Improving Security and Capacity for Arabic Text Steganography Using 'Kashida' Extensions

By :

Fahd Al-Haidari

Adnan Gutub

Khalid Al-Kahsah

Jamil Hamodi

Collage of Computer Sciences & Engineering
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
fahdhyd@kfupm.edu.sa, gutub@kfupm.edu.sa

Outlines:

- Introduction
- Background and related works
- Our approach and methodology
- Results and discussion
- Conclusion
- Q & A

Introduction

- Steganography
  - What is it?
  - What is the different between Steg. & Crypt. ?
  - Steg. Applications.
- Steganography mean issues
  - Security
  - Capacity
  - Robustness

Background

- Arabic language features:
  - Dots
  - Diacritics
  - Connectivity

Related works

- Doted letters
  - **Shifting dots up.** [Shirali-Shahreza & others, 2006]
    - negative robustness
    - depends on using the same fixed font
- Kashida
  - With doted letters. [A. Gutub & others, 2007]
    - solves the problem of the negative robustness
    - font independent
    - Less security
    - Less capacity

Related works

- Diacritics
  - One Diacritic. [M. Aabed & others, 2007]
    - high capacity
    - implemented easily
    - sending Arabic message with diacritics might raise suspicions nowadays
    - Arabic font has different encodings on different machines (computer dependant)
  - Multiple Diacritics. [Adnan Gutub & others, 2008]
    - High capacity
    - More than one scenario

Proposed approach

- Idea
- Algorithm
- Examples
- Implementation

Proposed approach

- The main idea
- Create a coding system based on:
  - locations of extendable characters
  - inserted Kashidahs
- E.g.
  - With 3 possible loactions
  - using at most 3 Kashidahs:
  - → Can represent 8 values (3 bits) .
- In general, the block size can be given as:

$$blocksize = \log_2 \sum_{m=0}^{k} \binom{m}{n}$$

values	3	2	1
0			
1			-
2		-	
3	-		
4		-	-
5	-		-
6	-	-	
7	-	-	-

Proposed approach

Algorithm 1 Hiding secret bits

Input: message, cover text
Output: stego text
initialize *k* with maximum Kashidas used per word.
while data left to embed **do**
  get next *word* from cover text
  n ← the number of possible extendable letters.
  if *n* > 0 **then**
    s ← calculate the block size based on *n* and *k*.
    get the value of the next *s*-bit message block.
    determine the *positions* that represent the value.
    insert Kashida into the *word* at *positions*.
    insert the *word* into stego text.
  end if
end while

Proposed approach

- Algorithm

Algorithm 2 Extracting secret bits

Input: stego text
Output: message
initialize *k* with maximum Kashidas used per word.
while word left to process **do**
  get next *word* from stego text
  n ← the number of possible extendable letters.
  if *n* > 0 **then**
    s ← calculate the block size based on *n* and *k*.
    determine the *positions* of existed Kashidas.
    get the *value* represented by *positions*
    get the *s*-bits block representing the *value*.
    insert the *s*-bits block into message.
  end if
end while

Proposed approach

- Example hide secret bits
- maximum Kashidas
  - K=3
- The word = “يسمع” ,
- secret bits=“10100”

- Block size
  - N=3 , extendable letters
  - possible values=1+3+3+1=8
  - Block_size, s =log 8 =3
- The value
  - 3-bits block= (100) , value =4
- The positions
  - Positions=table[N,value] = {1,2}

- Insert Kashida at position {1,2}
- Stego_text=“يسمع”

Table[pk,x]	list
[3,0]	0
[3,1]	1
[3,2]	2
[3,3]	3
[3,4]	1,2
[3,5]	1,3
[3,6]	2,3
[3,7]	1,2,3

Proposed approach

- Example retrieve the hidden bits

- Stego_text=“يسمع”
- maximum Kashidas
  - K=3
- Block size
  - N=3 , extendable letters
  - possible values=1+3+3+1=8
  - Block_size, s =log 8 =3
- Existed kashidas
  - Positions → {1,2}
- Extracted bits
  - Value ← 4 , From the table
  - 3 bits ← 100

Table[pk,x]	list
[3,0]	0
[3,1]	1
[3,2]	2
[3,3]	3
[3,4]	1,2
[3,5]	1,3
[3,6]	2,3
[3,7]	1,2,3

Proposed approach : implementation

```
for (int i = 0; i < split.Length; i++)
        {
          k = num_kasheda(split[i], k_positions);
          // get the proper block size
          switch (k)
           {
              case 1: case 2: case 3
                 block_size = k;
                 break;
             case 4: case 5: case 6: case 7: case 8:
                 block_size = k-1;
                 break;
           }
          // get the  block of secret bits
          if (hide_p >= hide.Length) { finished = true; break; }
          if ((hide.Length - hide_p) < block_size) block_size = hide.Length - hide_p;
          hide_block = hide.Substring(hide.Length - hide_p - block_size, block_size);
          hide_p += block_size; // advance the pointer in hiden text
          // get the corresponding value of the block
          value = get_value(hide_block);
          if (value != 0)
          {   // insert Kashidahs into  K_positions
              split[i] = insert_kasheda_3(split[i], k, k_positions, value);

          }
}
```

Proposed approach : implementation

```
for (int i = 0; i < split.Length; i++)
        {
          pk = num_kasheda(split[i], k_positions);
          ek = num_exist_kasheda(split[i], k_positions);
       // the block size
           switch (pk)
              {
                 case 1: case 2: case 3:
                    block_size = pk;
                    break;
                 case 4: case 5: case 6: case 7:
                    block_size = pk-1;
                    break;
              }

           value = 0;
       // locate the  coding table   at k_position, return the value
           for (int x1 = 0; x1 <= Math.Pow(2, block_size); x1++)
           {
                bool ok = true;
                for (int i1 = 0; i1 < ek; i1++)
                    if (table[pk, x1].list[i1] != k_positions[i1]) ok = false;
                if (ok)
                {
                   value = x1;
                   break;
                }
           }

       // convert it into binary bits
           int x=0;
           for (int j = 0; j < block_size; j++)
           {
              x = value >> j;
              x = x & 1;
             hide_block = x.ToString() + hide_block;
           }
}
```

Table[pk,x]	list
[3,0]	0
[3,1]	1
[3,2]	2
[3,3]	3
[3,4]	1,2
[3,5]	1,3
[3,6]	2,3
[3,7]	1,2,3

Results & discussion

- Capacity
- Security

Results & discussion

- Capacity
- The capacity can be evaluated
  - by the capacity ratio
    - dividing the amount of hidden bytes over the size of cover text
  - useable characters ratio.
    - using p for the ratio of characters capable of baring ‘1’
    - and q for the ratio of characters capable of baring ‘0’.
    - p=q, assuming that 50% of secret bits are ‘1’
    - Thus, usable characters ratio =(p+q)/2.

Results & discussion

Table 1 Capacity ratio of our proposed approach

Approach	Cover Size	Capacity (%)	Average capacity (%)
One Kashidah	70627	2.45	2.46325
	131233	2.465	
	235125	2.4526	
	525271	2.4854	
Two Kashidahs	70627	3.285	3.303293
	131233	3.2916	
	235125	3.2612	
	525271	3.37537	
Three Kashidahs	70627	3.73139	3.73715
	131233	3.71447	
	235125	3.7165	
	525271	3.78624	

Table 2 Reported Capacity Ratio in Literature

Approach	Cover Size	Capacity (%)	Average capacity (%)
Kashidah [1]	365181	1.215	1.22
	378589	1.172	
	799577	1.266	
	15112	1.244	
Diacritics [5]	318,632	3.25	3.27
	1,34,865	3.256	
	717,135	3.318	
	318,216	3.254	
Dots [3]	13619.2	1.172	1.37
	6983.68	1.467	
	6799.36	1.275	
	3604.48	1.529	

- The results show that capacity ratio increases when the number of Kashidah per word increases.
- The scenario of using three Kashidahs outperforms the other scenarios in our approach.
- Also, it outperforms the other existing approaches in literature in terms of capacity

Results & discussion

Table 3 Usable Characters Ratio of Our Approach

File size (bytes)	P	q	r	(p+r+q)/2
70627	0.39.18	0.3918	0	0.3918
131233	0.39	0.39	0	0.39
235125	0.3864	0.3864	0	0.3864
525271	0.392355	0.392355	0	0.392355

Table 4 Reported Usable Characters Ratio in Literature

Approach	p	q	r	(p+r+q)/2
Dots	0.2764	0.4313	0.0300	0.3689
Kashidah-Before	0.2757	0.4296	0.0298	0.3676
Diacritics	0.3633	0.3633	0	0.3633

- The results show that our proposed approach outperforms the other approaches in terms of usable characters ratio.
- Our proposed approach shows a ratio of about 0.39, where as, the others show a ratio of about 0.36 as it is shown in Table 4.

Results & discussion

- For the security, using a lot of extensions within a word in the cover text media, will reduces the security.
- In [1], one extension kashidah may represent one bit.
- E.g. the word "سنمتعهم" ; secret bits as "001010".
  - 6 extension kashidahs will be inserted in this word,
  - "سنمـتـعـهـم".
- our proposed approach improves this issue by restricting the inserted Kashidahs per word.
- E.g. Using at most three Kashidahs:
  - 2 kashidahs will be inserted at position 1,5 to hide (01010)
  - "سنمتعهم"

Conclusion

- A novel algorithm for text steganography
  - in Arabic language
  - and other similar Semitic languages.
- Implemented different scenarios
  - Using at most One, two, and three Kashidahs.
- Evaluated and compared in terms of capacity to the existing Arabic text steganography in literature.
  - It was superior in terms of capacity.
  - showed more security than the existing Kashidah approach