In Proceedings of *14th Nordic Conference on Secure IT Systems* (NordSec'09)
Springer LNCS 5838

Blunting Differential Attacks on PIN Processing APIs*

Riccardo Focardi[1], Flaminia L. Luccio[1], and Graham Steel[2]

[1] Università Ca' Foscari Venezia,
{focardi,luccio}@dsi.unive.it
[2] Laboratoire Spécification et Vérification, CNRS & INRIA & ENS de Cachan,
Graham.Steel@inria.fr

Abstract. We propose a countermeasure for a class of known attacks on the PIN processing API used in the ATM (cash machine) network. This API controls access to the tamper-resistant Hardware Security Modules where PIN encryption, decryption and verification takes place. The attacks are differential attacks, whereby an attacker gains information about the plaintext values of encrypted customer PINs by making changes to the non-confidential inputs to a command. Our proposed fix adds an integrity check to the parameters passed to the command. It is novel in that it involves very little change to the existing ATM network infrastructure.

Keywords: Security APIs, Financial Cryptography, PIN Verification.

1 Introduction

In the international ATM (cash machine) network, users' personal identification numbers (PINs) have to be sent encrypted from the PIN entry device (PED) on the terminal to the issuing bank for checking. Issuing banks cannot expect to securely share secret keys with every cash machine, and so the PIN is encrypted under various different keys as it passes through the network. Typically, it will first be encrypted in the PED under a key shared with the server or *switch* to which the ATM is connected. The PIN is then decrypted and re-encrypted under the key for an adjacent switch, to which it is forwarded. Eventually, the PIN reaches a switch adjacent to the issuing bank, by which time it may have been decrypted and re-encrypted several times. The issuing bank has no direct control over what happens in the intermediate switches, so to establish trust, the internationally agreed standards ANSI X9.8 and ISO 9564 stipulate the use of tamper proof cryptographic hardware security modules (HSMs). In the switches, these HSMs protect the PIN encryption keys, while in the issuing banks, they also protect the PIN derivation keys (PDKs) used to derive the customer's PIN from non-secret validation data such as their personal account number (PAN). All encryption, decryption and checking of PINs is carried out inside the HSMs.

* Work partially supported by Miur'07 Project SOFT: "*Security Oriented Formal Techniques*".

The HSMs have a carefully designed API providing functions for *translation* (i.e. decryption under one key and encryption under another) and *verification* (i.e. PIN correctness checking). The API has to be designed so that even if an attacker obtains access to the host machine connected to the HSM, he cannot abuse the API to obtain customer PINs.

In the last few years, several attacks have been published on the APIs in use in these systems [7, 10, 12]. Very few of these attacks directly reveal the PIN. Instead, they involve the attacker calling the API commands repeatedly with slightly different parameter values, and using the results (which may be error codes) to deduce the value of the PIN. High-profile instances of many PINs being stolen from a hacked switch has increased interest in the problem [1]. In April 2009, a Verizon Data Breach report and a subsequent interview with the report's author in the press confirmed publicly for the first time that PINs are being extracted from HSMs on a wide scale, and that organised criminal gangs are actually implementing attacks "that a year ago were thought to be only academically possible" [2, 3].

We are concerned with researching improvements to the PIN processing infrastructure that could mitigate these attacks. PIN recovery attacks have been formally analysed before, but previously the approach was to take a particular API configuration and measure its vulnerability to combinations of known attacks [16]. In recent work, we proposed an extension to language based information flow analysis to take account of cryptographic primitives designed to assure data integrity, in particular MACs [11]. We showed how PIN processing APIs could be extended with MACs to counteract differential attacks, and showed how this revised API type-checked under our framework. However, that work was rather theoretical, and did not attempt to explain how our proposal could be put into practice. In particular, for our proposal to be feasible in the short term, it needs to be adapted to take into account the constraints of the existing ATM infrastructure. In this paper we outline what we believe to be a practical scheme. We assess its impact on security and the amount of changes that would be required to the ATM network to put it into practice.

The rest of this report proceeds as follows: in section 2 we first review the PIN processing APIs we are interested in, and then in section 3 the known attacks. In section 4 we detail our fix, show how it improves security, and discuss the implementation of our proposal. In section 5 we compare our scheme to other proposals in the literature, we then conclude in section 6.

2 Background

In this section we first introduce the architectural model and we then revise some of the security APIs used for financial *Personal Identification Number* (PIN) verification. We focus on the ones that have been shown to be critical for PIN security when the HSM is in *operational mode*, i.e. when it is processing normal transactions from ATMs. Most HSMs support a further mode of operation such as *privileged mode* or *security officer login* which is used, e.g., for loading new

cryptographic keys, but generally this mode requires physical access to the HSM in order to enter codes on a keypad or present smartcards. Attacks at this level are considered out of scope for this paper.

2.1 The Architecture

Let us recall the ATM network structure [12]: the customer inserts the card and keys in the related PIN at and ATM of the *acquiring* bank. The PIN has to travel to the *issuing* bank[3], i.e. the bank where the customer has the account. Most of the time the acquiring bank and the issuing bank do not coincide, and may be located at great distance. In this case the PIN has to travel along a network of intermediary switches, encrypted via different symmetric keys: the issuing bank shares a key with the first neighbouring switch, this switch shares another key with the next switch, and so on. Thus, messages arriving at a switch have to be decrypted and re-encrypted with the new key. When messages containing the PIN arrive at the issuing bank, this bank has to confirm the association between the PIN and the *Personal Account Number* (PAN), i.e. an identifying account number associated with each users account.

2.2 PIN Management APIs

In the previous section we have observed how PINs travelling along the network have to be decrypted and re-encrypted under a different key: This is done using a *translation* API. There is a further complication however: PINs have to be formatted into 64-bit blocks for encryption under the 3DES cipher used in the network. Different countries, manufacturers, and banking organisations have proposed different formats for the block, some of which are now standardised in ISO 9564. The different block formats reflect the different capabilities of different nodes in the network (for example, some require the encrypting device to generate a fresh random number). Not all switches are able (or willing) to support all known PIN formats. Thus, the translation function might need to reformat messages under newer or older formats accepted by the next switch. In some cases, e.g., when the output format is ISO-0, the PAN related to the PIN has also to be specified. When the PIN reaches the issuing bank, its correspondence with the PAN is checked via a *verification* API. Even at the verification API, several different PIN block formats might be supported.

ISO-1 and ISO-0 formats. We now illustrate ISO-1 and ISO-0 formats, as we will often refer to them when describing the attacks. We report all the other commonly used formats in appendix A.

[3] Chip based cards now support the possibility of checking the PIN offline, i.e. checking it on the chip. However, under most schemes, this is intended for point-of-sales transactions. For cash advances and ATM transactions, an online PIN check is still required. Furthermore, large countries like the USA do not have chip-based card schemes in widespread use.

ISO-1 supports PINs from 4 to 12 digits in length. The formatted PIN appears as follows:

1	L	P	P	P	P	P/R	P/R	P/R	P/R	P/R	P/R	P/R	P/R	R	R

In particular, 1 identifies the ISO-1 format, L is the length of the PIN, P are the PIN digits while P/R is either a PIN digit or the pad random value R, depending on L. The random padding aims at avoiding codebook attacks.

ISO-0 is equivalent to ANSI X9.8, VISA-1, and ECI-1. This format is not randomized but uses the PAN to prevent codebook attacks. The 4-12 digits PIN is first formatted as follows:

0	L	P	P	P	P	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	F	F

0 stands for ISO-0, L is the length of the PIN, P are the PIN digits while P/F is either a PIN digit or the pad value F, depending on L. To obtain ISO-0, first the rightmost 12 digits of the PAN are written as follows:

0	0	0	0	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN

and then the two numbers above are xored:

0	L	P	P	P xor PAN	P xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	F xor PAN	F xor PAN

Translate and Verify API. We now specify more in detail the two APIs.

- PIN_Translate(k_I, k_O, $format_I$, $format_O$, PAN_I, PAN_O, EPB_I, EPB_O)

  This API takes a PIN block EPB_I encrypted with the key referred by k_I and formatted as $format_I$, extracts the PIN, reformats it as $format_O$ and re-encrypts it with key referred by k_O. When one of the two formats is ISO-0, the PAN is also provided via PAN_I or PAN_O.
  When the API succeeds, the obtained PIN block is returned in EPB_O. Otherwise, an error code is returned.

- PIN_Verify(k_I, k_V, $format_I$, PAN_I, EPB_I, v_data)

  This API takes a PIN block EPB_I encrypted with the key referred by k_I and formatted as $format_I$, extracts the PIN and checks its validity via the verification key referred by k_V and the verification data v_data. For example, in the IBM 3624 PIN calculation method [13], v_data is the triple ⟨offset, validation_data, dec_tab⟩: a user PIN of length[4] l is obtained by decimalising through dec_tab the first l digits of the encryption $enc(k_V, \text{validation\_data})$, and by digit-wise summing modulo 10 the offset. Once the user PIN is recovered, its equality is checked against the received encrypted PIN, i.e. the one inserted at the ATM.
  The API returns the result of the verification or an error code.

[4] The PIN length is either encoded in the format or passed as a further parameter, as mentioned below.

The above described APIs are slight abstractions and a simplifications of real ones. For example, referring to [13], we have that PIN_Translate(k_I, k_O, $format_I$, $format_O$, PAN_I, PAN_O, EPB_I, EPB_O) corresponds to Encrypted_PIN_Translate except that we have omitted parameters rule_array_count, rule_array, and sequence_number. The first two are used to specify whether the API should only *translate* or *translate-and-reformat* the encrypted PIN, and in the latter case a PIN extraction method has to be specified. Here, we are assuming the API always performs a *translate-and-reformat*, but we can recover the simpler *translate* case, e.g., by just passing NULL in the $format_O$ parameter. Finally, the parameter sequence_number is not used and always points to a constant.

PIN_Verify(k_I, k_V, $format_I$, PAN_I, EPB_I, v_data) corresponds to Encrypted_PIN_Verify, where we have omitted the parameters rule_array_count, rule_array, described above, and PIN_check_length, which is used to specify the length of the PIN for the format IBM 3624.

3 Known Attacks

In this section, referring to [7, 10, 12, 14], we give a brief overview of known attacks on cryptographic APIs. We will omit some of the details, mainly concentrating on the specification of the API calls involved. All the attacks have the same scenario: an attacker has obtained access to a host machine at a switch or verification facility, and is able to call any function of the API with any parameters. Additionally he is assumed to be able to intercept some genuine traffic arriving for processing at the HSM (i.e. encrypted PIN blocks and associated data from real ATM transactions). However, he is assumed to be *unable* to obtain, by brute force search or other means, the PIN encryption keys or PIN derivation keys in use in the system.

3.1 Attacks on Verification API

We now look at attacks on the verification API, which we split into two categories.

Decimalisation Table Attacks. This family of attacks was discovered by both Bond and Zieliński, [10], and Clulow, [12, §3.5.5]. Since a proposal by IBM (the so-called '3624 scheme'), many PIN schemes assign initial customer PIN values by encrypting a customer's PAN under a secret PIN derivation key (PDK), and then decimalising the result using a decimalisation table (or 'dectab'). A decimalisation table maps each hexadecimal value to a decimal. The 'standard' decimalisation table looks like this:

Hex. value	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dec. value	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5

So, if the first four digits of the result of encrypting a customer's PAN under the PDK are 4A6B hexadecimal, the assigned PIN will be 4061. Not all banks

use the same decimalisation table [6], so the table is passed as a parameter to the verify command. Some schemes allow the customer to change her PIN at an ATM. This is achieved by fixing an offset, which when added digitwise modulo 10 to the original PIN, gives the customer's chosen PIN. This offset is not considered to be security critical, since without the original PIN it provides no help in guessing the correct customer PIN.

Decimalisation table attacks do not determine the PIN digitwise, but rather determine first what digits are in the PIN, and then where these digits are. Suppose an attacker has an encrypted PIN block which, when supplied along with the standard decimalisation table and a known offset, correctly verifies inside the HSM (that is, the HSM reports that the PIN is correct when the PIN_Verify(.) function is called). Now suppose the attacker alters the decimalisation table like this:

Old value	0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
New value	1 1 2 3 4 5 6 7 8 9 1 1 2 3 4 5

He then calls PIN_Verify(.) again, with the modified dectab. If the verification still passes, then he knows there are no 0s in the PIN. If however the verification now fails, he knows there must be at least one 0 somewhere in the PIN. The problem now is to determine how many there are, and where. This can be accomplished by altering the offset. The attacker advances the offset by one at each position, and then at every combination of positions, until the PIN is once again reported as being correct. This reveals the location of the 0s in the PIN. The table below illustrates the process, for an example where the customer's PIN is 3060, and the offset is 0000. We assume the attacker has already tried the modified dectab shown above, and discovered that there is at least one 0 somewhere in the PIN.

Attacker set offset	Result from HSM	Knowledge of PIN
0001	Incorrect PIN	????
0010	Incorrect PIN	????
0100	Incorrect PIN	????
1000	Incorrect PIN	????
0011	Incorrect PIN	????
0101	Correct PIN	?0?0

The decimalisation table attack takes an average of 16.145 API calls to determine a four-digit PIN [16, §5].

Check Value Attack. Clulow has described an attack on the PIN verification API that takes advantage of the 'check value command' [12, §3.5.8]. Many APIs support such a command, used to ensure that a key has been imported correctly. The function of the command is to return a 64 bit block of 0s encrypted under a given key.

To make the attack, the attacker must also be able to supply a block of 0s as validation data to a command for verifying PINs such as the one described above. The first step is to obtain the check value of the PDK, and decimalise the

first four characters of the result using the standard decimalisation table. Store this as IPIN. Now, supply a PAN of 000000000000 to the function PIN_Verify(.), along with some encrypted PIN block (EPB) you want to crack. Start with offset 0000. Generally, the command will at first fail. Increase the offset by 1 until the command reports a successful verification. Store the final offset as OFFSET. Now we know that the PIN in the block verifies successfully when compared against IPIN + OFFSET (mod 10 each digit), and so this must be the customer's PIN.

On average, for a four-digit PIN, this attack would require one call to the check value command and 5000 calls to the PIN verify function.

3.2 Attacks on the Translate API

We now consider two different attacks on the translate API.

The codebook attack. This attack uses the translation API to get rid of variant information from the EPBs. This makes EPBs containing the same PIN equal, thus enabling standard codebook attacks. Notice that if EPBs contain random padding, as e.g. in the ISO-1 format, or if the PAN is xored to the PIN before it is encrypted, as happens for ISO-0, equal PINs in general produce different EPBs.

The attack can be applied both to switches and to verification facilities [7]. The attacker first generates all the EPBs for every possible PIN of a fixed length, in any ISO format. This can be done, e.g., by a network of attackers trying all of the 10000 PINs at different ATMs and intercepting the corresponding EPB at the hacked switch. A smarter method, described in [7], only requires 100 trials at the ATMs, but we do not describe it here for lack of space. Once the 10000 EPBs have been obtained, the attackers translate, at the switch where they have been intercepted, all these EPBs into ISO-0 format with a chosen fixed PAN, say PANA, and create a look-up table with all the EPBs and the associated PINs. At the arrival of real EPBs, they translate them into an ISO-1 format (removing the PAN) and then reformat them into an ISO-0 message together with the chosen PANA. The real PIN can now be revealed by searching this new encryption in the look-up table.

Wrong-format attack. This attack [12, 16] consists of translating a received encrypted message into an ISO-0 format with different PANs, and to acquire information depending on the return value of the translate API. More precisely, the attacker provides different PANs (PANA^i), which are xor-ed to the decrypted value to recover the PIN. When such an xor operation gives non-decimal values, the API returns an error code. This gives information about the actual PIN digits. In fact, for an n digits PIN, by repeating this operation $16(n-2)$ times it is possible to recover the $n-2$ rightmost PIN digits up to their parity. For example, for a 4 digits PIN we might discover that the third and fourth digits are 3 or 4 and 7 or 8, respectively. Notice that the first two digits cannot be recovered because they are not xor-ed with the PAN.

There is a trick to recover also the first two digits of the PIN: the attacker first translates a message in ISO-0 (where the PIN starts from the 3rd position) with a PAN of all 0s, and then claims it is in a VISA3 message format (that starts with the digits of the PIN ended by a sequence of Fs), and asks to translate it into ISO-0 with a PAN of all 0s. The result of this operation is that the PIN is shifted two positions to the right, thus permitting the attacker to use the previous attack on the entire PIN.

Finally, the choice between the pair of values can be narrowed with 4 more calls: Assume the pairs of values are $\{P_1, P_1+1\}$, $\{P_2, P_2+1\}$, $\{P_3, P_3+1\}$, $\{P_4, P_4+1\}$, i.e. $2^4 = 16$ possible PINs, the attacker can now pass a message in an ISO-0 format claiming is a VISA3, together with PAN $E_1 00000000000$, with $E_1 = E \oplus P_1$. If the HSM outputs an error, the right PIN digit value is P_1, else is $P_1 + 1$. In fact, the given PAN is xored to the PIN and if the PIN digit value is P_1 we obtain $P_1 \oplus E_1 = P_1 \oplus E \oplus P_1 = E$ which is non-decimal and gives an error. To recover the other values P_2, P_3 and P_4, the attacker provides PANs $0E_2 0000000000$, $00E_3 000000000$, $000E_4 00000000$, respectively, where $E_i = E \oplus P_i$.

Attacks on other APIs Attacks have also been found on the functions used for customer PIN change [9] and secure messaging [4]. These are out of scope for the current paper, but we plan to treat them in future work.

4 Our Proposed Fix

To understand the rationale behind our fix, first observe that the attacks explained in section 3 are 'differential attacks' in the sense that they involve an attacker learning about the value of the PIN by tweaking parameters to the API commands. The name comes from the similarity to differential attacks on block ciphers, where certain bits in the inputs to the cipher are changed and the result observed [8]. One way to prevent these attacks would be to prevent 'unauthorised' queries to the API. In theoretical terms, we are trying to achieve 'robust declassification' [15]. We know that the functions of the API must declassify some data, specifically they must tell us whether the encrypted PIN is the correct one or not. Robust declassification means an intruder cannot influence what data is declassified. However, the existing APIs have no way of distinguishing a legitimate query from an illegitimate one.

In another paper [11], we show how in general differential attacks can be countered by the use of MACs. The idea is to add a MAC of the non-confidential parameters required for PIN processing to the input to the relevant API command. The command checks the MAC before performing the PIN operation. If the MAC check fails, the command halts without processing the PIN. This way we achieve 'robust declassification'. However, the infrastructure changes needed to add full MAC calculation to ATMs and switches are seen as prohibitive by the banks [6]. We propose here a way to implement a weaker version of our scheme whilst minimising changes to the existing infrastructure. We lose some security,

since our MACs now have an entropy of only 5 decimal digits ($2^{16} < 10^5 < 2^{17}$). We assess the effect of this change in section 4.3.

4.1 CVC/CVV Codes

We observe that cards used in the cash machine network already have an integrity checking value: the card verification code (CVC), or card verification value (CVV), is a 5 (decimal) digit number included on the magnetic stripe of most cards in use. It is, in effect, a MAC of the customer's PAN, expiry date of the card and some other data. The current purpose of the CVV is to make copying cards more difficult, since the CVV is not printed on the card and does not appear on printed POS receipts[5]. Below we give the algorithm for calculating the CVV. This is done at the card issuing facility. CVVs are checked at the verification facility.

PAN	Exp date	Service code	0 pad
16 digits max	4 digits	3 digits	9 digits max
Block B1	Block B2		

Each decimal digit is encoded in 4 bits. Note the partition of the CVV plaintext field into two blocks, B1 and B2. To construct the CVV, a two-part DES key is required. Call the two 64-bit parts key K1 and key K2. The hexadecimal CVV root is constructed as

$$CVV_{hex} := enc(K1, dec(K2, enc(K1, (enc(K1, B1) \oplus B2))))$$

The 5 digit decimal number is constructed using the Visa decimalisation scheme:

1. Extract all decimal digits from CVV_{hex}, preserving their order from left to right.
2. Left justify the result
3. Reduce any remaining digits by 10
4. Left justify the result and append it to the result of 2
5. The CVV is the first 5 digits (from left to right) of the result.

4.2 Packing the MAC into the CVC/CVV

Our proposal is to pack more information into the CVV at issue time, and to use this as a MAC at the verification facility.

In our formal scheme, we included the data of the decimalisation table and card offset. Observe that with the maximum 16 digits of the PAN being used, we still have 9 digits of zeros in the final field of block B2. Our idea is to use the CVV calculation method twice, in the manner of a hashed MAC or HMAC function.

[5] The CVV/CVC is not to be confused with the CVC2 or CVV2, which is printed on the back of the card and designed to counteract 'customer not present' fraud when paying over the Internet or by phone.

We will calculate the CVV of a new set of data, containing the decimalisation table and offset or PVV and a code for the PIN block format. Then we will insert the result of the original CVV calculation to produce a final 5-digit MAC.

Our second CVV, which we will call CVV', contains the following fields:

Dectab	Offset/PVV	PIN block format	original CVV	0 pad
16 digits max	4 digits	1 digit	5 digits	6 digits max
Block B1'	Block B2'			

The position of each field is not important (except that it has to be fixed and agreed). We calculate CVV' in the same way as the standard CVV. This makes for easy upgrade from the original infrastructure, because CVV generation and verification commands are already available in HSMs so will need minimal changes to the firmware. The PIN_Verify command of the HSM must be changed to check CVV' before performing a verification test. The PIN test is only performed if the CVV' of the inputs matches the supplied CVV'.

One could use the same keys for calculating the CVV' as for the CVV, or one could use different keys. Either way, the verifying HSM needs access to these keys. The use of different keys could be motivated by a desire to be able to check CVVs, and so to an extent to verify card authenticity, at other points in the ATM network, without giving these nodes the keys used to create CVV's.

4.3 Security of the CVV Based Scheme

In our formal scheme we gave a proof of security by typing [11]. We proved that the modified version of the PIN_Verify function, implementing the MAC check before checking the PIN, gives robust declassification of the correctness of the PIN, subject to the following assumptions: 1) the MAC is unforgeable, 2) the attacker only has one encrypted PIN block to test, containing the correct PIN for a single given PAN, 3) encryption is perfectly secure (though not necessarily randomised). In proposing a scheme with a 5-digit MAC value, we are admitting the possibility of brute-force attacks, and thus breaking assumption 1). However, to guess the CVV' for a given set of parameters should take an average of 50 000 trials. So, an attack like the dectab attack which previously required 15 calls to the API will now take 750 000 calls. HSMs typically perform something of the order of 1000 PIN verifications per second, so this change moves the attack time for a single PIN from 0.015 seconds to 750 seconds, or 12.5 minutes.

Of course, the security of the scheme depends on the attacker not being able to calculate CVV's by any other means that brute force. For example, the API of the HSM must not make available the functionality to allow the creation of CVV's on arbitrary data. However, one would imagine this functionality only being available at an issuing facility (much like the functionality that prints PIN mailers to send to customers).

4.4 Practicalities of Deploying our Proposal

Sources close to the banking industry have told us the hardest (i.e. most costly) things to change in the ATM network are the network itself and the messaging

formats. Our proposal has the advantage of requiring neither of these to change. ATMs and switches generally already send the CVV from the ATM to the issuing bank, so can easily be adapted to send CVV'. In fact, many ATMs blindly send all the 'Track 2 data' from the magnetic card - this includes the PAN, expiry date, and CVV. Under most schemes there is still space on the magnetic stripe for a further 5 digit code. Chip based cards should have no problem storing a further 20 bits of data. So, we could use CVV' and the original CVV, thus allowing old style CVVs to be checked separately if required.

The next most costly thing to change is the software in the bank that calls the HSM. Our proposal requires some small changes here, since the verification command will now have extra parameters (the CVV' and a handle for the CVV' key). Changing the firmware in the HSM is not considered to be a prohibitive expense, and we require only minor changes, since HSMs can already verify CVVs.

A further advantage of our proposal to put the MAC onto the magnetic stripe is that it does not require any changes to the firmware of the PED. No MACs need to be calculated at the ATM, and no MAC keys need to be distributed to ATMs.

A particular feature of our scheme is that it does not require banks not adopting the scheme to change anything. A bank can unilaterally decide to add CVV's to its cards and require them at its verification facilities. The CVV' will pass through the network along with the rest of the track 2 data. The bank will thereby gain assurance of security at its verification facilities. However, it is known that PINs are not always verified at the issuing bank. There are also 'stand-in' verification centres run by, e.g., Visa and Mastercard, deployed to assure availability or to deal with foreign ATM transactions in certain cases. Upgrading these will be more complex, since even if the new 'Verify with CVV" command is made available at the facility, if an intruder can still access the old command, he can *a priori* still make the old attacks. A facility could work around this by partitioning the verification keys (PDKs) such that PDKs designated for use in the new scheme can only be used under the new command. Features for this kind of *key separation* are already implemented in many HSMs, since it is known to be dangerous to allow, for example, a key for IBM PIN verification to be used for VISA PVV verification [12, §3.5.7].

4.5 Addressing Translation Attacks

So far, our proposal only addresses the problem of attacks at the verification facility. By preventing queries with altered dectabs, PANs, or offsets, we blunt the attacks given in section 3.1. However, the CVV' cannot be used to ensure integrity of parameters at translation nodes, in order to address the attacks of section 3.2. This is because firstly, we do not envisage distributing the CVV' keys to all the nodes in the network, and secondly, we need to add a parameter giving the format of the PIN block to the MAC to protect against 'wrong format' attacks, and this parameter will change as the EPB makes its way through the

network, requiring the MAC value to be recalculated. The CVV' is therefore not suitable for this purpose.

Point-to-point MACs. There are other ways to upgrade the network to protect against the translation attacks. For example, we can demand that each switch shares a MAC key with every switch to which it might send an EPB, just as it currently shares a PIN encryption key with every such node. It can then calculate a MAC for every outgoing block, and verify a MAC for every incoming block. This is the scheme we proposed in our theoretical paper [11]. In practice, we cannot assume that all switches in the network will be upgraded at once. As a consequence, there will be a time when there are some switches which can calculate MACs, and some that cannot. As a consequence there will be edges in the network (i.e, some communicating pairs of switches) that can deal with MACs and share a key, along with some that cannot. PINs travelling through non-upgraded switches will, of course, still be vulnerable to translation attacks, even if they come from, or travel to, upgraded nodes. In fact, once they reach non-upgraded switches, i.e. switches whose translation function is not MAC-enabled, all the old attacks will be possible. This is, in a sense, natural as we cannot guarantee the security of PINs passing through insecure subnetworks.

Note that even if a PIN does not travel through non-upgraded switches, it might reach a MAC-enabled switch that is in contact with non-upgraded ones (see figure 1). This 'borderline' switch will have to offer both MAC-enabled and standard translate functions in order to communicate with the old protocol. This leaves the door open for attacks as an intruder might just disregard the MAC and call the old translate function in order to mount all the old translate attacks. A switch, in fact, only becomes secure once the old translate command can be disabled. A way to mitigate this problem is to add new special (temporary) *borderline switch* (see figure 2) with the only aim of interfacing upgraded and non-upgraded subnets. This nodes would only be reached by EPBs going to/coming from an insecure, non-upgraded, subnetwork. In this way EPBs travelling inside an upgraded network are not exposed to attacks on borderline nodes.[6]

Single format. Since translation attacks are based on *reformatting* the EPB, another possibility to prevent them would be to enforce the use of one single block format. In this way a translate would only consist of decrypting and re-encrypting the PIN block. Even in this case, upgrading the whole network would require time, leading to the same situation discussed above.

This solution does not require any new modified HSMs and protocols, since it only aims at reducing the existing functionalities so to operate on a single format thus it is, in principle, very appealing. However there are some difficulties that have to be carefully considered: (*i*) the 'political' decision on which format should be elected as 'the one' is far from being easy to take since many different standards exist, proposed by different entities; (*ii*) the choice of the format

[6] There might be attacks on the PIN routing that force them to reach untrusted part of the networks. They are of course relevant but out of the scope of this work.

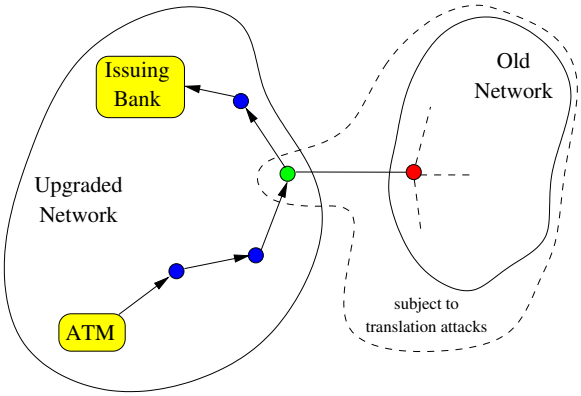

Fig. 1. Problem with borderline switches

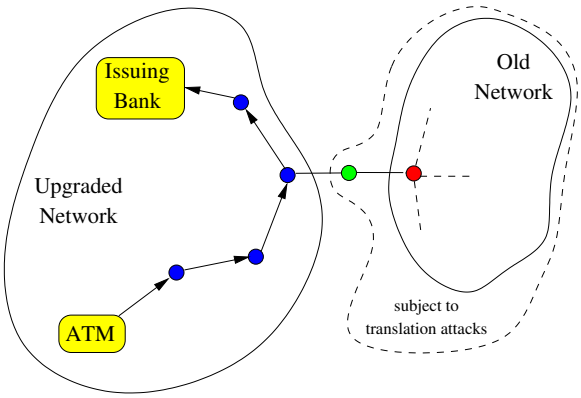

Fig. 2. Adding special borderline switches makes the upgraded network secure

might require upgrading some old HSMs or PEDs so to support it, especially for randomized formats like ISO1 or ISO3 which require good random number generators not present in old hardware.

5 Comparison to Other Proposals

Attacks on PIN processing APIs have been known for years [5, 7, 10, 12, 16], yet as other authors have observed, "proposals to counter such attacks are almost non-existent in the literature" [14]. In their decimalisation table attack paper, Bond and Zielinski suggest that the "decimalisation table input must be cryptographically protected so that only authorised tables can be used" [10, §6], though they do not give a scheme for doing so. Our proposal is in essence just such a scheme, though it additionally protects the other non-confidential inputs to the PIN verification command. Clulow includes the issue of 'Parameter integrity' in his discussion of solutions to the problem of PIN recovery attacks [12, §3.8].

He writes, "perhaps the most logical solution is to include the use of MACs (or similar methods) to ensure data integrity. For example, a decimalization table is supplied to a function call along with a MAC and the key for verifying that MAC." Our scheme achieves the same objective with a single MAC for all the non-confidential parameters. Clulow notes that "These solutions are fairly obvious. The challenge comes in ensuring interoperability between devices from different manufacturers". By basing our MAC around an existing standardised algorithm already in use in HSMs, and by putting the MAC onto the card rather than requiring it be calculated elsewhere in the network, we hope to address this challenge.

Finally, Mannan and van Oorschot have presented a scheme for 'salting' PINs [14]. The idea is to have banks fix a 128-bit salt value for each customer. This is written in plaintext to the bank card and stored encrypted at the verification facility. When a customer types her PIN at an ATM, the ATM generates a 12 (decimal) digit number as a pseudorandom function of the typed PIN and the salt. This is called the Transport Final PIN (TFP). This is sent in the EPB to the verification facility, where the process can be duplicated using the customer's true PIN, offset and encrypted salt and a comparison made to see if the PIN was correct. The main advantage of this is that at translation nodes, the known attacks reveal only the salted PIN, not the real one. Given the entropy of the salt, this is of little use to an attacker.

The Mannan/van Oorschot scheme shares some of the advantages of our proposal in that it does not require a change to messaging formats in the network, nor does it require changes to HSMs used in switches. However, their scheme does require that ATM PEDs are changed to be able to calculate a particular pseudorandom function, and also that verification HSMs have access to a database of all encrypted salts. Their proposals are primarily aimed at blunting the attacks of Berkmann and Ostrovsky [7], and do not seem to prevent, e.g., decimalisation table attacks at the verification facility. One could in fact deploy their scheme in conjunction with ours to provide effective protection against a larger set of attacks.

6 Conclusions

We have described a version of a MAC based scheme for ensuring integrity of queries to PIN processing APIs that is easy to implement and does not require wholesale changes to the ATM infrastructure. This ease is at the cost of some security, since the CVV codes can be cracked by brute force, but its implementation in the short term would make attack scenarios far less profitable. In the medium term we feel that the full MAC scheme should be used. The cost of this should be weighed against the cost of a complete overhaul of the way PIN processing is carried out in the ATM network.

References

1. Hackers crack cash machine PIN codes to steal millions. The Times online. http://www.timesonline.co.uk/tol/money/consumer_affairs/article4259009.ece.
2. PIN Crackers Nab Holy Grail of Bank Card Security. Wired Magazine Blog 'Threat Level'. http://blog.wired.com/27bstroke6/2009/04/pins.html.
3. Verizon Data Breach Investigations Report 2009. Available at http://www.verizonbusiness.com/resources/security/reports/2009_databreach_rp.pdf.
4. B. Adida, M. Bond, J. Clulow, A. Lin, R. Anderson, and R. L. Rivest. On the security of the EMV secure messaging API, 2008.
5. R. Anderson. Why cryptosystems fail. *Commun. ACM*, 37(11):32–40, 1994.
6. R. Anderson. What we can learn from API security (transcript of discussion). In *Security Protocols*, pages 288–300. Springer, 2003.
7. O. Berkman and O. M. Ostrovsky. The unbearable lightness of PIN cracking. In Springer LNCS vol.4886/2008, editor, *11th International Conference, Financial Cryptography and Data Security (FC 2007), Scarborough, Trinidad and Tobago*, pages 224–238, February 12-16 2007.
8. E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *J. Cryptology*, 4(1):3–72, 1991.
9. M. Bond. *Understanding Security APIs*. PhD thesis, University of Cambridge, England, 2004. http://www.cl.cam.ac.uk/$\sim$mkb23/research.html.
10. M. Bond and P. Zielinski. Decimalization table attacks for pin cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, 2003. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-560.pdf.
11. M. Centenaro, R. Focardi, F.L. Luccio, and G. Steel. Type-based Analysis of PIN Processing APIs. In *14th European Symposium on Research in Computer Security (ESORICS'09)*, LNCS, September 2009. To appear.
12. J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.
13. IBM Inc. CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors. Technical report, 2006. Releases 2.53–3.27. http://www-03.ibm.com/security/cryptocards/pcicc/library.shtml.
14. M. Mannan and P. van Oorschot. Reducing threats from flawed security APIs: The banking PIN case. *Computers & Security*, 28(6):410–420, September 2009.
15. A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, May 2006.
16. G. Steel. Formal Analysis of PIN Block Attacks. *Theoretical Computer Science*, 367(1-2):257–270, November 2006.

A Some standard 64-bit PIN block formats

IBM 3624 This format supports PINs from 1 to 16 digits in length. P are the PIN digits. Depending on the PIN length, P/X can be a PIN digit or a pad value, i.e. a value different from all the PIN digits and fixed a-priori.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X	P/X

ISO-0, ANSI X9.8, VISA-1, ECI-1 Supports PINs from 4 to 12 digits in length.

0	L	P	P	P	P	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	F	F

0 identifies the format, L is the length of the PIN, P are the PIN digits while P/F is either e PIN digit or the pad value F, depending on the PIN length. Then the rightmost 12 digits of the PAN are written as follows:

0	0	0	0	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN

and we now just xor the two numbers:

0	L	P	P	P xor PAN	P xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	P/F xor PAN	F xor PAN	F xor PAN

ISO-1 Supports PINs from 4 to 12 digits in length.

1	L	P	P	P	P	P/R	P/R	P/R	P/R	P/R	P/R	P/R	P/R	R	R

1 identifies the format, L is the length of the PIN, P are the PIN digits while P/R is either e PIN digit or the pad random value R, depending on the PIN length.

ISO-2 Supports PINs from 4 to 12 digits in length.

2	L	P	P	P	P	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	F	F

2 identifies the format, L is the length of the PIN, P are the PIN digits while P/F is either e PIN digit or the pad value F, depending on the PIN length.

ISO-3 Supports PINs from 4 to 12 digits in length. It is the same as ISO-0 except that the PIN is padded with random digits.

3	L	P	P	P	P	P/R	P/R	P/R	P/R	P/R	P/R	P/R	P/R	R	R

3 identifies the format, L is the length of the PIN, P are the PIN digits while P/R is either e PIN digit or the pad random value R, depending on the PIN length. As for ISO-0, the rightmost 12 digits of the PAN are written as follows:

0	0	0	0	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN	PAN

and we now just xor the two numbers:

3	L	P	P	P xor PAN	P xor PAN	P/R xor PAN	P/R xor PAN	P/R xor PAN	P/R xor PAN	P/R xor PAN	P/R xor PAN	P/R xor PAN	P/R xor PAN	R xor PAN	R xor PAN

VISA-3 Supports PINs from 4 to 16 digits in length. P are the PIN digits while P/F is either e PIN digit or the pad value F, depending on the PIN length.

P	P	P	P	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F	P/F