Homomorphically Encrypted Arithmetic Operations Over the Integer Ring

Chen Xu, Jingwei Chen(✉), Wenyuan Wu, and Yong Feng

Chongqing Key Laboratory of Automated Reasoning and Cognition,
Chongqing Institute of Green and Intelligent Technology,
Chinese Academy of Sciences, Chongqing 400714, China
{xuchen,chenjingwei,wuwenyuan,yongfeng}@cigit.ac.cn

Abstract. Fully homomorphic encryption allows cloud servers to evaluate any computable functions for clients without revealing any information. It attracts much attention from both of the scientific community and the industry since Gentry's seminal scheme. Currently, the Brakerski-Gentry-Vaikuntanathan scheme with its optimizations is one of the most potentially practical schemes and has been implemented in a homomorphic encryption C++ library HElib. HElib supplies friendly interfaces for arithmetic operations of polynomials over finite fields. Based on HElib, Chen and Guang (2015) implemented arithmetic over encrypted integers. In this paper, we revisit the HElib-based implementation of homomorphically arithmetic operations on encrypted integers. Due to several optimizations and more suitable arithmetic circuits for homomorphic encryption evaluation, our implementation is able to homomorphically evaluate 64-bit addition/subtraction and 16-bit multiplication for encrypted integers without bootstrapping. Experiments show that our implementation outperforms Chen and Guang's significantly.

Keywords: Fully homomorphic encryption · HElib · Arithmetic circuit · Integer operation · C++ implementation

1 Introduction

A fully homomorphic encryption (FHE) scheme is an encryption scheme that allows evaluation of arbitrarily functions on encrypted data. FHE was firstly pointed out by Rivest et al. [26] and was known to have a lot of applications in cryptography, especially in cloud security, but no secure scheme was known until Gentry's seminal work [11,12]. Since then, there are many works followed, e.g., [2–4,6,9,10,13,14,16,28], towards a practical FHE scheme. Among them, the BGV scheme [3] is one of the most efficient FHE shemes, and is considered as one of the most potentially practical ones, since it is based on the learning with error (LWE) assumption [25] or the ring-LWE (RLWE) assumption [21] and supports single-instruction-multiple-data (SIMD) operations under certain

F. Bao et al. (Eds.): ISPEC 2016, LNCS 10060, pp. 167–181, 2016.
DOI: 10.1007/978-3-319-49151-6_12

settings [28]. Also, the BGV scheme has already been implemented by Halevi and Shoup based on Shoup's number theory library NTL [27], named HElib [17].

More specifically, HElib includes implementations of all the basic functions in the BGV scheme with the support of SIMD operations [28] and the Gentry-Halevi-Smart optimizations [14]. As indicated by the authors of HElib in [19]: "... the lower-level of HElib ... is executed on a 'hardware platform' given by the underlying HE scheme", since the BGV scheme (besides almost all of the currently known FHE schemes) is designed for circuits. However, most often, when we think of computations, we do not think in terms of circuits, but in terms of RAM machines, or even high level programming languages. Therefore, for variants of more advanced applications, it is necessary to build some higher level functions based on HElib. For instance, the encrypted arithmetic operations over the integer ring should be included, since it is frequently used in, e.g., statistical functions such as mean, covariance, standard deviation, linear regression, etc.

In this paper, we use HElib to implement truly integer arithmetic operations via binary circuits, including addition, subtraction, multiplication and division with reminder. Our implementation is able to homomorphically evaluate 64-bit addition/subtraction and 16-bit multiplication for encrypted integers without bootstrapping; see Sects. 3 and 4 for details. To our best knowledge, the paper [5] by Chen and Guang is the first published work on this topic. In [5], the authors only reported their experiments of homomorphically encrypted arithmetic operations on integers with bits at most 4.

1.1 Related Work

Here we only focus on implementations of secure computation for integers, although there are a large number of other applications of FHE which have been implemented, such as AES [15]. In fact, before the appearance of FHE, there were already some work related to secure computation for integers. For instance, Kolesnikov et al. [20] presented several efficient garbled circuit constructions for integer addition, subtraction, multiplication, and comparison functions. With the development of FHE, some work related to FHE implementation appears. In [23], Naehrig et al. discussed integer arithmetic operations based on their implemention of a RLWE-based somewhat homomorphic encryption in the computer algebra system Magma [22]. It seems not relevant any more since it does not feature some key techniques, including modulus switching. Later on, Wu and Haven [29] presented their implementation for large scale statistical analysis based on HElib, including linear regression and mean and covariance computation. However, their method only supports arithmetic operations over $\mathbb{Z}_p$ with $p > 2^{128}$, which implies that the division of two integers (with remainder) can not be completely performed homomorphically and must be finished offline by the client, so does the DGHV scheme [10]. The DGHV scheme [10] and its optimizations [6] are aiming at secrue large integer arithmetic, however, the integer arithmetic is also over the integers modulo an even larger integer. We implement the carry computation in present work, so that our implementation supports arithmetic operations over the integer ring (not $\mathbb{Z}_p$). Chen and Guang [5] reported a similar

implementation of integer arithmetic over ciphertexts based on HElib. Both of [5] and ours use certain basic arithmetic circuits for corresponding integer operations without bootstrapping. The main difference is that we design such circuits more carefully. In particular, we design those circuits with less number of AND gates, since it is well-known that the number of AND gates of a circuit impacts heavily on the efficiency of FHE evaluation. For example, we adopt the integer addition circuit from [20, Sect. 3.1], which only needs one-half of AND gates used in [5, Sect. II]. In order to speed up further, we also implement a homomorphic carry-lookahead adder (CLA). Combining with several other optimizations leads that our implementation is not only more efficient than [5], but also able to deal with integers with larger size. In particular, our implementation supports 64-bit addition/subtraction and 16-bit multiplication with the multiplicative depth at most 17. We note that Cheon et al. [8] reported their implementation for binary integer addition (with equality test and comparison) based on SIMD circuits and HElib. The efficiency reported in [8] is very competitive. Comparing with theirs, our implementation supports integeral vector operations by means of SIMD, since we only use one slot for each computation.

2 Preliminaries

In this section, we give some basics related to FHE, the BGV scheme and HElib, which are useful for the rest of the paper. We refer to [3,11,18] for more details.

2.1 Fully Homomorphic Encryption

A public-key encryption scheme consists of three algorithms: `KeyGen`, `Enc`, and `Dec`. `KeyGen` is an algorithm that takes a security parameter λ as input, and outputs a secret key `sk` and a public key `pk`; `pk` defines a plaintext space $\mathcal{P}$ and a ciphertext space $\mathcal{C}$. `Enc` is an algorithm that takes `pk` and a plaintext $b \in \mathcal{P}$ as input, and outputs a ciphertext $c \in \mathcal{C}$. `Dec` takes `sk` and c as input, and outputs the plaintext b. The computational complexity of all of these three algorithms must be probabilistic polynomial time in λ. The correctness is defined as: if $(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathtt{KeyGen}$, $b \in \mathcal{P}$, and $c \leftarrow \mathtt{Enc}(\mathtt{pk}, b)$, then $\mathtt{Dec}(\mathtt{sk}, c) \rightarrow b$.

A homomorphic encryption (HE) scheme has an efficient algorithm `Eval` in addition to the three conventional algorithms. `Eval` takes as input the public key `pk`, a function f and a tuple of ciphertexts $\boldsymbol{c} = (c_1, \cdots, c_t)$, where $c_i \leftarrow \mathtt{Enc}(\mathtt{pk}, b_i)$ for $b_i \in \mathcal{P}$; it outputs a ciphertext $c \in \mathcal{C}$. The correctness is defined as follows: if $c \leftarrow \mathtt{Eval}(\mathtt{pk}, f, \boldsymbol{c})$, then $\mathtt{Dec}(\mathtt{sk}, c) \rightarrow f(b_1, \cdots, b_2)$. In almost all HE schemes, the function f to be homomorphically evaluated is described in a circuit model with XOR and AND gates, which correspond to binary addition and multiplication, respectively. Furthermore, a HE scheme is only able to evaluate circuits of limited depth as with increasing depth, the noise of ciphertexts increases so dramatically that `Dec` can not recover the correct plaintext from ciphertexts with large depth.

A fully homomorphic encryption (FHE) scheme is a HE scheme that is able to evaluate circuits with depth larger than its own Dec function. This condition allows to perform the so-called "bootstrapping" process successfully, and makes such a scheme is able to evaluate all computable circuits.

2.2 The BGV Scheme

The BGV scheme [3] can be seen as an improvement of the "second generation" of FHE given by Brakerski and Vaikuntanathan [4], which are based on standard assumptions supported by worst-case hardness of LWE or RLWE, while the "first generation" FHE constructions [10,12] are based on ad-hoc average case assumptions about ideal lattices and the approximation GCD problem. In addition, BGV is capable of evaluating arbitrary circuits of a priori bounded depth without the bootstrapping procedure. Here we only describe a variant of the basic BGV encryption schmeme that is implemented in HElib and works as follows.

- $\texttt{Setup}\,(1^\lambda)$. Given the security parameter λ as input, set an integer m (that defines the m-th cyclotomic polynomial $\Phi_m(x)$), an odd modulus q (we will work over $R_q = \mathbb{Z}_q[x]/\Phi_m(x)$), the noise distribution χ over R_q, and $N = \text{polylog}(q)$. Output $params = (m, q, \chi, N)$.
- $\texttt{KeyGen}\,(params)$. Sample $t \leftarrow \chi$. Let $\boldsymbol{s} = (1, t) \in R_q^2$. Set $\texttt{sk} = \boldsymbol{s}$. Generate $\boldsymbol{B} \leftarrow R_q^N$ uniformly at random and a column vector with "small" coefficients $\boldsymbol{e} \leftarrow \chi^N$. Set $\boldsymbol{b} = \boldsymbol{B}t + 2\boldsymbol{e}$. Output $\texttt{sk} = \boldsymbol{s}$ and the public key $\boldsymbol{A} = (\boldsymbol{b} \| -\boldsymbol{B})$.
- $\texttt{Enc}\,(params, \texttt{pk}, m)$. To encrypt a message $b \in R_2$, set $\boldsymbol{m} = (b, 0) \in R_q^2$, sample a colume vector with small coefficients $\boldsymbol{r} \leftarrow R_2^N$ and output the ciphertext $\boldsymbol{c} = \boldsymbol{m} + \boldsymbol{r}^T\boldsymbol{A} \in R_q^2$.
- $\texttt{Dec}\,(params, \texttt{sk}, \boldsymbol{c})$. Output the message $b = [[\langle \boldsymbol{c}, \boldsymbol{s} \rangle]_q]_2$.

Remark 1. We limit the plaintext space to R_2 in this paper, since it is convenient for integer arithmetic circuit design, although the scheme described above also handles plaintext spaces larger than R_2.

Note that the quantity $[\langle \boldsymbol{c}, \boldsymbol{s} \rangle]_q$ is called the *noise* of the ciphertext $\boldsymbol{c}$ under the secret key s. Decryption works correctly as long as we ensure that the noise of the ciphertext is small enough and does not warp around modulo q. Thus we have $[[\langle \boldsymbol{c}, \boldsymbol{s} \rangle]_q]_2 = [[\langle \boldsymbol{m} + \boldsymbol{r}^T\boldsymbol{A}, \boldsymbol{s} \rangle]_q]_2 = [[b + 2\boldsymbol{r}^T\boldsymbol{e}]_q]_2 = [b + 2\boldsymbol{r}^T\boldsymbol{e}]_2 = b$.

Homomorphic Evaluation. The BGV scheme supports homomorphic addition and multiplication. Let $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ be two ciphertexts of two plaintexts b_1 and b_2 under the same secret key $\boldsymbol{s}$, and suppose that the noise of $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ is bounded from above by B. The addition of two ciphertexts is simply a component-wise addition, i.e., $\boldsymbol{c}_+ = \boldsymbol{c}_1 + \boldsymbol{c}_2$ is a ciphertext of $b_1 + b_2$ under the secret key $\boldsymbol{s}$. The noise of $\boldsymbol{c}_+$ is at most $2B$. Multiplication is a bit more complicated, but we still have that $\boldsymbol{c}_\times = \boldsymbol{c}_1 \otimes \boldsymbol{c}_2$ is a ciphertext of $b_1 \cdot b_2$ under the new secret key $\boldsymbol{s} \otimes \boldsymbol{s}$, where $\otimes$ represents the tensor product. Furthermore, the noise of $\boldsymbol{c}_\times$

can only be bounded from above by B^2. To keep the secret key with small size and to decrease the noise of evaluated ciphertext, the key switching procedure and modulus switching procedure are used in the BGV scheme, respectively. Theoretically, in the BGV scheme, the cost of each homomorphical addition or multiplication increases fast as the circuit depth L grows. In the case of R_2, the cost is $\widetilde{\mathcal{O}}(\lambda \cdot L^3)$ (see [3] for more details).

Batching. Batching allows us to evaluate a function homomorphically in parallel on ℓ blocks of encrypted data. Batching works essentially by packing multiple plaintexts into one ciphertext. More specifically, when the plaintext space is limited to $R_2 = \mathbb{Z}[x]/\langle \Phi_m(x), 2\rangle$, where $\Phi_m(x)$ is the m-th cyclotomic polynomial, $\Phi_m(x)$ can be factorized into ℓ irreducible factors of same degree $d = \phi(m)/\ell$, i.e., $\Phi_m(x) = \prod_{i=1}^{\ell} f_i(x)$, where $\phi(\cdot)$ is the Euler's totient function. Each factor corresponds to a plaintext slot. Thus, for each $a \in R_2$, it can be represented as an ℓ-vector $(a \mod f_i)_{1 \le i \le \ell}$. Using the techniques in [14,28], one can perform SIMD operations on ℓ blocks of ciphertexts. Here we note that m is the dominating parameter for efficiency as it determines the size of computation.

2.3 HElib

HElib [17] is an open-source library which implements the BGV scheme with some optimizations such as ciphertext packing techniques (SIMD) [28] and optimizations in [14]. There are many useful functions in the library besides the evaluation of the AND gate and the XOR gate, including some initialization functions, and some helper classes like `EncryptedArray` which provides us with easy encryption and manipulation to the ciphertext slots.

In the library one ciphertext contains several large polynomials in R_q, where $q = \prod p_j$ is the modulus and each p_j is a small prime generated by the library. Every large polynomial is represented as a polynomial matrix. The matrix contains $\phi(m)$ columns and the i-th column represents the ciphertext modulo $f_i(x)$. The j-th row contains the FFT representation of a modulo p_j. So in HElib, the homomorphic addition corresponds to the polynomial addition in FFT form, and homomorphic multiplication corresponds to the polynomial multiplication in FFT form, which is element-wise multiplication.

From above, it is clear that both the size of matrices and the degree of the matrix entries depend only on $\phi(m)$, and hence the parameter m. In HElib, the parameter m is chosen such that

$$\phi(m) \ge \frac{(L_c(\log \phi(m) + 23) - 8.5)(\lambda + 110)}{7.2}, \tag{1}$$

where L_c is the minimum number of levels of modulus chain and λ is the security parameter; see the full version of [15].

In applications, the minimum number of levels in the modulus chain L_c in HElib is actually the number of modulus switches L_s plus one. And L_s is close but may not equal to the multiplicative depth L. This is because sometimes the

resulting ciphertext does not exceed the noise threshold after a multiplication, in which case, it is not necessary to perform the modulus switching process. What's more, although the effect is small, additions also accumulate noise which may contribute to modulus switching. In HElib, $L_s \approx 2\left\lceil \frac{L}{2} \right\rceil$, and thus $L_c \approx 2\left\lceil \frac{L}{2} \right\rceil + 1$.

From (1), a larger L_c implies a larger $\phi(m)$. This makes both addition and multiplication over ciphertexts less efficient. Thus, when we design a circuit for a certain application, we may choose those circuits with the multiplicative depth as less as possible.

3 Homomorphically Encrypted Arithmetic Operations

In BGV scheme, if we choose R_2 as the plaintext space, we can map the addition and multiplication in the scheme into AND ($\cdot$) and XOR ($\oplus$) logic gates. They are actually the foundations of the larger and more complex circuits (functions).

In this section, we present our implementation of integer arithmetic operations by using AND and XOR gate evaluation in HElib with several optimizations. Note that in FHE, AND gate evaluation is much more expensive than XOR gate evaluation because of the potential modulus switching, so the core problem here we try to solve is to minimize the multiplicative depth as well as the number of AND gates.

We use one ciphertext to represent one bit in our implementation, and a binary integer is a double-ended queue of ciphertext. In what follows, all bits we use are encrypted by the HElib function `EncryptedArray::encrypt`.

3.1 Addition

Addition is the most basic module in integer arithmetic and can be used in other arithmetic operations like subtraction, multiplication and division.

In this paper, we implement several different structures of adders, and use them in different scenarios. We first adapt the full-adder to the FHE, by reducing the number of AND gates. Based on that, we implement the Ripple Carry Adder (RCA) which has a simple structure but needs more multiplicative depth. In contrast, we also implement the Carry Lookahead Adder (CLA) which has a more complex structure but needs less multiplicative depth since the operations in the CLA can work in parallel. Besides, we build a "half adder chain" which is useful in division.

Full Adder. The basic modules of an adder include some half adders and some full adders. The difference is that a half adder does not accept the carry-in information while a full adder does. The implementation can be varied as long as the logic expressions of different implementations are equivalent. In [5], for example, the expressions of carry-in and sum of the full adder are as follows:

$$\begin{aligned} c_{i+1} &= a_i \cdot b_i \oplus c_i \cdot (a_i \oplus b_i), \\ s_i &= a_i \oplus b_i \oplus c_i, \end{aligned}$$

where a_i and b_i are the i-th bit of two summands, c_i is the i-th carry-in bit, and s_i is the i-th sum bit. In fact, the number of AND gates for the carry-out can be reduced from two down to one with the following optimization.

$$\begin{aligned} c_{i+1} &= a_i \cdot b_i \oplus c_i \cdot (a_i \oplus b_i) \\ &= a_i \cdot b_i \oplus a_i \cdot c_i \oplus b_i \cdot c_i \\ &= a_i \cdot b_i \oplus a_i \cdot c_i \oplus b_i \cdot c_i \oplus c_i \oplus c_i \cdot c_i \\ &= (a_i \oplus c_i) \cdot (b_i \oplus c_i) \oplus c_i, \end{aligned}$$

which can be found in, e.g., [20, Sect. 3.1]. In this way, it needs only one AND gate per bit, and hence the multiplicative depth of this full adder is $L = 1$.

Ripple Carry Adder (RCA). An n-bit RCA (Algorithm 1) is constructed by one half adder and $n - 1$ full adders. This adder adds one bit at a time, from the least significant bit to the most significant bit. The multiplicative depth is $L = n - 1$, since for every bit except MSB we need one AND gate and every next bit depends on the previous one.

Algorithm 1. (Ripple carry adder).

Input: n-bit number a, b
Output: the sum s.

1: $c_0 = 0$
2: **for** $i = 0$ to $n - 2$ **do**
3: $\quad s_i = a_i \oplus b_i \oplus c_i$
4: $\quad c_{i+1} = (a_i \oplus c_i) \cdot (b_i \oplus c_i) \oplus c_i$
5: **end for**
6: $s_{n-1} = a_{n-1} \oplus b_{n-1} \oplus c_{n-1}$
7: **return** s.

Carry Lookahead Adder (CLA). Since an n-bit ripple carry adder needs multiplicative depth of $n - 1$, the overload of polynomial computations soon becomes unacceptable as n increases. One way to solve this problem is to use CLA. Unlike the circuit of RCA whose structure is a chain, the circuit of CLA is like a tree with the root at the bottom. This adder needs more computations than RCA, but the multiplicative depth $L = \mathcal{O}(\log n)$ (see [24]) that is much smaller than RCA ($L = n - 1$) when n is large.

The two elements of CLA are *generate function* $g_i = a_i \cdot b_i$ and *propagate function* $p_i = a_i \oplus b_i$, which have the following properties: if g_i is one, c_{i+1} will be one and if p_i is one, c_{i+1} will be c_i. Thus, we have

$$c_{i+1} = g_i \oplus p_i \cdot c_i.$$

In our implementation, we use 4-bit CLA adder as a unit to construct the whole adder, thus an n-bit addition is divided into $\lceil \log_4 n \rceil$ levels, and at each

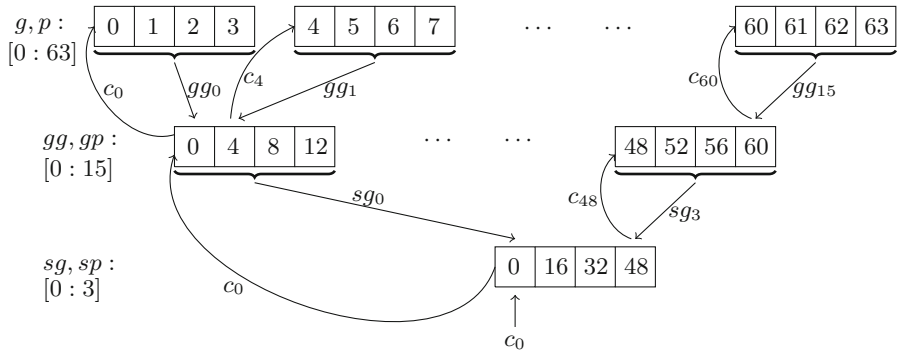

Fig. 1. The tree structure of CLA

level the dependent calculation is confined inside the 4-bit group, see Fig. 1. This is a recursive procedure: g and p in the lower level is determined by $f_g(g, p)$ and $f_p(g, p)$, and c_{j+4} in the lower level's group, correspondingly, is

$$c_{j+4} = f_g(g_{4j}, p_{4j}) \oplus f_p(g_{4j}, p_{4j}) \cdot c_j,$$

where

$$f_g(g_i, p_i) = g_{i+3} \oplus p_{i+3} \cdot g_{i+2} \oplus p_{i+3} \cdot p_{i+2} \cdot g_{i+1} \oplus p_{i+3} \cdot p_{i+2} \cdot p_{i+1} \cdot g_i,$$
$$f_p(g_i, p_i) = p_{i+3} \cdot p_{i+2} \cdot p_{i+1} \cdot p_i.$$

First, we compute all the g_i and p_i from $i = 0$ to 63. Then we use g_i and p_i to compute a 4-bit group generate function named gg_j and group propagate function gp_j, from $j = 0$ to 15. At this stage, there are $64/4 = 16$ groups in the circuit.

Then we use the same method to compute the super group generate and propagate function sg_k and sp_k, from $k = 0$ to 3. A super group is consisted with 4 groups. So there are $16/4 = 4$ super groups in the circuit.

Now we get to the base case, and we can compute the carry-in of each super group c_{16}, c_{32}, c_{48} with c_0, sg_k and sp_k. After that we use these carry-in along with gg_j and gp_j to compute the carry-in of each group (e.g., c_4, c_8, c_{12}, etc.). Finally, we use the carry-in of each group with g_i and p_i to compute the carry-in of each bit(e.g., c_1, c_2, c_3, etc.). At this stage, we have computed all the carry-in bits, then we have the sum bits $s_i = a_i \oplus b_i \oplus c_i$.

We describe 64-bit CLA in Algorithm 2. For 16-bit case, there are only two levels instead of three, but the idea is the same.

Half Adder Chain. In the division algorithm, we need to compute the additive inverse of an integer. This is achieved by adding 1 to the bit-complement of the number. Since no carry-in is needed in the procedure, it is better to simplify the addition by using half adders instead of full adders. We add the first bit of the number to 1, and for the rest of the bits we add the carry-in to the i-th bit to get the i-th sum bit.

Algorithm 2. (Carry Lookahead Adder).

Input: 64-bit number a, b
Output: the sum s.

1: **for** $i = 0$ to 63 **do** $g_i = a_i \cdot b_i$, $p_i = a_i \oplus b_i$, $c_i = 0$ **end for**
2: **for** $j = 0$ to 15 **do** $gg_j = f_g(g_{4j}, p_{4j})$, $gp_j = f_p(g_{4j}, p_{4j})$ **end for**
3: **for** $k = 0$ to 3 **do** $sg_k = f_g(gg_{4k}, gp_{4k})$, $sp_k = f_p(gg_{4k}, gp_{4k})$ **end for**
4: **for** $k = 0$ to 2 **do** $c_{16(k+1)} = sg_k \oplus sp_k \cdot c_{16k}$ **end for**
5: **for** $k = 0$ to 3, $j = 0$ to 2 **do**
$c_{16k+4(j+1)} = gg_{4k+j} \oplus gp_{4k+j} \cdot c_{16k+4j}$
end for
6: **for** $k = 0$ to 3, $j = 0$ to 3, $i = 0$ to 2 **do**
$c_{16k+4j+(i+1)} = g_{16k+4j+i} \oplus p_{16k+4j+i} \cdot c_{16k+4j+i}$
end for
7: Calculate $s_i = p_i \oplus c_i$ for $i = 0$ to 63
8: **return** s.

3.2 Subtraction

We can construct a subtractor in two general ways. One way is to derive the logic expressions of 1-bit subtractor and then chain the unit together like RCA. We call it Ripple Carry Subtractor (RCS). The logic expression of 1-bit subtractor is virtually the same as full adder, we just give the optimized expression of the difference bit d_i and the borrow bit c_i:

$$\begin{aligned} c_{i+1} &= (a_i \oplus c_i) \cdot (b_i \oplus c_i) \oplus b_i, \\ d_i &= a_i \oplus b_i \oplus c_i. \end{aligned}$$

Chaining the unit together, we get the n-bit RCS.

The other way is simpler because we can use adder to carry out subtraction. Since we use two's complement as the data representation, we have $a - b = a + \tilde{b} + 1$, where $\tilde{b}$ means the bit-wise complement of b, i.e., $\tilde{b_i} = b_i \oplus 1$. Thus if we first change b to $\tilde{b}$, then set the first carry-in bit c_0 to 1, we can do subtraction with an adder.

As we can see, the multiplicative depth of RCS is equal to that of RCA adder, since we have one AND gate for every borrow bit, and it is used to get the next borrow bit. Thus, the multiplicative depth of a n-bit RCS is $L = n - 1$. And since bit-wise complement only involves FHE addition, it does not increase multiplicative depth. We still have the multiplicative depth $L = \mathcal{O}(\log n)$ for the n-bit CLA subtraction.

3.3 Multiplication

Multiplication is constructed by additions, in a pencil and paper way. We use one binary number to multiply every bit of the other number, and thus we get n middle results. After that we left shift each middle results and add them together using the adder we mentioned above.

Here we have two techniques to reduce the number of AND gates. First, if we do not concern about the overflow of multiplication and just need a n-bit result, we can left align the integer and ignore the padding zeros on the right side. What's more, we carefully arrange the order of additions, thus to minimize the number of AND gates.

For example, if we multiply two 4-bit numbers, 2 and 3, we do the arithmetic as in Fig. 2. We first compute the 4 middle results 0010, 0010, 0000 and 0000, and shift the results to the correct position. Since we do not consider the overflow situation, we can truncate the higher bits in the left. Then we perform the addition in the following way. We add the first and the second number by adding three highest bits in the left, that is 001 and 010 showed in the dotted line. Since the lowest bit 0 on the right will not change after the addition, we just keep it. We also use the same way adding the third and the fourth number. In the second step, we add the two partial sums together in the same way and get the final result.

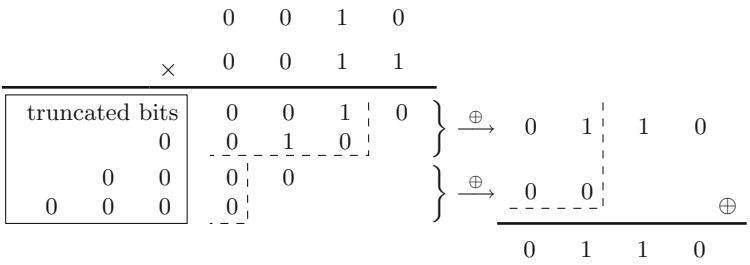

Fig. 2. Multiplying two integers 2 and 3 in a 4-bit binary circuit

Here we give the algorithm of multiplying two n-bit numbers in Algorithm 3. (Since the level of additions to sum all middle results is $\approx \log n$, we assume n is a power of 2 in the following algorithm description).

Algorithm 3. (Multiplier).

Input: n-bit encrypted number a, b
Output: the product c.

for $i = 1$ to $n - 1$ **do**
  $temp_i = a \cdot b_i$
end for
$level = \log_2 n, db = 1$
while $level > 0$ **do**
  for $i = 0$ to $size/(2 \cdot db)$ **do**
    $temp_{2i \cdot db} = temp_{2i \cdot db} + temp_{(2i+1)db}$
  end for
  $db = db \cdot 2$, $level = level - 1$
end while
$c = temp_0$
return c.

The multiplicative depth of multiplication is one level larger than the addition, since one level is used when we computing the middle results, and the addition of middle results costs $n-1$ levels. Therefore, the multiplicative depth $L = n$.

3.4 Division

We implement division using the non-restoring division method, which is the same as that in [5], and we omit the algorithmic description here. Note that this is not an efficient algorithm due to the large multiplicative depth brought by iterative addition of the partial reminder R and $\pm b$, where b is the divisor. The multiplicative depth L is about $\text{len}(a) \cdot \text{len}(b)$, where a is the dividend.

Nonetheless, We have a slight improvement on the algorithm. Since in the non-restoring division algorithm, we need to compute $R + b$ or $R + (-b)$ at each loop, but there is no need to compute $-b$ every time. For this reason, we precompute $-b$ at the very beginning. Furthermore, when using the $\tilde{b} + 1$ method to compute $-b$, we use a half adder chain rather than a full adder, as mentioned in Sect. 3.1. This can reduce three XOR gates per bit.

4 Experimental Results

In this section, we report the experimental results of our implementation described above and compare it with the similar implementation in [5].

Parameter Settings. There are many parameters in HElib interface, most of which are used to compute the integer m. The library provides a function `FindM()` which can determine a proper m according to the input parameters. Among these parameters, security level λ and levels in the modulus chain L_c are the most important ones, as we explained in Sect. 2.3.

In our experiments, we set the security level $\lambda = 80$ (that implies the breaking time of the encryption sheme is roughly 2^{80}) which is a reasonable value. Unfortunately, there is no good way to choose the parameter L_c, so we use the multiplicative depth L as a reference. First we choose a L_{init} which is a little larger than the estimated $L_c\,(\approx 2\lceil \frac{L}{2} \rceil + 1)$, then perform the calculation. After that, we use the library function `Ctxt::findBaseLevel()` to get the current level L_0 of the ciphertext. Then we set the $L_c = L_{init} - L_0 + 1$. Once λ and L_c are determined, we can compute the least integer m satisfying the Eq. (1).

Performance. We test our implementation, which is single-threaded, on a PC with a Intel Core i7 4790 CPU at 3.60 GHz and 8GB RAM. Table 1 provides the information about the running time of different arithmetic operations. In Table 1, the #bits column represents the current circuit supports #bits encrypted integer arithmetic, m is decided by the security parameter and L_c as in Eq. (1), the #slots column is the number of slots, and the timing is counted in seconds. Since in subtraction, multiplication and division we need addition as the fundamental module, we point out which adder we use to do the experiments in the circuit

Table 1. Performance of FHE Binary Arithmetic

Arithmetic	Circuit	#bits	m	#slots	L_c	time (s)
Addition	RCA	16	14351	504	17	2.16
	CLA	16	7781	150	7	2.53
	CLA	64	13981	600	13	37.69
Subtraction	RCS	16	14351	504	17	2.17
	CLA	16	7781	150	7	2.52
	CLA	64	13981	600	13	37.16
Multiplication	RCA	8	8191	630	9	4.62
	RCA	16	14351	504	17	46.32
Division	RCA	4	18631	720	21	14.63

column. Note that for the same L_c, we obtain the same integer m as in [5], although the authors claimed that their secuirty parameter was $\lambda = 128$.

From Table 1, it is clear that the RCA adder needs more multiplicative depth than CLA adder. Due to the heavy calculation inside the CLA adder, there is no obvious advantage for 16-bit integers. However, for 64-bit integers, the RCA adder needs $L_c = 64$ and $m = 55831$ which is such a large number that HElib do not have enough resource to continue computing and finally return an error message. In contrast, CLA adder only needs $L_c = 13$ and a 64-bit addition is carried out within 40s. The subtraction basically uses the same running time as addition, since they share the same structure.

Due to our description of multiplication, we know that there are two time-consuming parts in the multiplier. One is to compute middle results, and the other is to sum the middle results. In the experiment for 16-bit mulitplier with RCA adder, the first part takes about 31 s while the second part takes about 15s. Since both parts can be boosted in parallel, the performance of multiplier can be further improved.

Since the division needs the most multiplicative depth, it is the least efficient operation. Only for a 4-bit division, we have to set $L_c = 21$. In Chen and Guang's paper [5], they reported the arithmetic operations over encrypted integers with bits at most 4. For division with 4-bit encrypted integers, their implementation costs about 68s on a machine with 8 Intel Xeon E7-L8867 2.13 GHz processors and 512 GB RAM, while ours only costs about 15s.

At last but not least, thanks to the SIMD operation, our implementation supports integer vector calculation (element-wise computation with vector length at most #slots), since we only use the first slot of every ciphertext during the integer calculation. According to our tests, the cost is the same as that reported in Table 1 since their procedures of computation are identical.

5 Conclusion and Discussion

We presented our HElib-based implementation of homomorphic evaluation of integer arithmetic circuits on encrypted data without bootstrapping. Our implementation features different kind of adder circuits, among which we can choose for different applications. With several optimizations and careful choosing of circuits, our implementation significantly outperforms the implementation in [5].

We note that the latest version of HElib has included bootstrapping [19] and it seems going to support threadsafe mode in the very near future, and hence support parallel computation. With the help of these techniques, it is very hopeful to make above all integer arithmetic operations even faster, including the multiplication with CLA addition. Furthermore, how to design efficient SIMD circuits for integer arithmetic operations is a very interesting topic.

In addition, it would be very meaningful to design and implement FHE schemes for arithmetic operations over ranges larger than the integer ring, for instance, over the fixed or floating point number system. Very recently, the results from [1,7] seem to be good attempts in this area.

Acknowledgments. We would like to thank one of anonymous referees for pointing out us Cheon et al.'s work [8] on encrypted integer addition. The present work was partially supported by Natural Science Foundation of China (11471307, 11501540, 11671377), Chongqing Research Program of Basic Research and Frontier Technology (cstc2015jcyjys40001) and CAS "Light of West China" Program.

References

1. Arita, S., Nakasato, S.: Fully homomorphic encryption for point numbers. Cryptology ePrint Archive, Report 2016/402 (2016)
2. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32009-5_50
3. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012, pp. 309–325. ACM, New York (2012)
4. Brakerski, Z., Vaikuntanathan, V.: Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 505–524. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22792-9_29
5. Chen, Y., Gong, G.: Integer arithmetic over ciphertext and homomorphic data aggregation. In: Proceedings of 2015 IEEE Conference on Communications and Network Security, pp. 628–632. IEEE, Piscataway (2015)
6. Cheon, J.H., Coron, J.-S., Kim, J., Lee, M.S., Lepoint, T., Tibouchi, M., Yun, A.: Batch fully homomorphic encryption over the integers. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 315–335. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38348-9_20
7. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Floating-point homomorphic encryption. Cryptology ePrint Archive, Report 2016/421 (2016)

8. Cheon, J.H., Kim, M., Kim, M.: Search-and-compute on encrypted data. In: Brenner, M., Christin, N., Johnson, B., Rohloff, K. (eds.) FC 2015. LNCS, vol. 8976, pp. 142–159. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48051-9_11
9. Coron, J.-S., Mandal, A., Naccache, D., Tibouchi, M.: Fully homomorphic encryption over the integers with shorter public keys. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 487–504. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22792-9_28
10. Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13190-5_2
11. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University, Stanford (2009)
12. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) STOC 2009, pp. 169–178. ACM, New York (2009)
13. Gentry, C., Halevi, S., Peikert, C., Smart, N.P.: Field switching in BGV-style homomorphic encryption. J. Comput. Secur. **21**(5), 663–684 (2013)
14. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 465–482. Springer, Heidelberg (2012). doi:10.1007/978-3-642-29011-4_28
15. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 850–867. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32009-5_49
16. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 75–92. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40041-4_5
17. Halevi, S., Shoup, V.: HElib: an implementation of homomorphic encryption. https://github.com/shaih/HElib. Accessed June 2016
18. Halevi, S., Shoup, V.: Design and implementation of a homomorphic encryption library. https://github.com/shaih/HElib
19. Halevi, S., Shoup, V.: Bootstrapping for `HElib`. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 641–670. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46800-5_25
20. Kolesnikov, V., Sadeghi, A.-R., Schneider, T.: Improved garbled circuit building blocks and applications to auctions and computing minima. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 1–20. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10433-6_1
21. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 1–23. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13190-5_1
22. Computational Algebra Group, University of Sydney: Magma computational algebra system. http://magma.maths.usyd.edu.au/magma/
23. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Cachin, C., Ristenpart, T. (eds.) CCSW 2011, pp. 113–124. ACM, New York (2011)
24. Ofman, Y.P.: On the algorithmic complexity of discrete functions. Soviet Physics Doklady **7**(7), 589–591 (1963). Translated from Doklady Akademii Nauk SSSR 145(1), 48–51 (1962)
25. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) STOC 2005, pp. 84–93. ACM, New York (2005)

26. Rivest, R., Adleman, L., Dertouzos, M.: On data banks and privacy homomorphisms. In: DeMillo, R.A., Dobkin, D.P., Jones, A.K., Lipton, R.J. (eds.) Foundations of Secure Computation, pp. 165–179. Academic Press, Atlanta (1978)
27. Shoup, V.: NTL: a library for doing number theory. http://shoup.net/ntl/. Accessed June 2016
28. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. Des. Codes Crypt. **71**(1), 57–81 (2014)
29. Wu, D., Haven, J.: Using homomorphic encryption for large scale statistical analysis (2012). https://crypto.stanford.edu/people/dwu4/FHE-SI_Report.pdf