PolyFHEmus: Rethinking Multiplication in Fully Homomorphic Encryption

Charles Gouert and Nektarios Georgios Tsoutsos

University of Delaware
{cgouert, tsoutsos}@udel.edu

Abstract. Homomorphic encryption is a powerful technology that solves key privacy concerns in cloud computing by enabling computation on encrypted data. However, it has not seen widespread adoption due to prohibitively high latencies. In this article, we identify polynomial multiplication as a bottleneck and investigate alternative algorithms to accelerate encrypted computing.

Keywords: Homomorphic encryption · Hardware acceleration · Polynomial multiplication · Secure computing.

1 Introduction

Cloud computing has emerged as a ubiquitous paradigm that allows companies to leverage powerful remote servers to perform important computational tasks like analytics and classification. Indeed, this allows companies to pay for computational resources on demand instead of maintaining costly local infrastucture to perform important tasks. Unfortunately, cloud computing clients yield control of their data to the cloud service provider who owns the remote server. A curious service provider can view customer data stored on their own servers provided no confidentiality mechanisms are used to protect the data. Standard encryption techniques, such as the AES encryption scheme, can be used to accomplish this task and shield sensitive data from other parties, but it also renders the data unusable by the cloud. In essence, this strategy only allows for outsourced storage, but not meaningful computation.

A unique form of encryption called *homomorphic encryption* (HE) can simultaneously solve the problem of data privacy in the cloud while also enabling arbitrary computation over ciphertext data (i.e., the encrypted data). With modern HE schemes, users can encode their data as vectors of integers or floating point numbers into a single ciphertext. HE ciphertexts typically take the form of tuples of high-degree polynomials with large coefficients and operations between ciphertexts are composed of polynomial arithmetic. As a result, operations on encrypted data are significantly slower than computing on cleartext data (i.e., unprotected data in its original and natural form). State-of-the-art implementations of HE schemes leverage hardware acceleration in the form of ASICs (application-specific integrated circuits) and GPUs to reduce the latency and improve the throughput of encrypted operations. Even still, encrypted computation remains orders of magnitude slower than equivalent operations on cleartext data.

In the following sections, we identify polynomial multiplication as a key bottleneck in HE operations and examine alternative methodologies to replace widespread techniques utilized across the board in popular open-source implementations of homomorphic encryption schemes. Specifically, we observe that we can decompose the problem of polynomial multiplication in HE to the problem of multiplying two large integers. We find that the Schönhage–Strassen algorithm is well suited for multiplying very large integers with thousands of digits and further observe that we can optimize the technique further by substituting the fast fourier transform with the discrete Galois transform. We compare our approach against the Microsoft SEAL HE library and investigate the comparative performance of our techniques across several parameter sets that align with current use-cases of HE in the research community. Overall, this work aims to accomplish three goals:

– Analyze the performance of low-level HE primitive operations to definitively outline current performance bottlenecks;

Fig. 1. Secure Outsourcing with HE: The client encrypts sensitive data and then transmits the ciphertext over a network to the remote cloud server. The cloud server can then execute an algorithm over the ciphertexts and generate an encrypted answer, which is transmitted back to the client. Lastly, the client can decrypt and receive the result of the computation. No information about the inputs or outputs is revealed to the cloud.

- Propose a new strategy that adopts Kronecker substitution and the Schönhage–Strassen integer multiplication algorithm to enable faster polynomial multiplication in HE;
- Demonstrate meaningful speedups of HE primitives relative to the state-of-the-art.

2 Background

Before examining high-level implementation details and performance characteristics of state-of-the-art HE schemes, we will take a step back and look at the overarching types of homomorphic encryption schemes and what each form is capable of accomplishing. Additionally, the explicit threat model that we consider, which consists of identical assumptions to other works that rely on strictly HE, is outlined at the end of this section.

2.1 Homomorphic Encryption

All types of HE incorporate *malleable* ciphertexts that can be modified by computation to affect the underlying message of the ciphertext. However, not all HE schemes exhibit the same capabilities in terms of the number and types of operations that can be computed on the encrypted data. In general, all HE implementations can be delegated into one of three categories: partial HE (PHE), leveled HE (LHE), and fully HE (FHE).

PHE allows for one type of arithmetic operation over ciphertexts: unlimited addition or multiplication. While PHE schemes are typically efficient relative to the stronger forms of HE, the restricted capability relegates them to simple, niche applications. PHE will not be a focus of this article for this very reason, as it is only well-suited to a limited number of applications. We do note that, in some contexts, PHE can be combined with other privacy-preserving constructions, such as secure multi-party computation protocols, to realize more complex algorithms [13].

The next form of HE, called LHE, allows for both addition and multiplication in the encrypted domain. These capabilities constitute a functionally complete set of operations, meaning that any algorithm can be constructed with these arithmetic primitives. Ciphertexts take the form of tuples of high-degree polynomials and unlike PHE algorithms, most LHE schemes derive their security from a hard mathematical problem known as learning with errors (LWE) [14]. This problem is akin to solving a series of equations that boils down to finding s for all i instances: $b_i = a_i \cdot s_i + e_i$, where b_i and a_i are known and e_i is a small random error. At a high-level, plaintext values are converted to polynomials upon encryption and small errors are added to the coefficients. An unfortunate consequence of this is that the noise compounds as operations are conducted on the encrypted data. Specifically, ciphertext addition (which is composed of addition between the polynomials of the first ciphertext with the corresponding polynomials of the second ciphertext) causes the noise to grow linearly while multiplication causes exponential growth [4]. An important caveat of multiplication is that multiplying tuples of polynomials yields a product that is larger than the two inputs. Assume we have two ciphertexts $ct_x = (a_0, b_0)$ and $ct_y = (a_1, b_1)$ where a_n and b_n are polynomials. When we multiply these, we end up with $ct_z = (a_0 \times a_1, a_0 \times b_1 + a_1 \times b_0, a_1 \times b_1)$, which introduces an extra polynomial. Therefore,

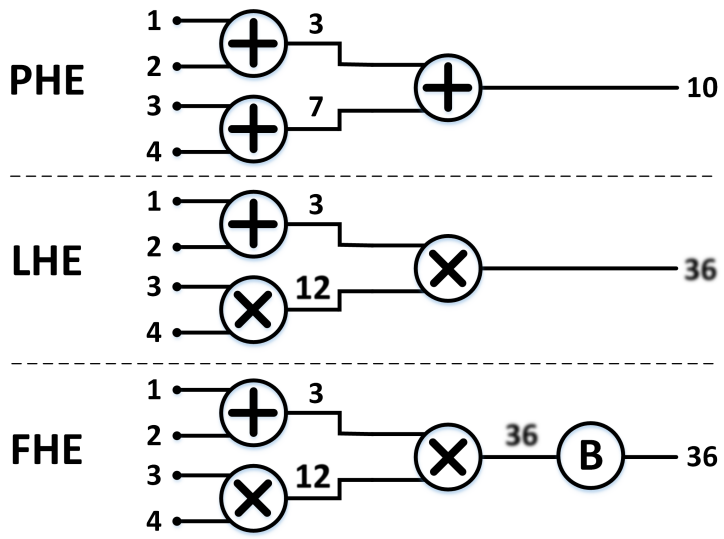

Fig. 2. Types of Homomorphic Encryption: PHE can only perform one type of operation, but is noiseless. On the other hand, LHE can perform both multiplication and addition, but accumulates noise and eventually the message becomes corrupted. Lastly, FHE has the same capabilities of LHE, but can refresh the ciphertext noise with a bootstrapping operation ("B" in the diagram).

the ciphertexts will continue to grow after each multiplication operation, which will cause an explosion in terms of both execution time and memory consumption. Luckily, an operation called key-switching (which consists primarily of polynomial multiplications with public key material) can map the 3-tuple product ciphertext back to a configuration matching the original dimensions of both inputs (i.e., 2-tuple). However, this operation results in noise growth and further exacerbates the noise problem.

Eventually, the noise will exceed a certain threshold after which it will begin corrupting the underlying message. In practice, this means that the final decryption will yield a non-deterministic answer (and will be incorrect with high probability). A noise mitigation measure called *modulus switching* can be used to somewhat reduce the magnitude of the accumulated noise, allowing for more operations. Notably, this operation reduces the size of the polynomial coefficients by a fixed amount and can only be used a finite number of times (eventually the coefficients cannot be decreased in size any more). Intuitively, increasing the size of the coefficients allows us to do more modulus switching, but this has a negative impact on security and can only be balanced by increasing the degree of the ciphertext polynomials. Therefore, we have to balance the number of modulus switches, or levels, with the ciphertext size and cost of polynomial operations. For applications that have a large depth, meaning that the number of subsequent multiplication operations over a ciphertext is high, LHE exhibits poor scalability as ciphertexts become unmanageably large and operations become more computationally intensive. This is because an LHE implementation needs to keep increasing the coefficient size to allow for more modulus switching operations, resulting in larger ciphertexts. As discussed, increasing the coefficient size also has the negative side effect of decreasing the overall security level: if the coefficient size is doubled, the polynomial degree of the ciphertexts must also be doubled to preserve the original security level. Therefore, the size of the ciphertexts increases in two dimensions to accommodate more possible modulus switching operations to evaluate a larger depth (i.e., more multiplications). This also makes the homomorphic operations significantly more expensive, as both the coefficients and the degree of the polynomials become larger.

LHE schemes can also support a technique called "batching", which allows users to encrypt vectors of plaintext values into a single ciphertext. As a case in point, the CKKS cryptosystem allows for vectors of up to $N/2$ floating point numbers to be encoded in a single ciphertext, where N is the chosen polynomial degree. Notably, the size of a batched ciphertext is identical to that of a ciphertext encrypting only a single plaintext

value. Computing upon these batched ciphertexts is akin to using SIMD (single-instruction multiple-data) operations, where each plaintext element is affected individually. Specifically, adding two batched ciphertexts behaves like vector addition and multiplying behaves like computing an element-wise product between the vectors.

The third and most computationally powerful form of HE is fully homomorphic encryption (FHE), which allows for unbounded addition and multiplication. Most FHE schemes also rely on the LWE problem and are hampered by the same noise growth issues as LHE schemes. However, a mechanism called *bootstrapping* is capable of refreshing the noise and can be invoked an unlimited number of times (unlike modulus switching). In fact, any LHE construction can become FHE with the introduction of this mechanism. While the relative latency of bootstrapping is significantly higher compared to all other HE operations, FHE evaluation exhibits superior scalability compared to LHE for applications that exhibit a very large multiplicative depth, such as neural networks with several layers. In effect, FHE mitigates the scalability problem as a single parameter set that supports bootstrapping can evaluate arbitrarily deep programs.

In this article, we compare against the popular Microsoft SEAL homomorphic encryption library [15], which supports strictly LHE contexts. Specifically, SEAL supports the BGV [4], BFV [7], and CKKS [5] cryptosystems. Both BGV and BFV are used to encrypt vectors of integers, while CKKS encrypts vectors of complex or floating point numbers. All three cryptosystems can be implemented as FHE constructions, but without hardware acceleration, the bootstrapping operation is prohibitively expensive for these schemes. Even though SEAL only supports LHE contexts, we remark that our proposed techniques outlined in the following sections are equally applicable to FHE, as the core primitive operations are similar and consist primarily of large polynomial operations.

2.2 Threat Model

Similarly to other works that rely solely on homomorphic encryption primitives, we assume that the computing party (i.e., the remote server owned by the cloud service provider) is *honest-but-curious*. This means that the cloud provider will faithfully execute the desired algorithm on the encrypted data, but has an incentive to view the sensitive client information being manipulated. In this scenario, homomorphic encryption instantiated with the proper, secure parameters ensures complete data confidentiality of inputs, intermediate results, and outputs. The only information that the cloud can glean is potentially the relative sizes of the plaintext of the encrypted data. As a simple example, assuming that a non-batching context is used and only scalars are encrypted, the cloud can infer that the plaintext consists of N integers or floating point numbers if the client sends N ciphertexts as input.

3 Faster Polynomial Arithmetic for HE Operations

We begin our investigation by profiling the Microsoft SEAL BFV implementation for a small degree-2 polynomial approximation consisting of a ciphertext multiplication and a ciphertext addition. With the `kcachegrind` tool, we find that approximately 40% of the evaluation time consists of the ciphertext multiplication, while the addition constitutes a negligible percentage of the overall runtime (i.e., $< 1\%$). The fast speed of the addition is due to the fact that the only operations involved include a set of element-wise modular additions across each pair of ciphertext polynomials. In fact, the modular reduction in this case is simplified relative to the reduction employed by the multiplication. By taking advantage of the fact that the sum will be less than twice the modulus, SEAL performs a simple check to see if the sum is greater than the modulus and subtracts the modulus if it is (otherwise no reduction is required). Additionally, the element-wise modular additions are embarrassingly parallel, which can be exploited to achieve very fast speeds. The remaining runtime percentage of the program consists primarily of generating keys, which we note is a one-time cost (the keys can be re-used for future HE programs assuming the same parameter set can be utilized).

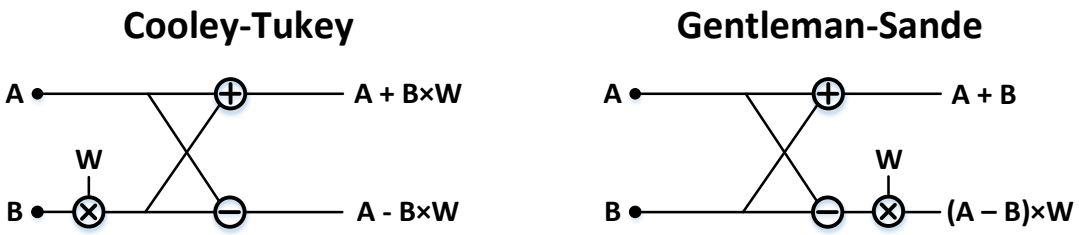

Fig. 3. A visual depiction of the two most popular DFT "butterflies": the Cooley-Tukey and Gentleman-Sande constructions. These are referred to as butterflies due to the shape created by the intersecting paths from the top and bottom wires. Both of these illustrate a radix-2 butterfly, which can be is used as the core primitive block of larger DFTs. A and B are both data inputs and W represents a twiddle factor, which is a multiplicative constant used in DFT computations.

3.1 Current Strategies for Polynomial Multiplication

The textbook method of multiplying polynomials is the most straightforward algorithm, but also exhibits the worst asymptotic performance ($\mathcal{O}(N^2)$) as each coefficient of the first polynomial needs to multiplied with every coefficient of the second polynomial. Considering that homomorphic ciphertext polynomials can plausibly contain up to 2^{17} coefficients, the poor scalability of this approach results in extremely significant performance penalties. Instead, the majority of HE libraries use an alternative approach with better scalability for large polynomial degrees based on the discrete Fourier transform (DFT).

This technique involves using a forward transform to convert the polynomials to the Fourier domain, where a polynomial multiplication translates to a point-wise multiplication between the coefficients. Ciphertexts are typically kept in this Fourier representation until an operation needs to be conducted in the original coefficient representation (e.g., for certain steps of keyswitching). Overall, this results in an asymptotic complexity of $\mathcal{O}(N \log N)$, which significantly outperforms the textbook, naive approach for the large polynomial degrees required for secure HE evaluation. To execute the DFT, two of the more widely used options are the fast Fourier transform (FFT) and number theoretic transform (NTT). Among these, most HE libraries opt for the NTT as it works naturally over the integers, whereas the FFT requires the coefficients of the ciphertext polynomials to be mapped to the floating point domain. Working in the floating point domain also has the negative side effect of rounding errors, which can exacerbate noise growth during computation. With these configurations, the core bottleneck of computation is the DFT; to give additional context about the scale of this problem, a dot product between two encrypted vectors of length 500 in the CGGI FHE cryptosystem results in over a billion NTT invocations [9].

3.2 Polynomial Multiplication with the Discrete Galois Transform

An alternative to both the FFT and NTT for computing the DFT is the discrete Galois transform (DGT). Indeed, this transform can offer significant advantages over the other DFT techniques as it allows us to cut down the overall transform size to $N/2$ instead of N [11]. Additionally, the DGT has been observed to have lower memory bandwidth requirements and higher arithmetic intensity in the context of GPU acceleration relative to state-of-the-art NTT and FFT constructions [2]. Similarly to the NTT, the DGT operates directly over integers, which is more natural in the context of the ciphertext polynomials with modular integer coefficients. However, the key difference is that the NTT operates in the finite field $\mathbb{F}_p$, where p is a constituent prime of the ciphertext modulus. On the other hand, the DGT operates in $\mathbb{F}_{p^2}$ and works over Gaussian integers, which contain real and imaginary parts in $\mathbb{Z}_p$ (i.e., the set of integers modulo p). In this case, we have that addition and subtraction of two Gaussian integers involve adding or subtracting the real parts and the imaginary parts independently, which can be done in parallel. Multiplication, similarly to addition and subtraction, is derived from the equivalent operations on regular complex numbers. For Gaussian integers $x = a + bi$ and $y = c + di$, the product of the two is defined as $z = (a \cdot c - b \cdot d) + (a \cdot d + b \cdot c)i$.

The DGT can utilize the same general data paths as the FFT or NTT, of which there are two primary variations: Cooley-Tukey [6] and Gentleman-Sande [8] butterflies, which are depicted in Figure 3. Both algorithms are very similar and utilize a constant known as a "twiddle factor". Twiddle factors are always complex roots of unity, where the nth root of unity is defined as $W_n = e^{-2\pi i/N}$. We note that the twiddle factors used in the computation of the DFT can be pre-generated prior to any homomorphic evaluation. The Cooley-Tukey construction multiplies the second input (i.e., B in the figure, which represents a ciphertext polynomial coefficient) by the twiddle factor before the sum and difference are computed. On the other hand, the Gentleman-Sande butterfly performs this multiplication after the difference of the coefficients A and B are computed. In practice, the Cooley-Tukey butterfly can be used to accomplish the forward transform while the Gentleman-Sande butterfly achieves the inverse transform [3, 12].

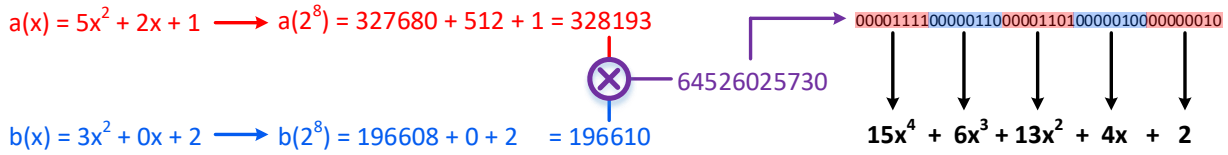

Fig. 4. Kronecker Substitution: Any polynomial can be converted to an integer representation by evaluating it at a power of 2 that is larger than any coefficient. After both input polynomials are converted, they can be multiplied in the integer domain. Then, the coefficients of the product polynomial can be derived directly from the binary representation of the product of the integer multiplication.

3.3 Schönhage–Strassen: Fast Multiplication for Big Integers

There have been a plethora of proposed algorithms for multiplication of very large integers (i.e., hundreds or thousands of digits long). Similarly to polynomial multiplication, the computational complexity of multiplying two integers of n digits is $\mathcal{O}(n^2)$. The first approach exhibiting better asymptotic complexity was Karatsuba in 1960, which achieves a complexity of $\mathcal{O}(n^{\log_2 3})$. Shortly after, the Toom-Cook algorithm generalized Karatsuba and achieved a computational complexity of $\mathcal{O}(n^{\log_k(2k-1)})$, where k is a parameter that refers to the number of chunks the input integers are broken up into. We note that $k = 2$ corresponds exactly to the Karatsuba algorithm, with the most popular variant being $k = 3$ and yielding an overall complexity of approximately $\mathcal{O}(n^{1.465})$.

While both Karatsuba and Toom-Cook outperform the standard integer multiplication algorithm, better techniques have emerged for extremely large integers with several thousand digits. The Schönhage–Strassen (SS) algorithm, introduced in 1971, utilizes the FFT recursively and achieves a complexity of $\mathcal{O}(n \times \log n \times \log(\log n))$, which is more efficient than both Karatsuba and Toom-Cook ($k = 3$) for a very large number of digits.

An immediate question is how the SS algorithm may be used in the context of efficient polynomial multiplication, since the size of the coefficients in HE ciphertexts are rarely larger than 2300 bits in length [10]. Our key observation is that there exists a mechanism that allows us to reduce the problem of the multiplying two polynomials to the problem of multiplying two integers by leveraging the *Kronecker substitution* technique that converts polynomials to integers. A simple example is depicted in Figure 4, which demonstrates multiplying the polynomial $5x^2 + 2x + 1$ with $3x^2 + 2$. Both polynomials are converted to the integer domain by evaluating at $x = 2^8$ and then multiplied. Finally, the expected product can be recovered by mapping chunks of 8 bits to the corresponding coefficient. As a case in point, the most significant byte of the integer product is equal to `00001111`, which maps to the polynomial term $15x^4$. With this approach, we can convert polynomials to large integers to compute the SS multiplication and then convert back to the polynomial domain.

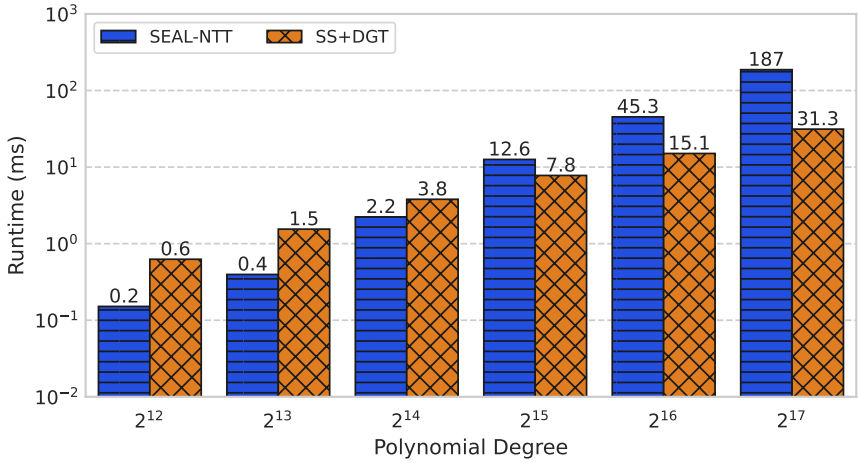

Fig. 5. Encrypted Multiplication Latency: This graph outlines the average latency of performing an encrypted multiplication across two "fresh" ciphertexts (i.e., ciphertexts that have not yet been computed upon). Notably, we strictly measure the multiplication step for both SEAL and our proposed approach. While we allow both programs to use as many threads as necessary, we remark that the SEAL library only utilizes a single thread.

3.4 Our Contribution: DGT using SS

Recall that the original Schönhage–Strassen utilizes the FFT construction to compute the integer multiplication; however, the FFT can easily be swapped out for another transform that can accomplish the general DFT. Since the DGT only requires a transform of half the size relative to the FFT, we can swap out the invocations to the FFT with DGT invocations. Next, we note that there are two primary methods to compute the polynomial multiplication. The first involves computing the SS+DGT algorithm across the ciphertext polynomials with very large coefficients (i.e., hundreds or thousands of bits in length). However, we observe that the forward Kronecker substitution, which involves evaluating the input polynomials with extremely large powers of 2, is quite costly. Even though each term of the polynomial can be shifted in parallel, summing all of the huge shifted coefficients becomes the bottleneck.

Instead, we can exploit more parallelism by adopting the residue-number system (RNS) to break up the original ciphertext polynomials into several polynomials with significantly smaller coefficients. After evaluation, these smaller polynomials can be combined to regenerate a single polynomial with large coefficients; this is made possible by exploiting the structure of the coefficient modulus, which is a product of primes: $q = p_0 \times p_1 \ldots \times p_N$, where q is the ciphertext modulus and p_i is a prime. Each constituent prime is typically chosen to be between 40 and 64 bits in length and the RNS decomposition will give us N polynomials, where N is the number of prime factors of the modulus. Now, the ith RNS polynomial will consist of coefficients modulo the prime p_i instead of q, allowing us to work with coefficients that are more naturally suited for the native word sizes supported by modern computers. Another key benefit of our proposed strategy is that it enables us to compute the SS+DGT across each polynomial with smaller coefficients in parallel, which can be recombined later to recover the original polynomial representation. Indeed, a similar approach is adopted by most modern HE libraries such as Microsoft SEAL [15] and OpenFHE [1].

4 Experimental Results

Our proposed approach has been implemented as an open-source multi-threaded C++ library that provides an interface for converting to and from polynomials (represented as a vector of integers) and arbitrary precision integers via Kronecker substitution, as well as a multiplication mechanism that operates across two integers of arbitrary precision. We compare our approach with the current polynomial multiplication strategy implemented in the Microsoft SEAL library, which utilizes the number theoretic transform (NTT)

to allow for pointwise multiplication. All experiments were ran on an `r5.24xlarge` AWS server with 784 GB of RAM and 96 vCPUs across parameter sets corresponding to 128 bits of security under current attack models. Our smallest parameter set utilizes a polynomial degree of 2^{12}, which allows for approximately 1-2 multiplications. We remark that smaller parameters are possible, but these are rarely used for the BGV, BFV, and CKKS cryptosystems. As the polynomial degree doubles, we also double the coefficient modulus, resulting in ciphertexts that are $4\times$ larger. Therefore, the ciphertexts generated using our largest parameter set with a polynomial degree of 2^{17} are $1024\times$ larger than the smallest parameter set we consider. These parameter sets run the entire gamut of plausible parameter sizes that are in use today for all use-cases. The results of our evaluations are depicted in Figure 5.

We observe that for smaller polynomial degrees SEAL outperforms our SS+DGT approach by a factor of $3\times$ for 2^{12}-degree polynomials and nearly $4\times$ for 2^{13}-degree polynomials. This confirms what we expect, as the SS multiplication is well-suited for very large inputs and performs suboptimally for smaller inputs. Therefore, for applications that only exhibit a low multiplicative depth (and can be executed correctly with small parameters) the NTT approach is a better choice, and applications such as matrix multiplication, logistic regression inference, and facial recognition via the squared Euclidean distance fall under this category. On the other hand, the SS+DGT multiplication begins to outperform the baseline SEAL approach when the polynomial degree grows over 2^{15} and offers significantly better scalability for larger sizes.

We expect this trend to continue for even larger polynomial degrees, yet in all current FHE cryptosystems, bootstrapping is typically implemented in the range of 2^{15} to 2^{17} for the polynomial degree (where our approach outperforms the state of the art). Further, the ciphertext sizes become infeasibly large after this range and the key generation becomes prohibitively expensive in terms of required RAM and execution time. This is primarily due to the complexity of computing the public key material used for operations such as key switching, which is required in order to perform multiplication operations. Overall, we remark that our proposed SS+DGT approach is a major improvement over the NTT-based polynomial multiplication for complex applications that either require bootstrapping or otherwise exhibit a large depth (e.g., deep neural networks, encrypted sorting, and image processing).

5 Concluding Remarks

In this article, we propose a new strategy for evaluating polynomial multiplication in the context of the homomorphic encryption. As this operation forms a key bottleneck of current HE schemes, optimizing it has large ramifications for the overall performance of encrypted applications. While many current HE libraries utilize the number theoretic transform to facilitate polynomial multiplication, we propose an efficient integer multiplication algorithm, namely the Schönhage–Strassen algorithm, and combine this approach with the discrete Galois transform using Kronecker substitution. We report that our proposed approach outperforms the popular Microsoft SEAL library for large polynomial degrees commonly used for general computation with homomorphic encryption.

Acknowledgments

This work has been supported by NSF Award #2239334.

References

1. Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 53–63, 2022.
2. Pedro Geraldo MR Alves, Jheyne N Ortiz, and Diego F Aranha. Faster homomorphic encryption over gpgpus via hierarchical dgt. In *International Conference on Financial Cryptography and Data Security*, pages 520–540. Springer, 2021.

3. Jonas Bertels, Michiel Van Beirendonck, Furkan Turan, and Ingrid Verbauwhede. Hardware acceleration of fhew. In *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 57–60. IEEE, 2023.
4. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
5. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.
6. James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
7. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.
8. W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966.
9. Charles Gouert, Vinu Joseph, Steven Dalton, Cedric Augonnet, Michael Garland, and Nektarios Georgios Tsoutsos. Arctyrex: Accelerated encrypted execution of general-purpose applications. *arXiv preprint arXiv:2306.11006*, 2023.
10. Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 114–148, 2021.
11. Guangyan Li, Donglong Chen, Gaoyu Mao, Wangchen Dai, Abdurrashid Ibrahim Sanka, and Ray CC Cheung. Algorithm-hardware co-design of split-radix discrete galois transformation for kyberkem. *IEEE Transactions on Emerging Topics in Computing*, 2023.
12. Suraj Mandal and Debapriya Basu Roy. Kid: A hardware design framework targeting unified ntt multiplication for crystals-kyber and crystals-dilithium on fpga. In *2024 37th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID)*, pages 455–460. IEEE, 2024.
13. Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T Lee, Hsien-Hsin S Lee, Gu-Yeon Wei, and David Brooks. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 26–39. IEEE, 2021.
14. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.
15. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.