The drunken bishop: An analysis of the OpenSSH fingerprint visualization algorithm

Dirk Loss[*], Tobias Limmer[†], Alexander von Gernler[‡]

September 20, 2009

Abstract

OpenSSH 5.1 introduced an ASCII-based visualization method for the remote servers' public key fingerprints. We explain the algorithm used to visualize the fingerprints and present some initial findings about its properties. Based on a Markov model and some brute-force attacks we were only able to produce some basic results. But we hope that our analysis will spur further research on this topic, so that eventually it will be found out whether the (heuristically designed) algorithm is secure enough for this purpose.

[*]mail@dirk-loss.de
[†]limmer@informatik.uni-erlangen.de
[‡]grunk@openbsd.org

Contents

1 Introduction

Bishop Peter finds himself in the middle of an ambient atrium. There are walls on all four sides and apparently there is no exit. The floor is paved with square tiles, strictly alternating between black and white. His head heavily aching—probably from too much wine he had before—he starts wandering around randomly. Well, to be exact, he only makes diagonal steps—just like a bishop on a chess board. When he hits a wall, he moves to the side, which takes him from the black tiles to the white tiles (or vice versa). And after each move, he places a coin on the floor, to remember that he has been there before. After 64 steps, just when no coins are left, Peter suddenly wakes up. What a strange dream!

When a Secure Shell (SSH) client connects to a server for the first time, the fingerprint of the server's public key is presented to the user. The user is prompted to verify the fingerprint and thereby attest the authenticity of the server:

```
$ ssh anoncvs.de.openbsd.org
The authenticity of host 'anoncvs.de.openbsd.org (131.188.40.91)'
can't be eastablished.
RSA key fingerprint is fc:94:b0:c1:e5:b0:98:7c:58:43:99:76:97:ee:9f:b7
Are you sure you want to continue connecting (yes/no)?
```

Even if the user has the correct version of the hash value available for comparison, comparing hexadecimal hash values is cumbersome and error-prone. So this mechanism against man-in-the-middle attacks[1] might not work so well in practice.

In order to ease the verification process, OpenSSH 5.1 introduced a new feature called "fingerprint visualization". If activated in the configuration file[1], each time you connect to an SSH server, a little picture is drawn on the screen (see figure 1).

The picture is algorithmically derived from the key fingerprint, so that different fingerprints generate different pictures. The idea is that if you have connected to this server before – e.g. from other hosts – you will probably recall the picture and trust the server ("Yes, that's a server I know."). More importantly, if an attacker poses as your server, the visualization for his fingerprint will be different. You will notice ("No, I haven't seen this picture before.") and cautiously decide not to log on.

In this paper we explain the algorithm used to visualize the fingerprints and present some initial findings about its properties. The visualization algorithm was

[1] Put `VisualHostKey yes` in `/etc/ssh_config`.

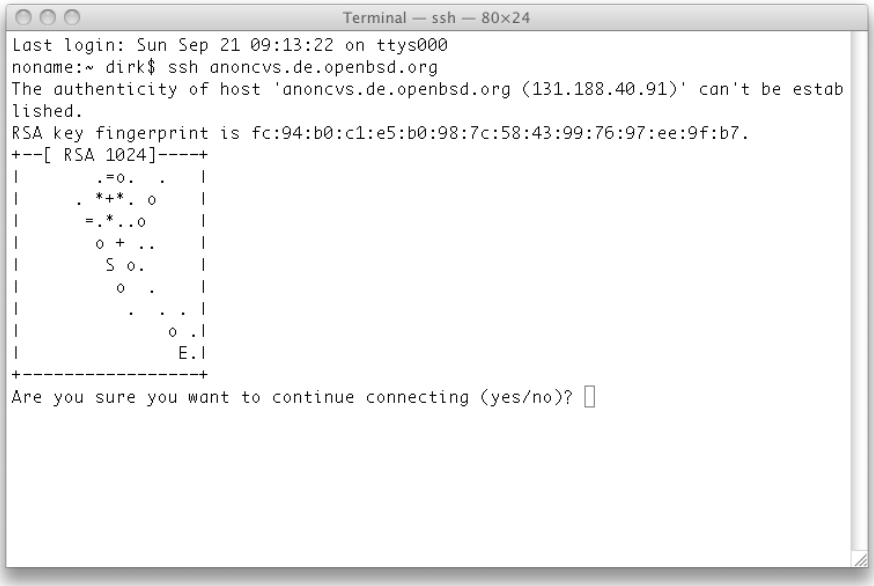

Figure 1: Fingerprint shown on first connection

devised and implemented by Alexander von Gernler during OpenBSD Hackathon 2008 [4] based on an idea by Perrig and Song [2] and the concept of "random art". The original C source code [3] is shown in the appendix of this paper.

Based on a Markov model and some brute-force attacks we were only able to produce some basic results. But we hope that our analysis will spur further research on this topic, so that eventually it will be found out whether the (heuristically designed) algorithm is secure enough for this purpose.

2 The algorithm

2.1 Field

As suggested by the introductory "drunken bishop" story the algorithm can be seen as a diagonal random walk on a bounded discrete plane. In the OpenSSH implementation (see the appendix for the original C source code) this *field* has a *width* of 17 characters and a *height* of 9 characters.[2] Figure 2 shows the starting position: an empty field with the start position in the middle. We enumerate the rows and columns, so that the field can be seen as a coordinate system. The origin is placed at the upper left corner.

```
             1111111
   01234567890123456
  +-----------------+x  (column)
 0|                 |
 1|                 |
 2|                 |
 3|                 |
 4|        S        |
 5|                 |
 6|                 |
 7|                 |
 8|                 |
  +-----------------+
  y
(row)
```

Figure 2: Empty field (start position marked)

There are $9 \cdot 17 = 153$ *positions* on the standard OpenSSH field. For our analysis each position is identified by its column x and row y. We use the notation $[x, y]$ to show the coordinates of a position. Alternatively a position can be given by a single number according to the following formula: position $= [x, y] = x + 17y$. For example, the start position `S` is at [8, 4] (position 76), as can be seen in figures 2 and 3.

If we take the positions as vertices and draw edges between those positions that can be reached in a single step, we get the graphs in figure 4 and figure 5. The graph is connected and contains cycles at the corner positions. It is also non-planar as can be seen from figure 5. The graphs were automatically generated by Graphviz (2D) and RTGraph3D (3D).

[2]Odd numbers are taken so that the marker for the start position can be placed exactly in the middle of the field. To get a square visualization on screen, the width of the field roughly

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84
85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101
102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118
119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135
136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152

Figure 3: Positions on the field given as numbers. Start position is marked

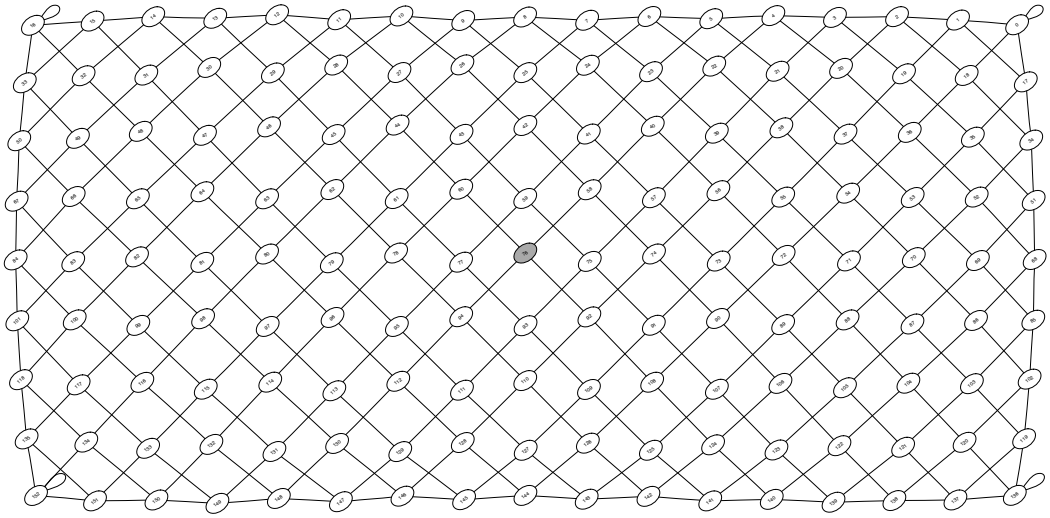

Figure 4: The field as a graph (2D)

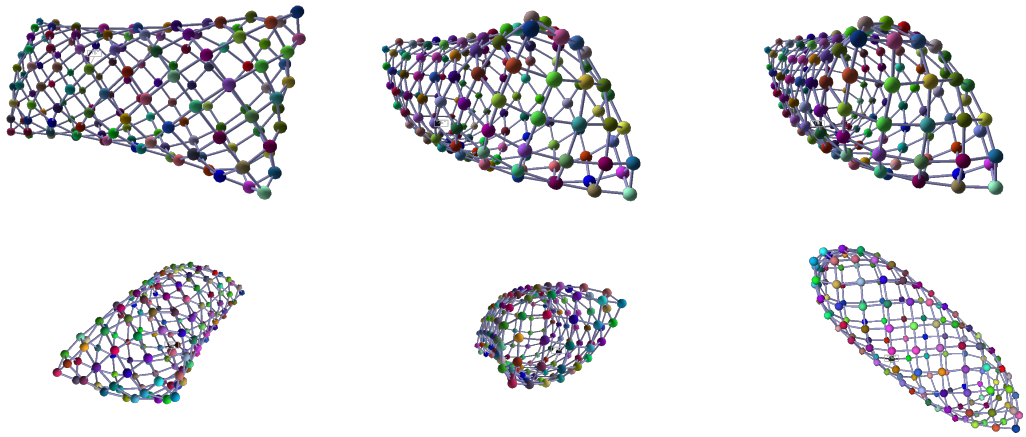

Figure 5: The field as a graph (3D)

2.2 Movement

OpenSSH uses the MD5 cryptographic hash function to generate a 128 bit fingerprint for the server's key. The visualization algorithm splits the fingerprint into 64 pairs of 2 bits. Figure 6 shows in which order these bit pairs are processed during the 64 steps of the algorithm: byte-wise from left to right and least significant bits first.

```
Fingerprint      fc      :      94      :     b0       :      c1       :    ...    :      b7

Bits        11 11 11 00  :  10 01 01 00  : 10 11 00 00  :  11 00 00 01  :    ...    :  10 11 01 11
             |  |  |  |      |  |  |  |     |  |  |  |      |  |  |  |                    |  |  |  |
Step         4  3  2  1      8  7  6  5    12 11 10  9      16 15 14 13       ...       64 63 62 61
```

Figure 6: Order of fingerprint processing

Each bit pair decides on the direction the bishop is heading at the current step (figure 7).[3])

Table 8 shows an example, the situation before the first move. From the start position in the middle of the field (76), bishop Peter can go to the positions located one step away in diagonal direction (58, 60, 92 or 94). The difference between the new position and the old position will be called *offset*.

These diagonal moves can only be performed if Peter is the middle of the field an no walls hinder him. These positions are marked as `M` in figure 10. If Peter is at the border of the field, he cannot go through the wall, but has to slide to the side.

doubles the height, because in standard terminal fonts the characters have double height.

[3]Each bit pair is taken as a tuple (V, H) where the first bit V decides on the vertical component (0=up, 1=down) and the second bit H decides on the horizontal component (0=left, 1=right).

Bits	Direction
00	↖
01	↗
10	↙
11	↘

Figure 7: Movement according to bit pairs

bits	direction	new position	offset
00	↖	58	58 - 76 = -18
01	↗	60	60 - 76 = -16
10	↙	92	92 - 76 = +16
11	↘	94	94 - 76 = +18

Figure 8: Possible movements from the start position (76)

In figure 10 these border positions are shown as `T` on the top, `B` at the bottom, `L` on the left and `R` on the right. The four positions in the corners of the field are shown as `a`, `b`, `c` and `d`.

Table 9 shows the real directions and the corresponding offsets for all types of positions:

2.3 Coverage

A position is called *empty*, if our bishop has never been there before. Otherwise it is called *visited*. A *walk* is the sequence of visited positions resulting from a particular fingerprint. It can be described as a 64-tuple of positions.

$$\text{walk} = (pos_{step1}, pos_{step2}, ..., pos_{step64}) \in N^{64}$$

For example, the fingerprint shown in the examples above
(fc:94:b0:c1:e5:b0:98:7c:58:43:99:76:97:ee:9f:b7)
generates the following walk:
(76, 58, 76, 94, 112, 94, 78, 62, 78, 60, 42, 60, 76, 60, 42, 24, 42, 26, 10, 26, 44, 26, 8, 26, 42, 24, 40, 24, 40, 22, 40, 58, 42, 24, 40, 24, 8, 26, 8, 7, 8, 9, 25, 9, 25, 41, 25, 43, 27, 45, 29, 13, 29, 45, 63, 79, 97, 115, 133, 117, 133, 151, 135, 152, 151)

The maximum number of visited positions is achieved if all 64 steps move the bishop to positions where he hasn't been before, i.e. the start position and 64 different other positions are covered:

$$\text{maximal coverage} = \frac{1+64}{153} \approx 42.48\%$$

pos	bits	headed dir.	real dir.	offset
M	00	↖	↖	-18
	01	↗	↗	-16
	10	↙	↙	+16
	11	↘	↘	+18
T	00	↖	←	-1
	01	↗	→	+1
	10	↙	↙	+16
	11	↘	↘	+18
B	00	↖	↖	-18
	01	↗	↗	-16
	10	↙	←	-1
	11	↘	→	+1
L	00	↖	↑	-17
	01	↗	↗	-16
	10	↙	↓	+17
	11	↘	↘	+18
R	00	↖	↖	-18
	01	↗	↑	-17
	10	↙	↙	+16
	11	↘	↓	+17
a	00	↖	no move	0
	01	↗	→	+1
	10	↙	↓	+17
	11	↘	↘	+18
b	00	↖	←	-1
	01	↗	no move	0
	10	↙	↙	+16
	11	↘	↓	+17
c	00	↖	↑	-17
	01	↗	↗	-16
	10	↙	no move	0
	11	↘	→	+1
d	00	↖	↖	-18
	01	↗	↑	-17
	10	↙	←	-1
	11	↘	no move	0

Figure 9: Possible movements for each type of position

```
              1111111
  01234567890123456
 +-----------------+x
0|aTTTTTTTTTTTTTTTb|
1|LMMMMMMMMMMMMMMMR|
2|LMMMMMMMMMMMMMMMR|
3|LMMMMMMMMMMMMMMMR|
4|LMMMMMMMMMMMMMMMR|
5|LMMMMMMMMMMMMMMMR|
6|LMMMMMMMMMMMMMMMR|
7|LMMMMMMMMMMMMMMMR|
8|cBBBBBBBBBBBBBBBd|
 +-----------------+
 y
```

Figure 10: Types of positions

So always more than half the field stays empty. For now, this is just an upper limit: It has to be verified if it is really possible to make 64 steps without re-visiting a position.

On the other hand, minimal coverage is reached if the bishop just goes back and forth between the start position and one of the four neighbor positions. This can take place in exactly four variants (figure 11).

$$\text{minimal coverage} = \frac{2}{153} \approx 1.31\%$$

1. Movement: ↖ ↘ ↖ ↘ ...
   Bit values: 11 00 11 00 binary = 0xcc hex
   Fingerprint: cc:cc:cc:cc:cc:cc:cc:cc:cc:cc:cc:cc:cc:cc:cc:cc
2. Movement: ↗ ↙ ↗ ↙ ...
   Bit values: 10 01 10 01 binary = 0x99 hex
   Fingerprint: 99:99:99:99:99:99:99:99:99:99:99:99:99:99:99:99
3. Movement: ↙ ↗ ↙ ↗ ...
   Bit values: 01 10 01 10 binary = 0x66 hex
   Fingerprint: 66:66:66:66:66:66:66:66:66:66:66:66:66:66:66:66
4. Movement: ↘ ↖ ↘ ↖ ...
   Bit values: 00 11 00 11 binary = 0x33 hex
   Fingerprint: 33:33:33:33:33:33:33:33:33:33:33:33:33:33:33:33

Figure 11: Movements leading to least coverage

2.4 Values

Each position holds a *value* that counts how often it was visited by the bishop. The values are represented graphically as the symbols in figure 12.[4] Initially all positions on the field are initialized with the value 0 (white space). The special values 15 and 16 (`S`) and the end value 16 (`E`) mark the start and end position of the walk and overwrite the real value of the respecting position.

Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Character		.	o	+	=	*	B	O	X	@	%	&	#	/	^	S	E

Figure 12: Symbols for all possible values

[4]The intention behind the choice of symbols was to mimic a line of ASCII characters which is getting thicker each time it is augmented upon intersecting with itself.

3 Markov analysis

The visualization algorithm works on 128 bits produced by the MD5 hash algorithm. Let us assume that this hash value is random. In this section we compute the probability that bishop Peter is on position p after his ith move.

3.1 Transistion matrix

Bishop Peter moves from position to position on the field. Each position he moves to can be seen as a state in a Markov model.

Using the offsets in the tables above we can now identify the probability to go from position i to position j:

$$p_{i,j} = 0.25 \qquad \text{if} j - i \in \text{offsets}(p)$$

$$p_{i,j} = 0 \qquad \text{if} j - i \notin \text{offsets}(p)$$

This gives a 153×153 transition matrix. Most elements are 0, but some are 0.25. The structure of the matrix is shown in figure 14: Black pixels are used for $p = 0.25$, grey pixels for $p = 0$.

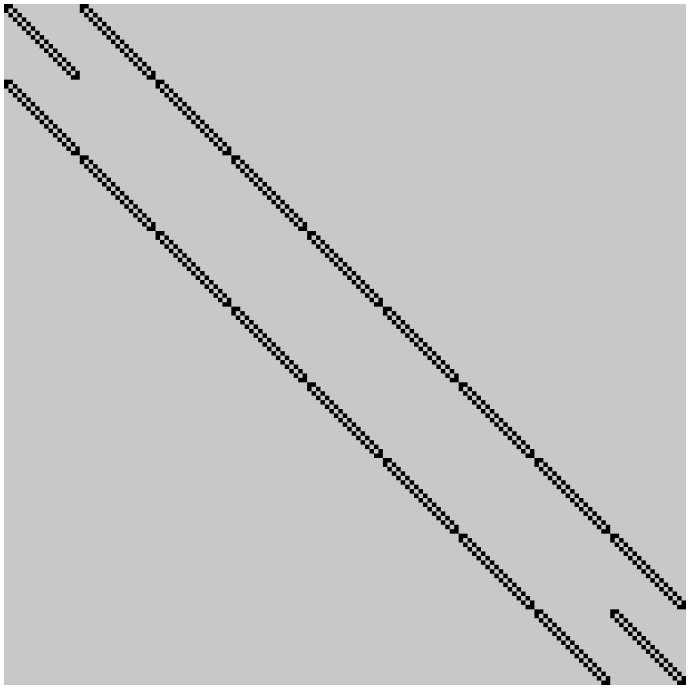

Figure 13: Transition matrix (black pixels for $p = 0.25$, grey pixels for $p = 0$)

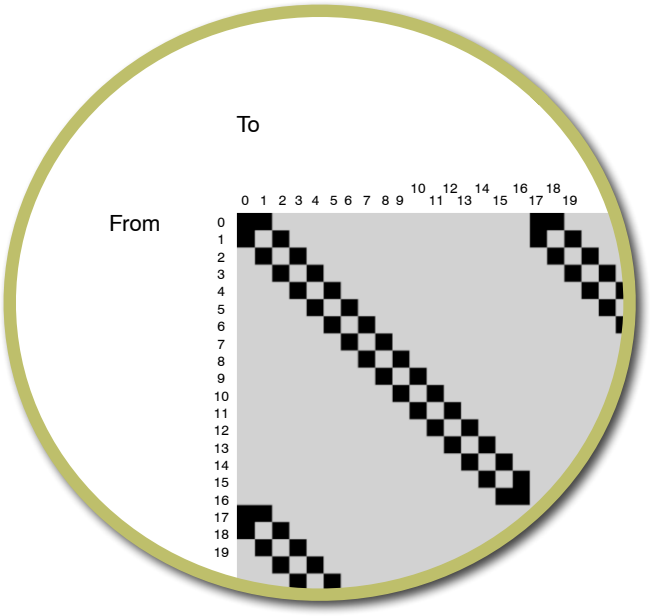

Figure 14: Upper left corner of the transition matrix

3.2 Probabilities for each position

The initial vector $\lambda = (0, ..., 0, 1, 0, ..., 0)$ shows the start of the walk: We have a probability of p=1 (certainty) to be on position 76 at the start.

By computing $p^{(s)} = \lambda M^s$ we get the probabilities to reach a given position after s steps: It gives a vector which holds the probability for each of the 153 positions.

By computing $p^{(64)} = \lambda M^{64}$ we can now tell how probable it is for each position that the walk ends there. Figure 15 shows the result graphically. The most probable positions for the end of the walk (i.e. the symbol `E`) are the start position and the positions diagonally next to it. The least probable positions are horizontally and vertically next to the start position.

How far is the bishop away from the start position after n steps? By weighing the distance of each position on the field with the probability that it is reached after n steps. Figure 16 shows the results for a taxi cab metric (distance $= \delta x + \delta y$), but other distance metrics could be used in the same way.

3.3 Reachability

By looking at $p^{(s)}$ we can see which positions are reachable after `s` steps: If the probability $p_i^{(s)}$ for a position i is zero, it cannot be reached. If probability is

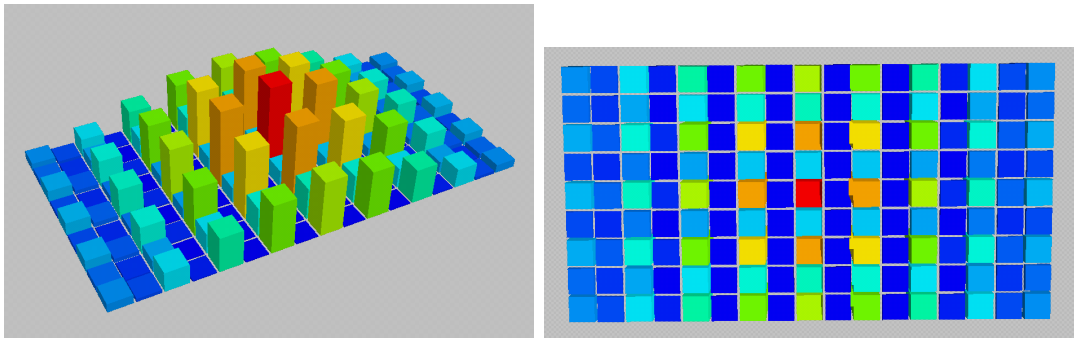

Figure 15: Probability for the end of the walk

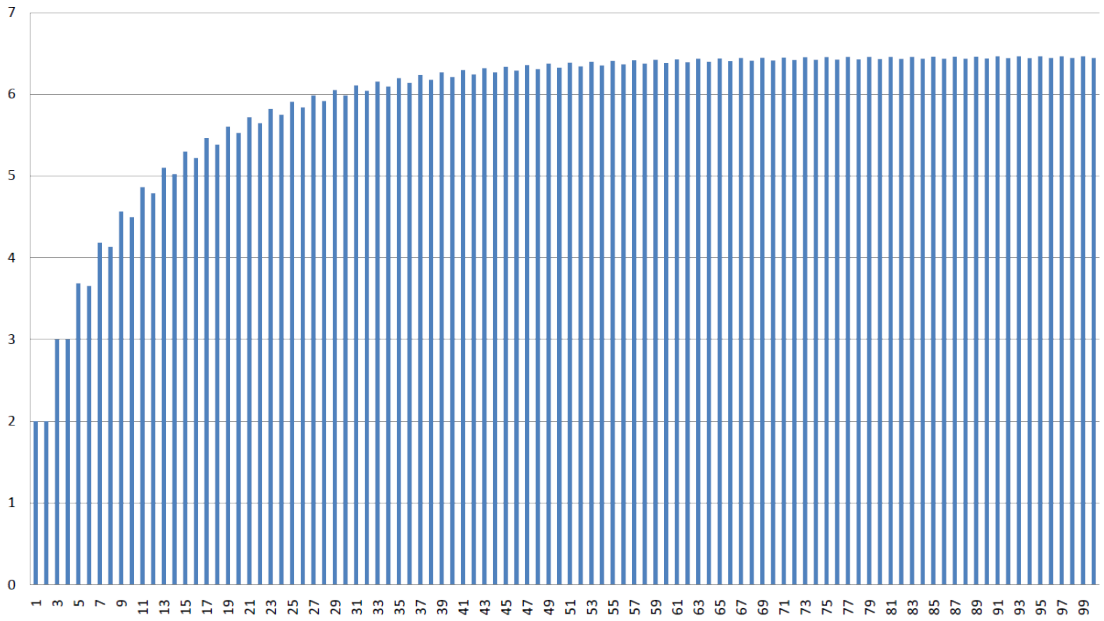

Figure 16: Mean distance from the start position after n steps (Taxi driver metric)

positive, it can.

All elements of $p^{(64)}$ are positive, so all positions are reachable in 64 steps.

By consecutively computing $p^{(s)}$ for each step and checking if the probability is zero for a position or not, we can identify after how many steps it is first reachable. Figure 17 shows the output.

8	7	6	5	4	5	4	5	4	5	4	5	4	5	6	7	8
8	7	6	5	6	3	6	3	6	3	6	3	6	5	6	7	8
8	7	6	7	4	7	2	7	2	7	2	7	4	7	6	7	8
8	7	8	5	8	3	8	1	8	1	8	3	8	5	8	7	8
8	9	6	9	4	9	2	9	2	9	2	9	4	9	6	9	8
8	7	8	5	8	3	8	1	8	1	8	3	8	5	8	7	8
8	7	6	7	4	7	2	7	2	7	2	7	4	7	6	7	8
8	7	6	5	6	3	6	3	6	3	6	3	6	5	6	7	8
8	7	6	5	4	5	4	5	4	5	4	5	4	5	6	7	8

Figure 17: Steps needed to reach a given position from the start position

4 Collisions

4.1 Random brute force attacks

We have an original fingerprint and its visual representation (the *picture*). Now we try to find similar fingerprints by randomly generating fingerprints and producing the pictures. These pictures are compared using a similarity function. This process is repeated multiple times. Finally, the generated fingerprint that have the highest similarity to the original are shown to the user.

The similarity metric should be fast, because we want to test a high number of fingerprints. And it should resemble human pattern recognition, because as humans remember and compare the visual fingerprints (pictures), the more the similarity metric is alike to human perception, the better the results of the attack is.

In our tests we used multiple metrics for determining the similarity. Simple distance metrics regard each point within the field seperately. If two points at the same position in both fields are equal, the distance index is not increased. If they differ, the distance index is incremented by a certain value. We tested mutiple methods here: using the difference between both field numbers, using the squared difference, and ignoring the exact field number and only distinguishing field number $n == 0$ and $n > 0$.

Figure 18 shows an example result. At the top is the picture for the original fingerprint. Shown below is a different fingerprint that generates a similar picture.

Even the best pictures from a set of several million candidates are still easily distinguishable from the original.

4.2 Constructing collisions based on graph theory

Instead of generating random pictures and testing them for similarity, we now want to generate fingerprints for which we know that they produce the same picture. The underlying observation is that it does not matter if the bishop goes clock-wise or anti-clock-wise: in both cases the picture will be a circle.

So we just repeatedly swap parts of the walk between two cut points.[5]. Cut points are two occurences of the same position in the walk, randomly selected. We generate the fingerprint from the resulting walk and store it in a list, if it is not already in there.

Here is an example: We try to find collisions for the fingerprint in figure 1: fc:94:b0:c1:e5:b0:98:7c:58:43:99:76:97:ee:9f:b7. Its walk is (76, 58, 76, 94, 112, 94, 78, 62, 78, 60, 42, 60, 76, 60, 42, 24, 42, 26, 10, 26, 44, 26, 8, 26, 42, 24, 40, 24,

[5]In the context of genetic algorithms, this is called the inversion operator (for mutation)

```
Original digest: 'fc94b0c1e5b0987c5843997697ee9fb7'
Original visualization:
+--[ n/a 9999]----+
|       .=o.  .   |
|     . *+*. o    |
|      =.*..o     |
|       o + ..    |
|        S o.     |
|         o  .    |
|          .  . . |
|              o .|
|               E.|
+-----------------+
53 : 731ee54c82233359e3d5e9f6ccf87e1f  distance=2263
+--[ n/a 9999]----+
|        o .. .   |
|       + +  o    |
|      = + ..o    |
|       + . *o    |
|        S o.o=   |
|         + .. +  |
|          .  . E |
|              . o|
|             ...o|
+-----------------+
```

Figure 18: Searching for similarities

40, 22, 40, 58, 42, 24, 40, 24, 8, 26, 8, 7, 8, 9, 25, 9, 25, 41, 25, 43, 27, 45, 29, 13, 29, 45, 63, 79, 97, 115, 133, 117, 133, 151, 135, 152, 151). Possible cut points are all the positions that occur more than once in the walk.

Figure 19 shows some of the generated collisions. All of them give the same picture.

4.3 Bruteforcing identical visual fingerprints

In this idea, we want to determine the number of possible collisions with a predefined visual fingerprint by brute-force. Given is a matrix that is generated by a normal visual fingerprint walk. The algorithm imitates the random walk and starts at the normal starting position in the center of the field. For each of the four directions, it checks if the target field was covered by the original fingerprint. If it was covered, the number within the field is decremented and the next step is determined like the first one. If no succeding step is possible any more, as their coverage numbers are equal to 0, the algorithm backtracks until a new direction is possible again. Using this exhaustive search, it is possible to find all hashes that produce an identical visual fingerprint.

As the exhaustive search has complexity $O(c^n)$ with n as the length of the given hash / fingerprint walk, it is very slow. In our experiments we did not succeed in finding all collisions on a hash with 128 bits within 2 weeks. To alleviate this problem, we reduced the length of the examined hashes to a manageable size. Our intention was to find trends in the number of collisions of hashes per hash size, so that we would be able to extrapolate information for long hashes from observed trends within the shorter hashes. Figure 20 shows the number of collisions within multiple hashes: the x-axis shows the number of hash bytes, the logarithmic y-axis shows the amount of collisions found for specific hashes. Each line corresponds to a specific hash that is analyzed from one byte to 10 bytes. Unfortunately, randomly chosen hashes did not show much similarity: at higher sizes, the number of collisions vary greatly and do not only depend on the number of covered fields, but also on the shape of the produced figure. The number of collisions also do not increase monotonously with hash size, but may even decrease in certain cases. Due to these features of the number of collisions, we do not expect to be able to extrapolate trends for higher hash sizes.

```
fc:94:b0:c1:e5:b0:98:7c:58:43:99:76:97:ee:9f:b7  (original)
[...]
09:1d:0f:da:c8:fd:e9:40:53:42:99:76:97:ee:9f:b7
09:1d:27:da:83:dc:fe:94:d0:40:99:76:97:ee:9f:b7
09:1d:8f:c8:3d:fe:94:80:75:42:99:76:97:ee:9f:b7
09:1d:9a:dc:3c:fe:94:50:0e:43:69:79:97:ee:9f:b7
09:1d:a7:9f:0e:34:8c:9c:f5:40:69:79:97:ee:9f:b7
09:1d:ca:d9:3c:fe:94:50:0e:43:69:79:97:ee:9f:b7
09:1d:ca:d9:3c:fe:94:50:0e:43:99:76:97:ee:9f:b7
09:1d:da:9f:0e:84:dc:13:e7:40:99:76:97:ee:9f:b7
09:1d:e3:9f:0e:84:9c:3d:4d:42:69:79:97:ee:9f:b7
09:1d:e3:9f:0e:84:c9:3d:4d:42:69:79:97:ee:9f:b7
09:1d:eb:4f:09:98:dc:43:4f:42:69:79:97:ee:9f:b7
09:1d:eb:4f:09:98:dc:43:e7:40:69:79:97:ee:9f:b7
09:35:3b:c8:ed:4f:09:d8:e5:40:69:79:97:ee:9f:b7
[...]
1c:71:a2:8d:b7:4f:09:c8:dd:40:69:79:97:ee:9f:b7
1c:71:a2:fd:e9:10:d2:c8:dd:40:69:79:97:ee:9f:b7
1c:71:ba:4f:09:d8:c2:d9:f4:40:99:76:97:ee:9f:b7
1c:71:ca:98:b7:4f:09:c8:f5:40:69:79:97:ee:9f:b7
1c:71:e2:9f:0e:21:9c:d9:dc:40:69:79:97:ee:9f:b7
1c:71:e2:fb:94:20:9c:d9:dc:40:69:79:97:ee:9f:b7
1c:71:f2:88:b7:4f:09:c8:d9:44:99:76:97:ee:9f:b7
1c:71:f2:c8:98:dc:9f:0e:21:45:99:76:97:ee:9f:b7
1c:74:0e:2b:c9:d9:9f:0e:c1:41:69:79:97:ee:9f:b7
1c:74:8a:87:c9:bd:4f:09:0d:43:69:79:97:ee:9f:b7
[...]
fc:94:d0:70:b2:c8:2d:d9:1b:44:69:79:97:ee:9f:b7
fc:94:d0:70:b2:c8:bd:81:87:45:69:79:97:ee:9f:b7
fc:94:d0:70:b2:c8:bd:81:87:45:99:76:97:ee:9f:b7
fc:94:d0:70:b2:d8:1b:c8:87:45:69:79:97:ee:9f:b7
fc:94:d0:70:e2:1b:92:c9:cd:41:69:79:97:ee:9f:b7
fc:94:d0:70:e2:1b:c8:99:c7:41:69:79:97:ee:9f:b7
fc:94:d0:70:e2:1b:c8:99:c7:41:99:76:97:ee:9f:b7
fc:94:d0:70:e2:c2:d9:1b:d8:41:99:76:97:ee:9f:b7
fc:94:d0:70:e2:c8:8d:79:1b:41:99:76:97:ee:9f:b7
fc:94:d0:a0:9c:bd:c1:78:71:42:69:79:97:ee:9f:b7
fc:94:d0:a4:78:1b:8c:c9:35:43:69:79:97:ee:9f:b7
fc:94:d0:a4:8c:99:7c:1b:34:43:69:79:97:ee:9f:b7
fc:94:d0:a4:92:dc:d8:1b:34:43:99:76:97:ee:9f:b7
fc:94:d0:a4:98:bd:c1:c2:35:43:69:79:97:ee:9f:b7
fc:94:d0:a4:98:cd:1b:c2:35:43:69:79:97:ee:9f:b7
fc:94:d0:a4:c2:bd:81:d9:34:43:69:79:97:ee:9f:b7
fc:94:d0:a4:c2:bd:81:d9:34:43:99:76:97:ee:9f:b7
fc:94:d0:a4:c2:d9:1b:d8:34:43:99:76:97:ee:9f:b7
fc:94:d0:b0:bc:81:dc:92:75:42:69:79:97:ee:9f:b7
fc:94:d0:b8:21:d9:c8:4d:27:43:99:76:97:ee:9f:b7
```

Figure 19: Collisions for the fingerprint of anoncvs.de.openbsd.org

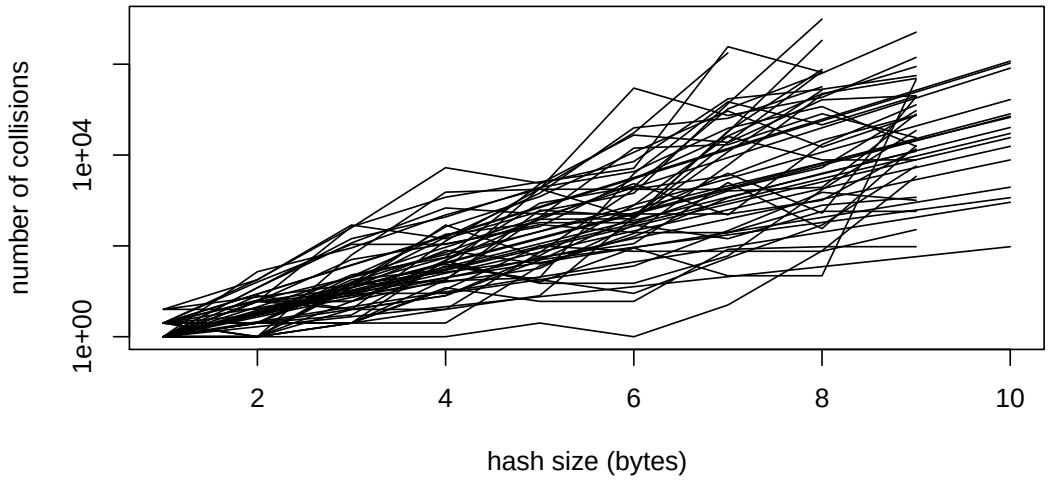

Figure 20: Number of collisions related to the size of hashes, each line corresponds to an individual hash

5 Ideas for further work

Based on a Markov model and some brute-force attacks we were only able to produce some basic results. But we hope that our analysis will spur further research on this topic. More questions could be asked. For example:

- How many fingerprints produce the same *picture* (same values for all positions) or the same *shape* (same visited positions, maybe with different values) as a given fingerprint?
- Can we measure how good a given picture is using those coun.
- How many different pictures and shapes can the algorithm produce?
- How can the hash value be reconstructed from a given visualization?
- How many visualizations can a person distinguish?
- How many self-avoiding walks (coverage = 65/153) exist on the standard field?

Some of those questions may be hard to answer. The following techniques could be used to make the problem easier:

- Reduce the field size. For example, take a 3x3 field instead of a 17x9 one.
- Change the algorithm to flip sides at the borders. This would turn the field into a torus.
- Remove the borders completely. This would lead to a planar graph, which could be analyzed more easily using graph theory.

In the end this kind of research might reveal whether the (heuristically designed) algorithm is secure enough for this purpose.

References

[1] Plasmoid of THC. Fuzzy fingerprints – attacking vulnerabilities in the human brain, 2003. `http://freeworld.thc.org/papers/ffp.pdf`.

[2] Adrian Perrig and Dawn Song. Hash visualization: a new technique to improve real-world security. In *Proceedings of the International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC)*, pages 131–138, July 1999.

[3] Alexander von Gernler. OpenSSH CVS commit message for revision 1.70 of key.c. `http://www.openbsd.org/cgi-bin/cvsweb/src/usr.bin/ssh/key.c?rev=1.70`, 2008.

[4] Alexander von Gernler. SSH fingerprint visualization support. c2k8 developer blog. `http://undeadly.org/cgi?action=article&sid=20080615022750`, 2008.