# Adaptive Successive Over-Relaxation Method for a Faster Iterative Approximation of Homomorphic Operations

Jungho Moon[1], Zhanibek Omarov[1], Donghoon Yoo[1], Yongdae An[1], and Heewon Chung[1]

[1]Desilo Inc., Seoul, Republic of Korea

August 30, 2024

**Abstract**

Homomorphic encryption is a cryptographic technique that enables arithmetic operations to be performed on encrypted data. However, word-wise fully homomorphic encryption schemes, such as BGV, BFV, and CKKS schemes, only support addition and multiplication operations on ciphertexts. This limitation makes it challenging to perform non-linear operations directly on the encrypted data. To address this issue, prior research has proposed efficient approximation techniques that utilize iterative methods, such as functional composition, to identify optimal polynomials. These approximations are designed to have a low multiplicative depth and a reduced number of multiplications, as these criteria directly impact the performance of the approximated operations.

In this paper, we propose a novel method, named as adaptive successive over-relaxation (aSOR), to further optimize the approximations used in homomorphic encryption schemes. Our experimental results show that the aSOR method can significantly reduce the computational effort required for these approximations, achieving a reduction of 2–9 times compared to state-of-the-art methodologies. We demonstrate the effectiveness of the aSOR method by applying it to a range of operations, including sign, comparison, ReLU, square root, reciprocal of m-th root, and division. Our findings suggest that the aSOR method can greatly improve the efficiency of homomorphic encryption for performing non-linear operations.

## 1 Introduction

Fully homomorphic encryption (FHE) is a fundamental cryptographic principle that enables the evaluation of an arithmetic or logical circuit representing a function while maintaining the privacy of the corresponding messages. By allowing

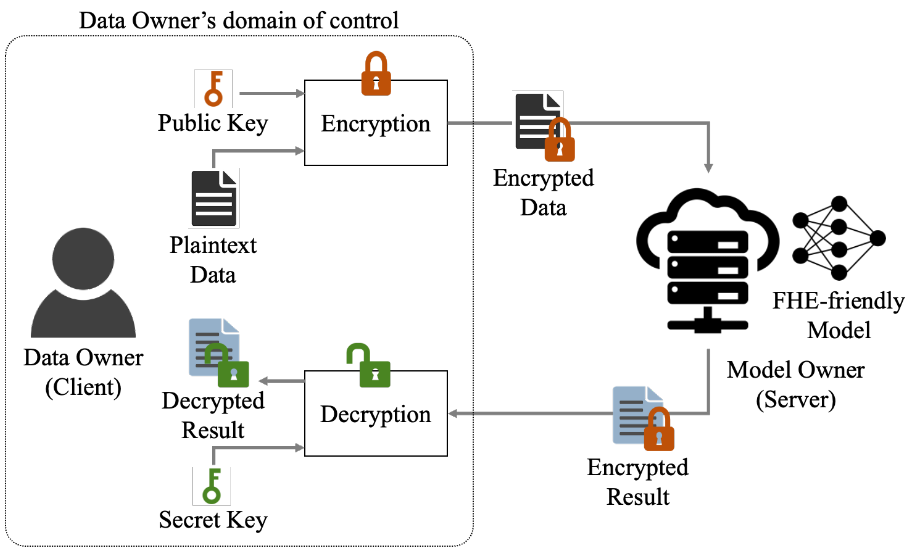


Figure 1: MLaaS with FHE for inference.

arithmetic and logical operations to be performed on encrypted messages, FHE guarantees the privacy while allowing meaningful computation. Clients can keep their data in encrypted format in the cloud and perform computations, such as inference, prediction, statistics, and more, directly on the encrypted data without revealing their confidential information. Similarly, the cloud-based service providers do not need to reveal their secrets, such as models or methods, to the clients.

As an example of application areas, FHE can be used as a technology for protecting the clients' privacy when deploying Machine Learning as a Service (MLaaS) on public clouds. In MLaaS, cloud servers can access clients' raw data, and hence potentially introduce privacy risks. The privacy of the clients' data can be protected using FHE as depicted in Figure 1. Since the client's data is encrypted by FHE before sending it to the server, the data never leaves the control of the client in plaintext. The server only receives the encrypted data and evaluates the machine learning model, which consists of FHE-friendly components as suggested in this work, with the given encrypted data. The result of the inference is also encrypted and only the client can decrypt and see the result. In this manner, the server acquires no knowledge of either the input or the output of such MLaaS pipeline, ensuring data privacy. This solution is called Privacy Preserving Machine Learning (PPML) [16, 22, 30, 25]. In addition to its deployment within PPML, FHE finds extensive applicability across various domains such as medical data analysis [28, 3, 35], financial data analysis [40], secure searchable encryption [43, 4], secure database [38, 41], and so on.

Gentry first introduced the blueprint of FHE scheme in [19], and since then several studies [6, 5, 18, 17, 9, 13] have aimed to construct efficient FHE schemes.

FHE schemes can be categorized into two types: word-wise FHE schemes (such as BGV [6], BFV [5, 18], and CKKS [9]) and bit-wise FHE schemes (such as FHEW [17] and CGGI [13]), depending on the primitive operations between ciphertexts. Word-wise FHE schemes support primitive operations of addition and multiplication on integers or complex numbers, while bit-wise FHE schemes support Boolean gates [17] and look-up table based operations [12].

Non-linear functions such as sign, square root, inverse, comparison, and activation functions found in neural nets (including ReLU, Max Pooling, Softmax, and others) are essential for practical real-world applications of FHE. Since word-wise FHE schemes only support addition and multiplication operations, such non-linear functions must be approximated through iterative methods, high-degree polynomials, or other sub-variations like polynomial compositions [10].

The efficiency of approximations of non-linear functions is determined by,

- Computational complexity — the total number of multiplications in the arithmetic circuit,
- Multiplicative depth — length of the longest chain of sequential multiplications. Depth can also be determined via the largest polynomial degree of an arithmetic circuit.

For example, $f(x) = x^4 + x^3 + 1$ has a depth of $\log_2 4 = 2$ and complexity of $2 + 2 + 0 = 4$[1].

The practicality of FHE is typically limited by the significant performance gap between FHE operations and ordinary calculations. There have been various efforts to bridge this gap, such as the development of hardware accelerators (e.g., CraterLake [39], Medha [33], SHARP [27]) and efficient FHE library implementations (e.g., HElib [21], Microsoft SEAL [34], OpenFHE [2], HEaaN [24], TFHE-rs [44]). This work aims to enhance the FHE performance orthogonally to these software and hardware optimizations, by improving non-linear function approximation methodologies.

## 1.1 Related Work

Cheon et al. [10] have reviewed evaluation methods for various basic functions such as inverse and square root which they used to construct an algorithm for maximum/minimum of numbers without using Boolean functions. Later, the authors [7] have substantially improved their results by introducing a new method for evaluating comparison function (equivalent to maximum/minimum) for which they needed to carefully devise two new sets of polynomials to achieve maximal performance. Lee et al. [29] have shown that the multiplicative depth can be further optimized by using compositions of special minimax approximate polynomials for approximation of comparison function. Panda [36] has demonstrated a fast evaluation method of inverse square root at the cost of additional evaluation of the comparison function.

[1] with reuse of computation results it further reduces to 3

Cheon et al.'s works [10, 7] have pioneered efficient polynomial approximations for non-linear homomorphic operations. The subsequent works generally tend to extend and/or improve their initial efforts. Panda's work [36] focuses on inverse square root while showing worse performance in addition to requiring an auxiliary computation. Lee et al. [29] have achieved performance on par with ours for comparison function, but their approach is limited to comparison function only. Our work stands out in that it directly extends well-known iterative methods, such as Newton's method and Goldschmidt's method, with simple-to-implement modification. Our approach is more versatile and can be applied to a wider range of iterative functions.

### 1.2 Our Contribution

In this paper, we further develop ideas introduced in the successive over-relaxation (SOR) [42, 15] method to improve the efficiency for iterative methods that are commonly used for homomorphic evaluation of non-linear functions. Our primary contribution, adaptive successive over-relaxation method (aSOR), is a general purpose method for a faster evaluation of iterative processes (Section 2.2). The term "adaptive" in aSOR refers to the continuous adjustment of the relaxation coefficient to ensure the fastest convergence at each step. SOR is a general-purpose method that is not limited to FHE or certain iterative functions. We demonstrate the practicality and high performance of aSOR by approximating non-linear functions such as comparison, sign, ReLU, square root, inverse square root, reciprocal of m-th root, inversion (division), max pooling, and softmax as possible applications of this methodology. Overall, we find that, in general, aSOR outperforms state-of-the-art iterative methods used in FHE, achieving a speedup ranging from 2 to 9 times faster, with the improvement increasing as the accuracy of the termination criteria becomes tighter.

**Organization** Section 2 presents definitions of homomorphic encryption and various approximation algorithms. In Section 3, we introduce the main idea of our work, which is adaptive successive over-relaxation (aSOR). We then describe the application of aSOR to several non-linear approximations in Section 4, including Cheon et al.'s comparison function [10, 7], Goldschmidt's inverse calculation [20, 10], Goldschmidt's square root and inverse square root calculation, and Newton-Raphson's method for square root calculation [1, 37]. Finally, in Section 5, we provide benchmarks with specific parameters.

## 2 Preliminaries

### 2.1 Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) allows for computations to be performed directly on ciphertexts without requiring decryption. FHE schemes are classified as bitwise and word-wise as we mentioned in Section 1. The basic operations of bitwise schemes are logic gates, whereas the basic operations of word-wise schemes are addition and multiplication. In this paper, we focus only on word-wise FHE schemes, and the terminology FHE refers to word-wise FHE.

An FHE scheme can be represented by a quintuple of probabilistic polynomial-time algorithms, denoted by $\mathsf{FHE} = (\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec},$
$\texttt{Add}, \texttt{Mult})$, which are defined as follows:

- $\texttt{KeyGen}(params) \to (pk, sk, evk)$; This algorithm takes a parameter $params$, which is determined by the security parameter $\lambda$ and the multiplicative depth $L$, and outputs a public key $pk$, a secret key $sk$, and an evaluation key $ek$.

- $\texttt{Enc}(pk, m) \to c$; This algorithm takes a message $m \in \mathsf{P}$ and a public key $pk$ as inputs, and outputs a ciphertext $c \in \mathsf{C}$, where $\mathsf{P}$ and $\mathsf{C}$ denote the plaintext space and the ciphertext space, respectively.

- $\texttt{Dec}(sk, c) \to m$: This algorithm takes a ciphertext $c \in \mathsf{C}$ and a secret key $sk$ as inputs, and outputs a message $m \in \mathsf{P}$.

- $\texttt{Add}(c_1, c_2, evk) \to c_{add}$; This algorithm takes ciphertexts $c_1 \in \mathsf{C}$ and $c_2 \in \mathsf{C}$ of plaintexts $m_1 \in \mathsf{P}$ and $m_2 \in \mathsf{P}$, respectively, with an evaluation key $evk$ as inputs, and outputs a ciphertext $c_{add} \in \mathsf{C}$ of $m_1 + m_2$.

- $\texttt{Mult}(c_1, c_2, evk) \to c_{mult}$; This algorithm takes ciphertexts $c_1 \in \mathsf{C}$ and $c_2 \in \mathsf{C}$ of plaintexts $m_1 \in \mathsf{P}$ and $m_2 \in \mathsf{P}$, respectively, with an evaluation key $evk$ as inputs, and outputs a ciphertext $c_{mult} \in \mathsf{C}$ of $m_1 \cdot m_2$.

The computational efficiency of an arithmetic circuit $f$ in an FHE scheme is primarily determined by its computational complexity, which corresponds to the number of multiplications, and its multiplicative depth. Therefore, minimizing these parameters is essential for achieving practical performance. In the remainder of the paper, we will concentrate on the non-scalar multiplicative depth and computational complexity, disregarding the number of additions and scalar multiplications in our analysis of computational efficiency.

## 2.2 Iterative Process

An iterative process is a computational technique used to refine an initial guess solution value $x_1$ until a terminal condition is met. Convergence rate and attraction basin, which denotes the range of initial values that converge, are important properties of any iterative process. Examples of well-known iterative processes include the Goldschmidt's division algorithm [20] and the Newton-Raphson method [1].

In an iterative process of the form $x_{n+1} = f(x_n)$, where $x_{\text{fp}} = f(x_{\text{fp}})$ and the subscript "fp" denotes the "fixed point", or convergence point, of $f$, the sequence $x_n$ $(n \geq 1)$ starting from $x_1$ converges to the true solution $x_{\text{fp}}$ provided that the initial guess $x_1$ lies within the convergence region. Since $f$ is not restricted to be linear, the commonly used notion of spectral radius is not applicable.

## 3 Adaptive successive over-relaxation method

The adaptive successive over-relaxation (aSOR) method introduced in this work is based on the conventional successive over-relaxation (SOR) method [42, 15], which aims to improve the convergence of iterative processes (as discussed in Section 2.2) by scaling the input using a relaxation factor $k$ (often denoted as $\omega$ in literature). The SOR method is typically used for faster solving of linear systems of equations in the form $A\mathbf{x} = \mathbf{b}$, where the relaxation factor remains constant across iterations. The focus of this work is on non-linear scalar valued iterative processes with variable relaxation factor $k_i$, where $i$ denotes an iteration number. All values of $k_i$ are predetermined, and no just-in-time decisions are made based on the output of any particular iteration.

Consider the function $f(x) = x(2-x)$ as a simple example, where the input range is $x_1 \in [\epsilon, 1]$ for $0 < \epsilon < 1$. The functional composition of the example function can be denoted as $x_3 = f(f(x_1)) = f^{(2)}(x_1)$, and more generally, $x_{n+1} = f^{(n)}(x_1)$. It is clear that $\lim_{n\to\infty} f^n(x) = 1$, and the fixed point of $f$ is $x_{\text{fp}} = 1$. We can call this function as the "almost sign" function because it makes any positive number $0 < x \leq 1$ converge to unity, although it does not work for negative numbers. The discussion of this particular function is still useful for its application in Goldschmidt's division algorithm, as discussed in Section 4.2.

Increasing the value of $n$ leads to an increasingly better approximation, as illustrated in Figure 2 (Left). We will restrict the initial input range to $x_1 \in [\epsilon_1, 1]$, where $\epsilon_1 \ll 1$ and the subscript $i$ indicates the iteration number. Consequently, the input range for the $i$'th iteration can be defined as $x_i \in [\epsilon_i, 1]$. Also, it is helpful to reinterpret this problem in terms of shrinking input ranges. For every pair of $i$ and $j$ with $0 < i < j$, we aim to have a narrower input range $x_j \in [\epsilon_j, 1]$ compared to $x_i \in [\epsilon_i, 1]$, where $\epsilon_j > \epsilon_i$. The narrowing of the input/output interval is achieved through iterative evaluation.

Instead of directly evaluating $f(x)$, the idea lies in computing $f(kx)$ even in the first iteration such that,

$$f(k_1 x_1) \in [\min(f(k_1\epsilon_1), f(k_1)), 1],$$

which is also the input range for the next iteration. That is, the idea of aSOR iterative composition is to evaluate,

$$\begin{aligned} x_n &= f(k_{n-1}x_{n-1}) \\ &= f(k_{n-1}f(k_{n-2}x_{n-2})) \\ &= f(k_{n-1}f(\ldots f(f(k_1x_1)))). \end{aligned}$$

In general, the optimal relaxation factor $k$ may differ for each iteration, and can be determined as follows:

$$k_i = \arg\max_{k_i}[\min(f(k_i\epsilon_i), f(k_i))].$$

The relaxation factors $k_i$ are to be chosen such that the output range constricts maximally at each iteration $i$. And this process yields the array of optimal

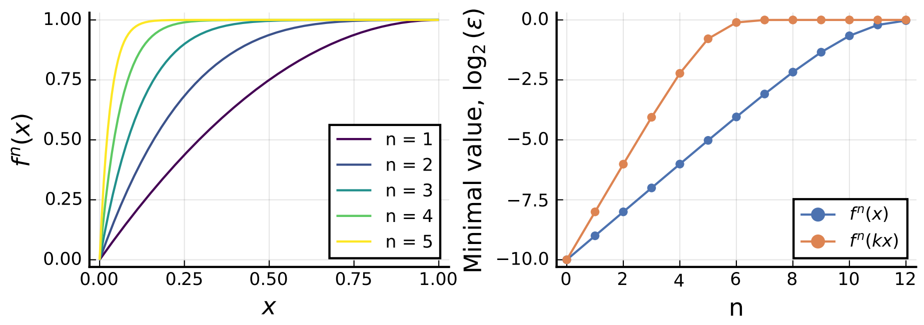


Figure 2: Left: Composite evaluation of "almost sign" $f(x) = x(2-x)$ function for various $n$ values. Right: Illustration of shrinkage of minimal input value $\epsilon$ for increasing $n$ with normal evaluation (blue) and aSOR method (orange). $n = 0$ corresponds to the initial input with $\epsilon_1 = 2^{-10}$.

$k_i$. For the specific example given, the maximum criterion is achieved when $f(\epsilon_i) = f(k_i\epsilon_i)$ or $k_i = 2/(\epsilon_i+1)$, as shown in Figure 3. Selecting the maximally optimal relaxation factor for each iteration is the core idea of aSOR method.

Assuming an initial input range floor of $\epsilon_1 = 2^{-10}$, and considering $n$ iterations to evaluate the function $f$, Figure 2 (Right) demonstrated the accelerated input range shrinkage achieved by the aSOR method in comparison to normal evaluation. As can be observed, the difference between the two methods increases naturally with larger values of $n$.

We can now formalize the aSOR method and the necessary conditions for any iterative function $f$.

**Definition 1 (Fixed point)** *Given a function $f$, a fixed point is the "convergence point" of the iterative evaluation for $f$ which can be achieved by $x_{\text{fp}} = \lim_{n\to\infty} f^n(x)$ for some small but finite $\varepsilon > |x - x_{\text{fp}}|$.*

**Definition 2 (Attraction basin)** *Given a function $f$, the attraction basin is defined as a maximal contiguous range $x \in [a,b]$ such that $x_{\text{fp}} = \lim_{n\to\infty} f^n(x)$ where $a \le x_{\text{fp}} \le b$.*

Assuming that $x_1 \in [a_1, b_1] \subset [a,b]$ is the expected input range at the first iteration, it implies that the initial expected input range is *strictly narrower* than the maximally allowed range (or the attraction basin). The iterative process can be interpreted as an array of intervals,

$$[a_n, b_n] \subset [a_{n-1}, b_{n-1}] \subset \cdots \subset [a_1, b_1] \subset [a, b],$$

where $[a_{i+1}, b_{i+1}] = f([a_i, b_i])$,

$$\lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n = x_{\text{fp}}.$$

Table 1: Example of sign approximation without aSOR

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $x_i$ | -0.25 | -0.36 | -0.52 | -0.71 | -0.89 | -0.98 | **-0.99** |

Table 2: Example of sign approximation with aSOR

| $i$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $k_i$ | - | 1.51 | 1.28 | 1.06 |
| $x_i$ | -0.25 | -0.54 | -0.87 | **-0.99** |

As long as $[k_i a_i, k_i b_i] \subset [a, b]$, such $k_i$ can be used without ruining the convergence criteria. We can compare the results of the $i$-th iteration with and without the relaxation factor $k_i$ as follows:

$$f([a_i, b_i]) = [a_{i+1}, b_{i+1}]$$
$$f([k_i a_i, k_i b_i]) = [a'_{i+1}, b'_{i+1}].$$

If $[a'i+1, b'i+1] \subset [a_{i+1}, b_{i+1}]$, then the relaxation factor $k_i$ is considered valid as it accelerates the convergence. Empirically, $k_i \geq 1$ and generally, $\lim_{n\to\infty} k_n = 1$. The aSOR method chooses an optimal $k_i$ at each iteration such that

$$k_i = \arg\max_{k_i} (c_{i+1} - c'_{i+1}), \tag{1}$$

where $c_{i+1} = b_{i+1} - a_{i+1}$ and $c'_{i+1} = b'_{i+1} - a'_{i+1}$ with $c_{i+1} > c'_{i+1}$.

## 4 Applications of aSOR method

We will demonstrate the aSOR method using several exemplary non-linear functions, including comparison, inversion, square root, and reciprocal of m-th root. Since only additions and multiplications are available as primitive operations in word-wise FHE schemes, non-linear functions cannot be evaluated directly. Instead, non-linear functions are approximated via polynomials. In practice, Goldschmidt's division algorithm [20] and Newton-Raphson method [37] are widely used for this purpose.

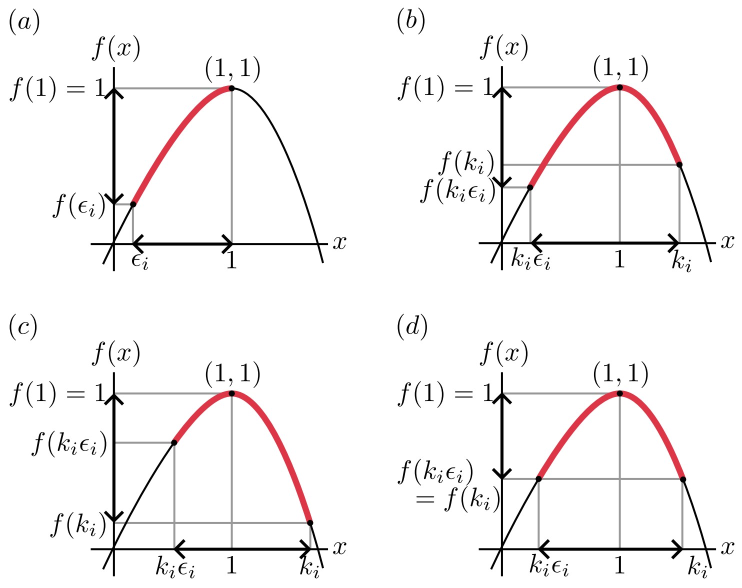


Figure 3: Graphical illustration of finding the optimal $k_i$. (a) when $k_i = 1$, $\epsilon_{i+1} = f(\epsilon_i) < f(1) = 1$. (b) when $k_i$ is chosen so that $\epsilon_{i+1} = f(k\epsilon_i) < f(k_i) < 1$, larger $k_i$ makes $\epsilon_{i+1}$ larger. $k_i$ is chosen too small. (c) when $k_i$ is chosen so that $\epsilon_{i+1} = f(k_i) < f(k_i\epsilon_i) < 1$, smaller $k_i$ makes $\epsilon_{i+1}$ larger. $k_i$ is chosen too large. (d) when $k_i$ is chosen so that $\epsilon_{i+1} = f(k_i\epsilon_i) = f(k_i) < 1$, we can get the largest $\epsilon_{i+1}$ while maximally reducing the number of needed iterations. $k_i$ is optimal.

## 4.1 Sign, Comparison and ReLU

For the sake of simplicity, we define a comparison function `Cmp` and a sign function `Sgn`: for $a, b \in \mathbb{R}$,

$$\texttt{Cmp}(a,b) = \begin{cases} 1 & \text{if } a > b \\ 1/2 & \text{if } a = b \\ 0 & \text{if } a < b \end{cases}$$

$$\texttt{Sgn}(a-b) = \begin{cases} 1 & \text{if } a > b \\ 0 & \text{if } a = b \\ -1 & \text{if } a < b \end{cases}$$

Note that `Cmp` does not satisfy linearity and it can be obtained from `Sgn` since

$$\texttt{Cmp}(a,b) = \frac{\texttt{Sgn}(a-b)+1}{2}.$$

With an additional multiplication after comparison evaluation, ReLU can be easily obtained via $\text{ReLU}(x) = x \cdot \texttt{Cmp}(x,0)$.

**Algorithm 1** RelaxationFactor($f, \alpha, \epsilon$)

**INPUT:** a polynomial $f(z)$, $z \in [\epsilon, 1]$, $\alpha, \epsilon \in \mathbb{R} \cap [0, 1]$
**OUTPUT:** a list of the relaxation factors

```
1: $\epsilon_1 \leftarrow \epsilon$ and $i \leftarrow 1$
2: $K \leftarrow []$
3: while $1 - \epsilon_i > 2^{-\alpha}$ do
4:     $k_i = \arg\max_{k_i}[\min(f(k_i\epsilon_i), f(k_i))]$
5:     append $k_i$ to $K$
6:     $\epsilon_{i+1} \leftarrow f(k_i)$
7:     $i \leftarrow i + 1$
8: end while
9: return $K$
```

In [7], they constructed a set of composite polynomials to approximate the sign function. We use

$$f(x) = -\frac{1}{2}x^3 + \frac{3}{2}x \tag{2}$$

which shows the best performance within the set of polynomials in [7] when applying the aSOR method.

The input range for the sign function is $x \in [\epsilon, 1]$, and the composition process is repeated for $n$ iterations until the convergence criteria $2^{-\alpha}$ is met. That is, the iterative composition $x_{i+1} = f(x_i)$ is repeated $n$ times until $|f(x_{n+1}) - \texttt{Sgn}(x)| < 2^{-\alpha}$. In other words, given the initial input range $\epsilon \leq x_1 \leq 1$ (or equivalently for negative input $-1 \leq x_1 \leq -\epsilon$, as Equation 2 is an odd function), we expect the final output range to be $1 - 2^{-\alpha} \leq x_{n+1} \leq 1$. Similar convergence criteria will be used the rest of the non-linear functions too.

To apply the aSOR method for the sign function, we first need to pre-calculate the relaxation factors $k_i$ with $1 \leq i \leq n$ as given in Algorithm 1.

Specifically, to get the $k_i$ for Equation 2, the maxima criteria occurs at $k_i = \sqrt{3/(\epsilon_i^2 + \epsilon_i + 1)}$. Next, we can immediately apply this method perform a faster sign function as given in Algorithm 2. Numerical evaluations are given in Table 1 and Table 2.

## 4.2 Inversion

**Goldschmidt's Division Algorithm** Goldschmidt's [20] division algorithm is a well-known algorithm to perform a division operation by iterative multiplications. More precisely, for $x \in (0,2)$ the inversion could be interpreted as,

$$\begin{aligned}\frac{1}{x} &= \frac{1}{1-(1-x)} \\ &= \frac{1+(1-x)}{1-(1-x)^2} \\ &= \frac{(1+(1-x))(1+(1-x)^{2^1})}{1-(1-x)^{2^2}} \\ &= \cdots \\ &= \frac{(1+(1-x))\cdots(1+(1-x)^{2^{n-1}})}{1-(1-x)^{2^n}} \qquad (3)\end{aligned}$$

and the denominator converges to 1 for a large enough $n$. Thus, by computing and multiplying $\left(1+(1-x)^{2^i}\right)$ iteratively, we can approximate $1/x$. The authors in [10] proposed a new method for computing inversion via Goldschmidt's division algorithm. For $x \in (0,2)$, Equation (3) implies that an approximated value of $1/x$ can be obtained by

$$\frac{1}{x} \approx \prod_{i=0}^{n} \left(1+(1-x)^{2^i}\right)$$

They prove that for $x \in [2^{-n}, 1)$ their construction required $\Theta(\log \alpha + n)$ iterations to converge to $1/x$ with an error bound of $2^{\alpha}$.

Instead of taking their approach, we take a different formulation of Goldschmidt's division algorithm. Let $a_i$ and $b_i$ be denominator and numerator after $i$ iterations, respectively. Then,

$$\begin{aligned}a_i &= 1-(1-x)^{2^i} \\ b_i &= \prod_{i=0}^{n} \left(1+(1-x)^{2^{i-1}}\right)\end{aligned}$$

and $a_{i+1}$ can be re-written as

$$\begin{aligned}a_{i+1} &= 1-(1-x)^{2^{i+1}} \\ &= \left(1-(1-x)^{2^i}\right)\left(1+(1-x)^{2^i}\right) \\ &= a_i(2-a_i) \qquad (4)\end{aligned}$$

**Algorithm 2** Sign($x, \alpha, \epsilon$)

**INPUT:** $x \in [-1, -\epsilon] \cup [\epsilon, 1]$, $\alpha, \epsilon \in \mathbb{R} \cap [0, 1]$
**OUTPUT:** an approximate value of 1 if $x > 0$, -1 if $x < 0$ and 0 otherwise

```
1: $f \leftarrow$ Equation 2.
2: $\epsilon_1 \leftarrow \epsilon$ and $i \leftarrow 1$
3: $K \leftarrow RelaxationFactor(f, \alpha, \epsilon)$
4: while $1 - \epsilon_i > 2^{-\alpha}$ do
5:     $k_i \leftarrow i$-th element of $K$
6:     $x \leftarrow f(k_i x)$
7:     $\epsilon_{i+1} \leftarrow f(k_i)$
8:     $i \leftarrow i + 1$
9: end while
10: return $x$
```

Similarly, $b_{i+1}$ becomes

$$\begin{aligned} b_{i+1} &= \prod_{i=0}^{n} \left(1 + (1 - x)^{2^{i+1}}\right) \\ &= \left(1 + (1 - x)^{2^{i}}\right) \prod_{i=0}^{n} \left(1 + (1 - x)^{2^{i-1}}\right) \\ &= b_i(2 - a_i). \end{aligned}$$

When $a_i$ is close to 1, $b_i$ is close to $1/x$. Therefore, Goldschmidt's inversion algorithm — Equation 4, for input $x \in \mathbb{R}$ can be re-expressed as evaluating $f(x) = x(2-x)$ iteratively. This function is similar to $f_n(x)$ used in the previous section: both functions get through the origin (0, 0) and (1, 1) and monotonically increase for $0 \leq x \leq 1$ and monotonically decrease for $x > 1$. Hence, the relaxation factor idea can also be utilized with $f(x) = x(2 - x)$. It then follows that,

$$\begin{aligned} b_{i+1} &= k_i b_i (2 - k_i a_i) \\ a_{i+1} &= k_i a_i (2 - k_i a_i) \end{aligned}$$

where $a_1 = x$, $y_1 = 1$, and $k_i$ is an relaxation factor for the $i$-th iteration of the composite evaluation.

This relaxation factor can also be computed by Algorithm 1. That is, once a set of $k_i$ for particular $\epsilon, \alpha$ are computed there is no need for repeat calculations. This has been summarized in Algorithm 3.

## 4.3 Inverse Square Root and Square Root

**Goldschmidt's inverse Square Root Algorithm** Goldschmidt's inverse square root algorithm is a numerical method achieved by iterative process. This

**Algorithm 3** Inverse($\alpha, \epsilon, x$)

**INPUT:** $x \in [\epsilon, 1]$, $\alpha, \epsilon \in \mathbb{R} \cap [0, 1]$
**OUTPUT:** an approximate value of $1/x$

```
1: $f \leftarrow z(2 - z)$
2: $\epsilon_1 \leftarrow \epsilon$ and $i \leftarrow 1$
3: $a \leftarrow x$
4: $b \leftarrow 1$
5: $K \leftarrow RelaxationFactor(f, \alpha, \epsilon)$
6: **while** $1 - \epsilon_i > 2^{-\alpha}$ **do**
7:     $k_i \leftarrow i$-th element of $K$
8:     $b \leftarrow k_i b(2 - k_i a)$
9:     $a \leftarrow k_i a(2 - k_i a)$
10:    $\epsilon_{i+1} \leftarrow k_i(2 - k_i)$
11:    $i \leftarrow i + 1$
12: **end while**
13: **return** $b$
```

algorithm utilizes composable $f(x) = x(3 - x)^2/4$ that converges to 1 for the input range of $(0, 1]$. This composite function also gets through the origin (0, 0) and (1, 1) and monotonically increases for $0 \leq x \leq 1$ and monotonically decreases for $x > 1$. Therefore, aSOR is to be applied here similarly. In Algorithm 1, the optimal relaxation factor $k_i$ can be achieved by finding the first positive $k_i$ that satisfies $f(k_i \epsilon_i) = f(k_i)$.

Formally, similar to the Goldschmidt's algorithm,

$$\begin{aligned} \frac{1}{x} = \frac{1}{a_0} &= \frac{(3 - a_0)^2/4}{a_0(3 - a_0)^2/4} = \frac{b_1^2}{a_1} \\ &= b_1^2 \frac{(3 - a_1)^2/4}{a_1(3 - a_1)^2/4} = b_1^2 \frac{b_2^2}{a_2} \\ &= b_1^2 b_2^2 \frac{(3 - a_2)^2/4}{a_2(3 - a_2)^2/4} = b_1^2 b_2^2 \frac{b_3^2}{a_3} \\ &= \cdots \\ &= b_1^2 b_2^2 \cdots b_{n-2}^2 b_{n-1}^2 \frac{(3 - a_{n-1})^2/4}{a_{n-1}(3 - a_{n-1})^2/4} \\ &= b_1^2 b_2^2 \cdots b_{n-2}^2 b_{n-1}^2 \frac{b_n^2}{a_n} \approx \left( \prod_i^n b_i \right)^2 \end{aligned} \tag{5}$$

$$\frac{1}{\sqrt{x}} = \prod_i^n b_i \tag{6}$$

When $a_n$ converges to 1, $\prod_i^n b_i$ converges to $\frac{1}{\sqrt{x}}$ according to Equations 5 and 6. If $a_n$ converges to 1 faster via relaxation factors $k_i$, $\prod_i^n b_i$ will also converge to $\frac{1}{\sqrt{x}}$. For the sake of completeness, the procedure of applying the relaxation factor for such iterative process is given below,

$$\begin{aligned}
\frac{1}{x} &= \frac{1}{a_0} = \frac{k_0(3-k_0a_0)^2/4}{k_0a_0(3-k_0a_0)^2/4} = \frac{b_1^2}{a_1} \\
&= b_1^2\frac{k_1(3-k_1a_1)^2/4}{k_1a_1(3-k_1a_1)^2/4} = b_1^2\frac{b_2^2}{a_2} \\
&= b_1^2b_2^2\frac{k_2(3-k_2a_2)^2/4}{k_2a_2(3-k_2a_2)^2/4} = b_1^2b_2^2\frac{b_3^2}{a_3} \\
&= \cdots \\
&= b_1^2b_2^2\cdots b_{n-2}^2b_{n-1}^2\frac{k_{n-1}(3-k_{n-1}a_{n-1})^2/4}{k_{n-1}a_{n-1}(3-k_{n-1}a_{n-1})^2/4} \\
&= \frac{b_1^2b_2^2\cdots b_n^2}{a_n} \approx \left(\prod_i^n b_i\right)^2. \qquad (7)
\end{aligned}$$

The algorithm is given in Algorithm 4.

**Goldschmidt's Square Root Algorithm** Since $\sqrt{x} = x/\sqrt{x}$, we can readily reuse the previous algorithm with minimal modification as,

$$\frac{1}{\sqrt{x}} = \prod_i^n b_i$$
$$\sqrt{x} = a_i\prod_i^n b_i.$$

The algorithm is also given in Algorithm 4, with a slight modification that $b \leftarrow x$ for square root calculation.

## 4.4 Reciprocal of m-th root

**Newton-Raphson Method** Newton–Raphson [1] method is a root-finding algorithm that returns approximate roots of a polynomial. Given a function $g$ defined over $x \in \mathbb{R}$, and its derivative $g'$, we begin with a first guess $y_1$ for the root of the function $g$. A better approximation for the root is $y_2$ is $y_2 = y_1 - \frac{g(y_1)}{g'(y_1)}$. Geometrically, $(y_1, 0)$ is the intersection with the $x$-axis of the tangent to the graph of $f$ at $(y_1, g(y_1))$. The process is repeated as

$$y_{i+1} = y_i - \frac{g(y_i)}{g'(y_i)} = f(y_i)$$

until a sufficiently accurate value is reached. In this paper, we utilize such an iterative procedure to approximate reciprocal m-th root of $x$, that is, $x^{-\frac{1}{m}}$.

Let $g(y_i) = y_i^{-\frac{1}{m}} - x$, by Newton-Raphson's method, we have

$$y_{i+1} = y_i - \frac{g(y_i)}{g'(y_i)} = y_i \left( \frac{(m+1) - x y_i^m}{m} \right) = f(y_i). \tag{8}$$

This iterative process can be repeated with a starting value of $y_1 = 1$ to obtain increasingly better approximations to $x^{-\frac{1}{m}}$. However, we cannot blindly apply the relaxation factor yet as this function $f$ does not pass through $(1, 1)$ for an arbitrary input $x$.

We can overcome this limitation by introducing a dummy intermediate variable $z_i = x^{\frac{1}{m}} \cdot y_i$ which changes the iterative updating rule to (from Equation 8),

$$\begin{aligned} x^{\frac{1}{m}} y_{i+1} &= x^{\frac{1}{m}} y_i \left( \frac{(m+1) - x y_i^m}{m} \right) \\ z_{i+1} &= z_i \left( \frac{(m+1) - z_i^m}{m} \right) \\ &= f(z_i). \end{aligned}$$

Here, $f(x)$ converges to 1 for the input range of $(0, 1]$. This composite function also passes the origin (0, 0) and (1, 1); monotonically increases for $0 \le x \le 1$; and monotonically decreases for $x > 1$. Therefore, aSOR can be applied here similarly.

If the range of $x$ is given as $0 < \epsilon_1 \le x \le 1$, then $x^{\frac{1}{m}} \in [\epsilon_1^{\frac{1}{m}}, 1]$ and $z_1 \in [\epsilon_1^{\frac{1}{m}} y_1, y_1]$ as $z_i = x^{\frac{1}{m}} y_i$. If $y_1 = 1$, then the minimum range limit of $z_1$ is $\epsilon_1^z = \epsilon_1^{\frac{1}{m}}$ and the maximum range limit of $z_1$ is 1, which is similar to the sign function with the range of $[\epsilon_1^z, 1]$. This enables us to use relaxation method for the reciprocal of m-th root.

The relaxation factor $k_i$ for each $i$-th iteration can be computed so that $f(k_i \epsilon_i^z) = f(k_i)$, and $y_{i+1} = k_i y_i ((m+1) - x(k_i y_i)^m)/m$. After $n$ iterations of composite polynomial, $y_n$ can be finally calculated where $n$ is pre-determined by $\alpha$. This has been summarized in Algorithm 5.

## 5 Performance Analysis

Importantly, the methods described in this work are not tied to CKKS [9] or even FHE in general. To our knowledge, CKKS happens to be the most efficient scheme for the described aSOR method. We will briefly provide reasons on why aSOR method could implemented efficiently with CKKS in the following section.

### 5.1 CKKS Scheme and Scale Adjustment

CKKS scheme [9] and its RNS variant [8] are the state-of-the-art FHE schemes that support approximate arithmetic for addition and multiplication operations. CKKS scheme encodes a raw data vector into a plaintext cyclotomic polynomial and then encrypts the encoded plaintext using the RLWE (Ring Learning with Errors) cryptosystem [32].

**Algorithm 4** (Inverse)SquareRoot$(x, \alpha, \epsilon)$

**INPUT:** $x \in [\epsilon, 1]$, $\alpha, \epsilon \in \mathbb{R} \cap [0, 1]$
**OUTPUT:** an approximate value of $1/x$

```
1: $f \leftarrow z(3 - z)^2/4$
2: $\epsilon_1 \leftarrow \epsilon$ and $i \leftarrow 1$
3: $a \leftarrow x$
4: $b \leftarrow x$
5: $b \leftarrow 1$ (In case of inverse square root)
6: $K \leftarrow RelaxationFactor(f, \alpha, \epsilon)$
7: while $1 - \epsilon_i > 2^{-\alpha}$ do
8:     $k_i \leftarrow i$-th element of $K$
9:     $b \leftarrow \sqrt{k_i} b(3 - k_i a)/2$
10:    $a \leftarrow k_i a(3 - k_i a)^2/4$
11:    $\epsilon_{i+1} \leftarrow k_i(3 - k_i)^2/4$
12:    $i \leftarrow i + 1$
13: end while
14: return $b$
```

One of the main challenges of the CKKS scheme is the approximation error that is inherent to almost every operation in the scheme [14, 26]. The CKKS scheme introduces several sources of error including encoding, encryption, rescaling, and relinearization errors. The CKKS error analysis and its impact on the aSOR method are presented in Appendix B.

The CKKS scheme uses fixed-point arithmetic. A data vector is scaled with a large integer, called the scaling factor $\Delta$, and then rounded to the integer before the encryption. When two data vectors encrypted with the CKKS scheme are multiplied homomorphically, the scaling factors of the two are also multiplied. This scaling factor should be reduced to the original value by using the *rescaling* operation. Each encoded plaintext can be interpreted as an integer vector $\mathbf{x}$, which could also potentially overflow just like machine integers, divided by its scale $\Delta$. As a mental analogy, it is helpful to imagine a ciphertext as $\mathbf{x}/\Delta$.

The scale of a ciphertext can be modified as follows:

$$\mathbf{x}/\Delta_1 \times k = \mathbf{x}/\Delta_2.$$

This scale adjustment results in an efficient multiplication/division of the ciphertext by a scalar $k$, which consumes no depth and could potentially reduce the scale at the same time. This is extremely similar to machine bit-shifting for fast multiplication/division by the powers of two. Therefore, such multiplications by a relaxation factor $k$ can be done nearly free of computational cost, which is why this method works so well for this particular FHE scheme.

We believe that this method has not been explored before in other fields due to the requirement of multiplication by a constant factor $k$, which can sometimes be as computationally expensive as running an additional iterative evaluation loop.

```
Algorithm 5 ReciprocalRoot($x, m, \alpha, \epsilon$)
INPUT: $x \in [\epsilon, 1]$, $\alpha, \epsilon \in \mathbb{R} \cap [0, 1]$
OUTPUT: An approximate value of $x^{-\frac{1}{m}}$

 1: $f \leftarrow z((m+1) - z^m)/m$
 2: $\epsilon_1 \leftarrow \epsilon$ and $i \leftarrow 1$
 3: $y \leftarrow 1$
 4: $K \leftarrow RelaxationFactor(f, \alpha, \epsilon)$
 5: while $1 - \epsilon_i > 2^{-\alpha}$ do
 6:     $k_i \leftarrow i$-th element of $K$
 7:     $y \leftarrow k_i y((m+1) - x(k_i y)^m)/m$
 8:     $\epsilon_{i+1} \leftarrow k_i((m+1) - k_i^m)/m$
 9:     $i \leftarrow i + 1$
10: end while
11: return $y$
```

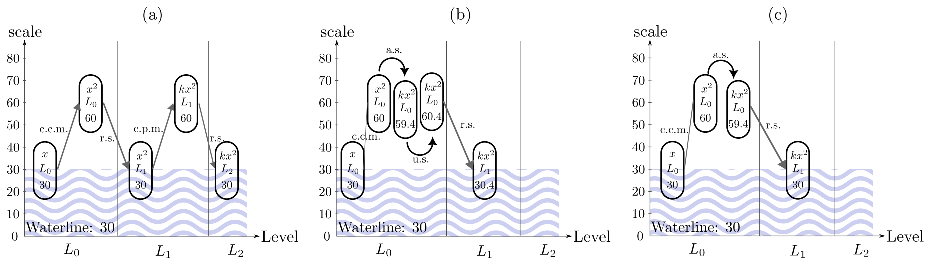
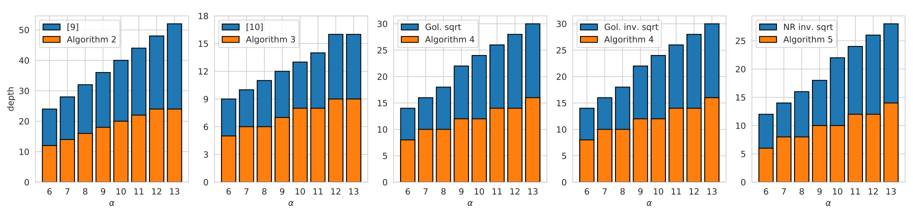


Figure 4: Comparison of the scale management of naive method and proposed method to compute $f(x) = kx^2$ when $k = 1.5$. c.c.m. : ciphertext-ciphertext multiplication, c.p.m. : ciphertext-plaintext multiplication, a.s. : adjusting scale r.s. : rescaling, u.s. : upscaling by $2^1$. (a) Without scale adjustment (b) With scale adjustment (c) With scale adjustment by pre-selected prime modulus.

CKKS scale management becomes more involved as it is now necessary to decide on the optimal rescaling moduli, based on the pre-calculated array of $k_i$ values with the scale adjustment in mind. In the most optimal scenario, the scale should be always rescaled back to the "waterline level" (defined as the lowest scale without loss of significant bits [31]) after each iterative evaluation to maximally suppress growth of the error bits in the ciphertexts.

Such scale adjustment might seem to come at the cost of having haphazard scales across ciphertexts. However, after performing any of the operations discussed before, the scale could be brought back to an arbitrary level via multiplication by unity of the needed scale. Such upscaling when necessary has been used before in CKKS compilers such as HECATE [31]. Ideally, our methodology should be embedded within a CKKS circuit compiler, where the compiler would automatically decide on the most optimal rescaling moduli to perform aSOR



(a) Sign (b) Inverse (c) Square root (d) Inverse sqrt (Gol.) (e) Inverse sqrt (NR)

Figure 5: Summary of depth results for basic non-linear functions.

optimally.

As an example, let us imagine a computation of $f(x) = kx^2$ with $k = 1.5$ for the sake of compatibility of the presented algorithms but the value of $k$, in principle, does not matter. Let us also assume that the scales, waterline scale, and the rescaling moduli are all equal to $2^{30}$ and the $x$ ciphertext starts at level 0 (assuming increasing level convention as we progress further along the computation). Naive computation of $f$ would require two rescaling operations and hence at least two levels (equivalent to computing $x^3$) as illustrated in Figure 4 (a).

As shown in Figure 4 (b) and (c), the calculation could be carried out more efficiently with scale adjustment via the following steps:

1. Calculate $\text{square}(x) = x^2$: scale $2^{60}$ and level 0.

2. Adjust the scale of $\text{square}(x) = x^2$ by $\log_2 1.5 = 0.585$ to complete the calculation of $kx^2$: scale $2^{60}/\log_2 1.5 = 2^{59.415}$ and level 0 (rescaling now would result in loss of significant bits as the result would fall below the waterline scale).

3. Upscale by multiplication of nearest integer scale unity $2^{60-59.00}$ to make the resulting ciphertext to have the scale $2^{60.415}$. Alternatively, the rescaling modulus could be chosen to be around $2^{29.415}$ to exactly match the needed rescaling value — Figure 4 (b) and (c); this would be even more efficient as we would not need perform cipher-plain multiplication.

4. Apply relinearization and rescaling: scale $2^{30.415}$ and level 1.

Performing this “scale trick” not only reduces the number of rescaling operations but could potentially remove one multiplication (if the next rescaling modulus is chosen accordingly) resulting in a much more efficient calculation overall. Additionally, if the values of $k$ are comparably large with rescaling moduli, this methodology could effectively “rescale” ciphertexts without explicitly performing the rescaling operation honestly.

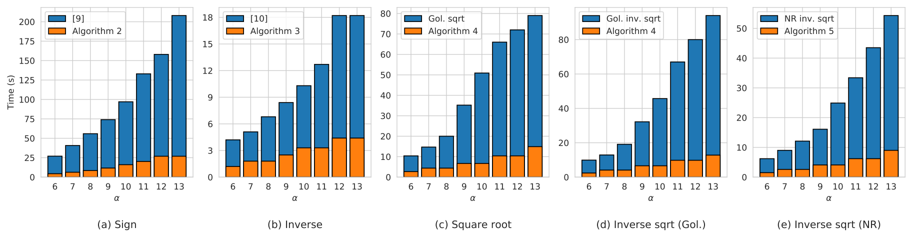


Figure 6: Summary of time results for basic non-linear functions.

## 5.2 Benchmarks of Basic Non-linear Functions

Our experiments were conducted on Linux Ubuntu 18.04.6 using AMD EPYC 7502-32 CPU. We used CKKS scheme implemented in Microsoft SEAL library [34] with the highest polynomial degree of $2^{17}$ and the initial scale $\Delta = 2^{40}$. In the experiment, we compare our algorithm to the previous work using three criteria: 1) the number of iterations, 2) the required multiplicative depth, 3) the computational complexity, and 4) the computational time in encrypted state. In this section, we will consider $\epsilon = 2^{-\alpha}$ to make the same precision of input and output of all operations. The results are concisely summarized in Table A.1, where *Depth* indicates multiplicative depth, and *Complexity* indicates computational complexity. All of the metrics tend to show better improved with increasing $\alpha$ value.

The baseline of our benchmarks is Cheon et al.'s works [10] and [7] as they reviewed a large set of functions while being simple in implementation. Multiplicative depth benchmarks are summarized in Figure 5, and time benchmarks are summarized in Figure 6. Complete benchmarks are given in Table A.1 to avoid visual clutter.

**Comparison** Figure 5 (a) shows the multiplicative depth comparison results for comparison function. The time in Figure 6 (a) is not amortized, i.e to get the amortized time per ciphertext slot the time should be divided by $2^{16}$. Algorithm 2 outperforms the previous work in all metrics — Table A.1. The multiplicative depth and the number of multiplications are both reduced, indicating a no-compromises improvement. Generally, the aSOR method reduces the computation time by about 6–9 times — Figure 6 (a).

Lee et al. [29] showed a similar performance improvement to this work, but they focused only on the comparison function. Additionally, their work requires sophisticated pre-calculations using dynamic programming.

**Inversion** Figure 5 (b) and Figure 6 (b) show that inverse operation is improved in depth, time and the ratios have a tendency to increase as $\alpha$ increases.

**Goldschmidt Square Root** Figure 5 (c) and Figure 6 (c) show that our method improves Goldschmidt Square Root algorithm in depth and especially in computation time considerably; almost 4 times faster when $\alpha$ is 6, which increases to about 8 times faster when $\alpha$ is 13.

Goldschmidt Square Root and Inverse Square Root are mostly the same algorithm except that they have a different initialization stage (Algorithm 4). Depth and time comparisons are given in Figure 5 (d) and Figure 6 (d).

**Goldschmidt and Newton Raphson Inverse Square Root** Figure 5 (e) and Figure 6 (e) compare the performance of aSOR for Inverse Square Root operation. Our method reduces the computation time significantly when applying it to both inverse square root algorithms — 4–9 times faster.

While Goldschmidt algorithm shows the better performance in complexity, Newton-Raphson method outperforms it in depth and time. For FHE applications, it is suggested to use Newton-Raphson method as it consumes less depth which is very much desirable in practice — Table A.1.

## 5.3 Applications

The methodologies developed in this work can be readily applied to various functions. Since word-wise FHE schemes are natively compatible with addition, subtraction and multiplication, the importance of an efficient methodology to perform the division operation alone (Algorithm 3) cannot be overstated. The following subsections demonstrate the possible applications of the presented methods in this paper.

**Softmax** The softmax function takes in a vector of real numbers and outputs a valid vector of probabilities. For a vector $X = (x_1, \ldots, x_t)$, softmax is defined as,

$$\text{softmax}(X) = \frac{1}{\sum_{i=i}^{t} \exp(x_i)} (\exp(x_1), \exp(x_2), \ldots, \exp(x_t)).$$

In order to approximate the exponential function, we utilized the elementary definition of $\exp(x)$ [11] as,

$$\exp(x) = \lim_{k \to \infty} \left(1 + \frac{x}{n}\right)^n \simeq \left(1 + \frac{x}{2^r}\right)^{2^r}.$$

Next, the inverse function can be approximately computed via Goldschmidt's division algorithm (Algorithm 3). Almost a two-fold improvement over the previous methodology [23] has been achieved as shown in Table 3 (3 classes, $\alpha = 2^{-7}$, $\epsilon = 2^{-9},\ 2^{-15},\ 2^{-29}$ per column).

**Max pooling** Max pooling is a common operation typically found in convolutional neural networks, which essentially reduces to computation of maximal value among 2-dimensional array of pixels. For $n$ distinct positive number of

Table 3: Summary of results for Softmax

| | Softmax range | $[-3,3]$ | $[-5,5]$ | $[-10,10]$ |
|---|---|---|---|---|
| Ref [23] | Depth | 17 | 29 | 46 |
| | Complexity | 30 | 49 | 68 |
| Ours | Depth | 12 | 20 | 29 |
| | Complexity | 20 | 31 | 34 |

Table 4: Summary of results for Max Pooling

| | Filter size | $2 \times 2$ | $3 \times 3$ | $4 \times 4$ |
|---|---|---|---|---|
| Ref [11] | Depth | 42 | 52 | 59 |
| | Complexity | 122 | 217 | 336 |
| Ours | Depth | 27 | 32 | 36 |
| | Complexity | 92 | 177 | 290 |

pixels $(a_1, a_2, ..., a_n)$, the maximal value $max(a_1, a_2, ..., a_n)$ could be calculated using the following identity [23],

$$\lim_{k\to\infty} \frac{a_i^k}{a_1^k + \cdots + a_n^k} = \begin{cases} 1, & \text{if } a_i \text{ is maximal.} \\ 0, & \text{otherwise.} \end{cases}$$

For this application $k = 2^7,\ \alpha = 2^{-7},\ \epsilon = 1/255$ were chosen to demonstrate the potential improvement in both complexity and depth of the algebraic circuits as shown in Table 4.

## 6 Conclusion

We propose adaptive successive over-relaxation method (aSOR). aSOR method derives inspiration from the usual SOR method and implements adaptive (varying) relaxation factor to always have the greatest impact on the convergence.

To our knowledge, the methodology performs the best when used in tandem with FHE schemes such as CKKS due to the fact that constant multiplications could be performed at practically free of computational cost. We have used SEAL library implementation of the CKKS scheme and performed our benchmarks using various input/output precision specifications. With CKKS scheme we find that the overall improvement of evaluation of the non-linear functions is around 2 to 9 times in terms of execution time.

Concrete and ready-to-use evaluation algorithms are presented in detail for sign, (inverse) square root, and negative inverse power functions. Given the today's needs for privacy preserving solutions for real-world applications in statistics, machine learning and other disciplines, such ready-to-use algorithms are crucial for usage of FHE in industrial and practical applications.

## A Detailed benchmarks

Detailed performance metrics for all the algorithms presented in this work are listed in Table A.1.

## B CKKS error analysis

Following Section 3, the iterative operations of

$$f([\epsilon_n, 1]) = [\epsilon_{n+1}, 1]$$

could incur errors during encryption and evaluations in CKKS scheme.

We adopt the notation of some distributions from [8, 9]. For a real $\sigma > 0$, $\mathrm{DG}(\sigma^2)$ samples a vector in $\mathbb{Z}^N$ by drawing its coefficient independently from the discrete Gaussian distribution of variance $\sigma^2$. For a positive integer $h$, $\mathrm{HWT}(h)$ is the set of signed binary vectors in $0, \pm 1^N$ for which Hamming weight is exactly $h$. Next, $l, B_i, \nu_i$ denote a level, an upper bound on the message $m_i$, and the noise of the encrypted message $m_i$ respectively.

Let us call $\mathcal{E}_n^{\text{enc}}$, $\mathcal{E}_n^{\text{add}}$, $\mathcal{E}_n^{\text{mul}}$, $\mathcal{E}_n^{\text{ks}}$, $\mathcal{E}_n^{\text{rs}}$ as maximal absolute errors which could occur during encoding and encryption, addition, multiplication, key switching (for relinearization and rotation), and rescaling respectively during the n-th iteration. Then, the errors can be defined as follows [9]:

$$\begin{aligned}
\mathcal{E}_n^{\text{enc}} &= 8\sqrt{2}\sigma N + 6\sigma\sqrt{N} + 16\sigma\sqrt{hN} \\
\mathcal{E}_n^{\text{add}} &= B_1 + B_2 \\
\mathcal{E}_n^{\text{mul}} &= \nu_1 B_2 + \nu_2 B_1 + B_1 B_2 \\
\mathcal{E}_n^{\text{ks}} &= B + \frac{8}{\sqrt{3}} N\sigma \frac{Q}{P}\sqrt{l^2+1} + (k+1)(\sqrt{3N} + \frac{8}{\sqrt{3}}\sqrt{hN}) \\
\mathcal{E}_n^{\text{rs}} &= q_l^{-1} B + \sqrt{3N} + \frac{8}{\sqrt{3}}\sqrt{hN}
\end{aligned}$$

The size of errors can be managed by choosing a proper FHE parameter set.

For convenience, let us denote $\mathcal{E}_n$ as the total error per iteration ($\mathcal{E}_{n>0}^{\text{enc}} = 0$). To completely guarantee the correctness of our algorithms in this work, the input ranges could be conservatively redefined as,

$$[\epsilon_{n+1}, 1] \rightarrow [\epsilon_{n+1} - \mathcal{E}_n, 1 + \mathcal{E}_n].$$

This range needs to re-normalized to $[\epsilon,\ 1]$-like form before proceeding with the next iteration. To do so, this range is divided by the maximum value $1 + \mathcal{E}_n$, such that

$$\left[\frac{\epsilon_{n+1} - \mathcal{E}_n}{1 + \mathcal{E}_n},\ 1\right]$$

is the new range. We can re-define a new minimum input value as,

$$\epsilon'_{n+1} = \frac{\epsilon_{n+1} - \mathcal{E}_n}{1 + \mathcal{E}_n},$$

and proceed with the next $n + 1$-th iteration as usual. In practice and in this work, such precautious measures are not necessary when the $\epsilon_i$ values are sufficiently large.

## References

[1] Saba Akram and Quarrat Ul Ann. “Newton Raphson method”. In: *International Journal of Scientific & Engineering Research* 6.7 (2015), pp. 1748–1752.

[2] Ahmad Al Badawi et al. “OpenFHE: Open-Source Fully Homomorphic Encryption Library”. In: *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC’22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 53–63. DOI: 10.1145/3560827.3563379. URL: https://doi.org/10.1145/3560827.3563379.

[3] Sagarika Behera et al. “Preserving the Privacy of Medical Data using Homomorphic Encryption and Prediction of Heart Disease using K-Nearest Neighbor”. In: *2022 IEEE International Conference on Data Science and Information System (ICDSIS)*. July 2022, pp. 1–6. DOI: 10.1109/ICDSIS55133.2022.9915983.

[4] Charlotte Bonte and Ilia Iliashenko. “Homomorphic String Search with Constant Multiplicative Depth”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*. CCSW’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 105–117. ISBN: 9781450380843. DOI: 10.1145/3411495.3421361. URL: https://doi.org/10.1145/3411495.3421361.

[5] Zvika Brakerski. “Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP”. In: *Advances in Cryptology - CRYPTO*. 2012, pp. 868–886.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) fully homomorphic encryption without bootstrapping”. In: *ITCS*. 2012, pp. 309–325.

[7] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. “Efficient Homomorphic Comparison Methods with Optimal Complexity”. In: *Advances in Cryptology - ASIACRYPT*. Vol. 12492. 2020, pp. 221–256.

[8] Jung Hee Cheon et al. “A full RNS variant of approximate homomorphic encryption”. In: *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer. 2019, pp. 347–368.

[9] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *Advances in Cryptology - ASIACRYPT*. 2017, pp. 409–437.

[10] Jung Hee Cheon et al. "Numerical Method for Comparison on Homomorphically Encrypted Numbers". In: *Advances in Cryptology - ASIACRYPT*. 2019, pp. 415–445.

[11] Jung Hee Cheon et al. "Numerical method for comparison on homomorphically encrypted numbers". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2019, pp. 415–445.

[12] Ilaria Chillotti et al. "Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE". In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Cham: Springer International Publishing, 2017, pp. 377–408. ISBN: 978-3-319-70694-8.

[13] Ilaria Chillotti et al. "TFHE: Fast Fully Homomorphic Encryption Over the Torus". In: *J. Cryptol.* 33.1 (2020), pp. 34–91.

[14] Anamaria Costache et al. *On the precision loss in approximate homomorphic encryption*. Cryptology ePrint Archive, Paper 2022/162. https://eprint.iacr.org/2022/162. 2022. URL: https://eprint.iacr.org/2022/162.

[15] Jr David M. Young. "Iterative methods for Solving Partial Difference Equations of Elliptic Type". 1950. URL: https://web.ma.utexas.edu/CNA/DMY/david_young_thesis.pdf.

[16] Nathan Dowlin et al. "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 201–210.

[17] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second". In: *Advances in Cryptology - EUROCRYPT*. 2015, pp. 617–640.

[18] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144. 2012. URL: https://eprint.iacr.org/2012/144.

[19] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *STOC*. 2009, pp. 169–178.

[20] Robert E Goldschmidt. "Applications of division by convergence". PhD thesis. Massachusetts Institute of Technology, 1964.

[21] Shai Halevi and Victor Shoup. "HElib: Software library for homomorphic encryption". http://github.com/shaih/HElib.git. 2018.

[22] Ehsan Hesamifard et al. "Privacy-preserving Machine Learning as a Service". In: *PoPETs* 2018.3 (2018), pp. 123–142.

[23] Seungwan Hong et al. "Secure tumor classification by shallow neural network using homomorphic encryption". In: *BMC genomics* 23.1 (2022), pp. 1–19.

[24] CryptoLab Inc. *HEaaN: Homomorphic Encryption for Arithmetic of Approximate Numbers*. `https://heaan.it/`. 2022.

[25] Nikola Jovanovic et al. "Private and Reliable Neural Network Inference". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1663–1677. ISBN: 9781450394505. DOI: `10.1145/3548606.3560709`. URL: `https://doi.org/10.1145/3548606.3560709`.

[26] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. "Approximate Homomorphic Encryption with Reduced Approximation Error". In: *Topics in Cryptology – CT-RSA 2022*. Ed. by Steven D. Galbraith. Cham: Springer International Publishing, 2022, pp. 120–144. ISBN: 978-3-030-95312-6.

[27] Jongmin Kim et al. "SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. Orlando, FL, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: `10.1145/3579371.3589053`. URL: `https://doi.org/10.1145/3579371.3589053`.

[28] Miran Kim et al. "Ultrafast homomorphic encryption models enable secure outsourcing of genotype imputation". In: *Cell Systems* 12.11 (2021), 1108–1120.e4. ISSN: 2405-4712. DOI: `https://doi.org/10.1016/j.cels.2021.07.010`. URL: `https://www.sciencedirect.com/science/article/pii/S240547122100288X`.

[29] Eunsang Lee et al. "Minimax approximation of sign function by composite polynomial for homomorphic comparison". In: *IEEE Transactions on Dependable and Secure Computing* (2021).

[30] Joon-Woo Lee et al. "Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network". In: *IEEE Access* 10 (2022), pp. 30039–30054.

[31] Yongwoo Lee et al. "HECATE: Performance-Aware Scale Optimization for Homomorphic Encryption Compiler". In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 193–204. DOI: `10.1109/CGO53902.2022.9741265`.

[32] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings". In: *Advances in Cryptology - EUROCRYPT*. 2010, pp. 1–23.

[33] Ahmet Can Mert et al. "Medha: Microcoded Hardware Accelerator for computing on Encrypted Data". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.1 (Nov. 2022), pp. 463–500. DOI: 10.46586/tches.v2023.i1.463-500. URL: https://tches.iacr.org/index.php/TCHES/article/view/9959.

[34] Microsoft. *SEAL: Simple Encrypted Arithmetic Library.* https://github.com/Microsoft/SEAL. 2022.

[35] Kundan Munjal and Rekha Bhatia. "A systematic review of homomorphic encryption and its contributions in healthcare industry". In: *Complex & Intelligent Systems* (May 2022), pp. 1–28. ISSN: 2199-4536. DOI: 10.1007/s40747-022-00756-z.

[36] Samanvaya Panda. *Polynomial Approximation of Inverse sqrt Function for FHE.* Cryptology ePrint Archive, Paper 2022/423. 2022.

[37] J Raphson. *Analysis Aequationum Universalis. London.* 1690.

[38] Xuanle Ren et al. "HEDA: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database". In: *Proc. VLDB Endow.* 16.4 (Dec. 2022), pp. 601–614. ISSN: 2150-8097. DOI: 10.14778/3574245.3574248. URL: https://doi.org/10.14778/3574245.3574248.

[39] Nikola Samardzic et al. "CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture.* ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 173–187. ISBN: 9781450386104. DOI: 10.1145/3470496.3527393. URL: https://doi.org/10.1145/3470496.3527393.

[40] M. Siva Sangari et al. "A Survey on Homomorphic Encryption for Financial Cryptography Workout". en. In: *Homomorphic Encryption for Financial Cryptography: Recent Inventions and Challenges.* Cham: Springer International Publishing, 2023, pp. 13–27. ISBN: 978-3-031-35535-6. DOI: 10.1007/978-3-031-35535-6_2. URL: https://doi.org/10.1007/978-3-031-35535-6_2.

[41] Olamide Timothy Tawose et al. "Toward Efficient Homomorphic Encryption for Outsourced Databases through Parallel Caching". In: *Proc. ACM Manag. Data* 1.1 (May 2023). DOI: 10.1145/3588920. URL: https://doi.org/10.1145/3588920.

[42] Wikipedia contributors. *Successive over-relaxation — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Successive_over-relaxation&oldid=1109147320. [Online; accessed 26-September-2022]. 2022.

[43] Ahmed El-Yahyaoui and Mohamed Dafir EC-Chrif El Kettani. “Fully Homomorphic Encryption: Searching over Encrypted Cloud Data”. In: *Proceedings of the 2nd International Conference on Big Data, Cloud and Applications*. BDCA'17. Tetouan, Morocco: Association for Computing Machinery, 2017. ISBN: 9781450348522. DOI: `10.1145/3090354.3090364`. URL: `https://doi.org/10.1145/3090354.3090364`.

[44] Zama. *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`. 2022.

Table A.1: Summary of benchmark results for basic non-linear functions

| | $\alpha$ | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|
| Sign algorithm as in [7] | Iterations | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 |
| | Depth | 24 | 28 | 32 | 36 | 40 | 44 | 48 | 52 |
| | Complexity | 24 | 28 | 32 | 36 | 40 | 44 | 48 | 52 |
| | Time (s) | 27.0 | 40.7 | 55.9 | 74.0 | 97.0 | 133.0 | 159.0 | 208.0 |
| Algorithm 2 *Sign* | Iterations | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 12 |
| | Depth | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 24 |
| | Complexity | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 24 |
| | Time (s) | 4.4 | 6.3 | 8.5 | 11.7 | 16.0 | 20.2 | 26.9 | 26.9 |
| Goldschmidt $1/x$ [10] | Iterations | 9 | 10 | 11 | 12 | 13 | 14 | 16 | 16 |
| | Depth | 9 | 10 | 11 | 12 | 13 | 14 | 16 | 16 |
| | Complexity | 9 | 10 | 11 | 12 | 13 | 14 | 16 | 16 |
| | Time (s) | 4.2 | 5.1 | 6.8 | 8.4 | 10.3 | 12.7 | 18.2 | 18.2 |
| Algorithm 3 *Inverse* | Iterations | 5 | 6 | 6 | 7 | 8 | 8 | 9 | 9 |
| | Depth | 5 | 6 | 6 | 7 | 8 | 8 | 9 | 9 |
| | Complexity | 5 | 6 | 6 | 7 | 8 | 8 | 9 | 9 |
| | Time (s) | 1.2 | 1.8 | 1.8 | 2.5 | 3.3 | 3.3 | 4.4 | 4.4 |
| Goldschmidt $\sqrt{x}$ | Iterations | 7 | 8 | 9 | 11 | 12 | 13 | 14 | 15 |
| | Depth | 14 | 16 | 18 | 22 | 24 | 26 | 28 | 30 |
| | Complexity | 14 | 16 | 18 | 22 | 24 | 26 | 28 | 30 |
| | Time (s) | 10.4 | 14.7 | 20.0 | 35.2 | 50.9 | 66.0 | 72.0 | 79.0 |
| Algorithm 4 *(Inverse)SquareRoot* | Iterations | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 |
| | Depth | 8 | 10 | 10 | 12 | 12 | 14 | 14 | 16 |
| | Complexity | 8 | 10 | 10 | 12 | 12 | 14 | 14 | 16 |
| | Time (s) | 2.7 | 4.4 | 4.4 | 6.7 | 6.7 | 10.4 | 10.4 | 14.9 |
| Goldschmidt $1/\sqrt{x}$ | Iterations | 7 | 8 | 9 | 11 | 12 | 13 | 14 | 15 |
| | Depth | 14 | 16 | 18 | 22 | 24 | 26 | 28 | 30 |
| | Complexity | 14 | 16 | 18 | 22 | 24 | 26 | 28 | 30 |
| | Time (s) | 9.9 | 12.9 | 19.1 | 32.2 | 45.7 | 67.0 | 80.0 | 94.0 |
| Algorithm 4 *(Inverse)SquareRoot* | Iterations | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 |
| | Depth | 8 | 10 | 10 | 12 | 12 | 14 | 14 | 16 |
| | Complexity | 7 | 9 | 9 | 11 | 11 | 13 | 13 | 15 |
| | Time (s) | 2.4 | 4.1 | 4.1 | 6.6 | 6.6 | 9.8 | 9.8 | 12.8 |
| Newton-Raphson $1/\sqrt{x}$ | Iterations | 6 | 7 | 8 | 9 | 11 | 12 | 13 | 14 |
| | Depth | 12 | 14 | 16 | 18 | 22 | 24 | 26 | 28 |
| | Complexity | 18 | 21 | 24 | 27 | 33 | 36 | 39 | 42 |
| | Time (s) | 6.2 | 9.0 | 12.1 | 16.1 | 24.9 | 33.4 | 43.5 | 54.3 |
| Algorithm 5 *ReciprocalRoot* | Iterations | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 |
| | Depth | 6 | 8 | 8 | 10 | 10 | 12 | 12 | 14 |
| | Complexity | 9 | 12 | 12 | 15 | 15 | 18 | 18 | 21 |
| | Time (s) | 1.5 | 2.6 | 2.6 | 4.1 | 4.1 | 6.2 | 6.2 | 9.0 |