Efficient Parallel Binary Operations on Homomorphic Encrypted Real Numbers

Jim Basilakis[1] and Bahman Javadi[2]

[1] Western Sydney University, j.basilakis@westernsydney.edu.au
[2] Western Sydney University

Abstract. A number of homomorphic encryption application areas, such as privacy-preserving machine learning analysis in the cloud, could be better enabled if there existed a general solution for combining sufficiently expressive logical and numerical circuit primitives to form higher-level algorithms relevant to the application domain. Logical primitives are more efficient in a binary plaintext message space, whereas numeric primitives favour a word-based message space before encryption. In a step closer to an overall strategy of combining logical and numeric operation types, this paper examines accelerating binary operations on real numbers suitable for somewhat homomorphic encryption. A parallel solution based on SIMD can be used to efficiently perform addition, subtraction and comparison operations in a single step. The result maximises computational efficiency, memory space usage and minimises multiplicative circuit depth. Performance of these primitives and their application in min-max and sorting operations are demonstrated. In sorting real numbers, a speed up of 25-30 times is observed.

Keywords: homomorphic encryption, SWHE, FHE, comparison, sorting

1 Introduction

Homomorphic Encryption (HE) is seen as an important technology for enabling computation and analytical processing on encrypted data outsourced to third parties. This would greatly facilitate mainstream utilisation of public cloud resources for sensitive data analysis with a significantly reduced chance of information leakage should any system breach occur. Fully Homomorphic Encryption (FHE) schemes, which support addition and multiplication operations on the same ciphertext, have only recently come about starting from the work on ideal lattices by Craig Gentry in 2009 [1]. Although the original scheme was highly inefficient, newer generation FHE schemes have since rapidly evolved that are several orders more efficient though not yet practical enough to be considered for mainstream application use.

Somewhat Homomorphic Encryption (SWHE) schemes remove the requirement of *bootstrapping* or *recryption* which typically takes up a very significant amount of the total FHE execution time. This however places a limit on the number of combined operations that can be performed on the 'noisy' ciphertexts before successful decryption is no longer possible. Considerable research has been dedicated so far to developing various optimisations and batching techniques to improve the efficiency of SWHE schemes and to maximise the depth of the circuits that can be evaluated without bootstrapping. A number of newer generation schemes were developed based on the more efficient and simple Learning with Error (LWE) problem introduced by Regev in 2009 [2] (and its later ring variant) rather than on lattices, although the hardness problems were related [3]. A FHE scheme based on *key-switching* and *modulus-switching* optimisation techniques (see Section 3.2) became known as the BGV scheme named after its authors Brakerski, Gentry and Vaikuntanathan. Other SWHE scheme variants include Ring-LWE (RLWE) scale-invariant versions and NTRU-based schemes [4].

Despite the aforementioned optimisations, there are still significant (space and time) demands on computational resources and network bandwidth to overcome. The fact is that SWHE schemes are obliged to have either a massive ciphertext expansion or an 'ugly' algebraic structure in order to maintain their security [5]. Ciphertext expansion ratios are typically in the order of GBs in size for only a relatively small amount of plaintext input, resulting in high computational and communication costs [6]. Ciphertext packing techniques such as homomorphic Single Instruction Multiple Data (SIMD) proposed by Smart and Vercauteren [7], maximise memory space usage and achieve parallelism of repeated operations by enabling the combination of multiple smaller ciphertexts into a single larger ciphertext.

Currently available SWHE implementations include additional optimisations to improve efficiency of processing large ciphertexts. A prominent open-source RLWE implementation of the BGV scheme called HElib[1] by Halevi and Shoup [8] incorporates many such optimisations (see Section 3.3), and is used to efficiently implement algorithms described in this paper.

Much research effort has also been dedicated around circuit design to support various HE application areas. The main aim is to optimise use of the SWHE tools as much as possible by designing algorithms in a way that reduces both the consecutive and total number of multiplications. The former minimises circuit depth to avoid bootstrapping while the latter reduces overall computational cost. SIMD techniques can be used to reduce the occurrence of both multiplication instances, in addition to maximising memory space usage. SWHE tools are often applied in highly tailored ways against the particular application scenarios, in order to deal with narrow parameter and computational efficiency constraints. Tightly coupling HE methods in this way limits their generalisability. The extent to which a HE solution can support general computing functions across a wide scope of application areas depends on the availability of sufficiently expressive

[1] github.com/shaih/HElib

composable cryptographic primitives designed to work efficiently within a HE context.

Developing even the most basic primitives such as simple numeric and logical operations require non-trivial choices, including whether to perform evaluations over binary or arithmetic circuits (or polynomial rings), as well as the choice of data encoding. These factors have a significant bearing on low-level primitive circuit performance, which have a further flow-on effect on the performance of higher-level algorithms (*eg.*, min-max and sorting methods), more directly relevant to practical use cases [9].

Binary circuits, which rely on encoding data at the bit-level before encrypting (*ie.*, use a plaintext message space of $\mathbb{Z}_2$) naturally support boolean comparison operations, whereas they are very slow in performing arithmetic operations [10]. *Word-based* encoding (*ie.*, encrypting as elements of a plaintext modulus of p or a $\mathbb{Z}_p$ message space, for a large enough p) allows for simpler and more efficient ciphertext multiplication and addition that are easily parallelised; this is however at the cost of more complex comparison and thresholding operations which are fundamental to many of the HE application areas [10].

In the area of data encoding, ciphertexts encoded as real numbers represent the most practical solution for most application areas. Techniques to represent fixed point numbers (encoded as polynomials) include scaled integer and fractional representations [11]. Encoding floating point numbers require independent storage of the encrypted significand from the exponent, necessitating separate homomorphic operations on each during calculations [12, 13]. Options also exist for encoding negative numbers, using either a sign-magnitude or two's complement approach [9]. The approaches used to represent real numbers (fixed or floating point) in either type of plaintext message space ($\mathbb{Z}_2$ or $\mathbb{Z}_p$) can have a significant impact on the efficiency of numeric and/or logical operations [9].

1.1 Our Contribution

The research presented in this paper aims to minimise circuit depth, and to maximise memory space utilisation and HE computational performance on real numbers under a binary message space. The approach combines primitive logical functions together with binary addition and subtraction operations using SIMD techniques. The result is to permit natural comparison and thresholding operations using binary circuits while at the same time facilitating an efficient amortised rate for addition and subtraction operations performed in parallel. Such a technique would find use in particular implementation instances across all HE applications areas where combinations of logical and numeric operations are required. The technique would also complement other strategies for performing arithmetic operations, including multiplication, in larger ($p > 2$) message spaces. Finally, based on this approach, performance of the fundamental basic primitive operations and their application to various more complex SWHE algorithm implementations on real numbers are demonstrated using HElib.

2 Related Work

Related work on encoding, addition, equality and comparison HE operations will be discussed during development of our binary circuit primitives in relevant parts of Sections 3 and 4. The outcome of our circuit development is to enable more efficient complex homomorphic operations, which are derived from these basic circuits, using parallel batching techniques. Two candidate algorithms for demonstrating potential application of these primitives including sorting and finding the minimum or maximum (min-max) from a group of encrypted numbers.

The task of sorting encrypted numbers using FHE was first implemented in 2013 by Aguilar-Melchor *et al.* [14] using a Bubble sort, and by Chatterjee *et al.* [15] using a two-stage Bubble and Insertion sort. The hcrypt[2] library, based on the work of Smart and Verteceuran [16] was used in both cases, although the former also experimented with an early BGV-based scheme. Chatterjee examined minimising the recryption operation and as was able to achieve sorting of 40 (32-bit) integers in 1399 seconds. The hcrypt library however does not take advantage of modern noise reduction or SIMD batching techniques [17], which are factors both relevant to additional improvements in sorting performance.

The seminal work by Çetin *et al.* [18] in 2015 implemented two low-depth sorting algorithms both relying on a matrix to perform all comparisons at the outset, and achieving a multiplicative circuit depth of $\mathcal{O}(\log N + \log n)$, where N is the total number elements to be sorted, and n is the number of bits encoding each element. Çetin also introduced the concept of using SIMD to accelerate sorting performance, although it is used in this case to sort multiple lists rather than accelerate sorting of a single list. The amortised running time per sort is reduced since it takes the same amount of time to sort one list compared to multiple groups of similar-sized lists up to the total number of available ciphertext slots, s. Sorting one list however inefficiently uses one ciphertext per bit, therefore the implementation is computationally expensive. Using an NTRU-based SWHE scheme, sorting 16 (32-bit) integers is inferred to take about 45 mins. Dai & Sunar [19] boosted the SWHE scheme's sorting performance using a GPU-accelerated library[3] to achieve a speed-up of 12-41 times (reportedly, 1 min 38 sec to sort 16 integers of size 32 bits).

Kim *et al.* [20], appears to be the first work to report using SIMD batching to accelerate sorting performance within a single list. The authors employ HElib to implement a bitonic sorting algorithm, which is a data independent (*ie.* constant number of comparisons irrespective of input) sorting algorithm, consisting of recursive sort and merge operations. Only half of the available data slots are used to insert elements within its ciphertext packing structure. The sorting network is made up of two bitonic (*ie.* increasing then decreasing) sub-sequences, each initially with two elements for the base case. The bitonic sort algorithm has a $\mathcal{O}(N \log^2 N)$ comparison complexity and a multiplicative depth of $\mathcal{O}(\log n \cdot$

[2] github.com/hcrypt-project/libScarab
[3] github.com/vernamlab/cuHE

$\log^2 N$). Additional details can be found in [20, 21]. Whilst benchmarking Çetin's work, [20] argues that the multiplicative circuit width in addition to its depth are important in determining the number of recryptions required for sorting large lists, which is the most dominant bottleneck in all FHE operations. The sorting algorithm based on the comparison matrix (having $\mathcal{O}(N^2)$ comparators) was shown to have a large width, and ultimately requiring more recryptions compared to the bitonic sort, despite the latter having a larger multiplicative circuit depth. An improvement in the number of recryptions was confirmed using real implementations of the sorting algorithms, although performance statistics on smaller sorting sets was not shown.

Literature specifically looking at finding min-max on encrypted number sets using FHE is limited. Kocabaş *et al.* [22] uses the HElib library for min-max heart rate computation involving a logical comparison circuit examined later in Section 5.3. A $\log_2 N$ stage binary tree is used to repeatedly applying min-max on N ciphertexts packed with a vector of n-bit integers, resulting in an overall circuit depth of $(\log_2 n + 2) \cdot \lceil \log_2 N \rceil$. It would appear further processing is necessary (additional circuit depth of $\log_2 n$) to extract the final minimum or maximum value within the output ciphertext in a step not apparently described; perhaps being recovered after decryption. The min-max algorithm described by Togan *et al.* [23] uses a recursive strategy to combine, in the base case, a one bit integer 'greater than' comparison relation with a selection operator both expressed in polynomial form. The overall circuit depth is similar to the approach by Kocabaş, although no ciphertext packing is involved. A HElib-based implementation finds the maximum value from 16, 8-bit integers in 21 minutes. Set at a 140 bit security level, the implementation is very memory intensive at 3.8 GB.

It is noteworthy that all literature reviewed only examines sorting and finding min-max from sets of encrypted integers rather than real numbers, despite both number types requiring an identical amount of processing time to sort when correctly encoded, for a given bit size.

3 Preliminaries

3.1 Homomorphic Encryption

Any HE scheme consists of four probabilistic polynomial time algorithms `Keygen`, `Enc`, `Dec`, `Eval`. In this case, all algorithms implicitly take in a number of parameters as input including dimension n, modulus q and error distributions, according to RLWE constructions:

- *Key generation* ($pk, sk \leftarrow$ `Keygen`(1^λ)): takes in security parameter λ as input and outputs a public encryption key pk and a secret decryption key sk.
- *Encryption* ($c \leftarrow$ `Enc`(pk, μ)): takes pk and a plaintext μ in a message space $\mathcal{M}$ defined by some ring $R_\mathcal{M}$. The output is a ciphertext c in space $\mathcal{C}$ (and ring $R_\mathcal{C}$).
- *Decryption* ($\mu \leftarrow$ `Dec`(sk, c)): decrypts c back to μ using the secret key sk.

– *Evaluation* ($c_f \leftarrow \texttt{Eval}(pk, f, c)$): characterises a HE scheme, taking as input the public key pk, a function $f : R_{\mathcal{M}}^l \rightarrow R_{\mathcal{M}}$ and a tuple of l ciphertexts $c_1, ..., c_l$, to output a ciphertext c_f.

In this work, f will be represented by an arithmetic circuit over `GF(2)` that can be evaluated by a SWHE scheme to a limited depth so that `Dec` can recover μ correctly without requiring bootstrapping.

3.2 The BGV SWHE Scheme

The RLWE variant of the BGV scheme can be represented as follows [3, 8]: With λ as the security parameter (*ie.*, all known valid attacks taking $\Omega(2^\lambda)$ bit operations) and an integer $m = \Omega(\lambda)$ that defines the m-th cyclotomic polynomial $\Phi_m(X)$, we define the polynomial ring $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$. m, which roughly corresponds to the dimension of the underlying lattice, determines the size of computation and hence represents efficiency of the encryption scheme. The plaintext message space is set to $\mathbb{A}_p := \mathbb{A}/p\mathbb{A} = \mathbb{Z}[X]/(\Phi_m(X), p)$ for $p \geqslant 2$ (although in this work we will always use $\mathbb{A}_2$). The ciphertext space is set to $\mathbb{A}_q := \mathbb{A}/q\mathbb{A}$ for an odd integer modulus q with a degree up to $\phi(m) - 1$[4]. Ciphertexts are 2-element vectors defined over $\mathbb{A}_{q_i}$ (or ring R_{q_i}) at any particular level i, *ie.*, $\boldsymbol{c} = (c_0, c_1) \in R_{q_i}^2$. In the *modulus-switching* procedure, by switching to a ciphertext with a smaller modulus (from q_i to q_{i-1}), an increase in the number of multiplications could occur from $\log L$ to L levels before bootstrapping is required and without involving a secret key [3, 24]. At each ciphertext level $(L, ...1, 0)$, the modulus decreases with every application of modulus-switching to reduce the noise during homomorphic evaluation. No further noise reduction is possible in level 0, requiring bootstrapping to enable further computation.

As there are a few variations to the RLWE assumptions involved in encryption, we use that proposed by Lyubashevsky, Peikert, Regev (LPR) as a practical approach [25]. In set-up of the scheme, a (discrete Gaussian) noise distribution χ is defined over the set of all ring elements $\mathbb{A}_q$ with a small norm bounded by some B. Sample $s \leftarrow \chi$ and $e \leftarrow \chi$. Generate $a_1 \leftarrow R_{q_L}$ uniformly at random and calculate $a_0 = [-(a_1 s + pe)]_{q_L}$. Set the secret key $sk = s$ and the public key $pk = (a_0, a_1)$.

To encrypt a plaintext polynomial $m \in \mathbb{A}_p$, sample $u, g, h \leftarrow \chi$ and compute the ciphertext which is initially in level L:

$$\boldsymbol{c} = (c_0, c_1) = (a_0 u + pg + m, a_1 u + ph) \bmod q_L \tag{1}$$

At any level, i the equality over $\mathbb{A}$ holds, $m = [[c_0 + sc_1]_{q_i}]_p$[5], which is the decryption formula. The term $[c_0 + sc_1]_{q_i}$ is the 'noise' in the ciphertext that grows with each homomorphic evaluation. The ciphertext is valid as long as this noise does not wrap around modulo q_i, which gives bound on the norm of the noise to be always below B. Under the RLWE assumption, the encryption scheme

[4] $\phi(m)$ is the Euler's totient function

[5] $[\cdot]_q$ is the reduction-mod-q function

is semantically secure (*ie.*, against honest-but-curious adversary behaviour) if the public key $(a_0 = -(a_1 s + pe), a_1) \in R_q^2$, and by extension the ciphertext (c_0, c_1), are computationally indistinguishable from uniform in R_q^2.

Homomorphic addition involves simply adding two ciphertext vectors under the same key s over $\mathbb{A}_{q_i}$. The noise of the added vector is at most $2B$. Homomorphic multiplication involves a tensor product over $\mathbb{A}_{q_i}$ resulting in squaring of both ciphertext and (new self-tensored) secret key dimensions. While the current modulus remains unchanged, the noise of the product ciphertext is bounded by B^2. A *key-switching* (or *relinearisation*) technique is required to reduce the dimension of tensored secret key and ciphertexts back to original dimensions. At each multiplication level, the tensored secret key would be encrypted under a different secret key as a hint to facilitate conversion to the new valid encryption. A single key however is only required for all levels if it can be assumed safe to encrypt the secret key under its own public key (*circular security* assumption) [3].

SIMD makes use of the fact that smaller plaintext spaces could collectively be considered a vector of independent plaintext slots (for performing component-wise addition or multiplication) when encrypted into much larger ciphertext spaces by virtue of the Chinese Remainder Theorem (CRT). Since the plaintext space is over $\mathbb{A}_p$, $\Phi_m(X)$ factors modulo p into s irreducible factors, $F_i(X)$ (*ie.*, $\Phi_m(X) = \prod_{i=1}^{s} F_i(X)$). Each factor corresponds to a 'plaintext slot' of degree d, which is the smallest integer such that $p^d = 1 \bmod m$, and thus the number of slots, $s = \phi(m)/d$. The plaintext polynomial $a \in \mathbb{A}_p$ therefore can be viewed as a vector of s different small polynomials, $(a \bmod F_i)_{i=1}^{s}$. The slot values can represent elements of the extension field $\mathbb{F}_{p^d}$ (or embedded elements of the subfield $\mathbb{F}_{p^n}$ where n divides d), rather than individual bits [26].

Implementing linear rotations and shifts to move data across plaintext slots can be achieved with an automorphism operation without any increase in ciphertext noise [8]. The automorphism is represented by the transform, $a \mapsto a^{(i)}$ where $a^{(i)}(X) = a(X^i) \bmod \Phi_m(X)$ for $a(X) \in \mathbb{A}$ and some $i \in \mathbb{Z}_m^*$. Often a combination of automorphism operations with selective slots masked out is necessary to achieve arbitrary permutations on packed ciphertexts. Key-switching is also required to transform a ciphertext involved in an automorphism into another valid ciphertext that is decryptable with respect to the original secret key. In this case, secret key and ciphertext dimensions remain unchanged. Note, there is a computational cost involved in combining automorphisms followed by key switching to achieve permutations on plaintext slots. For further details, refer to [26].

3.3 HElib

HElib includes many optimisations over the basic BGV scheme, including support for recryption and SIMD. Due to a specialised key-switching procedure, each q_i is a product of smaller set of primes p_j, (*ie.*, $q_i = \prod_{j=0}^{i} p_j$). As a result, ciphertext elements of the ring $\mathbb{A}_{q^i}$ are represented with respect to both polynomial vectors modulo each small prime and the primitive m'th roots of unity

evaluations generated using the Fast Fourier Transform (FFT). This so-called *DoubleCRT* format allows point-wise addition and multiplication of elements in $\mathbb{A}_{q^i}$ in linear time [27, 8]. The encryption method involved in HElib varies slightly compared to the aforementioned LPR method: It uses a native plaintext space of R_{p^r} for a prime power p^r. The secret key, $\boldsymbol{s}$ is the 2-vector $(1, s) \in R^2$, where s is chosen randomly from $\{0, \pm 1\}^{\phi(m)}$ with a recommended Hamming-weight (*ie.*, number of non-zero coefficients) of 64. If Q_{ct} is the product of all ciphertext primes, initially generate $c_1 \leftarrow \mathbb{A}_{Q_{ct}}$ uniformly at random and $e \leftarrow \chi$ where $\chi \in \mathbb{A}_{Q_{ct}}$, and set $c_0 = [p^r e - c_1 s]$. Then at any level i, the ciphertext $\boldsymbol{c} = (c_0, c_1)$ satisfies $[\langle \boldsymbol{s}, \boldsymbol{c} \rangle]_{q_i} = [c_0 + c_1 s]_{q_i} = m + p^r e$ in R where $m \in R_{p^r}$ is the message plaintext and $p^r e$ is the error term, which has a small norm relative to q_i [8].

4 Encoding Real Numbers

As mentioned, ease of implementation and efficiency of operations on binary circuits relies on the specific encoding method used to represent real numbers. A fixed point representation is the simplest one for demonstrating circuit operations, which can be applied to the polynomial plaintext ring, R_p and whose coefficients represent *bits* in a binary sequence. Two methods proposed by Dowling *et al.* [11] use scaled-to-integer and fractional encoding representations of fixed point real numbers, which are demonstrated by Costache *et al.* [28] to be isomorphic under the same power of two cyclotomic ring. Strictly speaking, fractional encoding cannot operate under a binary message space whereas the scaled-to-integer approach can perform computations over both binary or arithmetic circuits. It is nevertheless worthwhile considering both fixed point representations as researchers have previously sought to avoid costly bit operations by turning to the plaintext space, R_p [28].

The scaled-to-integer approach scales a binary representation of a real number by some power of 10 into an integer according to some fixed digit precision. Despite literature criticism about the requirement for complex book-keeping in order to ensure ciphertexts are correctly scaled in homomorphic operations [11, 28], we use the approach suggested by Jäschke *et al.* [9] of multiplying a real number by a power of two according to k bits of precision. In this case, the output of each homomorphic multiplication between two real numbers should be truncated by deleting the last k bits (equivalent to dividing by 2^k), to bring the product back to the required precision. Under this approach, there is no requirement to keep track of scaled products involved in computations before final conversion to the decrypted output, which could otherwise potentially leak data or computation information to an adversary.

The fractional encoding scheme is implemented over the cyclotomic ring $R = \mathbb{Z}[X]/(X^d + 1)$ where d is a power of two. The integer portion to the left of a decimal point can be represented in binary as usual while the fractional part of a real number to the right of a decimal point are represented by the highest polynomial coefficients. The latter are designated $-X^{d-i} b_{-i}$, where i is the bit

position to the right of the decimal point. Intuitively, due to the modulus of R, $X^d = -1$ and so the highest coefficients can also be represented as $X^{-i}b_{-i}$, which is as desired. Under this scheme, a number of coefficients are set aside for the fractional part of the encoded polynomial, $p \in R$. Negative numbers are represented by swapping coefficient signs. Under p, addition and multiplication work as expected and $p(2)$ decodes to the original real number [11, 28].

The problem with a fractional encoding scheme is that it does not appear to be compatible with SIMD operations, since the plaintext slots depend on irreducible modulus factors, $F_i(X)$, which is not the case with the modulus of R. The fractional encoding scheme can handle wrap-around modulo $X^d + 1$ with operations involving p, although with too many multiplications, coefficients reserved for the fractional and integral parts quickly grow towards each other to yield unexpected results when the two parts coalesce. In a power of two scaled-to-integer encoding, wrapping around slot values modulo $F_i(X)$ will yield unexpected results, placing a lower bound on the ring degree of polynomials involved in operations. In practice, this usually means ensuring that the degree of two polynomial inputs involved in multiplication are each less than half of the degree, d of the plaintext slots in order to prevent wrap-around. Due to truncation of the last k bits to correct for precision in multiplication, this has the effect of lowering the degree of the polynomial product, hence multiplication can continue indefinitely in this way, albeit within ciphertext noise limits.

We choose a slight refinement to Jäschke's approach when considering the scaled-to-integer representation of real numbers which, despite having identical encoding to the original method, is conceptuality simpler when applied computationally and for descriptive purposes. After scaling a real number by 2^k and encoding it in binary, dedicate k of the lowest polynomial coefficients to encode the fractional part of a real number with binary digits $b_{-1}b_{-2}...b_{-k}$, followed by the integral portion encoded with binary digits $b_{I^+}...b_1b_0$, and represented as: $\sum_{I^+ \geqslant i \geqslant -k} X^i b_i$. Although binary digits are shown, any base ($b > 2$) can be chosen to represent a real number using this scheme (whether balanced or not), similar to fractional encoding.

For negative numbers, a *two's complement* rather than a *sign-magnitude* encoding is chosen since the former is more efficient in comparison and addition operations [9], which have relevance to the circuit primitives discussed in the following section. A two's complement encoding of a negative real number is computed by inverting the scaled binary encoding of the positive real number and adding one. In this case, the 'highest' polynomial coefficient is reserved for encoding the negative value -2^{d-k-1}, and *sign extension* is used to extend the bit length of the encoding to d. Using an example, the real value -1.25 within a plaintext slot where $d = 8$ and 4 bits of precision are required, can be two's complement encoded as a ring polynomial under extension field $\mathbb{F}_{2^8}$ whose coefficients represent the bit sequence 11101100. The least four significant bits represent the fractional part while the Most Significant Bit (MSB) indicates the encoded value is negative (with two adjacent sign-extension bits). The final

result can be decoded as: $p(2) = -2^3 + 2^2 + 2^1 + 2^{-1} + 2^{-2} = -1.25$. Under this scheme, addition and multiplication also work as expected.

5 Circuit Primitives

Ultimately, primitive binary circuits for addition and equality are alone necessary to combine a range of logical operations together with addition and subtraction on real numbers under a binary plaintext space. Finding efficient versions of these primitives has a flow-on effect to more complex homomorphic operations that are derived from these two basic circuits. At a bit level, XOR ($\oplus$) and AND ($\cdot$) logic gates are implemented slot-wise over a binary plaintext space via corresponding homomorphic addition ($+_h$) and multiplication ($\times_h$) operations between ciphertexts respectively. All other logic functions can be derived from these two logic gates. This includes an OR operation which is implemented as the binary inverse (by adding 1 in binary) of an AND (*ie.*, NAND) operation. When combining a number of AND operations, ciphertexts are commonly arranged into a binary tree formation so that terms are multiplied from base to root rather than sequentially [29, 30, 6]. The effect is to reduce the multiplication depth of the circuit from $n-1$ to $\log n$, and can be readily implemented using SIMD operations.

The following sections describe various primitive circuits that can perform binary addition and equality operations, with the aim of finding optimal parallel designs to minimise the three resource parameters: memory usage, multiplicative depth and total number of multiplications. Circuits will be described generally with focus being given to the final most competitive versions. All circuits assume a ciphertext structure that takes up one slot per plaintext bit to work correctly including SIMD operations.

5.1 Addition Circuit

The most basic adder implementation is the n-bit Ripple Carry Adder (RCA) which employs n full adders, with the carry bit output from one adder acting as input into the next adder along a chain from least to most significant bits. This 'rippling' dependency of carry bits means that the circuit is unsuitable for parallel operations. The logic for a sum of two bits, a_i, b_i and carry-in bit, c_i of the full adder is expressed as, $s_i = a_i \oplus b_i \oplus c_i$. Logic for the carry-out bit, $c_{i+1} = a_i \cdot b_i \oplus c_i \cdot (a_i \oplus b_i)$ can be reduced to one AND gate per bit, $c_{i+1} = (a_i \oplus c_i) \cdot (b_i \oplus c_i) \oplus c_i$ [29]. The multiplicative depth of this adder hence is limited to $n-1$. There are bit-wise and packed ciphertext RCA protocol implementations available [31]. Despite the latter potentially making use of SIMD operations, there is no real benefit conferred with respect to resource parameters used and circuit depth (refer to Table 1).

A Carry Lookahead Adder (CLA) aims to avoid the 'rippling' effect by allowing the incoming carries to be computed in parallel. The original carry-out logic can be reformulated as, $c_{i+1} = g_i \oplus p_i \cdot c_i$ where $g_i = a_i \cdot b_i$ is called the

generate term, and $p_i = a_i \oplus b_i$, is called the *propagate* term. According to the carry-out logic, if g_i is one (a_i and b_i are both one), c_{i+1} is one; and if p_i is one (at least one of a_i or b_i is one), then c_i is propagated to c_{i+1}. Since neither g_i nor p_i depend on the carry, they can be determined beforehand and in parallel. A pattern emerges whereby for each carry bit:

$$c_{i+1} = g_i \oplus (p_i \cdot g_{i-1}) \oplus (p_i \cdot p_{i-1} \cdot g_{i-2}) \oplus \\ ... \oplus (p_i \cdot ... \cdot p_2 \cdot p_1 \cdot g_0) \oplus (p_i \cdot ... \cdot p_1 \cdot p_0 \cdot c_{in}) \quad (2)$$

The final sum is computed as $s_i = p_i \oplus c_i$. A CLA has a lower multiplicative depth but a higher computational cost compared to the RCA. By sub-dividing n-bits into smaller equal m-bit sized blocks, this cost can be reduced. The smaller m-bit adders can be arranged into a CLA between blocks while carries within a block are rippled (or vice-versa) to realise Equation 2. In this case, the circuit depth is $\Theta(m + \frac{n}{m})$ and minimised when $m = \Theta(\sqrt{n})$ [31]. Summary statistics for a CLA with an m-bit block ripple adder design [31] are presented in Table 1.

CLAs form the basis of another group of adders called Parallel Prefix Adders (PPAs), which perform parallel group carry operations. These adders are known to be amongst the most efficient of all digital binary addition circuits [32]. The Kogge-Stone Adder (KSA) is one of the fastest adders of this type where every propagate and generate bit is computed in parallel. The circuit has a minimum possible multiplication depth and scales logarithmically, although at the expense of long wiring, more gates and a larger area requirement in digital hardware implementations [32]. These disadvantages tend to be much less significant in computing systems. In fact, the KSA appears particularly amenable to homomorphic SIMD operations as the protocol entails applying identical instructions uniformly across all plaintext slots, staged between shifting slots relative to two interacting ciphertexts. An 8-bit KSA circuit is illustrated in Fig. 1.

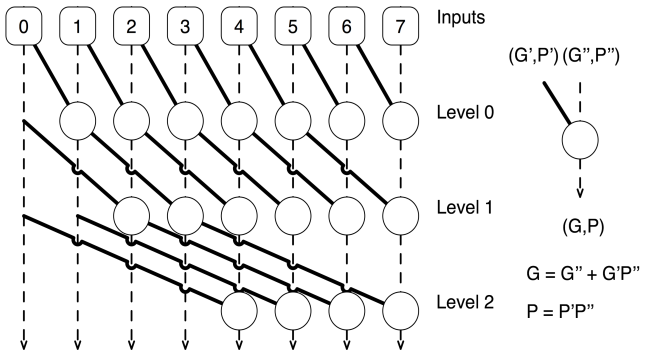

Fig. 1. The Kogge-Stone Adder Circuit [33].

As shown, each circuit node combines p and g recursively from two 'bit' columns to form a new set, $(g = g'' \oplus p'' \cdot g', p = p'' \cdot p')$. Starting at level 0, column i is combined with column $i + 2^k$ up to level $k = \lceil (\log_2 n) - 1 \rceil$ to eventually realise Equation 2. There are bit-wise and packed ciphertext protocol versions of the KSA [31]. In the latter case, all bits from a set of slots encoded within a base ciphertext can be combined in one step with another ciphertext whose set of slots are shifted in each level by 2^k towards the MSBs. To compute the final sum in parallel, a single multiplication depth is required initially to compute all g's, and subsequently between all k level carry operations, resulting in an overall multiplication depth of $\lceil \log_2 n \rceil$ and a total cost of $2 \cdot \lceil \log_2 n \rceil$ multiplications. The KSA appears to be the most efficient binary addition circuit that minimises all three main resource parameters (Table 1). These findings are reasonably comparable with those by King [31] and Kocabaş *et al.* [30] who employ the KSA. This adder appears to outperform various alternative techniques used for binary addition outlined in [9, 29, 34, 6].

Binary addition can also be used to perform subtraction using the trick of adding the first number with the two's complement negative encoding of the second number. This calculation can be incorporated into the adder algorithm by inverting the second encoded real number in addition to setting the first carry-in bit (c_{in} in Equation 2), to 1. The circuit requires only one further addition operation without any increase in its multiplicative depth.

Table 1. Comparison of addition and equality binary circuit parameter costs. n - bit number input; m - bits per block. Algorithms (Alg.) shown: RCA - Ripple Carry Adder; CLA - Carry Lookahead Adder; KSA - Kogge-Stone Adder; Eq - Equality; and Comp - Comparison Logic; includes bit-wise (b) and packed (p) ciphertext versions. Memory (Mem.) is the minimum number of ciphertexts required to fulfil the protocol. $\lg n$ represents $\lceil \log_2 n \rceil$.

Alg.	Add	Shift	Mult.	Depth	Mem.
RCA(b)	$4n - 2$	0	$n - 1$	$n - 1$	$2n + 3$
RCA(p)	n	n	$n - 1$	$n - 1$	4
CLA	$m + \frac{n}{m}$	$\lg m + m + \frac{n}{m} - 1$	$\lg m + m + \frac{n}{m} - 1$	$m + \frac{n}{m} - 2$	4
KSA(b)	$n[\lg n + 1] + 1$	0	$n[2 \lg n - 1] + 3$	$\lg n$	$3n$
KSA(p)	$\lg n + 2$	$2 \lg n$	$2 \lg n$	$\lg n$	4
Eq(p)	2	$\lg n$	$\lg n$	$\lg n$	2
Comp(p)	$2 \lg n + 3$	$2 \lg n + 1$	$\lg n + 2$	$\lg n + 1$	4

5.2 Equality Circuit

A common binary equality circuit implementation with optimisation for packed ciphertexts (albeit described for integers) is based on [34, 35, 18]. The equality test also holds for a two's complement scaled-to-integer encoding of a real number

and is depicted as: $\texttt{equal}(\boldsymbol{a}, \boldsymbol{b}) = \prod_{i=0}^{n-1}(a_i \oplus b_i \oplus 1)$. The output is 1 if $\boldsymbol{a}$ and $\boldsymbol{b}$ are equal, otherwise it is 0. In this case, $n-1$ multiplications are required which can be reduced to a multiplicative depth of $\lceil \log_2 n \rceil$ using a binary tree structure. The application of SIMD involves accumulating products of a packed ciphertext containing initial bit-wise values of $a_i \oplus b_i \oplus 1$. The slots are progressively shifted by 2^k in each level towards the MSBs and multiplied in parallel with the previous level's result. The MSB will contain the final output for the `equal` circuit after $\lceil \log_2 n \rceil$ levels [34]. An alternative divide-and-conquer strategy for determining integer equality with a similar multiplicative depth is described in [23], although $n-1$ homomorphic multiplications are required instead of $\log_2 n$.

5.3 Comparison Circuit

A comparison circuit compares the magnitude between two real numbers, indicating one number is larger or smaller compared to the other. Only one type of comparison circuit is required in combination with the `equal` circuit, to derive all other comparison relationships (*ie.*, $<, >, \leqslant, \geqslant$; see below). To enable a *less-than* comparison, there are two binary circuits of note [22, 18]. One determines the comparison logically while the other is derived from the addition circuit.

The comparison logic can be depicted in the following way: $\texttt{comp}(\boldsymbol{a}, \boldsymbol{b}) = \sum_{i=0}^{n-1}[(a_i < b_i)\prod_{i<j<n}(a_j = b_j)]$, where $(a_i < b_i) = (b_i \cdot (a_i \oplus 1))$ and $(a_j = b_j) = (a_j \oplus b_j \oplus 1)$. An implementation of the circuit using SIMD involves progressively shifting slots containing the term $a_j = b_j$ by 2^k in each level towards the Least Significant Bits (LSBs) and multiplied with the previous level's results. The final slot output is multiplied by the expression $a_i < b_i$ resulting in a multiplicative depth of $\lceil \log_2 n \rceil + 1$ and a total of $\lceil \log_2 n \rceil + 2$ multiplications (extra AND gate in the '$<$' term) [22]. Calculating a final sum of products using a binary tree structure, requires an additional $\lceil \log_2 n \rceil$ shifts and homomorphic additions to reveal the `comp` output. Comparison logic furthermore needs to address signed numbers in two's complement encoding. In particular, the inverse of the `comp` output should be returned when the signs differ between the two real numbers being compared. The final output therefore should be modified to, $\texttt{comp}(\boldsymbol{a}, \boldsymbol{b}) \oplus a_{n-1} \oplus b_{n-1}$ (not accounted for in Table 1).

Comparisons based on the binary addition circuit use subtraction to compare two real numbers followed by testing of the sign bit of the result [30]. In a *less-than* comparison, the following relationship holds: $\texttt{comp}(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} - \boldsymbol{b})_{n-1}$. Subtraction can be readily implemented using the KSA described earlier.

At first glance, it might appear that the logical comparison circuit, despite having a greater multiplicative depth by 1, is overall more efficient due to its lower cost of $\lceil \log_2 n \rceil + 2$ multiplications compared to $2 \cdot \lceil \log_2 n \rceil$ for the KSA comparator. However, the logical comparison algorithm has greater complexity around slot set-up and selection mask usage to perform sign correction. Such additional costs are not factored into the final parameter estimations presented in Table 1. Ultimately, the multiplication costs become comparable between both circuits (when $n \leqslant 64$ bits), while all other cost parameters are exceeded for the logical comparator. Alternative comparison circuits reporting a much higher

total number of homomorphic multiplications, albeit of similar multiplicative depth, appear in [34, 23].

Once $\mathtt{c} = \mathtt{comp}(\boldsymbol{a}, \boldsymbol{b})$ and $\mathtt{e} = \mathtt{equal}(\boldsymbol{a}, \boldsymbol{b})$ have been determined, all remaining comparison relationships can be derived using binary logic and at little additional cost as follows: $\boldsymbol{a} \leqslant \boldsymbol{b} = \mathtt{c + e}$; $\boldsymbol{a} > \boldsymbol{b} = \overline{\mathtt{c + e}}$; and $\boldsymbol{a} \geqslant \boldsymbol{b} = \overline{\mathtt{c}}$.

5.4 Further Optimisation

From the group of primitive binary circuits reviewed in Table 1, the packed versions of the KSA (for '+', '−' and '<' operations) and Equality (`equal`) circuits appear the most competitive from an overall memory, computation cost and multiplicative depth standpoint. Despite overall efficiency of these chosen schemes, each ciphertext input still only encodes a single real number, leaving many (potentially hundreds of) unused slots. Fortunately with very little modification to the basic circuits, both primitives can be further accelerated for free. Based on this approach, multiple real number computations can occur in one step within a single ciphertext per input. The parallelism across multiple primitive circuits is in addition to that occurring within each primitive (as outlined in Section 5), and both rely on SIMD. Arranging both KSA and Equality circuits in parallel to process multiple real number inputs and accelerate HE performance has not been previously described in the literature reviewed. Combining both parallel primitive circuits enables multiple arithmetic and logic operations, which are built from one or both primitives, to be executed in one step. Analyses and computations that benefit from parallel execution of the candidate sub-operations $(+, -, =, <, >, \leqslant, \geqslant)$, would have a significantly reduced amortised running time.

The packing structure of the ciphertext inputs are similar for both accelerated primitives, as illustrated in Fig. 2. Packed ciphertexts A and B contain multiple operand pairs providing input to a corresponding array of candidate sub-operations to be executed in parallel. A ciphertext consisting of s slots can pack $\lfloor s/n \rfloor$ data slots, each encoding an n-bit real number. The last (s mod n) slots are disregarded. Function ParallelPacking (outlined in Alg. 1 and Fig. 2) pre-processes packed ciphertext operands to prepare subsequent input to functions ParallelAdderSubtractor (Alg. 2) and ParallelEquality (Alg. 3). Three outputs resulting from the pre-processing stage are P, G and C_T. All three ciphertexts are required for input to Alg. 2 whereas only C_T is required for Alg. 3. Importantly, Alg. 2 also requires that the MSB for each data slot to be zero in order to work correctly, leaving the remaining $(n-1)$ bits to encode signed or unsigned real numbers. Masking of the MSBs is shown in Fig. 2 (and represented in Alg. 1, lines 4, 15) as a constant multiplication by $maskArray[]$ applied to all packed propagate and generate data slots (*resp.* P and G). Subsequently, both ciphertexts are right shifted by one bit (line 16) to move the zeroed bit to the LSB of the adjacent right data slot (in G the LSB represents c_{in}). All sub-operations except addition require corresponding inputs to be prepared for KSA subtraction. Fig. 2 demonstrates this as a constant add by $xorArray[]$ (Alg. 1, lines 6, 13) to obtain the inverse of relevant sub-inputs of Ciphertext B, in addition to setting the c_{in} bit to 1 via a constant add by $c0Array[]$ (lines 7, 17) to

the relevant data slots of G. The final outputs of ParallelAdderSubtractor and ParallelEquality are contained in respective ciphertexts of C_T. Both C_T outputs are homomorphically combined in a trivial post-processing stage to derive the final encrypted evaluations of all parallel sub-operations. The circuit block of parallel primitives illustrated in Fig. 2 from now will be referred to as Parallel Primitive Circuits (PPCs).

Both ParallelAdderSubtractor and ParallelEquality can flexibly process any n-bit number groups, not just powers of 2. The latter case however makes most efficient use of available slot resources. As a further optimisation, if the maximum bit length of numbers in all data slots is $l < n$, then this knowledge can reduce the multiplicative depth and computational cost of both parallel functions (*eg.* depth of $\lceil \log_2 l \rceil$ instead of $\lceil \log_2 n \rceil$). This property is used to significantly reduce the parameter costs of an accumulator function discussed in Section 6.1.

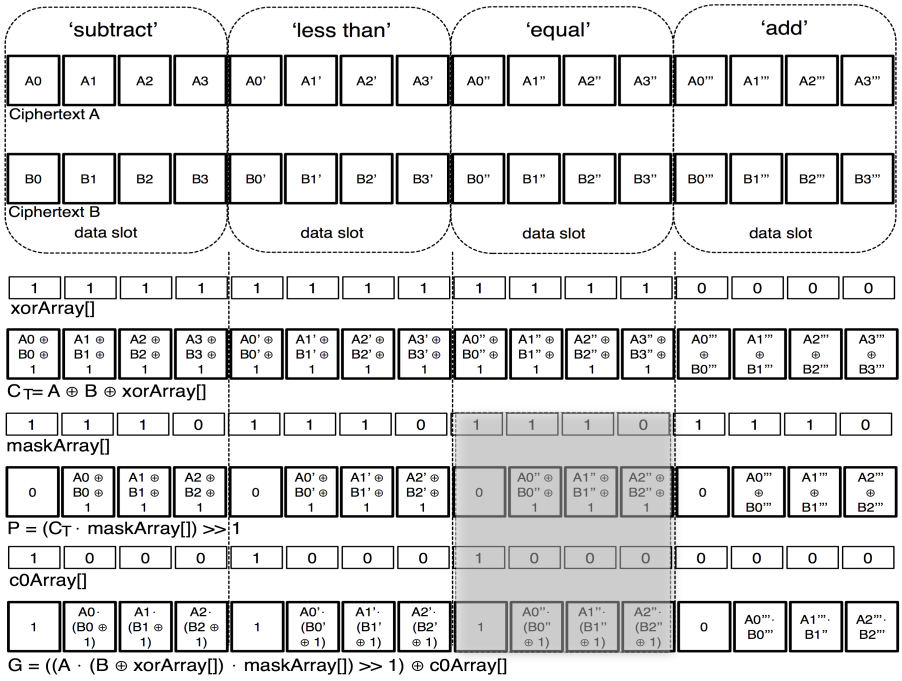

Fig. 2. Example ciphertext packing structure and preprocessing of inputs according to Alg. 1. Each ciphertext can fit 4 x 4-bit plaintext elements into 16 available slots. The greyed out area indicates variables not relevant to an 'equal' operation (as input to Alg. 3).

Algorithm 1 Preprocessing packed inputs to ParallelAdderSubtractor (Alg. 2) and ParallelEquality (Alg. 3)

INPUT: A, B, s, n and $Op[]$, where A and B are packed ciphertexts with s slots and each encoding an $(n-1)$-bit real number in each of its $\lfloor s/n \rfloor$ data slots. $Op[]$ is an unencrypted array of strings of size $\lfloor s/n \rfloor$ specifying the sub-operation types to occur in corresponding data slots between A and B (*eg.* 'add', 'subtr', 'eq', *etc.*).

OUTPUT: C_T, G and P: packed ciphertexts providing input to Algs. 2 (all 3 parameters) and 3 (only C_T required).

function ParallelPacking($A, B, n, Op[]$)
  $xorArray[s], maskArray[s], c0Array[s]$
  for $i \leftarrow 0$ **to** $\lfloor s/n \rfloor$ **do** ▷ for each data slot
    $maskArray$.insert($(n-1) \cdot [1] + [0]$) ▷ MSB = 0
    if $Op[i] \neq$ 'add' **then**
      $xorArray$.insert($n \cdot [1]$)
      $c0Array$.insert($[1] + (n-1) \cdot [0]$) ▷ LSB = 1
    else
      $xorArray$.insert($n \cdot [0]$)
      $c0Array$.insert($n \cdot [0]$)
    end if
  end for
  $B = B \oplus xorArray[]$
  $G \leftarrow A \cdot B$; $P \leftarrow A \oplus B$; $C_T \leftarrow P$
  $G = G \cdot maskArray[]$; $P = P \cdot maskArray[]$
  $G \gg 1$; $P \gg 1$
  $G = G \oplus c0Array[]$
  return C_T, G, P
end function

6 Implementation

The following section demonstrates acceleration of various sample privacy-preserving applications, by integrating the aforementioned parallel algorithms. The focus is on applications that are able to sort or find the min-max from a group of encrypted numbers. Depth optimised and unoptimised versions of both applications types will be illustrated. The main aim is to demonstrate broad utility, versatility and effectiveness of these parallel algorithms. Many other privacy-preserving applications and computations could potentially benefit from block execution of multiple binary operations, including other sorting algorithms. As a secondary aim, the high degree of acceleration achieved by the parallel algorithms yields very competitive performances on sorting and min-max compared to more specialised privacy-preserving applications of the same type found in the literature. Performance characteristics will be limited to multiplicative depth and parallel sorting times under a SWHE scheme, leaving analysis of number of recryptions and comparisons of larger sorting sets under a FHE scheme as future work.

Algorithm 2 Parallel Kogge-Stone Adder-Subtractor

INPUT: G, P, C_T and l, where G, P, C_T are packed ciphertexts with s slots and $\lfloor s/n \rfloor$ data slots, packed according to Algorithm 1. l is the maximum input bit length contained in all n-bit data slots ($l \leqslant n$).

OUTPUT: C_T : a ciphertext with component-wise addition or subtraction performed on every data slot pair.

1: **function** ParallelAdderSubtractor(G, P, C_T, l)
▷ G, P, C_T - ciphertext memory objects 1,2,3
2: **for** $i \leftarrow 0$ **to** $\lceil \log_2 l \rceil$ **do**
3: $tmp \leftarrow G$ ▷ ciphertext memory object 4
4: $tmp = P \cdot (tmp \gg 2^i)$
5: $G = G \oplus tmp$
6: **if** $i \neq \lceil \log_2 l \rceil - 1$ **then**
7: $tmp \leftarrow P$
8: $P = P \cdot (tmp \gg 2^i)$
9: **end if**
10: **end for**
11: **return** $C_T = C_T \oplus G$
12: **end function**

6.1 Sorting

The following demonstrators will look at accelerating sorting methods including the odd-even transposition sort and the Direct Sorting algorithm of Çetin's work [18]. Both are ideal candidates for the parallel primitive algorithms described in this paper.

Odd-Even Transposition Sort This simple bubble sort variant alternately compares odd and even pairs in an array of numbers in parallel. The steps for sorting 4 numbers is illustrated in Fig. 3(a). The algorithm is guaranteed to terminate after N parallel comparison steps. Normally the $N-1$ comparisons in each iteration are performed in parallel so the overall complexity is $\mathcal{O}(N)$. The algorithm is readily implemented using a block of PPCs with all sub-operations set to 'less than' for each comparison iteration. Numbers are compared to their neighbours by shifting one of two identical ciphertexts packed with (n-bit) real numbers, by n bits relative to the other ciphertext. Subsequently, the parallel comparison operators are applied using SIMD. Overall this requires N applications of the parallel KSA subtractor algorithm (circuit depth of $\lceil \log_2 n \rceil$) to sort an array of N numbers. Each iteration however would require preprocessing of inputs according to Algorithm 1, in addition to selection and swapping of alternate odd and even pairs within the packed ciphertext, which adds a total constant circuit depth of approximately 5 per iteration. Altogether a naive approach to sorting N x n-bit real numbers in one packed ciphertext has a minimum circuit depth of $[N \cdot (\lceil \log_2 n \rceil + 5)]$. Such an unoptimised sorting method would require recryption to sort a single packed ciphertext block due to the high circuit depth. Furthermore, sorting large number sets will further add to the

Algorithm 3 Parallel Equality

INPUT: C_T and l, where C_T is a packed ciphertext with s slots and $\lfloor s/n \rfloor$ data slots, packed according to Algorithm 1. l is the maximum input bit length contained in all n-bit data slots ($l \leqslant n$).

OUTPUT: C_T : a ciphertext with component-wise equality performed on every data slot pair. Results are located in the MSB of each data slot of C_T.

1: **function** ParallelEquality(C_T, l)
▷ C_T - ciphertext memory object 1
2: **for** $i \leftarrow 0$ **to** $\lceil \log_2 l \rceil$ **do**
3: **if** $i = \lceil \log_2 l \rceil - 1$ **and** $(l \bmod 2^i) \neq 0$ **then**
4: $shiftamt = l \bmod 2^i$
5: **else**
6: $shiftamt = 2^i$
7: **end if**
8: $tmp \leftarrow C_T$ ▷ ciphertext memory object 2
9: $C_T = C_T \cdot (tmp \gg shiftamt)$
10: **end for**
11: **return** C_T
12: **end function**

circuit depth, which involves computing parallel (SIMD) odd-even transposition sort spanning across multi-threaded packed ciphertext blocks. This particular sorting method will not be further considered in this paper.

Matrix Comparison Sort Çetin's work implemented two shallow circuit sorting algorithms, Direct and Greedy Sort, which both initially make use of a comparison matrix and have an overall circuit depth of $\mathcal{O}(\log_2 N + \log_2 n)$. Only the former sorting algorithm will be examined for PPC integration. The comparison matrix, $m_{ij}^{(\gamma)}$ for a given encrypted input vector, $E(X) = \langle X_0^{(\alpha)}, ..., X_{N-1}^{(\alpha)} \rangle$ is described as:

$$m_{ij}^{(\gamma)} = \begin{cases} m_{ij} = 1 \text{ if } X_i < X_j. \\ m_{ij} = 0 \text{ else } . \end{cases} \tag{3}$$

where $i, j < N$, $i < j$, and $m_{ii} = 0$ (the diagonal elements). The lower triangular part of the matrix, $j < i$ is computed as $m_{ji}^{(\gamma)} = (X_i \geqslant X_j) = m_{ij}^{(\gamma)} \oplus 1$. The Direct Sort algorithm relies on calculating the sum of all elements within each matrix column, which returns the index of each X_j in the sorted array. For further details see [18].

The parallel version replicates the comparison matrix by splaying out its rows across blocks of packed ciphertexts, which are readily processing by the PPCs. Two rows of ciphertexts per block contain the input combinations required to compute only the upper triangular part of the matrix, while the corresponding inputs for the lower triangular part are all 0. The comparison matrix's diagonal elements are removed, thereby reducing the number of comparisons required to

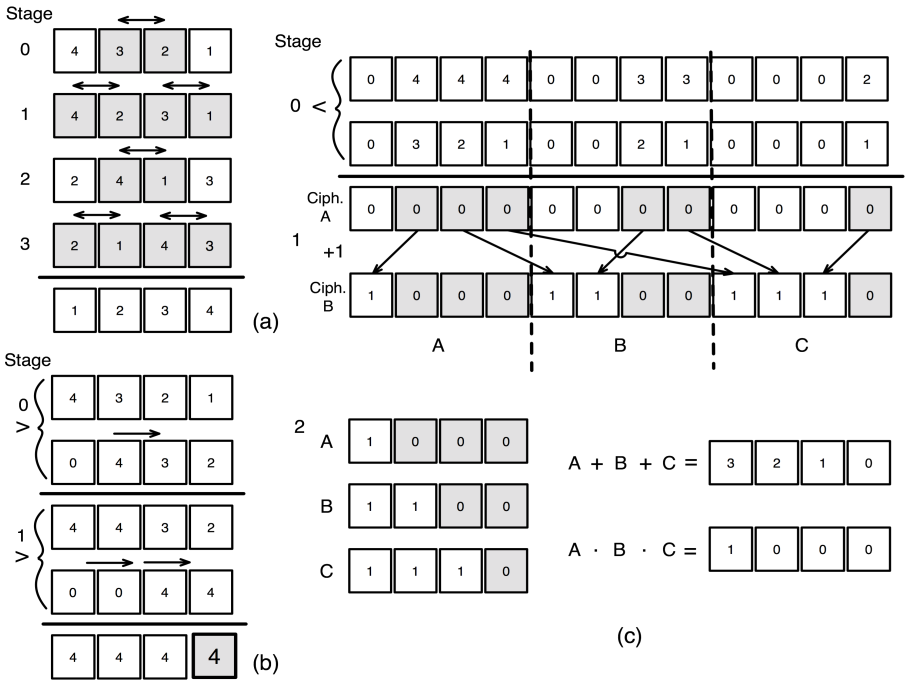

Fig. 3. Demonstrator algorithms: (a) Odd-Even Sort, (b) Min-Max, (c) Depth Optimised Min-Max and Sort. Squares represent data slots.

$N \cdot (N-1)$ instead of N^2. These steps are represented in Fig. 3(c). As shown, sorting an array of four numbers requires setting up all relevant input combinations across twelve n-bit data slots. In this case, all number combinations are contained in two input ciphertexts within one block. Parallel 'less than' sub-operations are performed on all input data slots by applying the PPCs to compute the upper triangular portion of the matrix. Remaining values are calculated by adding 1 (XOR) to the partial matrix and transforming the relevant data slots to complete the lower triangular portion of the matrix. Transposing half the comparison matrix in this way involves $N-1$ stages of slot shifts and insertions of XORed values at periodic intervals from the partial to completed matrix, represented by data slots within packed ciphertexts A and B respectively (as shown in Fig. 3(c)). The shift amount of ciphertext A and the periodic slot insertions to ciphertext B required at each stage are represented in Table 2. The steps in each stage are performed in parallel using SIMD and the algorithm can work to transform data slots across multiple ciphertext blocks. The algorithm has an overall circuit depth of one constant multiplication per stage due to the mask operation on ciphertext B.

Computing the index of every encrypted element in the sorted array requires adding the comparison matrix columns as shown in Fig. 3(c) - $(A+B+C)$. The

Table 2. Algorithm to transpose upper triangular portion of matrix encoded in Ciphertext A to form comparison matrix in Ciphertext B

Stage	**Ciphertext A**	**Mask to Ciphertext B**		
	Shift	**Start**	**Period**	**Times**
0	-1	0	$N+1$	$N-1$
1	$N-2$	N	$N+1$	$N-2$
2	$2N-3$	$2N$	$N+1$	$N-3$
$[\boldsymbol{i}]$	$[iN-i-1]$	$[iN]$	$[N+1]$	$[N-i-1]$
$\boldsymbol{N-2}$	$(N(N-3)+1)$	$N(N-2)$	$-$	1

corresponding columns are aligned with $N-2$ shifts of the data slots by $N \cdot n$ each shift. Elements in aligned columns require binary addition to compute the sorted indices. This *accumulator* operation (counting of elements) can be performed in parallel with an application of the PPCs, with all sub-operations set to 'add'. Since the highest index count resulting from the accumulator is expected to be $N-1$, the circuit depth of one application of the KSA is $\log_2(\lceil \log_2(N-1) \rceil)$ instead of $\log_2 n$ (note $l < n$ in Alg. 2). The accumulator function is applied $N-2$ times, therefore the overall circuit depth becomes $(N-2) \cdot \log_2(\lceil \log_2(N-1) \rceil)$. This circuit depth is relatively high. We consider instead using Çetin's approach to calculate the Hamming distance of the matrix columns by replacing the KSA with a Wallace Tree Adder (WTA) [36] to perform the accumulation. The latter adder itself is not computed in parallel using SIMD, but since it is applied n times - once for each matrix column, this part can be multi-threaded. The WTA has an overall minimal circuit depth of $\log_2(N/2)$.

Finally, the computed sorted index values need to be converted into an actual sorted array, which requires comparing each sorted index value, $\sigma_i^{(\gamma)}$ with an ascending list of all indices in the interval $[0, N-1]$. This function returns the encrypted output vector, $Y^\beta = \langle Y_0^{(\beta)}, ..., Y_{N-1}^{(\beta)} \rangle$ and is described by Çetin as follows:

$$Y^\beta = \sum_{i \in [N]} (\sigma_i^{(\delta)} = j) X_i^{(\alpha)} \text{ for } j \in [N]. \tag{4}$$

The equality expression in equation 4 can be computed in parallel with a second application of the PPCs, where all sub-operations are set to 'equal'. Similar to the accumulator, the maximum number of bits to be compared for equality are $\lceil \log_2(N-1) \rceil$, therefore the overall minimum multiplicative depth for the equality circuit becomes $\log_2(\lceil \log_2(N-1) \rceil)$.

Overall, there are two separate applications of the PPCs to compute a sorted array. Multiple PPCs processed in blocks are required to span the entire comparison matrix. Computations occur in groups of N data slots within blocks, called *segments*. For instance, to sort 8 x 32-bit real numbers using ciphertexts with 630 slots and $\lfloor 630/32 \rfloor = 19$ available data slots, will result in a maximum of $\lfloor 19/8 \rfloor = 2$ segments per ciphertext block. Thus, sorting 8 numbers requires

a comparison matrix spanning 7 segments across 4 blocks. Sorting 16 x 32-bit numbers requires 15 blocks using one segment per block. Similarly, multiple blocks are required to process the equality circuit in addition to the comparison matrix. As blocks are processed independently, there is significant potential for multi-threaded block processing to further accelerate sorting times. Data slot preparations across single or multiple blocks add more operations, such as slot shifting and masking, to achieve equivalent bit manipulations compared to unpacked ciphertexts (*ie.* encoding individual bits or numbers). These factors increase overall circuit depth estimations in homomorphic procedures involving packed ciphertexts.

6.2 Min-Max

Depth unoptimised Min-Max To simplify analysis of a depth unoptimised min-max circuit, we initially look at processing encrypted real numbers encoded within one ciphertext block. In this case, two ciphertexts are packed with identical data slot inputs. As shown in Fig. 3(b), slots of the lower ciphertext are progressively shifted by 2^k in each stage towards the MSB. 'Less than' comparison sub-operations are computed in parallel from the previous stage's output using a single application of the PPCs per stage. In each stage, the PPCs' output is used as a selection mask to choose data slots from either upper or lower ciphertext inputs depending on whether the final desired output is a minimum or maximum value, respectively. Each stage therefore has a minimum circuit depth of $\lceil \log_2 n \rceil + 1$. The MSB will contain the final min-max output after $\lceil \log_2 N \rceil$ stages, resulting in an overall minimum circuit depth of $\lceil \log_2 N \rceil \cdot (\lceil \log_2 n \rceil + 1)$.

To determine the min-max value over large number sets, the 'less than' PPCs are first repeatedly applied across B multiple ciphertext blocks using a $\log_2 B$ stage binary tree similar to [30]. This process has a minimum circuit depth of $\lceil \log_2 B \rceil \cdot (\lceil \log_2 n \rceil + 1)$. Output from this stage is subsequently reduced onto one ciphertext block which can be processed finally by the single-block min-max circuit of similar depth; resulting in an overall minimum depth of $(\lceil \log_2 B \rceil + \lceil \log_2 N \rceil) \cdot (\lceil \log_2 n \rceil + 1)$. Despite the relative lower complexity of depth unoptimised min-max compared to the corresponding sorting algorithm, recryption would likely be required to process one ciphertext block of inputs. We therefore turn our attention to the depth optimised min-max algorithm and do not further consider the unoptimised version in this paper.

Matrix Comparison Min-Max The depth optimised version of min-max is a derivation of Çetin's Direct Sort algorithm. All steps are identical to matrix comparison sorting described in Section 6.1, except during the final stage where all matrix columns are multiplied instead of added, as shown in Fig. 3(c) - $(A \cdot B \cdot C)$. Multiplication generates a mask pointing to the index of the maximum value for the number array input. If the comparison matrix is inverted beforehand (all data slots in ciphertext B are XORed), the output of multiplication points to the minimum value. Multiplication can be performed firstly across blocks using a binary tree structure, then across segments within blocks using SIMD,

resulting in a minimum circuit depth of $\lceil \log_2(B \cdot N) \rceil$ in the final stage. This stage replaces the combined accumulator and equality circuit stages present in sorting, resulting in an overall reduced circuit depth for the optimised version of min-max compared to sorting. Multi-threaded processes could accelerate homomorphic multiplication of multiple ciphertext blocks at each binary tree level (with the lowest potential closest to the root), in addition to the multi-threaded computation of the comparison matrix blocks.

7 Evaluation

We evaluate performance of the accelerated matrix comparison sort against Çetin's standard open source implementation[6], in addition to evaluating the depth optimised version of min-max. Our implementations are in C++ and use the HElib library for homomorphic operations with multi-threading turned on, and compiled using the NTL library version 10.5.0 and GMP version 6.1.2. All simulations are conducted using a m2.4xlarge AWS EC2[7] execution environment, which is reported to use an Intel Xeon E5-2686 v4 processor with 8 vCPU cores at 2.30GHz and 32GB RAM, running Ubuntu 16.04.3 LTS. As shown in Table 3, parameters are derived to ensure that the security of HElib's BGV-based SWHE scheme is conservatively higher than that of the NTRU-based LTV-SWHE scheme used in Çetin's work. It should be noted that matching the security of differing SWHE schemes is a tricky problem [37], including having potential issues surrounding NTRU's security assumptions [38, 39]. The security level of all conducted tests are upwards from 108-bit, estimated according to [27]. HElib defaults are used for parameters not listed in Table 3.

Table 3. Selected HElib parameters for evaluation: m - cyclotomic ring, $\phi(m)$ - lattice dimension, λ - security level, L - levels, s - number of slots.

m	$\phi(m)$	λ	L	Plaintext Space	s
15709	15004	108.5	15	GF(2^{22})	682
23311	23310	172.5-134.1	17-20	GF(2^{45})	518

Table 4 displays sorting and min-max performance statistics on 4, 8 and 16 elements encoding 31-bit real numbers (1 bit less due to ciphertext packing structure). Average execution times are shown for each algorithm, where 20 runs are conducted for each array size test. This amount is adequate given the low coefficient of variance shown for executions times within each test (normally < 0.01). Depth optimised algorithms are accelerated using multi-threaded operations across blocks involving PPCs, the impact of which is explicitly demonstrated (multi-threading 'off' vs 'on' in Table 4). Although WTA accumulator

[6] github.com/vernamlab/FLaSH
[7] aws.amazon.com/ec2

multi-threading could potentially accelerate N repeated Hamming distance computations of the matrix columns, this was not done to allow focus on parallel effects of the PPCs alone. For comparison purposes, Çetin's sorting code is run within the same execution environment as the PPC implementation.

Table 4 demonstrates a significant speed up of sorting performance by a factor of approximately 25-30 times as a result of batched SIMD binary operations. The circuit depth associated with each array size input is higher in the PPC version due to overheads associated with processing packed ciphertexts. Ciphertext sizes are also compared for each array size. In the case of Çetin's matrix sort, despite being $\frac{1}{10}$ the size comparatively, one ciphertext is generated per bit of input, therefore $N \cdot n$ ciphertexts are processed initially during input ($\sim$ 335MB for N=16, 32-bit numbers). In the PPC matrix sort, only 2 ciphertexts per block, B are necessary to process inputs according to Alg. 1 ($\sim$ 180MB for N=16, 32-bit numbers). Subsequently, only $4 \cdot B$ and $2 \cdot B$ ciphertexts are required to compute Algs. 2 and 3 respectively. Processing fewer packed ciphertexts effectively reduces memory and communication costs associated with a homomorphic application.

Table 4. Performance results for depth-optimised sorting (comparison with Çetin results) and min-max using 31-bit real number array inputs. blk - block, seg - segment, mthr - multi-threading

Algorithm	**Array Size**	**4**	**8**	**16**
Matrix Sort - PPCs	**level**	17	19	20
	blk x seg/blk	1 x 4	4 x 2	15 x 1
	time - mthr off	**12s**	**55s**	**4m 8s**
	time - mthr on	**11s**	**49s**	**3m 40s**
	ciph. size (MB)	5.3	6.0	6.0
Matrix Sort - Çetin [18]	**level**	11	12	13
	time	**5m 10s**	**24m 34s**	**1h 50m**
	ciph. size (MB)	0.565	0.610	0.655
Min-Max - PPCs	**level**	15	15	15
	blk x seg/blk	1 x 4	4 x 2	15 x 1
	time - mthr off	**2s**	**10s**	**42s**
	time - mthr on	**3s**	**8s**	**33s**
	ciph. size (MB)	3	3	3

8 Conclusion

This paper proposes a simple to implement, primitive circuit design providing a general solution for efficiently performing mixed combinations of parallel logical

$(=, <, >, \leqslant, \geqslant)$ and numerical $(+, -)$ binary operations on real numbers. The PPCs aggregate parallel KSA and equality circuits that operate on almost fully packed ciphertexts (*ie.* less one bit per data slot). The general applicability and versatility of the design is demonstrated through its application to problems such as homomorphic sorting and min-max. In matrix comparison sorting, two separate applications of the PPCs involved are comparison $(<)$ and equality $(=)$, whereas only the former is used in the depth optimised min-max algorithm. For the first time, input elements are real numbers and a competitive speed-up is achieved compared to more specialised existing schemes through flexible primitive circuit design, SIMD batching of densely packed ciphertexts, and a potential for multi-threading operations across ciphertext blocks. Data slot arrangements involved in parallel design normally result in higher multiplicative circuit depths compared to more memory-intensive, non-batched implementations. The circuit depth, multiplication cost and memory footprint of the PPCs were minimised as much as possible by analysing several primitive circuit designs before selecting a final combination. The challenge of developing efficient FHE applications that optimise computational resource and communication requirements is underpinned by primitive circuit combinations that balance effective parallel design with a minimum total number of recryptions.

References

[1] Craig Gentry. "A fully homomorphic encryption scheme". PhD thesis. Stanford University, 2009.

[2] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *Journal of the ACM (JACM)* 56.6 (Sept. 2009), p. 34. DOI: 10.1145/1568318.1568324.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. Association for Computing Machinery (ACM), 2012, pp. 309–325. DOI: 10.1145/2090236.2090262.

[4] Shai Halevi and Victor Shoup. "Algorithms in HElib". In: *International Cryptology Conference*. Springer Nature, 2014, pp. 554–571. DOI: 10.1007/978-3-662-44371-2_31.

[5] Kristian Gjøsteen and Martin Strand. "Can there be efficient and natural FHE schemes?" In: *IACR Cryptology ePrint Archive, Report 2016/105* (2016).

[6] Mihai Togan, Cristian Lupascu, and Cezar Plesca. "Homomorphic Evaluation of the Speck Cipher". In: *Proceedings of the Romanian Academy, Series A* 16 (2015), pp. 375–383.

[7] Nigel P Smart and Frederik Vercauteren. "Fully homomorphic SIMD operations". In: *Designs, codes and cryptography* (2014), pp. 1–25.

[8] Shai Halevi and Victor Shoup. "Design and implementation of a homomorphic-encryption library". In: *IBM Research (Manuscript)* 6 (2013), pp. 12–15.

[9] Angela Jäschke and Frederik Armknecht. "Accelerating homomorphic computations on rational numbers". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2016, pp. 405–423.

[10] Gizem S Çetin et al. "Arithmetic using word-wise homomorphic encryption". In: *IACR Cryptology ePrint Archive, Report 2015/1195* (2016).

[11] Nathan Dowlin et al. *Manual for using homomorphic encryption for bioinformatics*. Tech. rep. Technical report MSR-TR-2015-87, Microsoft Research, 2015.

[12] Jung Hee Cheon et al. "Homomorphic Encryption for Arithmetic of Approximate Numbers." In: *IACR Cryptology ePrint Archive, Report 2016/421* (2016).

[13] Seiko Arita and Shota Nakasato. "Fully Homomorphic Encryption for Point Numbers." In: *IACR Cryptology ePrint Archive, Report 2016/402* (2016).

[14] Carlos Aguilar-Melchor et al. "Recent advances in homomorphic encryption: A possible future for signal processing in the encrypted domain". In: *IEEE Signal Processing Magazine* 30.2 (Mar. 2013), pp. 108–117. DOI: 10.1109/msp.2012.2230219.

[15] Ayantika Chatterjee, Manish Kaushal, and Indranil Sengupta. "Accelerating Sorting of Fully Homomorphic Encrypted Data." In: *Lecture Notes in Computer Science*. Springer. 2013, pp. 262–273. DOI: 10.1007/978-3-319-03515-4_17.

[16] Nigel P Smart and Frederik Vercauteren. "Fully homomorphic encryption with relatively small key and ciphertext sizes". In: *International Workshop on Public Key Cryptography*. Springer Nature, 2010, pp. 420–443. DOI: 10.1007/978-3-642-13013-7_25.

[17] Abbas Acar et al. "A Survey on Homomorphic Encryption Schemes: Theory and Implementation". In: *CoRR* abs/1704.03578 (2017). arXiv: 1704.03578.

[18] Gizem S Çetin et al. "Depth optimized efficient homomorphic sorting". In: *International Conference on Cryptology and Information Security in Latin America*. Springer. Springer International Publishing, 2015, pp. 61–80. DOI: 10.1007/978-3-319-22174-8_4.

[19] Wei Dai and Berk Sunar. "cuHE: A homomorphic encryption accelerator library". In: *International Conference on Cryptography and Information Security in the Balkans*. Springer. 2015, pp. 169–186. DOI: 10.1007/978-3-319-29172-7_11.

[20] Pyung Kim, Younho Lee, and Hyunsoo Yoon. "Sorting Method for Fully Homomorphic Encrypted Data Using the Cryptographic Single-Instruction Multiple-Data Operation". In: *IEICE Transactions on Communications* 99.5 (2016), pp. 1070–1086. DOI: 10.1587/transcom.2015ebp3359.

[21] Nitesh Emmadi et al. "Updates on sorting of fully homomorphic encrypted data". In: *Cloud Computing Research and Innovation (ICCCRI), 2015 International Conference on*. IEEE. 2015, pp. 19–24.

[22] Ovunc Kocabaş and Tolga Soyata. "Utilizing homomorphic encryption to implement secure and private medical cloud computing". In: *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE. IEEE, June 2015, pp. 540–547. DOI: 10.1109/cloud.2015.78.

[23] Mihai Togan, Luciana Morogan, and Cezar Plesca. "Comparison-Based Applications for Fully Homomorphic Encrypted Data". In: *Proceedings of the Romanian Academy Series A* 16 (2015), pp. 329–338.

[24] Vinod Vaikuntanathan. "Computing blindfolded: New developments in fully homomorphic encryption". In: *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*. IEEE. Institute of Electrical and Electronics Engineers (IEEE), Oct. 2011, pp. 5–16. DOI: 10.1109/focs.2011.98.

[25] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2010, pp. 1–23.

[26] Craig Gentry, Shai Halevi, and Nigel P Smart. "Fully homomorphic encryption with polylog overhead". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2012, pp. 465–482.

[27] C Gentry, S Halevi, and NP Smart. "Homomorphic evaluation of the AES circuit (updated implementation)". In: *IACR Cryptology ePrint Archive, Report 2012/099* (2015).

[28] Anamaria Costache et al. "Fixed Point Arithmetic in SHE Scheme." In: *IACR Cryptology ePrint Archive, Report 2016/250* (2016).

[29] Chen Xu et al. "Homomorphically Encrypted Arithmetic Operations Over the Integer Ring". In: *International Conference on Information Security Practice and Experience*. Springer. 2016, pp. 167–181.

[30] Övünç Kocabaş. "Design and Analysis of Privacy-Preserving Medical Cloud Computing System". PhD thesis. 2016.

[31] Kevin C King. "Optimizing Fully Homomorphic Encryption". MA thesis. 2016.

[32] Megha Talsania and Eugene John. "A comparative analysis of parallel prefix adders". In: *Proceedings of the International Conference on Computer Design (CDES)*. 2013, p. 1.

[33] Tohoku University. *Hardware algorithms for arithmetic modules*. Tohoku University. Jan. 5, 2018. URL: http://www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html (visited on 01/05/2018).

[34] Jung Hee Cheon, Miran Kim, and Myungsun Kim. "Optimized search-and-compute circuits and their application to query evaluation on encrypted data". In: *IEEE Transactions on Information Forensics and Security* 11.1 (2016), pp. 188–199.

[35] Myungsun Kim et al. "On the Efficiency of FHE-based Private Queries". In: *IEEE Transactions on Dependable and Secure Computing* (2016).

[36] Christopher S Wallace. "A suggestion for a fast multiplier". In: *IEEE Transactions on electronic Computers* EC-13.1 (Feb. 1964), pp. 14–17. DOI: 10.1109/pgec.1964.263830.

[37] Ana Costache and Nigel P Smart. "Which ring based somewhat homomorphic encryption scheme is best?" In: *Cryptographers' Track at the RSA Conference*. Springer. 2016, pp. 325–340. DOI: 10.1007/978-3-319-29485-8_19.

[38] Martin Albrecht, Shi Bai, and Léo Ducas. "A subfield lattice attack on overstretched NTRU assumptions". In: *Annual Cryptology Conference*. Springer. Springer Berlin Heidelberg, 2016, pp. 153–178. DOI: 10.1007/978-3-662-53018-4_6.

[39] Martha Norberg Hovd. "The Handling of Noise and Security of Two Fully Homomorphic Encryption Schemes". MA thesis. Norwegian University of Science and Technology, 2017.