Automated Known-Plaintext Cryptanalysis of Short Hagelin M-209 Messages

George Lasry
Mishmar Hayarden 5
Givataim, 5358205 Israel
george.lasry@gmail.com

Nils Kopal
University of Kassel
Pfannkuchstr. 1
34121 Kassel, Germany
nils.kopal@uni-kassel.de

Arno Wacker
University of Kassel
Pfannkuchstr. 1
34121 Kassel, Germany
arno.wacker@uni-kassel.de

Abstract

The Hagelin M-209 portable encryption device was used by the US Army in World War II and the Korean War, as well as by other armies and in embassy settings. In this paper, we present a fully automated computerized known-plaintext attack, based on hillclimbing and a novel fitness function – the Aggregate Displacement Error. In our performance evaluation we show that we are able to recover key settings for messages as short as 50 characters. To validate our results, we solved several publicly available challenge messages, including a message with only 40 letters.

Keywords: Hagelin, M-209, Hillclimbing, Known-Plaintext Attack, Automated Cryptanalysis

1 Introduction

The Hagelin M-209, also known as CSP-1500, is a mechanical encryption device derived from the earlier C-36 which was developed by Boris Hagelin in 1936. It is built only of mechanical components, and does not require any power source. We show the mechanical internals of the device in Figure 1, and a functional diagram in Figure 2. The M-209 functions as a stream cipher, with a pseudo-random displacement sequence generator, and a Beaufort encoder, i.e. a Caesar cipher with an inverted alphabet, see Figure 1 (C). The pseudo-random displacement generator consists of two parts: A rotating cage with 27 bars (see Figure 1 (A) and Figure 2), and a set of six wheels (see Figure 1 (B) and Figure 2). The wheels are non-replaceable, unlike in later Hagelin models. Wheels 1, 2, 3, 4, 5, and 6 have 26, 25, 23, 21, 19 and 17 letters, respectively. Next to each letter, there is a pin which can be set to an effective or non-effective state. On each wheel, one of the pins is in the active position, against the bars of the cage. At each step of the encryption or decryption process, all the wheels rotate by exactly one step. Each bar in the cage has two movable lugs. Each lug may be set against any of the six wheels, or set to the neutral position (0), but both lugs may not be set against the same wheel. According to operating instructions [7], at least one of the two lugs should be set for each bar.

The operator usually changes the settings of the wheels pins and the lugs on a daily basis, according to key lists distributed periodically. For each message, he changes the initial position of the wheels, using an external indicator of 6 letters (see Figure 2). The operator encrypts the message letter by letter. He selects a plaintext letter using the disk on the left side of the device, and presses a power handle. The disk has 27 symbols, A to Z and a space symbol. Space symbols are internally replaced by the letter Z. When the power handle is pressed, all wheels rotate by one step, thus replacing the

The original submitted version is available at http://dx.doi.org/10.1080/01611194.2014.988370. This version can differ from the camera ready paper (due to copyright restrictions).
6 pins in the active position. In addition, the cage performs a full revolution around its 27 bars. For each bar, if any one of the two lugs was set against a wheel for which the pin in the active position is in effective state (see Figure 2), the bar is engaged and it moves to the left. The displacement used for encoding the current letter is equal to the number of bars engaged, and may have a value from 0 to 27. This displacement is then applied to the current plaintext letter, using a Beaufort scheme (see Figure 2), to form the ciphertext letter, as follows:

\[
CiphertextLetter[i] = (Z - PlaintextLetter[i] + Displacement[i]) \mod 26
\]

The letter A is represented by the number 0, B by 1, ... Z by 25. The device prints the ciphertext letters on a paper tape, on the left side of the device. The decryption process is essentially the same, except that encrypted messages do not contain the space symbol. Because the wheels have different number of pins, and those numbers are co-prime, the displacement sequence will not repeat itself until \(26 \cdot 25 \cdot 23 \cdot 21 \cdot 19 \cdot 17 = 101,405,850 \approx 2^{27}\) steps.

The rest of this paper is organized as follows: At first, we present in Section 2 an analysis of the keyspace of the M-209 as well as prior (manual) known-plaintext attacks. After that, we present our method for an automated known-plaintext attack in Section 3, including our hillclimbing algorithm, special transformations on pin settings and on lug settings, and our novel fitness function, the Aggregate Displacement Error. In Section 4 we present an evaluation of our attack, including performance measurements and an analysis of its work factor. We also applied our method to publicly available challenges and show the results. We finally conclude our paper in Section 5.

2 Cryptanalysis

In this chapter we first analyze the keyspace of the Hagelin M-209 machine. After that, we present an overview of prior known-ciphertext approaches.

2.1 Keypare

The settings of the device consists of the Wheels Settings which include the wheels pins and the initial position of the wheels, and of the Lug Settings - the settings of the lugs of the 27 bars. The overall keyspace consists of the combination of the keyspaces of the wheels settings and of the lug settings.

2.1.1 Wheels Settings Keyspace

Wheels 1, 2, 3, 4, 5 and 6 have 26, 25, 23, 21, 19 and 17 pins respectively, with a total of 131 pins. Each of the pins may be set to either effective or ineffective. Therefore, the size of the keyspace for the wheels pin settings is $2^{131}$.

In addition, the initial position of each wheel may be set by the operator, using the 6-letter external indicator. There are $26 \cdot 25 \cdot 23 \cdot 21 \cdot 19 \cdot 17 = 101,405,850 \approx 2^{27}$ distinct initial wheel positions settings. Usually, the operator modifies the initial wheels positions for each new message, while the pin and lug settings are changed daily. He also encrypts the 6 letters representing the initial positions of the wheels, and sends them encrypted, as part of the message preamble. There are various methods to encrypt the initial wheels positions, such as using the daily pin and lug settings and default "AAAAAA" initial wheels positions. In some rare cases, the initial positions of the wheels are sent in clear, or somehow they are known to the cryptanalyst. In those cases, it is necessary to take into account the initial wheels positions, as after recovering the pin and lug settings for one message, other messages on the same day and network may easily be decrypted, by just replacing the initial wheels positions. In our algorithm, as well as with the manual method developed by Morris (see Section 2.2), we can either use the initial wheel positions settings in the rare cases they are known, or simply assume default "AAAAAA" initial wheels positions, if they are not known. This is possible since any set of pin settings with initial wheels positions other than "AAAAAA", is logically equivalent to another set of pin settings in conjunction with the default "AAAAAA" initial wheels positions. To illustrate this, consider the following sample wheels pin settings, given the initial wheel positions "BBBBBB":
In this example, in wheel 1, pins #2,#3,#5,#6,#7,#10,#11,#15,#16,#17,#18,#23,#24 and #26 are in effective state, and all the other pins are in ineffective state. By rotating those wheels pin settings (for "BBBBBB") by one step to the right, using a cycling rotation, we can obtain wheels pin settings for the case of default "AAAAAA" initial wheel positions, which are cryptographically equivalent. Those equivalent pin settings are shown below:

| Wheel 1: 01101110011000111000011001 |
| Wheel 2: 001111000111010010100110 |
| Wheel 3: 0010111011110111101111 |
| Wheel 4: 01001110111101111011 |
| Wheel 5: 0101110111110111010 |
| Wheel 6: 0110111011110111 |

Listing 1: Example of Pin Settings with Initial Wheels Positions "BBBBBB"

For the attack presented here, as the initial wheels positions are either known, or assumed to be "AAAAAA", they do not affect the size of the wheels settings keyspace, which remains $2^{131}$.

**2.1.2 Lug Settings Keyspace**

Each one of the 27 bars in the cage has two movable lugs. Each lug can be set to be in front of any one of the 6 wheels, but the two lugs cannot be set to be in front of the same wheel. Also, in practical uses of the device, at least one of the lugs is always set. In the notation commonly used for lug settings, the lowest wheel number is specified first (e.g. 1-4, rather than 4-1), and if only one of the lugs is set, the number 0 is used instead of the first wheel (e.g. 0-1). An example of lug settings is shown in Table 1.

<table>
<thead>
<tr>
<th>Bar</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lug Settings</td>
<td>1-4</td>
<td>3-4</td>
<td>0-1</td>
<td>0-2</td>
<td>0-2</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td></td>
</tr>
<tr>
<td>Bar</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>Lug Settings</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td></td>
</tr>
<tr>
<td>Bar</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>Lug Settings</td>
<td>0-4</td>
<td>0-5</td>
<td>0-6</td>
<td>0-6</td>
<td>0-6</td>
<td>0-6</td>
<td>0-6</td>
<td>0-6</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Sample Lug Settings

There are 21 possible lug settings for each bar, as follows:

1. Only one of the two lugs is set to be against one of the 6 wheels, and the second is set to the neutral position (0). There are 6 possible such settings: 0-1, 0-2, 0-3, 0-4, 0-5 and 0-6.
2. Both lugs are set. This case is known as **lug overlap**. There are $\binom{6.5}{2} = 15$ possible lug settings with overlap. In the example shown in Table 1, bars #1 and #2 have lug settings with overlap, 1-4 and 3-4 respectively.
In theory, there should be $21^{27}$ possibilities to set up the bars lugs, or approximately $2^{118}$. From the cryptographic perspective, however, many of those settings are equivalent. In the encryption process, each one of the 27 bars independently contributes to the total displacement value applied by the Beaufort encoder to the current input letter. In the example shown in Table 1, bar #1 has lugs set to wheels 1-4, and bar #3 has only one lug set to wheel 1. Those settings are cryptographically equivalent to the settings shown in Table 2, where bar #1 has only one lug set to wheel 1, and bar #3 has lugs set to wheels 1-4.

<table>
<thead>
<tr>
<th>Bar</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lug Settings</td>
<td>0-1</td>
<td>3-4</td>
<td>1-4</td>
<td>0-2</td>
<td>0-2</td>
<td>0-2</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bar</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lug Settings</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
<td>0-3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bar</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lug Settings</td>
<td>0-4</td>
<td>0-5</td>
<td>0-6</td>
<td>0-6</td>
<td>0-6</td>
<td>0-6</td>
<td>0-6</td>
<td>0-6</td>
<td>0-6</td>
</tr>
</tbody>
</table>

Table 2: Cryptographically Equivalent Lug Settings

What actually matters is the number of bars with each one of the 21 distinct types of lug settings. We can, therefore, represent any lug settings for the device, by keeping a count of the bars with each one of the 21 possible types of lug settings. This concise form also represents any other equivalent set of lug settings. This is illustrated in Table 3 which shows this non-redundant alternative representation of the same lug settings as in Table 1 or Table 2. We shall use this concise and non-redundant representation throughout this paper, for our analysis and in our algorithms.

<table>
<thead>
<tr>
<th>Lug Settings Type</th>
<th>0-1</th>
<th>0-2</th>
<th>0-3</th>
<th>0-4</th>
<th>0-5</th>
<th>0-6</th>
<th>1-4</th>
<th>3-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Bars</td>
<td>1</td>
<td>3</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3: Non-redundant Representation of the Lug Settings from the Prior Example

After discarding the redundant settings, the cryptographically relevant size of the keyspace for the lug settings can now be calculated as follows: We need to distribute $k = 27$ indistinguishable elements (the bars) into $n = 21$ distinguishable buckets (the 21 distinct possible lug settings per each bar). Hence, according to the "bars and stars" formula $C(n + k - 1, k)$, there are $C(21 + 27 - 1, 27) = \frac{47!}{20!27!}$ possibilities. This is approximately $2^{43}$.

2.1.3 Combined Keyspace

The full, cryptographically relevant, keyspace of the Hagelin M-209 is therefore the combined keyspace for the wheels pin settings and for the lug settings, i.e. approximately $2^{131} \cdot 2^{43} = 2^{174}$.

2.2 Known-Plaintext Attacks

In general, known-plaintext attacks (or partially known-plaintext attacks) are often possible if a message contains stereotyped beginnings or endings, or when parts of the message (also known as "cribs") may be guessed. By determining the settings for a certain message using a known-plaintext attack, it is much easier to cryptanalyze other messages sent on the same day. Historically, messages sent on the same day and in the same net shared the same internal key settings (pin and lug settings), while only the external settings (initial wheels positions) of the Hagelin M-209 were different for each message.

Since the Hagelin M-209 encryption process is additive in nature, messages sent with the same settings can often be solved in-depth with the "Mots Probables" (probable words) method [1]. This is possible since the difference between corresponding plaintext letters at the same position in the two messages, is equal to the difference between the corresponding ciphertext letters. Therefore, if the analyst is able to guess a word in the first message, he can reproduce the letters of the second message at the corresponding positions by using a simple addition modulo 26 operation. As soon as
he has recovered enough plaintext with the probable words method, he can use a known-plaintext attack to recover the full key settings and decrypt the rest of the message as well as other messages.

In [5], Robert Morris describes such a known-plaintext attack for the recovery of the key settings. Since both the plaintext and the ciphertext are known, the displacement sequence (modulo 26) can be computed for all positions in the ciphertext. This attack is manual and is based on the iterative analysis and refinement of this displacement sequence. The method is complex and while some of the steps may be computerized, it relies heavily on the analyst’s judgment, and it requires a great deal of practice and trial-and-error. We illustrate here the main concept of the Morris attack, i.e. the displacement histograms. To do so, we use the following sample message (with 75 letters) and its corresponding ciphertext:

```
FROM Z GENERAL ZALEXANDERZTO
Z GENERAL ZPATTONZOPERATION
ZHUSSKYZSTOPZTHEZAMERICANZ
```

Listing 3: Sample Message Plaintext

```
Z Z Z L Z M T B F D I Y P E M L S B U J W S S D
S I J H M Q P L F N A V T L P G A J Z Y W Q M A A
```

Listing 4: Sample Message Ciphertext

At first, we compute the displacement sequence according to the following formula:

\[
\text{Displacement}[i] = (\text{CiphertextLetter}[i] + \text{PlaintextLetter}[i] - Z) \mod 26
\]

where \(Z = 25\). We obtain the following displacement sequence:

```
05, 17, 14, 05, 11, 06, 17, 07, 06, 23, 04, 20, 24, 16, 16, 17, 09, 19, 15, 24, 14, 14, 18, 12,
18, 15, 06, 19, 14, 01, 12, 08, 00, 15, 14, 20, 06, 01, 17, 13, 11, 01, 00, 18, 07, 19, 13, 13,
15, 13, 18, 16, 04, 00, 23, 15, 15, 04, 25, 02, 16, 21, 13, 19, 20, 06, 01, 22, 04, 16, 05, 19,
13, 14, 00
```

Listing 5: Displacement Sequence

Then, for each wheel, and for each one of its pins, we gather the relevant displacement values, and we compute the average displacement for each pin. At this stage we discard displacement values of 0 and 1, which may be ambiguous because of the modulo 26 operation. We assume that pin #1 of each wheel is at the active position when encrypting the first letter (position 0) of the message. We start with wheel 1, which has 26 pins. In our example, pin #1 of wheel 1 is active at position 0, then again 26 steps later (position 26), as well as in position 26 + 26 = 52. The displacement for position 0 is 5, it is 6 for position 26 and 4 for position 52. Therefore the average displacement for pin #1 is \(\frac{5+6+4}{3} = 5\). Similarly, pin #2 of wheel 1 is active at position 1 (displacement = 17) and at position 1 + 26 = 27 (displacement = 19). It is also active at position 1 + 52 = 53, but the displacement is 0. Therefore, it is ambiguous because of the modulo 26 operation, and we ignore it at this stage. Hence, the average displacement for pin #2 of wheel 1 is \(\frac{17+19}{2} = 18\). In the following listing we calculated the result for all pins of wheel 1:
We repeat the process for all the remaining wheels. We can now draw a histogram of the average
displacements, for each one of the 6 wheels. For example, for wheel 1 we have one pin with average
displacement 5 (pin #1), one pin with average displacement 6 (#6), one pin with average displacement
7 (#17), two pins with average displacement 8 (#8 and #11), and so on. The average displacement
histograms for all the wheels are shown below:

<table>
<thead>
<tr>
<th>pin 01</th>
<th>05,  06,  04,</th>
<th>Average:  05</th>
</tr>
</thead>
<tbody>
<tr>
<td>pin 02</td>
<td>17,  19,  00,</td>
<td>Average:  18</td>
</tr>
<tr>
<td>pin 03</td>
<td>14,  14,  23,</td>
<td>Average:  17</td>
</tr>
<tr>
<td>pin 04</td>
<td>05,  01,  15,</td>
<td>Average:  10</td>
</tr>
<tr>
<td>pin 05</td>
<td>11,  12,  15,</td>
<td>Average:  13</td>
</tr>
<tr>
<td>pin 06</td>
<td>06,  08,  04,</td>
<td>Average:  06</td>
</tr>
<tr>
<td>pin 07</td>
<td>17,  00,  25,</td>
<td>Average:  21</td>
</tr>
<tr>
<td>pin 08</td>
<td>07,  15,  02,</td>
<td>Average:  08</td>
</tr>
<tr>
<td>pin 09</td>
<td>06,  14,  16,</td>
<td>Average:  12</td>
</tr>
<tr>
<td>pin 10</td>
<td>23,  20,  21,</td>
<td>Average:  21</td>
</tr>
<tr>
<td>pin 11</td>
<td>04,  06,  13,</td>
<td>Average:  08</td>
</tr>
<tr>
<td>pin 12</td>
<td>20,  01,  19,</td>
<td>Average:  20</td>
</tr>
<tr>
<td>pin 13</td>
<td>24,  17,  20,</td>
<td>Average:  20</td>
</tr>
<tr>
<td>pin 14</td>
<td>16,  13,  06,</td>
<td>Average:  12</td>
</tr>
<tr>
<td>pin 15</td>
<td>16,  11,  01,</td>
<td>Average:  14</td>
</tr>
<tr>
<td>pin 16</td>
<td>17,  01,  22,</td>
<td>Average:  20</td>
</tr>
<tr>
<td>pin 17</td>
<td>09,  00,  04,</td>
<td>Average:  07</td>
</tr>
<tr>
<td>pin 18</td>
<td>19,  18,  16,</td>
<td>Average:  18</td>
</tr>
<tr>
<td>pin 19</td>
<td>15,  07,  05,</td>
<td>Average:  09</td>
</tr>
<tr>
<td>pin 20</td>
<td>24,  19,  19,</td>
<td>Average:  21</td>
</tr>
<tr>
<td>pin 21</td>
<td>14,  13,  13,</td>
<td>Average:  13</td>
</tr>
<tr>
<td>pin 22</td>
<td>14,  13,  14,</td>
<td>Average:  14</td>
</tr>
<tr>
<td>pin 23</td>
<td>18,  15,  00,</td>
<td>Average:  17</td>
</tr>
<tr>
<td>pin 24</td>
<td>12,  13,</td>
<td>Average:  13</td>
</tr>
<tr>
<td>pin 25</td>
<td>18,  18,</td>
<td>Average:  18</td>
</tr>
<tr>
<td>pin 26</td>
<td>15,  16,</td>
<td>Average:  16</td>
</tr>
</tbody>
</table>

Listing 6: Average Displacement for the 26 Pins of Wheel 1

<table>
<thead>
<tr>
<th>Wheel 1</th>
<th>Wheel 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000011111111111222222222</td>
<td>000000000001111111111112222222222</td>
</tr>
<tr>
<td>01234567890123456789012345</td>
<td>01234567890123456789012345</td>
</tr>
</tbody>
</table>

Listing 7: Average Displacement Histograms for Wheels 1 and 2
Next, we look for the histogram with the best "bimodal distribution", i.e. with displacement values around two main distinct "peak" values. The best candidates are wheels #2 and #3. We select wheel 3, as we can see that there is a peak and a cluster of values around the displacement value of 12, and another peak (and cluster) around the displacement value of 16. Those two peaks and clusters most probably represent the set of ineffective pins and the set of effective pins, respectively. From the analysis of this histogram, we may assume that pins of wheel 3 with an average displacement of 12 or below are most probably pins in ineffective state, and that pins with an average displacement of 16 and above are most probably in effective state. No conclusion may be yet drawn for the other pins. With those assumptions, we may also reach conclusions in regard to some of the ambiguous values (0 or 1) in the displacement sequence. Using this knowledge about the pins of wheel 3, we further refine the displacement sequence and the displacement histograms for other wheels, using interpolation techniques described in detail in Morris’s paper [5]. We look for the next wheel for which the refined displacement histogram has the best bimodal distribution and repeat the process. We iteratively process all the wheels, until all pin settings have been recovered. When all pin settings have been recovered, the lug settings can also be recovered. One of the main challenges with this method, especially with messages shorter than 100 letters, is that after processing the first 2 or 3 wheels, very often none of the remaining wheels will show any discernible bimodal distribution pattern, even after applying the interpolations techniques proposed by Morris.

According to the declassified TICOM I-175 report, the German cryptanalytic services in World War II were probably also able to reconstruct keys settings from known plaintext, although their method is not described in the report [6].

Note that a known-plaintext attack is possible even if the known or recovered plaintext is incomplete, or if it is not contiguous. Morris’s manual method requires at least 75-100 arbitrary characters of known plaintext to determine the full key settings. In contrast, our automated approach, which we present in the next section, in most cases requires only 50 arbitrary characters of known plaintext.
3 Automated Known-Plaintext Attack

In this section we present our known-plaintext attack which is also applicable to the case when only a part of the plaintext is known or can be guessed and/or the known plaintext is not contiguous. Only the number of ciphertext letters for which we know or can guess the corresponding original plaintext letter is relevant for our attack.

Our attack is a hillclimbing algorithm with three nested loops, which repeatedly transform the key in order to improve the corresponding fitness score. For the fitness score, we defined the Aggregate Displacement Error (ADE) function, as described in Section 3.3. For the key transformation, we defined two types of transformations on the pin settings (see Section 3.1), and two types of transformations on the lug settings (see Section 3.2). In the following we give an overview of the steps of our algorithm and we present the pseudo code of the complete algorithm in Appendix A.1:

1. Set the external initial wheels settings to the default value "AAAAAA", or to the original external settings if those are known.
2. Generate initial random pin and lug settings.
3. Repeat the following three loops (3a, 3b, and 3c) until the ADE no longer improves (i.e. no longer can be reduced):
   (a) Repeatedly perform the set of all the pin settings transformations as long as the ADE can be improved (reduced). For each transformation perform the following steps:
      i. Compute the ADE score.
      ii. If the ADE score with the new pin settings is lower than the score with the pin settings before the transformation, keep the new pin settings. Otherwise, discard the new pin settings.
   (b) Repeatedly perform a subset of the lug settings transformations which do not involve lugs overlap as long as the ADE can be improved (reduced). For each transformation perform the following steps:
      i. Compute the ADE score.
      ii. If the score with the new lug settings is lower than the score with the lug settings before the transformation, keep the new lug settings. Otherwise, discard the new lug settings.
   (c) Same as in (b), but for the complete set of lug settings transformations.
4. If an ADE score of 0 is reached, the correct settings have been found and the algorithm terminates. If neither loop 3a, 3b or 3c resulted in an improvement (reduction) of the ADE, repeat from Step 2, i.e. restart with new random settings.

In Appendix A.2 and Appendix A.3, we present the pseudo code of the two functions, "CheckPinsTransformations" and "CheckLugsTransformations", which implement the internal loops described in Step 3a, 3b, and 3c above.

3.1 Transformations on Pin Settings

In our automated known-plaintext attack we repeatedly perform transformations on the pin settings. The rationale behind each transformation is to make only a slight change in the pin settings starting from some initial settings. We never change the state of more than 1 or 2 pins in any single transformation. The full set of the pin settings transformations processed by hillclimbing includes:

1. "Toggle" Transformations: Toggle the state of one the pins of one of the wheels. This means that if the pin is currently effective, then set it to ineffective, and if it is currently ineffective, then set it to effective.
2. "Swap" Transformations: For a pair of pins, where the pins are not in the same state, toggle the state of both pins. For this transformation we consider any pair of pins, i.e. either two pins in the same wheel or in different wheels.
While any "Swap" transformation is logically equivalent to two "Toggle" transformations, the "Swap" transformations are needed to test the effect on the ADE of the two changes applied simultaneously, rather than sequentially computing the ADE after each one. This is necessary in case each one of the two equivalent Toggle transformations results in degrading the ADE, in which case such a transformation shall be discarded by the algorithm, while only the application of both changes at the same time (the "Swap" transformation) actually improves the ADE.

Since there is a total of 131 pins on the six wheels we have 131 possible "Toggle" transformations, and no more than \( \frac{131 \times 130}{2} = 8515 \) possible "Swap" transformations. Typically only half of the Swap transformations are relevant, as we consider only pairs of pins which are not in the same state.

### 3.2 Transformations on Lug Settings

In our automated know-plaintext attack we also iterate through a set of lug settings transformations. Each transformation affects a single bar. We use the concise and non-redundant representation presented in Section 2.1, basically an array counting the number of bars being set to each one of the possible 21 lug settings types. Those lug settings types include 6 types without overlap and 15 types with overlap. In each transformation, the count for one of the types is decreased, and the count for another type is increased. The full set of our lug settings transformations consists of the following categories:

1. Swap between two types of lug settings without overlap: For each one of the 6 non-overlap types, and each one of the remaining 5, reduce the count for the first type, and increase the count for the second type. If the first type already had a count of 0, we skip this case. Going back to our example in Table 3, an example of such a transformation would be to reduce the count for type 0-1 from 1 to 0, and to increase the count for type 0-2 from 4 to 5.

2. Swap between any two types of lug settings, with or without overlap: For each one of the 21 types, and each one of the remaining 20, reduce the count for the first type, and increase the count for the second type. If the first type already had a count of 0, we skip this case. Going back to our example in Table 3, an example of such a transformation would be to reduce the count for type 0-1 from 1 to 0, and increase the count for type 1-2 from 0 to 1.

While the second set of swap transformations also contains the set of simpler swap transformations (i.e. the ones without overlap), the distinction is necessary, as we will prefer to check first the simpler transformations, as long as we can improve the ADE, before checking the more complex ones.

With the above rules there are no more than \( 6 \times 5 + 21 \times 20 = 450 \) possible lugs transformations for our algorithm, and usually much less, as we can’t reduce the count of a certain type if it was already 0.

### 3.3 The Aggregate Displacement Error Score

The choice of an effective fitness function or score is often critical to the success of any hillclimbing algorithm. At a minimum, it should be more or less monotonic, i.e. more accurate key settings should result in a better score. With the fully correct key settings, the score should reach its optimum. But this is not enough: in order to be effective and allow convergence, and especially when starting with random settings with a large number of errors, the fitness function should also be able to detect and reflect subtle improvements or degradations in the key.

We first implemented our hillclimbing algorithm using a simple fitness function, by counting the number of plaintext letters correctly reproduced after decryption using a putative key, or more specifically, by counting incorrectly reproduced letters. With this simplistic function, the hillclimbing algorithm was able to recover the full key settings for messages with at least 300-400 known-plaintext letters. With 200 or less known-plaintext letters, our algorithm using this scoring method consistently failed. A major drawback of this scoring method is that in order to reproduce the correct letter at a given position, it is necessary to recover all the key settings elements which contribute to the displacement at this position, i.e. the active pin of each one of the 6 wheels, and most of the lug settings. Therefore a correction of only one of those elements is unlikely to be enough to correctly reproduce the plaintext letter, unless all the other elements were already correct.
We, therefore, designed an alternative scoring function, the Aggregate Displacement Error (ADE) to take into account the individual contribution of each one of the key elements affecting the decryption of each one of the letters in the message. Since our method is a known-plaintext attack, both the plaintext and the ciphertext are known, the expected displacement at each step may also be computed with the following equation:

\[ \text{ExpectedDisplacement}[i] = (\text{CiphertextLetter}[i] + \text{PlaintextLetter}[i] - Z) \mod 26 \] (3)

We now define the Displacement Error, for a putative key and for a certain position/letter in the ciphertext, as the absolute difference between the actual displacement, resulting from the current lugs and pin settings, and the expected displacement, as defined in the above equation. For a formal description, we provide the pseudo code for the displacement error computation in Appendix A.4. The ADE is simply the sum of the displacement error values for all the letters of the ciphertext for which the corresponding plaintext letter is known.

Special care is needed for expected displacement values 0 and 1, which can be ambiguous; the other values being non-ambiguous. Because of the modulo 26 operation, there is no way to differentiate between original displacement values of 0 and 26, as well as between displacement values of 1 and 27. In case of ambiguity, we use an optimistic approach and assume the lowest error from the two possible alternatives. For example, if the expected displacement at a given position \( i \) is 0, the original displacement could have been either 0 or 26. Same reasoning applies for 1 and 27. When we compare the actual displacement resulting from a putative key, which may have any value from 0 to 27, to the expected displacement value, which may have values only from 0 to 25, we assume the lowest (i.e. optimistic) possible displacement error value. For example, if \( \text{ExpectedDisplacement}[i] = 0 \) and \( \text{ActualDisplacement}[i] = 23 \), then the error is assumed to be \( 26 - 23 = 3 \), rather than an error of \( 23 - 0 = 23 \).

To verify the effectiveness of the ADE we compared it with the simple scoring method of counting incorrectly reproduced letters, using sample messages with 100 known-plaintext letters, encrypted with random key settings. For each case, we modify the correct key settings, by introducing a series of 100 consecutive settings errors, each time in either one of the pin settings or one of the lug settings. At each step, we decrypt the ciphertext with the key settings (with 0 to 100 errors), and compute the ADE and the number of incorrect letters after decryption. The average results for 1000 sample messages are shown in Figure 3. The count of incorrectly reproduced letters has been normalized by a factor of 5.94 to allow for convenient comparison. As expected, both the ADE and the incorrect letters count are 0 with the correct key, and they increase when additional settings errors are introduced, as long as the number of errors is not too high. However, the ADE graph is much smoother, and continues to be almost always monotonic, even with a high number of settings errors. This allows for the detection of subtle improvement or degradation after modifying key settings. Hence, the ADE is a more effective scoring method for our hillclimbing attack, as also shown by the overall performance of the full algorithm – only 50 known-plaintext letters vs. at least 300 required for full key recovery with the simplistic score.

4 Evaluation

In this section we show the soundness of our algorithm. At first we present its performance, which we analyzed with simulations. After that we analyze the work factor of our algorithm. Finally, we solved several publicly available challenges and we show the results.

4.1 Performance

We tested this algorithm with simulated cases of messages of various lengths, and encrypted with random keys settings. In the M-209 Technical Manual, a list of mandatory rules is provided, to avoid settings which may weaken the cryptographic security of the device [7]. One of the rules is that the number of lugs overlaps in bars should be at least 2 and 12 at most. Therefore, we tested the algorithm using lug settings with 2, 6 and 12 lugs overlaps. The number of known-plaintext letters was between 50 to 100. The results are shown in Table 4 and Figure 4. Each result is based on a sample of 100 simulation runs. A run is considered successful only if the full correct settings were
recovered within an arbitrary time-out limit of 10 hours. The tests were run on a 3.4 GHz Core i7 PC.

Key settings for messages with 65 letters or more can be easily recovered, often in less than a minute. For example, the 75 letter sample message given by Morris was solved in 5 seconds. For shorter messages, we can see that the number of lugs overlaps affects the success rate. It also affects the speed of convergence of the hillclimbing algorithm, as described in more detail in Section 4.2. Therefore, the lug overlap feature of the M-209 adds somehow to the cryptographic security of the M-209, at least for this type of attack. In comparison, Morris’s method requires at least 75-100 known plaintext characters, in order to fully recover the key settings.

Note that when hillclimbing fails to recover the full key settings, it is nevertheless often possible to recover most of the correct settings. If such a partial solution can be achieved, for example when only a small part of the message plaintext is known, this partial solution may help in revealing additional probable words or sentences in other parts of the message, even though they may show up with some errors after decryption. By using this additional known plaintext and running again the algorithm, the full key settings can then be recovered.

### 4.2 Analysis of Work Factor

As described in Section 3, each main cycle of the hillclimbing algorithm starts with a random key, i.e. with random pin and lug settings. It continuously performs loops of transformations of various types on the pin and lug settings. After each transformation, it decrypts the ciphertext with the resulting key, and computes the ADE. If the ADE has improved, the resulting new key is kept. The number of possible transformations of each type is summarized in Table 5 for pins and in Table 6 for lugs.

Hillclimbing always checks first the pin settings transformations, repeating cycles of testing all such transformations as long as the ADE improves. It then checks all possible lug settings transformations
Figure 4: Probability of Full Key Recovery vs. Length of Known Plaintext

<table>
<thead>
<tr>
<th>Transformation Type</th>
<th>Number of Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toggle Transformations</td>
<td>131</td>
</tr>
<tr>
<td>Swap Transformations</td>
<td>((131 \cdot 130)/2 = 8515)</td>
</tr>
<tr>
<td>Total:</td>
<td>On average about 4,300 since 50% of the possible swap transformations are irrelevant (the two pins involved must have different states).</td>
</tr>
</tbody>
</table>

Table 5: Number of Possible Pin Settings Transformations

not involving lugs overlaps. Last, it checks all possible lug settings transformations. The actual number of cycles for each type, as well as the number of overall start/restart cycles, varies according to the length of the message and the complexity of the key (number of lugs overlaps). However, in general about 80%-90% of the transformations actually tested are pins "Swap" transformations.

In Table 7 we present the total number of decryptions(transformations) required for recovering the full key settings, for several message lengths, using settings with 6 lugs overlaps. The average, best-case and worst-case results are shown. The data is based on batches of 100 successful simulation runs per each length. A run is considered successful if the full key was recovered within 10 hours (on a 3.4 GHz PC). Note that even in cases of unsuccessful runs, the majority of key settings can often be recovered.

The worst-case message with 50 letters and 6 lugs overlaps requires about 3,000,000,000 or \(2^{32}\) transformations, decryptions and ADE computations. This can easily be computed with a single high-end PC – in contrast to a brute-force search over the complete keyspace with a size of \(2^{174}\). On a 3.4 GHz PC and without multithreading, for messages with 50 known-plaintext letters, about

<table>
<thead>
<tr>
<th>Transformation Type</th>
<th>Number of Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swaps of types without lug overlap</td>
<td>(6 \cdot 5 = 30)</td>
</tr>
<tr>
<td>Swaps of any type</td>
<td>(21 \cdot 20 = 420)</td>
</tr>
<tr>
<td>Total:</td>
<td>On average about 120 transformations (about 30% of the total possible 450), since one of the two types involved must be non-zero.</td>
</tr>
</tbody>
</table>

Table 6: Number of Possible Lug Settings Transformations
<table>
<thead>
<tr>
<th>Message Length</th>
<th>Average Number of Decryptions per Successful Run</th>
<th>Minimum (Best Case)</th>
<th>Maximum (Worst Case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>384 000 000</td>
<td>21 000 000</td>
<td>2 925 000 000</td>
</tr>
<tr>
<td>75</td>
<td>1 783 000</td>
<td>268 000</td>
<td>5 962 000</td>
</tr>
<tr>
<td>100</td>
<td>311 000</td>
<td>160 000</td>
<td>597 000</td>
</tr>
</tbody>
</table>

Table 7: Number of Transformations Required for Full Key Recovery

300 000 transformations and ADE computations per second can be processed. For this worst-case message, this amounts to about 3 hours. The average time to solve a message with 50 known-plaintext letters (6 lugs overlaps) is approximately 20 minutes. With 100 letters, the process takes 1-2 seconds on average. In Figure 5, we present the work factor of our algorithm for additional scenarios, including 2, 6 and 12 lugs overlaps. We show the number of the required transformations on a log2 scale.

![Figure 5: Number of Transformations Required for Full Key Recovery](image)

The work factor of our method cannot be directly compared with Morris’s method for recovering key settings, which is a manual method. Some elements of Morris’s method, such as the computation of the displacement histograms, may be computerized. However, the analyst’s judgment is required at each step, especially on short messages, to decide on which wheel to focus, which pins to mark as known or ambiguous, and when to backtrack and start again. In contrast, our method is fully automated.

### 4.3 Challenges

We also applied the method to several challenges, such as those developed by Jean-Francois Bouchaudy. Bouchaudy is a French amateur codebreaker who has studied the Hagelin M-209 device, its historical uses and cryptanalytical methods. On his website, he published a series of challenges with increasing difficulty, and requiring a wide range of techniques for their solution [2]. In the bonus section, there is a known-plaintext challenge (#12) with 50 letters, which was easily solved by our method. There is also a challenge with 40 letters (#14), which we also solved. This took about 10 hours on a 4-core 3.4 GHz PC and using 8 threads. The author of the challenges has requested not to publish the solutions, but more details can be found on the challenge website [2], including a mention of our results.

We also applied the technique to the 4th and most difficult of the challenges that Morris himself published [5]. This message has 60 letters, but only 50 of the corresponding plaintext letters are given. Our program was able to find the settings in about 20 minutes. The lug settings have 6 overlaps. With those settings, the second ciphertext message from this challenge could be deciphered: “CONGRATULATIONSZGOODZBUDDYZZZZ”.
5 Conclusion

The use of a highly specialized and discriminative fitness function is often the key to the successful design of a hillclimbing method. In our case, the Aggregate Displacement Error score proved to be highly effective vs. an alternative simplistic scoring method. With the hillclimbing method described in this work, the process of recovering key settings based on short known (or guessed) plaintext, can be fully automated, and keys can be fully recovered with known plaintext as short as 50 letters, and often with less based on partial solutions.

We will release the source code of our attack as a component in the open source tool CrypTool 2.0 [3], thereby making this attack publicly available. A further direction for study would be to apply similar techniques for a known-plaintext attack on other Hagelin devices, such as the C-52 or the CD-57.

About the Authors

George Lasry is a computer scientist and consultant in the high-tech industry in Israel. Prior to this, he worked for many years in the development of communications systems, and also managed R&D and sales organizations. His primary interest in cryptographic research is the application of specialized optimization techniques for the computerized cryptanalysis of classical ciphers and cipher machines. Using such a technique, he solved in November 2013 [4] the Double Transposition (Doppelwürfel) cipher challenge which was published by Klaus Schmeh in 2007.

Nils Kopal is a Ph.D. student with the research group “Applied Information Security” (AIS) at the University of Kassel. He studied applied computer science at the University of Duisburg-Essen in Duisburg with focus on distributed and secure systems. He finished his studies with a master’s degree. With the AIS group he is the responsible coordinator for the development of the open source e-learning tool CrypTool 2.0 [3] which illustrates classical and modern cryptographic and cryptanalytic concepts.

Professor Arno Wacker is the head of the research group “Applied Information Security” (AIS) with the University of Kassel in Germany. He is also the lead of the open source project CrypTool 2.0 [3] and member of the steering group of MysteryTwister C3. His main research interests are modern security protocols for decentralized distributed systems, e.g. for volunteer computing scenarios, and computerized cryptanalysis of classical ciphers. Additionally, he teaches classical and modern cryptology in classes at the University of Kassel and at special workshops for pupils at local schools.

A Appendix

A.1 Pseudo Code of the Hillclimbing Algorithm

```
Input:
C // The ciphertext
P // The known plaintext letters
IWP // Initial wheels positions (or null if unknown)

Output:
BestKey // Wheels pin settings and lugs settings

AutomatedKnownPlaintextAttack(C, P, IWP) {
    // Local variables
    D // Expected Displacement Sequence
    TR_P_T // Generic Toggle transformations on pins (See Section 3.1)
    TR_P_S // Generic Swap transformations on pins (See Section 3.1)
    TR_L_N // Generic transformations on lugs – no overlap (see Section 3.2)
    TR_L // Generic transformations on lugs (see Section 3.2)
    BestKey = null // Best key globally
    BestKeyADE = MAX // Best ADE globally
```
// Compute the expected displacement sequence D.
// We add 26 before modulo 26 since we only want positive numbers (Z = 25)
for each position i in P {
    D[i] = (C[i] + P[i] − 'Z' + 26) modulo 26
}

// Prepare the sets of the generic transformations
Pre−compute TR_{P,T}, TR_{P,S}, TR_{L,N}, and TR_{L}

// Set the initial wheels positions if known, or to default "AAAAAA"
if (IWP == null) {
    SetInitialWheelsPositions("AAAAAA")
} else {
    SetInitialWheelsPositions(IWP)
}

StartTime = now() // Timeout limit
TimeLimit = 10 Hours

// Main loop – continue until either we have found the correct key (ADE = 0),
// or when time limit is exceeded
while ((BestKeyADE != 0) and ((now()−StartTime) < TimeLimit)) {
    // Start (or restart) with random pin and lug settings
    CurrentKey = GenerateRandomKey()
    CurrentKeyADE = DecryptAndComputeADE (C, CurrentKey, D)

    // Continue to sequentially test all types of transformations
    // as long as an improvement in ADE can be achieved ,
    // using any one of the pins or lug settings transformations
    do {
        AnyImprovement = false // true if any improvement was achieved

        // Repeatedly check all pin settings transformations as long as
        // an improvement in ADE can be achieved.
        (AnyImprovement, CurrentKey, CurrentKeyADE) =
            CheckPinsTransformations(C, D, TR_{P,T}+TR_{P,S}, CurrentKey, CurrentKeyADE)

        // Repeatedly check lug transformations without overlaps
        // as long as an improvement in ADE can be achieved.
        (AnyImprovement, CurrentKey, CurrentKeyADE) =
            CheckLugsTransformations(C, D, TR_{L,N}, CurrentKey, CurrentKeyADE)

        // Repeatedly check all lug settings transformations as long as
        // an improvement in ADE can be achieved.
        (AnyImprovement, CurrentKey, CurrentKeyADE) =
            CheckLugsTransformations(C, D, TR_{L}, CurrentKey, CurrentKeyADE)

        // Terminate this cycle if no more improvement could be achieved
        // using any of the pins or lug settings transformations
    } while (AnyImprovement)

    // Before restarting a new cycle, check whether the current key
    // is better than the best key so far. If yes, keep it.
    if (CurrentKeyADE < BestKeyADE) {
        BestKey.Pins = CurrentKey.Pins
        BestKey.Lugs = CurrentKey.Lugs
        BestKeyADE = CurrentKeyADE
    }
}
A.2 Pseudo Code for CheckPinsTransformations

Input:
C // The ciphertext
D // Expected Displacement Sequence
TR_P_SET // Set of transformations on pins
CurrentKey // Current key of the current run
CurrentKeyADE // ADE of the current key

Output:
AnyImprovement // Indicates whether any improvement was achieved
CurrentKey // Current key of the current run
CurrentKeyADE // ADE of the current key

CheckPinsTransformations(C, D, TR_P_SET, CurrentKey, CurrentKeyADE)
{
    // repeatedly check the set of transformations
    // on pins, as long as the ADE can be improved
    do
        Improved = false
        for each valid T in (TR_P_SET) {
            PutativeKey.Pins = ApplyPinsTransformation(T, CurrentKey.Pins)
            PutativeKeyADE = DecryptAndComputeADE (C, PutativeKey, D)
            if (PutativeKeyADE < CurrentKeyADE) {
                CurrentKey.Pins = PutativeKey.Pins
                CurrentKeyADE = PutativeKeyADE
                AnyImprovement = Improved = true
            }
        }
    } while (Improved)
    return (AnyImprovement, CurrentKey, CurrentKeyADE)
}

A.3 Pseudo Code for CheckLugsTransformations

Input:
C // The ciphertext
TR_L_SET // Set of transformations on lugs
CurrentKey // Current key of the current run
CurrentKeyADE // ADE of the current key

Output:
AnyImprovement // Indicates whether any improvement was achieved
CurrentKey // Current key of the current run
CurrentKeyADE // ADE of the current key

CheckLugsTransformations(C, D, TR_L_SET, CurrentKey, CurrentKeyADE)
{
    // repeatedly check the set of transformations
    // on lugs, as long as the ADE can be improved
    do
        Improved = false
        for each valid T in (TR_L_SET) {
            PutativeKey.Lugs = ApplyLugsTransformation(T, CurrentKey.Lugs)

            if (PutativeKey.Lugs should be ApplyLugsTransformation(T, CurrentKey.Lugs) in the code.
A.4 Pseudo Code of the Displacement Error Algorithm

```plaintext
A.4 Pseudo Code of the Displacement Error Algorithm

| Input: |
| expectedDisplacement \quad // The expected displacement |
| \quad // calculated using crib and ciphertext |
| \quad // It is equal to the original displacement, modulo 26. |
| actualDisplacement \quad // The actual displacement calculated |
| \quad // using decrypted ciphertext and ciphertext |

| Output: |
| DisplacementError \quad // The error of expectedDisplacement and |
| \quad // actualDisplacement |

| DisplacementError(expectedDisplacement, actualDisplacement) |
| { |
| \quad if (expectedDisplacement < 2) |
| \quad { |
| \quad \quad // Optimistic approach – we keep the lowest error value, either by: |
| \quad \quad // (a) Assuming the original displacement was >= 26 |
| \quad \quad // (b) Assuming the original displacement was < 2 |
| \quad \quad return \quad min(abs((expectedDisplacement + 26) - actualDisplacement), |
| \quad \quad \quad abs(expectedDisplacement - actualDisplacement)) |
| \quad } |
| \quad else |
| \quad { |
| \quad \quad return abs(expectedDisplacement - actualDisplacement) |
| \quad } |
| }

References


18