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

The main idea
Create a coding system based on:
- locations of extendable characters
- inserted Kashidahs

E.g.
- With 3 possible locations
- using at most 3 Kashidahs:
  - Can represent 8 values (3 bits).

In general, the block size can be given as:

\[
blocksize = \log \sum_{n=0}^{m} \binom{m}{n}
\]

<table>
<thead>
<tr>
<th>values</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
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 full message block. |
| determine the positions that represent the value. |
| insert Kashidas into the word at positions. |
| insert the word into stego text. |
| end if |
| end while |

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 $k$. |
| get the value of the next full message block. |
| 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 = "يسعم"
  - secret bits = "1010010100"

- Block size
  - $N=3$, extendable letters
  - possible values $= 1+3+3+1=8$
  - $Block\_size$, $s = \log 8 = 3$

- The value
  - 8-bit block (100), value = 4

- The positions
  - Positions = table[secret_bits] = (1, 2)

- Insert Kashida at position {1, 2}
- Stego_text = "ـسـيـعم"

---

Proposed approach

- Example: retrieve the hidden bits
  - Stego_text = "ـسـيـعم"
  - 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>
<thead>
<tr>
<th>Table: (x, k)</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>[3, 0]</td>
<td>0</td>
</tr>
<tr>
<td>[3, 1]</td>
<td>1</td>
</tr>
<tr>
<td>[3, 2]</td>
<td>2</td>
</tr>
<tr>
<td>[3, 3]</td>
<td>3</td>
</tr>
<tr>
<td>[3, 4]</td>
<td>1, 2</td>
</tr>
<tr>
<td>[3, 5]</td>
<td>1, 3</td>
</tr>
<tr>
<td>[3, 6]</td>
<td>2, 3</td>
</tr>
<tr>
<td>[3, 7]</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>
Proposed approach : implementation

```c
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.Length - hide_p) < block_size) block_size = hide.Length - hide_p;
    if (hide_p >= hide.Length) finished = true; break;
    // 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

```c
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;
    }
}
```

---

Proposed approach : implementation

```c
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

The capacity can be evaluated by dividing the amount of hidden bytes over the size of cover text.

- The capacity can be evaluated
  - by the capacity 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 \( = (p+q)/2 \).
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.

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 “سنتمتهم” ; secret bits as “001010”:
  - 6 extension kashidahs will be inserted in this word, “سنتمتهم”.
  - 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, to hide (01010)

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