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

Sebastien Canard[1], Sergiu Carpov[2], Donald Nokam Kuate[1,2], and Renaud Sirdey[1]

[1] Orange Labs, Applied Crypto Group, France
[2] CEA, LIST, France

**Abstract.** This paper is devoted to the study of the problem of running compression algorithms in the encrypted domain, using a (somewhat) fully homomorphic encryption (FHE) scheme. We do so with a particular focus on conservative compression algorithms. Despite of the encrypted domain Turing-completeness which comes with the magic of FHE operators, we show that a number of subtleties crop up when it comes to running compression algorithms and, in particular, that guaranteed conservative compression is not possible to achieve in the FHE setting. To illustrate these points, we analyze the most elementary conservative compression algorithm of all, namely Run-Length Encoding (RLE). We first study the way to regularize this algorithm in order to make it (meaningfully) fit within the constraints of a FHE execution. Secondly, we analyze it from the angle of optimizing the resulting structure towards (as much as possible) FHE execution efficiency. The paper is concluded by concrete experimental results obtained using the Fan-Vercauteren cryptosystem as well as the Armadillo FHE compiler.

## 1 Introduction

In the realm of algorithms, data compression ones are particular beasts. As everyone knows, at least in loose form, such an algorithm is expected, given an input $x$, to turn it into an output $y = \alpha(x)$ such that $y$ is much smaller than $x$ and that there exists $\alpha^{-1}$ such that $\alpha^{-1}(y) = x$ (in this paper we will consider only lossless compression[1]). In essence, and there are deep reasons why this is so, any data compression algorithm fails to achieve its goal on almost all of its possible inputs, most often generating $y$'s which are much larger than the corresponding $x$'s.

In this paper, we study the way such compression algorithm can be executed when the input data are encrypted, using fully homomorphic encryption (FHE).

[1] Lossy compression would be modeled as requiring the existence of a decompression algorithm $\beta$ such that $\beta(y)$ would be close enough to $x$ according to some criterion which may even be subjective as in audio or video coding.

It should be emphasized here that running compression algorithms in such encrypted domain is an important topic for the longer-term applications of FHE techniques. Indeed, stream transcoding (e.g., audio or video codec or resolution changes) in the encrypted domain would, if practical, unleash a number of very interesting applications.

Take for example the "Hello world!" of data compression which we are going to study in depth in this paper: the Run-Length Encoding (RLE) algorithm. Given an input sequence of symbols, this algorithm straightforwardly outputs a sequence of pairs counter/symbol which compresses symbols repetitions. Although this algorithm fails to compress straightforward non random sequences, e.g., of alternating symbols, it works well-enough in certain practical contexts such as low color depth image coding (e.g., as in the BMP format) or quantized DCT coefficient coding (as in most video coding standards, H264 ahead). Of course, there are many more sophisticated algorithms which avoid this kind of pitfalls and work quite well on more general purpose data.

In fact, one can define a universal (yet not computable) compression algorithm to output the smallest Turing machine able to output a given sequence[2]. But such algorithm still runs into the same kind of difficulties as the framework of Kolmogorov complexity theory precisely tells us that most strings are incompressible (that term being synonymous to random w.r.t. that theory). In essence, any compression algorithm has a very bad worst-case behavior and, *when ran in the encrypted domain using FHE*, any algorithm intrinsically realizes its worst-case behavior. The contribution of this paper is a first attempt to reconcile these two antagonistic points of view. Despite of its simplicity, the study of the RLE algorithm is rich enough for illustrating our points as well as drawing general conclusions.

The paper is organized as follows. In Sect. 2, after briefly reviewing the capabilities and limitations of the "FHE machine", we derive a regularized variant of the RLE algorithm which fits the constraints of that machine. Then, in Sect. 3, we make some attempt at effectively running several variants of this algorithm, more and more optimized towards "FHE friendlyness". In doing so, we try to point out general (both manual and automatic) optimization techniques and best practices for the "FHE programmer". Lastly, Sect. 5 concludes the paper with some remarks and perspectives.

## 2 Regularization of RLE

We now propose to study in depth, in the case of its execution in the encrypted domain, the most elementary compression algorithm there is: the RLE algorithm (still, it is practically used in a number of image formats, e.g., the BMP format,

[2] This universal compression algorithm would be able to compress strongly encrypted data as, unless another more compact encoding exists, it could at least be able to retrieve the cleartext data, the key and the encryption algorithm, then output a small Turing machine outputting the cleartext data as well as an another small Turing machine for the encryption step.

as well as for coding the quantized DCT coefficients in most video coding formats). The RLE algorithm simply consists, given a sequence of symbols $\alpha_1$, $\alpha_2$, ..., in producing a sequence of pairs counter/symbol which compresses symbols repetitions. For example, the sequence 0, 0, 2, 3, 3, 3, 4, 0, 0, 0, 0 would be turned into $\{2, 0\}$, $\{1, 2\}$, $\{3, 3\}$, $\{1, 4\}$, $\{4, 0\}$. Despite of its simplicity, this algorithm cannot be implemented per se to run homomorphically. We shall now look into the reasons as to why it is so and, most importantly, into how and to which extent an implementation (compliant with an encrypted domain execution) is yet possible in principle.

### 2.1 Operational capabilities of the FHE computer

When it works on $n$-bits integers (i.e., with $\mathbb{Z}_2$ as the cleartext domain), the FHE machine allows to perform all usual operators: addition, multiplication, sign change, subtraction, left and right shift (with or without sign extension) and so on. The reader is referred to [5] for details on how to implement such operators.

It is when performing comparisons, or more precisely when trying to use the results of comparisons between encrypted-domain values, that the true essence of the FHE machine is revealed. With the aforementioned operator, it is easy to create an operator $<: \mathbb{Z}_{2^n} \times \mathbb{Z}_{2^n} \longrightarrow \mathbb{Z}_2$ which enables any other comparison operator ($>$, $\geq$, $\leq$, $==$, ...). The subtlety is of course that the result of the evaluation of such an operator remains inaccessible to the computer running the algorithm. In particular, this means that our machine *can* evaluate any condition depending on encrypted-domain data but *cannot* exploit these results for example in order to influence the control flow of the program it is running!

Hence, the FHE machine cannot per se execute an "if ... then ... else" construct with a condition depending on encrypted-domain data, at least not directly. Also, it cannot directly execute a loop structure with a termination criterion depending on a data in the encrypted domain. More generally, relatively to encrypted-domain data, the FHE machine can only execute so-called static control structure programs (i.e., programs which can always be turned into a linear sequence of instructions).

Still, a number of regularization techniques do exist in order to turn dynamic control structure programs into static ones. As an example, for the "if ... then ... else" construct, all is needed is a conditional assignment operator (e.g. the C language :? operator) which can be implemented as

$$x := c?a : b \equiv x := c \otimes a \oplus (1 \oplus c) \otimes b.$$

One can then execute both branches of the construct and then appropriately recombine the results according to the condition value using the latter operator, without learning any information about the (encrypted) values $x$, $a$, $b$ and $c$.

In summary, we have to work with an unusual machine allowing only regularized conditional structures and loops with an a priori bounded number of iterations as well as having the property that algorithms always realize (at least) their worst case complexity (relatively to a "classical" machine) — keeping in mind that the running time of an algorithm on the FHE machine cannot *by construction* vary in terms of the encrypted data.

## 2.2 Regularization of the algorithm control flow

As a starting point, Fig. 1-RLE-0 provides some C-style pseudo-code of the usual implementation of the RLE algorithm (in the clear domain). As already hinted, this implementation is not compliant with the constraints of the FHE machine: the loop of line 9 progresses by variable increments which depend on the number of iterations of the loop of line 11 which stopping criterion depends on data which would be encrypted in an homomorphic execution (`input`). So let us attempt to regularize this algorithm in a step by step fashion. For this purpose, we give our intermediate results on Fig. 1-RLE-1 and Fig. 2-RLE-2 respectively. The final description we propose for the RLE algorithm is then given in Fig. 2-RLE-3 and will be explained in the next section.

The first step (Fig. 1-RLE-1) consists in getting rid of the non-constant increments of the loop index `i`. This is done at line 10 of RLE-1. Still, this requires to put the output of `j` (the symbol counter) and the associated symbol in an `if` statement which condition depends on the (encrypted) input sequence (either the symbol changes and the algorithm prints the symbol counter and the associated symbol or it does not change and the symbol counter is just incremented). Then, the assignment of `j` (lines 13 and 16 of RLE-1) can be regularized by means of a conditional assignment as discussed in section 2.1. This has been done at line 13 of the version RLE-2 (Fig. 2), in which we now focus.

So far, no significant restructuring of the algorithm has been necessary. Unfortunately, the core difficulties stand in the `if` statement at line 11 of RLE-2. We have seen that in order to regularize an `if` statement we have to (schematically) execute (unconditionally) both branches and then recombine their effects, using conditional assignments, consistently with the *encrypted* result of the evaluation of the condition. In particular, this means that going through either branches of an `if` must be indistinguishable in terms of effect[3]. For the `if` in question, the effect is to print something when the condition is satisfied and not to print anything otherwise (i.e., different effects on the program output). Hence, in order to have a similar effect whatever the branch we have no other choices than either:

– systematically print nothing (which is not very useful); or
– systematically print something but ensuring that the kind of "something" which is printed when there is nothing to print is indistinguishable from the kind of "something" which is printed when there is something to print.

This latter property is given us by the (semantic) security of the underlying cryptosystem. For example, we could print an (encrypted) zero for the counter value followed by an arbitrary symbol (this would be transparent to the RLE

[3] Whether we want it or not the "API" of the homomorphic machine and its underlying encryption system does not, by definition, let us do otherwise.

```
int main(void) {
  int n_chars;
  char *input;
  cin>>n_chars;
  input=new char[n_chars];
  assert(input);
  for(int i=0;i<n_chars;i++)
    cin>>input[i];
  for(int i=0;i<n_chars;) {
    int j=0;
    while(i+j<n_chars && input[i+j]==input[i])
      j++;
    cout<<j<<" "<<input[i]<<endl;
    i+=j;
  }
}
```

**RLE-0**

```
int main(void) {
  int n_chars;
  char *input;
  cin>>n_chars;
  input=new char[n_chars];
  assert(input);
  for(int i=0;i<n_chars;i++)
    cin>>input[i];
  int i,j=1;
  for(i=1;i<n_chars;i++) {
    if(input[i]!=input[i-1]) {
      cout<<j<<" "<<input[i-1]<<endl;
      j=1;
    }
    else
      j++;
  }
  cout<<j<<" "<<input[i-1]<<endl;
}
```

**RLE-1**

**Fig. 1.** Pseudo-code for variants of the RLE algorithm, from the usual implementation towards a variant which fits the constraints of an homomorphic execution. Would-be encrypted variables are in lightgray. See text.

```
int main(void) {
  int n_chars;
  char *input;
  cin>>n_chars;
  input=new char[n_chars];
  assert(input);
  for(int i=0;i<n_chars;i++)
    cin>>input[i];
  int i,j=1;
  for(i=1;i<n_chars;i++) {
    if(input[i]!=input[i-1])
      cout<<j<<" "<<input[i-1]<<endl;
    j=input[i]!=input[i-1]?1:j+1;
  }
  cout<<j<<" "<<input[i-1]<<endl;
}
```

**RLE-2**

```
int main(void) {
...
  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]:j;
      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_ctr[l]=l!=k?output_ctr[l]:j;
    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**

**Fig. 2.** Pseudo-code for variants of the RLE algorithm, from the usual implementation towards a variant which fits the constraints of an homomorphic execution. Would-be encrypted variables are in lightgray. See text.

decoder). This would mean replacing the if in question (lines 11 and 12 of RLE-2) by something like:

```
cout<<(input[i]!=input[i-1]?j:0)<<" "<<input[i-1]<<endl;
```

Of course, the issue is that (by construction) we do not compress anything anymore as the output stream is now systematically twice larger than the input one! As such, none of the above options is thus satisfactory.

## 2.3 Regularization of the algorithm outputs

Starting from the version given by RLE-2, let us now derive a meaningful "FHE-friendly" variant of the algorithm, which is described in Fig. 2-RLE-3.

In order to compress, we have to make the algorithm works with an *a priori given* compression rate and, as a consequence, relax another characteristic of the algorithm: its conservativeness. In order to do so, we have to work with a constant number of pairs counter/symbol both during the inner working of the algorithm as well as in its outputs. Then, the resulting variant (Fig. 2-RLE-3) needs an additional parameter, n_pairs, which allows to fix the number of (encrypted) pairs which are to be systematically outputted by the algorithm. All of these pairs are initialized to 0 (for the counter) and an arbitrary symbol (lines 13 to 16 of RLE-3). Then the compression step is run (described just after) and the algorithm prints all the n_pairs (encrypted) pairs counter/symbol. Then, regardless of its input, the algorithm always outputs the same number of (encrypted, indistinguishable from one another) pairs. Two scenarios are then possible:

1. the a priori fixed number of pairs is sufficient to represent the input sequence in which case the reconstruction of the sequence by the decoder will be correct but it can be that a number of pairs were unused (0 counter) and then that we have compressed less than possible (with RLE),
2. otherwise, the decoder will not be able to reconstruct the correct input sequence and some form of padding will be required for the decoder to at least output a sequence of correct length.

Some compression is achieved when

$$(\alpha + \beta) \times \mathsf{n\_pairs} < \alpha \times \mathsf{n\_chars}$$

where $\alpha$ and $\beta$ are the number of bits for respectively representing the symbols and the counters.

The compression step (lines 17 to 29 of RLE-3) works as follows. Integer k is initially set to 0 (or, more precisely, to an encryption of 0), this counter points to the current working pair in the pairs buffer. Then, we execute the loop on the input sequence symbols (line 18). The for loop of line 19 corresponds to an array assignment operator *with an encrypted index* (it is a loop because the complexity of such an assignment is linear in the array length as discussed in Sect. 2.1). Functionally, this loop is equivalent to doing output_ctr[k]=j and output_chr[k]=input[i-1] but with an encrypted k. Then, k and j are respectively incremented on symbol change and non-change by means of conditional assignments (lines 23 and 24). Lastly, once the loop of line 18 is done, the loop at line 26 is another (the last) assignment to the pairs buffer (still with encrypted k), this is the "equivalent" of the last print instruction of RLE-2 (line 15).

As an example, in order to avoid that the need for padding has an impact on the decoder, it is possible, still within the constraints of an homomorphic execution, to (i) count the number of symbols $\ell$ that is accounted for the output pairs (i.e., to sum, in the encrypted domain, the entries in output_ctr) and (ii) update the last counter (i.e., output_ctr[n_pairs-1]) in order to repeat the last symbol of the input sequence $\mathsf{n\_chars} - \ell$ more times. That way, it is the encoder which drives the padding which has sometimes to be done by the decoder. Other more complex strategies are of course possible, but they will remain lossy. It is also possible to add a flag to the compressed stream, indicating whether or not compression was lossless, and letting the decoder taking appropriate actions in either case. Of course, a simple sufficient condition for the compression to have been lossless is when the last counter is 0. This condition is most likely good enough for all practical purposes.

As a conclusion to this section, we hint at a more general negative result which forbids any form of lossless compression in the encrypted domain. Indeed, as long as a compression algorithm admits some difficulties to compress input sequences, then such a sequence will not fit within an a priori fixed memory budget smaller than its length. From a theoretical viewpoint, we know that there exists sequences which are not compressible for all compression algorithms — in fact almost all sequences are so. It follows that for any compression algorithm, when run homomorphically, either we (always) do not compress and we are lossless or we compress (at fixed rate) and we are lossy!

### 2.4 Remarks on RLE decoding

Note that, as far as running the decoder in the encrypted domain is concerned, there is no difficulty *in principle* as long as the number of pairs in the (compressed) input stream as well as the number of symbols in the (uncompressed) sequence are publicly known. Under the non straightforward assumption that a RLE decoder can be run efficiently in the encrypted domain, this technique (running a RLE encoder on the clear data before they are sent to the FHE computer and then homomorphically running an RLE decoder) might interestingly complement other available techniques for homomorphic ciphertext compression (e.g., [1, 3]) especially when very sparse data structures are sent such as in Private Information Retrieval-like protocols.

## 3 Experimental results

In the previous section, we have defined a variant of the RLE algorithm which can *in principle* be run homomorphically. Having done so, we are only half-way and we now turn to the issue of concrete refactoring and optimization of this variant to make it run homomorphically *as efficiently as possible*.

All the results in this paper have been obtained using the Fan-Vercauteren homomorphic scheme [4] (FV for short) and the parameters were computed as described by the authors. The scheme was dimensioned to assure 128-bits of security and the minimal multiplicative depth needed by the executed boolean circuits.

## 3.1 The Armadillo compiler

In our experiments, we have used the Armadillo FHE compiler[4] [2]. The Armadillo compilation chain is an easy to use environment which turns C++ programs into optimized boolean circuits which can then be executed homomorphically with the Armadillo FHE runtime environment. The compilation chain is composed of 3 layers: a front-end, a middle-end and a back-end. The front-end transforms code written in C++ into a boolean circuit representation. The middle-end layer optimizes the boolean circuit produced by the front-end using ABC[5] tuned for FHE-oriented circuit optimizations. The back-end then either constructs a binary (linked with a concrete FHE library) which homomorphically executes the boolean circuit or relies on the Armadillo RTE (which is essentially a FHE boolean circuit interpreter) to do so. In both cases, OpenMP is intensively used which allows to obtain almost linear speedups on SMP machine (say up to 100-200 cores). Two HE are presently supported by Armadillo: (i) an in-house implementation of [4] and (ii) the publicly available library HElib [6, 7]. The constraints of the FHE machine (Sect. 2.1) are enforced via the programming model which is exposed by the compiler (C++ types supporting only those operators compliant with the FHE constraints).

The Armadillo compiler is very useful when it comes to prototyping a given algorithm for homomorphic execution, allowing the programmer to have a short feedback loop when trying FHE-oriented optimizations.

## 3.2 A first trial

Before starting to run the RLE-3 variant (Fig. 2-RLE-3) which is not a "simple" algorithm (with respect to present homomorphic standards), we wanted to start with something a little bit more elementary. So our first test consisted in, given an input sequence of symbols, to output a sequence of booleans flagging the symbol changes. With the Armadillo compiler, we write the following seemingly standard code:

```
  Integer8 input[NUM_OF_CHARS];
```

[4] At present, the Armadillo compiler can be made available as part of research collaborations. Also, an open source release is under preparation at the time of writing.

[5] ABC (www.eecs.berkeley.edu/alanmi/abc) is a well-known open source System for Sequential Synthesis and Verification.

```
for(int i=0;i<NUM_OF_CHARS;i++)
  cin>>input[i];
for(int i=0;i<NUM_OF_CHARS-1;i++)
  cout<<(input[i]==input[i+1]);
```

Above, Integer8 is a type provided by the compiler environment for declaring encrypted integer variables, here on 8 bits. The NUM_OF_CHARS constant defines the number of symbols in the input stream. It should be emphasized that this latter constant must be known at compile-time in order to be able to build a (by definition static) boolean circuit. Note also that, in practice, it is not necessarily an issue. For example, in the context of a video coder, the RLE algorithm is executed only on a few (standardized) macro-block sizes (e.g. 4x4, 8x8, 16x8, 8x16, 16x16). Thanks to C++ operator overloading capabilities, our Integer8 variables are manipulated like usual integers. For example, the >> operator expresses a reading operation (in fine of encrypted bits stored in files which location will be provided to the runtime environment), the == operator allows to test the equality of two variables of type Integer8 (the result being and *encrypted* boolean) and the << operator will trigger the ouput of the Integer8 variables (in fine of encrypted bits stored in files which location will be provided to the RTE). Of course, there are many more operators, but our point is not to describe them exhaustively in this paper.

Letting the above program through the compiler gives a multiplicative depth 3, independently of the input sequence length (ABC is not able to optimize it further). This is in line with our expectations since this depth is due only to the == operator, implemented in Armadillo as an arborescent product of the bits $x_i \oplus y_i \oplus 1$ of $x$ and $y$ (depth 3 for 8 bits, then). Running this program with FV ($\lambda = 128$ and depth 3) is done in 0.270 s on the average 8 cores machine used throughout this paper unless otherwise stated. This corresponds to 1.726 s of *cumulated* computing time over the 8 cores of the machine (hence a speedup of 6 was obtained by the compiler *transparently* to the programmer), a duration which is dominated (without surprise) by the execution of the 63 homomorphic multiplications (AND gates) in fine required to execute the algorithm.

After this first test, let us move on to RLE-3.

## 3.3 Porting RLE-3

In order to start, we have simply rewritten RLE-3 (as in Fig. 2-RLE-3) using the API of Armadillo. Essentially, this means that we have declared all the supposedly encrypted variables in RLE-3 as Integer8 (that is input[NUM_OF_CHARS], j, k as well as output_chr[NUM_OF_PAIRS] and output_ctr[NUM_OF_PAIRS]) and that we have used the construction select(c,a,b) in lieu of c?a:b as the ?: construct cannot be overloaded in C++. For completeness, this latter code is provided in appendix A.

This time, it is not only the number of input symbols (NUM_OF_CHARS), but also the number of output pairs (NUM_OF_PAIRS) which must be a priori known. As discussed in Sect. 2, this is an intrinsic constraint as we can compress in the encrypted domain but only with a fixed rate. As an example, for 10 symbols and 5 pairs, passing that program through Armadillo gives a multiplicative depth of 22 — which does not appear prohibitive — but as we shall later see, this depth is not independent of the input sequence length. Still, in the same conditions as in the preceding section, the resulting program runs in around 3 min 41 s (taking into account a speedup of 7.75 (which is almost optimal for an 8 cores machine). This running time is dominated (99.7%) by the execution of the 1323 AND gates in the resulting boolean circuit. So far so good, but scaling to longer input sequences requires more work.

Before turning to the optimizations which will allow us to better scale to longer input sequences, let us first illustrate one of the benefits in using the Armadillo compiler. In order to do so, let us replace the following code (equivalent to lines 23 and 24 of RLE-3):

```
k=select(input[i]==input[i-1],k,k+1);
j=select(input[i]==input[i-1],j+1,Integer8(1));
```

by

```
Integer8::Bit b=input[i]==input[i-1];
k=select(b,k,k+1);
j=select(b,j+1,Integer8(1));
```

In the second above code snippet, the (FHE) programmer explicitly avoids to evaluate twice input[i]==input[i-1]. Still, when programming in C or C++ (with optimization options activated) that same programmer would expect this kind of elementary optimizations to be taken care of automatically by the compiler (and to be able to preserve the first of the above two snippets which is easier to read). As a far as the Armadillo compiler chain is concerned, this is indeed the case. In the first case, the compiler front-end outputs a larger boolean circuit, but *the same circuit* as in the second case is obtained once the ABC optimization step is applied.

## 3.4 Multiplicative depth analysis

Another way of benefiting from a compiler infrastructure is that it eases the characterization of programs or algorithms. Using Armadillo, we have thus attempted to get some insights on the behavior of the multiplicative depth of RLE in terms of both the input sequence length as well as the number of output pairs. Table 1 summarizes our results from a "toy example" with 10 symbols and 5 pairs up to a first "realistic" example with a 64 symbols input sequence compressed onto 8 pairs. The latter would correspond to e.g., the compression of an $8 \times 8$ macro-block (i.e., 64 bytes) with a 25% rate ($8 \times 2$ bytes). As such, it appears that the multiplicative depth is essentially driven by the input sequence length and that, running RLE-3 (in its raw form) on a sequence of more than 20 symbols appears irrealistic (we consider, from our experience, that a multiplicative depth of 30 is the threshold above which an homomorphic execution becomes prohibitive in time and memory). So we need to find ways of smashing that multiplicative depth down.

| | 10 | 20 | 30 | 40 | 50 | 60 | 64 |
|---|---|---|---|---|---|---|---|
| 1 | 24 | 34 | 44 | 54 | 64 | 74 | 78 |
| 2 | 21 | 31 | 43 | 53 | 61 | 71 | 75 |
| 5 | 22 | 32 | 42 | 52 | 62 | 72 | 76 |
| 8 | 21 | 31 | 41 | 51 | 61 | 73 | 75 |

**Table 1.** Sampling the multiplicative depth of raw RLE-3 (basic ABC optimization included). Each line gives for a given number of output pairs, the evolution of the multiplicative depth in terms of the input sequence length.

## 3.5 Optimization of the incrementation of k and j

Let us now focus on the following two lines:

```
(1) k=select(input[i]==input[i-1],k,k+1);
(2) j=select(input[i]==input[i-1],j+1,Integer8(1));
```

where input[i]==input[i-1] is at multiplicative depth 3 (Sect. 3.2).

If we then start to work with a program which computes (and outputs) the final value of k, then we indeed explain part of the multiplicative depth of our algorithm as we have a depth of 17 for an input sequence of length 10 and a depth of 67 for an input sequence of length 60. In the former case, the homomorphic running time is of around 20 secs on 8 cores.

The main issue with line (1) above is that there is a cumulative effect on k in the sense that the next value for k will be either the preceding value of k (or that preceding value incremented by 1) multiplied by an encrypted bit (the result of the condition evaluation, at depth 3). Hence, we accumulate 3 levels of multiplicative depth per loop iteration (except at the beginning since, as k is initialized to 0, a public constant, the first loop iteration will involve some hybridized cleartext/ciphertext calculations[6]).

In order to “break” this (multiplicative depth) accumulation, we can then rewrite our line as:

```
k+=input[i]!=input[i-1];
```

Although there still is an accumulation, of course, but this accumulation now occurs through an 8-bits adder and this does not result in an accumulation

[6] I.e. using simplified homomorphic operations when one of the operand is in clear form as: $[x] + 0 = [x]$, $[x] + 1 = [x] + [1]$, $[x] \times 0 = 0$ and $[x] \times 1 = [x]$.

w.r.t. the multiplicative depth[7]. Having done so, we end up with a depth of 10 ($= 3 + 7$ the sum of the depths of a comparison operator and of an adder on 8 bits) and the running time of the program computing only k gets down to 4 secs. Additionally, if we report this modification in RLE-3, the overall multiplicative depth (for an input sequence of length 10) gets down to 18 and the algorithms runs 1.7 times faster.

Line (2) above is more difficult to handle. Indeed, as for k, a program which computes and outputs the last value of j has a multiplicative depth dependent on the input sequence length (e.g., 17 for 10 symbols and 67 for 60) and, if the technique we used for k would allow to handle the incrementation part of j update, its conditional reset cannot be dealt with in the same way.

Let us first start by replacing line (2) above by:

```
j=Integer8(1)+j&(input[i]==input[i-1]);
```

which leads to a slightly better depth (without, however, breaking the dependency on the input sequence length).

Then, the leap of faith consists in considering that it may be interesting to perform *many* redundant computations (albeit with the hope of doing them at much lower depth) and look at the successive values of j produced during the execution of the loop.

Let $b_1$ =input[1]==input[0], ..., $b_i$ =input[i]==input[i-1], ...

Let also $j_0 = 1$. We then have,

$$
\begin{aligned}
j_1 &= 1 + j_0 b_1 = 1 + b_1 \\
j_2 &= 1 + j_1 b_2 = 1 + (1 + b_1) b_2 = 1 + b_2 + b_1 b_2 \\
j_3 &= \dots
\end{aligned}
$$

And, more generally,

$$
j_i = 1 + \sum_{l=1}^{i} \prod_{m=l}^{i} b_m, \text{ with term of highest degree } (i), \prod_{m=1}^{i} b_m.
$$

This latter term may then be evaluated with depth $\log_2 i$ by means of an arborescent product.

Hence, if we accept to proceed with a number of redundant multiplications, the calculation of $j_i$ can be done with depth $3+7+\log_2 i$ since we need to account for the depth of the $b_i$'s (3) as well as that of the 8-bits adder (7). Example code is provided in appendix B for completeness.

[7] Writing down the equations of an $n$-bit adder reveals an invariant w.r.t. the multiplicative depth: summing two "fresh" inputs leads to a result for which the $i$-th power bit is at multiplicative depth $i$, and summing two numbers with that property lead to a results also with that property. As a result, we can additively accumulate over $\mathbb{Z}_2^n$ with a multiplicative depth independent of the number of (encrypted) values which are summed.

When we observe the previous computation of $\mathsf{k}$ and $\mathsf{j}$, it is noted that by carrying out the additions like this: $\mathsf{k} += \mathsf{input[i]!= input[i\text{-}1]}$ and $\mathsf{j} += \prod_{m=l}^{i} b_m$; this invites us to use directly the 8-bits adder while we handle in the first time the binary values. The use of this 8-bits adder increases systematically the depth of 7. To avoid this, we will change this manner to make addition, and replace it by *arborescent addition*. This new way of adding will allow us to use a adder of $\log_2(i)$ instead of 8-bits. Thanks to this technique, we can reduce the depth of $\mathsf{k}$ from 10 to $3 + 1 + \log_2(i), 1 < i < 64$, and to 10 if $i \geq 64$; and that of $\mathsf{j}$ from $3 + 7 + \log_2(i)$ to $4 + \log_2(i) + \log_2(i-2), 2 < i < 66$ and to $10 + \log_2(i)$ if $i \geq 66$.

Thanks to this optimization, the depth of the program *computing only* $\mathsf{k}$ *and* $\mathsf{j}$ decreases from $\approx 20$ down to 11 (for an input sequence of length 10) and from $\approx 70$ down to only 16 for a more realistic input sequence length of 64 (corresponding, as said, to an $8 \times 8$ macroblock). The amount of computation redundancy could be further optimized.

So far, we have thus shown that a program for computing the last value of $\mathsf{k}$ and $\mathsf{j}$ is amenable to a reasonably low multiplicative depth. Although the impact of integrating these optimizations within the whole RLE algorithm is not negligible (execution time decreases from 2 min 9 s down to 1 min 8 s in the same conditions as before), the overall depth of the algorithm still remains problematic due to the output_chr and output_ctr array assignments (with encrypted index $\mathsf{k}$) which are the last residual "hotspot".

## 3.6 Optimization of the output arrays assignments

After working on the computation of $\mathsf{j}$ for optimization of the RLE algorithm, time has come to focus on the following loop:

```
for(int l=0;l<NUM_OF_PAIRS;l++)
{
  output_chr[l]=select(k==l,input[i-1],output_chr[l]);
  output_ctr[l]=select(k==l,j,output_ctr[l]);
}
```

which updates both the character table output_chr and the corresponding index counter table output_ctr.

To optimize this loop, we were inspired from the approach used for the previous calculation of $\mathsf{j}$: namely, expanding every line of code of this loop in order to express it as a function of the boolean factors $\mathsf{l{=}{=}k}$. Before beginning, we observe that the two lines of the loop, output_chr[l] and output_ctr[l], follow the same pattern. Then, we will use the generic expression,

$$c_l^{(i)} = \mathsf{select}\left(\mathsf{l{=}{=}k}^{(i)}, x_{i-1}, c_l^{(i-1)}\right),$$

with $i \in \{1, \ldots, \mathsf{NUM\_OF\_CHARS}\}$, $l \in \{0, \ldots, \mathsf{NUM\_OF\_PAIRS}\}$, and $c_l^{(0)} = c_0$, to carry out our expansion. For $i$ from 1 to $\mathsf{NUM\_OF\_CHARS}$, set the value

$k^{(i)} = \mathsf{select}\left(\mathsf{input[i]{==}input[i\text{-}1]}, k^{(i)}, k^{(i)}+1\right)$, with $k^{(0)} = 0$, and for $l$ from 1 to NUM_OF_PAIRS, set $b_l^{(i)} = \left(\mathsf{l} == \mathsf{k}^{(i)}\right)$. With these notations, we can write:

$$c_l^{(i)} = b_l^{(i)} x_{i-1} + \left(1 + b_l^{(i)}\right) c_l^{(i-1)} = c_l^{(i-1)} + b_l^{(i)} \left(x_{i-1} + c_l^{(i-1)}\right).$$

By recursively expanding this expression, and then factoring, we get the following general term,

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

where $\mathcal{P}(j+1,\ldots,i)$ is the power set of $\{j+1,\ldots,i\}$.

With this last equation, we can note that the term with highest degree of $b_l^{(i)}$ is the term for which $j = 1$ and $\mathcal{K} = \{2,\ldots,i\}$, or, in other words:

$$(c_0 + x_0) \prod_{u=1}^{i} b_l^{(u)}.$$

Again, this term can be evaluated with a multiplicative depth of $\text{depth}(b_l^{(i)}) + \log_2(i+1) = 12 + \log_2(i+1)$ if we carry out the multiplication in a tree-like manner as before. Using the last equation, we observe that we can obtain the values of output_chr[l] by replacing "$c_0$" by "1" and "$x_i$" by "input[i]". To obtain the values of output_ctr[l], we replace "$c_0$" by "0" and "$x_i$" by "$j_i$" for every $i$. The advantage of this formula is that we can reuse the results of the computations of the products $\prod_u b_l^{(u)}$ to compute both output_chr[l] and output_ctr[l].

## 4 Towards automatic multiplicative depth optimization

### 4.1 Heuristic boolean circuit rewriting

We have seen previously that there is a huge amount of work needed for by-hand optimization of non-straightforward boolean circuits multiplicative depth. As one can expect this will be the case for many other applications and not only the RLE algorithm. In this section, as a matter of perspective, we briefly present a heuristic for minimizing the multiplicative depth of boolean circuits which we are developing now as part of the Armadillo middle-end. Let us define the *direct multiplicative depth* of a node in a boolean circuit as the length of the longest path starting from circuit inputs to this node. Equivalently, the *reverse multiplicative depth* is the length of the longest path from this node to circuit outputs. The nodes for which these two values coincide are called *critical nodes*. The *critical circuit* contains all the critical nodes of a boolean circuit.

It is straightforward to see that optimizing critical circuit paths allows to minimize the overall multiplicative depth of a boolean circuit. We introduce a rewrite operator which when applied to the critical circuit will potentially decrease the original circuit multiplicative depth. The idea behind this operator is to rewrite critical paths of multiplicative depth 2 in such a way that its multiplicative depth decreases. A critical path of multiplicative depth 2 is a path in the critical circuit which starts and ends with AND gates and contains zero or more XOR gates in between[8]. Let $((((a_1 \& a_2) \oplus b_1) \oplus \ldots) \oplus b_m) \& a_3$ be the formula for a critical path of multiplicative depth 2 with $m$ intermediate XOR gates. Using XOR gate distributivity and AND gate associativity rules this path can be rewritten as $(((a_1 \& (a_2 \& a_3)) \oplus (b_1 \& a_3)) \oplus \ldots) \oplus (b_m \& a_3)$. Under certain conditions on the multiplicative depth of nodes $a_i$ the global multiplicative depth of the circuit will decrease.

The multiplicative depth minimization heuristic chooses, using a simple priority based on path length (shorter paths are privileged), a path of multiplicative depth 2 and rewrites it. The heuristic stops when no more critical paths, which minimize the global multiplicative depth, are available.

### 4.2 Experimental results

Here, our objective was to perform a test of running RLE-3 with a compression rate of 25% on a sequence of 64 values, representing an $8 \times 8$ macroblock. Thus with a budget of 16 bytes, which corresponds to 8 pairs symbol/counter.

Running our heuristic on RLE-3 circuit leads to the following results. We start from a circuit of depth $\approx 70$ and $\approx 12000$ AND gates to get to a circuit of depth 23 but with $\approx 60000$ AND gates. The multiplicative depth has therefore been smashed down but at the cost of almost 5 times more AND gates. Still, this is cost-effective: when we where simply not able to execute the initial depth-70 circuit, we have been able to execute the new depth-23 circuit in around 30 mins (2003 secs) on a 48 cores computer (taking into account a speedup of 47.3, yet again almost linear and obtained transparently through the use of Armadillo).

Fig. 3 provides the evolution of the depth and the number of AND gates during the successive iterations of the rewriting heuristic. Of course, if for some algorithm the optimum performances are not reached for the minimum multiplicative depth (and it experimentally appears to maximize the number of AND gates), the intermediary circuit which realizes the best multiplicative depth/AND gates trade-off can be kept.

## 5 Conclusion and future work

In this paper, we have tried to tackle the issue of running a compression algorithm over encrypted data, using homomorphic encryption. As already pointed at, running compression algorithms in the encrypted domain is an important topic for the longer-term applications of FHE techniques. In fact, stream transcoding in

[8] Without loss of generality we suppose that the boolean circuits contain only AND and XOR gates.

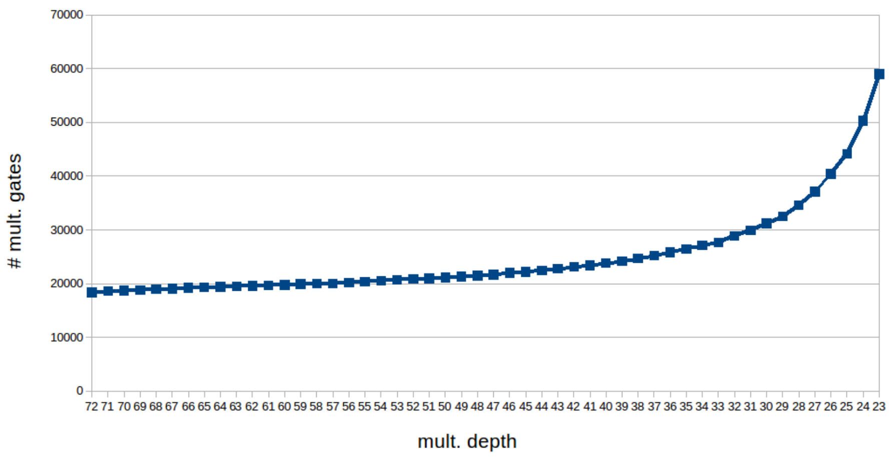

**Fig. 3.** Multiplicative depth vs number of AND gates during the execution of the circuit rewriting heuristic. See text.

the encrypted domain would, if practical, unleash a number of very interesting applications. In this work, and although we cannot claim to have achieved practical performances, we have tried to avoid sweeping under the rug the intricacies we had to effectively deal with in order to regularize and optimize the algorithm in order to be able to run it in the encrypted domain. We have done this with the hope that our return on experiment may be useful to other "FHE programmers".

## References

1. Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression. In *FSE*, volume 9783 of *Lecture Notes in Computer Science*, pages 313–333. Springer, 2016.
2. Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: A Compilation Chain for Privacy Preserving Applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, pages 13–19, 2015.
3. Sergiu Carpov and Renaud Sirdey. Another compression method for homomorphic ciphertexts. In *Proceedings of the 4rd International Workshop on Security in Cloud Computing*, pages 44–50, 2016.
4. Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.
5. Simon Fau, Renaud Sirdey, Caroline Fontaine, Carlos Aguilar Melchor, and Guy Gogniat. Towards Practical Program Execution over Fully Homomorphic Encryption Schemes. In *3PGCIC*, pages 284–290, 2013.
6. Shai Halevi and Victor Shoup. Algorithms in HElib. In *CRYPTO*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571, 2014.
7. Shai Halevi and Victor Shoup. Bootstrapping for HElib. In *EUROCRYPT*, volume 9056 of *Lecture Notes in Computer Science*, pages 641–670. Springer, 2015.

## A RLE-3 code

Concrete code initially inputed to the Armadillo compiler for the RLE-3 variant (Fig. 2-RLE3):

```
Integer8 input[NUM_OF_CHARS];
for(int i=0;i<NUM_OF_CHARS;i++)
  cin>>input[i];
Integer8 j=1,k=0;
Integer8 output_chr[NUM_OF_PAIRS];
Integer8 output_ctr[NUM_OF_PAIRS];
for(int i=0;i<NUM_OF_PAIRS;i++)
{
  output_chr[i]=1;
  output_ctr[i]=0;
}

int i;
for(i=1;i<NUM_OF_CHARS;i++)
{
  for(int l=0;l<NUM_OF_PAIRS;l++)
  {
    output_chr[l]=select(k==l,input[i-1],output_chr[l]);
    output_ctr[l]=select(k==l,j,output_ctr[l]);
  }
  k=select(input[i]==input[i-1],k,k+1);
  j=select(input[i]==input[i-1],j+1,Integer8(1));
}
for(int l=0;l<NUM_OF_PAIRS;l++)
{
  output_chr[l]=select(k==l,input[i-1],output_chr[l]);
  output_ctr[l]=select(k==l,j,output_ctr[l]);
}

for(int l=0;l<NUM_OF_PAIRS;l++)
{
  cout<<output_ctr[l];
  cout<<output_chr[l];
}
```

## B Optimized calculation of j

Starting for a precomputation of the $b_i$'s:

```
Bit B[NUM_OF_CHARS];
B[0]=0;
```

```
for(i=1;i<NUM_OF_CHARS;i++)
  B[i]=input[i]==input[i-1];
```

Then j updates is performed with the following procedure call:

```
j=computeJ(i,B);
```

With,

```
Bit treeAnd(const Bit B[],int l,int u)
{
  assert(u>=l);
  if(l==u)
    return B[l];
  if(l==u-1)
    return B[l]&B[u];
  int m=l+(u-l)/2;
  Bit v0=treeAnd(B,l,m);
  Bit v1=treeAnd(B,m+1,u);
  return v0&v1;
}

Integer8 computeJ(int i,const Bit B[])
{
  assert(i>=0);
  if(i==0)
    return Integer8(1);
  Integer8 j=Integer8(1);
  for(int l=1;l<=i;l++)
  {
    j+=treeAnd(B,l,i);
  }
  return j;
}
```