Design of a Flexible Schönhage-Strassen FFT Polynomial Multiplier with High-Level Synthesis to Accelerate HE in the Cloud

Kevin Millar
Department of Computer Engineering
Rochester Institute of Technology
kdm8162@rit.edu

Marcin Łukowiak
Department of Computer Engineering
Rochester Institute of Technology
mxleec@rit.edu

Stanisław Radziszowski
Department of Computer Science
Rochester Institute of Technology
spr@cs.rit.edu

***Abstract*—Homomorphic Encryption (HE) allows for encrypted data to be sent to, stored, and operated on by untrusted parties without the risk of privacy compromise. The benefits and applications of HE are far reaching, especially in regard to cloud computing. However, current HE solutions require a large number of resource intensive arithmetic operations such as high precision, high degree polynomial multiplication. This work aims to accelerate the multi-precision arithmetic operations used in HE with specific focus on an implementation of the Schönhage-Strassen Fast Fourier Transform (FFT)-based multiplication algorithm. It is planned to be incorporated into a larger HE library of arithmetic functions tuned for High-Level Synthesis (HLS) that enables flexible solutions for hardware/software systems on reconfigurable cloud resources. The developed FFT based polynomial multiplier exhibits flexibility in the selection of HE security parameters facilitating its use in a wide range of schemes and applications. The design yields substantial speedup over the polynomial multiplication functions implemented in the Number Theory Library (NTL) utilized by software based HE solutions.**

***Index Terms*—HE, HLS, FPGA, cloud computing.**

I. Introduction

As cloud computing grows in popularity, solutions like Amazon Web Services (AWS) [1] are becoming more desirable as an affordable means by which to utilize computing resources. Though cloud resources offer many benefits, the offloading of private data to third party systems for computation introduces new risks. Conventional cryptographic solutions do not solve this problem as protected data requires decryption to allow for operation on these shared computing resources. A potential solution to this problem is Homomorphic Encryption (HE) which allows for operations to be performed on encrypted data without exposing the underlying plaintext to untrusted parties. This removes the necessity of decrypting data before operation and retains the privacy of the data even when evaluated on cloud resources. HE schemes do exist, but they require a large number of complex arithmetic operations making current solutions computationally expensive and resource intensive. This work accelerates the resource intensive arithmetic operations heavily relied upon in many HE schemes. This is achieved through the continued development of a library containing arithmetic functions accelerated through HLS to allow for the flexible design of hardware/software systems on reconfigurable cloud resources. The development flow of an application utilizing this library is shown in Figure 1.

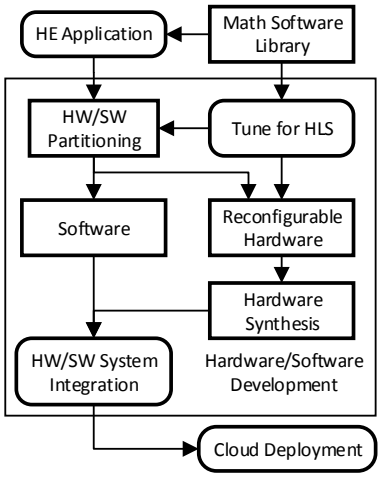

Fig. 1. Design flow of an HE application utilizing an HLS library

This diagram shows the design flow of an HE application in which the computationally intensive operations are partitioned for hardware acceleration through the HLS accelerated math library. These operations are then designated for execution on a hardware co-processor that is synthesized for the available target reconfigurable resources. Integration is performed by establishing communication between the main program running on a conventional CPU and the reconfigurable resources, and the full system is deployed within a cloud environment. This work focuses explicitly on the design and development of a flexible FFT based polynomial multiplier through HLS to offer improved performance over software solutions.

The organization of this paper is as follows: Related works are presented in Section II. Section III discusses the background of HE and FFT based polynomial multiplication. The HLS design of the FFT based polynomial multiplier is explored in Section IV. The results are analyzed in Section V, and Section VI presents the conclusions.

978-1-7281-1957-1/19/$31.00

II. Related Work

The hardware design of large-scale FFT multipliers for HE has been explored in various papers such as [2], [3], [4]. Each of these works designed custom hardware targeting either Application Specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA) platforms using conventional Hardware Description Language (HDL) design techniques. The work presented in [5] developed a flexible Karatsuba multiplier with Vivado HLS achieving a theoretical speedup up to 136 times over the Fast Library for Number Theory (FLINT) arithmetic software library. A similar accelerator for HE was created using the Karatsuba multiplication algorithm through the application of hardware/software co-design techniques in [6], specifically targeting the Fan-Vercauteren (FV) HE scheme introduced in [7]. A hardware accelerated FFT algorithm for HE was designed in [8] with Vivado HLS achieving a speedup of 6.9 times the same algorithm run on an Intel Core i7-5600U CPU at 2.6GHz. Mkhinini et. al designed a flexible residue number system (RNS) based large polynomial multiplier through HLS for HE and achieved significant speedup over software implementations [9], [10], [11].

III. Background

Homomorphic Encryption: HE cryptographic schemes allow for operations to be performed on encrypted data. Though Somewhat Homomorphic Encryption (SHE) schemes allowing for a limited number of operations on encrypted data have existed for a long time, the first Fully Homomorphic Encryption (FHE) scheme eliminating this constraint was introduced by Gentry in 2009 [12]. This breakthrough was achieved through a scheme based on bootstrapping, the principle by which an encryption scheme evaluates its own decryption circuit. Gentry later assisted in the development of a more efficient FHE scheme called Brakerski-Gentry-Vaikuntanathan (BGV) by utilizing a novel modulus switching technique [13], [14]. The BGV scheme is more efficient than previous schemes and can operate securely based on the Learning with Errors (LWE) or Ring Learning with Errors (RLWE) hardness assumptions.

Ring Learning with Errors: The RLWE problem is an extension of the LWE problem over algebraic rings first introduced in [15]. A basic definition of the RLWE problem is presented here for simplicity. Select a dimension $n \geq 1$ where n is a power of 2, a prime modulus $q \geq 2$ such that $1 = q \mod 2n$, and $f(x) = x^n + 1 \in \mathbb{Z}[x]$. Let $\mathbb{R} = \mathbb{Z}[x]/\langle f(x)\rangle$, $\mathbb{R}_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$, and χ be an error distribution. Establish a uniformly random secret polynomial $\mathbf{s} = \mathbf{s}(x) \in \mathbb{R}_q$. Choose a polynomial $\mathbf{a} = \mathbf{a}(x) \in \mathbb{R}_q$ uniformly at random, generate $e = e(x) \in \mathbb{R}_q$ based on the error distribution χ, and output$(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e)$. The RLWE problem is that given an arbitrary number of samples of the form $(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + e)$, it is computationally infeasible to determine $\mathbf{s}$ [16], [15].

Security: The security of RLWE is currently an area of research. It is heavily dependent upon the noise e added to the coefficients along with the selection of n and the prime modulus q. An ongoing effort is being made to standardize HE schemes based on RLWE for designated cyclotomic rings.

TABLE I
Parameters for RLWE-based HE Schemes

n	λ	$\log q$	Size (bits)
1,024	256	19	19,456
1,024	192	22	22,528
1,024	128	31	31,744
2,048	256	33	67,584
2,048	192	42	86,016
2,048	128	58	118,784
4,096	256	62	253,952
4,096	192	80	327,680
4,096	128	113	462,848
8,192	256	123	1,007,616
8,192	192	157	1,286,144
8,192	128	223	1,826,816
16,384	256	243	3,981,312
16,384	192	310	5,079,040
16,384	128	443	7,258,112
32,768	256	481	15,761,408
32,768	192	616	20,185,088
32,768	128	886	29,032,448

The analysis in [17] presents recommended parameters n and q for various levels of the security parameter λ. In this work we adopt a selection of these parameters as shown in Table I. These recommendations are made based on the LWE-estimator tool introduced in [18] which determines parameters for a given λ based on the estimated complexities of currently known attacks on RLWE. The typical plaintext is composed of n bits with a single bit encoded within each coefficient, but other packing techniques may be used.

FFT Modular Polynomial Multiplication: The Schönhage-Strassen algorithm first introduced multiplication of n-bit integers using the FFT with complexity of $O(n \log n \log\log n)$ [19], [20]. The heart of the algorithm recursively performs modular polynomial arithmetic on decomposed integers through an application of negative wrapped convolution, which is equivalent to polynomial multiplication modulo $x^n + 1$ [21]. Let ω be a primitive n-th root of unity and $\theta^2 = \omega$, then with

$$\hat{a} = (a_0, \theta a_1, \ldots, \theta^{n-1} a_{n-1}),$$
$$\hat{b} = (b_0, \theta b_1, \ldots, \theta^{n-1} b_{n-1}),$$

the negative wrapped convolution can be computed as

$$c = IFFT^{-1}(FFT(\hat{a}) \cdot FFT(\hat{b})).$$

Polynomial multiplication is performed with coefficients modulo a prime q of the form $q = 1 \mod 2n$ by selecting θ such that $\theta^2 = \omega \mod q$. This method to perform polynomial multiplication with the coefficients modulo q and the result modulo $x^n + 1$ is shown in Algorithm 1. In theory, the Schönhage-Strassen algorithm is applied recursively at each

multiplication to a depth such that the operands become small enough in size for a simpler multiplication algorithm to become more efficient. In this work, only the top layer of the algorithm was necessary so the multiplication performed in line 8 is not decomposed any further.

Algorithm 1 Schönhage-Strassen Polynomial Multiplication

Input: Polynomials $a(x)$ and $b(x)$ of maximum degree n with coefficients $a_i, b_i \in \mathbb{Z}_q$ for $i = 0, 1, \ldots, n-1$

Output: $c(x) = a(x) \cdot b(x) \mod (x^n + 1)$

1: Precalculate all powers of θ, ω, θ^{-1}, and ω^{-1} modulo q
2: weight_coeff : **for** $i \leftarrow 0$ **to** $n-1$ **do**
3: $\quad a_i \leftarrow a_i \cdot \theta^i \mod q$
4: $\quad b_i \leftarrow b_i \cdot \theta^i \mod q$
5: $a \leftarrow$ FFT(a, ω)
6: $b \leftarrow$ FFT(b, ω)
7: mult_coeff : **for** $i \leftarrow 0$ **to** $n-1$ **do**
8: $\quad c_i \leftarrow a_i \cdot b_i \mod q$
9: $c \leftarrow$ IFFT(c, ω^{-1})
10: unweight_coeff : **for** $i \leftarrow 0$ **to** $n-1$ **do**
11: $\quad c_i \leftarrow c_i \cdot \theta^{-i} \mod q$

IV. HLS Design

The modular polynomial multiplication algorithm outlined in Algorithm 1 was implemented in C++ targeting Xilinx FPGAs through Vivado HLS. High-level block diagram of the target circuit is shown in Figure 2. This image shows

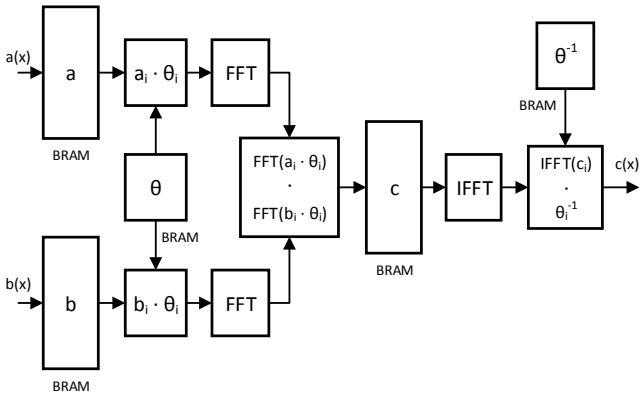

Fig. 2. Block diagram of the FFT circuit obtained from Algorithm 1

the storage of the input polynomials a and b into Block RAMs (BRAMs) and the operations necessary to produce the resulting product polynomial c. As the primitive n-th root of unity ω and the weighting parameter θ are not guaranteed to be powers of two, modular reduction of the coefficients was performed through Barrett's reduction algorithm. Furthermore, because the algorithm requires consecutive powers of ω and θ, and their inverses ω^{-1} and θ^{-1}, all modulo q, BRAMs were loaded with all precalculated powers of these values to prevent the expensive task of computing them in real time. These additional memories provided further benefits as the Inverse Fast Fourier Transform (IFFT) typically requires an additional step in which each of the vector elements is multiplied by $n^{-1} \mod q$. An optimization was performed to remove this step by precalculating the multiplicative inverse of n modulo q and multiplying each consecutive power of θ^{-1} by this value. The reduction of each resulting value modulo q was then stored in the BRAM and used in the unweighting step to perform both operations concurrently and retrieve the final result.

HLS Pipelining: To allow pipelining of the FFT component, a "ping-pong" buffer was introduced such that the operating memory was not both read from and written to within a single cycle. This was achieved through the addition of two memories to the FFT. The basic operation of the FFT with the "ping-pong" buffer is shown in Figure 3.

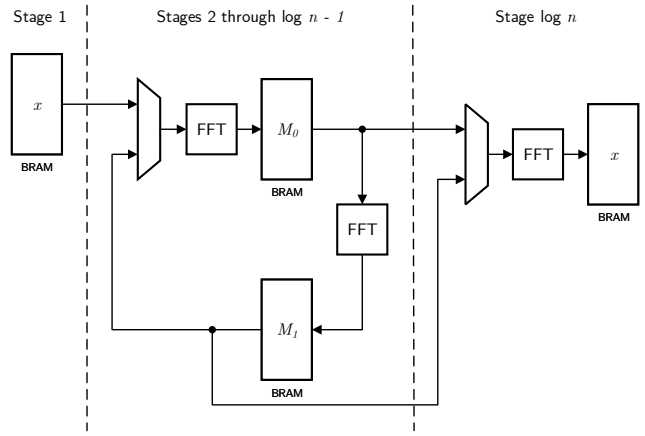

Fig. 3. Basic block diagram of the FFT with "ping-pong" memory buffer

The block diagram shows read from one memory and write to the other, swapping in functionality each stage, ensuring that all read and write operations were scheduled without contention. This pattern continues until the last stage of the FFT at which point the last memory to have been written was read and the results stored back into the input memory for further operation. The C++ source was modified so that the HLS tool would generate this desired hardware as outlined in Algorithm 2. The FFT functions were updated to include the "ping-pong" memory buffer through the addition of a 2D array. By default, HLS does not recognize these as two independent memories because they are declared under the same array structure. To ensure that each memory was implemented as its own BRAM, the HLS ARRAY_PARTITION directive was applied. The first iteration of the *fft_stage* was unrolled with the data read from the input array and the result stored in the first memory. The main *fft_stage* loop was modified to negate the address bit of the "ping-pong" array to swap the role of each memory between iterations. The final stage was also unrolled to read the data from the last written memory and write the result to the input array. The HLS PIPELINE directive was added to the *fft_stage_1*, *fft_group*, and *fft_stage_log_n* loops with a target Initiation Interval (II) of 1. To avoid data dependencies between stages, an HLS

Algorithm 2 FFT with "Ping-Pong" Memory Buffer

Input: Polynomial $a(x)$ of degree n with coefficients $a_i \in \mathbb{Z}_q$ for $i = 0, 1, \ldots, n-1$
Output: Transformed polynomial $a(x)$

1: $m = n/2$; $j = \log n - 1$; $addr = 0$
2: fft_stage_1 : **for** $k \leftarrow 0$ **to** $n/2 - 1$ **do**
3: $\quad i \leftarrow k$; $b \leftarrow k$
4: $\quad M_{addr,i} \leftarrow a_i + a_{i+m}$
5: $\quad M_{addr,i+m} \leftarrow \omega^b(a_i - a_{i+m})$
6: $m \leftarrow m/2$; $j \leftarrow j - 1$
7: fft_stage : **for** $s \leftarrow 1$ **to** $\log n - 1$ **do**
8: $\quad$ fft_group : **for** $k \leftarrow 0$ **to** $n/2 - 1$ **do**
9: $\quad\quad i \leftarrow k + \lfloor k/2^j \rfloor \cdot 2^j$
10: $\quad\quad b \leftarrow k \cdot 2^s$
11: $\quad\quad M_{\neg addr,i} \leftarrow M_{addr,i} + M_{addr,i+m}$
12: $\quad\quad M_{\neg addr,i+m} \leftarrow \omega^b(M_{addr,i} - M_{addr,i+m})$
13: $\quad m \leftarrow m/2$; $j \leftarrow j - 1$; $addr \leftarrow \neg addr$
14: fft_stage_log_n : **for** $k \leftarrow 0$ **to** $n/2 - 1$ **do**
15: $\quad i \leftarrow k \cdot 2$
16: $\quad a_i \leftarrow M_{addr,i} + M_{addr,i+m}$
17: $\quad a_{i+m} \leftarrow M_{addr,i} - M_{addr,i+m}$

UNROLL directive was applied to the *fft_stage* loop to ensure that the HLS tool would pipeline each iteration of the *fft_group* loop independently.

HLS Loop Unrolling: The inner loop operations within the multiplier performed computations between elements of the array in place and could potentially be performed fully in parallel. However, there is an inherent trade-off between loop unrolling and the area of the design. Though the latency of the loop is divided by the factor it is unrolled, this also multiplies the number of operations by the same factor resulting in an increase in the number of required hardware primitives. The *fft_group* loop was the primary target for loop unrolling as it was a nested loop with $\log n - 2$ occurrences. However, the memories within this loop contained two concurrent read and write operations per iteration making it impossible to unroll without array partitioning as the BRAM resources on the FPGA support a maximum of two read/write ports. Because each iteration of the *fft_group* loop operates between an element with an odd index and an element with an even index, it was possible to cyclically partition the input array into two. This enabled them to be individually indexed between loop iterations and implemented as separate dual port memories. The *fft_group* loop was manually unrolled by a factor of two with the FFT butterfly operation implemented with a C++ function template to ensure that each iteration of the *fft_stage* loop statically selected from which of the "ping-pong" memories to read and write. This was necessary because the HLS tool cannot resolve dependencies between function calls within a loop that operate on the same array and will always schedule the operations sequentially.

V. Results

The polynomial multiplier design developed in this work was synthesized with Vivado HLS 2018.3 using the default settings targeting the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit (xczu9eg-ffvb1156-2-i-es2) with a clock period of 4 ns for each of the configurations outlined in Table I. The synthesized design was benchmarked and verified against NTL [22], which is utilized within software HE solutions such as HElib [23]. The NTL library is built on top of the popular GNU Multiple Precision (GMP) library [24] and contains accelerated algorithms for common number theory operations. The average computation time required for the FFT polynomial multiplication function within the NTL software library was measured on a 4-core/4-thread 3.7 GHz AMD A10-7850k CPU with 16 GB of RAM. The timing results of the synthesized design and calculated speedup over NTL are shown in Table II.

TABLE II
Timing results of the synthesized design

n	$\log q$	Latency	FPGA (µs)	NTL (µs)	Speedup
1,024	19	7,390	30	1,231	42
1,024	22	7,390	30	1,211	41
1,024	31	7,399	30	1,211	41
2,048	33	15,614	62	2,486	40
2,048	42	15,686	63	2,337	37
2,048	58	15,686	63	3,248	52
4,096	62	33,184	133	7,301	55
4,096	80	33,199	133	10,436	79
4,096	113	33,210	133	14,582	110
8,192	123	70,120	280	25,367	90
8,192	157	70,120	280	30,280	108
8,192	223	70,150	281	40,644	145
16,384	243	148,011	592	98,995	167
16,384	310	148,011	592	121,281	205
16,384	443	148,041	592	228,205	385
32,768	481	311,920	1,248	470,785	377
32,768	616	311,978	1,248	570,413	457
32,768	886	311,948	1,248	901,426	722

TABLE III
Available resources for part xczu9eg-ffvb1156-2-i-es2

BRAM_18K	DSP48E	FF	LUT
1,824	2,520	548,160	274,080

This estimated speedup is in the range 37-722. The synthesized polynomial multiplier design was exported and implemented for a subset of the 256-bit security configurations outlined in Table I. The implementation results are presented in Table IV. The utilization percentage of each resource required by the implemented design refer to all resources available on target device as presented in Table III. The maximum achieved clock frequency for each configuration was calculated based on the Worst Negative Slack (WNS) resulting from place and route of the design on the target device. A maximum

TABLE IV
RESULTS OF THE IMPLEMENTED DESIGN FOR 256-BIT SECURITY CONFIGURATIONS TARGETING PART XCZU9EG-FFVB1156-2-I-ES2

n	$\log q$	BRAM_18K	DSP48E	FF	LUT	f_{max} (MHz)	Latency	FPGA (µs)	NTL (µs)	Speedup
1,024	19	3%	1%	2%	7%	299	7,390	25	1,231	50
2,048	33	4%	4%	3%	8%	272	15,614	58	2,486	43
4,096	62	11%	15%	6%	13%	256	33,184	129	7,301	56
8,192	123	41%	48%	8%	11%	200	70,120	350	25,367	72

clock frequency of 299 MHz was achieved for $n = 1024$ and $\log q = 19$ resulting in a speedup of 50 versus the HLS estimated speedup of 42. As the synthesis speedup results were based on a 250 MHz clock, the actual speedup that can be achieved is greater once implemented on the target device. Conversely, a minimum clock frequency of 200 MHz was achieved for $n = 8192$ and $\log q = 123$ resulting in a speedup of 72, lower than the speedup of 90 achieved by the synthesis estimation.

VI. CONCLUSIONS

The security benefits of HE are significant, especially in regards to cloud computing, as encrypted data can be operated on without revealing the underlying plaintext to untrusted parties. The acceleration of the computationally intensive high-precision high-degree polynomial arithmetic operations within FHE schemes is of the utmost importance to enable their widespread use. Our design achieved significant speedup over the polynomial multiplication operations performed by the NTL software library for various security configurations. Although the design does not exceed the performance of dedicated hardware solutions, the multiplier exhibits flexibility in the selection of both the polynomial degree and coefficient size allowing for it to be configured for the security level and target device. Our approach differs markedly from typical hardware acceleration approaches since we focus on the seamless HE application development from software perspective rather than focus on the hardware acceleration. Very few software developers clearly understand the intricacies of developing effective hardware-software co-design on large scale heterogeneous cloud.

REFERENCES

[1] Amazon. Amazon Web Services (AWS) - cloud computing services. [Online]. Available: https://aws.amazon.com

[2] W. Wang, X. Huang, N. Emmart, and C. Weems, "VLSI design of a large-number multiplier for fully homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 1879–1887, Sept 2014.

[3] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. C. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 1, pp. 157–166, Jan 2015.

[4] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, June 2015.

[5] M. J. Foster, M. Lukowiak, and S. Radziszowski, "Flexible HLS-based implementation of the Karatsuba multiplier targeting homomorphic encryption schemes," in *2019 MIXDES - 26th International Conference "Mixed Design of Integrated Circuits and Systems"*, June 2019.

[6] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat, "Hardware/software co-design of an accelerator for FV homomorphic encryption scheme using Karatsuba algorithm," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 335–347, March 2018.

[7] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Report 2012/144, 2012, https://eprint.iacr.org/2012/144.

[8] K. Kawamura, M. Yanagisawa, and N. Togawa, "A loop structure optimization targeting high-level synthesis of fast number theoretic transform," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, March 2018, pp. 106–111.

[9] A. Mkhinini, P. Maistri, R. Leveugle, R. Tourki, and M. Machhout, "A flexible RNS-based large polynomial multiplier for fully homomorphic encryption," in *2016 11th International Design Test Symposium (IDT)*, Dec 2016, pp. 131–136.

[10] A. Mkhinini, P. Maistri, R. Leveugle, and R. Tourki, "HLS design of a hardware accelerator for homomorphic encryption," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, April 2017, pp. 178–183.

[11] A. Mkhinini, P. Maistri, R. Leveugle, and R. Tourki, "Co-designed accelerator for homomorphic encryption applications," *Advances in Science, Technology and Engineering Systems Journal*, vol. 3, no. 1, pp. 426–433, 2018.

[12] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC '09*. ACM Press, 2009.

[13] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," Cryptology ePrint Archive, Report 2011/277, 2011, https://eprint.iacr.org/2011/277.

[14] ——, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference on - ITCS '12*. ACM Press, 2012.

[15] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23.

[16] O. Regev, "The learning with errors problem (invited survey)," in *2010 IEEE 25th Annual Conference on Computational Complexity*, June 2010, pp. 191–204.

[17] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Security of homomorphic encryption," HomomorphicEncryption.org, Redmond WA, USA, Tech. Rep., July 2017.

[18] M. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, 10 2015.

[19] Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, sep 1971.

[20] A. Schönhage, "Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients," in *Computer Algebra*, J. Calmet, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 3–15.

[21] S. Benz, "Fast multiplication of multiple-precision integers," Master's thesis, Rochester Institute of Technology, 1991.

[22] V. Shoup, "NTL: A library for doing number theory," 2018, version 11.3.2, https://www.shoup.net/ntl/.

[23] S. Halevi and V. Shoup, "HElib - an implementation of homomorphic encryption," Online, 2013. [Online]. Available: https://github.com/shaih/HElib/

[24] T. Granlund and the GMP development team, "GNU multiple precision arithmetic library," 2016, version 2.5.2,, https://gmplib.org/.