Exploring Generalization of Shoup Modular Multiplier

Oleg Mazonka
om22@nyu.edu
New York University Abu Dhabi
UAE

Mohammed Nabeel
mtn2@nyu.edu
New York University Abu Dhabi
UAE

Michail Maniatakos
michail.maniatakos@nyu.edu
New York University Abu Dhabi
UAE

ABSTRACT

Shoup's modular multiplication algorithm follows the idea of Barrett reduction algorithm. While Barrett reduction can be used to multiply two arbitrary numbers, Shoup's multiplier requires a pre-computed value for one of the operands. At the same time, Shoup is more efficient as it requires less computation. In this work, we extend Shoup's multiplier by adding functionality to operate on arbitrary operands in such a way that the multiplier can be used in both ways: using the original Shoup algorithm when one of the arguments can be pre-computed, or a general multiplier. The general multiplier reuses Shoup functionality in its core. We compare the performance of the multipliers in a software simulator and a hardware design.

ACM Reference Format:
Oleg Mazonka, Mohammed Nabeel, and Michail Maniatakos. 2024. Exploring Generalization of Shoup Modular Multiplier. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24), June 12–14, 2024, Clearwater, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649476.3660374

1 INTRODUCTION

Modular multiplication is at the core of Fully Homomorphic Encryption (FHE) computations (BFV [5], BGV [2], CKKS [3]). Currently, computing with FHE exhibits significant performance overhead, since plaintexts are transformed to polynomials with degrees in the orders of thousands, and coefficients sizes in the order of hundreds. Efficient operation using CPUs is somewhat limited on such long polynomials, thus there has been interest in building dedicated hardware accelerators. Operating on long polynomials of FHE schemes requires many concurrent modular multiplications. Hardware accelerators [8] need to incorporate as many fast multipliers as possible. In other words, the hardware multipliers must be as fast as possible and as small as possible, two targets typically conflicting during hardware design. Recent studies [10], [11] have primarily focused on conducting detailed performance analyses of popular modular multipliers, particularly within the context of FHE. However, there appears to be relatively less emphasis on exploring alternative algorithms for executing modular multiplication.

FHE schemes typically rely on Number Theoretic Transformation (NTT) for multiplication of polynomials. This efficient method allows the polynomial multiplication to be performed in $O(n \log n)$ instead of $O(n^2)$ time-memory constrain. Specifically, this method converts polynomials into the NTT form, multiplies polynomials by element-wise coefficient multiplication, and finally transforms back. NTT transformation needs modular multiplication between the polynomial coefficients and the twiddle factors; while the element-wise polynomial multiplication needs modular multiplication between the polynomial coefficients. Therefore there are two types of modular multiplications need for multiplying polynomials:

(1) NTT and inverse NTT: Multiplication between a polynomial coefficient and a twiddle factor – multiplication between two numbers where one number is always fixed and can be pre-computed; and
(2) Element-wise: Multiplication between a coefficient of one polynomial and the corresponding coefficient of the other polynomial – multiplication between two arbitrary numbers.

Clearly one generic multiplier can cover both above types of multiplications. On the other hand type 1 multiplication can be optimized and performed faster since one of the multiplication arguments is fixed, which is usually implemented in software polynomial multiplication [12].

Barrett multiplication [1] is often used as a generic multiplier due to its efficiency. Shoup multiplication [9], [6] is used as the specialized multiplier for the type 1 multiplication ([12], [7]). There are two choices to perform multiplication in the above scenario: 1) to use only the Barrett multiplier, or 2) to use both. In the first case, we lose some efficiency when multiplying type 1 with a generic multiplier. In the second case, when implementing in hardware, we bear the

GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA

ACM ISBN 979-8-4007-0605-9/24/06
https://doi.org/10.1145/3649476.3660374

overhead of having two different multipliers. It would be desirable to combine them into one multiplier without losing efficiency in both multiplication scenarios. While the Shoup multiplier is similar by nature and idea to Barrett, it is not straightforward to combine them without affecting their efficiency. It should be noted that in our study we do not address the Montgomery multiplier because, in the context of FHE, multiplications involve independent numbers, and the conversion to and from the Montgomery domain introduces significant delays.

Contribution. In this work, we present a design of a modular multiplier that can be used in two modes: 1) as a generic multiplier and 2) directly as a Shoup multiplier. We prove its correctness and demonstrate and compare its performance to Barrett.

2 BACKGROUND

2.1 Barrett multiplication

We use A and B as multiplication arguments (operands). M is the modulus of the multiplication. The original Barrett multiplication can be expressed in the following way:

$$t = x - \left\lfloor \frac{xr}{4^k} \right\rfloor M \qquad \text{with}$$

$$x = AB \qquad r = \left\lfloor \frac{4^k}{M} \right\rfloor$$

Value k is selected as $\lceil \log_2 M \rceil$ and r is pre-computed. At the end the result t must be reduced once, i.e. if $t < M$ then use t, else use $t-M$. This direct method requires 3 multiplications, where one multiplication, xr, is expensive as it needs multiplying argument x of $2k$ bit size by r of k bit size.

This big multiplication can be replaced with a smaller one, at the expense of extra reduction. Algorithm 1 shows the optimized version of the Barrett multiplier. From now on, number k is the size of the multiplier, not related to the size of M. Since the modulus M can be in size smaller than k, an adjustment must be done to enforce the Most Significant Bit (MSB) of M to be 1. It requires left shift of one operand and right shift of the result. In this optimized version the product x looses the lower $k-1$ bits while getting the quotient. Hence q gets the shift by $k+1$ instead of $2k$. The result of dropping the lower bits is that the quotient value can be off by 1; thus at the end we need two reductions instead of one as in the original Barrett. The compensation of bit (line 7) normally is not required in hardware if the subtractor is exactly of $k+1$ size.

Comparing to the original Barrett algorithm, this optimized version replaces the big multiplication $2k$ by k, by the smaller k by $k+1$ (line 4). As a result it requires two reductions at the end (lines 9,10) instead of one.

Algorithm 1 Optimized Barrett algorithm

Input: A, B, M', r, l
Conditions: $A < M$, $B < M$
Output: $t = AB \bmod M$
Pre-computation: $l = k - \lceil \log_2 M \rceil$, $M' = M \ll l$, $r = \left\lfloor 4^k / M' \right\rfloor$

1: $A' \leftarrow A \ll l$
2: $x \leftarrow A'B$
3: $x' \leftarrow x \gg (k-1)$
4: $q \leftarrow (x'r) \gg (k+1)$
5: $r_1 \leftarrow x \bmod 2^{k+1}$
6: $r_2 \leftarrow qM' \bmod 2^{k+1}$
7: **if** $r_1 < r_2$ **then** $r_1 \leftarrow r_1 + 2^{k+1}$ ▷ No need in HW
8: $t \leftarrow r_1 - r_2$
9: **if** $t \geq M'$ **then** $t \leftarrow t - M'$
10: **if** $t \geq M'$ **then** $t \leftarrow t - M'$
11: $t \leftarrow t \gg l$
12: **return** t

Algorithm 2 Shoup algorithm

Input: A, B, M, r
Conditions: $A < M$, $B < M$, $2M < 2^k$
Output: $t = AB \bmod M$
Pre-computation: $r = \left\lfloor 2^k B / M \right\rfloor$

1: $q \leftarrow (Ar) \gg k$
2: $r_1 \leftarrow AB \bmod 2^k$
3: $r_2 \leftarrow qM \bmod 2^k$
4: **if** $r_1 < r_2$ **then** $r_1 \leftarrow r_1 + 2^{k+1}$ ▷ No need in HW
5: $t \leftarrow r_1 - r_2$
6: **if** $t \geq M$ **then** $t \leftarrow t - M$
7: **return** t

Since A and M must match the size of the multiplier, a parameter l is added to shift left the input A (line 1) and shift right the output t (line 11). Its value is equal to number of bits M must be shifted left so the MSB in position k is 1.

2.2 Shoup multiplication

Shoup multiplier [9], [6] follows similar logic as Barrett. Its algorithm is shown in the listing 2. It is simpler comparing to the optimized Barrett since no shift is required and at the end there is only one reduction instead of two.

3 PROPOSED MULTIPLIER

3.1 Theory

Shoup multiplication follows the idea of Barrett with a difference that the pre-computed value incorporates one of the argument (B) to compute the quotient q. The idea of this work is to compute Shoup's "pre-computed" value to the

closest approximation so the Shoup algorithm can be used directly. Here we replace Shoup's value r_s with another value r_g that can be computed without division by introducing some arbitrary parameter α:

$$q_s = \left\lfloor \frac{Ar_s}{2^k} \right\rfloor \qquad r_s = \left\lfloor \frac{2^k B}{M} \right\rfloor \tag{1}$$

$$q_g = \left\lfloor \frac{Ar_g}{2^k} \right\rfloor \qquad r_g = \left\lfloor \frac{gB}{2^\alpha} \right\rfloor \qquad g = \left\lfloor \frac{2^{k+\alpha}}{M} \right\rfloor \tag{2}$$

The pair of values q_s, r_s in Shoup multiplier is replaced with new variables q_g, r_g, g. Without affecting functionality, we assume that

$$2^{k-2} \le M < 2^{k-1} \tag{3}$$

This is possible since modulus M and argument A can be shifted left, and at the end of computation the result is shifted right by the same number of bit positions.

The parameter α in Eq. 2 regulates the precision of the reciprocal of modulus M. Its optimal value is yet to be determined for the optimal design. It is clear that when α goes to infinity, then r_g converges to r_s. At the same time we try getting α as small as possible so $q_g \approx q_s$ and computing r_g is the most efficient. Any difference between q_g and q_s would require extra reductions of the final result. We find the best solution for α expressed in the following proposition.

Proposition: The value α set to $\alpha = k - 2$ in Eq. 2 ensures that q_g either equal to q_s or at most less by one, i.e.:

$$q_s - 1 \le q_g \le q_s \tag{4}$$

Proof: Let

$$\varepsilon = \frac{2^{2k-2}}{M} - \left\lfloor \frac{2^{2k-2}}{M} \right\rfloor$$

By definition $0 \le \varepsilon < 1$. The expression for q_g in Eq. 2 can be written as:

$$q_g = \left\lfloor 2^{-k} A \left\lfloor \frac{B(2^{2k-2}/M - \varepsilon)}{2^{k-2}} \right\rfloor \right\rfloor$$

by direct substitution r_s and replacing the floor term via ε. Since $\varepsilon \ge 0$, the right hand side of the above equation can only increase if ε is dropped:

$$q_g \overset{\{\varepsilon \ge 0\}}{\le} \left\lfloor 2^{-k} A \left\lfloor \frac{B(2^{2k-2}/M)}{2^{k-2}} \right\rfloor \right\rfloor = q_s$$

After cancellations, the expression is equal to q_s, which proves the right side of inequality Eq. 4.

The other side of the inequality requires a chain of transformations with the corresponding replacements that keep the inequality relation valid. Start the expansion of q_g:

$$q_g = \left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} - \frac{B\varepsilon}{2^{k-2}} \right\rfloor \right\rfloor$$

Since $\varepsilon < 1$, the right hand side can only be reduced by replacing ε with 1 as the holding term is negative:

$$\left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} - \frac{B\varepsilon}{2^{k-2}} \right\rfloor \right\rfloor \overset{\{\varepsilon<1\}}{\ge} \left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} - \frac{B}{2^{k-2}} \right\rfloor \right\rfloor$$

Furthermore, replacing B with 2^{k-1} would reduce the expression as $B < 2^{k-1}$ and the holding term is negative:

$$\left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} - \frac{B}{2^{k-2}} \right\rfloor \right\rfloor \overset{\{B<2^{k-1}\}}{\ge} \left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} - 2 \right\rfloor \right\rfloor$$

Now rearrange the expression for further reductions:

$$\left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} - 2 \right\rfloor \right\rfloor = \left\lfloor \frac{A}{2^k} \left(\left\lfloor \frac{2^k B}{M} \right\rfloor - 2 \right) \right\rfloor = \\ = \left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} \right\rfloor - \frac{A}{2^{k-1}} \right\rfloor$$

In similar way as before, A can be replaced (with potential reduction) with 2^{k-1} because $A < 2^{k-1}$:

$$\left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} \right\rfloor - \frac{A}{2^{k-1}} \right\rfloor \overset{\{A<2^{k-1}\}}{\ge} \left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} \right\rfloor - 1 \right\rfloor$$

Finally, the expression is reorganized into the form in terms of q_s:

$$\left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} \right\rfloor - 1 \right\rfloor = \left\lfloor \frac{A}{2^k} \left\lfloor \frac{2^k B}{M} \right\rfloor \right\rfloor - 1 = q_s - 1$$

that results in the inequality:

$$q_g \ge q_s - 1$$

which proves the left side of Eq. 4. This concludes the proof of the Proposition. □

The Proposition suggests that 1) q_g from Eq. 2 can be used instead of q_s with $\alpha = k - 2$; and 2) only one reduction is necessary when using q_g instead of q_s. With this information in hand, it is possible to reuse the Shoup multiplier for general modular multiplication with the following additional processing:

(1) Modulus M must be aligned according to Eq. 3. This requires shift-left of the operand A and shift-right of the result. This step is the same as in Barrett multiplier.
(2) Shoup's input r can be computed by the following multiplication:
$$r = \left\lfloor \frac{gB}{2^{k-2}} \right\rfloor, \quad \text{where} \quad g = \left\lfloor \frac{2^{2k-2}}{M} \right\rfloor$$
g is pre-computed.
(3) The result inside Shoup module may be greater than $2M$ (but less than $3M$) since q_g may be equal to $q_s - 1$. Therefore an extra reduction is necessary after Shoup returns.
(4) For the same reason, an extra bit is required for Shoup module to hold the subtraction result.

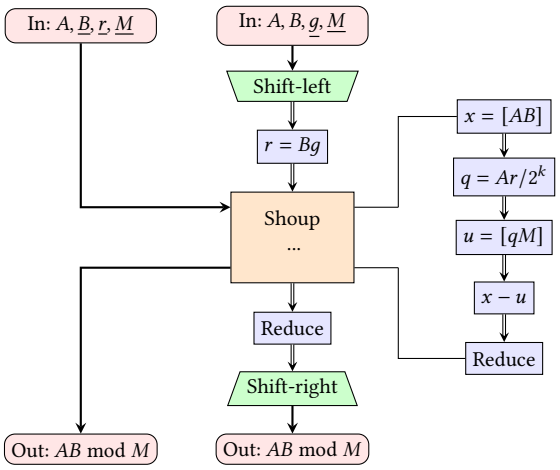

Figure 1: Overall diagram of the algorithm. Two pathways represent fast Shoup multiplication and full multiplication. Underlined inputs are fixed: they remain the same for different runs. Shoup block is expanded on the right side of the diagram.

Algorithm 3 Shoup modified

Input: A, B, M, r
Conditions: $A < M$, $B < M$, $2M < 2^k$
Output: $t = AB \bmod M$
Pre-computation: $r = \left\lfloor 2^k B/M \right\rfloor$ or $r = \left\lfloor gB/2^{k-2} \right\rfloor$

1: $q \leftarrow (Ar) \gg k$
2: $r_1 \leftarrow AB \bmod 2^{k+1}$ ▷ extra bit
3: $r_2 \leftarrow qM \bmod 2^{k+1}$ ▷ extra bit
4: **if** $r_1 < r_2$ **then** $r_1 \leftarrow r_1 + 2^{k+2}$ ▷ No need in HW
5: $t \leftarrow r_1 - r_2$ ▷ subtraction in k+1 bit size
6: **if** $t \geq M$ **then** $t \leftarrow t - M$
7: **return** t

Algorithm 4 Full multiplication

Input: A, B, M, g, l
Conditions: $A < M$, $B < M$
Output: $t = AB \bmod M$
Pre-computation: $l = k - 1 - \lceil \log_2 M \rceil$
$M' = M \ll l$
$g = \left\lfloor 2^{2k-2}/M' \right\rfloor$

1: $r \leftarrow (Bg) \gg (k-2)$
2: $A' \leftarrow A \ll l$
3: $t \leftarrow$ **Shoup modified [Algorithm 3]** $\{A', B, M', r\}$ ▷
4: **if** $t \geq M'$ **then** $t \leftarrow t - M'$
5: $t \leftarrow t \gg l$
6: **return** t

Now, it is possible to proceed with the overall algorithm design, combining both Shoup multiplication and full modular multiplication.

3.2 Algorithm

The modified Shoup algorithm is shown in Listing 3. It requires one extra bit. Also, the parameter r has a different meaning: it can represent the original Shoup pre-computed value or computed outside value when the multiplier is used as a full modular multiplier.

Listing 4 describes full modular multiplication using a modified Shoup multiplier as a core. It has four extra steps in comparison to the Shoup algorithms. First, there is multiplication to compute Shoup's input value r. The right shift by $k-2$ is not a computational operation since k is fixed. Argument A is left shifted. Then, the original Shoup multiplier of size k+1 can be used. Finally, reduction and the right shift are applied to the result.

The full multiplier effectively prepares the input to its core multiplier and post processes the result. If such multiplier is implemented in hardware, it can be used in two different scenarios: 1) as full modular multiplier; and 2) as Shoup multiplier using only its core. The idea is to reuse the faster core multiplication without adding a separate multiplier. Fig. 1 shows the two usages. If both arguments A and B are arbitrary values, the full multiplication is performed with pre-computed g parameter. If one of the arguments is known upfront, such as twiddle factors in NTT multiplication, and can be pre-computed, then the faster core multiplier operation is used.

4 EXPERIMENTS

4.1 Software evaluation

To estimate the efficiency of the proposed multiplier, we use the E3 programming framework [4], which allows gate counting from a C++ algorithm description. To compare the performance of different multiplier options we must consider both area and latency of the multiplier. Several multipliers can be placed on the same hardware chip to accommodate polynomial multiplication. Therefore, many smaller and slower multipliers may have similar performance as fewer larger and faster multipliers. We define efficiency as:

$$\textbf{Efficiency} = \frac{S^2}{P \cdot A}$$

where S is bit-size of the modular multiplier, P - the critical path, and A - area. This expression is deduced by regarding P/S as normalized latency and A/S as normalized area; hence, efficiency is a reciprocal of their product. The area and the critical path in E3 are computed as the total number of gates

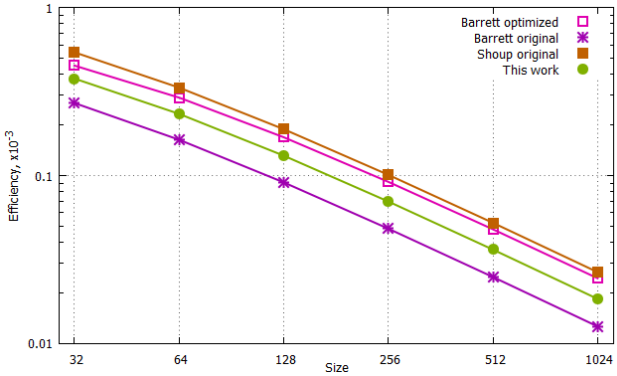

Figure 2: Efficiency comparison for standalone multiplier. X-axis is bit size of the multiplier. "This work" is the generalized Shoup and is depicted by green circles.

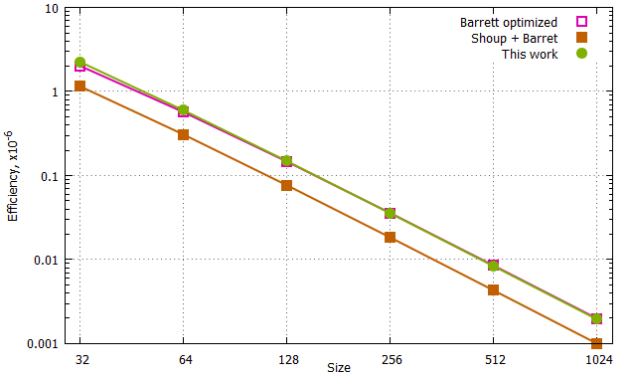

Figure 3: Efficiency comparison for simple polynomial multiplication with NTT conversions. X-axis is bit size of the multiplier. Barrett optimized goes along the curve of "This work".

and the longest chain of gates correspondingly required to complete the computation.

The results of using a standalone multiplier presented in Fig. 2 demonstrate consistently outperforming Shoup over Barrett and optimized Barrett over-generalized Shoup. On the other hand, when considering polynomial multiplication in hardware, reusing the area for the generalized Shoup improves the efficiency. A simple polynomial multiplication of the size N requires $N+N\log N$ modular multiplications with twiddle factors for NTT and inverse NTT, where N is the size of the polynomials plus N modular multiplications between coefficients. The first can be executed by Shoup. The second must be executed either by Barrett or generalized Shoup. In this case, the design of the multiplier setup can consist of either:

- Barrett for both type of modular multiplications; or
- Two multipliers: Shoup and Barrett; or
- Our proposed multiplier: generalized Shoup.

Fig. 3 shows the performance for these three options. In the graph Barrett's efficiency is almost hidden under the generalized Shoup (This work). This suggests that in our software performance estimation, the efficiency of the proposed multiplier can be as good as that of the Barrett-optimized multiplier.

4.2 Hardware evaluation

RTL for the above algorithms are written in Verilog and functionally verified using Synopsys VCS. Pipeline registers are inserted at the appropriate places in the design to control the critical path delay. For the Barrett design, pipeline registers are inserted at line 3, at line 4 before shifting, and at lines 8 and 11 in Algorithm 1. For Shoup, pipeline registers are inserted at lines 1, 2, 3, and 6 in Algorithm 3. Our multiplier has pipeline registers at lines 4 and 5 in Algorithm 4. To get the

Design	Size	Critical path, ns	Area, μm^2
Shoup	32	0.81	10148
	64	1.02	33353
	128	1.25	106312
	256	1.51	341616
Barrett optimized	32	1.02	12983
	64	1.31	36509
	128	1.61	113129
	256	1.95	367207
This work	32	0.86	15251
	64	1.10	48980
	128	1.32	153896
	256	1.63	482944

Table 1: Area and Delay Comparison of the multipliers.

area and performance numbers, RTL synthesis is done using Synopsys Design Compiler (DC) with TSMC 22nm Ultra Low Leakage (ULL) technology library. The library used is the one characterized by the worst corner, worst voltage of 0.81V, worst temperature 125°C), and worst process/parasitic (resistance and capacitance). This ensures that we achieve the operational frequency at the voltage of 0.9V±10% and at any temperature less than 125°C. We synthesized the multipliers to be as fast as possible, constraining the clock period to 250 ps so that the resultant critical path will give the minimum clock period the multiplier can be run at.

Table 1 lists the hardware design measurements: critical paths and area. Three different hardware implementations are presented: Shoup, Barrett, and Our multiplier. As expected, the critical path and area are increasing with the operand size of the multiplier. The area increases about three times when the size doubles, which roughly corresponds

to the increase in size of the underlying non-modular multipliers. Moreover, Barrett and Shoup have three underlying non-modular multipliers, and our multiplier has four. This explains why the area of our multiplier is about 30% greater than Barrett's. On the other hand, our multiplier performs faster than Barrett. The critical path within Barrett is between the pipeline registers at line 4 and line 8 in Algorithm 1. For Our multiplier, the critical path is within the Shoup multiplier, from the pipeline register at line 1 to the pipeline register at line 3 in Algorithm 3.

Our multiplier combines functionality of both Shoup and Barrett (full modular multiplication) multipliers. In case a design requires both functionalities in hardware Shoup and Barrett would occupy about 1.5 time more area than our multiplier.

5 CONCLUSION

In this paper, we explore the possibility of extending the Shoup multiplier for use as a core for full modular multiplication. We demonstrate that it is indeed feasible to make modifications to the Shoup multiplier, allowing its functionality to extend to full modular multiplication with some pre- and post-processing. Given the proven proposition stated in Eq. 4, we show that a mathematically efficient algorithm exists for blending both computations with and without one predefined argument. Such blending of computation is attractive from a hardware implementation standpoint, as it allows for the reuse of the processing unit for both types of multiplications.

Results show that our multiplier is faster in the critical path by about 20% than Barrett and slower than Shoup by only $\approx$ 8%. At the same time, it occupies more area. The main feature of our multiplier is that it can be used both as a Shoup multiplier and as full modular multiplier.

ACKNOWLEDGMENTS

The work was funded by the Center for Cyber Security Abu Dhabi (CCSAD) at New York University Abu Dhabi.

REFERENCES

[1] Paul Barrett. 1987. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Advances in Cryptology — CRYPTO' 86*, Andrew M. Odlyzko (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323.

[2] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption without Bootstrapping. Cryptology ePrint Archive, Report 2011/277. https://eprint.iacr.org/2011/277.

[3] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437.

[4] Eduardo Chielle, Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. 2018. E3: A Framework for Compiling C++ Programs with Encrypted Operands. Cryptology ePrint Archive, Report 2018/1013. https://eprint.iacr.org/2018/1013.

[5] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144.

[6] Tommy Färnqvist. 2005. Number Theory Meets Cache Locality – Efficient Implementation of a Small Prime FFT for the GNU Multiple Precision Arithmetic Library.

[7] David Harvey. 2014. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation* 60 (Jan 2014), 113–119. https://doi.org/10.1016/j.jsc.2013.09.002

[8] Mohammed Nabeel, Deepraj Soni, Mohammed Ashraf, Mizan Abraha Gebremichael, Homer Gamil, Eduardo Chielle, Ramesh Karri, Mihai Sanduleanu, and Michail Maniatakos. 2023. CoFHEE: A co-processor for fully homomorphic encryption execution. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.

[9] Victor Shoup. 2021. NTL: a library for doing number theory. http://www.shoup.net/ntl.

[10] Deepraj Soni, Mohammed Nabeel, Homer Gamil, Oleg Mazonka, Brandon Reagen, Ramesh Karri, and Michail Maniatakos. 2023. Design Space Exploration of Modular Multipliers for ASIC FHE accelerators. In *2023 24th International Symposium on Quality Electronic Design (ISQED)*. 1–8. https://doi.org/10.1109/ISQED57927.2023.10129292

[11] Deepraj Soni, Mohammed Nabeel, Negar Neda, Ramesh Karri, Michail Maniatakos, and Brandon Reagen. 2023. Quantifying the Overheads of Modular Multiplication. In *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. https://doi.org/10.1109/ISLPED58423.2023.10244324

[12] Duality Technologies. 2021. Palisade. https://palisade-crypto.org.palisade.