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

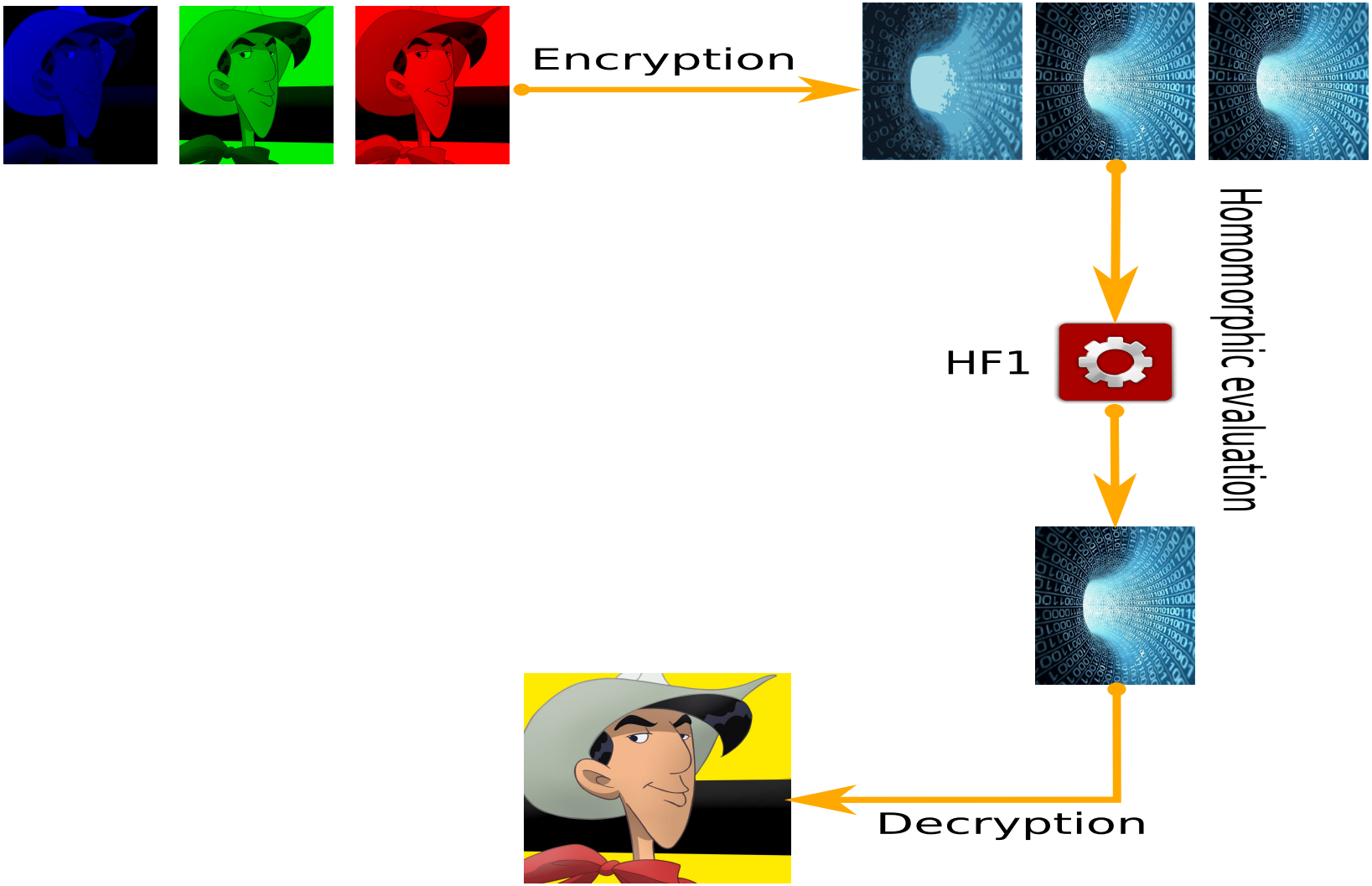
Agenda

Homomorphic encryption (HE)

Homomorphic encryption (HE)

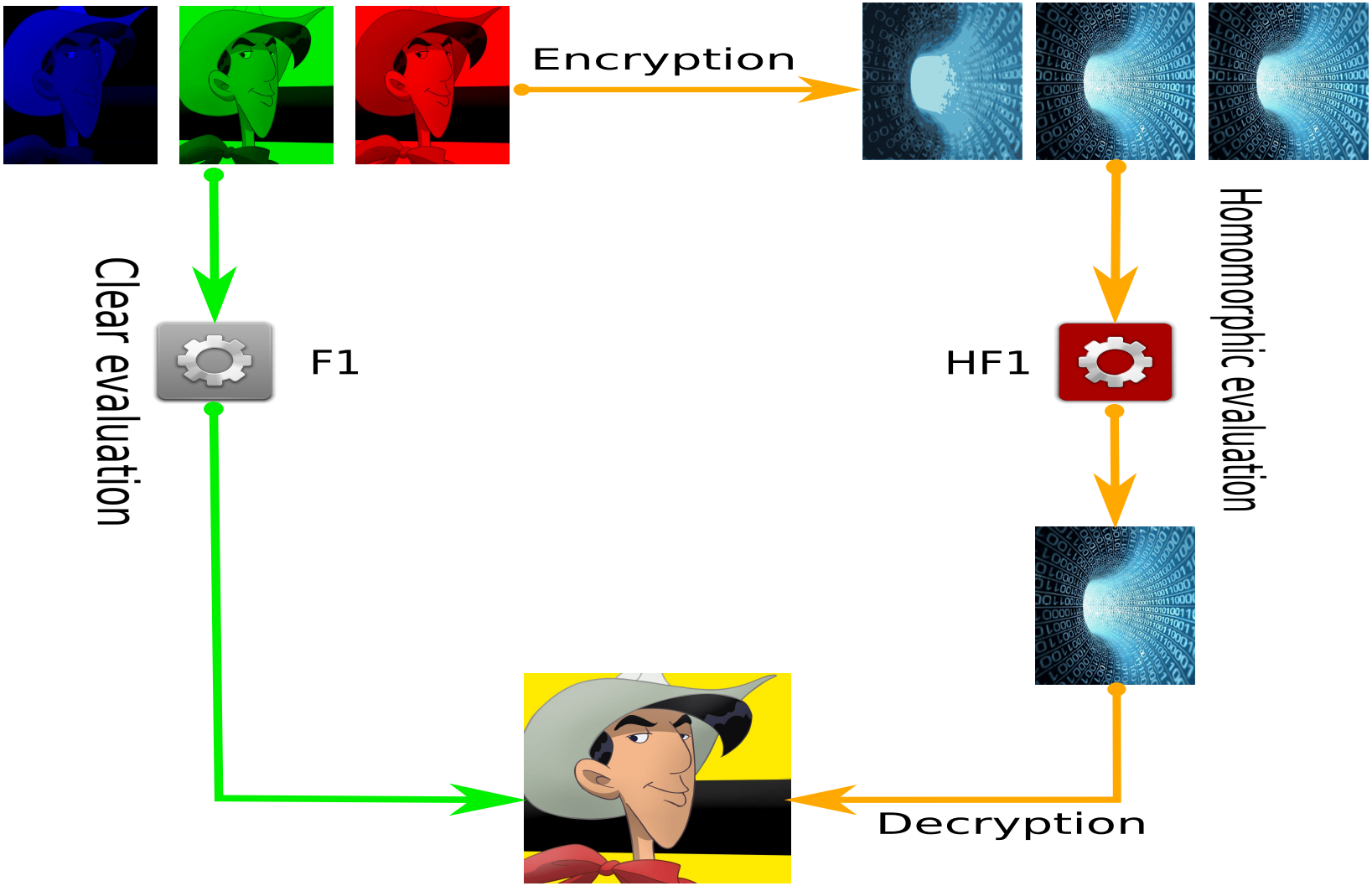
HF1: homomorphic concat & scale function

Homomorphic encryption (HE)



HF1: homomorphic concat & scale function

Homomorphic encryption (HE)



F1: concatenation & scale function
HF1: homomorphic concat & scale function

Examples and applications of HE

Examples

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

Examples and applications of HE

Examples

- partially HE (∞ "$+$" or ∞ "$\times$") :
  - RSA, ElGamal, Paillier
- somewhat HE (∞ "$+$" and finite "$\times$") :
  - YASHE, FV, BGV
- fully HE (∞ "$+$" and ∞ "$\times$") : 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

Armadillo compiler

Armadillo

- 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

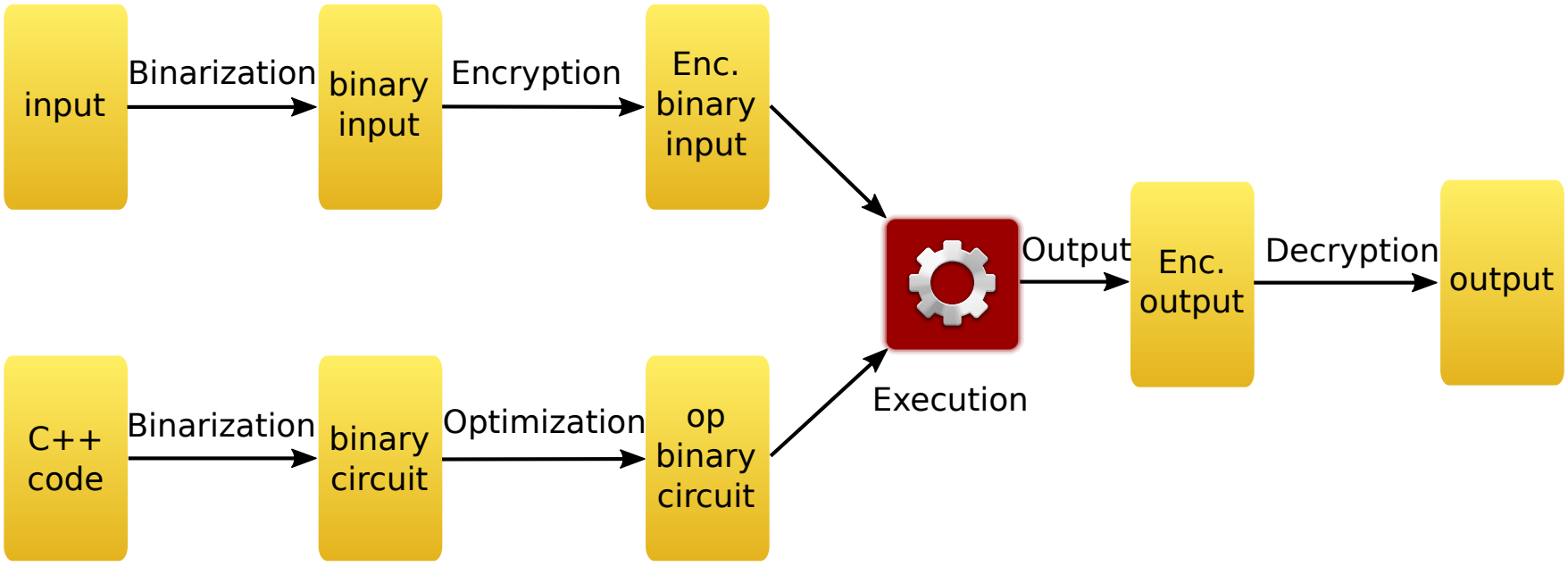


Figure – General structure of Armadillo

Agenda

Homomorphic encryption
- Definition
- Exemples and applications

Armadillo compiler
- Infos compiler
- General structure

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

RLE

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.

RLE

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 LLLLLLLUUUUKKKKKKKKEEEEEEEEEE is transformed in (L, 7), (U, 4), (K, 8), (E, 10).

RLE

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 LLLLLLLUUUUKKKKKKKKEEEEEEEEEE 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 regularization for Armadillo

A pseudo-code of RLE is :

```
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. }
```

RLE regularization for Armadillo

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;
15.  }
16. }
```

RLE regularization for Armadillo

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.   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 ;

```
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 ;

```
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 ;

RLE regularization for Armadillo

```
13.   for(int i=0;i<n_pairs;i++) {
14.     output_chr[i]='a';
15.     output_ctr[i]=0;
16.   }
17.   int i,j=1,k=0;
18.   for(i=1;i<n_chars;i++) {
19.     for(int l=0;l<n_pairs;l++) {
20.       output_ctr[l]=l!=k?output_ctr[l]:j;
21.       output_chr[l]=l!=k?output_chr[l]:input[i-1];
22.     }
23.     k=input[i]!=input[i-1]?k+1:k;
24.     j=input[i]!=input[i-1]?1:j+1;
25.   }
26.   for(int l=0;l<n_pairs;l++) {
27.     output_ctr[l]=l!=k?output_ctr[l]:j;
28.     output_chr[l]=l!=k?output_chr[l]:input[i-1];
29.   }
30.   for(int i=0;i<n_pairs;i++)
31.     cout<<output_ctr[i]<<" "<<output_chr[i]<<endl;
32. }
```

RLE-3

RLE regularization for Armadillo

```
13.  for(int i=0;i<n_pairs;i++) {
14.    output_chr[i]='a';
15.    output_ctr[i]=0;
16.  }
17.  int i,j=1,k=0;
18.  for(i=1;i<n_chars;i++) {
19.    for(int l=0;l<n_pairs;l++) {
20.      output_ctr[l]=l!=k?output_ctr[l]:j;
21.      output_chr[l]=l!=k?output_chr[l]:input[i-1];
22.    }
23.    k=input[i]!=input[i-1]?k+1:k;
24.    j=input[i]!=input[i-1]?1:j+1;
25.  }
26.  for(int l=0;l<n_pairs;l++) {
27.    output_ctr[l]=l!=k?output_ctr[l]:j;
28.    output_chr[l]=l!=k?output_chr[l]:input[i-1];
29.  }
30.  for(int i=0;i<n_pairs;i++)
31.    cout<<output_ctr[i]<<" "<<output_chr[i]<<endl;
32. }
```

output_ctr[k] = j
output_chr[k] = input[i-1]

RLE-3

First results

First results

# symbols	# pairs	depth	times(min)
10	5	22	7.20
16	5	28	21.0
64	8	75	-

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

First results

# symbols	# pairs	depth	times(min)
10	5	22	7.20
16	5	28	21.0
64	8	75	-

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 = intput[i] == intput[i-1] ?k :k+1 (line 23)
  - # symbols = 10 $\rightarrow$ depth = 17
  - # symbols = 64 $\rightarrow$ depth = 70
- j = intput[i] == intput[i-1] ?j+1 :1 (same as for k)

Improvements

First improvement

Change line
`k = intput[i] == intput[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

Improvements

First improvement

Change line
k = intput[i] == intput[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
j = intput[i] == intput[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

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

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

Second results

# symbols	# pairs	depth(old)	depth (new)	new times(min)
10	5	22	14	2.30
16	5	28	20	10.45
64	8	75	63	-

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

Second results

# symbols	# pairs	depth(old)	depth (new)	new times(min)
10	5	22	14	2.30
16	5	28	20	10.45
64	8	75	63	-

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) ;

Improvement

Third improvement

The lines 20 and 21 have the same structure which is of the form
$c_l^{(i)} = \left(l == k^{(i)}\right) ? x_{i-1} : c_l^{i-1}$
When we develop, we obtain the following expression

$$\begin{aligned} c_l^{(i)} &= c_0 + (c_0 + x_{i-1}) b_l^{(i)} \\ &\quad + \sum_{j=1}^{i-1} (c_0 + x_{j-1})\, b_l^{(j)} \left(1 + \sum_{\mathcal{K} \in \mathcal{P}(j+1,\ldots,i)} \left(\prod_{u \in \mathcal{K}} b_l^{(u)}\right)\right) \end{aligned}$$

where $b_l^{(i)} = \left(l == k^{(i)}\right), c_0 = c_l^{(0)}$.

This development allow us to drop the depth to the theoretical value of $12 + \log_2(N+1)$, $N =$ sequence length.

Conclusion

- DO
  - **regularized** and **executed** the RLE algorithm in the **homomorphic domain** ;
  - improved its depth ;
- TO DO
  - try to **reach** this **theoretical detph** ;
  - improved the **other block** from either side of RLE in **video compressing**.

Thanks !

Questions ?