Reducing Threats from Flawed Security APIs: The Banking PIN Case*

Mohammad Mannan and P.C. van Oorschot
School of Computer Science
Carleton University, Ottawa, Canada

Abstract

Despite best efforts from security API designers, flaws are often found in widely deployed security APIs. Even APIs with a formal proof of security may not guarantee absolute security when used in a real-world device or application. In parallel to spending research efforts to improve security of these APIs, we argue that it may be worthwhile to explore design criteria that would reduce the impact of an API exploit, assuming flaws cannot completely be removed from security APIs. We use such a design philosophy in dealing with PIN cracking attacks on financial PIN processing APIs; several of these attacks have been reported in the last few years, e.g., Berkman and Ostrovsky (FC 2007), Bond (CHES 2001). Our solution is called *salted-PIN*: a randomly generated salt value of adequate length (e.g., 128-bit) is stored on a bank card in plaintext, and in an encrypted form at a verification facility under a bank-chosen *salt key*. Instead of sending the regular user PIN, salted-PIN requires an ATM to generate a *Transport Final PIN* from a user PIN, account number, and the salt value (stored on the bank card) through, e.g., a pseudo-random function. We explore different attacks on this solution, and propose variants of salted-PIN that can protect against known attacks. Depending on the solution variation, attacks at a malicious intermediate switch now may only reveal the Transport Final PIN; both the user PIN and salt value remain beyond the reach of an attacker's switch. Salted-PIN requires modifications to service points (e.g., ATM, point-of-sale), issuer/verification facilities, and bank cards; however, changes to intermediate switches are not required.

1 Introduction

Attacks on financial PIN processing APIs revealing customers' PINs have been known to banks and security researchers for years, e.g., [10], [6], [8], [9], [7] (failure modes of ATM PIN encryption were first discussed in Anderson [2]). Apparently the most efficient of these *PIN cracking* attacks are due to Berkman and Ostrovsky [4].[1] However, proposals to counter such attacks are almost non-existent in the literature, other than a few suggestions; for example, maintaining the secrecy (and integrity) of some data elements related to PIN processing (that are considered security insensitive according to current banking standards) such as the "decimalization table" and "PIN Verification Values (PVVs)/Offsets" has been emphasized [8], [4]. However, implementing these suggestions requires modifications to all involved parties' Hardware Security Modules (HSMs). Commercial solutions such as the `PrivateServer Switch-HSM` [1] rely mostly on *tightly* controlling the key uploading process to a switch and removing unnecessary APIs or weak PIN block formats. Even if the flawed APIs are fixed, or non-essential attack APIs are removed to prevent these attacks, it may be difficult in practice to ensure that all intermediate (third-party controlled) switches are updated accordingly. Thus banks rely mainly on protection mechanisms provided within banking standards, and *policy-based* solutions, e.g., mutual banking agreements to protect customer PINs.

A solution such as Mobile Password Authentication (MP-Auth) [14] is apparently capable of preventing these attacks in addition to saving PINs from false ATM keypads and card reader attacks. However, MP-Auth relies on public key operations, and thus cannot be deployed without significant modifications to

*Version: March 31, 2009. Contact author: `mmannan@scs.carleton.ca`. A 6-page version of this manuscript appeared as a short paper in Financial Cryptography and Data Security (FC) 2008 [16].

[1] We encourage readers unfamiliar with financial PIN processing APIs and PIN cracking attacks to consult Section 2 for background, and Appendix A for a summary of attacks by Berkman and Ostrovsky [4].

ATMs, switches and verification facilities. Another obvious solution (as suggested in [8], [4]) is to update the PIN processing APIs, which also requires modifications to all involved parties' Hardware Security Modules (HSMs).[2]

Designing solutions to mitigate PIN cracking attacks pose some interesting challenges. PIN transfers in banking networks rely on symmetric key cryptography where the third-party controlled intermediate switches also possess shared keys to decrypt encrypted PINs (but have no access to issuer/verification keys). Although decrypted PINs (and the decryption key itself) are not (ideally) accessible from outside of an HSM, API flaws allow attackers to realistically extract enough information from the HSM (through legitimate API calls) to enable PIN cracking attacks. Thus PIN cracking solutions must protect user PINs that travel through third-party switches which may be less security conscious or even actively malicious. Our solution attempts to address threats from such an adversary as well as hostile parties at a verification facility with limited access (e.g., one who can call API functions from an HSM, but cannot access verification keys). However, we do not consider ATM frauds that are not scalable such as false keypads and card reader attacks [11].

One primary reason that PIN cracking attacks are possible is that actual user PINs, although encrypted, travel from ATMs to a verification facility through several (untrustworthy) intermediate switches. If, for example, hashed PINs were sent in an encrypted form, attackers may not be able to reveal user PINs even in the presence of API flaws. However, as PINs are generally short (4 digits), an offline dictionary attack may still easily allow recovery of actual PINs. From reviewing the history of API attacks, we also note that even a complete overhauling of PIN processing APIs may be subject to presently-unknown API flaws that might be exploited to reveal user PINs. Therefore we seek a solution that precludes real user PINs being extracted at verification facilities, and especially at switches (which are beyond the control of issuing banks), even in the presence of API flaws. One possible solution in this direction is not to send the actual user PIN itself through untrusted intermediate nodes. Our proposal follows such a direction.

While PIN cracking attacks get more expensive as the PIN length increases, it is unrealistic to consider larger (e.g., 12-digit) user PINs, for usability reasons.[3] As part of our proposal, we assume that a unique random *salt* value of sufficient length (e.g., 128 bits) is stored on a user's bank card, and used along with the user's regular four-digit PIN (*Final PIN*) to generate[4] a larger (e.g., 12 digit) *Transport Final PIN* (TFP). This TFP is then encrypted and sent through the intermediate switches. Thus we essentially expand the 4-digit PIN to 12 digits. We build our *salted-PIN* solution on this simple idea. Our proposal requires updating bank cards (magnetic-stripe/chip card), service-points (e.g., ATMs), and issuer/verification HSMs. However, our design goal is to avoid changing any intermediate switches, or requiring intermediate switches be trusted or compliant to anything beyond existing banking standards.

Salted-PIN provides the following benefits.

1. It does not depend on policy-based assumptions, and limits existing PIN cracking attacks even where intermediate switches are malicious.
2. It significantly increases the cost of launching known PIN cracking attacks; for example, the setup cost for the translate-only attack (see Appendix A) for building a complete Encrypted PIN Block (EPB) table now requires more than a trillion API calls in contrast to 10,000 calls as in Berkman and Ostrovsky [4].
3. Incorporating service-point specific information such as "card acceptor identification code" and "card acceptor name/location" (as in ISO 8583) into variants of salted-PIN, we further restrict attacks to be limited to a particular location/ATM.

Organization. Background on financial PIN processing is provided in Section 2. We outline the proposed salted-PIN solution in Section 3. Known attacks that apply to the basic version of salted-PIN are discussed in Section 4. In Section 5 we introduce three variants of salted-PIN to counter these attacks. Implementation challenges to salted-PIN are also briefly discussed in Section 5. Section 6 concludes. In Appendix A, we review several (representative) attacks as outlined by Berkman and Ostrovsky [4].

[2] For an overview of HSMs and related attacks, see Anderson et al. [3]

[3] A 12-digit PIN can be constructed by storing eight digits on the bank card while a user memorizes the other four digits as usual. However, as the real PIN is sent encrypted in this solution, attackers at a malicious switch can recover the PIN and create fake cards. (An anonymous FC 2008 referee pointed us to this idea and its relative advantages and disadvantages.)

[4] For example, through a pseudo-random function (PRF).

2 Background

In this section, we provide a basic overview of PIN processing and PIN block formats. More background on banking networks is discussed elsewhere (e.g., [10], [17]).

PIN Processing Architecture. When a user inputs her PIN at an ATM, the PIN is encrypted to form an Encrypted PIN Block (EPB) using a transport key shared between the ATM and the next switch connected to the ATM. A switch can be a stand-alone facility for PIN transportation (and other related bank network activities), or part of a bank's verification facility. PIN blocks are processed inside Hardware Security Modules (HSMs). Each switch shares a transport key with other switches that it is connected to. At a verification center, a switch may also have the issuer key (for PIN verification). A standardized set of PIN processing APIs is used for PIN creation, transportation, and verification. The intent is that this allows banks to protect user PINs from application programmers (or anyone having access to PIN processing APIs) at verification facilities as well as in switches.

There are several standardized PIN block formats (see below). An EPB may travel across several HSMs on its way to a verification site. When transmitted from one HSM to another, re-formatting (i.e., translating from one PIN block format to another) may be required. Thus all HSMs must implement translation APIs to allow reformatting of an EPB. A switch decrypts an EPB, checks the PIN block format (e.g., validity of PIN digits, PIN length), changes the format if required, and re-encrypts the PIN block with the destination switch's transport key. As all PIN operations are performed by HSMs, an application programmer (ideally) cannot learn anything about PINs transported as EPBs.

PIN Block Formats. We outline four PIN block formats from ISO 9564-1 [12], three of which are approved by VISA for online transactions (e.g., through ATMs). Assume that a PIN is four decimal digits long. A PIN block is composed of 16 hex digits, i.e., 64-bits. Let 'P' be a PIN digit (0 to 9), PAN the least significant 12 digits of a customer's Primary Account Number (excluding the check digit), and let 'A' be a PAN digit (0 to 9). An ISO-0 PIN block is calculated as follows.

$$\begin{aligned} \text{ISO-0 PIN Block} &= \text{Original PIN Block} \oplus \text{Formatted PAN} \\ \text{Here, Original PIN Block} &= \texttt{04 PPPP FFFF FFFF FF}, \\ &\quad \text{with 'F' denoting the hex digit F} \\ \text{Formatted PAN Block} &= \texttt{00 00AA AAAA AAAA AA} \end{aligned}$$

The leftmost zero in the original PIN block stands for ISO-0, and the digit 4 is the PIN length (which could be as high as 12). An ISO-0 PIN block is the result of XORing an original PIN block with a formatted PAN. The ISO-1 PIN Block format is `14 PPPP RRRR RRRR RR`, where 'R' is a random hex digit (0 to F). The ISO-2 PIN Block format is `24 PPPP FFFF FFFF FF`, which is used only when creating a card. An ISO-3 PIN block is calculated as follows.

$$\begin{aligned} \text{ISO-3 PIN Block} &= \text{Formatted PIN Block} \oplus \text{Formatted PAN} \\ \text{Here, Formatted PIN Block} &= \texttt{34 PPPP RRRR RRRR RR}, \\ &\quad \text{with 'R' a hex digit from A to F} \\ \text{Formatted PAN Block} &= \texttt{00 00AA AAAA AAAA AA} \end{aligned}$$

In summary, ISO-2 is the weakest PIN format; it is not allowed for online processing, and it has not been used in the PIN cracking attacks. ISO-0 and ISO-3 PIN blocks depend on a user PIN and account number. The ISO-1 format is not bound to a user's account number, and is recommended to be used in situations where the PAN is unavailable. Attacks exploiting translate-only APIs (see Section A.1) depend on the fact that any ISO-0 and ISO-3 PIN formats can be translated to the less secure ISO-1 format (as the ISO-1 format does not depend on the user PAN). Translation APIs are also generally implemented by all HSMs.

IBM Calculate-Offset API. IBM's calculate-offset API outputs an offset using a PAN and EPB. If the calculated offset value corresponds to the stored value for that PAN, then the PIN inside the EPB is verified. Offset values are assumed by the banking standards to be security insensitive, and are generally stored in plaintext. Fig. 1 illustrates how an offset value is calculated for PIN verification. Here, a *Natural PIN* is calculated from a customer's PAN, and the *Final PIN* is a customer-chosen PIN. Subtraction is digit by

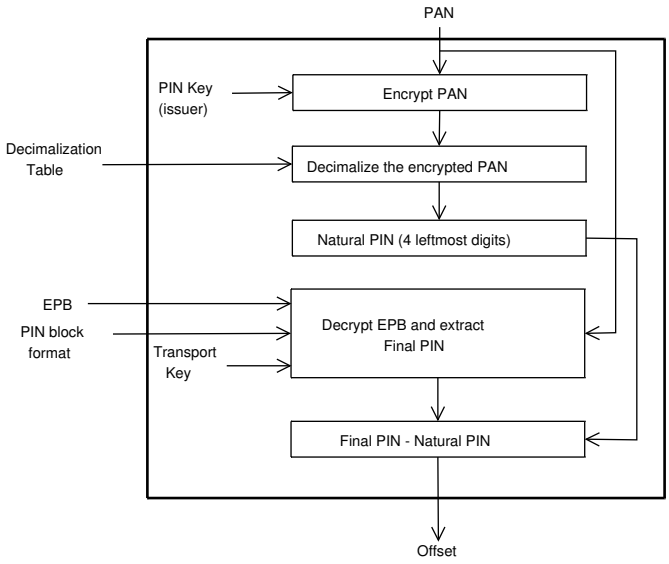

Figure 1: Offset calculation (adapted from [17])

digit modulo 10. An issuer key (residing inside an HSM) is used to encrypt a user's PAN. The encrypted PAN may contain hex digits (A to F), and it is decimalized using a decimalization table (mapping hex digits to decimal digits). The four left-most digits of the decimalized encrypted PAN constitute the user's Natural PIN. The Final PIN is extracted from the user's PAN, EPB (containing the user's encrypted Final PIN), the PIN block format, and the transport key (residing inside the HSM). The offset is calculated by subtracting the Natural PIN from the Final PIN.

VISA PIN Verification Value (PVV). Fig. 2 depicts how a VISA PIN Verification Value (PVV) is calculated. PVVs are used in a similar fashion as IBM offset values, and also (generally) stored in a plaintext database. A customer's PVV may be written on her bank card as well (for offline PIN verification).

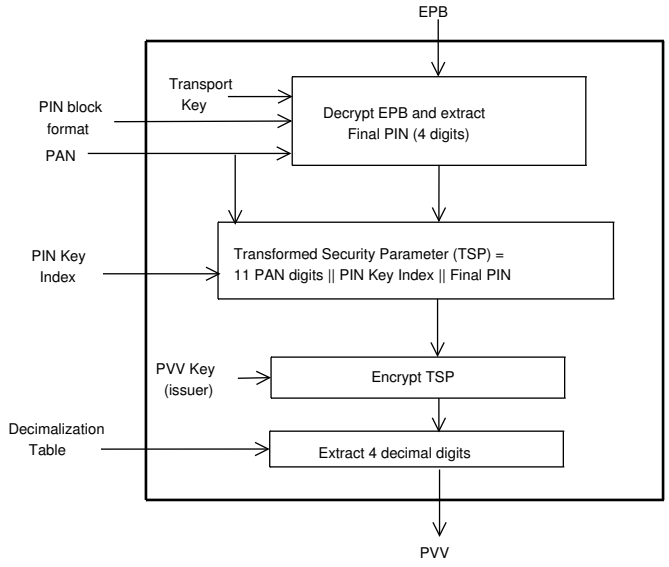

Figure 2: PVV calculation (adapted from [17], || denotes concatenation)

3 Salted PIN

Here we present the salted-PIN proposal in its simplest form.

Threat model and notation. Our threat model assumes attackers have access to PIN processing APIs and transaction data (e.g., Encrypted PIN Blocks, account number) at switches or verification centers, but do not have direct access to keys inside an HSM, or modify HSMs in any way. Attackers can also create fake cards from information extracted at switches or verification centers and use those cards (perhaps through outsider accomplices). We primarily consider large scale attacks such as those that can extract millions of PINs in an hour [4]. We do not address attacks that are not scalable, such as *card skimming*, attacks on EMV[5] PIN entry devices [11], or cases where an accomplice steals a card and calls an insider at a switch or verification center for an appropriate PIN. PIN cracking attacks that we consider are successful only when online PIN verification is applied (i.e., encrypted PINs are sent to a verification center for approval). In addition to magnetic-stripe cards, these attacks are also valid for chip/EMV cards except when offline/on-chip PIN verification is used (assuming card issuers allow EMV cards to fallback to magstripe processing for backward compatibility or chip failure). Note that, if and when chip cards are deployed worldwide, and offline PIN verification is the de facto mode of operation, most current PIN cracking attacks involving intermediate switches become invalid, consequently eliminating the need for new solutions. Although such cards are being gradually adopted, apparently magstripe cards (and the magstripe mode of operation of chip cards) along with existing flawed APIs will remain in operation for a long time to come. For example, we have seen that the transition away from DES and triple-DES to more modern cryptographic algorithms has taken far longer than many might have originally predicted. The following notation is used:

PAN	User's Primary Account Number (generally 14 or 16-digit).
PIN	User's Final PIN (e.g., 4-digit, issued by the bank or chosen by the user).
PIN_t	User's Transport Final PIN (TFP).
$Salt$	Long-term secret value shared between the user card and issuing bank.
$f_K(\cdot)$	A cryptographically secure Pseudo-Random Function (PRF).[6] Here K is the PRF key.

Generating salted-PINs. A randomly generated salt value of adequate length (e.g., 128 bits, to make dictionary attacks infeasible) is selected by a bank for each customer. The salt is stored on a bank card (chip-card or magstripe) in plaintext, and in an encrypted form at a verification facility under a bank-chosen *salt key*. API programmers (i.e., those who use HSM API) have access to this encrypted salt (but do not know the salt key or plaintext salt values). Encrypted salt values also cannot be overwritten by API programmers. A user inputs her PIN at an ATM, and the ATM reads the plaintext salt value from the user's bank card and generates a Transport Final PIN (TFP) as follows.

$$PIN_t = f_{Salt}(PAN, PIN) \tag{3.1}$$

The PRF output is interpreted as a number and divided by 10^{12}; the 12-digit remainder (i.e., PRF output modulo 10^{12}) is chosen as PIN_t and treated as the Final PIN from the user. Note that the maximum allowed PIN length by ISO standards is 12. The ATM encrypts PIN_t with the transport key shared with the adjacent switch, and forms an Encrypted PIN Block (EPB). An intermediate switch decrypts an EPB, (optionally) reformats the PIN block, and re-encrypts using the next switch's transport key. Additional functionalities are not required from these switches.

To set the initial offset or PIN verification value (PVV), an issuer generates a random PIN (e.g., 4 digits long) and salt for a user, and then uses equation (3.1) to generate PIN_t. The transport key of the verification HSM is used to encrypt PIN_t and form an EPB. This EPB is used to call a calculate offset/PVV function with the user's PAN and encrypted salt to generate the initial offset/PVV (note that each of these values is now 12 digits long).

Offset/PVV verification with salted-PIN. The salted-PIN verification for the IBM offset method (recall Section 2) is shown in Fig. 3. The Natural PIN is calculated from a PAN using an issuer's PIN key. The

[5] EMV is a growing standard for chip-based bank cards, initially developed by Europay, MasterCard, and VISA; see `http://www.emvco.com`.

[6] For example, as used in `PwdHash` [18].

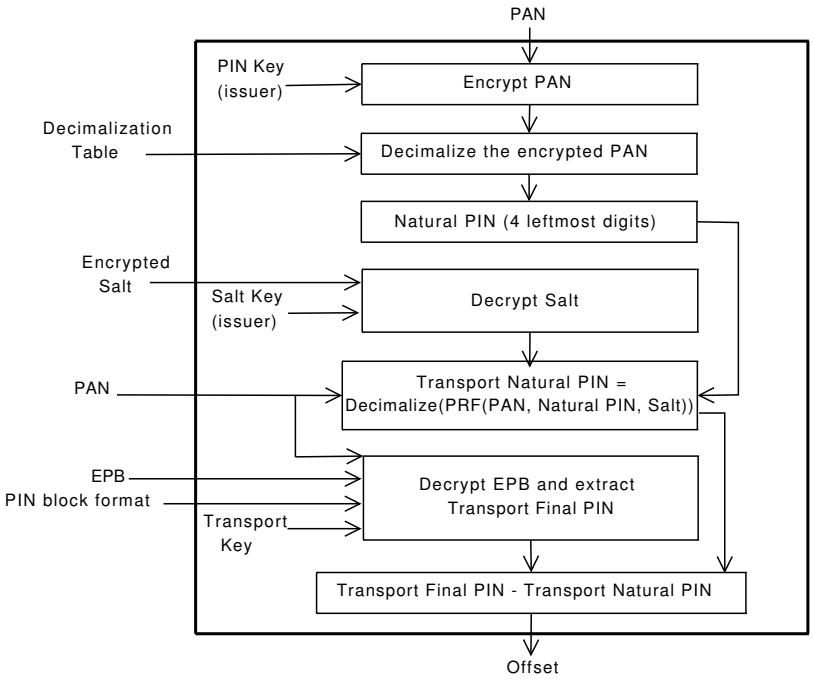

Figure 3: Salted-PIN verification for the IBM offset method

encrypted salt value corresponding to the PAN is decrypted using a salt key (like the PIN key and transport key, the salt key also resides inside an HSM). The Transport Natural PIN is generated from the Natural PIN using equation (3.1). The Transport Final PIN is extracted from an EPB, and the Transport Natural PIN is subtracted from it (digit by digit modulo 10 subtraction) to get the offset. This calculated offset value is compared with the corresponding PAN's stored (e.g., in a database) offset value. The salted-PIN verification for VISA PVV is shown in Fig. 4. The salt value is appended at the end of the Transformed Security Parameter (TSP), which is encrypted and decimalized to calculate the PVV. Note that we design the offset/PVV verification functions to keep them similar to the existing functions although these can be further simplified; for example, instead of storing offset/PVV values, EPBs directly may be stored and compared with incoming EPBs.

Salted-PIN protection against PIN cracking attacks. We discuss attacks (e.g., translate-only [4]) that reveal a user's TFP in Section 4. An attacker with write-access to the PVV database at a verification facility can choose any PIN for a specific account (see Section A.3). With the salted-PIN solution, an attacker can still choose any PIN to pack in an EPB and write the resulting PVV to a database. However, without knowing the salt value, overwriting a user's PVV does not help in an attack for the following reason. The salted-PIN verification function for PVV (Fig. 4) ensures use of the encrypted salt value as indexed by a user's PAN; thus for a successful PVV verification, a user's salt must be known or the encrypted salt value must be replaced.

4 Attacks on Salted-PIN

We now discuss attacks against the basic version of salted-PIN.

4.1 Enumerating EPBs through Translate-only Attacks

Here the goal of an attacker is to create a table of EPBs, and then crack all or a subset of user accounts. The following attacks in part follows an efficient variant of the translate attack as outlined by Berkman and Ostrovsky [4]. For these attacks, we assume an attacker M_i is an insider (e.g., application programmer) at a switch or verification center, and an outsider accomplice M_a who helps M_i in carrying out user input at an ATM. These attacks are possible for the following reason. Although a TFP is calculated from a long (e.g., 128 bits, sufficient to deter dictionary attacks) salt value, only 12 digits of the PRF output are used.

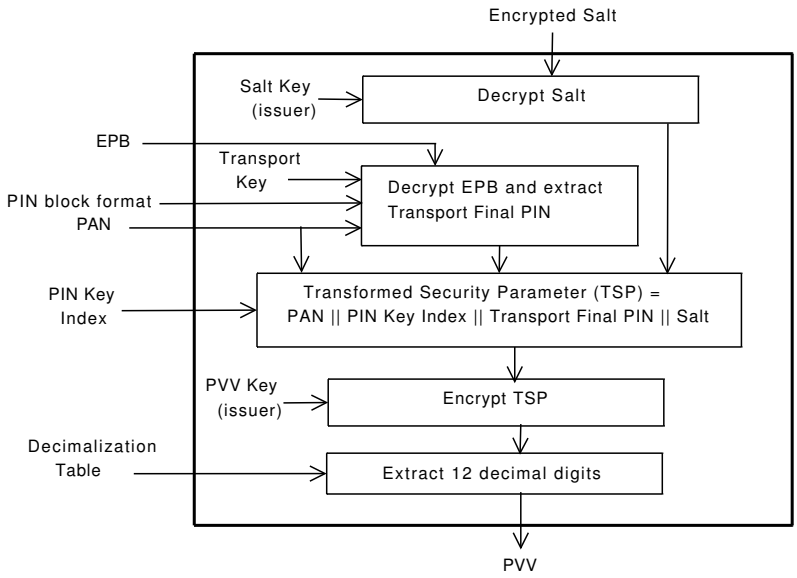

Figure 4: Salted-PIN verification for VISA PVV

Thus an attacker only requires *any* pair of salt and PIN combination that can generate a targeted account's TFP instead of finding the actual salt/PIN values.

Targeting all accounts. Assume that M_i extracts the salt value ($Salt_a$) and PAN from a card he possesses, and uses equation (3.1) to generate the 12-digit TFP PIN_{at} (through software or a hardware device, using any PIN PIN_a). Let PIN_{at} consist of $p_1p_2p_3 \dots p_{12}$ where each p_i ($i = 1$ to 12) is a valid PIN digit. Then M_a inserts this card to an ATM, and enters PIN_a. Assume that the generated PIN_{at} is encrypted by the ATM to form an EPB, E_1. M_i captures E_1 at a switch. If E_1 is not in the ISO-1 format, M_i translates it into ISO-1 (to disconnect E_1 from the associated PAN). Let the translated (if needed) E_1 in the ISO-1 format be E_1'. E_1' is then translated from ISO-1 to ISO-0 using $p_3p_4 \dots p_{12}00$ as the input PAN. This special PAN is chosen so that the XOR of PIN positions 3 to 12 with PAN positions 1 to 10 removes $p_3 \dots p_{12}$ when the translation API is called; i.e.,

PIN block inside $E_1' = 0$	C	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	F	F
Input PAN $= 0$	0	0	0	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	0	0
Resulting ISO-0 PIN block $= 0$	C	p_1	p_2	0	0	0	0	0	0	0	0	0	0	F	F

Assume the resulting EPB is $E_{p_1p_2}$ which is the same as the one containing a TFP $p_1p_20000000000$ with PAN 0. Now we can create all EPBs containing every 12 digit TFPs starting with p_1p_2 from $E_{p_1p_2}$. For example, an EPB with $p_1p_2q_3q_4 \dots q_{12}$ as the TFP can be generated through transforming $E_{p_1p_2}$ using PAN $q_3q_4 \dots q_{12}00$ (in ISO-0). Thus we can create all 10^{10} EPBs with TFPs from $p_1p_20 \dots 0$ to $p_1p_29 \dots 9$.

Starting from a different p_1p_2, all 10^{12} EPBs containing every 12 digit TFP can be generated as follows. M_a uses the previous bank card (i.e., the same salt and PAN) with different PINs (obviously, including wrong PINs) to calculate TFPs using software or a special device. When a TFP is found with the first two digits different than p_1p_2, the corresponding PIN is entered at an ATM. The attacker M_i at the switch then generates another set of 10^{10} EPBs containing TFPs starting with this different p_1p_2. The attack continues with different PINs until all 100 possible values of the initial two TFP digits are covered. Thus using these 100 EPBs containing TFPs starting with the different first two digits (i.e., from 00 to 99), M_i can create a table of EPBs for all possible TFPs (with corresponding PINs). The cost of building this table is slightly over 10^{12} API calls (for each 100 $E_{p_1p_2}$, at most two API calls are required). The cost of selecting the initial EPBs (i.e., that contain TFPs with two different starting digits) is insignificant as M_a can calculate TFPs offline, i.e., without involving any API calls to HSMs.

To launch an attack, a valid EPB of a target customer is collected. The EPB is translated to ISO-1 (to decouple it from the target account, if not already in ISO-1), then to ISO-0 with PAN 0. The resulting EPB is then located on the EPB table (as created in the setup phase). The corresponding PIN from the table can

now be used to exploit a card generated with the target's PAN, and the attacker's salt value (i.e., $Salt_a$). The cost of this attack is at most two API calls and a search of $O(10^{12})$, i.e., $O(2^{40})$.

In summary, the setup cost of this attack is about 10^{12} API calls with a per account cost of two API calls plus a search of $O(10^{12})$. The same translate-only attack by Berkman and Ostrovsky [4] on the current implementation of PIN processing requires only about 10,000 API calls as setup cost, and a per account cost of two API calls plus a search of $O(10^3)$.

Algorithm 1 Steps in the partial table attack

1: **for** $i = 0$ to $10^6 - 1$ **do**
2: $E_{c0} = \mathsf{Translate}_{\mathsf{ISO-0}}(E_c, i \times 100)$
3: **if** E_{c0} is in the table **then**
4: TFP in $E_c = 10^6 \times$ (six digit TFP from the table) $+ i$
5: Salt and PIN values corresponding to E_c is used to generate a fake card
6: exit
7: **end if**
8: **end for**

Trade-off between table size and per EPB attack cost. The per account cost of the above attack is not high enough to deter an attack. However, the setup cost of building the table with all one trillion EPBs is apparently significant (although this is a one-time cost). By reducing the table size, the attack can be launched with fewer API calls although the per EPB attack cost increases accordingly.

Assume that the attacker builds a table of 10^6 EPBs (i.e., one half of the original table size) containing TFPs ending with six zeros (000000), i.e., storing only the first six digits of a TFP. With this table, an attacker can calculate TFP of any target EPB E_c in 10^6 steps (assuming the EPB arrives in the ISO-1 format, or the attacker translates it into ISO-1); each step then requires one API call. The attack is described in Algorithm 1.

Now the cost of attacking N accounts is $10^6 + N \times 10^6$ API calls. The attacker can also vary the table size and the number of steps for each target account. For any table size 10^n for $n \in \{2, 3, \ldots, 12\}$, the required number of per account translate steps is 10^{12-n}. Thus in general the cost of attacking N accounts is $10^n + N \times 10^{12-n}$.

4.2 Replay Attack

In this attack, an adversary M_i at a switch or verification center collects a valid EPB E_c for a target PAN A_c, and then creates a fake card with the account number A_c (and any salt value). Note that M_i here does not know the actual salt value or PIN for the target account. An accomplice M_a uses the fake card with any PIN at an ATM, and the ATM generates a false EPB E_a. At the switch/verification center M_i locates E_a in transfer, and replaces E_a with the previously collected correct EPB E_c. Thus the fake card will be verified by the target bank, and M_a can access the victim's account.

Note that this attack works against the basic variant of salted-PIN as well as current PIN implementations without requiring any API calls. Although quite intuitive, this attack has not been discussed elsewhere to our knowledge.

5 Variants, Implementation Challenges and Lessons Learned

As we discussed in Section 4, the basic version of salted-PIN is vulnerable to several attacks. Other than the replay attack, the setup cost of launching these attacks is not trivial as previous PIN cracking attacks (cf. [4]) although the per account attack cost is apparently manageable. In this section, we outline three variants of salted-PIN to practically restrict these attacks by increasing the per account attack cost.

Variant 1: Service-point specific salted-PIN. If a fake bank card is created for a target account (e.g., through the attacks in Section 4), the card can be used from anywhere as long as it remains valid

(i.e., the issuing bank does not cancel it). To restrict such attacks, we modify equation (3.1) as follows.

$$PIN_t = f_{Salt}(PAN, PIN, spsi) \tag{5.1}$$

Here *spsi* stands for *service-point specific information* such as a "card acceptor identification code" and "card acceptor name/location" as in ISO 8583 (Data Elements fields). The verification center must receive *spsi* as used in equation (5.1). Although any PIN cracking attack (Section 4.1) can be used to learn a TFP or build a full/partial EPB table, the table is valid only for the particular values of *spsi*. Also, the replay attack (Section 4.2) may succeed only when the accomplice exploits a compromised card from a particular ATM. Thus this construct generates a localized TFP for each PIN verification, and thereby restricts the fake card to be used only from a particular location/ATM. Note that for this variant, the verification facility cannot use PVV or Offset values, because they would be different for each ATM. Another verification value would need to be designed.

Variant 2: Salted-PIN with double EPBs. ISO PIN block formats restrict PIN length to 12 digits in an EPB. This length limit enables a search of $O(2^{40})$ in a pre-built table (see in Section 4.1). As a variant, instead of choosing 12 digits from the result of equation (3.1), we can take 24 digits (i.e., PRF output modulo 10^{24}) and create two PIN_t blocks, each 12 digits long. As a result, two EPBs must be sent from an ATM, and a verification facility needs both EPBs to verify a user's PIN. However, intermediate switches may not need to be aware of this. An attack similar to Section 4.1 can be launched on each EPB separately, and two tables can be built for both parts of a 24-digit TFP; the cost of building the table simply doubles (two TFP tables, each has 10^{12} entries). Using the tables, a 24-digit TFP can be extracted from the two EPBs of any target account. However, determining a valid pair of salt value and PIN is not straightforward as the attack in Section 4.1. To generate a fake card (i.e., to find an appropriate salt value and PIN for the intended TFP) for this variant of salted-PIN, attackers must apparently carry out a computation of 10^{24} (i.e., $O(2^{80})$) steps. However, this variant is vulnerable to the replay attack (Section 4.2) when equation (3.1) is used. Again, service-point specific information as used in equation (5.1) for generating TFP can practically limit such attacks.

Variant 3: End-to-end PIN encryption/MAC. Using the stored salt as an encryption key, end-to-end PIN encryption can be achieved between an ATM and verification center. The salt value can also be used for calculating a message authentication code (MAC) for a user's Final PIN. This variant can secure PIN transportation to the extent of the algorithm used for encryption or MAC. Thus it can effectively eliminate PIN enumeration by an attacker at a switch or verification center. However, to restrict the replay attack (Section 4.2), one or more service-point specific items must be used with a PIN for encryption or MAC. Also, this variant will require updating intermediate switches.

Implementation challenges. One implementation challenge for salted-PIN could be the storage requirement for the salt (39 decimal digits or 128 bits) that must be stored on a bank card. There are four possible scenarios: (1) magnetic-stripe (magstripe) cards; (2) chip-card with a magnetic stripe at a magstripe reader terminal; (3) chip-card with online PIN verification; and (4) chip-card with offline PIN verification. For the last case, as a PIN does not leave the card, PIN cracking attacks are immaterial. For the first two cases, the amount of data that can be stored on a magnetic stripe is limited by ISO standards; for example, according to ISO-7811, track one in a magstripe bank card holds 79 six-bit characters (plus a parity check), and track two holds 40 four-bit (plus a parity) characters. These two tracks are generally present in most magstripe bank cards (there is also a third track on some cards). A salt may be stored on a magstripe card by overloading non-essential data fields in track one (e.g., discretionary data, name, expiration date), and redundant fields in track two (e.g., PAN). Chip-cards offer significantly more storage capability, and thus for the third case, accommodating the salt may not be an issue.

Salted-PIN requires that service points (e.g., ATMs, point-of-sale terminals) are capable of computing PRF as in equation (3.1). Thus another implementation challenge is posed by the limited computing ability of old magstripe reader terminals with limited CPU capabilities and cryptographic support of only a DES chip; recent terminals (e.g., Motorola's PD4750) generally operate on a 32-bit processor, and computing a PRF is not a computational issue.

Lessons learned and discussion. Now we briefly discuss the lessons learned from designing different variants of salted-PIN. These lessons, we believe, may help others in building robust security protocols.

1. **Minimizing disclosure of reusable information.** In the banking network, encrypted user PINs are sent through multiple switches to the verification center for user authentication. Such a scheme always bears the risk of exposing the long-term, reusable secret PIN. We argue that if long-term secrets are converted to one-time use passcodes, then the discovery of a flaw may not necessarily lead to the compromise of a reusable secret. Some techniques such as Lamport's hash chain [13] have been publicly known for decades. Unfortunately, applications of these schemes appear to be low.

2. **Reducing the value of disclosed information.** In general, currently attackers enjoy the benefit of compromising sensitive secrets once, and then reusing those multiple times. Localization of secrets or sensitive information as applied in the service-point specific salted-PIN variant, may also be useful in other settings, e.g., restricting identity fraud as a result of data breaches [15]. Making attacks *unattractive* (i.e., the reward is less than the required efforts) is an easier goal than making attacks *impossible*, and is often effective and sufficient. As defenders (such as API designers) and attackers are both humans, it makes little sense, at least on a philosophical ground, to believe that defenders can design protocols or techniques that cannot be defeated one way or another. However, we can "design for damage control", i.e., design protocols in such a way that when they break, they still do not expose long-term user secrets. Incorporating such damage control techniques into the design itself may make our protocols more resilient to attacks.

6 Conclusion

In the 30-year history of financial PIN processing APIs, several flaws have been uncovered. In this paper, we summarize some API attacks from Berkman and Ostrovsky [4] for context, and introduce a *salted-PIN* proposal and three of its variants to counter these attacks. Our preliminary analysis in this paper indicates that salted-PIN can provide a higher barrier to these attacks in practice by making them considerably more expensive (computationally). We have discussed some deployment issues, but acknowledge that this discussion is not exhaustive; deployment barriers may arise from unseen aspects. Salted-PIN is motivated primarily by the realistic scenario in which an adversary may control switches, and use any standard API functions to reveal a user's PIN; i.e., an attacker has the ability to perform malicious API calls to HSMs, but cannot otherwise modify an HSM.

Our proposal of salted-PIN is intended to stimulate further research and solicit feedback from the banking community regarding: (1) whether salted-PIN may improve PIN security in real terms; (2) practical barriers of deploying salted-PIN; and (3) any significant weaknesses of salted-PIN. We focus on providing a technical solution to update PIN processing APIs, some of which are well-known to be flawed. Instead of relying, perhaps unrealistically, on honest intermediate parties (who diligently comply with mutual banking agreements), we strongly encourage the banking community to invest effort in designing protocols that do not rely on such assumptions which end-users (among others) have no way of verifying. It has been speculated [4] that PIN cracking attacks may explain numerous unexplained *phantom* withdrawals [5] as reported by many ATM fraud victims. As reported [19] recently (June 20, 2008), the compromise of a third-party PIN processor may have been the reason for a large number of Citibank card fraud.

Acknowledgements

This work benefited substantially from discussion and/or feedback from a number of individuals, including: Bernhard Esslinger of University of Siegen, Joerg-Cornelius Schneider and Henrik Koy of Deutsche Bank, especially regarding attacks on the simple version of salted-PIN; a reviewer from a large Canadian bank; Glenn Wurster; and anonymous reviewers. The first author is supported in part by an NSERC CGS. The second author is Canada Research Chair in Network and Software Security, and is supported in part by an NSERC Discovery Grant, the Canada Research Chairs Program, and NSERC ISSNet.

References

[1] Algorithmic Research (ARX). PrivateServer Switch-HSM. White paper. `http://www.arx.com/documents/Switch-HSM.pdf`.

[2] R. Anderson. Why cryptosystems fail. *Communications of the ACM*, 37(11), Nov. 1994.

[3] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors – a survey. *Proceedings of the IEEE*, 94(2), Feb. 2006. Invited paper.

[4] O. Berkman and O. M. Ostrovsky. The unbearable lightness of PIN cracking. In *Financial Cryptography and Data Security (FC)*, volume 4886 of *LNCS*, Scarborough, Trinidad and Tobago, Feb. 2007.

[5] M. Bond. Phantom withdrawals: On-line resources for victims of ATM fraud. `http://www.phantomwithdrawals.com`.

[6] M. Bond. Understanding security APIs. Ph.D. Thesis, Computer Laboratory, University of Cambridge, 2004.

[7] M. Bond. Attacks on cryptoprocessor transaction sets. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Paris, France, May 2001.

[8] M. Bond and P. Zielinski. Decimalisation table attacks for PIN cracking. Technical report (UCAM-CL-TR-560), Computer Laboratory, University of Cambridge, 2003.

[9] M. Bond and P. Zielinski. Encrypted? Randomised? Compromised? (When cryptographically secured data is not secure). In *Workshop on Cryptographic Algorithms and their Uses*, Gold Coast, Australia, July 2004.

[10] J. Clulow. The design and analysis of cryptographic APIs for security devices. Masters Thesis, University of Natal, Durban, South Africa, 2003.

[11] S. Drimer, S. J. Murdoch, and R. Anderson. Thinking inside the box: System-level failures of tamper proofing. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2008.

[12] International Organization for Standardization (ISO). Banking – Personal Identification Number (PIN) management and security – Part 1: Basic principles and requirements for online PIN handling in ATM and POS systems, Apr. 2002. International Standard, ISO 9564-1.

[13] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11), Nov. 1981.

[14] M. Mannan and P. van Oorschot. Using a personal device to strengthen password authentication from an untrusted computer. In *Financial Cryptography and Data Security (FC)*, volume 4886 of *LNCS*, Scarborough, Trinidad and Tobago, Feb. 2007.

[15] M. Mannan and P. van Oorschot. Localization of credential information to address increasingly inevitable data breaches. In *New Security Paradigms Workshop (NSPW)*, Lake Tahoe, CA, USA, Sept. 2008.

[16] M. Mannan and P. van Oorschot. Weighing down "The Unbearable Lightness of PIN Cracking" (short paper). In *Financial Cryptography and Data Security (FC)*, volume 5143 of *LNCS*, Cozumel, Mexico, Jan. 2008.

[17] O. M. Ostrovsky. Vulnerabilities in the financial PIN processing API. Masters Thesis, Tel Aviv University, 2006.

[18] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security*, 2005.

[19] Wired.com. Citibank replaces some ATM cards after online PIN heist – update. Blog article (June 20, 2008). `http://blog.wired.com/27bstroke6/2008/06/citibank-issues.html`.

A Review of Earlier PIN Cracking Attacks

For convenience to the reader and for reference within, here we summarize several representative attacks from Berkman and Ostrovsky [4]. For reasons of brevity, we omit how some specific assumptions required by these attacks are met, as well as any efficiency analysis of these attacks (e.g., how many API calls are required for a given attack to succeed).

A.1 Translate PIN Block Attacks

We review the translate-only API attack which requires an attacker to generate/collect Encrypted PIN Blocks (EPBs) of all possible PINs, and access to the translate API function. This attack reveals plaintext PINs, and can be applied at a switch or verification facility. The steps in the attack are as follows.

1. Let A_x be any attacker chosen PAN.
2. Attackers collect/generate $10,000$ EPBs which pack all possible PINs in any ISO format (i.e., the format and PAN of those EPBs are immaterial). Suppose i is any 4-digit PIN, and E'_i packs i in any ISO format.
3. Translate all $10,000$ EPBs to ISO-0 EPBs using A_x as the PAN. Assume E_i is the resulting EPB from the translation API.

   $$E_i = \mathsf{Translate}_{\mathsf{ISO}-0}(E'_i, A_x), \text{where } i \in \{0000 \ldots 9999\}.$$

   Now E_i packs PIN i in the ISO-0 format (with respect to A_x). Make a table with the resulting EPBs and PINs, i.e., (E_i, i).
4. For any customer EPB, E_c, calculate

   $$E_t = \mathsf{Translate}_{\mathsf{ISO}-0}(\mathsf{Translate}_{\mathsf{ISO}-1}(E_c), A_x).$$

   Here, an attacker first converts the customer EPB to ISO-1 (which unlinks a PIN with the corresponding customer PAN), and then uses this result with the attacker's chosen PAN to generate an EPB in ISO-0 format.
5. Locate E_t in the table generated at step 3. The corresponding PIN is the PIN packed inside E_c.

A.2 Attacks Exploiting the IBM Calculate-Offset API

The steps in the IBM Calculate-Offset attack at a verification facility and intermediate switch are now outlined.

Calculate-Offset Attacks at a Verification Facility. Here the attacker is someone at a verification facility, e.g., an application developer. The steps in the attack are as follows.

1. Generate an EPB E_a that packs a known Final PIN PF_a.
2. For any customer account, A_c, calculate:

   $$\mathit{offset} = \mathsf{CalculateOffset}(E_a, A_c).$$

   If the customer's Natural PIN is PN_c, then $\mathit{offset} = PF_a - PN_c$. Here '$-$' is digit by digit modulo 10 subtraction; *offset* and PF_a are known to the attacker. Thus the attacker learns the customer's Natural PIN. If the attacker can read the plaintext offset value of the customer, then the customer's Final PIN is revealed.

Calculate-Offset Attack at a Switch. The steps of a calculate-offset attack at a switch are as follows.

1. Generate an EPB E_a that packs a known Final PIN PF_a.
2. Select any (random) PAN A_x.

3. Assume that attackers do not have access to the real issuer key at a switch. However, they can calculate a dummy offset using a dummy issuer key (i.e., whatever issuer key is available in the switch's HSM):

$$\mathit{offset}_{d1} = \mathsf{CalculateOffset}(E_a, A_x)$$

i.e., $\mathit{offset}_{d1} = PF_a - PN_{xd}$. Here PN_{xd} is the dummy Natural PIN with respect to the account A_x. So now PN_{xd} can be calculated as both PF_a and offset_{d1} are known.

4. For any customer EPB E_c which packs the customer's Final PIN PF_c, calculate:

$$\mathit{offset}_{d2} = \mathsf{CalculateOffset}(E_c, A_x)$$

i.e., $\mathit{offset}_{d2} = PF_c - PN_{xd}$. The value of PN_{xd} is known from the previous step, thus revealing the customer's Final PIN.

A.3 Attacks Exploiting the VISA PIN Verification Value (PVV)

The steps in the VISA PVV attack at a verification facility and intermediate switch are outlined below.

PVV Attacks at a Verification Facility. Attackers need an EPB with a known PIN, and may need write access to the issuer's PVV database. Again, like offset values, PVVs are considered security insensitive. The attack is as follows.

1. Generate an EPB E_a which packs a known Final PIN PF_a.
2. For any customer PAN A_c,

$$pvv = \mathsf{CalculatePVV}(E_a, A_c).$$

3. Use the calculated PVV with known PIN to create new bank cards (this may also require updating the PVV database at the verification facility).

PVV Attacks at a Switch. Using 10,000 EPBs which pack all possible PINs, attackers can reveal candidate PINs (less than two, on average) for any customer as follows. Note that the attack HSM here does not have access to the real issuer PVV key; the attack succeeds if *any* PVV key is available.

1. Choose any PAN A_x.
2. Generate EPBs for all possible PINs; assume E_i packs PIN i, where $i \in \{0000 \ldots 9999\}$.
3. For all EPBs generated in step 2, calculate PVVs with respect to A_x:

$$pvv_i = \mathsf{CalculatePVV}(E_i, A_x).$$

Now sort the values of pvv_i and build a table of entries (pvv_i, i). More than one (on average less than two) PINs may be indexed by a given PVV.

4. For any customer EPB E_c, compute

$$pvv = \mathsf{CalculatePVV}(E_c, A_x).$$

Use the resulting PVV as an index to the table built in step 3. The corresponding PIN is the customer's Final PIN PF_c; in case of multiple PIN values indexed by pvv, PF_c is one of those values; building the table using a different A_x may resolve collisions.