Ripple: Accelerating Programmable Bootstraps for FHE with Wavelet Approximations

Charles Gouert[1⋆], Mehmet Ugurbil[2⋆], Dimitris Mouris[2], Miguel de Vega[2], and Nektarios G. Tsoutsos[1]

[1] University of Delaware
{cgouert, tsoutsos}@udel.edu

[2] Nillion
{memo, dimitris, miguel}@nillion.com

Abstract. Homomorphic encryption can address key privacy challenges in cloud-based outsourcing by enabling potentially untrusted servers to perform meaningful computation directly on encrypted data. While most homomorphic encryption schemes offer addition and multiplication over ciphertexts natively, any non-linear functions must be implemented as costly polynomial approximations due to this restricted computational model. Nevertheless, the CGGI cryptosystem is capable of performing arbitrary univariate functions over ciphertexts in the form of lookup tables through the use of programmable bootstrapping. While promising, this procedure can quickly become costly when high degrees of precision are required. To address this challenge, we propose Ripple: a framework that introduces different approximation methodologies based on discrete wavelet transforms (DWT) to decrease the number of entries in homomorphic lookup tables while maintaining high accuracy. Our empirical evaluations demonstrate significant error reduction compared to plain quantization methods across multiple non-linear functions. Notably, Ripple improves runtime performance for several realistic benchmarks, such as logistic regression and cross-correlation, among others.

Keywords: Applied Cryptography · Homomorphic Encryption · Lookup Tables · Privacy-Enhancing Technologies · Encrypted Computation.

1 Introduction

Cloud computing enables corporations to leverage powerful computational resources, while avoiding the cost and upkeep associated with maintaining local computing infrastructure. Along with numerous benefits, this gives rise to privacy concerns over outsourced data as cloud service providers can possibly view data stored on their servers, and malicious actors have increasingly targeted cloud servers as they can be treasure troves of proprietary information from multiple clients [29,46]. While encryption techniques such as AES can be used to protect data confidentiality, the encrypted data must be static and the cloud cannot apply meaningful processing on ciphertexts (aside from storage). Thus, if a client wants to modify their data, they will have to download the ciphertexts, decrypt them, perform a computation to update the plaintext data, re-encrypt them, and re-upload the result to the cloud.

Homomorphic encryption (HE) is a powerful technique that helps address privacy concerns in cloud computing by allowing computation on encrypted data [19]. With HE, a client can encrypt sensitive data, upload the corresponding ciphertexts to the cloud, have the cloud apply an arbitrary algorithm such as image classification, and then receive a valid encryption of the results, which can only be decrypted with the client's secret key. In this way, the cloud learns nothing about the contents of the input data, intermediate results, or the output of the encrypted computation.

⋆ The first two authors have equal contribution and appear in alphabetical order.
This work has been supported by NSF CAREER award #2239334.

Nevertheless, for many HE schemes such as BGV [5], BFV [18], and CKKS [7], evaluating non-linear functions directly remains impossible, as only addition and multiplication operations can be executed over ciphertexts. Conversely, CGGI [9] allows encrypted lookup tables (LUTs) which allow non-linear functions to be computed exactly. Unfortunately, this operation is quite costly and becomes significantly more expensive as the plaintext modulus increases. Likewise, both the time and memory required to generate the lookup tables scale exponentially with the input's precision, with LUT sizes of 32 bits and larger becoming impractical.[3]

One solution to this problem is to simply quantize the LUT inputs (i.e., reduce the bit width) to lower the number of entries in the lookup table, resulting in faster evaluation and LUT generation times. The reduced precision caused by quantization, however, can negatively impact a wide variety of applications that require high precision. For instance, quantization in deep neural networks can result in non-negligible accuracy loss [26] and thus lead to incorrect classifications. Particularly, errors occurring due to quantization in early layers will propagate to subsequent layers, resulting in an avalanche effect, where the errors are compounded in each layer. Indeed, this effect is not limited to privacy-preserving inference and can happen in any application.

In this work, we propose the *Ripple* framework that offers new efficient techniques for encrypted LUT evaluation. Ripple provides all of the same benefits of quantization in terms of latency reduction while minimizing the accuracy loss resulting from reduced precision and bit widths. Our approach leverages the *discrete wavelet transform* (DWT) [42] to approximate non-linear functions and generate significantly smaller lookup tables while maintaining high accuracy. Moreover, Ripple employs multiple DWT families and introduces bespoke HE-friendly protocols tailored to each family to maximize accuracy and minimize latency. For instance, we find that some DWT families benefit from multiple LUTs while others need only a single LUT evaluation. We apply Ripple to a variety of non-linear functions that are widely used across several domains from machine learning [6,15] to statistics [39], as well as multiple realistic applications for homomorphic encryption [44,24], such as logistic regression inference and edge detection. Our contributions can be summarized as follows:

- We introduce Ripple to construct smaller encrypted LUTs with wavelet techniques without sacrificing accuracy.
- We propose multiple protocols for evaluating wavelet-encoded LUTs in the encrypted domain.
- We implement a suite of commonly adopted non-linear functions and optimize with Ripple, along with a set of representative benchmarks from various domains.

Roadmap: Section 2 provides a brief background on homomorphic encryption and the discrete wavelet transform. Sections 3 and 4 outline our encrypted lookup table methodology based on the DWT, as well as our novel optimizations, while Section 5 introduces a representative set of applications that can benefit from our methodologies. Section 6 presents our experimental evaluations for each application, demonstrating the efficacy of our approach, and Section 7 discussed related works and how they compare to Ripple. Lastly, our concluding remarks appear in Section 8.

2 Preliminaries

2.1 Homomorphic Encryption Overview

The key characteristic of all HE schemes is *malleability*, where ciphertexts can be manipulated to change the underlying plaintext data predictably. All HE schemes can be roughly divided into two categories depending on the primary computational domain: arithmetic-based schemes and Boolean-based schemes.

[3] We empirically observed even 32-bit encrypted LUTs with the state-of-the-art TFHE-rs [47] HE library require approximately 515 GB of RAM and 65 minutes. For reference, 30-bit LUTs took almost 15 minutes requiring over 120 GB.

2.1.1 Arithmetic-Based Schemes

This category consists of schemes that are primarily used for addition and multiplication over ciphertexts that encrypt modular integers or floating point numbers. We note that partial HE schemes such as Paillier [37] and unpadded RSA [40] fall into this category, but only allow for one type of operation (i.e., addition or multiplication) and are therefore only used in niche applications. The other cryptosystems in this category encompass schemes that can be instantiated as both *leveled* and *fully* homomorphic encryption, depending on whether or not a bootstrapping mechanism is added. The BGV [5] and BFV [18] cryptosystems are used for encrypting integers modulo a configurable plaintext modulus. On the other hand, an approximate scheme called CKKS [7] encrypts either complex or floating point numbers and the error that is inherent in homomorphic encryption is treated similarly to floating point arithmetic error.

In fact, this error is integral to all HE schemes that derive their security from the Learning With Errors (LWE) [38] or Ring-LWE (RLWE) [35] problems. Ciphertexts take the form of tuples of high-degree polynomials, and a small amount of random noise is injected into the coefficients upon encryption. The noise compounds as operations are conducted on the data; additively for ciphertext additions and multiplicatively for ciphertext multiplication. If the noise is allowed to grow past a certain threshold, it will begin to corrupt the underlying plaintext message.

This has huge ramifications and there are two primary mechanisms for noise reduction. The first is a computationally efficient procedure called modulus switching that reduces the size of the coefficients of the ciphertext polynomials. Reducing the coefficient modulus also serves to scale down the noise, which essentially reduces it. Aside from noise reduction, this technique will also speed up subsequent operations as the total size of the ciphertext is also reduced. Nevertheless, this technique can only be invoked a finite number of times, and eventually, it will not be possible to reduce the modulus size further.

Implementations that incorporate a *bootstrapping* procedure can reduce the noise indefinitely by homomorphically evaluating the decryption circuit of the cryptosystem (resulting in a ciphertext with lower noise). Also, the bootstrapping procedure regenerates the ciphertext modulus, allowing for more modulus switching operations in between bootstraps. These FHE contexts can evaluate arbitrarily deep arithmetic circuits, but the cost of bootstrapping remains extremely high relative to other HE operations. The overall latency of bootstrapping can range from several seconds to several minutes on a CPU depending on the parameter sets used.

Additionally, this class of schemes is not well-suited for evaluating non-linear operations as the only supported operations are addition and multiplication. For instance, many activation functions in machine learning constructions, like ReLU and sigmoid, can not be directly implemented. Instead, one must use a polynomial approximation that must be balanced to yield an acceptable trade-off between accuracy and latency. A high-degree polynomial can closely mimic the behavior of a non-linear function but requires several homomorphic multiplications and additions to evaluate.

2.1.2 Boolean-based Schemes

Instead of encrypting integers or floating point numbers, this class of schemes encrypts individual bits (or low-precision integers in certain cases). The addition and multiplication primitives are replaced by encrypted gate operations, such as `AND`, `OR`, and `NOT` gates. In practice, most logic gates are implemented as a series of linear operations between ciphertext polynomials followed by a *functional bootstrap*, which serves to scale the output to the expected value. Because Boolean schemes can support all standard logic gates, they are capable of executing arbitrary algorithms.

The DM cryptosystem [16] was the first of these schemes and is meant to operate solely as an FHE construction. Compared to the relatively slow bootstrapping speeds of the arithmetic-based schemes, DM is capable of evaluating a bootstrap in less than a second. A successor to this scheme, known as CGGI [8], boasts an even faster bootstrapping speed of approximately 10 milliseconds on a CPU. Both of these schemes lack a mechanism to enable *batching*, but what they lack in throughput, they make up for in terms of lower latency operations.

Additionally, compared to the prior class of HE schemes, Boolean schemes do not need to utilize polynomial approximations. Indeed, non-linear operations can be implemented exactly as a Boolean circuit. As

an example, the ReLU activation function is directly mapped to a multi-bit comparator circuit followed by a multiplexer. However, this may require a larger number of Boolean gates and the majority of gate types require at least one bootstrapping operation, resulting in relatively high latency for large circuits. Prior frameworks, such as the Google Transpiler [21], ArctyrEX [22], and HELM [23], exploit the inherent circuit-level parallelism to reduce the latency of circuit evaluation. Even still, the performance of these frameworks is limited by the critical path (or greatest depth) of the homomorphic circuit. Additionally, all three approaches rely on logic and/or high-level synthesis methods to convert input programs to optimized Boolean circuits, which results in very high pre-processing cost for non-trivial applications.

Alternatively, with proper parameter selection, Boolean schemes can also encrypt low-precision integers instead of bits. In the case of CGGI, this encoding allows for ciphertext addition and multiplication with a public constant, but not multiplication between two ciphertexts (distinguishing this mode from arithmetic-based schemes). Notably, it still retains the functional bootstrap and this can be utilized to evaluate N:N lookup tables, as discussed in the following subsection. This approach combines some of the key strengths of both arithmetic-based schemes and the Boolean mode of operation in the form of natively supported multi-bit arithmetic, as well as a mechanism for exactly evaluating non-linear functions.

The primary challenge is the restriction on the size of the underlying plaintext space, but this can be overcome by representing high-precision plaintext values as vectors of ciphertexts, where each encryption encodes a low-precision chunk of the original message. This very methodology is employed in the TFHE-rs library [47] in two different ways: a Chinese remainder theorem (CRT) method and a radix method. The former involves generating multiple residues by reducing an input message by a series of co-prime bases and encrypting each residue as a separate ciphertext. Upon decryption, the residues are combined to form the final higher-precision result. The second method involves decomposing the input data into a series of digits, each of which is decrypted individually. Ripple utilizes the latter approach as it provides a convenient way to truncate ciphertexts, which is an integral operation in the DWT protocols explained in the following section. Truncating the digits of a ciphertext array encoded with the radix decomposition can be done with negligible latency overhead as no FHE operations are required.

2.2 Programmable Bootstrapping (PBS)

A crucial feature of the DM and CGGI FHE cryptosystems is the *functional bootstrap*, which takes advantage of the programmability of the bootstrapping algorithm employed in these schemes. A polynomial with crafted coefficients that encodes the set of desired output messages is rotated by an encrypted value and the first encrypted coefficient corresponding to the constant term of the polynomial is extracted. These two procedures, called *blind rotation* and *extraction*, form the core bootstrapping steps.

By encoding chosen lookup table (LUT) entries in the coefficients of the polynomial to be rotated, one can evaluate a LUT T over a ciphertext. Essentially, this can be done by rotating the LUT polynomial by an encrypted amount (corresponding to the input ciphertext) and extracting the entry corresponding to the constant term. The result is a valid encryption that encodes the mapping from a LUT input to a desired LUT output. Thus, it allows computing arbitrary univariate functions by evaluating a function in the plaintext domain across all possible inputs and encoding them in the polynomial utilized during bootstrapping. This generalized bootstrapping technique is called *programmable bootstrapping (PBS)* [11,34]. Although the LUT needs to be relatively small to maintain efficient cryptographic parameter sets, it has two main advantages. First, it can encode any arbitrary univariate function, and second, it leads to a significant performance boost as it replaces expensive operations that otherwise would require multiple additions and multiplications.

2.3 The Discrete Wavelet Transform (DWT)

A Discrete Wavelet Transformation (DWT) is a process of splitting a discretely sampled signal into two parts: the approximation and the detail coefficients [42]. The former is half of the size of the original signal but encompasses the most interesting parts of it, while the latter contains information about the error incurred by the approximation coefficients. When combining both the detail coefficients and the approximation, one

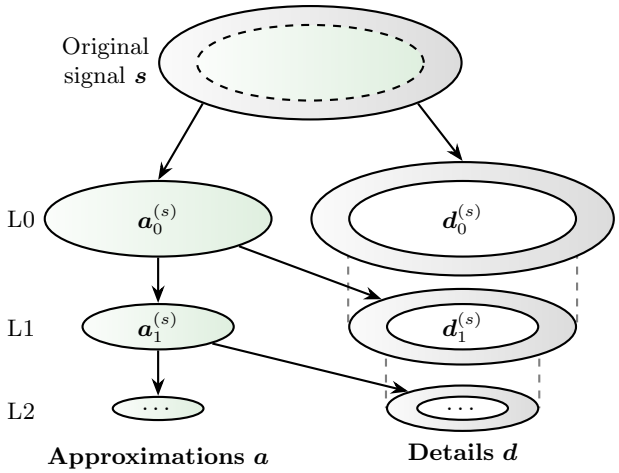


Fig. 1: Two DWT iterations. The signal $\boldsymbol{s}$ can be represented by the approximation $\boldsymbol{a}^{(s)}$ (green) and the details $\boldsymbol{d}^{(s)}$ (gray).

can reconstruct the original signal. Fig. 1 illustrates the original signal on top and three applications of the DWT. In the first level (L0), we deconstruct the signal on top to approximation ($\boldsymbol{a}_0^{(s)}$) and detail coefficients ($\boldsymbol{d}_0^{(s)}$). Then, we can repeat the same process by treating the L0 approximation as a new signal and thus get new approximation ($\boldsymbol{a}_1^{(s)}$) and detail coefficients ($\boldsymbol{d}_1^{(s)}$) at level 1, and so on. In Fig. 1, notice that given the L2 approximation and detail coefficients along with the L1 and L0 detail coefficients, it is sufficient for recovering the original signal.

There exist multiple wavelet families such as the Daubechy (Db) wavelets and the Biorthogonal wavelets. A special case of the Db wavelets is the Db-1 – or *Haar* – wavelet. The core idea in all families is a matrix multiplication with a constant matrix W, where W differs based on the family. Haar uses an orthogonal matrix to obtain the approximation coefficients, which are calculated by averaging every two consecutive points of the original signal. Biorthogonal, on the other hand, relies on two different matrices, where the transpose of one matrix is the inverse of the other. We delve more into the details of both wavelet families in Section 3.

2.3.1 Haar DWT

Haar applies a linear transformation to the input signal and generates the approximation and detail coefficients. Starting with a signal $\boldsymbol{s} = [s_0, s_1, \ldots, s_{2N-1}]$ of length $2N$, the Haar DWT generates the approximation coefficients $\boldsymbol{a} = [a_0, \ldots, a_{N-1}]$ and the detail coefficients $\boldsymbol{d} = [d_0, \ldots, d_{N-1}]$, each with N entries. More specifically, the approximations are generated as $a_k = (s_{2k} + s_{2k+1})/2$, while the details are generated by $d_k = (s_{2k} - s_{2k+1})/2$ for $k \in [0, \ldots, N)$.

It is easy to see that when the Haar DWT is applied to a one-hot vector, it results in yet another one-hot vector, albeit scaled. The index of the non-zero value in the new vector is in fact the old index divided by 2, which is akin to removing the last bit of the index. Therefore, we can simply truncate the value to get rid of the least significant bits and end up with the value that we want to do the lookup at. In effect, we have manually applied the Haar DWT transform to the one-hot vector using only truncation.

2.3.2 Biorthogonal DWT

The linear transformation in Haar can be seen as a matrix multiplication with some public orthogonal matrix.[4] The Biorthogonal DWT on the other hand, requires two different matrices, where the transpose of one matrix is the inverse of the other, i.e., $M_1^T = M_2^{-1}$. In this case, M_1 is used for the decomposition of the signal, while M_2 is used for the reconstruction.

In Haar, we observe that the points in the approximation coefficient are averaged from the input signal S and the details are complementary in order to be able to recover S. Conversely, in Biorthogonal wavelets, the approximations and details are computed with weighted averages [42]. Since the Biorthogonal wavelets have more non-zero filters, when the transform is applied to a one-hot vector, the result is not a one-hot vector. Even so, we can manually calculate the resulting vector as it is a weighted average of the two consecutive values starting at the most significant bits of the original index. The weights depend on the least significant bits of the original index; hence, by splitting the original index into the MSBs and LSBs, we can calculate the transformed vector.

3 The Ripple Framework

As mentioned in Section 2.2, a key feature of the CGGI cryptosystem is the ability to evaluate a lookup table with the programmable bootstrapping mechanism. Complex non-linear functions can now be encoded as LUTs and evaluated homomorphically, eliminating the need to perform expensive polynomial approximations. Unfortunately, as the size of the LUT grows, this technique becomes prohibitively expensive (recall footnote 3), and thus many non-linearities are impossible to evaluate in applications that require high precision.

We address this challenge by utilizing the DWT to reduce the size of LUTs without sacrificing correctness. Ripple is the first framework to explore wavelet approximations for FHE as a way to accelerate programmable bootstrapping. Our key observation is that if we apply the DWT to signals that represent smooth functions (e.g., logarithm, square root, sigmoid, etc.), then the detail coefficients are relatively small compared to the approximation coefficients. This means that our approximation is sufficient to represent the original signal and we can completely disregard the detail coefficients while maintaining a minimal error relative to the original function. Utilizing this observation, we can zero out the details in Fig. 1, and by just applying the DWT a single time, we can halve the size of the LUT. This signal might still be quite big, so we can repeat the same process and half the LUT size even further. Of course, as this is an approximation, the smaller the LUT size, the higher the error we might have. With Ripple, however, these errors are marginal as we show in Section 6.

Ripple's Key Observation. The core idea behind Ripple relies on the orthogonality or biorthogonality of the DWT transform applied to the inner product. Consider the fact that the inner product between two vectors $\boldsymbol{v}$ and $\boldsymbol{u}$ is equal to the product between the transpose of the first vector and the second, i.e., $\langle \boldsymbol{v}, \boldsymbol{u} \rangle = \boldsymbol{v}^T \cdot \boldsymbol{u}$. As described in Section 2.3, the DWT of both $\boldsymbol{v}$ and $\boldsymbol{u}$ is a multiplication with a matrix W, which we can view as $\mathsf{DWT}(\boldsymbol{v}) = W \cdot \boldsymbol{v}$. This results in a vector $\left[\frac{\boldsymbol{a}^{(v)}}{\boldsymbol{d}^{(v)}}\right]$, where $\boldsymbol{a}^{(v)}$ and $\boldsymbol{d}^{(v)}$ represent the approximation and detail coefficients of $\boldsymbol{v}$, respectively. Similarly for $\mathsf{DWT}(\boldsymbol{u}) = W \cdot \boldsymbol{u}$ we get $\boldsymbol{a}^{(u)}$ and $\boldsymbol{d}^{(u)}$. Observe that using orthogonality,

$$\begin{aligned} \langle \mathsf{DWT}(\boldsymbol{v}), \mathsf{DWT}(\boldsymbol{u}) \rangle &= \langle W \cdot \boldsymbol{v}, W \cdot \boldsymbol{u} \rangle \\ &= (W \cdot \boldsymbol{v})^T \cdot W \cdot \boldsymbol{u} \\ &= \boldsymbol{v}^T \cdot W^T \cdot W \cdot \boldsymbol{u} \\ &= \boldsymbol{v}^T \cdot I \cdot \boldsymbol{u} = \boldsymbol{v}^T \cdot \boldsymbol{u} = \langle \boldsymbol{v}, \boldsymbol{u} \rangle. \end{aligned} \tag{1}$$

We remark that the inner product of two vectors is equal to the sum of the inner product of the approximation coefficients and the inner product of the detail coefficients of the DWT transforms of the vectors, as

[4] An orthogonal matrix M has the property that $MM^T = I$, where I is the identity matrix. A matrix M is orthogonal if its transpose (M^T) is equal to its inverse (M^{-1}).

follows:

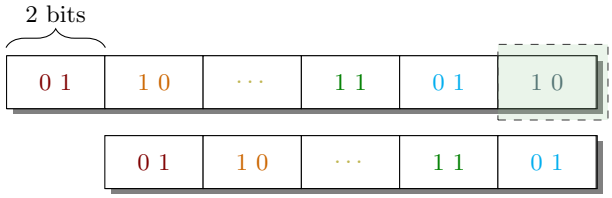
$$(W \cdot \boldsymbol{v})^T \cdot W \cdot \boldsymbol{u} = [\frac{\boldsymbol{a}^{(v)}}{\boldsymbol{d}^{(v)}}]^T \cdot [\frac{\boldsymbol{a}^{(u)}}{\boldsymbol{d}^{(u)}}]$$
$$= \boldsymbol{a}^{(v)T} \cdot \boldsymbol{a}^{(u)} + \boldsymbol{d}^{(v)T} \cdot \boldsymbol{d}^{(u)}.$$

In Ripple, we represent $\langle \boldsymbol{v}, \boldsymbol{u} \rangle$ as $\langle \boldsymbol{a}^{(v)}, \boldsymbol{a}^{(u)} \rangle + \langle \boldsymbol{d}^{(v)}, \boldsymbol{d}^{(u)} \rangle$ which is approximately equal to the inner product of their respective approximation coefficient vectors $\boldsymbol{a}^{(v)}$ and $\boldsymbol{a}^{(u)}$. By dropping the detail coefficients, we get the approximation of the original inner product via the inner product of the approximation coefficients, which is key to our proposed lookup methodology. Thus, $\langle \boldsymbol{v}, \boldsymbol{u} \rangle = \langle \mathsf{DWT}(\boldsymbol{v}), \mathsf{DWT}(\boldsymbol{u}) \rangle \approx \langle \boldsymbol{a}^{(v)}, \boldsymbol{a}^{(u)} \rangle$.

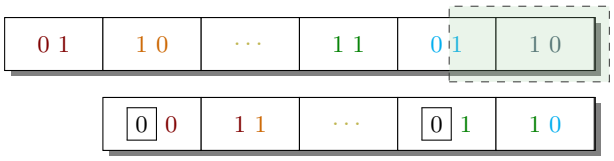
This works nicely for orthogonal DWTs, but for the Biorthogonal DWT, we need extra considerations. Instead of applying the same transformation to both $\boldsymbol{v}$ and $\boldsymbol{u}$, we have to apply the decomposition matrix to one, while applying the reconstruction matrix to the other. Since applying the reconstruction matrix is equivalent to applying the decomposition matrix with filters flipped, this can also be thought of as a decomposition. Then, by biorthogonality of these matrices, the same observation holds and $\langle \boldsymbol{v}, \boldsymbol{u} \rangle = \langle M_1 \cdot \boldsymbol{v}, M_2 \cdot \boldsymbol{u} \rangle \approx \langle \boldsymbol{a}^{(v)}, \boldsymbol{a}^{(u)} \rangle$.

Applying our Observation to the Encrypted Domain. In Eq. (1), we can view $\boldsymbol{v}$ as a one-hot vector where the non-zero value is at the index of the ciphertext and $\boldsymbol{u}$ as a public LUT T'. Then, the inner product $\langle \boldsymbol{v}, \boldsymbol{u} \rangle$ will yield the lookup value in table T' at the non-zero index of $\boldsymbol{v}$, which is the lookup value at the ciphertext. In particular, $\langle \boldsymbol{a}^{(v)}, \boldsymbol{a}^{(u)} \rangle$ will be an approximation of this lookup. Notably, the approximation of the lookup table is easy to calculate since it is in plaintext, while we need a way to efficiently calculate the approximation vector of the one-hot vector that represents the ciphertext. Fortunately, it turns out that this approximation vector can be calculated by a weighted sum of lookups.

Of course, the aforementioned technique is not practical in HE, but programmable bootstrapping (PBS) can be leveraged for this exact purpose. Starting with a ciphertext x' we can evaluate the LUT T' on x' homomorphically and obtain ciphertext $y' = T'(x')$. The novelty of Ripple here lies in the fact that x' has fewer bits than x and T' has fewer entries than T, while y' is a very close approximation to $y = T(x)$. Ripple focuses on the two most popular DWT families: Haar and Biorthogonal, which are both viable and constitute a tradeoff between accuracy and latency (as discussed in Section 6). In the following subsections, we describe how Ripple formulates and evaluates encrypted lookup tables with each of the two DWT families.



(a) Truncation of multiples of 2-bits.



(b) Truncation of odd bit sizes.

Fig. 2: Two scenarios of truncating the least significant bits of a radix ciphertext, where each digit encrypts two bits of data. With this configuration, it is far more efficient to truncate an even number of bits, while truncating an odd number of bits is computationally expensive.

Efficient Truncation. The speedups gained from the DWT techniques revolve around reducing the total size of the LUT to be evaluated. However, in the encrypted domain, it is impossible to reduce the number of LUT entries without also decreasing the underlying plaintext modulus. Simply shifting to isolate MSBs and LSBs for LUT evaluation will result in virtually no speedup as the total ciphertext size and dimensions will be identical to the original, and only the underlying message is manipulated. However, we can leverage the radix decomposition mode of TFHE-rs (where encrypted values are represented as vectors of ciphertexts) to effectively reduce the number of ciphertexts instead of shifting to isolate bits. An example of this approach is shown in Fig. 2a. If each digit of the encrypted value can encode two bits of information, a truncation can be performed by simply discarding digits; this approach is essentially free as no HE operations are required. At the same time, the total number of ciphertexts encompassing the encrypted value is reduced and the overall number of PBS required to evaluate a full lookup table is decreased. The key restriction of this approach is that the number of bits to be truncated must be a multiple of the bit-width of each digit. If this is not the case, as in Fig. 2b, the entire ciphertext array must be shifted, requiring multiple PBS operations and non-negligible overheads. After the shift is done, the blocks that encode all zeros (i.e., the left-most block in Fig. 2b) can be discarded. We observe that it is possible to avoid this inefficient truncation through careful parameterization of the DWT, as discussed in each of the following subsections.

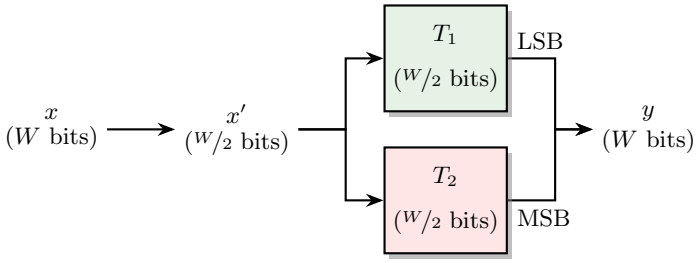


Fig. 3: Haar DWT LUT example approximating $y = T(x)$ by computing $y = T_1(x') \,\|\, T_2(x')$, where $x' = x \gg W/2$.

3.1 PBS with Haar DWT

Let W be the total bit width of our input radix-decomposed ciphertext vector. Naturally, we can represent the function that we want to approximate as a LUT T that maps k-bit inputs to k-bit outputs. For simplicity, we can assume that $k = W$. Creating a LUT of 2^W entries, however, is not always feasible. For instance, $W = 64$ requires generating and storing 2^{64} LUT entries, which is completely impractical. A straightforward option is to truncate our input ciphertexts by J bits and limit T to only have 2^{W-J} entries and operate over $W - J$ bit inputs and outputs. As it turns out, this approximation incurs high errors and is not sufficient for most applications.

Ripple takes a different approach: First, we apply the Haar DWT over the public LUT T iteratively J times by dropping the detail coefficients to end up with an approximation T' of our original function. The new table T' now operates over $W - J$ bits. Observe, however, that our input ciphertext vector is still W-bits long. To index T', Ripple truncates the vector to encode $W - J$ bits so it can be used during PBS to index the LUT. As long as $W - J$ bits is a multiple of the size of our radix digits, the truncation is free. Conveniently, this is the index needed for the approximation vector $\boldsymbol{a}^{(v)}$. This results in an output ciphertext vector also encoding $W - J$ bits.

As we started with an W-bit input ciphertext vector, after approximating the function we need to end up with a W-bit output as well. To do so, we repeat the same process twice by building two LUTs (T_1 and T_2); one for the LSBs and one for the MSBs. In both cases, the truncated $W - J$ bit input ciphertext vector is used to index the LUT. This is illustrated in Fig. 3, where $J = W/2$. Note that each of T_1 and T_2 has

$W/2$ bits and thus takes approximately half the time to be evaluated compared to a W-bit LUT (which is not even possible to practically create for large W). Additionally, both tables can be evaluated in parallel and finally, the two outputs can be concatenated to get the final encrypted result encoding W bits of data.

We remark that, in certain functions, we only need to evaluate the LSB table and we can avoid evaluating T_2 altogether. For instance, any function where the output can fit in less than half the bit width, such as the square root or Sigmoid activation function, only needs a single LUT evaluation.

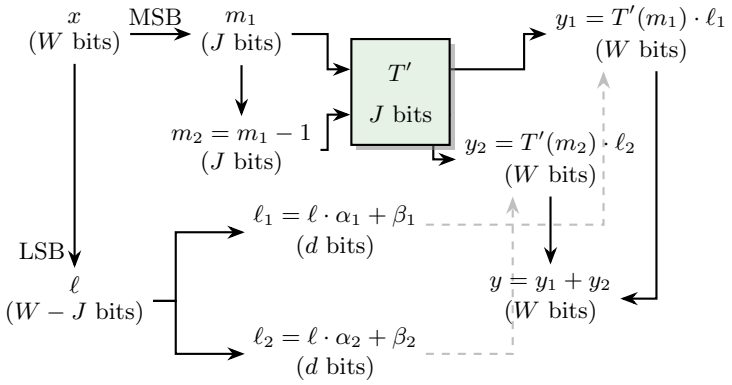


Fig. 4: Biorthogonal DWT LUT example approximating $y = T(x)$ by computing $y = T'(m_1) \cdot \ell_1 + T'(m_2) \cdot \ell_2$, where $m_1 = x \gg (W - J)$, $m_2 = m_1 - 1$, $\ell = x \mod 2^{W-J}$, $\ell_1 = \ell \cdot \alpha_1 + \beta_1$, and $\ell_2 = \ell \cdot \alpha_2 + \beta_2$ with $\alpha_1, \alpha_2, \beta_1, \beta_2$ being public scalars.

3.2 PBS with Biorthogonal DWT

The Haar DWT is quite efficient since its approximation vector is again one-hot and hence only has a single approximation that needs to be looked up; however, this benefit also comes at a limitation. Namely, all the points that have the same most significant bits evaluate to the same value. The Biorthogonal DWT overcomes this challenge by making use of the least significant bits of the input ciphertext, as well as the most significant bits. This comes at the cost of doing two lookups followed by two multiplications and an addition, but results in more accurate approximations. In the end, we get significantly better compression, in fact, the same compression used in JPEG2000 [41].

The Biorthogonal DWT follows a similar approach as the Haar DWT, yet incurs more operations as its linear transformation computes weighted averages. Contrary to Haar where we had to evaluate two separate LUTs, in Biorthogonal wavelets we need to evaluate a single LUT (T') across two different indices. We observe that this can also be done using two LUTs, where, as an optimization, the second one equals the first one shifted by an index.

Fig. 4 demonstrates how Ripple approximates an LUT with Biorthogonal wavelets. First, we extract the most significant bits (MSBs) from the input to be used for the lookup, and the least significant bits (LSBs), which will be used to combine the lookups via their weighted average. Starting with a W-bit input x, we split it into two parts of J and $W - J$ bits, which we call m_1 and ℓ, respectively. Here, J is equal to the depth of the DWT applied. The former (m_1) represents the J MSBs of x and it serves two purposes. First, from m_1, we create m_2 as $m_1 - 1$. Then, we use both m_1 and m_2 to index two consecutive entries of the DWT-encoded LUT and we end up with ciphertexts $T'(m_1)$ and $T'(m_2)$. The latter (ℓ) represents the $W - J$ LSBs of x and is used to compute two linear expressions with public constant values α_1, α_2, β_1, and β_2 as $\ell_1 = \ell \cdot \alpha_1 + \beta_1$, and $\ell_2 = \ell \cdot \alpha_2 + \beta_2$. These values come from the non-zero entries of the approximation coefficients vector $\boldsymbol{a}^{(v)}$: $\alpha_1 = -1$, $\alpha_2 = 1$, $\beta_1 = 2^J$, and $\beta_2 = 0$. Lastly, we multiply $T'(m_1)$ by ℓ_1 and $T'(m_2)$ by ℓ_2 computing

the inner product in the DWT domain, so we go back to W bits and sum the two ciphertexts together to get our final output.

We are usually able to compress the LUT for the Biorthogonal DWT into J bits as these values don't have to cover the entire output range of the function, unlike Haar. However, when this is not possible, we employ a similar method of evaluating multiple LUTs in parallel with the same J-bit inputs and combine the J-bit outputs to form a higher bit-width result.

4 Function-centric Compression

In this Section, we investigate two optimizations to further reduce the LUT sizes of specific classes of non-linear functions; namely, we propose optimizations for symmetrical functions and functions where the complex non-linearity converges to some value outside a certain interval.

4.1 Symmetrical Functions

First, we exploit the symmetry of certain functions to further reduce the LUT size. This class includes common functions in machine learning, such as reciprocal, sigmoid (σ), hyperbolic tangent (tanh), and the error function (erf). More formally, these functions exhibit the following property: $\frac{1}{-x} = -\frac{1}{x}$, $\sigma(-x) = 1 - \sigma(x)$, $\mathsf{tanh}(-x) = -\mathsf{tanh}(x)$ and $\mathsf{erf}(-x) = -\mathsf{erf}(x)$. Therefore, if we know the sign of the input, we can evaluate the function strictly in the positive domain and then use this intermediate result to calculate the actual value by taking into account the sign of the input.

To apply this technique, the function must exhibit symmetry around zero, but we note that any symmetric function can be shifted to exhibit this required symmetry. For example, sigmoid becomes symmetric around zero after it is moved down on the y-axis by 0.5: $\sigma(-x) - 0.5 = -(\sigma(x) - 0.5)$. We stress that this technique easily generalizes to any symmetric function and reduces the size of the LUT by $2\times$ as only half the domain needs to be evaluated.

In general, we call a function $f(x)$ symmetric around the symmetry point $(x_{\mathtt{sym}}, y_{\mathtt{sym}})$ if $f(x_{\mathtt{sym}} - x) - y_{\mathtt{sym}} = c_{\mathtt{sym}} \cdot (f(x_{\mathtt{sym}} + x) - y_{\mathtt{sym}})$, for symmetry constant $c_{\mathtt{sym}}$. The symmetry constant defines the symmetry relationship; for instance if $c_{\mathtt{sym}} = 1$ then the function is reflected along the y-axis (like $f(x) = x^2$) while if $c_{\mathtt{sym}} = -1$ then the function is reflected along the line $y = -x$ (like $f(x) = \mathsf{tanh}(x)$). It is easy to see that, given $f(x)$ at some value $x_{\mathtt{sym}} + \delta$, we can compute f at $x_{\mathtt{sym}} - \delta$ simply by noting:

$$\begin{aligned} f(x_{\mathtt{sym}} - \delta) &= f(x_{\mathtt{sym}} - \delta) - y_{\mathtt{sym}} + y_{\mathtt{sym}} \\ &= c_{\mathtt{sym}} \cdot (f(x_{\mathtt{sym}} + \delta) - y_{\mathtt{sym}}) + y_{\mathtt{sym}} \\ &= c_{\mathtt{sym}} \cdot f(x_{\mathtt{sym}} + \delta) + (1 - c_{\mathtt{sym}}) \cdot y_{\mathtt{sym}}. \end{aligned}$$

Fig. 5 demonstrates a function with $x_{\mathtt{sym}}, y_{\mathtt{sym}}$ and $c_{\mathtt{sym}} = -1$. Notice that for sigmoid, we have $x_{\mathtt{sym}} = 0, y_{\mathtt{sym}} = 0.5$ and $c_{\mathtt{sym}} = -1$, hence we get $\sigma(-x) = -\sigma(x) + 1$. Further, if we set $g(x) = f(x_{\mathtt{sym}} + x)$ and use the absolute value $|x - x_{\mathtt{sym}}|$ and $x - x_{\mathtt{sym}} < 0$:

$$\begin{aligned} g(x) =& (1 + (x - x_{\mathtt{sym}} < 0) \cdot (c_{\mathtt{sym}} - 1)) \cdot g(|x - x_{\mathtt{sym}}|) \\ &+ (x - x_{\mathtt{sym}} < 0) \cdot (1 - c_{\mathtt{sym}}) \cdot y_{\mathtt{sym}}. \end{aligned}$$

Note that this equation holds for all values x whether positive or negative. Thus, we build our LUT based on $g(x)$; we can use this equation to evaluate f at any value x.

4.2 Convergent Functions

Another class of functions that can be optimized are those that are nearly a piecewise polynomial function. This way we can evaluate polynomials and support them with a LUT for the complex parts.

A special case of this includes functions that are nearly constant outside a certain bounded interval. Coincidentally, sigmoid, tanh, and erf are also prominent examples of this. For instance, sigmoid is nearly 1 above a threshold of 8 and nearly 0 below -8 with less than 0.0003 of maximum difference. For tanh, the maximum difference is 0.0003 outside $[-4, 4]$, while for erf the difference is $1.54 \cdot 10^{-8}$ outside the same interval as tanh.

Given a function $f(x)$ and an interval $[x_{\texttt{left}}, x_{\texttt{right}}]$, we say this function is convergent if there exists a piecewise polynomial $p(x)$ such that for $x \notin [x_{\texttt{left}}, x_{\texttt{right}}]$, $|f(x) - p(x)| < \epsilon$ for a small constant ϵ. To approximate such convergent functions, we can check whether the input is inside the interval and select the lookup value $T(x)$ or the polynomial value accordingly as:

$$f(x) \approx (x \in [x_{\texttt{left}}, x_{\texttt{right}}]) \cdot \texttt{T}(x) + p(x).$$

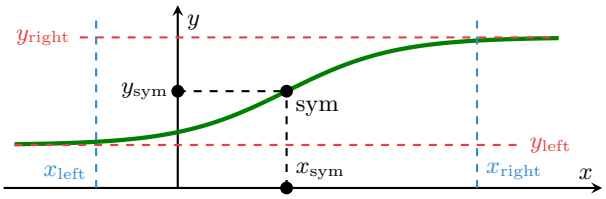


Fig. 5: Sigmoid like function where the symmetry point is x_{sym} and y_{sym}. x_{left} and x_{right} represent the bounds out of which the function converges to y_{right} and y_{left}, respectively.

4.3 Convergent Symmetrical Functions

Bringing together optimizations for symmetrical and convergent functions, we devise a function evaluation protocol in Alg. 1 (Note that only $f(x) - p(x)$ needs to be symmetric). We start by extracting the most significant bit, which is the sign bit of `ct` (called `ltz` for "less than zero" since this bit is 0 for positive values and 1 for negative). Then we calculate the sign as -1 (if `ltz` is 1) or 1 (if `ltz` is 0) and the absolute value (`abs` in line 5 in Alg. 1) of the `ct` by multiplying it by the `sign`. Next, we use the LUT method to look up the function value at `abs`. Note that this could be the desired DWT LUT and involves truncation and multiplication operations. We bring it all together using the symmetry of the function (in line 7 in Alg. 1). Next, we check if the `abs` is in the LUT domain. Notice that since the function is symmetric, we only have to check one end of $[x_{\texttt{left}}, x_{\texttt{right}}]$, which we call $\texttt{threshold} = x_{\texttt{right}} - x_{\texttt{sym}}$. Then we evaluate the polynomial approximation at `abs`. Finally, we merge the results using the convergence property.

For example, to evaluate sigmoid, we define the functions:

$$\begin{aligned} \text{PBS}(\texttt{abs}) &= \sigma(\texttt{abs}) \text{ for } \texttt{abs} \in [0, 8] \\ \textsc{Compare}(\texttt{abs}, \texttt{threshold}) &= \texttt{abs} < \texttt{threshold} \\ \textsc{Poly}(\texttt{abs}, \texttt{ltz}, \texttt{check}) &= (1 - \texttt{check}) \cdot (1 - \texttt{ltz}). \end{aligned}$$

Then we call Evaluate with $x_{\texttt{sym}} = 0, y_{\texttt{sym}} = 0.5$ and $c_{\texttt{sym}} = -1$. On the other hand, for a non-convergent symmetrical function like reciprocal $f(x) = \frac{1}{x}$, we set $\textsc{Compare}(\dots) = 1$ and $\textsc{Poly}(\dots) = 0$.

5 Real-World Applications

In this Section, we introduce the benchmarks utilized in our evaluation of Ripple. We provide specific methodologies to concretely showcase how Ripple can be incorporated into each of these various use-cases, which encompass several different problem domains.

Algorithm 1 LUT Evaluation for Symmetric Convergent Functions

Public Inputs: `func` ▷ Function to be evaluated.
`xsym` ▷ X-coordinate of the symmetry point.
`ysym` ▷ Y-coordinate of the symmetry point.
`csym` ▷ Symmetry coefficient.
`threshold` ▷ LUT boundary.
Private Input: `ct` ▷ Encrypted input value.

1: **procedure** EVALUATE(`func`, `xsym`, `ysym`, `csym`, `ct`)
2: `ct` ← `ct` - `xsym` ▷ Shift to symmetry point.
3: `ltz` ← EXTRACTSIGN(`ct`) ▷ Get the sign bit.
4: `sign` ← 1 - 2 · `ltz` ▷ Calculate the sign as -1 or 1.
5: `abs` ← `sign` · `ct` ▷ Compute the absolute value.
6: `eval` ← PBS(`abs`) ▷ The LUT evaluation on `abs`.
7: `sym` ← $(1 + \texttt{ltz} \cdot (\texttt{csym} - 1)) \cdot \texttt{eval} + \texttt{ltz} \cdot (1 - \texttt{csym}) \cdot \texttt{ysym}$
8: `check` ← COMPARE(`abs`, `threshold`) ▷ Is abs in LUT domain?
9: `poly` ← POLY(`abs`, `ltz`, `check`) ▷ Polynomial evaluation.
10: **return** `check` · `sym` + `poly`

5.1 Logistic Regression

Logistic Regression (LR) is a widely studied application in FHE from genome-wide association studies [30] to more generic applications [44,24] such as natural language processing [2]. This machine learning construction is particularly well-suited to binary classification problems and is akin to a single-layer neural network with a sigmoid activation. Overall, LR poses a challenge to run homomorphically as the whole benchmark is neither well suited to arithmetic or Boolean schemes. The fully-connected layer, which involves multiplying inputs with a set of trained plaintext weights and then accumulating the products, is efficient to compute with an arithmetic-based scheme but not a Boolean-based scheme. On the other hand, the Sigmoid activation function is non-linear, and therefore arithmetic schemes are not well equipped to evaluate it, while Boolean-based schemes are suited for it. The LUT-based approach that Ripple employs proves a great option for this benchmark, as the multi-bit encoding allows for efficient addition and multiplication while also allowing the sigmoid to be implemented directly with a programmable bootstrap. Prior works, which utilize arithmetic-based schemes, employ a Taylor series expansion of the sigmoid function and evaluate the first terms (e.g., $\sigma(X) = 1/2 + X/4 + (X^3)/48 + \dots$) [31] by upscaling the sigmoid to $\sigma(X) = 24 + 12X + X^3 + \dots$ and relying on the client to compute the scaled-down final result. Using more terms in the Taylor series increases both the latency as well as the noise growth due to the increasing numbers of additions and multiplications, but yields higher accuracy.

In Ripple, we take a different approach and replace the polynomial approximations with DWT-encoded lookup tables that compute directly the sigmoid activation function. This way, the end-to-end application is evaluated solely on the server side (i.e., without client-side scaling). First, the server pre-generates the lookup tables needed for the evaluation of the sigmoid activation. We note that this is a one-time preprocessing cost and the generated LUT can be re-used an arbitrary number of times for inference. The fully connected layer is implemented as a parallel series of multiplications between the plaintext weights and encrypted inputs, which is efficient in the CGGI cryptosystem. We then evaluate the non-linear sigmoid function directly with a DWT LUT construction and send the classification result back to the client. The client decrypts the result, which represents the probability that the encrypted input belongs to the first class.

For our implementation, we utilize the Palmer penguin dataset [27], where each input consists of eight attributes that correspond to physical characteristics of penguins (e.g., bill length, flipper length, body mass, etc.). The goal is to classify the inputs into the correct species of penguin: Adelie, Gentoo, or Chinstrap. Since logistic regression is particularly well-suited for binary classification, we remove entries in the dataset corresponding to the Chinstrap species.

Notably, an emergent way to do LR inference with FHE is to use the CKKS cryptosystem [7] for the fully connected layer and then scheme switch to CGGI to do the truncation and the DWT LUT. This would utilize

the best of both worlds and achieve fast additions and multiplications under CKKS and DWT-encoded PBS under CGGI. Unfortunately, the current state-of-the-art scheme switching techniques are not at the point yet where this is feasible. We empirically tested this theory with the OpenFHE framework [1] that implements both CGGI and CKKS, as well as scheme switching between the two. Nevertheless, OpenFHE lacks a radix ciphertext mechanism to achieve arbitrarily high precision and bit width. For the dataset that we employ in this work, we require a minimum of 24 bits of plaintext per ciphertext (or radix ciphertext), which is not achievable at the time of writing with the OpenFHE implementation of CGGI.

5.2 Euclidean Distance

Euclidean distance is a formula for computing the distance between two n-dimensional points $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)$ and $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$ in the Euclidean space. It has a plethora of applications from statistics and cluster analysis [6] to facial recognition [36] and has drawn the interest of recent FHE works [44,24]. The Euclidean distance can be computed by $d(\boldsymbol{u}, \boldsymbol{v}) = \sqrt{\sum_{i=1}^{n}(u_i - v_i)^2}$; however, computing non-linearities (e.g., the square root in the above formula) in the encrypted domain is not a trivial task. Thus, many prior FHE works resort to computing the squared Euclidean distance and return it to the user, who in turn, needs to compute the final square root in the clear after decrypting. This is problematic for applications that require computing Euclidean distances as intermediate steps, as the cloud would have to engage the client to decrypt, compute the square root, and then re-encrypt and upload the new ciphertext. As a case in point, the Euclidean distance is commonly used as a distance metric in K-Nearest Neighbor classifiers [12].

5.3 Correlation Coefficient

The correlation coefficient is a statistical metric for quantifying the degree of linear association between two variables, ranging from -1 to 1 [3]. Correlation coefficients are crucial in assessing the extent of association between different variables or datasets in various applications ranging from finance and medical analytics [17,33] to IoT and weather prediction [32,28]. More specifically, we are interested in the Pearson product-moment correlation coefficient, or *Pearson's* r, to measure linear relationships between two features $\boldsymbol{u}$ and $\boldsymbol{v}$. Pearson's r is defined as the ratio of the covariance of $\boldsymbol{u}$ and $\boldsymbol{v}$ (i.e., $\mathrm{Cov}(\boldsymbol{u}, \boldsymbol{v})$) to the product of their standard deviations (i.e., $\sigma_{\boldsymbol{u}}$ and $\sigma_{\boldsymbol{v}}$), more formally:

$$r_{\boldsymbol{u},\boldsymbol{v}} = \frac{\mathrm{Cov}(\boldsymbol{u}, \boldsymbol{v})}{\sigma_{\boldsymbol{u}}\sigma_{\boldsymbol{v}}} = \frac{\sum_{i=1}^{n}(u_i - \bar{\boldsymbol{u}})(v_i - \bar{\boldsymbol{v}})}{\sqrt{\sum_{i=1}^{n}(u_i - \bar{\boldsymbol{u}})^2}\sqrt{\sum_{i=1}^{n}(v_i - \bar{\boldsymbol{v}})^2}}.$$

This formula shows that if larger $\boldsymbol{u}$ values tend to correspond to larger $\boldsymbol{v}$ values (and vice versa), then r is positive with 1 denoting a perfect positive correlation. On the other hand, if $\boldsymbol{u}$ and $\boldsymbol{v}$ do not move in tandem, then r is negative, with -1 being the perfect negative correlation.

In Ripple, we are interested in the scenario where one party (i.e., the client) possesses multiple features of the same type and another party (i.e., the server) possesses a vector of features of a different type. The client is interested in learning whether there is a correlation between their private features and the server's features but neither party is willing to disclose them. For instance, imagine a company (the client) owning customer demographics, such as age or income, while another company (the server) holds sales data. We can utilize Ripple to encrypt the demographic information of the client and send it to the server, which in turn can compute the correlation coefficient between the income or age of customers and their purchasing habits. We further observe that we can combine several of the operations into a single LUT. For instance, note that one of the square roots in the denominator relies entirely on plaintext values (the cloud data does not need to be encrypted as it will never leave the cloud's server). We can therefore compute the full denominator in a single LUT, which will compute the square root over the encrypted data corresponding to vector $\boldsymbol{u}$, multiply it with a plaintext constant representing $\sqrt{\sum_{i=1}^{n}(v_i - \bar{\boldsymbol{v}})^2}$, and finally compute the reciprocal of the product. However, because TFHE-rs operates over integers, the reciprocal will always be equal to 0 or 1 (since the denominator will be greater than or equal to 1). To avoid this, instead of computing the reciprocal, we divide a large-scale factor (e.g., 1000) by the denominator. After the rest of the algorithm is computed, the client simply needs to perform a division on the decrypted value by the scale factor to recover the expected answer.

(a) Original.

(b) Filtered.

Fig. 6: Edge detection example using the Prewitt operator.

5.4 Edge Detection with the Prewitt Operator

Image processing applications have recently become prominent targets for FHE; from simple blurring and sharpening filters [23,21] to the Roberts Cross edge-detection operator [43]. Under the hood, all these applications perform convolutions as a kernel between an image and a filter, which moves pixel by pixel. More specifically, the Roberts cross operator applies a 2×2 filter and approximates the image gradient by taking the square root of the sum of squares of two distinct convolutions of the image. These convolutions capture the discrepancies between pixels that are diagonally adjacent to each other. Computing the square root over encrypted data in certain FHE cryptosystems is prohibitively expensive, which leads [43] to completely omit its calculation.

A more advanced edge detection algorithm compared to the Roberts cross is the Prewitt operator, which uses two 3×3 matrices to calculate the pixel gradients in a region, i.e., directional changes in image intensity or color [14]. Prewitt excels in images with some level of noise and clear grayscale gradients as it achieves noise reduction by averaging the pixels. At its core, for each pixel, it performs two matrix multiplications with the masks and then sums the resulting vectors to compute two values. Finally, the Prewitt operator computes a magnitude as the square root of the sum of the squares of these two values.

In Ripple, we implemented the Prewitt operator utilizing our DWT technique for computing the square root. In Fig. 6a, we show the original image, whereas in Fig. 6b we show how the resulting image looks like after applying the Prewitt operator with Ripple. We observe that we can modify the traditional Prewitt filters, which are originally defined as

$$f_1 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}, \quad f_2 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix},$$

and require a series of scalar multiplications when performing the convolution between f_1 and f_2 and the input image. Specifically, we observe that we can scale these filters to consist solely of powers of 2 or 0, which allows us to perform simple shifts instead. In this case, the filters become

$$f_1' = \begin{bmatrix} 2 & 2 & 2 \\ 0 & 0 & 0 \\ -2 & -2 & -2 \end{bmatrix}, \quad f_2' = \begin{bmatrix} 2 & 0 & -2 \\ 2 & 0 & -2 \\ 2 & 0 & -2 \end{bmatrix}.$$

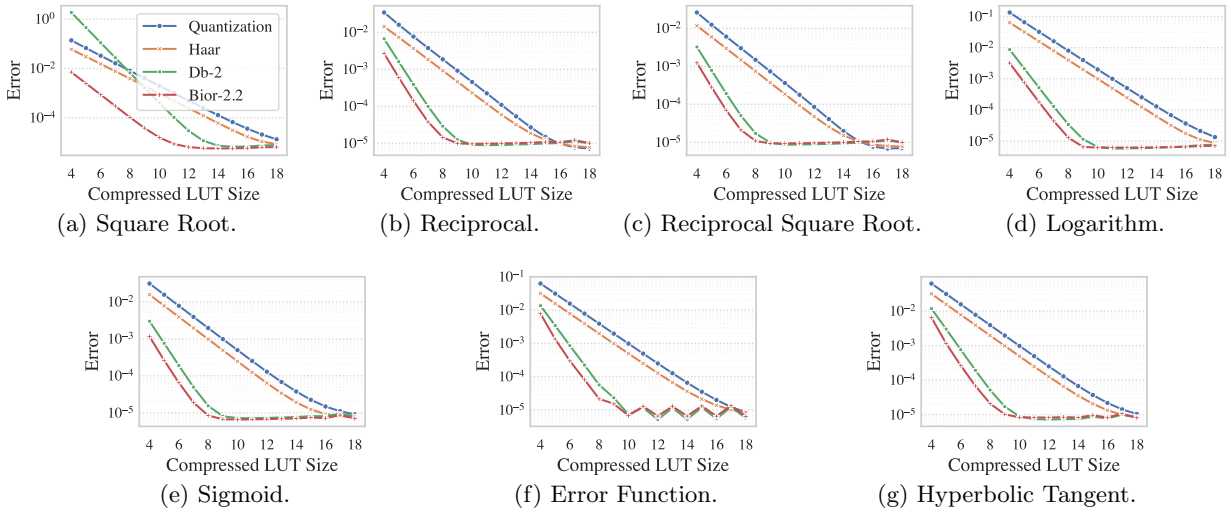
For 0 entries, we can avoid computation since the product is necessarily an encryption of 0 while the product with -2 can be computed as a left shift by 1 followed by a negation, which is efficient in CGGI.

6 Experimental Evaluation

6.1 Experimental Setup

We implemented Ripple using the state-of-the-art TFHE-rs [47] library.[5] In our experiments, we compare Ripple against HELM [23], Romeo [25], and Google Transpiler [21], as well as baseline implementations in

[5] Ripple is open-source at https://github.com/NillionNetwork/ripple.



(a) Square Root. (b) Reciprocal. (c) Reciprocal Square Root. (d) Logarithm.

(e) Sigmoid. (f) Error Function. (g) Hyperbolic Tangent.

Fig. 7: Approximation errors for multiple non-linear functions for varying compressed LUT sizes with Ripple.

TFHE-rs that use LUTs configured for the full bit width of each application. For Ripple, we implemented three variants: (a) a quantized version that is similar to the baseline, but, in this case, we truncate before applying the LUTs in a similar way to preparing inputs for a Haar DWT lookup, (b) a Haar DWT variant, and (c) a Biorthogonal DWT variant. It is expected that the quantized version will outperform both the Haar and Biorthogonal DWT-encoded LUTs in terms of latency, with the last two incurring significantly fewer errors.

We perform a series of experiments varying from simple non-linear functions (in Section 6.2) to more elaborate applications presented in Section 5 (in Section 6.3). For all experiments, we used a `c5.12xlarge` AWS EC2 instance with 48 virtual cores running Ubuntu 22.04. Also, for all TFHE-rs modes, we used a parameter set corresponding to approximately 128 bits of security [4]. Specifically, we utilized parameters that allow for each ciphertext to hold two data bits and two bits of carry used for intermediate computations. All radix ciphertexts are constructed as vectors of ciphertexts constructed with these parameters. Additionally, in all our experiments, the LUT generations are pre-computed offline by the server.

6.2 Non-Linear Functions

6.2.1 Ripple Approximations

In Table 1 we compare the Ripple approximations with both the Haar and Biorthogonal wavelet families against a baseline TFHE-rs implementation (using a non-DWT LUT with the full bit-width) and also a quantized TFHE-rs (i.e., approximation) version that truncates half of the bit-width of the inputs. Specifically, we consider a series of non-linear functions including reciprocal, square root, and the sigmoid activation function. Our comparisons emphasize the time to evaluate each non-linear function in seconds as well as the mean absolute error (MAE), which is the sum of absolute errors divided by the number of samples. We perform three blocks of experiments for the same functions with different combinations of word sizes (W), precision values (k), and LUT sizes (T). We experimentally observed that setting T to $\mathsf{W}/2$ yields the fastest runtime while minimizing the MAE. We set the precision k between the T and W because otherwise, quantization misses all the information in the fractional bits.

In Table 1, we observe that the runtime of all quantization, Haar DWT, and Biorthogonal DWT is faster than baseline, and increasingly so as the word size increases. This is intuitive as the LUTs are smaller, and therefore there are fewer PBS operations across the radix ciphertext. In the case of the Biorthogonal DWT, the extra computation associated with the multi-bit additions and multiplications is offset by the parallelism

Table 1: Overview of runtime improvement (in seconds) and mean absolute error (MAE) over non-linear operations for Ripple using Haar and Biorthogonal wavelets, as well as quantized TFHE-rs. Our baseline uses LUT sizes (T) to be equal to the word size (W), while the Ripple quantized version, as well as the Haar and Biorthogonal DWT variants, use approximations with T bits; k represents the precision.

Op.	W	k	T	Base-line	Quant.		Haar		Bior.	
					Time	MAE	Time	MAE	Time	MAE
$\sqrt{x}$				3.1	1.6	1.15e-2	1.6	5.41e-3	2.2	2.97e-4
$1/x$				3.1	1.6	3.83e-3	1.6	1.96e-3	2.2	2.21e-4
$1/\sqrt{x}$				5.9	3.2	2.79e-3	1.6	1.45e-3	2.1	1.97e-4
$\log x$	16	12	8	3.1	1.6	7.05e-2	1.6	2.17e-2	2.2	1.21e-2
$\sigma(x)^{\dagger}$				6.3	3.2	2.08e-3	1.6	1.02e-3	2.3	1.07e-4
$\mathsf{erf}(x)^{\ddagger}$				3.1	1.6	4.08e-3	1.6	2.05e-3	2.4	1.52e-4
$\mathsf{tanh}(x)^{\S}$				3.2	1.6	4.09e-3	1.6	2.07e-3	2.4	1.72e-4
$\sqrt{x}$				9.5	2.4	2.79e-3	2.3	1.37e-3	3.1	1.92e-5
$1/x$				9.5	2.4	2.45e-4	2.4	1.26e-4	3.1	8.62e-6
$1/\sqrt{x}$				18.1	4.8	2.19e-4	2.4	1.14e-4	3.1	8.24e-6
$\log x$	24	16	12	9.5	2.4	1.73e-3	2.3	8.61e-4	3.1	7.42e-6
$\sigma(x)^{*}$				19.0	4.8	1.97e-2	2.3	6.37e-5	3.3	6.73e-6
$\mathsf{erf}(x)^{*}$				9.4	2.4	3.96e-3	2.4	1.28e-4	3.3	8.89e-6
$\mathsf{tanh}(x)^{*}$				9.5	2.4	3.96e-3	2.4	1.28e-4	3.3	9.41e-6
$\sqrt{x}$				N/A	3.2	6.93e-4	3.2	3.45e-4	4.2	1.35e-6
$1/x$				N/A	3.2	1.57e-5	3.2	8.03e-6	4.2	5.37e-7
$1/\sqrt{x}$				N/A	6.3	1.49e-5	3.2	7.70e-6	4.1	5.07e-7
$\log x$	32	20	16	N/A	3.2	3.99e-4	3.2	1.47e-4	4.1	1.31e-4
$\sigma(x)^{*}$				N/A	6.3	7.68e-6	3.2	3.97e-6	4.3	4.20e-7
$\mathsf{erf}(x)^{*}$				N/A	3.1	1.54e-5	3.2	7.95e-6	4.3	5.31e-7
$\mathsf{tanh}(x)^{*}$				N/A	3.2	1.54e-5	3.2	7.91e-6	4.3	5.19e-7

† The Sigmoid function $\sigma(x) = 1/(1+e^{-x})$.

‡ The error function $\mathsf{erf}(x) = 2/\pi \int_0^x e^{-t^2}\, dt$.

§ The hyperbolic tangent function $\mathsf{tanh}(x) = (e^{x}-e^{-x})/(e^{x}+e^{-x})$.

* For these bounded functions where $\mathsf{W} - \mathsf{k} > 4$ we use the methodology explained in Section 4.

inherent in the algorithm. We observe that, for word size 16, the speedup is around $2\times$ while for word size 24, the speedup is closer to $3-4\times$, and for word size 32, the baseline is not possible to evaluate on our experimental server, as the RAM required to generate the encrypted LUTs exceeds 500 gigabytes for 32-bit tables. The Haar DWT is usually the same speed as quantization as both involve reducing the bit-width of the input by half and evaluating an LUT, but it is more accurate by halving the error observed. On the other hand, the Biorthogonal DWT is slower than the other two due to the overhead of multiplications, however, it is an order of magnitude more accurate.

6.2.2 DWT Compression

To better understand the accuracy difference between quantization and various DWT methods (i.e., Haar and Biorthogonal), we plot the average approximation errors for various compressed LUT sizes in Fig. 7. We also evaluate a third DWT variant, Db-2 (which is the second wavelet after Haar in the Debauchies family of wavelets), to illustrate the benefits of Haar and Biorthogonal. We start with a word size of 20 and a precision of 16 and proceed to compress the tables to varying sizes from 4 to 18 bits (the x-axes). Interestingly, the errors vary based on the approximated non-linear function.

From the trend, it is easy to see that the Haar DWT is twice as accurate as quantization throughout, while Db-2 and Biorthogonal DWTs are (in most cases) orders of magnitude more accurate. The difference in accuracy is most drastic around half the word size, which is 10 in this case. As the LUT size increases, the error of quantization and Haar DWT decreases linearly to the LUT size, while the respective errors of Db-2 and Biorthogonal DWTs decay exponentially to the LUT size until we reach a LUT size of 10-14 bits. The accuracy of the Biorthogonal DWT seems almost equivalent to Db-2 DWT and better by a factor of approximately 2 after some compression threshold. This suggests that Biorthogonal DWT gives better accuracy for the functions under evaluation.

Biorthogonal's superior accuracy, coupled with the fact that Db-2 is more expensive to evaluate due to an increased number of LUT evaluations, leads us to discard it altogether. On the other hand, since Biorthogonal is more expensive than Haar, we can evaluate the trade-off based on our application. If we need higher accuracy, we can go with Biorthogonal DWT, while if speed is of the essence, we can choose Haar DWT.

We observe the trends across the board, however, there are some differences between functions. For square root (Fig. 7a), Db-2 starts performing very poorly as the compressed LUT size decreases past 14. Reciprocal (Fig. 7b) and reciprocal square root (Fig. 7c) follow this trend, except for sizes 15-18 we have that Haar and quantization are more accurate than Biorthogonal and Db-2. Logarithm (Fig. 7d), sigmoid (σ, Fig. 7e), and tanh (Fig. 7g) follow the same trend with slight boundary effects (i.e., divergent behavior around the boundaries) for compressed LUT sizes over 16. Error function (Fig. 7f) experiences boundary effects when DWT is applied an odd number of times, hence we get a zigzag pattern for the error for compressed LUT sizes between 10 and 20 bits. Other functions experience this boundary effect for compressed LUTs with larger bit sizes, resulting in a paradoxically higher error when the compressed LUT size is above 16. The boundary effects can be attributed to DWT filters wrapping around the columns of the DWT matrix.

Table 2: Overview of bounded symmetrical function optimizations. Baseline: we compress a LUT with size equal to the word size (W) to LUT size (T); Optimization: we compress LUT with size (T+3) to LUT size (T). We report the maximum absolute error.

Op.	W	k	T	Haar			Bior.		
				Baseline	Opt.	Diff.	Baseline	Opt.	Diff.
$\sigma(x)$	24	16	12	7.82e-3	2.57e-4	30×	3.60e-5	1.65e-5	2.2×
erf(x)				3.52e-2	1.11e-3	32×	3.34e-4	3.43e-5	9.7×
tanh(x)				3.12e-2	9.82e-4	32×	2.67e-4	3.01e-5	8.9×
$\sigma(x)$	32	20	16	7.81e-3	1.61e-5	485×	3.16e-5	1.04e-6	30×
erf(x)				3.52e-2	6.93e-5	508×	3.16e-4	2.15e-6	147×
tanh(x)				3.12e-2	6.15e-5	507×	2.51e-4	1.90e-6	132×

6.2.3 Function-centric Compression

In Table 2 we compare Ripple's optimizations for convergent symmetrical functions with both Haar and Biorthogonal wavelet families, against a baseline without the optimizations. We start with a word size equal to 24 bits (or 32 bits) and using the optimizations, we bring this down to 19 bits (or 23 bits), respectively, by reducing the domain of the integer part of the LUT from $[-256, 256]$ to $[0, 8]$ (or from $[-2048, 2048]$ to $[0, 8]$). This effectively means that we now have 19 (or 23 bits) to represent a significantly smaller domain. We compare the maximum absolute error observed using the baseline and optimizations. Specifically, we observe a significant decrease in the error for Haar DWT on the order of 30× for 24 bits and 500× for 32 bits, while the decrease for Biorthogonal is about 10× for 24 bits and 100× for 32 bits, respectively. The improvement

in Biorthogonal is lower due to its already remarkable compression capabilities. Given the great increase in accuracy, applying these optimizations is beneficial when the computational overhead is acceptable.

6.3 Applications

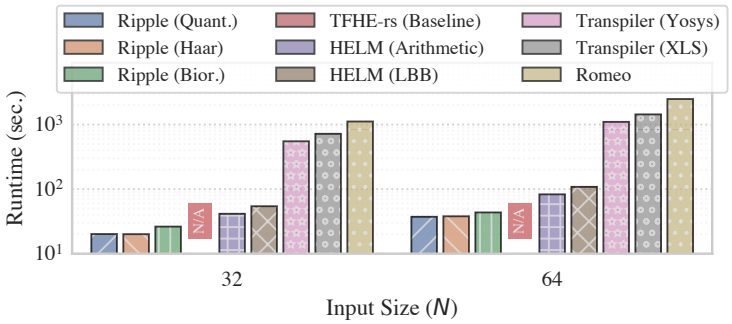


Fig. 8: Runtime comparisons for Euclidean distance between Ripple's three variants (Quantization, Haar DWT, and Biorthogonal DWT), TFHE-rs (baseline), HELM, Google Transpiler, and Romeo for vectors of 32 and 64 elements. Note that HELM, Transpiler, and Romeo only implement the squared Euclidean distance (i.e., without the square root computation). We use a word size W of 32 bits for all frameworks. Lastly, for 32 and 64 bits, the TFHE-rs baseline is not applicable (N/A) as the resources required for the LUT are impractical (see footnote 3).

6.3.1 Euclidean Distance

For this benchmark, shown in Fig. 8, we use $\mathsf{W} = 32$ bits and utilize a client and server vector of lengths 32 and 64 to demonstrate scalability. All Ripple variants perform the full Euclidean distance computation, while the related works compute the squared Euclidean distance and neglect the final square root calculation. We note that the TFHE-rs baseline is unable to evaluate the Euclidean distance with the required wordsize due to the astronomical cost of building 32-bit encrypted LUTs. For the Google Transpiler, we utilize both logic synthesis backends (i.e., Google XLS and Yosys), which optimize the circuit in different ways. For HELM, we utilize both LUT circuit modes (i.e., many-to-many LUTs for the arithmetic mode and a circuit of 2:1 LUTs for "lossless bidirectional bridging" or LBB). Overall, all three Ripple configurations outperform the related works in terms of latency while still taking into account the square root operation. However, as we observed in Table 1 the Haar and Biorthogonal approaches achieve significantly better approximations than the quantization variant. Notably, Haar also exhibits very competitive latencies across all non-linear functions and benchmarks.

6.3.2 Logistic Regression

Fig. 9 showcases our logistic regression inference benchmark for four attributes. While our chosen dataset is composed of entries with eight attributes, we truncate it to match the dimensions used in related works. Additionally, related works utilize a low-degree polynomial approximation for the sigmoid. We observe that Ripple is significantly faster than related works and also outperforms the TFHE-rs baseline using the full-bit width. The only exception to this is with $\mathsf{W} = 16$ bits, where the baseline outperforms the Biorthogonal DWT; however, for 24 bits, the Biorthogonal DWT exhibits lower latency than the baseline.

To achieve high accuracy with our chosen dataset (i.e., two species of penguins), we utilize all eight attributes with a wordsize of 24 bits. For this binary classification, all modes achieve 100% accuracy; the

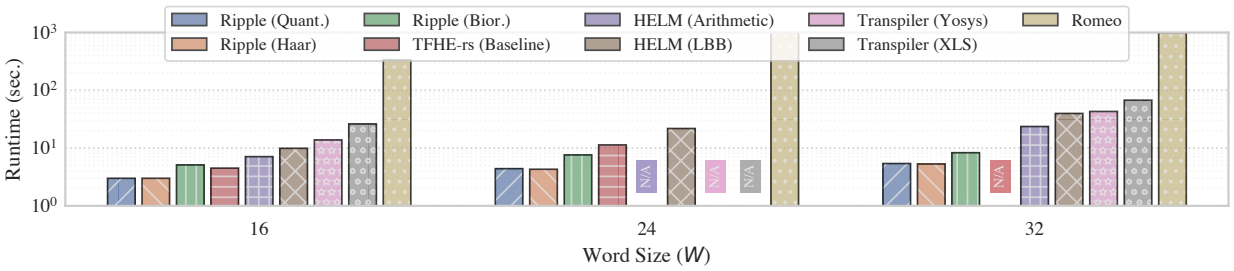


Fig. 9: Runtime comparisons for the logistic regression application for 4 attributes for word sizes of 16, 24, and 32 bits. For 24 bits, the arithmetic mode of HELM as well as both modes of the Google Transpiler are not applicable (N/A) as they rely on native word sizes. Lastly, for 32 bits, the TFHE-rs baseline is also N/A as the resources required for the LUT are impractical (see footnote 3).

baseline latency is 13.3 seconds per inference, while the Biorthogonal DWT variant classifies in 7.8 seconds. Lastly, the quantized variant that truncates half of the bits of the LUT input exhibits a latency of 6.2 seconds, while the Haar DWT slightly outperforms this with a latency of approximately 6 seconds.

6.3.3 Correlation Coefficient

We utilize $\mathsf{W} = 16$ to compute the correlation coefficient and find that the baseline implementation using the lookups with the full bit width requires 18.1 seconds. Similarly to the logistic regression with $\mathsf{W} = 16$, the Biorthogonal DWT exhibits a higher latency of approximately 19.4 seconds per evaluation. From these results and the prior benchmarks, we observe that the Biorthogonal performs suboptimally for small word sizes. However, Haar yields the lowest latency at 15 seconds per evaluation with the quantized variant taking 15.4 seconds on average.

6.3.4 Edge Detection

In our scenario, the client supplies an encrypted image where each monochrome pixel becomes an independent radix ciphertext (encoding 8 bits of information). We utilize the two modified filters described in Section 5.4, which are public and therefore not encrypted. We encrypt a 16×16 image as input and run the algorithm across the three Ripple modes and baseline. Lastly, we employ $\mathsf{W} = 16$ bits to avoid overflows during the intermediate computations. Overall, we observe that the majority of the execution time is consumed by all of the shifts and adds in the convolutions. The baseline implementation generates an encrypted output image in 482.6 seconds while the quantized version takes 476.9 seconds. The fastest variant, as in the prior benchmarks, is the Haar DWT at 471.2 seconds while the slowest is the Biorthogonal DWT at 588.5 seconds. The high latency of the Biorthogonal variant is primarily due to the fact that the application exhibits a very high degree of parallelism and therefore the parallelism inherent in the Biorthogonal LUT evaluation can not be properly exploited as all of the CPU cores are saturated.

7 Related Works

Ducas and Micciancio [16] first proposed the idea of using LUTs to evaluate arbitrary binary gates in HE, while Chillotti et al. [10] then extended this idea to evaluate arbitrary function evaluation as a tree of leveled multiplexers. Adoption was very limited, however, as it required expressing programs as deterministic automata and it needed the control inputs of the multiplexers to be fresh ciphertexts (i.e., could not perform computation with them before to multiplexer). The programmable bootstrapping technique (PBS) introduced in [11] allows for efficient and general-purpose LUT evaluation. HELM [23] built on this technique and introduced a framework for automated conversion from Verilog hardware description language (HDL) to

encrypted circuits. HELM employs three modes of operation, one that solely operates over binary gates, one that operates over integers and utilizes secure LUT evaluations, and a mixed mode that operates over binary circuits and "bridges" over to integers to securely evaluate a LUT and then "bridge" back to the binary domain. However, HELM is only compatible with very low-precision LUTs as bridging from an integer to multiple bits requires multiple N to 1 LUTs. Conversely, Ripple has high precision and requires smaller LUTs to encode the same amount of information with negligible errors.

Romeo [25] and Google Transpiler [21] follow a similar approach as HELM, in that of relying on an HDL and logic synthesis. The latter, provides two different front-ends, one based on Yosys [45] and another one based on Google XLS [20]. Both works, however, rely on Boolean circuits and neither of them supports LUTs, resulting in costly operations for evaluating non-linear functions.

In a different line of work, Chung et al. [13] evaluate LUTs under FHE with the BGV, BFV, and CKKS cryptosystems. Their idea is to transform LUTs into low-degree multivariate polynomials and utilize the packing (or batching) capabilities of the aforementioned cryptosystems to simultaneously evaluate multiple LUTs. They demonstrate their approach by evaluating AES on an A-100 GPU in over 9.5 minutes (while batching 2048 ciphertexts). Although the amortized cost per ciphertext is under a second, the latency of this approach is far from practical. On the other hand, Ripple shows a way to accelerate LUT evaluation without sacrificing correctness and is orthogonal to the underlying cryptosystem. As a matter of fact, our techniques introduced in Ripple can be extended for BGV, BFV, and CKKS as well and further accelerate LUT evaluation.

8 Concluding Remarks

In this work, we introduce the Ripple framework that leverages different approximation techniques based on discrete wavelet transform families to reduce the number of homomorphic LUT entries in PBS. Previous works focused either on polynomial approximations or on full-size LUTs; these techniques incur high-performance overheads when high precision is required while the former also introduced non-negligible errors. Ripple, on the other hand, maintains high accuracy while it reduces the LUT sizes. Our empirical evaluations have shown significant error reduction compared to plain quantization methods across various non-linear functions, varying from square root and reciprocal computations to more elaborate sigmoid and hyperbolic tangent functions. A key benefit of Ripple is that it improves performance for several realistic benchmarks without sacrificing accuracy, compared to equivalent applications that utilize the full bit widths and incur slower LUT evaluation runtimes.

Acknowledgments

The authors would like to acknowledge Manuel B. Santos for suggesting the correlation coefficient application. C. Gouert and N.G. Tsoutsos would like to acknowledge the support of the National Science Foundation (Award 2239334).

References

1. Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-Source Fully Homomorphic Encryption Library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, pages 53–63, New York, NY, USA, 2022. Association for Computing Machinery.
2. David D Lewis Alexander Genkin and David Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007.
3. Agustin Garcia Asuero, Ana Sayago, and AG González. The correlation coefficient: An overview. *Critical reviews in analytical chemistry*, 36(1):41–59, 2006.

4. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, *27th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 10–24, Arlington, VA, USA, January 10–12, 2016. ACM-SIAM.
5. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.
6. Randy L Carter, Robin Morris, and Roger K Blashfield. On the partitioning of squared euclidean distance and its applications in cluster analysis. *Psychometrika*, 54(1):9–23, 1989.
7. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.
8. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer, Heidelberg, Germany.
9. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.
10. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.
11. Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning*, pages 1–19, Cham, 2021. Springer International Publishing.
12. Kittipong Chomboon, Pasapitch Chujai, Pongsakorn Teerarassamee, Kittisak Kerdprasop, and Nittaya Kerdprasop. An empirical study of distance metrics for k-nearest neighbor algorithm. In *Proceedings of the 3rd international conference on industrial application engineering*, volume 2, page 4, Kitakyushu, Japan, 2015.
13. Heewon Chung, Hyojun Kim, Young-Sik Kim, and Yongwoo Lee. Amortized Large Look-up Table Evaluation with Multivariate Polynomials for Homomorphic Encryption. Cryptology ePrint Archive, Paper 2024/274, 2024. https://eprint.iacr.org/2024/274.
14. Wang Dong and Zhou Shisheng. Color Image Recognition Method Based on the Prewitt Operator. In *2008 International Conference on Computer Science and Software Engineering*, volume 6, pages 170–173, Wuhan, China, 2008. IEEE.
15. Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomput.*, 503(C):92–108, sep 2022.
16. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.
17. Orhan Erdem, Elvan Ceyhan, and Yusuf Varli. A new correlation coefficient for bivariate time-series data. *Physica A: Statistical Mechanics and its Applications*, 414:274–284, 2014.
18. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.
19. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
20. Google Research. Google XLS, 2020.
21. Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann, Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki, Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy, Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko, and Bryant Gipson. A general purpose transpiler for fully homomorphic encryption. Cryptology ePrint Archive, Report 2021/811, 2021. https://eprint.iacr.org/2021/811.
22. Charles Gouert et al. Accelerated encrypted execution of general-purpose applications. Cryptology ePrint Archive, Report 2023/641, 2023.
23. Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. HELM: Navigating Homomorphic Encryption through Gates and Lookup Tables. Cryptology ePrint Archive, Paper 2023/1382, 2023. https://eprint.iacr.org/2023/1382.

24. Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. SoK: New Insights into Fully Homomorphic Encryption Libraries via Standardized Benchmarks. *Proceedings on Privacy Enhancing Technologies*, 2023(3):154–172, July 2023.
25. Charles Gouert and Nektarios Georgios Tsoutsos. Romeo: conversion and evaluation of HDL designs in the encrypted domain. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20, Virtual Event, USA, 2020. IEEE Press.
26. Soheil Hashemi, Nicholas Anthony, Hokchhay Tann, R. Iris Bahar, and Sherief Reda. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, page 1478–1483, Leuven, BEL, 2017. European Design and Automation Association.
27. Allison M Horst, Alison Presmanes Hill, and Kristen B Gorman. Palmer archipelago penguins data in the palmerpenguins r package-an alternative to anderson's irises. *R Journal*, 14(1), 2022.
28. Shuai Hu, Yue Xiang, Hongcai Zhang, Shanyi Xie, Jianhua Li, Chenghong Gu, Wei Sun, and Junyong Liu. Hybrid forecasting method for wind power integrating spatial correlation and corrected numerical weather prediction. *Applied Energy*, 293:116951, 2021.
29. Nina Viktoria Juliadotter and Kim-Kwang Raymond Choo. Cloud attack and risk assessment taxonomy. *IEEE Cloud Computing*, 2(1):14–20, 2015.
30. Miran Kim, Yongsoo Song, Baiyu Li, and Daniele Micciancio. Semi-Parallel logistic regression for GWAS on encrypted data. *BMC Med Genomics*, 13(Suppl 7):99, July 2020.
31. Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, Xiaoqian Jiang, et al. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR medical informatics*, 6(2):e8805, 2018.
32. Lu Li, Mei Rong, and Guangquan Zhang. An internet of things qoe evaluation method based on multiple linear regression analysis. In *2015 10th International Conference on Computer Science & Education (ICCSE)*, pages 925–928, Cambridge, UK, 2015. IEEE.
33. Xiaodi Liu, Zengwen Wang, Shitao Zhang, and Harish Garg. Novel correlation coefficient between hesitant fuzzy sets with application to medical diagnosis. *Expert Systems with Applications*, 183:115393, 2021.
34. Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 130–160, Taipei, Taiwan, December 5–9, 2022. Springer, Heidelberg, Germany.
35. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.
36. M. D. Malkauthekar. Analysis of euclidean distance and manhattan distance measure in face recognition. In *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013)*, pages 503–507, Mumbai, 2013. IET.
37. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany.
38. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 84–93, Baltimore, MA, USA, May 22–24, 2005. ACM Press.
39. Maria Isabel Ribeiro. Gaussian probability density functions: Properties and error characterization. *Institute for Systems and Robotics, Lisboa, Portugal*, 2004.
40. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, February 1978.
41. Michael Unser and Thierry Blu. Mathematical properties of the jpeg2000 wavelet filters. *IEEE transactions on image processing*, 12(9):1080–1090, 2003.
42. P.J. Van Fleet. *Discrete Wavelet Transformations: An Elementary Approach with Applications*. Wiley, NY, USA, 2019.
43. Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. HECO: Fully homomorphic encryption compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4715–4732, Anaheim, CA, August 2023. USENIX Association.
44. Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy*, pages 1092–1108, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.
45. Clifford Wolf. Yosys Open SYnthesis Suite, 2016.

46. Pan Yang, Naixue Xiong, and Jingli Ren. Data security and privacy protection for cloud storage: A survey. *IEEE Access*, 8:131723–131740, 2020.
47. Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.