CHAPTER SIX

Generation, Distribution, and Installation of Cryptographic Keys[1]

Key generation is the process of producing the cryptographic keys required by a cryptographic system. *Key distribution* is the process of transporting or routing cryptographic keys through a cryptographic system for subsequent installation. *Key installation* is the process of entering cryptographic keys into cryptographic devices.

The procedures presented here for the generation, distribution, and installation of cryptographic keys are founded upon the key management scheme developed in Chapters 4 and 5. This scheme for key management distinguishes between key-encrypting keys and data-encrypting keys. The former are used to encipher other keys and are derived as part of the process of initializing the cryptographic system. They remain constant for relatively long periods—changed perhaps once a year. Data-encrypting keys, on the other hand, are generated dynamically during regular system operation. They remain in existence for the life of the data they protect. That period, for communication security, is the length of time the user is signed-on to the system—usually a matter of minutes. For file security, where data are stored in enciphered form, the data-encrypting keys may exist for relatively long periods of time.

One special key-encrypting key, the host master key, should be generated by some random process such as tossing a coin or throwing a die. All other key-encrypting keys are produced by using DES as a generator of pseudo-random numbers. The procedure can be performed under secure conditions on the computer. Data-encrypting keys are generated dynamically at the host processor by exploiting the randomness associated with the many different users and processes that normally are active on a system at any given time.

With DES, each 64-bit cryptographic key consists of 56 independent key bits and 8 bits (the last bit of each 8-bit byte) that may be used for error detection. If used, these bits assure that each byte in the key has odd parity (see Key Parity, Chapter 4).

[1] Reprinted in part from *IBM Systems Journal* **17**, No. 2, 1978 [1].

Since DES is a publicly known algorithm, cryptographic strength must be based on the secrecy of its cryptographic keys. Even though there are 2^{56} different possible keys, keys should be randomly selected so that an organized search for them would not be likely to meet with early success. If there were a known bias in the selection of keys, an opponent could try the more likely candidates first.

GENERATION OF THE HOST MASTER KEY

Regardless of the procedure used for key generation, organized predictable methods must be avoided. Any procedure based on one's telephone number, name and address, date of birth, or the like, is so frail that no real protection is provided. Also, the programs for generating pseudo-random numbers, which are available on many computer systems, are far too predictable to be used for this purpose and should be avoided.

Since the host master key, either directly or through one of its derived variants, provides protection (through encipherment) for all other keys stored in the system, and since the host master key will in all probability remain unchanged for long periods, great care must be taken to select this key in a random manner. The method recommended here is for the key to be created via a process performed by the user of the system.

Assume that a 64-bit parity-adjusted key is required for the selection process, and that odd parity is used (i.e., every eighth bit is adjusted so that the number of bits in the 8-bit group is odd). Since a change to the key is likely to cause its parity to become incorrect, parity can be used for error detection. For example, parity can be checked to ensure that a master key is properly specified for entry into a cryptographic facility. Or, during regular operation, parity can be checked to ensure that bits in the master key are not inadvertently changed (e.g., because of a malfunction in the cryptographic facility).

Tossing Coins

Let the bit values 0 and 1 in the cryptographic key be determined by the occurrence of heads and tails, respectively. Then toss 56 coins in eight groups of seven coins each, and record the results. Each group is then converted to its corresponding parity-adjusted hexadecimal digits (Table 6-1).

Each group of 7 bits in the 56-bit key is expanded to 8 bits by appending an additional parity bit (odd parity is maintained). This process can be performed with the aid of a table, if desired. In Table 6-2, for example, the first 4 bits index the table row and the last 3 bits the table column. Since every entry in the table has correct parity, a parity-adjusted key will be formed even if there should be an error in indexing. The cryptographic key (the value entered into the system and saved in a secure repository for back up purposes) is defined as that string of hexadecimal digits produced by the table reference process. The values used to index the table are then destroyed.

Trial	Result	Binary	Parity	Hex
(1)	HHHT HTH	0001 010	1	15
(2)	THTH HHT	1010 001	0	A2
(3)	TTHT HHH	1101 000	0	D0
(4)	THHH TTT	1000 111	1	8F
(5)	HHTT TTH	0011 110	1	3D
(6)	TTHH HHH	1100 000	1	C1
(7)	HHHT THT	0001 101	0	1A
(8)	HHTH THH	0010 100	1	29

Parity-Adjusted Key = Hex 15A2D08F3DC11A29

Table 6-1. Results of Coin-Tossing Converted to Binary and Hexadecimal Digits (heads (H) = binary 0, tails (T) = binary 1)

Throwing Dice

The method described above can also be used with dice. Instead of tossing seven coins, the user rolls seven dice. The binary digits can be obtained by an even roll (2, 4, or 6) to represent a 0 bit and an odd roll (1, 3, or 5) to represent a 1 bit.

	decimal	0	1	2	3	4	5	6	7
	binary	000	001	010	011	100	101	110	111
0	0000	01	02	04	07	08	0B	0D	0E
1	0001	10	13	15	16	19	1A	1C	1F
2	0010	20	23	25	26	29	2A	2C	2F
3	0011	31	32	34	37	38	3B	3D	3E
4	0100	40	43	45	46	49	4A	4C	4F
5	0101	51	52	54	57	58	5B	5D	5E
6	0110	61	62	64	67	68	6B	6D	6E
7	0111	70	73	75	76	79	7A	7C	7F
8	1000	80	83	85	86	89	8A	8C	8F
9	1001	91	92	94	97	98	9B	9D	9E
10	1010	A1	A2	A4	A7	A8	AB	AD	AE
11	1011	B0	B3	B5	B6	B9	BA	BC	BF
12	1100	C1	C2	C4	C7	C8	CB	CD	CE
13	1101	D0	D3	D5	D6	D9	DA	DC	DF
14	1110	E0	E3	E5	E6	E9	EA	EC	EF
15	1111	F1	F2	F4	F7	F8	FB	FD	FE

Table 6-2. Parity-Adjusted Hexadecimal Digits (Odd Parity)

Random Number Table

A random number table contains a list of numbers that have been generated by a random (or nearly random) process. Once recorded, the numbers are subjected to extensive statistical tests to uncover any nonrandomness. There is one book [2], for example, that lists 1 million random digits arranged in 20,000 rows and 50 columns. The table occupies 400 pages within the text. The basic problem with random number tables is that the opponent may be able to guess which table has been used. For all practical purposes, once this happens a number obtained from the table is no better than the randomly selected starting position.

For example, if a key is selected from the table by using one random starting point, and the table is read from left to right, then there are only 1 million different keys that may possibly be selected. In this instance, the key space is reduced from 72,057,594,037,927,936 (2^{56}) possible different keys to 1,000,000 (approximately 2^{20}) possible different keys. In addition, the tendency of books to open repeatedly to the same page or set of pages, and the tendency to choose numbers near the center of the page, reduce the possible candidates still further.

To overcome these problems, one can choose several random starting points. This would introduce an element of randomness into the routine for generating key bits based on the starting point. But this requires a random process (like coin-tossing) which could instead be used to generate the key directly. Hence the use of random number tables in key generation does not obviate a random process. The use of random number tables complicates, but does not strengthen the key generation process. In fact, the improper use of such tables could actually weaken the procedure. For this reason, random number tables are not recommended for generating keys.

GENERATION OF KEY-ENCRYPTING KEYS

When large numbers of key-encrypting keys are deployed throughout a cryptographic system, there is an increased chance that one or more of these keys will become known to an opponent. Therefore, the procedure for generating keys must be such that if one or more of the keys are discovered, the work factor will remain high enough to protect the remaining keys; that is, a knowledge of part of the keys will not provide a shortcut method to find any of the other keys.

It is recommended that the procedure for generating keys involve the host master key or one of its variants by executing one or more of the cryptographic key management operations. Not only will an opponent be forced to carry out part of his attack on the same host system, but because the operations themselves must be executed as part of the attack, the opponent is constrained by the particular operational characteristics of the host machine itself. For example, since the time it takes to encipher and decipher is known for a specific system, the minimum required computation time can be determined for a given attack.

A Weak Key-Generating Procedure

Suppose that RN is a 64-bit random number supplied as input to the key-generating procedure, and that keys are generated using an *encipher under master key* operation as follows:

Ki = the ith cryptographic key (i = 1, 2, . . . , n) obtained by adjusting each byte in Yi for odd parity

where

$$Y0 = RN$$

$$Yi = E_{KM0}(Yi - 1) \quad \text{for } i = 1, 2, \ldots, n$$

This procedure is too frail, however, since a compromise of only a single key, say Ki, would with little uncertainty allow an opponent to deduce Yi. By exercising the *encipher under master key* operation, the opponent could then generate the remaining keys, Ki + 1, Ki + 2, . . . , Kn.

A Strong Key-Generating Procedure

The procedure recommended here is to use DES as a generator of pseudo-random numbers and produce the entire set of keys with three 64-bit random values: RN1, RN2, and RN3. Let RN1 and RN2 be generated externally by a human using a random process (such as coin-tossing or rolling dice) similar to that used for generating the host master key, and let RN3 be derived internally within the host system. To defeat the key generating procedure, an opponent must compromise three secret parameters from at least two independent sources. (Three independent sources would be involved if RN1 and RN2 were generated and entered into the system by two different people.)

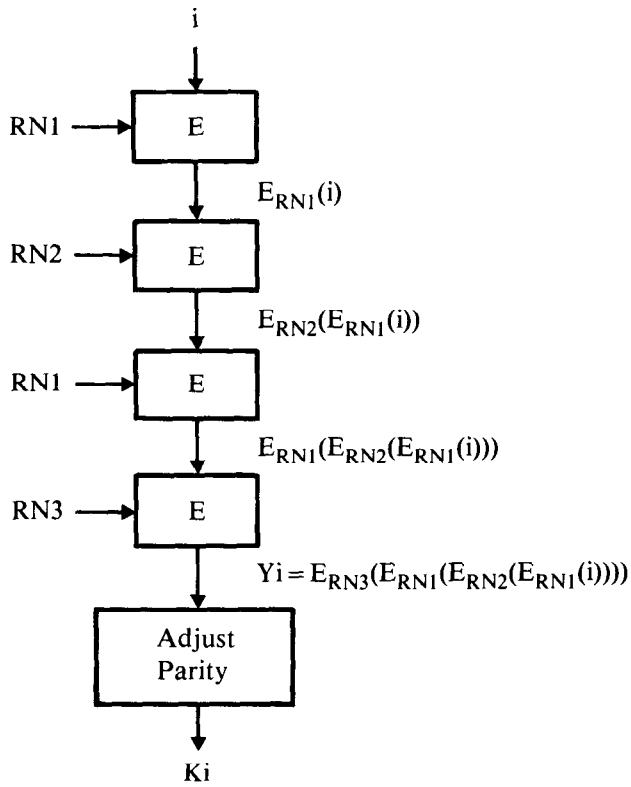
A straightforward approach for using RN1, RN2, and RN3 in the key-generating procedure is illustrated in Figure 6-1. (Multiple encryption is also discussed in Appendix D.) The procedure is described as follows:

Ki = the ith cryptographic key (i = 1, 2, . . . , n) obtained by adjusting each byte in Yi for odd parity

and

$$Yi = E_{RN3}(E_{RN1}(E_{RN2}(E_{RN1}(i)))) \quad \text{for } i = 1, 2, \ldots, n \qquad (6\text{-}1)$$

For an opponent to compromise a single key, say Ki, the values of RN1, RN2, and RN3 must be known. Thus even if a set of keys, Ki1, Ki2, . . . , Kij would become known, it would still be impossible to deduce the remaining unknown keys.



RN1, RN2 = random numbers supplied by the user.
RN3 = a random number generated by the host processor.
i = index number.
E = encipherment.
D = decipherment.

Figure 6-1. Basic Approach for Generating Keys Using the DES

The key-generating procedure illustrated in Figure 6-1 can be implemented within the host cryptographic system by exercising a *reencipher from master key* operation, as shown below:

$$\begin{aligned} &\text{RFMK: } \{\text{RN1, i}\} \longrightarrow \text{Ai} \\ &\text{RFMK: } \{\text{RN2, Ai}\} \longrightarrow \text{Bi} \\ &\text{RFMK: } \{\text{RN1, Bi}\} \longrightarrow \text{Ci} \\ &\text{RFMK: } \{\text{RN3, Ci}\} \longrightarrow \text{Yi} \\ &\text{Yi:} \xrightarrow{\text{adjust parity}} \text{Ki} \end{aligned} \qquad (6\text{-}2)$$

where

$$\begin{aligned} i &= \text{index number} \\ R1 &= D_{KM1}(RN1) \\ R2 &= D_{KM1}(RN2) \\ R3 &= D_{KM1}(RN3) \\ Yi &= E_{R3}(D_{KM0}(E_{R1}(D_{KM0}(E_{R2}(D_{KM0}(E_{R1}(D_{KM0}(i)))))))) \end{aligned}$$

and Ai, Bi, and Ci are 64-bit intermediate results.

Assuming that DES does not introduce any bias, then (for i = 1, 2, and 3) Ri is random if RNi is random. Thus the expressions for Yi given in Equations 6-1 and 6-2 are comparable, except for the extra deciphering operations under KM0 in Equation 6-2.

RN3 is generated from several independent readings of a *time-of-day* (TOD) clock. The idea is to issue n different input/output operations of indeterminate length so that the clock values, read at the completion of the operations, denoted

$$TOD1, TOD2, \ldots, TODn$$

are unpredictable.

One way to obtain an input/output operation of indeterminate length is to send a message to a terminal's user requesting that one or more terminal keys be struck in response to the issued message. Since the response time is different from user to user, the clock reading obtained at the completion of the event is unpredictable. By repeating the process, the required number of clock readings can be obtained.

If the TOD clock is a 64-bit counter, then a straightforward approach for generating RN3 is to exercise repeatedly the *reencipher from master key* operation:

$$\begin{aligned} &RFMK: \{TOD1, X0\} \longrightarrow X1 \\ &RFMK: \{TOD2, X1\} \longrightarrow X2 \\ &\qquad\vdots \\ &RFMK: \{TODn, Xn-1\} \longrightarrow RN3 \end{aligned}$$

where:

$$\begin{aligned} X0 &= 0 \\ Xi &= E_{D_{KM1}(TODi)}(D_{KM0}(Xi-1)) \qquad \text{for } i = 1, 2, \ldots, n \\ RN3 &= E_{D_{KM1}(TODn)}(D_{KM0}(\ldots(E_{D_{KM1}(TOD1)}(D_{KM0}(0)))\ldots)) \end{aligned}$$

The uncertainty in RN3 depends on the uncertainty in each clock value used in its computation. If, for example, each clock value has two possible out-

comes of equal likelihood, then there would be 2^n possible combinations for the n-tuple (TOD1, TOD2, . . . , TODn). A value of n equal to 64 would be more than enough to ensure that RN3 is random.

Since, in an actual implementation, the number of unpredictable outcomes for each clock value is much larger than 2, fewer clock readings are needed. For example, if the time a human takes to respond varies as much as one second, and if the responses are spread uniformly over the interval, then a clock with resolution to one microsecond has 2^{20} unpredictable outcomes. Three independent clock values may then be sufficient to generate a random value for RN3.

Where it is undesirable to involve a human in the creation of a random value for RN3, a single clock reading, taken at the time the key-generating procedure is invoked, can be used. However, a single clock value does not have enough different unpredictable combinations to allow RN3 to be used as a cryptographic key, since an opponent may be able to guess the approximate time when the keys were generated. But it does reduce the chance that a duplicate set of keys is accidentally regenerated, should it happen that RN1 and RN2 are inadvertently reentered during a subsequent execution of the key-generating procedure.

An Alternate Approach for Generating Key-Encrypting Keys

An alternate approach also uses DES as a generator of pseudo-random numbers. The basic idea is that a 64-bit random number, RN, can be used in conjunction with the DES algorithm to produce the entire set of key-encrypting keys (except the host master key). RN, in this case, is generated externally by a random process similar to that used in generating the host master key (e.g., coin-tossing or dice-throwing). Yi is the ith pseudo-random number generated in the process, and Ki, which is obtained from Yi by adjusting each byte for odd parity, equals the ith cryptographic key (i = 1, 2, . . . , n).

The approach described here is to use one of the host processor's cryptographic operations so that each value of Yi (i = 1, 2, . . . , n) is a function of the host master key as well as a function of RN. This approach makes use of the *reencipher from master key* (RFMK) operation, as shown below:

$$\text{RFMK: } \{\text{RN, TOD} + \text{i}\} \longrightarrow \text{Ai}$$

$$\text{RFMK: } \{\text{RN, Ai}\} \longrightarrow \text{Yi}$$

$$\text{Yi:} \xrightarrow{\text{adjust parity}} \text{Ki}$$

where

$$R = D_{KM1}(RN)$$

$$Yi = E_R D_{KM0}(E_R(D_{KM0}(TOD + i))))$$

and Ai is a 64-bit intermediate result. Again, assuming that DES does not introduce any bias, the quantity $R = D_{KM1}(RN)$ is random if RN is random.

Therefore, Yi is a function of two secret, independently selected cryptographic keys: one (RN) supplied by the user, the other (KM0) supplied by the system.

Because of the DES algorithm's property of noninvertibility and because of the manner in which the parameters (RN and KM0) are used to compute Yi, a knowledge of several clear keys (Ki1, Ki2, . . . , Kij) or, in fact, even a knowledge of the corresponding Y-values (Yi1, Yi2, . . . , Yij) will not permit RN or KM0 to be deduced. Therefore, a knowledge of one or more of the generated keys will not allow any of the remaining keys to be deduced.

Note that the procedure described above does *not* depend on the randomness provided by the TOD clock. A clock value is introduced in this case to reduce the likelihood that the user will inadvertently regenerate a duplicate list of keys by reentering the same value of RN. A duplicate RN could be reentered, for example, if the medium on which a new value of RN is recorded was accidentally replaced with one containing an old value of RN.

Encipherment of Keys under the Master Key's Variants

Enciphering under the variants of the host master key could be done with operations similar to the *encipher under master key* (EMK) operation. If EMK1 and EMK2 designate encipherment under KM1 and KM2, respectively, then these operations can be described as follows:

$$\text{EMK1: } \{X\} \longrightarrow E_{KM1}(X)$$

$$\text{EMK2: } \{X\} \longrightarrow E_{KM2}(X)$$

In this approach, use of the EMK1 and EMK2 operations must be carefully controlled (see Additional Considerations, Chapter 5). For example, if an opponent could encipher a known value X under KM1, then knowledge of X and $E_{KM1}(X)$ would permit the opponent to transform $E_{KM0}(Y)$ to $E_X(Y)$ and thus allow the decipherment of any arbitrary quantity (Y) under the host master key (KM0). This would enable recovery of session keys and file keys in clear form. To protect against this threat and thus provide control over enciphering under the host master key's variants, the EMK1 and EMK2 operations could be designed so that they are activated (made operational) only through the use of a physical key-operated switch.

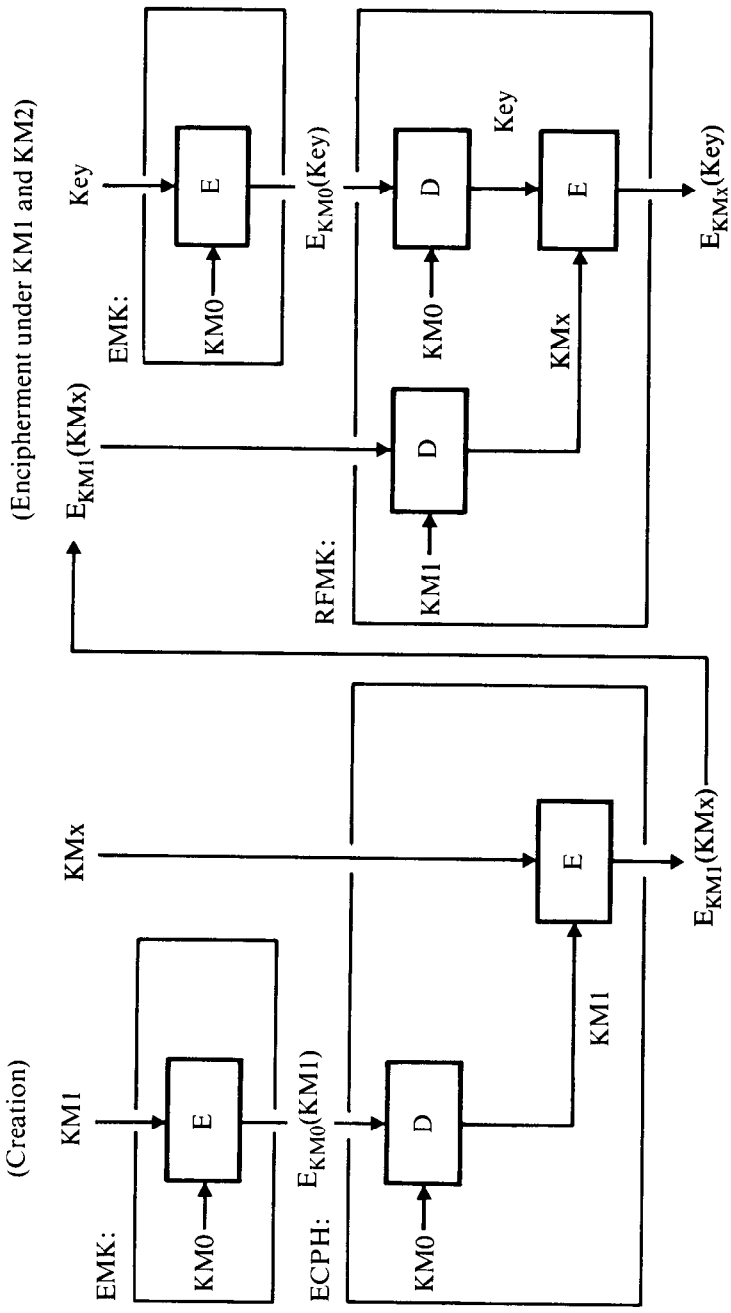
In an alternate approach, encipherment under the variants KM1 and KM2, could be accomplished by using the quantities $E_{KM1}(KM1)$ and $E_{KM1}(KM2)$, respectively. These quantities are called *system activation keys*, and encipherment of an arbitrary value X is accomplished by using a *reencipher from master key* operation, as shown below:

$$\text{EMK: } \{X\} \longrightarrow E_{KM0}(X)$$

$$\text{RFMK: } \{E_{KM1}(KM1), E_{KM0}(X)\} \longrightarrow E_{KM1}(X)$$

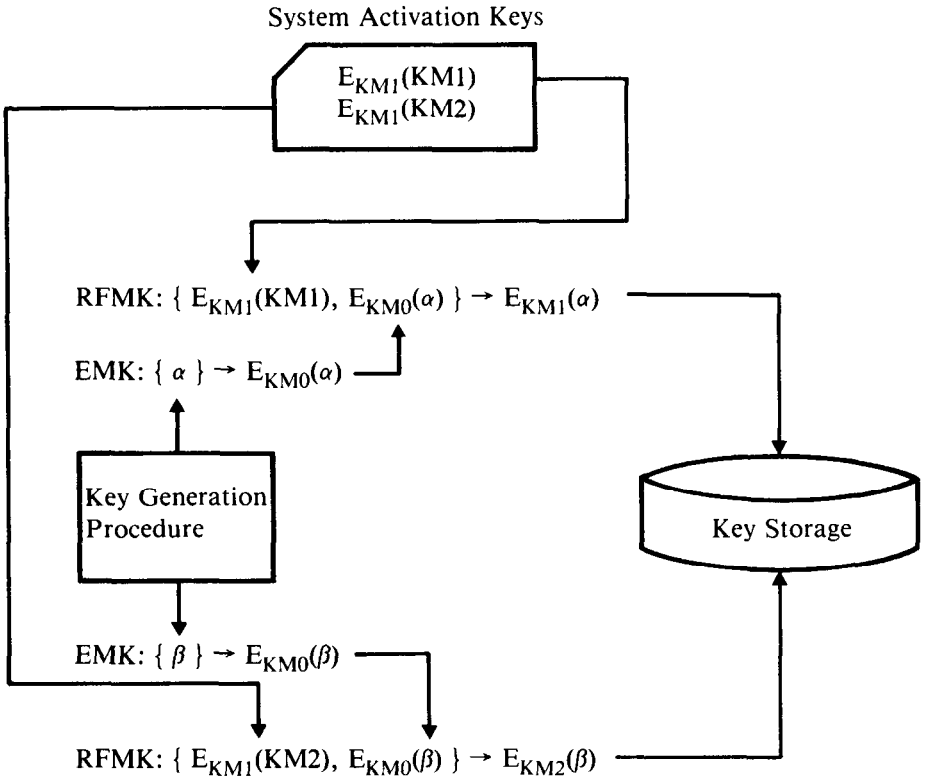
$$\text{RFMK: } \{E_{KM1}(KM2), E_{KM0}(X)\} \longrightarrow E_{KM2}(X)$$

How the system activation keys can be created and used for enciphering under the host master key's variants, KM1 and KM2, is further illustrated in the block diagram in Figure 6-2.



KMx is a variant of the master key (x = 1 or 2).

Figure 6-2. Creation of the System Activation Keys, $E_{KM1}(KM1)$ and $E_{KM1}(KM2)$, and Encipherment Under KM1 and KM2



α = key to be enciphered under 1st variant, KM1.
β = key to be enciphered under 2nd variant, KM2.

Figure 6-3. Encipherment of Cryptographic Keys for Local Storage at a Host Processor

Once created, the system activation keys may be saved and later reentered as input parameters to the key-generating procedure (Figure 6-3).

To prevent unauthorized enciphering under KM1 and KM2, the system activation keys are maintained as secret parameters of the cryptographic system. Since deciphering under KM1 and KM2 is not possible with the system activation keys, the design adheres to the strategic principle of providing the cryptographic system with only the minimum functional capability needed to generate and manage its keys.

Alternatively, the quantities $E_{KM0}(KM1)$ and $E_{KM0}(KM2)$ could be used with an ECPH operation to encipher under the respective variants, KM1 and KM2. However, these quantities could also be used with a DCPH operation to decipher under KM1 and KM2. Because of this, it would be less desirable to use $E_{KM0}(KM1)$ and $E_{KM0}(KM2)$—rather than $E_{KM1}(KM1)$ and $E_{KM1}(KM2)$—as system activation keys.

Transforming Cryptographic Keys

It may happen that the host master key is changed within the cryptographic system without the other keys in the system also being changed. Thus a procedure is needed that permits keys stored under the encipherment of variants of the old host master key to be reenciphered under variants of the new host master key.

Let KM0* and KM0 represent the old and new host master keys, respectively. The required function is obtained as follows. KM0* and KM0 are read into the main memory of the host system where the variants KM1*, KM2*, KM1, and KM2 are derived by inverting appropriate bits in KM0* and KM0, respectively. The new host master key, KM0, is then written into the host's cryptographic facility using a set master key operation.

An *encipher under master key* operation is then used to generate the following quantities:

$$\text{EMK: } \{KM1\} \longrightarrow E_{KM0}(KM1)$$
$$\text{EMK: } \{KM2\} \longrightarrow E_{KM0}(KM2)$$

These quantities are used with an *encipher data* (ECPH) operation to generate the following additional quantities:

$$\text{ECPH: } \{E_{KM0}(KM1), KM1\} \longrightarrow E_{KM1}(KM1)$$
$$\text{ECPH: } \{E_{KM0}(KM1), KM2\} \longrightarrow E_{KM1}(KM2)$$
$$\text{ECPH: } \{E_{KM0}(KM2), KM1^*\} \longrightarrow E_{KM2}(KM1^*)$$
$$\text{ECPH: } \{E_{KM0}(KM2), KM2^*\} \longrightarrow E_{KM2}(KM2^*)$$

If α represents a secondary key stored enciphered under the first variant of the host master key, then reencipherment is accomplished as follows:

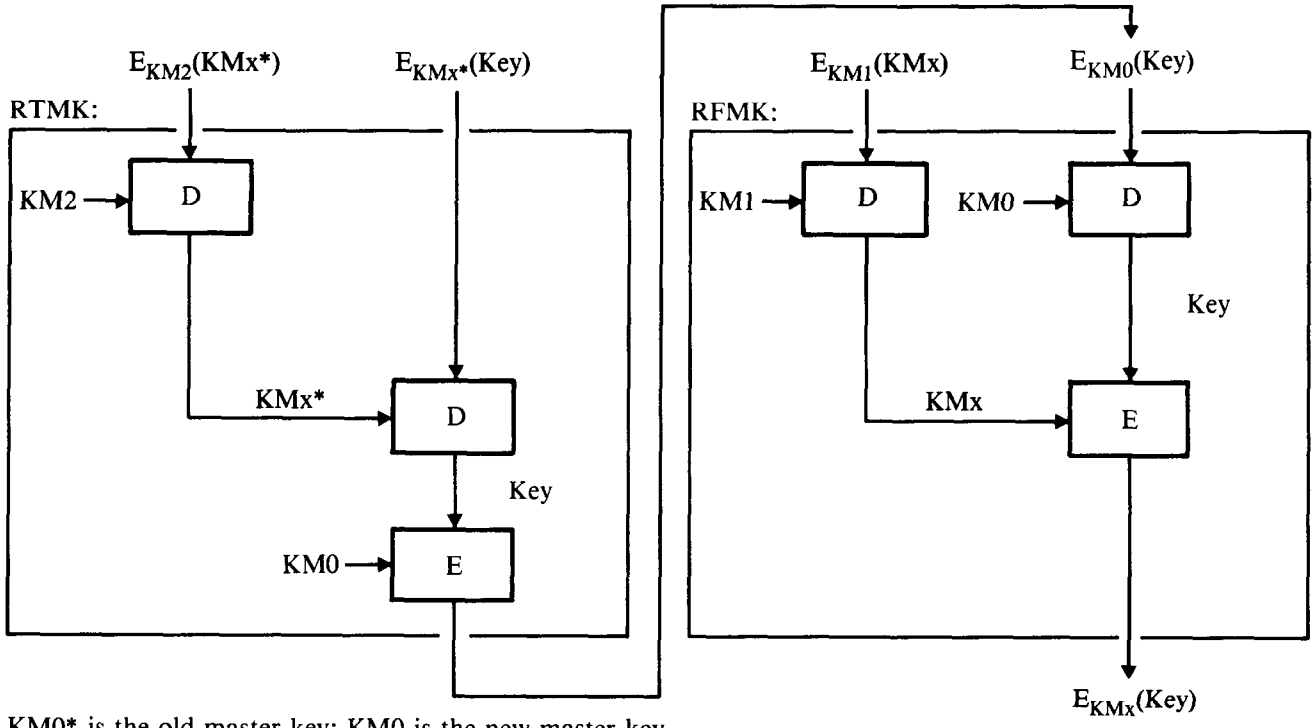
$$\text{RTMK: } \{E_{KM2}(KM1^*), E_{KM1^*}(\alpha)\} \longrightarrow E_{KM0}(\alpha)$$
$$\text{RFMK: } \{E_{KM1}(KM1), E_{KM0}(\alpha)\} \longrightarrow E_{KM1}(\alpha)$$

If β represents a secondary key stored enciphered under the second variant of the host master key, then reencipherment is accomplished as follows:

$$\text{RTMK: } \{E_{KM2}(KM2^*), E_{KM2^*}(\beta)\} \longrightarrow E_{KM0}(\beta)$$
$$\text{RFMK: } \{E_{KM1}(KM2), E_{KM0}(\beta)\} \longrightarrow E_{KM2}(\beta)$$

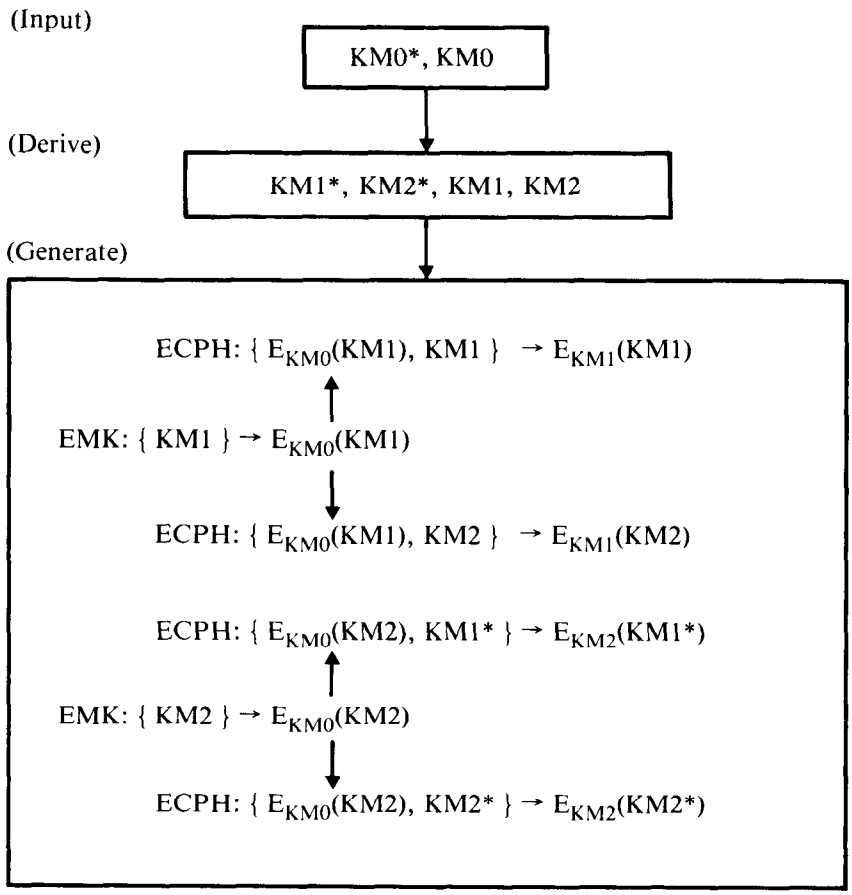
Reencipherment using the RTMK and RFMK operations is further illustrated in the block diagram in Figure 6-4. The general procedure for reencipherment of keys in the cryptographic key data set (CKDS) is illustrated in Figure 6-5.

To ensure that the procedure for transforming cryptographic keys is performed properly, one should verify that the old host master key read into



KM0* is the old master key; KM0 is the new master key.
$E_{KM2}(KMx^*)$ and $E_{KM1}(KMx)$ are assumed to be pregenerated.
x = 1 or 2.

Figure 6-4. Reencipherment of Cryptographic Keys from an Old to a New Master Key



KM0*, KM1*, KM2* are old master key and variants.
KM0, KM1, KM2 are new master key and variants.

Figure 6-5. General Procedure for Reencipherment of Keys

the main memory of the host system is equal to the actual host master key stored in the cryptographic facility. This can be accomplished by using the following procedure. Let

KM0* = the old host master key previously written into the host's cryptographic facility

KM0′ = the copy of the old host master key read into main memory of the host system for verification

The following operations are now performed:

$$\text{EMK: } \{KM0'\} \longrightarrow E_{KM0^*}(KM0')$$

$$\text{DCPH: } \{E_{KM0^*}(KM0'), E_{KM0^*}(KM0')\} \longrightarrow q$$

where

$$q = D_{KM0'}(E_{KM0^*}(KM0'))$$

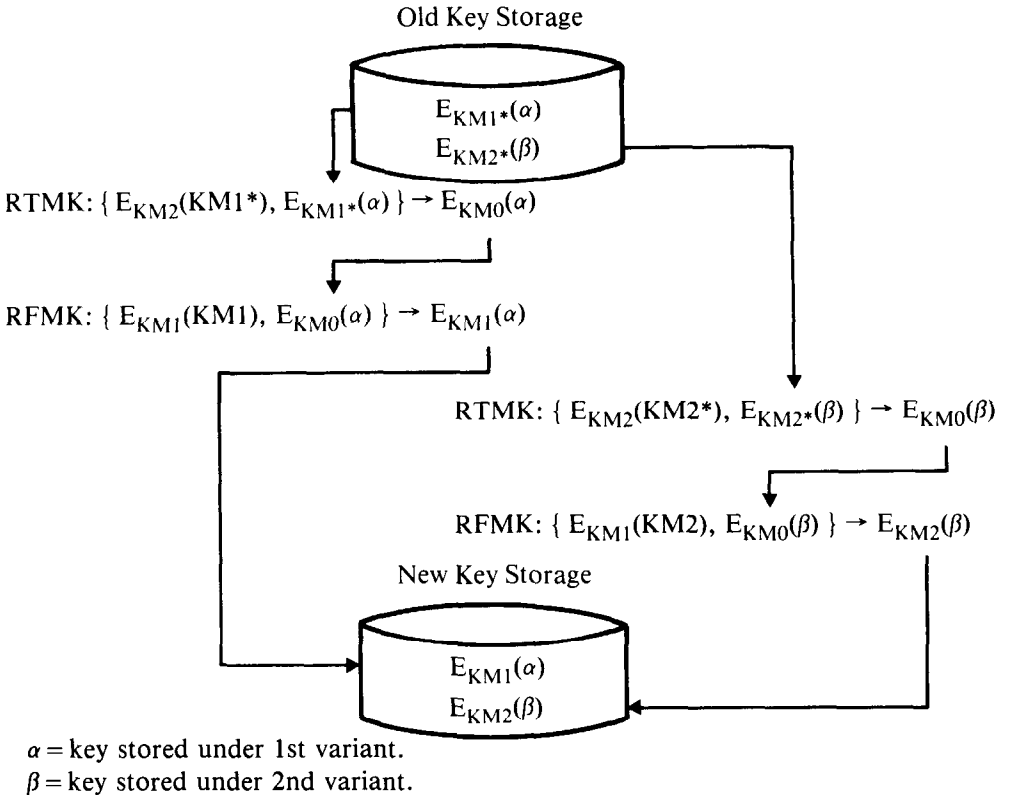


Figure 6-5 (cont'd). General Procedure for Reencipherment of Keys

If KM0′ equals KM0*, then it is always true that q equals KM0′. However, q may equal KM0′ when KM0* is not equal to KM0′, although for DES this is an extremely unlikely event. Assuming that DES is a good pseudo-random number generator, so that each bit in a generated number is equally likely to be a 0 or 1, it follows that the probability of the event that q equals KM0′ and KM0* does not equal KM0′ is about 2^{-64}.

For all practical purposes, if q equals KM0′, then KM0* equals KM0′ (i.e., the entered value of the host master key is equal to the host master key in the cryptographic facility).

GENERATION OF DATA-ENCRYPTING KEYS

A data-encrypting key is produced from a 64-bit random or pseudo-random number (RN) by defining the number to be the desired data-encrypting key already enciphered under a key-encrypting key known to the cryptographic system. For example, in communication security, RN is defined as the session key (KS) enciphered under the host's master key (KM0):

$$RN \equiv E_{KM0}(KS)$$

On the other hand, in file security, RN is defined as the file key (KF) enciphered under a secondary file key (KNF):

$$RN \equiv E_{KNF}(KF)$$

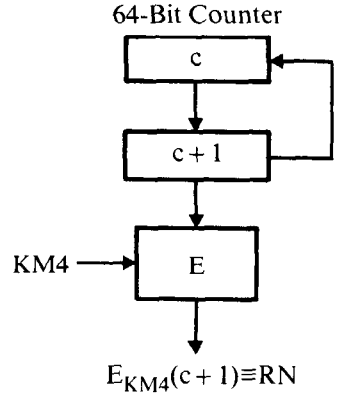
With this strategy, it is not necessary to generate first a data-encrypting key in clear form and then encipher it under the appropriate key-encrypting key. Data-encrypting keys are never exposed in clear form. Rather, they are dynamically produced as needed by the cryptographic system. This is accomplished by using the DES algorithm as a generator of pseudo-random numbers.

An Approach for Generating Keys with the Cryptographic Facility

One way to generate pseudo-random numbers (enciphered data-encrypting keys) is by using a nonresettable counter, or nonvolatile storage, that can be read and incremented only by the cryptographic facility. With this approach, the counter receives the same protection as the master key, working key, or any other intermediate value produced by one of the cryptographic operations.

A pseudo-random number is generated by incrementing the counter and enciphering the resultant value (c + 1) with a special variant of the host master key (KM4), as illustrated in Figure 6-6.

Because the pseudo-random numbers produced from this procedure (RN1, RN2, . . . , RNn) are each based on a different counter value, it follows that RN1 ≠ RN2 ≠ . . . ≠ RNn. The period of the pseudo-random number gen-



KM4 is a special variant of the host master key used only for the generation of RN. RN is defined as a data-encrypting key enciphered under a key-encrypting key known to the cryptographic system.

Figure 6-6. Procedure for Generating Pseudo-Random Numbers with the Cryptographic Facility

erating process is equal to the period of the counter (n). The procedure is such that it is not computationally feasible to deduce the secret key (KM4) or subsequent unknown values of RN using knowledge of one or more prior values of RN. Furthermore, this process is completely isolated from other cryptographic functions because KM4 is used in the procedure only for generating pseudo-random numbers.

Since the counter can be accessed only by the cryptographic facility, it is not possible for an opponent to reset its value and thereby cause prior data-encrypting keys to be regenerated. Moreover, if the counter is made large enough, say on the order of 64 bits, an opponent cannot cause the counter to recycle by making repeated requests for pseudo-random numbers. Thus previously generated numbers are never recreated. (Note that the counter value need not be kept secret to provide adequate cryptographic strength, but keeping the counter values secret will increase cryptographic strength.)

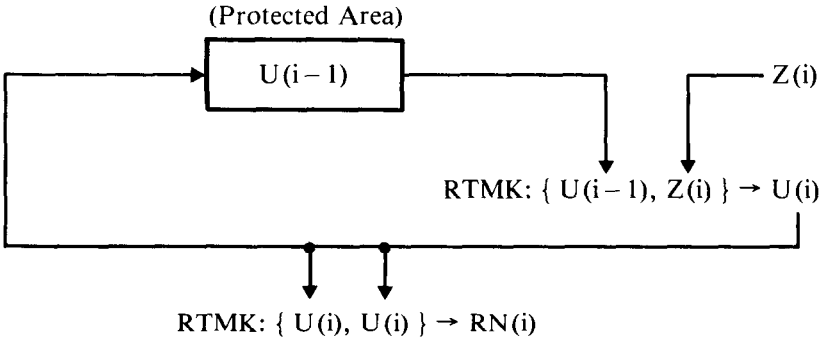
An Alternate Approach for Generating Data-Encrypting Keys

Basically, a pseudo-random number RN (defined as an enciphered data-encrypting key) is generated within the host processor as a result of the dynamically changing and unpredictable nature of the resource demands placed upon the system by its users.

The alternate approach for generating pseudo-random numbers makes use of the *reencipher to master key* (RTMK) operation in conjunction with two seed values, U and Z. These seed values are derived internally within the processor and are used as input parameters to an RTMK operation. The output of the operation is then used to derive RN. Two independent seed values provide the procedure with added strength, since both values must be compromised before a successful attack is possible. Figure 6-7 illustrates the described procedure for generating pseudo-random numbers.

For example, consecutive seed values Z(1), Z(2), . . . , Z(i) could be generated by combining two or more independent TOD clock values. Independent clock values can be achieved by interleaving an input-output operation of unpredictable duration between successive clock readings. The seed value U(0) could be derived from a combination of user-dependent and process-dependent information stored in the volatile memory of the host processor. Each value U(i), for any value of i greater than zero, is defined as the output of an RTMK operation whose input consists of U(i − 1) and Z(i). Hence it follows that U(i) is a function of U(0) and of Z(1), Z(2), . . . , Z(i).

It should be noted that RN(i) is a function of two independent quantities, U(i − 1) and Z(i), each having enough different values or combinations to prevent discovery by direct search. Each of the U-values, U(1), U(2), . . . , U(i), is generated internally by feeding back the result from the previous RTMK operation. A second RTMK operation is used to produce RN(i) from U(i) and to ensure that it is not possible to deduce U(i) from RN(i). Hence knowledge of one or more of the generated values of RN will not permit an opponent to deduce other (prior or subsequent) values of RN.



Where:

U(0) = arbitrary value.
Z(i) = function of two or more readings of the time-of-day clock.
RN(i) = generated pseudo-random number.
i = index number.

The specific relations are:

$$U(i) = E_{KM0}(D_{D_{KM2}(U(i-1))}(Z(i)))$$
$$RN(i) = E_{KM0}(D_{D_{KM2}(U(i))}(U(i)))$$

Figure 6-7. DES-Based Pseudo-Random Number Generator for Data-Encrypting Keys

To subvert this process for generating pseudo-random numbers, an opponent must contend with both a changing and unpredictable Z-value and a secret U-value that itself is a function of U(0) and all prior Z-values. Even if one of the seed values should be compromised, the other provides enough cryptographic strength so that an exhaustive attack intended to recover a set of eligible RN(i) would be computationally infeasible.

ENTERING A MASTER KEY AT THE HOST PROCESSOR

For reasons of security, the master key cannot be read once it has been installed in a cryptographic facility. However, the following procedure will allow a security officer to validate, with a high level of confidence, that the master key stored in the cryptographic facility is the one that was intended.

Some function (ϕ) of the master key is computed externally to the system and compared with a similar value computed within the system. For example, KM0 could be used as a key to encipher a 64-bit random number (RN);

$$\phi(KM0) \equiv E_{KM0}(RN)$$

Once the master key has been installed in a cryptographic facility, the

encipher under master key (EMK) operation could be used to produce the same quantity:

$$\text{EMK: } \{\text{RN}\} \longrightarrow E_{\text{KM0}}(\text{RN})$$

Comparison of these two values could then establish whether the keys used in the two routines were identical. The procedure is not foolproof, however, because the same incorrect key could have been entered in both routines. The objective, therefore, is to reduce the likelihood that an incorrect key is installed in the cryptographic facility in the first place.

Hard-Wired Entry

The reading of temporarily stored keys into the main memory of a system can be avoided by providing a direct wire connection between the point where the key is entered and the nonvolatile key storage area of the cryptographic facility. In this case, the key is entered by means of toggle switches, dials, or the like. The direct wire connection should be so constructed, as by shielding, that probing or tapping of transmitted information is not possible.

Even with hard-wired entry, there is still some chance that the key entered into the cryptographic facility will be different from the key that was intended. The following analysis provides an estimate for **p(UE)**, the probability of undetected error in the entered key (i.e., that an incorrect master key becomes installed in the cryptographic facility).

Of the sources of error affecting the key entry process, human error is the most critical. Mechanical or machine error occurs relatively infrequently, and therefore can be disregarded. At first, it is assumed that only one of the 16 hexadecimal digits entered into the cryptographic facility might be in error. A special case of multiple errors (i.e., double digit transposition of adjacent hexadecimal digits) is considered subsequently.[2]

In the situation described, a mistake in entering the key will not be detected if the parity of the incorrectly entered hexadecimal digit is correct. Since there are 16 possible hexadecimal digits, of which only 8 have odd parity, eliminating the correct digit leaves 7 combinations that have correct parity out of 15 possible combinations. Let

A = the event that the entered value of KM0 has correct parity

B = the event that KM0 is incorrectly entered

Then the probability of an undetected error, P(UE), is given by

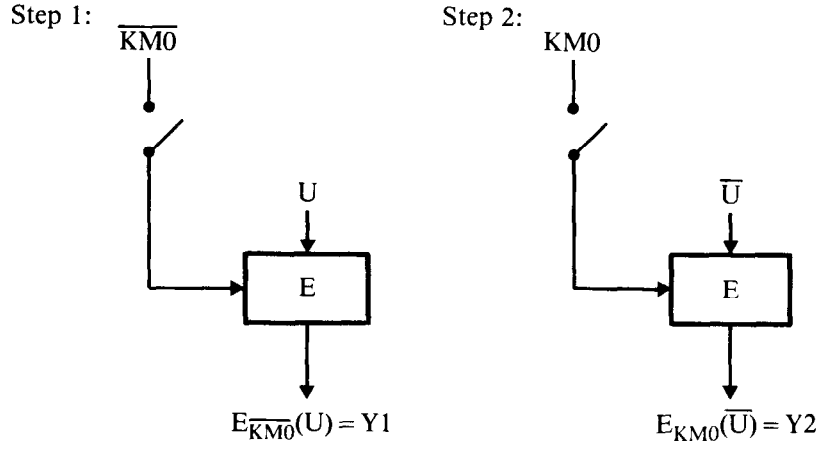
$$\begin{aligned} p(\text{UE}) &= p(\text{A and B}) \\ &= p(\text{A}|\text{B})p(\text{B}) \\ &= (7/15)p(\text{B}) \\ &= 0.47p(\text{B}) \end{aligned}$$

[2] It is assumed, as previously stated, that host and terminal master keys are entered in the form of 16 parity-adjusted hexadecimal digits.

where p(A|B) designates the probability of event A given that event B has occurred.

To improve the situation, two quantities, KM0 and a function (ϕ) of KM0, are specified in such a way that errors associated with the entry of KM0 and ϕ(KM0) are statistically independent. The choice of ϕ equal to the identity function would not be practical. This would amount to entering KM0 twice, so that an error in the first entry might well be repeated in the second, implying that the errors in KM0 and ϕ(KM0) could not be considered statistically independent. However, the choice of ϕ(KM0) = $\overline{KM0}$ (where overbar indicates the complement) is satisfactory. In this case, the complementary property[3] of the DES algorithm can be advantageously used to validate that the intended KM0 is submitted for entry in the host's cryptographic facility.

The procedure consists of first installing the complement of KM0 ($\overline{KM0}$) in the cryptographic facility and enciphering the arbitrary value U, then installing KM0 in the facility and enciphering the complement of U ($\overline{U}$). The output values are defined as Y1 and Y2, respectively. By the complementary property of the DES, KM0 can be assumed to be installed properly in the cryptographic facility whenever Y1 equals the complement of Y2 ($\overline{Y2}$). Any corruption that may occur in the master key during its transmission between the entry point and the cryptographic facility is also detected with this procedure. Figure 6-8 illustrates this entry procedure.



Notes: "——" indicates "the complement of."
Accept KM0 if Y1 = $\overline{Y2}$, otherwise reject KM0.

Figure 6-8. Validation of the Master Key as it is Entered at the Host Processor

[3] The complementary property of the DES algorithm stipulates that EK(X) equals $\overline{E_K(\overline{X})}$ for every key (K) and plaintext (X). (See also Classes of Ciphers, Chapter 3.)

The following analysis illustrates the superiority of validating the master key using KM0 and $\overline{KM0}$. Let

E1 = the event that the entered value of $\overline{KM0}$ is in error, but that the entered value has correct parity

E2 = the event that the entered value of KM0 is in error, but that the entered value has correct parity

E3 = the event that the entered value of $\overline{KM0}$ and the entered value of KM0 are complements of each other

An undetected error can occur only if all events, E1, E2, and E3, occur simultaneously. The notation (E1, E2, E3) denotes this joint event. The probability of an undetected error is thus given by

$$\begin{aligned} p(UE) &= p(E1, E2, E3) \\ &= p(E3|E1, E2)p(E2|E1)p(E1) \end{aligned}$$

Since it can reasonably be assumed that the events E1 and E2 are statistically independent, it follows that

$$p(E2) = p(E1) = (7/15)p^1$$

and therefore that

$$p(UE) = p(E3|E1, E2)(7/15)^2(p^1)^2 \qquad (6\text{-}3)$$

where p^1 is the probability that one digit in a key is incorrectly entered (i.e., is in error).

However, given that the events E1 and E2 do occur and that only one hexadecimal digit is in error, it follows that there is a 1/16 probability that the errors in $\overline{KM0}$ and KM0 occur in the same digital position, a requirement if the errors are to go undetected. Moreover, given that the errors occur in the same digital position, then there is a 1/7 probability that the two incorrect digits will be complements of each other. This is because there are seven incorrect hexadecimal digits with correct parity and only one of these that is the complement of the other incorrectly entered digit. Therefore

$$\begin{aligned} p(UE) &= (1/16)(1/7)(7/15)^2(p^1)^2 \\ &= 0.00194(p^1)^2 \end{aligned}$$

A more general result that takes into account the possibility that multiple errors may occur can be obtained in the following way. Let

E1,i = the event that exactly i digits (i = 1, 2, . . . , 16) in the entered value of $\overline{KM0}$ are in error, but the entered value has correct parity

E2,i = the event that exactly i digits (i = 1, 2, . . . , 16) in the entered value of KM0 are in error, but the entered value has correct parity

E3,i = the event that the entered value of $\overline{KM0}$ and the entered value of KM0, as given by E1,i and E2,i, are complements of each other

Then p(UE) can be expressed as

$$p(UE) = \sum_{i=1}^{16} p(E1,i, E2,i, E3,i)$$

$$= \sum_{i=1}^{16} p(E2,e, E3,i \mid E1,i)p(E1,i) \qquad (6\text{-}4)$$

As an example, let double digit transposition of adjacent hexadecimal digits be considered. (This is probably one of the most frequent multiple errors.) Let p2 be the probability of such an event. There are 15 error combinations for this case. Eight of these affect only 1 byte, case (a), with probability (8/15)p2 and 7 of these affect 2 bytes, case (b), with probability (7/15)p2. In the case where transposition takes place within 1 byte, parity is not lost, and thus $p(E1,2) = p(E2,2) = (8/15)p2$. In the case where transposition affects 2 bytes, the probability of obtaining correct parity is $p(E1,2) = p(E2,2) = (7/15)^2p2$, since there are 7 out of 15 hexadecimal combinations leading to the correct parity.

The conditional probability $p(E2,2, E3,2 \mid E1,2)$ is equal to (1/15)p2. This is so because the event E3,2, describing the complementary property, occurs only if the same digit pairs are affected in KMO and $\overline{KMO}$. (Otherwise the transposed digits destroy the complementary property of the entered values.) Thus only one transposition combination out of the 15 satisfies this condition. Given that E1,2 occurs, it also follows that parity is not affected in KMO if the same digit pairs are transposed in both KMO and $\overline{KMO}$. (Note that if parity is unaffected when a pair of digits in KMO (or $\overline{KMO}$) are transposed, then parity will also be unaffected when the corresponding pair of digits in KMO (or $\overline{KMO}$) are transposed.)

Since the events of cases (a) and (b) are mutually exclusive, probabilities can be added, and for i = 2 Equation 6-4 can be rewritten as

$$p(UE) = p2^2(1/15)\,[(8/15) + (7/15)^2] = 0.05p2^2$$

To reduce P(UE) still further, the key entry procedure could cause the key and its complement value to be displayed prior to the key being written in the cryptographic facility. This would allow an additional visual check to be made on the entered values.

Indirect Entry

To enter the master key indirectly, it is read into the main storage of the host processor and a *set master key* (SMK) operation is then used to write the key into the nonvolatile storage area of the cryptographic facility. Once the master key has been transferred, the copy of the master key in main storage is erased.

To reduce the likelihood of human error resulting in an incorrect master key being initialized in the cryptographic facility, it is recommended that the key be entered from a nonvolatile medium such as a punched card or magnetic tape. The card or tape could be stored in a secure location (e.g., a safe or vault) when not being used. As part of the procedure, the master key would be defined as that value which is recorded on the medium—provided, of course, that it has correct parity. Thus any human error committed in recording the key on the medium would be of no real consequence.

Any corruption of the master key between its entry point and the cryptographic facility resulting from either a hardware or software error could be detected by using the previously described procedure in which both the host master key and its complement value are entered (see Hard-Wired Entry).

ATTACK VIA EXTERNAL MANIPULATIONS

While the importance of having a physically protected area for the storage of the master key has been emphasized, the following illustrates that care must also be exercised in choosing a key entry procedure that is safe.

Let K denote an unknown master key and R the contents of the master key storage area in the cryptographic facility. Assume that K and R consist of 16 hexadecimal digits:

$$K = K1, K2, \ldots, K16$$
$$R = R1, R2, \ldots, R16$$

where K1 is stored in R1, K2 in R2, and so forth.

Assume further that the master key is entered into the cryptographic facility in a series of 16 consecutive steps. At each step, a single 4-bit hexadecimal digit of the key is entered into the next available location within R. The procedure is accomplished via manual switches. The storing of digits into R is controlled by an indexing circuit that indicates the next available location. When the switches are deactivated, the index is reset to point R1. However, the procedure has a weakness that can be exploited.

The unknown value of K1 can be attacked by systematically setting R1 to the values 0 through (hexadecimal) F. This can be done by activating the switches for entering the key, entering a trial digit, and deactivating the switches. A list of test messages, previously enciphered under the unknown master key, is now enciphered under each of the 16 different trial master keys. Since R2, R3, . . . , R16 are not changed in the procedure, the value of K1 can be determined by observing which entered digit causes the enciphered test messages to produce the correct ciphertext.

In like manner, K2 can be determined by systematically setting R2 to the values 0 through F. This is done by activating the switches, entering the known value of K1 into R1, entering a trial digit into R2, and deactivating the switches. The list of test messages is again enciphered under each of the 16 different trial master keys. Since R1 is set to K1 and R3, R4, . . . , R16 are not changed in the procedure, the value of K2 can be determined by ob-

serving which entered digit causes the enciphered test messages to produce the correct ciphertext.

Repeating the procedure, one can determine the digits K3 through K16. An average of 128 trials, or a maximum of 256 trials, are required. The described attack is thwarted by ensuring that R is automatically overwritten whenever the switches for entering the master key are activated. The process of overwriting the key is called *key zeroization* [3].

MASTER KEY ENTRY AT A TERMINAL

A terminal master key can be set by means of switches, dials, or a hand-held key loading device, or it can be entered at a keyboard. Again, because the terminal master key (KMT) cannot be read once it has been set, one should validate that the key has been properly initialized in the cryptographic facility.

On-Line Checking

One way of determining whether the proper master key has been initialized in a terminal's cryptographic facility is to establish a communications session with the host processor. If the installed terminal master key differs from the copy stored at the host, the session key initiated between the host and terminal will be different, and it will not be possible to send and receive an agreed-upon message.

A simple handshaking protocol could be adopted as part of the process of initiating a session. For example, the terminal could encipher a value N = (N1, N2) with the session key (KS) and send the resulting value to the host processor. Via the established protocol, the host would decipher the received quantity, apply some function to N such as switching the N1 and N2 to produce N′ = (N2, N1), reencipher the value N′ under KS, and send the result to the terminal. At the terminal, a check could then be made to ensure that the first and last halves of the deciphered value of N′ are equal to the last and first halves of N, respectively. If the values agree, the terminal master key would be accepted. Other approaches are possible (see Handshaking, Chapter 8).

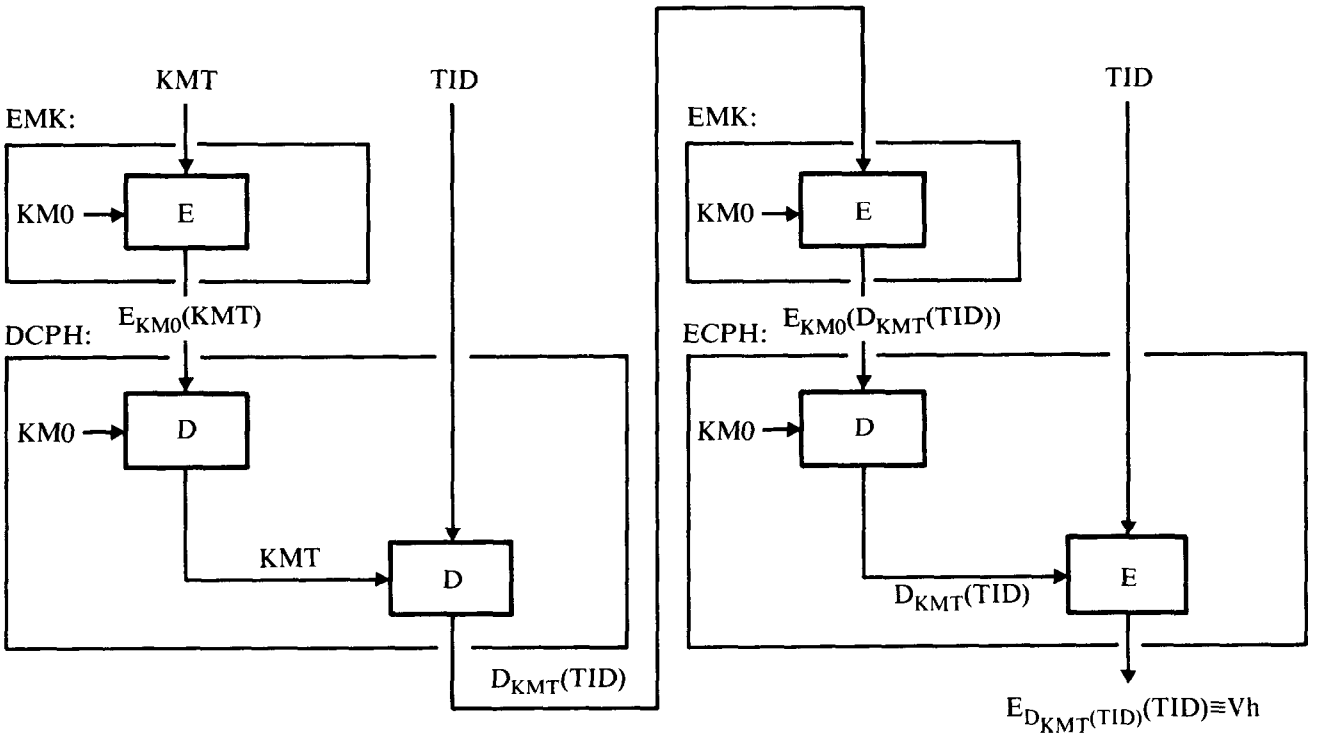
Off-Line Checking

It is often desirable to check KMT directly at a terminal without involving the host processor. To do so one can use a validating pattern. This pattern is a nonsecret function of KMT, and is created as part of the process of generating the key.

When a key is generated, the *encipher under master key* (EMK), *encipher data* (ECPH), and *decipher data* (DCPH) operations are used to produce a validating pattern (Vh) at the host system, as follows:

$$\text{EMK: } \{\text{KMT}\} \longrightarrow E_{KM0}(\text{KMT})$$

$$\text{DCPH: } \{E_{KM0}(\text{KMT}), \text{TID}\} \longrightarrow D_{KMT}(\text{TID})$$



TID = terminal identification number, unique to each terminal in a network.
Vh = validating pattern generated at a host processor.

Figure 6-9. Procedure, at the Host Processor, to Create a Validating Pattern for the Terminal Master Key

$$\text{EMK: } \{D_{KMT}(TID)\} \longrightarrow E_{KM0}(D_{KMT}(TID))$$

$$\text{ECPH: } \{E_{KM0}(D_{KMT}(TID)), TID\} \longrightarrow E_{D_{KMT}(TID)}(TID) = Vh$$

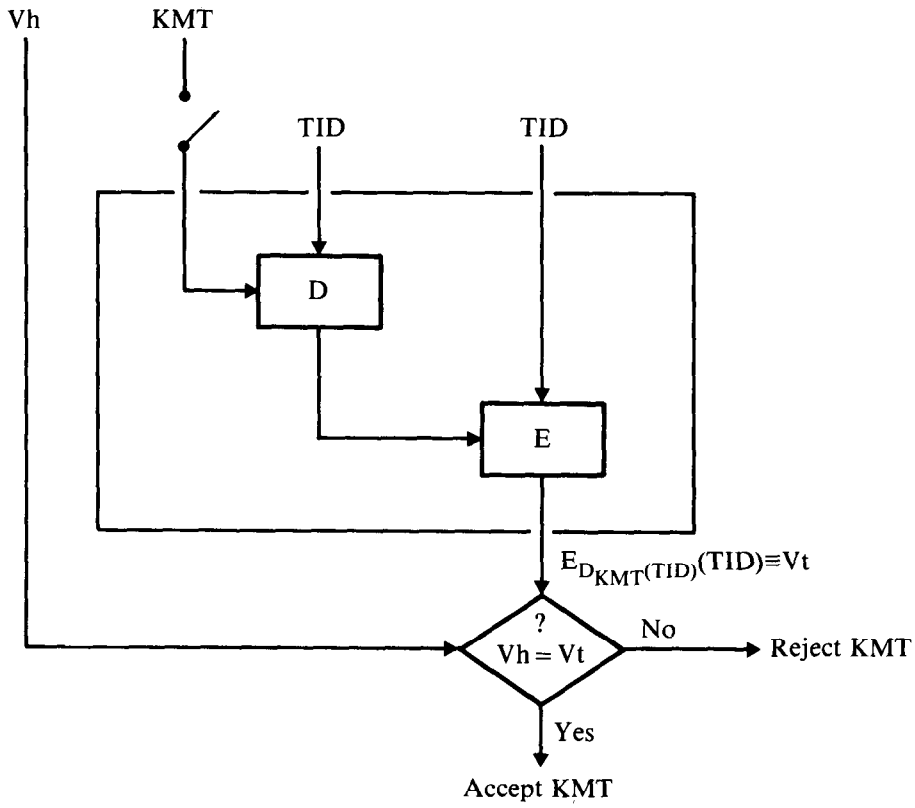
where TID is a terminal identification number unique to each terminal.

The steps to create Vh are shown in block diagram form in Figure 6-9. Later, at the terminal, the *decipher key* (DECK) and *encipher* (ENC) operations are used to produce a similar validating pattern (Vt), as follows:

$$\text{DECK: } \{TID\} \longrightarrow D_{KMT}(TID)$$

$$\text{ENC: } \{TID\} \longrightarrow E_{D_{KMT}(TID)}(TID) = Vt$$

It is assumed here that the terminal can ensure the integrity of TID and that an opponent is not able to cause the terminal to use a value other than TID in the computation of Vt.



Vh = validating pattern generated at a host processor.
Vt = validating pattern generated at a terminal.
TID = terminal identification number, unique to each terminal in a network.

Figure 6-10. Procedure for Entering and Validating Terminal Master Keys

The person who is authorized to enter KMT at the terminal is given the quantity Vh. As part of the procedure for entering KMT, the terminal will cause the value Vt to be generated and displayed. The operator can determine whether KMT has been entered correctly by comparing Vh with Vt. The steps to create Vt and validate KMT are shown in block diagram form in Figure 6-10.

If Vh is stored in nonvolatile storage at the terminal, frequent checks can be made on the correctness of the terminal master key. Alternatively, Vh could be written down and posted in a conspicuous location at the terminal. A keyboard entry command causing Vt to be generated and displayed would thus allow Vt and Vh to be compared periodically by the terminal operator.

DISTRIBUTION OF CRYPTOGRAPHIC KEYS

Whenever data-encrypting keys (session and file keys) occur outside the cryptographic facility, they are maintained under the encipherment of some key-encrypting key. This allows data-encrypting keys to be routed through the system over paths that are nonsecure. A data-encrypting key can be recovered in a usable form only if the recipient possesses the key-encrypting key under which the data-encrypting key has been encrypted.

Key-encrypting keys are distributed through the system in an altogether different way. One cannot always rely on encryption as a means of protecting the secrecy of these keys, since each node must have at least one key installed initially in clear form. That key must be sent to the node over a path with an acceptable degree of security (i.e., the probability of interception of the key must be very low). One such method is to use a courier, normally the safest and most secure means of transporting keys. Of course, security in this case depends on the reliability of the courier.

Although not necessarily recommended, other means of transmitting keys are by registered mail and by private telephone conversation. These methods are less secure than using a courier because there is a greater chance that an opponent could intercept the key during transmission. The probability of compromise could be reduced, however, by transmitting two or more bit patterns over independent paths and combining them, (e.g., by using an Exclusive-OR operation) at the final destination.

The same approach could be used when entering the key itself. For example, a different bit pattern could be entered into the cryptographic facility by each of several persons. These bits could then be combined within the cryptographic facility to produce the desired key. For the key to be compromised, this protocol would require the collusion of all persons involved in the key entry process.

The procedure for routing keys can be expressed more formally using statistical measures. Let T1, T2, . . . , Tn denote n different bit patterns of 64 bits each, and ϕ a nonsecret function used to produce a cryptographic key (K):

$$K = \phi(T1, T2, \ldots, Tn)$$

For example, ϕ could denote an operation that Exclusive-ORs the bit patterns together, as shown below:

$$K = T1 \oplus T2 \oplus \ldots, \oplus Tn$$

If desired, the technique would permit each of the different bit patterns to be entered into the cryptographic facility by a different person. This, of course, would require that function ϕ be available within the cryptographic facility itself. More likely, the bit patterns would be separately transmitted to a single person, who would in turn combine them using function ϕ and then enter the resulting key into the cryptographic facility.

Let Ai denote the event that bit pattern Ti is transmitted to a designated receiver without being compromised. Note that the complement of Ai ($\overline{Ai}$) denotes the event that Ti is compromised during transmission. Assume, as with the Exclusive-OR operation, that function ϕ is such that T1 through Tn must be compromised in order for the cryptographic key (K) to be compromised. Thus if B represents the event that key K is compromised, then the probability of event B, p(B), can be expressed as follows:

$$p(B) = p(\overline{A1})p(\overline{A1}|\overline{A2}) \ldots p(\overline{An}|\overline{A1}, \overline{A2}, \ldots, \overline{An})$$

If the events A1, A2, . . . , An are statistically independent, then p(B) can be expressed as the product of the probabilities;

$$p(B) = p(\overline{A1})p(\overline{A2}) \ldots p(\overline{An})$$

Even though individual values for $p(\overline{A1})$ through $p(\overline{An})$ may not be small enough to justify the transmission of only a single bit pattern, the product of the probabilities may be small enough to be acceptable.

Since assessment of the likelihood that cryptographic keys may become compromised is highly subjective, it is unreasonable to expect that accurate values for the various probabilities in question can ever be obtained. The model is useful mainly to demonstrate the underlying principle involved.

To illustrate this idea, suppose that T1 is sent by registered mail, T2 is sent by telegram, T3 is sent by private conversation, and that K is produced by Exclusive-ORing T1, T2, and T3. Since the bit patterns, T1 through T3, are transmitted via different paths, an assumption of statistical independence ought to hold, in which case, the probability that K is compromised should be the product of the probabilities that T1 is compromised, T2 is compromised, and T3 is compromised. Assuming that the probabilities of the events, p(T1), p(T2), and p(T3), are less than one, it follows that the product of these probabilities is less than any one of the values. Thus, sending the key as T1, T2, and T3 involves less risk than sending the key directly using only one of the paths.

LOST CRYPTOGRAPHIC KEYS

Remember that it is as difficult for the properly authorized user of the system to decrypt data when the key is unknown as it is for the hostile

cryptanalyst who never had the key in the first place. Consequently, if for any reason the cryptographic key required to decrypt data should become lost or unknown, the data will not be recoverable. Every effort should be made to adopt a set of administrative procedures and controls that will minimize the probability of losing cryptographic keys.

A copy of all pregenerated key-encrypting keys should be stored in a secure area (e.g., a safe or vault) in the event they are needed for the purpose of backup (see also A Procedure for Authentication of Cryptographic Keys, Chapter 8).

Cryptographic keys may become lost or unknown as a result of hardware malfunction, software error, or human failure. An undetected modification of a cryptographic key stored within the cryptographic system, or failure to use the proper key in the cryptographic facility may cause ciphering operations to proceed using an unknown key. In communication security, two nodes may attempt to communicate using different session keys, or in file security, stored data may be enciphered under a key that is different from that used for recovery.

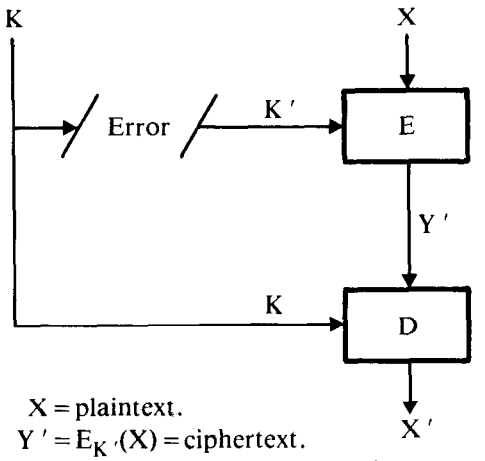
A simple handshaking procedure at session initiation ensures that both communicants are using the same session key. Message-authentication procedures can also be used to test that plaintext has been recovered with a proper key. Authentication techniques based upon the host master key permit keys to be validated prior to their use. As an extra measure, at the time data are enciphered they could also be deciphered to make sure that recovery is possible. (See Authentication Techniques Using Cryptography, Chapter 8.)

RECOVERY TECHNIQUES

In situations where a cryptographic key will not properly recover plaintext from ciphertext, it may still be possible to decipher the data using techniques for recovery. The underlying principle is that even though the exact key used to encipher the data is not known, one may still be able, using trial and error, to search the key space in a preferred order of likely candidates. If the list of likely candidates is small enough, then such a search may be successful.

Any error by a human, a machine, or software that causes a cryptographic key, K, to be changed to an incorrect key, K′, can be thought of as a function, ϕ, which maps the space of possible keys to itself. Hence recovery can be handled as a two-step procedure: (1) all available information concerning the nature of the error is used to compute a list of functions, $\phi 1, \phi 2, \ldots, \phi n$, identifying the most probable candidates, $\phi 1(K), \phi 2(K), \ldots, \phi n(K)$, for the incorrect key, K′, and (2) the data are then decrypted with each of these candidate keys.

In the present discussion, it is assumed that an incorrect key, K′, is used to encipher data and that the correct key, K, is used to recover the data (Figure 6-11). Alternatively, it may happen that the plaintext is enciphered with the correct key, K, but recovery is later attempted using a corrupted key, K′. In that case, a trial key is in the set of keys to be searched, provided that one of the functions, $\phi 1, \phi 2, \ldots, \phi n$, maps it to K′.



X = plaintext.
$Y' = E_{K'}(X)$ = ciphertext.
$X' = D_K(E_{K'}(X))$ = recovered plaintext.
K = key.
K' = corrupted key.

Figure 6-11. Encipherment Using a Corrupted Key and Decipherment Using the Correct Key

Consider the example of a data file enciphered with a key that in turn is written down on a slip of paper (as 16 hexadecimal digits) and stored in a vault. It is discovered later that the key will not decipher the data. Furthermore, it is suspected that an error occurred in recording the key. Since human error will most likely involve only one, two, or at most, three digits (the occurrence of multiple errors is small) recovery could be attempted by changing each digit, or combination of digits, in the incorrect key to its other possible values and then attempting to decipher and file with each of the keys. If this method does not succeed, it means that more digits were in error. Searching additional keys may be uneconomical, since many trials would be required.

SUMMARY

In connection with the key management scheme discussed in Chapters 4 and 5, two kinds of keys have been described: data-encrypting keys, which protect either data in transit (primary communication keys or session keys) or stored data (primary file keys or file keys); and key-encrypting keys, which encipher other keys—for example, host master keys, secondary communication keys (of which the terminal master key is a special case), and secondary file keys.

Generally speaking, the best method for generating a given class of cryptographic keys depends on the expected number of each type of key that will be needed and the time when the keys will be used. In many cases, the keys can be created dynamically (on demand), but sometimes they are required ahead of time in order to initialize the system.

The host master key is generated by a random process such as tossing

coins or throwing dice. Human involvement to that extent is reasonable in the process of generating keys because only one master key is required for each host processor, and the master key is likely to remain unchanged for a relatively long time. Since the master key protects all other keys stored at the host processor, special care should be taken to ensure that it is generated and installed in the cryptographic facility in a secure manner.

It is reasonable to anticipate that the total number of key-encrypting keys (excluding the host master key) may be large enough to warrant mechanical (nonhuman) generation procedures. The desired keys can be produced using the DES algorithm as a generator of pseudo-random numbers. The seed values used in this procedure are generated by the user, employing a random process similar to that used in generating the host master key. Since the key-encrypting keys are used in initializing the cryptographic system, they must be generated ahead of time. This can be accomplished under secure conditions using a computer.

Data-encrypting keys are also required in large numbers (one for each session and file using encryption), but they need not be generated until specifically requested (i.e., until they are needed to protect a communications session or stored data). Hence data-encrypting keys either could be generated ahead of time and stored in table form until needed, or they could be generated dynamically (on demand). Disadvantages in generating them ahead of time are that the keys would be exposed longer to possible compromise by an opponent, and they would require additional storage. One approach for dynamically generating data-encrypting keys is to make use of the randomness associated with the many users and processes normally active on the system at any one time.

Among the more important principles to be followed in key generation is that the compromise of one or more keys should not make it possible for any of the remaining keys to be deduced. With regard to key distribution, it was shown that security can be increased whenever two or more bit patterns of 64 bits are transmitted over different paths and combined at the final destination. To enhance the security of key installation, it is suggested that two different related values, the key and a function of the key, be entered into the cryptographic facility. Any errors that occur in both values will be statistically independent, so the likelihood of an undetected error (i.e., the probability that a wrong key will be installed) will be greatly reduced.

REFERENCES

1. Matyas, S. M. and Meyer, C. H., "Generation, Distribution, and Installation of Cryptographic Keys," *IBM Systems Journal,* **17**, No. 2, 126–137 (1978).
2. The RAND Corporation, *A Million Random Digits With 100,000 Normal Deviates,* Free Press, Glencoe, IL, 1955.
3. Federal Standard 1027, *Telecommunications: General Security Requirements for Equipment Using the Data Encryption Standard,* General Services Administration, Washington, D.C. (April 14, 1982).