Theory Comput Syst (2012) 50:52–71
DOI 10.1007/s00224-011-9340-9

# Guessing Bank PINs by Winning a Mastermind Game

**Riccardo Focardi · Flaminia L. Luccio**

Published online: 17 June 2011


**Abstract** In this paper we formally prove that the problem of cracking, i.e., correctly guessing, bank PINs used for accessing Automated Teller Machines and the problem of solving the Generalized Mastermind Game are strictly related. The Generalized Mastermind Game with $N$ colors and $k$ pegs is an extension of the well known Mastermind game, played with 6 colors and 4 pegs. The rules are the same, one player has to conceal a sequence of $k$ colored pegs behind a screen and another player has to guess the exact position and colors of the pegs using the minimal number of moves. We first introduce a general game, called the *Extended Mastermind Game* (*EMG*), and we then formally prove it includes both the Generalized Mastermind Game and the PIN cracking Problem. We then present some experimental results that we have devised using a computer program that optimizes a well known technique presented by Knuth in 1976 for the standard Mastermind game. We finally show that the program improves the as state-of-the-art Mastermind solvers as it is able to compute strategies for cases which were not yet covered. More interestingly, the same solving strategy is adapted also for the solution of the PIN cracking problem.

**Keywords** Security APIs · PIN processing · Hardware security modules · Mastermind

A preliminary version of this paper has been presented at the *5th International Conference on Fun with Algorithms (FUN'10)* [14].

R. Focardi · F.L. Luccio (✉)
c/o Dipartimento di Scienze Ambientali, Informatica e Statistica (DAIS),
Università Ca' Foscari Venezia, via Torino 155, 30172, Venezia, Italy
e-mail: luccio@unive.it

R. Focardi
e-mail: focardi@dsi.unive.it

## 1 Introduction

The Mastermind game was invented in 1970 by the Israeli postmaster and telecommunication expert Mordecai Meirowitz. It was then brought on the market in 1972 by the English company Invicta Plastics Ltd. The game is played as a board game between two players, a *codebreaker* and a *codemaker* (which can be a human or a computer) [22]. The codemaker chooses a linear sequence of colored pegs and conceals them behind a screen. Duplicates of colored pegs 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 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 marker* represents only the right guess of a color but at the wrong position.

An apparently completely unrelated problem is the one of protecting a user's Personal Identification Number (PIN) when withdrawing some money at an Automated Teller Machine (ATM). International bank networks are structured in such a way that an access to an ATM implies that the user's PIN is sent from the ATM to the issuing bank for the verification. While traveling, the PIN is decrypted and re-encrypted by special tamper-resistant devices called Hardware Security Modules (HSMs) which are placed on each traversed network switch, as illustrated in Fig. 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 neighbor acquiring bank. While traveling from node to node, the encrypted PIN is decrypted and re-encrypted by the HSM located in the switch with another key shared with the destination node. 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 [6, 8, 12]. 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, however, in this paper we concentrate only on the so-called *decimalization table (dectab) attack* [8]. We will formally illustrate this attack in the next section. Intuitively, the PIN verification API at the issuing bank verifies the correctness of a PIN by checking the equality between the trial

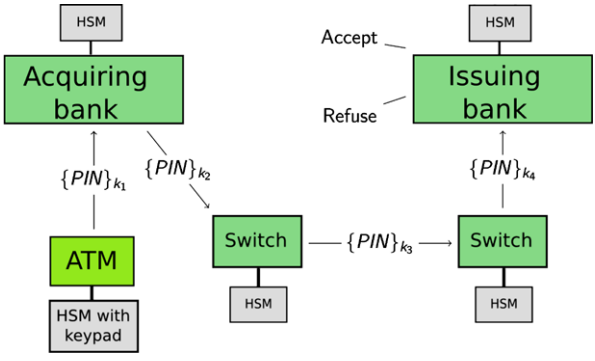


**Fig. 1** Bank network

PIN, i.e., the PIN that arrives encrypted to the bank, and the user PIN. The user PIN is computed by encrypting the validation data (i.e., some public data which include, e.g., the user Personal Account Number (PAN)) with the PIN derivation key (stored at the bank) and obtaining a 16 hexadecimal digit string. A substring of this string is extracted and decimalized via a decimalization table that maps each hexadecimal digit into a decimal number. Finally, an *offset* is added. The dectab attack consists of deducing the PIN digits by modifying some information, e.g., the way numbers are decimalized and by observing if this affects the result of the verification. The position of the guessed PIN digits is reconstructed by manipulating the offset of the PIN which is a public parameter. By combining all this information the attacker is able to reconstruct the whole PIN.

### 1.1 Our Contribution

In this paper we formally show how 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. Modifying the dectab, corresponds to disclosing the presence of certain digits in the PIN, analogously to the white marker in the Mastermind Game. On the other hand, manipulating the dectab and the offset together is similar to asking the codemaker to disclose both the color and the position of one PIN digit, in case the guess is correct, similarly to what happens with a black marker of the game.

The observation that the Mastermind game and the problem of extracting PIN digits via dectab attacks are closely related is not entirely new, as it is briefly mentioned in [7]. However, in the following we formalize the above intuition by showing how PIN cracking and Mastermind can be seen as instances of a more general problem, or game, the Extended Mastermind Game. We also suggest a new way of improving the dectab attack: The idea is to allow the codebreaker (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.

To this aim, we develop a computer program that optimizes a well known technique presented by Knuth in [19] for the standard Mastermind game and extend it to our more general setting. We perform experiments showing that the program is almost as precise as state-of-the-art Mastermind solvers [17] but faster, being it able to compute strategies for cases not yet covered. More interestingly, the very same solving strategy can be adapted to the solution of the PIN cracking problem. We have also noticed that this algorithm can be further refined using a few heuristics that use more sophisticated guesses only when the algorithm is not able to reduce the solution space. This leads to new interesting upper bounds on the number of average API calls for performing the attacks on PINs of length 4 and 5.

### 1.2 Paper Structure

In Sect. 1.3 we briefly summarize the related literature. In Sect. 2 we formally define the two problems, i.e., the Generalized Mastermind Problem and the PIN Cracking Problem. In Sect. 3 we introduce the Extended Mastermind Problem, i.e., a general problem whose instances are the above mentioned problems. In Sect. 4 we expose some experimental results, and we conclude in Sect. 5.

### 1.3 Related Literature

*Mastermind* In [19] Donald Knuth considered the standard Mastermind game, played using pegs of 6 different colors, in a sequence of length 4. The codebreaker has to uncover a secret sequence selected from $6^4 = 1296$ elements. Knuth 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 full enumeration technique based on recursive backtracking methods that uses at most 6 guesses but decreases the expected number to 4.340 or to 4.341 if only 5 guesses are allowed [20]. The idea is to reduce the search space by considering only one of the so-called case-equivalent combinations. Other different approaches for the solution of the standard Mastermind problem have also been presented, as genetic algorithms [5, 18] and an evolutionary technique [4].

For the Generalized Mastermind Game with N colors and sequences of length $k$ some solutions have been presented. In [10] the authors propose a bound related to the one for finding a hidden code by asking questions. They also show that $\lceil \frac{k}{N} \rceil + 2N \log N + 2N + 2$ guesses are sufficient to solve the problem. In [17] the authors present some new bounds for this generalized 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.

Finally, different variants of the game have been proposed, e.g, in [11], 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 secret sequence. The Mastermind Satisfiability Problem (MSP), based on this static version, has been proved to be NP-complete [22].

*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 analyzing PIN recovery attacks and API-level attacks in general [21]. In particular, different models have been proposed, e.g., in [8] 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 [21]. In [9] we have presented, together with other authors, a language-based setting for analyzing PIN processing API via a type-system. We have formally modeled 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 [13] we have also have proposed a low-impact, easily implementable fix that adds an integrity check to the parameters passed to the PIN processing API that controls access to the tamper-resistant HSMs where PIN encryption, decryption and verification takes place. This

fix involves very little change to the existing ATM network infrastructure and 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*).

It is convenient to give here some general notions: both problems aim at guessing a secret $s$, picked from a set $\mathcal{S}$. When placing a guess $g$, from the set of possible guesses $\mathcal{G}$, the player obtains an answer $a_s(g) \in \mathcal{A}$, that is determined by the placed guess $g$ and the (unknown) secret $s$. This allows the player to deduce that the secret $s$ is in the set $\{v \in \mathcal{S} \mid a_v(g) = a_s(g)\}$ of *surviving candidates*, i.e., the set of all values in $\mathcal{S}$ that give the same answer as the one just received, when placing guess $g$. Notice that this set can be computed without knowing $s$, once $a_s(g)$ is known.

**Definition 1** (Guessing game) A *guessing game* is a triple $\langle \mathcal{S}, \mathcal{G}, a \rangle$ with $a : \mathcal{S} \times \mathcal{G} \to \mathcal{A}$. A *play* is a sequence $\langle g_1, \mathcal{S}_1 \rangle, \ldots, \langle g_l, \mathcal{S}_l \rangle$, where $\mathcal{S}_i = \{v \in \mathcal{S} \mid a_v(g_i) = a_s(g_i)\}$, $\{s\} \neq \mathcal{S}_1 \cap \cdots \cap \mathcal{S}_{l-1}$ and $\{s\} = \mathcal{S}_1 \cap \cdots \cap \mathcal{S}_l$.

Intuitively, $\langle g_i, \mathcal{S}_i \rangle$, represents the guess $g_i$ at step $i$ and the set $\mathcal{S}_i$ of surviving candidates computed from $g_i$ and the answer $a_s(g_i)$. The constraints $\{s\} \neq \mathcal{S}_1 \cap \cdots \cap \mathcal{S}_{l-1}$ and $\{s\} = \mathcal{S}_1 \cap \cdots \cap \mathcal{S}_l$ implies that the last guess $g_l$ is exactly the one that reveals the secret $s$.

A (*deterministic*) *strategy* is a way of computing the next guess $g \in \mathcal{S}$, given a sequence $\langle g_1, \mathcal{S}_1 \rangle, \ldots, \langle g_i, \mathcal{S}_i \rangle$ of previous guesses. A strategy can give rather different outcomes when varying the secret since this, of course, affects the answers and the relative surviving candidates. It is thus interesting to look for strategies that are minimal. A strategy is

*w-minimal* if it minimizes the number of guesses in the worst-case;
*a-minimal* if it minimizes the number of guesses in the average-case, assuming that secrets are uniformly picked from $\mathcal{S}$.

For the sake of simplicity, in the following we will use the term minimal in order to indicate either a *w-minimal*or an *a-minimal*strategy.

### 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 human (the "codemaker"). The codemaker chooses a linear sequence of $k$ colored pegs (duplicates are allowed), 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


**Fig. 2** An example of a Mastermind game

codebreaker and a response of the codemaker. The response can be empty (nothing has been correctly guessed), can contain a black or a white marker, i.e., is a sequence of at most $k$ 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.

As we have previously pointed out, in the standard Mastermind game the set of all possible solutions has size $6^4$, in the Generalized Mastermind Game, however, the size explodes to $N^k$, thus running plain exhaustive search techniques might become problematic when $N$ and $k$ increase too much.

An example of the standard Mastermind game, played with $N = 6$ colors and $k = 4$ pegs, is shown in Fig. 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 $W$. 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 $B$ and a white $W$ peg, an so on. At the last step the response is a sequence of 4 black markers, i.e., $B, B, B, B$.

More formally, let $\mathcal{C} = \{0, 1, \ldots, N-1\}$ be the set of colors, and let $\mathcal{S} = \mathcal{G} = \mathcal{C}^k$, i.e., both the secret $s \in \mathcal{S}$ and guesses $g \in \mathcal{G}$ are tuples of $k$ colors respectively noted $(c_1^s, c_2^s, \ldots, c_k^s)$ and $(c_1^g, c_2^g, \ldots, c_k^g)$. Answers $a_s(g)$ are pairs $(b_s(g), w_s(g))$ where $b_s(g)$ and $w_s(g)$ are respectively the number of black and white markers, as defined below.

**Definition 2** (Black markers) The number of black markers is $b_s(g) = |\{i \in [1,k] \mid c_i^s = c_i^g\}|$.

The number of white markers, instead, is the number of matching colors between the secret and the guess which are not in the same position, i.e., we exclude black markers. To formalize this we first compute the number of occurrences of a color

$j \in \mathcal{C}$ in the secret code as $p_j = |\{i \in [1,k] \mid j = c_i^s\}|$, and in the guess as $q_j = |\{i \in [1,k] \mid j = c_i^g\}|$. 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.

**Definition 3** (White markers) The number of white markers is $w_s(g) = \sum_{j=1}^{N} \min(p_j, q_j) - b_s(g)$.

Let us illustrate the above definitions with a simple example

*Example 1* Let $N = 6$, $s = (c_1^s, c_2^s, c_3^s, c_4^s) = (1,2,3,1)$ be the secret and $g = (c_1^g, c_2^g, c_3^g, c_4^g) = (1,3,1,3)$ be the guess. We compute $b_s(g) = |\{i \in [1,k] \mid c_i^s = c_i^g\}| = |\{1\}| = 1$. In fact only the first '1' is in the right position, giving a black marker. Then we have

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

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 $w_s(g) = 3 - b_s(g) = 2$. 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 $\min(p_j, q_j)$.

The Generalized Mastermind Problem (*GMP*) consists of devising a minimal strategy for the Generalized Mastermind Game played on $N$ colors and $k$ pegs. Formally:

**Definition 4** (The Generalized Mastermind Problem—*GMP*) Devise a minimal strategy for the game $\langle \mathcal{C}^k, \mathcal{C}^k, a \rangle$ with $a_s(g) = (b_s(g), w_s(g))$.

### 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 traveling 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 (PIN_V for short). We focus our attention on PIN_V and we report its code in Table 1. PIN_V 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. This latter PIN arrives encrypted under key $k$ as *EPB* (Encrypted PIN block).

**Table 1** The *verification* API

```
PIN_V (PAN, EPB, len, offset, vdata, dectab) {
    x1 := enc_pdk(vdata);
    x2 := left(len, x1);
    x3 := decimalize(dectab, x2);
    x4 := sum_mod10(x3, offset);
    x5 := dec_k(EPB);
    x6 := fcheck(x5);
    if (x6 = ⊥) then return("format wrong");
    if (x4 = x6) then return("PIN correct");
                 else return("PIN wrong")}
```

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 decimalized 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) [1] 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 [8]. 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*, position by position, until the API returns again that the PIN is correct.

[1] Note that there are attacks, not addressed in this paper, that rely on change of format in the message. Thus, $x_6$ fails given that the arrived EPB, once decrypted, has a format that is different from the expected one.

We illustrate the attack through a simple example.[2]

*Example 2* Assume that $dectab = 0123456789012345$, as

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

and that EPB $= \{|5997, r|\}_k$, and $len = 4$, $offset = 4732$.
Given $x_1 = \mathsf{enc}_{pdk}(vdata) = \text{BC6595FDE32BA101}$ the correct solution, unknown to the intruder, is the following.

$$\left|\begin{array}{l|l} x_2 = \mathsf{left}(4, BC6595FDE32BA101) & = \text{BC65} \\ x_3 = \mathsf{decimalize}(dectab, BC65) & = 1265 \\ x_4 = \mathsf{sum\_mod10}(1265, 4732) & = 5997 \\ x_5 = \mathsf{dec}_k(\{|5997, r|\}_k) & = (5997, r) \\ x_6 = \mathsf{fcheck}(5997, \mathsf{r}) & = 5997 \\ x_6 \neq \perp & \\ x_4 = x_6 & \mathsf{return}(''PIN\ correct'') \end{array}\right|$$

The attacker, unaware of the value of the PIN, first tries to discover whether or not 0 appears in $x_3$, so it changes the dectab, which is a public parameter, as

$$dectab' = \underline{1}123456789\underline{1}12345,$$

i.e., it replaces the two 0's by 1's. Invoking the API with $dectab'$ he obtains

$$\mathsf{decimalize}(dectab', BC65) = \mathsf{decimalize}(dectab, BC65) = 1265,$$

that is $x_3$ remains unchanged and 0 does not appear in it. The attacker proceeds by replacing the 1's of *dectab* by 2's: with

$$dectab'' = 0\underline{2}23456789\underline{2}2345,$$

he has

$$\left|\begin{array}{l|l} x_3 = \mathsf{decimalize}(dectab'', BC65) & = 2265 \neq 1265 \\ x_4 = \mathsf{sum\_mod10}(2265, 4732) & = 6997 \neq 5997 \\ x_5 = \mathsf{dec}_k(\{|5997, r|\}_k) & = (5997, r) \\ x_6 = \mathsf{fcheck}(5997, \mathsf{r}) & = 5997 \\ x_6 \neq \perp & \\ x_4 \neq x_6 & \mathsf{return}(''PIN\ wrong'') \end{array}\right|$$

[2]For simplicity we consider the "standard" decimalization table, in practice the mapping is much more complicated.

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 $1+4 \bmod 10 = 5$. He then proceeds using the same technique to discover the other PIN digits.

We can now formulate our problem. Let $\mathcal{S} = [0,9]^k$, i.e., the secret $s \in \mathcal{S}$ is a tuple of $k$ digits $(d_1^s, d_2^s, \ldots, d_k^s)$, analogously to the Generalized Mastermind played with 10 colors, i.e., with $N = 10$. Guesses $\mathcal{G}$ instead are malicious calls to the API. As we focus on dectab-based attacks, we only consider the value of the dectab and the offset. Moreover, we only consider changes of dectab that preserve equal mappings, i.e., if the original dectab mapped $d_1$ and $d_2$ to the same decimal value, even the modified dectab will do it. The reason is that we are not interested in deducing the hexadecimal digits but just the decimal ones. We will more precisely discuss this issue when we will compare PIN cracking with Mastermind.

Let *dectab* be the original dectab. We call $\mathcal{T}$ the set of functions $dectab'$: $[0,15] \mapsto [0,9]$ such that $dectab(d_1) = dectab(d_2)$ implies $dectab'(d_1) = dectab'(d_2)$. Since an offset is a tuple of $k$ digits, we let $\mathcal{G} = \mathcal{T} \times [0,9]^k$. The answer $a_s(g)$ is the one given by the PIN verification API, called with all the original parameters (noted _), except for the dectab and offset which are the ones included in the guess $g$.

**Definition 5** (The PIN Cracking Problem—*PCP*) Devise a minimal strategy for the game

$$\langle [0,9]^k, \mathcal{T} \times [0,9]^k, a \rangle$$
$$\text{where } a_s(dectab', offset') = \mathsf{PIN\_V}(\_, \_, \_, offset', \_, dectab').$$

Notice that, since we do not change the encrypted PIN, the only possible answers are "PIN correct" and "PIN wrong". The wrong format error, in fact, can only happen if the EPB, once decrypted, is not in the expected format.

## 3 Extended Mastermind

In this section we extend the Generalized Mastermind Problem previously presented, 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 sufficient that one of the pegs in the set is correct to get a black

or a white marker. E.g., given $N = 6$ colors, $k = 4$ positions, $(1, 5, 3, 1)$ the secret and $(\{1\}, \{3, 4, 5\}, \{1, 2\}, \{0\})$ the guess, then the result is two black markers for the first two positions of the guess, i.e., $B, B$, and one white marker for the third position of the guess, i.e., $W$. We show the correspondence between the values of the secret and the values in the guess as underlined values in the guess, i.e., $(\{\underline{1}\}, \{3, 4, \underline{5}\}, \{\underline{1}, 2\}, \{0\})$.

As for the Generalized Mastermind, let $\mathcal{C} = \{0, 1, \ldots, N-1\}$ be the set of colors, and let $\mathcal{S} = \mathcal{C}^k$, i.e., the secret $s \in \mathcal{S}$ be a tuple of $k$ colors, noted $(c_1^s, c_2^s, \ldots, c_k^s)$. The set of guesses, instead, is now $\mathcal{G} = (2^{\mathcal{C}})^k$. In fact, a guess $g$ is a tuple of $k$ sets of colors noted $(C_1^g, C_2^g, \ldots, C_k^g)$, with $C_1^g, \ldots, C_k^g \subseteq \mathcal{C}$.

In this new extended game, the number of black markers represents the number of colors in the secret that belong to the corresponding (i.e., in the same position) set of colors in the guess, i.e., the matching condition $c_i^s = c_i^g$ of Definition 2 is changed into $c_i^s \in C_i^g$.

**Definition 6** (Black markers) The number of black markers is $\hat{b}_s(g) = |\{i \in [1, k] \mid c_i^s \in C_i^g\}|$.

The number of white markers is computed similarly to Definition 3 as

$$\hat{w}_s(g) = \sum_{j=1}^{N} \min(p_j, q_j) - \hat{b}_s(g),$$

where $q_j$ now reflects the existence of the extended guesses, thus: $q_j = |\{i \in [1, k] \mid j \in C_i^g\}|$.

*Example 3* Consider again $N = 6$, and secret $(c_1^s, c_2^s, c_3^s, c_4^s) = (1, 2, 3, 1)$ as in Example 1. We have shown that the guess $(1, 3, 1, 3)$ gives 1 black (the 1 in the first position) and 2 whites (one of the 3's and the 1 in the third position). Consider now the extended guess $(C_1^g, C_2^g, C_3^g, C_4^g) = (\{1\}, \{3\}, \{1\}, \{1, 3\})$ which includes set $\{1, 3\}$ in the last position. In this case we have $\hat{b}_s(g) = |\{i \in [1, k] \mid c_i^s \in C_i^g\}| = |\{1, 4\}| = 2$, as the first and the fourth pegs belong to the corresponding sets $\{1\}$ and $\{1, 3\}$ in the guess.

The computation of $p_j$ is the same as the one of Example 1, while $q_j$ differs for what concerns color 1, i.e., $q_1 = |\{1, 3, 4\}| = 3$. In fact, color 1 now also appears in the set $\{1, 3\}$, in the fourth position. This however does not affect the computation of $\sum_{j=1}^{N} \min(p_j, q_j)$ since $p_1 = 2$ and we still get $\sum_{j=1}^{N} \min(p_j, q_j) = 3$. Intuitively, there are still 3 matching pegs (two of the 1's and one of the 3's). The number of white markers, however, differs as we now have two black markers to subtract, giving $\hat{w}_s(g) = 3 - \hat{b}_s(g) = 1$. Color 3 in the second position of the guess is the only matching peg in the wrong position since the first and the fourth 1s in the secret are computed as black markers.

We can now define the Extended Mastermind problem played on $N$ colors and $k$ pegs.

**Definition 7** (The Extended Mastermind Problem—*EMP*) Devise a minimal strategy for the game $\langle \mathcal{C}^k, (2^{\mathcal{C}})^k, a\rangle$ with $a_s(g) = (\hat{b}_s(g), \hat{w}_s(g))$.

### 3.1 Connecting the Games

We now show that the first two games, *GMP* and *PCP*, are instances of *EMP*, i.e., can be played as a restricted version of the Extended Mastermind Problem. In a sense, this proves that *EMP* is a super-game of the other two.

**Definition 8** A game $\langle \mathcal{S}, \mathcal{G}, a\rangle$ with $a\colon \mathcal{S} \times \mathcal{G} \to \mathcal{A}$ is an instance of another game $\langle \mathcal{S}, \mathcal{G}', a'\rangle$ with $a'\colon \mathcal{S} \times \mathcal{G}' \to \mathcal{A}'$, iff there exist two functions $\alpha\colon \mathcal{G} \to \mathcal{G}'$ and $\beta\colon \mathcal{A}' \to \mathcal{A}$ such that $\beta(a'_s(\alpha(g))) = a_s(g)$, for all $s \in \mathcal{S}$.

Intuitively, $\alpha$ maps a guess of the first game into a guess of the second game and $\beta$ maps the answer of the second game back into an answer for the first game. If this coincides with the actual answer in the first game, we have that we can use an instance of the second game to solve the first one, as stated in the following proposition.

**Proposition 1** *Let $\langle \mathcal{S}, \mathcal{G}, a\rangle$ be an instance of $\langle \mathcal{S}, \mathcal{G}', a'\rangle$, $\hat{\mathcal{G}} = \mathrm{img}(\alpha)$ and $\hat{a}_s(\hat{g}) = \beta(a'_s(\hat{g}))$, for all $\hat{g} \in \hat{\mathcal{G}}$. Then, the problem of devising a minimal strategy for $\langle \mathcal{S}, \mathcal{G}, a\rangle$ is equivalent to the one of devising a minimal strategy for $\langle \mathcal{S}, \hat{\mathcal{G}}, \hat{a}\rangle$.*

*Proof* We prove the proposition for both *w-minimal* and *a-minimal* strategies. Strategies for the games $\langle \mathcal{S}, \mathcal{G}, a\rangle$ and $\langle \mathcal{S}, \hat{\mathcal{G}}, \hat{a}\rangle$ are formalized as family of functions $\sigma_i : (\mathcal{G} \times 2^{\mathcal{S}})^i \to \mathcal{G}$ and $\hat{\sigma}_i : (\hat{\mathcal{G}} \times 2^{\mathcal{S}})^i \to \hat{\mathcal{G}}$, respectively. In fact, they take sequences in the form $\langle g_1, \mathcal{S}_1\rangle, \ldots, \langle g_i, \mathcal{S}_i\rangle$ and $\langle \hat{g}_1, \mathcal{S}_1\rangle, \ldots, \langle \hat{g}_i, \mathcal{S}_i\rangle$, and respectively return the next guesses $g_{i+1}$ and $\hat{g}_{i+1}$. We now define how to construct a strategy $\hat{\sigma}_i$ from $\sigma_i$ and vice-versa.

$$\hat{\sigma}_i(\langle \hat{g}_1, \mathcal{S}_1\rangle, \ldots, \langle \hat{g}_i, \mathcal{S}_i\rangle) = \alpha(\sigma_i(\langle \alpha^{-1}(\hat{g}_1), \mathcal{S}_1\rangle, \ldots, \langle \alpha^{-1}(\hat{g}_i), \mathcal{S}_i\rangle)) \tag{1}$$

$$\sigma_i(\langle g_1, \mathcal{S}_1\rangle, \ldots, \langle g_i, \mathcal{S}_i\rangle) = \alpha^{-1}(\hat{\sigma}_i(\langle \alpha(g_1), \mathcal{S}_1\rangle, \ldots, \langle \alpha(g_i), \mathcal{S}_i\rangle)) \tag{2}$$

for some $\alpha^{-1} : \hat{\mathcal{G}} \to \mathcal{G}$ such that $\forall \hat{g} \in \hat{\mathcal{G}}, \alpha(\alpha^{-1}(\hat{g})) = \hat{g}$.

From now on, we will write $\hat{\sigma}_i = \alpha(\sigma_i)$ and $\sigma_i = \alpha^{-1}(\hat{\sigma}_i)$ as shortcuts for (1) and (2) above. We now prove that any play in game $\langle \mathcal{S}, \hat{\mathcal{G}}, \hat{a}\rangle$ following $\hat{\sigma}_i$ can be naturally mapped into a play of the same length, in game $\langle \mathcal{S}, \mathcal{G}, a\rangle$ following $\alpha^{-1}(\hat{\sigma}_i)$. More specifically, consider $\langle \hat{g}_1, \mathcal{S}_1\rangle, \ldots, \langle \hat{g}_l, \mathcal{S}_l\rangle$ such that $\hat{g}_{i+1} = \hat{\sigma}_i(\langle \hat{g}_1, \mathcal{S}_1\rangle, \ldots, \langle \hat{g}_i, \mathcal{S}_i\rangle)$, for $1 \le i \le l$.

Let us now show that the corresponding play following $\sigma_i = \alpha^{-1}(\hat{\sigma}_i)$ in the first game is exactly $\langle \alpha^{-1}(\hat{g}_1), \mathcal{S}_1\rangle, \ldots, \langle \alpha^{-1}(\hat{g}_l), \mathcal{S}_l\rangle$. We first show that when playing $\hat{g}_i$ and $\alpha^{-1}(\hat{g}_i)$ in games $\langle \mathcal{S}, \hat{\mathcal{G}}, \hat{a}\rangle$ and $\langle \mathcal{S}, \mathcal{G}, a\rangle$, we obtain the same set of surviving candidates $\mathcal{S}_i$. In the former game we have $\mathcal{S}_i = \{v \in \mathcal{S} \mid \hat{a}_v(\hat{g}_i) = \hat{a}_s(\hat{g}_i)\} = \{v \in \mathcal{S} \mid \beta(a'_v(\hat{g}_i)) = \beta(a'_s(\hat{g}_i))\}$. In the latter game, instead, the set is computed as $\{v \in \mathcal{S} \mid a_v(\alpha^{-1}(\hat{g}_i)) = a_s(\alpha^{-1}(\hat{g}_i))\}$. Notice now that $\hat{g}_i = \alpha(\alpha^{-1}(\hat{g}_i))$ and, by Definition 8, we have $\beta(a'_s(\hat{g}_i)) = a_s(\alpha^{-1}(\hat{g}_i))$ for all $s \in \mathcal{S}$, thus $\{v \in \mathcal{S} \mid a_v(\alpha^{-1}(\hat{g}_i)) = a_s(\alpha^{-1}(\hat{g}_i))\} = \{v \in \mathcal{S} \mid \beta(a'_v(\hat{g}_i)) = \beta(a'_s(\hat{g}_i))\} = \mathcal{S}_i$.

We now proceed by induction on index $i$ of $g_i$. The base case is $i = 1$. By (2) we have $\sigma_0() = \alpha^{-1}(\hat{\sigma}_0()) = \alpha^{-1}(\hat{g}_1)$. Assume now the thesis for $i \geq 0$ and consider $\sigma_i(\langle\alpha^{-1}(\hat{g}_1), \mathcal{S}_1\rangle, \ldots, \langle\alpha^{-1}(\hat{g}_i), \mathcal{S}_i\rangle) = \alpha^{-1}(\hat{\sigma}_i(\langle\hat{g}_1, \mathcal{S}_1\rangle, \ldots, \langle\hat{g}_i, \mathcal{S}_i\rangle)) = \alpha^{-1}(\hat{g}_{i+1})$.

Similarly, we can prove that any play $\langle g_1, \mathcal{S}_1\rangle, \ldots, \langle g_l, \mathcal{S}_l\rangle$ following $\sigma_i$ is naturally mapped into the play $\langle\alpha(g_1), \mathcal{S}_1\rangle, \ldots, \langle\alpha(g_l), \mathcal{S}_l\rangle$ following $\alpha(\sigma_i)$. Thus, transforming strategies via (1) and (2) preserves the length of the plays and so even the worst and average cases, when playing with different secrets $s$ picked from $\mathcal{S}$.

Let now $\hat{\sigma}_i$ be a minimal strategy for $\langle\mathcal{S}, \hat{\mathcal{G}}, \hat{a}\rangle$ and assume, by contradiction, that $\sigma_i = \alpha^{-1}(\hat{\sigma}_i)$ is not the minimal strategy for $\langle\mathcal{S}, \mathcal{G}, a\rangle$ since there exist a better strategy $\sigma_i'$. By applying $\alpha$ we obtain that $\alpha(\sigma_i')$ produces plays of the same length as $\sigma_i'$ and is thus better than $\hat{\sigma}_i$, giving a contradiction. The same reasoning can be followed to show that if $\sigma_i$ is minimal for $\langle\mathcal{S}, \mathcal{G}, a\rangle$ then $\alpha(\sigma_i)$ is minimal for $\langle\mathcal{S}, \hat{\mathcal{G}}, \hat{a}\rangle$. □

**Proposition 2** *GMP is an instance of EMP.*

*Proof* Recall that *GMP* is defined as $\langle\mathcal{C}^k, \mathcal{C}^k, a\rangle$ with $a_s(g) = (b_s(g), w_s(g))$ while *EMP* is the game $\langle\mathcal{C}^k, (2^{\mathcal{C}})^k, a'\rangle$ with $a'_s(g) = (\hat{b}_s(g), \hat{w}_s(g))$. Notice that we have the same set $\mathcal{S} = \mathcal{C}^k$. We now consider the following $\alpha(c_1, \ldots, c_k) = (\{c_1\}, \ldots, \{c_k\})$ and $\beta(b, w) = (b, w)$. Intuitively, a guess in *GMP* is mapped into a guess in *EMP* of singleton sets, while the answer is just mapped back as it is. We now show that $\beta(a'_s(\alpha(g))) = a_s(g)$. Since $\beta$ is the identity function we just prove that $a'_s(\alpha(g)) = a_s(g)$, i.e., $a'_s(\{c_1\}, \ldots, \{c_k\}) = a_s(c_1, \ldots, c_k)$. This amounts to prove that $\hat{b}_s(\{c_1\}, \ldots, \{c_k\}) = b_s(c_1, \ldots, c_k)$ and $\hat{w}_s(\{c_1\}, \ldots, \{c_k\}) = w_s(c_1, \ldots, c_k)$. These are easily proved by observing that requiring $c_i^s \in C_i^g$ is trivially equivalent to $c_i^s = c_i^g$ whenever $C_i^g = \{c_i^g\}$, which is in fact the case here. □

Thus, by Proposition 1 we have that *GMP* is equivalent to *EMP* when guesses are restricted to tuples of singletons. In fact, in the proof we use $\alpha(c_1, \ldots, c_k) = (\{c_1\}, \ldots, \{c_k\})$ and $\mathrm{img}(\alpha)$ is exactly the set of all tuples of $k$ singleton sets of colors. This is somehow expected, as *EMP* generalizes *GMP* by using sets instead of single colors.

More interestingly, we prove that the PIN cracking problem is also an instance of *EMP*.

**Proposition 3** *PCP is an instance of EMP played with* 10 *colors.*

*Proof* Recall that *PCP* is defined as $\langle[0, 9]^k, \mathcal{T} \times [0, 9]^k, a\rangle$ where $\mathcal{T}$ is the set of functions $dectab' : [0, 15] \mapsto [0, 9]$ such that $dectab(d_1) = dectab(d_2)$ implies $dectab'(d_1) = dectab'(d_2)$ and $a_s(dectab', offset') = \mathsf{PIN\_V}(\_, \_, \_, offset', \_, dectab')$.

Let $(c_1^s, \ldots, c_k^s) \in [0, 9]^k$ be the secret PIN. We write $offset = o_1, \ldots, o_k$ and $dectab = d_0, \ldots, d_{15}$. Moreover, we let

$$\Delta^{off} = (\Delta_1^{off}, \ldots, \Delta_k^{off}) = (o'_1 - o_1, \ldots, o'_k - o_k) \mod 10$$

$$\Delta^{dec} = (\Delta_0^{dec}, \ldots, \Delta_{15}^{dec}) = (d'_0 - d_0, \ldots, d'_{15} - d_{15}) \mod 10$$

Intuitively, $\Delta^{off}$ and $\Delta^{dec}$ are differences (modulo 10) between the new and old offset digits and dectab mappings. From the condition above, we know that $dectab(d_1) = dectab(d_2)$ implies $\Delta^{dec}_{d_1} = \Delta^{dec}_{d_2}$.

We define $\alpha(dectab', offset') = (C_1, \ldots, C_k)$ where $C_i = \{d_j + o_i \mod 10 \mid j \in [0, 15] \wedge \Delta^{off}_i \equiv -\Delta^{dec}_j (\mathrm{mod}\ 10)\}$. Intuitively, $C_i$ contains all PIN digits for which the new dectab and offset have compensating variations. $\beta(b, w)$ is as follows:

$$\beta(b, w) = \begin{cases} \textit{“PIN correct”} & \text{if } b = k \wedge w = 0 \\ \textit{“PIN wrong”} & \text{otherwise} \end{cases}$$

Recall that *EMP* is the game $\langle \mathcal{C}^k, (2^{\mathcal{C}})^k, a' \rangle$ with $a'_s(g) = (\hat{b}_s(g), \hat{w}_s(g))$. We want to prove that $\beta(a'_s(\alpha(dectab', offset'))) = a_s(dectab', offset')$. We consider the case $a_s(dectab', offset') = \textit{“PIN correct”}$. By the code of PIN_V of Table 1, we know that $x_4 = \mathsf{decimalize}(dectab', x_2) + offset' \mod 10$ and the call returns *“PIN correct”* if and only if this value is equal to the one computed using the original values of offset and dectab, i.e., $\mathsf{decimalize}(dectab, x_2) + offset \mod 10$. Let us focus on an single $i$-th digit of $x_4$. Using the above notation, obtain $a_s(dectab', offset') = \textit{“PIN correct”}$ if and only if, for all $1 \le i \le k$ we have $d'_{x^i_2} + o'_i \equiv d_{x^i_2} + o_i (\mathrm{mod}\ 10)$ that is $o'_i - o_i \equiv d_{x^i_2} - d'_{x^i_2} (\mathrm{mod}\ 10)$ or, equivalently, $\Delta^{off}_i \equiv -\Delta^{dec}_{x^i_2} (\mathrm{mod}\ 10)$.

By definition of $\alpha$ and $\beta$ we also know that $\beta(a'_s(\alpha(dectab', offset'))) = \textit{“PIN correct”}$ if and only if $a'_s(C_1, \ldots, C_k) = (k, 0)$, i.e., if and only if we have $k$ black markers meaning that $\hat{b}_s(C_1, \ldots, C_k) = |\{i \in [1, k] \mid c^s_i \in C_i\}| = k$. This happens if and only if $c^s_i \in C_i$ for $1 \le i \le k$. To conclude we need to prove that this holds if and only if $\Delta^{off}_i \equiv -\Delta^{dec}_{x^i_2} (\mathrm{mod}\ 10)$. Since $c^s_i = d_{x^i_2} + o_i \mod 10$ we have that $\Delta^{off}_i \equiv -\Delta^{dec}_{x^i_2} (\mathrm{mod}\ 10)$ implies $d_{x^i_2} + o_i \mod 10 \in C_i$ that is $c^s_i \in C_i$. To see the opposite inclusion, assume that $c^s_i \in C_i$. There must exists $j$ such that $c^s_i = d_j + o_i \mod 10$ because of $\Delta^{off}_i \equiv -\Delta^{dec}_j (\mathrm{mod}\ 10)$. From $c^s_i = d_{x^i_2} + o_i \mod 10 = c^s_i = d_j + o_i \mod 10$ we get $d_{x^i_2} = d_j$ which we have noticed to imply $\Delta^{dec}_{x^i_2} = \Delta^{dec}_j$ from which the thesis. □

Thanks to Proposition 1 we know that we can solve the PIN cracking problem by playing an extended Mastermind game restricted to guesses in $\mathrm{img}(\alpha)$, and only considering answers containing $k$ black markers (all the other answers are ‘collapsed’ into the same one, i.e., *“PIN wrong”*). It is easy to see that $\mathrm{img}(\alpha)$ is composed of all extended guesses in which the sets of colors (or digits) are either disjoint or the same. For example, for $k = 4$, we might have $(\{1, 4\}, \{7, 2\}, \{5, 8\}, \{7, 2\})$. In fact, the intuition behind $\alpha$, in the proof, is to ‘collect’ in the sets all the PIN digits for which the modified dectab compensates the change in the offset. These sets are actually either disjoint or exactly the same, since the modified dectab can either compensate or not the modification in the offset. Consider again set $(\{1, 4\}, \{7, 2\}, \{5, 8\}, \{7, 2\})$, and suppose, for example, that the original offset is $(1, 5, 4, 2)$ and it is changed into $(0, 3, 1, 0)$, giving a change of $(-1, -2, -3, -2)$. Now if the dectab changes the

mappings producing 1 and 4 of the first set into 2 and 5 (by adding 1), these two digits will actually fall in the first set of the guess, as this compensate the $-1$ variation in the first position of the offset. Notice that 7 and 2 in the second and fourth guess need to be changed into 9 and 4 (by adding 2) since the offset variation is $-2$ in both those positions, and so on.

The example above also illustrates why we extended the Mastermind game to allow for guessing sets of colors. In fact, in order to deal with simultaneous variations of the dectab we need to be able to place guesses that only deduce partial information on the color in a certain position. Consequently, it is not possible to map PIN cracking into the Generalized Mastermind. This idea of simultaneous variations of the dectab is also the key ingredient for the improvements on known bounds presented in the next section.

Finally, the above proposition is based on the assumption that variations of the dectab preserve equal mappings. This is necessary to map the *PCP* problem into *EMP* with 10 colors. Dropping this assumption would still allow to prove that *PCP* is an instance of *EMP* with 16 colors. Intuitively, the proof would follow the above one focusing on guessing the first $k$ digits of variable $x_2$ instead of $x_4$, i.e., the PIN before decimalization and the offset sum.

## 4 Experimental Results

We have devised a program which is an optimized extension of the original program for Mastermind presented by Knuth in [19]. It works as follows:

1. Try all the possible guesses. For each guess, compute the number of 'surviving' solutions related to each possible outcome of the guess;
2. pick the guess from the previous step which minimizes the maximum number of surviving solutions among all the possible outcomes and perform the guess:
   (a) For each possible outcome, store the corresponding surviving solutions and recursively call this algorithm;
   (b) stop 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 heuristic 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 [17].

By applying the optimized algorithm to the Generalized Mastermind Problem we were able to find new upper bounds on the minimal number of moves for unknown values (see values in bold of Table 2). As a matter of fact, as it is mentioned in [16], Knuth's idea does not define an optimal strategy, it is however very close to the optimal. In [17] 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 known exact ones. We were also able to efficiently find bounds on 2 colors and 9 and 10 pegs and 3 colors and 8 pegs, and we also provided the exact sequence of moves to be followed. As the authors of [17] state, the computation of the above

**Table 2** Our optimization of Knuth's algorithm

| Colors/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 3** Bounds from [17]

| Colors/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 | | | | |

new values would probably take "many weeks" with their strategy, whereas in our case most computations took just few seconds, others few minutes.

Our experiments have also been oriented towards the investigation of the solution of the PIN cracking problem. We have applied the very same algorithm and in this case we have noticed that using sets with more than two elements in the guesses did not improve much the solutions. With sets of size at most two the algorithm performs quite well and in [14, 15] we have been able to improve the results of [21] for cracking PINs of length 4 from an average number of calls of 16.145 to 14.484.[3] The 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 illustrated in [8] and studied in [21]. Notice that in [8, 21] 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 $(\mathcal{C}, \ldots, \mathcal{C})$, with

[3]The values here reported are the correct ones. As it was mentioned during the presentation of [14], the published values for the PIN cracking problem (PINs of length 4 and 5) are slightly different due to a small error in the code that sometimes used non disjoint sets for guesses. A corrected version of paper [14] is available at [15].

$\mathcal{C}$ containing the digits whose presence has to be checked. In [14, 15] we also found a new bound for PINs of length 5 giving an average of 20.88 calls, but with a running time of about 10 hours. Moreover, to reduce complexity, we have limited the search to guesses with sets of size 1.

In this paper we have further refined the algorithm so to better perform on the PIN cracking problem. We have noticed that the shape of the guesses used in the found strategies is quite regular. We have thus added a few heuristics that incrementally extend the set of guesses when needed: in particular we start from the 'dectab-only' guesses mentioned above, and we use more sophisticated guesses only when the algorithm is not able to reduce the solution space. We have parametrized the code so that the user can choose the size of the sets and can force the program to use all the guesses after the size of the solution space is below a certain limit. Interestingly, this has improved the results of [14, 15]. This is mainly due to the fact we have focused on the 'most useful' guesses and this has allowed us to perform a more complete search on that specific subsets, for example using guesses with sets of size 3 and 4. Moreover, recall that Knuth's algorithm is not exhaustive. Thus, by forcing some apparently non-convenient guesses at the beginning, later on it may happen that the search tree becomes more balanced.

Table 4 reports the pseudo-code of the algorithm. The full program (written in Python) is available at http://www.dsi.unive.it/~focardi/MM_PIN/. We have omitted the implementation details and all the output. The recursive function `do_guess` is initially invoked with the whole $\mathcal{S}$ set of possible secrets. It then finds a guess that minimizes the maximum number of surviving solutions on the two possible answers and it recurse on the two sets in which the actual set is partitioned. The resulting average number of guesses to break PINs of length 4 and 5 are, respectively, 14.47 and 19.3, improving the 14.48 and 20.88 of [14, 15]. The running times are, respectively, 18.3 seconds and 18.4 minutes on a Toshiba Portégé Laptop (2Gb RAM) running Ubuntu Linux 10.04 and Python 2.6.5, which is about 30 times faster than [14, 15]. The choice of Python is due to the fact that complex data structures such as nested lists and sets are native in the language and the resulting code is very compact and readable. Re-implementing the code in more performing languages such as C, the execution time would further reduce. Finally, note that the bound for PINs of length 5 was previously not known in the literature before [14], and our algorithm can find it in a matter of minutes.

It is finally worth noticing that our results are close to the optimal as it is shown by the following:

**Theorem 9** *The average number of API calls for the solution of the PIN cracking problem is at least* 13.362 *for PINs of length* 4 *and at least* 16.689 *for PINs of length* 5.

*Proof* Recall that the output of each API call is '*PIN correct*' and '*PIN wrong*', i.e., there are just two possible choices. It is well known that given $n$ solutions that have to be placed at the leaves of a tree, the binary tree that minimizes the average of the sum of the lengths to the leaves is an almost-balanced binary tree. Thus the base 2 logarithm already gives a theoretical bound to the optimal average number of moves. In

**Table 4** Pseudocode of the *EMP*-based solution of *PCP*

```
do_guess(S):
if |S| != 1:                  # not a single solution yet

 min = |S|                    # starting value, we want to decrease it
 rnd = 1                      # round 1, for heuristics
 guesses = []                 # empty list of guesses

 while min == |S| and rnd <= 2:
  # depending on the round and the number of surviving solutions, generates the
       guesses
  if rnd == 1: # heuristic 1 - dectab only
   # here we have guesses of the form [[list],....,[list]], the 'dectab only'
   # guesses, checking whether or not the PIN digits belong to the given list.
   # list is the list of all 'survived' colors
   guesses = guesses + < dectab only guesses >

  if r==2 or |S| < EXT_LIMIT: # heuristic 2, round 2 or size below a limit
   # the following guesses are of the form [[list1],[list2],....] with list1
   # and list2 complementary and no other list occurring in the guess.
   # list1 and list2 are made of 'survived' colors
   guesses = guesses + < complementary list guesses >


  for g in guesses:
   M_SOLS= S & surviving(g)  # intersect S and surviving solutions for g

   n_sol = max(|m_sols|,|S|-|m_sols|) # count matching and not and take the max

   if mas < min:                      # if we got a minimum, let's store it
     min = n_sol
     MIN_SOLS = M_SOLS

 rnd = rnd+1                          # next round (for heuristics)

if min < |S|:  # we are decreasing the surviving solution set size
               # let us perform the guess
 do_guess(MIN_SOLS)    # guess was right, we explore solutions in MIN_SOLS
 do_guess(S - MIN_SOLS)# guess was wrong, we explore solutions in S - MIN_SOLS

else:
 FAIL         # we are looping
```

practice, we can compute the average number of moves on the best almost-balanced binary tree by computing $s = n - 2^l$, with $l = \lfloor \log_2 n \rfloor$, representing the solutions that exceed the leaves of a perfectly balanced tree of depth $l$. Now, we have to accommodate for such leaves using depth $l + 1$. Notice that, when doing so, we need to use half of $s$ leaves at level $l$ as parents. The solution is to accommodate for twice $s$ at depth $l + 1$. In fact this requires exactly $s$ leaves to become parents. We would have $2^l - s$ at level $l$ and $2s$ leaves at level $l + 1$ for a total of $2^l + s = n$.

To compute the average number of API calls we can thus use the expression

$$avg = \frac{(2^l - s)l + 2s(l+1)}{n} = \frac{(2^{l+1} - n)l + (2\,n - 2^{l+1})(l+1)}{n}$$

For lengths of 4 and 5 we have:

$$avg_4 = \frac{(2^{14} - 10000) \times 13 + (20000 - 2^{14}) \times 14}{10000} = 6384 \times 13 + 3616 \times 14$$
$$\approx 13.362$$
$$avg_5 = \frac{(2^{17} - 100000) \times 16 + (200000 - 2^{17}) \times 17}{10000} = 31072 \times 13 + 68928 \times 14$$
$$\approx 16.689 \qquad \square$$

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 API calls from the value 16.145 of [21] to 14.484 in [14, 15], and further to 14.47 in this paper. We also found a new bound for PINs of length 5 giving an average number of API calls of first 20.88 in [14, 15], and then 19.3 here. 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 [17] to the PIN cracking setting.

**Acknowledgements** We would like to thank Graham Steel for his helpful comments and suggestions. Work partially supported by the RAS Project "*TESLA: Techniques for Enforcing Security in Languages and Applications*".

## 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. Bento, L., Pereira, L., Rosa, A.: Mastermind by evolutionary algorithms. In: Proc. ACM Symp. Applied Computing, San Antonio, Texas, 28 February–2 March, pp. 307–311. ACM Press, New York (1999)
5. Berghman, L., Goossens, D., Leus, R.: Efficient solutions for Mastermind using genetic algorithms. Technical report KBI 0806, Katholieke Universiteit Leuven, Department of Decision Sciences and Information Management, 2006

6. Berkman, O., Ostrovsky, O.M.: The unbearable lightness of PIN cracking. In: 11th International Conference, Financial Cryptography and Data Security (FC 2007), Scarborough, Trinidad and Tobago, February 12–16. Lecture Notes in Computer Science, vol. 48862, pp. 224–238. Springer, Berlin (2007)
7. Bond, M., Clulow, J.: Extending security protocol analysis: new challenges. Electron. Notes Theor. Comput. Sci. **125**, 13–24 (2005)
8. Bond, M., Zielinski, P.: Decimalization table attacks for pin cracking. Technical report UCAM-CL-TR-560, University of Cambridge, Computer Laboratory, 2003. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-560.pdf
9. Centenaro, M., Focardi, R., Luccio, F., Steel, G.: Type-based analysis of PIN processing APIs. In: Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS 09). Lecture Notes in Computer Science, vol. 5789, pp. 53–68. Springer, Berlin (2009)
10. Chen, Z., Cunha, C., Homer, S.: Finding a hidden code by asking questions. In: Computing and Combinatorics Second Annual International Conference (COCOON 96), Hong Kong, June 17–19. Lecture Notes in Computer Science, vol. 1090, pp. 50–55. Springer, Berlin (1996)
11. Chvatal, V.: Mastermind. Combinatorica **3**, 325–329 (1983)
12. Clulow, J.: The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003
13. Focardi, R., Luccio, F., Steel, G.: Blunting differential attacks on PIN processing APIs. In: Proceedings of the 14th Nordic Conference on Secure IT Systems (NORDSEC 09), October. Lecture Notes in Computer Science, vol. 5838/2009, pp. 88–103. Springer, Berlin (2009)
14. Focardi, R., Luccio, F.L.: Cracking bank pins by playing mastermind. In: Proc. Fith International Conference Fun with algorithms (FUN 10), June 2–4. Lecture Notes in Computer Science, vol. 6099, pp. 202–213. Springer, Berlin (2010)
15. Focardi, R., Luccio, F.L.: Cracking bank pins by playing mastermind. http://www.dsi.unive.it/~focardi/MM_PIN/FUN10corrected.pdf (2010). Corrected version of FUN10
16. Goddard, W.: Mastermind revisited. J. Comb. Math. Comb. Comput. **51**, 215–220 (2004)
17. Jäger, G., Pezarski, M.: The number of pessimistic guesses in generalized mastermind. Inf. Process. Lett. **109**, 635–641 (2009)
18. Kalisker, T., Camens, D.: Solving mastermind using genetic algorithms. In: Proc. Genetic and Evolutionary Computation Conference (GECCO 03), July 12–16. Lecture Notes in Computer Science, vol. 2724, pp. 1590–1591. Springer, Berlin (2003)
19. Knuth, D.: The computer as a master mind. J. Recreat. Math. **9**, 1–6 (1976)
20. Koyama, M., Lai, T.: An optimal mastermind strategy. J. Recreat. Math. **25**, 251–256 (1993)
21. Steel, G.: Formal analysis of PIN block attacks. Theor. Comput. Sci. **367**(1–2), 257–270 (2006)
22. Stuckman, J., Zhang, G.: Mastermind is NP-complete. INFOCOMP J. Comput. Sci. **5**, 25–28 (2006)