RevoLUT : Rust Efficient Versatile Oblivious Look-Up-Tables

Sofiane Azogagh
azogagh.sofiane@courrier.uqam.ca
Univ Québec à Montréal
Canada

Zelma Aubin Birba
birba.zelma_aubin@courrier.uqam.ca
Univ Québec à Montréal
Canada

Marc-Olivier Killijian
killijian.marc-olivier.2@uqam.ca
Univ Québec à Montréal
Canada

Félix Larose-Gervais
larose-gervais.felix@courrier.uqam.ca
Univ Québec à Montréal
Canada

ABSTRACT

In this paper we present RevoLUT, a library implemented in Rust that reimagines the use of Look-Up-Tables (LUT) beyond their conventional role in function encoding, as commonly used in TFHE's programmable boostrapping. Instead, RevoLUT leverages LUTs as first class objects, enabling efficient oblivious operations such as array access, elements sorting and permutation directly within the table. This approach supports oblivious algortithm, providing a secure, privacy-preserving solution for handling sensitive data in various applications.

KEYWORDS

Privacy, Homomorphic encryption, Oblivious algorithm, Library

1 INTRODUCTION

For many years, fully homomorphic encryption was limited to vanilla arithmetic operations - multiplication and addition of ciphertexts. Non-linear functions, such as ReLu, had to be approximated using polynomial interpolation. A breakthrough came with the development of FHEW-like schemes [8, 12, 14], which introduced new bootstrapping operations that enabled an important mechanism: programmable bootstrapping (PBS). During a PBS operation, any function, linear or non-linear, can be evaluated at no cost. The outputs of the function are discretized in a Look-Up-Table (LUT) and then applied during the PBS on the bootstrapped data. Other works are also trying to apply this paradigm to other encryption schemes such as the CKKS scheme [9] like in [2] but, to the best of our knowledge, at the moment PBS is only available in pratice in TFHE.

Look-Up-Tables are a well-established concept in computer science, where they serve as data structures that store precomputed values to avoid expensive runtime computation. However, in the context of fully homomorphic encryption, LUTs have evolved further their traditional role. They have become essential building blocks for many applications ranging from storing an activation function in neural networks [6, 7] to designing a 8-bit FHE processor abstraction [16].

With RevoLUT, we take the innovation a step further by introducing LUTs as first-class objects. The library offers a range of oblivious mechanisms to manipulate, compute, access, permute, and even sort these objects. This approach provides a novel abstraction, empowering engineers to design efficient oblivious algorithms for computing on encrypted data.

2 PRELIMINARIES

2.1 Notation

Let p be a power of 2. We denote by $\mathbb{Z}_p$ the set of messages and by $[\![m]\!]$ the TFHE encryption of a message $m \in \mathbb{Z}_p$. For N a power of 2, we define $\mathcal{R}$ as the quotient ring $\mathbb{Z}[X]/(X^N+1)$ and $\mathcal{R}_q$ as the same ring modulo q, that is $\mathbb{Z}_q[X]/(X^N+1)$. Unless otherwise specified, all operations in this paper are performed in the ring $\mathcal{R}_q$. We also make use of the Kronecker delta function $\delta_{i,j}$, which equals 1 when $i = j$ and 0 otherwise. Using this notation, we can define the one-hot encoding of an integer i as the bit vector $\delta_i = (\delta_{i,0}, \dots, \delta_{i,p-1}) \in \{0,1\}^p$, which contains a single 1 at index i and 0s elsewhere. Other notations are defined in the text when needed.

2.2 The TFHE Cryptosystem

The TFHE encryption scheme, proposed in 2016 [10, 11], is based on the security of the Learning With Errors (LWE) problem and its ring variant, the Ring-LWE (RLWE) problem.

2.2.1 Ciphertext Types. In TFHE, several types of ciphertexts are defined depending on the nature of the plaintext and the encryption method employed. A commonly used type in this paper is the General LWE (GLWE) ciphertext, defined as follows:

GLWE Ciphertexts. A message $m \in \mathbb{Z}_p$ can be encrypted under the secret key $s = (s_0, \dots, s_{k-1}) \xleftarrow{\$} \mathbb{Z}_2^k$ as a GLWE ciphertext $(a, b) \in \mathcal{R}_q^{k+1}$, where $a = (a_0, \dots, a_{k-1}) \xleftarrow{\$} \mathcal{R}_q^k$ and $b = \sum_{i=0}^{k-1} a_i \cdot s_i + \Delta m + e$, with $\Delta = \frac{q}{p}$ and e being a noise term sampled from a Gaussian distribution. The vector a is called *mask* and b *body*.

Specifically, when $N = 1$, the ciphertext is referred to as an LWE ciphertext. When $k = 1$ and $N > 1$, it is termed an RLWE ciphertext. In this case, an LWE ciphertext encrypts a message in $\mathbb{Z}_q$, while an RLWE ciphertext encrypts a polynomial in $\mathbb{Z}_q[X]$ modulo $X^N + 1$.

LUT Ciphertexts. Additionally, [4] introduced the concept of Look-Up-Table (LUT) ciphertexts, which are essentially RLWE ciphertexts that include some redundancy. A Look-Up-Table in TFHE is a vector $(m_i)_{0 \le i < p}$ of $\mathbb{Z}_p$ elements represented as a polynomial $M(X) \in \mathcal{R}_q$ of the form:

$$M(X) = \sum_{i=0}^{p-1} \sum_{j=0}^{\frac{N}{p}-1} m_i X^{i\frac{N}{p}+j}$$

This polynomial is then encrypted as an RLWE ciphertext to form a LUT ciphertext as illustrated in Figure 1.

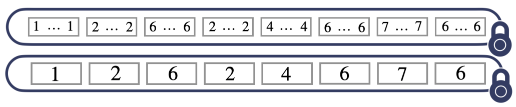

Fig. 1. Illustration of a RLWE ciphertext (top) with redundancy shown in gray boxes, which implements a LUT ciphertext (bottom) where each box represents an element in $\mathbb{Z}_p$ (here $p = 8$).

In this paper, ciphertexts are denoted within brackets to indicate their type. For instance, $[\![M]\!]_{\text{LUT}} = [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}}$ represents the message $M = (m_0, \ldots, m_{p-1})$ encrypted as a LUT ciphertext, while $[\![m]\!]_{\text{LWE}}$ is an LWE ciphertext and $[m]_{\text{LWE}}$ is a trivially encrypted LWE ciphertext (that is a ciphertext whose mask and noise are set to 0).

2.2.2 TFHE's operations. TFHE provides several building blocks for performing homomorphic operations on ciphertexts. The main operations used in this paper are:

- **Blind Rotation (BR):** $([\![\star]\!]_{\text{LWE}}, [\![\star]\!]_{\text{LUT}}) \rightarrow [\![\star]\!]_{\text{RLWE}}$. This operation is used to privately rotate the polynomial $M(X)$ (encrypted as an RLWE ciphertext) by $[\![i]\!]_{\text{LWE}}$ coefficients.
- **Sample Extraction (SE):** $(\star, [\![\star]\!]_{\text{RLWE}}) \rightarrow [\![\star]\!]_{\text{LWE}}$.
  This operation extracts a coefficient from the polynomial $M(X) = \sum_{i=0}^{N-1} m_i X^i$ encrypted as an RLWE ciphertext, resulting in an LWE ciphertext $[\![m_j]\!]_{\text{LWE}}$. The LWE ciphertext is generated by selecting specific coefficients from the RLWE input.
- **Key Switching (KS):** $[\![\star]\!]_{\text{LWE}} \rightarrow [\![\star]\!]_{\text{LWE}}$.
  This operation switches the secret key or parameters of an LWE ciphertext to new ones by homomorphically re-encrypting the ciphertext with a different key.
- **Public Functional Key Switch (PFKS):** $\{[\![\star]\!]_{\text{LWE}}\} \rightarrow [\![\star]\!]_{\text{RLWE}}$.
  Introduced in [12] (Algorithm 2), this operation allows for the compact representation of multiple LWE ciphertexts into a single RLWE ciphertext, effectively packing several LWE ciphertexts into one.

The redundancy in a LUT ciphertext is mainly important to guarantee the correctness of the bootstrapping operation. Indeed, the LWE ciphertext used in the Blind Rotation operation serves as an index to select the correct coefficient from the LUT ciphertext. However, this LWE ciphertext incorporates a gaussian noise e which is bounded by N/p after the so-called Modulus Switching operation (see [13] for more details). This bound gives exactly the size of the redundancy of the coefficients in the RLWE ciphertext implementing the LUT. These sequences of consecutive coefficients in the RLWE ciphertext implementing a LUT are generally called *boxes*. During the (functional) bootstrapping operation, each box corresponds to a specific message m_i of the LUT ciphertext. When the Blind Rotation is performed, $[\![i]\!]_{\text{LWE}}$ points to the i-th box containing the message m_i in the LUT. Thus, the redundancy ensures that, despite the random error present in $[\![i]\!]_{\text{LWE}}$, the Sample Extraction operation will still correctly select the message m_i as long as the noise e is smaller than the redundancy.

3 REVOLUT'S OPERATIONS

The aforementioned operations implemented in [17] led us to design a library named RevoLUT that leverages the LUT ciphertexts and enables manipulating data obliviously in those recipients. In the following, we present the main operations implemented in RevoLUT and used in our sorting algorithm. We refer the interested reader to [1] for more details and more operations.

3.1 Reading operations

Look-Up-Tables seen as ciphertexts of arrays and TFHE's operations offer some reading operations that can be used to implement oblivious algorithms.

Blind Array Access (BAA). Introduced in [3], this operation is the most basic reading operation on a LUT ciphertext. It is basically the *programmable bootstrapping* of TFHE. Given a LUT ciphertext of an array and a LWE ciphertext of an index, BAA returns a LWE ciphertext of the array element at the given ciphered index.

Algorithm 1: Blind Array Access (BAA)

Input : An encrypted index $[\![i]\!]_{\text{LWE}}$
A LUT ciphertext $[\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}}$
Output: A LWE ciphertext $[\![m_i]\!]_{\text{LWE}}$

```
[[rotated]]_LUT ← BR([[i]]_LWE, [[m_0, ..., m_{p-1}]]_LUT)
[[m_i]]_LWE ← SE(0, [[rotated]]_LUT)
return [[m_i]]_LWE
```

Blind Matrix Access (BMA). Introduced in [4], this operation is the generalization of BAA to matrices. Given a matrix encoded as a row first vector of LUT ciphtertexts, and a pair of indices as LWE ciphertexts, BMA returns the value stored at the row and column as a LWE ciphertext.

Algorithm 2: Blind Matrix Access (BMA)

Input : A vector of LUT $\begin{pmatrix} [\![m_{0,0}, \ldots, m_{0,n}]\!]_{\text{LUT}} \\ \vdots \\ [\![m_{n,0}, \ldots, m_{n,n}]\!]_{\text{LUT}} \end{pmatrix}$
A pair of LWE ciphertext $[\![r]\!]_{\text{LWE}}, [\![c]\!]_{\text{LWE}}$
Output: A LWE ciphertext $[\![m_{r,c}]\!]_{\text{LWE}}$

```
for i ← 0 to n do
  | [[m_{i,c}]]_LWE ← BAA([[m_{i,0}, ..., m_{i,n}]]_LUT, [[c]]_LWE)
end
[[m_{0,c}, ..., m_{n,c}]]_LUT ← PFKS([[m_{0,c}]]_LWE, ..., [[m_{n,c}]]_LWE)
[[m_{r,c}]]_LWE ← BAA([[m_{0,c}, ..., m_{n,c}]]_LUT, [[r]]_LWE)
return [[m_{r,c}]]_LWE
```

3.2 Writing operations

Blind Array Add (BAAdd). Introduced in [5], BAAdd is a primitive that enables blind writing in a LUT ciphertext by adding a value to specific position. Given a LUT ciphertext, an encrypted index i, and an encrypted value x, BAAdd adds x to the element at position i while keeping all other elements unchanged. A related operation, Blind Array Assignment, could be implemented by first reading the current value at position i using BAA, subtracting it from $[\![x]\!]_{\text{LWE}}$, and then using BAAdd with the difference. While this would enable arbitrary value assignments, it requires an additional blind rotation.

Algorithm 3: Blind Array Add (BAAdd)

Input : An encrypted index $[\![i]\!]_{\text{LWE}}$
A LUT ciphertext $[\![m_0, \ldots, m_i, \ldots, m_{p-1}]\!]_{\text{LUT}}$
An encrypted value $[\![x]\!]_{\text{LWE}}$
Output: A LUT ciphertext $[\![m_0, \ldots, m_i + x, \ldots, m_{p-1}]\!]_{\text{LUT}}$

$[\![x\delta_0]\!]_{\text{LUT}} \leftarrow PFKS([\![x]\!]_{\text{LWE}})$
$[\![x\delta_i]\!]_{\text{LUT}} \leftarrow BR(-[\![i]\!]_{\text{LWE}}, [\![x\delta_0]\!]_{\text{LUT}})$
return $[\![m]\!]_{\text{LUT}} + [\![x\delta_i]\!]_{\text{LUT}}$

A caveat of this approach is that the $[\![x\delta_i]\!]_{\text{LUT}}$ is most likely misaligned due to the noise present in the rotation index. This affects the frontiers of the redundancy boxes present in LUT ciphertexts as shown in Figure 2. A way to avoid error propagation is to bootstrap the LUT by extracting every message and pack them in a fresh LUT, as described in section 3.4.

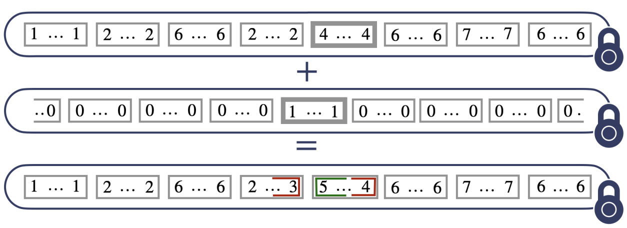

Fig. 2. Illustration of $BAAd([\![4]\!]_{\text{LWE}}, [\![1, 2, 6, 2, 4, 6, 7, 6]\!]_{\text{LUT}}, [\![1]\!]_{\text{LWE}})$ with $p = 8$. The red areas at the boundaries of the redundancy boxes represent errors due to the noise in the LWE encryption of $[\![4]\!]_{\text{LWE}}$. If the noise in the LWE ciphertext were zero, the boxes would be perfectly aligned. However, since we have no control over this noise, except that it does not exceed $(N/2p)$, only the center of the boxes remains accurate.

3.3 Ordering operations

3.3.1 Blind Permutation. The permutation primitive is a fundamental building block in many privacy-preserving application (e.g cloud storage [15], private information retrieval, private set intersection, etc). Blind Permutation is an homomorphic primitive that allows to permute the elements of an encrypted vector without revealing the permutation. It leverages some building blocks of TFHE's bootstrapping operations such as Blind Rotation, Sample Extraction and Public Functional Key Switch. Specifically, the elements to be permuted are in an encrypted LUT and the permutation indices, the destination of each slot, are encrypted as an LWE ciphertext.

The permutation is done by extracting each element of the encrypted LUT as LWEs, then for each extracted element, a Public Functional Key Switch is applied to create new LUTs with the extracted element at its first position. Then we apply a Blind Rotation to the new LUTs using the LWEs representing the permutation indices. And finally, as all the other elements of the LUTS are zeros, we can sum them all to get the permuted LUT. To be sure that we can apply this indefenitely, we apply a Sample Extraction for all the elements and we create a new LWE ciphertext with the permuted elements.

Algorithm 4: Blind Permutation (BP)

Input : A LUT ciphertext $[\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}}$
A permutation vector $([\![\pi_0]\!]_{\text{LWE}}, \ldots, [\![\pi_{p-1}]\!]_{\text{LWE}})$
Output: A permuted LUT $[\![m_{\pi_0}, \ldots, m_{\pi_{p-1}}]\!]_{\text{LUT}}$

// Constants in ciphertexts are trivially encrypted

$[\![res]\!]_{\text{LUT}} \leftarrow [\![0, \ldots, 0]\!]_{\text{LUT}}$
for $i \leftarrow 0$ **to** $p - 1$ **do**
  $[\![m_i]\!]_{\text{LWE}} \leftarrow SE(i, [\![m_0, \ldots, m_{p-1}]\!]_{\text{LUT}})$
  $[\![r_i]\!]_{\text{LUT}} \leftarrow PFKS([\![m_i]\!]_{\text{LWE}}, [\![0]\!]_{\text{LWE}}, \ldots, [\![0]\!]_{\text{LWE}})$
  $[\![r_i]\!]_{\text{LUT}} \leftarrow BR([\![\pi_i]\!]_{\text{LWE}}, [\![r_i]\!]_{\text{LUT}})$
  $[\![res]\!]_{\text{LUT}} \leftarrow [\![res]\!]_{\text{LUT}} + [\![r_i]\!]_{\text{LUT}}$
end
return $[\![res]\!]_{\text{LUT}}$

3.3.2 Blind Sort. In RevoLUT, we implemented two sorting algorithms that leverage one or more of the operations presented in the previous sections. The first one is an homomorphic version of the counting sort algorithm and the second one is another sorting algorithm that we will call *Double Blind Permutation.*

Blind Counting Sort. The idea here is to exploit the known size of the array and range of values that are given. It functions in a similar manner to the classical counting sort algorithm, first building a count table of every entries and then rebuilding the sorted array from that count table.

Double Blind Permutation. The idea behind this approach is that, to sort a vector of p elements within the range $[0, p-1]$, the required permutation is simply the vector itself. Each element of the vector corresponds to its own index in the sorted vector. This concept aligns perfectly with LUT ciphertext, as the range of elements and the number of elements in the vector are identical. Therefore, we can utilize this property to sort a vector of size p by using it as a permutation vector in the Blind permutation algorithm. However, issues arise when the vector contains fewer than p distinct elements. Applying the Blind permutation as explained earlier will result in a sparse vector where the non-sequential elements are separated by 0. To address this issue, we propose using an additional Blind Permutation where the permutation indices are computed on the fly after the first Blind Permutation. The goal of the second permutation is to compact the vector by moving the empty slots (represented as 0) to the end of the vector.

3.4 LUT Bootstrapping

Certain operations in RevoLUT, such as BAAdd (Algorithm 3) and BP (Algorithm 4), can disrupt the redundancy boxes as detailed in Section 3.2 and shown in Figure 2. This disruption occurs because noise in the LWE ciphertext during Blind Rotation can cause the boxes to become misaligned. To address this issue, we can extract each message from the LUT using Sample Extract and repack them into a fresh LUT before the box centers become corrupted. This repacking process ensures that the redundancy boxes in the new LUT ciphertext remain properly aligned.

Algorithm 5: Blind Counting Sort (BCS)

Input : A LUT ciphertext $[\![m_0, \dots, m_{p-1}]\!]_{\text{LUT}}$
Output: A sorted LUT

$[\![count]\!]_{\text{LUT}} \leftarrow [\![0, \dots, 0]\!]_{\text{LUT}}$
for $i \leftarrow 0$ **to** $p-1$ **do**
  // $count_{m_i} \leftarrow count_{m_i} + 1$
  $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \dots, m_{p-1}]\!]_{\text{LUT}})$
  $[\![count]\!]_{\text{LUT}} \leftarrow BAAdd([\![m_i]\!]_{\text{LWE}}, [\![count]\!]_{\text{LUT}}, [1]_{\text{LWE}})$
end
for $i \leftarrow 1$ **to** $p-1$ **do**
  // $count_i \leftarrow count_i + count_{i-1}$
  $[\![x]\!]_{\text{LWE}} \leftarrow BAA([i-1]_{\text{LWE}}, [\![count]\!]_{\text{LUT}})$
  $[\![count]\!]_{\text{LUT}} \leftarrow BAAdd([i]_{\text{LWE}}, [\![count]\!]_{\text{LUT}}, [\![x]\!]_{\text{LWE}})$
end
$[\![res]\!]_{\text{LUT}} \leftarrow [\![0, \dots, 0]\!]_{\text{LUT}}$
for $i \leftarrow p-1$ **to** 0 **do**
  // $count_{m_i} \leftarrow count_{m_i} - 1$
  $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \dots, m_{p-1}]\!]_{\text{LUT}})$
  $[\![count]\!]_{\text{LUT}} \leftarrow BAAdd([\![m_i]\!]_{\text{LWE}}, [\![count]\!]_{\text{LUT}}, [-1]_{\text{LWE}})$
  // $res_{count_{m_i}} \leftarrow m_i$
  $[\![m_i]\!]_{\text{LWE}} \leftarrow BAA([i]_{\text{LWE}}, [\![m_0, \dots, m_{p-1}]\!]_{\text{LUT}})$
  $[\![count_{m_i}]\!]_{\text{LWE}} \leftarrow BAA([\![m_i]\!]_{\text{LWE}}, [\![count]\!]_{\text{LUT}})$
  $[\![res]\!]_{\text{LUT}} \leftarrow BAAdd([\![count_{m_i}]\!]_{\text{LWE}}, [\![res]\!]_{\text{LUT}}, [\![m_i]\!]_{\text{LWE}})$
end
return $[\![res]\!]_{\text{LUT}}$

4 EXPERIMENTAL RESULTS

In this section, we present few experimental results of our Blind operations. All experiments are performed on a computer running Ubuntu 24.04 with an Intel i9-11900KF CPU clocked at 3.5GHz and 64GB of RAM.

p	BAA	BMA	BAAdd	BP	BCS
4	7 ms	20 ms	8 ms	32 ms	116 ms
8	17 ms	56 ms	19 ms	140 ms	462 ms
16	30 ms	120 ms	34 ms	422 ms	1.34 s
32	60 ms	471 ms	70 ms	1.8 s	5.54 s
64	128 ms	2.8 s	162 ms	8.3 s	26.09 s

Table 1: Runtimes (in ms) of some blind operations of RevoLUT. The LUTs contains p elements of $\mathbb{Z}_p$

5 APPLICATIONS OF REVOLUT

RevoLUT's read/write operations on LUT ciphertexts can be used to implement oblivious algorithms. In this section, we present some use cases where RevoLUT can be used to implement privacy-preserving algorithms, especially in the context of outsourcing computation.

5.1 PROBONITE

Proposed in [3], PROBONITE, which stands for Private One-Branch-Only Non Interactive decision Tree Evaluation, is a privacy-preserving algorithm that allows to evaluate decision trees on encrypted data in a context of Machine-Learning as a service. The proposed algorithm minimizes the number of comparisons by only evaluating the relevant branch of the decision tree. It leverages two primitives: *Blind Node Selection* which is based on Private Information Retrieval techniques and *Blind Array Access* presented in this paper. The former is used to select the appropriate branch of the decision tree to evaluate, while the latter is used to fetch the appropriate feature vector to compare to the client's request.

5.2 Private k-NN

In [5], we demonstrate how RevoLUT's Blind Counting Sort (BCS) can be leveraged as a subroutine to implement an efficient top-k algorithm in a tournament style. We demonstrate the effectiveness of this Blind top-k algorithm by implementing a privacy-preserving k-Nearest Neighbor. By using BCS instead of comparison-based sorting methods, we achieve significant performance improvements over the state-of-the-art. Traditional FHE-based sorting approaches rely on TFHE comparators that require additional bits to handle negacyclicity. In contrast, BCS operates directly with the precision of the values treated (*i.e* elements of $\mathbb{Z}_p$), as it uses counting rather than comparisons. This key difference allows our k-NN implementation to process larger batches of data more efficiently while maintaining the same accuracy. This angle of optimization is particularly interesting as it allows more room for the noise in the LWE ciphertexts and thus more leveled operations (addition, scalar multiplication etc..) with the same elements precision.

5.3 Oblivious Turing Machine

In [4], we present a construction of an Oblivious Turing Machine using Blind Matrix Access (BMA). This construction enables clients to securely outsource both their Turing Machine and its computations to an untrusted server. The proposed construction encodes the machine's instructions as a matrix of integers in $\mathbb{Z}_p$ and the tape as a LUT ciphertext, allowing the server to execute the machine's operations without learning anything about the program or its data.

6 CONCLUSION

In this paper, we introduced RevoLUT, a Rust library built upon tfhe-rs that reimagines Look-Up-Tables as first-class objects for homomorphic encryption. Moving beyond their traditional role in function encoding during programmable bootstrapping, we demonstrated how LUTs can serve as versatile data structures enabling efficient oblivious operations. The library provides a comprehensive set of primitives for reading (BAA, BMA), writing (BAAdd), and ordering (BP, BCS) operations on encrypted data, forming a powerful toolkit for implementing privacy-preserving algorithms. Our experimental results demonstrate the practical viability of RevoLUT's operations, with reasonable execution times even for larger LUT sizes. Through concrete applications in the context of Machine Learning and Turing Machine evaluation, we showcased how RevoLUT can be leveraged to build complex privacy-preserving systems.

REFERENCES

[1] 2024. RevoLUT: Rust efficient versatile library for oblivious Look-Up Tables. https://github.com/sofianeazogagh/revoLUT.

[2] Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. 2024. General Functional Bootstrapping using CKKS. Cryptology ePrint Archive, Paper 2024/1623. https://eprint.iacr.org/2024/1623

[3] Sofiane Azogagh, Victor Delfour, Sébastien Gambs, and Marc-Olivier Killijian. 2022. PROBONITE: PRivate One-Branch-Only Non-Interactive decision Tree Evaluation. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Los Angeles, CA, USA) *(WAHC'22)*. Association for Computing Machinery, New York, NY, USA, 23–33. https://doi.org/10.1145/3560827.3563377

[4] Sofiane Azogagh, Victor Delfour, and Marc-Olivier Killijian. 2024. Oblivious Turing Machine. In *2024 19th European Dependable Computing Conference (EDCC)*. IEEE, 17–24.

[5] Sofiane Azogagh, Marc-Olivier Killijian, and Félix Larose-Gervais. 2024. A non-comparison oblivious sort and its application to private k-NN. Cryptology ePrint Archive, Paper 2024/1894. https://eprint.iacr.org/2024/1894

[6] Adrien Benamira, Tristan Guérand, Thomas Peyrin, and Sayandeep Saha. 2023. TT-TFHE: a Torus Fully Homomorphic Encryption-Friendly Neural Network Architecture. *arXiv preprint arXiv:2302.01584* (2023).

[7] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2017. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. Cryptology ePrint Archive, Paper 2017/1114. https://eprint.iacr.org/2017/1114

[8] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. 2018. New techniques for Multi-value input Homomorphic Evaluation and Applications. Cryptology ePrint Archive, Paper 2018/622. https://eprint.iacr.org/2018/622

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*.

[10] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10031)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). 3–33. https://doi.org/10.1007/978-3-662-53887-6_1

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2018. TFHE: Fast Fully Homomorphic Encryption over the Torus. *IACR Cryptol. ePrint Arch.* (2018), 421. https://eprint.iacr.org/2018/421

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[13] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2020. CONCRETE: Concrete operates on ciphertexts rapidly by extending TfhE. In *WAHC 2020-8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*.

[14] Léo Ducas and Daniele Micciancio. 2015. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 617–640.

[15] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. 2014. The Melbourne shuffle: Improving oblivious storage in the cloud. In *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II 41*. Springer, 556–567.

[16] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. 2024. Designing a General-Purpose 8-bit (T)FHE Processor Abstraction. Cryptology ePrint Archive, Paper 2024/1201. https://eprint.iacr.org/2024/1201

[17] 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.