A Scalable Implementation of Fully Homomorphic Encryption Built on NTRU*

Kurt Rohloff David Bruce Cousins
Raytheon BBN Technologies
10 Moulton St.
Cambridge, MA, USA
{krohloff,dcousins}@bbn.com

February 16, 2014

Abstract

In this paper we report on our work to design, implement and evaluate a Fully Homomorphic Encryption (FHE) scheme. Our FHE scheme is an NTRU-like cryptosystem, with additional support for efficient key switching and modulus reduction operations to reduce the frequency of bootstrapping operations. Ciphertexts in our scheme are represented as matrices of 64-bit integers. The basis of our design is a layered software services stack to provide high-level FHE operations supported by lower-level lattice-based primitive implementations running on a computing substrate. We implement and evaluate our FHE scheme to run on a commodity CPU-based computing environment. We implemented our FHE scheme to run in a compiled C environment and use parallelism to take advantage of multi-core processors. We provide experimental results which show that our FHE implementation provides at least an order of magnitude improvement in runtime as compared to recent publicly known evaluation results of other FHE software implementations.

1 Introduction

Recent breakthroughs in Homomorphic Encryption have shown that it is theoretically possible to securely run arbitrary computations over encrypted data without decrypting the data [10, 11]. There has been recent work on designing and implementing variations of Somewhat Homomorphic Encryption (SHE) and Fully Homomorphic Encryption (FHE) schemes [2, 6, 9, 12, 13, 15, 18, 23, 24, 28]. These implementations have become increasingly practical with published results on both the runtime of isolated EvalAdd and EvalMult operations for some implementation [12, 23, 24] and evaluations of composite functions like AES [9, 15, 28].

Current approaches to design FHE schemes rely on bootstrapping to arbitrarily increase the size of computation supported by an underlying SHE scheme. Many current implementations of SHE and FHE schemes rely on the the manipulation of very large integers so that the schemes are both secure and capable of supporting the evaluation of sufficiently large circuits. Prior SHE and FHE implementation designs [12, 15, 23, 24], for the most part, rely on single-threaded execution on commodity CPU-type hardware, partially due to the difficulty of or lack of native support for multi-threaded execution with underlying software libraries [20, 25]. This, in addition to the inherent computational cost of secure computing using known SHE and FHE schemes, prevented the practical use of SHE and FHE.

*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.)

In this paper we report on our work to design, implement and evaluate a scalable Fully Homomorphic Encryption (FHE) scheme which addresses the limitations for secure arbitrary computation. Our implementation uses a variation of a not previously implemented bootstrapping scheme [1] simplified for power-of-2 rings. We also use a "double-CRT" representation of ciphertexts which was also discussed in [15]. With this double-CRT representation, we can select parameters so that ciphertexts are secure when represented as matrices of 64-bit integers, but still support the secure execution of programs on commodity computing device without expending unnecessary computational overhead manipulating large multi-hundred-bit or even multi-thousand-bit integers.

We implement in software specialized lattice primitives such as Ring Addition, Ring Multiplication and the Chinese Remainder Transform (CRT). We use our primitive implementations to construct the FHE operations of Key Generation (KeyGen), Encryption (Enc), Decryption (Dec), Evaluation Addition (EvalAdd), Evaluation Multiplication (EvalMult) and Bootstrapping (Boot). We use supporting Modulus Reduction (ModReduce), Ring Reduction (RingReduce) and Key Switching (KeySwitch) operations to augment the EvalMult operation and support larger depth computations without bootstrapping or decreasing the security of our scheme.

We implemented this scheme to run in a compiled C environment and use parallelism to take advantage of multi-core processors. Taken together, our implementation of these concepts points the way to a practical implementation of FHE with a more efficient (and less frequent) use of the bootstrapping operation. We evaluate the performance of our software library as a set of compiled executables in a commodity CPU-based multi-core Linux environment. The evaluated performance of our library compares favorably with evaluations of the reported experimental CPU-based evaluation results of other recent SHE and FHE schemes implemented in software such as in [12, 23, 24].

This paper is organized as follows. In Section 2 we discuss how we represent ciphertexts in our implementation. In Section 3 we define our NTRU-based FHE scheme. In Section 4 we discuss parameter selection for our NTRU-based scheme to provide practical secure computing on commodity computing hardware. In Section 5 we discuss our experimental results from our FHE scheme implemented in Matlab. We conclude the paper with a discussion of our insights and next steps in Section 6. Data tables experimental runtime results can be seen in Appendix A.

2 Double-CRT Ciphertext Representation

Previous SHE/FHE designs and implementations use two primary parameters to tune the security provided and the supported depth of homomorphic computation (without resorting to bootstrapping): the ring dimension n and the ciphertext modulus q. With these parameters, fresh ciphertexts are typically represented as n-element integer arrays, where each array element consists of at least $\log_2(q)$ bits. In previous implementations the ring dimension n typically ranged from 512 (2^9) to 16384 (2^{14}) and beyond, while several hundred to several thousand bits was typically required to represent q. In the previous implementations that use this "large-q" approach, the practicality challenge derives from the difficulty of supporting both a large ring dimension n (which provides comparatively better security) and a large q (which increases the depth of computation supported).

The requirement of a very large q is potentially problematic, because the number of clock cycles to support mod-q operations using naive "big integer" arithmetic grows at least linearly (and often quadratically) with the number of bits used to represent q for even the simplest operations, e.g., modular addition and multiplication. We use a variation of the double-CRT approach discussed in [15] to circumvent this problem using the standard technique of a "residue number system" (based on the Chinese remainder theorem over the integers) to represent ciphertexts as t length-n integer vectors of mod-q_i values instead of a single integer vector mod q where $q = q_1 * \cdots * q_t$ for pairwise coprime moduli q_i. For our ciphertext representation we use t length-n integer vectors of mod-q_i values represented as a $n \times t$ integer matrix. With our double-CRT approach, the number of moduli (t) grows to support the secure execution of larger programs, but more bits are not required to represent the moduli $q_1, \cdots, q_t$. Our implementation supports the secure execution of depth $t-1$ programs with t moduli.

The double-CRT representation is an extension of the Chinese Remainder Transform (CRT) [19]

representation used in prior SHE and FHE implementations. Chinese remainder transforms are used to convert ciphertexts from the natural "power basis" representation to the double-CRT representation. This conversion can mathematically be represented as a multiplication by square $n \times n$ matrices, but admits a fast, highly parallel evaluation procedure that is closely related to the Cooley-Tukey Fast Fourier Transform (and others.)

As we discuss more in Section 4 below, each of the moduli $q_1, \cdots, q_t$ can be represented as 64-bit integers and still support the secure execution of non-trivial programs. These 64-bit representations greatly improve the practicality of our approach to SHE and FHE. By using 64-bit modular operations to manipulate ciphertexts, keys, etc., we support faster low-level execution of the SHE operations on commodity 64-bit (or even 32-bit) processors.

An advantage of our double-CRT NTRU approach is that the FHE operations can be highly parallelized. Similar to the standard CRT representation, by using a double-CRT representation, the EvalAdd, EvalMult operations and key sub-operations in Bootstrapping, Modulus Reduction, Ring Switching and Key Switching can become t naively parallelized operations. This greatly simplifies the secure execution of programs using our FHE implementation as compared to other, non-CRT representations of ciphertexts.

3 Cryptosystem

In this section we describe the somewhat homomorphic cryptosystem we use that is very similar to the NTRU system [16], though it was not until recently that its homomorphic properties were noticed independently by López-Alt et al. [18] and Gentry et al. [14].

For ease of implementation and design simplicity, we limit our description to power-of-2 cyclotomic rings. For ring dimension n which is a power of 2, define the ring $R = \mathbb{Z}[x]/(x^n + 1)$ (i.e., integer polynomials modulo $x^n + 1$). For a positive integer q, define the quotient ring $R_q = R/qR$ (i.e., integer polynomials modulo $x^n + 1$, with coefficients from $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$).

3.1 Basic NTRU-Type System

In this subsection we provide a mathematical description of a somewhat homomorphic NTRU-based scheme. The message space is R_p for some integer $p \geq 2$, and most arithmetic operations are performed modulo some $q \gg p$ that is relatively prime with p. Fast addition and multiplication in R_q can be performed by using the mod-q Chinese Remainder Transform (CRT) representation of elements. The basic operations of the scheme are as follows:

- Gen: choose a short $f \in R$ such that $f = 1 \bmod p$ and f is invertible modulo q, and a short $g \in R$. Output $pk = h = g \cdot f^{-1} \bmod q$ and $sk = f$.

  Note that f is invertible modulo q if and only if each of its mod-q CRT coefficients is nonzero. The CRT coefficients of f^{-1} (modulo q) are just the mod-q inverses of those of f.

  Concretely, the short elements f and g can be chosen from discrete Gaussians. E.g., we can let $f = p \cdot f' + 1$ for some Gaussian-distributed f'. Note that such an f will have expectation (center) 1. Using a zero-centered f can have some advantages, and may be chosen using a more sophisticated sampling algorithm.

- $\mathsf{Enc}(pk = h, \mu \in R_p)$: choose a short $r \in R$ and a short $m \in R$ such that $m = \mu \bmod p$. Output $c = p \cdot r \cdot h + m \bmod q$.

  Concretely, m can naively be chosen as $m = p \cdot m' + \mu$ for a Gaussian-distributed m', but again, such an m is not zero-centered. It is typically better to choose m as a zero-centered random variable congruent to μ modulo p.

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

The homomorphic operations are defined as follows:

- $\mathsf{EvalAdd}(c_0, c_1)$: output $c = c_0 + c_1 \bmod q$.
- $\mathsf{EvalMult}(c_0, c_1)$: output $c = c_0 \cdot c_1 \bmod q$.

With the use of $\mathsf{EvalMult}$, the decryption procedure needs to be modified. Define the "degree" of ciphertexts as follows: a freshly generated ciphertext has degree 1, and the degree of $c = \mathsf{EvalMult}(c_0, c_1)$ is the sum of the degrees of c_0 and c_1. Then decryption of a ciphertext c of degree at most d is the same as above, except that we instead compute $\bar{b} = f^d \cdot c \bmod q$.

3.2 Key Switching

Key switching converts a ciphertext of degree at most d, encrypted under a secret key f_1, into a degree-1 ciphertext c_2 encrypted under a secret key f_2 (which may or may not be the same as f_1). This requires publishing a "hint"

$$a_{1\to 2} = m \cdot f_1^d \cdot f_2^{-1} \bmod q,$$

for a short $m \in R$ congruent to 1 modulo p. (Concretely, we can choose $m = p \cdot e + 1$ for a Gaussian-distributed e, though a zero-centered m is better.)

- $\mathsf{KeySwitch}(c_1, a_{1\to 2})$: output $c_2 = a_{1\to 2} \cdot c_1 \bmod q$.

Note that $a_{1\to 2}$, c_1, c_2 can all be stored and operated upon in CRT form, so key switching is very efficient: the hint is just one ring element, and the procedure involves just one coordinate-wise multiplication of the CRT vectors. This compares quite favorably to key-switching procedures for other cryptosystems, which typically require decomposing a ciphertext into several short ring elements and performing several ring multiplications.

3.3 Ring Reduction

Ring reduction maps a ciphertext from ring n to smaller ring $n' = n/2^a$, where typically $a = 1$. Although we describe a ring reduction operation for power-of-2 rings, more general ring switching approaches exist and can be obtained from simple generalizations of the approach we describe here.

The basic ring switching operation is a Decompose algorithm, which maps a dimension n ring to dimension n' elements. Decompose(c) works as follows:

- Let $c = (c_0, ..., c_{n-1})$ be in the power basis and let $w = n/n'$.
- We output ciphertexts c'_i for each $i = 0, ..., w-1$ where $c'_i = (c_i, c_{w+i}, c_{2w+i}, ..., c_{(m'-1)w+i})$. I.e., c'_i just consists of those entries of c whose indices are $i \bmod w$.

Before applying Decompose we first key-switch the ciphertext to one which can be decrypted by a "sparse" secret key sk, whose only nonzero entries in the power basis are at indices equal to $0 \bmod w$. We perform the ring-switching on a ciphertext c, by performing key-switching on c to get cp (encrypted under sk), then call Decompose(cp) to get the $/c'_i/$. The ciphertext c should only have plaintext data only in its indices $0 \bmod w$. Otherwise, this data is lost during the ring reduction operation.

3.4 Modulus Reduction

Modulus reduction, initially proposed in [3], converts a ciphertext from modulus q to a smaller modulus (q/q'), where q' divides q (and so is also relatively prime with p), while also reducing the underlying noise by about a q' factor.

The basic description is as follows: given a ciphertext $c \in R_q$, we add to it a small integer multiple of p that is congruent to $-c \bmod q'$. This ensures that the underlying noise remains small, the plaintext remains unchanged, and the resulting ciphertext is divisible by q'. Then we can divide both the ciphertext and modulus by q', which reduces the underlying noise term by a q' factor as well.

Note that the final step (of dividing by q') implicitly multiplies the underlying message by $(q')^{-1} \bmod p$. We can either keep track of these extra factors as part of the ciphertext and correct for them as the final step of decryption, or we can just ensure that $q' = 1 \bmod p$, so that division by q' does not affect the underlying message.

The following formal procedure uses the fixed (ciphertext-independent) value $v = (q')^{-1} \bmod p$, which can be computed in advance and stored.

- ModReduce(c, q, q'):
  1. compute a short $d \in R$ such that $d = c \bmod q'$.
  2. compute a short $\Delta \in R$ such that $\Delta = (vq' - 1) \cdot d \bmod (pq')$. E.g., all of Δ's integer coefficients can be in the range $[-pq'/2, pq'/2)$.
  3. let $d' = c + \Delta \bmod q$. By construction, d' is divisible by q'.
  4. output $(d'/q') \in R_{(q/q')}$.

Following [15], the above is most efficient to implement when $q = q_1 \cdots q_t$ is the product of several small, pairwise relatively prime moduli; when q' is one of those moduli (say, $q' = q_t$ without loss of generality); and when c is represented in "double-CRT" form, i.e., each of c's mod-q CRT coefficients is itself represented in (integer) CRT form as a vector of mod-q_i values, one for each i. Then the above steps can be computed as follows:

1. Computing d is done by inverting the mod-q_t CRT on the vector of mod-q_t components of c (leaving the other mod-q_i components unused), and interpreting the resulting coefficients as integers in $[-q_t/2, q_t/2)$.
2. Computing Δ is done by multiplying the coefficients of d by the fixed scalar $(vq_t - 1)$ modulo pq_t.
3. Adding Δ to c is done by computing the double-CRT representation of Δ (i.e., applying each mod-q_i CRT to Δ), and adding it entry-wise to c's double-CRT representation.

   Note that the mod-q_t CRTs of Δ and c are just the negations of each other (by construction), so their sum is the all-zeros vector. Therefore, there is no need to explicitly compute the mod-q_t CRT of Δ.
4. Computing d'/q_t is done by dropping the mod-q_t components in the double-CRT representation of d' (which are all zero anyway), and multiplying every mod-q_i component by the fixed scalar $q_t^{-1} \bmod q_i$. (These scalars can be computed in advance and stored.)

3.5 Composed EvalMult

We use the Key Switching, Ring Reduction and Modulus Reduction operations as supporting functions with EvalMult to improve noise management and enable more computation between calls to the Bootstrapping operation. Taken together, we form a composite operation, which we call ComposedEvalMult, from the sequential execution of an EvalMult, Key Switching and Modulus Reduction operation.

Ring Reduction is called during some ComposedEvalMult operations, depending on the level of security provided by a ciphertext resulting from the result of the Ring Reduction operation. As Modulus Reduction operations are performed the security provided by a ciphertexts increases (as described in 4.) Ring Reduction correspondingly reduces the level of security provided by a ciphertext. We implemented our FHE library such that a minimum level of security δ' is provided at all times, and this level of δ' is a parameter selectable by the library user. If a call to a Ring Reduction operation will result in a level of security $\delta \leq \delta'$, then the RingReduction is performed in the ComposedEvalMult operation.

Our conception is that due to the ModReduction and RingReduction component of ComposedEvalMult, it is feasible to coordinate the choice of the original ciphertext width t and the scheduling of ComposedEvalMult operations so that the final ciphertext resulting from secure circuit evaluation and which needs to be decrypted is only one column wide with respect to a single modulus q_1 and provides

a level of security at least as great as the original ciphertexts resulting from the encryption operation. More explicitly, if we need to support a depth $t-1$ computation, the initial encryptions should only be t columns wide to ensure that the final ciphertext is 1 column wide. Whereas the runtime of Encryption, EvalAdd, ComposedEvalMul depend on the ring dimension and depth of computation supported, the Decryption operation would hence depend only on the final ring dimension after all ring switching has been completed. If we need to decrypt a ciphertext that has multiple columns we our double-CRT representation, we could perform multiple ModReduction operations to reduce this $t > 1$ ciphertext until we are left with a single mod-q_1 column.

3.6 Bootstrapping

The basis of our bootstrapping approach comes from a new approach to homomorphic rounding. This approach to bootstrapping is described in detail in [1]. We provide a high-level overview of this operation here, simplified for our restriction to power-of-2 rings. This operation has the following steps:

1. *Round the ciphertext*: For each entry v for residue i, we output $round(v * q/q_i)$, where the inner expression is rational, and "round" means taking the nearest integer. Generally $q = 2^\ell$ is chosen experimentally, but as small as possible.
2. *Convert the plaintext modulus*: This is no-op under our simplifying assumptions.
3. *Lift the ciphertext and plaintext moduli*: This is also a no-op under our simplifying assumptions.
4. *Scale the ciphertext*: We scale up the ciphertext by a Q/q' factor (rounding to nearest integers in the power basis), and embed into dimension N (new ring dimension) as well. The plaintext modulus is still q'.
5. *Compute the homomorphic trace*: The following steps are performed iteratively $\log_2(N)$ times:
   (a) "Lift" the ciphertext modulus to $2Q$, which has the effect of making the plaintext modulus 2q.
   (b) Apply the automorphism from [1], with appropriate key switching to put the result into the same key as the original ciphertext in the iteration.
   (c) Sum the original and resulting ciphertexts.
   (d) Divide the ciphertexts by 2.
6. *Perform a homomorphic rounding*: This operation is described in Appendix B of [1].

4 Parameter Selection

The selection of n and $q_1, \ldots, q_t$ depends heavily on the plaintext modulus p, the depth of computation that needs to be supported, and the desired security level. We capture the primary concerns influencing the selection of a ring dimension n and the moduli $q_1, \ldots, q_t$ at a high level as follows:

- The necessary ring arithmetic should be easily supported on the computation substrate – i.e., that mod-q_i operations (for $i \in \{1, \ldots, t\}$) require few clock cycles.
- The moduli $q_1, \ldots, q_t$ are sufficiently large to enable sufficient noise shrinkage via modulus reduction.
- The ring dimension n and noise parameters are sufficiently large so the scheme provides adequate security.
- The ring dimension n is not so large that it becomes overly time-consuming and memory-intensive to manipulate the ciphertexts.
- The plaintext modulus p and any noise added to the ciphertext during encryption is sufficiently small that we can evaluate reasonably sized circuits with correct decryption.

Table 1: Dependence of bit lengths of moduli q_i, as a function of 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

We choose to add discrete Gaussian noise to the fresh ciphertexts where $r = 6$ represents the selected probability distribution parameter. We have found theoretically that the smallest modulus q_1 needs to satisfy the expression

$$q_1 > 4pr\sqrt{n}w \tag{1}$$

in order to ensure successful decryption, where the parameter $w \approx 6$ represents an "assurance" measure for correct decryption (essentially, the probability of decryption failure is bounded by the probability that a normally distributed variable is more than $w\sqrt{2\pi}$ standard deviations from its mean), and $p \cdot r$ is the Gaussian parameter of the noise used in fresh ciphertexts. (Hence r is the Gaussian parameter of the underlying NTRU-like problem.)

After selecting q_1, we select the remaining $q_i \in \{q_2, \ldots, q_t\}$ such that

$$q_i > 4p^2r^5n^{1.5}w^5, \tag{2}$$

which ensures that modulus reduction by a factor of q_i sufficiently reduces the noise after a ComposedEvalMult operation. For implementation simplicity, we set q_1 to be the smallest feasible solution to $q_1 > 4p^2r^5n^{1.5}w^5$. Consequently all q_i are represented by $\log_2(q_t)$ bits, leading to simpler implementations.

Table 1 shows how many bits are required to represent $q_1, \ldots, q_t$ for varying ring dimensions for $p = 2$. Note that all $q_1, \ldots, q_t$ can be represented in less than 64 bits.

Following [5, 17, 22, 26], we use the standard "root Hermite factor" δ as the primary measure of concrete security for a set of parameters. The most recent experimental evidence [5] suggests that $\delta = 1.007$ would require roughly 2^{40} core-years on recent Intel Xeon processors to break. Using the estimates from [17, 22], we found that in order to achieve a security level δ for a depth of computation $d = t - 1$ using the t moduli $q_1, \ldots, q_t$, we need to ensure that

$$n \geq \lg(q_1 \cdots q_t)/(4\lg(\delta)). \tag{3}$$

Table 2 shows how δ varies as a function of the ring dimension and depth of computation supported. Based on our analysis, if we impose the requirement that $\delta \leq 1.007$, then we would need to use ring dimension $n = 16324$ to support depth $d = 13$ computations.

Table 2: Security level δ, as a function of depth of computation supported and ring dimension for $p = 2$.

Dim. \ Depth	1	3	5	7	9	11	13	15	17	19
512	1.015	1.045	1.077	1.109	1.143	1.178	1.213	1.250	1.288	1.327
1024	1.007	1.023	1.038	1.054	1.070	1.087	1.104	1.121	1.138	1.155
2048	1.004	1.012	1.020	1.028	1.036	1.044	1.053	1.061	1.069	1.078
4096	1.002	1.006	1.010	1.014	1.018	1.022	1.026	1.030	1.035	1.039
8192	1.0011	1.003	1.005	1.007	1.009	1.011	1.013	1.016	1.018	1.020
16384	1.0005	1.0016	1.003	1.003	1.005	1.006	1.007	1.008	1.009	1.010

5 Evaluation Experiments

We implemented our scheme in the Mathworks Matlab environment and used the Matlab coder toolkit [21] to generate an ANSI C representation of our implementation. We subsequently hand-modified our

auto-generated ANSI C to incorporate the pthreads library [4] to leverage parallelism. We compiled this ANSI C using gcc to run as an executable in a Linux environment. We believe that additional performance improvements could be obtained by implementing our FHE scheme natively in C.

We chose to implement our scheme in Matlab because it provides an interpreted computation environment for rapid prototyping with native support for vector and matrix manipulation which simplifies implementation development. We found the Matlab syntax to be a natural fit for writing software to support the primitive lattice operations needed for our double-CRT NTRU-based SHE design.

We wrote our Matlab implementation of our double-CRT NTRU SHE scheme using the Matlab fixed-point toolbox. The Matlab fixed-point toolbox also provides a path toward generated HDL implementations of our design that can be deployed for practical use on highly parallel computing hardware such as FPGAs. Part of our vision for the use of our SHE design is to develop an FPGA implementation of FHE [7, 8].

We ran our compiled implementation on a 64core 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 and 20 cores during the evaluation of our software implementation.

We collected data on the runtime of the Encryption, EvalAdd, ComposedEvalMult, Decryption and Bootstrapping operations over selections of depth of computation supported and ring dimension. We ran 100 iterations of this collection procedure for each combination of t and ring dimension. We used different randomly selected key sets, plaintexts and encryption noise on every iteration to mitigate minor variations in performance that may arise due to these experimental random variables on every iteration. Tables of the raw mean runtime results can be seen in Tables 3 through 7 in Appendix A.

We collected data on the runtime of the Encryption, EvalAdd and ComposedEvalMult operations for settings of $t \in \{2, 4, 6, ..., 20\}$ and for ring dimensions $n \in \{512, 1024, 2048, 4096, 8192, 16384\}$. We collected data on the runtime of the Decryption operation of final ciphertexts, for computations with fresh (input) ciphertexts with ring dimensions $n \in \{512, 1024, 2048, 4096, 8192, 16384\}$ and depth of computation $t - 1$ for $t \in \{2, 4, 6, ..., 20\}$. Note that due to ring switching, decryption runtime is dependent only on the dimension of the final ciphertext, which is a function of the initial ciphertext and depth of computation. We collected data on the runtime of the Bootstrapping operation for settings of the "maximum" ring dimensions $n \in \{512, 1024, 2048, 4096, 8192, 16384\}$ ciphertexts are expressed in where the resulting ciphertext supports a depth one computation before another bootstrapping operations is required. As discussed in [1], the depth of computation required for bootstrapping is logarithmic in the ring dimension. We are currently exploring practical trade-offs associated with the impacts on the scheduling of bootstrapping to enable more computation between bootstrapping calls.

Our experimental results shows that run times grow linearly with ring dimension n and the ciphertext width t where $t-1$ is the depth of computation supported before bootstrapping or decryption could still be performed and have a high probability of recovering a correctly decrypted ciphertext. This makes intuitive sense because as we double either the ring dimension or the ciphertext width, we roughly double the amount of computation that needs to be performed with every Encryption, EvalAdd and ComposedEvalMult operation. Similar results hold for Decryption (Table 6) which shows a linear dependence of runtime on ring dimension, but under the assumption that decryption occurs after $t - 1$ ModReduction operations, including ModReduction operations bundled in ComposedEvalMult operations. Our initial results show that Bootstrapping runtime is similarly linear with respect to the maximum ring dimension. As compared to the results reported in [12, 23, 24], our FHE software implementation provides order-of-magnitude improvements in the runtime of the FHE operations.

6 Discussion and Looking Forward

Our FHE implementation is part of our long-term vision to support a general, practical and secure computing capability through a layered services architecture. Part of our vision is to provide software interfaces in our design for our highly optimized implementations of the basic FHE operations (KeyGen, Encrypt, EvalAdd, EvalMult, Decrypt) for users to construct general applications that require secure computation on encrypted data with automated calls to supporting operations such as Ring Switching,

Key Switching, Modulus Reduction and Bootstrapping. Inherent to this architecture vision is our FHE implementation of lattice-based computational primitives which form a lower layer of our envisioned architecture. We use these primitives such as ring addition, ring multiplication, modulus operations and the Chinese Remainder Transforms to run on commodity computing devices such as CPUs and FPGAs. We designed this modular approach to the implementation of the SHE operations and the underlying core primitives which allows us to 1) augment these operations with additional operations such as a bootstrapping operation (which enables FHE), or 2) replace the implementations of a subset of the operations or primitives as implementation advances are made.

A further aspect of our layered architecture vision is our ability to mix-and-match a computing substrate at lower levels of our architecture. Although not an immediate focus of the results reported here, the double-CRT representation, coupled with the 64-bit integer representation, simplifies parallelization of our FHE scheme for easier porting to other, high-performance and low-cost parallel computing environments such as FPGAs [7, 8] and possibly even GPUs [27]. If ported to a dedicated FPGA co-processor, the runtime of our underlying SHE/FHE implementation can be greatly improved upon as compared to the runtime of the corresponding interpreted CPU-only implementation which we discuss herein.

Taken together, we see our design and experimentation with our NTRU-based FHE scheme as a stepping-stone to a practical implementation of FHE through our layered architecture vision. Our primary path forward is to increasingly leverage the inherent parallelism of our design at multiple levels of our implementation. At a low level we are working to port our lattice-based primitives to operate on commodity FPGAs. This higher level parallelism offers the possibility of more practical SHE and FHE on both multi-core CPUs or multiple parallel FPGAs operating as "FHE co-processors".

Acknowledgement

The authors wish to acknowledge the helpful feedback and guidance of Prof. Chris Peikert in preparing the material discussed in this paper.

References

[1] Jacob Alperin-Sheriff and Chris Peikert. Practical bootstrapping in quasilinear time. In Ran Canetti and JuanA. Garay, editors, *Advances in Cryptology CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2013.

[2] JoppeW. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *Cryptography and Coding*, volume 8308 of *Lecture Notes in Computer Science*, pages 45–64. Springer Berlin Heidelberg, 2013.

[3] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS*, pages 97–106, 2011.

[4] David Butenhof. *Programming with POSIX (R) threads*. Addison-Wesley Professional, 1997.

[5] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.

[6] JungHee Cheon, Jean-Sbastien Coron, Jinsu Kim, MoonSung Lee, Tancrde Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch fully homomorphic encryption over the integers. In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 315–335. Springer Berlin Heidelberg, 2013.

[7] David Bruce Cousins, Kurt Rohloff, Chris Peikert, and Rick Schantz. SIPHER: Scalable implementation of primitives for homomorphic encryption - FPGA implementation using Simulink. In *Fifteenth Annual Workshop on High Performance Embedded Computing (HPEC)*, HPEC '11, 2011.

[8] David Bruce Cousins, Kurt Rohloff, Chris Peikert, and Rick Schantz. An update on scalable implementation of primitives for homomorphic encryption - FPGA implementation using simulink. In *Sixteenth Annual Workshop on High Performance Embedded Computing (HPEC)*, HPEC '12, 2012.

[9] Yarkin Doroz, Yin Hu, and Berk Sunar. Homomorphic aes evaluation using ntru. Cryptology ePrint Archive, Report 2014/039, 2014. `http://eprint.iacr.org/`.

[10] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, Stanford, CA, USA, 2009. AAI3382729.

[11] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[12] Craig Gentry and Shai Halevi. Implementing Gentry's fully homomorphic encryption scheme. In Kenneth Paterson, editor, *Advances in Cryptology EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer Berlin / Heidelberg, 2011.

[13] Craig Gentry and Shai Halevi. HElib. `https://github.com/shaih/HElib`, 2014.

[14] Craig Gentry, Shai Halevi, Vadim Lyubashevsky, Chris Peikert, Joseph Silverman, and Nigel Smart, 2011. Personal communication.

[15] Craig Gentry, Shai Halevi, and Nigel Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer Berlin / Heidelberg, 2012.

[16] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer Berlin Heidelberg, 1998.

[17] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In *CT-RSA*, pages 319–339, 2011.

[18] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*, pages 1219–1234, 2012.

[19] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13190-5.

[20] MAGMA. V2.18-11, 2012. Available at `http://magma.maths.usyd.edu.au/magma/`.

[21] MATLAB. *R2012b*. The MathWorks Inc., Natick, Massachusetts, 2012.

[22] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In *Post Quantum Cryptography*, pages 147–191. Springer, February 2009.

[23] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.

[24] Henning Perl, Michael Brenner, and Matthew Smith. Poster: an implementation of the fully homomorphic Smart-Vercauteren cryptosystem. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 837–840, New York, NY, USA, 2011. ACM.

[25] Victor Shoup. *NTL: A Library for doing Number Theory*. Courant Institute, New York University, New York, NY, 2012. Available at `http://shoup.net/ntl/`.

[26] Joop van de Pol. Quantifying the security of lattice-based cryptosystems in practice. In *Mathematical and Statistical Aspects of Cryptography*, 2012.

[27] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Accelerating fully homomorphic encryption on GPUs. In *Proceedings of the IEEE High Performance Extreme Computing Conference*, 2012.

[28] David Wu and Jacob Haven. Using homomorphic encryption for large scale statistical analysis. 2012.

A Experimental Results

Table 3: Encryption Runtime (ms) vs. Depth of Computation Supported and Ring Dimension for $p = 2$.

Dim. \ Depth	1	3	5	7	9	11	13	15	17	19
512	2.32	2.83	2.86	3.27	3.39	3.25	4.38	4.64	5.35	5.66
1024	3.87	5.33	5.17	5.98	5.68	5.63	6.94	8.40	9.04	9.20
2048	6.26	6.48	7.01	7.47	7.94	8.78	12.70	13.03	13.05	14.52
4096	12.08	12.27	13.04	14.87	17.38	17.65	20.73	17.46	21.57	22.13
8192	24.53	25.18	26.13	29.07	30.81	32.15	34.43	32.46	36.16	37.90
16384	52.30	55.02	58.05	59.71	60.29	61.98	63.44	64.99	69.96	72.89

Table 4: EvalAdd Runtime (ms) vs. Depth of Computation Supported and Ring Dimension for $p = 2$.

Dim. \ Depth	1	3	5	7	9	11	13	15	17	19
512	0.21	0.32	0.42	0.54	0.64	0.73	1.26	2.11	2.90	3.12
1024	0.30	1.04	0.47	0.57	0.72	0.74	1.40	2.72	2.85	2.93
2048	0.37	0.45	0.55	0.67	0.80	1.00	1.97	3.00	3.04	3.24
4096	0.56	0.65	0.74	0.91	1.92	2.07	2.25	2.43	3.73	3.54
8192	0.89	1.01	1.20	1.36	2.46	2.70	3.69	3.23	5.05	5.44
16384	1.58	1.82	2.12	2.39	3.99	4.19	4.27	4.77	7.16	7.29

Table 5: ComposedEvalMult Runtime (ms) vs. Depth of Computation and Ring Dim. for $p = 2$.

Dim. \ Depth	1	3	5	7	9	11	13	15	17	19
512	16.03	22.73	23.32	22.65	22.87	22.96	24.35	25.24	25.37	25.78
1024	29.15	37.85	39.05	39.11	38.79	39.24	39.49	39.59	39.52	39.68
2048	49.17	66.31	66.77	67.41	67.15	68.38	68.22	69.27	69.45	71.09
4096	99.56	140.42	140.71	141.42	141.26	142.75	143.52	145.51	144.61	148.31
8192	196.83	279.37	280.42	284.40	283.98	285.69	289.59	286.55	292.69	295.69
16384	463.92	623.19	622.74	628.87	630.43	633.37	639.52	642.80	651.20	659.88

Table 6: Decryption Runtime (ms) vs. Depth of Computation Supported and Initial Ring Dim. for $p = 2$.

Dim. \ Depth	1	3	5	7	9	11	13	15	17	19
512	0.40	0.26	0.13	0.14	0.10	0.10	0.06	0.06	0.06	0.06
1024	0.87	0.38	0.18	0.11	0.11	0.11	0.11	0.11	0.05	0.05
2048	1.92	0.84	0.38	0.38	0.22	0.22	0.22	0.22	0.12	0.12
4096	3.36	1.70	0.84	0.86	0.37	0.39	0.38	0.22	0.22	0.21
8192	7.22	3.43	1.67	1.72	0.85	0.87	0.86	0.87	0.39	0.40
16384	15.36	7.18	3.37	3.37	1.67	1.67	1.67	1.73	0.87	0.85

Table 7: Bootstrapping Runtime (s) vs. Ring Dimension for $p = 2$.

Ring Dimension	512	1024	2048	4096	8192	16384
Runtime (s)	5.8	13	26	60	125	275