Efficient software implementation of homomorphic encryption for addition and multiplication operations

Yongwoo Oh, Taeyun Kim, and Hyoungshick kim

Sungkyunkwan University,
(16419) 2066, Seobu-ro, Jangan-Gu, Suwon-Si, Gyeonggi-Do, Republic of Korea
{dyddn7997,taeyun1010,hyoung}@skku.edu

Abstract. Fully homomorphic encryption enables any type of calculation on encrypted data. There are several crypto libraries that provide such fully homomorphic encryption. However, since most libraries only support single level binary circuit operations, it is required for developers to efficiently implement basic arithmetic algorithms such as addition, subtraction, multiplication, and division for their own applications. In this paper, we propose fast *binary* addition and multiplication algorithms to support various bit-wise operations. To show the feasibility of the proposed algorithms, we implemented the proposed algorithms for 16, 32, 48, and 64 bits integers using the `TFHE` library. Our experiment results demonstrate that the proposed addition operation decreases the running time by 11 to 12 percent, and our multiplication implementation is about 3 to 4 times faster than the non-threaded method for 16, 32, 48 and 64 bits integers.

Keywords: homomorphic encryption · binary operation · concurrent calculation

1 Introduction

Homomorphic encryption is an encryption scheme that allows certain calculations to be performed on encrypted data. Since fully homomorphic encryption allows any calculation on encrypted data, it can be applied to various applications protecting user's confidential data.

Homomorphic encryption can be used in cloud computing, biometrics, medical data [1]. Also, homomorphic encryption can be used for blind auctions requiring fair processing among clients [8].

There are several fully homomorphic encryption libraries. However, they only support bit-level circuit operations (binary AND, OR, NAND, NOR, NOT, XOR and multiplexer circuit operations) on encrypted data. Therefore, for developers, it is needed to implement high-level algorithms with those basic circuits. The performance of such implementation can be greatly varied with the developer's programming skills and experience because homomorphic encryption operations are typically too slow. TFHE (https://github.com/tfhe/tfhe), one of the fastest

homomorphic crypto libraries, takes on average 13ms to complete one normal circuit operation and 26ms to complete a multiplexer circuit operation. These results show that homomorphic encryption operations are significantly slower than normal circuit operations.

There were several previous studies that aimed to improve the speed of homomorphic cryptosystem. For example, Brakerski et al. [2] suggested fully homomorphic encryption without bootstrapping, which is called BGV cryptosystem. Ilaria et al. [4] proposed a technique to accelerate bootstrapping that is a performance bottleneck in homomorphic encryption operations.

However, only a few studies (e.g., [5]) have tried to reduce the execution times of arithmetic operations for homomorphic encryption in application level. In this paper, we propose two implementation methods to improve the performance of the multiple bit addition and multiplication operations. Our key contributions are summarized as follows:

- We propose new homomorphic encryption algorithms to fully support various bit-wise addition and multiplication operations. For addition, we calculate concurrently on carry-out bit and sum bit using threads. For multiplication, we hierarchically group operand bits by two and calculating each group concurrently using threads in an iterative manner.
- We evaluate the performance of proposed addition and multiplication algorithms compared with non-threaded versions to show the efficiency of the proposed algorithms.

For evaluation, TFHE library was used in our implementation. However, we claim that our addition and multiplication algorithms can also be implemented by using any homomorphic encryption library with only slight change in a variable form and circuit operation functions.

2 Background

2.1 Homomorphic cryptosystem

Homomorphic encryption is an encryption scheme that enables calculations on encrypted data [6]. There are two types of homomorphic encryption scheme: partially homomorphic encryption scheme and fully homomorphic encryption scheme. Fully homomorphic encryption scheme supports all kinds of calculation on encrypted data but partially homomorphic one does not. Fully homomorphic encryption scheme uses two keys: a private key to encrypt and decrypt data, and an evaluation key to calculate on encrypted data.

2.2 Efficient implementation of arithmetic operations for homomorphic encryption

There have been several attempts to speed up arithmetic operations on homomorphically encrypted data. For example, Seo et al. [7] implemented an arithmetic adder for multiple variables using HElib (https://github.com/shaih/HElib).

Their implementation took 130 seconds for 6 variables to be added and took 195 seconds for 32 variables' addition under 32-bit environment. In particular, their implementation is efficient when a sequence of addition operations should only be performed on encrypted data. In their method, however, when multiplication and addition operations are performed in a mixed manner, their performance was significantly degraded because this paper was targeting for improving multiple addition-only operations. Yao et al. [3] also implemented arithmetic operations (i.e., addition, subtraction, multiplication and division) on encrypted data using HElib. However, the performance of their operations is not sufficient for real-world applications. In this paper, we present new addition and multiplication implementation techniques using multiple threads to speed up the performance of those arithmetic operations. To the best of our knowledge, this is the first implementation of arithmetic operations using concurrent programming for homomorphic encryption.

2.3 Adder circuit

When implementing addition and multiplication operations on encrypted data using homomorphic encryption library, we used two adder circuits: a half adder and a full adder. In big-endian environment, a half adder runs on least significant bit, and a full adder runs on the remaining bits.

In a half adder circuit, two result bits, a carryout bit and a sum bit, are calculated as follows:

- Sum = $a \oplus b$
- CarryOut =$a \wedge b$

Here, $\oplus$ and $\wedge$ represent bit-wise XOR and AND operations, respectively.

In a full adder circuit that uses a multiplexer, two result bits, a carryout bit and a sum bit are calculated as follows:

- Sum = $a \oplus b \oplus CarryIn$
- CarryOut =$MUX_{a \oplus b}(a, CarryIn)$

where $MUX_a(b, c)$ is equal to b if $a = 0$; otherwise, $MUX_a(b, c)$ is equal to c if $a = 1$.

3 Methodology

3.1 Addition

Let two encrypted integers be $A = A_1 \cdots A_n$, $B = B_1 \cdots B_n$, and sum of A and B be $S = S_1 \cdots S_n$ where A_i, B_i, and $S_i \in \{0, 1\}$.

First, we calculate the sum of two encrypted integers from a least significant bit (LSB) using a half adder. In this step, we can see that XOR and AND operations are not dependent to each other (see Fig. 1). Therefore, we

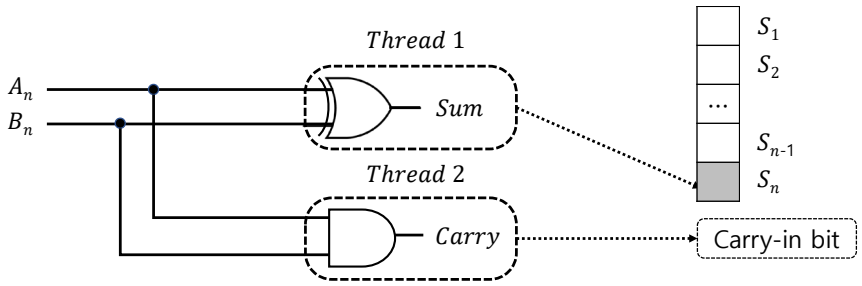

Fig. 1. Half adder in n-bit additions for homomorphic encryption. Here, A_n and B_n are LSB of two encrypted integers (A and B). XOR and AND operations can be independently performed.

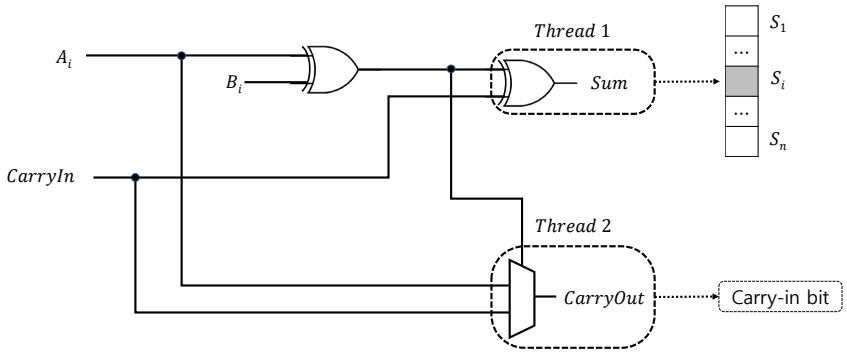

Fig. 2. Full adder in n-bit addition for homomorphic encryption. Here, A_i and B_i are ith bits of two encrypted integers (A and B). XOR and multiplexer operations can be independently performed.

can concurrently calculate those operations to avoid the time delay when those operations are sequentially executed.

After obtaining a carry bit from the LSB calculation, we should successively calculate the sum of the next bits and a carryout bits. We first calculate $A_i \oplus B_i$. Next, we use $A_i \oplus B_i$ and $CarryIn$ bit which is calculated as $CarryOut$ in previous step to calculate a sum bit and a $CarryOut$ bit. As shown in Fig. 2, we can calculate these two bits concurrently. Similar to the half adder, we repeat this process sequentially to calculate all carryout and sum bits.

3.2 Multiplication

For a n-bit multiplication operation, we sequentially perform shift and addition operations n times. Because each addition operation takes $O(n)$ time, n addition operations take $O(n^2)$ time. Therefore, we aim to perform addition operations in a parallel manner in order to avoid the time delay caused by the sequential execution of those operations.

Suppose there are two encrypted integers $A = A_1 \cdots A_n$, $B = B_1 \cdots B_n$ for multiplications where A_i and $B_i \in \{0, 1\}$. If we truncate the overflow part of multiplication, we can use the following equation of $A * B = M_1 + M_2 + \cdots + M_n$ where $M_n = A_n \wedge (B_1 B_2 \cdots B_n)$, $M_{n-1} = A_{n-1} \wedge (B_2 \cdots B_n 0)$, $M_{n-2} = A_{n-2} \wedge (B_3 \cdots B_n 00)$, $\cdots$, $M_1 = A_1 \wedge (B_n 0 \cdots 0)$.

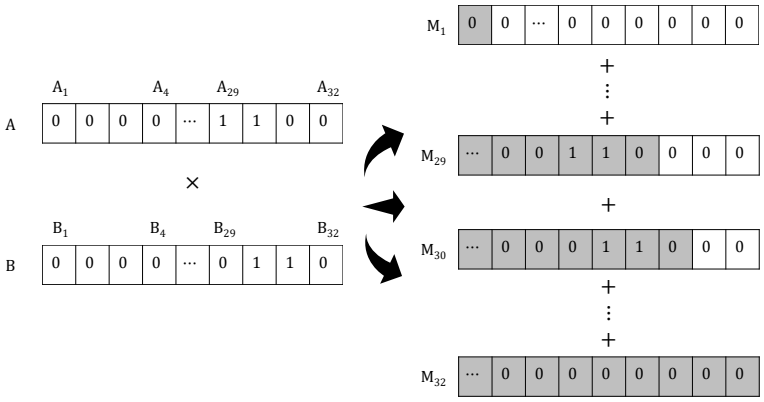

Fig. 3. First step for 32 bit variable multiplication.

In the proposed multiplication algorithm, the first step is to calculate $M_1, M_2, \cdots, M_n$. Interestingly, at this step, all $M_1, M_2, \cdots, M_n$ can be concurrently calculated because each M_i is independent from the other M_j where $i \neq j$. Fig. 3 shows an example of multiplication of two 32-bit integers A and B. When calculating M_i, only grey area is to be calculated from A and B, and white area is filled with a constant bit 0. Therefore, all M_i can be concurrently calculated using threads.

Given $M_1, M_2, \cdots, M_n$ for multiplication, we group them by two terms (e.g., M_i and M_j) and calculate them independently. Because every M_k has k valid upper bits, we have to calculate $min(i-1, j-1)$ bits to calculate $M_i + M_j$. Thus, if we match and group by (M_1, M_n), (M_2, M_{n-1}), $\cdots$, $(M_{\frac{n}{2}}, M_{\frac{n}{2}+1})$ to calculate each group concurrently, only $\frac{n}{2} - 1$ bit operation time is needed for performing all operations. Since we grouped terms by two, the number of $\frac{n}{2}$ calculations is needed. Let these be $M1_1$, $M1_2$, $\cdots$, $M1_{\frac{n}{2}}$. These results sequentially have

$n-1$, $n-2$, $\cdots$, $\frac{n}{2}$ number of zeros in the lower bits. To obtain $M2_1 \cdots M2_{\frac{n}{4}}$, there are $\frac{3n}{4}-1$ bit operations as every group can be calculated concurrently. If we repeat this process recursively until the last one is reached, we can obtain $\frac{n}{2}-1$, $\frac{3n}{4}-1$, $\frac{7n}{8}-1$, $\frac{15n}{16}-1$, $\cdots$ bit operation time for each level. Because the number of operations at each level is between $\frac{n}{2}-1$ and n, the maximum level is $O(\log n)$. Thus, the time complexity of the entire procedure would be $O(n \log n)$. Fig. 4 shows the process of 32-bit multiplication operation. We can see that calculations are performed hierarchically from 16 groups until only one remains.

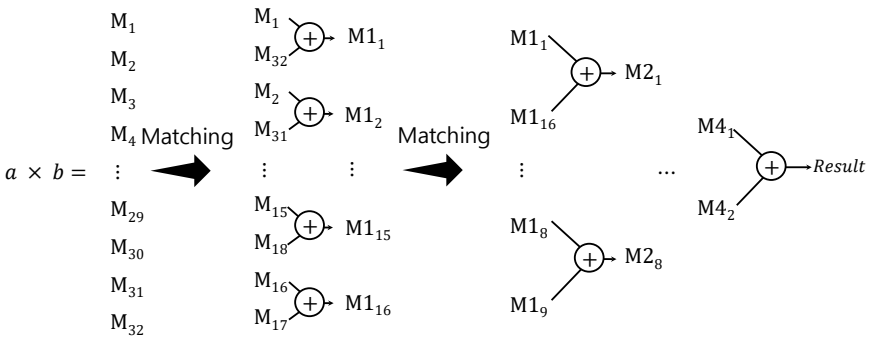

Fig. 4. Example of calculation of M_1 to M_{32} in two 32-bit integers. We group by two at each level and add each other to merge the results iteratively.

If the operand size is not a power of 2, we can pad the remaining M_i with encrypted zeros to increase its size to the next power of 2, and then apply the above procedure.

4 Evaluation

To show the feasibility of the proposed implementation techniques, we implemented the proposed addition and multiplication algorithms with varying bit sizes (16, 32, 48 and 64 bits). In our experiments, we used i5-7600K CPU, 16GB memory and Samsung EVO 850 pro 250GB SSD running Ubuntu 16.04 64-bit operating system. To measure the performance of execution time, we used the `perf` program that is widely used in performance measurement. For comparison, we also implemented *non-threaded* addition and multiplication operations with the same condition. To avoid bias, we performed both addition and multiplication operations of our implementations 100 times, respectively, for each condition. The experiment results are shown in Fig. 5 and 6.

Fig. 5 shows the performance of binary addition operations. Our proposed implementation method always produced better results for all bit operations

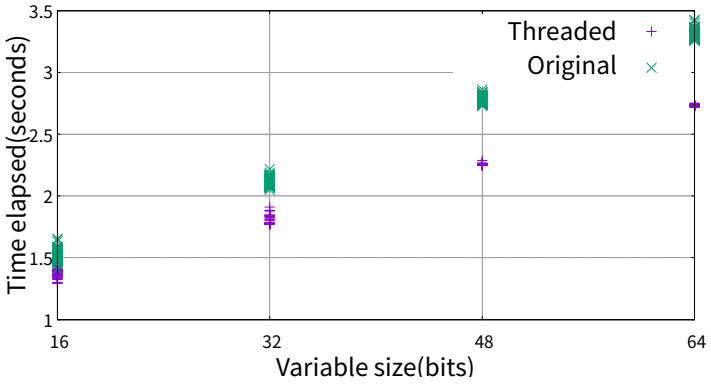

Fig. 5. Performance of addition operations.

than non-threaded implementation. Overall, the proposed method reduced the execution time by about 11 to 12 percent. Interestingly, the performance of the proposed method is relatively stable compared with the non-threaded implementation for 48 and 64 bit operations.

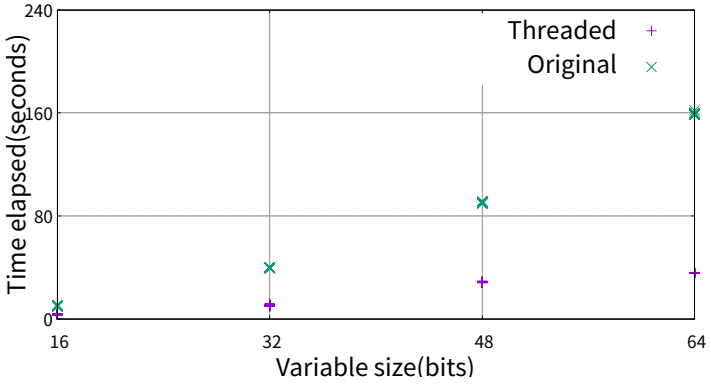

Fig. 6. Performance of multiplication operations.

Fig. 6 shows performance of binary multiplication operations. Because the proposed implementation technique significantly increases concurrency during the calculation, we achieved the performance improvement of 2.95, 3.93, 3.19 and 4.48 times for 16, 32, 48 and 64 bit operations, respectively. Again, in the case of multiplication operations, our proposed implementation method always produced better results for all bit operations than non-threaded implementation.

5 Conclusion and future work

We proposed new software implementation techniques to improve the performance of addition and multiplication operations on homomorphically encrypted data. To efficiently implement addition and multiplication operations, we suggest the use of threads to increase concurrency in performing operations. To show the feasibility of the proposed algorithms, we implemented addition and multiplication operations with varying bit sizes (16, 32, 48 and 64 bit). According to our experiments, the execution times of those operations can be significantly decreased by about 12 and 448 percent, respectively, compared with non-threaded implementations for 64 bit operations.

As part of future work, we plan to consider other techniques for rearranging to improve parallelism for basic arithmetic operations. We will also extend the proposed algorithms to other homomorphic cryptography libraries to generalize our results.

6 Acknowledgements

This work was supported by the ICT R&D programs (No.2017-0-00545) and the National Research Foundation (NRF) funded by the Ministry of Science and ICT (2017H1D8A2031628).

References

1. Archer, D., Chen, L., Cheon, J.H., Gilad-Bachrach, R., Hallman, R.A., Huang, Z., Jiang, X., Kumaresan, R., Malin, B.A., Sofia, H., Song, Y., Wang, S.: Applications of homomorphic encryption. Tech. rep. (2017)
2. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully Homomorphic Encryption without Bootstrapping. Tech. rep. (2011), https://eprint.iacr.org/2011/277
3. Chen, Y., Gong, G.: Integer arithmetic over ciphertext and homomorphic data aggregation. In: IEEE Conference on Communications and Network Security (2015)
4. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds. Tech. rep. (2016), https://eprint.iacr.org/2016/870
5. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Improving TFHE: faster packed homomorphic operations and efficient circuit bootstrapping. Tech. rep. (2017), https://eprint.iacr.org/2017/430
6. 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.) Advances in Cryptology – CRYPTO 2013. Springer Berlin Heidelberg (2013)
7. Seo, K., Kim, P., Lee, Y.: Implementation and Performance Enhancement of Arithmetic Adder for Fully Homomorphic Encrypted Data. Journal of the Korea Institute of Information Security & Cryptology **27**, 413–426 (2017)
8. Suzuki, K., Yokoo, M.: Secure generalized vickrey auction using homomorphic encryption. In: International Conference on Financial Cryptography. pp. 239–249. Springer (2003)