Accelerating Fully Homomorphic Encryption in Hardware

Yarkın Doröz, Erdinç Öztürk, Berk Sunar

Abstract—We present a custom architecture for realizing the Gentry-Halevi fully encryption (FHE) scheme. This contribution presents the first full realization of FHE in hardware. The architecture features an optimized multi-million bit multiplier based on the Schönhage Strassen multiplication algorithm. Moreover, a number of optimizations including spectral techniques as well as a precomputation strategy is used to significantly improve the performance of the overall design. When synthesized using 90 nm TSMC technology the presented architecture achieves to realize the encryption, decryption, and recryption operations, in 18.1 msec, 16.1 msec, and 3.1 sec, respectively, and occupies a footprint of less than 30 million equivalent gates.

I. INTRODUCTION

One of the most significant developments in cryptography in the last few years has been the introduction of the first fully homomorphic encryption scheme by Gentry [5]. Gentry’s lattice-based scheme appears to be secure and hence settles an open problem posed by Rivest et al. in 1978 [6]. Since addition and multiplication on any non-trivial ring constitute a universal set of gates, a fully homomorphic encryption scheme – if made efficient– allows one to employ untrusted computing resources without risk of revealing sensitive data. Computation is carried out directly on ciphertexts and the true result of circuit evaluation is not revealed until after decryption.

FHE holds great promise for numerous applications including private information retrieval and search, data aggregation, electronic voting, biometrics, etc. In general, by deploying FHE we may mitigate the vulnerabilities stemming from imperfect software. FHE allows sensitive data to be processed in ciphertext form directly on untrusted servers. Only when the information is to be accessed or visualized it will be downloaded and decrypted on the users trusted machine.

Unfortunately, FHE has not yet sufficiently matured to be used in real-life deployments. The bandwidth overhead caused by ciphertext expansion is prohibitive. More significantly, after every few bit operations the ciphertext needs to be homomorphically re-encrypted to manage the growth in noise. Recryption is a computationally expensive operation that takes in the order of seconds even on high end platforms. In [3], an FPGA implementation draft for improving the speed of FHE primitives was proposed. However, no implementation results were presented.

While, more efficient schemes are appearing in the literature, e.g. see [7], [8], [9], [10], we would like to take a snapshot of the attainable hardware performance of the Gentry-Halevi FHE variant. We are motivated by the fact that the first commercial implementations of most public key cryptographic schemes, e.g. RSA, Elliptic Curve DSA schemes have been via limited functionality hardware products due to the efficiency shortcomings on general purpose computers. We speculate that a similar growth pattern to emerge in the maturation process of FHEs. While the obvious efficiency shortcomings of FHE’s are being worked out, we would like to pose the following question: How far are FHE schemes from being offered as a hardware product? Clearly answering this question would be a major overtaking deserving the evaluation of all FHE variants with numerous implementation techniques. Here we only provide initial results.

In this work, we tackle the performance problem head-on by introducing a custom ASIC design for the Gentry-Halevi FHE. To the best of our knowledge this is the first ASIC realization of the full scheme (excluding key generation). Our hardware architecture supports the encryption, decryption and recryption primitives for the 2048-dimension instantiation of the Gentry-Halevi scheme. We utilize a number of optimizations, including reformulation of the operations, use of spectral techniques and precomputation to speed up the arithmetic operations. Another contribution of independent interest is the number theoretical transform based fast million bit multiplier, which lies at the heart of all the primitives.

II. BACKGROUND

A. Gentry’s Fully Homomorphic Scheme

While the details are very technical, a high-level description of Gentry’s scheme is as follows. The scheme is based on identifying ideals \( I \) in polynomial quotient rings \( \mathbb{Z}[x]/(f(x)) \) (with \( \deg(f) = n \)) with Euclidean lattices \( \mathbb{Z}_I \subseteq \mathbb{R}^n \) by mapping each residue polynomial \( r(x) = a_0 + \cdots + a_{n-1}x^{n-1} \) to its vector of coefficients \( (a_0, \ldots, a_{n-1}) \). Gentry calls these objects ideal lattices. Ideal lattices provide both additive and multiplicative homomorphisms modulo a public-key ideal that is represented as a lattice. Having both types of operations permits evaluation of general circuits. That is, we obtain an encryption procedure Encrypt such that \( \text{Encrypt}(x_1) + \text{Encrypt}(x_2) = \text{Encrypt}(x_1 + x_2) \) and \( \text{Encrypt}(x_1) \cdot \text{Encrypt}(x_2) = \text{Encrypt}(x_1 \cdot x_2) \). Therefore, any circuit \( C \) with efficient description can be evaluated homomorphically. However, this somewhat fully homomorphic scheme is not perfect. Due to the noisy nature of the scheme, in each iteration of a homomorphic addition or multiplication the noise term in the partial result grows. After the evaluation of only a logarithmic depth circuit, the decryption fails to recover the correct result.
To make the scheme work, Gentry uses a number of tricks. He introduces a re-encryption procedure called Recrypt that takes a noisy ciphertext and returns a noise-reduced version. In a brilliant move, Gentry manages to obtain Recrypt again from the somewhat homomorphic encryption scheme by simply homomorphically evaluating the decryption circuit using encrypted secret key bits on the noisy ciphertext. Thereby, a “refreshed” ciphertext can be obtained to reduce noise buildup. To make this work, the somewhat homomorphic scheme needs to be able to handle circuits that are deeper than its own decryption circuit before the level of noise becomes too large. Somewhat homomorphic schemes that carry this property are called bootstrappable.

B. The Gentry-Halevi FHE

The first (partial) implementation of a specialized version of Gentry’s scheme was presented by Smart and Vercauteren [16]. Smart and Vercauteren specialized Gentry’s scheme to principal-ideal lattices, and forced the determinant of the lattice to be a prime number. While this specialization improves the efficiency, it does not allow efficient key generation to support the full scheme including bootstrapping and Recrypt for practical key sizes [16]. Gentry and Halevi remove the primality restriction by introducing a special hermitian normal form for the bases. Further optimizations such as choosing sparse polynomials, batching polynomial evaluations, customized resultant and inversion algorithm for \( f(x) = x^{2^l} \pm 1 \) allowed the first software implementation of an FHE scheme.

Since in this paper we present a hardware implementation of the Gentry-Halevi FHE scheme, we find it necessary to give a high-level description of the primitives as follows. Following [16] we use \( \lceil N \rceil \) to denote the round to nearest integer operation, \( \lfloor N \rfloor_d = (N \mod d) - d \), and \( [N] = \{0, 1, \ldots, N - 1\} \).

**Key Generation.** The key generation phase is rather involved but can be summarized in the following steps:

1) Set \( f_n(x) = x^n \pm 1 \). Choose a random \( n = 2^d \)-dimensional integer lattice represented by a randomly chosen polynomial \( v(x) \) where \( v_i \) are chosen from the set of \( t \)-bit signed integers.

2) Compute \( w(x) \) such that \( w(x)v(x) = d \mod f_n(x) \) where \( d \) represents a constant integer. This task may be achieved by using the polynomial version of the Extended Euclidean Algorithm\(^1\).

3) Compute \( r = w_0/w_1 \mod d \) and check if \( w_i = w_{i+1}r \mod d \) for all \( i = 1, \ldots, n - 2 \). If the inverse of \( w_1 \) does not exist restart the key generation procedure by picking a new random \( v(x) \) polynomial.

4) In order to facilitate reencryption, randomly choose bit-vectors \( \sigma_i \) for \( i = 0, \ldots, S - 1 \) where each vector has Hamming weight one. Choose \( w' \) as any one of the odd coefficients of \( w(x) \). Randomly choose \( x_i \in \mathbb{Z}_d \) for \( i = 0, \ldots, s - 1 \) such that \( \sum_{j=0}^{s-1} \sigma_{i(j)}x_jR^i \mod d = w' \). The parameter \( R \in \mathbb{Z} \) may be chosen as a power of 2 for efficiency.

5) Let \( l = \lceil 2\sqrt{S} \rceil \). For reencryption, pick bits \( \eta_{i,j} \) for \( i \in [s], j \in [d] \) where \( \eta_{i,j} \) has Hamming weight 2 when viewed as an \( l \)-dimensional vector for fixed \( i \) values. Then encrypt each \( \eta_{i,j} \) obtaining \( \beta_{i,j} = \text{Encrypt}(\eta_{i,j}) \).

6) The public key is \( \text{PK} = (r, d, \{\eta_{i,j} : i \in [s], \beta_{i,j} : i \in [s], j \in [d]\}) \) and the secret key is \( \text{SK} = (w', \sigma_0, \sigma_1, \ldots, \sigma_{S-1}) \).

**Encryption.** To encrypt a bit \( m \in \{0, 1\} \) first choose an \( n \)-th degree sparse random polynomial \( u(x) \) where the coefficients are chosen from \( \{0, 1, -1\} \) with probability of \( \rho \) being \( \{0\} \). Using the public key parameters \( (r, d) \) we may compute the encryption by evaluating \( u(x) \) as follows: \( \text{Encrypt}(m) = \left[m + 2 \sum_{i=0}^{n-1} u_i r^i\right]_d \). Note that when multiple bits are to be encrypted, one may batch the computation yielding a significant speedup. Indeed, \( k \) simultaneous encryptions may be computed at cost only \( O(\sqrt{\rho}) \) times more than a single bit encryption using simultaneous polynomial evaluation techniques.

**Decryption.** We decrypt a ciphertext \( c \in \mathbb{Z}_d \) using the secret key \( \text{SK} = (w_i) \) simply by computing a modular multiplication as \( \text{Decrypt}(c) = [cw_i]_d \mod 2 \).

**Reencryption.** The goal of reencyption is to remove the noise buildup experienced during homomorphic circuit evaluations. We may only evaluate circuits to a constant (small) depth depending on the specific choice of parameters. To continue homomorphic evaluations we apply the reencyption procedure. Informally, reencypt works by homomorphically decrypting the ciphertext using encrypted secret key bits. A given ciphertext \( c \) is reencypted by taking the following steps:

1) Compute \( y_{j,i} = cz_iR^i \mod d \) for \( i = 0, \ldots, S - 1 \) and \( j = 0, \ldots, s - 1 \).

2) Compute \( z_{j,i} = y_{j,i}d \) as the \( p = \lceil \log_2(s + 1) \rceil \) bit approximation to the right of the binary point.

3) For \( j \in [s] \) compute the quotients \( q_j = \sum_{a \in [l]} \beta_{j,a} \left( \sum_{b \in [l]} \beta_{j,b}z_{j,i(a,b)} \right) \mod d \) where the index function is defined as \( i(a, b) = ad - (a + 1) + (b - a) \). Note that the \( \beta_{j,a}z_{j,i} \) products are realized as conditional additions in \( \mathbb{Z}_d \) (since \( z_{j,i} \) are bits in cleartext) and only the product of the result of the inner summation with \( \beta_{j,a} \) requires multiplication in \( \mathbb{Z}_d \).

4) Finally compute the reencyption of \( c \in \mathbb{Z}_d \) is achieved by homomorphically evaluating the decryption circuit on \( c \) in encrypted form. Using a number of optimization the decryption operation is expressed the following form:

\[
\text{Decrypt}_{SK}(c) = \left[ \sum_{j \in [s]} \sum_{i \in [l]} \sigma_{j(i)}z_{j,i} \right] + \sum_{j \in [s]} \sum_{i \in [l]} \sigma_{j(i)}(y_{j,i} \mod 2) \mod 2
\]

Note that the inputs \( d \) and \( y_{j,i} \) are in cleartext form while the secret key \( \sigma_{j} \) are in encrypted form during the

---

\(^1\)Note that Section 4 of [16] presents a significantly more efficient technique for computing \( w(x) \).
evaluation of the decryption circuit. The first summation is homomorphically computed on the individual bits (in encrypted form) via grade school addition of $s$ fixed point numbers expressed using $p$-bits. Therefore, in the computation of the actual recrypt operation $\sigma_j(i)$ are replaced with their recoded and encrypted form, i.e. $\beta_j,i$ and inner summation of the first term with $q_j$. During homomorphic evaluation the $(\mod 2)$ additions and multiplications turn into additions and multiplications in $\mathbb{Z}_d$, respectively. The depth of the circuit evaluating the carry output may be shown to be bounded by $O(s^2)$. Hence, we end up computing in the order of $O(s^2)$ multiplications in $\mathbb{Z}_d$ to figure out the carry bit and reflect it to the LSB in encrypted form using a $\mathbb{Z}_d$ addition operation. The second summation multiplies bits (in cleartext) by ciphertexts in $\mathbb{Z}_d$.

C. Number Theoretic Transform Based Arithmetic

The Number Theoretic Transformation (NTT) is a special form of Fourier Transform over rings. This special form, eliminates the error prone structure of Fourier Transform because of the floating point arithmetic. We use NTT as the backbone of the million-bit arithmetic operations. Especially, it is effective in large integer multiplication (in million-bit range) which lies at the heart of all the primitives. Common multiplication schemes (Karatsuba Algorithm, Classical school-book multiplication method) become infeasible for large integer multiplications. Schönhage-Strassen algorithm is currently asymptotically the fastest algorithm for very-large large integer multiplications. Schönhage-Strassen algorithm is not included in timing calculations.

Schönhage Strassen Algorithm. The Schönhage-Strassen Algorithm is a NTT-based large integer multiplication algorithm, with a runtime of $O(N \log N \log \log N)$ [20]. For a $N$-digit number, NTT is computed using the ring $R_N = \mathbb{Z}/(2^N + 1)\mathbb{Z}$, where $N$ is a power of 2. In [25], the algorithm is explained as follows:

In the algorithm, for the NTT evaluation we sample the numbers $A$ and $B$ that fits into $N$-digits with a sampling size $\epsilon$. The selected $p$ is a prime number with a primitive root $w$, i.e. $w^p = 1 \pmod{p}$. Then, we can represent the NTT forms of the numbers as $A_k = \sum_{k=0}^{N-1} w^k a_k$ and $B_k = \sum_{k=0}^{N-1} w^k b_k$. Later, the components are multiplied to form $c_k = A_k \cdot B_k \pmod{p}$. Using the inverse-NTT we compute $C_k = \sum_{k=0}^{N-1} w^{-k} c_k$. In the last step, we accumulate the carry additions to finalize the evaluation of $C$. To realize the Schönhage-Strassen Algorithm efficiently, it is crucial to employ fast NTT and inverse-NTT computation techniques. We adopted the most common method for computing Fast Fourier Transforms (FFTs), i.e. the Cooley-Tukey FFT Algorithm [18]. The algorithm computes the Fourier Transform of a sequence $X$ as $X_k = \sum_{j=0}^{N-1} x_j e^{-2\pi i k j / N}$, by turning the length $N$ transform computation into two $N/2$ size Fourier Transform computations as follows

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} \theta^m + e^{2\pi i k} \sum_{m=0}^{N/2-1} x_{2m+1} \theta^m,$$

where $\theta = e^{-2\pi i k / N}$. We change $e^{2\pi i k}$ with powers of $w$ and perform the divisions into two halves with odd and even indices, recursively. With the use of fast transform technique, we can evaluate the Schönhage-Strassen Multiplication Algorithm in $O(N \log N \log \log N)$ time.

Block-wise Arithmetic. In the NTT form, we can compute multiple addition, subtractions and multiplications, e.g. $\sum_j NTT(X_j)[i]$ which $i$ is the digit index and $j$ is the index of the integer. The arithmetic operation is performed by performing the additions digit-wise and can be done in parallel. We can group the digits of the integers in $l$ digits, and complete the additions for the first group (first $l$ digits). Later, we select the next group to compute the second result. Doing this iteratively, we compute the overall arithmetic in groups. This type of computation simplifies our implementation by selection of small $l$. We refer it as block-wise (or block) computation in further sections.

III. OVERVIEW OF OUR ARCHITECTURE

A high level depiction of the overall architecture is presented in Figure 1. The architecture contains five sub-architecture blocks called Large Integer Multiplier, Barrett Reduction Unit, Decryption Completion Unit, Encryption Unit and Recryption Unit. These are controlled by the Master Control Unit (MCU). Each of the Encryption, Decryption and Recryption primitives require large integer multiplications. However, providing a dedicated Large Integer Multiplier Unit for each primitive is too costly. Therefore, in our design we incorporate one Large Integer Multiplier that will be shared between these primitives.

To realize each primitive, the MCU controls the units to complete its operation, handling the order of operations and I/O between the units and the external memory. Since the operands are in the range of millions of bits, data transactions between units are impractical. Therefore, we suppose an external memory unit (RAM) in the design for storage, but it is not included in the evaluation of the area. We utilize a 64-bit bus for I/O transactions between the units and the external RAM. The RAM holds the public keys and acts as a shared memory between the units when the primitive operations are realized in sequence. The operation time for a unit is then considered as total time spent from RAM to the computation unit, the time spent for arithmetic operations and the time needed to write the result back to RAM. With effective addressing and utilizing prefetching from RAM, the initial address decoding overhead can be eliminated, therefore is not included in timing calculations.

A. Parameter Selection

Large Integer Arithmetic Parameters. The parameters for the NTT based implementation is based on [25], [1]. We choose 64-bit word size for fast and efficient computations, sampling size as $\epsilon = 2^{34}$ and the modulus as $p = 2^{64} - 2^{32} + 1$. We chose $p$ from a special family of prime numbers which are called Solinas Primes [2]. This allowed us to realize a modular reduction using only a primitive arithmetic operations. A
128-bit number is denoted as \( z = 2^{26}a + 2^{64}b + 2^{32}c + d \).
Using the selected \( p \), we perform \( z \pmod{p} \) operation as \( 2^{32}(b + c) - a - b + d \).
The large integer size parameter \( N \) is chosen to satisfy \( \frac{N}{2}((\epsilon - 1)^2 < p \) to prevent the digit-wise overflows in the intermediary computations in NTT. Also, \( N \) should be big enough to cover million-bit multiplication with smallest possible value, so that the computation time is small.
The best candidate for \( N \) is determined as \( 3 \cdot 2^{15} \). This parameter achieves one million-bit multiplication with minimum memory overhead. Finally, we determine the \( N^{th} \) primitive root of unity, i.e. \( w \), from the equation \( w^N \equiv 1 \pmod{p} \), we determine \( w = 3511764839390700819 \).

In Cooley-Tukey FFT Algorithm, each recursive halving operation is referred as a stage and is denoted as \( S_i \), which \( i \) is the stage index. The size of the smallest NTT block is selected 12 digits and it is referred as the 0th stage, i.e. \( S_0 \). The remaining stages are reconstruction stages and require different arithmetic operations from the ones in \( S_0 \). As the nature of the NTT algorithm, in each reconstruction stage process the block size is doubled. Therefore, it takes 13 reconstruction stages to complete the NTT operation. In terms of NTT operations, every stage and operation is identical to NTT. Only difference is selection of \( w^i \). It is computed as: \( w^i = w^{-i} \pmod{p} \).

In Barrett Reduction operation, it uses \( \mu \) and \( m \) values for multiplicative purposes, so we store them in NTT form. These values are used in a straight digit-wise multiplication, so we do not have any overheads resulting from an algorithm. Therefore, we select \( \epsilon \) same with large integer multiplication as 24-bits.

**FHE Primitive Parameters.** In terms of Encryption, Decryption and Recryption operation parameters, we use the smallest parameters in [16]. The Table I shows the selected parameters.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( l )</th>
<th>( s )</th>
<th>( p )</th>
<th>( S )</th>
<th>( p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>46</td>
<td>15</td>
<td>5</td>
<td>512</td>
<td>16/2048</td>
</tr>
</tbody>
</table>

**TABLE I**

**SELECTED PARAMETERS**

The other parameter selection is selection of the sampling size, i.e. \( c \), for different FHE Primitive operations. We selected \( \epsilon \) values as following:

- **Encryption:** In Encryption operation, we hold powers of \( r \) in NTT form and perform additive an multiplicative operations. Using the Encryption equation, we set the sampling equation as: \( \frac{n}{2} \sum_{i=0}^{n-1} (\epsilon - 1)^2 < p \). In this equation, the sum of \( n \) numbers is taken into account for the worst–case scenario. Although additions and subtractions have the same probability to occur and they will cancel out the bit size growth in a best–case scenario, the worst case scenario should be taken into account to
prevent overflows for any cases. Using the parameters above, we select \( \epsilon \) as 16-bits.

- **Recryption:** The Recryption operation has a similar equation to Encryption operation. Its algorithm also consist of additive and multiplicative operations. Using the Recryption equation, we set the sampling equation as: 
  \[ \frac{1}{2} \sum_a (\epsilon - 1) \sum b_i (\epsilon - 1) < p \]  
  In terms of values \( a \) and \( b_i \), we have \( S \) inner additions and each digit is multiplied by another. The sampling equation can be rewritten as: 
  \[ \frac{1}{2} \sum S (\epsilon - 1)^2 < p \]  
  Similar to Encryption, we select the sampling size as 16-bit.

IV. LARGE INTEGER ARCHITECTURE

The operations for the described fully homomorphic scheme are based on efficient large integer arithmetic, which necessitates the implementation of an optimized modular multiplication architecture. In the following, design details of the large integer multiplier and modular reduction architecture is given.

A. Large Integer Multiplier

![Fig. 2. Overview of The Large Integer Multiplier](image)

**Architecture Overview.** Our architecture is composed of a data cache, a multiplier control unit, two routing units and a function unit, which is illustrated in Figure 2. The architecture is designed to perform a restricted set of special functions. There are four functions for handling the input/output transactions and five functions for arithmetic operations:

- **Sequential Load:** It is used to store a million-bit number to the cache.
- **Sequential Unload:** The cache releases its contents starting from the least significant to most significant.
- **Butterfly Load:** In NTT an important step is the distribution of the digits into the right indices using the butterfly operation.
- **Scale & Unload:** In the last step of the NTT computation the digits need to be scaled by \( N^{-1} \mod p \) and accumulate the carries. The Scale Unit overlaps scale and carry accumulation of the digits with output, since it is the final step in million-bit multiplication.
- **12x12 NTT/INTT:** The smallest NTT/INTT computation is for 12 digits. 12x12 NTT/INTT Unit takes the digits sequentially and computes the 12 digit NTT/INTT by using simple shifts and addition operations.
- **Stage-Reconstruction:** This function is used for reconstruction of the stages. According to the given input, it reconstructs an NTT or INTT stages. Reconstructions only complete a stage at a time, with the given stage index input. In order to complete a full reconstruction, i.e. completing evaluation of NTT, it is recalled for all 13 stages.
- **Inner-Multiplication:** This operation is used to compute the digit-wise modular multiplications. For this we utilize the multipliers used in **Stage-Reconstruction Units**.

Using the functions outlined above, we can compute the product of million-bit numbers \( A \) and \( B \) using the following sequence of operations:

1) \( A \) is loaded into cache by using **Butterfly Load**.
2) The NTT of number \( A \), i.e. NTT(\( A \)), is computed by calling; first 12x12 NTT function, and afterwards **Stage-Reconstruction** function for all stages.
3) NTT(\( A \)) is stored to RAM using **Sequential Unload**.
4) Using the same sequence of functions as above, we also compute NTT(\( B \)).
5) The cache can only hold the half of the digits of NTT(\( A \)) and NTT(\( B \)). Therefore, the numbers are divided into lower and upper halves: 
   \[ \text{NTT}(A) = \{\text{NTT}(A)_h, \text{NTT}(A)_l\} \]  
   \[ \text{NTT}(B) = \{\text{NTT}(B)_h, \text{NTT}(B)_l\} \]  
6) We use **Sequential Load** function to store NTT(\( A \))\(_h\) and NTT(\( B \))\(_h\).
7) Later on, an **Inner Multiplication** function is used to calculate the digit-wise modular multiplications:
   \[ C_h[i] = \text{NTT}(A)_h[i] \times \text{NTT}(B)_h[i] \]
8) The result is stored to the RAM by **Sequential Unload**.
9) We repeat above three steps to compute the lower part:
   \[ C_l[i] = \text{NTT}(A)_l[i] \times \text{NTT}(B)_l[i] \]
10) The result digits, i.e. \( C[i] \), are loaded into the cache by **Sequential Load**. At this point the cache will contain the multiplication result, but still in the NTT form.
11) The result is converted to integer form by using, 12x12 INTT function which is followed by a complete **Stage-Reconstruction** functions \( C' = \text{INTT}(C[i]) \).
12) In the last step, the result is scaled and the carries are accumulated by **Scale & Unload** function to finalize computation of \( C \): 
   \[ C[i + 1] = C'[i + 1] + [C'[i]/p] \mod R \]

**Multiplier Cache System.** The size of the cache is important for the timing of multiplications. In each **Stage-Reconstruction** process of the NTT algorithm, we need to match the indices of odd and even digits. The index difference of the odd and even digits in a reconstruction stage is computed as: 
   \[ S_{\text{off}} = 12 \cdot 2^{i-1} \]  
   where \( i \) is the index of reconstruction stages, i.e. \( 1 \leq i \leq 14 \). Since, in later stages we require digits from distant indices, an adequate sized cache is chosen to reduce the number of input/output transactions between the cache and RAM.

Let's call \( N' \) as the chosen cache size. Then, we can divide the \( N \) digits into \( 2^t = N/N' \) blocks, i.e. \( N = \{N_{2^{t-1}}, N_{2^{t-2}}, \ldots, N_0\} \). Once a block is given as input, we
can compute the reconstruction stages until \( N' < S_i, \text{diff} \) for the \( i^{th} \) stage. Then, starting from the \( i^{th} \) stage, \( N_j \) requires digits from \( N_j+1 \) which \( j \) is block index. So, we need to divide \( N_j \) and \( N_j+1 \) into halves and match the upper halves of \( N_j \) with \( N_j+1 \), and lower halves of \( N_j \) with \( N_j+1 \). This matching process adds \( 2N' \) clock cycles for each block. Then, the total input/output overhead is evaluated as \( 2N \cdot \log_2 \left( \frac{N}{N'} \right) \), where \( \log_2 \left( \frac{N}{N'} \right) \) is the number of the stages that requires digit matching from different blocks. In our implementation, we aim to optimize the speed by selecting \( N' \) as \( N \).

Although a huge sized cache is important for our design, a straight cache implementation is not sufficient to support parallelism. The main arithmetic functions utilized in the multiplication process, such as \( 12 \times 12 \) NTT/INTT, \( ^2 \), STAGE-RECONSTRUCTION and INNER MULTIPLICATION, are highly suitable for parallelization. In order to achieve parallelization, the cache should be able to sustain required bandwidth for multiple units. In order to sustain the bandwidth, we build up the cache by combining small, equal size caches or as we refer them sub-caches. Combining these sub-caches on a top level, we can select the cache to be used as a single-cache or a multicache system. In case of linear functions, such as \( \text{S}_0 \), we can select the cache to be used as a single-cache or a multi-cache system. In our implementation, we aim to optimize the speed by selecting \( N' \) as \( N \).

### Routing Unit

The Routing Units play an important role to match the \( \text{odd} \) and \( \text{even} \) digits to the arithmetic units. As stated previously, the index difference of the digits is \( (12 \cdot 2^{j-1}) \). Therefore, in last \( \log 2m \) reconstruction stage, \( \text{odd} \) and \( \text{even} \) digits fall into different sub-cache. The assignment of sub-caches to proper arithmetic units\(^3\) for each STAGE-RECONSTRUCTION is shown in Table II. In the Table, arithmetic units are referred as \( \text{arith}_i \), which \( i \) is the index number.

### Function Unit

The Function Unit can be divided into three parts, i.e. the SCALER Unit, the \( 12\times12 \) NTT/INTT Unit and multiple STAGE-RECONSTRUCTION Unit.

**SCALER Unit**: As mentioned earlier after the \( 12\times12 \) INTT operation, as a final step we need to perform a modular multiplication between the digits and inverse of \( N \). Denoting the digits as \( d_i \) and including the carries as \( c_i \), final operation can be shown as \( d_i \times N^{-1} + c_i \pmod{p} \). For our parameter choice \( N^{-1} \pmod{p} = 0x123456789abcdef \) a constant number with a special form that can be computed by simple shift and add operations with a low number or arithmetic operations.

**12 \times 12 \) NTT/INTT Unit**: This unit can be used for either computing the first stage of NTT or for INTT according to the required operation. This first NTT/INTT stage is basically computed using the NTT formula: \( X_j = \sum_{i=0}^{n} \left( w^i \right)^{j} \times d_i \pmod{p} \). In the equation, \( j \) is the index of the digit to be computed, and \( d_i \) refers to the the digit value with index \( i \). The parameter \( w^j = w^{2^j} \pmod{p} \), since the coefficient is squared in each stage. The parameter \( n \) is the smallest NTT block value to be evaluated and in our architecture it is chosen to be 12.

The overhead of the first stage can be decreased significantly by exploiting the special structure of the constant coefficients. Indeed, we can realize the modular multiplications with simple shift and add operations following by a modular reduction at the end. Furthermore, note that in INTT \( w^i \) = 0x10000 and in NTT \( w^i \) = 0x123456789abcdef and therefore only a few additions and shifts are needed. These simple operations can be squeezed into few clocks and pipelined to optimize throughput. Also coefficients repeat after a while, since \( (w^j)^{12} = w^{2^{(j+6)}} = 1 \pmod{p} \). This eases the construction of the NTT/INTT units, since we can reuse the same circuit in the process. Due to pipelining all operations can be performed in \( N \) clock cycles.

**STAGE RECONSTRUCTION Unit**: This unit holds 64 bit multipliers and is responsible for the operation of two functions STAGE-RECONSTRUCTION and INNER MULTIPLICATION. The architecture is more complex than the other units in the FUNCTION Unit. The architecture is illustrated in Figure 3. The pipelined INNER MULTIPLICATION function computes a 64-bit multiplication, additions and a modular reduction. The unit can output a modular multiplication product in every two clock cycles after the initial startup cost of the pipeline. The whole function takes \( \frac{3N}{2} \) multiplications and with \( m \) multipliers it will cost \( \frac{3N}{2m} \) clock cycles.

**Fig. 3. Stage Reconstruction Unit**

In the STAGE-RECONSTRUCTION function, a more complex control mechanism is required compared to other functions. In each stage we need to compute

\[
\begin{align*}
O_{i,j} &= E_{i-1,j} - O_{i-1,j} \times w_{i-1}^{j \pmod{n_{i-1}}} \pmod{p}, \\
E_{i,j} &= E_{i-1,j} + O_{i-1,j} \times w_{i-1}^{j \pmod{n_{i-1}}} \pmod{p}.
\end{align*}
\]

(1)

In the equation, \( i \) denotes the stage index from 1 to 13, \( j \) denotes the index of the digits, \( w_{i} \) is the coefficient of stage \( i-1 \) and finally \( n_{i-1} \) is the modular reduction to select the appropriate power of the \( w_{i} \). We may compute the terms \( n_{i} \) iteratively as \( n_{i+1} = 2 \times n_{i} \) where \( n_{0} = 12 \). Ideally we need

<table>
<thead>
<tr>
<th>( S_{1} )</th>
<th>( S_{2} )</th>
<th>( S_{3} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>arith(_0)</td>
<td>( \text{arith}(_1) )</td>
<td>( \text{arith}(_2) )</td>
</tr>
<tr>
<td>( S_{1} )</td>
<td>( S_{2} )</td>
<td>( S_{3} )</td>
</tr>
<tr>
<td>8c0-8c1</td>
<td>8c2-8c3</td>
<td>8c4-8c5</td>
</tr>
</tbody>
</table>

**Table II**: Assignment Table
to store all the coefficients along with the \textit{odd} digits. However, this will require another large cache of size \((12+24+48+\cdots+49152) \times N\) digits. The cache can be reduced to almost half in size, since during the stage computations each stage uses all of the coefficients from the previous stage. Therefore, the size can be reduced to \(12+(24+48+\cdots+49152)/2 = 49152 \times N\).

In order to reduce the storage requirement, we can compute the coefficients efficiently by using pre-stored coefficients as follows:

1) The coefficients required in two consecutive stages are as follows: \(S_{i+1} = w_1^0, w_1^1, \ldots, w_1^{n_i}\) and \(S_{i+2} = w_1^{n_i+1}, w_1^{n_i+2}, \ldots, w_1^{2n_i}\).

2) Then \(S_{i+2} = w_1^{n_i+1}, w_1^{n_i+2}, \ldots, w_1^{2n_i}\), since it holds that \(n_{i+1} = 2 \times n_i\).

3) Further, \(S_{i+2} = w_1^0, w_1^1, \ldots, w_1^{2n_i}\), since \(w_1 = w_1^2\).

4) This shows that half of the coefficients of \(S_{i+2}\) are same as \(S_{i+1}\) and the other half are the square roots of the coefficients of \(S_{i+1}\).

5) We can compute the square roots by multiplying each \(w_i^j\) with \(w_i^{j+1}\).

Using the properties described above, we construct the \textit{Coefficient Table} by storing two columns of coefficients. In the first column, since our smallest computation block is 12, we compute and store all \(w_0^j\) coefficients for \(11 \geq j \geq 0\). We denote these coefficients as \(w_{\text{first},i}\), where \(i\) denotes the index of the coefficient. In the second column, for each of the remaining stages we compute and store \(w_i^1\). The second column coefficients are denoted by \(w_{\text{second},i}\). This makes a total of 24 coefficients which we can use to compute any of the \(w_i^j\) values. When we include also the coefficients for the INTT operations, our table contains 48 coefficients. The computation of an arbitrary coefficient using the table can be achieved as

\[
w_i^j = w_{\text{first},i} \times \prod_{t=0}^{j} w_{\text{second},t}^i.
\]

The values of \(l\) and \(e\) are functions of \(i\) and \(j\). Also \(e\) is a value equal to 1 or 0. Therefore we can omit the multiplications whenever \(e = 0\). The total number of multiplications for computing \(w_i^j \times O_i\) can be calculated as follows:

1) In every reconstruction stage we start by multiplying \textit{odd} digits with \(w_{\text{first},i}\)’s. This step makes a total of \(\frac{N}{2}\) multiplications.

2) Apart from the first reconstruction stage, in each stage we also require coefficients from \(w_{\text{second},0}\) to \(w_{\text{second},i-1}\). Since we cannot store the coefficients, in each stage we need to rebuild the previous stage coefficients to build up the coefficients. We are using half of the previous stage values so in each stage we need \(\frac{N}{4}\) additional multiplications.

3) The total multiplications will then becomes \(\sum_{t=0}^{i=12} \left( \frac{N}{2} + i \times \frac{N}{4} \right) = 26 \times N\).

\textbf{Multiplier Control Unit.} The \textit{Multiplier Control Unit} contains a state machine to handle the functions that are given as inputs. It is mainly responsible to start the requested function given as input to the system. Since \(12 \times 12\) INTT/INTT

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
 & NTT(A) & NTT(B) \\
\hline
2 BUTTERFLY LOAD & 2 & 2 \\
2 STAGE-RECON & 12 & 12 \\
2 SEQUENTIAL LOAD & 2 & 2 \\
2 SEQUENTIAL UNLOAD & 2 & 2 \\
\hline
AxB & 2 & 2 \\
2 SEQUENTIAL LOAD & 2 & 2 \\
2 SEQUENTIAL UNLOAD & 2 & 2 \\
\hline
INTT(AxB) & BUTTERFLY LOAD & 2 & 2 \\
12 \times 12 INTT & 12 & 12 \\
STAGE-RECON & 12 & 12 \\
SCALE UNLOAD & 12 & 12 \\
\hline
TOTAL & 52 & 52 \\
\hline
\end{tabular}
\caption{Clock Cycle Counts of Functional Blocks}
\end{table}

\textbf{B. Modular Reduction}

A million-bit multiplication operation produces a two million-bit result, which is required to be reduced back to million bits. A cost-efficient reduction algorithm is required in order to compute \(r = x \pmod{m}\). We used the Barrett Modular Reduction Algorithm [4].

Since precomputed \(\mu\) value and \(m\) constant terms, we can avoid two INTT conversions by storing them in their INTT form. Also, by selecting a special base as \(2^{64}\), we can perform the division operations by changing the reading address of \(x\). The operation \(\left\lfloor \frac{x}{m} \right\rfloor\) is done by reading the digits after the \(b^{k-1}\)th bit and for \(\left\lfloor \frac{x}{m} \right\rfloor\) reading after the \(b^{k+1}\)th bit. Once a Barrett Reduction is requested, the state machine controls the million-bit multiplier and perform \(q_3 \cdot m\) and \(q_1 \cdot \mu\).

After the multiplications, terms \(r_1\) and \(r_2\) is loaded for subtraction that is evaluated in digits by a simple subtractor with word length 64-bits. Since, reading both \(r_1\) and \(r_2\) uses huge portion of the bandwidth, a \(k\)-digit local cache is added to store partial results to prevent the I/O collisions from/to
the RAM. The local cache is based on two parallel $\kappa/2$-digit FIFOs so that we can output 2 digits per clock. In case of an addition for the if case, it will convert the sign bits into zero, which will be equal to omitting after $b_{k+1}$. In the while loop, the inputs are used as $r_1 \rightarrow r$ and $r_2 \rightarrow m$. Starting from most significant digits, $r$ and $m$ are loaded into comparator unit to decide $r \geq m$. In case of $r \geq m$, we perform subtraction using the same subtracter. The decision $r \geq m$ is done by implementing a 512-bit (8-digit) comparator. If the first 8-digits are equal then it loads the next 8-digits for decision until they are not equal.

The time for a Barrett Reduction is heavily depends on the multiplications and the subtractions have a small effect. As stated previously by storing $\mu$ and $m$ in NTT form, we can complete the multiplications totally in $2 \cdot 36.5N$ clock cycles. In case of subtractions for 3 million-bit I/O transactions\textsuperscript{4} will require $\approx 23500$ clock cycles. Since in a typical Barrett Reduction subtraction iterations are between 2 and 3, we neglect the overhead caused by the subtractions. The total time required for a Barrett reduction is $\approx 73N$.

V. FHE PRIMITIVES

A. Decryption

The decryption operation is a rather simple operation that requires a modular multiplication operation followed by a modulo 2 reduction, i.e., $\text{Decrypt}(c) = [cw_i]_d \pmod{2}$. During the decryption operation, the MASTER CONTROL UNIT uses the LARGE INTEGER MULTIPLIER and the BARRETT REDUCTION units to realize $[cw_i]_d$. This is followed by the application of the DECRYPTION COMPLETION UNIT, which contains a simple arithmetic circuit that takes the lowest significant digit of the modular multiplication result and pads it with zeroes to match the operand length, i.e., million-bit size for the chosen parameter sizes.

We can reduce the large integer multiplication time by storing the $w_i$ in NTT form. The conversion operation is only applied to the ciphertext $c$. Therefore the multiplication takes in the order of $36.5N$ clock cycles. The modulo 2 reduction realized by reading the last digit and by forming the large integer result with padding takes less than 8,000 clock cycles. Since $8,000 \ll 36.5N$ we neglect this quantity. Including the Barrett Reduction operation, the overall decryption operation takes $109.5N$ clock cycles.

B. Encryption

\begin{equation}
\text{Encrypt}(m) = \text{INTT} \left[ M + 2 \sum_{j=0}^{\frac{\kappa}{2}-1} R^j \left( \sum_{i=0}^{k-1} u_{(j+k+i)} R^i \right) \right]_d
\end{equation}

where we use uppercase symbols to denote the NTT form of the variables, e.g., $R = \text{NTT}(r)$ and $R^2 = \text{NTT}(r^2)$. Since the message $m$ is a bit, we simplify $M = (0, 0, \ldots, 0)$ if $m = 0$, else $M = (1, 1, \ldots, 1)$ if $m = 1$. The equation eliminates NTT convolutions in multiplications and requires only one inverse-NTT and a single modular reduction at the end. Since conversions take 90% of the time in multiplication, by moving virtually all computations into the NTT domain we gain a significant speedup.

\text{Algorithm A’} proposed in [12] is suitable for efficient polynomial evaluation in hardware.
The NTT based arithmetic operations in aggregate, referred to as NTT-Encryption, are evaluated with what we call the Encryption Unit. The remainder of the operations are completed by utilizing the Large Integer Multiplier Unit and the Barrett Reduction Unit. To realize the encryption primitive, the Master Control Unit runs the Encryption Unit, the Large Integer Multiplier and the Barrett Reduction units in order.

**Encryption Unit.** The Encryption Unit is designed as a semi-systolic architecture as illustrated in Figure 5. The architecture contains a Control Unit, a storage unit for u and Encryption Processing Elements (EPEs). Since NTT based arithmetic can be rather efficiently parallelized, the RAM access latency becomes the bottleneck in the design. We can achieve the maximum throughput by incorporating #EPEs=bandwidth/frequency processing elements in the design.

**Encryption Processing Element (EPE).** The EPE as shown in Figure 6 is designed to evaluate NTT-Encryption of a block size κ. The parameter κ also represents the size of the local cache. The local cache acts as a temporary variable t and it is used to reduce the number of I/O transactions. It is also important to note that the unit is fully pipelined with 10 clock delay. Therefore each block operation will have an extra 10 clock cycle delay in time evaluation, i.e. we multiply the total delay. Therefore each block operation will have an extra 10 is used to reduce the number of I/O transactions. It is also important to note that the unit is fully pipelined with 10 clock delay. Therefore each block operation will have an extra 10 clock cycle delay in time evaluation, i.e. we multiply the total delay.

The unit evaluates encryption in the following two steps:

- The first step evaluates the window summations and the scaling operation is shown in the Algorithm 2. With built-in local cache, a window summation can be evaluated in at most k input and 1 output transactions. If a U value is zero, then R(k) is not loaded into the system. Therefore, the probability of the coefficients of the u polynomial being 0 directly gives us the cost of the operations. The total number of I/O transactions is κ·(1−κ)+2, where 1−κ is the probability of non-zero terms and plus 2 is for input of the scaler and output terms.

**Algorithm 2:** Window Sum & Scale Operation

```
Input: r = \{\{P(0), P(1),\ldots, P(k-1)\},
\{R(0), R(1),\ldots, R(n-k)\}\}, u = \{u_0, u_1,\ldots, u_{n-1}\}
Output: Inner Polynomial Block
P^{(j)} = \sum_{i=0}^{k-1} u_i R^{(j)}
```

```
1 for j = 0 to \(\frac{n}{k} - 1\) do
2 t ← 0
3 for i = 0 to k - 1 do
4 if u_i ≠ 0 then t ← t + u_i R^{(j)}
5 t ← t · R^{(j)}
6 P^{(j)} ← t
```

- The second step computes the window summations along with the doubling and addition of the message bit to finalize the NTT-Encryption. The algorithm is shown in Algorithm 3. The total number of I/O transactions for the window summations also depends on the probability ρ. If ρ is large enough the probability of \(P^{(j)} = 0\), i.e. \(\rho^k\), will be sufficiently large that they can be ignored during the additions. Then, the total I/O transaction number is \(\frac{n}{k}(1-\rho^k)+1\), where \(\frac{n}{k}\) represents the number of windows and plus 1 is for the output term.

```
Algorithm 3: NTT Encryption

Input: P = \{P^{(0)}, P^{(1)}, \ldots, P^{(n-1)}\}
Output: R_l = 2 \sum_{i=0}^{n-1} P^{(i)} + M_l
1 t ← 0;
2 for j = 0 to \(\frac{n}{k} - 2\) do
3 if P(j) ≠ 0 then t ← t + P(j) + P^{(j)}
4 if P(j) ≠ 0 then t ← t + P^{(n-1)} + P^{(n-1)} + M_l;
5 R_l ← t
```

The EPE is controlled by signals double, operation, add_bit, read, write and clear. Each EPE is connected with a 64-bit bus, which is utilized to load the powers of r into the system. During the computation of the first algorithm, double and add_bit signals are inactive and the input is directly fed to the Modular Arithmetic Unit. The unit consist of a 64-bit modular subtracter, an adder and a multiplier. The operation signal enables the required modular arithmetic operation. In case of \(u_i = ±1\) modular adder/subtractor is used to compute \(t = t ± R^k\). For the scaling operation, modular multiplier is enabled by the operation signal to compute \(t = t · R^{j-k}\). The second algorithm is realized using two 64-bit modular adders that are controlled by the double and add_bit signals. If the double signal is active, the input is added to itself to double the window summations: \(2P = P + P\). If the add_bit signal is active, the message bit \(m\) is added to the summation in NTT form. Using these two signals, the final equation is evaluated as \(2u(R) + M = 2 \sum_{i=0}^{n-1} P^{(i)} + M\). Additionally, it is important to note that the clear signal is used in case of setting \(t = 0\) and read/write signals are used to read and update \(t\) values.

**Control Unit.** The Control Unit is a state machine that manages the encryption operation. The inputs are the message bit \(m\), random polynomial \(u\) and its outputs are the operation, double, add_bit and clear signals. Once the \(u\) polynomial is loaded into the storage, the Control Unit performs an encryption operation as follows:

1) Take message bit \(m\) as input.
2) Request for the \( u \) polynomial for the first window, \( \{u_0, u_1, \ldots, u_{k-1}\} \).
3) Using clear signal, reset the cache units at \( t \leftarrow 0 \).
4) Check the values of \( u_i \) iteratively and skip if it is zero. In case of \( \pm 1 \), \( \beta \) blocks of the powers \( r_i^k \) are loaded into the bus and operation signal is selected. Each arithmetic core is assigned with different blocks to evaluate \( t = t + r_i^k \).
5) Iterate index \( i \). Computation of the window sum is completed. To scale the sum, the necessary \( \beta \) blocks (powers of \( r_i^k \)) are loaded into the cache and the operation signal is set to enable multiplication. The term \( t \) is updated as: \( t = t \cdot R_i^k \). Now \( t \) holds the result of a scaled window summation: \( P_{l}(j) = R_i^k \sum_{0}^{k-1} u_i R_i^j \). Since there are \( \beta \) blocks, the window sum is evaluated for the \( \beta \) blocks as \( P_{sub}^{(j)} = \{P_{\beta -1}^{(j)}, \ldots, P_1^{(j)}, P_0^{(j)}\} \).
6) Sequentially write the results \( P_{sub}^{(j)} \) back to the main memory.
7) Using the steps 3–5, process each block to finish the computation of a window: \( P^{(j)} = \{P_{bs -1}^{(j)}, \ldots, P_1^{(j)}, P_0^{(j)}\} \), where \( bs \) is the block size.
8) Repeating steps 3–7, Compute all of the windows: \( \{P^{(0)}, P^{(1)}, \ldots, P^{(s-1)}\} \).
9) Using the clear signal, clear all caches to 0.
10) Assert the double signal starting from \( j = 0 \), \( \beta \) blocks of \( P^{(j)} \) is loaded if \( P^{(j)} \neq 0 \). This evaluates \( t = t + 2 \cdot P^{(j)} \) up to \( j = \frac{n}{\beta} - 1 \). The add bit signal is activated for the case \( j = \frac{n}{\beta} - 1 \). This will add the message bit \( m: t = t + 2 \cdot P^{\frac{n}{\beta} - 1} \).
11) Every arithmetic core unit writes the result sequentially to the main memory.
12) Using Steps 9–12 process each block to finish the computation of the equation \( R = 2 \cdot \sum_{j=0}^{\frac{n}{\beta} - 1} P^{(j)} + m \).

Parameters selection makes a significant difference for the time efficiency of the architecture. Since an \( r^t \) term is included if \( u_i \) is not 0, the probability distribution of \( u_i = \{0, 1, -1\} \) is important to evaluate the timings. We select the window size as 64, and let us remind that the probability is selected \( \rho = 16/2048 \). Since we only have 16 non-zero values, we need to evaluate 16 of 32 windows in the worst case scenario. If only 16 of these 16 windows are evaluated which will cost \( 16 \cdot 3N \) cycles. Addition of these 16 windows will take \( 17N \) clock cycles. Including the number of EPE’s, inverse-NTT and Barrett Reduction operations, total cost of the operations will amount to \( \frac{66N^2}{\beta} \cdot (1 + 10/\kappa) + 89N \) cycles.

C. Recryption

The Recryption evaluation is computed as

\[
\text{Decrypt}_{SK}(c) = \left[ \sum_{j \in \mathbb{S}} \sum_{i \in \mathbb{S}} \sigma_j(i) z_{j,i} \right] \pmod{2} + \sum_{j \in \mathbb{S}, i \in \mathbb{S}} \sigma_j(i) (y_{j,i} \pmod{2}) \pmod{2}.
\]

5We divide the degree 2048 into 64 windows of equal degree polynomials.

The first summation can also be rewritten in the following form.

\[
q_j = \sum_{a \in \mathbb{S}} \beta_j,a \left( \sum_{b \in \mathbb{S}} \beta_j,b z_{j,i(a,b)} \right) \pmod{d}
\]

We can take advantage from the fact that the public keys \( \beta_j,l \) are known ahead of time after KEYGEN. By precomputing and storing the keys in NTT form, i.e. \( B = \text{NTT}(\beta) \), we can eliminate many costly large integer multiplications. The equation is rewritten as

\[
q_j = \text{INTT} \left( \sum_{a \in \mathbb{S}} B_{(j,a)} \sum_{b \in \mathbb{S}} B_{(j,b)} z_{j,i(a,b)} \right) \pmod{d}
\]

The new equation eliminates most of the NTT and inverse-NTT conversions. Only one inverse conversion and a single modular reduction is required at the end of the computation. Furthermore, we benefit by computing the \( z_{j,i} \) terms first and storing them in a table. This allows us to compute the NTT based arithmetic parts in blocks, since we are able to re-read the \( z_{j,i} \) for each block. Otherwise, we would have needed to add \( \hat{N} \)-sized cache increasing the area overhead significantly. Using these tricks, we can divide the above equation into four parts. The first two parts are computed with a unit which we call the RECRIPTION UNIT. The rest of the operations are computed by the MASTER CONTROL UNIT using the LARGE INTEGER MULTIPLIER and the BARRETT REDUCTION UNIT.

The four steps of the Recryption operation are as follows:

- Evaluation of the precision bits \( z_{j,i} \).
- Sum of Public Keys \( S_j = \sum_{a \in \mathbb{S}} B_{(j,a)} \left( \sum_{b \in \mathbb{S}} B_{(j,b)} z_{j,i(a,b)} \right) \).
- Inverse-NTT conversion and Barrett Reduction of \( S_j \).
- Grade-School Addition.

1) Evaluating Precision Bits: The precision bits \( z_{j,i} \) are \( p \)-bit result of the quotient of \( y_{j,i}/d \). We divide the computation of \( z_{j,i} \) terms into two units. First the BINARY COMPUTATION UNIT evaluates the precision bits \( z_{j,i} = \{b_{j,i}^{(0)}, \ldots, b_{j,i}^{(p-1)}\} \). In the equation, \( j \) is the public key index, \( i \) is the hamming weight index and \( p = \lceil \log_2(s + 1) \rceil + 1 \) is the number of precision bits. The second unit used in the computation is the MODULAR COMPUTATION UNIT which evaluates \( y_{j,i} = c \cdot x_j \cdot R^i \pmod{d} \) by computing \( y_{j,i} = y_{j,i-1} \cdot R \pmod{d} \). The units are designed to make the evaluations for one public key. Therefore, for a public key of size \( s \) we reuse the units for each \( x_j \). In the following we give the design details.

Binary Computation Unit. The BINARY COMPUTATION UNIT is illustrated in Figure 7. It consists of \( p \) bit quotient evaluation units, a \( p \) bit buffer and a storage table that has size of \( p \cdot S \) and denoted as Precison Bit Table. As shown in the figure, the quotient evaluation unit is an architecture that performs binary division by shift and subtraction operations. By using this design, we have a smaller area and the timing overhead will still remain small compared to the overall timing. The evaluation of precision bits, for a specific value of \( j, i \) is as follows:
1) The **Quotient Evaluation Unit** takes the first $k_1$ bits of the values $y_{j,i}$ and $d$ which are loaded into storage denoted as $y_{head}$ and $d_{head}$.

2) Using a comparator $y_{head}$ and $d_{head}$ is compared:

$$\begin{align*}
if \ y_{head} & \geq d_{head} \quad b^{(l)}_{i,j} = 1 \\
else \quad b^{(l)}_{i,j} = 0
\end{align*}$$

3) The precision bit $b^{(l)}_{i,j}$ is loaded into the buffer. The value $d_{head}$ is updated as $d_{head} = d_{head} \gg 1$ using a 1-bit shifter. Also, $y_{head}$ is updated according to the value of $b^{(l)}_{i,j}$ as:

$$\begin{align*}
if \ b^{(l)}_{i,j} = 1 \quad y_{head} = y_{head} - d_{head} \\
else \quad y_{head} = y_{head}
\end{align*}$$

4) We iterate Steps 2 and 3 until all the precision bits are calculated.

5) The **Precision Bit Table** has $S$ rows and each evaluated precision bit is loaded to the $i^{th}$ row of the table from the buffer.

With 64-bit word size arithmetic; the loading operation takes $k_1/64$ cycles, each update of the values takes $k_1/64$ cycles and each precision bit evaluation with comparison takes 1 cycle. The process of one precision bit evaluation takes $(\frac{p+1}{64} + 1)$ cycles.

![Fig. 7. Binary Computation Unit](image)

**Modular Computation Unit.** The **Modular Computation Unit** is used to evaluate $y_{j,i}$ using the equation $y_{j,i} = y_{j,i-1} \cdot R \pmod{d}$ and setting $y_{j,0} = c \cdot x_j \pmod{d}$. The value $R$ is a special number equal to $2^{103}$. This simplifies the multiplication into a simple shift operation. Also, the modulus reduction operation can be evaluated by a scaled subtraction operation, i.e. $y_{j,i} - t \cdot d = y_{j,i} \pmod{d}$, where $t$ is the largest coefficient that ensures $t \cdot d < y_{j,i}$. By combining these two, the computation can be expressed as $y_{j,i} = (y_{j,i-1} \ll 103) - t \cdot d$.

In the equation coefficient $t$ is at most 103-bit value, so we are able to design a fast modular reduction unit that computes and multiplies the small coefficients with million-bit numbers. The design is illustrated in Figure 8, which consists of 103-bit quotient evaluation unit, 64x128 bit multiplier unit, a carry accumulate unit, 103-bit shifter, 64 bit subtractor and a local storage. The evaluation of $(y_{i-1,j} \ll 103) - t \cdot d$ is performed with the following steps:

1) The 103-bit **Quotient Evaluation Unit** takes the first $k_2$ bits of the values $y_{j,i-1}$ and $d$.

2) The **Quotient Evaluation Unit** evaluates the $t$ value as a 103-bit number as explained in Binary Computation Unit.

3) Since the evaluations output one bit at a time, the bits are loaded into a 128-bit buffer. The buffer will hold 103-bit $t$ with a 25-bit zero padding to feed $t$ as a constant value.

4) After computing $t$, we can evaluate $y_{j,i} = y_{s} - t \cdot d = (y_{j,i-1} \ll 103) - t \cdot d$.

The evaluation of $y_t$ by the **Quotient Evaluation Unit** takes $(\frac{104k_2}{64} + 1)$ cycles. In the rest of the computations, the design inputs two million-bit numbers and outputs a million-bit result. The design is fully pipelined and able to generate a result at each clock. Also, the pipeline delay is small that we can neglect in the timings. Therefore, the transactions takes 47000/bandwidth cycles to finish an evaluation, where $\text{bandwidth}$ is the rate of digits per clock cycle.

**Filling the Precision Bits Table.** In order to complete the operations and fill the Precision Bits Table, we use Modular Computation Unit and Binary Computation Unit in turns. Using a local Control Unit, we iterate the modular computation and binary computation for $S$ times to complete the table for a single public key. Each public key has the initial modular multiplication $(c \cdot w_j \pmod{d})$ which we can reduce eliminate extra operations by; converting $c$ to NTT form for once and using in each public key evaluation, pre-storing the public keys in NTT from. Therefore, we only perform digit multiplications, inverse-NTT conversions and the Barrett Reduction. Then, completing the table for a single public key takes:

$$93.5N + \left(\frac{104k_2 + (p + 1)k_1}{64} + 2 + \frac{47000}{\text{bandwidth}}\right) \times S$$

cycles. Using the same units for other public keys adds a factor of $s$ to the overall timing, i.e. $s \cdot \text{time}_{\text{table}} + 16N$. However, each public key has an independent operation which we can benefit by using multiple of these units. Still we need to increase the bandwidth by the number of units times to achieve the speedup.

![Fig. 8. Modular Computation Unit](image)

2) **Evaluating the Sum of Public Keys:** Recall the equation for the summation of the public keys:

$$s_j = \sum_{a \in [l]} \beta_{j,a} \left( \sum_{b \in [l]} \beta_{j,b} z_{j,i(a,b)} \right) \pmod{d}.$$  

As before we chose to store the $\beta$’s in NTT form to eliminate the conversions and rewrite the equation as:

$$s_j = \sum_{a \in [l]} B_{(j,a)} \left( \sum_{b \in [l]} B_{(j,b)} z_{j,i(a,b)} \right).$$

$^6$Adds an initial $16N$ clock cycles
Since $z_{j,i}(a,b)$ is a $p$-bit value, it is denoted as $z_{j,i} = \{b_{j,i}^{(p-1)}, \ldots, b_{j,i}^{(1)}, b_{j,i}^{(0)}\}$. Then, $s_j$ turns into a $p$ sized array that each bit computation is performed separately. By denoting $s_j = \{s_{j}^{(p-1)}, \ldots, s_{j}^{(1)}, s_{j}^{(0)}\}$, we can expand the equations as:

$$s_{j}^{(k)} = \sum_{a \in \{0\}} B_{j,a} \left( \sum_{b \in \{0\}} B_{j,b} b_{j,i}^{(k)} \right)$$

where $k$ is the bit index. For the evaluation of the equation, we designed a Recryption Processing Element (RPE) which includes small local storage for rapid calculations. As clearly shown in the equation the same $B$ inputs are used in all the evaluations. Therefore, we formed an array of $p$ RPE units and distribute each $b_{j,i}^{(k)}$ to a unit. By doing that, we compute all $s_{j}^{(k)}$ evaluations with a single transactions rather than $p$. Furthermore, we can replicate the RPE array for processing multiple blocks since the evaluations are in NTT form. Likewise encryption, we can replicate RPE arrays to speed up the computations. The bandwidth limits the number of RPE arrays we can utilize. The design illustrated in Figure 9 consists of the RPE Arrays, the Precision Bits Table\(^8\) and a Control Unit. In the following we give design details of the units and describe their working mechanisms.

![Recryption Architecture](image)

Recryption Processing Element. The design of the Recryption Processing Element is illustrated in Figure 10. The unit consist of two 64-bit modular adders, one 64-bit modular multiplier, a multiplexer and two local storage units. The local storage units are referred as $\text{up}_c$ and $\text{low}_c$ and has size of $\kappa$-digits each. The unit is fully pipelined with a total of 11 clock-cycle depth.

A complete block evaluation of the equation, performed by RPE is shown in Algorithm 4.

**Algorithm 4: Recryption Algorithm**

**Input:** $B_j = \{B_{j,l}, \ldots, B_{j,1}, B_{j,0}\}, b_{j,i}^{(k)} \in \{b_{j,i}^{(p-1)}, \ldots, b_{j,i}^{(1)}, b_{j,i}^{(0)}\}$

**Output:** $s_j = \sum_{a} B_{j,a} \left( \sum_{b} B_{j,b} \cdot b_{j,i}^{(k)} \right)$

1. $\text{low}_c = 0$
2. for $a = 0 \rightarrow l - 1$
   3. $\text{up}_c = 0$
   4. for $b = a + 1 \rightarrow l - 1$
      5. if $b_{j,i}^{(k)} = 1$
         6. \[\text{up}_c = \text{up}_c + B_{j,b}\]
      7. $\text{low}_c = \text{low}_c + \text{up}_c \cdot B_{j,b}$

Fig. 10. Recryption Processing Element

Control Unit. The Control Unit incorporates a state machine to handle the transactions and compute the Recryption operation. It controls the Precision Bits Table for requesting the required $z_{j,i}$ bits with index $i$ and they are directly fed to the RPEs. The unit controls the request_bits, read/write and clear signals to perform the arithmetic. The architecture is illustrated in Figure 9. Including the output transactions the operation requires $(S + x \cdot p) \cdot N \cdot (1 + l / 11 / \kappa) \cdot s$ clock cycles, in which factor of $s$ comes from the public key number and $\phi$ comes from the number of RPE arrays. Using $S = 512$ and $p = 5$, we evaluate $l$ as 46 and $x$ as 14, then the timing will be $531 \cdot N \cdot (1 + l / 11 / \kappa) \cdot s$.

3) Conversion and Reduction of Sum of Public Keys: Once the precision bits for each public key are evaluated, they need to be converted back from the NTT domain. Using an inverse-NTT and Barret Reduction algorithms, the conversions will take $89\cdot N$ clock cycles. Having $s$ public keys and $p$ precision bits, the total operation will take $s \cdot p \cdot 89\cdot N$ cycles. The operations can be parallelized by using multiple large multiplier units. This will increase the area by the number of multiplier units, but will reduce the computation time by the same factor.

4) Grade School Addition: In this section we explain the method we used to add five 15-bit numbers, where each bit of every number is in encrypted form. All the bits are represented by a very large number and a conventional addition algorithm does not apply. Assume we are realizing the bitwise
addition operation: \( \{c, s\} = x + y \) where \( c \) is the resulting carry bit of the addition operation in V-C4 and \( s \) is the sum. The logic realizing this operation is called a half adder. The result \( c \) and \( s \) can be represented as follows: \( c = x \text{ AND } y \) and \( s = x \text{ XOR } y \). Given that we realize this half-adder on the ciphertext, we can modify the equations as follows: For \( \{C, S\} = X + Y \) we write \( C = X \times Y \) and \( S = X + Y \) where \( A, B, C \) and \( S \) are ciphertext and multiplication and addition operations are large-number modular arithmetic.

Since 15 5-bit numbers in ciphertext form need to be added, we utilize Wallace tree [28] approach to minimize the number of large multiplications. For this operation, we utilize a total of 78 large multiplication operations and a total of \( 33 + 33 + 27 + 14 = 139 \) large addition operations. The large multiplication operations require modular reductions, so that the bit growth is prevented. In the evaluation of large multiplication operations and a total of 256-digit cache size. The Timing and a single EPE takes only about 26% of the entire Encryption operation. Thus the area overhead of additional EPE units does not yield any significant performance return.

The encryption operation is the most significant yet slowest operation. Therefore, we aimed at optimizing this operation as much as possible. Using the recommended bandwidth settings, we reduced the total time of the encryption operation to 3.1 seconds. For the initial two steps of the operation, the evaluation of \( z_{j,i} \) and sum of PK, the same elements can be utilized to realize the evaluation for each public key. Since they are independent, the timing of the first two steps can be reduced to \( 1.075/\lambda \), in which \( \lambda \) is the number of arithmetic units utilized for these steps. This will increase the area and bandwidth requirements for those operations by a factor of \( \lambda \). For the last two steps of the operation, we can reduce the timing by adding more LARGE INTEGER MULTIPLIER units. Using \( \kappa \) multipliers, the latency of the I-NTT & Reduction operation can be reduced by a factor of \( \lambda \) and the delay of the Wallace-Tree can be reduced by close to a factor of \( \lambda \). In multiplication, I/O transactions take 20% of the total operation. Therefore, with \( \kappa \) multipliers we can perform multiple operations without increasing the bandwidth. This will reduce the operation latency by a factor of \( \kappa \).

The design is synthesized with local cache sizes of 256, 128 and 64 digits and the area results are shown in Table V. The local cache sizes affect the area of ENCRYPTION, RECRYPTION and BARRETT REDUCTION units only. Among these three units, the RECRYPTION Unit covers the largest area with 1.17 million gates for a 256-digit cache size. The DECRIPTION Unit, whose only purpose is to take the least significant bit and to augment it with zero bits, does not have any local cache and consumes only about 200 gates and additional rewiring. The LARGE INTEGER MULTIPLIER has a fixed size cache that makes it the largest hardware unit in the design with 26.7 million gates.

<table>
<thead>
<tr>
<th>Operation</th>
<th># of Clock Cycles</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Integer Multiplication</td>
<td>52.5N</td>
<td>7.75 msec</td>
</tr>
<tr>
<td>Barrett Reduction</td>
<td>75N</td>
<td>10.70 msec</td>
</tr>
<tr>
<td>Decryption</td>
<td>109.5N</td>
<td>16.16 msec</td>
</tr>
<tr>
<td>Encryption</td>
<td>EPE &amp; Reduction</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>( \frac{209}{89N} )</td>
<td>4.98 msec</td>
</tr>
<tr>
<td></td>
<td>(1.039)</td>
<td>13.12 msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.10 msec</td>
</tr>
<tr>
<td>Recryption</td>
<td>Evaluation of ( z_{j,i} )</td>
<td></td>
</tr>
<tr>
<td>Sum of PK</td>
<td>( \frac{88N + 8405}{2} )</td>
<td>0.488 sec</td>
</tr>
<tr>
<td>Grade School</td>
<td>( \frac{511N}{2} \cdot (1.042) )</td>
<td>0.612 sec</td>
</tr>
<tr>
<td></td>
<td>( s \cdot \frac{89N}{2} )</td>
<td>0.985 msec</td>
</tr>
<tr>
<td></td>
<td>( 6967N )</td>
<td>1.027 sec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.112 sec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>64-digit Cache</th>
<th>128-digit Cache</th>
<th>256-digit Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Integer Mul.</td>
<td>26.7</td>
<td>25.7</td>
<td>26.7</td>
</tr>
<tr>
<td>Barrett Reduction</td>
<td>0.029</td>
<td>0.0356</td>
<td>0.0647</td>
</tr>
<tr>
<td>Decryption</td>
<td>0.0002</td>
<td>0.0002</td>
<td>0.0002</td>
</tr>
<tr>
<td>Encryption</td>
<td>0.1047</td>
<td>0.136</td>
<td>0.206</td>
</tr>
<tr>
<td>Recryption</td>
<td>0.674</td>
<td>0.7741</td>
<td>1.177</td>
</tr>
</tbody>
</table>

| TABLE V | AREA INFORMATION OF HARDWARE BLOCKS |

**VI. IMPLEMENTATION RESULTS**

The design was synthesized with Synopsys Design Compiler using 90 nm TSMC Library. Timing analysis shows a maximum frequency of 666 MHz. A moderate memory speed of 1333 MTps (Megatransfers per second) is selected for the Main Memory (RAM).

The ratio between the memory and the main circuit frequency, i.e. \( \beta = \phi = \frac{1333}{90} = 2 \), results in I/O speed of 2 transactions (digits in our case) per clock cycle, which led us to set our Encryption Processing Element and Recryption Processing Element Array numbers as 2 each. Also, local cache sizes of the Recryption Processing Element, Encryption Processing Element and Barrett Reduction Unit was selected as 256 digits to have a smaller pipeline delay, i.e. \( \kappa = 256 \).

Finally, as mentioned before, we incorporated a single multiplier into our design. Under these settings, the timings are found to be as shown in Table IV.

Large Integer Multiplication, Barrett Reduction and Decryption operations share a single Large Integer Multiplier. Therefore their latencies are directly related to the latency of the multiplier. By increasing the number of EPE units as well as the bandwidth, we can reduce the latency of the Encryption operation. However, the encryption latency is already small and a single EPE takes only about 26% of the entire Encryption operation. Thus the area overhead of additional EPE units does not yield any significant performance return.
the same scheme. In Table VII, we compare our design with the software implementation of GH-FHE scheme in [16] and the GPU implementation in [14].

In **large integer multiplication**, the time performance of our design is close to that of the Xeon software implementation and ~10 times slower than GPU implementation. Decryption is 20% faster compared to Xeon software but it is 6.5 times slower than the GPU implementation. Encryption is 101 times faster than the Xeon software and 12.3 times faster than the GPU implementation.

However, the most significant and time-consuming operation is Recryption and we are 10 times faster than the Xeon software implementation. Moreover, our design is still 1.1 second faster than the GPU implementation. We should note however that our hardware runs at a slower frequency with a significantly lower gate count of less than 30 million equivalent gates. In contrast, the NVidia GPU in [14] contains approximately 900 million and the Xeon processor contains 205 million gates. This shows the benefit of our ASIC design compared to general purpose CPU and GPU implementation with much higher performance at lower area cost.

### VIII. Conclusion

In this work we took initial steps to remedy the efficiency bottleneck of fully homomorphic encryption schemes by introducing the first custom FHE architecture. For this we introduced a novel large integer multiplier design realizing the famous Schönhage Strassen algorithm in hardware. We further extended the multiplier by Barrett’s reduction algorithm to handle million bit modular multiplication operations. Using this core we implemented the Gentry-Halevi FHE primitives, e.g. encryption, decryption, and recryption. Among these primitives we managed to improve the efficiency of the challenging recryption operation to the point where we are surpassing its software implementation performance on a high end GPU processor at a fraction of the footprint. Not surprisingly, this shows that a custom application specific FHE design can significantly outperform current software implementations. Furthermore, during the completion of this work, several new (theoretical) FHE constructions emerged whose performance could also benefit greatly by the presented approach.

### References


<table>
<thead>
<tr>
<th></th>
<th>Multiplication</th>
<th>Decrypt</th>
<th>Encrypt</th>
<th>Recrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>7.750</td>
<td>16.1</td>
<td>18.1</td>
<td>3.1</td>
</tr>
<tr>
<td>GPU [14]</td>
<td>0.765</td>
<td>2.5</td>
<td>220</td>
<td>4.2</td>
</tr>
<tr>
<td>Xeon [16]</td>
<td>6.667</td>
<td>20.0</td>
<td>1300</td>
<td>32</td>
</tr>
</tbody>
</table>

**Table VI** Timing Comparison