Management of Secret Keys: Dynamic Key Handling

Joan Daemen

Banksys
Haachtesteenweg 1442
B-1130 Brussel, Belgium
Daemen.J@banksys.be

Abstract. In this paper we describe mechanisms for the management of secret keys, used in the securization of financial applications. The mechanisms Dynamic Key Handling and Reversed Dynamic Key Handling can be seen as particular methods to diversify keys in time, compatible with the present diversification by terminal and/or electronic purse.

1 Introduction

In this paper we describe two mechanisms that are used for handling secret keys used in the cryptographic protection of financial applications.

The first mechanism is the so-called Dynamic Key Handling (DKH). Basically, DKH is a method to synchronize cryptographic keys between the secure communication module (SCM) of an ATM, an EFT-POS or a Proton merchant terminal on one hand and the Host Security Module (HSM) of the central system on the other. The keys are used for several cryptographic services, related to the protection of the messages exchanged between the terminal and the Host. The most important of those services are the generation and verification of message authentication codes (MAC) and the encipherment of PINs.

The second mechanism is called Reversed Dynamic Key Handling (RDKH). RDKH is a method used to realize the regular update of cryptographic keys in the Proton electronic purses and the secure application modules (SAM) of Proton Merchant terminals without sacrificing full purse-terminal interoperability. The keys are used to secure the off-line purchase transactions between Proton purses and Proton merchant terminals. Banksys has patented the Reversed Dynamic Key Handling mechanism in Belgium and an International patent is pending.

2 Preliminaries

The communication and application security of the majority of financial applications today is based on the security services of message and entity authentication and encipherment, realized by using secret-key cryptographic functions. In general, the secret keys are protected by storing them in "tamper-resistant" devices (e.g., Smart Cards), capable of executing the necessary cryptographic functions.

The security of the system therefore relies on the tamper resistance of the security modules that hold the secret keys.

However, provably tamper-resistant modules do not exist and a possible key disclosure cannot be excluded. Therefore, it is wise to build systems in such a way that the consequences of the disclosure of key material are as small as possible and that a recovery to a secure situation is facilitated. The basic mechanism to obtain this is key diversification:

Hierarchic Key Derivation: terminal security modules and electronic purses are loaded with keys that are derived from master keys by means of one-way functions, diversified with purse or terminal dependent parameters. If a derived key is compromised, the secrecy of the corresponding master key is protected by the one-way property of the derivation function.

Session Keys: keys that are present in the terminal security modules and/or purses are not used directly for the calculation of authentication codes or for encipherment. Unique session keys are used instead. These session keys are also derived using a one-way function. Session key repetitions are prevented by the use of dynamic key handling and diversification using terminal and/or purse transaction numbers.

Regular Updates: Specific keys in the terminals and purses are updated regularly when they come online with the Purse Provider's Host/HSM. The necessity for compatibility with the key hierarchy has resulted in the Reversed Derived Key Handling mechanism.

The two mechanisms that are the subject of this paper, DKH and RDKH, can be seen as particular methods to diversify keys in time, compatible with the present diversification by component.

3 Dynamic Key Handling

The basic goal that underlies the application of the Dynamic Key Handling is the following:

The cryptographic keys that are inside a terminal may not allow to derive keys that have been used in any previous communication sessions.

DKH is in fact an implementation of the mechanism of Key Transformation as mentioned in ISO standard 11568-3 [1].

Every terminal secure communication module (SCM) has a unique identifier (ID_{SCM}) and is loaded initially with a unique seed key. To facilitate the key management at the HSM/Host side, they are derived from a single master key, stored inside the HSM. The derivation process has the following properties:

- The derivation function is a one-way function based on triple-DES.

– The derivation is parameterized by ID_{SCM} to assure uniqueness of seed keys.

Due to the one-way property of the derivation function, it is infeasible to derive the master key from any set of seed keys. Moreover, given any number of seed keys, corresponding to a set of terminals, it is infeasible to calculate a seed key of any other terminal. The HSM can at any time calculate the seed key corresponding to a terminal with a particular ID_{SCM}.

The key inside a terminal evolves irreversibly according to a non-trivial procedure. In the following subsections, we will describe and motivate the design of this procedure.

3.1 Linear Key Derivation

According to our basic goal, a cryptographic key that has been used by a terminal during a communication session with the Host/HSM must be deleted from the memory of the terminal SCM. Still, there must be key material available for future use. Moreover, the HSM must be able to derive the key used during a session.
Conceptually the simplest mechanism to obtain this is the following:

After use of a key, it is replaced by a value obtained by performing a one-way transformation on the key.

This is illustrated in Figure 1.

Fig. 1. Key evolution according to a linear structure.

With this solution, the workload of the terminal is one application of the one-way transformation per session. For the HSM we can choose between two possibilities:

– The HSM can keep the most recently used key of every terminal in a table and use it to derive the next key with.
– From the seed key (derived from master keys and ID_{SCM}) and the sequence number (e.g., communicated at the beginning of a session), the HSM can derive the used key by iterating the one-way transformation the appropriate number of times.

In the first case a large table containing a key for every terminal has to be managed by the Host/HSM. This can be realized by storing the keys in enciphered form on the Host system but this gives rise to an important key management

overhead. In the second case, the workload of the HSM grows to an unacceptable level as terminals in the field accumulate sessions. Therefore another approach has been chosen that allows the efficient derivation of keys in the HSM without the need for large key tables.

3.2 The Principle of Tree-Like Key Derivation

A key can be derived into several other keys by applying the one-way transformation for different values of a parameter. If for instance every key is derived into two other keys, the resulting diagram is a balanced binary tree: every node has two child nodes. An example of such a tree is depicted in Figure 2. It can be seen that a binary tree with depth d gives rise to $2^d - 1$ keys. The number of iterations of the one-way transformation to compute a key starting from the seed key (node 0) is called its *path length* and is determined by the depth of the node in the tree. Clearly, the maximum path length in a tree with depth d is $d - 1$.

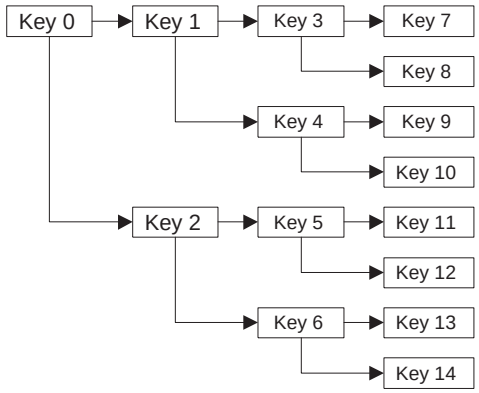

Fig. 2. Key derivation according to a tree structure.

Example 1. In a balanced binary tree with about one million nodes, the maximum path length is 19 and the mean path length is less than 16. For a linear key chain this is respectively 999,999 and 500,000.

Each key can be derived into more than two keys. If this number is equal for all nodes (except those of maximum depth) the tree is called *balanced*. For a balanced tree of depth d in which every node has n child nodes, the number of nodes is given by $(n^d - 1)/(n - 1)$.

According to our central goal, a key must be deleted from the memory of the terminal SCM after the session in which it is used. This implies that before deletion, its child keys (if any) must be computed and stored in the terminal SCM. Hence, for the binary tree diagram every use of a key below depth d gives rise to the storage of 2 new keys. This implies that a *table* of keys must be stored, rather than a single key. The key table consists of a number of *slots*, each capable of storing one key. A slot containing a key is called *occupied*, otherwise it is *free*.

The total amount of slots needed depends strongly on the order in which the keys are used. If the keys in Figure 2 are used in the order 0, 1, 2, ..., the maximum number of slots needed is 8. This situation occurs after the use of key 6, when keys 7 to 14 have to be stored. For a general balanced tree, this amounts to n^{d-1} slots needed. Hence, for this order, the number of slots required is comparable to the total number of keys in the tree.

By imposing another order, the number of slots required can be significantly reduced. Consider the following order in our binary tree example: 0, 1, 3, 7, 8, 4, 9, 10, 2, 5, 11, 12, 6, 13, 14. The number of slots needed is only 4. The maximum number of keys have to be stored after the use of key 3, when the key table contains keys 2, 4, 7 and 8. For a binary tree of depth d the required number of slots is d; for a general balanced tree, this is $(n-1)d+1$. This illustrates that using a limited number of slots and a limited depth, a balanced tree with a large number of keys can be generated.

Key evolution can be depicted in a so-called *key evolution diagram*, with the table slots set out vertically and the key evolution set out horizontally. The use and destruction of a key is denoted by a dot, key derivation is indicated by arrows. Figure 3 gives an example of a key evolution diagram for a balanced binary tree of depth 5.

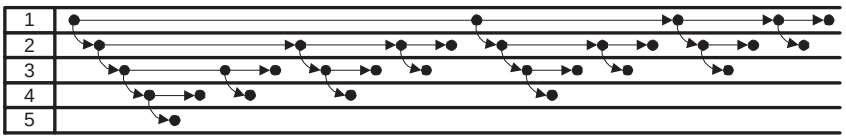

Fig. 3. Key evolution diagram with a binary tree structure.

Balanced trees may be the easiest case to study, they are certainly not the optimum solution. In Figure 3 it can be seen that the 5-th slot is only occupied during a single cycle. In actual implementations it is preferable to make optimal use of the available resources.

Example 2. Figure 4 contains a key evolution diagram in which the number of child keys of a key is equal to the number of available slots and which has maximum depth 5. This diagram requires only three slots, can generate 35 keys

and has a mean path length smaller than that of the binary balanced tree of depth 5 containing only 31 keys.

Fig. 4. Key evolution diagram with a non-binary tree structure.

Real-world constraints for a DKH scheme are :

- The number of required slots (SCM memory).
- The mean path length of the *used* nodes (average HSM workload).
- The total number of nodes (SCM potential life-time).
- The overhead (time and memory) of determining the position of a key in the key evolution diagram (SCM memory and workload).

3.3 Banksys DKH

In the key evolution as adopted by Banksys the position of a key in the key evolution diagram is indicated by its (unique) key *tag*. This key tag is used by the terminal to determine the keys that have to be derived after use of the key and by the HSM to derive the corresponding key from the seed key.
The two parameters for the Banksys key evolution are :

S: Number of slots,
K: Length of the key tag, expressed in bits.

Properties of the One-Way (Derivation) Transformation The DKH One-Way transformation has the following properties:

- It is a one-way transformation based on DES. It is infeasible to calculate the input given the output and the parameters.
- It is parameterized by the position of the last 1-bit in the key tag to diversify the different child keys derived from the same key.
- It is parameterized by the SCM identifier $\mathrm{ID}_{\mathrm{SCM}}$ to make certain that two equal keys for different SCMs do not give rise to two equal child keys.

Derivation of Keys After usage of a key, its derived keys (if any) are generated and stored in free slots. This includes the slot of the used key, that will be overwritten. Clearly, the number of derived keys from a key is upper bounded by the number of free slots available (say h) in the key table. Given the tag of a key, the tags of the derived keys are determined by the following procedure:

– Determine the number of free slots h. This includes the slot that holds the current key, so we have $1 \leq h \leq S$.
– Locate the rightmost all-0 bit string (say of length ℓ) in the key tag. We have $0 \leq \ell \leq K$.
– For i from 1 to ℓ, do the following: construct a tag by replacing the all-0 bit string by a bit string with a single 1 on position i. Derive the corresponding key and store it in a free slot. Mark the slot as occupied.

If $\ell = 0$, i.e., if the tag ends in a 1, no keys are derived. The key used in the previous session is deleted from memory and its slot is marked as free. Clearly the number of keys that are derived after a session is $\min(h, \ell)$.

Example 3. Say $K = 10$. The possible tags derived from key with tag `1011100000` are given by:
`1011110000`
`1011101000`
`1011100100`
`1011100010`
`1011100001`

If $h = 3$, only keys corresponding to the first 3 tags are generated and all 3 free slots are filled. If $h = 7$, two slots stay empty.

It can be seen that if the tags are considered as binary coded unsigned integers, the tag of a key is always higher than that of its parent key.

Derivation of a key in the HSM Given the key tag, the HSM is able to derive the corresponding key from the seed key. From the key tag derivation it follows that the parent of a key tag is formed by replacing the rightmost 1-bit by a 0-bit. This can be performed iteratively to arrive at the all-0 key tag of the seed key. The path length of a key, i.e., the number of iterations of the one-way transformation needed to calculate it, is equal to the Hamming weight (number of 1-bits) of the tag.

Example 4. the parent key of key with tag `1011100010` has tag `1011100000`. The complete path is given by:
`0000000000`
`1000000000`
`1010000000`
`1011000000`
`1011100000`
`1011100010`
where every key is derived from the key with tag in the previous line.

Determination of Successor Key Tag in the terminal After usage of a key, the key of the following session must be determined. This is called the *successor* key. The basic principle governing the order of the keys is:

The most recently derived key is used first.

This implies that the use of a key is followed directly by the use of all its descendents. The most recently derived key is simply the key in the table with the lowest tag, since it can be shown:
Property 1: The keys in the table have been generated in the reverse order of their tag values.

Proof. (by induction)
Initial situation:
The seed key with the all-0 tag is in the first slot. Since there is only one key in the table, the key with the lowest tag and the most recently derived key coincide.

Induction step:
Assume Property 1 applies for the current state of the key table. We will demonstrate that this is also the case for the successor state.

Case 1: No keys are derived from the current key.
The state transition simply consists of the deletion of the last key. Clearly, if Property 1 is valid for the current state, it is also valid for the successor state.

Case 2: There are keys derived from the current key.
The tags of the keys that are derived from the current key are obtained by replacing the rightmost all-0 bit string of the current key tag by `1000...`, `0100...`, etc. For these keys, it can be seen that the one with the lowest tag is generated last. It remains to be demonstrated that the tags of these keys are lower than the tags of all other keys in the table.
Assume the table contains a key (say X) with a tag that is lower than at least one of the new tags. Since Property 1 applies for the current state, the tag of key X must necessarily be higher than the tag of the current key. These two constraints impose that the tag of X must be equal to the tag of the current key, with the exception of the rightmost all-0 bit string. This implies that key X would be a descendent of the current key, and therefore necessarily generated more recently. This contradicts the hypothesis that Property 1 applies for the current state.

The key with an all-1 tag has no successor since it has no children and has the highest possible tag.

This scheme can be implemented by storing the tags together with the keys in the table. After generation of the child keys (if any), the successor key is determined by selecting the one with the lowest tag. In practice, the redundancy in the different key tags is used to reduce the amount of memory needed for their storage.

Example 5. If $h = 3$, the successor of key with tag `1011100000` has tag `1011100100`. If $h = 8$, the successor of key with tag `1011100000` has tag `1011100001`.

In fact, the tag of the successor key can be easily constructed from that of the current key, without having to resort to the tags stored in the table. We have

Property 2: The tag of the successor of a key with tag ending in a 1-bit is given by replacing the rightmost 0-bit and the trailing 1-bits of the current key by a 1-bit and an equal number of trailing 0-bits.

Proof. :

- The current key is the last descendent of a key X with tag given by replacing the rightmost all-1 bit string by a single 1 followed by an all-0 bit string.
- Consider the key Y with the tag constructed by taking the tag of X and interchanging the rightmost 1-bit and the 0 before it.
- Clearly key Y and X have the same parent key and were generated during the same step, Y just before X. Since the descendents of X are exhausted, Y is the most recent key in the table.

Example 6. The successor of key with tag `1011100111` has tag `1011101000`. `1011100111` descends from `1011100100`. Together with `1011100100`, the keys with tags
`1011110000`
`1011101000`
`1011100010`
`1011100001`
were generated. The last two have already been used before `1011100100`, the first two are still available in the table. The most recently generated is the one with tag `1011101000`.

Property 2 can be used to reduce the amount of memory needed for the table. Next to the actual key values, only the tag of the current key must be stored and the order in which the keys were generated.

3.4 Statistics

The two parameters S and K determine the total number of keys M and the mean path length p. In this subsection we derive expressions for these quantities. We have:
Property 3: The number of occupied key slots is given by the number of 0-bits before the rightmost 1-bit of the tag of the current key plus 1.

Proof. (by induction)
Initial situation:
Consider the successor state of the initial state (in which the key table only contains the seed key.) The current key has a tag with a single 1 in position S (the lowest occurring tag in the table) and the total number of occupied slots

is S. Clearly the number of 0-bits before the rightmost 1-bit of the tag of the current key is $S-1$, hence Property 3 holds.

Induction step:
Assume Property 3 applies for the current state of the key table. We will demonstrate that this is also the case for the successor state.

Case 1: No keys are derived from the current key.
The state transition simply consists of the deletion of the last key. The key tag of the successor can be constructed by replacing the rightmost `01111...` bitstring by `10000...`. The number of 0-bits before the rightmost 1-bit of the tag is diminished by 1 and so is the number of occupied slots.

Case 2: There are keys derived from the current key.
The keys that are derived from the current key are obtained by replacing the rightmost all-0 bit string by `1000...`, `0100...`, etc. The successor key of the current key is the one with the smallest tag. The additional number of 0-bits before the rightmost 1-bit of the successor key tag with respect to the current key tag is the number of newly generated keys minus one. This is equal to the additional number of occupied slots, hence Property 3 is preserved.

The number of possible keys with path length w is given by

$$\binom{\min(S+w-1,K)}{w}$$

This follows from the fact that the tags with Hamming weight w can be enumerated with the positions of the 1-bits. These bits are confined to the first $(S+w-1)$ positions of the tag. This can be seen as follows:

- The number of 1-bits is w.
- The number of 0-bits before the rightmost 1-bit is at most $S-1$, since the number of occupied slots may not exceed S (Property 3). Hence the rightmost 1-bit must be in position $S+w-1$ or lower. If the tag is shorter than $(S+w-1)$, obviously the 1-bits are confined to the K bits of the tag. Observe that for small Hamming weights this expression is independent of the size of the key tag K.

The total number of keys M is given by summing over all path lengths:

$$M = \sum_{w=0}^{K} \binom{\min(S+w-1,K)}{w}$$

This can be rewritten as:

$$M = \sum_{w=0}^{K} \binom{\min(S+w-1,K)}{w}$$

$$= \sum_{w=0}^{K-S} \binom{S+w-1}{w} + \sum_{w=K-S+1}^{K} \binom{K}{w}$$
$$= \sum_{w=0}^{K-S} \binom{S-1+w}{w} + \sum_{w=0}^{S-1} \binom{K}{w}$$
$$= \binom{K}{S} + \sum_{w=0}^{S-1} \binom{K}{w}$$

Hence, the total number of keys is

$$M = \sum_{w=0}^{K} \binom{K}{w}$$

The mean path length p is given by taking the average for all possible keys:

$$p = \frac{1}{M} \sum_{w=0}^{K} \binom{\min(S+w-1, K)}{w} w$$

Table 1 gives M and p for some typical values of K and S.

	$K = 16$		$K = 24$		$K = 32$	
S	M	p	M	p	M	p
4	2,517	10.60	12,951	16.92	41,449	23.29
5	6,885	10.33	55,455	16.85	242,825	23.46
6	14,893	9.92	190,051	16.55	1,149,017	23.32
7	26,333	9.47	536,155	16.12	4,514,873	23.00
8	39,203	9.02	1,271,626	15.60	15,033,173	22.53
9	50,643	8.63	2,579,130	15.04	43,081,973	22.00
10	58.651	8.34	4.540.386	14.47	$1.076\ 10^8$	21.40
11	63.019	8.14	7.036.530	13.92	$2.966\ 10^8$	20.77
12	64.839	8.05	9.740.686	13.40	$4.624\ 10^8$	20.13

Table 1. M and p for some typical values of S and k.

3.5 Session Key Derivation and Usage

The terminal uses the keys in its key table to :

- secure PIN transmission by encipherment,
- provide message authentication by generation of MAC,
- provide application-level authentication codes.

The keys that are in the key table are not used directly for these purposes, but are the input to a one-way function to derive the actual session keys.

In order for the HSM to derive the current key of a terminal, it must know its tag. Since the Host/HSM does not keep track of the tags of all the terminals, the HSM can only use a key for the protection of a message towards the terminal if the terminal has already contacted the HSM and communicated its tag previously.

4 Reversed Dynamic Key Handling

PROTON is a electronic purse scheme exploited (Belgium) and sold (abroad) by Banksys. The security of its architecture is to a large extent realised by the use of an elaborated scheme of differentiated cryptographic keys.

Proton purchase transactions, i.e., transactions between Proton purses and merchant terminals are done off-line. There is no connection to the centralized Host system. A purchase transaction is secured by session keys that are derived from keys inside the purse and the secure application module (SAM) of the merchant terminal. For every purchase transaction a new session key is calculated by purse and SAM. This session key is determined by the pair of master keys (KM_A and KM_B) and a number of session-specific parameters.

This is realized by a two-level hierarchical key structure whose detailed description is out of the scope of this paper. The SAMs are subdivided in a number of families, each of which receives a different key derived from KM_A. These "family keys" KD_A are again derived by Purse ID to keys KDD_A and written into the purses. Likewise, the purses are subdivided in a number of groups, each of which receives a different key derived from KM_B. These "group keys" are again derived by SAM ID and written into the SAMs. At the beginning of a purchase transaction a unique session key is calculated by purse and SAM partly using, partly deriving common key material.

To allow the replacement of compromised keys, the keys derived from KM_A are dynamically updated during the life of the Proton System. This is where the Reversed Dynamic Key Handling is applied. For simplicity, we will in our explanation of RDKH ignore the double key hierarchy.

The keys KD_A are dynamically updated during the life of the Proton system. The time interval between two key updates is called a *period*. The subsequent periods are identified by the period number PN_{PP}, that is managed centrally by the Purse Provider. Obviously, an update of the keys KD_A affects the keys KDD_A. Hence this dynamic key mechanism implies both key updates for the SAMs as for the purses. Since a key update requires the SAM or purse to be on-line with the Purse Provider Host/HSM, a simultaneous update of all these keys is not realistic. It is therefore necessary to have a mechanism that allows transactions between purses and SAMs with debit keys corresponding to different periods. This is realised in the following way.

A SAM must contact the Purse Provider Host/HSM at fixed intervals to collect its revenues. When the PN_{PP} is incremented with respect to the previous

collection, a new period has started and the key KD_A is loaded. A merchant terminal is assumed to be on-line at least once a period, hence its SAM contains the debit keys corresponding to the current period PN_{PP} or the previous period $PN_{PP}-1$.

A purse comes on-line with the Purse Provider for load transactions. When the PN_{PP} is incremented with respect to the previous load transaction of the purse, the keys KDD_A corresponding to the previous period ($PN_{PP}-1$) are loaded. Since it may take several periods between two subsequent load transactions of a purse, a purse may carry keys corresponding to old periods.

The keys KD_A have been generated in such a way that given KD_A of a certain period, the corresponding key of the preceding period can be calculated, and by induction the keys KD_A of all previous periods. However, the one-way property of the function used to generate these keys makes it infeasible to calculate keys corresponding to future periods.

It can be seen that for any SAM/purse pair, the PN of the SAM keys is larger than or equal to the PN of the purse keys. In a transaction between a SAM and a purse with $PN_{purse} < PN_{SAM}$, the SAM starts by calculating the key KD_A corresponding to the PN_{purse}. This consists of applying the one-way function ($PN_{SAM} - PN_{purse}$) times to its current key.

The dependencies between the keys involved in RDKH is illustrated in Figure 5.

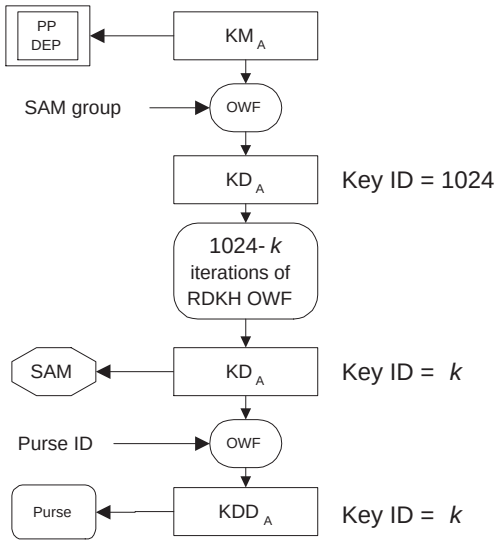

Fig. 5. RDKH key derivation

5 Conclusions

We have presented two methods for the dynamic handling of diversified secret keys.

References

1. ISO 11568-3, *Banking - Key management (retail) - Part 3: Key management techniques for symmetric ciphers.* DIS 11568, ISO, 1993.