Article

An Efficient BGV-type Encryption Scheme for IoT Systems

Wei Yuan ^{1,2} and Han Gao ^{3,*}

¹ College of Computer Science and Technology, Jilin University, Changchun 130012, China; yuanwei1@126.com

² College of Mathematics, Jilin University, Changchun 130012, China

³ School of Information Engineering, China University of Geosciences, Beijing 100000, China

* Correspondence: 1004171212@cugb.edu.cn

Received: 24 June 2020; Accepted: 6 August 2020; Published: 19 August 2020

Abstract: Internet of Thing (IoT) systems usually have less storage and computing power than desktop systems. This paper proposes an efficient BGV-type homomorphic encryption scheme in order fit for secure computing on IoT system. Our scheme reduces the storage space for switch keys and ciphertext evaluation time comparing with previous BGV-type cryptosystems. Specifically, the switch key in homomorphic computations can be a constant but no longer one for each level. Moreover, the product of two ciphertexts can be at the same sublayer as them and the multiplication operations can be repeated between two sublayers. As a result, the multiplication times will not be limited by L in an L -level circuit and, thus, the ciphertext evaluation time will decrease significantly. We implement the scheme with the C language. The performance test shows that the efficiency of the improved scheme is better than Helib in same configurations.

Keywords: BGV; homomorphic encryption; IoT security

1. Introduction

The Internet of Thing (IoT) systems have been widely used in people's daily life [1,2]. In an IoT system, one can control his/her remote devices [3] to finish expected tasks. That requires the devices to have secure computing power [4]. Homomorphic encryption allows a device to perform arbitrary computations on encrypted data without user secret key. When compared with non-homomorphic encryptions, it removes the trust on the device and makes us ease to use the computing power of the device. Hence, homomorphic encryption has broad application prospects [5–11]. In a breakthrough work [12], Gentry demonstrated that fully homomorphic encryption was theoretically possible based on ideal lattices. In Gentry's work, the ciphertext can be regarded as some "noise" attached to the plaintext, and the noise will increase along with evaluation times. When the noise exceeds a threshold value, which makes decryption fail, a bootstrapping program can refresh the noise introduced by evaluations and make the ciphertext as a new one generated by the encryption algorithm. However, the efficiency of the scheme is far from practical application.

Following Gentry's work, many researchers tried to improve the performance of homomorphic encryption. Based on the learning with error assumption [13,14], Brakerski and Vaikuntanathan [15,16] (BV) presented a dimension-modulus reduction technique and constructed a leveled homomorphic encryption without the requirement of a bootstrappable encryption. Brakerski, Gentry, and Vaikuntanathan [17] (BGV) then dramatically improved the performance of the BV-type homomorphic encryption. Moreover, Gentry and Halevi and Smart reduced the size of public keys and ciphertexts [18] of the BGV-type scheme at the cost of increasing the probability of key recovery attack. Subsequently, they optimized the execution times of fast Fourier transform (FFT) and Chinese reminder theorem (CRT) by reducing the

number of times to convert polynomials between coefficient and evaluation representations [16,17,19]. Brakerski [20] further reduced the ciphertext noise growth rate of multiplication operations. Halevi and Shoup [21–23] continuously and roundly optimized BGV-type scheme in the algorithm implementation aspect. Castryck, Iliashenko, and Vercauteren [24] improved parallel performance to packet more data within one instruction. Öztürk, Doröz, Savas and Sunar [25] designed a custom hardware accelerator to improve the performance of polynomial multiplier. Eric, Chris, and Chad [26] provided a compiler for homomorphic encryption and let the programmer with no special knowledge of homomorphic encryption easily use it. In addition, many efforts [22,27–34] have been paid for improving the performance of bootstrapping. As Chen, Chillotti, and Song [34] declared, the bootstrapping time per plaintext slot had been reduced to about 1 s.

This paper primarily focuses on improving the storage and computation efficiency of BGV-type cryptosystem. In previous works, we find that only [19] mentioned this problem. However, the main goal of [19] is to reduce the representation transformation times of FFT and CRT, but not reduce the size of public key, simplify the multiplication algorithm, or further control the noise. Hence, the improvement directions presented in [19] are different from our work.

Organization. In Section 2, we introduce the plaintext and ciphertext encoding manners and the principle of BGV. Afterwards, in Section 3, we briefly introduce a BGV-type scheme and then our modifications on BGV. In Section 4, our scheme is given and a proof of concept implementation with the C language and a performance test are provided in Section 5. Section 6 concludes the paper. At last, all of the source codes of our proof of concept implementation are given in Appendix A and the codes of benchmark implementations of Helib and RSA are given in Appendixes B and C. These appendices may help reviews.

2. Background Knowledge

2.1. Plaintext Encoding

Based on the underlying algebra, we know that a cyclotomic polynomial can be factorized into l irreducible polynomials as $\Phi_m(x) = \prod_{i=1}^l F_i(x) \bmod p$. Hence, the quotient ring $R_p = \mathbb{Z}_p[x]/\Phi_m(x) \cong \mathbb{Z}_p[x]/F_1(x) \otimes \cdots \otimes \mathbb{Z}_p[x]/F_l(x)$. Let d be the order of (m, p) , such that $p^d = 1 \bmod m$. We have $l = \phi(m)/d$. Here, $\phi()$ is Euler function, d is the degree of each $F_i(x)$, and l is the number of $F_i(x)$, which represents the number of plaintext slots. Subsequently, a message can be encoded with the coefficients and modulus of a polynomial ring as the plaintext.

As a concrete example, assume $p = 17$ and $m = 8$. We have $(p = 17)^{d=1} = 1 \bmod (m = 8)$ and thus $l = (\phi(m) = 4)/(d = 1) = 4$. As a result, $\Phi_8(x) = x^4 + 1 = (x - 2)(x - 2^3)(x - 2^5)(x - 2^7) \bmod 17$. For a message $5491 = 17^3 + 2 \times 17^2 + 3 \times 17 + 4$, we can express it as $x^3 + 2x^2 + 3x + 4 \bmod (x^4 + 1, 17)$. Because

$$\begin{aligned} x^3 + 2x^2 + 3x + 4 &\cong 9 \bmod (x - 2, 17) \\ &\cong 5 \bmod (x - 2^3, 17) \\ &\cong 15 \bmod (x - 2^5, 17) \\ &\cong 4 \bmod (x - 2^7, 17) \end{aligned}$$

$x^3 + 2x^2 + 3x + 4$ can be further represented by a vector (9,5,15,4). Next, we have $(x^3 + 2x^2 + 3x + 4) \cdot (x^3 + 2x^2 + 3x + 4) \bmod x^4 + 1 = 3x^3 + 7x^2 + 3x + 6$. Meanwhile, $(9,5,15,4) \cdot (9,5,15,4) \bmod 17 = (81,25,225,16) \bmod 17 = (13,8,4,16)$. We may verify that $3x^3 + 7x^2 + 3x + 6$ can be represented by (13,8,4,16). Similarly, $(x^3 + 2x^2 + 3x + 4) + (x^3 + 2x^2 + 3x + 4) \bmod x^4 + 1 = (2x^3 + 4x^2 + 6x + 8) \cong (1,10,13,8) = (9,5,15,4) + (9,5,15,4) \bmod 17$. Hence, the encoded vector supports multiplication and addition simultaneously. According to CRT on the polynomial $\Phi_m(x)$ and modulus p , a vector can also recover the polynomial and, thus, the plaintext.

2.2. Ciphertext Encoding

Similar to the plaintext case, the ciphertext space in BGV [17] is defined by a quotient ring $R_q = \mathbb{Z}_q[x]/\Phi_m(x)$, where q is far greater than p . Naturally, we wonder what is the detailed relationship between p and q . Reference [19] indicates that q can be the product of a series of prime numbers $p_0 \cdots p_{L-1}$. L is the depth of the circuit in a leveled homomorphic encryption scheme and a new ciphertext is with respect to q_{L-1} . Each time two ciphertexts multiply, the new ciphertext of the product is mapped from R_{q_i} to $R_{q_{i+1}}$, where $q_i = \prod_{j=0}^i p_j$ for $i = 0, \dots, L-1$. ([19], Section 3.2) introduces the procedure for mapping an element in R_{q_i} to the corresponding element in $R_{q_{i+1}}$ in the modulus-reduction process. Hence, L is the upper bound of the multiplication and addition times. As an optimization of keeping ciphertexts and key switch matrices in evaluation (CRT) representation, Ref. [19] also suggests that $p_i \equiv 1 \pmod{m}$ for each $i = 0, \dots, L-1$.

As a toy example of ciphertext encoding, assume $p_0 = 11$, $p_1 = 23$, and $p_2 = 67$. We have $q_0 = p_0 = 11$, $q_1 = p_0 p_1 = 11 \times 23 = 253$, and $q_2 = p_0 p_1 p_2 = 16951$. Let 10 be the ciphertext $ct^{(1)} \in R_{q_1}$, we can compute its equivalent ciphertext $ct^{(2)} = (ct^{(1)} \times p_2) \in R_{q_2} = 10 \times 67 \pmod{16951} = 670$. To directly verify the correctness of the reduction, we may map $ct^{(2)}$ to $ct^{(1)}$ by multiplying the inverse mapping element rp_1 from R_{q_2} to R_{q_1} (p_2 and rp_1 are mutually inversed when mapping elements between R_{q_1} and R_{q_2}). Here, we directly give that element $rp_1 = 34$ in the toy example. Please see ([19], Section 3.2) to find the way to compute it. Then we may test $670 \times 34 \pmod{253} = 10$.

2.3. Security Assumption of BGV

The security of BGV-type cryptosystems is based on the learning with errors over rings (RLWE) assumption [13,35]. The $\text{RLWE}(\lambda, q, \chi)$ assumption is to distinguish two distributions $(a, a \cdot s + \epsilon)$ and (a, b) , where a , s , and b are randomly selected from R_q and ϵ is selected from χ referencing security parameter λ . This assumption has been proved hard over ideal lattices in [35].

2.4. Principle of BGV

Let p, q be two prime numbers and $p < q$. R_p and R_q are groups formed by p and q according to algebra. For any number $m \in R_p$, if $0 < m + \epsilon p < q$, we have $[(m + \epsilon p) \pmod{q}] \pmod{p} = m$. For example, let $p = 5$, $q = 19$, and $m = 3$. If $\epsilon = 3$, $m + \epsilon p = 18$. $18 \pmod{19} \pmod{5} = 3$. Otherwise, if $\epsilon = 4$, $m + \epsilon p = 23$. $23 \pmod{19} \pmod{5} = 4$. If $\epsilon = -1$, $m + \epsilon p = -2$. $-2 \pmod{19} \pmod{5} = 2$. Let m be the plaintext and $(m + \epsilon p)$ be the ciphertext. If the operations on the ciphertext do not make the intermediate result exceeding the modulus q , all the errors can be eliminated by moding q and then moding p . That is because the error term is multiplied by p . In our opinion, the BGV and BGV type schemes construct cryptosystems based on this principle and try their best to control the error growth in computing.

3. Introduction of BGV

In original BGV [17], public key and switch keys are matrices. ([18], Appendix D.2)

indicates those keys can also be vectors over polynomial ring. Subsequently, the key size can be reduced. In this section, we describe BGV in vector form. As all the other encryption schemes, BGV includes four algorithms, Setup, Keygen, Encrypt, and Decrypt as the basic encryption-decryption part. Moreover, it includes Add and Multiply to perform homomorphic calculations. These algorithms are as follows.

Setup: given a security parameter λ and level L , the setup algorithm generates L large prime numbers $q_0, \dots, q_{L-1}$, where $q_0 < \dots < q_{L-1}$ and selects the discrete Gaussian distribution χ as the error distribution. It defines the plaintext space by selecting m and p and computing l and d . $q_0, \dots, q_{L-1}, \chi, p$, and l are public parameters.

KeyGen: given the public parameters, first, the key generation algorithm selects a random vector s as the secret key. Then, it computes $b = -(a \cdot s + p \cdot \epsilon) \pmod{q_{L-1}}$, where a is a random element in $R_{q_{L-1}}$, ϵ is sampled from χ , and p is the plaintext modulus, and let (a, b) be the public key. Next, it computes

$b_i = -(a_i \cdot s + p \cdot \epsilon_i - t_i \cdot s^2) \bmod t_i \cdot q_i$, where a_i a random element in R_{q_i} , t_i is an integer, and ϵ_i is sampled from χ , and finally let $(a_0, b_0, t_0, 0), \dots, (a_{L-1}, b_{L-1}, t_{L-1}, L-1)$ be the switch key.

Encrypt: given a plaintext m , the encryption algorithm randomly selects a vector v , where each $v_i \in \{0, \pm 1\}$ with probability $\{\frac{1}{2}, \frac{1}{4}\}$, samples e_0 and e_1 from χ , and computes $c_0 = b \cdot v + p \cdot e_0 + m \bmod q_{L-1}$, $c_1 = a \cdot v + p \cdot e_1 \bmod q_L$. $ct = (c_0, c_1, L-1)$ is the initial ciphertext.

Decrypt: given a ciphertext $ct = (c_0, c_1, i)$ and the secret key s , the decryption algorithm computes $m \leftarrow c_0 + c_1 \cdot s \bmod q_i \bmod p$.

Mul: given two ciphertexts $ct' = (c'_0, c'_1, i)$ and $ct'' = (c''_0, c''_1, i)$ under same secret key, the multiplication algorithm first computes $d_0 = c'_0 \cdot c''_0$, $d_1 = c'_0 \cdot c''_1 + c'_1 \cdot c''_0$, and $d_2 = c'_1 \cdot c''_1$. Then it compresses (d_0, d_1, d_2) to $ct^* = (c_0^*, c_1^*)$, where $c_0^* = t_i d_0 + b_i \cdot d_2 \bmod t_i q_i$ and $c_1^* = t_i d_1 + a_i \cdot d_2 \bmod t_i q_i$, with the switch key (a_i, b_i, t_i, i) . Next, $ct^* \in R_{t_i q_i}$ is first converted into $ct^* \in R_{q_i}$ and then the output ciphertext $ct \in R_{q_{i-1}}$ by SwitchModulu.

Add: given two ciphertexts $ct' = (c'_0, c'_1, i)$ and $ct'' = (c''_0, c''_1, i)$, the addition algorithm computes the sum of the two ciphertext as $ct^* = (c'_0 + c''_0 \bmod q_i, c'_1 + c''_1 \bmod q_i)$, and then converts $ct^* \in R_{q_i}$ to $ct \in R_{q_{i-1}}$ as the output.

SwitchModulu: given a ciphertext $ct = (c_0, c_1, i) \bmod q_i$, two modulus q_i and q_j where $i > j$, the modulu switch algorithm computes the modulo inverse element $rp_j = \frac{q_j}{q_i}$ in q_j , and output the new ciphertext $ct' = (c'_0, c'_1, j)$, where $c'_1 = c_1 \cdot rp_j \bmod q_j$ and $c'_2 = c_2 \cdot rp_j \bmod q_j$.

Correctness. first, given $ct = (c_0, c_1, L-1)$ and corresponding secret key s , $\langle ct, s \rangle = c_0 + c_1 \cdot s = \{b \cdot v + p \cdot e_0 + m + (a \cdot v + p \cdot e_1) \cdot s\} \bmod q_{L-1} = \{m - a \cdot s \cdot v - p \cdot \epsilon \cdot v + p \cdot e_0 + a \cdot v \cdot s + p \cdot e_1 \cdot s\} \bmod q_{L-1} = \{m - p \cdot \epsilon v + p(e_0 + e_1 s)\} \bmod q_{L-1}$. Let error be $-p \cdot \epsilon v + p(e_0 + e_1 s)$. If $0 < \text{error} < q_{L-1}$, $\langle ct, s \rangle \bmod p = m$.

Second, given $ct = (c_0, c_1, i-1)$ and secret key s , where ct is the product of $ct_1 = (c'_0, c'_1, i)$ and $ct_2 = (c''_0, c''_1, i)$ outputted by Mul, let rt_i be the inverse element of t_i that maps element from $R_{t_i q_i}$ to R_{q_i} , we have $\langle ct, s \rangle = c_0 + s \cdot c_1 = rt_i(t_i d_0 + b_i d_2) + s[rt_i(t_i d_1 + a_i d_2)] = d_0 + s d_1 + rt_i(b_i + a_i s) d_2 = d_0 + s d_1 + rt_i(t_i s^2 - p \epsilon_i) d_2 = d_0 + s d_1 + s^2 d_2 - rt_i p \epsilon_i d_2$. Because

$$d_0 + d_1 \cdot s + d_2 \cdot s^2 = \{c'_0 \cdot c''_0 + (c'_0 \cdot c''_1 + c'_1 \cdot c''_0)s + c'_1 \cdot c''_1 s^2\} = \{m' m'' + v' v'' (p \epsilon)^2 + p \epsilon v'' [m' + p(e'_0 + e'_1 s)] + p \epsilon v' [m'' + p(e''_0 + e''_1 s)] + p^2 (e'_0 + e'_1 s)(e''_0 + e''_1 s) + p(m' + m'') \cdot (e'_0 + e'_1 s)\} \bmod q_{i-1}$$

and $rt_i p \epsilon_i d_2 = rt_i p \epsilon_i c'_1 \cdot c''_1 = rt_i p \epsilon_i (a v' + p e'_1)(a v'' + p e''_1) \bmod q_{i-1}$

Let error represents $v' v'' (p \epsilon)^2 + p \epsilon v'' [m' + p(e'_0 + e'_1 s)] + p \epsilon v' [m'' + p(e''_0 + e''_1 s)] + p^2 (e'_0 + e'_1 s)(e''_0 + e''_1 s) + p(m' + m'')(e'_0 + e'_1 s) - rt_i p \epsilon_i (a v' + p e'_1)(a v'' + p e''_1)$. If $0 < \text{error} < q_{i-1}$, $\langle ct, s \rangle \bmod p = m' m''$.

Third, given $ct = (c_0, c_1, i-1)$ and secret key s , where ct is the sum of $ct_1 = (c'_0, c'_1, i)$ and $ct_2 = (c''_0, c''_1, i)$ outputted by Add, we have $\langle ct, s \rangle = c_0 + s \cdot c_1 = (c'_0 + c''_0) + s(c'_1 + c''_1) = \{m' + p(e'_0 + \epsilon - s e'_1) + m'' + p(e''_0 + \epsilon - s e''_1)\} \bmod q_{i-1}$. Let error be $p(e'_0 + \epsilon - s e'_1) + p(e''_0 + \epsilon - s e''_1)$. If $0 < \text{error} < q_{i-1}$, $\langle ct, s \rangle \bmod p = m_1 + m_2$.

By selecting small parameters, like v, ϵ, e_0, e_1 , etc., the components of a ciphertext may be far less than modulu. However, with the increasement of multiplication and addition times, errors will eventually exceed the modulu. The function of SwitchModulu is to continually scale cumulative errors to $\frac{q_{i-1}}{q_i}$ by map components from R_{q_i} to $R_{q_{i-1}}$. That is why the maximum multiplication and addition times are limited by the predefined parameters L . In the multiplication algorithm, the direct product of two ciphertexts is a new ciphertext with three elements, key switch is to recompress it to two elements and the errors are still the same.

3.1. Modifications on BGV-type Cryptosystems

Based on the BGV-type Cryptosystems, we make following changes. We may observe the decryption processes after the modifications.

First, let $q_0 = p_0 = p$, $p_i = 1 \bmod p$, and $q_i = \prod_{j=0}^i p_j$ for $i = 1, \dots, L-1$ in the setup algorithm. Second, replace t_i with p_{i+1} in switch key generation and key switch algorithms, and then remove the SwitchModulu operations that map elements from $R_{p_{i+1} \cdot q_i}$ to R_{q_i} . That is to say, each $b_i = -(a_i \cdot s +$

$p \cdot \epsilon_i - p_{i+1} \cdot s^2) \bmod p_{i+1} \cdot q_i (= q_{i+1})$. Third, move the SwitchModulu operation in Mul from the last step to the first step.

Because the encryption, decryption, and addition algorithms do not change, we focus on the multiplication algorithm. Given two ciphertexts $ct_1^{(i)} = (c_0^{(1)}, c_1^{(1)}, i)$ and $ct_2^{(i)} = (c_0^{(2)}, c_1^{(2)}, i)$ of the plaintexts m_1 and m_2 , the multiplication algorithm first SwitchModulu them to $ct_1^{(i-1)} = (c_0^{(1)}, c_1^{(1)}, i-1)$ and $ct_2^{(i-1)} = (c_0^{(2)}, c_1^{(2)}, i-1)$ and then SwitchKey the product of $ct_1^{(i-1)}$ and $ct_2^{(i-1)}$ to $ct^{(i)} = (c_0, c_1, i)$ as below: $c_0 \leftarrow \{p_i c_0^{(1)} c_0^{(2)} + b_i c_1^{(1)} c_1^{(2)}\} \bmod q_i$ and $c_1 \leftarrow \{p_i (c_0^{(1)} c_1^{(2)} + c_0^{(2)} c_1^{(1)}) + a_i c_1^{(1)} c_1^{(2)}\} \bmod q_i$

Given the secret key s , we have

$$\begin{aligned} \langle ct, s \rangle &= c_0 + s \cdot c_1 = \{p_i c_0^{(1)} c_0^{(2)} + p_i s c_0^{(1)} c_1^{(2)} + p_i s c_0^{(2)} c_1^{(1)} + p_i s^2 c_1^{(1)} c_1^{(2)} - p \epsilon_i c_1^{(1)} c_1^{(2)}\} \bmod q_i \\ &= p_i (c_0^{(1)} + s c_1^{(1)}) (c_0^{(2)} + s c_1^{(2)}) \bmod q_i - p \epsilon_i c_1^{(1)} c_1^{(2)} \bmod q_i \end{aligned}$$

If the error terms are less than q_i , we have $\langle ct, s \rangle \bmod p = m_1 m_2$. Obviously, the correctness condition is the same as the BGV scheme.

Observation 1. Next, we replace the normal switch key epk_i with another sublayer's switch key epk_j in Mul and observe the decryption process. In this condition, the ciphertext outputted by Mul is as follows: $c_0 \leftarrow \{p_i c_0^{(1)} c_0^{(2)} + b_j c_1^{(1)} c_1^{(2)}\} \bmod q_i$ and $c_1 \leftarrow \{p_i (c_0^{(1)} c_1^{(2)} + c_0^{(2)} c_1^{(1)}) + a_j c_1^{(1)} c_1^{(2)}\} \bmod q_i$.

Given the secret key s , we have

$$\begin{aligned} \langle ct, s \rangle &= c_0 + s \cdot c_1 = \{p_i c_0^{(1)} c_0^{(2)} + p_i s c_0^{(1)} c_1^{(2)} + p_i s c_0^{(2)} c_1^{(1)} + p_j s^2 c_1^{(1)} c_1^{(2)} - p \epsilon_j c_1^{(1)} c_1^{(2)}\} \bmod q_i \\ &= p_i \bmod q_i \cdot c_0^{(1)} c_0^{(2)} \bmod q_i + p_i \bmod q_i \cdot s c_0^{(1)} c_1^{(2)} \bmod q_i + p_i \bmod q_i \cdot s c_0^{(2)} c_1^{(1)} \bmod q_i + p_j \bmod q_i \\ &\quad \cdot s^2 c_1^{(1)} c_1^{(2)} \bmod q_i - p \epsilon_j c_1^{(1)} c_1^{(2)} \bmod q_i \\ &\neq p_i (c_0^{(1)} + s c_1^{(1)}) (c_0^{(2)} + s c_1^{(2)}) \bmod q_i - p \epsilon_i c_1^{(1)} c_1^{(2)} \bmod q_i \end{aligned}$$

However, if $p_i = 1 \bmod p$ and $q_i = \prod_{j=0}^i p_j$ for $i = 1, \dots, L$, we have $p_i \bmod q_i \bmod p = p_j \bmod q_i \bmod p = 1$. Thus, $\langle ct, s \rangle \bmod p = (c_0^{(1)} + s c_1^{(1)}) (c_0^{(2)} + s c_1^{(2)}) \bmod q_i \bmod p - p \epsilon_i c_1^{(1)} c_1^{(2)} \bmod q_i \bmod p$. The correctness condition is the same as BGV. If the error terms are less than q_i , we also have $\langle ct, s \rangle \bmod p = m_1 m_2$.

As a result, we do not need to generate a switch key for each level after the modification. Only one l -dimension switch key is enough for multiplication. Thus, the total size of public keys can be reduced from linear to constant.

Observation 2. Two ciphertexts and their product are at the same sublayer after the modifications. The reason that each b_i should modulu $t_i \cdot q_i$, but not q_i in BGV is the SwitchKey operation in Mul will lift elements from R_{q_i} to $R_{t_i \cdot q_i}$. After replacing t_i with p_{i+1} , the elements will be lifted from R_{q_i} to $R_{q_{i+1}}$. The reason we move the SwitchModulu operation in Mul from the last step to the first step is to satisfy this requirement. Because all of the input elements belong to R_{q_i} in Mul, the accumulative errors are far less than $R_{q_{i+1}}$. Thus, the decryption can succeed and another SwitchModulu operation can be omitted.

The advantage of this change is all of the multiplication operations can be repeatedly done between R_{q_i} and $R_{q_{i+1}}$. Subsequently, the multiplication times will not be limited by L in an L -level circuit and, thus, naturally become unlimited without bootstrapping. That is to say, L can be very small. We may use small number of large primes to replace large number of small primes. As a matter of fact, the product of several large primes is far less than the product of many small primes. For example, we know $100^2 \ll 2^{100}$. As a result, the ciphertext evaluation time will decrease significantly.

4. Scheme Description

In this section, we describe a variant BGV-type scheme under our modifications. Because the scheme is to optimize the ciphertext evaluation processes. The description of encoding a message to a vector as plaintext is ignored. Moreover, the basic secret key generation, public key encryption, encryption, and decryption algorithms are the same as known schemes [5,19]. The details are as follows:

Setup: given a security parameter λ , the setup algorithm selects a prime number $p(\lambda)$ and a number $m(\lambda)$ and computes corresponding parameters d and l to define the plaintext space. Afterwards, it selects a series of prime number $p_0(\lambda), \dots, p_{L-1}(\lambda)$, where $p_0 = p, p_1(\lambda) = \dots = p_{L-1}(\lambda) = 1 \bmod p(\lambda)$, and computes $q_i = \prod_{j=0}^i p_j$, for $i = 0, \dots, L-1$. All of the ciphertexts and keys are defined in $R_{q_{L-1}}$. The public parameters include all above parameters.

SecretKeyGen: given the public parameters, the secret key generation algorithm selects an l -dimension vector s . Each element in s is chosen from a noise distribution χ . Output $SK = s$.

PublicKeyGen: given the public parameters and an l -dimension vector s , the public key generation algorithm selects an l -dimension vector a from $R_{q_{L-1}}$, an l -dimension error vector e from distribution χ , and computes an l -dimension vector $b = a \cdot s + p \cdot e \bmod q_{L-1}$. Output $PK = (a, b)$.

SwitchKeyGen: given the public parameters and an l -dimension vector s , the switch key generation algorithm selects an l -dimension vector a uniformly from $R_{q_{L-1}}$, an l -dimension error vector e from distribution χ , and computes an l -dimension vector $b = a \cdot s + p \cdot e - s^2 \cdot (p \cdot r + 1) \bmod q_{L-1}$, where r is a random number. Output $EPK = (a, b)$.

Encrypt: given the public parameters, a public key $PK = (a, b)$ and a message $m \in R_p^l$, the encryption algorithm selects an l -dimension vector v . Each of its element is chosen from $\{-1, 0, 1\}$ with probability of $(\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$. Then it selects two l -dimension error vectors e_0 and e_1 , which their elements are sampled from the discrete Gaussian distribution with variance σ . The ciphertext ct consists of two elements $c_0, c_1 \in R_{q_{L-1}}$ and the depth is $L-1$. $c_0 = b \cdot v + p \cdot e_0 + m \bmod q_{L-1}$ and $c_1 = a \cdot v + p \cdot e_1 \bmod q_{L-1}$. $ct = (c_0, c_1, L-1)$ is outputted.

Decrypt: given the public parameters, a secret key $SK = s$ and a ciphertext $ct = (c_0, c_1, i)$, the decryption algorithm computes $m \leftarrow ((c_0 - s \cdot c_1) \bmod q_i) \bmod p$.

SwitchModulu: given a ciphertext $ct = (c_0, c_1, i) \bmod q_i$, two modulus (i, q_i) and (j, q_j) where $i > j$, the modulu switch algorithm computes the modulo inverse element $rp_j = \frac{q_i}{q_j}$ in q_j , and output the new ciphertext $ct' = (c'_0, c'_1, j)$, where $c'_1 = c_1 \cdot rp_j \bmod q_j$ and $c'_2 = c_2 \cdot rp_j \bmod q_j$. Usually, $i = j + 1$ is enough if L is just fit for security requirements. The modulo inverse element rp_j can be computed by the extended Euclid algorithm (see `ex_gcd` in Appendix A.2).

CTLinearization: given two ciphertexts $ct_1 = (c_0^{(1)}, c_1^{(1)}, i)$ and $ct_2 = (c_0^{(2)}, c_1^{(2)}, i)$, the ciphertext relinearization algorithm outputs the three-dimension ciphertext $d = (d_0, d_1, d_2, i + 1) = (ct_1 \otimes ct_2, i + 1) = \{(c_0^{(1)} \cdot c_0^{(2)}), (c_0^{(1)} \cdot c_1^{(2)} + c_0^{(2)} \cdot c_1^{(1)}), (c_1^{(1)} \cdot c_1^{(2)}), i + 1\}$.

SwitchKey: given a three-dimension ciphertext $d = (d_0, d_1, d_2, i)$ and a switch key $EPK = (a, b)$, the key switch algorithm computes $c_0 = p_i \cdot d_0 - b \cdot d_2 \bmod q_i$ and $c_1 = p_i \cdot d_1 - a \cdot d_2 \bmod q_i$ and outputs $ct = (c_0, c_1, i)$.

Mul: given two ciphertexts $ct_1 = (c_0^{(1)}, c_1^{(1)}, i)$ and $ct_2 = (c_0^{(2)}, c_1^{(2)}, i)$, the multiplication algorithm invokes $ct_1 \leftarrow \text{SwitchModulu}(ct_1)$ and $ct_2 \leftarrow \text{SwitchModulu}(ct_2)$, and then invokes $d \leftarrow \text{CTLinearization}(ct_1, ct_2)$. Finally, it invokes $ct \leftarrow \text{SwitchKey}(d)$ and outputs ct .

Add: given two ciphertexts $ct_1 = (c_0^{(1)}, c_1^{(1)}, i)$ and $ct_2 = (c_0^{(2)}, c_1^{(2)}, i)$, the addition algorithm computes $c_0 = c_0^{(1)} + c_0^{(2)} \bmod q_i$ and $c_1 = c_1^{(1)} + c_1^{(2)} \bmod q_i$, and then outputs $ct = (c_0, c_1, i)$.

Correctness. Above scheme is nearly same as BGV except that the switch key is fixed and the key switch algorithm does not switch ct to a lower level one at last.

First, given a ciphertext $ct = (c_0, c_1, L-1)$ outputted by **Encrypt**, a secret key $SK = s$, we have $\langle ct, s \rangle \bmod p = c_0 - s \cdot c_1 \bmod p = (b \cdot v + p \cdot e_0 + m - sa \cdot v - sp \cdot e_1) \bmod q_{L-1} \bmod p = m + p(e_0 + e - s \cdot e_1) \bmod q_{L-1} \bmod p = m$.

Second, given a secret key $SK = s$, a ciphertext $ct = (c_0, c_1, i)$ outputted by Mul, where ct is the product of $ct_1 = (c_0^{(1)}, c_1^{(1)}, i)$ and $ct_2 = (c_0^{(2)}, c_1^{(2)}, i)$, we have $\langle ct, s \rangle \bmod p = c_0 - s \cdot c_1 \bmod p$. As the observations in Section 3.1, $c_0 - s \cdot c_1 \bmod p = (c_0^{(1)} - s c_1^{(1)})(c_0^{(2)} - s c_1^{(2)}) - p e c_1^{(1)} c_1^{(2)} \bmod q_i \bmod p = (m_1 + p(e_0^{(1)} + \epsilon - s \cdot e_1^{(1)}))(m_2 + p(e_0^{(2)} + \epsilon - s \cdot e_1^{(2)})) - p e c_1^{(1)} c_1^{(2)} \bmod q_i \bmod p = m_1 m_2$.

Third, given a secret key $SK = s$, a ciphertext $ct = (c_0, c_1, i)$ outputted by Add, where ct is the sum of $ct_1 = (c_0^{(1)}, c_1^{(1)}, i)$ and $ct_2 = (c_0^{(2)}, c_1^{(2)}, i)$, we have $\langle ct, s \rangle \bmod p = c_0 - s \cdot c_1 \bmod p = (c_0^{(1)} + c_0^{(2)}) - s(c_1^{(1)} + c_1^{(2)}) \bmod q_i \bmod p = m_1 + p(e_0^{(1)} + \epsilon - s \cdot e_1^{(1)}) + m_2 + p(e_0^{(2)} + \epsilon - s \cdot e_1^{(2)}) \bmod q_i \bmod p = m_1 + m_2$.

Security. All of our changes are on the homomorphic computation parts, including the switch key generation algorithm and the multiplication algorithm. The basic encryption-decryption function parts, including the secret key generation algorithm, the public key generation algorithm, the encryption algorithm, and the decryption algorithm are the same as known schemes [5,19]. As introduced in the background, the security of BGV-type cryptosystems is based on the RLWE assumption. In our scheme, the public key (a, b) where $b = a \cdot s + p \cdot \epsilon \bmod q_{L-1}$ has similar form as that of the distribution $(a, a \cdot s + \epsilon)$. The difference is p and q_{L-1} are exposed to the adversary in the setup algorithm. However, this does not reveal more information, because the error is provided by ϵ but not p . We may regard the distribution in RLWE assumption as $(a, a \cdot s + p \cdot \epsilon)$ where $p = 1$. In our scheme, p is a prime number referencing the security parameter. Then the forms of the assumption are same. Logically, only modifying the multiplication algorithm will not affect the security of the basic encryption-decryption function. Thus, the security of our scheme does not need to be re-proved.

5. Implementation and Performance Analysis

In this section, we instantiate the above scheme in the range of “int”, which occupies 32 bits. “int” is the most intuitive data type in the C language. It expresses numbers between -2147483648 and 2147483647 . We remove the negative part and let all computation results within $[0, 2147483647]$. Although “int” is too small to prevent attacks, the correctness and performance of above scheme can be verified. In practical applications, large number, like “bigint” in C++ or “biginteger” in Java can easily replace it. In our implementation, the depth L is set to 3. $p = 11$, $m = 5$, $d = 1$, and $l = (\phi(5) = 4)/(d = 1) = 4$. Moreover, $p_0 = 11$, $p_1 = 23 (= 2 \times 11 + 1)$, and $p_2 = 67 (= 6 \times 11 + 1)$. Correspondingly, $q_0 = 11$, $q_1 = 253 (= 11 \times 23)$, $q_2 = 16951 (= 11 \times 23 \times 67)$. The moduli inverse $rp_1 = 34$ can be computed such that $p_2 \cdot rp_1 = 67 \times 34 = 1 \bmod 253$. Detailed implements to find it are given in Appendix A.2 The reason we set $p = 11$ is that the two smallest prime numbers those equal 1 moduli 11 are 23 and 67. Then $16951^2 = 287336401 < 2147483647$ can ensure all of the intermediate computation results do not exceed the representation range of “int”. If we set $p = 13$, the two smallest prime numbers are $p_1 = 53$ and $p_2 = 79$. Then we have $q_1 = 689$, $q_2 = 54431$. Because $54431^2 = 2962733761 > 2147483647$, the decryption may fail at last. Of course, p can be set to smaller prime numbers than 11 in the toy implementation. The initial ciphertext generated by the encryption algorithm, the secret key, the public key, and the switch key generated by the key generation algorithms are all at level 2. To simplify the implementation, all of the random numbers are chosen from the random number generator in the C library but not χ .

Without loss of generality, we simply select a four-dimension vector (1,2,3,4) as the initial plaintext and generate corresponding ciphertext ct_0 . Subsequently, we compute $ct_{i+1} = ct_i \cdot ct_0$ and $ct_{i+2} = ct_{i+1} + ct_0$ for $i = 0, 2, 4, 6, 8$. For example, ct_1 should be (1,4,9,5) and ct_2 should be (2,6,1,9), etc. The result of correctness validation is in Figure 1.

The vectors, $pt[0], \dots, pt[10]$, are plaintext multiplication and addition results. The vectors, $dpt[0], \dots, dpt[10]$, are the decryption results of the ciphertext of corresponding plaintext. We can see that $dpt[i]$ s are the same as corresponding $pt[i]$ s.

Next, we test the performance of our scheme. The source code of our implementation is given in Appendix A. It includes three files, `stdafx.h`, `stdafx.cpp`, and `unlimited homomorphic encryption.cpp`.

“stdafx.h” defines all the data structures. “stdafx.cpp” defines all of the functions. The functions are invoked in “unlimited homomorphic encryption.cpp”. Appendix B lists the source code of an implementation based on the Helib library. The implementation uses similar parameters as ours. Figure 2 lists their executing time. Our test machine is a ThinkPad X1 laptop with Intel(R) Core(TM)i7-8750H CPU @2.2GHz 8 cores Visual Studio 2005, Microsoft, Redmond, WS, USA).

```

#pragma once
#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define L 4 //length of the ciphertext
#define L 3 //depth of the ciphertext
#define err 10 //the range of the error
#define pt_g 11 //For int, if pt_g is less than 2, it is less than 2
#define test_time 10 //The test time

typedef struct
{
    int p[L];
    int q[L];
    int rp[L];
    PubPara;
}

typedef struct
{
    int s[L];
} SecKey;

typedef struct
{
    int a[L];
    int b[L];
}

int main()
{
    //Initialization
    p[0] = 11, q[0] = 11, rp[0] = 1
    p[1] = 22, q[1] = 253, rp[1] = 1
    p[2] = 67, q[2] = 16951, rp[2] = 34
    sk = (41, 1516, 6334, 9549)
    pk = a: (45, 7247, 12533, 14590), b: (1152, 11488, 15227, 2215)
    epk = a: (48, 9091, 751, 12168), b: (11144, 10612, 49, 1457)

    pt[0] = (1, 2, 3, 4)
    dpt[0] = (1, 2, 3, 4)
    pt[1] = (1, 4, 9, 5)
    dpt[1] = (1, 4, 9, 5)
    pt[2] = (2, 6, 1, 9)
    dpt[2] = (2, 6, 1, 9)
    pt[3] = (2, 1, 3, 3)
    dpt[3] = (2, 1, 3, 3)
    pt[4] = (3, 3, 6, 7)
    dpt[4] = (3, 3, 6, 7)
    pt[5] = (3, 6, 7, 6)
    dpt[5] = (3, 6, 7, 6)
    pt[6] = (4, 8, 10, 10)
    dpt[6] = (4, 8, 10, 10)
    pt[7] = (4, 5, 8, 7)
    dpt[7] = (4, 5, 8, 7)
    pt[8] = (5, 7, 0, 0)
    dpt[8] = (5, 7, 0, 0)
    pt[9] = (5, 3, 0, 0)
    dpt[9] = (5, 3, 0, 0)
    pt[10] = (6, 5, 3, 4)
    dpt[10] = (6, 5, 3, 4)
    pt mul time: 0.000000 seconds
    ct mul time: 0.000000 seconds
  
```

Figure 1. Example of five Multiplications and five Additions.

Figure 2. Efficiency Comparison.

Intuitively, the homomorphic calculation time grows linearly in our scheme, but sub-exponentially in the Helib-based scheme. Meanwhile, the execution time of our scheme is much less than the Helib-based one.

We know that RSA is an encryption scheme only supporting multiplicative homomorphic and it has been applied into many security applications. As a further comparison, we list the Mul operation time of RSA. Since the ciphertext modulu of our scheme is $q_2 = 16951$, we select $p = 127$ and $p = 131$ in

the toy RSA example, such that their product $n = 16637$ is very close to q_2 . The execution time of our scheme is about 100 times of RSA. The source code of the benchmark RSA is in Appendix C.

Finally, Figure 3 lists the storage costs of Helib and ours. The size of public key in Helib is linear growth with maximum multiplications times, while the size of public key in our scheme is a constant.

Figure 3. Storage Comparison.

6. Conclusions

In this paper, we present an efficient BGV-type scheme for IoT systems. Our scheme reduces the storage space and evaluation time, and it is more suitable for secure IoT systems. The performance test of the proof of concept implementation shows that our new scheme is more practical than the current method.

Author Contributions: Conceptualization and methodology, W.Y.; software, H.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by: National Key R&D Plan of China under Grant No. 2017YFA0604500, and by National Sci-Tech Support Plan of China under Grant No. 2014BAH02F00, and by National Natural Science Foundation of China under Grant No. 61701190, and by Youth Science Foundation of Jilin Province of China under Grant No. 20160520011JH & 20180520021JH, and by Youth Sci-Tech Innovation Leader and Team Project of Jilin Province of China under Grant No. 20170519017JH, and by Key Technology Innovation Cooperation Project of Government and University for the whole Industry Demonstration under Grant No. SXGJSF2017-4, and by Key scientific and technological R&D Plan of Jilin Province of China under Grant No. 20180201103GX, Project of Jilin Province Development and Reform Commission No. 2019FGWTZC001.

Acknowledgments: We gratefully acknowledge the helpful comments and suggestions of the reviewers and editors.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Source Code of Our Implementation

This section gives the source code of our scheme. The program is written with C on visual studio 2005. One can create a new project named “unlimited homomorphic encryption” and click finish. The project will create three files `stdafx.h`, `stdafx.cpp` and `unlimited homomorphic encryption.cpp`. Then one can paste the following codes to the three files and then compile and run the project. To easily see the correctness, the default test times are 10 with 5 multiplications and 5 additions. One can change the test times freely.

Appendix A.1. `stdafx.h`

```
#pragma once

#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <tchar.h>
```

```

#include <stdlib.h>
#include <time.h>

#define l 4 //length of the vector
#define L 3 //depth of the circuit
#define err 10 //the range of error
#define pt_q 11 //For int, if either the intermediate number or the final number exceeds 2,147,483,647,
the decrypt will fail. If  $p_0 = 11$ ,  $q_2^2$  is less than 2,147,483,647. If  $p_0 = 13$ ,  $q_2^2$  will exceed 2,147,483,647. So
we set the mode of the plaintext space to 11. It also can be smaller prime number in int space.

#define test_time 10 //The test times of “add” and “mul”. It can be set to larger number, e.g., 100,
1000, 10,000.
typedef struct
{
    int p[L];
    int q[L];
    int rp[L];
} PubPara;
typedef struct
{
    int s[l];
} SecKey;
typedef struct
{
    int a[l];
    int b[l];
} PubKey;
typedef struct
{
    int c0[l];
    int c1[l];
    int depth;
} Ciphertext;
typedef struct
{
    int d0[l];
    int d1[l];
    int d2[l];
    int depth;
} DCiphertext;
typedef struct
{
    int m[l];
} Plaintext;
int mod(int a, int n);
PubPara& Setup(int p);
SecKey& SecKeyGen(PubPara pp);
PubKey& EncryptionPubKeyGen(PubPara pp, SecKey sk);
PubKey& SwitchKeyGen(PubPara pp, SecKey sk);
Ciphertext& Encrypt(PubPara pp, PubKey pk, Plaintext pt, int depth = L-1);

```

```

Plaintext& Decrypt(PubPara pp, SecKey sk, Ciphertext ct);
Ciphertext& add(PubPara pp, Ciphertext ct1, Ciphertext ct2);
Ciphertext& mul(PubPara pp, PubKey epk, Ciphertext ct1, Ciphertext ct2);
Ciphertext& mul2(PubPara pp, PubKey epk, Ciphertext ct1, Ciphertext ct2);

```

Appendix A.2. stdafx.cpp

```
#include "stdafx.h"
```

int mod(int a, int n) //The definition of % in C99 is different from our mod. This gadget reduces any a into 0, 1,..., $n - 1$

```

{
    long b=a;
    if (b<0)
    {
        b = -b;
        b = b%n;
        b = (n-b)%n;
    }
    else
    {
        b = b%n;
    }
    return b;
}
int ex_gcd(int a, int b, int &x, int &y) //Extended Euclid Algorithm
{
    if(b==0)
    {
        x=1;
        y=0;
        return a;
    }
    int r=ex_gcd(b,a%b,x,y);
    int t=x;
    x=y;
    y=t-a/b*y;
    return r;
}

```

int mod_reverse(int a,int n) // $ax=1 \pmod n$. x is the reverse element mod n

```

{
    int d,x,y;
    d=ex_gcd(a,n,x,y);
    if(d==1) return (x%n+n)%n;
    else return -1;
}

```

int isprime (int n) //Test whether n is a prime, the algorithm works but is not efficient

```

{
    int flag=1;
    for(int i=2;i<n;i++)

```

```

        {
            if(n%i==0)
            {
                flag=0;
                break;
            }
        }
        return flag;
    }
}
int proper_prime(int q, int start) //Test whether q is proper and larger than start
{
    int i, result=0;
    for(i=start;;i++)
    {
        if(isprime(i))
        {
            if((i-1)%q==0)
            {
                result=i;
                break;
            }
        }
    }
    if(!result)
    {
        printf("error\n");
    }
    return result;
}
Ciphertext& SwitchModulus(PubPara pp, Ciphertext ct)
{
    Ciphertext sm_ct;
    sm_ct.depth = ct.depth-1;
    //printf("the depth before SM is %d\t",ct.depth);
    for(int i=0;i<l;i++)
    {
        sm_ct.c0[i] = mod(ct.c0[i],pp.q[sm_ct.depth]);
        sm_ct.c0[i] = sm_ct.c0[i]*pp.rp[sm_ct.depth];
        sm_ct.c0[i] = mod(sm_ct.c0[i],pp.q[sm_ct.depth]);
        sm_ct.c1[i] = mod(ct.c1[i],pp.q[sm_ct.depth]);
        sm_ct.c1[i] = sm_ct.c1[i]*pp.rp[sm_ct.depth];
        sm_ct.c1[i] = mod(sm_ct.c1[i],pp.q[sm_ct.depth]);
    }
    //printf("the depth after SM is %d\n",sm_ct.depth);
    return sm_ct;
}
Ciphertext& SwitchModulus(PubPara pp, Ciphertext ct, int depth)
{
    Ciphertext sm_ct;
    //printf("The depth before SM is %d\t",ct.depth);

```

```

    if(ct.depth < depth)
    {
        printf("error, the depth of ct has been less than the input depth\n");
        return sm_ct;
    }
    else
    {
        sm_ct.depth = ct.depth-1;
    }
    while(sm_ct.depth>depth)
    {
        sm_ct.depth--;
        for(int i=0;i<l;i++)
        {
            sm_ct.c0[i] = mod(ct.c0[i],pp.q[sm_ct.depth]);
            sm_ct.c0[i] = sm_ct.c0[i]*pp.rp[sm_ct.depth];
            sm_ct.c0[i] = mod(sm_ct.c0[i],pp.q[sm_ct.depth]);
            sm_ct.c1[i] = mod(ct.c1[i],pp.q[sm_ct.depth]);
            sm_ct.c1[i] = sm_ct.c1[i]*pp.rp[sm_ct.depth];
            sm_ct.c1[i] = mod(sm_ct.c1[i],pp.q[sm_ct.depth]);
        }
    }
    //printf("The depth after SM is %d\n",sm_ct.depth);
    return sm_ct;
}
DCiphertext& CTLinearization(PubPara pp, Ciphertext ct1, Ciphertext ct2)
{
    DCiphertext dct;
    if(ct1.depth != ct2.depth)
    {
        printf("the depth of the two cts are not equal");
        return dct;
    }
    //printf("the depth before CL is %d\t",ct1.depth);
    dct.depth=ct1.depth+1;
    for(int i =0;i<l;i++)
    {
        dct.d0[i]=ct1.c0[i]*ct2.c0[i];
        dct.d0[i]=mod(dct.d0[i],pp.q[dct.depth]);
        dct.d1[i]=ct1.c0[i]*ct2.c1[i]+ct1.c1[i]*ct2.c0[i];
        dct.d1[i]=mod(dct.d1[i],pp.q[dct.depth]);
        dct.d2[i]=-ct1.c1[i]*ct2.c1[i];
        dct.d2[i]=mod(dct.d2[i],pp.q[dct.depth]);
    }
    //printf("the depth after CL is %d\n",dct.depth);
    return dct;
}
Ciphertext& SwitchKey(PubPara pp, PubKey epk, DCiphertext dct)
{
    Ciphertext ct;

```

```

    ct.depth=dct.depth;
    for(int i=0;i<L;i++)
    {
        ct.c0[i]=dct.d0[i]*pp.p[ct.depth]+dct.d2[i]*epk.b[i];
        ct.c0[i]=mod(ct.c0[i],pp.q[ct.depth]);
        ct.c1[i]=dct.d1[i]*pp.p[ct.depth]+dct.d2[i]*epk.a[i];
        ct.c1[i]=mod(ct.c1[i],pp.q[ct.depth]);
    }
    // Ciphertext rct = SwitchModulus(pp, ct);
    return ct;
}
PubPara& Setup(int p)
{
    PubPara pp;
    pp.p[0]=p;
    pp.q[0]=pp.p[0];
    pp.rp[0]=1;
    printf("p[0]:%d\tq[0]:%d\trp[0]:%d\n",pp.p[0],pp.q[0],pp.rp[0]);
    for(int i=0;i<L-1;i++)
    {
        pp.p[i+1]=proper_prime(pp.p[0],pp.p[i]+1);
        pp.q[i+1]=pp.p[i+1]*pp.q[i];
        pp.rp[i]=mod_reverse(pp.p[i+1],pp.q[i]);
        printf("p[%d]:%d\tq[%d]:%d\trp[%d]:%d\n", (i+1), pp.p[i+1], (i+1), pp.q[i+1], (i), pp.rp[i]);
    }
    return pp;
}
SecKey& SecKeyGen(PubPara pp)
{
    srand(1);
    SecKey sk;
    for(int i=0;i<L;i++)
    {
        sk.s[i]=rand()%pp.q[L-1];
    }
    /*
    printf("sk:");
    for(int j=0;j<L;j++)
    {
        printf("%d",sk.s[j]);
        if(j!=L-1)printf(",");
    }
    printf("\n");
    */
    return sk;
}
PubKey& EncryptionPubKeyGen(PubPara pp, SecKey sk)
{
    srand(2);
    PubKey pk;

```



```

    unsigned long eps[l];
    for(int i=0;i<l;i++)
    {
        pk.a[i]=mod(rand(),pp.q[L-1]);
        eps[i]=mod(rand(),err);
        pk.b[i]=pk.a[i]*sk.s[i]+pp.p[0]*eps[i];
        pk.b[i]=mod(pk.b[i],pp.q[L-1]);
    }
    printf(" pk: a:");
    for(int j=0;j<l;j++)
    {
        printf("%d",pk.a[j]);
        if(j!=l-1)printf(",");
    }
    printf("\n");
    for(int j=0;j<l;j++)
    {
        printf("%d",pk.b[j]);
        if(j!=l-1)printf(",");
    }
    printf("\n");
    return pk;
}
PubKey& SwitchKeyGen(PubPara pp, SecKey sk)
{
    srand(3);
    PubKey pk;
    unsigned long eps[l];
    for(int i=0;i<l;i++)
    {
        pk.a[i]=mod(rand(),pp.q[L-1]);
        eps[i]=mod(rand(),err);
        int rnd = rand()%err; //rnd cannot be too much, otherwise b will exceed the representation
range of int and decrypt will fail
        pk.b[i]=pk.a[i]*sk.s[i]+pp.p[0]*eps[i]-sk.s[i]*sk.s[i]*(rnd *pp.p[0]+1);
        pk.b[i]=mod(pk.b[i],pp.q[L-1]);
    }
    printf("epk: a:");
    for(int j=0;j<l;j++)
    {
        printf("%d",pk.a[j]);
        if(j!=l-1)printf(",");
    }
    printf("\n");
    for(int j=0;j<l;j++)
    {
        printf("%d",pk.b[j]);
        if(j!=l-1)printf(",");
    }
    printf("\n");
}

```

```

        return pk;
    }
    Ciphertext& Encrypt(PubPara pp, PubKey pk, Plaintext pt, int depth)
    {
        srand(3);
        int v[l], e0[l], e1[l];
        Ciphertext ct;
        for(int i=0;i<l;i++)
        {
            v[i]=rand()%2;
            e0[i]=rand()%err;
            e1[i]=rand()%err;
            ct.c0[i]=pk.b[i]*v[i] + pp.p[0]*e0[i]+pt.m[i];
            ct.c0[i]=mod(ct.c0[i],pp.q[depth]);
            ct.c1[i]=pk.a[i]*v[i] + pp.p[0]*e1[i];
            ct.c1[i]=mod(ct.c1[i],pp.q[depth]);
        }
        ct.depth=depth;
        return ct;
    }
    Plaintext& Decrypt(PubPara pp, SecKey sk, Ciphertext ct)
    {
        Plaintext pt;
        for(int i=0;i<l;i++)
        {
            pt.m[i] = ct.c0[i] - sk.s[i]*ct.c1[i];
            pt.m[i] = mod(pt.m[i],pp.q[ct.depth]);
            pt.m[i] = mod(pt.m[i],pp.p[0]);
        }
        printf("");
        for(int j=0;j<l;j++)
        {
            printf("%d",pt.m[j]);
            if(j!=l-1)printf(",");
        }
        printf("\n");
        return pt;
    }
    Ciphertext& add(PubPara pp, Ciphertext ct1, Ciphertext ct2)
    {
        //printf("in add\n");
        Ciphertext ct;
        if(ct1.depth != ct2.depth)
        {
            printf("the depth of the two cts are not equal");
            return ct;
        }
        ct.depth=ct1.depth;
        for(int i=0;i<l;i++)
        {

```

```

        ct.c0[i] = ct1.c0[i] + ct2.c0[i];
        ct.c0[i] = mod(ct.c0[i],pp.q[ct.depth]);
        ct.c1[i] = ct1.c1[i] + ct2.c1[i];
        ct.c1[i] = mod(ct.c1[i],pp.q[ct.depth]);
    }
    return ct;
}
Ciphertext& mul(PubPara pp, PubKey epk, Ciphertext ct1, Ciphertext ct2)
{
    //printf("in mul\n");
    int depth1 = ct1.depth;
    int depth2 = ct2.depth;
    //printf("ct1 depth:%d,ct2 depth:%d\n",depth1,depth2);
    Ciphertext s_ct1,s_ct2;
    if(depth1 == depth2)
    {
        s_ct1 = SwitchModulus(pp,ct1);
        s_ct2 = SwitchModulus(pp,ct2);
    }
    else if(depth1 < depth2)
    {
        s_ct1 = SwitchModulus(pp,ct1);
        s_ct2 = SwitchModulus(pp,ct2,depth1-1);
    }
    else
    {
        s_ct1 = SwitchModulus(pp,ct1,depth2-1);
        s_ct2 = SwitchModulus(pp,ct2);
    }
    DCiphertext dct = CTLinearization(pp,s_ct1,s_ct2);
    Ciphertext mct = SwitchKey(pp,epk,dct);
    return mct;
}
Ciphertext& mul2(PubPara pp, PubKey epk, Ciphertext ct1, Ciphertext ct2)
{
    //printf("in mul\n");
    DCiphertext dct = CTLinearization2(pp,ct1,ct2);
    Ciphertext mct = SwitchKey(pp,epk,dct);
    return mct;
}

```

Appendix A.3. Unlimited Homomorphic Encryption.cpp

```

#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    //—setup——
    PubPara pp = Setup(pt_q);
    //—keygen——
    SecKey sk=SecKeyGen(pp);
}

```

```

    PubKey pk=EncryptionPubKeyGen(pp,sk);
    PubKey epk=SwitchKeyGen(pp,sk);
//—encrypt—for correctness test——
    Plaintext pt[test_time+1]; //plaintext
    Ciphertext ct[test_time+1]; //ciphertext
    Plaintext dpt[test_time+1]; //decrypted plaintext
/*
//—encrypt—for performance test——
    Plaintext pt[3]; //plaintext
    Ciphertext ct[3]; //ciphertext
    Plaintext dpt[3]; //decrypted plaintext
*/
//—pt-init——
    for(int i=0;i<l;i++)
    {
        pt[0].m[i]=i+1;
    }
//—ct-init——
    ct[0] = Encrypt(pp,pk,pt[0]);
//——test-start——
    double result1,result2;
    clock_t start,end;
    start = clock();
//——
    for(int i=0;i<test_time-1;i+=2)
    {
        for(int j=0;j<l;j++)
        {
//for correctness test
//——pt-mul——
            pt[i+1].m[j]= pt[i].m[j]*pt[0].m[j];
            pt[i+1].m[j] = mod(pt[i+1].m[j],pp.p[0]);
//——pt-add——
            pt[i+2].m[j]= pt[i+1].m[j]+pt[0].m[j];
            pt[i+2].m[j] = mod(pt[i+2].m[j],pp.p[0]);
/*for performance test
//——pt-mul——
            pt[1].m[j]= pt[1].m[j]*pt[0].m[j];
            pt[1].m[j] = mod(pt[1].m[j],pp.p[0]);
//——pt-add——
            pt[2].m[j]= pt[1].m[j]+pt[0].m[j];
            pt[2].m[j] = mod(pt[2].m[j],pp.p[0]);
*/
        }
    }
    end = clock();
    result1 = (double)(end - start);
    start = clock();
    for(int i=0;i<test_time-1;i+=2)
    {

```

```

//for correctness test
//-----ct-mul-----
    ct[i+1]= mul(pp,epk,ct[i],ct[0]);
    //ct[i+1]= mul2(pp,epk,ct[i],ct[0]);
//-----ct-add-----
    ct[i+2]= add(pp,ct[i+1],ct[0]);
/*for performance test
//-----ct-mul-----
    ct[1]= mul(pp,epk,ct[0],ct[0]);
    //ct[1]= mul2(pp,epk,ct[0],ct[0]);
//-----ct-add-----
    ct[2]= add(pp,ct[1],ct[0]);
*/
}
end = clock();
result2 = (double)(end - start);
/**/
for(int i=0;i<=test_time;i++)
{
    printf(" pt[%d]:(",i);
    for(int j=0;j<1;j++)
    {
        printf("%d",pt[i].m[j]);
        if(j!=1-1)printf(",");
    }
    printf(")\n");
    printf("dpt[%d]:",i);
    dpt[i] = Decrypt(pp,sk,ct[i]);
}
printf("pt mul time:%f seconds\n",result1/CLOCKS_PER_SEC);
printf("ct mul time:%f seconds\n",result2/CLOCKS_PER_SEC);
char a;
scanf("%c",&a);
return 0;
}

```

Appendix B. Benchmark of Helib

This section gives the source code of a benchmark of Helib. To run this program, you should install Helib at first.

Appendix B.1. FHE_op.h

```

#include <NTL/ZZ.h>
#include <NTL/BasicThreadPool.h>
#include <NTL/lzz_pXFactoring.h>
#include "FHE.h"
#include "time.h"
#include <iostream>

```

Appendix B.2. Test_FHE_op.cpp

```

#include "FHE_op.h"
using namespace std;
int main(int argc, char **argv)
{
    long r = 1; //lifting
    long c = 2; //number of columns in key-switching matrices,  $(a, b)$ 
    long k = 32; //security parameter, corresponding to the 32-bit "int" space
    long s = 0; //minimum number of slots
    long d = 0; //degree of the field extension
    long w = 64; // Hamming weight of secret key

    long m = 5;
    long p = 11;
    long L = 5000; //Number of levels in the modulus chain, decide the max multiplication time

    cout << "Initializing context..." << flush;
    FHEcontext context(m,p,r); //initialize context
    buildModChain(context, L, c); //modify the context
    cout << "OK!" << endl;

    cout << "Creating polynomial..." << flush;
    NTL::ZZX G = context.alMod.getFactorsOverZZ()[0]; //creates the polynomial
    cout << "OK!" << endl;

    cout << "Generating keys..." << flush;
    FHESecKey secretKey(context); //construct a secret key structure
    const FHEPubKey& publicKey = secretKey; //FHESecKey is a subclass of FHEPubKey
    secretKey.GenSecKey(w); //actually generate a secret key with Hamming weight w
    cout << "OK!" << endl;

    Ctxt Acc(publicKey);
    Ctxt ctxt(publicKey);
    publicKey.Encrypt(Acc, NTL::to_ZZX(1));
    publicKey.Encrypt(ctxt, NTL::to_ZZX(2));

    NTL::ZZX ptAcc;
    clock_t start, finish;
    double Total_time;
    start = clock();

    for(int i=0;i<int(L/5);i+=2) //To ensure the capacity of homomorphic operations
    {
        Acc*=ctxt;
        //secretKey.Decrypt(ptAcc, Acc);
        //cout << "ptAcc " << i+1 << "=" << ptAcc[0] << endl;
        Acc+=ctxt;
        //secretKey.Decrypt(ptAcc, Acc);
        //cout << "ptAcc " << i+1 << "=" << ptAcc[0] << endl;
    }
}

```



```

    }
    finish = clock();
    Total_time = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("\n running time: %f seconds \n", Total_time);

    return 0;
}

```

Appendix C. Benchmark of RSA

This section gives the source code of a benchmark of RSA.

```

#include <stdio.h>
#include <tchar.h>
#include <time.h>
#define test_time 1000000
int p,q,n,phin,e,d;
void setup_and_key_gen()
{
    p=127,q=131;
    n=p*q;
    phin=(p-1)*(q-1);
    printf("p:%d,q:%d,n:%d,phi(n):%d\n",p,q,n,phin);
    e=53;
    for(d=1;;d++)
    {
        if(d*e%phin==1)break;
    }
    printf("e:%d,d:%d\n",e,d);
}
int enc(int pt,int e,int n)
{
    int ct=1;
    for(int i=0;i<e;i++)
    {
        ct = ct * pt;
        ct = ct % n;
    }
    return ct;
}
int dec(int ct,int d,int n)
{
    int dpt=1;
    for(int i=0;i<d;i++)
    {
        dpt = dpt * ct;
        dpt = dpt % n;
    }
    return dpt;
}
int _tmain(int argc, _TCHAR* argv[])

```

```

{
    setup_and_key_gen();
    int pt=2;
    int ct=enc(pt,e,n);
    int ct1=enc(pt,e,n);
    double result;
    clock_t start,end;
    start = clock();
    for(int i=0;i<test_time-1;i++)
    {
        ct = ct*ct1;
        ct = ct%n;
    }
    end = clock();
    result = (double)(end - start);
    printf("rsa mul for %d times, time:%f seconds\n", test_time, result / CLOCKS_PER_SEC);
    char c;
    scanf("%c",&c);
    return 0;
}

```

References

1. Cheng, X.; Pitziolis, A.; Lasebae, A. Implementing Fingerprint Recognition on One-Time Password Device to Enhance User Authentication. In *International Symposium on Cyberspace Safety and Security*; Springer: Cham, Switzerland, 2019; pp. 448–461.
2. Zhang, X.; Yang, S.; Srivastava, G.; Chen, M.Y.; Cheng, X. Hybridization of cognitive computing for food services. *Appl. Soft Comp.* **2020**, *89*, 106051. [\[CrossRef\]](#)
3. Ali, B.; Cheng, X. Security Solution Based on Raspberry PI and IoT. In *International Symposium on Cyberspace Safety and Security*; Springer: Cham, Switzerland, 2019; pp. 162–171.
4. Lv, S.; Ye, J.; Yin, S.; Cheng, X.; Feng, C.; Liu, X.; Li, R.; Li, Z.; Liu, Z.; Zhou, L. Unbalanced private set intersection cardinality protocol with low communication cost. *Future Gener. Comp. Syst.* **2020**, *102*, 1054–1061. [\[CrossRef\]](#)
5. Sun, X.; Zhang, P.; Liu, J.K.; Yu, J.; Xie, W. Private machine learning classification based on fully homomorphic encryption. *IEEE Trans. Emerg. Top. Comp.* **2020**, *8*, 352–364. [\[CrossRef\]](#)
6. Cheon, J.H.; Kim, J. A Hybrid Scheme of Public-Key Encryption and Somewhat Homomorphic Encryption. *IEEE Trans. Inform. Forens. Sec.* **2015**, *10*, 1052–1063. [\[CrossRef\]](#)
7. Chen, H.; Laine, K.; Rindal, P. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the ACM Conference on Computer and Communications Security 2017*, Dallas, TX, USA, 30 October–3 November 2017; pp. 1243–1255.
8. Liu, S.; Li, Z.; Zhang, Y.; Cheng, X. Introduction of Key Problems in Long-Distance Learning and Training. *MONET* **2019**, *24*, 1–4. [\[CrossRef\]](#)
9. Roy, S.S.; Vercauteren, F.; Vliegen, J.; Verbauwhede, I. Hardware Assisted Fully Homomorphic Function Evaluation and Encrypted Search. *IEEE Trans. Comp.* **2017**, *66*, 1562–1572. [\[CrossRef\]](#)
10. Bourse, F.; Minelli, M.; Minihold, M.; Paillier, P. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In *Proceedings of the International Cryptology Conference (CRYPTO) 2018*, LNCS 10993, Santa Barbara, CA, USA, 19–23 August 2018; pp. 483–512.
11. Malavolta, G.; Thyagarajan, S.A.K. Homomorphic Time-Lock Puzzles and Applications. In *Proceedings of the International Cryptology Conference (CRYPTO) 2019*, LNCS 11692, Santa Barbara, CA, USA, 18–22 August 2019; pp. 620–649.
12. Gentry, C. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC) 2009*; ACM: New York, NY, USA, 2009; pp. 169–178.

13. Regev, O. On lattices, learning with errors, random linear codes, and cryptography. In *ACM Symposium on Theory of Computing (STOC) 2005*; ACM: New York, NY, USA, 2005; pp. 84–93.
14. Peikert, C. Public-key cryptosystems from the worst-case shortest vector problem: Extended abstract. In *ACM Symposium on Theory of Computing (STOC) 2009*; ACM: New York, NY, USA, 2009; pp. 333–342.
15. Brakerski, Z.; Vaikuntanathan, V. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Proceedings of the International Cryptology Conference (CRYPTO) 2011*, Santa Barbara, CA, USA, 14–18 August 2011; pp. 505–524.
16. Brakerski, Z.; Vaikuntanathan, V. Efficient fully homomorphic encryption from (standard) LWE. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS) 2011*, Palm Springs, CA, USA, 22–25 October 2011; pp. 97–106.
17. Brakerski, Z.; Gentry, C.; Vaikuntanathan, V. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the Conference on Innovations in Theoretical Computer Science (ITCS)*, Cambridge, MA, USA, 8–10 January 2012; ACM: New York, NY, USA, 2012; pp. 309–325.
18. Gentry, C.; Halevi, S.; Smart, N.P. Fully homomorphic encryption with polylog overhead. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) 2012*, LNCS 7237, Cambridge, UK, 15–19 April 2012; pp. 465–482.
19. Gentry, C.; Halevi, S.; Smart, N.P. homomorphic evaluation of the AES circuit. In *Proceedings of the International Cryptology Conference (CRYPTO) 2012*, LNCS 7417, Santa Barbara, CA, USA, 14–18 August 2012; pp. 850–867.
20. Brakerski, Z. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Proceedings of the International Cryptology Conference (CRYPTO) 2012*, LNCS 7417, Santa Barbara, CA, USA, 14–18 August 2012; pp. 868–886.
21. Halevi, S.; Shoup, V. Algorithms in HELib. In *Proceedings of the International Cryptology Conference (CRYPTO) 2014*, Santa Barbara, CA, USA, 17–21 August 2014; pp. 554–571.
22. Halevi, S.; Shoup, V. Bootstrapping for HELib. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) 2015*, Sofia, Bulgaria, 26–30 April 2015; pp. 641–670.
23. Halevi, S.; Shoup, V. Faster Homomorphic Linear Transformations in HELib. In *Proceedings of the International Cryptology Conference (CRYPTO) 2018*, LNCS 10993, Santa Barbara, CA, USA, 19–23 August 2018; pp. 93–120.
24. Castryck, W.; Iliashenko, I.; Vercauteren, F. Homomorphic SIMD Operations: Single Instruction Much More Data. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) 2018*, LNCS 10820, Tel Aviv, Israel, 29 April–3 May 2018; pp. 338–359.
25. Öztürk, E.; Doröz, Y.; Savas, E.; Sunar, B. A Custom Accelerator for Homomorphic Encryption Applications. *IEEE Trans. Comp.* **2017**, *66*, 3–16. [[CrossRef](#)]
26. Eric, C.; Chris, P.; Chad, S. ALCHEMY: A Language and Compiler for Homomorphic Encryption Made easy. In *Proceedings of the ACM Conference on Computer and Communications Security 2018*, Toronto, ON, Canada, 15–19 October 2018; pp. 1020–1037.
27. Alperin-Sherief, J.; Peikert, C. Practical bootstrapping in quasilinear time. In *Proceedings of the International Cryptology Conference (CRYPTO) 2013*, Santa Barbara, CA, USA, 18–22 August 2013; pp. 1–20.
28. Alperin-Sherief, J.; Peikert, C. Faster bootstrapping with polynomial error. In *Proceedings of the International Cryptology Conference (CRYPTO) 2014*, Santa Barbara, CA, USA, 17–21 August 2014; pp. 297–314.
29. Brakerski, Z.; Vaikuntanathan, V. Lattice-based FHE as secure as PKE. In *Proceedings of the Conference on Innovations in Theoretical Computer Science 2014*, Princeton, NJ, USA, 12–14 January 2014; pp. 1–12.
30. Ducas, L.; Micciancio, D. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) 2015*, Sofia, Bulgaria, 26–30 April 2015; pp. 617–640.
31. Li, Z.; Ma, C.; Du, G.; Weiping, O. Efficient fully homomorphic encryption from (standard) LWE. In *Proceedings of the Conference on Parallel and Distributed Systems 2016*, Wuhan, China, 13–16 December 2016; pp. 1169–1174.
32. Chen, H.; Han, K. Homomorphic Lower Digits Removal and Improved FHE Bootstrapping. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) 2018*, LNCS 10820, Tel Aviv, Israel, 29 April–3 May 2018; pp. 315–337.
33. Cheon, J.H.; Han, K.; Kim, A.; Kim, M.; Song, Y. Bootstrapping for Approximate Homomorphic Encryption. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) 2018*, LNCS 10820, Tel Aviv, Israel, 29 April–3 May 2018; pp. 360–384.

34. Chen, H.; Chillotti, I.; Song, Y. Improved Bootstrapping for Approximate Homomorphic Encryption. In Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) 2019, LNCS 11477, Darmstadt, Germany, 19–23 May 2019; pp. 34–54.
35. Lyubashevsky, V.; Peikert, C.; Regev, O. On ideal lattices and learning with errors over rings. In Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques 2010, French Riviera, France, 30 May–3 June 2010; pp. 1–23.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).