Technical Report

UCAM-CL-TR-560
ISSN 1476-2986

Number 560

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Decimalisation table attacks for PIN cracking

Mike Bond, Piotr Zieliński

February 2003

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Decimalisation table attacks for PIN cracking

Mike Bond, Piotr Zieliński

Abstract

We present an attack on hardware security modules used by retail banks for the secure storage and verification of customer PINs in ATM (cash machine) infrastructures. By using adaptive decimalisation tables and guesses, the maximum amount of information is learnt about the true PIN upon each guess. It takes an average of 15 guesses to determine a four digit PIN using this technique, instead of the 5000 guesses intended. In a single 30 minute lunch-break, an attacker can thus discover approximately 7000 PINs rather than 24 with the brute force method. With a £300 withdrawal limit per card, the potential bounty is raised from £7200 to £2.1 million and a single motivated attacker could withdraw £30–50 thousand of this each day. This attack thus presents a serious threat to bank security.

1 Introduction

Automatic Teller Machines (ATMs) are used by millions of customers every day to make cash withdrawals from their accounts. However, the wide deployment and sometimes secluded locations of ATMs make them ideal tools for criminals to turn traceable electronic money into clean cash.

The customer PIN is the primary security measure against fraud; forgery of the magnetic stripe on cards is trivial in comparison to PIN acquisition. A street criminal can easily steal a cash card, but unless he observes the customer enter the PIN at an ATM, he can only have three guesses to match against a possible 10,000 PINs and would rarely strike it lucky. Even when successful, his theft still cannot exceed the daily withdrawal limit of around £300 . However, bank programmers have access to the computer systems tasked with the secure storage of PINs, which normally consist of a mainframe connected to a "Hardware Security Module" (HSM) which is tamper-resistant and has a restricted API such that it will only respond to with a YES/NO answer to a customer's guess.

A crude method of attack is for a corrupt bank programmer to write a program that tries all PINs for a particular account, and with average luck this would require about 5000 transactions to discover each PIN. A typical HSM can check maybe 60 trial PINs per second in addition to its normal load, thus a corrupt employee executing the program during a 30 minute lunch break could only make off with about 25 PINs.

However, HSMs implementing several common PIN generation methods have a flaw. The first ATMs were IBM 3624s, introduced widely in the US in around 1980, and most PIN generation methods are based upon their approach. They calculate the customer's original PIN by encrypting the account number printed on the front of the customer's card with a secret DES key called a "PIN generation key". The resulting ciphertext

is converted into hexadecimal, and the first four digits taken. Each digit has a range of '0'-'F'. In order to convert this value into a PIN which can be typed on a decimal keypad, a "decimalisation table" is used, which is a many-to-one mapping between hexadecimal digits and numeric digits. The left decimalisation table in Figure 1 is typical.

0123456789ABCDEF	0123456789ABCDEF
0123456789012345	0000000100000000

Figure 1: Normal and attack decimalisation tables

This table is not considered a sensitive input by many HSMs, so an arbitrary table can be provided along with the account number and a trial PIN. But by manipulating the contents of the table it becomes possible to learn much more about the value of the PIN than simply excluding a single combination. For example, if the right hand table is used, a match with a trial pin of 0000 will confirm that the PIN does not contain the number 7, thus eliminating over 10% of the possible combinations. We first present a simple scheme that can derive most PINs in around 24 guesses, and then an adaptive scheme which maximises the amount of information learned from each guess, and takes an average of 15 guesses. Finally, a third scheme is presented which demonstrates that the attack is still viable even when the attacker cannot control the guess against which the PIN is matched.

Section 2 of the paper sets the attack in the context of a retail banking environment, and explains why it may not be spotted by typical security measures. Section 3 describes PIN generation and verification methods, and section 4 describes the algorithms we have designed in detail. We present our results from genuine trials in section 5, discuss preventative measures in section 6, and draw our conclusions in section 7.

2 Banking Security

Banks have traditionally led the way in fighting fraud from both insiders and outsiders. They have developed protection methods against insider fraud including double-entry book-keeping, functional separation, and compulsory holiday periods for staff, and they recognise the need for regular security audits. These methods successfully reduce fraud to an acceptable level for banks, and in conjunction with an appropriate legal framework for liability, they can also protect customers against the consequences of fraud.

However, the increasing complexity of bank computer systems has not been accompanied by sufficient development in understanding of fraud prevention methods. The introduction of HSMs to protect customer PINs was a step in the right direction, but even in 2002 these devices have not been universally adopted, and those that are used have been shown time and time again not to be impervious to attack [1, 2, 5]. Typical banking practice seeks only to reduce fraud to an acceptable level, but this translates poorly into security requirements; it is impossible to accurately assess the security exposure of a given flaw, which could be an isolated incident or the tip of a huge iceberg. This sort of risk management conflicts directly with modern security design practice where robustness is crucial. There are useful analogues in the design of cryptographic algorithms. Designers who make "just-strong-enough" algorithms and trade robustness for speed or

export approval play a dangerous game. The cracking of the GSM mobile phone cipher A5 is but one example [3].

And as "just-strong-enough" cryptographic algorithms continue to be used, the risk of fraud from brute force PIN guessing is still considered acceptable, as it should take at least 10 minutes to guess a single PIN at the maximum transaction rate of typical modules deployed in the 80s. Customers are expected to notice the phantom withdrawals and report them before the attacker could capture enough PINs to generate a significant liability for the banks. Even with the latest HSMs that support a transaction rate ten times higher, the sums of money an attacker could steal are small from the perspective of a bank.

But now that the PIN decimalisation table has been identified as an security relevant data item, and the attacks described in this paper show how to exploit uncontrolled access to it, brute force guessing is over two orders of magnitude faster. Enough PINs to unlock access to over £2 million can be stolen in one lunch break!

A more sinister threat is the perpetration of a smaller theft, where the necessary transactions are well camouflaged within the banks audit trails. PIN verifications are not necessarily centrally audited at all, and if we assume that they are, the 15 or so transactions required will be hard for an auditor to spot amongst a stream of millions. Intrusion detection systems do not fare much better – suppose a bank has an extremely strict audit system that tracks the number of failed guesses for each account, raising an alarm if there are three failures in a row. The attacker can discover a PIN without raising the alarm by inserting the attack transactions just before genuine transactions from the customer which will reset the count. No matter what the policies of the intrusion detection system it is impossible to keep them secret, thus a competent programmer could evade them. The very reason that HSMs were introduced into banks was that mainframe operating systems only satisfactorily protected data integrity, and could not be trusted to keep data confidential from programmers.

So as the economics of security flaws like these develops into a mature field, it seems that banks need to update their risk management strategies to take account of the volatile nature of the security industry. They also have a responsibility to their customers to reassess liability for fraud in individual cases, as developments in computer security continually reshape the landscape over which legal disputes between bank and customer are fought.

3 PIN Generation & Verification Techniques

There are a number of techniques for PIN generation and verification, each proprietary to a particular consortium of banks who commissioned a PIN processing system from a different manufacturer. The IBM CCA supports a representative sample, shown in Figure 2. We IBM 3624-Offset method in more detail as it is typical of decimalisation table use.

3.1 The IBM 3624-Offset PIN Derivation Method

The IBM 3624-Offset method was developed to support the first generation of ATMs and has thus been widely adopted and mimicked. The method was designed so that offline ATMs would be able to verify customer PINs without needing the processing power and

Method	*Uses Dectables*
IBM 3624	yes
IBM 3624-Offset	yes
Netherlands PIN-1	yes
IBM German Bank Pool Institution	yes
VISA PIN-Validation Value	
Interbank PIN	

Figure 2: Common PIN calculation methods

storage to manipulate an entire database of customer account records. Instead, a scheme was developed where the customer's PIN could be calculated from their account number by encryption with a secret key. The account number was made available on the magnetic stripe of the card, so the ATM only needed to securely store a single cryptographic key. An example PIN calculation is shown in Figure 4.

The account number is represented using ASCII digits, and then interpreted as a hexadecimal input to the DES block cipher. After encryption with the secret "PIN generation" key, the output is converted to hexadecimal, and all but the first four digits are discarded. However, these four digits might contain the hexadecimal digits `'A'-'F'`, which are not available on a standard numeric keypad and would be confusing to customers, so they are mapped back to decimal digits using a "decimalisation table" (Figure 3).

```
0123456789ABCDEF
0123456789012345
```

Figure 3: A typical decimalisation table

```
Account Number        4556 2385 7753 2239
Encrypted Accno       3F7C 2201 00CA 8AB3

Shortened Enc Accno   3F7C

                 0123456789ABCDEF
                 0123456789012345

Decimalised PIN       3572

Public Offset         4344

Final PIN             7816
```

Figure 4: IBM 3624-Offset PIN Generation Method

The example PIN of `3F7C` thus becomes `3572`. Finally, to permit the cardholders to change their PINs, an offset is added which is stored in the mainframe database along with the account number. When an ATM verifies an entered PIN, it simply subtracts the offset from the card before checking the value against the decimalised result of the encryption.

3.2 Hardware Security Module APIs

Bank control centres and ATMs use Hardware Security Modules (HSMs), which are charged with protecting PIN derivation keys from corrupt employees and physical attackers. An HSM is a tamper-resistant coprocessor that runs software providing cryptographic and security related services. Its API is designed to protect the confidentiality and integrity of data while still permitting access according to a configurable usage policy. Typical financial APIs contain transactions to generate and verify PINs, translate guessed PINs between different encryption keys as they travel between banks, and support a whole host of key management functions.

The usage policy is typically set to allow anyone with access to the host computer to perform everyday commands such as PIN verification, but to ensure that sensitive functions such as loading new keys can only be performed with authorisation from multiple employees who are trusted not to collude.

IBM's "Common Cryptographic Architecture" [6] is a financial API implemented by a range of IBM HSMs, including the 4758, and the CMOS Cryptographic Coprocessor (for PCs and mainframes respectively). An example of the code for a CCA PIN verification is shown in Figure 5.

```
Encrypted_PIN_Verify(
    A_RETRES , A_ED ,                 // return codes 0,0=yes 4,19=no
    trial_pin_kek_in , pinver_key ,   // encryption keys for enc inputs
    (UCHAR*)"3624    " "NONE    "     // PIN block format
    "       F"                        // PIN block pad digit
    (UCHAR*)"            " ,
    trial_pin ,                       // encrypted_PIN_block
    I_LONG(2) ,
    (UCHAR*)"IBM-PINO" "PADDIGIT" ,   // PIN verification method
    I_LONG(4) ,                       // # of PIN digits = 4
    "0123456789012345"                // decimalisation table
    "123456789012    "                // PAN_data (account number)
    "0000            "                // offset data
                   );
```

Figure 5: Sample code for PIN verification in CCA

The crucial inputs to `Encrypted_PIN_Verify` are the `decimalisation table`, the `PAN_data`, and the `encrypted_PIN_block`. The first two are supplied in the clear and are straightforward for the attacker to manipulate, but obtaining an `encrypted_PIN_block` that represents a chosen trial PIN is rather harder.

3.3 Obtaining chosen encrypted trial PINs

Some bank systems permit clear entry of trial PINs from the host software. For instance, this functionality may be required to input random PINs when generating PIN blocks for schemes that do not use decimalisation tables. The appropriate CCA command is `Clear_PIN_Encrypt`, which will prepare an encrypted PIN block from the chosen PIN. It should be noted that enabling this command carries other risks as well as permitting our attacks. If there is not randomised padding of PINs before they are encrypted, an attacker could make a table of known trial encrypted PINs, compare each arriving encrypted PIN against this list, and thus easily determine its value. If it is still necessary to enable clear PIN entry in the absence of randomised padding, some systems can enforce that the clear PINs are only encrypted under a key for transit to another bank – in which case the attacker cannot use these guesses as inputs to the local verification command.

So, under the assumption that clear PIN entry is not available to the attacker, his second option is to enter the required PIN guesses at a genuine ATM, and intercept the encrypted PIN block corresponding to each guess as it arrives at the bank. Our adaptive decimalisation table attack only requires five different trial PINs – `0000` , `0001` ,`0010` , `0100` , `1000`. However the attacker might only be able to acquire encrypted PINs under a block format such as ISO-0, where the account number is embedded within the block. This would require him to manually input the five trial PINs at an ATM for each account that could be attacked – a huge undertaking which totally defeats the strategy.

A third and more most robust course of action for the attacker is to make use of the PIN offset capability to convert a single known PIN into the required guesses. This known PIN might be discovered by brute force guessing, or simply opening an account at that bank.

Despite all these options for obtaining encrypted trial PINs it might be argued that the decimalisation table attack is not exploitable unless it can be performed without a single known trial PIN. To address these concerns, we created a third algorithm (described in the next section), which is of equivalent speed to the others, and does not require any known or chosen trial PINs.

4 Decimalisation Table Attacks

In this section, we describe three attacks. First, we present a 2-stage simple *static* scheme which needs only about 24 guesses on average. The shortcoming of this method is that it needs almost twice as many guesses in the worst case. We show how to overcome this difficulty by employing an *adaptive* approach and reduce the number of necessary guesses to 22. Finally, we present an algorithm which uses PIN offsets to deduce a PIN from a single correct encrypted guess, as is typically supplied by the customer from an ATM.

4.1 Initial Scheme

The initial scheme consists of two stages. The first stage determines which digits are present in the PIN. The second stage consists in trying all the possible pins composed of those digits.

Let D_{orig} be the original decimalisation table. For a given digit i, consider a binary decimalisation table D_i with the following property. The table D_i has 1 at position x if and only if D_{orig} has the digit i at that position. In other words,

$$D_i[x] = \begin{cases} 1 & \text{if } D_{\text{orig}}[x] = i, \\ 0 & \text{otherwise.} \end{cases}$$

For example, for a standard table $D_{\text{orig}} =$ `0123 4567 8901 2345`, the value of D_3 is `0001 0000 0000 0100`.

In the first phase, for each digit i, we check the original PIN against the decimalisation table D_i with a trial PIN of `0000`. It is easy to see that the test fails exactly when the original PIN contains the digit i. Thus, using only at most 10 guesses, we have determined all the digits that constitute the original PIN.

In the second stage we try every possible combination of those digits. Their actual number depends on how many different digits the PIN contains. The table below gives the details.

Digits	Possibilities
`A`	`AAAA`(1)
`AB`	`ABBB`(4), `AABB`(6), `AAAB`(4)
`ABC`	`AABC`(12), `ABBC`(12), `ABCC`(12)
`ABCD`	`ABCD`(24)

The table shows that the second stage needs at most 36 guesses (when the original PIN contains 3 different digits), which gives 46 guesses in total. The expected number of guesses is, however, as small as about 23.5.

4.2 Adaptive Scheme

The process of cracking a PIN can be represented by a binary search tree. Each node v contains a guess, i.e., a decimalisation table D_v and a pin p_v. We start at the root node and go down the tree along the path that is determined by the results of our guesses. Let p_{orig} be the original PIN. At each node, we check whether $D_v(p_{\text{orig}}) = p_v$. Then, we move to the right child if yes and to the left child otherwise.

Each node v in the tree can be associated with a list $\mathcal{P}_v$ of original PINs such that $p \in \mathcal{P}_v$ if and only if v is reached in the process described in the previous paragraph if we take p as the original PIN. In particular, the list associated with the root node contains all possible pins and the list of each leaf should contain only one element: an original PIN p_{orig}.

Consider the initial scheme described in the previous section as an example. For simplicity assume that the original PIN consists of two binary digits and the decimalisation table is trivial and maps $0 \to 0$ and $1 \to 1$. Figure 6 depicts the search tree for these settings.

The main drawback of the initial scheme is that the number of required guesses depends strongly on the original PIN p_{orig}. For example, the method needs only 9 guesses for $p_{\text{orig}} =$ `9999` (because after ascertaining that digit 0–8 do not occur in p_{orig} this is the only

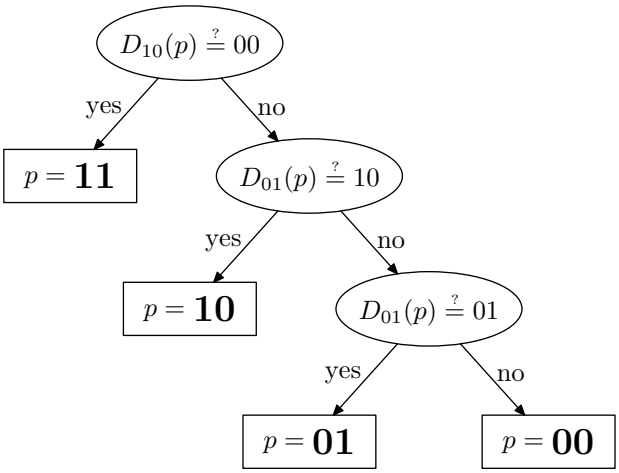

Figure 6: The search tree for the initial scheme. D_{xy} denotes the decimalisation table that maps $0 \to x$ and $1 \to y$.

possibility), but there are cases where 46 guesses are required. As a result, the search tree is quite unbalanced and thus not optimal.

One method of producing a perfect search tree (i.e., the tree that requires the smallest possible numbers of guesses in the worst case) is to consider all possible search trees and choose the best one. This approach is, however, prohibitively inefficient because of its exponential time complexity with respect to the number of possible PINs and decimalisation tables.

It turns out that not much is lost when we replace the exhaustive search with a simple heuristics. We will choose the values of D_v and p_v for each node v in the following manner. Let $\mathcal{P}_v$ be the list associated with node v. Then, we look at all possible pairs of D_v and p_v and pick the one for which the probability of $D_v(p) = p_v$ for $p \in \mathcal{P}_v$ is as close to $\frac{1}{2}$ as possible. This ensures that the left and right subtrees are approximately of the same size so the whole tree should be quite balanced.

This scheme can be further improved using the following observation. Recall that the original PIN p_{orig} is a 4-digit *hexadecimal* number. However, we do not need to determine it exactly; all we need is to learn the value of $p = D_{\text{orig}}(p_{\text{orig}})$. For example, we do not need to be able to distinguish between `012D` and `ABC3` because for both of them $p =$ `0123`. It can be easily shown that we can build the search tree that is based on the value of p instead of p_{orig} provided that the tables D_v do not distinguish between `0` and `A`, `1` and `B` and so on. In general, we require each D_v to satisfy the following property: for any pair of hexadecimal digits x, y: $D_{\text{orig}}[x] = D_{\text{orig}}[y]$ must imply $D_v[x] = D_v[y]$. This property is not difficult to satisfy and in reward we can reduce the number of possible PINs from $16^4 = 65\,536$ to $10^4 = 10\,000$. Figure 7 shows a sample run of the algorithm for the original PIN $p_{\text{orig}} =$ `3491`.

4.3 PIN Offset Adaptive Scheme

When the attacker does not know any encrypted trial PINs, and cannot encrypt his own guesses, he can still succeed by manipulating the offset parameter used to compensate for customer PIN change. Our final scheme has the same two stages as the initial scheme, so

No	Possible pins	Decimalisation table D_v	Trial pin p_v	$D_v(p_{\text{orig}})$	$p_v \stackrel{?}{=} D_v(p_{\text{orig}})$
1	10000	1000 0010 0010 0000	0000	0000	yes
2	4096	0100 0000 0001 0000	0000	1000	no
3	1695	0111 1100 0001 1111	1111	1011	no
4	1326	0000 0001 0000 0000	0000	0000	yes
5	736	0000 0000 1000 0000	0000	0000	yes
6	302	0010 0000 0000 1000	0000	0000	yes
7	194	0001 0000 0000 0100	0000	0001	no
8	84	0000 1100 0000 0011	0000	0010	no
9	48	0000 1000 0000 0010	0000	0010	no
10	24	0100 0000 0001 0000	1000	1000	yes
11	6	0001 0000 0000 0100	0100	0001	no
12	4	0001 0000 0000 0100	0010	0001	no
13	2	0000 1000 0000 0010	0100	0010	no

Figure 7: Sample output from adaptive test program

our first task is to determine the digits present in the PIN.

Assume that an encrypted PIN block containing the correct PIN for the account has been intercepted (the vast majority of arriving encrypted PIN blocks will satisfy this criterion), and for simplicity that the account holder has not changed his PIN and the correct offset is 0000. Using the following set of decimalisation tables, the attacker can determine which digits are present in the correct PIN.

$$D_i[x] = \begin{cases} D_{\text{orig}}[x] + 1 & \text{if } D_{\text{orig}}[x] = i, \\ D_{\text{orig}}[x] & \text{otherwise.} \end{cases}$$

For example, for $D_{\text{orig}} =$ 0123 4567 8901 2345, the value of D_3 is 0124 4567 8901 2445. He supplies the correct encrypted PIN block and the correct offset each time.

As with the initial scheme, the second phase determines the positions of the digits present in the PIN, and is again dependent upon the number of repeated digits in the original PIN. Consider the common case where all the PIN digits are different, for example 1583. We can try to determine the position of the single 8 digit by applying an offset to different digits and checking for a match.

Guess Offset	Guess Decimalisation Table	Customer Guess	Customer Guess + Guess Offset	Decimalised Original PIN	Verify Result
0001	0123 4567 9901 2345	1583	1584	1593	no
0010	0123 4567 9901 2345	1583	1593	1593	yes
0100	0123 4567 9901 2345	1583	1683	1593	no
1000	0123 4567 9901 2345	1583	2583	1593	no

Each different guessed offset maps the customer's correct guess to a new PIN which may or may not match the original PIN after it is decimalised using the modified table. This procedure is repeated until the position of all digits is known. Cases with all digits different will require at most 6 transactions to determine all the position data. Three

different digits will need a maximum of 9 trials, two digits different up to 13 trials, and if all the digits are the same no trials are required as there are no permutations. When the parts of the scheme are assembled, 16.5 guesses are required on average to determine a given PIN.

5 Results

We first tested the adaptive algorithm exhaustively on all possible PINs. The distribution in Figure 8 was obtained. The worst case has been reduced from 45 guesses to 24 guesses, and the average has fallen from 24 to 15 guesses. We then implemented the attacks on the IBM Common Cryptographic Architecture (version 2.41, for the IBM 4758), and successfully extracted PINs generated using the IBM 3624 method. We also checked the attacks against the API specifications for the VISA Security Module (VSM) , and found them to be effective. The VSM is the forerunner of a whole range of hardware security modules for PIN processing, and we believe that the attacks will also be effective against many of its successors.

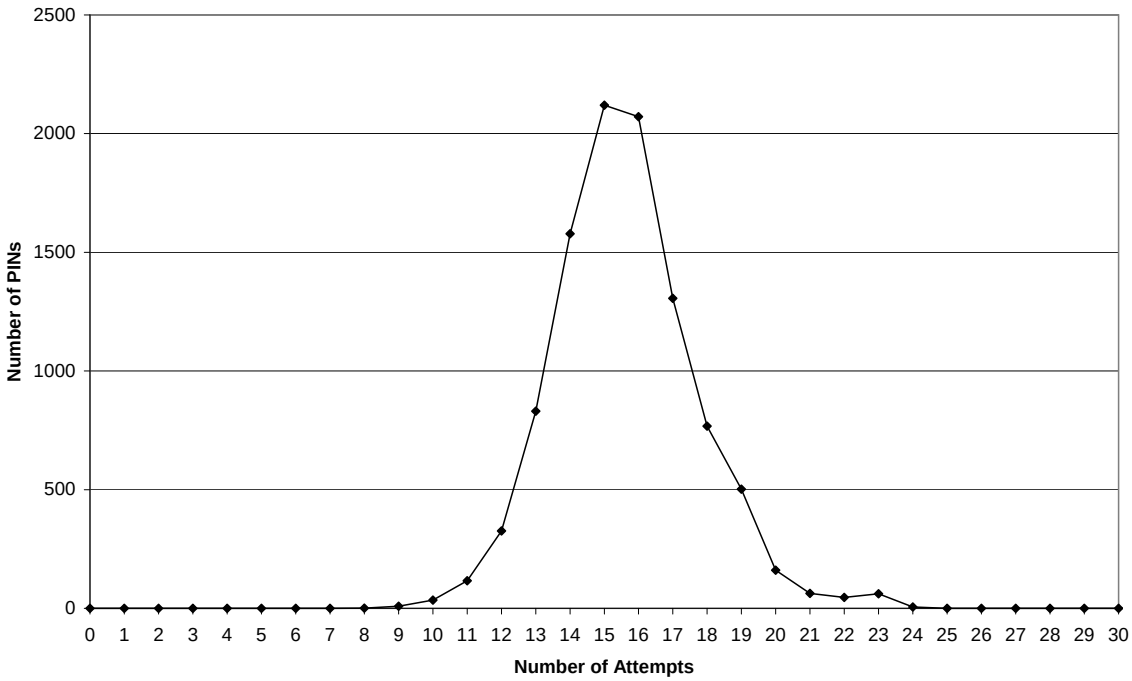

Figure 8: Distribution of guesses required using adaptive algorithm

6 Prevention

It is easy to perform a check upon the validity of the decimalisation table. Several PIN verification methods that use decimalisation tables require that the table should be `0123456789012345` for the algorithm to function correctly, and in these cases the API need only enforce this requirement to regain security. However, PIN verification methods that support proprietary decimalisation tables are harder to fix. A checking procedure that ensures a mapping of the input combinations to the maximum number of possible output combinations will protect against the first two decimalisation table attacks, but not against the attack which exploits the PIN offset and uses only minor modifications to the genuine decimalisation table. To regain full security, the decimalisation table input must be cryptographically protected so that only authorised tables can be used.

The only short-term alternative to the measures above is to use more advanced intrusion detection measures, and it seems that the long term message is clear: continuing to support decimalisation tables is not a robust approach to PIN verification. Unskewed randomly generated PINs stored encrypted in an online database such as are already used in some banks are significantly more secure.

7 Conclusions

We are currently starting discussions with HSM manufacturers with regard to the practical implications of the attacks. It is very costly to modify the software which interacts with HSMs, and while update of the HSM software is cheaper, the system will still need testing, and the update may involve a costly re-initialisation phase. Straightforward validity checking for decimalisation tables should be easy to implement, but full protection that retains compatibility with existing mainframe software will be hard to achieve. It will depend upon the intrusion detection capabilities offered by each particular manufacturer. We hope to have a full understanding of the impact of these attacks and of the optimal preventative measures in the near future.

Although HSMs have existed for two decades, formal study of their security APIs is still in its infancy. Previous work by one of the authors [5, 4] has uncovered a whole host of diverse flaws in APIs, some at the protocol level, some exploiting properties of the underlying crypto algorithms, and some exploiting poor design of procedural controls. The techniques behind the decimalisation table attacks do not just add another string to the bow of the attacker – they further confirm that designing security APIs is one of the toughest challenges facing the security community. It is hard to see how any one methodology for gaining assurance of correctness can provide worthwhile guarantees, given the diversity of attacks at the API level. More research is needed into methods for API analysis, but for the time being we may have to concede that writing correct API specifications is as hard as writing correct code, and enter the traditional arms race between attack and defence that so many software products have to fight.

Acknowledgements

We would like to thank Richard Clayton and Ross Anderson for their helpful contributions and advice. Mike Bond was able to conduct the research thanks to the funding received from the UK Engineering and Physical Research Council (EPSRC) and Marconi plc. Piotr Zieliński was supported by a Cambridge Overseas Trust Scholarship combined with an ORS Award, as well as by a Thaddeus Mann Studentship from Trinity Hall College.

References

[1] R. Anderson: Why Cryptosystems Fail *Communications of the ACM*, 37(11), pp32–40 (Nov 1994)

[2] R. Anderson: The Correctness of Crypto Transaction Sets *Proc. Cambridge Security Protocols Workshop 2000* LNCS 2133, Springer-Verlag, pp 125–127 (2000)

[3] A. Biryukov, A. Shamir, D. Wagner Real Time Cryptanalysis of A5/1 on a PC *Proceedings of Fast Software Encryption 2000*

[4] M. Bond, R. Anderson API-Level Attacks on Embedded Systems *IEEE Computer Magazine*, October 2001, pp 67–75

[5] M. Bond: Attacks on Cryptoprocessor Transaction Sets *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES 2001)*, LNCS 2162, Springer-Verlag, pp 220–234 (2001)

[6] IBM Inc.: IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide for the IBM 4758-001, Release 1.31. IBM, Armonk, N.Y. (1999) `http://www.ibm.com/security/cryptocards/bscsvc02.pdf`