INF8750: BlindSort

Félix Larose-Gervais

April 30, 2024

Contents

1 Introduction

1.1 Motivation

Currently, when a client needs some computation from a server, cryptography is often used to protect the client's data in transit. There will first be an asymmetric key exchange that will allow the client and server to agree on a key for symmetric encryption for the rest of the communication. This model protects the confidentiality of the user's data against an eavesdropper, however, the server decrypts and learn the user's data.

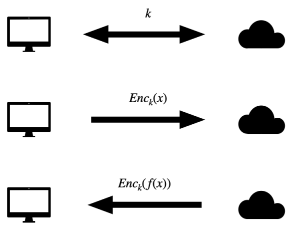

Figure 1: Key exchange followed by symmetric encryption

To enhance the user's privacy, we can use Fully Homomorphic Encryption (FHE). It allows computation to be made on ciphertexts without neeeding to decrypt them. This in turn enables building privacy preserving services that can act on their client's data without needing to see it in clear.

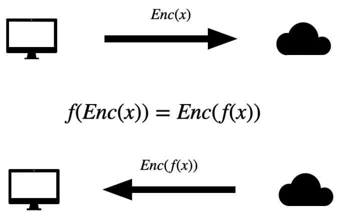

Figure 2: Homomorphic encryption (The server doesn't learn the user's data)

Even though FHE has evolved a lot since its inception, its rather low performance is still hindering adoption. The goal of this project is to investigate performance of a modern FHE framework by implementing some sorting algorithms and comparing them with each other and the state of the art.

The first two algorithms are variants of the Direct Sort, first proposed by [Çetin *et al.*, 2015]. One using the Blind Matrix Access and Blind Permutation offered by RevoLUT [Azogagh, 2024], the other being a simple port of Direct Sort to TFHE-rs [Zama, 2022]. We also present a new algorithm based on two passes of Blind Permutation that speeds up the process significantly for certain types of arrays.

1.2 Homomorphic encryption

Shortly after publishing the RSA cryptosystem [Rivest *et al.*, 1978b], its authors noticed the scheme was in fact partially homomorphic [Rivest *et al.*, 1978a]. Then they asked whether a fully homomorphic encryption scheme was possible.

Definition 1 *A mapping $\varphi : \mathbb{G} \to \mathbb{H}$ is called an* ***homomorphism*** *if $\forall x, y \in \mathbb{G}$ we have:*

$$\varphi(xy) = \varphi(x)\varphi(y)$$

For example, $\varphi_q : \mathbb{Z} \to \mathbb{Z}_q$, $x \mapsto x \mod q$ is an homomorphism because it respects addition.

Definition 2 *An encryption scheme does* ***homomorphic encryption*** *if its encryption procedure is an homomorphism (modulo decryption).*

For example, an homomorphic encryption scheme $\Pi = (Gen, Enc, Dec)$ will satisfy:

$$Enc(x + y) \equiv_{Dec} Enc(x) \oplus Enc(y)$$

With respect to a $+$ a plaintext operation and its corresponding ciphertext operation $\oplus$.

Definition 3 *An encryption scheme is said to be* ***partially homomorphic*** *if it only allows certain computation on its ciphertexts without having to decrypt them.*

For example, the encryption scheme $RSA = (Gen, Enc, Dec)$ is partially homomorphic, because multiplications can be computed on ciphertexts.

$$RSA = \begin{cases} Gen(1^n) & := (N, e, d) \leftarrow GenRSA \\ Enc(m) & := m^e \mod N \\ Dec(c) & := c^d \mod N \end{cases}$$

Where $GenRSA$ computes $N = pq$ the product of two random primes and e, d inverses in $\mathbb{Z}^{\times}_{\varphi(N)}$. Given $m_1, m_2 \in \mathbb{Z}_N$, such that $c_1 = Enc(m_1)$ and $c_2 = Enc(m_2)$, we can verify:

$$\begin{aligned} Enc(m_1 m_2) &\equiv (m_1 m_2)^e \pmod N \\ &\equiv m_1^e m_2^e \pmod N \\ &\equiv Enc(m_1)Enc(m_2) \end{aligned}$$

We can see that decryption of the multiplication of ciphertexts yields the same result as multiplicating the corresponding plaintexts. Other cryptosystems have been shown to be partially homomorphic, notably [ElGamal, 1985], [Benaloh, 1994] and [Paillier, 1999].

Definition 4 *An encryption scheme is said to be* ***fully homomorphic*** *if it allows any computation on its ciphertexts without having to decrypt them.*

The existence of a fully homomorphic encryption scheme was an open problem for 30 years. It was shown possible by [Gentry, 2009], who constructed the first fully homomorphic encryption scheme and presented a new idea, called bootstrapping, that allows to adapt some partially homomorphic schemes to make them fully homomorphic.

After his work, many fully homomorphic encryption schemes were proposed. In particular, TFHE [Chillotti *et al.*, 2020], implemented in the TFHE-rs library, greatly improved performance over the initial construction.

1.3 Fast Fully Homomorphic encryption over the Torus (TFHE)

Fully Homomorphic Encryption over the Torus (TFHE) [Chillotti *et al.*, 2020] is a FHE scheme based on the Learning With Errors (LWE) problem. The security of the scheme is itself reduced to the difficulty of the Shortest Vector Problem (SVP) in lattices [Regev, 2009].

For $p < q$ powers of two with $\Delta = q/p$, a message $m \in \mathbb{Z}_p$ can be encrypted into $(a, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$

$$LWE = \begin{cases} Gen(1^n) & := s \overset{R}{\leftarrow} \{0,1\}^n \\ Enc_s(m) & := (a, \sum a_i s_i + \Delta m + e),\ a \overset{R}{\leftarrow} \mathbb{Z}_q^n, e \overset{R}{\leftarrow} \chi_\sigma \\ Dec_s(a,b) & := \lfloor b - \sum a_i s_i \rceil / \Delta \end{cases}$$

Where a (the mask) is chosen uniformly at random and e is sampled from a centered Gaussian distribution with standard deviation σ. We can verify this schema is additively homomorphic.

$$\begin{aligned} Enc_s(m_1) + Enc_s(m_2) &= (a_1, a_1 \cdot s + \Delta m_1 + e_1) + (a_2, a_2 \cdot s + \Delta m_2 + e_2) \\ &= (a_1 + a_2, (a_1 + a_2) \cdot s + \Delta(m_1 + m_2) + (e_1 + e_2)) \\ &= Enc_s(m_1 + m_2) \end{aligned}$$

Similarly, we can encrypt arrays of fixed size N (a power of 2) by encoding them as elements of $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$, that is the polynomials with integer coefficients (modulo q) of degree at most $N - 1$.

$$RLWE = \begin{cases} Gen(1^n) & := S \overset{R}{\leftarrow} \mathcal{R}_2 \\ Enc_s(M) & := (A, AS + \Delta M + E),\ A \overset{R}{\leftarrow} \mathcal{R}_q, E \overset{R}{\leftarrow} \chi_\sigma \\ Dec_s(A,B) & := \lfloor B - AS \rceil / \Delta \end{cases}$$

It's important to note that blind operations on the (R)-LWE ciphertexts accumulate noise. Therefore, after a certain amount of operations, that noise would overflow in the most significant bits that are reserved for the message, thus rendering the ciphertext un-decipherable. To solve this issue, [Gentry, 2009] proposed the idea of bootstrapping; that is evaluating the decryption procedure homomorphically, given an encrypted key and doubly encrypted message. Repeating this procedure allows keeping the noise under a certain thresold.

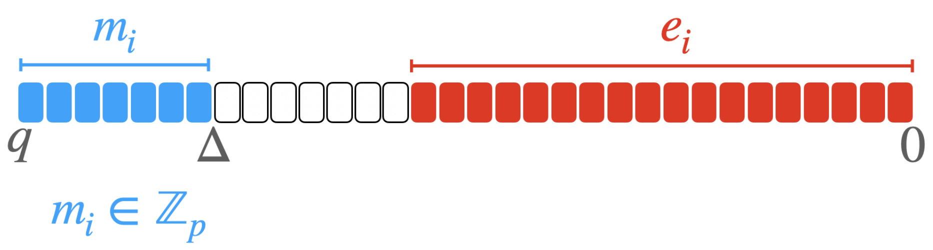

Figure 3: Visual representation of a LWE ciphertext[Zama.ai, 2022a]

1.4 RevoLUT

The RevoLUT project, based on TFHE-rs, exposes a structure called a Look-Up Table (LUT). This structure is a fixed size array of ciphertexts, for which many new primitives where implemented such as blind permutation, blind array access and blind matrix access. These are all made possible thanks to TFHE's blind rotation, which is also at the core of their bootstrapping implementation.

Blind Rotation

Blind Rotation is a primitive offered by TFHE-rs that allows to rotate a RLWE ciphertext by an encrypted integer amount. Given an encrypted polynomial $P = \sum a_i X^i$ and an encrypted integer α, it computes $PX^{-\alpha}$. Due to the noise in α this operation is slightly inaccurate. To solve this, we introduce redundancy in the coefficients of the encrypted polynomials to accomodate for any noise.

Blind Array Access

A Blind Array Access allows to access the encrypted coefficient of a LUT at an encrypted integer index α. This is done by first performing a Blind Rotation of the LUT by α, then a Sample Extraction (also provided by TFHE-rs) of the rotated LUT at index 0.

Blind Matrix Access

A Blind Matrix Access allows to access the encrypted coefficient of an array of LUTs at an encrypted index (α, β). This is done by first performing a Blind Array Access into all the LUTs at index α, packing the results into a new LUT and performing a Blind Array Access into this new LUT at index β.

Blind Permutation

A Blind Permutation of a given LUT T by an array of integer indices σ is performed by obtaining the n coefficients T_i through Sample Extraction, and constructing n LUTs starting with T_i, each rotating them blindly by the corresponding σ_i and then summing the results into the output LUT.

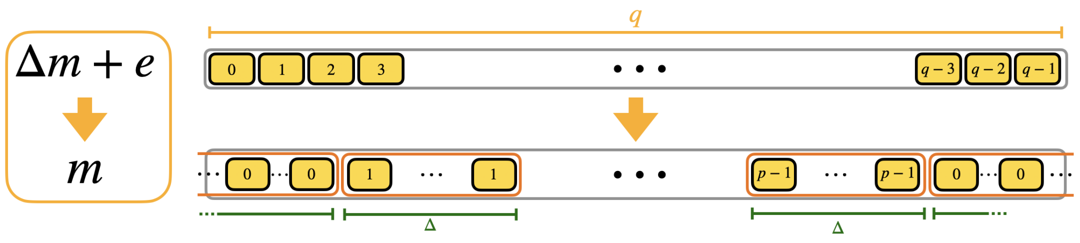

Figure 4: Visual representation of a LUT[Zama.ai, 2022b]

2 Simple Sort

2.1 Main algorithm

This first algorithm is the simplest, it is a variation of insertion sort documented in [Fung, 2021]. Its purpose is to serve as a baseline to compare to.

Algorithm 1 Simple Sort

```
function BlindSort(T: [FheUint])
    for $i = 0$ to $n$ do
        for $j = 0$ to $i$ do
            $min = BlindMin(T_i, T_j)$
            $max = BlindMax(T_i, T_j)$
            $T_j \leftarrow min$
            $T_i \leftarrow max$
        end for
    end for
end function
```

Notes

It is interesting to see how easily certain algorithms can be ported to FHE. In this one, the indices being accessed in the array do not depend on the result of blind comparisons so implementation is pretty straightforward.

It would have been desirable to also port an efficient algorithm to FHE for comparison purpose, but it is not as obvious how partition based algorithms like quicksort and mergesort would transfert. That is due to array accesses at indices depending on the result of blind comparisons.

2.2 Implementation

This algorithm was implemented using TFHE-rs, that already provides $BlindMin$ and $BlindMax$.

Analysis

The procedure runs in $O(n^2)$ steps, each costing 2 blind operations. This makes the algorithm seem really inefficient. Despite this, it requires fewer blind operations than other more complicated algorithms presented after and therefore performs surprisingly well.

3 Direct Sort

3.1 Main algorithm

The following is an adaptation of the direct sort proposed by [Çetin *et al.*, 2015]. The main idea of the algorithm is to construct a permutation from blind comparisons between pairs of elements of the given array, then apply it blindly.

Algorithm 2 Direct Sort

```
function BlindSort(T: [LWE])
    $\sigma \leftarrow [0, \dots, 0]$
    for $i = 0$ to $n$ do
        for $j = 0$ to $i$ do
            $b \leftarrow BlindLt(T_i, T_j)$
            $\sigma_i \leftarrow \sigma_i + b$
            $\sigma_j \leftarrow \sigma_j + 1 - b$
        end for
    end for
    return $BlindPermutation(T, \sigma)$
end function
```

In the first step, we conceptually construct a comparison matrix $C = (C_{i,j})$ such that:

$$C_{i,j} = \begin{cases} BlindLt(T_i, T_j) & \text{if } i < j \\ 1 - C_{j,i} & \text{otherwise} \end{cases}$$

For example, given the LUT $T = [2, 1, 3, 2]$, we have:

$$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

We can view $\sigma = (1, 0, 3, 2)$ as the sums of the rows of the comparison matrix C.
Then, when applied to T we get the sorted result $\sigma(T) = [1, 2, 2, 3]$.

Notes

It is important to note that the blind comparison only happens for $i < j$. This was originally a performance concern due to the slowness of $BlindLt$ in RevoLUT (making only $\frac{n^2-n}{2}$ calls instead of n), but it ended up being important for correctness in arrays containing duplicates.

If C was defined as $(C_{i,j})$ such that $C_{i,j} = BlindLt(T_i, T_j)$, we'd have:

$$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Which would yield $\sigma = (1, 0, 3, 1)$ and would result in incorrect sorting.

3.2 Implementations

RevoLUT

$BlindLt(x, y)$ is defined as a $BlindMatrixAccess(L, x, y)$ with $L = (i < j)_{0 \leq i,j < n}$:

$$L = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

The matrix L is trivially encrypted; that is it's encrypted like with the LWE encryption procedure described earlier, except the mask a is only zeroes instead of being sampled from a uniform distribution. This trivial encryption offers no security as it doesn't hide the message at all, but it allows running blind operations on it, and the result of those are properly confidential.

As for BlindPermutation, it is already implemented in RevoLUT.

TFHE-rs

This second implementation works very similarly to the first one, except that $BlintLt$ is already provided by TFHE-rs, but not $BlindPermutation$. It can be implemented like [Çetin *et al.*, 2015] in the following way:

Algorithm 3 Blind Permutation in TFHE-rs

```
function BlindPermutation(T: [TfheUint], $\sigma$: [TfheUint])
    $R \leftarrow [0, \dots, 0]$
    for $i = 0$ to $n$ do
        for $j = 0$ to $n$ do
            $R_i \leftarrow R_i + BlindEq(\sigma_j, i) \times T_j$
        end for
    end for
    return R
end function
```

$BlindEq$ is provided by TFHE-rs.

Analysis

The main differences between the two implementations are in the algorithmic complexities of $BlindLt$ and $BlindPermutation$.

For RevoLUT, $BlindLt \in O(n^2)$ because $BlindMatrixAccess$ requires $n + 1$ blind rotations, each of which are linear in n. $BlindPermutation \in O(n^2)$ because it requires n blind rotations. This totals to $O(n^4)$.

However, in the second implementation, because the blind comparisons are implemented as bivariate functions over radix integers, they are essentially logarithmic in n. Thus, we have $BlindLt \in O(\log(n))$ and $BlindPermutation \in O(n^2 \log(n))$. That makes the whole procedure run in time $O(n^2 \log(n))$.

4 Double Blind Permutation

4.1 Main algorithm

The idea of this third algorithm is to interpret the given array as a permutation, apply it to itself to get a partially ordered sparse array. Then fix it by percolating null values to the end.

Algorithm 4 Permutation Sort

```
function BlindSort(T: [LWE])
    $R \leftarrow BlindPermutation(T, T)$
    $\sigma \leftarrow [0, \ldots, 0]$
    $z \leftarrow 0$
    for $i = 0$ to $n$ do
        $z \leftarrow z + BlindEq(R_i, 0)$
        $\sigma_i \leftarrow R_i - z$
    end for
    return $BlindPermutation(R, \sigma)$
end function
```

For example, given an array

$$T = [5, 2, 7, 3, 0, 0, 0, 0]$$

We can construct

$$\sigma_1 = (5, 2, 7, 3, 0, 0, 0, 0)$$

And apply it to itself to get

$$R = \sigma_1(T) = [0, 0, 2, 3, 0, 5, 0, 7]$$

Then we conceptually construct Z such that Z_i counts zeroes in R up to (and including) i

$$Z = [1, 2, 2, 2, 3, 3, 4, 4]$$

We compute $\sigma_2 = R - Z \mod n$

$$\sigma_2 = (7, 6, 0, 1, 5, 2, 4, 3)$$

And then finally we return $BlindPermutation(R, \sigma_2)$

$$\sigma_2(R) = [2, 3, 5, 7, 0, 0, 0, 0]$$

Notes

There are a few limitations to this approach. First, it doesn't work on duplicate entries in its current form (the first permutation will lose information). Secondly, it doesn't consider 0 as a value provided by the user.

4.2 Implementations

RevoLUT

$BlindPermutation$ is already implemented in RevoLUT. And $BlindEq(R_i, 0)$ can be expressed as $BlindArrayAccess(L, R_i)$ for the trivially encrypted $L = [1, 0, \dots, 0]$.

TFHE-rs

$BlindPermutation$ as defined earlier in Algorithm 3 can be re-used here, and $BlindEq$ is already provided by TFHE-rs. That makes this implementation fairly straightforward.

Analysis

For the first implementation, recall that $BlindPermutation \in O(n^2)$ in RevoLUT, and the n passes of the linear $BlindEq$ add up to a total of $O(n^2)$.

As for the second, the middle loop clocks in at a faster $O(n\log(n))$ thanks to TFHE-rs' fast bivariate functions, but the whole time is dominated by the 2 slower blind permutation that are running in $O(n^2 \log(n))$.

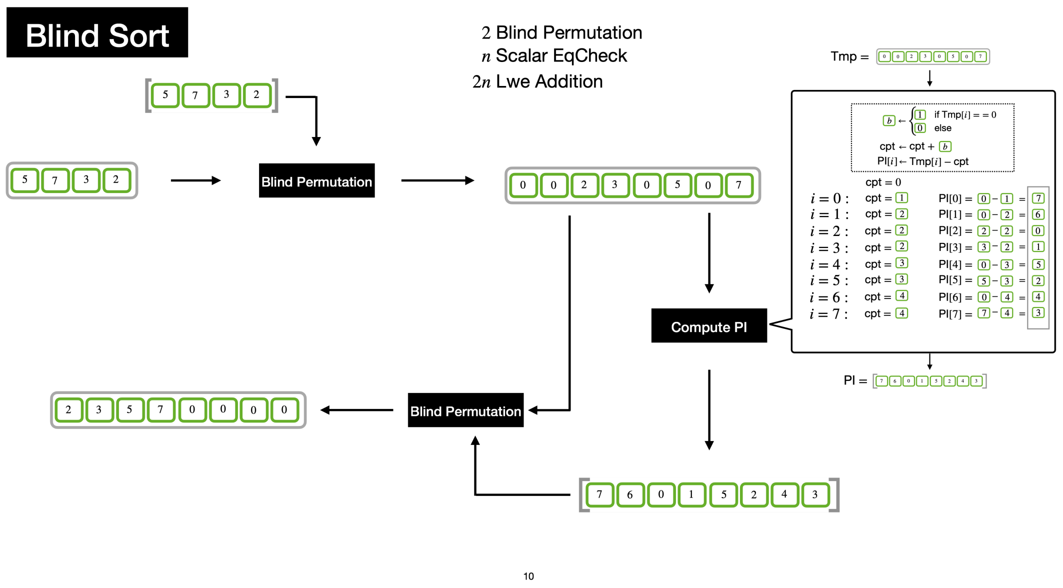
Blind Sort
2 Blind Permutation
n Scalar EqCheck
$2n$ Lwe Addition
[5 7 3 2]
5 7 3 2
Blind Permutation
0 0 2 3 0 5 0 7
Compute PI
Tmp = 0 0 2 3 0 5 0 7
$b \leftarrow \begin{cases} 1 & \text{if Tmp}[i] == 0 \\ 0 & \text{else} \end{cases}$
cpt ← cpt + b
PI[i] ← Tmp[i] − cpt
cpt = 0
$i = 0$: cpt = 1 PI[0] = 0 − 1 = 7
$i = 1$: cpt = 2 PI[1] = 0 − 2 = 6
$i = 2$: cpt = 2 PI[2] = 2 − 2 = 0
$i = 3$: cpt = 2 PI[3] = 3 − 2 = 1
$i = 4$: cpt = 3 PI[4] = 0 − 3 = 5
$i = 5$: cpt = 3 PI[5] = 5 − 3 = 2
$i = 6$: cpt = 4 PI[6] = 0 − 4 = 4
$i = 7$: cpt = 4 PI[7] = 7 − 4 = 3
PI = [7 6 0 1 5 2 4 3]
2 3 5 7 0 0 0 0
Blind Permutation
[7 6 0 1 5 2 4 3]
10

Figure 5: Illustration of the Double Blind Permutation sorting algorithm

5 Results

5.1 Methodology

To compare the performance of the different sorting algorithms implemented, we used the statistics-driven micro-benchmarking framework Criterion. Multiple runs of randomly generated encrypted arrays of various sizes where sorted with all implementations.

Also, the Blind Permutation of RevoLUT is compared to the naive one we implemented in TFHE-rs to gain insights on the performance characteristics of the two sorting algorithms.

5.2 Benchmarks

The first two figures are extracted from the cited papers, other numbers where obtained by running the experiment code available on github.com at filedesless/blindsort and sofianeazogagh/revoLUT. The experiments were ran on a 13-inch macbook pro m1 2020.

	Array Size			
	4	8	16	32
Direct Sort ([Çetin *et al.*, 2015])	500ms	5s	50s	7m
Direct Sort ([Iliashenko et Zucca, 2021])	80ms	700ms	6s	50s
Direct Sort (RevoLUT)	2s	20s	2m	-
Direct Sort (TFHE-rs)	2s	10s	40s	2m
Naive Blind Permutation (TFHE-rs)	1s	4s	16s	1m
RevoLUT Blind Permutation (RevoLUT)	400ms	800ms	3.6s	10s
Double Blind Permutation (THFE-rs)	3s	9s	33s	2m
Double Blind Permutation (RevoLUT)	1s	2s	5s	21s
Simple Sort (TFHE-rs)	1s	5s	23s	104s

5.3 Remarks

First we can observe that even though Blind Permutation is faster within RevoLUT than TFHE-rs, this isn't enough to overcome the slowness of the Blind Matrix Access used in the Blind Comparison. However, it makes a great difference for the second algorithm where we get very interesting figures.

The Double Blind Permutation shows promising results, being much faster than the state of the art. However, there is still some outstanding issues that need to be resolved and the numbers from the papers should be reproduced on the same hardware to obtain a more fair comparison.

Finally, the most surprising result was the relative performance of Simple Sort. This is probably due to the fewer blind operations being performed in total.

6 Conclusion

In conclusion, the results show that the fastest proposed algorithm is the Double Blind Permutation, implemented with RevoLUT's fast Blind Permutation.

6.1 Caveats

However, the algorithm is limited to sorting distinct non null values. We have some ideas to extend it, but at an unclear performance cost. In order to support duplicates, we would have to somehow count them and adjust the permutations accordingly.

It would also be nice to run the state of the art benchmarks on the same hardware as our proposed solution. Currently, timing numbers are being compared from different sources which is fairly inaccurate.

6.2 Future work

Other research avenues include investigating the feasability of porting partition based sorting scheme like quicksort or mergesort to FHE. The indices being accessed in the encrypted array depending on the result of blind operations make this non-obvious, but there might be a way to use Blind Rotations to achieve the same effect at a reasonable performance cost.

Another interesting option is to consider non comparison based sorting algorithms like counting sort. The fixed size nature of RevoLUT seem to make it a good candidate for algorithms whose complexity depends on the size of the structure and the size of the stored elements.

References

[Azogagh, 2024] Azogagh, S. (2024). Revolut. https://github.com/sofianeazogagh/revoLUT.

[Benaloh, 1994] Benaloh, J. (1994). Dense probabilistic encryption. Dans *Proceedings of the workshop on selected areas of cryptography*, 120–128.

[Chillotti *et al.*, 2020] Chillotti, I., Gama, N., Georgieva, M. et Izabachène, M. (2020). TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology*, *33*(1), 34–91. http://dx.doi.org/10.1007/s00145-019-09319-x. Récupéré le 2024-02-05 de http://link.springer.com/10.1007/s00145-019-09319-x

[ElGamal, 1985] ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, *31*(4), 469–472.

[Fung, 2021] Fung, S. P. Y. (2021). Is this the simplest (and most surprising) sorting algorithm ever?

[Gentry, 2009] Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. Dans *Proceedings of the forty-first annual ACM symposium on Theory of computing*, STOC '09, 169–178., New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/1536414.1536440. Récupéré le 2024-03-19 de https://doi.org/10.1145/1536414.1536440

[Iliashenko et Zucca, 2021] Iliashenko, I. et Zucca, V. (2021). Faster homomorphic comparison operations for BGV and BFV. *Proceedings on Privacy Enhancing Technologies.* Récupéré le 2024-01-25 de `https://petsymposium.org/popets/2021/popets-2021-0046.php`

[Paillier, 1999] Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. Dans J. Stern (dir.). *Advances in Cryptology — EUROCRYPT '99*, 223–238., Berlin, Heidelberg. Springer Berlin Heidelberg.

[Regev, 2009] Regev, O. (2009). On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. arXiv:2401.03703 [quant-ph], `http://dx.doi.org/10.48550/arXiv.2401.03703`. Récupéré le 2024-03-20 de `http://arxiv.org/abs/2401.03703`

[Rivest *et al.*, 1978a] Rivest, R. L., Adleman, L. et Dertouzos, M. L. (1978a). On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, 169–179.

[Rivest *et al.*, 1978b] Rivest, R. L., Shamir, A. et Adleman, L. (1978b). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, *21*(2), 120–126. `http://dx.doi.org/10.1145/359340.359342`. Récupéré le 2024-01-26 de `https://dl.acm.org/doi/10.1145/359340.359342`

[Zama, 2022] Zama (2022). TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. `https://github.com/zama-ai/tfhe-rs`.

[Zama.ai, 2022a] Zama.ai (2022a). Glwe encoding coefficients. Récupéré de `https://www.zama.ai/post/tfhe-deep-dive-part-1`

[Zama.ai, 2022b] Zama.ai (2022b). Glwe encoding coefficients. Récupéré de `https://www.zama.ai/post/tfhe-deep-dive-part-4`

[Çetin *et al.*, 2015] Çetin, G. S., Doröz, Y., Sunar, B. et Savaş, E. (2015). Depth Optimized Efficient Homomorphic Sorting. In K. Lauter et F. Rodríguez-Henríquez (dir.), *Progress in Cryptology – LATINCRYPT 2015*, volume 9230 61–80. Cham: Springer International Publishing. Series Title: Lecture Notes in Computer Science