A Better Key Schedule for DES-like Ciphers

Uri Blumenthal
uri@watson.ibm.com
IBM
☎ +1–914–945–1267

Steven M. Bellovin
smb@research.att.com
AT&T Bell Laboratories
☎ +1–908–582–5886

June 6, 1996

Abstract

Several DES-like ciphers aren't utilizing their full potential strength, because of the short key and linear or otherwise easily tractable algorithms they use to generate their key schedules. Using DES as example, we show a way to generate round subkeys to increase the cipher strength substantially by making relations between the round subkeys practically intractable.

1 Introduction

A Feistel cipher consists of two parts:

- the key schedule, which produces subkeys for each round, normally from the "generating" or main key; and
- scrambling—the "real" encryption, which mixes the subkey bits with the input bits to produce the output bits.

While *scrambling* has been studied very thoroughly, *key schedule* generation has not enjoyed the attention it rightfully deserves until recently [Kn93]. It has been shown that poor key schedules can break an otherwise perfectly good cipher [Bi94], but relatively little has been said about what it means for a key schedule to be strong. Lars Knudsen [Kn93] suggests that a **strong key schedule** has the following properties:

1. Given any s bits of the r round keys, derived from an unknown generating key (where s is less than the total amount of round keys material), it is *hard* to find any of the remaining key bits from the s known bits.

2. Given some relation between two generating keys, it is *difficult* to predict the relations between any of the round keys derived from these two generating keys.

Several attacks, notably differential [BS93] and linear cryptanalysis [Mat94a], even though they theoretically recover the subkeys directly and independently from each other, in practice utilize the fact that the relation between the bits of the subkeys of different rounds is rather linear, thus avoiding considerable amount of work.

The idea, therefore, is to generate the subkeys in such a way, that the relation between any subkey bits of any rounds is *practically* intractable. As a side-effect, generating a key schedule this way takes longer than traditional key schedule setup; this, coupled with the fact that we allow for variable key lengths, helps foil brute force attacks as well.

Looked at another way, a DES-like cipher is really two separate cryptosystems, F and G. The user sees

$$G = \{g_k : k \in \{0,1\}^{56}\}.$$

But there is really underlying cryptosystem

$$F = \{f_k : k \in \{0,1\}^{768}\}$$

and a mapping $\phi : \{0,1\}^{56} \rightarrow \{0,1\}^{768}$ such that $f_{\phi(k)} = g_k$ for all input keys k. We assert that the standard mapping ϕ has some undesirable properties, and propose a new mapping ϕ'. As noted, our function is more general, in that it maps $\{0,1\}^*$ to $\{0,1\}^{768}$.

Section 2 describes how we generate the key schedule. We use DES [NBS77] as an example, and hence such magic numbers as 168 and 768, but any similar block cipher such as IDEA or SKIPJACK could be used instead. Section 3 presents the justification for each step. We discuss possible attacks in Section 4, and give a final analysis in Section 5.

2 Design

We first define a primitive called *n-folding*, which takes a variable-length input block and produces a fixed-length output sequence. The intent is to give each input bit approximately equal weight in determining the value of each output bit. Note that whenever we need to treat a string of bytes as a number, the assumed representation is Big Endian — Most Significant Byte first (see the implementation in appendix B on page 11).

To n-fold a number X, replicate the input value to a length that is the least common multiple of n and the length of X. Before each repetition, the input is rotated to the right by 13 bit positions. The successive n-bit chunks are added together using 1's-complement addition (that is, addition with end-around carry) to yield a n-bit result denoted $\langle X \rangle_n$.

We take the 64-fold of the user's input K to produce a seed value

$$S = \langle K \rangle_{64}. \tag{1}$$

This seed value is used both as the key and data to produce an initialization vector IV and an intermediate key I. The extra 8 bits are ignored when S is used as a DES key. If this is unacceptable, the 56-fold of K could be taken to produce S', the key. We think this extra operation (i.e. *folding* 64 bits to 56 bits) is unnecessary in this case. However, if the key size and data size differ sufficiently, as with IDEA or SKIPJACK, this extra step would be needed.

IV is produced by encrypting the seed twice using that seed as a key; I is produced by decrypting the seed twice using the seed as a key:

$$IV = \{(\{S\}S)\}S \tag{2}$$
$$I = \{(\{S\}S^{-1})\}S^{-1} \tag{3}$$

where $\{P\}K$ denotes the encryption of plaintext P under control of key K and $\{P\}K^{-1}$ the corresponding decryption.

Note that we need **two** key schedules. One is the "secondary", to be used in preparation of the "primary" key schedule that will be used in data encryption. This paper is concerned with generation of the primary key schedule. Requirements for the secondary are less stringent; we believe that the DES-like generation of round keys is good enough (but that the regular shift of the IDEA generating key during round key generation isn't such a good idea). A shift with a regular step, interspaced by a few irregular steps—just as DES does—seems sufficient for our purposes.

The intermediate key I and IV are used to produce a generator key G. First, K is 192-folded to create three 64-bit chunks to be the keys for triple-DES. $\langle K \rangle_{192}$ is then encrypted under control of I using CFB-64 mode with initialization vector IV:

$$G = \{\langle K \rangle_{192}\}I \tag{4}$$

The bits of G are used to form keys for triple-DES in encrypt-decrypt-encrypt mode (parity bits are ignored). This time, we use triple-DES EDE CFB-64 mode on $\langle K \rangle_{768}$, using keys G and the same IV as before. The output of this encryption is used as the 768-bit key schedule for ordinary DES data encryption. It is obvious, that one can obtain as many bits of key material as needed by using $\langle K \rangle_M$, where M is the number of key bits required. This is useful for cases like described in [BB94] and in appendix on page 9.

To formalize, the key schedule generating algorithm is:

Algorithm 1 Generate Key Schedule $Sched$ from user key K of n bits

Require: $n \geq 40 \wedge n \leq 256$

$S \Leftarrow \langle K \rangle_{64}$;
$t \Leftarrow \{S\}S$; $IV \Leftarrow \{t\}S$; { single DES ECB}
$t \Leftarrow \{S\}S^{-1}$; $I \Leftarrow \{t\}S^{-1}$; { single DES^{-1} ECB}
$G \Leftarrow \{\langle K \rangle_{192}\}I$; { single DES CFB-64 using key I and init vector IV}
$Sched \Leftarrow \{\langle K \rangle_{768}\}G$; { triple DES EDE CFB-64 with key G and init vector IV}

3 Theory of Operation

Our basic assumption is that the underlying block cipher used is fundamentally strong, but possibly has an inappropriate key size, either too small (as in DES [DH77, Wie94]) or too large (as for an exportable cipher [DML$^+$93, DML$^+$94]). A fundamental design goal is to ensure that all input key bits have an equal effect on the generated key schedule.

We use n-folding (equation 1) to change the length of a bit string. Replication to the l.c.m. of the input and output lengths ensures that each input bit has equal weight. The rotation by 13 bits—a number that is likely to be relatively prime to any input key length of interest—serves to blur the effect of regularities in the input value [Bi94].

One's-complement addition of words of width n is used to generate the final fixed-length output value. Because we are generating a bit string, rather than a number, we use one's-complement addition, rather than the more common two's-complement addition. The arithmetic results are similar though not identical—the answers are in the range $[-(2^n - 1), +(2^n - 1)]$, rather than the more tractable reduction modulo 2^n. Also, this way carry bit is preserved.

If the input value to n-folding is already of the desired length, the operation has no effect; this is quite acceptable, since subsequent steps will scramble it sufficiently.

The next step, generation of I (equation 3) and IV (equation 2), is accomplished via a one-way function. Depending on the input value, n-

folding may not be strong enough, so we use the underlying block cipher as our primitive. Applying encryption and decryption twice (equations 2 and 3) serves to defeat the effects of weak input keys. For example, if K were 0, S would be 0 as well, which would result in I and IV having the same value if single encryption were used.[1]

While we know of no way to exploit this property, we still prefer to avoid the regularity.

Equation 4 is used to produce three DES keys for encryption of the key schedule bits. Triple encryption is necessary to guard against brute-force searches for the generating key. Assume an attack that can recover the key schedule bits used for, say, the first round. This is not an unreasonable supposition; there have been attacks published against shortened versions of DES [BS93]. In that case, a brute-force search could be used to find the generating key G, which could conceivably be used in an attempt to find the other subkey bits.

The final step produces the actual key schedule. We use 768-folding as an additional way to spread out the effects of the input key, but other mechanisms would probably work as well. One possibility would be the standard key schedule generation mechanism. We do recommend that the plaintext for the CFB-64 operation be key-dependent, to frustrate known-plaintext attacks on the generator key.

4 Attacks

Biham and Shamir have already studied the strength of DES with independent subkeys against **differential cryptanalysis** [BS93]. While the bits of our subkeys are not truly independent, they are sufficiently scrambled that there is no exploitable relationship between them. Either way, differential cryptanalysis becomes considerably more difficult, as every bit of every subkey of every round has to be recovered independently. On the order of 2^{61} chosen plaintexts is needed to mount a successful sliding attack, compared to $O(2^{47})$ when standard DES key schedule used [BS93].

A similar argument can be made with respect to **linear cryptanalysis** [Mat94a]. Unfortunately, the only successful results published were related to DES with a 56-bit key and the standard key schedule. Since the linear

[1] For ciphers with unpleasant property of $\{0\}0 = 0$, this may not hold, and 0 will be a weak key always. LOKI89 is one example of such cipher. In general, $\{K\}K = K$ is a bad property of a cipher to begin with, and we do not have a cure for it. Reminder: our method improves already good but imperfect ciphers, it does not turn bad ones into good ones.

expressions were computed for the assumption of independent subkeys, their probabilities stand. However, in our case it allows an attacker to recover only 12 bits from each the first and the last round subkeys (using $n-2$ approximation) with 2^{43} complexity and requiring 2^{43} known plaintext pairs. It is obvious however, that in order to recover the other 36 bits of the round subkeys in question, other approximations with weaker probabilities will have to be used. Biham [BB94] estimated the complexity of linear attack for independent subkeys $O(2^{60})$. Thus, linear attack seems impractical, though of course possible.

Furthermore, even if any subkey bits are recovered, going from them back to the input key—and hence possibly to some known plaintext with which to attack a key exchange dialog—would require cryptanalyzing triple DES with very little ciphertext. Such an attack is clearly infeasible. This makes the key generator[2] immune to potential cryptanalysis in the highly unlikely case that all the subkeys are recovered.

Since we allow variable-length input keys, a **brute-force** attack may be feasible if the user's key is too short. This is hardly a surprise; that is often the point of using short keys. Even so, our cipher is somewhat stronger than it might appear; key setup time is quite long, which would hinder any brute-force attack.

We do not see any shortcut attacks on this scheme. Even if there are some weaknesses in the early part, the final step—triple DES encryption of a moderately random string—should produce high-quality random numbers. We do not think it possible to cryptanalyze this operation. The only feasible attack would appear to be a cryptanalysis of DES itself that does not rely on the key schedule to aid in full key recovery.

Matsui-san [Mat94b] showed that if the order of the S-boxes is changed to "`56412738`", then DES becomes immune to linear cryptanalysis, but its resistance to differential attack drops to $O(2^{45})$ from $O(2^{47})$. This order of S-boxes could be used with our key schedule generation algorithm, since a differential attack would not be feasible. The same paper also showed that S-boxes order "`24673158`" is immune to both differential and linear attack. Combining this with our key schedule and a generating key length of at least 64 bits again results in a practically unbreakable cipher.

[2]It produces generating keys for users.

5 Analysis

What we have done is to specify a new algorithm for generating a key schedule. Many details of our scheme could easily be changed without affecting the essence.

The least important design detail is the n-folding algorithm. Other mechanisms for converting a user-specified key to output strings of different lengths would work as well. The properties we consider necessary are first, that all input bits should have an equal weight for all output lengths; and second, that the function should be at least weakly one-way. Arguably, ours does not go far enough in that direction, since it uses only linear functions.

The second detail is how we produce the generator key and IV. Again, the exact mechanism isn't important, though the same two properties should hold.

Our final design choice is how we actually generate the key schedule; that in turn reflects two decisions, our desire to use DES, and the precise mechanism used to produce the plaintext. The former is much more important: we have a great deal of confidence in the basic strength of DES, and in the randomness of its output. While not perfect [Mat94a], we do not think there are any serious threats here; the amount of ciphertext is quite low, and known plaintext should be all but impossible to obtain.

Clearly, any other one-way pseudorandom function could be used instead. But if, say, MD5 [Riv92] were used, it would introduce another element to analyze. Besides, there is both a conceptual and an implementation gain in elegance by using only one cryptographic function. Of course, using DES slows down key setup; as discussed below, this is not necessarily a disadvantage.

There is more room for choice in the selection of the plaintext. While we have made ours key-dependent, another possibility would be to make it key family-dependent, much as with [Bl94b] and the Soviet GOST cipher [GOST, Ch+94]. This would allow for separate communities of interest without the risks run by tinkering with the S-boxes.

Most of the strength of our cipher is derived from the strength of the underlying block cipher. This is intentional; in common with the designers of CDMF [DML+93, DML+94], we prefer to build on foundations that are known to be strong. We could change some aspects of the DES design — for example, the E-box would no longer need to be tied to PC2, since the latter would no longer exist—but we are very reluctant to upset a delicate engineering balance. This brings to light yet another advantage of this approach — any DES hardware can utilize it, as long as it allows for downloading the key schedule (vs. insisting on downloading 56-bit key and deriving the subkeys internally, by itself). We are aware of at least one manufacturer, that produces DES hardware capable of downloading key schedule directly in.

Very clearly, though, our key schedules have no more information than the input key. If 40 bits are fed in, a brute-force attack on those 40 bits will work, though the attack would be much more expensive than with CDMF; our key setup requires 43 encryption operations, rather than two.

The longer key setup time could be problematic for things like network encryptors, which need to maintain very many encryption contexts. Still, memory is cheap, and caching the key schedule is often done even in today's implementations. The total key schedule is only 96 bytes; most likely, other state that must be kept per security association would be at least that large. So the total storage requirements per key would not increase that much.

In another vein, our cipher will encrypt and decrypt as quickly as DES. This is an advantage compared with other secure ciphers like triple-DES or DES with 32 rounds and the traditional key schedule. In short, we have traded setup time for run time, which is almost always a good deal.

Our key schedule also eliminates weak keys and the complementation property of DES. To see that the former holds, observe that weak keys in DES arise because of regularities in the key schedule algorithm. If the bits of our key schedule are uniformly randomly distributed—a fair assumption, given the known randomization properties of DES—it is exceedingly improbable that, say, $K_1 = K_{16}$, let alone that that would hold for all 8 pairs of key schedule registers. The complementation property occurs because selections of the key bits are directly exclusive-ORed with plaintext fields; our scrambling steps eliminate that regularity.

Our approach can be used together with other proposed and published improvements of DES, such as [BB94]. In this case, all the key material can be derived from the main user key. This approach improves the strength of any DES or DES-like variation, including different or key-dependent S-boxes.

We recommend our approach to be used with every iterated block cipher, as it maximizes the cipher's strength while preserving it's bulk encryption speed.

6 Acknowledgments

We are indebted to Eli Biham, Don Coppersmith and Jim Reeds; they made many valuable suggestions during the writing of this paper.

Appendix A DES-SK/N Family

Following the example of R. Rivest [Riv94], we define a DES-SK family with variable number of rounds (between 16 and 64), where each round is equal to a round of standard DES, except in the way round key is obtained. We will use our key schedule to generate round keys for any number of rounds.

Judging by the results of differential and linear cryptanalysis, we conjecture, that the number of rounds should be multiple of 8 and in total preferably 24 or more.

Not to be limited by original number of rounds in ANSI DES, we define a "secondary" key schedule (used to generate the "primary" key schedule) in the following way:

Algorithm 2 Generate secondary key schedule for DES-SK/N

Require: Number of rounds r, key K of n bits
Require: $n \geq 40 \;\; \wedge \;\; n \leq 256$
Require: $r \geq 16 \;\; \wedge \;\; r \leq 64$
Require: r is multiple of 8
Ensure: n relatively prime to 13 and 23 that used for shifts
  Shift will be 13 bits circular to the right for all rounds
  Shift for rounds 2, 3, $\lfloor r/2 \rfloor - 3$, $\lfloor r/2 \rfloor + 2$, $r - 5$, $r - 3$ will be 23 bits circular to the right
  $Q \Leftarrow \langle K \rangle_{136}$
  for $i = 1$ to r **do**
    Take first 6 bytes of Q and copy into subkey of round i
    Rotate Q to the right by $shift$ bits
  end for

As a primary key schedule generator we can use algorithm 1 on page 4, or algorithm 3 on the next page.

Appendix A.1 DES-SK/32 with 32 rounds

It appears, that 32 rounds is as little as one can comfortably have. Note, that both SKIPJACK and GOST use 32 rounds. We'll use key schedule as described above (see algorithm 1 on page 4 and evaluate the strength of the resulting cipher[3], which for the sake of simplicity we call DES-SK/32 (vs.

[3] We can use key schedule algorithm 3 on the following page, but it will require further analysis to prove that secondary key schedule (algorithm 2) is not weaker than ANSI DES. Experts suggested that algorithm 2 is good enough, especially when r is 32 and up.

Algorithm 3 Generate primary Schedule for DES-SK/N

Require: Secondary schedule set up (see 2 on the preceding page)
Require: Number of rounds r, key K of n bits
Require: $n \geq 40 \;\wedge\; n \leq 256$
Require: $r \geq 16 \;\wedge\; r \leq 64$
Require: r is multiple of 8

$S \Leftarrow \langle K \rangle_{80}$
$IV \Leftarrow \{\{S\}S\}S$ { DES-SK CBC}
$I \Leftarrow \{\{S\}S^{-1}\}S^{-1}$ { DES-SK CBC}
$G \Leftarrow \{\langle K \rangle_{240}\}I$ { DES-SK CFB-64 key I init vector IV}
$Sched \Leftarrow \{\langle K \rangle_{r*48}\}G$ { triple DES-SK EDE CFB-64 key G init vector IV}

traditional DES of 16 rounds). Let us select generating key 80 bits long.

Before proceeding, it's worth to mention that if an adversary has 2^{64} known plaintexts, he has a complete dictionary for the given key, and more that 2^{64} plaintexts simply doesn't exist .

We will deal with dictionary attack and possible defense against it later.

- **Brute force**: obviously, since the generating key is 80 bits — ...
- **Performance**: conjectured to be one half of "normal" single DES performance, hardware- or software-based.
- **Differential attack**: conjectured to require at least 2^{122} chosen plaintexts. Therefore we are immune to this attack.
- **Linear attack**: not bothering with probability calculations, conjectured to require at least $O(2^{86})$ known plaintexts[4]. Therefore this attack is impossible.
- **Weak keys**: doesn't have.
- **Complementary property**: doesn't have.

Note, that the algorithm is built on well-known and deeply researched cryptographic primitives. And the hardware exists, that can utilize it.

[4] $O(2^{120})$ seems more likely.

Appendix A.2 DES-SK/48 or Triple-DES?

We notice, that 3DES in essence is nothing more than 48 physical (or about 46 actual[5]) rounds of standard DES, where the regular key schedule is "disrupted" in three places, because different parts of it are generated by different bits of the main key. However, the relations, that make the attack easier, are still there. I.e. every 16 rounds exhibit tractable relations between the subkeys.

On the other hand, using our key schedule and applying 48 rounds of straight DES scrambling, we get better security due to lack of exploitable relations between the round keys and much higher barrier on cryptanalytic attacks, with performance (hardware or software) better than that of 3DES. But clearly, both 3DES and DES-SK/48 seem an overkill, as DES-SK/32 will do as good a job with only about 65% of computational expenses.

Appendix B Reference Source Code

Appendix B.1 Copyright

```
/*********************************************/
/* Copyright (C) 1996 Uri Blumenthal, IBM Corp. */
/* Use as you wish, as long you hold the author */
/* and IBM harmless. */
/*********************************************/

/**********************************************/
/* Reference (illustration) code for Key Schedule */
/* Generation defined in "Better Key Schedule for */
/* DES-like ciphers" article published in 1996 in */
/* PragoCrypt'96 conference proceedings. */
/* This code was designed bulletproof, foolproof, */
/* speedproof and thoughtproof. The only criteria */
/* it had to satisfy was performing according to */
/* the algorithm, specified in this article. And */
/* so it does. */
/**********************************************/
```

Appendix B.2 GCD and LCM

```
#include <stdlib.h>
```

[5]Since there is no swap between the last round of one "sweep" of single DES and the first round of the next "sweep" of single DES — these two rounds are not as strong as "full-fledged" two DES rounds.

```
/ * *********************************************/
/* Copyright (C) 1996 Uri Blumenthal, IBM Corp. */
/* Use as you wish, as long you hold the author */
/* and IBM harmless. */
/ * *********************************************/

/* Modern Euclidian GCD Algorithm from "The Art of */
/* Computer Programming" by Knuth vol.II page 320. */

int gcd (int u, int v)
{
  int r = 0;

  if (u < 0) u = -u; if (v < 0) v = -v;
  if (u < v) {
    r = u; u = v; v = r;
  }

  while (v ≠ 0) {
    r = u % v; u = v; v = r;
  }
  return u;
}

int lcm (int u, int v)
{
  int g = gcd(u, v);
  if (g == 0) return 0;
  return (u * v / g);
}

#ifdef GCD_TEST
#include <stdio.h>
int main(int argc, char **argv)
{
  void usage(char *s);
  int a, b, c;

  printf("GCD and LCM tester...\n");
  if (argc > 1 && argc ≠ 3) usage(argv[0]);
  if (argc == 1) {
    a = 40902; b = 24140;
    printf("Expected gcd(%d, %d) = 34\n", a, b);
    printf("Expected lcm(%d, %d) = 29040420\n", a, b);
  } else {
```

```
    a = atoi(argv[1]); b = atoi(argv[2]);
  }
  printf("gcd(%d, %d) = ", a, b);
  c = gcd(a, b);
  printf("%d\nlcm(%d, %d) = ", c, a, b);
  c = lcm(a, b);
  printf("%d\n", c);
  exit(0);
}

void usage(char *s)
{
  printf("Usage:  %s [first-integer second-integer]\n", s);
  exit(1);
}
#endif                                   /* GCD_TEST */
```

Appendix B.3 String Rotation

```
#include <memory.h>
#include "copyright.h"

#ifdef ROTATE_TEST
#include <stdio.h>
#endif                                   /* ROTATE_TEST */

/* rot13(byte *dst, byte *src, int length) */
/* Rotates a byte string 13 bits to the right. */
/* Can deal with overlapping buffers. */
/* Passing a NULL pointer will crash this code. */
/* Error discovery could be put in, but... */

/* Basically, we shift every byte by 5 bits to the right */
/* and then "renumber" the bytes, so that: */
/* dst[0] <- src[n]; */
/* dst[1] <- src[0]; */
/* ................. */
/* dst[n] <- src[n-1]. */

void rot13 (unsigned char *dst, unsigned char *src, int len)
{
  int i;
  unsigned char b;
  unsigned char *buf;
```

```
  if (len == 0 || src == NULL || dst == 0) return;
  buf = (unsigned char *) malloc (len + 1);

  b = src[len - 1] & 0x1F;

  for (i = len - 1; i > 0; i--)
    buf[i+1] = ((src[i] >> 5) & 0x7) | ((src[i-1] & 0x1F) << 3);
  buf[1] = ((src[0] >> 5) & 0x7) | (b << 3);
  buf[0] = buf[len];
  for (i = 0; i < len; i++) dst[i] = buf[i];

  free (buf);
}

#ifdef ROTATE_TEST
int main (int argc, char **argv)
{
  char buf[10];
  int i;

  printf("ROT13 Test...\n");
  memset (buf, 0, sizeof(buf));
  if (argc > 1) i = atoi(argv[1]) % 10;
  else i = 9;
  buf[i] = 1;
  printf("Buffer before rot:  ");
  for (i = 0; i < 10; i++) printf("%02x", buf[i]); printf("\n");
  rot13(buf, buf, 10);
  printf("Buffer after rot:  ");
  for (i = 0; i < 10; i++) printf("%02x", buf[i]); printf("\n");
  exit(0);
}
#endif                                          /* ROTATE_TEST */
```

Appendix B.4 One's Complement Addition

```
/* Multiprecision One-Complement addition */
/* All numbers are represented Big Endian */

void ocadd(unsigned char *a,                    /* first additive */
           unsigned char *b,                    /* second additive */
           unsigned char *dst,                  /* result (sum) */
           int len)          /* how many bytes represent the numbers */
{
  int k = 0;
```

```
  int i;

#ifdef DEBUG_OCADD
  int m;
  printf("\tAdding numbers:\n");
  for (m = 0; m < len; m++) {
    printf("%02x", a[m]);
    if ((m + 1) % 20 == 0) printf("\n");
  }
  printf("\n\tand\n\t");
  for (m = 0; m < len; m++) {
    printf("%02x", b[m]);
    if ((m + 1) % 20 == 0) printf("\n");
  }
  printf("\n");
#endif

  for (i = len - 1; i >= 0; i--) {
    dst[i] = (a[i] + b[i] + k) % 256;
    k = (a[i] + b[i] + k) / 256;
  }

#ifdef DEBUG_OCADD
  printf("\taddition gave:\n");
  for (m = 0; m < len; m++) {
    printf("%02x", dst[m]);
    if ((m + 1) % 20 == 0) printf("\n");
  }
  printf("\n\tcarry bit is %1d\n", k);
  m = 0;
#endif

  while (k > 0) {                          /* guaranteed to run at most twice */
#ifdef DEBUG_OCADD
    printf("Doing %d-th round of add/propagate carry bit\n", ++m);
#endif
    for (i = len - 1; i >= 0; i--) {
      dst[i] = (dst[i] + k) % 256;
      k = (dst[i] + k) / 256;
    }
  }

  return;
}
```

```
#ifdef OCADD_TEST
#include <stdio.h>
#include <memory.h>

int main (int argc, char **argv)
{
  unsigned char a[11], b[11], c[11];
  int k = 0;
  memset(a, 0, sizeof(a));
  memset(b, 0, sizeof(a));
  memset(c, 0, sizeof(a));
  printf("OCADD Test...\n");
  a[0] = 0xff; b[0] = 9;
  printf("a:  ");
  for (k = 0; k < sizeof(a); k++) printf("%02x", a[k]);
  printf("\n");
  printf("b:  ");
  for (k = 0; k < sizeof(b); k++) printf("%02x", b[k]);
  printf("\n");
  ocadd(a, b, c, 11);
  printf("c:  ");
  for (k = 0; k < sizeof(c); k++) printf("%02x", c[k]);
  printf("\n");
  exit(0);
}
#endif                          /* OCADD_TEST */
```

Appendix B.5 N-Folding

```
#include <stdlib.h>
#include <memory.h>
#if defined(NFOLD_TEST) || defined(DEBUG_NFOLD)
#include <stdio.h>
#endif                  /* TEST_NFOLD | DEBUG_NFOLD */

/****************************************/
/* Provides n-folding of the input string */
/* of m bytes long to a string of n bytes */
/* long. Buffers better not overlap! */
/****************************************/
int nfold(unsigned char *in, int m,
          unsigned char *out, int n)
{
  unsigned char *buf = NULL;              /* tmp storage */
  unsigned char *a = NULL;                       /* save "in" for rot13 */
```

```
  int LowestCommonMult = 0;                    /* LCM(m, n) */
  int i = 0;
#ifdef DEBUG_NFOLD
  int k = 0;
#endif

  /* functions we need */
  extern int lcm(int u, int v);
  extern int rot13(unsigned char *a,
                   unsigned char *b,
                   int len);
  extern int ocadd(unsigned char *add1,
                   unsigned char *add2,
                   unsigned char *sum,
                   int len);

  /* Stupidity isn't welcome */
  if (n <= 0 || m <= 0) return -1;

  /* Prepare tmp place for the input so that */
  /* we don't damage the original with rot13 */
  a = (unsigned char *) malloc (m);
  if (a == NULL) return -2;                    /* no memory, bad */
  memcpy(a, in, m); /* save the input string */

  /* Compute Lowest Common Multiplier of the */
  /* input string length and desired output */
  /* string length. */
  LowestCommonMult = lcm(m, n);
#ifdef DEBUG_NFOLD
  printf("Input string is:\n");
  for (k = 0; k < m; k++) {
    printf("%02x", in[k]);
    if (((k + 1) % 20) == 0) printf("\n");
  }
  printf("\n");
  printf("Computed LCM(%d, %d) = %d\n", m, n,
         LowestCommonMult);
#endif

  buf = (unsigned char *) malloc(LowestCommonMult);
  if (buf == NULL) return -3;                  /* no mem, bad */

  /* Replicate the input th the LCM length */
  for (i = 0; i < (LowestCommonMult / m ); i++) {
```

```
    memcpy((unsigned char *)&buf[i * m], a, m);
    rot13(a, a, m);
#ifdef DEBUG_NFOLD
    printf("after %d-th replication we rotated input to:\n", i+1);
    for (k = 0; k < m; k++) {
      printf("%02x", a[k]);
      if (((k + 1) % 20) == 0) printf("\n");
    }
    printf("\n");
#endif
  }

  memset(out, 0, n); /* just in case */

#ifdef DEBUG_NFOLD
  printf("Note - result of last replication is thrown out.\n");
  printf("\nReplicated string (buffer filled) is:\n");
  for (k = 0; k < LowestCommonMult; k++) {
    printf("%02x", buf[k]);
    if (((k + 1) % 20) == 0) printf("\n");
  }
  printf("\n");
#endif


  /* Now we view the buf as set of n-byte strings */
  /* Add the n-byte long chunks together, using */
  /* one's complement addition, storing the */
  /* result in the output string. */
  for (i = 0; i < (LowestCommonMult / n); i++) {
    ocadd(out, (unsigned char *)&buf[i * n], out, n);
#ifdef DEBUG_NFOLD
    printf("after %d-th one's complement addition sum is:\n", i+1);
    for (k = 0; k < n; k++) {
      printf("%02x", out[k]);
      if (((k + 1) % 20) == 0) printf("\n");
    }
    printf("\n");
#endif
  }

  free(buf); free(a); /* don't be hoggish */

  return 0;                    /* A.O.K. */
}
```

```
#ifdef NFOLD_TEST
#include <stdio.h>
int main(int argc, char **argv)
{
  unsigned char key[41];
  unsigned char res[64];
  int i = 0, rc = 0;

  printf("N-Fold test (strings/numbers treated Big Endian)\n");

  memset(key, 0, sizeof(key));
  memset(res, 0, sizeof(res));
  for (i = 0; i < 40; i++)
    key[i] = (char) (060 + (i % 10));
  printf("We will 64-fold 40-bit long string:\n");
  printf("\"%s\" in ASCII\n\n", key);

  rc = nfold(key, 40, res, 64);

  printf("\nN-Fold returned %d\nResulting string is:\n", rc);

  for (i = 0; i < sizeof(res); i++) {
    printf("%02x", res[i]);
    if (((i + 1) % 20) == 0) printf("\n");
  }
  printf("\n");

}
#endif                                        /* NFOLD_TEST */
```

Appendix B.6 Output of N-Fold

```
N-Fold test (strings/numbers treated Big Endian)
We will 64-fold 40-bit long string:
"012345" in ASCII

Input string is: 303132333435

Computed LCM(6, 8) = 24
after 1-th replication we rotated input to: a1a981899199
after 2-th replication we rotated input to: 8ccd0d4c0c4c
after 3-th replication we rotated input to: 626466686a60
after 4-th replication we rotated input to: 530313233343
Note - result of last replication is thrown out.
```

```
Replicated string (buffer filled) is:
303132333435a1a9818991998ccd0d4c0c4c626466686a60

after 1-th one´s complement addition sum is: 303132333436a2a9
after 2-th one´s complement addition sum is: b2bbc4cdc003b0f6
after 3-th one´s complement addition sum is: be072631266b1a56

N-Fold returned 0
Resulting string is: be072631266b1a56
```

References

[BS93] Eli Biham and Adi Shamir. *Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, Berlin, 1993.

[Bi94] Eli Biham. New Types of Cryptanalytic Attacks Using Related Keys. In *Advances in Cryptology: Eurocrypt '93*, pages 413–424. Springer-Verlag, 1994.

[BB94] Eli Biham and Alex Biryukov. How to Strengthen DES Using Existing Hardware. In *Advances in Cryptology: ASIACRYPT '94*, pages 398–412. Springer-Verlag, 1995.

[Bl94a] Uri Blumenthal. Improvements to DES. In *SECURITY, 5th IBM Interdivisional Technical Symposium, 1994*, pages 979–982.

[Bl94b] Uri Blumenthal. Improvements to SEAL. In *SECURITY, 5th IBM Interdivisional Technical Symposium, 1994*, pages 976–978.

[Ch+94] C. Charnes, L. O'Connor, J. Pierzyk, R. Safavi-Naini, Y. Zheng. Further Comments on the Soviet Encryption Algorithm. *TR, University of Wollongong, NSW 2500, AUSTRALIA*, 1994.

[DH77] Whitfield Diffie and Martin E. Hellman. Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6):74–84, June 1977.

[Kn93] Lars R. Knudsen. Practically secure Feistel ciphers. In *Fast Software Encryption. Cambridge Security Workshop, Proceedings*, pages 211–221. Springer-Verlag, 1994.

[DML+93] D.B.Johnson, S.M. Matyas, A.V. Le, , and J.D. Wilkins. Design of the commercial data masking facility data privacy algorithm. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 93–96, November 1993.

[DML+94] D.B.Johnson, S.M. Matyas, A.V. Le, , and J.D. Wilkins. The commercial data masking facility (CDMF) data privacy algorithm. *IBM Jour. of Research and Development*, 38(2):217–226, March 1994.

[Mat94a] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology: Eurocrypt '93*, pages 386–397. Springer-Verlag, 1994.

[Mat94b] Mitsuru Matsui. On Correlation Between the Order of S-boxes and the Strength of DES. In *Proceedings of EUROCRYPT'94*.

[NBS77] NBS. Data encryption standard, January 1977. Federal Information Processing Standards Publication 46.

[Riv92] R. Rivest. The MD5 message-digest algorithm. Request for Comments (Informational) RFC 1321, Internet Engineering Task Force, April 1992.

[Riv94] R. Rivest. The RC5 encryption algorithm. In *Proceedings of the Workshop on Cryptographic Algorithms*, K.U.Leuven, December 1994, To appear.

[Wie94] Michael J. Wiener. Efficient DES key search. Technical Report TR-244, School of Computer Science, Carleton University, Ottawa, Canada, May 1994. Presented at the Rump Session of Crypto '93.

[GOST] ГОСТ 28147-89. Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования. *National Soviet Bureau of Standards. Data Processing systems.Cryptographic Protection. Cryptographic Transform Algorithm. GOST 28147-89*, 1989.