A Fully Pipelined Reconfigurable Montgomery Modular Multiplier Supporting Variable Bit-Widths

Hao Zhou, Changxu Liu, Lan Yang, Li Shang *Member, IEEE,* and Fan Yang *Member, IEEE*

***Abstract*—Recently, there has been increased emphasis on privacy-preserving computation technologies such as homomorphic encryption (HE) and Zero-knowledge proof (ZKP). Modular multiplication is a critical component for both HE and ZKP. Variable bit-width is a must for many applications of privacy-preserving computation, due to variable bit-width requirements for different cryptography schemes. However, the majority of modular multipliers that support variable bit-width configurations exhibit relatively low throughput. This work presents a fully pipelined Montgomery modular multiplier with variable bit-width support. Truncated multipliers are introduced to reduce the resources of modular multipliers in our approach. In order to meet different bit-width requirements, the proposed modular multiplier can be dynamically reconfigured. The proposed design can support widely used bit-width configurations, specifically, 384-bit, 256-bit, and 128-bit. 256-bit and 128-bit modes support parallel computation of 2 and 6 sets of operands, respectively. Compared with existing variable bit-width modular multipliers, the proposed reconfigurable modular multiplier significantly improves the throughputs with even lower resources.**

***Index Terms*—Montgomery modular multiplication, reconfigurable, variable bit-width , parallel computation, privacy-preserving computation**

I. Introduction

In recent years, Zero-knowledge proof (ZKP) has garnered great attention due to the growing need for analyzing sensitive blindly. ZKP has been deployed across diverse landscapes [1]–[4], ranging from the electronic voting to online auctions and smart contracts on blockchains. The data processed by ZKP typically exhibit a large bit-width up to several hundreds, requiring considerable resources. Nevertheless, the efficiency of privacy-preserving computations on general-purpose computing processors like CPUs and GPUs can not meet the requirements of various applications [5]. For instance, generating a Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) proof may take a long time, potentially a few minutes even for a singular payment transaction [6]. In contrast, accelerators for privacy-preserving computing can greatly improve the efficiency.

In order to improve the efficiency of privacy-preserving computing, hardware acceleration methodologies rooted in a fully pipelined design have yielded commendable acceleration outcomes [6]–[9]. As a foundational arithmetic unit of the hardware accelerators, modular multipliers exert a direct influence on the performances of accelerators [10]–[13]. Pipelined modular multiplication can greatly improve the speed of hardware accelerators.

The ability to support multiple bit-widths is also an important issue that needs to be considered in modular multiplication. There are a variety of protocols and algorithms used in privacy-preserving computing. The bit-width requirements of modular multipliers for ZKP can vary greatly. In ZKP systems, Number-Theoretic Transform (NTT) and Multi-Scalar Multiplication (MSM) on the most widely used elliptic curves (ECs), such as BN-128, BLS12-381, and BLS12-377 [14], need 256-bit and 384-bit modular multiplications. In previous works, only single bit-width is supported by NTT or MSM design. After fabrication, the designs cannot support other protocols with different bit-width. Therefore, a reconfigurable modular multiplication supporting multiple bit-widths can greatly improve the flexibility of ASIC accelerators to support various protocols or emerging more efficient algorithm after fabrication.

Both pipelined and multiple bit-width supporting modular multipliers have been explored in prior research.

A. Pipelined Modular Multipliers

Most pipelined modular multipliers are either based on Barrett modular multiplication [15] or Montgomery modular multiplication [16]. Barrett modular multiplication is similar to commonly used modulo operation and also requires integer multiplication to find the quotient. Compared to Barrett algorithm, the Montgomery algorithm needs transforms of the operands between Montgomery forms and normal forms. However, for a computing task, only the transforms from the normal forms to Montgomery forms for the inputs, and the transforms from the Montgomery forms to the normal forms for outputs are needed. The computation cost of these transforms is usually ignorable for complex tasks. If the computational cost of these transforms is not ignorable, Barrett algorithm can be used. The resource consumption for implementation of Montgomery algorithm is lower than that of Barrett algorithm [17]. The length of the critical path of Montgomery algorithm is also shorter than Barrett algorithm [17]. More importantly, for variable bit-width support, the Montgomery algorithm and its improved version have more flexibility than the Barrett algorithm. To the best of our knowledge, the Barrett algorithm is not word-level-friendly and cannot efficiently support variable bit-width.

In Barrett modular multiplication, part of the multiplications only needs to calculate the upper half or the lower half bits

This research is supported partly by National Key R&D Program of China 2023YFB2704600, partly by National Natural Science Foundation of China (NSFC) research projects 92373207 and 62090025. Hao Zhou, Changxu Liu, Lan Yang and Fan Yang are with State Key Lab of Integrated Chips & Systems, and School of Microelectronics, Fudan University, Shanghai, China. Li Shang is with State Key Lab of Integrated Chips & Systems, and School of Computer Science, Fudan University, Shanghai, China. Corresponding Author: Fan Yang (yangfan@fudan.edu.cn).

of results. Therefore, truncated multipliers are introduced in pipelined Barrett modular multipliers to reduce the resources. For instance, a fully pipelined modular multiplier based on the Barrett and Shoup algorithm is presented in [18] for homomorphic encryption. To reduce resources, unnecessary DSPs and registers are removed from the lower and upper half integer multipliers. In [7], [19], both the Most Significant Bit multiplier (MSB-Mult) and the Least Significant Bit multiplier (LSB-Mult) are integrated into the pipelined Barrett modular multiplier. However, truncated multipliers in these Barrett modular multipliers are inefficient when directly adapted to the Montgomery algorithm.

The most popular Montgomery algorithms are the classical Montgomery algorithm [16] and the word-based Montgomery algorithm proposed by Tenca-Todorov-Koc in [20]. The former is most commonly used in fully pipelined Montgomery modular multipliers. In [9], a fully pipelined Montgomery modular multiplier tailored for MSM is realized. This modular multiplier comprises two 384-bit full-scale integer multipliers (FULL-Mults) and a custom multiplier based on non-adjacent form (NAF) representation. The design in [21] implements a fully pipelined Montgomery modular multiplier using the KO-3 algorithm-based LSB-Mult. The LSB-Mult here is more efficient than [7], [18], [19]. But it is also possible to apply MSB-Mult in the Montgomery algorithm, which was ignored in this work. More importantly, these works are limited to supporting a single bit-width, restricting their versatility in applications.

B. Modular Multipliers Supporting Variable Bit-Widths

Word-based Montgomery algorithms are all realized by scanning input multiple times. The times of iterations determine the bit-width supported by modular multiplier [20], [22]–[24]. These iterations take several cycles to complete one modular multiplication and result in remarkably low throughput, adversely impacting both the pipeline and the throughput of the upper-level modules. This poses a significant challenge, particularly in applications like smart contracts on blockchains, where generation speed and throughput are important [25].

C. Our contributions

In this paper, we introduce a fully pipelined Montgomery modular multiplier supporting variable bit-widths. To optimize the utilization of integer multipliers across multiple bit-widths, we present an enhanced word-based modular multiplication algorithm. This algorithm employs the Karatsuba-Ofman (KO) algorithm, which minimizes the required integer multipliers by fully utilizing all partial products in the initial phase. By carefully analyzing the Montgomery algorithm, we design a truncated multiplier to remove the unnecessary components in Montgomery modular multiplier. Considering the commonly used bit-widths (256-bit and 384-bit) in the ZKP system, the bit-width of basic integer multipliers is set as 128-bit to maximize resource utilization efficiency. Apart from supporting 256-bit and 384-bit, our design also implements 128-bit modular multiplication due to almost no extra costs needed. Moreover, several recent works, like private smart contract ZeeStar [26], blockchain-based private transaction scheme [27], and federated learning solutions [28]–[31], have integrated both Homomoriphc Encryption (HE) and ZKP into one system to improve the ability of private data protection. The 128-bit modular multiplication exactly meets the requirement for HE [32], like the designs in RPU [33] and CoFHEE [34]. There is great potential for our design to be integrated into a reconfigurable accelerator supporting HE and different ZKP protocols simultaneously. Our contributions can be summarized as follows.

- We propose an enhanced word-based modular multiplication algorithm. This algorithm incorporates a suitable KO algorithm to reduce the required integer multiplications. Based on this algorithm, the utilization of multiplication resources is 100% for all modes.
- We introduce truncated multipliers, including the LSB multiplier and MSB multiplier, aimed at eliminating calculations for the non-essential parts. For truncated multipliers, we provide the mathematical proof. This is the first time the MSB multiplier is introduced in the word-based Montgomery modular multiplier.
- We design a fully pipelined Montgomery modular multiplier supporting variable bit-widths. The versatility of this architecture allows for concurrent processing of 1-way 384-bit, 2-way 256-bit, or 6-way 128-bit pipelined modular multiplications in parallel. According to this architecture, we can scale up it to implement modular multipliers supporting larger bit-widths.
- We evaluate our reconfigurable fully pipelined modular multiplier with TSMC 28 nm technology and FPGA platform. Experimental results show that with TSMC 28nm technology, our proposed Montgomery modular multiplier can achieve up to 7.0× improvement in efficiency compared to the state-of-the-art designs. Compared to software-based solutions, our design achieves 53×-100× speedup. For FPGA platform, our proposed Montgomery modular multiplier can achieve 4.8×, 8.5× improvements in 128-bit and 256-bit modes, respectively, compared to the state-of-the-art designs. For 384-bit mode, our proposed Montgomery modular multiplier can achieve significantly higher throughput. Though the efficiency is slightly lower than the state-of-the-art design due to resources introduced to support multiple bit-widths, it exhibits more flexibility.

The remainder of this paper is organized as follows. In Section II, the background is briefly reviewed. In Section III, we introduce truncated multipliers to the modular multiplier and propose an improved word-based modular multiplication algorithm. In Section IV, we present our fully pipelined modular multiplier capable of handling multiple bit-widths. In Section V, we evaluate our design and compare it with previous works. In Section VI, we conclude the paper.

II. Background

A. Montgomery Modular Multiplication

Montgomery modular multiplication is a widely used algorithm in modular arithmetic. The traditional modular mul-

Algorithm 1 Montgomery Modular Multiplication [16]

Preprocess: $M^{'} = -M^{-1} \bmod R$
Input: $X, \bar{Y} \in [0, M)$, $R = 2^n > M$, $\gcd(R, M) = 1$
Output: $Z = X\bar{Y}R^{-1} \bmod M \in [0, M)$
1: $Z = X\bar{Y}$
2: $Q_m = (Z \bmod R)M^{'} \bmod R$
3: $Z = (Z + Q_m M)/R$
4: **if** $Z \geq M$ **then**
5: $\quad Z = Z - M$
return Z

tiplication algorithm requires division, which is implemented by multiple multiplications in hardware. In privacy-preserving computing scenarios, the bit-widths can vary from hundreds to thousands. The Montgomery algorithm [16] streamlines the modular multiplication by replacing the division with a more efficient shifting operation, reducing resource consumption and enhancing the efficiency. Algorithm 1 outlines the Montgomery modular multiplication procedure, where modular M is a prime number, n is the smallest positive integer ensuring coprimality between $R = 2^n$ and M (i.e., $\gcd(R, M) = 1$), M' represents the modular inverse of M with respect to the modular R. The multiplier Y undergoes conversion to the Montgomery form $\bar{Y}$ during preprocessing via Equation 1.

$$\bar{Y} = YR \ mod \ M. \tag{1}$$

In Algorithm 1, Y is converted to the Montgomery form, and thus $XY \bmod M$ is transformed to $X\bar{Y}R^{-1} \bmod M$. The Montgomery algorithm transforms divisions into shift operations. However, a direct right shift of $X\bar{Y}$ by n bits may lead to the loss of the lower n bits, resulting in a precision loss. To avoid this, a suitable Q_m is generated in Step 2 of Algorithm 1 to ensure the lower n bits of $Z + Q_m M$ are precisely zero. After right shift n bits of $Z + Q_m M$, the result of Step 3 meets Equation 2.

$$\begin{cases} X\bar{Y} < M^2, Q_m < R, M < R \\ \frac{X\bar{Y}+Q_m M}{R} < \frac{M^2+RM}{R} < \frac{RM+RM}{R} = 2M. \end{cases} \tag{2}$$

The final result is obtained after Step 4 and Step 5.

B. Scalable Montgomery Modular Multiplication

The Montgomery algorithm has indeed elevated the computational efficiency of large integer modular multiplications. However, both the algorithm and its corresponding hardware implementations face limitations in the bit-widths of operands and the flexibility. Based on the Montgomery algorithm, Tenca-Todorov-Koc introduced a word-based algorithm with scalable bit-width operations, as shown in Algorithm 2 [20]. This word-based approach not only conserves resources but also significantly enhances the efficiency. In Algorithm 2, the outer loop divides X into multiple k-bit words, sequentially multiplying them by the words of Y in each iteration. During the i-th iteration, the i-th word of X undergoes multiplication with the first word (Y_0) of Y and is added to the first word (Z_0) of the preceding iteration's result (Z). In Step 4, Q_{mi} is obtained in a way similar to Algorithm 1. After this Step, the result Q_{mi} ensures that the lower k bits of $(Z_0 + Q_{mi}M_0)$ are zero. In the inner loop (from Step 6 to Step 9 of Algorithm 2), s iterations are employed to scan both Y and M with k-bit Step. The subsequent outer loop proceeds to scan the next word of X following a right shift of k bits. This process continues, sequentially scanning the words of X and Y until the end of the algorithm.

It's worth noting that Algorithm 2 demands s^2 iterations. For operands with large bit-width, the required number of iterations could be significantly large, significantly impacting the algorithm's overall efficiency. A fully pipelined modular multiplier supporting multiple bit-width is possible to address this issue.

Algorithm 2 Scalable Montgomery Modular Multiplication [20]

Preprocess: $M^{'} = -M^{-1} \bmod R$, $s = \lceil \frac{n}{k} \rceil$
Input: $X, Y \in [0,M)$, $R = 2^n > M$, $\gcd(R, M) = 1$
Output: $Z = XYR^{-1} \bmod M \in$[0,M)
1: $Z = 0$
2: **for** i=0 to n-1 step k **do** ▷ Outer loop
3: $\quad (C_a, Z_0) = Z_0 + X[i]Y_0$
4: $\quad Q_{mi} = (Z_0 \bmod 2^k)M' \bmod 2^k$
5: $\quad (C_b, Z_0) = Z_0 + Q_{mi}M_0$
6: $\quad$ **for** j=1 to s-1 **do** ▷ Inner loop
7: $\quad\quad (C_a, Z_j) = C_a + Z_j + X[i]Y_j$
8: $\quad\quad (C_b, Z_j) = C_b + Z_j + Q_{mi}M_j$
9: $\quad\quad Z_{j-1} =$ Concatenate($Z_j[k\text{-}1{:}0]$, $Z_{j-1}[2k\text{-}1{:}k]$)
10: $\quad C_a = C_a$ or C_b
11: $\quad Z_{s-1} =$ sign ext(C_a, $Z_{s-1}[2k\text{-}1{:}k]$)
12: **if** $Z \geq M$ **then**
13: $\quad Z = Z - M$
return Z

C. KO Algorithm for Large Integer Multiplication

Karatsuba-Ofman algorithm (KO algorithm) is a well-known algorithm for large integer multiplication. The algorithm is based on the divide-and-conquer approach and splits the n-bit multiplicand and the multiplier into two smaller terms.

$$\begin{cases} X = x_1 * 2^{n/2} + x_0 \\ Y = y_1 * 2^{n/2} + y_0. \end{cases} \tag{3}$$

The product of $X * Y$ is denoted as:

$$\begin{aligned} Z &= X * Y \\ &= x_1 y_1 * 2^n + (x_1 y_0 + x_0 y_1) * 2^{n/2} + x_0 y_0. \end{aligned} \tag{4}$$

According to KO algorithm, $x_1 y_0 + x_0 y_1$ is capable of being replaced by:

$$x_1 y_0 + x_0 y_1 = (x_0 + x_1) * (y_0 + y_1) - x_0 y_0 - x_1 y_1. \tag{5}$$

Denote p_0, p_1 and p_{01} the partial products $x_0 y_0$, $x_1 y_1$ and $(x_0 + x_1) * (y_0 + y_1)$, respectively, Equation 4 can be rewritten as:

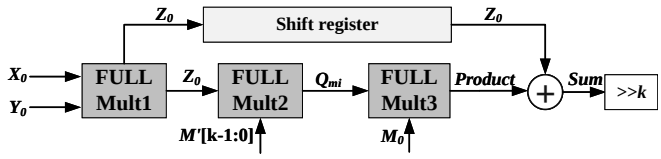
$$\begin{aligned} Z &= p_1 * 2^n + (p_{01} - p_0 - p_1) * 2^{n/2} + p_0 \\ &= Z_2 * 2^n + Z_1 * 2^{n/2} + Z_0. \end{aligned} \tag{6}$$

The KO algorithm can significantly reduce the number of multiplications necessary for large integer multiplication. Notably, it trims down the requirement from 4 multiplications in the schoolbook multiplication algorithm to 3 multiplications. This streamlined approach not only directly enhances the computational efficiency but also has the potential to reduce the hardware consumption, particularly when dealing with operands with large bit-widths.

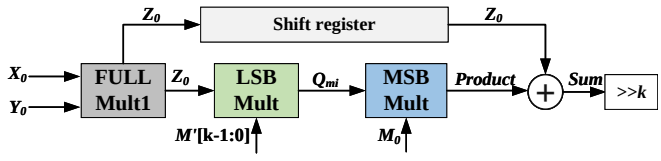
III. Improved modular multiplication algorithm

In this section, we propose an enhanced word-based Montgomery modular multiplication that employs the KO (or KO-3) algorithm to compute all partial products of inputs before Montgomery reduction. This approach reduces resource consumption and better accommodates reconfigurable modular multipliers. Additionally, as shown in Fig. 1a, multiplications employed in Steps 3 to 5 of Algorithm 2 are all full-scale multiplications (FULL-Mult) that perform calculations completely. However, according to the Montgomery algorithm, Q_{mi} only needs to take the low half of the multiplication result (FULL-Mult2), while bits in the low half of $Z_0 + \mathrm{FULL\text{-}Mult3}(\mathrm{Q_{mi}}, \mathrm{M_0})$ are all 0 and will be discarded in subsequent steps. Therefore, partial resources in FULL-Mult2 and FULL-Mult3 are unnecessary.

For computing the partial multiplication results, we introduce two truncated multiplications: MSB multiplication (MSB-Mult) and LSB multiplication [21] (LSB-Mult). In MSB-Mult, only the higher partial product is computed. In LSB-Mult, the calculation for the higher partial product is omitted. The improved version is shown in Fig. 1b. Unlike previous MSB-Mult approaches [9], [19], which introduce larger errors, our proposed MSB-Mult incurs an error of only 1, making it more compatible with word-based Montgomery modular multiplication.



(a) Partial multiplications in Algorithm 2



(b) Improved version introducing truncated multiplications

Fig. 1: Multiplications in Algorithm 2 and its improved version

A. Word-based Modular Multiplication with KO Algorithm

In Algorithm 2, partial products like $X[i]Y_0$ (or $X[i]Y_j$) are computed in each loop during Step 3 (or Step 7). In scenarios where inputs are divided into s words, s^2 multipliers are needed for calculating these partial products. However, when applied to a reconfigurable modular multiplier, this approach is not efficient. By leveraging the KO (or KO-3) algorithm to compute all partial products of inputs before Montgomery reduction, we can significantly reduce the number of required multipliers. In comparison to Algorithm 2, Algorithm 3 manages to save 1 or 3 large integer multiplications, each with a bit-width of up to 128 in our design.

B. MSB Multiplication

In the Montgomery algorithm, the lower segments of $Z + Q_m M$ (Algorithm 1) and $Z_0 + Q_{mi} M_0$ (Algorithm 2) are all zero, which can be eliminated directly through shifting operations. When only the higher part of the product and addition are calculated, the resources employed by the multiplier and adder, as well as the latency induced by the carry chain, would inevitably decrease. In contrast to the LSB-Mult, the computation for the lower part cannot be directly discarded here. Doing so might introduce substantial errors due to the absence of carry from the lower part. Hence, we introduce MSB-Mult, which omits the calculation for the lower part, with a predictable error.

Similar truncated multipliers have been introduced in prior works for Barrett modular multiplication [7], [19]. The errors introduced by truncated multipliers employed in these works will be up to 5. As a result, the error in the final result is only a few times the modulus, which is easy to compensate for the Barrett modular multiplication. However, this approach is not well-suited for word-based Montgomery algorithms in which compensating only work under error being 1.

In the context of word-based Montgomery modular multiplication, errors introduced by MSB-Mult necessitate immediate compensation before initiating the next reduction to prevent error propagation. However, it is impossible to correct unpredictable errors in this step. Predictable error is thus necessary. Moreover, in addition to not being completely suitable for the Montgomery algorithm, the approach in [19] is inefficient due to taking more resources when introducing greater error. In [7], the details regarding bit-width allocation and proof are not provided.

We present our MSB-Mult as follows. We use a strategy similar to the divide-and-conquer approach. Each recursive level of MSB multiplication is divided into two segments, as shown in Equation 3. The computation of each level is shown in Equation 4 which omits $x_0 y_0$, while $x_1 y_1$ is still calculated in the KO algorithm. Subsequently, $x_1 y_0$ and $x_0 y_1$ are also partitioned and computed in the subsequent recursive level. However, the difference from the traditional methods is that the word length of x_0 (or y_0) is not $k/2$, but as shown in Theorem 1.

Theorem 1. *If the allocation scheme of each level in MSB-Mult meets Equation 7*

$$R_l = \begin{cases} k/2 - 1 & \text{if } l = 1 \\ \lfloor R_1 / 2^{l-1} \rfloor & \text{if } 1 < l \leqslant L, \end{cases} \tag{7}$$

where R_l is the word length of x_0 (or y_0) in l-th recursive level of MSB-Mult, k is the width of MSB-Mult input and h is

an integer satisfying $k = 2^L(h+1)$, L is the selected number of recursive levels of MSB-Mult and $L \in \{1,2,3,4\}$, then we have

$$\lfloor \mu \rfloor - \lfloor \widetilde{\mu} \rfloor = 1,$$

where $\mu = (Z_0 + Q_{mi}M_0)/2^k$ is the result without error, $\widetilde{\mu} = [\lfloor Z_0/2^{k-c} \rfloor + \text{MSB-Mult}(Q_{mi}, M_0)]/2^c$ is the result with error introduced by MSB-Mult and discarded part in Z_0. Here, the value c is the width of word reserved in lower half part of Z_0, and $c = h+1$. The value $k-c$ represents the width of the discarded part in Z_0.

Proof. Please find the proof of Theorem 1 in appendix. □

Therefore, once we have calculated $(Z_0 + Q_{mi}M_0)/2^k$ approximately by MSB-Mult, we only need to correct the error of the result by adding 1.

An illustrative example with $k = 128$ and $L = 3$ is depicted in Fig. 2. The implementation of MSB-Mult is structured into three levels, with $R_1 = 63$, $R_2 = 31$, and $R_3 = 15$. Within Fig. 2, the regions shaded in gray mean that this particular section of the result is computed using the KO algorithm. Conversely, the areas highlighted in green indicate that the calculation of the result adheres to Equation 4, excluding the $x_0 * y_0$ term at each hierarchical level. Although we illustrate a 3-level structure as an example, it's worth noting that for larger bit-width scenarios, a similar multi-level structure can also be obtained.

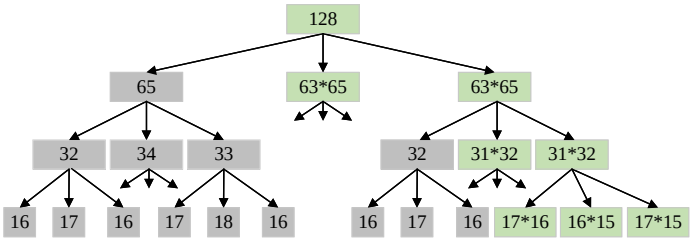


Fig. 2: The MSB multiplier.

C. LSB Multiplication

As depicted in Step 1 of Algorithm 1 and Step 4 of Algorithm 2, the higher partial product of $(Z_0 \mod 2^k)M'$ has no influence on the result. Hence, one proposition is to omit the calculation of this part and exclusively compute the product of the lower part. However, Equation 6 reveals that no multipliers are spared in the truncated multiplier based on the KO algorithm. Similar to the full-scale multiplier, the computations for p_0, p_1, and p_{01} are necessary. However, the KO-3 algorithm, which decomposes X and Y into three terms, as expressed in Equation 8, has the capability to reduce the multiplications in LSB-Mult [21], [35].

$$\begin{cases} X = x_2 * 2^{2k/3} + x_1 * 2^{k/3} + x_0 \\ Y = x_2 * 2^{2k/3} + y_1 * 2^{k/3} + y_0. \end{cases} \tag{8}$$

Based on Equation 8, $X * Y$ is expressed as

$$\begin{aligned} Z =& X * Y \\ =& x_2y_2 * 2^{4k/3} + \\ & (x_1y_2 + x_2y_1) * 2^{3k/3} + \\ & (x_0y_2 + x_2y_0 + x_1y_1) * 2^{2k/3} + \\ & (x_1y_0 + x_0y_1) * 2^{k/3} + x_0y_0. \end{aligned} \tag{9}$$

If x_0y_0, x_1y_1, x_2y_2, $(x_0+x_1)*(y_0+y_1)$, $(x_0+x_2)*(y_0+y_2)$ and $(x_1+x_2)*(y_1+y_2)$ are represented as p_0, p_1, p_2, p_{01}, p_{02} and p_{12} respectively, Equation 9 can be rewritten as

$$\begin{aligned} Z =& X * Y \\ =& p_2 * 2^{4k/3} + \\ & (p_{12} - p_1 - p_2) * 2^{3k/3} + \\ & (p_{02} - p_0 - p_2 + p_1) * 2^{2k/3} + \\ & (p_{01} - p_0 - p_1) * 2^{k/3} + p_0 \\ =& Z_4 * 2^{4k/3} + Z_3 * 2^{3k/3} + \\ & Z_2 * 2^{2k/3} + Z_1 * 2^{k/3} + Z_0. \end{aligned} \tag{10}$$

In Equation 10, only the calculation for 5 partial products, namely p_0, p_1, p_2, p_{01}, and p_{02}, in the lower part of LSB-Mult is necessary, with the computation for p_{12} being omitted. Compared to the utilization of the KO algorithm, LSB-Mult using the KO-3 algorithm exhibits superior performance.

D. Improved Modular Multiplication Algorithm

Combining LSB-Mult and MSB-Mult, we propose a improved word-based Montgomery multiplication as follows.

Algorithm 3 Improved modular multiplication algorithm

Preprocess: M' = $-M^{-1}$ mod R, s=$\lceil \frac{n}{k} \rceil$
Input: $X, Y \in$[0,M), R = $2^n > M$, gcd(R, M) = 1
Output: $Z = XYR^{-1}$ mod $M \in$[0,M)

$Z = 0$
$\{Z_0, Z_1, ..., Z_{2s-2}\}$ = KO (or KO3) $algorithm(X, Y)$
for i=0 to s-1 **do** ▷ Outer loop
  Q_{mi} = LSB-Mult(Z_0 mod 2^k, M')
  Z_0 = $\lfloor Z_0/2^{k-c} \rfloor$ + MSB-Mult(Q_{mi}, M_0)
  $sum_{i/0}$ = $\lfloor Z_0/2^c \rfloor$ + 1
  for j=1 to $s-1$ **do** ▷ Inner loop
    $sum_{i/j}$ = Z_j + $Q_{mi}M_j$
  Z_0 = $sum_{i/0}$ + $sum_{i/1}$
  for l=1 to $2s-3-i$ **do**
    if $1 \leqslant l < s-1$ **then**
      Z_l = $sum_{i/l+1}$
    else if $s-1 \leqslant l \leqslant 2s-3-i$ **then**
      Z_l = Z_{l+1}
Z = $Z_{s-2} * 2^{k(s-2)}$ + ... + $Z_1 * 2^k$ + Z_0
if $Z \geq M$ **then**
  $Z = Z - M$
return Z

In Algorithm 3, the calculation of all partial products p leverages the KO (or KO-3) algorithm before Montgomery reduction. When s exceeds 1, the number of multiplications

required for $X * Y$ falls below s^2 [35]. Subsequent steps performing Montgomery reduction on $Z_0, Z_1, ..., Z_{2s-2}$ can reuse the outputs of the KO algorithm. Each outer loop iterates through the lower k bits of the lower order coefficients (Z_0) sequentially to obtain Q_{mi} as described in Step 4 of the algorithm. Notably, this step discards the higher k bits of the product. To optimize resource utilization, LSB-Mult replaces FULL-Mult in this context.

Similar to Algorithms 1 and 2, the lower k bits of $Z_0 + Q_{mi} * M_0$ are always zero. This enables the introduction of MSB-Mult for calculating $Q_{mi} * M_0$, followed by error correction. Meanwhile, $Q_{mi} * M_j$ $(j > 0)$ is calculated and added to coefficients Z_j to update $sum_{i/j}$. Subsequently, Z_l is updated and the algorithm proceeds to the next outer loop for further reduction. After s iterations of the outer loop, Step 15 calculates the weighted sum of $s-1$ coefficients. Detecting the sign bit in the difference between the sum and the modular value yields the final result.

IV. Proposed Reconfigurable Modular Multiplier

In this section, we will present our proposed reconfigurable modular multiplier.

A. Overview of the Architecture

The general structure of the fully pipelined multiplier implementing Algorithm 3 is shown in Fig. 3. While the input bit-width n varies in different modes, the radix k remains constant at 128. The design adapts resource allocation based on the chosen bit-width mode, leveraging Algorithm 3 and Fig. 3 to fulfill computational requirements. Blue arrows denote the utilization of only lower k bits, whereas brown arrows indicate discarded data below $(k-c)$-th bit. Corrections word with c bits are discarded after Step5 of Algorithm 3, as depicted by red arrows in Fig. 3.

Taking advantage of the word-based algorithm, we can devise a computational array comprising several k-bit multipliers, adders, and shift registers, as illustrated in Fig. 4. In this work, the computational array includes 9 k-bit and 3 (k+1)-bit full-scale multipliers (FULL-Mult), 3 k-bit LSB-Mults, 3 k-bit MSB-Mults, 9 shift register arrays, and several adders to accommodate 384-bit, 256-bit, and 128-bit modular multiplications. The region (a) incorporates three k-bit and three (k+1)-bit FULL-Mults, responsible solely for computing partial products p. In 384-bit mode, the KO-3 algorithm is employed, while in 256-bit mode, the KO algorithm is utilized. Consequently, the multipliers required for 2-way 256-bit inputs in Step 2 of Algorithm 3 precisely meet those for 1-way 384-bit and 6-way 128-bit inputs.

The k-bit LSB-Mults in the region (b) are used to calculate Q_{mi}. The k-bit MSB-Mults in the region (c) are used to calculate $Q_{mi} * M_0$. The remaining k-bit FULL-Mults in region (d) are utilized for calculating either Q_{mi} or $Q_{mi} * M_j$ depending on the selected mode. According to Algorithm 3, s^2+s multipliers are required for the reduction step. Therefore, 12 multipliers present in these three regions can fulfill the requirements for 1-way 384-bit. Based on the chosen bit-width, these resources can be reassembled, following Algorithm 3 and Fig. 3, in a pipelined manner through the reconfigurable connections.

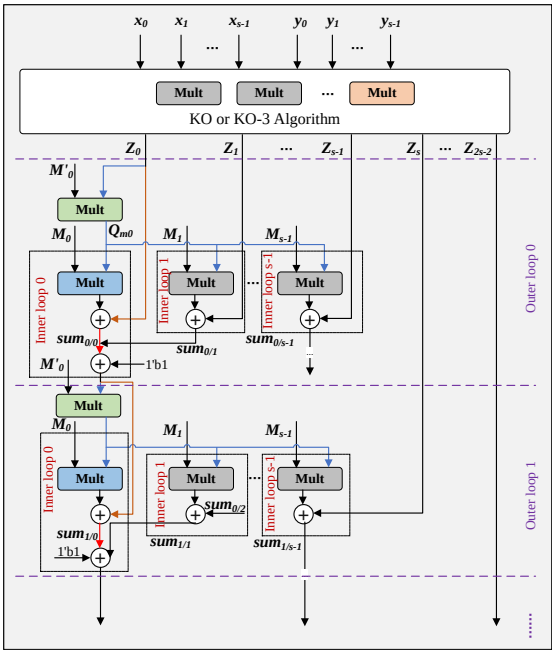


Fig. 3: The pipelined data flow of Algorithm 3.

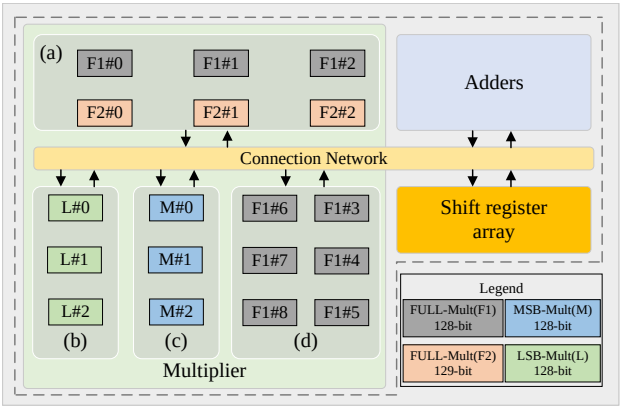


Fig. 4: The overall architecture of proposed modualr multiplier. The 128-bit and 129-bit FULL-Mult are denoted by $F1$ and $F2$, respectively. M and L denote MSB-Mult and LSB-Mult, respectively.

In 256-bit and 128-bit modes, the available multiplier resources are not only sufficient for one modular multiplication pipeline. Utilizing the reconfigurable connections, the remaining multipliers are used to construct more modular multiplication pipelines, enabling these pipelines to operate in parallel. In these modes, s is either 2 or 1, respectively. These 12 multipliers also precisely meet 2-way 256-bit (needs $2*(2^2+2)$ multipliers) and 6-way 128-bit (needs $6*(1^2+1)$ multipliers) calculations. Therefore, the utilization of multiplication resources is 100% for all modes.

B. Reconfigurable Pipelined Modular Multiplier in Different Modes

According to configuration, resources in Fig. 4 are organized as shown in Fig. 3.

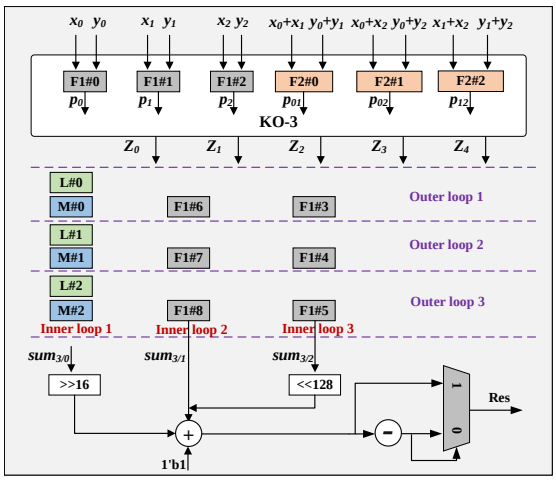


Fig. 5: The resource allocation and data flow for 384-bit mode. The 128-bit and 129-bit FULL-Mult are denoted by $F1$ and $F2$, respectively. M and L denote MSB-Mult and LSB-Mult, respectively.

384-bit mode: Fig. 5 illustrates the data flow of the modular multiplier in 384-bit mode. The KO-3 algorithm is employed to compute the product $X * Y$, utilizing six FULL-Mults located in region (a) of Fig. 4. Three 128-bit FULL-Mults are dedicated to calculating p_0, p_1 and p_2, respectively, while three 129-bit FULL-Mults handle the calculations for p_{01}, p_{02} and p_{12}. Following the addition and subtraction operations, $Z_0, Z_1, ..., Z_4$ are forwarded to the next pipeline stage. The lower portion of Z_0 is fed to LSB-Mult0, while the higher portion, encompassing the higher 128 bits and correction bits, is fed to shift registers for the calculation of $Z_0 + Q_{mi}M_0$. The circuit implementation details for each inner and outer loop are presented in Fig. 3. After three inner loops and three outer loops, the corrected sum is subtracted from the modular value, and the final result is chosen based on the sign bit.

256-bit mode: Due to the reduced number of multipliers required for both inner and outer loops in 256-bit mode, the remaining multipliers, as shown in Fig. 6, are reassigned and reassembled to handle the modular multiplication of another set of operands. Notably, to minimize area and latency by reducing multiplexer usage, we prioritize minimal changes to the data path compared to the 384-bit mode. For instance, only the inputs to FULL-Mult #1 and FULL-Mult #2 are changed. Similarly, only the inputs to 129-bit FULL-Mult #1/#2 are replaced, while LSB-Mult #0/#1, MSB-Mult #0/#1, and 128-bit FULL-Mult #6/#7, responsible for the first set of data's modular multiplication, remain unchanged. The detailed circuit implementation for each inner and outer loop remains similar to that presented in Fig. 3.

128-bit mode: In 128-bit mode, only three multipliers are required for a single modular multiplication. All the multipliers shown in Fig. 4 are precisely sufficient for the modular multiplication of six sets of operands simultaneously as shown in Fig. 7.

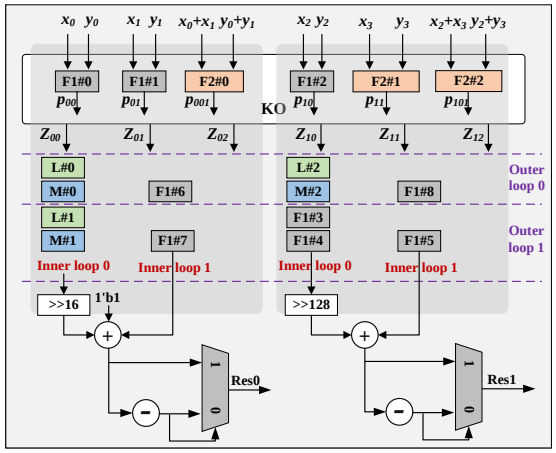


Fig. 6: The resource allocation and data flow for 256-bit mode. The 128-bit and 129-bit FULL-Mult are denoted by F1 and F2, respectively. M and L denote MSB-Mult and LSB-Mult, respectively.

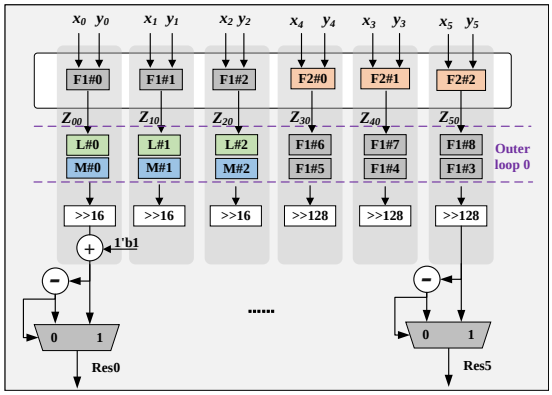


Fig. 7: The resource allocation and data flow for 128-bit mode. The 128-bit and 129-bit FULL-Mult are denoted by F1 and F2, respectively. M and L denote MSB-Mult and LSB-Mult, respectively.

V. Evaluation

This section presents an implementation of a reconfigurable modular multiplier tailored for the ASIC platform. To underscore the efficacy of our truncated multipliers, we further show the comparison between FULL-Mult and truncated multipliers. Leveraging the Synopsys Design Compiler, we implement and synthesize these works with a TSMC 28 nm technology.

As prior fully pipelined architectures primarily being implemented on FPGA platforms, we also replicate and evaluate our design on the same platform for a fair comparison. However, it should be noted that the reconfigurability inherent in our approach showcase greater advantages in ASIC platforms. We implement two reconfigurable NTT pipelines with multiple bit-widths support to benchmark the performance of our design. NTT is one of the computations needed to be accelerated in ZKP and HE.

A. Evaluation of Truncated Multipliers

We implement and compare 128-bit FULL-Mult and truncated multipliers used in our reconfigurable pipelined modular multiplier to show the enhancement of truncated multipliers. To obviously show the enhancement of our truncated multipliers compared with previous work, we also conduct a comparison with the truncated multipliers in Barrett modular multiplier proposed in [19]. However, due to the unavailability of the Quartus Prime Pro Edition software and the Intel Stratix10 FPGA, we replicate the methodology employed in the aforementioned work with the 28nm technology. Therefore, we also implement 1024-bit FULL-Mult and truncated multipliers based on Section III with the 28nm technology.

TABLE I: Comparison between truncated multipliers and FULL-Mults

Design		Ours†		[19]†
Bit-Width		128	1024	1024
Area ($10^3 um^2$)	FULL-Mult	28.9	1519.6	1519.6
	LSB-Mult‡	21.5(25%)	1102.5(27%)	1475.8(3%)
	MSB-Mult‡	22.8(21%)	1263.0(16.8%)	1266.6(16.6%)
Power (mW)	FULL-Mult	25.47	1962.7	1962.7
	LSB-Mult‡	15.53(39.0%)	1531.6(22%)	1899.3(3.2%)
	MSB-Mult‡	16.59(34.9%)	1673.7(14.7%)	1677.2(14.5%)

† The 128-bit multipliers are synthesized based on TSMC 28 nm library with 1000MHz clock, while 1024-bit multipliers with 800MHz.
‡ Data in parentheses show the area and power improvements of truncated multipliers compared to FULL-Mult.

We synthesize these works under 800MHz and 1000MHz clock frequency. The corresponding results and enhancement of FULL-Mults and truncated multipliers with 128-bit and 1024-bit are shown in Table I.

In our 128-bit multipliers, FULL-Mult is implemented using the KO algorithm with recursive 3 levels. While LSB-Mult and MSB-Mult are all implemented as described in Section III. Compared to the FULL-Mult, the LSB-Mult saves area by 25%, while the MSB-Mult saves area by 21%. More appreciably, the LSB-Mult and MSB-Mult achieve 39.0% and 34.9% reductions in power consumption compared to FULL-Mult.

Our 1024-bit FULL-Mult is implemented using the KO algorithm with recursive 5 levels. While our 1024-bit LSB multiplier utilizes the KO-4 algorithm (KO algorithm by-4 decomposition) [35], eliminating two 256-bit multipliers. The MSB-Mult is implemented with a recursive 5-level structure similar to that described in III-B. Our LSB-Mult achieves 27% area reduction compared to FULL-Mults and 25% reduction compared to [19]. The MSB multiplier exhibits a 16.8% area reduction compared to FULL-Mult. Our design exhibits a similar area reduction compared to [19], while [19] discards more information and introduces larger errors up to 5. However, only an error of 1 is introduced into our design.

For power consumption, our LSB-Mult and MSB-Mult realize 22% and 14.7% reductions compared to FULL-Mult, respectively. The enhancement of these two truncated multipliers to [19] is similar to that in area. Our LSB-Mult achieves 19.4% reduction compared to [19], while the power consumption results of two MSB-Mults are similar. The total areas of the two modular multiplication designs amount to 3897.6 $10^3 um^2$ and 4279.7 $10^3 um^2$, respectively. Our work, therefore, realizes a 9% area reduction compared to [19].

B. Evaluation of Reconfigurable Pipelined Modular Multiplier with ASIC Technology

The performance of the modular multiplier is evaluated through various metrics, including frequency (MHz), area ($10^3 um^2$), power (mW), cycles, throughput (Gbps) and efficiency (Gbps/$10^3 um^2$). The cycles metric denotes the number of cycles for one modular multiplication. The efficiency is defined as follows

$$\text{Efficiency} = \frac{\text{Throughput}}{\text{Area}}. \tag{11}$$

Design in [36] employs carry-save adders inside the iteration without carry propagation, realizing an extremely short critical path. When processing 128-bit and 256-bit modular multiplication, this design can work under 1886MHz and 1785MHz. However, it needs multiple iterations even in pipelined version. The number of cycles needed to complete one modular multiplication is 18. This has a very negative impact on throughput and efficiency. Compared to this work, our design, in 128-bit mode, can process 6 modular multiplications per cycle and exhibits much higher throughput. Though our design takes more area and has a lower frequency, it achieves a 5.8× enhancement in efficiency than [36]. While in 256-bit mode, our design can process 2 modular multiplications per cycle and realizes an enhancement by 7.0×. In 384-bit mode, our design can only complete one modular multiplication per cycle, but the efficiency is still relatively high. More importantly, this flexibility in bit-width configuration caters to the diverse needs of privacy-preserving applications.

Our design occupies an area of 564.0 $10^3 \mu m^2$ and consumes 471.6 mW of dynamic power. In 128-bit mode, the pipeline fills up after 6 clock cycles, subsequently generating six results per cycle with a throughput of 750 Gbps. For 256-bit mode, the first two results are generated after 12 clock cycles, achieving a throughput of 500 Gbps. Similarly, in 384-bit mode, the respective values are 17 clock cycles and 375 Gbps.

We also compare our proposed modular multiplier in different bit-widths with software-based implementation. Based on Google Benchmark library [37], we benchmark the Montgomery modular multiplication in ctbignum library [38] on a platform with 2.6 GHz Intel Xeon Platinum 8358 processor with 1TB memory. Only one thread is used in the test. We use g++ compiler to compile the benchmark with -O2 and -O3 optimization, respectively. Based on this processor and configuration, we run the 128-bit, 256-bit, and 384-bit Montgomery modular multiplication. As shown in Table III, we provide results in terms of the number of processed modular multiplications per second on different platforms and the speedup of our design with respect to CPU. In 128-bit and 256-bit modes, our design can process 6 and 2 modular multiplications per cycle in parallel. Under a 1GHz clock, 6,

TABLE II: Performance of reconfigurable modular multiplier

Design	Library	Bit-Width	Frequency (MHz)	Area ($10^3 um^2$)	Power (mW)	Cycles	Throughput (Gbps)	Efficiency (Gbps/$10^3 um^2$)	Improvement in Efficiency
[36]	TSMC 22 nm	128	1886	58.1	-	18	13.4	0.23	-
		256	1785	200	-	18	25.4	0.127	-
Ours	TSMC 28 nm	128†	1000	564.0	471.6	1	750	1.33	5.8×
		256†	1000	564.0	471.6	1	500	0.862	7.0×
		384	1000	564.0	471.6	1	375	0.665	-

† In 128-bit and 256-bit modes, our design can process 6 and 2 modular multiplications in parallel per cycle, respectively.

TABLE III: Comparison of modular multipliers on different platforms. OP/s means one modular multiplier per second here.

Bit-Width	CPU (-O2)	CPU (-O3)	**Ours**	Speedup	
				Ours/CPU(-O2)	**Ours**/CPU(-O3)
128	68 MOP/s	98 MOP/s	6 GOP/s	88	61
256	23 MOP/s	37 MOP/s	2 GOP/s	87	54
384	10 MOP/s	19 MOP/s	1 GOP/s	100	53

2, and 1 billion modular multiplications in different modes are processed by our implementation. Compared to software-based implementation with -O2 optimization, our design achieves 87× to 100× speedup. While compared to -O3 optimized software implementation, our design still achieves 53× to 61× speedup.

C. Evaluation of Reconfigurable Pipelined Modular Multiplier on FPGA platform

We also implement our proposed fully pipelined modular multiplier on XILINX XCVU9P and Virtex-6 FPGAs. Our design on the FPGA platform is implemented in a similar way as described in Section III and Section IV. When configured in 128-bit mode, our design can complete 6 modular multiplications in parallel per cycle. While in 256-bit mode, it can process 2 modular multiplications per cycle. In 384-bit mode, it can only process one modular multiplication per cycle. The empirical results are presented in Table IV. It is important to note that the throughput and efficiency values for 256-bit and 128-bit modes are calculated assuming full capacity operation.

Table IV presents a more granular comparison of our work with previous fully pipelined works. To ensure a fair comparison, we introduce a metric of Equivalent Gates to represent the equivalent gate count of LUTs and FFs. The equivalent gates for 6-input LUTs and FFs are 15 [40] and 12 [41], respectively. The DSPs are also equivalently converted into LUTs required to implement a single DSP multiplier and finally represented by equivalent gates. In the XILINX XCVU9P platform, one DSP can be equivalently expressed as 485 LUTs. The performance of the modular multiplier is evaluated across various metrics, including frequency (MHz), latency (ns), throughput (Mbps), area (FFs, LUTs, DSPs, and Equivalent Gates), and efficiency (Mbps/1K Gates).

Design in [21] implements a fully pipelined Montgomery modular multiplier without supporting multiple bit-widths. Suffering from the lower frequency, the efficiency of this modular multiplier is low. Compared with [21], our implementation shows significant improvements. Our implementation utilizes more resources but achieves a higher frequency and can process 2 modular multiplications per cycle. Therefore, it offers higher throughput. Specifically, our design provides 2.2× higher efficiency when compared to [21].

Design in [39] implements a low-latency modular multiplication. However, despite consuming a much higher area, this work does not offer ideal latency. In 128-bit mode, compared with [39], our design exhibits an enhancement by 4.8× in efficiency. While in 256-bit mode, our design exhibits an enhancement by 8.5×.

Design in [9] implements a fully pipelined Montgomery modular multiplier used in Multi-scalar multiplication (MSM). This modular multiplier can only support one 384-bit modular multiplication per cycle. Compared to this work, our implementation requires a larger area but achieves higher frequency and throughput. The reconfigurable connection increases the area, resulting in decreased efficiency. For a fixed-bit-width design, as demonstrated in Table IV, due to the introduction of truncated multipliers, the efficiency would be higher. Though our design has lower efficiency compared to [9] but exhibits higher flexibility, which is particularly valuable for ASIC designs. This feature will significantly broaden its potential applications.

The results presented in Table IV demonstrate that the use of truncated multipliers has a significant impact on reducing area and latency. This is important for effectively saving the area and increasing the frequency of the proposed modular multiplier.

D. Scalability Analysis of Architecture

In the above subsections, we discuss the performance of the proposed reconfigurable modular multiplier and corresponding architecture in different bit-width modes. To explore the scalability of the proposed architecture, we scale up the architecture to support larger bit-width. In the ZKP, 768-bit is another most used bit-width, such as the MNT4753 curve. According to the proposed method, we therefore implement a larger reconfigurable modular multiplier supporting 768-bit operands. At the same time, this modular multiplier can

TABLE IV: Comparison of pipelined Montgomery modular multiplier implementations on FPGA.

Device	Design	Bit-Width	Frequency (MHz)	Latency (ns)	Area				Throughput (Mbps)	Efficiency (Mbps/1K Gates)	Improvement in Efficiency
					FFs	LUTs	DSPs	Equivalent Gates			
Virtex-6	[21]	256	68	14.7	-	187.9K	0	2818.5K	17408	6.18	-
	Ours	256 †	127.6	7.8	37582	294.3K	0	4865.5K	65536	13.47	2.2×
XCVU9P	[39]	128	-	11.85	1472	24120	81	968.7K	10802	11.15	-
		256	-	17.55	2952	87622	289	3452.2K	14587	4.23	-
	[9]	384	250	4	46866	43144	324	3566.7K	96000	26.92	-
	Ours	128 †	319	2.71	33143	47978	474	4565.7K	244992	53.66	4.8×
		256 †	319	2.77	33143	47978	474	4565.7K	163328	35.77	8.5×
		384	319	3.04	33143	47978	474	4565.7K	122496	26.83	0.99× ‡
		384♯	333	3.00	24373	33808	451	4080.6K	127872	31.3	1.16×

† In 128-bit and 256-bit modes, our design can process 6 and 2 modular multiplications in parallel per cycle, respectively.
‡ The efficiency is slightly lower due to resources introduced to support multiple bit-widths, while it exhibits more flexibility.
♯ Leveraging truncated multipliers, we also implement a fixed bit-width 384-bit modular multiplier.

also process 3-way 384-bit, 6-way 256-bit, and 20-way 128-bit modular multiplications. Under the 1000 MHz clock, the scaled-up modular multiplier, as shown in Fig. 8, improves the throughput by 3.1× on average, in 128-bit, 256-bit, and 384-bit modes. Synthesized under this clock, this modular multiplier takes up 1818 $10^3\mu m^2$ which scales up 3.2× compared to the proposed modular multiplier. With 3.2× resources, the scaled-up modular multiplier achieves almost the same improvement in terms of throughput. Therefore, as shown in Fig. 8, the area efficiency of the scaled-up modular multiplier is similar to that of the proposed multiplier in the lower bit-width mode.

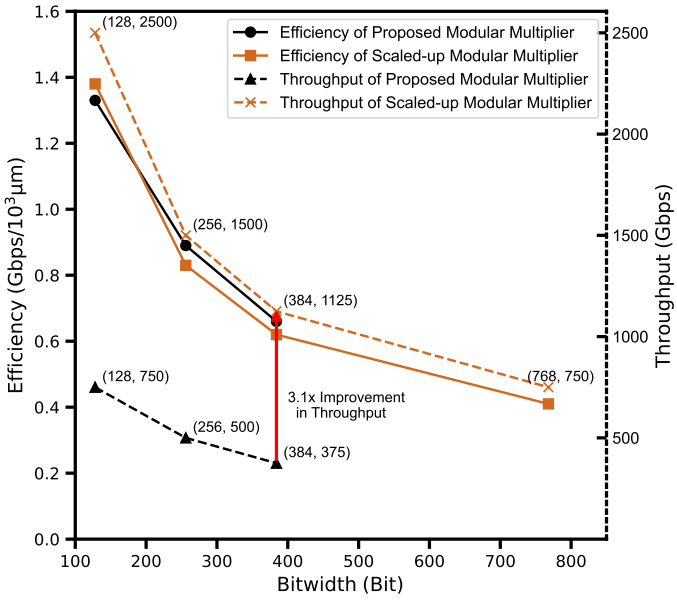


Fig. 8: Throughput and efficiency in each bit-width mode of proposed and scaled-up modular multiplier under 1000 MHz clock.

TABLE V: Evaluation of NTT constructed by the proposed reconfigurable modular multiplier.

Design	Platform	Frequency (MHz)	Area (mm^2)	Bit-Width	Times(μs)	
					2^{16}	2^{20}
SAM [43]	Xilinx Alveo U250	100	594K LUTs 6.7K DSPs	256	1240	12610
GZKP [44]	GPU V100	-	-	256	90	1070
PipeZK [6]	UMC 28nm	300	15.04	256	281	11000
RPU [33]	GF 12nm	1680	20.5	128	6.7	-
Ours	TSMC 28nm	800	20.02	128	14.9	222.9
		800	20.02	256	41.9	658.3
		800	20.02	384	82.9	1313.7

E. Reconfigurable NTT Using Proposed Modular Multiplier

Based on the proposed reconfigurable modular multiplier, we construct two reconfigurable SDC-structure-based [42] NTT pipelines with variable bit-width support to benchmark our work in real-world applications. Determined by the parallelism of reconfigurable modular multiplier, each NTT pipeline can process 6-lane 128-bit, 2-lane 256-bit, or 1-lane 384-bit NTT in parallel. Two NTT pipelines therefore meet the requirement of processing 12-lane 128-bit, 4-lane 256-bit, or 2-lane 384-bit NTT in parallel. The performance of our NTT based on the proposed modular multiplier and both NTT works based on FPGA, GPU, and ASIC are presented in Table V.

Compared to FPGA-based implementation in SAM [43], our implementation has an obvious speedup that benefits from the native frequency advantage of the ASIC platform. At the same time, we retain some flexibility due to modular multipliers with variable bit-width support. Based on the proposed modular multipliers, the NTT here achieves 29.6× and 19.2× speedups against SAM. Even compared to faster work GZKP [44] implemented on GPU, our NTT still exhibits average enhancement by 1.9×. Compared to the design in PipeZK [6] which is synthesized using Synopsys' Design Compiler (DC)

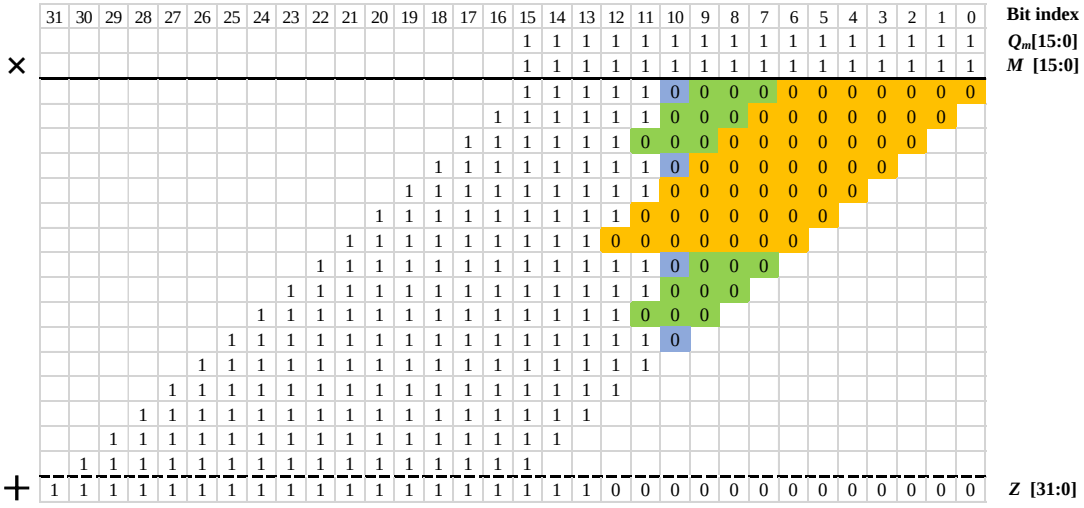


Fig. 9: The 16-bit MSB multiplier.

with UMC 28nm technology, we provide higher flexibility in addition to obvious enhancements. For PipeZK, only one bit-width is supported. In 128-bit mode, the NTT in RPU [33] is synthesized using DC with advanced technology and achieves higher frequency and better performance. If we scaled two designs to the same frequency, our implementation would have a similar throughput to RPU. Furthermore, our NTT can support more bit-width modes.

VI. Conclusion

This paper presents a fully pipelined Montgomery modular multiplier that supports variable bit-widths. The design allows for flexible reconfiguration of computation units, including FULL-Mults, LSB-Mults, MSB-Mults, adders, and shift registers, to meet different bit-width requirements. Our design supports modular multiplication of 384-bit, 256-bit and 128-bit. For small bit-width data such as 256-bit and 128-bit, the design enables the processing of multiple sets of data in parallel without wasting resources. The experimental results demonstrate that the proposed modular multiplier has very low latency and high throughput. Through the introduction of LSB-Mults and MSB-Mults and employing a suitable KO algorithm, we were able to efficiently reduce the area of the design. The high throughput and flexibility of our design provide efficient support for privacy-preserving computing applications such as HE and ZKP.

VII. Appendix

As shown in Theorem 1, the word length of x_0 (or y_0) in each level is $\frac{k}{2}-1$ or $\lfloor \frac{R_1}{2^{l-1}} \rfloor$. When k meets the Equation 7 and L is the number of recursive levels of MSB-Mult, the word lengths of x_0 (or y_0) of each level are

$$\begin{cases} R_1 = 2^{L-1}h + 2^{L-1} - 1 \\ R_2 = 2^{L-2}h + 2^{L-2} - 1 \\ \cdots \\ R_L = 2^0 h + 2^0 - 1. \end{cases} \tag{12}$$

Under the assumption that all bits of omitted parts (x_0 and y_0) are all 1, the maximum error, introduced by neglecting $x_0 y_0$, in each level is Δ_1, Δ_2, Δ_3, and Δ_L, respectively, as shown in Equation 13.

$$\begin{cases} \Delta_1 = (2^{R_1} - 1)^2 \\ \Delta_2 = (2^{R_2} - 1)^2 * 2^{R_1} * 2 \\ \Delta_3 = (2^{R_3} - 1)^2 * 2^{R_1+R_2} * 4 \\ \cdots \\ \Delta_L = (2^{R_L} - 1)^2 * 2^{R_1+R_2+\cdots+R_{L-1}} * 2^{L-1}. \end{cases} \tag{13}$$

Combining Equation 12 and Equation 13, we obtain the maximum error Δ_{msb_mult} caused by MSB-Mult:

$$\begin{aligned} \Delta_{msb\_mult} &= \Delta_1 + \Delta_2 + \Delta_3 + \cdots + \Delta_L \\ &= L * 2^{2^L(h+1)-2} - 2^{(2^L-1)(h+1)} + 1 \\ &= L * 2^{k-2} - 2^{k-(h+1)} + 1. \end{aligned} \tag{14}$$

If the width of the correction word c is set as $h+1$, the maximum error introduced by discarding bits below the $(k-c)$-th bit of Z is $2^{k-(h+1)} - 1$. Thus, we can find that

$$\mu - \widetilde{\mu} \leqslant \frac{\Delta_{msb\_mult} + 2^{k-(h+1)} - 1}{2^k} = \frac{L * 2^{k-2}}{2^k} = \frac{L}{4}, \tag{15}$$

where μ is $(Z_0 + Q_{mi}M_0)/2^k$ without error while $\widetilde{\mu}$ is coarse result with error. Omitted parts in each recursive level are impossible to be 0 in Montgomery algorithm, therefore, the Equation 16 is true.

$$0 < \mu - \widetilde{\mu} \leqslant 1, L \in \{1, 2, 3, 4\}. \tag{16}$$

Additionaly, the lower k bits of $Z_0 + Q_{mi}M_0$ are all zero. Thus, we can obtain

$$\lfloor \mu \rfloor - \lfloor \widetilde{\mu} \rfloor = 1. \tag{17}$$

We show an example here. A compact MSB multiplier example, configured for k = 16 and L = 3, is depicted in Fig. 9. Note that the number 1 in the figure only indicates that this bit is calculated, while 0 indicates that the bit is not computed. Based on Equation 7, R_1, R_2 and R_3 are 7, 3 and

1, respectively. The orange area in the figure denotes the part of the first level where no calculations are performed, while the green area and blue area indicate the part without being computed in level 2 and level 3, respectively. The maximum errors introduced by each area and their sum are shown in Equation 18

$$\begin{cases} \Delta_1 = (2^7-1)^2 \\ \Delta_2 = (2^3-1)^2 * 2^7 * 2 \\ \Delta_3 = (2^1-1)^2 * 2^{7+3} * 4 \\ \Delta_{msb\_mult} = \Delta_1 + \Delta_2 + \Delta_3 \\ \quad = 2^{15}+1. \end{cases} \tag{18}$$

The maximum error caused by discarding bits below $(k-c)$-th bit of Z is $2^{14}-1$. Hence, the maximum error in $(Z + Q_m M)/2^k$ introduced by two terms is $(2^{14}+2^{15})/2^{16} = 3/4$. Thus, we have

$$0 < \mu - \widetilde{\mu} \leqslant \frac{3}{4} < 1. \tag{19}$$

REFERENCES

[1] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno, "Pinocchio coin: building zerocoin from a succinct pairing-based proof system," in *Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies*, 2013, pp. 27–30.

[2] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno, "Cinderella: Turning shabby x. 509 certificates into elegant anonymous credentials with the magic of verifiable computation," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 235–254.

[3] H. S. Galal and A. M. Youssef, "Verifiable sealed-bid auction on the ethereum blockchain," in *Financial Cryptography and Data Security: FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers 22*. Springer, 2019, pp. 265–278.

[4] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.

[5] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 459–474.

[6] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "Pipezk: Accelerating zero-knowledge proof with a pipelined architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 416–428.

[7] C. F. Xavier, "Pipemsm: Hardware acceleration for multi-scalar multiplication," Cryptology ePrint Archive, Paper 2022/999, 2022, https://eprint.iacr.org/2022/999. [Online]. Available: https://eprint.iacr.org/2022/999

[8] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[9] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. Varela, "Fpga acceleration of multi-scalar multiplication: Cyclonemsm," Cryptology ePrint Archive, Paper 2022/1396, 2022, https://eprint.iacr.org/2022/1396. [Online]. Available: https://eprint.iacr.org/2022/1396

[10] R. L. Rivest, L. Adleman, M. L. Dertouzos *et al.*, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[11] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[12] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[13] V. S. Miller, "Use of elliptic curves in cryptography," in *Conference on the theory and application of cryptographic techniques*. Springer, 1985, pp. 417–426.

[14] S. Lab, "libsnark: A c++ library for zksnark proofs," 2018, https://github.com/scipr-lab/libsnark. [Online]. Available: https://github.com/scipr-lab/libsnark

[15] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.

[16] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[17] Y. Kong and B. Phillips, "Comparison of montgomery and barrett modular multipliers on fpgas," in *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, 2006, pp. 1687–1691.

[18] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2019, pp. 1–8.

[19] M. Langhammer and B. Pasca, "Efficient fpga modular multiplication implementation," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 217–223.

[20] A. F. Tenca and Ç. K. Koç, "A scalable architecture for modular multiplication based on montgomery's algorithm," *IEEE Transactions on computers*, vol. 52, no. 9, pp. 1215–1221, 2003.

[21] R. Liu and S. Li, "A design and implementation of montgomery modular multiplier," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–4.

[22] A. F. Tenca and Ç. K. Koç, "Scalable methods and apparatus for montgomery multiplication," May 16 2006, uS Patent 7,046,800.

[23] B. Zhang, Z. Cheng, and M. Pedram, "High-radix design of a scalable montgomery modular multiplier with low latency," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 436–449, 2021.

[24] A. A. Abd-Elkader, M. Rashdan, E.-S. A. Hasaneen, and H. F. Hamed, "Fpga-based optimized design of montgomery modular multiplier," *IEEE Transactions on Circuits and Systems II: express briefs*, vol. 68, no. 6, pp. 2137–2141, 2020.

[25] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 459–474.

[26] S. Steffen, B. Bichsel, R. Baumgartner, and M. Vechev, "Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 179–197.

[27] R. Wang, Y. Tang, X. Pei, S. Guo, and F. Zhang, "Block-chain privacy protection scheme based on lightweight homomorphic encryption and zero-knowledge proof," *Computer Science*, vol. 48, no. S2, pp. 547–551, 2021.

[28] B. Zhang, G. Lu, P. Qiu, X. Gui, and Y. Shi, "Advancing federated learning through verifiable computations and homomorphic encryption," *Entropy*, vol. 25, no. 11, p. 1550, 2023.

[29] G. R. Rao, H. Ghanimi, V. Ramachandran, D. Al-Qahtani, P. Dadheech, and S. Sengan, "Enhanced security in federated learning by integrating homomorphic encryption for privacy-protected, collaborative model training," *Journal of Discrete Mathematical Sciences and Cryptography*, vol. 27, pp. 361–370, 01 2024.

[30] S. Lu, J. Zheng, Z. Cao, Y. Wang, and C. Gu, "A survey on cryptographic techniques for protecting big data security: present and forthcoming," *Science China Information Sciences*, vol. 65, no. 10, p. 201301, 2022.

[31] J. Zhao, H. Zhu, F. Wang, R. Lu, H. Li, Z. Zhou, and H. Wan, "Accel: An efficient and privacy-preserving federated logistic regression scheme over vertically partitioned data," *Science China. Information Sciences*, vol. 65, no. 7, p. 170307, 2022.

[32] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter *et al.*, "Homomorphic encryption standard," *Protecting privacy through homomorphic encryption*, pp. 31–62, 2021.

[33] D. Soni, N. Neda, N. Zhang, B. Reynwar, H. Gamil, B. Heyman, M. Nabeel, A. Al Badawi, Y. Polyakov, K. Canida *et al.*, "Rpu: The ring processing unit," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 272–282.

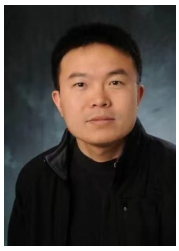
[34] M. Nabeel, D. Soni, M. Ashraf, M. A. Gebremichael, H. Gamil, E. Chielle, R. Karri, M. Sanduleanu, and M. Maniatakos, "Cofhee: A co-processor for fully homomorphic encryption execution," in *2023 Design,*

Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2023, pp. 1–2.
[35] P. L. Montgomery, "Five, six, and seven-term karatsuba-like formulae," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 362–369, 2005.
[36] O. Mazonka, E. Chielle, D. Soni, and M. Maniatakos, "Fast and compact interleaved modular multiplication based on carry save addition," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2022, pp. 1–9.
[37] H. F. Eric, Dominic, "Google benchmark - a library to support the benchmarking of functions," https://github.com/google/benchmark, 2024.
[38] N. J. Bouman, "Multiprecision arithmetic for cryptology in C++ - compile-time computations and beating the performance of hand-optimized assembly at run-time," *CoRR*, vol. abs/1804.07236, 2018. [Online]. Available: http://arxiv.org/abs/1804.07236
[39] E. Öztürk, "Design and implementation of a low-latency modular multiplication algorithm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 6, pp. 1902–1911, 2020.
[40] I. Cadence Design Systems, "Asic prototyping simplified," [Online], https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/pcb-design-analysis/asic-prototyping-tp.pdf.
[41] "Gate count capacity metrics for fpgas," 1997. [Online]. Available: https://api.semanticscholar.org/CorpusID:13161763
[42] A. Cortes, I. Velez, and J. F. Sevillano, "Radix r^k ffts: Matricial representation and sdc/sdf pipeline implementation," *IEEE Transactions on Signal Processing*, vol. 57, no. 7, pp. 2824–2839, 2009.
[43] C. Wang and M. Gao, "Sam: A scalable accelerator for number theoretic transform using multi-dimensional decomposition," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.
[44] W. Ma, Q. Xiong, X. Shi, X. Ma, H. Jin, H. Kuang, M. Gao, Y. Zhang, H. Shen, and W. Hu, "Gzkp: A gpu accelerated zero-knowledge proof system," ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 340–353. [Online]. Available: https://doi.org/10.1145/3575693.3575711

Hao Zhou received the B.S. degree in Electronic Science and Technology from Jilin University, Jilin, China, in 2017 and the M.S. degree in Electronics and Communication Engineering from University of Chinese Academy of Sciences, Beijing, China, in 2020. He is currently working toward the Ph.D degree in Electronic and Information Engineering with the School of Microelectronics, Fudan University, Shanghai, China. His research interests include zero-knowledge proof, fully homomorphism encryption and VLSI implementation of digital systems.

Changxu Liu received the B.S. degree in Microelectronics Science and Engineering from Wuhan University, Wuhan, China, in 2022. He is currently pursuing the Ph.D degree with the State Key Laboratory of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China. His current research interests include hardware-software Co-design for privacy-preserving computing applications and digital IC design.

Lan Yang received the B.S. degree in Microelectronic Science and Engineeringy from Fudan University, Shanghai, China, in 2023. She is currently pursuing the Ph.D degree with the State Key Laboratory of Integrated Chips and Systems, School of Microelectronics, Fudan University, Shanghai, China. Her research interests include homomorphic encryption in privacy-preserving technologies and hardware acceleration.



Li Shang (Member, IEEE) received the Ph.D. degree from Princeton University, Princeton, NJ, USA. He was the Deputy Director and Chief Architect of Intel Labs China, and an Associate Professor of CUBoulder, Boulder, CO, USA. He is a Professor with the School of Computer Science, Fudan University, Shanghai, China. His research interests include computer systems, human-centered computing, machinelearning, VLSI and EDA, with over 100 publications, multiple best paper awards, and nominations. Dr. Shang was a recipient of the NSF Career Award.

Fan Yang (Member, IEEE) received the B.S. degree from Xi'an Jiaotong University, Xi'an, China, in 2003, and the Ph.D. degree from Fudan University, Shanghai, China, in 2008. He is currently a Full Professor with the Microelectronics Department, Fudan University. His research interests include model order reduction, circuit simulation, high-level synthesis, acceleration of artificial neural networks, acceleration of privacy-preserving computing, and yield analysis and design for manufacturability.