Fast Software Encryption Functions

Ralph C. Merkle
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

Encryption hardware is not available on most computer systems in use today. Despite this fact, there is no well accepted encryption function designed for software implementation -- instead, hardware designs are emulated in software and the resulting performance loss is tolerated. The obvious solution is to design an encryption function for implementation in software. Such an encryption function is presented here -- on a SUN 4/260 it can encrypt at 4 to 8 megabits per second. The combination of modern processor speeds and a faster algorithm make software encryption feasible in applications which previously would have required hardware. This will effectively reduce the cost and increase the availability of cryptographic protection.

Introduction

The computer community has long recognized the need for security and the essential role that encryption must play. Widely adopted, standard encryption functions will make a great contribution to security in the distributed heavily networked environment which is already upon us. IBM recognized the coming need in the 1970's and proposed the Data Encryption

Copyright (c) 1998, Springer-Verlag

Standard, or DES [19]. Although controversy about its key size has persisted, DES has successfully resisted all public attack and been widely accepted. After some debate its use as a U.S. Federal Standard was reaffirmed until 1992 [14]. However, given the inherent limitations of the 56 bit key size used in DES [16] it seems clear that the standard will at least have to be revised at some point. A recent review of DES by the Office of Technology Assessment [15] quotes Dennis Branstad as saying the "useful lifetime" of DES would be until the late 1990's.

Despite the widespread acceptance of DES the most popular software commercial encryption packages (for, e.g., the IBM PC or the Apple Macintosh) typically offer both DES encryption and their own home-grown encryption function. The reason is simple -- DES is often 50 to 100 times slower than the home-grown alternative. While some of this performance differential is due to a sub-optimal DES implementation or a faster but less secure home-grown encryption function, it seems that DES is inherently 5 to 10 times slower than an equivalent, equally secure encryption function designed for software implementation. This is not to fault DES. One of the design objectives in DES was quite explicitly a fast hardware implementation; when hardware is available, DES is an excellent choice. However, a number of design decisions were made in DES reflecting the hardware orientation which result in slow software performance -- making the current extensive use of DES in software both unplanned-for and rather anomalous.

Having offered a rationale for an encryption function designed for software implementation, we now turn to the design principles, followed by the actual design.

Basic Principles

The basic design principles in DES seem sound. The fact that DES has not been publicly broken speaks in their favor. However, upon examining specific design decisions in DES, we find that several should be revised -- either in light of the software orientation of the new encryption function, or because of the decreased cost of hardware since the early '70's. Examining the basic design decisions one by one, and in no particular order, we can decide

Copyright (c) 1998, Springer-Verlag

what reasonably should be changed.

The selection of a 56 bit key size is too small, a problem which can be easily remedied. This subject has already been debated extensively, and while 56 bits seems to offer just sufficient protection for many commercial applications, the negligible cost of increasing the key size virtually dictates that it be done.

The extensive use of permutations is expensive in software, and should be eliminated -- provided that a satisfactory alternative can be found. While permutations are cheap in hardware and provide an effective way to spread information (also called "diffusion" [21]) they are not the best choice for software. In the faster implementations of DES, the permutations are implemented by table look-ups on several bits at once. That is, the 48-to-32 bit permutation P is implemented by looking up several bits at once in a table -- where each individual table entry is 32 bits wide and the table has been pre-computed to contain the permutation of the bits looked up. Using a table-lookup in a software encryption function seems a good idea and can effectively provide the desired "diffusion" -- however there seems no reason to limit such a table to being a permutation. Having once paid the cost of looking up an entry in a table, it seems preferable that the entry should contain as much information as possible rather than being arbitrarily restricted to a small range of possibilities.

Each individual S-box in DES provides only 64 entries of 4 bits each, or 32 bytes per S-box. Memory sizes have greatly increased since the mid 1970's when DES was designed, and larger S-boxes seem appropriate. More subtly, DES uses 8 S-boxes and looks up 8 different values in them simultaneously. While this is appropriate for hardware (where the 8 lookups can occur in parallel) it seems an unreasonable restriction for software. In software, each table lookup must follow the preceding lookups anyway -- for that is the nature of sequential program execution. It seems more valuable cryptographically to make each lookup depend upon the preceding lookup. This means that the cascade of unpredictable changes that are so central to DES-type encryption functions can achieve greater depth with fewer lookups. Looked at another way, DES has a maximum circuit depth of 16 S-boxes, even though it has a total of 128 S-box lookups. If those same 128 S-box operations were done sequentially, with the output of each lookup operation altering the input to the next

Copyright (c) 1998, Springer-Verlag

lookup, then the maximum circuit depth would be 128 S-boxes -- eight times as many and almost certainly providing greater cryptographic strength. This change would have very little impact on the running time of a software implementation on a typical sequential processor. We conclude that a larger S-box size and sequential (rather than parallel) S-box usage should be adopted.

The initial and final permutations in DES are widely viewed as cryptographically pointless -- or at least, not very important. They are therefore discarded.

The key schedule in DES has received mild criticism for not being sufficiently complicated[9]. In practice, all of the faster DES software implementations pre-compute the key schedule. This pre-computation seems a good idea when large volumes of data are being encrypted -- the pre-computation allows a more leisurely and careful arrangement of the encryption tables and means the actual encryption function can more rapidly scramble the data with less effort. A more complex key pre-computation therefore seems desirable.

Finally, the design criteria used for the DES S-boxes were kept secret. Even though there is no particular reason to believe that they conceal a trap door, it would seem better if the criteria for S-box selection were made explicit, and some sort of assurances provided that the S-boxes were actually chosen randomly in accordance with the published criteria. This would both quiet the concerns about trap doors, and also allow a fuller and more explicit consideration of the S-box selection criteria.

With this overview of design principles we can now proceed to the design.

Khufu, Khafre and Snefru

There are actually two encryption functions named Khufu and Khafre, and a one-way hash function named Snefru. All three names were taken from the Pharoahs of ancient Egypt following a suggestion by Dan Greene. To quote the Encyclopedia Britannica "The ideal pyramid was eventually built by Snefru's successor, Khufu, and the first --- the Great Pyr-

Copyright (c) 1998, Springer-Verlag

amid at Giza --- was the finest and must successful." Khafre was Khufu's son.

The basic hardware model around which they are optimized is a 32-bit register oriented microprocessor. The basic operations are 32-bit load, store, shift, rotate, "xor" and "and".

The two encryption functions are optimized for somewhat different tasks, but use similar design principles. Khufu is designed for fast bulk encryption of large amounts of data. To achieve the fastest possible speed, the tables used in encryption are pre-computed. This pre-computation is moderately expensive, and makes Khufu unsuited for the encryption of small amounts of data. The other encryption function -- Khafre -- does not require any pre-computation. This means Khafre can efficiently encrypt small amounts of data. On the other hand, Khafre is somewhat slower than Khufu for the encryption of large volumes of data because it takes more time to encrypt each block.

The one-way hash function -- Snefru -- is designed to rapidly reduce large blocks of data to a small residue (perhaps 128 or 256 bits). Snefru requires no pre-computation and therefore can be efficiently applied to small arguments. Snefru provides authentication and does not provide secrecy. Snefru is discussed in a separate paper[24]. The C source for Snefru is available by anonymous ftp from arisia.xerox.com (13.1.100.206) in directory /pub/hash

We first discuss the design of Khufu.

Khufu

Khufu is a block cipher operating on 64-bit blocks. Although increasing block size was a very tempting design alternative, the 64-bit block size of DES has not been greatly criticized. More important, many systems built around DES assume that the block size is 64 bits. The pain of using a different encryption function is better minimized if the new encryption function can be easily "plugged in" in place of the old -- which can be done if the block size is the same and the key size is larger. The new encryption function essentially looks exactly like the old encryption function -- but with some new keys added to the

Copyright (c) 1998, Springer-Verlag

key space. Increasing the block size might have forced changes in more than just a single subroutine -- it might (for example) have forced changes in data formats in a communications systems.

Khufu, like DES, is a multi-round encryption function in which two 32-bit halves (called L and R for Left and Right) are used alternately in the computations. Each half is used as input to a function F, whose output is XORed with the other half -- the two halves are exchanged and the computation repeated until the result appears to be random (no statistically detectable patterns). Khufu uses a different F-function than DES -- and uses multiple different F-functions during the course of encryption. One round of DES uses an F-function defined by 8 table lookups and associated permutations. By contrast, one round of Khufu uses a single table lookup in a larger S-box. In addition, in the first step of encryption (prior to the main loop) the plaintext is XORed with 64 bits of key material, and again in the final step of encryption (following the main loop) the 64-bit block is XORed with another 64 bits of key material to produce the ciphertext.

We will need to refer to the 4 bytes in a 32-bit word, and will adopt the "big-endian" convention. Byte 0 is the leftmost (most significant) byte while byte 3 is the rightmost (least significant) byte. The 8 bytes in a 64-bit block will be numbered 0 through 7, again with byte 0 being the leftmost (most significant) byte while byte 7 is the rightmost (least significant) byte.

The algorithm proceeds as follows: the 64-bit plaintext is first divided into two 32-bit words designated L and R. L is bytes 0 through 3, and R is bytes 4 through 7 of the 64-bit plaintext. L and R are then XORed with two 32-bit words of auxiliary key material. Then the main loop is started, in which byte 3 (the least significant byte) of L is used as the input to a 256-entry S-box. Each S-box entry is 32-bits wide. The selected 32-bit entry is XORed with R. L is then rotated to bring a new byte into position, after which L and R are swapped. The S-box itself is changed to a new S-box after every 8 rounds (we shall sometimes call 8 rounds an "octet"). This means that the number of S-boxes required depends on the number of rounds of encryption being used: one new S-box for every octet. Finally, after the main loop has been completed, we again XOR L and R with two new 32-bit auxiliary key values

Copyright (c) 1998, Springer-Verlag

to produce the ciphertext.

For efficiency reasons, we restrict the number of rounds to be a multiple of 8, i.e., an integral number of octets. If the main encryption loop is always executed a multiple of 8 times, then it can be unrolled 8 times -- which is substantially more efficient than the definitionally correct but inefficient versions given in this paper. For this reason, the variable "enough" given below must be an exact multiple of 8. Various integer calculations will not work correctly for values of "enough" that are not multiples of 8. Encryption of a single 64-bit plaintext by Khufu can be viewed algorithmically as follows:

```
L, R: int32;
enough: integer;  -- the security parameter, default of 16 seems appropriate.
                  -- values of 8, 16, 24, 32, 40, 48, 56, and 64 are possible.
SBoxes: ARRAY [1 .. enough/8] OF ARRAY [0 .. 255] OF int32;  -- key material
AuxiliaryKeys:  ARRAY[1 .. 4] OF int32;  -- additional key material
rotateSchedule: ARRAY [1 .. 8] = [16,16,8,8,16,16,24,24];
octet: integer;  -- really (round+7)/8, it keeps track of which
                 -- 8-round "octet" we are currently in

L = L  XOR AuxiliaryKeys[1];
R = R  XOR AuxiliaryKeys[2];

octet = 1;

FOR round = 1 TO enough DO  -- Note that "enough" must be a multiple of 8
Begin
   R = R XOR SBoxes[octet] [L AND #FF];
   L = RotateRight[L, rotateSchedule[(round-1) mod 8 + 1] ];
   SWAP[L,R];
   if (round mod 8 = 0) then octet = octet+1;
```

Copyright (c) 1998, Springer-Verlag

```
End;

L = L  XOR AuxiliaryKeys[3];
R = R  XOR AuxiliaryKeys[4];
```

Notationally, it will be convenient to index the different variables at different rounds. This indexing is explicitly given by re-writing the above algorithm and replacing L and R with arrays. In addition, we add the array "i" to denote the indices used to index into the S-box.

```
L, R: ARRAY [-1 .. enough+1] OF int32;
enough: integer;  -- the security parameter, default of 16 seems appropriate.
                  -- values of 8, 16, 24, 32, 40, 48, 56, and 64 are possible.
i: ARRAY[0 .. enough] OF int8;  -- 8-bit bytes
SBoxes: ARRAY [1 .. enough/8] OF ARRAY [0 .. 255] OF int32;  -- key material
AuxiliaryKeys:  ARRAY[1 .. 4] OF int32;  -- additional key material
rotateSchedule: ARRAY [1 .. 8] = [16,16,8,8,16,16,24,24];
octet: integer;  -- really (round+7)/8, it keeps track of which 8-round
                 -- "octet" we are currently in

L[0] = L[-1] XOR AuxiliaryKeys[1];
R[0] = R[-1] XOR AuxiliaryKeys[2];

octet = 1;

FOR round = 1 TO enough DO  -- Note that "enough" must be a multiple of 8
Begin
   i[round] = L[round-1] AND #FF
   L[round] = R[round-1] XOR SBoxes[octet] [ i[round] ];
   R[round] = RotateRight[L[round-1], rotateSchedule[ (round-1) mod 8 + 1] ];
   if (round mod 8 = 0) then octet = octet+1;
End;
```

Copyright (c) 1998, Springer-Verlag

L[enough+1] = L[enough] XOR AuxiliaryKeys[3];
R[enough+1] = R[enough] XOR AuxiliaryKeys[4];

The plaintext is (by definition) [L[-1], R[-1]], while the ciphertext is [L[enough+1], R[enough+1]]. By definition, round 1 computes L[1] and R[1] from L[0] and R[0], using index 1 -- or i[1]. Similarly, round n computes L[n] and R[n] from L[n-1] and R[n-1] using i[n]. We shall sometimes say that "round" 0 computes L[0] and R[0] from L[-1] and R[-1], and that "round" enough+1 computes L[enough+1] and R[enough+1] from L[enough] and R[enough].

The primary purpose of the rotation schedule is to bring new bytes into position so that all 8 bytes of input are used in the first 8 rounds (or first octet). This means that a change in any single input bit is guaranteed to force the use of a different S-box entry within 8 rounds, and so initiate the cascade of unpredictable changes needed to scramble the input. A secondary purpose of the rotation schedule is to maximize the number of rotates by 16 because they tend to be faster on many microprocessors. For example, the 68000 has a SWAP instruction which is equivalent to rotating a 32-bit register by 16 bits. Also, rotation by 16 tends to be very fast on processors with 16 bit registers -- simply by altering one's viewpoint about which register contains the lower 16 bits and which register contains the upper 16 bits it is possible to perform this operation with no instructions at all. The final purpose of the rotation schedule is to restore the data to its original rotational position after each octet of 8 rounds. Thus, the sum of the rotations is equal to 0 modulo 32.

A different S-box is used after each octet of encryption. This has two beneficial effects: first, it means that the same S-box entry will never be used twice with the same rotational alignment. That is, if a single S-box were used for all octets, then it might be that i[1] (the index used to select an S-box entry on the first round) and i[9] might be the same -- and therefore the same S-box entry would be used in rounds 1 and 9. These identical S-box entries would cancel each other out because a value XORed with itself produces 0. (If i[1] = i[9], then SBox[i[1]] XOR ...stuff... XOR SBox[i[9]] would equal ...stuff...) Both i[1] and

Copyright (c) 1998, Springer-Verlag

i[9] would have had no effect on the encryption process. This would weaken the encryption function. If, however, the S-box is changed after every octet then even if i[1] = i[9], cancellation is very unlikely to occur (because SBoxes[1][i[1]] is almost certainly different from SBoxes[2][i[9]], even though i[1]=i[9]). A second beneficial effect is to insure that the encryption process is entirely different during the second octet than in the first octet. If the same S-box were used, then the second octet would compute the same function as the first octet -- which can be a serious weakness.

The parameter "enough" is used because encryption must continue for enough rounds to obscure and conceal the data. The exact number of rounds that is sufficient will no doubt be a matter of considerable debate -- it is left as a parameter precisely so that those who wish greater security can use more rounds, while those who are satisfied with fewer rounds can encrypt and decrypt data more rapidly. It seems very unlikely that fewer than 8 rounds (one octet) will ever be used, nor more than 64 rounds (8 octets). The author expects that almost all applications will use 16, 24, or 32 rounds. Values of "enough" that are not multiples of 8 are banned.

It is interesting to note that DES uses 16 rounds, and that it requires 5 rounds before each bit of input and key influences every bit of the block being encrypted[17]. That is, a change in a single bit of the input or of the key will not influence all 64 bits in the block being encrypted for 5 rounds. We might refer to this number as the "mixing interval," and say that DES has a mixing interval of 5 rounds. In Khufu, it requires 9 rounds before every bit of input and key influences every bit of the block being encrypted. It requires 8 rounds before every bit influences the selection of an S-box entry, and a 9th round for that change to influence the other 32-bit half of the 64-bit block being encrypted. The mixing interval in Khufu would therefore be 9 rounds. An interesting number is the total number of rounds divided by the mixing interval, which we will call the "safety factor." In DES, the safety factor is 16/5 = 3.2. In Khufu with 16 rounds, the safety factor is 16/9 = 1.8. It would seem that Khufu with 16 rounds suffers in this comparison, although we need to remember that the S-boxes in Khufu are secret, whereas the S-boxes in DES are public. Secret S-boxes are presumably more effective than publicly known S-boxes in concealing the data. If we increase the number of rounds in Khufu to 32, then the safety factor becomes 32/9 = 3.6,

Copyright (c) 1998, Springer-Verlag

which seems more likely to be satisfactory. While this metric seems useful, it should be viewed with caution: a large safety factor is no guarantee of security, nor is there any guarantee that 3.2 (the safety factor for DES) should be imbued with special significance. Given, however, that the task of selecting the number of rounds is difficult, it seems plausible to seek guidance by examining related systems.

Pre-Computing the S-Boxes

While 128 bits of key material is used at the start and finish of the encryption process (e.g., 64 bits at the start and 64 bits at the finish from the 128-bit array "auxiliaryKeys"), most of the key material is mixed in implicitly during the encryption process by selection of entries from the S-boxes. All the S-boxes (along with the 128 bits of auxiliary key material) are pre-computed from a (presumably short) user supplied key. The S-boxes *are* most of the key. This raises the question of how the S-boxes are computed and what properties they have. While the specific method of computing the S-boxes is complex, the essential idea is simple: generate the S-boxes in a pseudo-random fashion from a user supplied key so that they satisfy one property: all four of the one-byte columns in each S-box must be permutations. Intuitively, we require that selection of a different S-box entry change all four bytes produced by the S-box. More formally, (where "#" means "not equal to" and SBoxes[o][i][k] refers to the kth byte in the ith 32-bit entry of the SBox used during octet "o"): for all o, i, j, k; i # j implies SBoxes[o][i][k] # SBoxes[o][j][k].

We can divide the pre-computation of a pseudo-random S-box satisfying the desired properties into two parts: first, we generate a stream of good pseudo-random bytes; second, we use the stream of pseudo-random bytes to generate four pseudo-random permutations that map 8 bits to 8 bits. These four pseudo-random permutations are the generated S-box. We repeat this process and compute additional S-boxes until we have enough for the number of rounds of encryption that we anticipate.

We could generate a stream of pseudo-random bytes using an encryption function -- but we have no S-box to use in such an encryption function! To circumvent this circularity prob-

Copyright (c) 1998, Springer-Verlag

lem, we can assume the existence of a single "initial" S-box. Although we must get this initial S-box from somewhere, for the moment we assume it exists and satisfies the properties described earlier. We will discuss where it comes from later.

We (rather arbitrarily) adopt a 64-byte "state" value for our pseudo-random byte-stream generator. That is, the user-provided key is used to initialize a 64-byte block (which effectively limits the key size to 512 bits -- this does not seem to be a significant limit). This 64-byte block is then encrypted using Khufu (using the standard S-box for all octets, and setting the auxiliary keys to 0) in cipher block chaining mode. (Although use of a single S-box for all rounds will result in occasional cancelations as described earlier, this is acceptable for this particular application.) This provides 64 pseudo-random bytes. When these 64 bytes have been used, the 64-byte block is again encrypted, providing an additional 64 pseudo-random bytes. This process is repeated as long as more pseudo-random bytes are needed.

Now that we have a stream of pseudo-random bytes, we must convert them into the needed permutations. We adopt the algorithm given in Knuth Vol II. In this algorithm, we start with some pre-existing (not neccessarily random) permutation. For our purposes, we can start with the initial S-box. We then interchange each element in the initial permutation with some other randomly chosen element, thus producing a random permutation. In a pseudo programming language we have:

```
FOR octet = 1 TO enough/8 DO
  SBox = initialSBox;
  FOR column = 0 TO 3 DO
    BEGIN
      FOR i = 0 TO 255 DO
      BEGIN
        randomRow = RandomInRange[i,255];  -- returns a random number
                                           -- between i and 255, inclusive
        SwapBytes[  SBox[i,column], SBox[randomRow,column]  ];
```

Copyright (c) 1998, Springer-Verlag

```
        END;
      END;
    SBoxes[octet] = SBox;
END;
```

The routine "RandomInRange" uses the stream of random bytes to actually generate a number in the requested range.

Khafre

The design of Khafre is similar to the design of Khufu except that Khafre does not pre-compute its S-box. Instead, Khafre uses a set of standard S-boxes (discussed in the next section -- note that the standard S-boxes are different from the one initial S-box). The use of standard S-boxes means that Khafre can quickly encrypt a single 64-bit block without the lengthy pre-computation used in Khufu; however it also means that some new mechanism of mixing in key material must be adopted because the standard S-boxes can not serve as the key. The mechanism of key-mixing is simple -- key material is XORed with the 64-bit data block before the first round and thereafter following every 8 rounds. A consequence of this method is that the key must be a multiple of 64 bits -- it is expected that 64-bit and 128-bit key sizes will typically be used in commercial applications. Arbitrarily large key sizes can be used, though this will slow down encryption.

We can summarize Khafre as follows:

```
L, R: int32;
standardSBoxes: ARRAY [1 .. enough/8] OF ARRAY [0 .. 255] OF int32;
key:  ARRAY [0 .. keySize-1] OF ARRAY [0 .. 1] of int32;
keyIndex: [0 .. keySize-1];
rotateSchedule: ARRAY [1 .. 8] = [16,16,8,8,16,16,24,24];
```

Copyright (c) 1998, Springer-Verlag

```
L = L XOR key[0][0];
R = R XOR key[0][1];
keyIndex = 1 MOD keySize;
octet = 1;
FOR round = 1 TO enough DO
BEGIN
  L = L XOR standardSBoxes[octet] [R AND #FF];
  R = RotateRight[R, rotateSchedule[round mod 8 + 1] ];
  SWAP[L,R];
  IF round MOD 8 = 0 THEN
    BEGIN
      L = L XOR rotateRight[ key[keyIndex][0], octet];
      R = R XOR rotateRight[ key[keyIndex][1], octet];
      keyIndex = keyIndex + 1;
      IF keyIndex = keySize THEN keyIndex = 0;
      octet = octet+1;
    END;
END;
```

keySize is the number of 64-bit blocks of key material used for encryption.
rotateRight [a, b] rotates the 32-bit word "a" right by "b" bits.

We again require that the number of rounds be a multiple of 8 for efficiency reasons.

In order to decrypt correctly, we have to compute the correct value of "keyIndex" to use when decryption begins. For example, if we used a 128-bit key (keySize = 2) for 32 rounds to encrypt a 64-bit plaintext, then the final entry used in the key array would be key[1]. When we began to decrypt, we would have to begin with key[1] rather than key[0]. In general, we will have to start decryption from key[(enough/8 + 1) MOD keySize]. This com-

Copyright (c) 1998, Springer-Verlag

putation is extremely easy in the common case where keySize is 1, for any integer taken modulo 1 is 0. The other common case, in which keySize is 2, is also very easy to compute. Computing an integer modulo 2 requires only that we examine the bottom bit of the integer. While the modulo operation is more complex in some other cases, these cases are likely to be rare. If a particular case should prove to be frequent, simple special case code could be used to insure that computing the MOD function would not take excessive computer time.

Khafre will probably require more rounds than Khufu to achieve a similar level of security because it uses a fixed S-box. In addition, each Khafre round is somewhat more complex than each Khufu round. As a consequence of these two factors, Khafre will take longer than Khufu to encrypt each 64-bit block. In compensation for this slower encryption speed, Khafre does not require pre-computation of the S-box and so will encrypt small amounts of data more quickly than Khufu.

In Khafre used with a 64-bit key, the mixing interval is again 9 rounds. Here, because the S-boxes are public as in DES, it seems that the safety factor of 16/9 = 1.8 is more directly comparable with the safety factor of 16/5 = 3.2 for DES. Increasing the number of rounds from 16 to 24 or 32, yielding safety factors of 24/9 = 2.7 or 32/9 = 3.6, would seem more in keeping with the DES values. Further increases would be justified either because a safety factor larger than that of DES would be viewed as prudent, or because the "quality" of the mixing done by 9 rounds of Khafre might be viewed as less effective than 5 rounds of DES. Use of a key with more than 64 bits increases the mixing interval, and so would presumably require increases in the total number of rounds to yield commensurate increases in real security. Further study of these issues is warranted.

Making the Initial and Standard S-Boxes

We need an initial S-box to generate a pseudo-random stream of bytes. We also need a set of standard S-boxes to use in Khafre during the encryption process. In both applications, we need assurances about how the S-boxes were generated. This was a major question in DES -- whether any structure (intentional or accidental) might be present in the S-boxes

Copyright (c) 1998, Springer-Verlag

that would weaken the encryption function. Because the method of selecting the DES S-boxes was kept secret, the published articles on the structure of DES are necessarily incomplete. Published discussions of the structure in the DES S-boxes makes it clear that very strong selection criteria were used, and much of the structure actually found can reaonably be attributed to design principles intended to strengthen DES. The purpose behind some of the structure detected is unclear; though it does not appear to weaken DES it would be useful to know if the structure serves some purpose or whether it occured as an unintended consequence of the particular method chosen to actually generate the S-boxes.

To avoid these questions, the standard S-boxes will be generated from the initial S-box according to the standard (and public) algorithm for generating a set of S-boxes from a key. The key selected for the standard S-boxes will be the null (all 0) key. In this way, not only the standard S-boxes but also the algorithm for generating them are made public and can be examined to determine if there are any weaknesses.

The initial S-box must be generated from some stream of random numbers. In order to insure that the initial S-box does not have hidden or secret structure, we adopt the following rules:

1.) The program that generates the initial S-box from a stream of random numbers will be public.

2.) The stream of random numbers used as input to the program should be above reproach -- it should be selected in such a fashion that it could not reasonably have been tampered with in a fashion that might allow insertion of a trap-door or other weakness.

The first criteria can be met by making the code for generation of the S-boxes available along with the code for Khufu and Khafre. The second criteria is met by using the random numbers published in 1955 by the RAND corporation in "A Million Random Digits with 100,000 Normal Deviates" (available on magnetic tape for a nominal fee).

Copyright (c) 1998, Springer-Verlag

Methods of Cryptanalysis

Questions about the security of a new cryptographic algorithm are inevitable. Often, these questions are of the form "Have you considered the following attack..." It is therefore useful to describe the attacks that were considered during the design process. This serves two purposes. First, it reassures those who find their attack has already been considered (and presumably found non-threatening). Second, it tells those who are considering a new attack that the matter might not be settled and is worth pursuing further. A second question typically asked is "How many rounds are enough?" This will vary with three factors: the value of the data being encrypted, the encryption speed (delay) that is acceptable, and the estimated cost of cryptanalysis. This last cost is inferred by considering how many rounds are sufficient to thwart various certificational attacks.

Attacks can be broadly divided into a number of categories -- starting with chosen plaintext, known plaintext and ciphertext only. We shall primarily consider attacks of the chosen plaintext variety -- a system secure against chosen plaintext attacks is presumably also secure against the two weaker attacks. Some consideration will be given to known plaintext and ciphertext only attacks. Protection against casual browsers is valuable and can be provided more cheaply (i.e., with fewer rounds in the encryption process and hence less delay). An attack by a casual browser is better modeled by a ciphertext only attack. At the same time, the cryptographic resources the casual browser is likely to bring to bear are markedly inferior. Finally, the cost of encryption (in user inconvenience or delay) might be significant and the value of the data might not justify much inconvenience -- the choice might be between rapid encryption that offers protection against casual attack or no encryption at all.

Without further ado, and in no particular order, we discuss the major attacks considered during the design phase.

We first consider attacks against Khufu with a reduced number of rounds. We shall here consider attacks against an 8 round Khufu and will start with a chosen plaintext attack. We assume that the objective is to determine the entries in the S-box and the values of the aux-

Copyright (c) 1998, Springer-Verlag

iliary keys. While it might theoretically be possible to take advantage of the fact that the S-box was generated in a pseudo-random fashion from a smaller key (effectively limited to 64 bytes) this has so far not proven to be the case. The pseudo-random method of generating the S-box from the key is sufficiently complex that the 64-byte to 1024-byte expansion involved in this process appears quite strong. This is probably due to the relaxed computational time requirements on the pre-computation, i.e., the pre-computation is probably over-kill, but in most applications an additional fixed delay of some tens of milliseconds probably won't be noticed, so it wasn't further optimized.

An 8 round encryption can be readily broken under a chosen plaintext attack by noting that each round of the encryption process is affected by only a single byte of the initial plaintext. Therefore, given 8 rounds and 8 bytes of plaintext, some byte of plaintext is used last; e.g., in the 8th round. By encrypting two plaintext blocks that differ only in this last byte, we obtain two ciphertext blocks in which the encryption process differs only in the 8th round, and therefore in which the two left halves have the same difference as two S-box entries. That is, if the two ciphertext left halves are designated L[8] and L'[8] and if the indices of the S-box entries used in the 8th rounds are designated i[8] and i'[8], then L[8] XOR L'[8] equals SBox[i[8]] XOR SBox[i'[8]]. L[8] and L'[8] are known, as are i[8] and i'[8], so this provides an equation about two S-box entries. After we recover roughly 256 equations we will almost be able to solve for the 256 entries in the S-box. At this point, the recovery of the S-box will not quite be complete -- we can arbitrarily set the value of a single S-box entry and determine values for the rest of the entries that will satisfy the equations we have. Further equations will not help us, for if we have one solution to the equations, we can generate another solution by complementing the ith bit in every proposed S-box entry. The new set of values will also satisfy the equations, for every equation XOR's two S-box entries, and hence complementing the ith bit in both entries will leave the XOR of the two bits unchanged. We need another method for resolving this last ambiguity. This is conceptually easy (in the worst case, we could simply examine all 2^{32} possibilities) but an efficient algorithm is difficult to explain in a short space -- we therefore leave this as an exercise for the reader. Once the S-box entries are known, it is also relatively simple to determine the auxiliary keys.

Copyright (c) 1998, Springer-Verlag

If we consider a known plaintext attack against an 8 round encryption, we find the problem is more difficult. Certainly, we could request a large number of plaintext-ciphertext pairs and hope that at least some of the pairs differed only in the final few bytes (e.g., the bytes that are used only on the 7th and 8th rounds of encryption) but this would require many millions of such pairs. This, of course, presumes that the plaintext is selected randomly -- which implies that cipher block chaining (or some other pseudo-randomization method) is used to insure that patterns in the plaintext are eliminated prior to encryption. Direct encryption (without some "whitening" or pre-randomization) of sufficient text would probably result in 8-byte blocks that differed only in a single byte -- which might allow use of the method described above.

Finally, a ciphertext only attack against an 8-round Khufu appears to be a difficult problem. More sophisticated attacks can be mounted[22] that use various "hill-climbing" strategies. While we have not directly mounted such an attack, we would speculate that it would succeed for 8 rounds, though this is not certain.

Fundamentally, statistical or "hill-climbing" attacks must rely on statistically detectable differences between various alternatives. If the statistics are flat, then such techniques will fail. An important question with Khufu is the number of rounds required to achieve a statistically flat distribution. Preliminary results indicate that 16 rounds produces flat statistics.

The use of auxiliary keys were largely adopted for three reasons: first, it seemed intuitively reasonable that randomization of the input by XORing an unknown quanitity would assist in the encryption process. Second, four additional register-to-register XOR operations are cheap to implement. Finally, the auxiliary keys foil a specific chosen plaintext attack. This attack depends on the observation that, although the S-box has 256 entries, the encryption process does not use all entries for each plaintext-ciphertext pair. Even worse, although a typical 8-round encryption will use 8 different S-box entries it doesn't have to: some entries could be repeated. In the worst case, a single entry would be repeated 8 times -- which would effectively mean that only 32 bits of key material was used. If the auxiliary keys were not present then we could simply guess at the value of one of the S-box entries,

Copyright (c) 1998, Springer-Verlag

and then confirm our guess if we could find a plaintext-ciphertext pair that used only that entry for all 8 rounds. Because each of the 8 rounds uses a single byte from the plaintext, we could actually construct the plaintext needed to confirm our guess (if the auxiliary keys were not present). For example, if we guess that the 0^{th} S-box entry has some specific value, then we would select the first byte of our specially-built plaintext (or i[1], the byte of plaintext used as an index into the S-box in the first round) to be 0. Then, knowing what happens in the first round, we can select the second byte of the plaintext (or i[2]) so that the 0^{th} entry is again selected on the second round -- which would tell us what happens in the third round. By repeating this process for 8 rounds, we can construct a plaintext which, when enciphered, will tell us whether or not the 0^{th} S-box entry does or does not have a specific value. After trying 2^{32} values we will surely find the correct one. If we then repeat this whole process for the 1^{st} entry, and then the 2^{nd} entry, etc. we could determine the values of all the entries in the S-box.

The auxiliary keys prevent this attack because they effectively inject 64 bits of key material into the encryption process prior to selecting S-box entries. Thus, correctly guessing a 32-bit S-box entry is insufficient because we would also have to guess the 64-bit value XORed with the plaintext prior to encryption. If we guessed a single such bit incorrectly, then the incorrectly guessed bit would (within the first 8 rounds) cause selection of an uncontrolled S-box entry which would then initiate the uncontrolled avalanche of changes that we rely upon to provide cryptographic strength.

Although this attack is actually rather inefficient compared with our first chosen ciphertext attack, it does point out that there is no guarantee that multiple different S-box entries have actually been used during encryption. Instead, we must assure ourselves that the risk of this occuring is sufficiently low by explicitly computing the probability of its occurence.

Another attack in this general class is the cancelation attack. In this attack, we alter the first byte of the 8 bytes in the plaintext, and then attempt to cancel the effects of this alteration by altering the other 32-bit half in a compensating fashion. That is, by altering the first byte of plaintext used in the first round, we cause a change in the second round that we can understand. Because we can also change the other 32-bit half, this understandable change

Copyright (c) 1998, Springer-Verlag

in the second round can be cancelled. (Notice that the auxiliary keys have very little impact on this attack. We shall assume that the auxilliary keys are 0 for this analysis.). Now, if the first byte were 3, and we changed it to a 5, then this would produce a predictable change in the value XORed with the other 32-bit half, R, in the first round. This first round is computed as:

i[1] = L[0] AND #FF;
L[1] = R[0] XOR SBox[i[1]];

For the first plaintext we encrypted, this would become:

L[1] = R[0] XOR SBox[3];

while for the second plaintext encrypted, this would become:

L'[1] = R'[0] XOR SBox[5];

Therefore, if we select R'[0] = R[0] XOR SBox[3] XOR SBox[5], then the second equation becomes:

L'[1] = R[0] XOR SBox[3] XOR SBox[5] XOR SBox[5]

or

L'[1] = R[0] XOR SBox[3]

But this means that L'[1] = R[0] XOR SBox[3] = L[1]

In other words, L[1] and L'[1] are identical -- by knowing SBox[3] XOR SBox[5] we were able to cancel out the change that should have taken place in L[1]. This, of course, means that the avalanche of changes upon which encryption so critically depends has been thwarted at the very start. Notice that after the first round of encryption, the two blocks dif-

Copyright (c) 1998, Springer-Verlag

fer only in the first byte -- that is, the byte used in the first round. After 8 rounds of encryption, the resulting ciphertexts will also differ in only this one byte.

In practice, this attack seems to require that you first guess the correct value of SBox[i] XOR SBox[j] for two different values of i and j. This is a 32-bit value, and so on average it seems necessary to try 2^{32} different values before encountering the correct one. After 8 rounds of encryption, however, the fact that we have determined the correct 32-bit "cancelation value" will be obvious because the final 64 bits of ciphertext generated by the two different plaintexts will differ in only a single byte.

It might not at first be obvious, but we can in fact modify this attack so that only $2 * 2^{16}$ plaintext-ciphertext pairs are required in order to find the correct cancelation value. Although as described above, it would seem that we need 2^{32} pairs of plaintext-ciphertext pairs to test each possible 32-bit cancelation value, this is not the case. We can generate two lists of plaintext-ciphertext pairs, and then by selecting one plaintext-ciphertext pair from one list and the other plaintext-ciphertext pair from the other list, we can generate 2^{32} possible combinations of entries from the two lists. If we select the plaintexts used to generate the lists carefully, then every 32-bit cancelation value can be represented by one entry from the first list, and one entry from the second list.

When we consider this attack on a 16 round Khufu it is much weaker. If we can determine the correct 32-bit cancelation value it will cause collapse of the encryption process up until the changed byte is again used. If the first byte has been changed, then it will again be used on the 9^{th} round -- this means that in a 16-round Khufu a cancelation attack will effectively strip off 8 rounds. The remaining 8 rounds must then provide sufficient cryptographic strength to resist attack. Empirical statistical tests indicate that 8 rounds in which changes take place in the first one or two rounds will result in apparently random output -- though of course, this result demonstrates only that the output was random with respect to the specific statistical tests used, not that all possible statistical tests would reveal no pattern.

An attack proposed by Dan Greene is based on the observation that each 32-bit half is being

Copyright (c) 1998, Springer-Verlag

XORed with values selected (possibly with a rotation) from the S-box. Once the key has been chosen this S-box is fixed -- so at most 256 different values can be used and each value can be rotated (in the first 8 rounds of Khufu) in only four different ways. That is, we are at best applying a fixed and rather limited set of operations to each half. If we focus on the right half, R, (and if we neglect the effect of the auxiliary keys) then we find that:

R8 = R0 XOR ROTATE[SBox[i1],0] XOR ROTATE[SBox[i3],16] XOR
ROTATE[SBox[i5],24] XOR ROTATE[SBox[i7],8]]

R8 designates the right half following 8 rounds of Khufu, i.e., the right half of the ciphertext. R0 designates the right half before encryption begins, i.e., the right half of the plaintext. Although the indices used to select the S-box entries are computed during encryption, we are going to ignore their actual values. Instead, we will assume that i1, i3, i5 and i7 are selected randomly. This should not weaken the encryption function, so any cryptanalytic success we have using this assumption indicates weakness in the original system as well.

If we define

Y = Y[i1, i3, i5, i7] = ROTATE[SBox[i1],0] XOR ROTATE[SBox[i3],16] XOR ROTATE[SBox[i5],24] XOR ROTATE[SBox[i7],8]]

we can re-express the earlier equation as:

R8 XOR R0 = Y[i1, i3, i5, i7]

The left side of this equation is readily computed from a plaintext-ciphertext pair, and with enough such pairs we can compute the probability distribution of (R8 XOR R0). The right side should determine the same distribution (if we assume the actual indices are more or less random -- which should be a good approximation if the plaintext is random!). The 4 8-bit indices clearly could generate at most 2^{32} possible values for Y, but it seems more

Copyright (c) 1998, Springer-Verlag

plausible that some values for Y will be produced more than once while other values for Y will not be produced at all. That is to say, the distribution of Y's will not be uniform. If we can compute this distribution from enough plaintext-ciphertext pairs, and if it is non-uniform, could we then cryptanalyze an 8 round Khufu? Statistical evidence gathered on a 16-round Khufu suggests that this attack will fail for 16 rounds, but its success for 8 rounds is still unclear. Even given the distribution of Y's it is not clear (at the time of writing) how to determine the actual S-box entries.

Summary

An 8-round Khufu can be broken by several attacks, though it is somewhat resistant to ciphertext only attack. A 16-round Khufu has so far resisted the modest level of attack that has been mounted. Preliminary statistical analysis suggests that a 16-round Khufu produces random output. We are hopeful that a 16-round Khufu will be useful for general commercial encryption, but this conclusion is tentative. Increasing the number of rounds to 32 or more should be effective in increasing the complexity of cryptanalysis. Further study concerning the number of rounds required to prevent cryptanalysis is warranted.

The analysis of Khafre has been less detailed. It seems probable that Khafre will require more rounds of encryption to provide equivalent security than Khufu, because the S-boxes used with Khafre are public. Khufu, by contrast, generates different S-boxes for each key and keeps the S-boxes secret -- and so uses more key material per round than Khafre.

Any reader seriously considering use of these encryption functions is advised that (1) waiting for one to two years following their publication should allow sufficient time for their examination by the public cryptographic community and (2) current information about their status should be obtained by contacting the author.

Copyright (c) 1998, Springer-Verlag

Acknowledgements

The author would like to particularly thank Dan Greene for his many comments and the many hours of discussion about cryptography in general and the various design proposals for Khufu in particular. Thanks are also due to Luis Rodriguez, who implemented the C version of Khufu and gathered most of the statistics. Thanks are also due the many researchers at PARC who provided insight, technical comments, and encouragement. I would also like to thank Dan Swinehart, John White, Mark Weiser, Frank Squires, John Seely Brown, Ron Rider, and the rest of PARC management for their persistent support of this work.

Bibliography

1.) "Secrecy, Authentication, and Public Key Systems", Stanford Ph.D. thesis, 1979, by Ralph C. Merkle.

2.) "A Certified Digital Signature", Crypto '89.

3.) Moti Yung, private communication.

4.) "A High Speed Manipulation Detection Code", by Robert R. Jueneman, Advances in Cryptology - CRYPTO '86, Springer Verlag, Lecture Notes on Computer Science, Vol. 263, page 327 to 346.

5.) "Another Birthday Attack" by Don Coppersmith, Advances in Cryptology - CRYPTO '85, Springer Verlag, Lecture Notes on Computer Science, Vol. 218, pages 14 to 17.

6.) "A digital signature based on a conventional encryption function", by Ralph C. Merkle, Advances in Cryptology CRYPTO 87, Springer Verlag, Lecture Notes on Computer Science, Vol. 293, page 369-378.

7.) "Cryptography and Data Security", by Dorothy E. R. Denning, Addison-Wesley 1982, page 170.

8.) "On the security of multiple encryption", by Ralph C. Merkle, CACM Vol. 24 No. 7, July 1981 pages 465 to 467.

9.) "Results of an initial attempt to cryptanalyze the NBS Data Encryption Standard", by Martin Hellman et. al., Information Systems lab. report SEL 76-042, Stanford Univer-

Copyright (c) 1998, Springer-Verlag

sity 1976.

10.) "Communication Theory of Secrecy Systems", by C. E. Shannon, Bell Sys. Tech. Jour. 28 (Oct. 1949) 656-715

11.) "Message Authentication" by R. R. Jueneman, S. M. Matyas, C. H. Meyer, IEEE Communications Magazine, Vol. 23, No. 9, September 1985 pages 29-40.

12.) "Generating strong one-way functions with cryptographic algorithm", by S. M. Matyas, C. H. Meyer, and J. Oseas, IBM Technical Disclosure Bulletin, Vol. 27, No. 10A, March 1985 pages 5658-5659

13.) "Analysis of Jueneman's MDC Scheme", by Don Coppersmith, preliminary version June 9, 1988. Analysis of the system presented in [4].

14.) "The Data Encryption Standard: Past and Future" by M.E. Smid and D.K. Branstad, Proc. of the IEEE, Vol 76 No. 5 pp 550-559, May 1988

15.) "Defending Secrets, Sharing Data: New Locks and Keys for Electronic Information", U.S. Congress, Office of Technology Assessment, OTA-CIT-310, U.S. Government Printing Office, October 1987

16.) "Exhaustive cryptanalysis of the NBS data encryption standard", by Whitfield Diffie and Martin Hellman, Computer, June 1977, pages 74-78

17.) "Cryptography: a new dimension in data security", by Carl H. Meyer and Stephen M. Matyas , Wiley 1982.

18.) "One Way Hash Functions and DES", by Ralph C. Merkle, Crypto '89.

19.) "Data Encryption Standard (DES)", National Bureau of Standards (U.S.), Federal Information Processing Standards Publication 46, National Technical Information Service, Springfield, VA, Apr. 1977

21.) "Cryptography and Computer Privacy", by H. Feistel, Sci. Amer. Vol. 228, No. 5 pp 15-23, May 1973

22.) "Maximum Likelihood Estimation Applied to Cryptanalysis", by Dov Andelman, Stanford Ph.D. Thesis, 1979

23.) IBM has recently proposed a specific one-way hash function which has so far resisted attack.

24.) "A Fast Software One-Way Hash Function," submitted to the Journal of Cryptology. The C source for this method is available by anonymous FTP from arisia.xerox.com (13.1.100.206) in directory /pub/hash.

Copyright (c) 1998, Springer-Verlag