An Approach for Text Steganography Based on Markov Chains

H. Hernan Moraldo

Abstract. A text steganography method based on Markov chains is introduced, together with a reference implementation. This method allows for information hiding in texts that are automatically generated following a given Markov model. Other Markov - based systems of this kind rely on big simplifications of the language model to work, which produces less natural looking and more easily detectable texts. The method described here is designed to generate texts within a good approximation of the original language model provided.

Keywords: steganography, Markov chain, Markov model, text, linguistics

1 Introduction

Steganography is the field that deals with the problem of sending a message from a sender A to a recipient B through a channel that can be read by a so-called Warden, in a way that the Warden doesn't suspect that the message is there.

Steganographic techniques exist for hiding messages in images, audio, videos, and other media. In particular, text steganography studies information hiding on texts. There are many techniques for this, as summarized on [1] [2]. One of the simplest steganographic methods for texts works by encoding a fixed amount of bits per word, using a table that maps words to codes, and vice versa. A disadvantage of this trivial technique is that the text will be obviously random at a syntactic level, as words are generated in a way that is independent of context.

There are other simple methods that store data in the text format, using spacing, capitalization, font or HTML tags. For example SNOW [3] hides information in tabs and spaces at the end of each line, that are usually not visible on text viewers. Also, some techniques start from a base text (the covertext), and modify it in some way: for example by switching words to near synonymous, or by changing sentences from their original grammatical structure to another one that preserves the meaning. The technique shown in [4] hides information by modifying words in a way that resembles ortographical or typographical errors. There are other techniques that rely on translation [5].

In other cases, texts are generated using a grammar model; this kind of system has the advantage of producing texts that make sense at a grammatical level, although not at a semantical level.

And there are techniques, like the one described on this paper, that are based on using Markov models to generate texts that encode some hidden message on them. Weihui Dai et al. [6] [7] explore a method for encoding data on this way; [8] shows a simple implementation of a similar concept.

This article explores a specific method for using Markov chains for text steganography. How this method compares to other similar methods and how it works is explored further in the next sections. A reference implementation of the method described here is also included in the open source program Markov-TextStego [9].

2 Related Work

Many methods for text steganography that are not based on Markov chains are known. An example is NiceText [10], which shows a way to encode ciphertext to text, that uses custom styles, Context Free Grammars and dictionaries.

The approach used in [6] [7] is based on Markov chains. When encoding, some data is provided as input, and the system generates a text as output using a given Markov chain. The stegotexts are generated in a way that simulates that they were generated by the Markov chain.

However, to avoid complex calculations, the Markov model is simplified by assuming that all probabilities from a given state to any other state are equal. This can change the quality of the texts generated by the Markov chain significantly. For example words like "the" and "naturally" are both potential starts of a phrase, but the former should be much more frequent than the latter; and this difference is not preserved by the simplification.

Other Markov - based models or similar models require of similar simplifications of the Markov chain, typically by making all outbound probabilities of each state equal (as in the previous example), or by replacing them by other ones, either explicitly or implicitly through the operation of the encoding algorithm [8] [11] [12].

The method described here aims to be an answer to the question of whether it is possible to preserve the probabilities in the Markov models to higher levels of accuracy. The method is not optimally precise, but it generates texts that use a language model that is a good approximation of the provided Markov model.

3 Markov Chain Models

A Markov chain is a model for a stochastic process. A sequence of random variables $X = (X_1, ..., X_T)$ with values from a finite set S is a Markov chain, if it has the Markov properties [13] [14]:

Limited Horizon property:

$$P(X_{t+1} = s_k | X_1, ..., X_t) = P(X_{t+1} = s_k | X_t) \tag{1}$$

Time Invariant property:

$$P(X_{t+1} = s_k | X_t) = P(X_2 = s_k | X_1) \tag{2}$$

The first property means that the Markov chain doesn't have memory of any states, beyond the last one. The second property means that the conditional probabilities for all states do not depend on the position (time) on the sequence.

Diagrams like the one shown in Fig. 1 are frequently used to represent the transitions in Markov chains. All nodes in the graph represent states (elements of S), and any arrow from s_j to s_k with a value of p means that $P(s_k|s_j) = p$. We call any state s_k an outbound state of s_j, if there is an arrow from s_j to s_k. For every s_j that doesn't have an arrow to another s_k state, $P(s_k|s_j) = 0$.

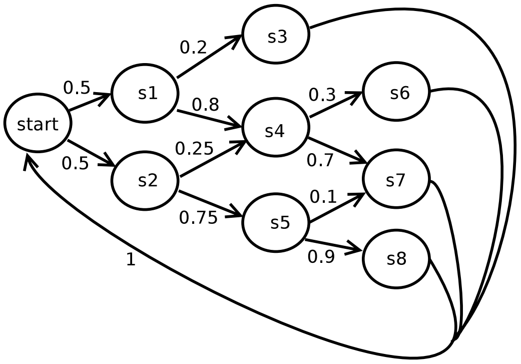

Fig. 1. Example Markov chain. In the context of text steganography, each state is a word. The special state "*start*" marks both the start and end of a sentence.

Markov chains and models are frequently used to model language [13]; when that's the case, states in the chain are used, for example, to represent words, characters, or n-grams. Also, Markov models are used in steganography (as described above), and in steganalysis [15] [16].

A Markov language model may be useful to compute probabilities for phrases, from the n-gram probabilities. For example given the Markov chain shown in Fig. 1, if the process were to start from "*start*" (symbol that we use both for start and end of a sentence), the probability of generating the text composed by the sequence of words or states $[s1, s4, s7]$ would be 0.28.

These models can also be used to generate random texts. For this, a random source is used that can pick a next state s_k with probability $P(s_k|s_j)$, given the current state s_j. The algorithm for generating the random text starts by setting "*start*" to be the current state; then, in every iteration it uses the random source to pick the next word, which also becomes the new current state in the next iteration. To generate a single sentence, the process can be made to stop when the state "*start*" is reached.

Although Markov chains only have memory of a single previous state, every state can be a bigram, or an n-gram. This way, a Markov language model can have memory for more than a single word. Although this article only describes the steganographic method based on states that are single words, the reference implementation of the system [9] allows using both unigrams and bigrams as states, and it is possible to extend it to support n-grams with $n > 2$. Table 3 compares the results of the encoding procedure when using unigrams and bigrams.

A Markov language model with states as single words can be computed from the frequencies of all bigrams, and all unigrams in a text:

$$P(w_n|w_{n-1}) = \frac{count(w_{n-1}, w_n)}{count(w_{n-1})} \tag{3}$$

where $count(a, b)$ is the number of occurrences of word a followed immediately by word b in the text, and $count(a)$ is the number of occurrences of word a.

As discussed above, some steganographic methods are based on these Markov language models. The language model is usually simplified in some way; for example [6] sets all $P(x|w)$ with w fixed, to a fixed k.

In these models, once the simplification is done, the Markov chain is used to encode data into text; every word stores some fixed or variable amount of bytes, and every bigram in the generated text is required to have conditional probability $P(w_n|w_{n-1}) > 0$ in the Markov chain. A decoding algorithm that reverses the process, transforming the text into data, is also defined.

The approach shown in this article avoids much of the simplification in the probabilities of the Markov chain. Although there is still some precision loss in the model, for the most part, the proportions between the frequencies of different n-grams are preserved, specially for long texts.

4 Fixed-size Steganography

A main objective in this article is to describe two functions, *encode*, and *decode*, that are used to create a text out of a data input, and to get the original data out of an encoded text. In steganography literature, it would be said that *encode* generates a stegotext out of the input plaintext, while *decode* does the reverse process. The *encode* function is not cryptographically secure; it assumes that its input is a plaintext, or some data that has already been encrypted using an independent system. In the latter case, *encode*'s input can be called ciphertext.

We require the encoding function to be invertible; that is, for every input data d_1 and d_2, $encode(d_1) = encode(d_2)$ only if $d_1 = d_2$. Also *decode* is the inverse of *encode*, so for every input d, $decode(encode(d)) = d$. The encoding function *encode* is required to work on all the domain of data d; the required domain of *decode* however needs only be the image of *encode*.

The *encode* and *decode* functions will be built out of simpler functions, for fixed-size encoding and decoding. These functions are $encode_{fixed}(data, datasize)$, and $decode_{fixed}(text, datasize)$. Both the Markov chain and the starting symbol are actually required for these functions too, but they are left out for simplicity. Only when it is required for the purposes of the explanation, a third argument is added to both functions, for the start symbol: $encode_{fixed}(data,\ datasize,\ startsymbol)$, and $decode_{fixed}(text, datasize, startsymbol)$.

In this system, the size of d in bits is known beforehand both for $encode_{fixed}$ and for $decode_{fixed}$. The requirements for both functions are weaken compared to those for their non-fixed counterparts; it is required that for every input data d_1 and d_2 such that $length(d_1) = length(d_2)$, $encode_{fixed}(d_1, length(d_1)) = encode_{fixed}(d_2, length(d_2))$ only if $d_1 = d_2$ (where $length(d)$ is the size of d in bits). This weaker restriction means that the encoder may produce the same text for two different data inputs, only if they have different lengths, as can be seen in the examples in Table 1.

Also, $decode_{fixed}(z, length(z)) = d$ with $z = encode_{fixed}(d, length(d))$.

4.1 Mapping of Probabilities to Ranges

A basic component for encoding and decoding is the function named *subranges*, that maps all outbound states from a given state, to subranges of a given range. These subranges are a partition of the original range.

$$subranges(mc, s, r) = [(s_1, r_1), ..., (s_n, r_n)] \tag{4}$$

where mc is a Markov chain, s is a state in S, and r is a range of natural numbers $[a, b]$. The result is a list that must have some properties that are described below.

The behavior of this function is that it maps outbound states of a Markov chain to subranges of a given range, in a way that approximately matches the proportion between the sizes of the different subranges, to the proportion between the probabilities of the respective states. For example, if mc is the chain in Fig. 1, $s = start$, and $r = [0, 3]$, the expected result would be $[(s_1, [0, 1]), (s_2, [2, 3])]$. That is, because each outbound state has a 0.5 probability, it has to get half of the full range. If $s = s_2$, the expected result would be $[(s_4, [0, 0]), (s_5, [1, 3])]$, where again the length of the subranges matches the proportion of their respective probabilities.

This partitioning method will be used in an iterative way, both for encoding and for decoding. Fig. 2 (in Section 4.2) shows how this is done, although the details of the operation are described in the next sections.

The returned value for *subranges* in Equation 4 is a list of pairs (s_k, r_k), where s_k is a state such that $P(s_k|s) > 0$, and r_k is a subrange of r. The subranges of r returned by *subranges* are a partition of r.

A good implementation of the function generates a mapping between subranges r_k and states s_k, such that the fraction of the total range length for each r_k is approximately equal to the probability of the respective state s_k. That is:

$$\frac{length(r_k)}{length(r)} \approx P(s_k|s) \tag{5}$$

Where the length of a range is defined to be $length([a, b]) = b - a + 1$.

The property in Equation 5 is not a strict requirement, as even without this condition the encoding function will still generate texts that are decoded correctly. However, only when this condition is held the texts that are generated will be approximately described by the original Markov language model.

[**Condition of Minimal Length**] The following condition is actually required for the steganographic system to work, however. Any time that there are at least two different states s_1 and s_2 such that $P(s_1|s) > 0$ and $P(s_2|s) > 0$ (that is, every time s has at least two outbound states), it is required that *subranges* returns a list with at least two elements.

This restriction is necessary for ensuring that both the encoder and decoder methods halt for all inputs. It might produce precision loss in many cases however, as in the following example: an s state has two outbound states s_1 and s_2, with conditional probabilities $P(s_1|s) = 0.99$ and $P(s_2|s) = 0.01$, and the input range to process is $r = [0, 1]$.

In this case, it would seem that the best output would map s_1 to the full range: the returned value for this would be $[(s_1, [0, 1])]$. However this value doesn't hold the Condition of Minimal Length, as the list has a single element, despite s having more than one outbound state.

Because of this, the only valid results for this example would be $[(s_1, [0, 0]), (s_2, [1, 1])]$ and a symmetrical one (same ranges but switching states). As can be seen, these valid options are worse approximations to the input conditional probabilities, than just mapping s_1 to the full state; however the condition described disallows this better approximation.

The following three functions are used by the encoding and decoding methods.

As described, *subranges* returns a list that maps ranges to states. The function *subrangeForState* uses the list to return the subrange that is assigned to a given state:

$$\begin{aligned} subrangeForState(mc, s_k, r, s_l) = r_k \\ \text{from } [..., (s_k, r_k), ...] = subranges(mc, s_k, r) \text{ such that } s_k = s_l \end{aligned} \tag{6}$$

The function *subrangeForNumber* returns the subrange in the list that contains a given number:

$$\begin{aligned} subrangeForNumber(mc, s_k, r, number) = \text{ subrange } r_k \\ \text{from } [..., (s_k, r_k), ...] = subranges(mc, s_k, r) \text{ such that } number \in r_k \end{aligned} \tag{7}$$

The function *stateForNumber* returns the state that is assigned to the subrange returned by *subrangeForNumber*:

$$\begin{aligned} stateForNumber(mc, s_k, r, number) = \text{ state } s_k \\ \text{from } [..., (s_k, r_k), ...] = subranges(mc, s_k, r) \text{ such that } number \in r_k \end{aligned} \tag{8}$$

These functions will be used in the next sections.

4.2 Encoding Fixed-size Data Using Markov Chains

The function $stateForNumber$, can also be seen as a function that encodes data to a single word. Given a Markov chain, a state, a range and a number (the input data), it finds the corresponding state or word in the chain for that number. Related to that, $subrangeForNumber$ also defined above, returns the subrange that corresponds to the word returned by $stateForNumber$.

Based on these two functions, a sequence of states s_t and a sequence of ranges r_t can be generated as described in the following two equations. These sequences are computed given a Markov chain mc, an initial state s_0 (typically "$start$"), an input data ($number$), and an initial range r_0 (typically $[0, 2^n - 1]$, where n is the length of the data to store):

$$s_t = stateForNumber(mc, s_{t-1}, r_{t-1}, number) \tag{9}$$

$$r_t = subrangeForNumber(mc, s_{t-1}, r_{t-1}, number) \tag{10}$$

Both sequences are defined to be finite (as we want to encode data to a finite sequence of words); the final element for both is T such that $length(r_T) = 1$. This means that we stop encoding when the sequence of words describes a single number.

Finally, $encode_{fixed}(data, length(data)) = [s_1, ..., s_T]$.

This encoding process works by partitioning an input range in a way that matches the outbound states of a given state, and then selecting the outbound state whose subrange contains the number to encode. After this is done, the selected subrange and state are used as the input for the next iteration of the algorithm. When the process finishes, the encoded text is the sequence of states that the algorithm went through.

The $subranges$ function is restricted by the Condition of Minimal Length in Section 4.1 to always split a range in more than one subrange, whenever possible; therefore the iteration of this process will produce ranges that are smaller and smaller. (Even though it is possible that a Markov chain that is computed from a text contains states with only one outbound state, those will eventually lead to $start$, which will have more than one outbound state.) Also all subranges must contain at least one element, so the iterative generation of subranges converges to a subrange of length 1.

Because the selected subrange length converges to 1, the process has to finish, and when it finishes there is a subrange around a single number (the original input) and a list of states (words). For every input data, there is a final result.

For every number d of size n, this final result can be seen as a path that points to d, as every state in the word sequence tells which subrange to choose from the partitions generated by $subranges$. Using this path intuition, it can be seen that if d_1 and d_2 are two different numbers of the same size n, their encoded texts are necessarily different, as they lead to different numbers. In the same way, the decoding system can find d using the text as a path to the length 1 subrange.

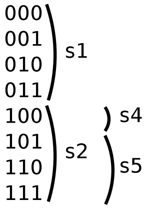

Fig. 2. Example encoding of 100 into $[s_2, s_4]$. This and more examples can be seen in Table 1.

Fig. 2 shows this partitioning process. The example in the figure uses the range $[0, 8]$, with the numbers encoded in binary. If we use the Markov chain shown in Fig. 1 and we start from *start*, in a first step the range has to be split in half, because the probabilities for the two states s_1 and s_2 are both 0.5. The subrange assigned to s_2 can then be split in two other parts, now for the states s_4 and s_5, but the proportions are 0.25 and 0.75 in this case. This shows that if we were to encode the binary number 100, with a fixed size $n = 3$ bits, we would get the text $[s_2, s_4]$. If we were trying to encode the binary number 111, we would need to continue partitioning the range for $s5$, until there is only a single number in the last subrange.

Table 1 shows the output of $encode_{fixed}$ for a number of inputs. The reference implementation [9] was used, and the results may vary in other implementations, depending on specific details of the range partitioning algorithm. All examples use the Markov chain shown in Fig. 1, with "*start*" as the starting state. In particular, it is possible to see that 100 indeed encodes to $[s_2, s_4]$, as described above.

4.3 Decoding of Fixed-size Data Using Markov Chains

Decoding of fixed-size data is based on $subrangeForState$, which was described on Section 4.1. It was previously described as a function that returns the subrange that is assigned to a given state; but it can also be seen as a decoder from states to numbers. In this way, the function $subrangeForState(mc, w_k, r, w_l)$ decodes a single word state w_l, given that the previous state was w_k. The decoded value is not a number, but a range of numbers: $[a, b]$ where both a and b are natural numbers.

Given an input sequence of states or words w_t (where w_0 is taken to be the initial state used for encoding) and an initial range r_0 (typically $[0, 2^n - 1]$) we define the sequence of ranges r_t as:

$$r_t = subrangeForState(mc, w_t, r_{t-1}, w_{t-1}) \tag{11}$$

The output of the $decode_{fixed}$ is the value of the range r_T, where T is the first t such that $length(r_t) = 1$. Since when that happens the range covers a single number, the decoding process can just return that number.

data	n	encoded text
0	1	[s1]
1	1	[s2]
00	2	[s1, s3]
01	2	[s1, s4]
10	2	[s2, s4]
11	2	[s2, s5]
000	3	[s1, s3]
001	3	[s1, s4, s6]
010	3	[s1, s4, s7, start, s1]
011	3	[s1, s4, s7, start, s2]
100	3	[s2, s4]
101	3	[s2, s5, s7]
110	3	[s2, s5, s8, start, s1]
111	3	[s2, s5, s8, start, s2]
0000	4	[s1, s3, start, s1]
0001	4	[s1, s3, start, s2]
0010	4	[s1, s4, s6, start, s1]
0011	4	[s1, s4, s6, start, s2]
0100	4	[s1, s4, s7, start, s1, s3]
0101	4	[s1, s4, s7, start, s1, s4]
0110	4	[s1, s4, s7, start, s2, s4]
0111	4	[s1, s4, s7, start, s2, s5]
1000	4	[s2, s4, s6]
1001	4	[s2, s4, s7]
1010	4	[s2, s5, s7]
1011	4	[s2, s5, s8, start, s1, s3]
1100	4	[s2, s5, s8, start, s1, s4, s6]
1101	4	[s2, s5, s8, start, s1, s4, s7]
1110	4	[s2, s5, s8, start, s2, s4]
1111	4	[s2, s5, s8, start, s2, s5]
00000	5	[s1, s3, start, s1, s3]
11111	5	[s2, s5, s8, start, s2, s5, s8, start, s2]

Table 1. Table of example encodings using $encode_{fixed}$. Two different inputs can encode to the same text only if they have a different length, as happens with 00 and 000. The frequencies of the different bigrams approximate the probabilities $P(s_n|s_{n-1})$ from the Markov chain, and this approximation gets better as n grows (because the space is bigger, and because of the Condition of Minimal Length, which has a higher effect in smaller inputs). Also this table shows a very low capacity, because the Markov chain used is very small. More comments about capacity in Section 6.

A valid output isn't guaranteed for all texts (sequences of words), only for words that have been generated by using the $encode_{fixed}$ process described above.

The decoding process works because it follows the same path that the encoder process followed when generating the text, and this path leads to the original input data. The encoder writes a sequence of words while refining subranges until finding a range that has length 1. The decoding process follows the states written by the encoder, which lead to exactly the same sequence of subranges. This means that $decode_{fixed}$ will reach the input of $encode_{fixed}$, when feed with the output of $encode_{fixed}$. This makes $decode_{fixed}$ acts as the inverse for $encode_{fixed}$, for fixed n.

An additional property of $decode_{fixed}$ as it is defined here is that if $data = decode_{fixed}(text)$, then also $data = decode_{fixed}(text + text_2)$, where $text_2$ is any text and "+" is the list concatenation operation. This is because the fixed decoding algorithm finishes computing the value for $data$ when the last subranges converge to a single number, and that happens at the same place in the text sequence for $text$ and for $text + text_2$.

This property is useful because it allows us to concatenate encoded texts, and they can be decoded directly as the decoder can tell where every text starts and ends. This is applied to the variable encoding algorithm discussed in Section 5.

4.4 Implementation Details

A direct implementation of the algorithms described above would require that many operations are applied to the n bit ranges in every iteration of encoding and decoding. For example, in every iteration of the fixed-size decoding algorithm, a call to $subranges$ needs to be done with a range of numbers with n bits of size, until the length of the selected range is 1 (so that the range matches the original input). This is very inefficient both regarding memory usage and processing time.

It is possible to avoid processing on the full n bits on every iteration, by making some changes to the underlying algorithms. Some data with length n can be processed more efficiently if only a short, moving window of a few bits is processed in every iteration. We define $subranges_{fast}$:

$$subranges_{fast}(mc, s, r_{mbits}, n) = subranges(mc, s, expand(r_{short}, n)) \qquad (12)$$

where $r_{short} = [a, b]$ is defined to be a range where a and b are two numbers that can be expressed in up to m bits, and $expand(range, m, n)$ computes $[a_2, b_2]$, with a_2 identical to a in all its leftmost m bits, and 0 in the remaining bits, and with b_2 identical to b in all its leftmost m bits, and 1 in the remaining bits. This means that we can use $expand$ to convert short ranges like $[01, 10]$ (in binary) to the longer 4 bit range $[0100, 1011]$, if $n = 4$.

An efficient implementation of $subranges_{fast}$ returns all subranges in short form, for any input. When the ranges have to be split in a way that requires infinite or long precision (for example if there are two states, with $P(s_1|s) =$

0.3 and $P(s_2|s) = 0.7$), this is only possible if a precision limit is set in the implementation. This precision limit can be set to mean that regardless of the input of $subranges_{fast}$, there is a maximum number of bits that can be used for the partitioning process.

For example, with $n = 100$ and the probabilities described above, the ranges returned could be: [00000000, 01001101] for s_1, and [01001110, 11111111] for s_2. In this case, s_1 really has about 0.305 of the numbers of the total range, so using 8 of the 100 bits is a good approximation. If we were to use only 4 bits in $subranges_{fast}$ for this case, it would return: [0000, 0100] for s_1, and [0101, 1111] for s_2. In this case s_1 maps to about 0.312 numbers of the total range; this is a slightly worse approximation, but it might be better as it requires using only half the amount of bits.

Both for encoding and decoding, a bit stream data structure will be needed. For encoding, this stream of bits will be read; when decoding, it will be used to write the data output, in a bitwise fashion.

When encoding, in every iteration $subranges_{fast}$ will require a small number of bits to be read from the bit stream. As soon as those bits are read, they can be discarded from the bit stream. Also, $subranges_{fast}$ will generate new ranges in every call, and in every iteration these ranges will be more precise, that is, ranges that cover a smaller amount of numbers. This means that the subranges will require more bits to be stored.

However, if the precision for $subranges_{fast}$ is set to a finite value (as described above), the number of bits at the right of the range that differ from each other will be at most k, for some k. This means that with every iteration, the ranges will grow in size n, but the leftmost bits will at the same time converge bitwise to the same values (for range $[a, b]$, leftmost bits of a and b will be identical). The leftmost bits can then be discarded, as they are already known to match the leftmost bits in the input data.

This process ensures that in every iteration of encoding, $subranges_{fast}$ only has to deal with a moving window that has a limited number of bits, related to the precision set to the system in the implementation.

Similarly for decoding; in very iteration, the range that $subranges_{fast}$ returns will grow in size (as measured in bits). However, while the range grows in size, the leftmost bits converge, so they can be removed, and added to an output bit stream. When the process finishes, the output bit stream will contain the full output of the decoding algorithm: all the bits of the converged range.

5 Variable Size Encoding and Decoding

The encoding and decoding process described above only allows to decode data from a text, given that the size of the data is known beforehand. However, requiring the recipient of a steganographic system to know the size of the hidden data before it is decoded is not optimal. An extension of the encoding and decoding methods for variable-size data solves this problem.

image	chain states	encoding time	decoding time	encoded size	$\frac{\text{encoded size}}{\text{file size}}$
example.zip (12 kB)	unigrams	91.8 s *(0.1 kB/s)*	95.3 s *(0.1 kB/s)*	81 kB *(zip: 32 kB)*	6.7 *(zip: 2.7)*
	bigrams	53.7 s *(0.2 kB/s)*	55.8 s *(0.2 kB/s)*	149 kB *(zip: 58 kB)*	12.4 *(zip: 4.8)*
example.jpg (19 kB) (zip: 12 kB)	unigrams	141 s *(0.1 kB/s)*	150.5 s *(0.1 kB/s)*	119 kB *(zip: 38 kB)*	6.3 *(zip: 3.2)*
	bigrams	93.4 s *(0.2 kB/s)*	97.6 s *(0.2 kB/s)*	216 kB *(zip: 66 kB)*	11.4 *(zip: 5.5)*
example.png (39 kB)	unigrams	299.6 s *(0.1 kB/s)*	317.2 s *(0.1 kB/s)*	269 kB *(zip: 104 kB)*	6.9 *(zip: 2.7)*
	bigrams	194.1 s *(0.2 kB/s)*	181.8 s *(0.2 kB/s)*	494 kB *(zip: 188 kB)*	12.7 *(zip: 4.8)*

All benchmarks were run on a computer with processor Intel Core i7-2670QM CPU at 2.20GHz x 8, with 7.7 GiB RAM.

Samples of the encoded texts:

– Example.zip (unigrams): ”Be limited and secondly because Pierre suddenly realized. Und die and secondly. Monotonous sound of the man who too late. Monsieur Kiril Andreevich nicknamed the hour later grasped the Russian commanders. Dressed for the new building with a year period of the two or an example.”
– Example.zip (bigrams): ”Be a square for fuel and kindled fires there. Secondly it was hard to hide behind the cart and remained silent. He feels a pain in the now cold face appeared that the man continually glanced at her as though they stumbled and panted with fatigue. With a deep.”
– Example.jpg (unigrams): ”He had been her neighbors and friends that these wrinkles and then there’s no lambskin cap and saw that Russian expedition. Under a largish piece of me all all is going on the lot of that moment I have an all four abreast. Having evidently relating to scrutinize the nunnery. Every moment.”
– Example.jpg (bigrams): ”He had something on both sides and. Secondly it was tete a tete. You did me the duty of a month ago. He’s having a good humored amiable smiles. Pierre pointed to a series of actions that follows therefrom. After playing out a passage she had all the forms of town life perished. Tell him Here.”
– Example.png (unigrams): ”Rostov a short fingers and exhausted and the driver a bright lilac dress. And the locomotive by all seemed to the most profitable source of Karataev and secondly. Even remember that it an enormous movements and had to tell you want of the other troops standing. If it’s high.”
– Example.png (bigrams): ”Rostov looked inimically at Pierre and addressing all present and rested on them. But seeing before him. Princess Mary thought only of how Princess Mary for Prince Vasili saw that Platon did not forget what I consider myself bound to Princess Mary will take the covert at once abandoned all their decorations.”

Fig. 3. Example benchmarks and results when running MarkovTextStego [9] with Markov chains generated from War and Peace by Tolstoy. MarkovTextStego uses the method discussed in this article, and an extension that uses bigrams as states in the Markov chain (as discussed briefly in Section 3). The bigram-based encoder will produce higher quality texts, however they will be larger than those produced by the unigram-based encoder.

For variable size encoding and decoding it is required that an integer m is shared beforehand. This number is not the data size, but the size used for a header; it is typically a small value like 16 or 32. Texts c_1 and c_2 are encoded as shown below, using the three arguments version of $encode_{fixed}$.

The header is encoded first, into c_1. This is done using the fixed-data encoding algorithm, with the fixed size m that is known both for encoder and decoder:

$$n = length(data) \tag{13}$$

$$c_1 = encode_{fixed}(n, m, start) \tag{14}$$

Once the header was encoded, the actual data is encoded into c_2. We use w as starting symbol, to ensure that there isn't an interruption in the flow of the generated text between the last symbol in c_1 and the first one in c_2:

$$w = \text{ last word in } c_1 \text{ text sequence} \tag{15}$$

$$c_2 = encode_{fixed}(data, n, w) \tag{16}$$

Finally, $encode(data)$ is defined simply as:

$$encode(data) = c_1 + c_2 \tag{17}$$

That is, the encoded data is just the header text followed by the data text. As w was used as starting symbol for generating c_2, there will be no interruption in the flow between both texts.

For decoding an input $text$, we define:

$$n\prime = decode_{fixed}(text, m, start) \tag{18}$$

That is, $decode_{fixed}$ is used to extract the length information from the header, using the shared value m.

$$text_1 = \text{ list of words used in decoding } n\prime \tag{19}$$

$$text_2 = \text{ list of words not used in decoding } n\prime \tag{20}$$

$$w\prime = \text{ last of } text_1 \tag{21}$$

Finally, $decode$ can be defined:

$$decode(text) = decode_{fixed}(text_2, n\prime, w\prime) \tag{22}$$

It can be seen that when $data\prime = decode(encode(data))$, it follows that: $n\prime = n$, $text_1 = c_1$, $text_2 = c_2$, and $w\prime = w$. For this reason, $data\prime = data$, which means that $decode$ is the right decoding function.

It is also possible to extend *encode*, without changing this last property, in this way:

$$encode(data) = c_1 + c_2 + randomText(z) \tag{23}$$

where z is the last word in c_2, and $randomText(symbol)$ generates a random text that ends in period, using the Markov chain and starting from the given state. This can be used to ensure that all texts generated by *encode* have a final sentence that is complete, and finishes with period. Adding any text won't affect the decoding at all, as explained in Section 4.3.

Depending on the kind of data that is being transmitted, it might be useful to encode into c_1 the length of the data in bytes, instead of encoding it in bits. Also, the way the length is actually represented into bits matters; if big endian is used to represent a multi-byte length into bytes, short encoded lengths will start with a sequence of 0 bits; this could produce the encoded texts to always start with the same words, or with a small variety of different words (because all leftmost bits are zero). For this reason, either little endian or a representation that reverses the bits of big endian would be preferable.

6 Conclusions and Future Research

This article presented a steganographic method based on Markov chains that differs from other similar models in the way precision loss in the language model is avoided. A reference implementation for this method was also presented.

The examples shown in Table 1 could seem to show that the system has very low capacity. However this is only because of the Markov chain used; if the system uses a small Markov chain, it will have low capacity, but if it uses a bigger Markov chain it will typically have a higher capacity.

Preliminary results of empirical tests using a big Markov chain computed from an actual literary text show that the encoded data takes the size of about 6 - 7 times the size of the original data, with an n value that is big enough (for very small n, yet bigger than a few bytes, this factor can be higher, e.g. around 9). Because the produced output is a text, it can be compressed with a high ratio; the compressed size of the texts is about 2 times the size of the original data. However, these results require a more complete and thorough analysis.

Other possibilities for further research are: to combine this method to other known language based steganographic systems, for producing an overall better steganographic text generation method; to analyze what is the actual, measured performance for this new algorithm, and how this new algorithm compares to other existing algorithms, in terms of stegoanalysis.

References

1. Bennett K.: Linguistic Steganography: Survey, Analysis, and Robustness Concerns for Hiding Information in Text. CERIAS Tech Report 2004-13, Purdue University. (2004)

2. Nechta, I., Fionov, A.: Applying Statistical Methods to Text Steganography. CoRR. (2011)
3. Kwan M.: SNOW. `http://www.darkside.com.au/snow/manual.html` (1996)
4. Topkara M., Topkara U., Atallah M.J.: Information Hiding Through Errors, A Confusing Approach. Proceedings of the SPIE International Conference on Security, Steganography, and Watermarking of Multimedia Contents. (2007)
5. Grothoff, C., Grothoff, K., Alkhutova, L., Stutsman, R., Atallah, M.: Translation-Based Steganography. Proceedings of the 2005 Information Hiding Workshop (IH 2005). Paper 1624. (2005)
6. Dai, W., Yu, Y., Dai, Y., Deng, B.: Text Steganography System Using Markov Chain Source Model and DES Algorithm. Journal of Software, vol. 5, issue 7, pp. 785-792. (2010)
7. Dai, W., Yu, Y., Deng, B.: BinText Steganography Based on Markov State Transferring Probability. 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (ICIS '09). (2009)
8. `http://1010.co.uk/markov\_stego.py`
9. Moraldo, H.H.: MarkovTextStego. `https://github.com/hmoraldo/markovTextStego` (2012)
10. Chapman, M.: Hiding the Hidden: a Software System for Concealing Ciphertext as Innocuous Text. Master's thesis, University of Wisconsin-Milwaukee (1997)
11. Siefkes, C.: NL Stego. `http://www.siefkes.net/software/nlstego/` (2005)
12. Siefkes, C.: Natural Language Steganography Based on Statistical Text Generation. `http://www.siefkes.net/software/nlstego/slides/slides-nlstego.sxi` (2005)
13. Manning, C.D., Schütze, H.: Foundations of Statistical Natural Language Processing. The MIT Press. (1999)
14. Russell, S., Norvig, P.: Artificial Intelligence, a Modern Approach. Prentice Hall, 2nd Edition. (2002)
15. Sidorov, M.: Hidden Markov Models and Steganalysis. 2004 workshop on Multimedia and security (MM&Sec '04). (2004)
16. Taskiran, C.M., Topkara, U., Topkara, M, Delp, E.J.: Attacks on Lexical Natural Language Steganography Systems. SPIE International Conference on Security, Steganography, and Water-marking of Multimedia Contents. (2006)