Cracking bank PINs by playing Mastermind [⋆]

Riccardo Focardi and Flaminia L. Luccio

Università Ca' Foscari Venezia,
{focardi,luccio}@dsi.unive.it

Abstract. The bank director was pretty upset noticing Joe, the system administrator, spending his spare time playing Mastermind, an old useless game of the 70ies. He had fought the instinct of telling him how to better spend his life, just limiting to look at him in disgust long enough to be certain to be noticed. No wonder when the next day the director fell on his chair astonished while reading, on the newspaper, about a huge digital fraud on the ATMs of his bank, with millions of Euros stolen by a team of hackers all around the world. The article mentioned how the hackers had 'played with the bank computers just like playing Mastermind', being able to disclose thousands of user PINs during the one-hour lunch break. That precise moment, a second before falling senseless, he understood the subtle smile on Joe's face the day before, while training at his preferred game, Mastermind.

Keywords: Security APIs, PIN processing, Hardware Security Modules, Mastermind.

1 Introduction

The Mastermind game was invented in 1970 by Mordecai Meirowitz. The game is played as a board game between two players or as a one player game between a single player and the computer (in both cases called the *codebreaker* and the *codemaker*, respectively) [19]. The codemaker chooses a linear sequence of colored pegs and conceals them behind a screen. Duplicates are allowed. The codebreaker has to guess, in different trials, both the color and the position of the pegs. During each trial he learns something and based on this he decides the next guess: in particular, a response consisting of a *black peg* (which we will call *black marker*) represents a right guess of the color and the position of a peg (but the marker does not indicate which one is correct), a response consisting of a *white peg* (called *white marker*) represents only the right guess of a color but at the wrong position.

An apparently completely unrelated problem is the one of protecting user's Personal Identification Number (PIN) when withdrawing some money at an Automated Teller Machine (ATM). International bank networks are structured

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

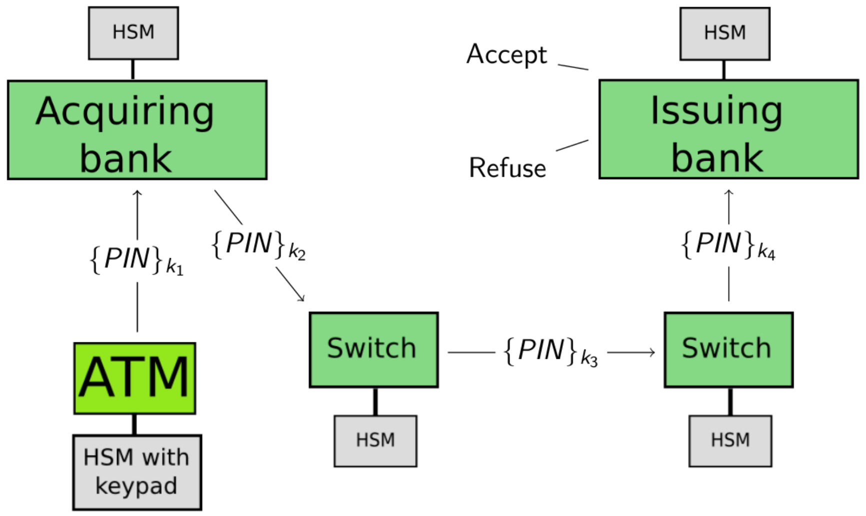

Fig. 1. Bank network.

in such a way that an access to an ATM implies that the user's PIN is sent to the issuing bank for the verification. While travelling, the PIN is decrypted and re-encrypted by special tamper-resistant devices called Hardware Security Modules (HSMs) which are placed on the traversed network switches, as illustrated in figure 1. The first PIN encryption is performed by the ATM keypad which is an HSM itself, using a symmetric key k_1 shared with the neighbour acquiring bank. While travelling from node to node, the encrypted PIN is decrypted and re-encrypted with another key shared with the destination node, by the HSM located in the switch. The final verification and acceptance/refusal of the PIN is done by the issuing bank.

Although this setting seems to be secure, several API-level attacks have been discovered on these HSMs in the last years [5, 6, 10]. These attacks work by assuming that the attacker is an insider gaining access to the HSM at some bank switch and performing subtle sequences of API calls from which he is able to deduce the value of the PIN. There are many examples of such attacks, the one we are considering in this paper is the so-called *dectab attack* [6], which we will illustrate in the detail in the next section. Intuitively, while verifying the PIN, the PIN verification API at the issuing bank HSM takes as an input different parameters, some of which are public. One of these parameters is a decimalization table that maps an intermediate hexadecimal representation of the user PIN into a decimal number. By manipulating some information, e.g., by modifying the way numbers are decimalized and by observing if this affects the result of the verification, the attacker can deduce which are the actual PIN digits. The position of the guessed PIN digits is reconstructed by manipulation another public parameter, i.e., the *offset* of the PIN. By combining all this information the attacker is able to reconstruct the whole PIN.

Our contribution. In this paper we show that decimalization attacks can be seen as playing an extended Mastermind game. Each API call represents a trial of the codebreaker and the API return value is the corresponding answer of the codemaker. Manipulating the dectab and the offset together is similar to asking the codemaker to disclose the color and the position of one PIN digit, in case the guess is correct, similarly to what happens with a black marker of Mastermind. Modifying the dectab only, instead, corresponds to asking for the presence of certain digits in the PIN, analogously to the white marker in the game.

We make the above intuition formal by showing how PIN cracking and Mastermind can be seen as instances of a more general problem, or game. This extended problem suggests a new way of improving the dectab attack. The idea is to allow the player (i.e, the attacker) to ask for sets of colors (i.e., digits), instead of just single colors, for each position. This, in fact, can be implemented in the PIN cracking setting by modifying multiple entries of the dectab, as we will show in detail. We show that this reduces the known bounds on the number of average API calls for performing the attacks from 16.145 to 14.484 which is close to the optimal value of 13.362.

To this aim, we develop a computer program that optimizes a well known technique presented by Knuth in [15] for the standard Mastermind game and extend it to our setting. We perform experiments showing that the program is almost as precise as state-of-the-art Mastermind solvers [13] but faster, being it able to compute strategies for cases not yet covered. More interestingly, the very same solving strategy is adapted to the PIN cracking problem proving the above mentioned new bound on the average number of API calls required in dectab attacks.

Paper structure. In section 1.1 we briefly summarize the related literature. In section 2 we formally define the two problems, i.e., the Generalized Mastermind Problem and the PIN Cracking Problem. In section 3 we introduce the Extended Mastermind Problem, i.e., a general problem whose instances are the Generalized Mastermind Problem and the PIN Cracking Problem. In section 4 we expose some experimental results, and we conclude in section 5.

1.1 Related literature

Mastermind. In [15] Donald Knuth presented an algorithm for the solution of the standard Mastermind game, which is played using pegs of 6 different colors, in a sequence of length 4. He showed how the codebreaker can find the pattern in five moves or fewer, using an algorithm that progressively reduces the number of possible patterns. Each guess is made so that it minimizes the maximum number of remaining possibilities. The expected number of guesses is 4.478. In 1993 Kenji Koyama and Tony W. Lai proposed a technique that uses at most 6 guesses but decreases the expected number to 4.340 or to 4.341 if only 5 guesses are allowed [17]. In [4, 14], the authors apply evolutionary and genetic algorithms to solve the Mastermind problem.

Different variants of the game have been proposed, e.g, in [9], Chvatal mentions a problem, suggested by Pierre Duchet, called the *static Mastermind*. This problem consists of finding the minimum number of guesses made all at once (i.e., without waiting for the responses), that are required to determine the code.

In [8] the authors propose a bound for finding a hidden code by asking questions. This problem relates to the Generalized Mastermind Game with N colors and sequences of length k. The authors show that $\lceil \frac{k}{N} \rceil + 2NlogN + 2N + 2$ guesses are sufficient to solve the problem.

Finally, in [13] the authors present some new bounds to the Generalized Mastermind Game. Using a computer program they compute some new exact values of maximum number of guesses. They also provide theoretical bounds for the case of sequences of length 2, 3 and 4, and for the general case of N colors and length k.

PIN cracking. API-level attacks on PINs have recently attracted attention from the media [1, 3]. This has increased the interest in studying formal methods for analysing PIN recovery attacks and API-level attacks in general [18]. In particular, different models have been proposed, e.g., in [6] the authors prove that in average 16.5 API calls are required to reconstruct the PIN and this bound was decreased to 16.145 in [18]. In [7] we have presented, together with other authors, a language-based setting for analysing PIN processing API via a type-system. We have formally modelled existing attacks, proposed some fixes and proved them correct via type-checking. These fixes typically require to reduce and modify the HSM functionality by, e.g., sticking on a single format of the transmitted PIN or adding MACs for the integrity of user data. Notice, in fact, that the above mentioned attack is based on the absence of integrity on public user data such as the *dectab* and the *offset*. As upgrading the bank network HSMs worldwide is complex and very expensive in [11] we have also have proposed a low-impact, easily implementable fix requiring no hardware upgrade which makes attacks 50000 times slower, but yet not impossible.

2 The two problems

In this section we give a formal definition of the two problems we will be relating. We first define the *Generalized Mastermind Problem (GMP)*, i.e., the problem of solving a Generalized Mastermind Game, and we then present the problem of attacking a PIN using the decimalization table, and we call it the *PIN Cracking Problem (PCP).*

2.1 The Generalized Mastermind Game

The Generalized Mastermind Problem is a game that is played between a player (the "codebreaker") and a computer or another player (the "codemaker"). The

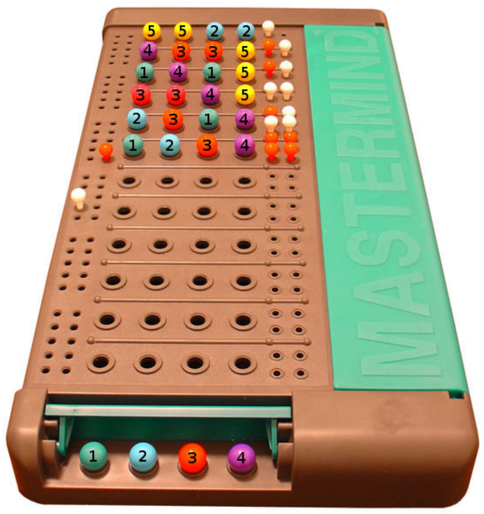

Fig. 2. An example of a Mastermind game.

codemaker chooses a linear sequence of k colored pegs, which we call *secret* and conceals them behind a screen. The colors range in a set $\{0, 1, \ldots, N-1\}$. The codebreaker has to guess the secret, i.e., both the color of the pegs and their exact position. The game is played in steps, each of which consists of a guess of the codebreaker and a response of the codemaker. The response can be empty, can contain a black or a white marker, i.e., is a sequence of at most 4 markers chosen in the set $\{B, W\}$. The black marker represents a correct guess both of the color and the position of a peg, there is no indication however of its position, the white marker only represents the correct guess of the color.

An example of the standard Mastermind game, i.e., played with $N = 6$ colors and $k = 4$ pegs, is shown in figure 2 taken from [2]. In this example black markers are depicted in red. We have added numbers to identify different colors. At the first step the codebreaker only finds a right color, i.e., a cyan peg (2), in a wrong position, thus the response is a white marker, i.e., W. At the next step he correctly guesses a red peg (3) in the right position and a purple peg (4) in a wrong position, thus the response is a black and a white peg, i.e., B, W, an so on. At the last step the response are 4 black markers, i.e., B, B, B, B.

Note that in the standard Mastermind game the set of all possible solutions has size 6^4, in the Generalized Mastermind Game the size explodes to N^k, thus running plain exhaustive search techniques might become problematic when N and k increase too much.

We can now formulate our problem.

The Generalized Mastermind Problem (GMP). Given a Generalized Mastermind Game played on N colors and k pegs, devise a minimal sequence of guesses for the correct disclosure of the secret.

```
PIN_V( $PAN, EPB, len, offset, vdata, dectab$ ) {
    $x_1 := \mathsf{enc}_{pdk}(vdata)$;
    $x_2 := \mathsf{left}(len, x_1)$;
    $x_3 := \mathsf{decimalize}(dectab, x_2)$;
    $x_4 := \mathsf{sum\_mod10}(x_3, offset)$;
    $x_5 := \mathsf{dec}_k(EPB)$;
    $x_6 := \mathsf{fcheck}(x_5)$;
    if ($x_6 = \perp$) then return("format wrong");
    if ($x_4 = x_6$) then return("PIN correct");
                    else return("PIN wrong")}
```

Table 1. The *verification* API.

2.2 API-level attacks in bank networks

In this section we show in detail a real API-level attack to the bank PINs. As we have mentioned in the introduction, a PIN travelling along the network has to be decrypted and re-encrypted under a different key, and this is done using a so called *translation* API. While the PIN reaches the issuing bank, its correspondence with the *validation data*, i.e., a value that is typically an encoding of the user Personal Account Number (PAN) and possibly other 'public' data, such as the card expiration date or the customer name, is checked via a *verification* API. We focus on this latter API, called PIN_V and reported in table 1, that checks the equality of the actual *user* PIN, derived through the PIN derivation key *pdk*, from the public data *offset*, *vdata*, *dectab*, and the *trial* PIN inserted at the ATM that arrives encrypted under key k as *EPB* (Encrypted PIN block). The API returns the result of the verification or an error code.

PIN_V behaves as follows:

- The user PIN of length *len* is computed by first encrypting validation data *vdata* with the PIN derivation key *pdk* (x_1) and obtaining a 16 hexadecimal digit string. Then, the first *len* hexadecimal digits are chosen (x_2), and decimalised through *dectab* (x_3), obtaining the 'natural' PIN assigned by the issuing bank to the user. decimalize is a function that associates to each possible hexadecimal digit (of its second input) a decimal one as specified by its first parameter (*dectab*). Finally, if the user wants to choose her own PIN, an *offset* is calculated by digit-wise subtracting (modulo 10) the natural PIN from the user-selected one (x_4).
- To recover the trial PIN *EPB* is first decrypted with key k (x_5), then the PIN is extracted by the formatted decrypted message (x_6). This last operation depends on the specific PIN format adopted by the bank. In some cases, for example, the PIN is padded with random digits so to make its encryption

immune from codebook attacks. In this case, extracting the PIN involves removing this random padding.
– Finally, if x_6 fails ($\perp$ represents failure) then a message is returned, moreover the equality between the user PIN and the trial PIN is verified.

An API attack on PIN_V. We now illustrate a real attack on PIN_V first reported in [6]. The attack works by iterating the following two steps, until the whole PIN is recovered:

1. To discover whether or not a decimal digit d is present in the user 'natural' PIN contained in x_3 the intruder picks digit d, changes the *dectab* function so that values previously mapped to d now map to d+1 mod 10, and then checks whether the system still returns '*PIN correct*'. If this is the case d is not contained in the 'natural' PIN.

2. To locate the position of the digit previously discovered by a '*PIN wrong*' output the intruder also changes the *offset* until the API returns again that the PIN is correct.

We illustrate the attack through a simple example.

Example 1. Assume *len*=4, *dectab*=5753108642143210, as

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

offset=4732. The correct solution, unknown to the intruder, is the following.

$$\left|\begin{array}{l} x_2 = \mathsf{left}(4, AD7295FDE32BA101) \\ x_3 = \mathsf{decimalize}(dectab, AD72) \\ x_4 = \mathsf{sum\_mod10}(1265, 4732) \\ x_5 = \mathsf{dec}_k(\{\!|5997, r|\!\}_k) \\ x_6 = \mathsf{fcheck}(5997, \mathsf{r}) \end{array}\right| \begin{array}{l} = \text{AD72} \\ = 1265 \\ = 5997 \\ = (5997, r) \\ = 5997 \end{array}\right|$$

Since x_6 is different from $\perp$ and $x_4 = x_6$, the API returns '*PIN correct*'.

The attacker, unaware of the value of the PIN, first changes the dectab, which is a public parameter, as $dectab'$=57531$\underline{1}$864214321$\underline{1}$, i.e., it replaces the two 0's by 1's. The aim is to discover whether or not 0 appears in x_3. Invoking the API with $dectab'$ we obtain $\mathsf{decimalize}(dectab', AD72) = \mathsf{decimalize}(dectab, AD72) =$ 1265, that is 0 does not appear in x_3. The attacker proceeds by replacing the 1's of *dectab* by 2's: with $dectab''$=5753$\underline{2}$08642$\underline{2}$432$\underline{2}$0 he has $\mathsf{decimalize}(dectab'', AD72)$ $= 2265 \neq \mathsf{decimalize}(dectab, AD72)$=1265, reflecting the presence of 1's in the original value of x_3. Then, x_4=$\mathsf{sum\_mod10}(2265, 4732)$ =6997 instead of 5997 returning '*PIN wrong*'.

The intruder now knows that digit 1 occurs in x_3, and to discover its position and multiplicity, he now varies the offset so to 'compensate' for the modification of the *dectab*. In particular, he tries to decrement each offset digit by 1. For example, testing the position of one occurrence of one digit amounts to trying the following offset variations: $\underline{3}732, 4\underline{6}32, 47\underline{2}2, 473\underline{1}$. Notice that, in this specific case, offset value 3732 makes the API return again '*PIN correct*'.

The attacker now knows that the first digit of x_3 is 1. Given that the *offset* is public, he also calculates the first digit of the user PIN as $1 + 4 \bmod 10 = 5$.

We can now formulate our problem.

The PIN Cracking Problem (PCP). Given a bank network the *PCP* consists of recovering an encrypted (i.e., secret) PIN by devising a malicious sequence of calls to the *verification* API.

3 Extended Mastermind

We exend the Generalized Mastermind Problem presented in previous section, by allowing the codebreaker to pose an extended guess composed of k *sets* of colored pegs, instead of just k pegs. Intuitively, the sets represent alternative guesses, i.e., it is enough that one of the peg in the set is correct to get a black or a white marker.

More formally, let $\mathcal{C} = \{0, 1, \ldots, N-1\}$ be the set of colors. We note $(S_1, S_2, \ldots, S_k)$, with $S_1, \ldots, S_k \subseteq \mathcal{C}$, an extended guess, and $(c_1, c_2, \ldots, c_k)$, with $c_1, \ldots, c_k \in \mathcal{C}$, the secret.

Intuitively, the number of black markers represents the number of colors in the secret belonging to the corresponding set. Formally:

Definition 1 (Black markers). *The number b of black markers is computed as $b = |\{i \in [1,k] \mid c_i \in S_i\}|$.*

The number of white markers, instead, corresponds to the number of colors in the secret belonging to sets in the guess, but not the ones in the corresponding position. To formalize this we first compute the number of occurrences of a color J $\in \mathcal{C}$ in the secret code as $p_j = |\{i \in [1,k] \mid j = c_i\}|$, and in the guess as $q_j = |\{i \in [1,k] \mid j \in S_i\}|$. Now $min(p_j, q_j)$ represents the number of matching pegs of color j. If we sum over all the colors we obtain the overall number of matching pegs. From this we need to subtract the ones giving black markers, in order to obtain the number of white markers.

Definition 2 (White markers). *The number w of white markers is computed as $w = \sum_{j=1}^{N} min(p_j, q_j) - b$.*

Let show the above definitions with a simple example

Example 2. Let $N = 6$, $(1, 2, 3, 1)$ be the secret and $(1, 3, 1, 3)$ be the guess[1] We compute $b = |\{i \in [1,k] \mid c_i \in S_i\}| = |\{1\}| = 1$. In fact only the first '1' is in the right position, giving a black marker. Then we have

$$\begin{array}{ll|ll}
p_0 = |\{\}| & = 0 & q_0 = |\{\}| & = 0 \\
p_1 = |\{1,4\}| & = 2 & q_1 = |\{1,3\}| & = 2 \\
p_2 = |\{2\}| & = 1 & q_2 = |\{\}| & = 0 \\
p_3 = |\{3\}| & = 1 & q_3 = |\{2,4\}| & = 2 \\
p_4 = |\{\}| & = 0 & q_4 = |\{\}| & = 0 \\
p_5 = |\{\}| & = 0 & q_5 = |\{\}| & = 0
\end{array}$$

[1] we omit the set notation for singletons, i.e., we write $(1, 3, 1, 3)$ in place of $(\{1\}, \{3\}, \{1\}, \{3\})$.

Now $\sum_{j=1}^{N} min(p_j, q_j) = 3$ meaning there are 3 matching pegs (the two 1's and one of the 3), but one of them is already counted as a black. Thus we obtain $w = 3 - b = 2$. Notice that the two 3's in the guess are counted just once, as only one 3 appears in the secret code. This is why we need to take $min(p_j, q_j)$.

To see how this scales to set consider the extended guess $(1, 3, 1, \{1, 3\})$. In this case we have $b = 2$ (the first and the last pegs) and $w = 3 - b = 1$, i.e., there is one peg in the wrong position (i.e., the '3').

Definition 3 (The Extended Mastermind Problem - EMP). *Given an Extended Mastermind Game played on N colors and k pegs, devise a minimal sequence of guesses for the correct disclosure of the secret.*

We now show how the two previous problem can be seen as instances of EMP.

Lemma 1. *GMP is an instance of EMP.*

Proof. It is sufficient to restrict sets in guesses to singletons to recover the Generalized Mastermind Game.

More interestingly, we see how the PIN cracking problem can be seen as particular instance of GMP.

Lemma 2. *PCP is an instance of EMP.*

Proof. We restrict the extended guesses $(S_1, \ldots, S_k)$ so that, once the offset digits have been subtracted from each member of the corresponding set, the resulting sets are either equal or disjoint. Let $\hat{S}_1, \ldots, \hat{S}_t$, $t \leq k$, be such sets. We modify the dectab of each digit $d \in \hat{S}_i$ by mapping it into $d + i$. This mapping is well-defined given that sets $\hat{S}_i$ are disjoint. At the same time we modify the digits of the offset at the positions where $\hat{S}_i$ occurs (recall it may occur more than once in case of equal sets) by decreasing them by i. As a result, since we have changed the offset, the only way to obtain a 'PIN correct' is that the digits of the intermediate PIN calculation at those positions have been increased by i and this only holds if they appears in $\hat{S}_i$. Iterating this on all the sets we easily see that 'PIN correct' corresponds to having $c_i \in S_i$ for all $i = 1, \ldots, k$, i.e., having four black markers. Thus, we say that the codemaker answers 'yes' when the answer is 4 black markers and 'no' otherwise. Notice that the player can use extended guesses and ask for sets of values and not just singletons, so it is not necessary to guess the exact code to get a 'yes'.

4 Experimental results

We have devised a program which is an optimized extension of the original program for Mastermind presented by Knuth in [15]. It works as follows:

1. Tries all the possible guesses. For each guess, computes the number of 'surviving' solutions related to each possible outcome of the guess;

Colours/Pegs	2	3	4	5	6	7	8	9	10
2	3	4	4	5	6	6	7	**7**	**8**
3	4	4	4	4	5	6	**6**		
4	4	4	4	5	6				
5	5	5	5						
6	5	5	5						
7	6	6	6						
8	6	6	6						
9	7	7	7						
10	7	7	8						

Table 2. Our optimization of Knuth's algorithm.

2. Picks the guess from the previous step which minimizes the maximum number of surviving solutions among all the possible outcomes and performs the guess:
   (a) For each possible outcome, stores the corresponding surviving solutions and recursively calls this algorithm;
   (b) stops whenever the number of surviving solutions is 0 (impossible outcome) or 1 (guessed the right sequence).

In order to reduce the complexity of the exhaustive search over all possible guesses we have implemented an optimization which starts working on a subset of the colors (the one used up to the current guess) and adds new colors only when needed by the guesses. This is similar, in the spirit, to what is done in [13].

We first show some results obtained by running this optimized algorithm to the Generalized Mastermind Problem. Note that most computations took few seconds, others few minutes, and we were also able to find new upper bounds on the minimal number of moves for unknown values (see Table 2, values in bold). As a matter of fact, as it is mentioned in [12], Knuth's idea does not define an optimal strategy, it is however very close to the optimal. In [13] some empirical optimal values were computed (see Table 3) and some theoretical bounds were presented. Note that our values differ at most by one from the exact ones. Moreover, we were able to efficiently find bounds on 2 colors and 9 and 10 pegs and 3 colors and 8 pegs, and to list the exact sequence of moves to be followed, whereas the program of [13], as the authors state, would probably take "many weeks" of computation.

We have then applied the very same algorithm to the PIN cracking problem. In this case we have noticed that using sets with more than two elements in the guesses did not improve the solutions. With sets of size at most two the algorithm performs quite well and we have been able to improve the results of [18] by finding a strategy with an average number of calls of 14.484 instead of 16.145. This improvement is based on the idea of extended guesses, in which sets of values can be queried by simultaneously changing their mapping in the dectab of a same quantity. This idea is new, and extends the attack strategy

Colours/Pegs	2	3	4	5	6	7	8
2	3	3	4	4	5	5	6
3	3	4	4	4	5	5	
4	4	4	4	5	5		
5	4	5	5	5			
6	5	5	5				
7	5	6	6				
8	6	6	6				
9	6	6	7				
10	7	7	7				

Table 3. Bounds from [13].

illustrated in [6] and studied in [18]. Notice that in [6, 18] special 'dectab-only' API calls, where the offset is left untouched, are exploited in order to immediately discover whenever a digit appears as one of the intermediate PIN digits. In our approach, these calls are generalized to sets by performing guesses (once the offset is subtracted) of the form $(\hat{S}, \ldots, \hat{S})$, with $\hat{S}$ containing the digits whose presence has to be checked. We also found a new bound for PINs of length 5 giving an average of 20.88 calls. It is worth noticing that our results are close to the optimal partitioning of the solutions into an almost-balanced binary tree. In fact, it can be easily computed that this would give a number of average calls of 13.362 and 16.689 for PINs of length 4 and 5, respectively.

All the files containing the detailed strategies for Mastermind and PIN cracking can be downloaded at `http://www.dsi.unive.it/~focardi/MM_PIN/`

5 Conclusion

In this paper we have considered two rather different problems, Mastermind and PIN cracking, and we have shown how they can be seen as instances of an extended Mastermind game in which guesses can contain sets of pegs. We have implemented an optimized version of a classic solver for Mastermind and we have applied it to PIN cracking, improving the known bound on the number of API calls. The idea of using sets in the guesses has in fact suggested a new attacking strategy that reduces the number of required calls. By combining 'standard' attacks with this new strategy we have been able to reduce the average number of calls from 16.145 to 14.484. We also found a new bound for PINs of length 5 giving an average of 20.88 calls. Both average cases are close to the optimum.

As a future work we intend to study the extension of more involved techniques such as the ones of [13] to the PIN cracking setting. As a final note, we would solicit the bank director of our abstract, and other serious people to be more open-minded and never assume that something is useless just because it is funny, "sooner or later society will realize that certain kinds of hard work are in fact admirable even though they are more fun than just about anything else" [16].

Acknowledgements. We would like to thank Graham Steel for his helpful comments and suggestions.

References

1. Hackers crack cash machine PIN codes to steal millions. The Times online. `http://www.timesonline.co.uk/tol/money/consumer_affairs/article4259009.ece`.
2. Mastermind. `http://commons.wikimedia.org/wiki/File:Mastermind.jpg`.
3. PIN Crackers Nab Holy Grail of Bank Card Security. Wired Magazine Blog 'Threat Level'. `http://blog.wired.com/27bstroke6/2009/04/pins.html`.
4. L. Bento, L. Pereira, and A. Rosa. Mastermind by evolutionary algorithms. In New York ACM Press, editor, *Proc. ACM Symp. Applied Computing, San Antonio, Texas*, page 307311, 28 February-2 March 1999.
5. O. Berkman and O. M. Ostrovsky. The unbearable lightness of PIN cracking. In Springer LNCS vol.4886/2008, editor, *11th International Conference, Financial Cryptography and Data Security (FC 2007), Scarborough, Trinidad and Tobago*, pages 224–238, February 12-16 2007.
6. M. Bond and P. Zielinski. Decimalization table attacks for pin cracking. Technical Report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, 2003. `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-560.pdf`.
7. M. Centenaro, R. Focardi, F. Luccio, and G. Steel. Type-based analysis of PIN processing APIs. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, pages 53–68. Springer, LNCS 5789, 2009.
8. Z. Chen, C. Cunha, and S. Homer. Finding a hidden code by asking questions. In Springer LNCS vol. 1090/1996, editor, *Computing and Combinatorics Second Annual International Conference (COCOON '96) Hong Kong*, pages 50–55, June 1719 1996.
9. V. Chvatal. Mastermind. *Combinatorica*, 3:325–329, 1983.
10. J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.
11. R. Focardi, F. Luccio, and G. Steel. Blunting differential attacks on PIN processing APIs. In *Proceedings of the 14th Nordic Conference on Secure IT Systems (NordSec 2009)*. Springer, LNCS 5838, October 2009.
12. W. Goddard. Mastermind revisited. *J. Combin. Math. Combin. Comput.*, 51:215–220, 2004.
13. G. Jäger and M. Pezarski. The number of pessimistic guesses in generalized mastermind. *Information Processing Letters*, 109:635–641, 2009.
14. T. Kalisker and D. Camens. Solving mastermind using genetic algorithms. In Springer, editor, *Lect. Notes Comput. Sci., 2724*, page 15901591, 2003.
15. D. Knuth. The Computer as a Master Mind. *Journal of Recreational Mathematics*, 9:1–6, 1976.
16. Donald E. Knuth. *The Stanford GraphBase: a platform for combinatorial computing.* Addison-Wesley Professional, 1993.
17. M. Koyama and T. Lai. An Optimal Mastermind Strategy. *Journal of Recreational Mathematics*, 25:251–256, 1993.
18. G. Steel. Formal Analysis of PIN Block Attacks. *Theoretical Computer Science*, 367(1-2):257–270, November 2006.
19. J. Stuckman and G. Zhang. Mastermind is NP-Complete. *INFOCOMP Journal of Computer Science*, 5:25–28, 2006.