This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

A Custom Accelerator for Homomorphic Encryption Applications

Erdinç Öztürk, Yarkın Doröz, Erkay Savaş and Berk Sunar

Abstract—After the introduction of first fully homomorphic encryption scheme in 2009, numerous research work has been published aiming at making fully homomorphic encryption practical for daily use. The first fully functional scheme and a few others that have been introduced has been proven difficult to be utilized in practical applications, due to efficiency reasons. Here, we propose a custom hardware accelerator, which is optimized for a class of reconfigurable logic, for López-Alt, Tromer and Vaikuntanathan's somewhat homomorphic encryption based schemes. Our design is working as a co-processor which enables the operating system to offload the most compute–heavy operations to this specialized hardware. The core of our design is an efficient hardware implementation of a polynomial multiplier as it is the most compute–heavy operation of our target scheme. The presented architecture can compute the product of very–large polynomials in under 6.25 ms which is 102 times faster than its software implementation. In case of accelerating homomorphic applications; we estimate the per block homomorphic AES as 442 ms which is 28.5 and 17 times faster than the CPU and GPU implementations, respectively. In evaluation of Prince block cipher homomorphically, we estimate the performance as 52 ms which is 66 times faster than the CPU implementation.

Index Terms—Somewhat homomorphic encryption, NTT multiplication, FPGA, Accelerator for homomorphic encryption

1 Introduction

Fully homomorphic encryption (FHE) schemes are introduced to enable blinded server–side computations. Although the idea was proposed in 1978 [1], first working FHE scheme was constructed by Gentry in 2009 [2], [3]. Despite heavy efforts to develop practical implementations of this scheme since its construction, such as rendering expensive bootstrapping evaluations obsolete [4] and parallel processing through *batching* of multiple data bits into a ciphertext [5], [6], [7], it was not possible to realize an efficient hardware or software implementation. For instance, an implementation by Gentry et al. [8] homomorphically evaluates the AES circuit in about 36 hours resulting in an amortized per block evaluation time of 5 minutes. In [9] and later in [10] Doröz et al. present an architecture for ASIC that implements a full set of FHE primitives including bootstrapping. Another implementation by Doröz et al. [11] manages to evaluate AES roughly an order of magnitude faster than [8]. Doröz et al. implemented Prince encryption algorithm which is more suitable for homomorphic encryption with low depth circuit and less computation complexity in [12] with a runtime of 57 minutes. Cousins et al. report the first reconfigurable logic implementations in [13], [14], in which Matlab Simulink was used to design the FHE primitives. This was followed by further FPGA implementations [15], [16], [17], [18]. Cao et al. [16] proposed a number theoretical transform (NTT)-based large integer multiplier combined with Barrett reduction to alleviate the multiplication and modular reduction bottlenecks required in many FHE schemes. Parallel to these efforts on Gentry scheme, new FHE schemes, suchs as lattice-based [19], [20], [21], integer-based [22], [23], [24] and learning-with-errors (LWE) or (ring) learning with errors ((R)LWE) based encryption [25], [26], [27] schemes, were introduced. The encryption step in the proposed integer based FHE schemes by Coron et al. [23], [24] were designed and implemented on a Xilinx Virtex-7 FPGA. The synthesis results show speed up factors of over 40 over existing software implementations of this encryption step [16].

It is clear that these implementations are not practical FHE solutions. Exhausting efforts targeting software and hardware implementations, researchers began to investigate GPUS as alternative platforms for FHE applications. Using GPUs, Wang et al. [28] managed to accelerate the recryption primitive of Gentry and Halevi [26] by roughly an order of magnitude. In [18], Wang et al. present an optimized version of their result [17], which achieves speed–up factors of 174, 7.6 and 13.5 for encryption, decryption and the recryption operations on an NVIDIA GTX 690, respectively, when compared to results of the implementation of Gentry and Halevi's FHE scheme [19] that runs on an Intel Core i7 3770K machine. A more recent work by Dai et al. [29], [30] reports GPU acceleration for NTRU based FHE evaluating Prince and AES block ciphers, with 103 times and 7.6 times speedup values, respectively, over an Intel Xeon software implementation.

In Table 1, we summarize previous FHE implementations. As can be seen from Table 1, FPGA and ASIC implementations are promising for significant performance

Y. Yarkın Doröz and B. Sunar are with Worcester Polytechnic Institute, {ydoroz,sunar}@wpi.edu.
E. Öztürk is with Istanbul Commerce University, eozturk@ticaret.edu.tr
E. Savaş is with Sabanci University erkays@sabanciuniv.edu

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

TABLE 1
Overview of specialized FHE Implementations. GH-FHE: Gentry & Halevi's FHE scheme; CMNT-FHE: Coron et al.'s FHE schemes [23], [24] [19]; NTRU based FHE, e.g. [31], [32]

DESIGN	SCHEME	PLATFORM	PERFORMANCE
CPU			
AES [8]	BGV-FHE	2.0 GHz Intel Xeon	5 min / AES block
AES [11]	NTRU-FHE	2.9 GHz Intel Xeon	55 s / AES block
Full FHE [33]	NTRU-FHE	2.1 GHz Intel Xeon	275 s / per bootst.
Prince [12]	NTRU-FHE	3.5 GHz Intel i7	3.3 s / Prince Block
GPU			
NTT mul / reduction [28]	GH-FHE	NVIDIA C250 GPU	0.765 ms
NTT mul [28]	GH-FHE	NVIDIA GTX 690	0.583 ms
AES [29]	NTRU-FHE	NVIDIA GTX 690	7 s / AES block
Prince [29]	NTRU-FHE	NVIDIA GTX 690	1.28 s / Prince block
Prince [30]	NTRU-FHE	NVIDIA GTX 690	32 ms / Prince block
FPGA			
NTT transform [17]	GH-FHE	Stratix V FPGA	0.125 ms
NTT modmul / enc. [16]	CMNT-FHE	Xilinx Virtex7 FPGA	13 ms / enc.
ASIC			
NTT modmul [9]	GH-FHE	90nm TSMC	2.09 s
Full FHE [10]	GH-FHE	90nm TSMC	3.1 s / recrypt

gains. Much of the development so far has focused on the Gentry-Halevi FHE [19], which intrinsically works with very large integers. Therefore, a good number of research work focused on developing FFT/NTT based large integer multipliers [9], [28], [28], [10]. Currently, the only full-fledged (with bootstrapping) FHE hardware implementation is the one reported by Doröz et al. [10], which also implements the Gentry-Halevi FHE. At this time, there is a lack of hardware implementations of the more recently proposed FHE schemes, i.e. Coron et al.'s FHE schemes [23], [24], BGV-style FHE schemes [4], [19] and NTRU based FHE, e.g. [31], [32]. We, therefore, focus on providing hardware acceleration support for one particular family of FHE's: NTRU-based FHE schemes, where arithmetic with very large polynomials (both in degree and coefficient size) is crucial for performance.

Our Contribution. In this work, we present an FPGA architecture to accelerate NTRU based FHE schemes. Our architecture may be considered as a proof-of-concept implementation of an external FHE accelerator that will speed up homomorphic evaluations taking place on a CPU. Specifically, the architecture we present manages to evaluate a full polynomial multiplication efficiently, for large degrees 2^{14} and 2^{15}, by utilizing a number theoretical transform based approach. Using this FPGA core we can evaluate multiplication of 2^{14} degree polynomial 72 times faster than a CPU and 25.7 times faster than a GPU implementations. In case of 2^{15} degree polynomials, it can evaluate the multiplications 102 and 36.5 times faster than a CPU and a GPU, respectively. Furthermore, by facilitating efficient exchange using a PCI Express connection, we evaluate the overhead incurred in a sustained homomorphic computations of deep circuits. For instance, by including data transfer clock cycles, our hardware can evaluate a full 10 round AES circuit in under 440 ms per block. In case of Prince, our hardware achieves amortized run time of 52 ms per block.

2 BACKGROUND

In this section we briefly outline the primitives of the López-Alt, Tromer and Vaikuntanathan's fully homomorphic encryption based schemes, and later discuss the arithmetic operations that will be necessary in its hardware realization.

2.1 LTV-Based Fully Homomorphic Encryption

While the arithmetic and homomorphic properties of NTRU have been long known by the research community, a full-fledged fully homomorphic version was proposed only very recently in 2012 by López-Alt, Tromer and Vaikuntanathan (LTV) [31]. The LTV scheme is based on a variant of NTRU introduced earlier by Stehlé and Steinfeld [32]. The LTV scheme uses a new operation called relinearization and existing technique modulus switching for noise control. While the LTV scheme can support homomorphic evaluation in a multi-key setting where each participant is issued their own keys, here we focus only on the single user case for brevity.

The primitives of the LTV scheme operate on polynomials in $\mathrm{R}_q = \mathbb{Z}_q[x]/\langle x^N + 1\rangle$, i.e. with degree N, where the coefficients are processed using a prime modulus q. In the scheme an error distribution function χ – a truncated discrete Gaussian distribution – is used to sample random, small B-bounded polynomials. The scheme consists of four primitive functions:

KeyGen. We select decreasing sequence of primes $q_0 > q_1 > \cdots > q_d$ for each level. We sample $g^{(i)}$ and $u^{(i)}$ from χ, compute secret keys $f^{(i)} = 2u^{(i)} + 1$ and public keys $h^{(i)} = 2g^{(i)}(f^{(i)})^{-1}$ for each level. Later we create evaluation keys for each level: $\zeta_\tau^{(i)}(x) = h^{(i)}s_\tau^{(i)} + 2e_\tau^{(i)} + 2^\tau(f^{(i-1)})^2$, where $\{s_\tau^{(i)}, e_\tau^{(i)}\} \in \chi$ and $\tau = [0, \lfloor \log q_i \rfloor]$.

Encrypt. To encrypt a bit b for the i^{th} level we compute: $c^{(i)} = h^{(i)}s + 2e + b$ where $\{s, e\} \in \chi$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

Decrypt. In order to compute the decryption of a value for specific level i we compute: $m = c^{(i)} f^{(i)} \pmod 2$.

Evaluate. The multiplication and addition of ciphertexts correspond to XOR and AND operations, respectively. The multiplication operation creates a significant noise, which is handled with using relinearization and modulus switching. The relinearization computes $\tilde{c}^{(i)}(x) = \sum_\tau \zeta_\tau^{(i)}(x)\tilde{c}_\tau^{(i-1)}(x)$, where $\tilde{c}_\tau^{(i-1)}(x)$ are 1-bounded polynomials that are equal to $\tilde{c}^{(i-1)}(x) = \sum_\tau 2^\tau \tilde{c}_\tau^{(i-1)}(x)$. In case of modulus switching, we do the computation $\tilde{c}^{(i)}(x) = \lfloor \frac{q_i}{q_{i-1}} \tilde{c}^{(i)}(x) \rceil_2$ to cut the noise level by $\log(q_i/q_{i-1})$ bits. The operation $\lfloor \cdot \rceil_2$ is matching the parity bits.

2.2 Arithmetic Operations

To implement the costly large polynomial multiplication and relinearization operations we follow the strategy of Dai et al. [29]. For instance, in the case of polynomial multiplication we first convert the input polynomials using the Chinese Remainder Theorem (CRT) into a series of polynomials of the same degree, but with much smaller word-sized coefficients. Then, pairwise product of these polynomials is computed efficiently using Number Theoretical Transform (NTT)-based multiplier as explained in subsequent sections. Finally, the resulting polynomial is recovered from the partial products by the application of the inverse CRT (ICRT) operation.

2.2.1 CRT Conversion

As an initial optimization we convert all operand polynomials with large coefficients into many polynomials with small coefficients by a direct application of the Chinese Remainder Theorem (CRT) on the coefficients of the polynomials: $\texttt{CRT} : \mathsf{A}_j \longrightarrow \{\mathsf{A}_j \bmod p_0, \mathsf{A}_j \bmod p_1, \cdots, \mathsf{A}_j \bmod p_{l-1}\}$, where p_i's are selected small primes, l is the number of these small primes, and A_j is a coefficient of the original polynomial. Through CRT conversion we obtain a set of polynomials $\{A^{(0)}(x), A^{(1)}(x), \cdots, A^{(l-1)}(x)\}$ where $A^{(i)}(x) \in \mathrm{R}_{p_i} = \mathbb{Z}_{p_i}[x]/\Phi(x)$. These small coefficient polynomials provide us the advantage of performing arithmetic operations on polynomials in a faster and efficient manner. Any arithmetic operation is performed between the reduced polynomials with the same superscripts, e.g. the product of $A(x) \cdot B(x)$ is going to be $\{A^{(0)}(x) \cdot B^{(0)}(x), A^{(1)}(x) \cdot B^{(1)}(x), \cdots, A^{(l-1)}(x) \cdot B^{(l-1)}\}$. A side benefit of using the CRT is that it allows us to accommodate the change in the coefficient size during the levels of evaluation, thereby yielding more flexibility. When the circuit evaluation level increases, since q_i gets smaller, we can simply decrease the number of primes l. Therefore, both multiplication and relinearization become faster as we proceed through the levels of evaluation. After the operations are completed, a coefficient of the resulting polynomial, $\mathsf{C}(x)$ is computed by the Inverse CRT (ICRT):

$$\mathsf{ICRT}(\mathsf{C}_j) = \sum_{i=0}^{l-1} \left(\frac{q}{p_i}\right) \cdot \left(\left(\frac{q}{p_i}\right)^{-1} \cdot C_j^{(i)} \mod p_i\right) \mod q,$$

where $q = \prod_{i=0}^{i=l-1} p_i$. Note that we will drop the superscript notation used for the reduced polynomials by the CRT for clarity of writing since we will deal with mostly the reduced polynomials henceforth in this paper.

2.2.2 Polynomial Multiplication

The fundamental operation in the LTV scheme, during which the majority of execution time is spent, is the multiplication of two polynomials of very large degrees. More specifically, we need to multiply two polynomials, $A(x)$ and $B(x)$ over the ring of polynomials $\mathbb{Z}_p[x]/(\Phi(x))$, where p is an odd integer and degree of $\Phi(x)$ is $N = 2^n$. Namely, we have $A(x) = \sum_{i=0}^{N-1} A_i x^i$ and $B(x) = \sum_{i=0}^{N-1} B_i x^i$. The classical multiplication techniques such as the schoolbook algorithm have quadratic complexity in the asymptotic case, namely $\mathcal{O}(N^2)$. In general, the polynomial multiplication requires about N^2 multiplications and additions and subtractions of similar numbers in $\mathbb{Z}_p$. Other classical techniques such as Karatsuba algorithm [34] can be utilized to reduce the complexity of the polynomial multiplication to $\mathcal{O}(N^{\log_2 3})$. Nevertheless, the classical techniques do not yield feasible solutions for large N.The NTT-based multiplication achieves a quasi-linear complexity $\mathcal{O}(N \log N \log\log N)$ for polynomial multiplication, which is especially beneficial for large values of N.

The NTT can essentially be considered as a Discrete Fourier Transform defined over the ring of polynomials $\mathbb{Z}_p[x]/(\Phi(x))$. Simply speaking, the forward NTT takes a polynomial $A(x)$ of degree $N-1$ over $\mathbb{Z}_p[x]/(\Phi(x))$ and yields another polynomial of the form $\mathcal{A}(x) = \sum_{i=0}^{N-1} \mathcal{A}_i x^i$. The coefficients $\mathcal{A}_i \in \mathbb{Z}_p$ are defined as $\mathcal{A}_i = \sum_{j=0}^{N-1} A_j \cdot w^{ij} \bmod p$, where $w \in \mathbb{Z}_p$ is referred as the twiddle factor. For the twiddle factor we have $w^N = \bmod p$ and $\forall i < N \; w^i \neq 1 \bmod p$. The inverse transform can be computed in a similar manner $A_i = N^{-1} \cdot \sum_{j=0}^{N-1} \mathcal{A}_j \cdot w^{-ij} \bmod p$. Once the NTT is applied to two polynomials, $A(x)$ and $B(x)$, their multiplication can be performed using coefficient-wise multiplication over $\mathcal{A}_i$ and $\mathcal{B}_i$ in $\mathbb{Z}_p$; namely we compute $\mathcal{A}_i \times \mathcal{B}_i \bmod p$ for $i = 0, 1, \ldots N-1$. Then, the inverse NTT (INTT) is used to retrieve the resulting polynomial $C(x) = INTT(NTT(A(x)) \odot NTT(B(x)))$, where the symbol $\odot$ denotes the coefficient-wise multiplication of $\mathcal{A}(x)$ and $\mathcal{B}(x)$ in $\mathbb{Z}_p$. Note that the polynomial multiplication yields a polynomial $C(x)$ of degree $2N-1$. Therefore, before applying the forward NTT, $A(x)$ and $B(x)$ should be padded with N zeros to have exactly $2N$ coefficients. Consequently, for the twiddle factor we should have $w^{2N} = 1 \bmod p$ and $\forall i < 2N \; w^i \neq 1 \bmod p$.

Cooley–Tukey algorithm [35], described in Algorithm 1, is a very efficient method of computing forward

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

and inverse NTT. The permutation in Step 2 of Algorithm 1 is implemented by simply reversing the indexes of the coefficients of A_i. The new position of the coefficient A_i where $i = (i_n, i_{n-1}, \ldots, i_1, i_0)$ is determined by reversing the bits of i, namely $(i_0, i_1, \ldots, i_{n-1}, i_n)$. For example, the new position of A_{12} when $N = 16$ is 3. The inverse NTT can also be computed with Algorithm 1, using the inverse of the twiddle factor, i.e. $w^{-1} \bmod p$. Therefore, we can use the same circuit for both forward and inverse NTT. Note that the NTT-based multiplication technique returns a polynomial of degree $2N-1$, which should be reduced to a polynomial of degree $N-1$ by diving it by $\Phi(x)$ and keeping the remainder of the division operation. When the reduction polynomial $\Phi(x)$ is of a special form such as $x^N + 1$, the NTT is known as Fermat Theoretic Transform (FTT) [36] and the polynomial reduction can be performed easily as described in [37] and [38].

ALGORITHM 1: Iterative Version of Number Theoretic Transformation

input : $A(x) = A_0 + A_1 x + \ldots + A_{N-1}x^{N-1}$, $N = 2^n$, and w
output: $\mathcal{A}(x) = \mathcal{A}_0 + \mathcal{A}_1 x + \ldots + \mathcal{A}_{N-1}x^{N-1}$

for $i = N$ **to** $2N-1$ **do**
  $A_i = 0$;
end
$(\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_{2N-1}) \leftarrow \text{Permutation}(A_0, A_1, \ldots, A_{2N-1})$;
for $M = 2$ **to** $2N$ **do**
  for $j = 0$ **to** $2N-1$ **do**
    for $i = 0$ **to** $\frac{M}{2} - 1$ **do**
      $x \leftarrow i \times \frac{2N}{M}$;
      $\mathcal{I} \leftarrow j + i$;
      $\mathcal{J} \leftarrow j + i + \frac{M}{2}$;
      $\mathcal{A}[\mathcal{I}] \leftarrow \mathcal{A}[\mathcal{I}] + w^{x \bmod 2N} \times \mathcal{A}[\mathcal{J}] \bmod p$;
      $\mathcal{A}[\mathcal{J}] \leftarrow \mathcal{A}[\mathcal{I}] - w^{x \bmod 2N} \times \mathcal{A}[\mathcal{J}] \bmod p$;
      $i \leftarrow i + 1$;
    end
    $j \leftarrow j + M$;
  end
  $M \leftarrow M \times 2$;
end

2.2.3 Relinearization

Relinearization takes a ciphertext and set of evaluation keys ($EK_{i,j}$) as inputs, where $i \in [0, l-1]$ and $j \in [0, \lceil \log(q)/r \rceil - 1]$, l is the number of small prime numbers and r is the level index. Algorithm 2 describes relinearization as implemented in this work. We precompute the CRT and NTT of the evaluations keys (since they are fixed) and in the computations we perform the multiplications and additions in the NTT domain. The result is evaluated by taking l INTT and one ICRT at the end. An r-bit windowed relinearization involves $\lceil \log(q)/r \rceil$ polynomial multiplications and additions, which are performed again in the NTT domain.

ALGORITHM 2: Relinearization with r bit windows

input : Polynomial c with $(n, \log(q))$
output: Polynomial d with $(2n, \log(nq\log(q)))$

$\{\tilde{c}_\tau\} = \mathsf{CRT}(c)$;
$\{\tilde{C}_\tau\} = \mathsf{NTT}(\{\tilde{c}_\tau\})$;
for $i = 0$ **to** $l-1$ **do**
  load $EK_{i,0}, EK_{i,1}, \cdots, EK_{i,\lceil \log(q)/r \rceil - 1}$;
  $\{D_i\} = \{\sum_{\tau=0}^{\lceil \log(q)/r \rceil - 1} \tilde{C}_\tau \cdot EK_{i,\tau}\}$;
end
$\{d_i\} = \mathsf{INTT}(\{D_i\})$;
$d = \mathsf{ICRT}(\{d_i\})$;

Since operand coefficients are kept in residue form, before relinearization we need to compute the inverse CRT of $\tilde{c}_\tau$.

3 Architecture Overview

3.1 Software/Hardware Interface

The performance of the NTRU based FHE scheme heavily depends on the speed of the large degree polynomial multiplication and relinearization operations. Since the relinearization operation is reduced to the computation of many polynomial multiplications, a fast large degree polynomial multiplication is the key to achieve a high performance in the NTRU-FHE scheme. Having a large degree N increases the computation requirements significantly, therefore a standalone software implementation on a general-purpose computing platform fails to provide a sufficient performance level for polynomial multiplications. The NTT-based polynomial multiplication algorithm is highly suitable for parallelization, which can lead to performance boost when implemented in hardware. On the other hand, the overall scheme is a complex design demanding prohibitively huge memory requirements (e.g., in LTV-AES [11] key requirements exceed 64-GB of memory). Therefore, a standalone architecture for SWHE fully implemented in hardware is not feasible to meet the requirements of the scheme.

In order to cope with the performance issues we designed the core NTT-based polynomial multiplication in hardware, where the polynomials have relatively small coefficients (i.e., 32-bit integers) to use it in more complicated polynomial multiplications and relinearization evaluations. The designed hardware is implemented in an FPGA device, which is connected to a PC with a high speed interface, e.g. PCI Express (PCIe). The PC handles simple and non-costly computations such as memory transactions, polynomial additions and etc. In case of a large polynomial multiplication or NTT conversion (in case of relinearization), the PC using the CRT technique, computes an array of polynomials whose coefficients are 32-bit integers from the input polynomials of much larger coefficients. The array of polynomials with small coefficients are sent in chucks to the FPGA via the high-speed PCIe bus. The FPGA computes the desired operation: polynomial multiplications or only NTT conversion. Later, the PC receives the resulting polynomials

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

from the FPGA and if necessary, i.e. before modulus switching or relinearization, evaluates the inverse-CRT to compute the result.

3.2 PCIe Interface

The PCIe is a serial bus standard used for high speed communication between devices which in our case are PC and the FPGA board. As the target FPGA board, we use Virtex-7 FPGA VC709 Connectivity Kit and can operate at 8 GT/s, per lane, per direction with each board having 8 lanes. The system is capable of sending the data packets in bursts. This allows us to achieve real time data transaction rate close to the given theoretical transaction rate as the packet sizes become larger.

3.3 Arithmetic Core Units

In order to achieve multiplication of two large degree polynomials, we designed hardware implementations for basic arithmetic building blocks to perform operations on the polynomial coefficients such as modular addition, modular subtraction and modular multiplication.

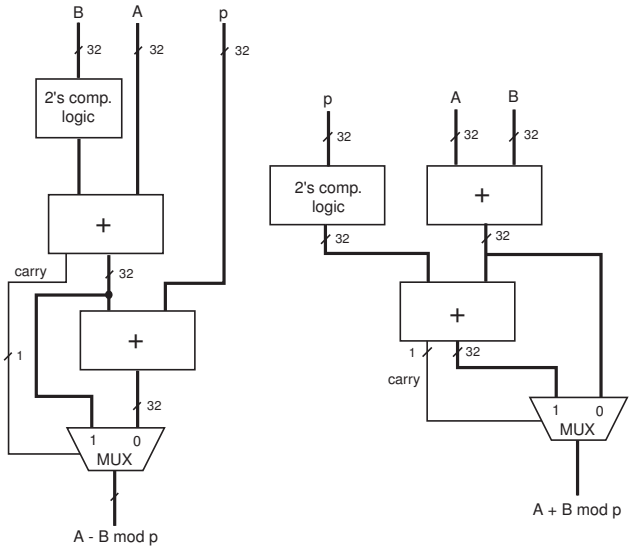
For compute–heavy operations using a large number of multiplication operations such as modular exponentiation and polynomial multiplication, it is a common practice, especially on word-oriented architectures, to perform partial reduction for the intermediate operations [39]. For example, when multiplying two 32–bit numbers with respect to a 32–bit modulus p, it is sufficient to achieve a result that is 32 bits in length, which can still be larger than the modulus p. This increases complexity of modular addition and modular subtraction operations because of the massive number of operations realized in a single clock cycle for multiplication of two polynomials of degree 2^{14} and 2^{15}. Therefore, we conclude that the most efficient method for the these modular operations is to achieve full modular reduction, and we design our building blocks to work with only fully reduced integers. Also, we base our design on an architecture to perform modular arithmetic operations for 32–bit numbers.

3.3.1 32-bit Modular Addition

The modular addition circuit, which is illustrated in Figure 1b, takes one clock cycle to perform one modular addition operation where operands A, B and the modulus p are all 32-bit integers and $A, B < p$. As noted before, it is guaranteed that the result will be less than the modulus p. Since the largest values of A and B are $p-1$, and thus the largest value of $A+B$ is $2p-2$, at most one final subtraction of the modulus p from $A+B$ will be sufficient to achieve full modular reduction after addition operation.

3.3.2 32-bit Modular Subtraction

The modular subtraction circuit, which is designed in a similar manner to modular addition circuit, is illustrated in Figure 1a. Similarly the subtraction unit is optimized to take one clock cycle to finish one modular subtraction operation on a target device. Since the largest values of A and B are $p-1$, and the smallest values of A and B are 0, the largest value of their subtraction can be $p-1$, and the smallest value can be $-p+1$, which indicates that one final addition of the modulus p will be sufficient to achieve full modular reduction after subtraction operation.



(a) Modular Subtracter (b) Modular Adder

Fig. 1. Modular Adder/Subtracter Circuits.

3.3.3 Integer Multiplication

The target FPGA device features many DSP units that are capable of performing very fast multiply and accumulate operations. A DSP unit takes three inputs A, B and C, which are 18 bits, 25 bits and 48 bits, respectively. A and B are multiplicand inputs, and C is the accumulate input. The output is a 48–bit integer, which can be defined as $D = A \times B + C$. Therefore, we can accumulate the results of many 18×25–bit multiplications without overflow. Since our operands are 32 bits in length, first we need to perform a full multiplication operation of 32–bit numbers. The operand lengths of the DSP units dictate that we need to perform four 16×16–bit multiplication operations to achieve a 32–bit multiplication operation. Utilizing four separate DSP slices, we could perform a 32–bit multiplication with 1 clock cycle throughput. However, this brings additional complexity to the hardware and because of the overall structure of the polynomial multiplication algorithm, 1–cycle throughput is not crucial for our design. Therefore, we decided to utilize a single DSP unit and perform the required multiplication operations to achieve a 32–bit multiplication operation on the same DSP unit. This results in a 4–cycle throughput as explained below.

In our design, however, we use Barrett's algorithm [40] for modular reduction, which requires 33×33–bit multiplication operations, for which the utilized method

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

ALGORITHM 3: 33×33–bit integer multiplication

input : $A = \{A_1, A_0\}$, $B = \{B_1, B_0\}$, where A_1, B_1 are high 17 bits and A_0, B_0 are low 16 bits of A and B, respectively
output: $C = A \times B$

1 $R1 \leftarrow A_0 \times B_0 + 0$;
2 $R2 \leftarrow A_0 \times B_1 + R1_H (R1_H = R1 >> 16)$;
3 $R3 \leftarrow A_1 \times B_0 + R2$;
4 $R4 \leftarrow A_1 \times B_0 + R3_H (R3_H = R3 >> 16)$;
5 $C \leftarrow \{R4, R3_L, R1_L\} (R1_L = R1 \& 0xFFFF, R3_L = R3 \& 0xFFFF)$;

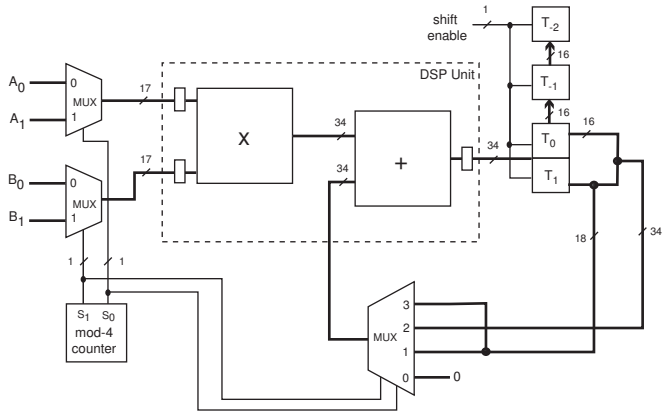


Fig. 2. Multiplier Circuit.

is described in Algorithm 3. Therefore, we use DSP slices to perform 17×17–bit integer multiplications at a time as illustrated in Figure 2, instead of 16×16–bit multiplications, where both operations have exactly the same complexity. To minimize critical path delays, we utilize the optional registers for the multiplicand inputs and the accumulate output ports of the DSP unit as shown in Figure 2. These registers increase the latency of a single 33×33-bit multiplication to 6 clock cycles. On the other hand, the throughput is still four clock cycles, which allows the multiplier unit to start a new multiplication every four clock cycles.

We use classical multiplication algorithm and accumulate the result of the previous multiplication immediately after a 17×17–bit multiplication operation. The result will be in the registers T_1, T_0, T_{-1}, T_{-2}. Note that the wire widths in Figure 2 indicate the sizes of the operands and the intermediate values in our application, not the actual widths of the corresponding wires in the DSP units.

3.3.4 32-bit Modular Multiplication

We use Barrett's modular reduction algorithm [40] to perform modular multiplication operations. The Montgomery reduction algorithm [41], which is a plausible alternative to the Barrett reduction, can also be used for modular multiplication of 32-bit integers. However, the Montgomery arithmetic requires transformations to and from the residue domain, which can lead to complications in the design. Therefore, we prefer using the Barrett's algorithm in our implementation to alleviate the mentioned complications in the design.

We use the algorithm adapted for 32-bit modular multiplication operations as illustrated in Algorithm 4. The comparison operation (and associated addition with 2^{33}) in Step 9 is not needed in hardware implementation, as it is equivalent to checking the carry output of addition of U and 2's complement of V after Step 8. More specifically, when the operation $U - V$ results in a negative number, the actual operation in hardware, where two's complement arithmetic is used, produces no carry. Consequently, if we use exactly the 33 bits of the result ignoring whether there is a carry or not, we will always obtain the correct result.

ALGORITHM 4: Barrett Modular Multiplication Algorithm for 32-bit Modulus

input : A, B, p, and T, where $A, B < p < 2^{32}$ and $T = \lfloor \frac{2^{64}}{p} \rfloor$
output: $C = A \times B \bmod p$

1 $X \leftarrow A \times B$;
2 $Q \leftarrow X >> 31$;
3 $R \leftarrow Q \times T$;
4 $S \leftarrow R >> 33$;
5 $Y \leftarrow S \times p$;
6 $U \leftarrow X \bmod 2^{33}$;
7 $V \leftarrow Y \bmod 2^{33}$;
8 $W \leftarrow U - V$;
9 **if** $W < 0$ **then**
  $W \leftarrow W + 2^{33}$;
  end
10 **if** $W - 2p > 0$ **then**
  $C \leftarrow W - 2p$;
  end
11 **else if** $W - p > 0$ **then**
  $C \leftarrow W - p$;
12 **else**
  $C \leftarrow W$;
  end

The subtraction $W \leftarrow U - V$ in Step 8 can be at most a 33-bit number, more precisely $3p - 1$ as explained in [42]. Therefore, two subtractions in Steps 10–11 can be necessary to obtain the final complete result at the end. As we want to finish Steps 10–11 in a single clock cycle, we perform both subtractions in the hardware implementations simultaneously, namely $W - 2p$ and $W - p$, and select the correct result using the carry bits of the subtraction results and a multiplexer as illustrated in Figure 3. If $W - 2p$ is positive, it is guaranteed that it is a number in the range $0 \leq W < p$, and we select this result as the output. However, if $W - 2p$ is a negative number and $W - p$ is a positive number, we select $W - p$ as the correct output. If both subtractions yield negative results, we select W as the output.

Our implementation of the Barrett algorithm, which is illustrated in Figure 3 takes 19 clock cycles to complete one modular multiplication of 32–bit integers whereas its throughput is four clock cycles. We will refer the first four clock cycles as the *warm up cycles* of the multiplier

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

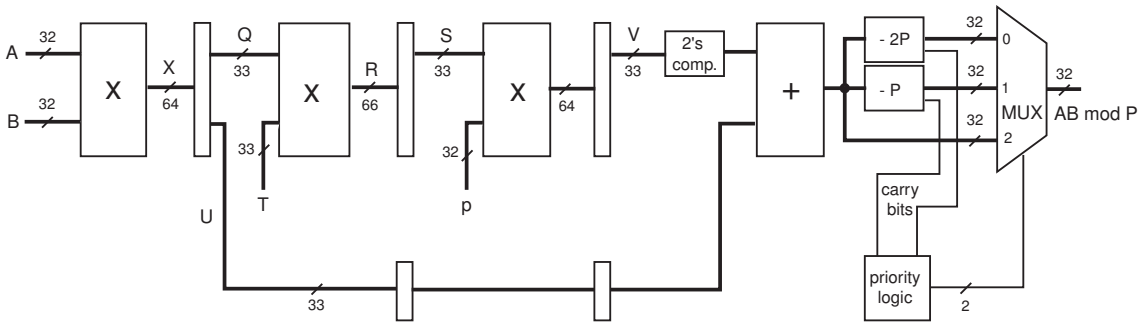


Fig. 3. Architecture for 32-bit Modular Multiplier.

and the last 15 clock cycles as the *tail cycles*. These periods of clock cycles are important for the first and last multiplication operations performed in the pipeline architecture in Figure 3. We will need these pieces of information to accurately estimate the number of clock cycles needed in our computations in subsequent sections.

4 $2^n \times 2^n$ Polynomial Multiplier

We implemented a $2^n \times 2^n$ polynomial multiplier, with 32–bit coefficients. Throughout the paper, we will use the term 2^n to denote the $2^n \times 2^n$ polynomial multiplier. We do not utilize any special modulus, to achieve a generic and robust polynomial multiplier as we use Barrett's reduction algorithm for coefficient arithmetic. Instead of the classical schoolbook method for polynomial multiplication, we utilized the NTT–based multiplication algorithm, as explained in Section 2.2 and described in Algorithm 5. It should be noted that Step 5 of Algorithm 5 is implemented by coefficient–wise 32–bit modular multiplications.

ALGORITHM 5: NTT–based 2^n polynomial multiplication

input : $A(x) = A_0 + A_1 x + \cdots + A_{2^n-1} x^{2^n-1}$,
$B(x) = B_0 + B_1 x + \cdots + B_{2^n-1} x^{2^n-1}$, p
output: $C(x) = A(x) \times B(x)$

1 $NTT_A(x) \leftarrow$ NTT of polynomial $A(x)$;
2 $NTT_B(x) \leftarrow$ NTT of polynomial $B(x)$;
3 $NTT_C(x) \leftarrow$ Inner products of polynomials $NTT_A(x)$ and $NTT_B(x)$;
4 $T(x) \leftarrow$ Inverse NTT of polynomial $NTT_C(x)$;
5 $C(x) \leftarrow T(x) \times ((2^n)^{-1} \bmod p)$;

4.1 NTT Operation

4.1.1 NTT Algorithm

We apply the NTT operation on a polynomial $A(x)$ of degree $2^n - 1$ over $\mathbb{Z}_p[x]/(\Phi(x))$. Since the result of the NTT–based multiplication will be of degree $2^{(n+1)}$, we need to zero–pad the polynomial $A(x)$ to make it also a polynomial of degree $2^{(n+1)}$ as follows $A(x) = \sum_{j=0}^{2^n-1} A_j \cdot x^j + \sum_{j=2^n}^{2^{(n+1)}-1} 0 \cdot x^j$. When we apply the NTT transform on $A(x)$, the resulting polynomial is $\mathcal{A}(x) = \sum_{i=0}^{2^{(n+1)}-1} \mathcal{A}_i \cdot x^i$, where the coefficients $\mathcal{A}_i \in \mathbb{Z}_p$ are defined as $\mathcal{A}_i = \sum_{j=0}^{2^{(n+1)}-1} A_j \cdot w^{ij} \bmod p$, and $w \in \mathbb{Z}_p$ is referred as the twiddle factor. Since the size of the NTT operation is actually $2^{(n+1)}$, we need to choose a twiddle factor w which satisfies the property $w^{2^{(n+1)}} \equiv 1 \bmod p$ and $\forall i < 2^{(n+1)}\ w^i \neq 1 \bmod p$.

To achieve fast NTT operations, we utilize the Cooley–Tukey approach, as explained in Section 2.2. Cooley–Tukey approach works by splitting up the NTT–transform into two parts, performing the NTT operation on the smaller parts, and performing a final reconstruction to combine the results of the two half–size NTT transform results into a full–sized NTT operation. If the NTT operation is defined as:

$$\mathcal{A}_i = \sum_{j=0}^{2^{(n+1)}-1} A_j \cdot w^{ij} \bmod p,$$

we can split up this operation as follows

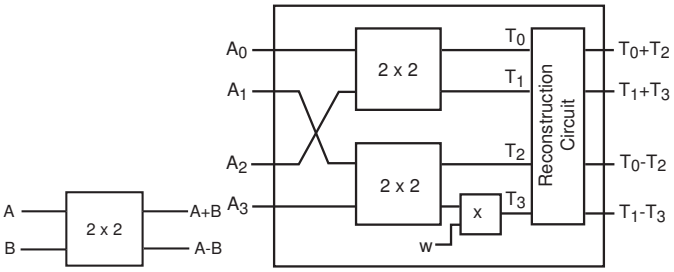
$$\mathcal{A}_i = \sum_{j=0}^{2^n-1} A_{2j} \cdot w^{i(2j)} \bmod p + \sum_{j=0}^{2^n-1} A_{2j+1} \cdot w^{i(2j+1)} \bmod p,$$

which can also be expressed as $\mathcal{A}_i = E_i + w^i O_i$, where E_i and O_i represent the i^{th} coefficients of the 2^n NTT operation on the even and odd coefficients of the polynomial $A(x)$, respectively. It is important to note that if the twiddle factor of the $2^{(n+1)}$ NTT operation is w, the twiddle factor of the smaller 2^n operation will be w^2. Because of the periodicity of the NTT operation, we know that $E_{i+2^n} = E_i$ and $O_{i+2^n} = O_i$. Therefore, we have $\mathcal{A}_i = E_i + w^i O_i$ for $0 \leq i < 2^n$ and $\mathcal{A}_i = E_{i-2^n} + w^i O_{i-2^n}$ for $2^n \leq i < 2^{(n+1)}$. For the twiddle factor, it holds that $w^{i+2^n} = w^i \cdot w^{2^n} = -w^i$. Consequently, we can achieve a full $2^{(n+1)}$ NTT operation with two small 2^n NTT operations utilizing the following reconstruction operation

$$\begin{aligned} \mathcal{A}_i &= E_i + w^i O_i, \\ \mathcal{A}_{i+2^n} &= E_i - w^i O_i. \end{aligned} \quad (1)$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

The reconstruction operation is performed iteratively over very large number of coefficients. To better explain the iterative Cooley–Tukey approach, we would like to give a toy example of the NTT operation. First, we show the smallest NTT-transform circuit used in our design, which is shown in Figure 4a. Here, the NTT operation is applied over a polynomial of degree 1, with $w^2 \equiv 1 \bmod p$. Therefore, the two outputs of the circuit are $A+B$ and $A+wB \equiv A-B \bmod p$. Utilizing the 2×2



(a) 2×2 NTT circuit (b) 4×4 NTT circuit

Fig. 4. Construction of the 4×4 NTT circuit from 2×2 circuits

NTT circuit, we can perform a 4×4 NTT operation as shown in Figure 4b. Here, since we are constructing a 4×4 NTT circuit, we have $w^4 \equiv 1 \bmod p$.

In a similar fashion, we can achieve an 8×8 NTT operation utilizing two 4×4 NTT operations, as shown in Figure 5. Here, since we are constructing an 8×8 NTT circuit, we have $w^8 \equiv 1 \bmod p$. Also in Figure 5, we can

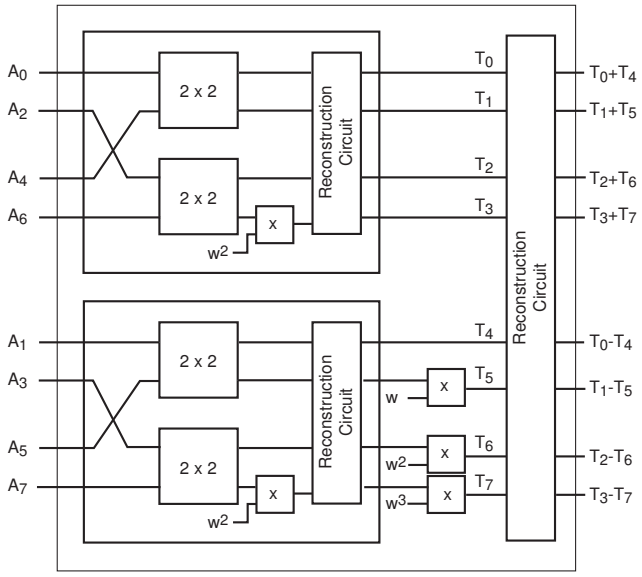


Fig. 5. Construction of the 8×8 NTT circuit iteratively.

see that if the twiddle factor of the 8×8 NTT operation is w, the twiddle factor of the 4×4 NTT operation is w^2. The overall architecture for iterative computation of NTT is shown in Figure 6. Note that, in a full $2^{(n+1)}$

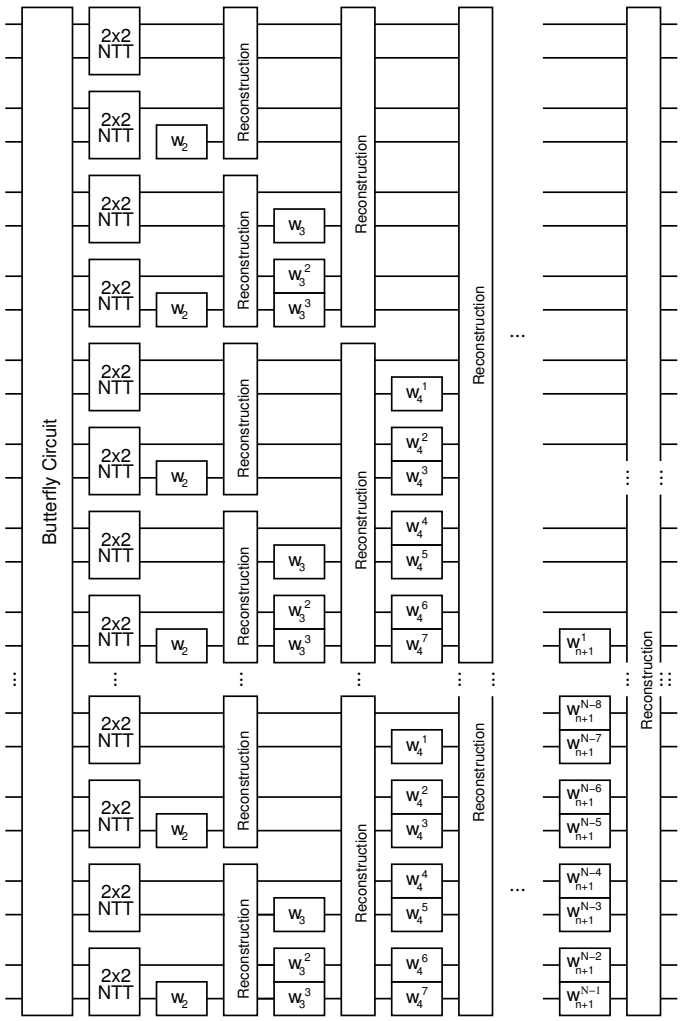


Fig. 6. NTT Circuit

NTT circuit, the twiddle factor w^{16484} is used in 8×8 NTT circuits.

4.1.2 Coefficient Multiplication and Accumulation

In order to parallelize multiplication and accumulation operations we utilize $3 \cdot K$ DSP units to achieve K modular multiplications in parallel, with a 4–cycle throughput, where K is a design parameter that depends on the number of available DSP units in the target architecture. In our design, K is chosen as a power of 2.

To be able to feed the DSP units with correct polynomial coefficients during multiplication cycles, we utilize K separate Block RAMs (BRAM) to store the polynomial coefficients as shown in Figure 7 (e.g. $K = 128$). The algorithm used to access the polynomial coefficients in parallel is described in Algorithm 6. The algorithm takes the BRAM content (i.e., the coefficients of $A(x)$), the degree $N = 2^n$, the current level m, and the number of modular multipliers $K = 2^\kappa$ as input, and generates the indexes in a parallel manner. Every four clock cycles, we try to feed modular multipliers the number of coef-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

ficients which is as close to K as possible. Ideally, it is desirable to perform exactly K modular multiplications in parallel, which is not possible due to the access pattern to the powers of w. Algorithm 6, on the other hand, achieves a good utilization of modular multiplication units.

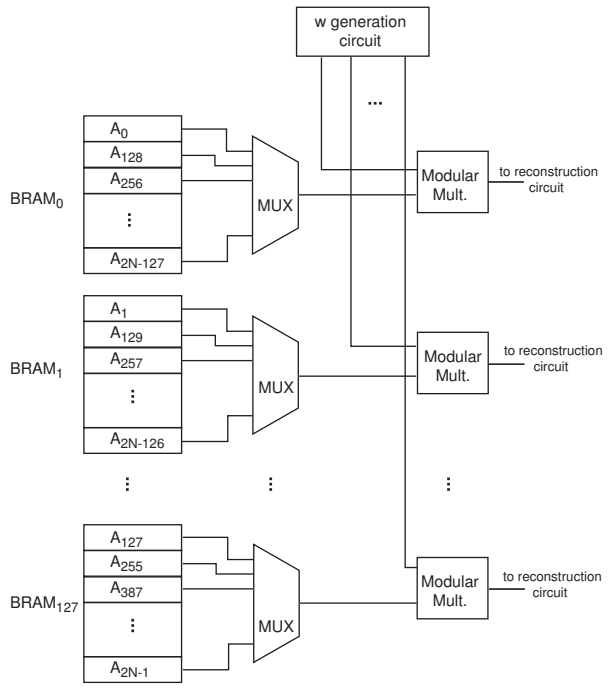


Fig. 7. The architecture for NTT transformation of a polynomial of degree N over F_p, where $\lceil \log_2 p \rceil = 32$.

ALGORITHM 6: Parallel access to polynomial coefficients

input : $A(x) = A_0 + A_1 x + \ldots + A_{2N-1}x^{2N-1}$, n, m, and $\kappa < n$
output: $B_i[j]$

```
1  mCnt ← 2^(m−1) − 1 ; /* number of multiplications in a block */
2  bSize ← 2^m ;                /* size of a block */
3  BRAMCnt ← 2^(κ−2) ;          /* number of BRAMs */
4  if bSize ≤ 2^(κ−2) then
       for t = 0 to 1024 do
           for i = 0 to BRAMCnt do in parallel
               for j = i + bSize − mCnt to i + bSize do
                   for k = 0 to 3 do
5                      Access BRAM_j[t + 2k] ;
6                      Access BRAM_j[t + 2k + 1] ;
                       k ← k + 1;
                   end
                   j ← j + 1;
               end
               i ← i + bSize;
           end
           t ← t + 8;
       end
   end
7  else
       for i = 0 to BRAMCnt do in parallel
           for j = 0 to 1024 do
               for k = 2^(m−κ+1) to 2^(m−κ+2) do
8                  Access BRAM_i[k + j] ;
                   k ← k + 1;
               end
               j ← j + 2^(m−κ+2);
           end
           i ← i + 1;
       end
   end
```

For level m, we use the $2^m \times 2^m$ NTT circuit. The coefficients are arranged in $2^m \times 2^m$ blocks. For example when $K = 256$, for the first level of the NTT operation, where $m = 2$, we need to multiply every $4th$ coefficient of the polynomial with $w_2 = w^{16384}$. Since the coefficients are perfectly dispersed, we can read 256 coefficients to feed the 256 multipliers in four clock cycles. This is perfect as the throughput of our multipliers are also four cycles. When the multiplication operations are complete, with an offset of 19 cycles (four clock cycles are for the warm up of the pipeline whereas 15 clock cycles are the tail cycles necessary in a pipelined design to finish the last operation), the results are written back to the same address of the RAM block as the one the coefficients are read from.

We provide formulae for the number of multiplications in each level and an estimate of the number of clock cycles needed for their computation in our architecture. Suppose $N = 2^n$ and $K = 2^\kappa$ $(n > \kappa)$ are the number of coefficients in our polynomial and the number of modulo multipliers in our target device, respectively. The coefficients are stored in BRAMs, with a word size of 32 bits and an address length of 10 bits (1024 coefficients per BRAM). For ideal case, the number of modular multipliers should be 4 times the number of BRAMS required to store a single polynomial. The formula for the number of multiplications for the level $m > 1$ can be given as $\mathcal{M} = 2^{n+1-m} \cdot (2^{m-1} - 1)$. Also, using $K = 2^\kappa$ multipliers, the number of clock cycles to compute all multiplications in a given level $1 < m \leq n + 1$ can be formulated as

$$CC_m = \begin{cases} 4 + 4 \cdot \left\lfloor \dfrac{\mathcal{M}}{\alpha \cdot \lfloor K/\alpha \rfloor} \right\rfloor + 15 & \kappa \geq m \\ \\ 4 + 4 \cdot (\frac{\beta}{K} + 1) \cdot 2^{n+1-m} + 15 & \kappa < m, \end{cases}$$

where $\alpha = 2^{\kappa-m} \cdot (2^{m-1} - 1)$ and $\beta = 2^{m-1} - 2^\kappa$. In the formula, the first (4) and the last terms (15) account for the warm up and the tail cycles.

As an example, Table 3 shows the number of multiplication operations required for each stage of the iterative Cooley-Tukey NTT operation, for a 32768–coefficient (64K-point) NTT operation, when the number of modular multipliers is 256. (i.e., $N = 2^{15}$ and $K = 256$).

As mentioned before, the modulo multipliers are not always fully utilized during the NTT computation. For example when $K = 2^8$ and $N = 2^{15}$, for $m = 2$, we have to read every 4^{th} coefficient from the BRAMs. Because the coefficients are perfectly dispersed throughout the 64

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

TABLE 2
Powers of w needed in different levels of NTT circuit

Level (m)	Block size	powers of w
2	4×4	$w^{2^{14}}$
3	8×8	$w^{2^{13}}, w^{2\cdot 2^{13}}, w^{3\cdot 2^{13}}$
4	16×16	$w^{2^{12}}, w^{2\cdot 2^{12}}, \ldots, w^{(2^3-1)\cdot 2^{12}}$
5	32×32	$w^{2^{11}}, w^{2\cdot 2^{11}}, \ldots, w^{(2^4-1)\cdot 2^{11}}$
6	64×64	$w^{2^{10}}, w^{2\cdot 2^{10}}, \ldots, w^{(2^5-1)\cdot 2^{10}}$
7	128×128	$w^{2^9}, w^{2\cdot 2^9}, \ldots, w^{(2^6-1)\cdot 2^9}$
8	256×256	$w^{2^8}, w^{2\cdot 2^8}, \ldots, w^{(2^7-1)\cdot 2^8}$
9	512×512	$w^{2^7}, w^{2\cdot 2^7}, \ldots, w^{(2^8-1)\cdot 2^7}$
10	1024×1024	$w^{2^6}, w^{2\cdot 2^6}, \ldots, w^{(2^9-1)\cdot 2^6}$
11	2048×2048	$w^{2^5}, w^{2\cdot 2^5}, \ldots, w^{(2^{10}-1)\cdot 2^5}$
12	4096×4096	$w^{2^4}, w^{2\cdot 2^4}, \ldots, w^{(2^{11}-1)\cdot 2^4}$
13	8192×8192	$w^{2^3}, w^{2\cdot 2^3}, \ldots, w^{(2^{12}-1)\cdot 2^3}$
14	16384×16384	$w^{2^2}, w^{2\cdot 2^2}, \ldots, w^{(2^{13}-1)\cdot 2^2}$
15	32768×32786	$w^2, w^{2\cdot 2}, \ldots, w^{(2^{14}-1)\cdot 2}$
16	65536×65536	$w, w^2, \ldots, w^{2^{15}-1}$

TABLE 3
Details of NTT computation in our architecture for 32768 coefficients and and 256 multiplier units.

NTT blocks	number of blocks	number of modular multiplications	number of clock cycles
4×4	16384	16384	275
8×8	8192	24576	531
16×16	4096	28672	
32×32	2048	30720	
64×64	1024	31744	
128×128	512	32256	
256×256	256	32512	
512×512	128	32640	
1024×1024	64	32704	
2048×2048	32	32736	
4096×4096	16	32752	
8192×8192	8	32760	
16384×16384	4	32764	
32768×32768	2	32766	
65536×65536	1	32767	
Total clock cycles			**7709**

BRAMS, we can only read $16 \cdot 2 = 32$ coefficients every clock cycle, which yields a number of 128 concurrent multiplications every four clock cycles. Consequently, we can finish all the modular multiplications in the first level in $4 + 128 \cdot 4 + 15 = 531$ clock cycles. Since we can use half the modular multipliers, we achieve half utilization in the first level. However, when $m = 3$, we have to read every 6^{th}, 7^{th} and 8^{th} out of every 8 coefficients. We can read $24 \cdot 2 = 48$ coefficients every clock cycle from the BRAMs. This means we can only utilize 192 out of 25 modular multipliers since the irregularity of the access to the polynomial coefficients. This, naturally, results in a slightly low utilization. However, since we can read 2 coefficients from each BRAM every clock cycle, we are at almost perfect utilization, resulting in $4+128\cdot 4+15 = 531$ clock cycles for this and the rest of the stages.

Since the operands of the both operations are accessed in a regular manner, the number of clock cycles spent on modular additions and subtractions are calculated as $\frac{2^{(n+1)}\cdot(n+1)}{2^\tau}$, when there are 2^τ modular adders and 2^τ subtractors.

4.1.3 w Generation

Theoretically, we need an N-th root of unity in F_p for NTT of polynomials of degree N. Due to the polynomial padding in our case, we need an $2N$-th root of unity $w \in F_p$ such that $w^{2^{(n+1)}} = 1 \bmod p$ and $\forall i < 2^{(n+1)}$, $w^i \neq 1 \bmod p$.

In every level of the NTT circuit, we use different powers of w. For the level m, where we use the $2^m \times 2^m$ butterfly circuit and the coefficients are arranged in $2^m \times 2^m$ blocks, we need $w_m^1, w_m^2, \ldots, w_m^{2^{m-1}-1}$ where $w_m = w^{2^{16-m}}$. For instance, $w^{2^{14}}$ is used in every multiplication in the 4×4 butterfly circuit while $w^{2^{13}}, w^{2\cdot 2^{13}}, w^{3\cdot 2^{13}}$ are used in the multiplications in 8×8 butterfly circuit.

For the powers of w that are used in different levels of computation for a 2^{16}–point NTT operation, see Table 2. In summary, for the 2^{16}–point NTT we need $2^{15} - 1 = 32767$ powers of w; namely $w, w^2, w^3, \ldots, w^{32767}$. In case of 2^{14} polynomial multiplier we require up to 2^{15}–point NTT arithmetic which we only need $2^{14}-1 = 16383$ coefficients for powers of w, e.g. $w^2, w^{2\cdot 2}, w^{2\cdot 3}, \ldots, w^{2\cdot 16383}$. We precompute and store these powers of w in block RAMs in a distributed fashion similar to the coefficients of the polynomials as illustrated in Figure 7. Alternatively, the powers of w can be computed on-the-fly for area efficiency. However, since we have sufficient number of block RAMs in the target reconfigurable device, we prefer the precomputation approach.

4.1.4 Reconstruction

Once we are done with the multiplications, we utilize 64 modular adders and 64 modular subtracters to realize the addition and subtraction operations as shown in Equation 1.

4.2 Inner Multiplication

Inner multiplication of two 2^n polynomials is trivial for our hardware design. We can load 256 coefficients from each polynomial every 4 cycles and feed the multipliers, without increasing the 4–cycle throughput. For a 2^n polynomial inner multiplication we spend $2^{(n+1)} \cdot 4/256 + 15$ clock cycles.

4.3 Inverse NTT

The Inverse NTT operation is identical to the NTT operation, except that instead of the twiddle factor w, we use the twiddle factor $w_i = w^{-1} \bmod p$. The precomputed twiddle factors of the inverse NTT are stored in the same block RAMs as the forward NTT twiddle factors, with an address offset. Therefore, the same control block can be utilized with a simple address change for the w coefficients for the inverse NTT operation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

4.4 Final Scaling

Final scaling is similar to the inner multiplication phase. We load each coefficient of the resulting polynomial, and multiply them with the precomputed scaling factor. Similar to the inner multiplication phase, we can load 256 coefficients from the resulting polynomial in 4 cycles cycle and feed the multipliers, without increasing the 4–cycle throughput. For a 2^n polynomial final scaling operation, we spend $2^{(n+1)} \cdot 4/256 + 15$ clock cycles.

5 Implementation Results

We developed the architecture described in the previous section into Verilog modules and synthesized it using Xilinx Vivado tool for the Virtex 7 XC7VX690T FPGA family. The synthesis results are summarized in Table 4. We synthesized the design and achieved an operating frequency of 250 MHz for multiplication of polynomials of degrees $N = 16,384$ for Prince and $N = 32,768$ for AES with a small word size of $\log p = 32$ bit [1]. In Table 5 we summarize the timing results of the synthesized small word size polynomial multiplier.

Although we can scale our architecture for larger parameters, it becomes hard to synthesize, since we are using 50 percent of the LUTs already. Another problem is that with larger hardware it is harder to do the routing because of the butterfly circuit mapping at each level. Also, it becomes harder to fit all the necessary components, i.e. polynomials, powers of ω and resulted polynomial in the FPGA. Therefore, it becomes impossible to process a multiplication without extra I/O transactions when computing the NTT conversions.

TABLE 4
Virtex-7 XC7VX690T device utilization of the multiplier

N $= (16,384/32,768)$	**Total**	**Used**	**Used (%)**
Slice LUTs	433,200	219,192	50.59
Slice Registers	866,400	90,789	10.47
RAMB36E1	1470	193	13.12
DSP48E1	3600	768	21.33

TABLE 5
Timing results for 32-bit coefficient polynomial multiplier for various degree N sizes

N	**NTT**	**Mult**	**PCIe**	**Total**
$16,384$	24.5 μs	73.4 μs	26 μs	99.4 μs
$32,768$	50.9 μs	152 μs	79 μs	231 μs

The FPGA multiplier is used to process each component of the CRT representation of our large coefficient ciphertexts with $\log q = 500$ bits for Prince and $\log q = 1271$ bits for AES implementation. In fact we keep all ciphertexts in CRT representation and only compute the polynomial form when absolutely necessary, e.g. for parity correction during modulus switching and before relinearization. We assume any data sent from the PC through the PCIe interface to the FPGA is stored in onboard BRAM units.

CRT Computation Cost. To facilitate efficient computation of multiplication and relinearization operations we use a series of equal sized prime numbers to construct a CRT conversion. In fact, we chose the primes p_i's such that $q = \prod_{i=0}^{l} p_i$. During the levels of homomorphic evaluation, this representation allows us to easily switch modulus by simply dropping the last p_i following by a parity correction. Also, since we have an RNS representation on the coefficients we no longer need to reduce by q. This also eliminates the need to consider any overflow conditions. Thus, $l = \log(q)/\log(p_i)$ is 25 and 41 for Prince and AES implementations, respectively. We efficiently compute the CRT residue in software on the CPU for each polynomial coefficient as follows:

- Precompute and store $t_k = 2^{64 \cdot k} \pmod{p_i}$ where $k \in [0, \lceil \log(q/64) - 1 \rceil]$.
- Given a coefficient of c, we divide it into 64-bit blocks as $c = \{\ldots, w_k, \ldots, w_0\}$.
- We compute the CRT result by evaluating $\sum t_k \cdot w_k \pmod{p_i}$ iteratively.

The CRT computation cost for 41 primes p_i per ciphertext polynomial is in the order of 89 ms and for 25 primes p_i per ciphertext polynomial is in the order of 14.5 ms on the CPU. The CRT inverse is similarly computed (with the addition of a word carry) before each modulus switching operation at essentially the same cost.

Communication Cost. The PCIe bus is only used for transactions of input/output values, NTT constants and transport of evaluation keys to the FPGA board. With 8 lanes each capable of supporting 8 GT/s transport speed the PCIe is capable to transmit a 1 MB ciphertext in about 0.13 ms. Note that the NTT parameters used during multiplication also need to be transported since we do not have enough room in the BRAM components to keep them permanently. We have two cases to consider:

- Multiplication: We transport two polynomials of 5 MB / 1 MB each along with the NTT parameters of 5 MB / 1 MB and receive a polynomial of 10 MB / 2 MB, which costs about 3.25 ms / 0.65 ms per multiplication for AES/Prince implementation.
- Relinearization: We need to transport the ciphertext we want to relinearize, the NTT parameters and a set of $\frac{\log(q)}{16} \approx 80$ / $\frac{\log(q)}{16} \approx 32$ evaluation keys (ciphertexts), where a window size of 16-bit is used, resulting in a 52 ms / 10 ms delay for AES/Prince implementation.

Multiplication Cost. We compute the product of two polynomials with coefficients of size $\log(p) = 32$ bits using 256 modular multipliers in $12720/6120$ cycles,

1. We use the same hardware architecture for both applications. The only difference is that compared to $N = 16,384$ case, the architecture is used almost twice many times in $N = 32,768$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

which translates to 152 μs / 73.4 μs for AES/Prince implementation. This figure is comprised of two NTT and one inverse NTT operations and one inner product computation. The addition of I/O transactions increase the timing by 79 μs / 26 μs for AES/Prince implementations. The latency of large polynomial multiplication may be broken down as follows:

- Cost of small coefficient polynomial multiplications is $41 \cdot 152$ μs $= 6.25$ ms for AES and $25 \cdot 73.4$ μs $= 1.84$ ms for Prince.
- The PCIe transaction of the two input polynomials, the NTT coefficients and the double sized output polynomial is 3.25 ms / 0.64 ms for AES/Prince implementation.

Thus, the total latency for large polynomial multiplication in the CRT representation is computed in 9.51 ms and 2.48 ms for AES and Prince implementations respectively.

Polynomial Modular Reduction. Since all operations are computed in a polynomial ring with a characteristic polynomial as modulus without any special structure, we use Barrett's reduction technique to perform the reductions. Note that precomputing the constant polynomial $x^{2N}/\Phi(x)$ (truncated division) in the CRT representation we do not need to compute any CRT or inverse CRT operations during modular reduction. Thus we can compute the reduction using two product operations in about 19 ms and 4.9 ms for AES and Prince implementations respectively.

Modulus Switching. We realize the modulus switching operation by dropping the last CRT coefficient followed by parity correction. To compute the parity of the cut polynomial we need to compute an inverse CRT operation. The following parity matching and correction step takes negligible time. Therefore, modulus switching can be realized using one inverse CRT computation in 89 ms and 14.5 ms for AES and Prince implementations respectively.

Relinearization Cost. To realinearize a ciphertext polynomial

- We need to convert the ciphertext polynomial coefficients into integer representation using one inverse CRT operation, which takes 89 ms / 14.5 ms for AES/Prince implementation.
- The evaluation keys are kept in NTT representation, therefore we only need to compute two NTT operations for one operand and the result. For $l = 41/25$ primes and $\frac{\log(q)}{16} \approx 80/32$ products the NTT operations take 331 ms / 38 ms for AES/Prince implementation.
- We need to transport the ciphertext, the NTT parameters and $80/32$ evaluation keys (ciphertexts) resulting in a 52 ms / 4 ms delay for AES/Prince implementation.
- The summation of the partial products takes negligible time compared to the multiplications and the PCIe communication cost.

Then, the total relinearization operation takes 526 ms and 61.2 ms for AES and Prince implementation respectively. With the current implementation, the actual NTT computations still dominate over the other sources of latency such as PCIe communication latency and the CRT computations. However, if the design is further optimized, e.g. by increasing the number of processing units on the FPGA or by building custom support for CRT operations on the FPGA, then the PCIe communication overhead will become more dominant. The timing results are summarized in Table 6.

TABLE 6
Primitive operation timings including I/O transactions.

	AES Timings (ms)	Prince Timings (ms)
CRT	89	14.5
Multiplication	9.51	2.48
NTT conversions	6.25	1.8
PCIe cost	3.26	0.64
Modular Reduction	19	4.95
Modulus Switch	89	14.5
Relinearization	526	61.2
CRT conversions	89	14.5
NTT conversions	331	38.2
PCIe cost	52	4

6 Comparison

To understand the improvement gained by adding custom hardware support in leveled homomorphic evaluation of a deep circuit, we *estimate* the homomorphic evaluation time for the AES and Prince circuits and compare it with a similar software implementations by Doröz et al [11], [12] and by Wei et al [29], [30].

Homomorphic AES evaluation. Using the NTRU primitives we implemented the depth 40 AES circuit following the approach in [11]. The tower field based AES SBox evaluation is completed using 18 Relinearization operations and thus 2,880 Relinearizations are needed for the full AES. The AES circuit evaluation requires 5760 modular multiplications. During the evaluation we also compute 6080 modulus switching operations. This results in a total AES evaluation time of 15 minutes. Note that during the homomorphic evaluation with each new level the operands shrink linearly with the levels thereby increasing the speed. We conservatively account for this effect by dividing the evaluation time by half. With 2048 message slots, the amortized AES evaluation time becomes 439 ms.

We have also modified Doröz et al.'s homomorphic AES evaluation code to compute relinearization with 16-bits windows (originally single bit). This simple optimization dramatically reduces the evaluation key size and speeds up the relinearization. The results are given in Table 7. We also included the GPU optimized implementation by Dai et al. [29] on an NVIDIA GeForce GTX

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2574340, IEEE Transactions on Computers

TABLE 7
Comparison of multiplication, relinearization times and AES estimate

	Mul (ms)	Speedup	Relin (s)	Speedup	AES (s)	Speedup
CPU [11]	970	1×	103	1×	55	1×
GPU [29]	340	2.8×	8.97	11.5×	7.3	7.5×
CPU (16-bit)	970	1×	6.5	16×	12.6	4.4×
Ours	9.5	102×	0.53	195×	0.44	125×

680. With custom hardware assistance we obtain a significant speedups in both multiplication and relinearization operations. The estimated AES block evaluation is also improved significantly where some of the efficiency is lost to the PC to FPGA communication and CRT computation latencies.

Homomorphic Prince evaluation. Using the NTRU primitives we implemented the depth 24 Prince circuit following the approach in [12]. The algorithm is completed using 1152 relinearizations, 1920 multiplications, 3072 modular reductions and 2688 modular switch operations. An important thing to note that as we did in AES implementation, we divide the evaluation time by half. The reason is that since during the homomorphic evaluation with each new level, the operands shrink linearly so the evaluation speed increases linearly. These results in a total time of 53 seconds and an amortized time of 52 ms with batching 1024 messages. Here in Table 8, we compare the results of homomorphic Prince implementation of Doröz et al. [12] which is implemented using a CPU. Also, we include the homomorphic Prince implementations of Dai et al. [29], [30] on GPUs which are significantly faster compared to the CPU implementation.

TABLE 8
Comparison of multiplication, relinearization times and Prince estimate

	Mul (ms)	Speedup	Relin (s)	Speedup	Prince (s)	Speedup
CPU [12]	180	1×	10.9	1×	3.3	1×
GPU [29]	63	2.85×	0.89	12.3×	1.28	2.58×
GPU [30]	n/a	n/a	n/a	n/a	0.032	103×
Ours	2.5	72×	0.06	181×	0.05	66×

7 Conclusions

We presented a custom hardware design to address the performance bottleneck in leveled somewhat homomorphic encryption evaluations. For this, we design a large NTT based multiplier, which is able to compute large degree polynomial multiplications using the Cooley-Tukey FTT technique. We extend the support of the custom core to be capable of multiplying large degree polynomials with large coefficients by using CRT representation on the coefficients. Using numerous techniques the design is highly optimized to speedup the NTT computations, and to reduce the burden on the PC/FPGA interface. Our design achieves remarkable improvements in speed of modular multiplication and relinearization of the LTV SWHE scheme compared to the previous software implementations. In order to show the acceleration that our architecture may provide, we estimated the homomorphic AES and Prince evaluation performances and determined a speedup of about 28 and 66 times respectively. Finally, we would like to note that these estimates are only to get a sense of the improvement that our architecture brings in. This custom accelerator architecture can be more useful in many other practical homomorphic evaluation applications in practice.

Acknowledgments

Funding for this research was in part provided by the US National Science Foundation CNS Award #1319130 and the Scientific and Technological Research Council of Turkey project #113C019.

References

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of Secure Computation*, pp. 169–180, 1978.

[2] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[3] ——, "Fully homomorphic encryption using ideal lattices," in *STOC*, 2009, pp. 169–178.

[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping," *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 18, p. 111, 2011.

[5] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Designs, Codes and Cryptography*, vol. 71, no. 1, pp. 57–81, 2014.

[6] Z. Brakerski, C. Gentry, and S. Halevi, *Public-Key Cryptography – PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. Packed Ciphertexts in LWE-Based Homomorphic Encryption, pp. 1–13.

[7] J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun, *Advances in Cryptology – EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. Batch Fully Homomorphic Encryption over the Integers, pp. 315–335.

[8] C. Gentry, S. Halevi, and N. P. Smart, *Advances in Cryptology – CRYPTO 2012: 32nd Annual Cryptology Conference, 2012. Proceedings*. Springer Berlin Heidelberg, 2012, ch. Homomorphic Evaluation of the AES Circuit, pp. 850–867.

[9] Y. Doröz, E. Öztürk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," in *Digital System Design (DSD), 2013 16th Euromicro Conference on*, 2013.

[10] ——, "Accelerating fully homomorphic encryption in hardware," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, 2015.

[11] Y. Doröz, Y. Hu, and B. Sunar, "Homomorphic aes evaluation using the modified ltv scheme," *Designs, Codes and Cryptography*, pp. 1–26, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10623-015-0095-1

[12] Y. Doröz, A. Shahverdi, T. Eisenbarth, and B. Sunar, *Financial Cryptography and Data Security: FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ch. Toward Practical Homomorphic Evaluation of Block Ciphers Using Prince, pp. 208–220.

[13] D. Cousins, K. Rohloff, R. Schantz, and C. Peikert, "SIPHER: Scalable implementation of primitives for homomorphic encrytion," Internet Source, September 2011.

[14] D. Cousins, K. Rohloff, C. Peikert, and R. E. Schantz, "An update on SIPHER (scalable implementation of primitives for homomorphic encRyption) - FPGA implementation using simulink," in *HPEC*, 2012, pp. 1–5.

[15] C. Moore, N. Hanley, J. McAllister, M. O'Neill, E. O'Sullivan, and X. Cao, "Targeting FPGA DSP slices for a large integer multiplier for integer based FHE," *Workshop on Applied Homomorphic Cryptography*, vol. 7862, 2013.

[16] X. Cao, C. Moore, M. O'Neill, N. Hanley, and E. O'Sullivan, "Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction," *Under Review*, 2013.

[17] W. Wang and X. Huang, "FPGA implementation of a large-number multiplier for fully homomorphic encryption," in *ISCAS*, 2013, pp. 2589–2592.

[18] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 698–706, March 2015.

[19] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *EUROCRYPT*, 2011, pp. 129–148.

[20] ——, "Fully homomorphic encryption without squashing using depth-3 arithmetic circuits," in *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science*, ser. FOCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 107–109. [Online]. Available: http://dx.doi.org/10.1109/FOCS.2011.94

[21] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Public Key Cryptography*, 2010, pp. 420–443.

[22] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *EUROCRYPT*, 2010, pp. 24–43.

[23] J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully homomorphic encryption over the integers with shorter public keys," in *CRYPTO*, 2011, pp. 487–504.

[24] J.-S. Coron, D. Naccache, and M. Tibouchi, "Public key compression and modulus switching for fully homomorphic encryption over the integers," in *EUROCRYPT*, 2012, pp. 446–464.

[25] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *FOCS*, 2011, pp. 97–106.

[26] C. Gentry, S. Halevi, and N. P. Smart, *Public Key Cryptography – PKC 2012: 15th International Conference on Practice and Theory in Public Key Cryptography, 2012. Proceedings*, 2012, ch. Better Bootstrapping in Fully Homomorphic Encryption, pp. 1–16.

[27] ——, *Advances in Cryptology – EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2012. Proceedings*, 2012, ch. Fully Homomorphic Encryption with Polylog Overhead, pp. 465–482.

[28] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *HPEC*, 2012, pp. 1–5.

[29] W. Dai, Y. Doröz, and B. Sunar, "Accelerating NTRU based homomorphic encryption using GPUs," in *HPEC*, 2014.

[30] W. D. andand Berk Sunar, "cuHE: A homomorphic encryption accelerator library," in *BalkanCryptSec2015*, 2015.

[31] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *STOC*, 2012.

[32] D. Stehlé and R. Steinfeld, "Making NTRU as secure as worst-case problems over ideal lattices," *Advances in Cryptology – EUROCRYPT '11*, pp. 27–4, 2011.

[33] K. Rohloff and D. Cousins, "A scalable implementation of somewhat homomorphic encryption built on NTRU," in *2nd Workshop on Applied Homomorphic Cryptography (WAHC'14)*, 2014.

[34] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akad. Nauk SSSR*, vol. 145, no. 293–294, p. 85, 1962.

[35] 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.

[36] R. C. Agarwal and C. S. Burrus, "Fast convolution using fermat number transforms with applications to digital filtering," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 22, no. 2, pp. 87–97, Apr 1974.

[37] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *LATINCRYPT*, ser. Lecture Notes in Computer Science, A. Hevia and G. Neven, Eds., vol. 7533. Springer, 2012, pp. 139–158.

[38] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient fpga implementations of lattice-based cryptography," in *HOST*. IEEE, 2013, pp. 81–86.

[39] T. Yanık, E. Savas, and C. Koc, "Incomplete reduction in modular arithmetic," *IEE Proceedings: Computers and Digital Techniques*, vol. 149, no. 2, pp. 46–52, 2002, cited By (since 1996)22.

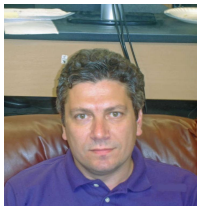
[40] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology CRYPTO 86*, ser. Lecture Notes in Computer Science, A. Odlyzko, Ed., 1987, vol. 263, pp. 311–323.

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

[42] S. V. Darrel Hankerson, Alfred J. Menezes, *Guide to Elliptic Curve Cryptography*. Springer Professional Computing, 2004.

Erdinç Öztürk Erdinç Öztürk received his BS degree in Microelectronics from Sabanci University at 2003. He received his MS degree in Electrical Engineering at 2005 and PhD degree in Electrical and Computer Engineering at 2009 from Worcester Polytechnic Institute. His research field was Cryptographic Hardware Design and he focused on efficient Identity Based Encryption implementations. After receiving his 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.

Yarkın Doröz Yarkın Doröz received a BSc. degree in Electronics Engineering at 2009 and a MSc. degree in Computer Science at 2011 from Sabanci University. Currently he is working towards a Ph.D. degree in Electrical and Computer Engineering at Worcester Polytechnic Institute. His research is focused on developing hardware/software designs for Fully Homomorphic Encryption Schemes.



Erkay Savaş Erkay Savaş received the BS (1990) and MS (1994) degrees in electrical engineering from the Electronics and Communications Engineering Department at Istanbul Technical University. He completed the PhD degree in the Department of Electrical and Computer Engineering (ECE) at Oregon State University in June 2000. He has been a faculty member at Sabanci University since 2002. His research interests include applied cryptography, data and communication security, security and privacy in data mining applications, embedded systems security, and distributed systems. He is a member of IEEE, ACM, the IEEE Computer Society, and the International Association of Cryptologic Research (IACR).

Berk Sunar Berk Sunar received his BSc degree in Electrical and Electronics Engineering from Middle East Technical University in 1995 and his Ph.D. 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, Sunar has joined Worcester Polytechnic Institute faculty. He is currently heading the Vernam Applied Cryptography Group. Sunar received the prestigious National Science Foundation Young Faculty Early CAREER award in 2002 and IBM Research Pat Goldberg Memorial Best Paper Award in 2007.