PIN Recovery Attacks

Jolyon Clulow
Prism
jolyonc@prism.co.za

Abstract
The author has discovered several weaknesses in the functions of the standard financial crypto transaction set (or API). These weaknesses lead to a variety of PIN recovery attacks applicable to hardware security modules (HSM) or cryptocoprocessors. The attacks are extremely fast, taking only a couple of seconds

Keywords: PIN recovery attacks, tamper resistant/responding security modules, API attacks

Table of Contents

1. Introduction

Tamper resistant/responding security modules (TRSM) fulfill an important role in financial transaction networks by providing a secure trusted environment within which to store and manipulate sensitive data. The monetary value of the data protected by TRSMs is indeed significant, representing the sum of bank, debit and credit card transactions.

Much work has been done in evaluating the security of such devices from both a physical resistance to attack perspective as well as the logical security for which [1] provides a comprehensive introduction and overview. In this paper, we develop several API attacks (as in [2], [3] and [4]) but against the standard PIN functions that are common to APIs used in financial transaction networks (or ATM networks). These PIN recovery attacks are computation trivial and extremely fast requiring only seconds to successfully extract a PIN.

While it is outside of the scope of this paper to detail a 'standard PIN API', we provide a brief overview. A raw PIN is formatted it into one of a number of possible PIN block formats, which is then encrypted (typically using 3DES). This encrypted PIN block is sent from the originating point, across a financial network to the account holder's institution, to be verified. Every two connected nodes on the network, share a unique key thereby establishing a secure zone between the two parties. When it is 'switched' through the network, it becomes necessary to 'translate' the encrypted PIN block from one zone key to another. At the same time, it may be required to 'reformat' the PIN from the existing format to a new one. Thus a standard PIN API, will have the ability to verify an encrypted PIN, translate an encrypted PIN between zone keys and to reformat an encrypted PIN. It obviously supports some set of PIN formats. A more comprehensive discussion including details of the algorithms involved is included in the appendix, which the reader is encouraged to consult. [5] serves as a comprehensive reference API.

2. Known Attacks and Assumed Level of Security

The level of security offered by any cryptographic protection (function) is measured by the effort required to defeat it. Thus, we briefly examine the known attacks against PIN transactions to establish the 'assumed' level of security.

2.1. Exhaustive Key Search (Brute force)

Exhaustive key search is (almost) always a possibility. It is an understood upper bound on the level of security that can be achieved - and so any technique that does not improve on it is of little interest. The strength of a secure algorithm is measured by the length of the key (effectively the size of the key space required to be searched). The current standard PIN transaction systems are 3DES (ref VISA reqs) - although many (typically historic) single DES systems still exist. Note, this is not an attack against correctness. In our current reality, a 3DES system is practically immune to such an attack.

2.2. Exhaustive Pin Search

The PIN space is considerable smaller than the key space. It is thus critical to prevent an adversary from being able to mount such an attack (since he would surely and rapidly succeed). It is probably for this reason that the ANSI X9.8 standard (Personal Identification Number (PIN) Management and Security) states "The system shall not be capable of being used or misused to determine a PIN by exhaustive trial and error". One obvious potential weakness would be any implementation, which accepts clear PINs to either the PIN verification or generation functions. An API that allows such functionality typically restricts it to a secure or authorized mode, thereby ensuring that it is not misused. At some point, the user has to be able to enter a PIN in the clear. However, this is a manual process (usually at a trusted interface) and hence cannot be easily automated. In addition, it is common practice to detect and prevent a user trying many combinations.

2.3. The Code Book Attack

To mount such an attack, the adversary will build up a 'code book' containing every PIN and the result of that PIN encrypted under a given key. To recover an unknown encrypted PIN, the attacker simply consults the codebook to find the encrypted PIN block and hence identify the associated PIN. Should the PIN be encrypted under a different key to the one used for the codebook, the attacker simply translates it to encryption under the codebook's key. As with the exhaustive key search, it should not be possible for the attacker to build up such a codebook and the same practical limitations apply.

For an ANSI X9.8 format n digit PIN, 10^n encrypted PIN blocks are required to be stored (for a given account number). This represents a trivial memory requirement and is insignificant to search. Naively, attacking formats containing random padding would appear to require a larger codebook, but this is not the case. All formats are 'equally' vulnerable, since the format can be changed in the translate(reformat) call (to the weakest one). The same technique can be used to 'eliminate' the variation offered by the PAN. It is noteworthy that regardless of format, key and pan, all encrypted pins are potentially vulnerable to a single codebook.

2.4. Key Separation Attacks

Key separation is a mechanism, which enforces that a given key is used as intended. The well known attack scenario, is supplying an encrypted PIN block and the PIN encrypting key to a standard data decrypt call, which would result in the clear PIN block being returned. This is an attack on the correctness of the transaction set and demonstrates the necessity to 'separate' PIN encrypting keys from data decrypting keys. There are two common methods for achieving this. The first is to use different master keys for encrypting different types of keys. Incorrectly using an encrypted key, would result in the key being decrypted under the incorrect master key, yielding a 'random' result. This may be detected if parity checking of keys is enforced or enabled. The second method, involves the creation of a variant of the master keys based on the type of key. Perhaps the most widely used system, is IBM's control vector method. A unique control vector is associated with each type of key. To encrypt or decrypt the key, the master key variant is created by xor-ing the master key with the control vector. Again, using an encrypted key as incorrect type, results in a 'random' result.

3. Attack Models

We have the following inputs:

- Query access to a (potentially tamper proof) device with the typical PIN transaction set (as described above)

- An encrypted pin encrypting key which is valid for the device above (i.e. the PIN encrypting key is itself encrypted under a (master) key resident in the device)
- A valid encrypted PIN block (EPB), which was encrypted under the encrypted PIN encrypting key

It is our goal to find superior techniques to those listed under `Known Attacks', which exploit potential lack of correctness of the typical PIN transaction sets.

4. Manipulation Techniques

We describe a set of techniques that allow for manipulation in some or other useful manner. These techniques form our toolkit for the attacks that follow.

We assume that encrypted PIN block is in ANSIX9.8 format and it's associated PAN is the 12 digits = "000000000000". This is a trivial assumption, since we can always use the reformat call to reformat any other pin block to this. For simplicity of representation, we also assume that the PIN is of length 4.

4.1. Modifying the PIN Block

Our intention here is to obtain a new EPB', which has the same PIN as the original EPB but for which the clear PIN Block PB' = PB ⊕ 0000RRRRRRRRRRRR.
This is trivial. We translate from the original PAN, to a new PAN' = PAN ⊕ RRRRRRRRRRRRRRR. This can also be represented by P2' = P2 ⊕ 0000RRRRRRRRRRRR. Observe the process

PB	=	$d_{(K)}$(EPB)
P1	=	PB ⊕ P2
	=	0DPPPPFFFFFFFFFFFF
P	=	PPPP
P1'	=	0DPPPPFFFFFFFFFFFF
PB'	=	P1' ⊕ P2'
	=	P1 ⊕ (P2 ⊕ 0000RRRRRRRRRRRR)
	=	(P1 ⊕ P2) ⊕ 0000RRRRRRRRRRRR)
	=	PB ⊕ 0000RRRRRRRRRRRR

This call will always succeed, provided that the value P2' obeys any rules associated with it. Note that the value of P1 has remained constant. The operation is denoted

$$\text{ANSI X9.8}_{\text{Source PAN}} \rightarrow \text{ANSI X9.8}_{\text{Target PAN}}$$

Example:

```
PIN                  1234
PAN                  012345678901
Clear PIN Block      041235DCBA9876FE

PIN enc key          0505050505050505
Encrypted PIN Block  DC674029B47666C3

In the translate call, we specify a new PAN'
PAN'             112345678901 (PAN' = PAN ⊕ 100000000000)
Clear PIN BLOCK      041225DCBA9876FE (= PB ⊕ 0000100000000000)
Encrypted PIN Block' 011720D9BF9D73FB
```

Hence we have succeeded in modifying the clear PIN block.

4.2. Modifying the PIN

Definition

PAN Casting (PC) is the process of interpreting an ANSI X9.8 PIN block with a given PAN (called the *source* PAN) as an ANSI X9.8 PIN block with a chosen PAN (called the *applied* PAN).

This operation is achievable (even when restricted to working with encrypted PIN blocks) through the use of the translate function. Given an encrypted ANSI X9.8 PIN block with *source* PAN, supply it to the reformat call, specifying the *applied* PAN as the input PAN for the encrypted PIN block and selecting a suitable output format (called the *target* format). The implementation may fail the call if the modified PIN block does not satisfy the rules of the implementation. We denote the operation as :

$$\text{ANSI X9.8}_{\text{Source PAN}}\text{: PC(ANSI X9.8}_{\text{Applied PAN}}) \rightarrow \text{ANSI X9.8}_{\text{Target PAN}}$$

Our intention here is to obtain a new EPB', with a modified PIN (written as P' = PPPP ⊕ 00RR). We perform a reformat call with an incorrect input PAN' (P2' = P2 ⊕ 000RR0000000000). The operation can be written as

$$\text{ANSI X9.8}_{\text{PAN}}\text{: PC(ANSI X9.8}_{\text{PAN' = PAN} \oplus \text{RR0000000000}}) \rightarrow \text{ANY}.$$

Observe the process

$$\begin{aligned} PB &= d_{(K)}(EPB) \\ P1' &= PB \oplus P2' \\ &= (P1 \oplus P2) \oplus (P2 \oplus RR0000000000) \\ &= P1 \oplus 0000RR0000000000 \\ P' &= P \oplus 00RR \end{aligned}$$

P' is then formatted into PB' and encrypted to yield EPB'. This call is successful provided that P2' and P1' are deemed to be valid.

Example:

```
PIN                  1234
PAN                  012345678901
Clear PIN Block      041235DCBA9876FE

PIN enc key          0505050505050505
Encrypted PIN Block  DC674029B47666C3

In the translate call, we provide PAN' (as opposed to the 'correct' PAN)
PAN'             112345678901 (PAN' = PAN ⊕ 100000000000)
Clear PIN BLOCK      041235DCBA9876FE

Extracted PIN        1224 (= PIN ⊕ 0010)
```

Hence we have succeeded in modifying the PIN.

Definition

Format Casting (FC) is the process of interpreting a PIN block of a given format (called the *source* format) as a PIN block of a chosen format (called the *applied* format).

Again, this operation remains achievable even when restricted to working with encrypted PIN blocks, through the use of the translate function. Given an encrypted PIN block of *source* format, supply it to the reformat call, specifying the *applied* format as the input format of this PIN and selecting a suitable output format (called the *target* format). We denote the operation as

Source Format : FC (Applied Format) $\rightarrow$ Target Format

The implementation may fail the call if the source format PIN block cannot be successfully interpreted as a PIN block of applied format, according to the rules of the implementation. This operation can be used in a variety of useful ways to manipulate the PIN. Perhaps the most powerful application of this technique is in the modification of the length of the PIN (both to extend and reduce). We demonstrate it's use in extending the length of the PIN.

4.3. Modifying the length of the PIN

Our intention here is to obtain a new EPB', with a modified PIN (P' = 0LPPPP). We perform the following operation

ANSI $X9.8_{PAN=0}$: FC(VISA-3) $\rightarrow$ ANSI $X9.8_{PAN = 0}$

The implementation performs the following steps:

$$
\begin{aligned}
PB &= d_{(K)}(EPB) \\
&= 0LPPPPFFFFFFFFFF \oplus P2 \\
&= 0LPPPPFFFFFFFFFF
\end{aligned}
$$

As a VISA-3 format PIN, the PIN is extracted as

P' = 0LPPPP

and reformatted to

$PB' = 0L'0LPPPPFFFFFFFF \oplus P2'$

where P2' is the output pan specified. Note that L' = L+2. This call will succeed provided P' is deemed to be valid.

5. Extending Known Attacks

5.1. The Code Book and Exhaustive Search Revisited

The ability to modify PIN length has immediate and dramatic ramifications since it allows us to greatly reduce the data collection and search effort. Using variations on the techniques described, it is possible to reduce the PIN space. For example, using

VISA-3 : FC(ECI-2) $\rightarrow$ ANY
(yielding $P' = P_1P_2P_3P_4$)

we can attack the first 4 digits of any PIN independently while

VISA-3 : FC(IBM 3621) $\rightarrow$ ANY
(yielding P' = $P_5P_6P_7P_8P_9P_{10}P_{11}P_{12}$)

allows exposes the remaining digits. We can repeat the process, ultimately turning a 12 digit PIN into 3 separate 4 digit PINs. Thus, regardless of length, all PINs are equally vulnerable to a 4 digit PIN electronic codebook or exhaustive search attack!

Given the 4 digit PIN P = $P_1P_2P_3P_4$. Let the sequence S_1 be the operations

VISA-2 : FC(VISA-3) $\rightarrow$ ANY
(yielding P' = $4P_1P_2P_3P_40$)

ANSI X9.8 : FC(IBM 3621) $\rightarrow$ ANY
(yielding P'' = $P_2P_3P_40$)

and let sequence S_2 be the operation
ANSI X9.8$_{(PAN = 00FF00000000)}$: FC(IBM 3621) $\rightarrow$ ANY
(yielding P' = P_3P_400)

Note by applying S_2 followed by S_1, we obtain PIN P' = P_4000.

By applying the next operation three times consecutively to each of the PINS in the set {$P_1P_2P_3P_4$, $P_2P_3P_40$, P_3P_400, P_4000}, we obtain the set { $P^*_i = 444P_i \mid i \in 1..4$ }.

VISA-2 : FC(ECI-2) $\rightarrow$ VISA-2
(yielding P' = 4 $P_1P_2P_3$)

Combining it all, we can reduce a L digit PIN to the set {$P^*_i = 444P_i \mid i \leq L$}. The attacker's requirements have been reduced to a code book of 10 known 4 digit PINs, namely {$P^*_i = 444i \mid i = 0..9$}, and access to the reformat function in order to recover any PIN!

6. Oracles

6.1. Introduction

This section follows a simple strategy. We define an oracle and investigate the properties thereof, describing how the oracle could be used to identify a 'number' and an algorithm to do so. We then show how to instantiate such an oracle using standard PIN functions, ultimately yielding a method to recover the digits of the PIN (i.e. a PIN recovery attack).

6.2. Theory

We begin by defining three useful oracles. For x, a, b, n $\in$ Z (a unknown).

Definition

Given a query x, oracle $g_{(a,n)}$ returns *true* if $a \oplus x = n$, else *false*.

Definition
Given a query x, oracle $h_{(a,n)}$ returns *true* if $a \oplus x < n$, else *false*.

Definition
Oracle $l_{(a,b)}$ returns *true* if a = b, else *false*.

Lemma
Given $x, n \in Z$, (n even), then $x \oplus 1 < n$ iff $x < n$.

Theorem
Given $a, n \in Z$, (a unknown, a < n).

1. It is possible to uniquely identify a by querying oracle $g_{(a, n)}$.
2. If n is even, it is not possible to supply a query $x \in Z$ to oracle $h_{(a, n)}$ which can distinguish between a and $a \oplus 1$.
3. If n is even, n/2 odd, it is possible to identify a as either a or $a \oplus 1$ by querying oracle $h_{(a, n)}$.

These lead to obvious algorithms for identifying a.

Algorithm: Using $g_{(a,n)}$ to recover $\{a, a \oplus 1\}$

1. For each possible value of a_i, $i \in Z_n$
   1.1. Calculate $x = a_i \oplus (n-1)$.
   1.2. Return a_i iff $g_{(a, n)}(x)$ = *true*.

Algorithm: Using $h_{(a,n)}$ to recover $\{a, a \oplus 1\}$

1. For each possible value of a_i, $i \in Z_n$
   1.1. Calculate $x_1 = a_i \oplus (n-1)$ and $x_2 = a_i \oplus n$.
   1.2. Submit x_1, x_2 to $h_{(a,n)}$
   1.3. Return $\{a_i, a_i \oplus 1\}$ iff $h_{(a,n)}(x_1)$ = *true* and $h_{(a,n)}(x_2)$ = *false*.

For n = 10, there is a more efficient algorithm using $h_{(a, 10)}$.

Algorithm: Efficient algorithm for identifying $\{a, a \oplus 1\}$ using $h_{(a, 10)}$, a < 10

For $a \in \{0,1,2,3,4,5,6,7,8,9\}$, notice $a \oplus 8 < 10$ for $a \in \{0,1,8,9\}$, while $a \in \{2,3,4,5,6,7\}$ will fail $h_{(a, 10)}$.

1. Query $h_{(a, 10)}(8)$.
   1.1. If $a \in \{0,1,8,9\}$, $a \oplus 4 < 10$ for $a \in \{0,1\}$. Hence query $h_{(a, 10)}(4)$ will distinguish between {0,1} and {8,9}.
   1.2. If $a \in \{2,3,4,5,6,7\}$,
   $a \oplus A < 10$ for $a \in \{2,3\}$,
   $a \oplus C < 10$ for $a \in \{4,5\}$, and
   $a \oplus E < 10$ for $a \in \{6,7\}$.
   Hence the queries $h_{(a, 10)}(A)$, $h_{(a, 10)}(C)$ and $h_{(a, 10)}(E)$ will distinguish between {2,3}, {4,5} and {6,7}.

This algorithm takes only 3 operations to identify $\{a, a \oplus 1\}$.

6.3. Instantiating the Oracles

Let $A(l, i, x)$ be l hexadecimal digits long and all zero's, except for the i^{th} digit which is x (e.g. A(12, 2,7) = 070000000000).

6.3.1. Realization of the oracles $h(P_{i+2}, 10)$, $l(PB_{i+4}, F)$

Note: Using the translate function with consistency checking.

Consider

$$\text{ANSI X9.8}_{PAN}: PC(\text{ANSI X9.8}_{PAN' = PAN \oplus A(12, i, x)}) \rightarrow \text{ANSI X9.8}_{PAN}$$

$$\begin{aligned}
PB &= d_{(K)}(EPB) \\
P1' &= PB \oplus P2' \\
&= (P1 \oplus P2) \oplus (P2 \oplus A(16, i+4, x)) \\
&= P1 \oplus A(16, i+4, x) \\
P' &= P \oplus A(L, i+2, x) \\
EPB' &= e_{(K)}(P1')
\end{aligned}$$

This operation will succeed if P1' and P' are deemed valid.

If format checking is applied to ensure that all PIN digits $P' = P'_1P'_2...P'_L$ are valid (i.e. $P'_i < 10$), then for i = 1 .. (L-2), the call will fail if $P_i \oplus x \geq 10$. This yields the oracle $h(P_{i+2}, 10)$ for i = 1 .. (L-2)! If format checking is applied to ensure that all the padding digits in P1' are correct, then i = (L-1)..12, the call will fail. If we set x = 1, then for i = 1..L-2, the call must pass by our lemma, while for i = L-1..12, the call will fail. This yields oracle $l(PB_{i+4}, F)$. This leads to the following PIN recovery attack.

6.4. ANSI X9.8 PIN Length Determination

Algorithm: ANSI X9.8 PIN Length Determination

1. Since the minimum value of L is 4 and the maximum 12, we iterate i = 3..11
   1.1. If $l(PB_{i+4}, F)$ is *true* then return L = i + 1.

The running time of the algorithm is L-3.

6.5. ANSI X9.8 (Partial) PIN Recovery attack

Algorithm: ANSI X9.8 (Partial) PIN Recovery attack

1. For i = 1 .. L-2
   1.1. Use the efficient algorithm to determine P_{i+2}, $P_{i+2} \oplus 1$ using the oracle $h(P_{i+2}, 10)$.

The running time of the algorithm is 3(L-2). The algorithm does not recover any information about the first 2 digits of the PIN, but identifies the remaining L-2 digits as one of 2 candidates. In total, we have reduced the PIN space from 10^L to $10^2 \bullet 2^{L-2}$. An important point is that the attack only makes use of the PAN Casting technique.

6.6. ANSI X9.8 (Extended) PIN Recovery attack

There is an obvious extension to expose the first two digits of the pin. We simply extend the PIN by 2 digits to the left using the

ANSI $X9.8_{PAN=0}$: FC(VISA-3) $\rightarrow$ ANSI $X9.8_{PAN = 0}$

operation. The original first 2 digits P_1P_2, have been 'shifted' to vulnerable positions in the new PIN P' = 0L'$P_1P_2..P_L$ where L' = L+2. This allows us to reduce the pin space to 2^L. This attack now relies on both the PC and FC techniques. Some extra manipulation is required for PINs of length $L \geq 9$.

6.7. PIN Recovery attack without consistency checking

6.7.1. Realization of the oracle $l(PB_{i+4}, F)$

Using the translate function without consistency checking.

But what if the implementation does not enforce consistency checking? Consider the sequence

ANSI $X9.8_{PAN}$: PC(ANSI $X9.8_{PAN \oplus A\_(12, i, 1)}$) $\rightarrow$ VISA-3
VISA-3 $\rightarrow$ ANSI $X9.8_{PAN}$

Without consistency checking, the first call will always succeed yielding P' = P $\oplus$ A(L, i+2, 1) if $i \leq L-2$; otherwise P' = P. Note that regardless of the value of i, P' is a valid PIN (since $P_{i+2} < 10$, by our lemma $P_{i+2} \oplus 1 < 10$). If i > L-2, then the PIN is unchanged and so the final encrypted PIN block will be identical to the original one. If $i \leq L-2$, then the final encrypted must differ since the $P' \neq P$. Thus we obtain another realization of $l(PB_{i+4}, F)$.

6.7.2. Realization of the oracle $g(P_{i+2}, F)$

Consider the sequence,

ANSI $X9.8_{PAN}$: PC(ANSI $X9.8_{PAN \oplus A (12, i, x)}$) $\rightarrow$ VISA-3
VISA-3 $\rightarrow$ ANSI $X9.8_{PAN \oplus A(12, i, x)}$

Again we assume no consistency checking, ensuring the first call will always succeed. For simplicity we assume $2 < i \leq -2$ and so P' = P $\oplus$ A(L, i+2, x), $P'_{i+2} = P_{i+2} \oplus x$. If $x = P_{i+2} \oplus F$, then $P'_{i+2} = F$ which is viewed as a delimiter and is used to identify the end of the PIN for the VISA-3 format. This reduces the PIN to length L' = 2 + i. This can be identified either by recalculating the PIN length or simply reformatting the result back to it's former format and checking if the former and resultant encrypted PIN blocks are identical (if the PIN length has changed, they will differ).
For i < 2 and $P'_{i+2} = F$, the implementation should fail the operation (hence be easily identifiable) since the PIN length is less than the minimum of 4 (however a particular implementation may behave differently). Nonetheless, given our ability to extend the PIN length, we can overcome the potential obstacle. In the end, we have obtained the oracle $g(P_{i+2}, F)$.

6.8. PIN Recovery attack with consistency checking(2)

6.8.1. Realization of the oracle $g(P_{i+2}, F)$

Let $A^*(l, i, x) = A(l,i,x) \oplus A(l,i+1,P_{i+2+1} \oplus F) \oplus \ldots \oplus A(l,L-2,P_L \oplus F)$ for $i < L$, else $A^*(l, i, x) = 0$.

ANSI X9.8$_{PAN = 0}$ $\rightarrow$ ANSI X9.8$_{A^*(12,i,x)}$
ANSI X9.8$_{A^*(12,i,x)}$: FC(VISA-3) $\rightarrow$ ANY

The first call modifies the clear PIN block yielding

$PB' = 0LP_1..P_{i+1}P'_{i+2}F..FF,$

$P'_{i+2} = P_{i+2} \oplus x,$

which is then interpreted as a VISA-3 format PIN block. In order for the consistency checking to pass, $L < 10$ and P'_{i+2} must be valid. Note that if $x = P_{i+2} \oplus F$, then $P'_{i+2} = P_{i+2} \oplus x = F$ is valid. It also has the side affect of changing the length of the PIN to one less than if this were not the case. This change can easily be detected by our earlier methods. Thus we obtain an oracle for $g(P_i, F)$.

Algorithm: PIN Recovery attack with consistency checking

1. For i = 2 .. L
   1.1. For each possible value j = 0..9
      1.1.1. Submit $j \oplus F$ to oracle $g(P_i, F)$.
      1.1.2. $P_i = j$ iff the oracle returns *true*.

7. Other Attacks

These are attacks against weak algorithms or poor implementations.

7.1. The Check Value Attack Against Offsets

The check value function encrypts a 64 bit binary zero under the supplied key. The PIN verify function (for IBM and GBP algorithms) encrypts a 64 bit user supplied validation data. The resulting ciphertext is decimalized via a user supplied decimalization table and termed the intermediate PIN (IPIN). The offset is the result of subtracting the IPIN from the customer selected PIN modulo 10 (OFFSET = PIN - IPIN modulo 10). The key observation is the similarity in operation between the check value function and the verification function.

Consider the case when the validation data is a 64 bit binary zero. The result of the encryption stage is the same as the result from the check value call (or more accurately, the first n bytes are the same for a n byte check value). The decimalization table is known, so it is trivial to calculate the IPIN value. It is also possible to recover the value of the offset by exhaustive search (again using the verify function). With knowledge of both the IPIN and offset, it is a trivial calculation to determine the PIN.

Algorithm:

1. Calculate check value of key
2. Decimalize the check value and store as IPIN
3. Search for the OFFSET (with validation data = 000000000000000)
4. Return PIN = IPIN + OFFSET

Example

```
PIN                  1234
PIN ver key          0505050505050505

Validation data      0000000000000000
Decimalization Table 0123456789012345
OFFSET(searched)     3034

Check Value          8CA0964
IPIN                 8200 (= the check value decimalized)

Calculated PIN       1234 (= OFFSET + IPIN)
```

A few observations. The attack (as always) accepts an encrypted PIN block and its associated (encrypted) PIN encrypting key. The actual value of the PIN verification key is not important, hence one can merely conjure the key (if possible) or use any other PIN verification key in the system. The attack is fairly efficient, requiring only a single search for the offset (10000 + (n-4) • 10 queries).

7.2. The Decimalization Data Attack against Offsets

Since the user supplies the decimalization table to the call, it is possible to repeatedly query the target with modifications to the table. Suppose we know the offset for a given PIN block (using a given decimalization table). Consider the effect of changing a single element in the table (the i[th] digit in the table mapping $\{i \rightarrow j_i,\ i \in Z_{16},\ j_i \in Z_{10}\}$.

If the hexadecimal digit i was not found in the first n digits of ciphertext, then there is no change in the value of IPIN, and the same value of the offset will pass the verify call. However, for each instance of i in the ciphertext, the corresponding digit of IPIN will be remapped to j_i'. The original offset will now fail the verify call. Using this approach we can identify the possible values of the hexadecimal digits in the first n digits of the ciphertext.

This technique can be further strengthen by setting $j_i' = j_i + k$ where k is a known (non zero) value (addition is modulo 10). Due to the simple relationship between the offset and the IPIN, we know that by adding k to a digit in the IPIN, the corresponding digit in the offset is reduced by k. Thus we can search through all possible ciphertext digit locations that contain i, by modifying the offset value and supplying the modified offset and decimalization table to the verify function. After at most 2^n queries, we will have identified all digit locations in the ciphertext with the value i. By repeating through all possible values of i, we can uniquely determine the value of the first n digits of ciphertext and thus IPIN. Again since we know the offset and IPIN, we can trivially calculate the value of the PIN.

Algorithm:

1. Search for offset (decimalization table is $i \rightarrow i \bmod 10$)
2. For i = 0..15
   - 2.1. Replace the entry $i \rightarrow i \bmod 10$, with $i \rightarrow i + k \bmod 10$
   - 2.2. For each possible location of i in the ciphertext (including none).
     - 2.2.1. Test the corresponding modified offset.
     - 2.2.2. If 'pass', then store locations of i in ciphertext.
3. Decimalize the 'recovered' ciphertext and store as IPIN
4. Return PIN = IPIN + OFFSET

Example:

```
PIN                  6598
PIN ver key          0505050505050505
Validation data      1122334455667788
Decimalization Table 0123456789012345

Ciphertext           E481FC5658391418
IPIN                 4481
OFFSET               2117

Modification 1
Dec Table (0)        1123456789012345
IPIN                 4481
OFFSET               2117 (will pass)
Hence the digit '0' is not found in the first four digits of the ciphertext.

Modification 2
Dec Table (1)        0223456789012345
IPIN                 4482
OFFSET               2116 (will pass)
Thus we have identified that the fourth digit of the ciphertext is 1.
```

Initially we require the value of the offset (requiring 1 search or 10000 + (n-4) • 10 queries). The rest of the algorithm requires at most 16 • 2^n queries. The same comments apply to the PIN verification key as in the previous attack.

8. Key Separation Attacks

As mentioned before, the principle of key separation between generic key types (e.g. data and PIN keys) is understood and implemented by many API's. However, the granularity of separation required, is not understood. We show attacks against a surprisingly fine level of granularity.

We describe IBM's naming (and separation) of PIN keys, since they offer the highest level of granularity of the commercial API's investigated by the authors. This naming (and separation) is also consistent across the IBM ICSF, IBM TSS and IBM *CCA* API's - all of which use control vectors to achieve separation.

- A *PINGEN* key, is a PIN generating key, and is the only key type allowed for use in the generate PIN function
- A *PINVER* key, is a PIN verifying key, and is the only key type allowed for use in performing the verification in the verify function
- An *IPINENC* key, is a input PIN encrypting key. Encrypted PIN blocks supplied to any of the PIN functions, are encrypted under a key of this type.
- An *OPINENC* key, is a output PIN encrypting key. The output encrypted PIN blocks from any of the PIN function, are encrypted under a key of this type.

More details can be found in the IBM manuals.

8.1. Exhaustive PIN search and Code Book Attacks based on the failure to separate PINGEN/PINVER and IPINENC/OPINENC keys

We describe a exhaustive PIN search attack exploits a lack of separation between PINGEN/PINVER and IPINENC/OPINENC keys to perform the search. Since we have no separation, we may interchange the use of the pin encrypting key and the pin verification key. We wish to recover the PIN (P), encrypted under some key k. We have the associated encrypted PIN block EPB and the encrypted key k.

Our first step, is to translate the EPB to ANSIX9.8 with PAN = 000000088888. The clear PIN block is of the form PB = $04P_1P_2P_3P_4$FFFFF77777. Let $P^i = P^i{}_1P^i{}_2P^i{}_3P^i{}_4$ be a guess for P (for simplicity of notation, we have used a 4 digit pin). We format a 16 hexadecimal validation data string as VAL = 04 $P^i{}_1P^i{}_2P^i{}_3P^i{}_4$FFFFF77777. We now use this validation data as input to say the IBM Pin Generation algorithm using the encrypted key k. The validation string is encrypted under the clear value of k, decimalized and the first 4 digits returned as the generated PIN (P_{gen}). We simply compare the first 4 decimalized digits of our translated encrypted PIN block to P_{gen}. If P^i = P, then the values will be equal. We can expect multiple collisions due to the decimalization process and the fact that we only have 4 digits to compare. However, we can eliminate false witnesses by repeating the process and modifying the validation data and the PAN (e.g. use PAN = 000000088887 and VAL = $04P_1P_2P_3P_4$FFFFF77778). Simply by iterating through all possible values of P^i we can identify P. Alternatively, we can build up a code book.

One may question why we didn't use a null PAN, and format the validation data as VAL = $04P_1P_2P_3P_4$FFFFFFFFFF. The reason lies in the fact that some API's take as input, effectively a 11 hexadecimal digit validation string, and extract the last 5 digits from the rightmost 5 digits of the PAN. Other API's require the 12^{th} digit to specify a key index in the range 1 to 6. The description above, will satisfy that the addition requirement that the PANs supplied to generate and translate functions consists of decimal digits (for the above description, the generate call uses a PAN = 000000077777).

A limitation of this attack, is that it requires access to the generate function. However, as mentioned before, this is a security risk in itself, usually protected against by restricted its usage to a secure or authorized mode, or alternatively, encrypting the output (or requiring encrypted inputs). If the output is encrypted, then we must obtain the first 4 decimalized digits of the encrypted PIN block as an encrypted PIN for comparison. This in itself is perhaps not a strenuous requirement. However, there is a more powerful version of the attack, using the verify function (which is unlikely to have access control restriction). We proceed as before, generating the validation as before, but supplying it to the verification function (using offsets) and supplying the encrypted pin block. We require one extra piece of information, namely the offset. If P^i = P, the validation same is identical to the clear pin block as before, the intermediate PIN will equal the first 4 decimalised digits of the encrypted PIN block (which is known). We denote this value by IPIN.
The offset is the result of the intermediate PIN subtracted the clear PIN modulo 10. Hence the offset will equal will equal the sum IPIN and P^i modulo 10. Thus for each value P^i we calculate OFFSET = P^i - IPIN and supply it to the verification call. If P^i = P, the call will pass. False witnesses can be eliminated as before.

8.2. Exhaustive PIN search based on failure to separate between PINVER keys for different verification algorithms

In this attack, we play two different verification algorithms against each other. This is a fascinating result since it shows that taking two (potentially) individually secure verification functions, and allowing both in a single API can destroy it's security. It serves as a warning for designers (and implementers) of API's against blindly adding functionality, even though the function may be secure on its own. It also shows the need for extremely fine granularity of separation of keys associated with different functions, even though the functions are of similar nature.

The VISA PVV algorithm encrypts the concatenation of the 11 digit transformed security parameter (TSP), a key index (a digit in the range 1 to 6) and the first 4 digits of the PIN extracted from the encrypted PIN block (i.e. TSP || Key Index || PIN). The result is decimalised in a unique way. Scanning from left to right, the first 4 decimal digits encountered are returned as the PVV. Should there be less than 4 digits, a second scan is performed, in which the non decimal digits are converted to decimal digits (by subtraction modulo 10), and the first 4 resulting digits returned as the PVV.

The probability that the first 4 digits are indeed all decimal digits (and hence form the PVV) is ($\left(\frac{10}{16}\right)^4 = 0.153$). Thus for a given key, by trying 7 different values for the TSP, we can expect one to exhibit this property.

Consider now the result of supplying the value (TSP || Key Index || PIN) as validation data to the (IBM) offset algorithm under the same verification key and using a decimalization table that maps $n \rightarrow n \bmod 10$. The first 4 digits recovered from the encryption are the same as the PVV, and unchanged by the IBM decimalization process (i.e. IPIN = PVV). These 4 digits are subtracted from the clear value of the PIN to obtain the offset (or OFFSET = PIN - PVV).

It is possible to use this relationship to search for the PIN. By iterating through all possible value for the PIN (denoted P^i), and testing whether $OFFSET^i = P^i - PVV$, satisfies the verification algorithm with validation data (TSP || Key Index || P^i). As indicated earlier, the entire process needs to be repeated 7 times on average to witness the true value for the PIN. We can expect some false witnesses for the PIN but these can be eliminated by increasing the number of repetitions (say to 14 when we can expect the real PIN to have been witnessed twice).

Algorithm:

1. Loop i: (i = 1..7)
   - 1.1. TSP^i <= i || 1 || 1234
   - 1.2. Search for PVV^i
   - 1.3. Loop j: (j = 0...9999)
     - 1.3.1. PIN^j = j
     - 1.3.2. $OFFSET^{i,j} = PIN_j - PVV^i$
     - 1.3.3. $VAL\ DATA^{i,j}$ = i || 1 || j
     - 1.3.4. Test if $OFFSET^{i,j}$ verifies the PIN, if so then j is a candidate.

Example:

```
PIN           1234
Verification Key (K) 0505050505050505

i = 1:
TSP1            1000000000011234
E(K)(TSP1)      D84355EF089EF293
PVV1            8435
Will not correctly witness the PIN.

i = 2:
TSP2            2000000000011234
E(K)(TSP2)      9D15A1C0BE1DD129
PVV2            9151
Will not correctly witness the PIN.
```

```
i = 3:
PVV2            3000000000011234
E(K)(TSP3)      6787FA69080E3C93
PVV3            6787
Will correctly witness the PIN.

  j = 1
  VAL_DATA3,1    = 3000000000110001
  OFFSET3,1      = 4324
  PIN_verification call returned FALSE

  ...

  j = 1234
  VAL_DATA3,1    = 300000000011234
  OFFSET3,1      = 5557
  PIN verification call returned TRUE.  Store j = 1234 as candidate for the PIN.

  ...
```

Changing the decimalization table and calling the offset verify function again can eliminate some false witnesses (the digits 0 to 9 must remain mapped to themselves). E.g. {$n \rightarrow n$ ($n < 10$), else $n \rightarrow n+1 \bmod 10$ ($n \geq 10$).

9. References

1. R.J. Anderson, "Security Engineering - A Guide to Building Dependable Distributed Systems", Wiley, 2001
2. R.J. Anderson, "The Correctness of Crypto Transaction Sets", Security Protocols – 8th International Workshop, Springer-Verlag, 2000
3. R.J. Anderson and M. Bond, "API-Level Attacks on Embedded Systems", IEEE Computer Magazine October 2001, 2001, pp 67-75
4. M. Bond, "Attacks on Cryptoprocessor Transactions Sets", Cryptographic Hardware and Embedded Systems CHES 2001 Third International Workshop, Springer-Verlag , 2001, pp 220-234
5. CCA API Release 2.41 available at http://www-3.ibm.com/security/cryptocards/html/release241.shtml

10. Appendix

A Standard Financial PIN Transaction Set

10.1. PIN Block Formats

We briefly describe a subset of the common PIN block formats that are used later. The notation and descriptions are reproduced from [5] and included here for completeness.

PIN Notation

```
P = A 4-bit decimal digit that is one digit of the PIN value.
C = A 4-bit control value. The valid values are X'0' and X'1'.
L = A 4-bit hexadecimal digit that specifies the number of PIN digits.
F = A 4-bit field delimiter of value X'F'.
f = A 4-bit delimiter filler that is either P or F, depending on the
    length of the PIN.
D = A 4-bit decimal padding value. All pad digits in the PIN block
    have the same value.
X = A 4-bit hexadecimal padding value. All pad digits in the PIN block
    have the same value.
x = A 4-bit hexadecimal filler that is either P or X, depending on the
    length of the PIN.
R = A 4-bit hexadecimal random digit. The sequence of R digits can each
    take a different value.
r = A 4-bit random filler that is either P or R, depending on the
    length of the PIN.
Z = A 4-bit hexadecimal zero (X'0').
z = A 4-bit zero filler that is either P or Z, depending on the
    length of the PIN.
S = A 4-bit hexadecimal digit that constitutes one digit of a sequence
    number.
A = A 4-bit decimal digit that constitutes one digit of a user-specified
    constant.
```

ANSI X9.8

```
 (ISO format 0, VISA  format 1, VISA format 4, ECI format 1)

P1 = CLPPPPffffffffFF
P2 = ZZZZAAAAAAAAAAAA
PIN Block (PB) = P1 XOR P2
where C = X'0' and L = X'4' to X'C'
```

ISO Format 1 `(ECI format 4)`

```
PIN Block (PB) = CLPPPPrrrrrrrrRR
where C = X'1' and L = X'4' to X'C'
```

VISA Format 2

```
PIN Block (PB) = LPPPPzzDDDDDDDDD
where L = X'4' to X'6'
```

VISA Format 3

```
This format specifies that the PIN length can be 4-12 digits inclusive.
The PIN start from the left most digit and ends by the delimeter ('F').
An example of a 6 digit PIN

PIN Block (PB) = PPPPPPFXXXXXXXXX
```

IBM 3621 Format

```
This format requires the program to specify the delimiter, X, for
determining the PIN length.

PIN Block (PB) = SSSSPPPPxxxxxxxx
```

ECI Format 2

```
This format defines the PIN to be 4 digits.

PIN Block (PB) = PPPPRRRRRRRRRRRR
```

10.2. Functions

A given financial API may have many PIN related commands in its transaction set. There are however, three functions, which form the core a PIN, based system. These are:

- PIN generation (the process of generating a PIN)
- PIN verification (the process of verifying that the PIN contained in an encrypted PIN block is the 'correct' PIN for a given account holder)
- PIN translation (the process of translating an encrypted PIN block between PIN formats and encrypting keys)

10.3. Algorithms

PIN generation can be achieved in a variety of ways, either by means of an algorithm or chosen by a user (or both). An example of the algorithmic approach is the IBM 3624 PIN Generation Algorithm, which generates a PIN based on account- or person-related data, called the validation data. The validation data is enciphered under a PIN generating key, decimalised and the desired number of digits selected as the PIN.

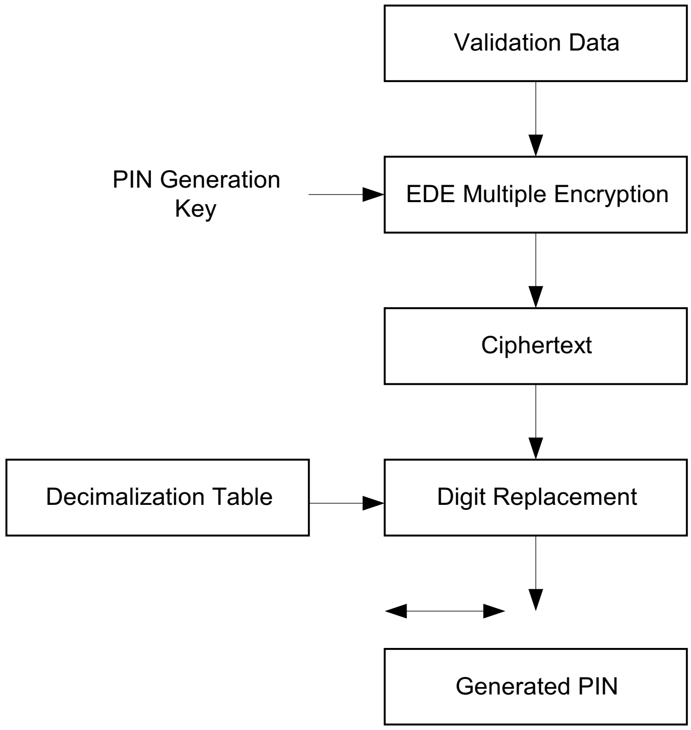

Figure 10-1PIN Generation Algorithm

Verification is achieved by repeating the process (except that now the key may be called the PIN verification key) and comparing the calculated PIN with the PIN that is extracted from the encrypted PIN block.

Generating algorithms can be extended to cater for chosen PINs. This is achieved through the use of offsets, which relate the algorithm generated PIN to the chosen PIN. The normal generation algorithm is run (with the same parameters) and the output called an intermediate PIN. The offset is calculated by subtracting modulo 10 the chosen PIN digits from some subset of the intermediate PIN digits (usually either the leftmost or rightmost digits).

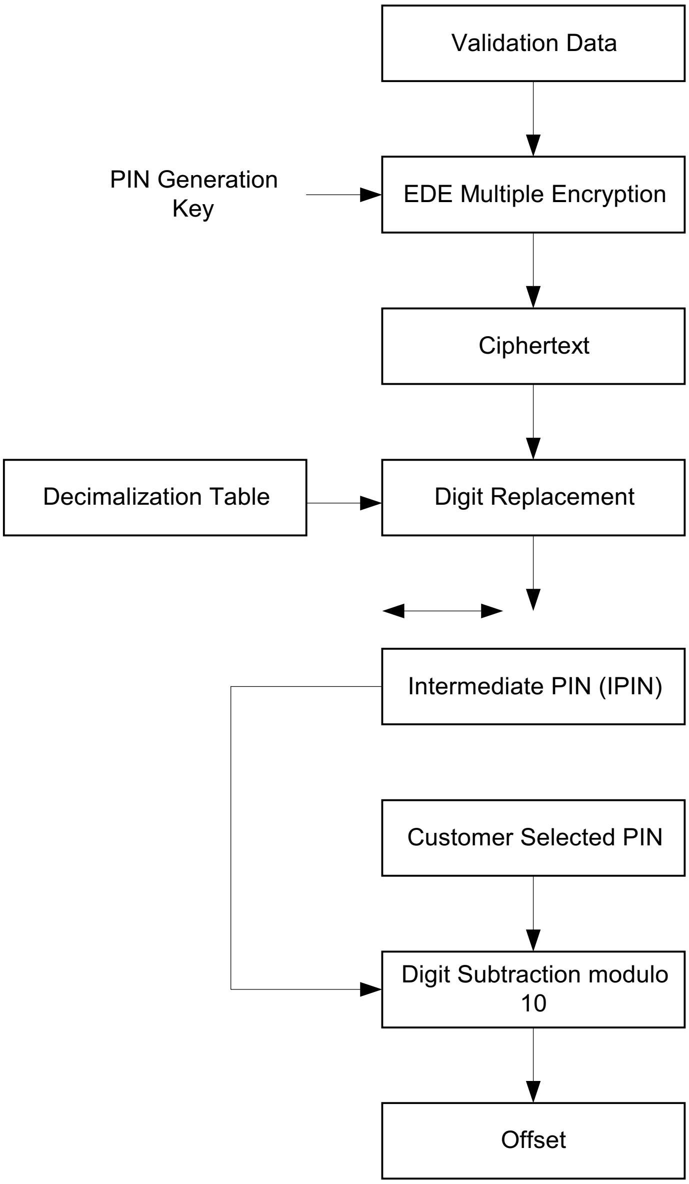

Figure 10-2PIN Offset Generation Algorithm

Verification requires that the offset to be supplied to the call as well.

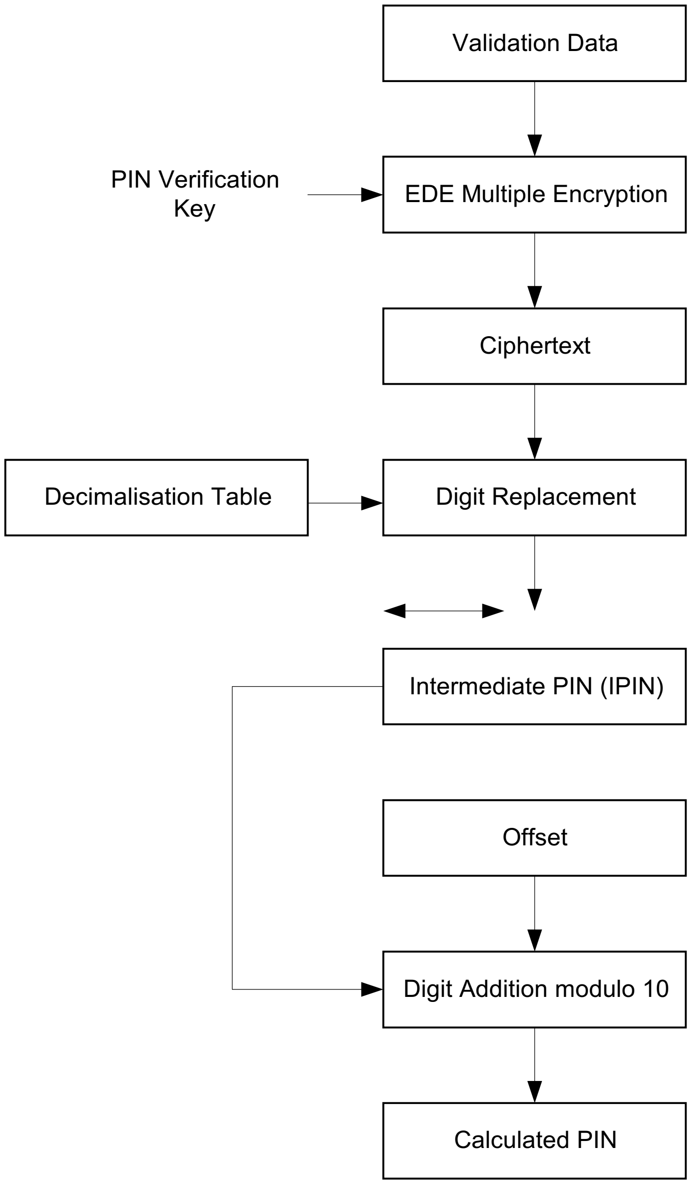

Figure 10-3PIN Verification Algorithm

For a chosen PIN, the PIN generation algorithm accepts a PIN (either clear or encrypted) as input and must calculate a PIN verification value (PVV), which is stored

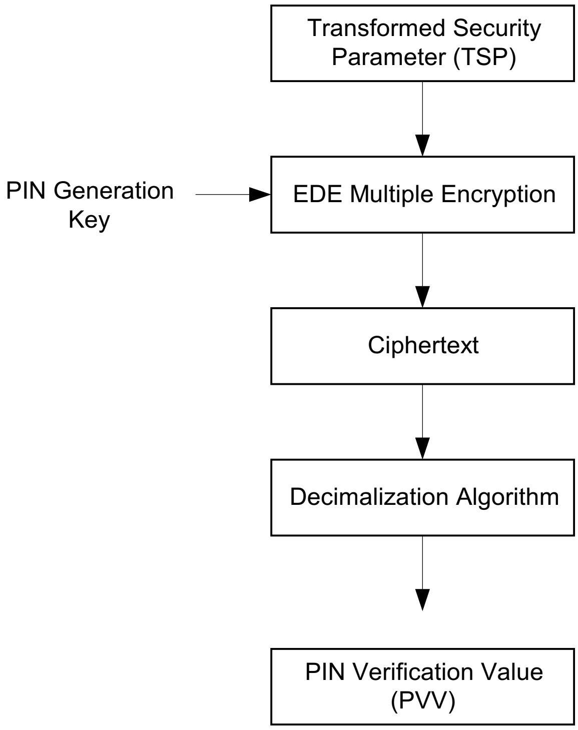

Figure 10-4PVV Generation Algorithm

Verification involves extracting the PIN, calculating the verification value and comparing it to a supplied PVV. An example is the VISA PIN verification algorithm.

Finally, the translate PIN function extracts the PIN from an encrypted PIN block according to the rules of the PIN block's format. The function then reformats it into the requested target format and encrypts under the target PIN encrypting key.