent on the plaintext. Diffusion is important because if any metric can be defined under which small input changes result in small output changes, an adversary could mount a "key-clustering" attack and vastly reduce the search space for the key. The following examples show how many bits change for a single bit change in either the key or the plaintext. In these examples, the base key and plaintext are all zeroes, and each example shows either one bit of plaintext or one bit of key set to one, and the results are compared with the all zeroes case. For ease of bit comparison, the ciphertext is shown in hexadecimal, and the number of bits changed is shown on a nibble-by-nibble basis.

		0	0	0	0	0	0	0	0
ROUND	1	0	0	0	0	142	78	247	25
ROUND	2	65	84	204	7	142	78	247	25
ROUND	3	65	84	204	7	189	175	188	174
ROUND	4	202	4	221	221	189	175	188	174
ROUND	5	202	4	221	221	111	120	221	202
ROUND	6	240	2	201	145	111	120	221	202
ROUND	7	240	2	201	145	47	131	211	165
ROUND	8	24	170	127	231	47	131	211	165
ROUND	9	24	170	127	231	25	82	11	73
ROUND	10	20	196	224	2	25	82	11	73
ROUND	11	20	196	224	2	109	16	168	77
ROUND	12	78	112	159	210	109	16	168	77
ROUND	13	78	112	159	210	69	210	97	74
ROUND	14	140	83	170	118	69	210	97	74
ROUND	15	140	83	170	118	44	24	139	110
ROUND	16	60	155	200	52	44	24	139	110
ROUND	17	60	155	200	52	56	187	199	249

		0	0	0	0	0	0	1	0
ROUND	1	0	0	0	0	142	78	246	25
ROUND	2	65	84	204	222	142	78	246	25
ROUND	3	65	84	204	222	189	175	189	179
ROUND	4	168	4	221	213	189	175	189	179
ROUND	5	168	4	221	213	197	120	220	63
ROUND	6	213	42	126	0	197	120	220	63
ROUND	7	213	42	126	0	93	233	7	75
ROUND	8	144	70	230	253	93	233	7	75
ROUND	9	144	70	230	253	15	173	54	23
ROUND	10	254	36	249	0	15	173	54	23
ROUND	11	254	36	249	0	165	187	197	155
ROUND	12	172	159	115	46	165	187	197	155
ROUND	13	172	159	115	46	220	212	53	182
ROUND	14	26	90	167	189	220	212	53	182
ROUND	15	26	90	167	189	35	12	243	51
ROUND	16	84	93	88	189	35	12	243	51
ROUND	17	84	93	88	189	228	155	178	48

Figure 3. Round by round diffusion is illustrated using the key of 31,41,59,26,53,58,97,93,238,46,26,43,38,32,79 and input data of 0,0,0,0,0,0,0,0 and 0,0,0,0,0,0,1,0.

```
K= 00    P=00          C=A2 17 60 54 F5 8B 34 58  (base case)
Key[0] = 1             C=6C BA AB D0 05 94 AD 05
# of bits changed:       23 23 23 11 40 14 22 23 (total=35)

K=  00    P=00          C=A2 17 60 54 F5 8B 34 58   (base case)
B0 = 1                  C=8A 93 0D D6 86 61 FA D5
# of bits changed:        11 11 23 11 32 32 23 13 (total=30)
```

When all 120 single-bit key changes and all 64 single-bit plaintext changes are examined as above, the total number of ciphertext bit changes ranges from 24 to 42, which is about what one would expect from a good pseudo-random function. This shows that even though the algorithm is byte-oriented, because of the non-linear mixing of the f function, every bit of key and plaintext can affect every bit of the output.

Another way to measure diffusion is to count how many times a given input byte appears in the expressions for the output bytes of each round. For example, after three rounds, the modified B4 can be expressed in terms of the original values of B0...B7 as:

```
B4^f[B0^Key[0]]^f[B0^f[B7^f[B3^Key[3]]^Key[6]]^Key[7]]
```

in which B0 appears two times. The table of Figure 4 shows how many times B0 appears in each round. Notice that at the end of 17 rounds, B0 figures into each output byte at least 834 ways. And since any two occurrences of B0 are separated by at least one application of the randomizing function, f, there will not be any generalized "folding" or simplification of these expressions.

NON-LINEARITY OF f

Since all the operations of NEWDES are linear except for f, it is important that f be subjected to the same scrutiny as the S-boxes of DES. A quick check will show that f is not linear or affine. But that is not enough. It should not even be very close to affine. Because of the size of f, it is very difficult to search through various modifications to f looking for linearity. Here a probabilistic argument may be useful. An affine transformation is defined by an eight by eight matrix of ones and zeros and an eight bit vector. Hence there are 2^{72} affine transformations. On the other hand, there are 256! different permutations. If we assume that the method of generation used for f is pseudo-random enough to generate uniformly distributed permutations with respect to the set of affine transformations, then the probability of a randomly generated permutation being affine is $2^{72}/256!$. (It is actually even lower than this since not all affine transformations are permutations.) Using

Stirling's formula we approximate this probability as 2^{-1611}. Suppose we call a permutation "nearly affine" if it lies in a neighborhood of an affine transformation under some metric, and that the size of each neighborhood is 2^{1500}. Then we would still have the acceptably low probability of 2^{-111} that a randomly generated permutation is nearly affine. But just to be complete, I did do a search for solutions to the equation:

f[x^y] ^ f[y] = f[x^z] ^ f[z] = b

for $1 \leq x \leq 255$, $0 \leq y < z \leq 255$ and (x^y^z) not zero, keeping a histogram of the values of b for which solutions were found. If f were affine, the equation would hold for all x, y, and z. On a purely random basis, I would expect 32,385 of the 8,290,560 parameter sets to yield a solution, and then the values of b should be uniformly distributed. In fact, there are 32,304 solutions, and the histogram for values of b ranged from 48 to 204 occurences.

ROUND	B0	B1	B2	B3	B4	B5	B6	B7
1	1	0	0	0	1	0	0	0
2	1	1	1	0	1	0	0	0
3	1	1	1	0	2	1	1	0
4	1	3	4	1	2	1	1	0
5	1	3	4	1	3	4	5	1
6	2	6	11	6	3	4	5	1
7	2	6	11	6	5	10	16	7
8	9	11	26	22	5	10	16	7
9	9	11	26	22	14	21	42	29
10	38	25	61	64	14	21	42	29
11	38	25	61	64	52	46	103	93
12	131	77	159	167	52	46	103	93
13	131	77	159	167	183	123	262	260
14	391	260	465	429	183	123	262	260
15	391	260	465	429	574	383	727	689
16	1080	834	1422	1156	574	383	727	689
17	1080	834	1422	1156	1654	1217	2149	1845

Figure 4. Diffusion of input byte B0 is illustrated by counting the number of occurrences of B0 in the expressions for the values of B0...B7 at each round. Since B0 affects B4 in an odd-numbered round, the entry under B4 after one round is one. After round five, B0 appears three and four times in B4 and B5 respectively. Since, in round six, B2 is replaced by B2 exclusive-or a function of B4 and B5, B2 will acquire seven more occurrences of B0. Hence the appearance count in the B2 column goes up from four to 11.

CYCLE STRUCTURE OF f

Another interesting feature of a permutation is its decomposition into cycles. The f function we have generated is undistinguished in this respect. It has four cycles of length 1, 19, 79, and 157. The one fixed point is actually quite likely among randomly generated permutations, and it does not appear to weaken the algorithm since the output of the f function is not applied directly to the input. I have yet to find any special case of key and plaintext for which the results of encryption is "special" in any sense.

BIT CHANGE EXPANSION

If f is to be used to propagate bit changes, then a single bit change in the input to f ought to produce an apparently uncorrelated change in the output. This property was verified by computing

```
f[x^(00000001)] ^ f[x] ... f[x^(10000000)] ^ f[x]
```

for all x, and keeping a cumulative total of the number of bits changed. It turned out that over all values of x and over all 8 bits of each value of x, on the average, 3.97 bits changed at the output of f for each single bit change at the input, which is again in agreement with what a good pseudorandom function should do.

BACK TO HARDWARE

NEWDES has been designed to make software implementation both easy and fast, but there are applications in which software is not fast enough. In that case, NEWDES can be implemented in hardware and run at nearly the same speeds as DES since there are about the same number of rounds, and each round has the same sequential delay. The only disadvantage for NEWDES in hardware is the cost of the table function. To process the operations for one round in parallel requires four separate copies of f, each of which is a 256 by eight-bit ROM. This can be compared with DES's eight S-boxes, each of which is a 64 by four-bit ROM. NEWDES then would use four times as much table ROM as DES (not counting key scheduling control ROMs, which are probably comparable). Given the current cost of DES chips, it is unlikely that this increase in ROM space would have a significant effect on the cost of the device.

CORRECTNESS TESTS

To check an implementation of NEWDES for correctness the data of Figure 5 can be used for comparison. Since the most likely error would be miscopying an f table entry, the examples of 30-fold encryption have been picked so that each one involves accesses to every element of the f table. If an implementation of NEWDES generates the correct results for these examples, then the f tables can be assumed to be correct. If the results are different, then by the diffusion of the algorithm, they will probably be very different, with no clue as to what went wrong. If this occurs, the round-by-round examples in Figure 3 may be helpful.

KEY = 31,41,59,26,53,58,97,93,238,46,26,43,38,32,79

INPUT BLOCK								ENCRYPTED ONCE ENCRYPTED 30 TIMES							
0	0	0	0	0	0	0	0	60	155	200	52	56	187	199	249
								183	106	97	58	239	9	231	129
1	2	3	4	5	6	7	8	226	4	14	237	143	244	145	46
								147	230	194	164	78	67	16	202
10	20	30	40	50	60	70	80	235	132	9	3	48	102	14	159
								168	49	142	49	176	120	76	188
2	2	2	2	2	2	2	2	242	185	63	238	33	248	82	149
								48	69	0	118	84	220	31	48
101	102	103	104	105	106	107	108	104	111	143	150	57	158	116	152
								149	218	155	243	197	170	192	204
12	23	34	45	56	67	78	89	99	248	54	243	75	200	59	40
								15	242	115	140	9	175	69	106

Figure 5. An implementation of NEWDES can be verified by performing the following encryption operations and checking the results. To completely check the table function f, the encryption algorithm was applied to the input blocks, and the results were reencrypted, for the 30 iterations, so that for each example, every entry in f is used.

CONCLUSION

What I have attempted to do is to fill what I see as a gap in applicable encryption tools. There are small operating systems in use today that provide file encryption via simulation of World War II rotor machines or running key ciphers. Obviously something is needed for the software applications that is both secure and easy to program. But unless the design is in some sense "transparent", few users will trust it. I believe NEWDES is secure, transparent, and easy to implement. It would be well suited for a variety of custom in-house applications as well as a standard to be used in public networks.

REFERENCES

1. National Bureau of Standards. 1977. Data Encryption Standard. Federal Information Processing Standard (FIPS) Publication No. 46.

2. Diffie, D. and M. E. Hellman. 1979. Privacy and Authentication: An Introduction to Cryptography. Proc. IEEE. 67(3).

3. Hellman, M. E. et al. 1980. Results of an Initial Attempt to Cryptanalyze the NBS Data Encryption Standard. In A Special Tutorial Seminar: Cryptography and Data Security. Copyright Martin Hellman.

4. Hellman, M. E. et al. 1977. Exhaustive Cryptanalysis of the NBS Data Encryption Standard. Computer. 10(6): 74-84.

5. Merkle, R. and M. E. Hellman. 1981. On the Security of Multiple Encryption. CACM 24: 465-467.

6. Meushaw, R. 1979. The Standard Data Encryption Algorithm: Part 2. Byte. April: 110-130.

7. Cushman, R. H. 1982. Data-encryption Chips Provide Security - Or Is It False Security? EDN. Feb 17: 39.

8. Grappel, R. 1982. Hemenway Corporation, private communication. Nov.

9. Proposed Federal Standard 1026.