ISBN 978-981-11-3671-9
Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering (WCSE 2017)
Beijing, 25-27 June, 2017, pp. 956-960
doi: 10.18178/wcse.2017.06.166

Faster Binary Arithmetic Operations on Encrypted Integers

Jingwei Chen [1], Yong Feng [1], Yang Liu [2 +] and Wenyuan Wu [1]

[1] Chongqing Key Laboratory of Automated Reasoning and Cognition, Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences, Chongqing, China

[2] College of Information Science and Engineering, Chongqing Jiaotong University, Chongqing, China

Abstract. Fully homomorphic encryption (FHE) makes a large number of applications available in cloud computing environment. Following Gentry's seminal work, many FHE schemes have been presented. Among known FHE schemes, the Brakerski-Gentry-Vaikuntanathan scheme with corresponding optimizations is one of the most potential candidates for practical use, and has been implemented by Shoup and Halevi in a homomorphic encryption C++ library HElib. Based on HElib, Xu et al. (2016) reported their implementation of binary arithmetic operations over encrypted integers. In this paper, we optimize their implementation further. More specifically, the multi-thread technique is used to accelerate the arithmetic circuits. Moreover, ciphertext slots are fully used in our implementation. As a result, these techniques enable us to evaluate 600 additions of two 64-bit integers simultaneously within 25 seconds, about 0.042 seconds for per-addition time on average.

Keywords: FHE, HElib, integer arithmetic

1. Introduction

Nowadays, more and more individuals save and process sensitive data on cloud servers. Naturally, the security is the biggest problem that clients worry about. Fully homomorphic encryption (FHE) allows cloud servers to running any computable function on encrypted data, so that it can be used in an extensively large number of applications, such as consumer privacy in advertising, medical application, data mining, etc.

Although the usefulness of FHE had already been noticed by Rivest et al. [1] as early as 1978, the first breakthrough was not made until Gentry's work [2] in 2009. Followed Gentry's seminal work, many FHE schemes appear, for instance [3]–[12]. As indicated by Gentry et al. in [13], the (Ring-)LWE based Brakerski-Gentry-Vaikuntanathan (BGV) [6] scheme is one of the few variants that seem the most likely to yield "somewhat practical" homomorphic encryption. The BGV scheme, along with many optimizations in [7], [12] to make the homomorphic evaluation faster, has been implemented in a C++ library by Halevi and Shoup, named HElib [14]. HElib focuses on effective use of the Smart-Vercauteren ciphertext packing techniques [12], which make single-instruction-multiple-data (SIMD) operations available for homomorphic evaluation. Since almost all known FHE schemes, including BGV, are designed for circuits, HElib supplies only the basic circuit functions. For example, HElib does not include a function for homomorphic evaluation of the decimal addition of two integers. However, it is undoubted that this kind of decimal integer arithmetic operations is frequently used in practice. Based on HElib, Chen and Gong [15] and Xu et al. [16] implemented decimal arithmetic operations for integers via binary circuits.

In this paper, we optimize the above routine further. More precisely, the multi-thread technique is used to accelerate the arithmetic circuits, which makes our implementation nearly 2× faster than that of Xu et al.'s implementation; see Section IV for details. Moreover, ciphertext slots are fully used in our implementation, while it seems that only one slot is used in Xu et al's implementation. As a result, these techniques enable us

[+] Corresponding author. Tel.: + 86 23 65 93 56 58; fax: +86 23 65 93 50 00.
E-mail address: ly1246@qq.com.

to evaluate 600 additions of two 64-bit integers simultaneously within 25 seconds, i.e., about 0.042 seconds for per-addition time on average; see Section IV for more experimental results.

1.1. Related Work

Homomorphically encrypted integer arithmetic operations are the fundamental of higher level applications. Naehrig et al. [17] implemented a RLWE-based somewhat homomorphic encryption scheme in MAGMA. Wu and Haven [18] reported their implementation based on HElib as well, however, they utilized large plaintext spaces over $\mathbb{Z}_p$ with prime $p > 2^{128}$. Their implementation does not include the homomorphic evaluation of the carry bits, so does the implementation [10] of the DGHV FHE scheme [3]. To the best of our knowledge, Chen and Gong [15] seems the first published work to implement the binary integer arithmetic by using HElib, however, they only reported encrypted integer arithmetic with at most 4 bits. Xu et al. [16] improved the efficiency by using several optimizations and designing the circuits more carefully. In this paper, we optimize the above implementation further and obtain some practical average per-operation performances. Cheon et al. [19] reported their implementation for binary integer addition (with equality test and comparison) based on well-designed SIMD circuits and HElib. The way of using SIMD of our implementation is different from Cheon et al.'s. This leads to our implementation allowing element-wise integer vector arithmetic operations.

2. Backgrounds

In this section, we recall the BGV scheme and HElib. We refer to [6], [20], [21] for more details.

2.1. The BGV Scheme

The BGV scheme [6] is an improvement of Brakerski and Vaikuntanathan [4], which is based on standard assumptions supported by worst-case hardness of LWE [22] or RLWE [23]. In addition, BGV is capable of evaluating arbitrary circuits of a priori bounded depth without the bootstrapping procedure. Here we use a variant of the basic BGV encryption scheme that is implemented in HElib; see, e.g., [16, Se. 2.2].

We limit the plaintext space to $\mathrm{R}_2 = \mathbb{Z}_2[x]/\Phi_m(x)$ 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. We also note that m is the dominating parameter for efficiency as it determines the size of computation. The BGV scheme support ciphertext packing. The ciphertext packing technique allows us to evaluate a function homomorphically in parallel on ℓ blocks of encrypted data. It works essentially by packing multiple plaintexts into one ciphertext. More specifically, when the plaintext space is limited to $\mathrm{R}_2 = \mathbb{Z}_2[x]/\Phi_m(x)$, where $\Phi_m(x)$ is the m-th cyclotomic polynomial, $\Phi_m(x)$ can be factorized into ℓ irreducible factors of same degree $d = \varphi(m)/\ell$, i.e., $\Phi_m(x) = f_1(x) \cdots f_\ell(x)$ where $\varphi(*)$ is the Euler's totient function. Each factor corresponds to a plaintext slot. Thus, for each $a \in \mathrm{R}_2$, it can be represented as an ℓ-vector $(a \bmod f_i)_i$. Using the techniques in [7], [12], one can perform SIMD operations on ℓ blocks of ciphertexts. Just like for integer Chinese Remaindering, addition and multiplication in R_2 correspond to element-wise addition and multiplication of the vectors of slots.

2.2. The HElib Implementation

HElib [14] is an open-source C++ implementation of the BGV scheme based on the C++ Number Theory Library (NTL) [24]. 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. Due to the parallelization, SIMD operations bring much better amortized per-bit timing. We refer to the document [21] and the source code for the exact usage of HElib. Here we only focus on some parameter settings which play important roles in practice. In the library, the ciphertext space is $\mathrm{R}_q = \mathbb{Z}_q[x]/\Phi_m(x)$, where $q = p_1 p_2 \cdots p_n$ is the modulus and each p_j is a small prime generated by the library. By double CRT representation, each ciphertext is represented as an $n \times \varphi(m)$ matrix. Each entry of the matrix is an evaluation of the ciphertext polynomial at certain point modulo p_j for some j. According to [13, Appendix C] (in the full version), the parameter m is chosen such that

$$\varphi(m) \geq (\lambda + 110)(L_c(\log \varphi(m) + 23) - 8.5)/7.2, \tag{1}$$

where L_c is the minimum number of levels of modulus chain and λ is the security parameter.

In applications, the minimum number of levels in the modulus chain L_c in HElib is the number of modulus switches L_s plus one. In HElib, $L_s \approx 2\lceil L/2 \rceil$, and thus $L_s \approx 2\lceil L/2 \rceil + 1$, where L is the multiplicative depth we want to support. From (1), a larger L_c implies a larger $\varphi(m)$. This makes both addition and multiplication of ciphertexts less efficient. Thus, when we design a circuit for a certain application, we should choose those circuits with the multiplicative depth as less as possible. Totally speaking, the security parameter λ and L_c determine the computing overhead.

3. Implementation of Arithmetic Operations

In this section, we present our implementation of integer arithmetic operations by using AND and XOR gate evaluation in HElib. We assume that every integer is written in a little-endian two's complement representation. We use one ciphertext to represent one bit in our implementation, and a double-ended queue of ciphertexts to represent an encrypted binary integer. In what follows, all bits we use are encrypted by the HElib function EncryptedArray::encrypt under the same public key.

3.1. Binary Integer Arithmetic Algorithms

We implement several different adders, including Ripple Carry Adder (RCA) and Carry Lookahead Adder (CLA), and hence several different circuits for subtraction and multiplication as well.

We implement an n-bit RCA according to [16, Algorithm 1]. 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. We also implemented the CLA adder for both 16 bits encrypted integers and 64 bits integers, respectively, as in [16, Algorithm 2]. We note that this adder requires more bit operations than RCA, but with lower multiplicative depth. More precisely, the CLA adder circuit has multiplicative depth $L = O(\log n)$ (see [25]).

For subtraction, we can obviously obtain two algorithms which correspond to the RCA adder and the CLA adder, respectively, and their multiplicative depths are $n - 1$ and $O(\log n)$, respectively.

For multiplication, we use the school method, i.e., using one binary integer to multiply every bit of the other number and then adding all the middle results together. The multiplicative depth of the multiplication circuit is one level larger than the addition.

For integer division with remainder, we use the non-restoring method, which can be found, e.g., [15]. The multiplicative depth is about $\mathrm{len}(a) \cdot \mathrm{len}(b)$ for $a \div b$, where $\mathrm{len}(a)$ is the bit length of a.

3.2. Using Full of Slots

In fact, one of the most important reasons that we choose the one-ciphertext-one-bit representation of encrypted integers is that we want to make our implementation support encrypted integer vector operations. Since that BGV and HElib support SIMD, such a target can be implemented by using full of ciphertext slots.

For example, suppose that $\boldsymbol{a} = (a_1, a_2, \cdots, a_k)$ and $\boldsymbol{b} = (b_1, b_2, \cdots, b_k)$ are two k-dimensional integer vectors to be added and that the bit length of each a_i and b_i is at most n. First, we choose appropriate parameters such that there are at least k slots supported by the encryption scheme. Then, in the r-th slot of one ciphertext, we only encrypt one bit for a_r or b_r. So after running EncryptedArray::encrypt at most $2n$ times, we obtain $2n$ ciphertexts, each of which represents one encrypted bit for a_i or b_i. At last, we can directly revoke the adder we mentioned previously and obtain the elementwise encrypted addition results. The multiplicative depth of this process is the same as that of the corresponding adder that we use.

In addition, due to using R_2 as our plaintext space, our implementation is in fact an embedding of relatively small plaintexts into large ciphertexts. The full use of ciphertext slots therefore enable more efficient use of both space and computational resources.

3.3. Multi-thread Implementation

As mentioned previously, HElib is based on NTL. Thanks to thread safe mode of NTL, HElib is eventually thread safe since March 2015. So we can adapt our implementation accordingly such that it supports multi-thread computation.

First, we add #include <NTL/BasicThreadPool.h> into our test.cpp file and then set the number of threads we want to use by SetNumThreads(nthreads). During the main body of the code, if it is suitable for parallelization, we only use the higher-level macros for writing simple parallel ‘for’ loops. Namely, before and after the ‘for’ loop, we add NTL_EXEC_RANGE(n, first, last) and NTL_EXEC_RANGE_END, respectively.

4. Experiments

In this section, we show the performance of our implementation and the comparison with that in [16]. We use the HElib function FindM() to decide the parameter m, which is very important for the performance. We set the security level as a reasonable value $\lambda = 80$ (the same as [15], [16]). Besides, we set the parameter L_c (see (1)) the same as [16] as well. All time are obtained on a PC with a Intel Core i7 4790 CPU of 3.60 GHz and 8 GB RAM by using 8 threads. In Table 1, the #bits column represents the current circuit supports #bits encrypted integer arithmetic, m is decided by the security parameter λ and L_c as in (1), the #slots column is the number of slots, the Xu et al. column is the performance of the implementation in [16] and the timing is counted in seconds.

Table 1: Performance of Binary Encrypted Integer Arithmetic

	circuit	#bits	m	#slots	L_c	Xu et al.	Time
+	RCA	16	14351	504	17	2.16	1.16
	CLA	16	7781	150	7	2.53	2.05
	CLA	64	13981	600	13	37.69	24.36
−	RCS	16	14351	504	17	2.17	1.20
	CLA	16	7781	150	7	2.52	2.02
	CLA	64	13981	600	13	37.16	24.73
×	RCA	8	8191	630	9	4.62	2.63
	RCA	16	14351	504	17	46.32	29.34
÷	RCA	4	18631	720	21	14.63	7.74

From Table 1, it is obvious that our implementation outperforms that of [16]. Moreover, duo to full use of slots, the time in last column is the cost for doing #slots operations simultaneously. For instance, our implementation is able to evaluate 600 additions of two 64-bit integers simultaneously within 25 seconds, i.e., about 0:042 seconds for per-addition time on average.

5. Conclusion and Discussion

In this paper, we implemented the arithmetic operations for encrypted integers by using HElib. Our implementation extends Xu et al.’s work [16] in two folds. For one, we use full of ciphertext slots such that our implementation supports encrypted integer vector operations element-wise, which leads to a practical average performance. For another, our implementation supports multi-thread mode, which leads to a speedup.

However, we also read from Table 1 that the speedup is not good as 8× faster. The reason is that our parallel strategy is in the code level. Namely, all circuits we use are not designed for parallel computing. For example, it is well known that the RCA adder is not suitable for parallelization. For CLA, the parallelization is under consideration. In addition, it is an intriguing topic to design and implement circuits for encrypted fixed point number system.

6. Acknowledgements

The present work was partially supported by NSFC (11501540, 11471307, 11671377), Chongqing Research Program of Basic Research and Frontier Technology (cstc2015jcyjys40001) and the Light of West China Program of CAS, Key Programs of CAS (QYZDB-SSW-SYS026), and Research Program of Chongqing Municipal Education Commission (KJ1705121).

7. References

[1] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In: R. A. DeMillo et al. (eds.). *Foundations of Secure Computation*. Atlanta: Academic Press. 1978, pp. 165–179.

[2] C. Gentry. Fully homomorphic encryption using ideal lattices. In: M. Mitzenmacher (ed.). *Proceedings of the 41st STOC*. NewYork: ACM. 2009, pp. 169–178.

[3] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In: H. Gilbert (ed.). *Proceedings of EUROCRYPT 2010*. Berlin: Springer. 2010, pp. 24–43.

[4] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: P. Rogaway (ed.). *Proceedings of CRYPTO 2011*. Berlin: Springer. 2011, pp. 505–524.

[5] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In: P. Rogaway (ed.). *Proceedings of CRYPTO 2011*. Berlin: Springer. 2011, pp. 487–504.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In: S. Goldwasser (ed.). *Proceedings of the 3rd ITCSC*. New York: ACM. 2012, pp. 309–325.

[7] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In: D. Pointcheval and T. Johansson (eds.). *Proceedings of EUROCRYPT 2012*. Berlin: Springer. 2012, pp. 465–482.

[8] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In: R. Safavi-Naini and R. Canetti (eds.). *Proceedings of CRYPTO 2012*. Berlin: Springer. 2012, pp. 868–886.

[9] C. Gentry, S. Halevi, C. Peikert, and N. P. Smart. Field switching in BGV-style homomorphic encryption. *Journal of Computer Security*. 2013, **21**(5): 663–684.

[10] J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, et al. Batch fully homomorphic encryption over the integers. In: T. Johansson and P. Q. Nguyen (eds.). *Proceedings of EUROCRYPT 2013*. Berlin: Springer, 2013, pp. 315–335.

[11] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: R. Canetti and J. A. Garay (eds.). *Proceedings of CRYPTO 2013*. Heidelberg: Springer, 2013, pp. 75–92.

[12] N. P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Des., Codes and Crypt.* 2014, **71**(1): 57–81.

[13] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In: R. Safavi-Naini and R. Canetti (eds.). *Proceedings of CRYPTO 2012*. Berlin: Springer. 2012, pp. 850–867.

[14] S. Halevi and V. Shoup. HElib: An Implementation of homomorphic encryption. https://github.com/shaih/HElib.

[15] Y. Chen and G. Gong. Integer arithmetic over ciphertext and homomorphic data aggregation. In: *Proceedings of 2015 IEEE CNS – The 1st Workshop on Security and Privacy in the Cloud*. Piscataway: IEEE. 2015, pp. 628–632.

[16] C. Xu, J. Chen, W. Wu, and Y. Feng. Homomorphically encrypted arithmetic operations over the integer ring. In: F. Bao et al. (eds.). *Proceedings of ISPEC 2016*. Cham: Springer. 2016, pp. 167–181.

[17] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In: C. Cachin and T. Ristenpart (eds.). *Proceedings of the 3rd ACM WCCSW*. New York: ACM. 2011, pp. 113–124.

[18] D. Wu and J. Haven. Using homomorphic encryption for large scale statistical analysis. 2012. Available at https://crypto.stanford.edu/people/dwu4/FHE-SI_Report.pdf.

[19] J. H. Cheon, M. Kim, and M. Kim. Optimized search-and-compute circuits and their applications to query evaluation on encrypted data. *IEEE Transactions on Information Forensics and Security*. 2016, **11**(1): 188–199.

[20] C. Gentry. A fully homomorphic encryption scheme. Ph.D. dissertation, Stanford University, Stanford, 2009.

[21] S. Halevi and V. Shoup. Design and implementation of a homomorphic encryption library, 2013. Available from https://github.com/shaih/HElib.

[22] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*. 2009, **56**(6): 34:1–40.

[23] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In: H. Gilbert (ed.). *Proceedings of EUROCRYPT 2010*. Berlin: Springer. 2010, pp. 1–23.

[24] V. Shoup. NTL: A library for doing number theory. Available at http://shoup.net/ntl/.

[25] Y. P. Ofman. On the algorithmic complexity of discrete functions. *Soviet Physics Doklady*, 1963,7(7): 589–591.