The Unbearable Lightness of PIN Cracking

Omer Berkman[1] and Odelia Moshe Ostrovsky[2,3,⋆]

[1] The Academic College of Tel Aviv Yaffo, School of Computer Science
[2] Algorithmic Research Ltd.,
www.arx.com
[3] Tel Aviv University, School of Computer Science

Abstract. We describe new attacks on the financial PIN processing API. The attacks apply to switches as well as to verification facilities. The attacks are extremely severe allowing an attacker to expose customer PINs by executing only one or two API calls per exposed PIN. One of the attacks uses only the translate function which is a required function in every switch. The other attacks abuse functions that are used to allow customers to select their PINs online. Some of the attacks can be applied in switches even though the attacked functions require issuer's keys which do not exist in a switch. This is particularly disturbing as it was widely believed that functions requiring issuer's keys cannot do any harm if the respective keys are unavailable.

Keywords: Security API, API attack, Financial PIN Processing API, HSM, Insider attack, Phantom Withdrawal, VISA PVV, IBM 3624, EMV.

1 Introduction

Personal Identification Number (PIN) is the means used by a bank account holder to verify his/her identity to the issuing bank. When a PIN is entered by the card holder at a service point (e.g., an Automatic Teller Machine), the PIN and account number are sent to the verification facility (the issuing bank or other authorized entity) for verification. To protect the PIN on transit, it is formatted into a PIN block, the PIN block is encrypted under a transport key and the resulting Encrypted PIN Block (EPB) is sent for verification. As there usually isn't direct communication between the service point and the verification facility, the PIN goes through switches. Each switch decrypts the EPB, verifies the resulting PIN block format (so the format serves as some form of Message Authentication Code), re-formats the PIN block if necessary, and re-encrypts the PIN block with a transport key shared with the next switch (or the verification facility when arriving there). Switches may be part of other issuers' verification facilities or may be stand alone. There is generally no connection between a switch facility that handles an incoming EPB, and the issuer of the respective

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-540-77366-5_37

⋆ The work of this author was carried out as part of an MSc thesis in Tel Aviv University.

account number. Additionally, switches may be physically far from the issuer (for example, when a customer withdraws money overseas).

To protect the PIN and the encryption keys both in switches and in the issuer's environment, all operations involving a clear PIN are handled within a Hardware Security Module (HSM). Such operations are controlled by an application at the site using a cryptographic API. The Financial PIN Processing API is a 30-years old standard which includes functions for, e.g., PIN issuing, PIN verification, PIN reformatting, and PIN change.

The issuer's environment is usually physically separated into an issuing facility and an online verification facility. The issuing facility where customer PINs are generated and printed for delivery is usually isolated logically and physically from the rest of the issuer's environments. The verification facility as well as switches on the other hand, are connected to the outside world and required to be online so they are much more prone to attack. Much of the required functionality in the issuing facility is sensitive, so HSMs implementing the Financial PIN processing API should separate (at least logically) the functionality required for the issuing facility from that of the verification facility. Switches are treated as verification facilities in this respect so HSMs in switches should contain (at least logically) only functions required for the verification facility.

In this paper we describe attacks on the Financial PIN Processing API, which result in discovering customers PINs. The attacks can be applied in switches as well as in verification facilities. The attacks require access (i) to the HSM in the attacked facility for executing API calls; (ii) to EPBs incoming to the attacked facility. Applying such attacks thus requires the help of an insider in the attacked facility. However, when the attacks are applied on a switch, one cannot relate to them as insider attacks. Since the switch, and the issuer whose EPBs are attacked on the switch, are unrelated, an insider of the switch facility is an outsider from the issuer's point of view. The issuer has no control, neither on the environment nor on the employees in the attacked facility. We stress that our attacks only require the use of API functions (and only the ones approved for the verification facility) and **do not** assume that the attacker can perform sensitive operations such as loading known keys into the attacked HSM.

Attack 1 uses a single API function denoted *translate*. The *translate* function allows to reformat an EPB in any PIN block format to an EPB in another PIN block format. It also allows to change the transport key which encrypts the PIN block. It is a required function in every switch, and exists also in verification facilities as part of the API. The attack executes a (one-time) preprocessing step of 20,000 HSM calls in which a (small) look-up table is built. This table allows revealing the PIN packed in each EPB arriving to the attacked switch using one or two HSM calls.

Attack 2 requires the use of one of two API functions - *calculate offset* or *calculate* PVV - which are used primarily for allowing customers to select their PINs online. The attack has four variants. These variants allow discovering a PIN given its respective account number (Attack 2.1), discovering a PIN from its EPB (Attack 2.2), setting a new value for a customer's PIN given the respective account number (Attack 2.3), and partitioning all EPBs arriving to the attacked facility into groups having the same PIN (Attack 2.4). The attacks on account numbers (2.1 and 2.3) are applied in a verification facility. The attacks on EPBs (2.2 and 2.4) can be applied both in switches and in verification facilities. Each of the four variants can be performed in one or two HSM calls per attacked entity (account number or EPB), and requires no preprocessing.

Both *calculate offset* and *calculate PVV* functions require issuer keys so it is quite surprising that they can be attacked in switches as switches do not contain issuer keys. This is particularly disturbing as it is widely believed that functions requiring issuer's keys cannot do any harm if the respective keys are unavailable.

In some of the cases above, the attacked functions are not used by the application at the site. For example, the *calculate offset* and *calculate PVV* functions are generally not required in switches and *translate* is generally not required in a verification facility. It is important in such cases to irreversibly disable these functions (as well as other unused functions) if this capability is offered. Issuers certainly have the incentive to apply such measures in their verification (and issuing) facilities. However, it is not clear how to verify that switch facilities adhere to these measures.

The attacks abuse integrity and secrecy weaknesses in the financial PIN processing API, some of which are well known ([1,2,3,4,5,6], see also Section 3). For example, integrity in the financial PIN processing API is so weak that one can easily trick API functions into accepting a customer's EPB together with an account number which is not the customer's.

As the attacks target the standard itself, they apply to all common commercial HSMs implementing the API and affect all financial institutions. The attacks apply also to systems employing the EMV standard ([7]) when on-line verification takes place, as is the case in ATM transactions.

The attacks enable the discovery of several thousand customer PINs per attacked HSM per second enabling an attacker to apply serious attacks on issuing banks, such as simultaneous withdrawals of aggregate large sums of money. The attacks may also explain cases of phantom withdrawals where a cash withdrawal from an ATM has occurred, and neither the customer nor the bank admits liability.

To prevent the attacks described in this paper, changes in the standard must be introduced. Such changes require worldwide modifications in ATMs, HSMs and other components implementing the Financial PIN processing API.

The rest of the paper is organized as follows. Section 2 discusses the threat model. In Section 3 we describe known vulnerabilities in the standard. Sections 4 and 5 describe our attacks. A discussion of the attacks is given in Section 6 and is followed by concluding remarks in Section 7. Finally, an appendix contains information on the attacked functions for reference.

2 Threat Model

A potential attacker is an insider of the attacked facility - a switch or a verification facility. Such an insider should have logical access to the HSM in the facility and should be able to generate API calls (the required API functions depend on the attack). In many cases this is easy as the HSM is connected to the organization's internal network. When this is not the case, the attacker can, for example, interfere with or masquerade as the legal application working with the HSM in the attacked facility.

In most of the attacks the attacker is required to generate EPBs which contain known PINs and which share a transport key with the attacked HSM. To do this, the attacker can use any banking card (genuine or fake) and enter a desired PIN at an ATM adjacent physically or logically to the attacked HSM. The attacker then needs to record the EPB when it arrives to the attacked facility. This can be done in various ways, e.g., by a program that reads the EPB on its way from the application to the HSM in the site. In the same way an attacker is able to record EPBs incoming to the switch, e.g., in order to expose the PINs they hide.

In order to prevent insider attacks on their HSMs, a few banks install a (hardware) mechanism by which their on-line application timeouts whenever the HSM is used by a different entity. However, it is hard to figure out whether a short timeout really signifies an attack. Furthermore, it is possible to attack a system employing such a mechanism by, for example, physically intervening with the low-level communication between the application and the HSM.

All API functions use cryptographic keys. The standard does not specify how keys should be input to an API function but most implementations either keep keys outside the HSM encrypted by a master key, or keep them inside the HSM. In the first case, HSMs accept encrypted keys in each API call. In this case, an attacker is only required to record the desired encrypted key buffer from a real transaction. The same encrypted key can then be used in the attacker's API calls to the HSM. In the second case where keys are stored and managed inside the HSM, the attacker only needs to know the required key ID. In this case, however, the HSM may also handle user access rights to the keys. To use the required keys in such cases the attacker can, as before, interfere with or masquerade as the legal application working with the HSM in the attacked facility. In any case, we never assume that the attacker has any knowledge of the value of cryptographic keys.

Transport keys sometimes change. However, parameters to the API functions that control the keys to be used in the API function come from the outside so the attacker can always direct the HSM to use the same key. Additionally, when required, the attacker can use the *translate* function to translate an EPB encrypted with one transport key to an EPB encrypted with another.

It should be noted that an attacker is not required to be an authorized user - a maintenance employee or after-hours cleaner can generally do the job on the attacker's behalf. Moreover, in all variants of Attack 2, the attacker can also be an insider programmer that applies the attack innocently, believing that the PVVs or offsets (see Section 5 and the appendix for definitions) he/she was asked to supply are required for legitimate purposes as they are output in clear from the relevant API functions, and treated as non-sensitive. In switches, insiders (perhaps even high-ranking) may conduct the attacks but target EPBs of foreign banks only, dramatically decreasing the chances of their being caught.

3 Basics and Previous Work

As we mentioned in the introduction, on its way for verification, the PIN is formatted into a PIN block and the result is encrypted using a transport key to generate an Encrypted PIN Block (EPB). Specifically, [8] describes four different PIN Block formats. ISO-0, ISO-1, ISO-2, and ISO-3, which differ in whether the customer's account number and/or random data is involved in the format in addition to the PIN itself. ISO-0 uses only account number, ISO-1 uses only random data, ISO-2 uses neither account number nor random data, and ISO-3 uses both account number and random data. [9] approves ISO-0, ISO-1, and ISO-3 for online PIN transactions. ISO-2 is **not** approved for online PIN transactions since an EPB based on ISO-2 (and on a given transport key) has only 10,000 possible values (assuming the PIN is of length 4) enabling the use of a look-up table.

Our attacks abuse the following known weaknesses:

1. The *translate* API function allows reformatting an EPB from any of the approved formats (ISO-0, ISO-1, or ISO-3) to another ([3,6,4]).
2. The ISO-1 format is independent of any account number ([6]).
3. A result of Weaknesses 1 and 2 is that an EPB in ISO-0 (or ISO-3) associated with a given account number can be converted (by going through ISO-1) to an EPB in ISO-0 associated with a different account number ([1], [3] and [6]). Note that doing this unties the link between the customer's account number and the customer's PIN and creates a fabricated link between a different account number and this customer's PIN.
4. An EPB based on ISO-0 and a particular account number has only 10,000 possible values enabling the use of a look-up table ([3,1,6]). We note that this weakness implies that for a particular account number, the ISO-0 format is as weak as ISO-2.
5. As mentioned in the introduction, the format of PIN block serves as a form of Message Authentication Code (MAC). The weakness is that in ISO-0 format, digits of the PIN are XORed with digits of the account number, making it impossible to correctly authenticate neither ([6,4,3]).

We are not aware of previous attacks abusing the *calculate offset* and *calculate PVV* functions but note that Attack 2.1 which abuses the *calculate offset* function is reminiscent of an attack (described in [10,11,6]) on a **non-API** function which was added temporarily to the implementation of the API in a certain bank in order to enable changing all customers' account numbers without re-issuing new PINs.

Previous API-level attacks appear in [12,13,14]. Previous attacks on the Financial PIN Processing Standard appear in the references above as well as in [11]. Among these, of particular interest is the decimalisation-table attack ([5]), which targets the *verify* function in verification facilities. It allows revealing a PIN from its account number or EPB by executing 15 HSM calls on the average. Contrary to our attacks, the decimalisation-table attack does not apply to switches. Additionally, it targets only one of the two verification methods in the standard (see Section 5 for details) while we attack both.

4 Attack 1 - Attacking the Translate Function

The attack we describe in this section, enables revealing for any EPB arriving to the attacked switch (or verification facility), the PIN that the EPB packs. The attack uses at most two API calls per EPB. The attack requires also a one-time preprocessing step consisting of 20,000 API calls (assuming the PIN is of length 4 as is normally the case). The attack uses the translate API function only.

We start by observing that Weakness 3 (in Section 3) degrades the security of the system to the strength of ISO-2 (recall that ISO-2 is the weak and thus non-approved PIN block format): Fixing an account number to some value A and translating all EPBs arriving to the attacked switch to ISO-0 with account number A (going through ISO-1) ensures that all resulting EPBs are based on account number A, thus degrading their strength to that of ISO-2. This observation has extremely serious implications on the security of the Financial PIN Processing API, as it implies that a single look-up table of size 10,000 is all that is required in order to discover the PIN packed in **every EPB arriving to the attacked switch**, regardless of its account number. Clulow ([6]) was evidently aware to this saying "it is noteworthy that regardless of format, key and pan, all encrypted pins are potentially vulnerable to a single codebook", but his words seem to have gone almost unnoticed probably because he and others had no efficient way of building the required table.

In this section we do exactly this. Specifically, we show how to generate a table of 10,000 EPBs, where the ith EPB, $1 \leq i \leq 10,000$ contains the PIN whose value is i, and such that each EPB in the table is formatted in ISO-0 using a fixed account number A.

One obvious way such a table can be generated is by brute force - generating 10,000 EPBs by ATMs: For each i, $1 \leq i \leq 10,000$ use a card with any account number and type PIN value i. When the respective EPB arrives at the attacked HSM, translate it to ISO-0 using account number A (by one or two calls to the *translate* function depending on the format of the incoming EPB). Using different account numbers when generating EPBs via ATMs would make it harder to discover the attack.

We now describe a much more practical method of building the table. Instead of generating an EPB per each possible PIN as in the brute force manner above, this method uses Weakness 5 to generate an EPB per 100 possible PINs. Thus, by generating 100 EPBs we can build the whole 10,000-entries look-up table (it is also possible to generate less than 100 EPBs and build a partial look-up table).

We start by describing the ISO-0 PIN block format. Denote the PIN $P_1P_2P_3P_4$ and the respective account number $A_1A_2\ldots A_{12}$ (only 12 digits of the account number are used in the ISO formats). The PIN block is the XOR of two 16-hexadecimal digits blocks. An *original block* containing the PIN ("F" stands for the hexadecimal value F)

0	4	P_1	P_2	P_3	P_4	F	F	F	F	F	F	F	F	F	F

with an *account number block* containing the account number

0	0	0	0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}

When an API function receives an EPB in ISO-0 as a parameter, it also receives its associated account number. To use the PIN packed in the EPB, the function decrypts the EPB, and XORs the result with the account number block to recreate the original block. It then authenticates the result by verifying that the values of the first two digits of the original block are 0 and 4, that the last 10 digits are hexadecimal F and that the PIN is composed of decimal digits.

Weakness 5 - the fact that two digits of the PIN are XORed with two digits of the account number - is used for generating the table. The attacker generates in ATMs 100 EPBs packing, respectively, PIN values $0000, 0100, \ldots, 9900$. Each of these EPBs is formatted in ISO-0 and associated with account number $00A_3 \ldots A_{12}$ where the values $A_3, \ldots, A_{12}$ are immaterial to the attack and can be different for each PIN value to make the attack more innocent. (It is also possible to generate the 100 EPBs in ATMs using completely arbitrary account numbers and then change the account number of each EPB to the desired one using the *translate* function.)

We complete our description by showing how the attacker generates an EPB containing PIN value $xyuv$ for any decimal values x, y, u, v:

To generate an EPB that packs PIN value $xyuv$, the attacker uses the EPB packing $xy00$ which was generated by ATM. Using the *translate* function this EPB is reformatted to ISO-1 but instead of using the original account number $00A_3 \ldots A_{12}$ the attacker provides the *translate* function with account number $uvA_3 \ldots A_{12}$.

The *translate* function decrypts the EPB and gets a block which is the XOR of the original block

0	4	x	y	0	0	F	F	F	F	F	F	F	F	F	F

and the original account number block

0	0	0	0	0	0	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}

Note that the function only sees the decrypted block - the XOR of these two blocks. It then XORs the decrypted block with the following:

0	0	0	0	u	v	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}

to get

0	4	x	y	u	v	F	F	F	F	F	F	F	F	F	F

This resulting block will be authenticated (its first two digits are 0 and 4, its 10 last digits are hexadecimal F, and the PIN consists of decimal digits). Consequently, PIN value $xyuv$ will be packed in an EPB in ISO-1 PIN block and returned. The attacker can now translate this EPB to ISO-0 with the desired account number A.

5 Attack 2 - Attacking Functions Allowing PIN Change

In the Financial PIN Processing API, the PIN is verified using one of two approved methods - the IBM 3624 or the VISA PIN verification value (PVV) methods. In both methods the input to the verify function is as follows:

- An EPB containing the PIN presented by the customer.
- The customer's account number.
- A four decimal digits *customer's verification value* (called *offset* in the first method and PVV in the second).

This customer's verification value is not secret. It is kept either in a database or on the customer's card.

Denote by P the PIN packed in the EPB, by A the customer's account number, and by V the customer's verification value.

The verify function decrypts the EPB, authenticates it by verifying the PIN block format, extracts P from the EPB, and verifies whether $V = f(P, A)$ where f is a function which depends on an issuer's secret key. The function f and the issuer's key are different between the two methods.

In order to allow customers to select their PINs online, the Financial PIN Processing API contains two functions (one for each method) that allow recalculating the customer's verification value when the customer's PIN changes. The functions are denoted *calculate offset* and *calculate PVV*. Both functions receive the following input:

- An EPB containing the customer's selected PIN.
- The customer's account number.

The functions return $V = f(P, A)$ where P, A, V and f are as before. We note that in both functions, the value V is pseudo random as a result of using the random issuer's key in f.

The main weakness in both functions regardless of f is that the new PIN supplied to the function (packed in an EPB) is not bound to the old PIN. Indeed, the main step in each of the four variants 2.1-2.4 abuses this weakness, so the variants have much in common.

Note that since an attacker can carry out the attack by directly using the API, it would not be enough to check the above binding by the application at the site. Note also that it would not be enough to change the API by adding each of these functions a parameter consisting of an EPB that packs the customer's old PIN (and a means to verify it, i.e., the respective verification value) since the attacker can record a customer's real EPB on its way for verification, and use it as the additional parameter.

To attack any of the two API functions, we are required to send as parameters an EPB and an account number. In all four attacks, the EPB would be generated by one customer and the account number would belong to another. To force the attacked function to accept the non-matching parameters, weaknesses 1 and 2 of Section 3 are utilized: We use the *translate* function to reformat each EPB to

ISO-1 before sending it to the respective API function. Because ISO-1 does not depend on account number, there would be no inconsistency between the EPB parameter and the account number parameter. When describing the attacks below, we do not mention this reformatting any more (but we count it in the number of HSM calls required). Note that restricting the *calculate offset* or *calculate PVV* functions to accept only EPBs with a certain format would not thwart the attacks, as we can reformat the EPB to that format. Note also that with the exception of Attack 2.4, all the attacks below can be applied (though in a more restricted form) even if the *translate* function is disabled. See Section 5.3.

In Section 5.1 we describe our attacks on the *calculate offset* function. In Section 5.2 we describe our attacks on the *calculate PVV* function. It is worth noting that except for assuming that the value V is pseudo random, the attacks on *calculate PVV* do not use any properties of the respective f (so, for example, they apply also to *calculate offset*).

We use a shorthand $O = offset(E, A)$ (respectively, $V = PVV(E, A)$) to denote calling the *calculate offset* function (respectively, the *calculate PVV* function) with an EPB E and an account number A.

5.1 Attacks on the Calculate Offset Function

The specific function f in *calculate offset* is $V = P - g(A)$ where P is the PIN packed in the EPB parameter, A is the account number parameter, V is the returned offset, g is a function that depends on an issuer's key and computes a 4 decimal digits number, and "$-$" is minus modulo 10 digit by digit.

Attack 2.1 - Attacking Account Numbers in a Verification Facility. This attack reveals for every customer account number associated with the attacked issuer, the respective customer's PIN. It requires one HSM call per attacked account number. In addition, it requires generating by ATM an EPB that packs a known PIN. This single EPB will be used to attack all account numbers associated with the issuer.

We start by generating an EPB in an ATM that packs an arbitrary known PIN (the account number associated with this EPB is immaterial as we reformat the EPB to ISO-1 prior to using it). This EPB, denoted E_a (for attacker's EPB) is used to attack the account numbers of all customers.

For each customer's account number A_c, compute $O = Offset(E_a, A_c)$. Denote by P_a the value of the known PIN packed in the attacker's EPB, by P_c the required customer PIN, and by O_c the customer's offset (stored in the issuer's database or on the magnetic stripe of the card). We thus have $O = P_a - g(A_c)$. Since P_a is known, $g(A_c)$ can be easily computed. We also know that $O_c = P_c - g(A_c)$. Since $g(A_c)$ is known and since the value of O_c is not secret (it can be recorded during a transaction or read from the database or from the card) the customer's PIN P_c can be trivially calculated. Note that the exact nature of g is immaterial, but the attack requires that the real value of $g(A_c)$ be used (since the value of O_c depends on it) so it needs to be applied in the verification facility where the required issuer's key exists.

Attack 2.2 - Attacking EPBs Incoming to a Switch. The attack reveals for each customer's EPB arriving to the attacked switch, the PIN it packs. It requires one or two HSM calls per attacked EPB. In addition, it requires generating by ATM an EPB that packs a known PIN. This single EPB will be used to attack all EPBs arriving to the attacked switch. We note that the attack can be applied also in verification facilities.

Generate an EPB in an ATM that packs a known PIN and denote it E_a. Fix an arbitrary account number B. Compute $O_1 = Offset(E_a, B)$. For each customer's EPB arriving to the attacked switch compute $O_2 = Offset(E_c, B)$ where E_c is the customer's EPB.

Denote by P_a and P_c the values of PINs packed in the attacker's and customer's EPBs, respectively. We thus have $O_1 = P_a - g(B)$ and $O_2 = P_c - g(B)$. Since the value of P_a is known, the value of P_c can be trivially calculated. Note that the value of $g(B)$ is immaterial, so the attack can be applied in a switch which does not contain the required issuer's key.

5.2 Attacks on the Calculate PVV Function

Attack 2.3 - Attacking Account Numbers in a Verification Facility. This attack reveals for any account number associated with the attacked issuer, the PVV that corresponds to this customer's account number and an attacker's chosen PIN. Replacing the verification value on the card or in the database (depending on the system) enables withdrawing money from the customer's account using the chosen PIN. The attack requires one HSM call per account number attacked. In addition, it requires generating by ATM an EPB that packs a known PIN. This single EPB will be used to attack the account numbers of all customers.

Generate an EPB in an ATM that packs an arbitrary known PIN and denote it E_a. For each customer's account number A_c, compute $V = PVV(E_a, A_c)$.

The computed PVV value V corresponds to the customer's account number and the chosen PIN. Since the attack takes place in the verification facility, the required issuer's key is used, and the PVV is valid.

It remains to explain how the attacker can replace the customer's original PVV used by the system by the PVV computed in the attack.

According to [9], the clear PVV can be stored on the card's magnetic stripe or in a PVV database. In case the PVV is stored on both, the PVV is taken from the database. In many implementations the PVV is stored only on the card as long as the customer uses the initial PIN generated by the issuer.

Setting the customer's PVV to the computed PVV can be done as follows:

Case 1: The PVV is stored only on the card. Generate a card containing the customer's details and set the PVV value on the magnetic stripe to the PVV that was calculated by the attacker. In this case the fabricated card (associated with the attacker's chosen PIN) and the customer's original card (associated with the customer's PIN) will both be valid at the same time. It is important to note that in this case, issuing a new PIN to a customer will not prevent the attack as the fabricated card with the false PVV will remain valid.

Case 2: The PVV entry of this customer exists in the PVV database. In this case the attacker needs write access to the PVV database. As both PVVs and offsets are not considered sensitive, access to this database is generally not restricted. In many banks, for example, HSM service personnel can access this database. As a result of the attacks published in this paper, the attitude towards PVVs and offsets is now being changed. Anyway, given write access, the attacker can do one of the following:

– Delete the PVV entry (and then apply the steps described in Case 1).
– Set the customer's entry in the PVV database to the PVV that was calculated by the attacker. If the entry does not exist - create it. In this case the fabricated card will be the only valid card.

Attack 2.4 - Attacking EPBs Incoming to a Switch. Consider all customer EPBs arriving to the attacked switch. The attack discovers for each such EPB (and its associated account number) a list of other EPBs having the same PIN (with high probability). It requires one or two HSM calls per attacked EPB. We note that the attack can be applied also in verification facilities.

We use a table of 10,000 entries. The table is indexed by values of computed PVVs. Each entry of the table contains customer EPBs (and their associated account numbers). Initially all entries are empty.

Fix an arbitrary account number B. We show how to attack any customer's EPB arriving to the switch. Denote by E_c the customer's EPB.

1. $V = PVV(E_c, B)$.
   The computed PVV value V equals $f(P_c, B)$ where P_c is the PIN packed in the customer's EPB.
2. Add the customer's EPB E_c to the table entry corresponding to the resulting PVV value V.
   For example, if V is 5678 then E_c will be added to table entry 5678.

The computed PVV value V depends only on P_c, B, and on the key k used by the function f. Since the attack is applied in a switch, k is not the issuer's key as required, but some other arbitrary value (not known to the attacker). The value of k is immaterial to the attack. All that we require is that the value V be a pseudo random function of P_c, B, and k. Since B and k are fixed, V can be regarded as a pseudo random function of P_c only.

Suppose we have performed the above with many EPBs. What actually happens in steps 1 and 2 above is that all EPBs that pack the same PIN value are thrown into the same table entry. Since the process is random, a table entry may be empty, may contain EPBs corresponding to a single value of PIN, or may contain EPBs corresponding to several PIN values. Combinatorically, the process is equivalent to throwing balls (PINs) to bins (table entries) and asking questions on the number of balls (distinct PINs) in each bin. It can be shown that when the number of balls and bins is the same (10,000 in our case) the average number of balls in a non-empty bin is less than 2. In other words, EPBs that ended in the same table entry correspond to less than 2 distinct PINs on the average.

To decrease the probability that EPBs in the same table entry correspond to more than a single PIN, we repeat the procedure with respect to the EPBs in each table entry using a different fixed account number C. EPBs from a given table entry that again end together in the same table entry, have high probability of having the same PIN.

5.3 What If Reformatting Is Disabled

In each of the attacks 2.1, 2.2, and 2.3 above, the attacker uses the translate function to reformat EPBs to ISO-1. This enables using a single recorded EPB for attacking all account numbers in attacks 2.1 and 2.3 (and all EPBs in Attack 2.2). Could the attacks still work if the *translate* function (or its reformatting capability) is disabled and all EPBs are in either ISO-0 or ISO-3 format?

An easy solution is for the attacker to record for each account number attacked, an EPB associated with that account number. Although such an attack cannot be applied on very large scale, it can still be harmful (especially when the attacker is interested in attacking specific customers).

Moreover, using the overlapping between PIN digits and account number digits (Weakness 5 in Section 3), a single recorded EPB in ISO-0 format may be used to attack up to 100 account numbers. Using Weakness 5 together with overlapping between account number digits and random digits, an EPB in ISO-3 format may be used to attack up to $6 \cdot 10^9$ account numbers. The details appear in [15].

6 Discussion

Attack 1 is perhaps the most hard to handle as the *translate* function is a required function in every switch. Attacks 2.1 and 2.3 deserve special attention as they do not even require that the customer know his/her PIN. Moreover, if one is interested in attacking a specific account number, the translate function is not required for the attack (as discussed in Section 5.3). Attack 2.2 is surprising, as it implies that customer fake cards having PINs different from the customer's original PIN, can be valid together with the customer's genuine card. Attack 2.4 does not require generating an EPB in an ATM. On the negative side, Attack 1 requires generating in ATMs 100 EPBs, and all variants of Attack 2 require that *calculate PVV* or *Calculate offset* be available.

Our recommendation to issuers is as follows. In their facilities, issuers should disable the *calculate offset* and *Calculate PVV* functions as well as the reformatting capability of translate, even for the price of eliminating customer selected PINs or other capabilities. Warning mechanism (e.g., with respect to the number of times API functions are called) may also be useful. With respect to switches, issuers should ensure good control over their country's local switches and apply detection and other mechanisms with respect to overseas transactions.

7 Conclusions

We have shown in this paper that the Financial PIN processing API is exposed to severe attacks on the functions *translate*, *calculate PVV* and *calculate offset* inside and outside of the issuer environment.

The attacks we describe provide possible explanations to many Phantom Withdrawals. The attacks are so simple and practical that issuers may have to admit liability not only for future cases but even retroactively. The attacks can be applied on such a large scale (in some of the attacks up to 18,000,000 PINs can be discovered in an hour) that banks' liability can be enormous.

As some of the attacks apply to switches, which are not under the issuers control, countermeasures in the issuers environment do not suffice. To be protected from this attack, countermeasures in all verification paths to the issuer must be taken. As this is unrealistic, solutions outside the standard must be sought.

We have also shown that physical and/or logical separation of the issuing and verification facilities does not prevent severe attacks, as part of the API functionality intended for use in verification facilities is vulnerable.

We have demonstrated that reformatting capability between different PIN block formats, can go further than degrading the security of the system to the weakest format, as weaknesses of several formats may be abused. Our attacks also show that the ISO-1 format is extremely weak and thus should be immediately removed from the list of approved interchange transaction formats.

Another interesting insight from the attacks described is that the offset and the PVV values may reveal as much information as the PIN itself. One possible remedy is treating them as secret values.

In addition to all implementations of this API, systems applying the EMV standard ([7]) and using online (rather than off-line) PIN verification are also vulnerable to the attacks.

The vulnerabilities exposed in this paper require worldwide modifications in ATMs, HSMs and other components implementing the PIN processing API.

Acknowledgements. We are indebted to Ezer Farhi, Uri Resnitzky, and Jony Rosenne for fruitful discussions and helpful comments, and to Amos Fiat for suggestions which significantly improved the presentation of the paper. We are also thankful to Mike Bond and Jolyon Clulow for comments on earlier versions of the paper.

References

1. Anderson, R.J., Bond, M., Clulow, J., Skorobogatov, S.: Cryptographic processors - a survey. Proceedings of the IEEE 94(2), 357–369 (2006)
2. Bond, M.: Understanding Security APIs. PhD thesis, University of Cambridge (2004), `http://www.cl.cam.ac.uk/mkb23/research.html`
3. Bond, M., Clulow, J.: Encrypted? randomised? compromised? In: Workshop on Cryptographic Algorithms and their Uses (2004)

4. Bond, M., Clulow, J.: Extending security protocols analysis: New challenges. In: Automated Reasoning and Security Protocols Analysis (ARSPA), pp. 602–608 (2004)
5. Bond, M., Zielinski, P.: 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
6. Clulow, J.: The design and analysis of cryptographic APIs. Master's thesis, University of Natal, South Africa (2003), http://www.cl.cam.ac.uk/jc407
7. EMV: Integrated circuit card specifications for payment systems (2004), http://www.emvco.com
8. ISO: Banking – personal identification number (PIN) management and security – part 1: Basic principles and requirements for online PIN handling in ATM and POS systems (2002)
9. VISA: PIN security requirements (2004), http://partnernetwork.visa.com/st/pin/pdfs/PCI_PIN_Security_Requirements.pdf
10. Anderson, R.: The correctness of crypto transaction sets. In: Christianson, B., Crispo, B., Malcolm, J.A., Roe, M. (eds.) Security Protocols. LNCS, vol. 2133, pp. 128–141. Springer, Heidelberg (2001)
11. Andersson, R.J.: Why cryptosystems fail. Communications of the ACM 37(11), 32–40 (1994)
12. Longley, D.: Expert systems applied to the analysis of key management schemes. Computers and Security 6(1), 54–67 (1987)
13. Rigby, S.: Key management in secure data networks. Master's thesis, Queensland Institute of Technology, Australia (1987)
14. Steel, G., Bundy, G.: Deduction with XOR constraints in security API modelling. In: McAllester, D. (ed.) CADE-17. LNCS, vol. 1831, Springer, Heidelberg (2000)
15. Moshe-Ostrovsky, O.: Vulnerabilities in the financial PIN processing API. Master's thesis, Tel Aviv University (2006)

Appendix

We describe below, for reference, the attacked functions parameters, and the computation performed by each function.

Translate	**Calculate PVV**	**Calculate offset**
EPB	EPB	EPB
account number	account number	account number
input PIN block format	PIN block format	PIN block format
input transport key	transport key	transport key
output PIN block format	issuer's PVV key	issuer's PIN key
output transport key		

The *translate* function extracts the PIN from the EPB by decrypting the EPB using the input transport key and authenticating the result using the account number and input PIN block format (Section 4 describes the authentication process). It then re-formats the PIN into a PIN block using the output PIN block format and account number, and re-encrypts the result using the output transport key. The resulting EPB is the output of the function.

The *calculate PVV* function extracts the PIN from the EPB (as in *translate*). It then concatenates the PIN to the account number, encrypts the result using the issuer's PVV key, and extracts four decimal digits from the encrypted result. These four digits constitute a PIN Verification Value (PVV) which is the output of the function.

The *calculate offset* function extracts the PIN from the EPB. It then encrypts the account number using the issuer's PIN key and extracts four decimal digits denoted *natural PIN* from the encrypted result. The natural PIN is then subtracted (modulus 10) from the PIN. The result constitute an *offset* which is the output of the function.