Accelerating Fully Homomorphic Encryption in Hardware

Yarkın Doröz, Erdinç Öztürk, and Berk Sunar

Abstract—We present a custom architecture for realizing the Gentry-Halevi fully homomorphic 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 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 gates.

Index Terms—Fully homomorphic encryption, application specific hardware, cryptographic accelerators, large-integer multiplication

1 Introduction

One of the most significant developments in the field of cryptography in the last few years has been the introduction of the first fully homomorphic encryption (FHE) scheme by Gentry [1]. Gentry's lattice-based scheme appears to be secure and hence settles an open problem posed by Rivest et al. [2].

Since addition and multiplication in any non-trivial ring constitute a universal set of gates, a fully homomorphic encryption scheme allows one to employ untrusted computing resources without risk of revealing sensitive data. Using FHE, computation carried out directly on ciphertext carries over to the underlying plaintext, therefore private computation becomes possible. FHE holds great promise for numerous applications including private information retrieval and search, data aggregation, electronic voting, biometrics, etc. For instance, using FHE we may keep sensitive databases, e.g. medical and financial records, in encrypted form and perform private queries without any risk of compromising privacy. In general, by deploying FHE we may mitigate the vulnerabilities stemming from imperfect software.

Unfortunately, FHE has not yet sufficiently matured to be used in real-life deployments. The bandwidth due to 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 without any implementation results. Here we take a snapshot of the attainable hardware performance of the Gentry-Halevi FHE variants. 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 hardware products due to the inefficiency of general purpose computers. We speculate a similar growth pattern to emerge in the maturation process of FHEs. While the 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. During the progression of this work potentially more efficient schemes appeared in the literature, e.g. see [4], [5], [6], [7]. Since most of these schemes are defined over ideal-rings they also rely on arithmetic with large numbers or polynomials with large coefficients. The IP-cores we developed for fast number theoretic transform based multipliers with efficient modular reduction may also be used to accelerate these newer schemes.

In this paper, 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 encryption, decryption and recryption primitives for a 2,048-dimension instantiation of the Gentry-Halevi scheme. We utilize a number of optimizations, including reformulation of the operations, use of spectral techniques and precomputations 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.

- *Y. Doröz and B. Sunar are with the Department of Electronics and Computer Engineering, Worcester Polytechnic Institute, Worcester, MA 01906. E-mail: {ydoroz, sunar}@wpi.edu.*
- *E. Öztürk is with the Department of Electrical and Electronics Engineering, Istanbul Commerce University, Istanbul, Turkey. E-mail: eozturk@ticaret.edu.tr.*

Manuscript received 21 Apr. 2013; revised 25 Jan. 2014; accepted 26 June 2014. Date of publication 4 Aug. 2014; date of current version 13 May 2015. Recommended for acceptance by J. Plusquellic. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2014.2345388

2 Background

2.1 Gentry's Fully Homomorphic Scheme

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 $^\circ(f) = n$) with euclidean lattices $\mathbb{L}_I \subseteq \mathbb{R}^n$ 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 additive and multiplicative homomorphisms modulo a public key ideal. We obtain an encryption procedure Encrypt such that $\mathsf{Encrypt}(x_1) + \mathsf{Encrypt}(x_2) = \mathsf{Encrypt}(x_1 + x_2)$ and $\mathsf{Encrypt}(x_1) \cdot \mathsf{Encrypt}(x_2) = \mathsf{Encrypt}(x_1 \cdot x_2)$. Therefore, any circuit C with efficient description can be evaluated homomorphically. However, this **somewhat** fully homomorphic scheme (SWHE) is not perfect. Due to the noisy nature of the scheme, with each homomorphic gate evaluation 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 SWHE scheme by simply homomorphically evaluating the decryption circuit using encrypted secret key bits on the noisy ciphertext. To make this work, the SWHE needs to be able to handle circuits that are deeper than its own decryption circuit before the level of noise becomes too large. SWHE schemes with this property are called **bootstrappable**.

2.2 The Gentry-Halevi FHE

Smart and Vercauteren specialized Gentry's scheme to principal-ideal lattices, and forced the determinant of the lattice to be a prime number [8]. While this specialization improves the efficiency, it does not allow construction of the full scheme including bootstrapping and Recrypt for practical key sizes [8]. 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. Here we give a high-level description of the primitives as follows. Let $\lceil N \rfloor$ denote the round to nearest integer operation, $[N]_d = (N \pmod d) - d$, and $[N] = \{0, 1, \ldots, N-1\}$.

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

1) Set $f_n(x) = x^n \pm 1$. Choose a random $n = 2^\theta$-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 $w(x)v(x) = d \,(\mathrm{mod} f_n(x))$ where d represents a constant integer. This task may be achieved by using the polynomial version of the Extended Euclidean Algorithm.[1]
3) Compute $r = w_0/w_1 (\mathrm{mod}\ d)$ and check if $w_i = w_{i+1}r \pmod d$ 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 re-encryption, randomly choose bit-vectors σ_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 $\sum_{j=0}^{s-1}\sum_{i=0}^{S-1} \sigma_i(j)x_jR^i \pmod d = w'$. The parameter $R \in \mathbb{Z}$ may be chosen as a power of 2.
5) Let $l = \lceil 2\sqrt{S} \rceil$. For re-encryption, pick bits $\eta_{i,j}$ for $i \in [s], j \in [l]$ where $\eta_{i,j}$ has Hamming weight 2 when viewed as an l-dimensional vector. Then encrypt each to obtain $\beta_{i,j} = \mathsf{Encrypt}(\eta_{i,j})$.
6) The public key is $\mathsf{PK} = (r, d, \{x_i : i \in [s]\}, \{\beta_{i,j} : i \in [s], j \in [l]\})$ and the secret key is $\mathsf{SK} = (w', \sigma_0, \sigma_1, \ldots, \sigma_{S-1})$.

Encryption. To encrypt a bit $m \in \{0, 1\}$ first choose an nth degree sparse random polynomial $u(x)$ with coefficients from $\{0, 1, -1\}$, for which probability of a coefficient being 0 is ρ. Using the PK parameters (r, d) encryption is computed as follows: $\mathsf{Encrypt}(m) = [m + 2\sum_{i=0}^{n-1} u_i r^i]_d$. When multiple bits are to be encrypted, one may *batch* the computation yielding a significant speedup, e.g. k encryptions may be computed with only $O(\sqrt{k})$ times more costly than a single bit encryption using simultaneous polynomial evaluation.

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

Recryption. The goal of recryption 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 recrypt procedure. Informally, recrypt works by homomorphically decrypting the ciphertext using encrypted secret key bits. A given ciphertext c is recrypted by taking the following steps:

1) Compute $y_{j,i} = cx_jR^i \pmod d$ 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 $q_j = \sum_{a\in[l]} \beta_{j,a}(\sum_{b\in[l]} \beta_{j,b} z_{j,i(a,b)}) \pmod d$ where the index function is defined as $i(a,b) = al - \binom{a+1}{2} + (b-a)$. 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, the re-encryption of $c \in \mathbb{Z}_d$ is achieved by homomorphically evaluating the decryption circuit on c in encrypted form. After a number of optimizations, the decryption operation $\mathsf{Decrypt}_{\mathsf{SK}}(c)$ is expressed in the following form:

$$\left\lceil \sum_{j\in[s]}\sum_{i\in[S]} \sigma_j(i)z_{j,i} \right\rfloor + \sum_{j\in[s],i\in[l]} \sigma_j(i)(y_{j,i}) \pmod 2.$$

Note that the inputs d and $y_{j,i}$ are in cleartext form while the secret key σ_j are in encrypted form during the 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

1. Note that Section 4 of [8] presents a significantly more efficient technique for computing $w(x)$.

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 $\pmod 2$ 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. The second sum multiplies bits by ciphertexts in $\mathbb{Z}_d$.

2.3 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 for large integer multiplication (in million-bit range) which lies at the heart of all the primitives. Common multiplication schemes (Karatsuba Algorithm [9], schoolbook multiplication method) become infeasible for very-large integer multiplications. Schönhage-Strassen algorithm [10] is currently asymptotically the fastest algorithm for very-large numbers. It has been shown that it outperforms classic schemes for operand sizes larger than 2^{17} bits [11]. FFT-based large integer multiplier architectures were presented in [12], [13], [14].

Schönhage strassen algorithm. The Schönhage-Strassen Algorithm is an NTT-based large integer multiplication algorithm, with a runtime of $O(N \log N \log \log N)$ [10]. For an 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. A summary of the algorithm is as follows. For an in-depth review of the Schönhage Strassen algorithm see [15]. We sample the numbers A and B that fits into N digits with a sampling size ϵ. 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 $c_k = A_k \cdot B_k \bmod p$. Using the inverse-NTT (INTT) 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 INTT computation techniques. We adopted the most common method for computing Fast Fourier Transform (FFT), i.e. the Cooley-Tukey FFT Algorithm [16]. The algorithm computes the Fourier Transform of a sequence X as $X_k = \sum_{j=0}^{N-1} x_j e^{-i2\pi k \frac{j}{N}}$, by turning the length N transform computation into two $\frac{N}{2}$ size Fourier Transform computations as follows:

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

where $\theta = e^{-2\pi k \frac{i}{N/2}}$. We change $e^{\frac{-2\pi i k}{N}}$ 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.

Modular reduction. We may use Barrett Modular Reduction (BMR) algorithm [17] to compute $r \equiv x \pmod M$ as following:

$$r \equiv \underbrace{x \pmod{b^{k+1}}}_{r_1} - \underbrace{\left\lfloor \frac{\lfloor x/b^{k-1} \rfloor \mu}{b^{k+1}} \right\rfloor M \pmod{b^{k+1}}}_{r_2}.$$

In the equation b is the radix and other parameters are $k = \log_b M + 1$ and $\mu = \lfloor b^{2k}/M \rfloor$. According to [17] r has the following equality: $r < 3M$. Therefore; after evaluating r, first we check if it is negative and perform $r = r + b^{k+1}$ and later we subtract M from r while $M < r$.

Block-wise arithmetic. In the further sections of the paper, we refer to block-wise (or block) computations. The term defines separation of a large integer in the NTT form into computational blocks where each block contains l digits. Each integer that is in NTT form will be formed of N/l number of blocks. Since the NTT structure of these large integers is suitable for performing parallel arithmetic, these blocks are distributed among arithmetic units.

3 Overview of Our Architecture

The overall architecture presented in Fig. 1 contains five components: 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 for each primitive is too costly. 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, handling the order of operations and I/O between the units and the external memory. Since the operands are in the range of million-bits, data transactions between units are impractical. We assume an external memory (RAM) in the design for storage. We utilize a 64-bit bus for I/O transactions between the units and the external RAM. Holding the public keys, RAM acts as a shared memory between the units when the primitive operations are realized. The total operation time of a unit is computed as the sum of the time needed to read data from the RAM, the latency of the 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.

3.1 Parameter Selection

In the following we explain the details of the parameter selection for Large Integer Arithmetic to support million-bit multiplication operation. Next, we give parameters for FHE primitives and show some potential trade-offs.

Large integer arithmetic parameters. The parameters for the NTT-based implementation is based on [18], [19]. We choose a 64-bit word size, and a sampling size of $\epsilon = 2^{24}$ with modulus $p = 2^{64} - 2^{32} + 1$, a Solinas prime [20]. This allows us to realize a modular reduction using a few primitive

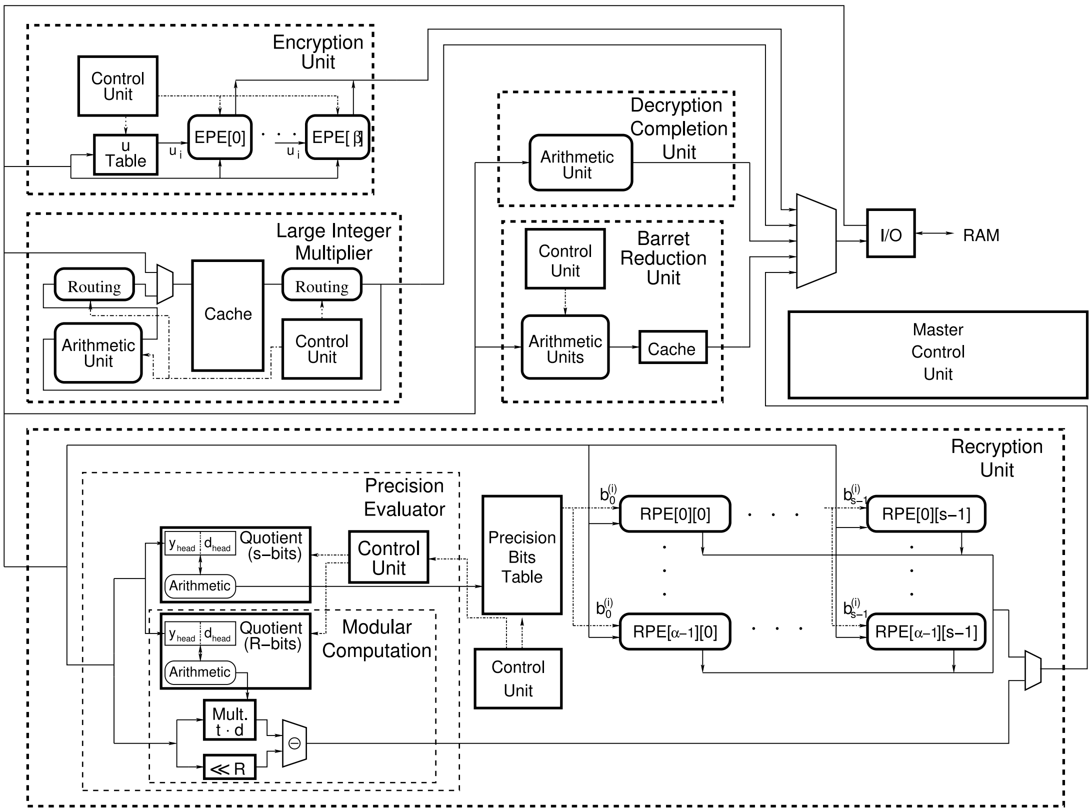

Fig. 1. Overview of the full architecture.

arithmetic operations. A 128-bit number is denoted as $z = 2^{96}a + 2^{64}b + 2^{32}c + d$. 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 overflow. Also, N should be large enough to enable million-bit multiplication with smallest possible value, i.e. $2 \text{ million bits} < N \cdot \epsilon$. The best candidate for N is determined as $3 \cdot 2^{15}$.[2] Given the parameters and Equation $w^N \equiv 1 \bmod p$, $w = 3511764839390700819$.

In Cooley-Tukey FFT, each recursive halving operation is referred as a stage and it is denoted as S_i, where i is the stage index. The size of the smallest NTT block is selected as 12 digits and it is referred as the 0th stage, i.e. S_0. The remaining 13 stages are reconstruction stages and require different arithmetic operations from the ones in S_0. In terms of INTT operations, every stage and operation is identical to NTT. Only difference is selection of w', which is computed as: $w' = w^{-1} \bmod p$.

FHE primitive parameters. We instantiate the scheme for the smallest parameters in [8] as; $n = 2{,}048$, $l = 46$, $s = 15$, $p' = 5$, $S = 512$, $\rho = \frac{2032}{2048}$ and $\log d = 7{,}85{,}000$.

For FHE primitives, we utilize addition operations in NTT form, i.e. $\frac{u}{2}\sum_{i=1}^{j} u_i(\epsilon-1)^2 < p$. The j value is equal to S and $(1-\rho)\cdot n$ in Recryption and Encryption respectively. Therefore, we choose a different sample rate ϵ to prevent overflow. Selecting $\epsilon = 2^{16}$, we support primitive operations up to 786,432 bits, which is larger than $\log d$.

2. We choose the digit size as a power of two which ease arithmetic computations.

4 Large Integer Architecture

The FHE primitives are based on efficient large integer arithmetic. In the following, we give the design details of a large integer multiplier and a modular reduction architecture.

4.1 Large Integer Multiplier

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 Fig. 2. The architecture is designed to perform a restricted set of special functions. There are four functions for handling the input/output transactions and three functions for arithmetic operations:

- *Sequential load.* Stores a million-bit number in the cache.
- *Sequential unload.* The cache releases its contents starting from the least significant to most significant.
- *Butterfly load.* Distributes the digits into the right indices using the butterfly operation.
- *Scale and unload.* Scales and outputs the result, i.e. $N^{-1} \pmod p$.
- 12×12 *NTT/INTT.* The smallest NTT/INTT computation is for 12 digits. NTT Unit[3] takes the digits sequentially and computes the 12-digit NTT/INTTby using simple shift and add operations.
- *Stage-reconstruction.* This function is used for reconstruction of a stage. In order to complete a full reconstruction, it is called for 13 stages.

3. We refer to 12×12 NTT/INTT Unit as NTT Unit.

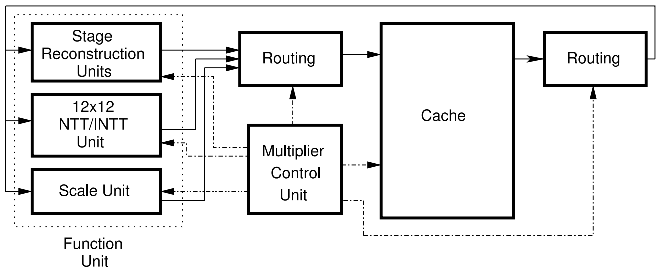

Fig. 2. Overview of the large integer multiplier.

- *Inner-multiplication.* Computes 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. $\mathsf{NTT}(A)$, is computed by calling; first 12×12 NTT function, and afterwards Stage-Reconstruction function for all stages.
3) $\mathsf{NTT}(A)$ is stored to RAM using Sequential Unload.
4) Using steps 1-2-3 above, we also compute $\mathsf{NTT}(B)$.
5) The cache can only hold the half of the digits of $\mathsf{NTT}(A)$ and $\mathsf{NTT}(B)$ together. Therefore, the numbers are divided into lower and upper halves: $\mathsf{NTT}(A) = \{\mathsf{NTT}(A)_h, \mathsf{NTT}(A)_l\}$ and $\mathsf{NTT}(B) = \{\mathsf{NTT}(B)_h, \mathsf{NTT}(B)_l\}$.
6) Sequential Load stores $\mathsf{NTT}(A)_h$ and $\mathsf{NTT}(B)_h$.
7) Inner Multiplication computes modular multiplication of the upper halve: $C_h[i] = \mathsf{NTT}(A)_h[i] \ * \mathsf{NTT}(B)$ $h[i]$.
8) The result is stored to the RAM by Sequential Unload.
9) We repeat above three steps to compute the lower part: $C_l[i] = \mathsf{NTT}(A)_l[i] * \mathsf{NTT}(B)_l[i]$.
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, 12×12 INTT function which is followed by a complete Stage-Reconstruction function $C' = \mathsf{INTT}(C[i])$.
12) In the last step, the result is scaled and the carries are accumulated by Scale & Unload function to finalize computation of C: $C[i+1] = C'[i+1] + \lfloor C'[i]/p \rfloor$ and $C[i] = C'[i](modR)$.

Multiplier cache system. The size of the cache is important for the latency 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: $S_{i,diff} = 12 \cdot 2^{i-1}$, where i is the index of reconstruction stages, i.e. $1 \leq i \leq 13$. 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.

Lets 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 $N' < S_{i,diff}$ for the ith stage. Then, starting from the ith stage, N_j requires digits from N_{j+1} where 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 $2N \cdot \log_2(N/N')$, where $\log_2(N/N')$ 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.

TABLE 1
Assignment Table

	arith$_0$	**arith$_1$**	**arith$_2$**	**arith$_3$**
S_1-S_{10}	sc_0-sc_1	sc_2-sc_3	sc_4-sc_5	sc_6-sc_7
S_{11}	sc_0-sc_1	sc_2-sc_3	sc_4-sc_5	sc_6-sc_7
S_{12}	sc_0-sc_2	sc_1-sc_3	sc_4-sc_6	sc_5-sc_7
S_{13}	sc_0-sc_4	sc_1-sc_5	sc_2-sc_6	sc_3-sc_7

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 12×12 $\mathsf{NTT/INTT}$,[4] Stage-Reconstruction and Inner Multiplication, are highly suitable for parallelization. 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, we can select the cache to be used as a single-cache or a multi-cache system. In case of linear functions, such as Sequential Load, Butterfly Load, etc., the cache works as a single-cache with one I/O port, where as for parallel functions, it works as a multi-cache system with multiple I/O ports. The number of sub-caches should be equal to $2 \times m$ (double the size of Stage-Reconstruction Unit number) to eliminate access read/write to the same sub-cache in the reconstruction process. Each sub-cache has a size of $N/(2 \times m)$ and we denote them as; $\{sc_0, sc_1, \ldots, sc_{2m-1}\}$.

Routing unit. The Routing Unit matches the *odd* and *even* digits to the arithmetic units. As stated previously, the indice difference of the digits is $(12 \cdot 2^{i-1})$. Therefore, in last $\log 2m$ reconstruction stage, *odd* and *even* digits fall into different sub-caches. The assignment of sub-caches to proper arithmetic units[5] for each Stage-Reconstruction is shown in Table 1. In the Table, arithmetic units are referred as $arith_i$, which i is the index number.

Function unit. The Function Unit is divided into three parts, i.e. the Scaler Unit, the NTT Unit and multiple Stage-Reconstruction Unit.

Scaler Unit. Denoting the digits as d_i and including the carries as c_i, digits of the result is $d_i \times N^{-1} + c_i \pmod p = \{c_{i+1}, r_i\}$. As $N^{-1} \pmod p = 0\text{xFFFF}555455560001$—a constant number with a special form, we implemented the product using simple shift and add circuit.

4. We used one 12×12 $\mathsf{NTT/INTT}$ unit for this function. For few number of arithmetic units for multiplier, i.e. $m = 4$, the performance gain is percent 3. However, for larger m such as 64, performance gain will go up to percent 20.

5. Arithmetic units are the Stage-Reconstruction Units.

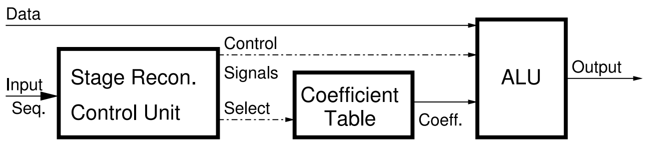

Fig. 3. Stage reconstruction unit.

NTT Unit. The unit computes 12-digit NTT and INTT using the formula: $x_i = \sum_{i=0}^{11} (w')^i \times d_i \bmod p$, where d_i is the given 12 digit input. The parameter w' is set as; $w' = w^{2^{13}} \pmod p$ and $w' = (w^{-1})^{2^{13}} \pmod p$ for NTT and INTT operations respectively. Note that in NTT $w' = 0\times10{,}000$ and in INTT $w' =$ 0xFFFFEFFFF00010001. These constant multiplications are implemented using simple add and shift circuits. These simple operations can be squeezed into few clock cycles and pipelined to optimize throughput. Due to pipelining, 12-digit NTT of the large integer is completed in N clock cycles.

Stage Reconstruction Unit. The unit in Fig. 3 is responsible for two functions; Stage-Reconstruction and Inner Multiplication. The Arithmetic Logic Unit (ALU) in Fig. 4 consists of 32-bit multipliers, adders and a reduction circuit to complete 64-bit modular multiplications.

In Inner Multiplication 64-bit numbers are fed into Odd and Coeff bus. Even is fed with zero, so that ALU only performs modular multiplication. The ALU can output a modular multiplication product in every two clock cycles after the initial startup cost of the pipeline. The whole function takes $\frac{N}{2}$ multiplications and with m multipliers it will cost $\frac{N}{m}$ clock cycles.

In Stage-Reconstruction we compute. $O_{i,j} = E_{i-1,j} - O_{i-1,j} \times w_{i-1}^{j \bmod n_{i-1}} \pmod p$ and $E_{i,j} = E_{i-1,j} + O_{i-1,j} \times w_{i-1}^{j \pmod{n_{i-1}}} \pmod p$ where, 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. The following equation is true for n_i parameters; $n_{i+1} = 2 \times n_i$ with initial setting of $n_0 = 12$. Ideally we need to store all the coefficients along with the *odd* digits. However this will require another large cache of size $(12 + 24 + 48 + \cdots + 49{,}152) \approx N$ digits. Although we save half the memory size by reuse of the powers for different stages, necessity of storing w^{-1} doubles the size requirement. We reduce the memory requirement by using memory-time tradeoff. The coefficients are computed efficiently as follows:

1) The coefficients required in two consecutive stages are as follows: $S_{i+1} : w_i^0, w_i^1, \ldots, w_i^{n_i}$ and $S_{i+2} : w_{i+1}^0, w_{i+1}^1, \ldots, w_{i+1}^{n_{i+1}}$.
2) Then $S_{i+2} : w_{i+1}^0, w_{i+1}^1, \ldots, w_{i+1}^{2n_i}$ since it holds that $n_{i+1} = 2 \times n_i$.
3) Further, $S_{i+2} : w_i^{\frac{0}{2}}, w_i^{\frac{1}{2}}, \ldots, w_i^{\frac{2n_i}{2}}$, since $w_i = w_{i+1}^2$.
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 compute square roots by multiplying each w_i^j with w_{i+1}^1.

Thus, we construct the 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

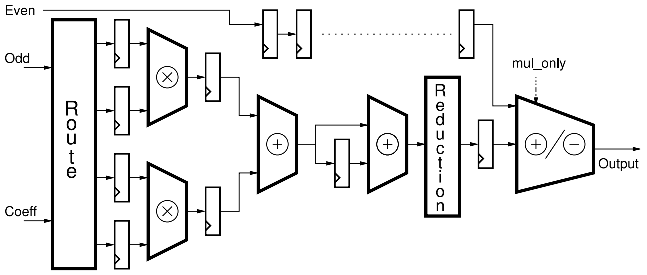

Fig. 4. ALU of the stage reconstruction unit.

coefficients as $w_{first,i}$, 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 $w_{second,i}$. 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 $w_i^j = w_{first,l} \times \prod_{t=0}^{i} w_{second,t}^e$.

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 evaluating $w_i^j \times O_i$, is computed as

1) In every reconstruction stage we start by multiplying *odd* digits with $w_{first,l}$'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 $w_{second,0}$ to $w_{second,i-1}$. 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 for each stage we need $\frac{N}{4}$ additional multiplications.
3) The total number of multiplications becomes $\sum_{i=0}^{i=12} (\frac{N}{2} + i \times \frac{N}{4}) = 26 \times N$.

Multiplier control unit. The Multiplier Control Unit includes a state machine for a large integer multiplication operation. The main job is to send correct indices to Function Units to complete arithmetic functions, such as Inner Multiplication and Stage-Reconstruction. NTT Unit and Scale Unit consist only of datapath.

The Multiplier Control Unit also handles I/O addressing of the sub-caches. Sequential operation requires incremental addressing for each sub-cache. In Stage-Reconstruction operation, addressing is computed according to the stage level, which is basically updated with the index range of dependent *odd* and *even* digits.

Performance analysis. The latency of each functional block is given in Table 2. In order to perform a complete multiplication we require two complete NTT, two Inner Multiplication and one INTT operations.

4.2 Modular Reduction

lIn Barrett Reduction, the result is evaluated using two large integer multiplications and a few subtractions. The values μ and M are stored in NTT form to avoid conversion costs. Selecting $b = 2^{64}$ simplifies the arithmetic. Division and

TABLE 2
Clock Cycle Counts of Functional Blocks

NTT(A) , NTT(B)	2 Butterfly Load	$2N$
	2 12×12 NTT	$2N$
	2 Stage-Recon	$26N$
	2 Sequential Unload	$2N$
A×B	2 Sequential Load	$2N$
	2 Inner Multiplication	$\frac{N}{2}$
	2 Sequential Unload	$2N$
INTT(A×B)	Butterfly Load	N
	12×12 INTT	N
	Stage-Recon	$13N$
	Scale Unload	N
	TOTAL	$52.5N$

reduction operations, such as $\lfloor x/b^{k-1} \rfloor$ and $x \bmod b^{k+1}$, are accomplished by loading from different memory address. For division, bits are read from b^{k-1} to most significant bit and for modular arithmetic, bits are read from least significant to b^{k+1}.

Once Barrett Reduction is requested, the state machine performs the multiplications with M and μ, and computes r_1 and r_2 values. r_1 and r_2 are loaded for subtraction that is evaluated in digits by a simple subtracter with word length of 64 bits. Since reading both r_1 and r_2 occupies a huge portion of the bandwidth, a κ-digit local cache is added to store partial results to prevent the I/O collusions from/to the RAM. The local cache is based on two parallel $\kappa/2$-digit FIFOs so that we can output two digits per clock cycle. Later, r is checked if it is negative and it is corrected by setting zero after b^{k+1}. For comparison r and M are loaded into comparator unit to decide $r \geq M$. The decision is done by implementing a 512-bit comparator. If the first eight digits are equal then it loads the next eight digits for decision until they are not equal. If comparison is true, r is updated as $r = r - M$.

The time for a Barrett Reduction heavily depends on the multiplications and the subtractions have a small effect. The multiplications are completed in $2 \cdot 36.5N$ clock cycles. Subtractions take $\approx 3 \times 23{,}500$ clock cycles, which is omitted. The total time required for a Barrett reduction is $\approx 73N$ clock cycles.

5 FHE Primitives

5.1 Decryption

The decryption operation is a rather simple operation that requires a modular multiplication operation followed by a modulo 2 reduction, i.e. $\mathsf{Decrypt}(c) = [cw_i]_d \pmod 2$. During decryption, 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 least significant digit of the modular multiplication result and pads it with zeroes to match the operand length.

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 operation takes $36.5N$ clock cycles. The modulo 2 reduction is 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 Barrett Reduction, the overall decryption operation takes $109.5N$ clock cycles.

5.2 Encryption

The most time consuming part of encryption is evaluating powers of r. In [21], these are computed using a recursive algorithm. While asymptotically faster such a recursive approach is not suitable for hardware implementations. Instead we utilize a window-based serial evaluation scheme. Algorithm A′ proposed in [22] is suitable for efficient polynomial evaluation in hardware.

Algorithm 1. Algorithm A′

```
1. Define $u(r) = u_0r^0 + u_1r^1 + \cdots + u_{n-1}r^{n-1}$ and a window size $k$ as $k < n$ and $k \mid n$.
2. Group coefficients of $u(r)$ using powers of $r^k$ as:
   $(u_0r^0 + \cdots + u_{k-1}r^{k-1})r^0+$
   $(u_kr^0 + \cdots + u_{2k-1}r^{k-1})r^k + \cdots +$
   $(u_{n-k}r^0 + \cdots + u_{n-1}r^{k-1})r^{n-k}$.
3. Define Inner Polynomials as:
   $P^{(j)} = r^{j \cdot k}(\sum_{i=0}^{k-1} u_{(j \cdot k+i)} r^i)$
4. Then, the $u(r)$ polynomial can be rewritten as:
   $u(r) = \sum_{j=0}^{\frac{n}{k}-1} P^{(j)} = \sum_{j=0}^{\frac{n}{k}-1} r^{j \cdot k}(\sum_{i=0}^{k-1} u_{(j \cdot k+i)} r^i)$
```

The algorithm divides the evaluation into three steps. First, the polynomial terms are grouped into windows of k digits where each grouping is multiplied by increasing powers of r^k. After the summation operations, the window sums are scaled by the proper powers of r^k. The last step is to aggregate the scaled window sums. The algorithm reduces the number of multiplications to $k + \frac{2n}{k}$. A further speed-up is achieved by storing two tables: $\{r^0, r^1, \ldots, r^{k-1}\}$ and $\{r^k, r^{2k}, \ldots, r^{n-k}\}$. Since r is set during the KeyGen step the lookup tables can be precomputed. With the introduction of the lookup tables, the only multiplication operations needed are the ones computed when the window sums are multiplied by the power of r^k. Using the lookup tables, the number of multiplications are reduced further to $\frac{n}{k} - 1$ with a storage requirement of $\frac{n}{k} + k - 2$.

The algorithm can be further improved by realizing the operations entirely in the NTT domain. By storing the table elements in NTT form, an encryption operation may be realized as

$$\mathsf{Encrypt}(m) = \mathsf{INTT}\left[M + 2 \sum_{j=0}^{\frac{n}{k}-1} R^{j \cdot k} \left(\sum_{i=0}^{k-1} u_{(j \cdot k+i)} R^i \right) \right]_d ,$$

where we use uppercase symbols to denote the NTT form of the variables, e.g. $R = \mathsf{NTT}(r)$ and $R^j = \mathsf{NTT}(r^j)$. Since the message m is a single bit, we simplify $M = (0, \ldots, 0)$ if $m = 0$, else $M = (1, \ldots, 1)$ if $m = 1$. The equation eliminates NTT conversions and requires only one INTT and a single modular reduction at the end.

NTT-based arithmetic operations in aggregate, referred to as NTT Encryption, are evaluated with what we call the Encryption Unit. Remainder of the operations are completed

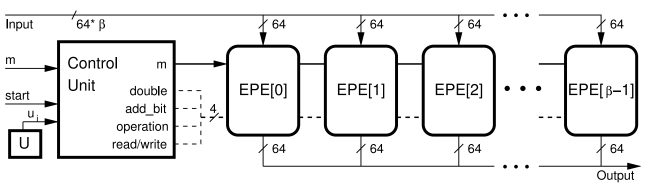

Fig. 5. Encryption architecture.

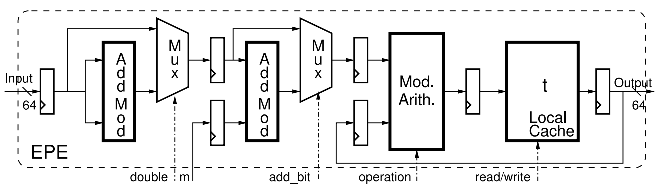

Fig. 6. Encryption processing element.

by utilizing the Large Integer Multiplier Unit and the Barrett Reduction Unit. To realize the encryption primitive, Master Control Unit runs Encryption Unit, Large Integer Multiplier and Barrett Reduction units in order.

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

Encryption processing element. EPE as shown in Fig. 6 is designed to evaluate NTT-Encryption of a block size κ. 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 stages. Therefore each block operation will have an extra 10 clock cycles for time evaluations, i.e. we multiply the total timing with $(1 + 10/\kappa)$. The unit evaluates encryption with 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_i value is zero, then $R^i_{(l)}$ 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 $k \cdot (1-\rho) + 2$, where $1-\rho$ is the probability of non-zero terms and plus 2 is for input of the scaler and output terms
.

Algorithm 2. Window Sum and Scale Operation

```
Input: $r = \{\{R^0_{(l)}, R^1_{(l)}, \ldots, R^{k-1}_{(l)}\}, \{R^k_{(l)}, R^{2k}_{(l)}, \ldots, R^{n-k}_{(l)}\}\}$,
       $u = \{u_0, u_1, \ldots, u_{n-1}\}$
Output: Inner Polynomial Block $P^{(j)}_l = R^{j\cdot k}_{(l)} \sum_0^{k-1} u_i R^i_{(l)}$
1 for $j = 0 \rightarrow \frac{n}{k} - 1$ do
2     $t \leftarrow 0$
3     for $i = 0 \rightarrow k - 1$ do
4         if $u_i \neq 0$ then $t \leftarrow t + u_i R^i_{(l)}$;
5     $t \leftarrow t \cdot R^{j\cdot k}_{(l)}$
6     $P^{(j)}_l \leftarrow t$
```

- 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. ρ^k, will be sufficiently large that they can be ignored during the additions. Then, the total number of I/O transactions is $\frac{n}{k} \cdot (1 - \rho^k) + 1$, where $\frac{n}{k}$ represents the number of windows and plus 1 is for the output term.

Algorithm 3. NTT Encryption

```
Input: $P = \{P^{(0)}_l, P^{(1)}_l, \cdot, P^{(\frac{n}{k}-1)}_l\}$
Output: $R_l = 2\sum_0^{\frac{n}{k}-1} P^{(i)}{}_l + M_{(l)}$
1 $t \leftarrow 0$;
2 for $j = 0 \rightarrow \frac{n}{k} - 2$ do
3     if $P^j \neq 0$ then $t \leftarrow t + P^{(j)}_l + P^{(j)}_l$;
4 if $P^{(j)} \neq 0$ then $t \leftarrow t + P^{(\frac{n}{k}-1)}_l + P^{(\frac{n}{k}-1)}_l + M_{(l)}$;
5 $R_l \leftarrow t$
```

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 $u_i = \pm 1$ modular adder/subtracter is used to compute $t = t \pm R^i$. For the scaling operation, modular multiplier is enabled by the operation signal to compute $t = t \cdot R^{j\cdot k}$. 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 $2u(R) + M = 2\sum_0^{\frac{n}{k}-1} P^{(i)} + M$. 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 for 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, 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 $t \leftarrow 0$.
4) Check the value of u_i iteratively and skip if it is zero. In case of ± 1, β blocks of the powers r^i_l is loaded into the bus and operation signal is selected. Each arithmetic core is assigned with different blocks to evaluate $t = t + r^i_l$.

5) Iterate index i. Computation of the window sum is completed. To scale the sum, the necessary β blocks (powers of r_l^k) are loaded into the cache and the operation signal is set to enable multiplication. The term t is updated as: $t = t \cdot R_l^k$. Now t holds the result of a scaled window summation: $P_l^{(j)} = R_l^{j \cdot k} \sum_0^{k-1} u_i R_l^i$. Since there are β blocks, the window sum is evaluated for the β 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: $\{P^{(0)}, P^{(1)}, \ldots, P^{(\frac{n}{k}-1)}\}$.
9) Using the clear signal, clear all caches to 0.
10) Assert the double signal starting from $j = 0$, β blocks of $P^{(j)}$ is loaded if $P^{(j)} \neq 0$. This evaluates $t = t + 2 \cdot P_l^{(j)}$ up to $j = \frac{n}{k} - 1$. The add_bit signal is activated for the case $j = \frac{n}{k} - 1$. This will add the message bit m: $t = t + 2 \cdot P_l^{(\frac{n}{k}-1)} + m$.
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 $R = 2 \cdot \sum_{i=0}^{\frac{n}{k}-1} P^{(j)} + m$.

Parameter selection affects the efficiency of the architecture significantly. Since an r^i term is included if u_i is not 0, the probability distribution of $u_i = \{0, 1, -1\}$ is important to evaluate the timing. We select the window size as 64, and the probability is selected $\rho = 16/2{,}048$. Since we only have 16 non-zero values, we need to evaluate 16 of 32 windows in the worst case scenario.[6] If 16 of these windows are evaluated, it will take $16 \cdot 3N$ cycles. Addition of these 16 windows will take $17N$ clock cycles. Including the number of EPE's, INTT and Barrett Reduction operations, total cost of the operations will be $\frac{65N}{\beta} \cdot (1 + 10/\kappa) + 89N$ cycles.

5.3 Recryption

Recryption operation $\mathsf{Decrypt}_{\mathsf{SK}}(c)$ is evaluated as

$$\left\lceil \sum_{j\in[s]} \sum_{i\in[S]} \sigma_j(i) z_{j,i} \right\rfloor + \sum_{j\in[s], i\in[l]} \sigma_j(i)(y_{j,i}) \pmod 2.$$

The first summation has following form: $q_j = \sum_{a\in[l]} \beta_{j,a} (\sum_{b\in[l]} \beta_{j,b} z_{j,i(a,b)}) (\mathrm{mod}\ d)$. We can take advantage of 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. $B = \mathsf{NTT}(\beta)$, we can eliminate many costly large integer multiplications. The equation is rewritten as $q_j = \mathsf{INTT}(\sum_{a\in[l]} B_{(j,a)} (\sum_{b\in[l]} B_{(j,b)} z_{j,i(a,b)})) (\mathrm{mod}\ d)$ The new equation eliminates most of the NTT and INTT conversions. Only one inversion and one modular reduction is required at the end. Furthermore, we benefit by precomputing the $z_{j,i}$ terms 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

6. We divide the degree 2,048 into 64 windows of equal degree polynomials.

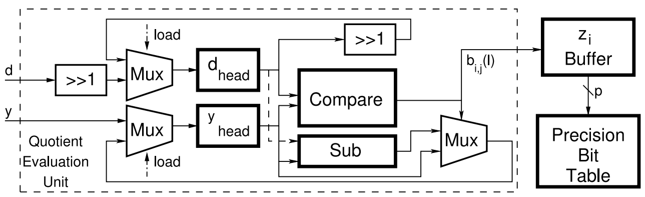

Fig. 7. Binary computation unit.

the $z_{j,i}$ for each block. We divide the above equation into four steps:

- Evaluation of the precision bits $z_{j,i}$.
- Sum of PKs $S_j = \sum_{a\in[l]} B_{(j,a)} (\sum_{b\in[l]} B_{(j,b)} z_{j,i(a,b)})$.
- INTT conversion and Barrett Reduction of S_j.
- Grade-School Addition.

First two steps are computed by Recryption Unit as shown in Fig. 9. Last two steps are computed by the Master Control Unit using the Large Integer Multiplier and the Barrett Reduction Unit.

5.3.1 *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 $z_{j,i} = \{b_{j,i}^{(0)}, \ldots, b_{j,i}^{(p'-1)}\} = \frac{y_{j,i}}{d}$. 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 Fig. 7. It consists of p' bit quotient evaluation units, a p'-bit buffer and a storage table that has size of $p' \cdot S$, 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 timing overhead will still remain small compared to the overall timing. The evaluation of precision bits, for values 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: if $y_{head} >= d_{head}$ then $b_{i,j}^{(l)} = 1$ else $b_{i,j}^{(l)} = 0$.
3) The precision bit $b_{i,j}^{(l)}$ 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 using the value of $b_{i,j}^{(l)}$ as: if $b_{i,j}^{(l)} == 1$ then $y_{head} = y_{head} - d_{head}$ else $y_{head} = y_{head}$.
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 bits are loaded to the ith row of the table from the buffer.

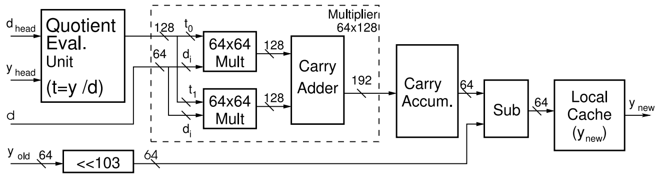

Fig. 8. Modular computation unit.

With 64-bit word size arithmetic, load 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 $(\frac{(p'+1)k_1}{64}+1)$ cycles.

Modular computation unit. 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 $y_{j,0} = c \cdot x_j \pmod{d}$. R is a special number equal to 2^{103}. This simplifies the multiplication into a simple shift operation. Also, the modular 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 $y_{i,j} = (y_{i-1,j} \ll 103) - t \cdot d$. In the equation coefficient t is at most 103 bits, so we are able to design a fast modular reduction unit that computes and multiplies the small coefficients with million-bit numbers. The design in Fig. 8 consist of a 103-bit quotient evaluation unit, a $64x128$-bit multiplier unit, a carry accumulate unit, a 103-bit shifter, a 64-bit subtracter and a local storage. Evaluation of $(y_{i-1,j} \ll 103) - t \cdot d$ is performed with the following steps:

1) 103-bit Quotient Evaluation Unit takes the first k_2 bits of the values $y_{j,i-1}$ and d.
2) 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, bits are loaded into a 128-bit buffer. The buffer will hold a zero-padded 103-bit t.
4) After computing t, we can evaluate $y_{j,i} = ys - t \cdot d = (y_{j,i-1} \ll 103) - t \cdot d$

The evaluation of 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 cycle. Also, the pipeline delay is small that we can neglect for timing computations. Therefore, transactions take $47{,}000$/bandwidth cycles to finish an evaluation, where *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 $(c \cdot w_i \pmod{d})$, so we eliminate extra NTT of ciphertext c by only converting it once[7] and using it in each public key multiplication. Furthermore, we store the public keys in NTT form and eliminate the conversion operations, which reduces the timing significantly. Therefore, we only perform digit multiplications, INTT conversions and Barrett Reduction. Then, completing the table for a single public key takes: $\tau = 93.5N + \left(\frac{104k_2+(p'+1)k_1}{64} + 2 + \frac{47000}{\text{bandwidth}}\right) \times S$ cycles. Using the same units for other public keys adds a factor of s to the overall timing, i.e. $s \cdot \tau + 16\ N$. 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 to achieve a speedup.

5.3.2 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 β's in NTT form to eliminate the conversions and rewrite the equation as: $s_j = \sum_{a\in[l]} B_{(j,a)} (\sum_{b\in[l]} B_{(j,b)} z_{j,i(a,b)})$ Since $z_{j,i(a,b)}$ is a p'-bit value, it is denoted as $z_{j,i} = \{b_{j,i}^{(p'-1)}, \dots, b_{j,i}^{(1)}, b_{j,i}^{(0)}\}$.[8] Then, s_j turns into a p' sized array that each bit computation is performed separately. By denoting $s_j = \{\tilde{s}_j^{(p'-1)}, \dots, \tilde{s}_j^{(1)}, \tilde{s}_j^{(0)}\}$, we can expand the equations as

$$\tilde{s}_j^{(k)} = \sum_{a\in[l]} B_{(j,a)} \left(\sum_{b\in[l]} B_{(j,b)} b_{(j,i)}^{(k)} \right),$$

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 $\tilde{s}_j^{(k)}$ 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 Fig. 9 consists of the RPE Arrays, the Precision Bits Table[9] and a Control Unit. In the following we give design details of the units.

Recryption processing element. The design of the RPE is illustrated in Fig. 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 up_c and low_c and has size of κ-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 2).

7. Only adds an initial 16 N clock cycles in overall operation.

8. The values a, b are omitted for simplicity.

9. The table formed in Evaluating Precision Bits.

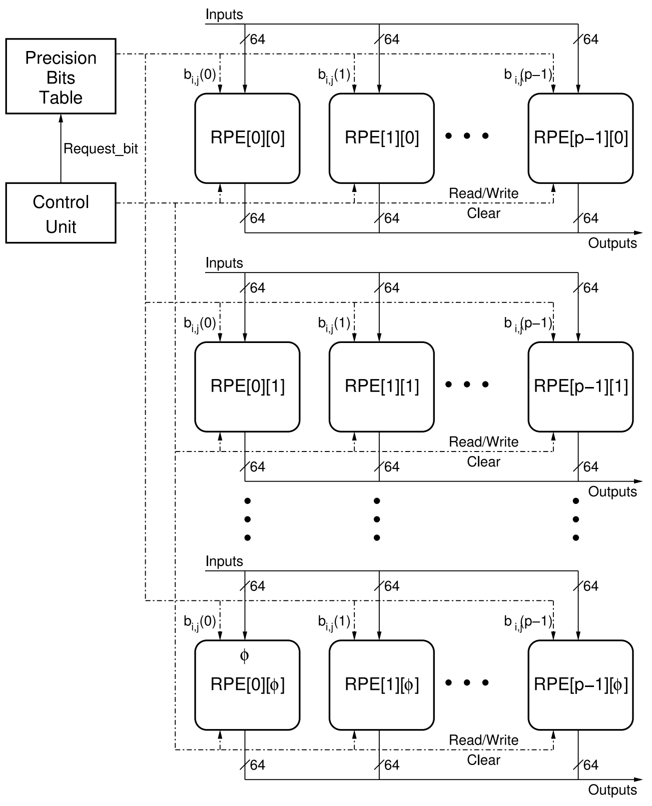

Fig. 9. Recryption architecture.

```
Algorithm 4. Recryption Algorithm
Input: $B_j = \{B_{j,l-1)}, \ldots, B_{j,1)}, B_{j,0)}\}, b^{(k)}_{(j,i)} \in$
        $\{b^{(p'-1)}_{(j,i)}, \ldots, b^{(1)}_{(j,i)}, b^{(0)}_{(j,i)}\}$
Output: $s_j = \sum_a B_{j,a)}(\sum_b B_{j,b)} \cdot b^{(k)}_{j,i})$
1  low_c $= 0$
2  for $a = 0 \rightarrow l-1$ do
3      up_c $= 0$
4      for $b = a+1 \rightarrow l-1$ do
5          if $b^{(k)}_{j,i} == 1$ then
6              up_c = up_c $+ B_{j,b)}$
7      low_c = low_c + up_c $\cdot B_{j,a)}$
```

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. Including the output transactions the operation requires $\frac{(S+x+p')\cdot N\cdot(1+11/\kappa)}{\phi} \cdot s$ clock cycles, in which factor of s comes from the public key number and ϕ comes from the number of RPE arrays. Given the FHE primitive parameters and x being 14, the timing is $\frac{531\cdot N\cdot(1+11/\kappa)}{\phi} \cdot s$.

5.3.3 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 INTT and Barret Reduction algorithms, the conversions will take $89N$ clock cycles. Having s public keys and p' precision

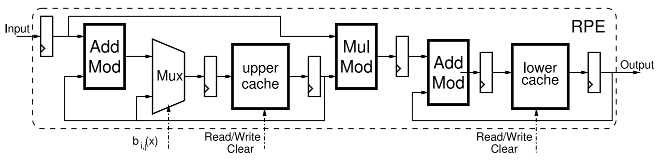

Fig. 10. Recryption processing element.

bits, the total operation will take $s \cdot p' \cdot 89N$ 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.

5.3.4 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 5.3.4 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 [23] approach to minimize the number of large multiplications. For this operation, we utilize a total of 78 large multiplication operations and a total of $33 + 33 + 32 + 27 + 14 = 139$ large addition operations. The large multiplication operations require modular reductions, so that the bit growth is prevented. In the evaluation of C, we use two multiplications followed by additions, so the multiplications is reduced after the additions. This reduce the Barrett Reduction operation by half. Then, the total timing is equal to $78 \cdot 52.5N + 39 \cdot 73N + 25N$, in which $25N$ is approximate cost of the additions.

6 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 1,333 MTps (Megatransfers per second) is selected for the Main Memory (RAM). The ratio between the memory and the main circuit frequency, i.e. $\beta = \phi = \frac{1333}{666} = 2$, results in I/O speed of two transactions (digits in our case) per clock cycle, which led us to set our EPE and RPE Array numbers as 2 each. Also, local cache sizes of the RPE, EPE 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 3. 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

TABLE 3
Arithmetic Operation Timings

Operation		# of Clock Cycles	Timing
Large Integer Multiplication		$52.5N$	7.75 msec
Barrett Reduction		$73N$	10.70 msec
Decryption		$109.5N$	16.16 msec
Encryption	EPE	$\frac{65N}{2} \cdot (1.039)$	4.98 msec
	INTT and Reduction	$89N$	13.12 msec
	Total		**18.10 msec**
Recryption	Evaluation of $z_{j,i}$	$(93N + 840\ S) \cdot s + 16\ N$	0.488 sec
	Sum of PK	$\frac{531 \cdot N}{2} \cdot (1.042) \cdot s$	0.612 sec
	INTT and Reduction	$s \cdot p \cdot 89N$	0.985 msec
	Grade School	6,967N	1.027 sec
	Total		**3.112 sec**

and a single EPE takes only about 26 percent of the entire Encryption operation. The recryption 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 recryption operation to 3.1 seconds. For the initial evaluation of $z_{j,i}$ and sum of PKs, we can utilize same elements. Since they are independent, the timing of the first two steps can be reduced to $1.075/\lambda$, in which λ 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 λ. For the last two steps of the operation, we can reduce the timing by adding more LARGE INTEGER MULTIPLIER units. Using κ multipliers, the latency of the INTT and Reduction operation can be reduced by a factor of λ and the delay of the Wallace-Tree can be reduced by close to a factor of λ. In multiplication, I/O transactions take 20 percent of the total operation. Therefore, with five multipliers we can perform multiple operations without increasing the bandwidth.

The design is synthesized with local cache sizes of 256, 128 and 64 digits and the area results are shown in Table 4. 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 DECRYPTION 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 26.5 million gates for cache and 0.2 million gates for $m = 4$ arithmetic units. By increasing the number of multipliers m we may further reduce the execution time. The time for the large integer multiplication $(\frac{158N}{m} + 13N)$ is tabulated for various m in Table 5. Note that the time-area product is optimal for $m = 64$ improving multiplication speed by 3.4 times while the area is increased by only 11 percent. We estimate a two times speedup in Recryption.

TABLE 4
Area of Hardware Blocks (Millions of Gates)

	64-digit Cache	128-digit Cache	256-digit Cache
Large Integer Mul.	26.7	26.7	26.7
Barrett Reduction	0.0209	0.0356	0.0647
Decryption	0.0002	0.0002	0.0002
Encryption	0.1047	0.1360	0.2060
Recryption	0.6740	0.7741	1.1770

7 COMPARISON

We presented the first hardware implementation of a fully homomorphic encryption system with the goal of exploring the limits of the GH-FHE scheme in hardware. Hence a direct comparison with other FHE implementations is not possible. While a comparing to software implementations would not be fair, we find it useful to summarize these results alongside ours in Table 6.

In LARGE INTEGER MULTIPLICATION, the time performance of our design is close to that of the Xeon software implementation and $\sim$10 times slower than GPU implementation. Decryption is 20 percent 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 critical 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 much lower gate count of less than 30 million equivalent gates. In contrast, the NVidia GPU in [24] 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. In Table 6 (bottom) we normalize the timings with the clock rates of the chips. In the most critical primitive, i.e. Recrypt, our implementation requires fewer than half the clock cycles compared to GPU and 46.6 times fewer clock cycles compared to Xeon implementations at a

TABLE 5
Time-Area Trade-Off

m	4	8	16	32	64	128
Time (in N)	52.5	32.7	22.8	17.9	15.4	14.2
Area	26.7	26.9	27.3	28.1	29.7	32.9
Time $\times$ Area	**1401**	**879.6**	**622.4**	**502.9**	**457.3**	**1,817**

TABLE 6
Times in MSEC (Top) and in Million Cycles (Bottom)

	Multiplication	Decrypt	Encrypt	Recrypt
Ours	7.750	16.1	18.1	3,100
GPU [24]	0.765	2.5	220	4,200
Xeon [8]	6.667	20.0	1,800	32,000
Ours	5.1	10.7	12	2,000
GPU [24]	0.8	2.8	253	4,800
Xeon [8]	20	60	5,400	96,000

much lower footprint. Our implementation can benefit at least a few times speedup, if synthesized with a smaller technology than 90 nm like Xeon and GPU processors (40-45 nm).

8 Conclusion

In this work we took initial steps to remedy the efficiency bottleneck of FHE schemes by introducing the first custom FHE architecture. For this we introduced a novel large integer modular multiplier design realizing the Shönhage Strassen algorithm and Barrett's reduction in hardware. 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.

Acknowledgments

This work was supported in part by NSF Awards #1117590 and #1319130. Funding was also provided by The Scientific and Technological Research Council of Turkey, project number 113C019.

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, 2009, pp. 169–178.
[2] R. Rivest, L. Adleman, and M. Dertouzos, "On data banks and privacy homomorphisms," in *Foundations of Secure Computation*. New York, NY, USA: Academic, 1978, pp. 169–177.
[3] D. Cousins, K. Rohloff, C. Peikert, and R. Schantz, "An update on Sipher (scalable implementation of primitives for homomorphic encryption)–FPGA implementation using simulink," in *Proc. IEEE Conf. High Perform. Extreme Comput.*, Sep. 2012, pp. 1–5.
[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," in *Proc. 3rd Innovations Theoretical Comput. Sci. Conf. (ITCS'12)*, New York, NY, USA: ACM, 2012, pp. 309–325.
[5] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *Proc. Found. Comput. Sci.*, Oct. 2011, pp. 97–106.
[6] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-LWE and security for key dependent messages," in *Proc. Adv. Cryptology*, 2011, vol. 6841, pp. 505–524.
[7] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proc. 44th Annu. ACM Symp. Theory Comput.*, 2012, pp. 1219–1234.
[8] C. Gentry and S. Halevi, " Implementing Gentry's fully-homomorphic encryption scheme," in *Proc. Adv. Cryptology*, 2011, vol. 6632, pp. 129–148.
[9] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," in *Proc. USSR Acad. Sci. 145*, 1962, pp. 293–294.
[10] A. Schnhage and V. Strassen, "Schnelle multiplikation groer zahlen," *Computing*, vol. 7, no. 3–4, pp. 281–292, 1971.
[11] L. C. C. García, "Can Schönhage multiplication speed up the RSA decryption or encryption?" in *Proc. Conf. Cryptology—Moravia-Crypt*, 2005, http:/www.cdc.informatik.tu-darmstadt.de/reports/reports/xaschonh.pdf
[12] S. Craven, C. Patterson, and P. Athanas, "Super-sized multiplies: How do FPGAS fare in extended digit multipliers," presented at the 7th Int. Conf. Military Aerospace Programm. Logic Devices, Washington, DC, USA, 2004.
[13] S. Yazaki and K. Abe, "An optimum design of FFT multi-digit multiplier and its VLSI," *Bull. Univ. Electro-Commun.*, vol. 18, no. 1, pp. 39–45, 2006.
[14] K. Kalach and J. David, "Hardware implementation of large number multiplication by FFT with modular arithmetic," in *Proc. IEEE-NEWCAS Conf.*, Jun. 2005, pp. 267–270.
[15] J. V. Z. Gathen and J. Gerhard, *Modern Computer Algebra*, 2nd ed, New York, NY, USA: Cambridge Univ. Press, 2003.
[16] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.
[17] P. Barrett, " Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Proc. Adv. Cryptology*, 1987, vol. 263, pp. 311–323.
[18] N. Emmart and C. Weems, "High precision integer multiplication with a GPU," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, May 2011, pp. 1781–1787.
[19] N. Emmart and C. Weems, "High precision integer addition, subtraction and multiplication with a graphics processing unit," *Parallel Process. Lett.*, vol. 20, no. 04, pp. 293–306, 2010.
[20] J. A. Solinas, "Generalized Mersenne numbers," Dept. CO, Univ. Waterloo, Waterloo, ON, CA, Tech. Rep. CORR-39, 1999.
[21] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford Univ., Stanford, CA, USA, 2009.
[22] I. Munro and M. Paterson,"Optimal algorithms for parallel polynomial evaluation," *J. Comput. Syst. Sci.*, vol. 7, no. 2, pp. 189–198, 1973.
[23] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electron. Comput.*, vol. EC-13, no. 1, pp. 14–17, Feb. 1964.
[24] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *Proc. High Perform. Extreme Comput.*, Sep. 2012, pp. 1–5.

Yarkın Doröz received the BSc degree in electronics engineering in 2009 and the MSc degree in computer science in 2011 from Sabanci University. Currently, he is working toward the PhD degree in electrical and computer engineering at Worcester Polytechnic Institute. His research is focused on developing hardware/software designs for Fully Homomorphic Encryption Schemes.

Erdinç Öztürk received the BS degree in microelectronics from Sabanci University in 2003, the MS degree in electrical engineering in 2005, and the PhD degree in electrical and computer engineering in 2009 from Worcester Polytechnic Institute. His research field was cryptographic hardware design and he focused on efficient Identity Based Encryption implementations. After receiving the PhD degree, he worked at Intel in Massachusetts for almost 5 years as a hardware engineer, before joining Istanbul Commerce University as an assistant professor.

Berk Sunar received the BSc degree in electrical and electronics engineering from Middle East Technical University in 1995 and the PhD degree in electrical and computer engineering from Oregon State University in 1998. After briefly working as a member of the research faculty at Oregon State University, he joined Worcester Polytechnic Institute faculty. He is currently heading the Vernam Applied Cryptography Group. He received the prestigious National Science Foundation Young Faculty Early CAREER award in 2002 and IBM Research Pat Goldberg Memorial Best Paper Award in 2007.