Designing an FPGA-Accelerated Homomorphic Encryption Co-Processor

David Bruce Cousins,Kurt Rohloff,Daniel Sumorok

Abstract—In this paper we report on our advances designing and implementing an FPGA-based computation accelerator as part of a Homomorphic Encryption Processing Unit (HEPU) co-processor. This hardware accelerator technology improves the practicality of computing on encrypted data by reducing the computational bottlenecks of lattice encryption primitives that support homomorphic encryption schemes. We focus on accelerating the Chinese Remainder Transform (CRT) and inverse Chinese Remainder Transform (iCRT) for power-of-2 cyclotomic rings, but also accelerate other basic ring arithmetic such as Ring Addition, Ring Subtraction and Ring Multiplication. We instantiate this capability in a Xilinx Virtex-7 FPGA that can attach to a host computer through either a PCI-Express port or Ethernet. We focus our experimental performance analysis on the NTRU-based LTV Homomorphic Encryption scheme. This is a leveled homomorphic encryption scheme, but our accelerator is compatible with other lattice-based schemes and recent improved bootstrapping designs to support arbitrary depth computation. We experimentally compare performance with a reference software implementations of the CRT and iCRT bottlenecks and when used in a practical application of encrypted string comparison.

Index Terms—Applied Cryptography, Hardware Acceleration, Homomorphic Encryption

1 Introduction

Recent advances in lattice-based Homomorphic Encryption (HE) have shown that it can be practical to securely run arbitrary computations over encrypted data [1], [2]. In addition to supporting encrypted computing, lattice-based HE schemes are attractive because they are post-quantum public-key schemes [3], meaning they are resistant to quantum computing attacks. Despite recent advances, HE is still not widely practical, partially because of computational bottlenecks in lattice-based HE schemes when run on commodity CPU-based computation devices. It would be valuable to accelerate the execution of core HE operations, possibly in a low-cost but computationally efficient and highly optimized hardware co-processor.

We discuss our experience designing and implementing an FPGA-based computation accelerator and co-processor for lattice-based Homomorphic Encryption (HE). We instantiate this capability in Xilinx Virtex-7 FPGAs that attach to a host computer through a PCI-Express port as Homomorphic Encryption Processing Unit (HEPU) co-processors.

Our design is intended to be adaptable and extensible, with a modular software architecture that supports rapid prototyping with alternative HE schemes, easier integration of the HEPU into broader computing infrastructures, and increased parallelism for efficient execution performance. While prior HE implementations have reduced absolute runtime, we focus on system engineering issues to improve overall performance when supporting broader system integration and practical application.

Although we experimentally evaluate performance with respect to the LTV scheme [4], which is itself a variant of the NTRU cryptosystem [5], our co-processor is also applicable to other lattice-based schemes such as the BGV [6] and GSW [7] designs. There has been some early work in the area of FPGA implementations of circuits that support lattice-based HE [8], [9], [10], [11], but little research into end-to-end workflows for a co-processor design.

Our strategy is to accelerate key computational bottlenecks common across HE schemes which can be much more rapidly executed on FPGA architectures. We particularly focus on the Chinese Remainder Transform (CRT) and inverse Chinese Remainder Transform (iCRT) for power-of-2 cyclotomics. We also accelerate lesser bottlenecks, including basic ring arithmetic: Ring Addition, Ring Subtraction and Ring Multiplication. Our approach yields experimentally 2 orders-of-magnitude improvement in runtime, as compared to a reference CPU-based implementation, for both the core CRT and iCRT performance bottlenecks and a real-world use case for encrypted ASCII string comparison.

The paper is organized as follows. In Section 2 we describe our representational LTV-based HE cryptosystem that motivates our design choices in the hardware accelerator. Section 3 discusses the design of the FPGA circuits for our lattice encryption primitives. Section 4 discusses the overall architecture of the FPGA based HE accelerator. Section 5 discusses communications between the host and hardware accelerator for more efficient integration of the HEPU into a computational workflow. Section 6 discusses experimental evaluation of the FPGA accelerator as compared to CPU based computation. Section 7 discusses related work. We provide a concluding discussion in Section 8.

- *D. Cousins and D. Sumorok are with Raytheon BBN Technologies, Cambridge, MA, 02138. E-mail: {dave,dsumorok}@bbn.com*
- *K. Rohloff is with the College of Computing Sciences, New Jersey Institute of Technology, Newark, NJ, 07102. E-mail: rohloff@njit.edu*

Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. FA8750-11-C-0098. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited.)

2 NTRU Representational Cryptosystem

We provide a brief overview of the LTV cryptosystem, defined with proofs of correctness and security in [4], to identify a workflow of primitive computational operations needed to support level homomorphic encryption. A more extensive reference deign and software implementation is discussed in [12].

For mathematical preliminaries, for the ring dimension n, we define the ring $R = \mathbb{Z}[x]/(x^n+1)$ (i.e., integer polynomials modulo $x^n + 1$). For the ciphertext modulus q, define the ciphertext space $R_q = R/qR$ (i.e., integer polynomials modulo $x^n + 1$, represented as length n vectors with mod-q coefficients). The plaintext space is R_p, so plaintexts are represented as length-n vectors of integers modulus p.

The modulus q is often very large for non-trivial computations. Representing a ciphertext c can require hundreds of bits of precision for every integer coefficient and incurs a very high computational overhead. We counter this with a double-CRT representation where a ciphertext c is a collection of t vectors $\{c_1, \cdots, c_t\}$ where vector c_i is an n-element vector of mod-q_i integers for $i \in \{1, \ldots, t\}$ [1]. This allows us to represent a large integer q by a collection of smaller residues, that can each fit within a conventional 64-bit integer.

We convert data into the double-CRT representation with the core computational primitive called the "Chinese Remainder Transform" (or CRT for short) [13]. The CRT is a modulo arithmetic variation of the Discrete Fourier Transform (DFT) [14]. The primary design decision is to exclusively support power-of-2 cyclotomics so that plaintext and ciphertext vectors are of length n which is a power-of-2. This design decision enables more efficient hardware and software implementation at the expense of less efficient bootstrapping operations (needed to provide arbitrary-depth Fully Homomorphic Encryption (FHE) capabilities).

To support key generation and encryption operations, we often need to generate samples from a discrete Gaussian distribution over R_q with a distribution parameter r, denoted as $\chi^r_{R_q}$. We do this using techniques in [15] to make n independent samples from a scalar discrete Gaussian distribution χ^r with a distribution parameter $r = 6$, and use the CRT to convert to the double-CRT representation.

With these preliminaries, the following functions are then defined in our scheme:

KeyGen: Sample $f \in \chi^r_{R_q}$ such that $f = 1 \bmod p$ and f is invertible modulo $q_1, \cdots, q_t$. Sample $g \in \chi^r_{R_q}$. The double-CRT representations of f_i^{-1} (modulo q_i) is the mod-q_i inverse of f_i. Output the secret key $sk = f$ and public key $pk = h = g \cdot f^{-1}$.

Enc($pk = h, \mu \in R_p$): Sample $e, s \in \chi^r_{R_q}$. Output $c = p \cdot e \cdot h + p \cdot s + \mu$, in double-CRT representation.

Dec($sk = f, c \in R_q$): Compute $\bar{b} = f \cdot c$ and lift it to the integer polynomial $b \in R$ with coefficients in $[-q/2, q/2)$. Output $\mu = b \bmod p$.

From [4], the scheme natively supports the homomorphic operations:

EvalAdd(c_0, c_1): Output $c = c_0 + c_1$.

EvalMult(c_0, c_1): Output $c = c_0 \cdot c_1$.

2.1 Supporting Operations

We improve upon this basic HE scheme with HEPU-accelerated supporting operations:

- Key switching to reduce the complexity of key coordination during coordination [6].
- Modulus reductions to decrease noise growth and increase the size of computation supported without compromising security [6].
- Ring switching to keep ciphertexts smaller and computations more efficient [16].

Key switching converts a ciphertext of degree at most d, encrypted under secret key f_1, into a degree-1 ciphertext c_2 encrypted under a secret key f_2. We perform this operation after every EvalMult to ensure the ciphertexts are decrypted by the same secret key that is independent of the depth of computation performed. This requires a "hint":

KeySwitchHintGen(f_1, f_2): Output $a_{1\to 2} = (p \cdot e+1) \cdot f_1^2 \cdot f_2^{-1}$ for a Gaussian-distributed e.

We define key switch as:

KeySwitch($c_1, a_{1\to 2}$): Output $c_2 = a_{1\to 2} \cdot c_1$

Modulus reduction reduces noise growth in ciphertexts. It is performed after every KeySwitch operation after the EvalMult operations and converts a ciphertext c that is a double-CRT collection of t vectors $\{c_1, \cdots, c_t\}$ into a ciphertext c' that is a collection of $t - 1$ vectors $\{c'_1, \cdots, c'_{t-1}\}$. We do this by adding a small integer multiple of p that is congruent to $-c \bmod q_t$. This ensures that the underlying noise remains small, that the plaintext remains unchanged, and that the resulting ciphertext is divisible by q_t. Then we can divide both the ciphertext and modulus by q_t, which reduces the underlying noise term by a q_t factor as well. To implement this we use the independent value $v = (q_t)^{-1} \bmod p$, which we compute in advance:

ModReduce(c):

1) Compute $d = c \bmod q_t$.
2) Let $d' = (vq_t - 1) \cdot d \bmod (pq_t)$, with all of d''s entries in $[-pq_t/2, pq_t/2)$.
3) Let $d_t = c + d' \bmod q$.
4) Output $(d_t/q_t) \in R_{(q/q_t)}$.

Computing $d = c \bmod q_t$ is most efficiently done in coefficient form, which means inverting the mod-q_t CRT on the vector of mod-q_t components of c. Computing d' is done by multiplying the coefficients of d by the fixed scalar $(vq_t - 1)$ modulo pq_t. Adding d' to c is done by computing the double-CRT representation of d' (i.e., applying each mod-q_i CRT to d'), and adding it entry-wise to c's double-CRT representation. Computing d_t/q_t is done by multiplying every mod-q_i component of d' by the fixed scalar $q_t^{-1} \bmod q_i$, which is computed in advance.

Ring reduction maps a ciphertext from ring n to smaller ring $n' = n/2$ by first key-switching the ciphertext a "sparse" secret key sk, whose only nonzero entries in the evaluation domain are at even indices.

RingReduce(c): Output $\{c'_1, \cdots, c'_t\}$ for each $i = 0, \ldots, w - 1$ where c'_i consists of the even entries of c_i.

We group the key switching, modulus reduction and ring reduction operations into a "Composed" Evaluation Multiply (CompEvalMult) operation which includes a sequence EvalMult, KeySwitch and ModReduce every time

TABLE 1
Bits of moduli q_i vs. ring dimension for $p = 2$.

Ring dimension n	512	1024	2048	4096	8192	16384
Bit length $\log_2(q_i)$	44	45	47	48	50	51

CompEvalMult is called. From [12] running a modulus reduction operation results in a more secure ciphertext, but ring reduction reduces security. Starting from a fresh ciphertext c with security level δ, we perform CompEvalMult operations until the security of running ring reduction would result in a ciphertext at least as secure as the original ciphertext c. The equation we use to estimate a bound on δ and schedule RingReduce is seen in Subsection 2.2.

Our scheme also supports a recent more efficient bootstrapping operation [17] called after t CompEvalMult operations to obtain a refreshed ciphertext and support arbitrary depth computation. We do not target our HEPU design to optimize a full bootstrapping execution because of a core bootstrapping step that requires a larger ciphertext modulus than we can support in the HEPU hardware. However, our HEPU accelerates internal bootstrapping operations including CRT and iCRT bottlenecks.

2.2 Parameter Selection

From a correctness perspective in [4], we choose the smallest modulus q_1 so that it satisfies the expression $q_1 > 4prn^{1/2}w$ to ensure successful decryption as the last double-CRT ciphertext modulus after all possible modulus reduction operations have been performed. The parameter $w \approx 6$ represents an "assurance" measure for correct decryption. We select the remaining $\{q_2, \ldots, q_t\}$ such that $q_i > 4p^2r^5n^{1.5}w^5$ so modulus reduction by a factor of q_i sufficiently reduces the noise after a (CompEvalMult) operation to allow successful decryption after the modulus reduction operations. For all reasonable parameter selections, $q_1 < \{q_2, \ldots, q_t\}$, so the size of $4p^2r^5n^{1.5}w^5$ is critical in the HEPU design. Table 1 shows the maximum number of bits required to represent $\{q_2, \ldots, q_t\}$ for varying ring dimensions for $p = 2$. We can attain potentially large performance optimizations in the HEPU because the moduli use less than 64 bits.

To ensure security, we use the standard "root Hermite factor" δ as the security parameter. Experimental evidence [18] suggests that $\delta = 1.007$ requires roughly 2^{40} core-years on recent Intel Xeon processors to break. This setting is currently believed to be adequately secure and provide at least 80 bits of security [19]. Using estimates from [20], [21], we need to ensure that $n \geq \lg(q_1 \cdots q_t)/(4\lg(\delta))$. Based on tradeoff analysis in [12], we would need to use ring dimension $n = 16324$ to support depth $d = 13$ computations securely without bootstrapping. A depth $d = 13$ is adequate for many practical applications, such as secure encrypted ASCII text comparison, which we explore experimentally below.

Recent results have shown weaknesses in the NTRU scheme when there is a large composite ciphertext modulus q [22], [23]. In particular, there are known polynomial time attacks against this NTRU scheme when q is larger than $2^{\sqrt{n}}$, which provides an additional constraint on the choice of the ring dimension and modulus. However, despite these recent findings, security of our experimental settings and the validity of our acceleration are not impacted, and these results do not impact the security of other schemes the HEPU could support such as the BGV [6] scheme, among others.

3 FPGA Based Circuits for HE Modulo Arithmetic Ring Operations

3.1 Pipelined VHDL for Fast Modulus Arithmetic

We implemented elementary operations in the form of ring operations in Matlab and Simulink using the Fixed Point toolbox. Matlab programs are coded into Simulink discrete models, adding pipeline stages in the process (modeled as unit delays and described in detail below). The resulting Simulink is compared against the initial Matlab implementation to verify correctness. VHDL code is then generated automatically using Mathworks' HDL Coder tool. The resulting VHDL can be validated against the original models using the HDL Verifier tool.

Software implementations of the modulus operation usually use some form of trial division to determine a remainder. Implementing efficient modulus arithmetic requires special numerical algorithms, such as Montgomery Reduction [24], and Barrett Reduction [25]. These algorithms avoid naïve division by q_i by scaling the integers so divisions can be performed by a powers of 2, requiring only simple bit shifts.

We implement the CRT and iCRT operations as an EvalMult of the input with a "CRT twiddle table" vector of length n, followed by a pipelined Number Theoretic Transform (NTT) based on an Fast Fourier Transform (FFT) circuit [26]. This NTT uses modulo integer instead of complex arithmetic and uses a standard radix 2 "Butterfly" operations which consists of one addition, one subtraction and one multiply, all modulo the residue q_i. Thus we need to implement modulo subtraction in hardware as well. Taken together, the CRT twiddle EvalMult and the NTT transform an input ring into a form that allows an equivalent of efficient polynomial multiplication, by simply performing an EvalMult of the transformed inputs. The iCRT is a similar combination of circuits, though connected in reverse order (as discussed below.)

All the circuits we developed were heavily pipelined to increase the maximum synthesizable clock speed, i.e. there is a pipeline of many stages of arithmetic or logical operations separated by clocked registers. Additionally, the circuits are designed to operate on a stream of inputs. For example, with our `RingAdd`, `RingSub` and `RingMul` circuits, one element of each input ring is fed as input into the operation, and one output is generated each clock cycle. Each circuit is built with an input port which will pass a modulus table index i along with the two input values, selecting a particular modulus q_i for the operation. This requires us to fix the values of q_i in a lookup table at FPGA compile time. In operation, we concatenate all values of each double-CRT ciphertext into their respective input streams along with the appropriate t index, and thus can achieve a steady state where one modulus operation is performed each clock cycle. We process streams of input with several

thousand entries, so the extra processing latency needed to fill the pipeline is trivial as compared to total execution time.

3.1.1 Pipelined `RingAdd` and `RingSub` Circuits

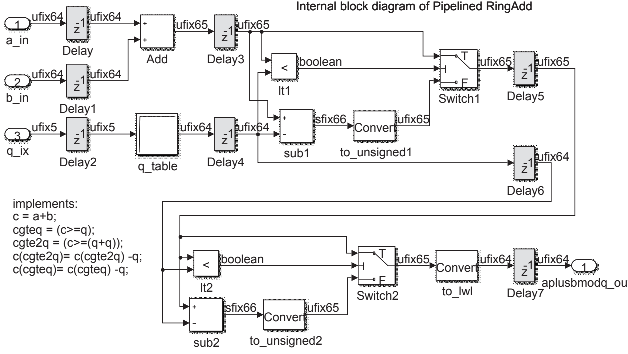

Fig. 1. Structure of `RingAdd` streaming modulo addition circuit implemented using Simulink fixed point arithmetic for VHDL synthesis

When the inputs to the modulo add and subtract operations are bounded by the modulus q_i, the two operations simplify dramatically and do not require Montgomery or Barret reductions.

Figure 1 shows the Matlab code for for EvalAdd and the resulting `RingAdd` pipelined Simulink model. The circuit has a latency of four clock cycles. The modulo subtraction operation and `RingSub` circuits are designed similarly with the addition of some conditional logic in the pipeline to handle "negative" modulo values.

3.1.2 Pipelined `RingMul` Circuit

Modulo multiplication is a much more complicated operation than either addition or subtraction, even if the input multiplier and multiplicand are bounded by q_i. This is because the range of the output of the latter two are bounded by a small integer multiple of q_i, and can be adjusted within the range of $[0 \dots q_i - 1]$ by simple comparisons and subtraction of q_i. For multiplication the product is approximately twice the bit width of q_i. Furthermore, large bit width multiplications can severely restrict the resulting clock rates of the circuits. To address these constraints, we adopted an interleaved modular multiplication based on a generalized Barrett reduction [27]. This multiplier has the following properties:

- Long words of bit length L can be represented by N smaller words of bit length S (i.e. four 16 bit words to represent a 64 bit modulus).
- The multiplication is performed in N stages, where each stage performs one modulo multiplication that is $L + S$ bits long. The stage can be pipelined to perform one modulo multiply per clock cycle.
- Each stage has a Barrett modulus performed on the partial product, which reduces overall bit growth of the partial products to $L + S$. Each stage requires 3 multiplies, and all divisions required by the Barrett algorithm are implemented as simple bit shifts.
- We chose to implement this circuit with $L = 64$, $S = 16$ resulting in an $N = 4$ stage implementation as shown in Figure 2.

The algorithm from [27] is in the form of a loop that executes N times. We unroll that loop and perform the operations in a pipeline N stages long.

Figure 2 merits discussion to clarify the modular multiplication. Data flows into the circuit from left to right, through the $N = 4$ stages of the `RingMul` modulo multiplier, each shown by the large component box labeled with `pipeline_Barrett_Mult` $\star[0 \dots 3]$. Each stage is pipelined and takes 47 clocks for the data to pass through. On each clock, a new set of input data is presented, so that at any point in time there are 47 inputs in various stages of being processed by each block.

Below each stage is a set of 47 step shift registers (indicated by z^{47}) to propagate the inputs to the next stage. Note that the shift registers ensure that data arriving on each clock stay synchronized throughout the pipeline. The input `x_in` is passed subsequently to each stage. The other input `y_in` is divided up into 16 bit words, with the most significant word passed to the first stage, the next 16 to the second stage and so on. Finally, each staqe is identical, incrementally accumulating the contribution of the product of input `x_in` with one of the 16 bit words derived from input `y_in`. The input to each stage, `z_in` is an 80 bit accumulator that is initially set to zero, processed and then sent as `z_out` to be used as the `z_in` for the next stage. Each stage also has an output value `out`, which is ignored in all stages but the last one, where it provides the final output $(x * y) \mod q_i$.

The circuit can support a full tower of multiple moduli since all modulus related variables are stored in lookup tables (i.e. constant coefficients mu_i, moduli q_i and bit shift sizes as discussed below) and are indexed by the input `q_in` in the same manner as our `RingAdd` circuit. A constraint of our Mathworks tool-chain is that all integer values be at most 128 bits wide. Our selection of the algorithm word sizes ensures that no single operation output exceeds this limit. The details of a single stage of the modulo multiplier are shown in Figure 3. This stage performs one iteration of the loop from the algorithm in [27]. Readers are referred to that source for full details of the algorithm. The circuit is heavily pipelined and consists of seven internal stages:

1) A left shift of `z_in` and simultaneous product of `z_in` and 16 bits of `y_in`;
2) The accumulation the shifted `z_in` and the product and a subsequent right shift by a specific number of bits determined by the bit size of the modulus selected;
3) Another 128 bit product of the shifted result and a pre-computed value `mu_i` based on the modulus;
4) Another right shift (based on the modulus) to form the term q' in the algorithm;
5) A final multiply of q' and the selected modulus q_i;
6) The subtraction of this product from the accumulated z of the second stage, generating `z_out`;
7) A final stage that adjusts the value of `out` if `z_out`$\leq q_i$.

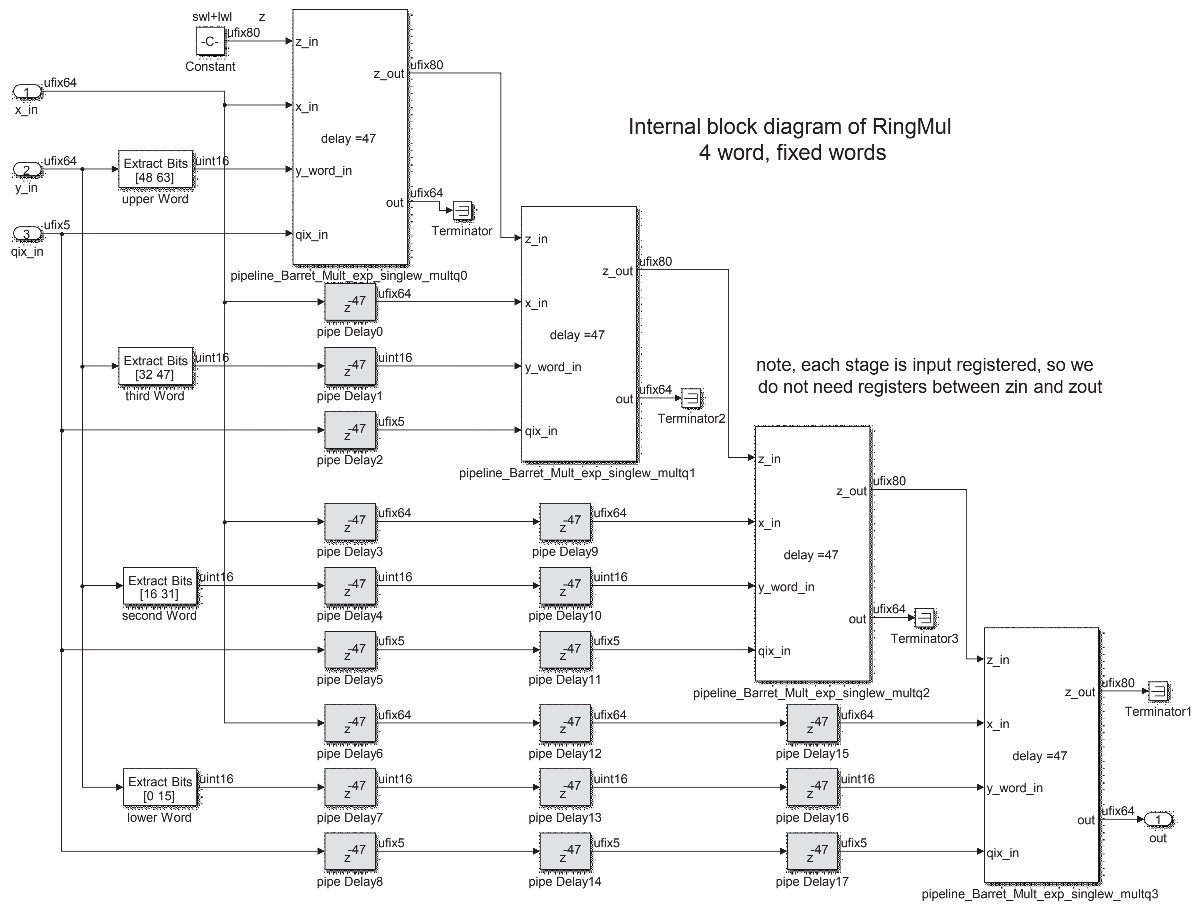

Fig. 2. The top level structure of the four-stage Barrett 64x64 bit `RingMul` streaming modulo multiply circuit

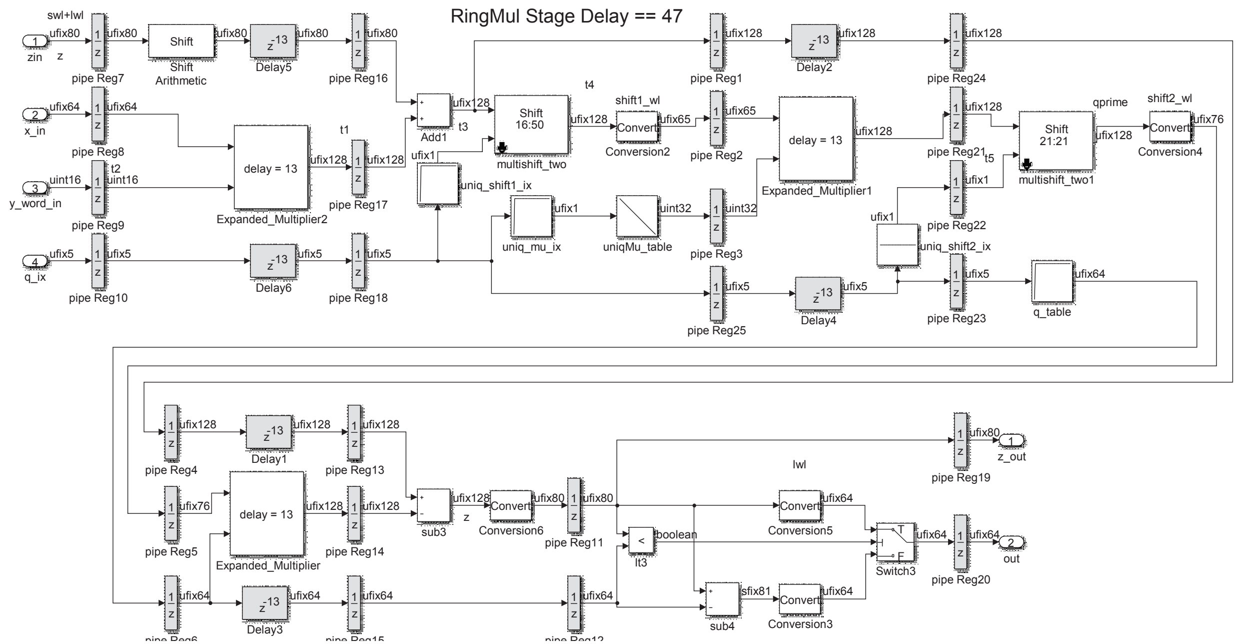

Fig. 3. A single stage of the `RingMul` modulo multiply

The three 128-bit multipliers in Figure 3 are themselves further pipelined in order to operate at a faster clock rate than a non-pipelined version would. These components were re-implemented as four parallel 32×32 bit products of combinations of the upper and lower 32 bits of the two inputs, with a pipelined accumulation of partial sums. Adding additional pipelines of length four, both before and after each resulting smaller product block allowed the VHDL compiler, and place and route optimizer to map these product blocks onto multiple hardware multipliers in a distributed fashion. The resulting the `RingMul` circuit will run at clock speeds in excess of 350 MHz. However, memory access constraints limit the operation of the circuit to a much lower rate, as will be seen later. The implementation is agnostic to the particular FPGA technology used, and is tunable for different FPGA technologies. The implementation results in total of 47 pipeline stages for each of the four multiplier stages, for a total pipeline latency of 188 clock cycles for the Barrett modulo multiplier. Considering our system operates on large vectors of data, and that each clock cycle can support any of the 32 moduli defined in the tower, the impact of the pipeline latency is minimal.

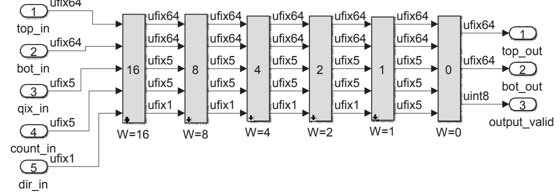

Fig. 4. Top level structure of the pipelined `NTT` circuit (size $N = 64$)

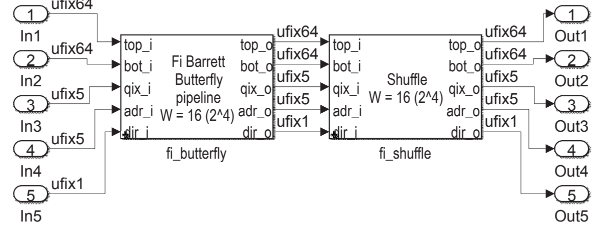

Fig. 5. The structure of a single stage in pipelined `NTT` circuit ($W = 16$)

3.2 VHDL Fast Forward and Inverse CRT

3.2.1 Pipelined `NTT` Circuit

We developed a pipelined circuit for performing a fast CRT. We implemented the NTT using a standard pipeline decimation in frequency FFT architecture known as the Radix 2, Multipath Delay Commutator [28]. The fundamental structure of the Simulink model that performs the NTT is identical to a complex version that computes the standard FFT. The only difference is the use of modulo vs. complex arithmetic in the radix 2 butterfly, and the replacement of the complex roots of unity (known as the FFT twiddle factors) with the modulus roots of unity. Specifically, in an full N point FFT, each stage of the FFT requires a subset of the N complex roots of unity for use in its specific butterfly stages according to a fixed relationship of input index $k \in [0 \ldots N-1]$. In the first stage, all N twiddle factors are required (i.e. $t(k) = \exp(-2j\pi k/N)$). Consequent stages each require half of the twiddles used in the previous stage (as will be seen later). Similarly, the NTT uses N twiddles t s.t. $t^N \bmod q = 1$ and $t^k \bmod q \neq 1$ for $k < N$. There is a one for one mapping of complex to modulo roots of unity. The flexibility of our model-based approach allows us to debug the code using complex input and complex butterflies, and then use the same exact structure for the NTT with only a change to the arithmetic in the butterfly block. One additional design consideration, is that while the forward and inverse NTT each have identical structures, each requires a different set of twiddles (corresponding exactly to the differences between the forward and inverse FFT, as detailed in [26]). Thus, an input logic line `dir_in` is propagated through the circuit to select between the two sets of twiddle tables within each butterfly circuit.

The design trades off area for processing speed on the chip, in that a pair of input values are read in and a pair of output values written out each clock cycle, with no stalling of the pipeline. Figure 4 shows the circuit for an $N = 64$ point transform, where $\log_2(N)$ radix 2 Butterflies are required, though the last butterfly does not require multiplies. This circuit simply requires the addition of a new stage for every power of two added to the ring size. Each stage is written so that one parameter W (related to a power of 2) controls the size of all tables and delay lines in each stage. A single stage is shown in Figure 5, consisting of a computation section (left) and a shuffle block (right). The computation section is shown in Figure 6 consisting of a `RingButterfly` and switching logic driven by `dir_in` to select the appropriate forward or inverse twiddle table (thus, the same circuit computes either a forward or inverse NTT). The `RingButterfly` is a modulo arithmetic version of the FFT butterfly, and is shown in Figure 7, consisting of a `RingAdd`, `RingSub`, and a `RingMul`, appropriately pipelined. Inputs `a_in`, `b_in`, `c_in`, and `qix_in`, are the two inputs to the butterfly, the current twiddle factor and the selector for the modulus to use, respectively. In addition to the computation block, each stage has a shuffle block as shown in Figure 8. Note that $3/2N - 2$ delay elements are required for the shuffle blocks. The shuffle block enables on-the-fly reordering of the input vectors, and is a direct implementation of the commutator detailed in [28]. A key point is that this implementation results in the fastest possible computation rate of the `NTT`, with one pair of output samples being generated every clock cycle. By concatenating input vectors sequentially, we keep the pipeline full without stalling.

3.2.2 Hand Optimizations

The only difference between the forward and inverse CRT is whether the `NTT` is pre- or post-multiplied with a special "CRT Twiddle vector" (different from the NTT/FFT twid-

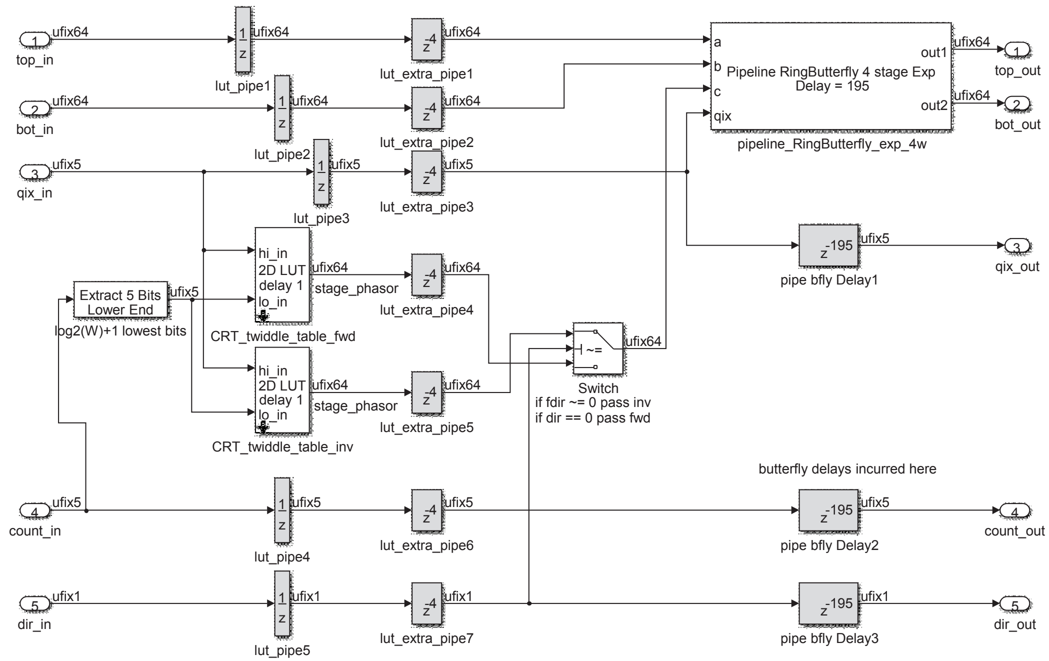

Fig. 6. The computation section of a single stage in the pipelined NTT circuit ($W = 16$) showing forward and reverse twiddle lookup table selection by dir_in and twiddle indexing by count_in. These inputs are fed to a standard NTT modulo arithmetic butterfly circuit

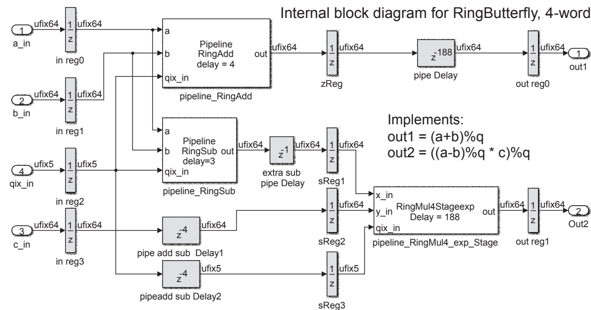

Fig. 7. Pipelined RingButterfly circuit showing implementation of the butterfly formulas and modulus selection by qix_in

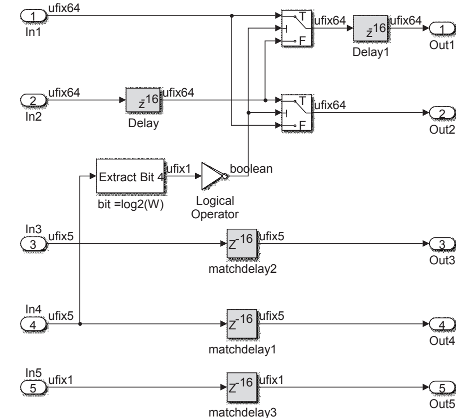

Fig. 8. Shuffle section of a single stage in pipelined NTT circuit ($W = 16$). The top two inputs and outputs are the data top and bottom inputs, the next three pass qix_in, count_in, and dir_in to the next stage. count_in is used to drive the shuffle selection circuit

dles). We programmed our VHDL wrapper that feeds the data to the NTT and RingMul circuits so that this pre- or post-multiplication is done in a pipelined manner. The input and output data needs to be presented to the circuit in two parallel streams, with the top stream containing the first half of the input vector and the bottom stream containing the second half. The resulting output is in bit reverse order. Rather than implement this in Simulink we incorporated

these data manipulations into the VHDL wrapper around the `NTT` portion of the CRT. These wrappers use double buffering to keep the pipeline full.

Shortcomings of this design is that it utilizes a large FPGA area, and each stage has its own "NTT Twiddle memory". In practice, every stage's twiddle memory is composed of exactly the same even entries in the twiddle memory preceding it in the pipeline. Duplicating this memory in each stage is inefficient.

Each stage's Twiddle memory is a Read Only Memory (ROM) table, implemented with FPGA block RAM. The first stage has the largest table, and each successive stage's twiddle table is half the size of the previous stage's twiddle table. To reduce wasting critical FPGA block RAM, the first four stages share a table. One reason this is possible is that the twiddle tables have been designed so each twiddle table contains the same values as the values at the even addresses in the previous stage's table. The twiddle table for the first stage contains the values for all the other tables. The reason each stage needs its own table, however, is that each stage needs to access its table simultaneously in order for the CRT module to achieve the desired data throughput. Fortunately, the FPGA Block RAM primitives are Dual Port, which means that it possible to simultaneously read from two independent addresses. By clocking the dual port block RAM at twice the rate (200 MHz) of the rest of the CRT logic (100 MHz), we are able to construct a virtual quad port block RAM. For each cycle of the CRT clock (100 MHz), two addresses are presented to each port of the large twiddle table, one on each cycle of its faster clock (200 MHz). As a result, the large twiddle table is able to sustain a throughput of four independent reads per clock cycle, and saves the resources required by the twiddle tables for stages 2048, 1024, and 512, as shown in Figure 9. After these savings, our design still uses 98% of the available FPGA Block RAM (BRAM) resources of our selected FPGA.

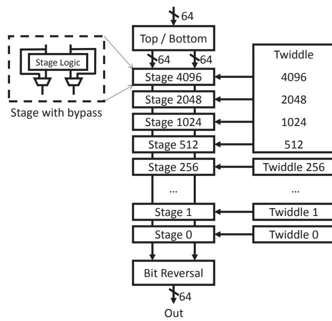

Fig. 9. VHDL wrapper that performs required I/O reordering, sharing of twiddle ROM and stage bypass in the NTT.

3.3 Pipelined `RingRound` Circuit

In addition to our implemented ring and CRT operations, we implemented a function used in our Composed Eval Mult (CompEvalMult) called `RingRound`. The CompEvalMult function is implemented as a C function that calls several FPGA primitives. Since `RingRound` requires modulus operations using $2 * q_i$ instead of q_i it is necessary to implement it in hardware with its own `RingMul` dedicated to this new set of moduli. Figure 10 shows the Simulink Model of `RingRound` which consists of a modified `RingMul` circuit (with a modified set of moduli $2q_i$), and a pair of operations selected by the range of the result which ensure the output is bounded within the appropriate range. The operations are all performed with pipelined circuits.

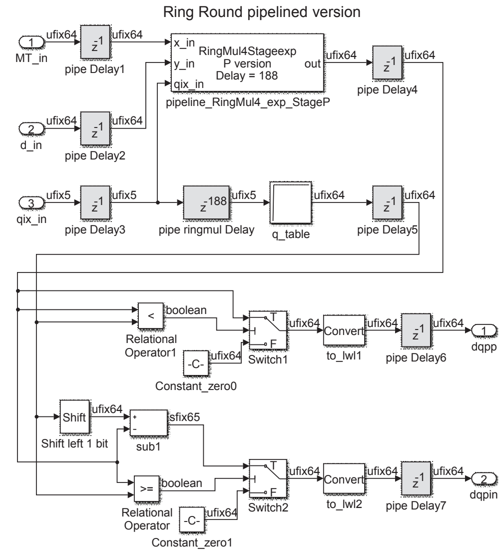

Fig. 10. Structure of `RingRound` circuit

3.4 Further VHDL Optimization

Xilinx advises that selecting synchronous vs. asynchronous reset in the Simulink HDL generation parameters will result in VHDL that is mapped more efficiently onto the the DSP48E blocks on the Virtex 7 FPGA, resulting in faster clock rates [29]. Most of our circuits utilize lookup tables, both for storing the moduli q_i and for storing various `NTT` twiddle table entries. Our initial implementation of the table lookup using the Simulink `Lookup` block maps the resulting ROM directly into FPGA gate circuitry. This can increase the place and route time drastically for very large tables, and also can result in slower circuits. Placing an additional delay line, with a "ResetType = none" HDL property lets the Xilinx tools map the table to block ram in the FPGA, for more efficient resource utilization [29].

A possible way to increase performance is to increase clock rates. It would be possible to increase the clock rate with some re-design, but this is nontrivial engineering task that would require additional design effort. Cycle time is constrained by the critical path, which is the path with the longest delay between two flip flops. The critical path depends on many factors, including fanout, routing density and the amount of logic between flip flops. The fact that we are able to run the lookup tables at twice the clock rate does not reduce the critical path in our design, and even further constrains the placement and routing.

3.5 FPGA Hardware Selection

We identified that we needed an off-the-shelf FPGA with evaluation board that provides a large number of hardware

multipliers and large amount of block RAM on the chip so that accelerator would be both capable and low-cost. We selected the Virtex 7 VC707 evaluation board as a solution that met our FPGA sizing requirements. Our largest ring size of 2^{14} requires 87% of DSP48 blocks and 98% of block RAM on the board's Virtex 7 485T chip. This evaluation board has a PCI Express (PCIe) interface for connecting with the host PC motherboard when hosted in the same PC chassis. Additionally, during development, the on-board DDR3 memory was used for storage of encrypted variables, and high speed Ethernet was used for connecting the board with multiple PC hosts in the development environment.

4 HEPU ARCHITECTURE

4.1 System Architecture

The HEPU was designed to operate as an attached processor to accelerate HE primitive operations. We wanted flexibility to operate as either a stand-alone network attached processor or as a PCIe-based device in a host computer. This gives the most flexibility in the use of the device, but PCIe-based is more efficient.

We designed and developed an attached processor in which a software programmable micro controller would manage I/O communications with the host via Ethernet or PCIe memory map, manage on board data storage in the form of an encrypted register file, and manage data transfer to and from the HE primitive modules in as efficient manner as possible. We used the Xilinx Platform Studio Microblaze soft core processor and AXI4 interconnect architecture to implement the HE processor. Figure 11 shows a system block diagram of the FPGA system for both the Ethernet and PCIe hosted configurations. The Xilinx platform studio enables us to implement our HE primitives as streaming co-processors on the AXI bus. An AXI4 lite bus is used to set control parameters of our Ring operation circuits, such as ring size and tower size.

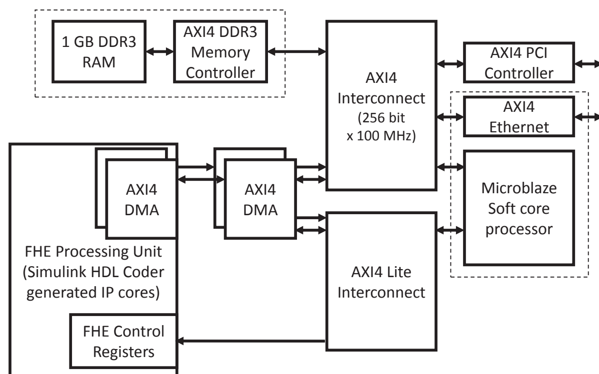

Fig. 11. System Block Diagram showing major components and the AXI4 interconnect for the various implementations of the HEPU. The components inside the dotted boxes are included for the network attached co-processor build and removed for the PCIe hosted build.

A high level diagram of the PCIe hosted HE accelerator architecture is shown in Figure 12. At the top of the diagram

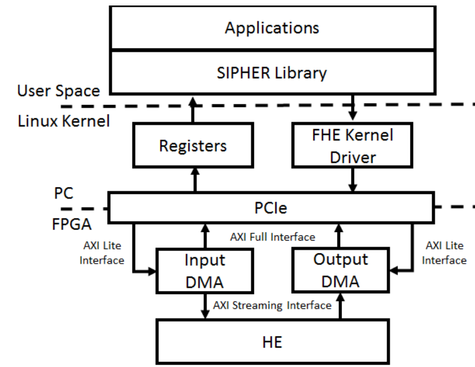

Fig. 12. HE hardware accelerator system architecture

is a block labeled "Applications". This represents user applications that use the HE accelerator by communicating with the HE Kernel Driver, which exposes a Linux "character device". The user application sends commands and data to the HE Kernel driver by writing packets of data to the HE character device, and likewise the user application receives responses and status via data packets from the HE character device. The packet packed-based communication protocol allows the user application to be agnostic to whether the HE Accelerator is on a local PCIe bus, on separate network-connected device.

The core HE Kernel Driver is written in C. The same code runs in the Linux kernel or on a soft-core Microblaze processor inside the FPGA depending on if it is configured for PCIe or ethernet operation respectively. This portable driver code instantiates a set registers, in memory, which store the HE input and output data, as well as intermediate results. When the code runs on Microblaze, these registers exist in the FPGA board's DDR3 RAM. When the code runs in the Linux Kernel, these register exist in PC memory.

In Figure 12 the (AXI Full Interface) arrow between the Input DMA module and the PCIe interface originates at the DMA module, even though the data flows in the other direction. This is because the DMA modules are both a Master, meaning they initiate the data transfers. One DMA Master (Input) reads from PC memory, and the other (Output) writes to PC memory. The DMA modules also have an AXI slave port, for control, which is written to and from by the PC (via the PCIe interface). Communications packets (described below) are sent between the application and the Kernel Driver. In the PCIe implementation this is a direct memory socket whereas in the Ethernet implementation data packets are sent via Gigabit Ethernet rather than to mapped PCIe memory. We do not describe the Ethernet implementation in detail because it was used for development purposes only.

4.2 Control of HE Circuits

Figure 13 depicts the interior of the HE Processing Unit module. An HE Primitives module is fed by two 256-bit wide data streams (each containing four sequential 64 bit words) in the AXI clock domain, but internally processes two 64-bit streams in a different clock domain called the "math clock" domain. The conversion from 256 to 64 bit words are done using a simple de-multiplexing circuit,

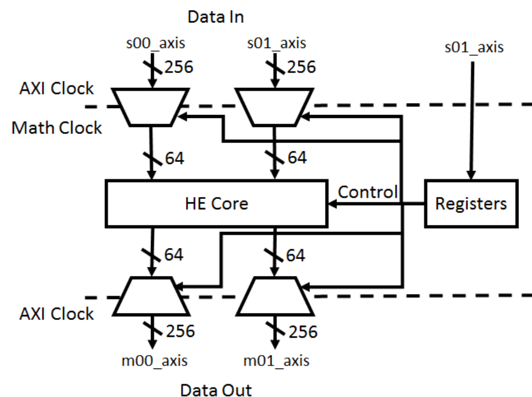

Fig. 13. Integration of HE Core containing primitives with the AXI stream I/O data streams

converting one 256 bit word into four 64 bit words clocked sequentially into the HE domain). The large I/O bitwidth was selected to maximize the data in and out of the HE Primitives module. In this design, the math clock runs at 100 Mhz, and the AXI clock runs at 125 MHz. Separating the HE math clock domain from the PCIe clock domain give design flexibility. The frequency of the AXI clock is determined by the speed of the PCIe interface. The math clock frequency can be set to the maximum supportable HE Core logic clock frequency. We are conservative with the math clock frequency because we have internal (NTT twiddle) memories that are run at twice this frequency. As the HE core circuits are improved and become more efficient, we will be able to increase the rate of the math clock. The AXI clock may also be doubled if we move to a PCIe Gen 3 architecture, something we were not able to do within the project budget. When running with Ethernet, the main AXI4 interconnects remain a 256 bit bus connecting the DDR3 RAM with the HE core. In this mode, the I/O rate into and out of DDR3 memory limits the overall processing speed of the system. The memory controller used was the standard Xilinx Virtex 7 IP block configured for 256-bit wide data access to the DDR3 RAM present on our VC707 evaluation board.

Figure 14 shows the internal structure of the HE core and how the inputs and output streams are routed to the various HE primitive functions. The core HE block is configured by control lines and performs one of several operations on the data using the `RingAdd`, `RingSub`, `RingMul`, `NTT`, or `RingRound` circuits. For forward CRTs, the `NTT` block is fed with input In0, and the `NTT` module's output is then multiplied with input In1 using the `RingMul` module. Similarly, for inverse CRTs, inputs In0 and In1 are fed into the `RingMul` module, and that output is fed into the `NTT` module. This architecture conserves FPGA resources since the `RingMul` inputs for CRT operations (the CRT twiddle vector) arrive from In1 (via DMA from DDR3 ram or PC memory depending on hosting) and do not need to be stored locally as ROM tables in the FPGA. This reduces the amount of FPGA block memory used by the design. Each of the operations in Figure 14 receives their input data pipelined first by ring elements, then by tower indices. Thus all input and output for a complete double CRT representation is streamed in one operation. The CRT operation requires slightly different interfaces that change the order of the input and output data.

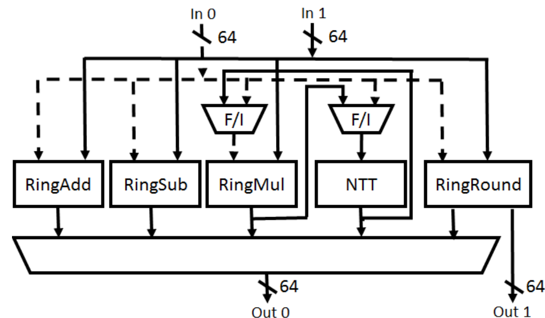

Fig. 14. HE Core

TABLE 2
Application Level Control Protocol Keywords

Keyword	Function
LOAD	Transfer the contents of the message (ASCII) into a particular Input register.
GET	Request the contents of an output register to be loaded into a message buffer and sent to host.
STATUS	Generates a short report on the FPGA board console for debugging register contents and program loaded.
PROG	Loads a sequence of operations to be performed on the register data.
RUN	Starts a software Finite State Machine to run the stored program to completion.
CRT, ICRT, CEM	A single command that will LOAD two registers, perform a forward CRT, inverse CRT or CompEvalMult on them and GET the output.
RESET	Resets the system to its original state.

5 HEPU Communication Protocols

A Linux Driver is used to interface with the user Application code. It translates the Application Level text interface messages to a binary format message and handles all I/O to the FPGA board. The software controlling the system on the Microblaze is written in C code. Registers are allocated out of the PCIe Kernel memory. All Microblaze code is executed in Linux Kernel Space.

5.1 Application Level Programming Interface

The communication protocol between the user application code and Linux Driver is ASCII message based. Messages can contain multiple instructions separated by a `newline` sequence. Each instruction starts with a keyword that defines the rest of the instruction format. The list of allowed keywords are shown in Table 2. The user application can string commands together to program the FPGA to operate on several pieces of encrypted data in the form of an assembly language. The FPGA accelerators assembly language has the syntax shown in Table 3.

An example simple program in now given in Algorithm 1. The program first moves encrypted data from input register 0, to scratch register 0, then repeats the process for a second input variable to register 1. It then computes a

TABLE 3
Table of Available Opcodes for Application Program

Opcode	Description
LOAD	Moves data from an input register to scratch register, all active tower elements are moved.
STORE	Moves data from a scratch register to output register, all active tower elements are moved.
RADD	Sets up DMAs of the two input and one output registers to the RingAdd circuit. All active tower elements are processed in one large data flow.
RSUB	Same as RADD, except the RingSub circuit is the target of the DMAs.
RMUL	Same as RADD, except the RingMul circuit is the target of the DMAs.
CRT	Sets up DMAs of the one input and one output registers to the CRT circuit. All active tower elements are processed in one large data flow.
ICRT	Same as CRT except an inverse CRT circuit is used.
EMULC	Executes a CompEvalMult, in software which in turn executes several ring primitives. Output register is one tower smaller than the input registers.

RingAdd and RingMul using the two inputs, and storing the result in scratch registers 2 and 3. It then stores those three results in output registers 0 and 1.

```
R0 = LOAD(In0)
R1 = LOAD(In1)
R2 = RADD(R0,R1)
R3 = RMUL(R0,R1)
Out0 = STORE(R2)
Out1 = STORE(R3)
```

Algorithm 1: Sample HEPU Program

Once the program is loaded, typical system operation would be for the user to execute two LOAD commands to load the contents of input registers 0 and 1 with encrypted data (the encryption being done on the secure host). The user then executes a RUN command to allow the Homomorphic operations to be run on the unsecured FPGA processor. Then subsequent calls to GET commands will transfer the resulting encrypted result data back to the host. Finally decryption would be done on the secure host.

6 Performance Evaluation

We evaluated the performance of our FPGA co-processor of the bottleneck CRT operation, for the core Composed Evaluation Mmultiply (CompEvalMult) and a broader use-case for encrypted string comparison. We made apples-to-apples experimental comparisons between these operations implemented in:

1) Native Matlab run in the Matlab interpreter.
2) Compiled C run on a commodity Linux desktop.
3) Our PCIe-based FPGA co-processor accelerator with operations called from Matlab.

We also compare performance with recent comparable work on LTV implementations and lattice crypto hardware accelerators, inclusive of [10], [11], [30], [31]

We ran our Matlab and compiled implementations on a single core on a server with 2.1GHz Intel Xeon processors and 1TB of RAM in a CentOS environment. Although we had access to many resources, we used at most 10 GB of memory. We on purpose do not include FPGA setup time in our analysis as this is amortized over long-term computations using the FPGA accelerator. We developed software reference implementations in the Mathworks Matlab environment and used the Matlab coder toolkit to generate single-threaded ANSI C representation of our implementation [12].

6.1 Evaluating CRT Performance

Experimental results on CRT execution times for various ring dimensions can be seen in Figure 15. There is more than one and a half orders of magnitude performance improvement by offloading CRT computations from the CPU to the FPGA, even though we include the round trip call time time for the CRT operation on the FPGA in our experimental analysis to increase operational reality. The FPGA curve has a distinct “J” shape with slightly increased runtimes at smaller ring dimensions on the FPGA. This is a result of the FPGA communication time overhead and is a common artifact of PCIe bus co-processor behavior.

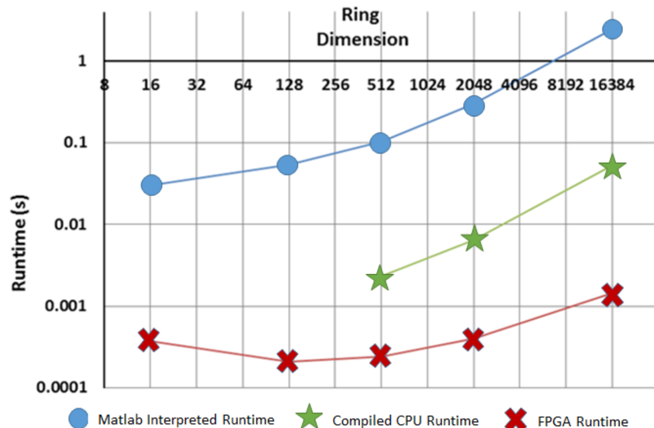

Fig. 15. Runtimes of CRT in various CPU and FPGA Accelerator Configurations

In Table 4, we provider further comparison between the experimentatal observations of CRT runtimes in the FPGA-based HEPU and our reference software implementation run on a CPU, with the runtimes of related hardware-accelerated CRT implementations in [10], [11], [31]. As can be seen in the table, [10], [11] are difficult to compare to our results as experimental results from this prior work is run at a higher ring dimension than our largest experiments on the HEPU. However, although the ring dimensions for results in [10], [11] are double the maximum ring dimension we experimentally support, the runtimes of the HEPU are lower than the prior reported results from all of these three prior works. There is a chance that [10] may report runtimes comparable to results from our HEPU implementation at ring dimension 16384 if run at that level.

6.2 Evaluating CompEvalMult Performance

When designing circuits for programs run over homomorphic encryption schemes, especially LTV implementations,

TABLE 4
Table of Comparative CRT Performance.

Ring Dimension	Compiled CPU Runtime	FPGA Runtime	Runtime Reported in [10]	Runtime Reported in [11]	Runtime Reported in [31]
512	2.32	0.239	NA	NA	NA
1024	3.87	NA	NA	NA	NA
2048	6.48	0.399	NA	NA	NA
16384	52.3	1.457	NA	NA	4
32768	NA	NA	5.743	89	NA

TABLE 5
Table of Comparative CompEvalMult Performance.

Ring Dimension	Compiled CPU Runtime	FPGA Runtime	Runtime Reported in [30]
512	16.03	3.7	NA
1024	29.15	8.7	92
2048	49.17	15.5	NA
16384	463.92	105	NA

the primary practical bottleneck at a circuit program level is the CompEvalMult operation. One of our goals in designing the HEPU was to accelerate the CompEvalMult operation, primarily by outsourcing the CRT and its inverse operations in the ModReduce operation to the HEPU, in addition to other supporting operations as possible.

Our experimental results on the CompEvalMult operation can be seen in Table 5. We show experimental results from our reference software implementation run on a commodity CPU, our HEPU runtimes supported by an FPGA, and experimental results from another LTV software as a basis for comparison. We are unaware of prior publications on hardware accelerators with comparable reported experimental results for the LTV CompEvalMult operation.

Note that the results reported in [30] are slightly slower than the experimental results from our software reference implementation. This gives some confidence that our reference software implementation is relatively effecicient and is a reasonable basis of comparison for the relative performance of hardware-accelerated homomorphic encryption operations in our HEPU.

6.3 Evaluating String Comparison Performance

We evaluate performance improvement with the more operationally relevant encrypted string comparison operation EvalCompare() from [32]. We run the encrypted string matching algorithm with $n = 16384$ and $p = 2$, over a range of signature lengths t. This provides a operationally realistic level of security ($\delta < 1.007$), which is currently believed to be adequately secure and provide at least 80 bits of security [19]. The results of the performance comparison can be seen in Figure 16. Similar with the CRT results, our FPGA-accelerated implementation is more than two orders of magnitude faster than the other implementations.

7 Related Work

Our work is driven by recent advances in Fully Homomorphic Encryption (FHE), inclusive of theoretical and protocol

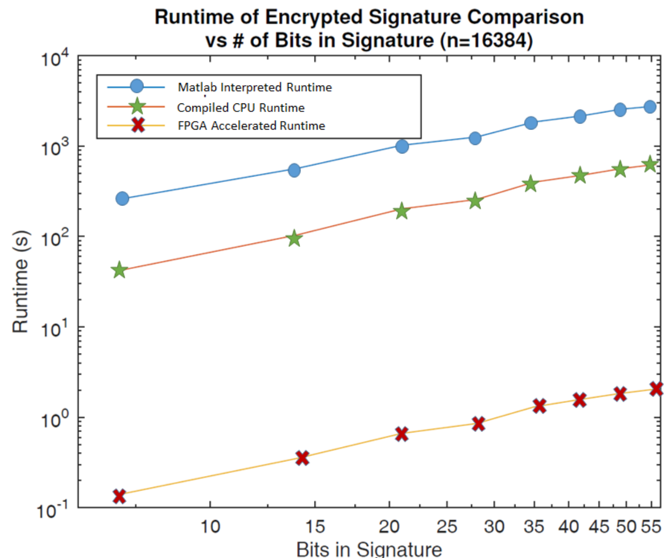

Fig. 16. Encrypted Signature Comparison Runtime vs. Signature Length for Various Computation Devices

advances [4], [6], and implementation advances in software [2] and hardware [8], [9], [10], [11], [31], [33], [34]. Early discussions of our design methodologies are in [35], but earlier papers did not discuss the an HEPU co-processor design, experimental results or provide a detailed discussion of the encryption system design. There has been some prior public work on FPGA implementations for HE which focus on improving efficiency in CRT circuits [9], [10], [11], [31], [33]. This most relevant prior work are specialized to specific classes of operations, rather than as a co-processor.

Despite advances in HE implementations, there has been little published on the application. The design of encrypted string search method we use here as a benchmark is discussed in more depth in [32]. It is also common to use AES circuits as benchmarks for HE implementations instead of the more general CRT operation [36], [37].

8 Discussion and Ongoing Activities

As general lattice encryption and specifically HE technology matures, native high performance implementations in hardware co-processors can be increasingly optimized and will increasingly allow for easier practical adoption of encrypted computing technologies. After designing an FPGA-based lattice encryption accelerator and co-processor, we identify several areas for further advanced refinement. Specifically, our design decision on double-CRT ring element representations with a maximum tower size of 32 is larger than what is needed for viable bootstrapping at high security. Reducing this number will allow use of smaller FPGA chips, at lower cost. The system currently runs 8lane x8 PCIe Gen1, which provides approximately 2 Gb/sec per lane for a total 16 Gb/sec throughput. If we moved to using PCIe Gen2, the throughput would double to 4 Gb/sec per lane. While our current hardware supports Gen2 PCIe, however the PCIe IP block used did not. It should be possible to further optimize by moving to Gen2 PCIe by upgrading the FPGA tools or developing new PCIe interface logic.

As an alternative hardware acceleration approach, dramatic recent improvement in usable GPU technology has brought wider bit width operations needed for secure and practical HE closer to reality. GPUs have a deeper penetration into the cloud computing commodity market than FPGAs currently have, and as such, may prove a valuable implementation platform for acceleration of HE. There have been some successful early efforts to support homomorphic encryption with GPU-based implementation [31], [37], [38]. Future steps include expanding upon this prior work to support GPU-computing arrays, where homomorphic encryption operations are performed on parallel GPU devices. A research challenge in this area is entailed by potential inter-process communication between GPU devices, and the need to coordinate and schedule operations across GPU devices to minimize interprocess communication, reduce possible interprocess communication bottlenecks and maximize overall processing throughput.

ACKNOWLEDGMENTS

The authors wish to acknowledge input received from Chris Peikert of the University of Michigan, Drew Dean from Qualcomm Research and Vinod Vaikuntanathan from MIT.

REFERENCES

[1] C. Gentry, S. Halevi, and N. Smart, "Homomorphic evaluation of the AES circuit," in *Advances in Cryptology, CRYPTO 2012*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds. Springer Berlin / Heidelberg, 2012, vol. 7417, pp. 850–867.

[2] S. Halevi and V. Shoup, "HElib-an implementation of homomorphic encryption," https://github.com/shaih/HElib, 2014.

[3] D. Micciancio, "Lattice-based cryptography," in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 713–715.

[4] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "Multikey fully homomorphic encryption and on-the-fly multiparty computation," *IACR Cryptology ePrint Archive*, vol. 2013, p. 94, 2013, full Version of the STOC 2012 paper with the same title. [Online]. Available: http://eprint.iacr.org/2013/094

[5] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic Number Theory*, ser. Lecture Notes in Computer Science, J. P. Buhler, Ed. Springer Berlin Heidelberg, 1998, vol. 1423, pp. 267–288. [Online]. Available: http://dx.doi.org/10.1007/BFb0054868

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, p. 13, 2014.

[7] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 75–92.

[8] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction." *IACR Cryptology ePrint Archive*, vol. 2013, p. 616, 2013.

[9] E. Öztürk, Y. Doröz, B. Sunar, and E. Savaş, "Accelerating somewhat homomorphic evaluation using FPGAs," Cryptology ePrint Archive, Report 2015/294, Tech. Rep., 2015.

[10] Y. Doröz, E. Öztürk, E. Savaş, and B. Sunar, "Accelerating LTV based homomorphic encryption in reconfigurable hardware," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2015, pp. 185–204.

[11] S. S. Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular hardware architecture for somewhat homomorphic function evaluation," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2015, pp. 164–184.

[12] K. Rohloff and D. B. Cousins, "A scalable implementation of fully homomorphic encryption built on NTRU," in *Proceedings of the 2nd Workshop on Applied Homomorphic Cryptography (WAHC)*, 2014.

[13] V. Lyubashevsky, C. Peikert, and O. Regev, "A toolkit for ring-LWE cryptography." in *EUROCRYPT*, vol. 7881. Springer, 2013, pp. 35–54.

[14] D. D. A. Schönhage and V. Strassen, "Schnelle Multiplikation grosser Zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, 1971.

[15] N. C. Dwarakanath and S. D. Galbraith, "Sampling from discrete Gaussians for lattice-based cryptography on a constrained device," *Applicable Algebra in Engineering, Communication and Computing*, vol. 25, no. 3, pp. 159–180, 2014.

[16] C. Gentry, S. Halevi, C. Peikert, and N. P. Smart, "Ring switching in BGV-style homomorphic encryption," in *Proceedings of the 8th International Conference on Security and Cryptography for Networks*, ser. SCN'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 19–37. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32928-9_2

[17] J. Alperin-Sheriff and C. Peikert, "Practical bootstrapping in quasilinear time," in *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 1–20.

[18] Y. Chen and P. Q. Nguyen, "BKZ 2.0: Better lattice security estimates," in *ASIACRYPT*, ser. Lecture Notes in Computer Science, vol. 7073. Springer, 2011, pp. 1–20.

[19] T. Lepoint and M. Naehrig, *A Comparison of the Homomorphic Encryption Schemes FV and YASHE*. Cham: Springer International Publishing, 2014, pp. 318–335. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06734-6_20

[20] R. Lindner and C. Peikert, "Better key sizes (and attacks) for LWE-based encryption," in *CT-RSA*, 2011, pp. 319–339.

[21] D. Micciancio and O. Regev, "Lattice-based cryptography," in *Post Quantum Cryptography*. Springer, February 2009, pp. 147–191.

[22] M. Albrecht, S. Bai, and L. Ducas, "A subfield lattice attack on overstretched NTRU assumptions: Cryptanalysis of some FHE and graded encoding schemes," Cryptology ePrint Archive, Report 2016/127, 2016.

[23] J. H. Cheon, J. Jeong, and C. Lee, "An algorithm for NTRU problems and cryptanalysis of the GGH multilinear map without an encoding of zero," Cryptology ePrint Archive, Report 2016/139, 2016.

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

[25] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor." in *Crypto*, vol. 86. Springer, 1986, pp. 311–323.

[26] L. R. Rabiner and B. Gold, "Theory and application of digital signal processing," *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1975. 777 p.*, vol. 1, 1975.

[27] M. Knežević, F. Vercauteren, and I. Verbauwhede, "Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods," *Computers, IEEE Transactions on*, vol. 59, no. 12, pp. 1715–1721, 2010.

[28] M. Arioua, S. Belkouch, M. Agdad, and M. Hassani, "VHDL implementation of an optimized 8-point FFT/IFFT processor in pipeline architecture for OFDM systems," in *Multimedia Computing and Systems (ICMCS), 2011 International Conference on*. IEEE, 2011, pp. 1–5.

[29] "7 series FPGA configurable logic block," Xilinx, Tech. Rep., 2014, uG474 (v1.7).

[30] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Manual for using homomorphic encryption for bioinformatics," Technical report MSR-TR-2015-87, Microsoft Research, Tech. Rep., 2015.

[31] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[32] K. Rohloff, "Privacy-preserving data exfiltration monitoring using homomorphic encryption," in *The 2nd IEEE International Conference on Cyber Security and Cloud Computing*, 2015.

[33] S. S. Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular hardware architecture for somewhat homomorphic function evaluation," Cryptology ePrint Archive, Report 2015/337, 2015, http://eprint.iacr.org/.

[34] C. Gentry and S. Halevi, "Implementing Gentry's fully homomorphic encryption scheme," in *Advances in Cryptology, EUROCRYPT 2011*, ser. Lecture Notes in Computer Science, K. Paterson, Ed. Springer Berlin / Heidelberg, 2011, vol. 6632, pp. 129–148.

[35] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An FPGA co-processor implementation of homomorphic encryption,"

in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.
[36] Y. Doröz, Y. Hu, and B. Sunar, "Homomorphic AES evaluation using NTRU," *IACR Cryptology ePrint Archive, Report 2014/039*, vol. 2014, p. 39, 2014.
[37] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption on GPUs," in *Proceedings of the IEEE High Performance Extreme Computing Conference*, 2012.
[38] W. Dai, Y. Doröz, and B. Sunar, "Accelerating NTRU based homomorphic encryption using GPUs," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.
[39] C. Gentry, "Computing arbitrary functions of encrypted data," *Commun. ACM*, vol. 53, no. 3, pp. 97–105, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1666420.1666444
[40] D. Micciancio, "A first glimpse of cryptography's holy grail," *Commun. ACM*, vol. 53, no. 3, pp. 96–96, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1666420.1666445
[41] C. Moore, M. O'Neill, E. O'Sullivan, Y. Doröz, and B. Sunar, "Practical homomorphic encryption: A survey," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 2792–2795.
[42] C. Peikert, "A decade of lattice cryptography," *Foundations and Trends® in Theoretical Computer Science*, vol. 10, no. 4, pp. 283–424, 2016.
[43] T. ElGamal, *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 10–18.
[44] P. Paillier, *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238.

APPENDIX
HOMOMORPHIC ENCRYPTION INTRODUCTION

In this appendix, we provide a brief introduction to homomorphic encryption. More extensive introductory material on homomorphic encryption can be found in [39], [40]. A survey of recent relevant implementation efforts can also be found here [41].

Homomorphic encryption schemes, like all encryption schemes, make it computationally difficult to recover any partial information about a plaintext from its encrypted ciphertext (under precisely formulated hardness assumptions) [42]. This is no different from any modern encryption scheme such as AES and RSA. Unlike other modern encryptions schemes, homomorphic encryption enables computing on encrypted data without decrypting it.

Homomorphic encryption can be used to:

- Encrypt plaintext data into ciphertext.
- Perform special algebraic operations on the ciphertext to obtain new ciphertexts.
- Decrypt the resulting output ciphertext into new plaintext.
- The decrypted output plaintext is equivalent to running special algebraic operations corresponding to the algebraic operations performed on the ciphertext.

Homomorphic encryption can thus be used to perform computations on encrypted data, without exposing the data itself. Examples of homomorphic encryption schemes include the ElGamal [43] and Paillier [44] encryption schemes. ElGamal is multiplicatively homomorphic, in that the decrypted product of two ciphertexts is equal to the product of the corresponding plaintext. Paillier is additively homomorphic, in that the decrypted product of two ciphertexts is equal to the sum of the corresponding plaintext.

The lattice-based encryption schemes which we can support with our HEPU can support arbitrary computations, beyond just addition or multiplication operations. Examples of these lattice-based homomorphic schemes include [4], [6], [7]. These more general homomorphic encryption scheme provide a circuit-style computation model with EvalAdd and EvalMult as the core operations performed on ciphertext, with the corresponding plaintext operations being polynomial addition and multiplication, respectively.