Running compression algorithms in the encrypted domain: a case-study on the homomorphic execution of RLE

Donald Nokam Kuate
Sebastien Canard
Sergiu Carpov
Renaud Sirdey

28 avril 2017
Agenda

Homomorphic encryption
  Definition
  Examples and applications

Armadillo compiler
  Infos compiler
  General structure

Run-Length Encoding (RLE)
  Definition RLE
  Regularization of RLE
  Improvements and results
Homomorphic encryption (HE)
Homomorphic encryption (HE)

HF1: homomorphic concat & scale function

Homomorphic evaluation

HF1: homomorphic concat & scale function
Homomorphic encryption (HE)

F1: concatenation & scale function
HF1: homomorphic concat & scale function
Clear evaluation
Homomorphic evaluation
Encryption
Decryption

F1: concatenation & scale function
HF1: homomorphic concat & scale function
### Examples and applications of HE

#### Examples

- **partially HE** ($\infty +$ or $\infty \times$):
  - RSA, ElGamal, Paillier
- **somewhat HE** ($\infty +$ and finite $\times$):
  - YASHE, FV, BGV
- **fully HE** ($\infty +$ and $\infty \times$): first time define in 2009 by Craig Gentry who introduces bootstrapping. Fully HE = somewhat HE+bootstrapping.
Examples and applications of HE

Examples

- partially HE ($\infty$ “ + ” or $\infty$ “ $\times$ ”): RSA, ElGamal, Paillier
- somewhat HE ($\infty$ “ + ” and finite “ $\times$ ”): YASHE, FV, BGV
- fully HE ($\infty$ “ + ” and $\infty$ “ $\times$ ”): first time define in 2009 by Craig Gentry who introduces bootstrapping. Fully HE = somewhat HE + bootstrapping.

Applications

- cloud computing;
- electronic voting;
- video transcoding and image processing.
Agenda

Homomorphic encryption
  Definition
  Exemples and applications

Armadillo compiler
  Infos compiler
  General structure

Run-Length Encoding (RLE)
  Definition RLE
  Regularization of RLE
  Improvements and results
Armadillo compiler

- Compiler developed by CEA;
- developed in C++;
- use the FV homomorphic scheme, but other HE scheme too;
- the main operations are $+$, $\times$, and $x = c?a:b$
General structure of Armadillo

Figure – General structure of Armadillo
Agenda

Homomorphic encryption
  Definition
  Examples and applications

Armadillo compiler
  Infos compiler
  General structure

Run-Length Encoding (RLE)
  Definition RLE
  Regularization of RLE
  Improvements and results
RLE

Is a **lossless data compression algorithm**, which consists in transforming a sequence of symbols where some symbols have a **consecutive repetition**, in a sequence of (symbol, counter) much shorter.

Example

The sequence `LLLLLUUUUKKKKKKKKEEEEEEEE` is transformed in `(L, 7), (U, 4), (K, 8), (E, 10)`.

Applications

▶ loss or lossless image compressing (BMP, JPEG);
▶ MPEG and H26x video compressing.
RLE

Is a lossless data compression algorithm, which consists in transforming a sequence of symbols where some symbols have a consecutive repetition, in a sequence of (symbol, counter) much shorter.

Example

The sequence LLLLLLUUUUKKKKKKKKKKEEEEEEEEEEE is transformed in (L, 7), (U, 4), (K, 8), (E, 10).
RLE

Is a **lossless data compression algorithm**, which consists in transforming a sequence of symbols where some symbols have a **consecutive repetition**, in a sequence of (symbol, counter) much shorter.

**Example**

The sequence LLLLLLUUUUKKKKKKKKKEEEEEEEEEE is transformed in (L, 7), (U, 4), (K, 8), (E, 10).

**Applications**

- loss or lossless image compressing (BMP, JPEG);
- MPEG and H26x video compressing.
A pseudo-code of RLE is:

```c
01. int main(void) {
02.    int n_chars;
03.    char *input;
04.    cin>>n_chars;
05.    input=new char[n_chars];
06.    assert(input);
07.    for(int i=0;i<n_chars;i++)
08.       cin>>input[i];
09.    for(int i=0;i<n_chars;)
10.       int j=0;
11.       while(i+j<n_chars && input[i+j]==input[i])
12.          j++;
13.       cout<<j<<" "<<input[i]<<endl;
14.       i+=j;
15.   }
16. }
```
Steps of regularization

- make incrementation of i constant and transform the while loop;

07. `for(int i=0; i<n_chars; i++)`
08. `cin >> input[i];`
09. `for(int i=0; i<n_chars; ) {
  10.    int j=0;
  11.    while(i+j<n_chars && input[i+j]==input[i])
  12.       j++;
  13.    cout << j << " " << input[i] << endl;
  14.    i+=j;
  }
16. }`
Steps of regularization

- make incrementation of i constant and transform the while loop;

```cpp
07. for(int i=0;i<n_chars;i++)
08.     cin>>input[i];
09. int i,j=1;
10. for(i=1;i<n_chars;i++) {
11.     if(input[i]!=input[i-1]) {
12.         cout<<j<<" "<<input[i-1]<<endl;
13.         j=1;
14.     }
15. else
16.     j++;
17. }
18. cout<<j<<" "<<input[i-1]<<endl;
19. }
```

RLE-1
RLE regularization for Armadillo

Steps of regularization

- condition the counter j;

```cpp
07. for(int i=0; i<n_chars; i++)
08.   cin>> input[i];
09. int i,j=1;
10. for(i=1; i<n_chars; i++) {
11.   if(input[i] != input[i-1]) {
12.     cout<< j <<" " << input[i-1] << endl;
13.     j=1;
14.   }
15.   else
16.     j++;
17. }
18. cout<< j <<" " << input[i-1] << endl;
19. }

RLE-1
RLE regularization for Armadillo

Steps of regularization

▶ condition the counter j;

```cpp
07. for(int i=0; i<n_chars; i++)
08.     cin>>input[i];
09.    int i,j=1;
10.   for(i=1; i<n_chars; i++) {
11.     if(input[i]!=input[i-1])
12.       cout<<j<<" "<<input[i-1]<<endl;
13.       j=input[i]!=input[i-1]?1:j+1;
14.   }
15.   cout<<j<<" "<<input[i-1]<<endl;
16. }
```
RLE regularization for Armadillo

Steps of regularization

- condition the outputs;
  - set a fix rate of compression;
  - referencement on encrypted indeces;
for(int i=0; i<n_pairs; i++) {
    output_chr[i] = 'a';
    output_ctr[i] = 0;
}

int i,j=1,k=0;
for(i=1; i<n_chars; i++) {
    for(int l=0; l<n_pairs; l++) {
        output_ctr[l] = l!=k ? output_ctr[l];
        output_chr[l] = l!=k ? output_chr[l];
    }
    k = input[i] != input[i-1] ? k+1 : k;
    j = input[i] != input[i-1] ? 1 : j+1;
}
for(int l=0; l<n_pairs; l++) {
    output_ctr[l] = l!=k ? output_ctr[l];
    output_chr[l] = l!=k ? output_chr[l];
}
for(int i=0; i<n_pairs; i++)
    cout << output_chr[i] << " " << output_chr[i] << endl;
RLE regularization for Armadillo

```cpp
... for(int i=0; i<n_pairs; i++) {
  output_chr[i] = 'a';
  output_ctr[i] = 0;
}

int i, j = 1, k = 0;
for(i = 1; i < n_chars; i++) {
  for(int l = 0; l < n_pairs; l++) {
    output_chr[l] = l != k ? output_chr[l] : input[i-1];
  }
  k = input[i] != input[i-1] ? k + 1 : k;
  j = input[i] != input[i-1] ? 1 : j + 1;
}
for(int l = 0; l < n_pairs; l++) {
  output_chr[l] = l != k ? output_chr[l] : input[i-1];
}
for(int i = 0; i < n_pairs; i++)
  cout << output_ctr[i] << " " << output_chr[i] << endl;
}
```

RLE-3
# First results

<table>
<thead>
<tr>
<th># symbols</th>
<th># pairs</th>
<th>depth</th>
<th>times (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>22</td>
<td>7.20</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>28</td>
<td>21.0</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>75</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table** – Depth of homomorphic RLE in function of number of symbols and number of output pairs on 2 core cpu and 3.00GHz machine
First results

<table>
<thead>
<tr>
<th># symbols</th>
<th># pairs</th>
<th>depth</th>
<th>times (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>22</td>
<td>7.20</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>28</td>
<td>21.0</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>75</td>
<td>-</td>
</tr>
</tbody>
</table>

Table – Depth of homomorphic RLE in function of number of symbols and number of output pairs on 2 core cpu and 3.00GHz machine

Remarks

- $k = \text{input}[i] == \text{input}[i-1] ? k : k+1$ (line 23)
  - # symbols = 10 → depth = 17
  - # symbols = 64 → depth = 70
- $j = \text{input}[i] == \text{input}[i-1] ? j+1 : 1$ (same as for k)
First improvement

Change line

```
k = input[i] == input[i-1] ? k : k+1 by
k+1 if input[i] != input[i-1];
```

the depth down to 8 for 10 symbols and to 10 for 64 symbols
**Improvements**

**First improvement**

Change line

```c
k = input[i] == input[i-1] ? k : k+1 by
k+=input[i] != input[i-1];
```

the depth down to 8 for 10 symbols and to 10 for 64 symbols

**Second improvement**

Change line

```c
j = input[i] == input[i-1] ? j+1 : 1 by
j = 1 + j&(input[i] != input[i-1]) and remark that if we set
b_i = input[i] != input[i-1] then
```

\[ j = 1 + \sum_{i} \prod_{m=l}^{i} b_m. \]

The depth down to 11 for 10 symbols and to 14 for 64 symbols
Second results

<table>
<thead>
<tr>
<th># symbols</th>
<th># pairs</th>
<th>depth(old)</th>
<th>depth (new)</th>
<th>new times(min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>22</td>
<td>14</td>
<td>2.30</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>28</td>
<td>20</td>
<td>10.45</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>75</td>
<td>63</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table** – Second results: depth of homomorphic RLE in function of number of symbols and number of output pairs on 2 core cpu and 3.00GHz machine.

Remark: The algorithm depth is reduced, but still not yet enough. This is due to lines:

```
output_chr[l]=(k==l) ?input[i-1] :output_chr[l]
```
(line 20);

```
output_ctr[l] = (k==l) ?j :output_ctr[l]
```
(line 21);
Second results

<table>
<thead>
<tr>
<th># symbols</th>
<th># pairs</th>
<th>depth(old)</th>
<th>depth (new)</th>
<th>new times(min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>22</td>
<td>14</td>
<td>2.30</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>28</td>
<td>20</td>
<td>10.45</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>75</td>
<td>63</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table** – Second results: depth of homomorphic RLE in function of number of symbols and number of output pairs on 2 core cpu and 3.00GHz machine

**Remark**

The algorithm depth is reduced, but still not yet enough. This is due to lines

```c
output_chr[l]=(k==l) ?input[i-1] :output_chr[l] (line 20);
output_ctr[l] = (k==l) ?j :output_ctr[l] (line 21);
```
Third improvement

The lines 20 and 21 have the same structure which is of the form
\[ c_i^{(i)} = (l == k^{(i)}) \cdot x_{i-1} : c_i^{i-1} \]

When we develop, we obtain the following expression

\[
c_i^{(i)} = c_0 + (c_0 + x_{i-1}) b_i^{(i)}
+ \sum_{j=1}^{i-1} (c_0 + x_{j-1}) b_i^{(j)} \left( 1 + \sum_{K \in \mathcal{P}(j+1, \ldots, i)} \left( \prod_{u \in K} b_i^{(u)} \right) \right)
\]

where \( b_i^{(i)} = (l == k^{(i)}) \), \( c_0 = c_i^{(0)} \).

This development allows us to drop the depth to the theoretical value of \( 12 + \log_2 (N + 1) \), \( N = \text{sequence length} \).
Conclusion

- **DO**
  - regularized and executed the RLE algorithm in the homomorphic domain;
  - improved its depth;

- **TO DO**
  - try to reach this theoretical depth;
  - improved the other block from either side of RLE in video compressing.
Thanks!

Questions?