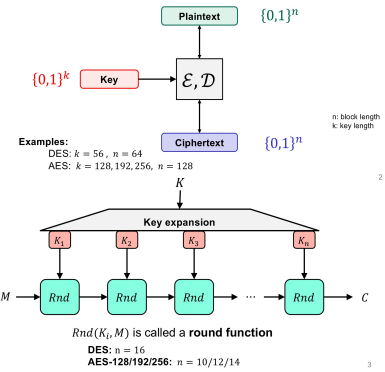
* AES (Advanced Encryption Standard)

- block cipher -> 평문을 블록 단위로 암호화 한다. (AES, DES)
- Block cipher 방식 -> 그냥 블록 단위로 평문그대로 (ECB 모드), 조금더 (CBC, CTR)
-> DES 취약점 (key 짧음) -> AES 탄생

- 패딩 => 평문 받으면 블록크기 배수로 만듦. PCM)이 으로 패딩하면 원문 복구 가능해야함. but 단순 0 붙이기는 원래것 알수없어 힘듦



Homomorphic Evaluation of the AES Circuit

(Updated Implementation)

계산

Craig Gentry
IBM Research

Shai Halevi
IBM Research

Nigel P. Smart
University of Bristol

January 3, 2015

Abstract

We describe a working implementation of leveled homomorphic encryption (with or without bootstrapping) that can evaluate the AES-128 circuit. This implementation is built on top of the HElib library, whose design was inspired by an early version of this work. Our main implementation (without bootstrapping) takes about 4 minutes and 3GB of RAM, running on a small laptop, to evaluate an entire AES-128 encryption operation. Using SIMD techniques, we can process upto 120 blocks in each such evaluation, yielding an amortized rate of just over 2 seconds per block.

상환률

For cases where further processing is needed after the AES computation, we describe a different setting that uses bootstrapping. We describe an implementation that lets us process 180 blocks in just over 18 minutes using 3.7GB of RAM on the same laptop, yielding amortized 6 seconds/block. We note that somewhat better amortized per-block cost can be obtained using "byte-slicing" (and maybe also "bit-slicing") implementations, at the cost of significantly slower wall-clock time for a single evaluation.

In this article we describe many of the optimizations that went into this implementation. These include both AES-specific optimizations, as well as several "generic" tools for FHE evaluation (which are incorporated in the HElib library). The generic tools include (among others) a different variant of the Brakerski-Vaikuntanathan key-switching technique that does not require reducing the norm of the ciphertext vector, and a method of implementing the Brakerski-Gentry-Vaikuntanathan modulus-switching transformation on ciphertexts in CRT representation.

↳ 중국인 나머지 정리

Keywords. AES, Fully Homomorphic Encryption, Implementation

An early version of this work was published in CRYPTO 2012. The current report describes also more recent implementation work, done over the last two years.

For the early version, the first and second authors were partly sponsored by DARPA under agreement number FA8750-11-C-0096. The U.S. Government is authorized to reproduce and distribute reprints of the early version for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

For the same early version, the third author was sponsored by DARPA and AFRL under agreement number FA8750-11-2-0079. The same disclaimers as above apply. He is also supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant COED–EP/I03126X, and by a Royal Society Wolfson Merit Award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the European Commission or EPSRC.

Contents

1 Introduction

In his breakthrough result [13], Gentry demonstrated that fully-homomorphic encryption was theoretically possible, assuming the hardness of some problems in integer lattices. Since then, many different improvements have been made, for example authors have proposed new variants, improved efficiency, suggested other hardness assumptions, etc. Some of these works were accompanied by implementation [28, 14, 8, 29, 21, 9], but these implementations were either "proofs of concept" that can compute only one basic operation at a time (at great cost), or special-purpose implementations limited to evaluating very simple functions. In the early version of this work we reported on the first implementation powerful enough to support an "interesting real world circuit," specifically the AES-128 encryption operation. To this end, we implemented a variant of the leveled FHE-without-bootstrapping scheme of Brakerski, Gentry, and Vaikuntanathan [5] (BGV). In the current article we report on an updated implementation of the same circuit, using the "general purpose" open-source HElib library [18], whose design was inspired by that early version of our work. (As of December 2014, we made our new implementation available as part of HElib.)

Why AES? We chose to shoot for an evaluation of AES since it seems like a natural benchmark: AES is widely deployed and used extensively in security-aware applications (so it is "practically relevant" to implement it), and the AES circuit is nontrivial on one hand, but on the other hand not astronomical. Moreover the AES circuit has a regular (and quite "algebraic") structure , which is amenable to parallelism and optimizations. Indeed, for these same reasons AES is often used as a benchmark for implementations of protocols for secure multi-party computation (MPC), for example [26, 10, 19, 20]. Using the same yardstick to measure FHE and MPC protocols is quite natural, since these techniques target similar application domains and in some cases both techniques can be used to solve the same problem.

Beyond being a natural benchmark, homomorphic evaluation of AES decryption also has interesting applications: When data is encrypted under AES and we want to compute on that data, then homomorphic AES decryption would transform this AES-encrypted data into an FHE-encrypted data, and then we could perform whatever computation we wanted. (Such applications were alluded to in [21, 29, 6]).

Why BGV? Our implementation is based on the (ring-LWE-based) BGV cryptosystem [5], which is one of the few variants that seem the most likely to yield "somewhat practical" homomorphic encryption. Other variants are the NTRU-like cryptosystem of Lòpez-Alt et al. [23], the ring-LWE-based scale-invariant cryptosystem of Brakerski [4]. These three variants offer somewhat different implementation tradeoffs, but they all have similar performance characteristics. We don't expect the differences between these variants to be very significant, and moreover most of our optimizations for BGV are useful also for the other two variants. (Another interesting approach if to implement the newer cryptosystem of Gentry et al. [16], or some combination thereof.)

Contributions of this work. Our implementation is based on a variant of the BGV scheme [5, 7, 6] (based on ring-LWE [24]), using the techniques of Smart and Vercauteren (SV) [29] and Gentry, Halevi and Smart (GHS) [15], and we introduce many new optimizations. Some of our optimizations are specific to AES, these are described in Section 4. Most of our optimization, however, are more general-purpose and can be used for homomorphic evaluation of other circuits, these are described in Section 3.

Many of our general-purpose optimizations are aimed at reducing the number of FFTs and CRTs that we need to perform, by reducing the number of times that we need to convert polynomials between coefficient and evaluation representations. Since the cryptosystem is defined over a polynomial ring, many of

the operations involve various manipulation of integer polynomials, such as modular multiplications and additions and Frobenius maps. Most of these operations can be performed more efficiently in evaluation representation, when a polynomial is represented by the vector of values that it assumes in all the roots of the ring polynomial (for example polynomial multiplication is just point-wise multiplication of the evaluation values). On the other hand some operations in BGV-type cryptosystems (such as key switching and modulus switching) seem to require coefficient representation, where a polynomial is represented by listing all its coefficients.[1] Hence a "naive implementation" of FHE would need to convert the polynomials back and forth between the two representations, and these conversions turn out to be the most time-consuming part of the execution. In our implementation we keep ciphertexts in evaluation representation at all times, converting to coefficient representation only when needed for some operation, and then converting back.

We describe variants of key switching and modulus switching that can be implemented while keeping almost all the polynomials in evaluation representation. Our key-switching variant has another advantage, in that it significantly reduces the size of the key-switching matrices in the public key. This is particularly important since one limiting factor for evaluating "interesting" circuits is the ability to keep the key-switching matrices in memory. Other optimizations that we present are meant to reduce the number of modulus switching and key switching operations that we need to do.

Our Implementation and tests. Many of the optimizations described in this work were incorporated in the HElib C++ library, which is built on top of NTL (and GnuMP). We tested our implementation on a two years old Lenovo X230 laptop with Intel Core i5-3320M running at 2.6GHz, on which we run an Ubuntu 14.04 VM with 4GB of RAM and with the g++ compiler version 4.9.2. The detailed results of our tests are described in Section 4.4, the one-line summary is that we can evaluate AES-128 homomorphically on 120 blocks in 245 seconds on that commodity laptop. Also, if we need to incorporate extra processing then we can use bootstrapping and get evaluation on 180 blocks in under 18 minutes. All of our programs are single-threaded, so only one core was used in the computations.

We note that there are a multitude of optimizations that one can perform on our basic implementation. Most importantly, there are great gains to be had by making better use of parallelism: Unfortunately, the HElib library is not yet thread safe, which severely limits our ability to utilize the multi-core functionality of modern processors. Much of the work in homomorphic-AES is "embarrassingly parallelizable" and so we expect a fully parallel implementation to have a speedup factor roughly equal to the number of active cores (with parallelization opportunities not running our until perhaps 100x of current implementation). The byte-sliced and bit-sliced implementations (which we did not implement on top of HElib) obviously offer even more room for parallelism.

Organization. In Section 2 we review the main features of BGV-type cryptosystems [6, 5], and briefly survey the techniques for homomorphic computation on packed ciphertexts from SV and GHS [29, 15]. Then in Section 3 we describe our "general-purpose" optimizations on a high level, with additional details provided in Appendices A and B. A brief overview of AES and a high-level description and performance numbers is provided in Section 4.

[1] The need for coefficient representation ultimately stems from the fact that the noise in the ciphertexts is small in coefficient representation but not in evaluation representation.

2 Background

2.1 Notations and Mathematical Background

For an integer q we identify the ring $\mathbb{Z}/q\mathbb{Z}$ with the interval $(-q/2, q/2] \cap \mathbb{Z}$, and use $[z]_q$ to denote the reduction of the integer z modulo q into that interval. Our implementation utilizes polynomial rings defined by cyclotomic polynomials, $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$. The ring $\mathbb{A}$ is the ring of integers of a the mth cyclotomic number field $\mathbb{Q}(\zeta_m)$. We let $\mathbb{A}_q \stackrel{\text{def}}{=} \mathbb{A}/q\mathbb{A} = \mathbb{Z}[X]/(\Phi_m(X), q)$ for the (possibly composite) integer q, and we identify $\mathbb{A}_q$ with the set of integer polynomials of degree upto $\phi(m) - 1$ reduced modulo q.

Coefficient vs. Evaluation Representation. Let m, q be two integers such that $\mathbb{Z}/q\mathbb{Z}$ contains a primitive m-th root of unity, and denote one such primitive m-th root of unity by $\zeta \in \mathbb{Z}/q\mathbb{Z}$. Recall that the m'th cyclotomic polynomial splits into linear terms modulo q, $\Phi_m(X) = \prod_{i \in (\mathbb{Z}/m\mathbb{Z})^*} (X - \zeta^i) \pmod q$.

We consider two ways of representing an element $a \in \mathbb{A}_q$: Viewing a as a degree-$(\phi(m)-1)$ polynomial, $a(X) = \sum_{i<\phi(m)} a_i X^i$, the *coefficient representation* of a just lists all the coefficients in order $\mathbf{a} = \langle a_0, a_1, \ldots, a_{\phi(m)-1} \rangle \in (\mathbb{Z}/q\mathbb{Z})^{\phi(m)}$. For the other representation we consider the values that the polynomial $a(X)$ assumes on all primitive m-th roots of unity modulo q, $b_i = a(\zeta^i) \bmod q$ for $i \in (\mathbb{Z}/m\mathbb{Z})^*$. The b_i's in order also yield a vector $\mathbf{b} \in (\mathbb{Z}/q\mathbb{Z})^{\phi(m)}$, which we call the *evaluation representation* of a. Clearly these two representations are related via $\mathbf{b} = V_m \cdot \mathbf{a}$, where V_m is the Vandermonde matrix over the primitive m-th roots of unity modulo q. We remark that for all i we have the equality $(a \bmod (X - \zeta^i)) = a(\zeta^i) = b_i$, hence the evaluation representation of a is just a polynomial Chinese-Remaindering representation.

In both representations, an element $a \in \mathbb{A}_q$ is represented by a $\phi(m)$-vector of integers in $\mathbb{Z}/q\mathbb{Z}$. If q is a composite then each of these integers can itself be represented either using the standard binary encoding of integers or using Chinese-Remaindering relative to the factors of q. We usually use the standard binary encoding for the coefficient representation and Chinese-Remaindering for the evaluation representation. (Hence the latter representation is really a *double CRT* representation, relative to both the polynomial factors of $\Phi_m(X)$ and the integer factors of q.)

2.2 BGV-type Cryptosystems

Our implementation uses a variant of the BGV cryptosystem due to Gentry, Halevi and Smart, specifically the one described in [15, Appendix D] (in the full version). In this cryptosystem both ciphertexts and secret keys are vectors over the polynomial ring $\mathbb{A}$, and the native plaintext space is the space of binary polynomials $\mathbb{A}_2$. (More generally it could be $\mathbb{A}_p$ for some fixed $p \geq 2$, but in our case we will always use $\mathbb{A}_2$.)

At any point during the homomorphic evaluation there is some "current integer modulus q" and "current secret key $\mathbf{s}$", that change from time to time. A ciphertext $\mathbf{c}$ is decrypted using the current secret key $\mathbf{s}$ by taking inner product over $\mathbb{A}_q$ (with q the current modulus) and then reducing the result modulo 2 *in coefficient representation*. Namely, the decryption formula is

$$a \leftarrow [\, \underbrace{[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q}_{\text{noise}} \,]_2 \,. \tag{1}$$

The polynomial $[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q$ is called the "noise" in the ciphertext $\mathbf{c}$. Informally, $\mathbf{c}$ is a *valid ciphertext* with respect to secret key $\mathbf{s}$ and modulus q if this noise has "sufficiently small norm" relative to q. The meaning of "sufficiently small norm" is whatever is needed to ensure that the noise does not wrap around q when performing homomorphic operations, in our implementation we keep the norm of the noise always below some pre-set bound (which is determined in Appendix C.2).

Following [24, 15], the specific norm that we use to evaluate the magnitude of the noise is the "canonical embedding norm reduced mod q", specifically we use the conventions as described in [15, Appendix D] (in the full version). This is useful to get smaller parameters, but for the purpose of presentation the reader can think of the norm as the Euclidean norm of the noise in coefficient representation. More details are given in the Appendices. We refer to the norm of the noise as *the noise magnitude*.

The central feature of BGV-type cryptosystems is that the current secret key and modulus evolve as we apply operations to ciphertexts. We apply five different operations to ciphertexts during homomorphic evaluation. Three of them — addition, multiplication, and automorphism — are "semantic operations" that we use to evolve the plaintext data which is encrypted under those ciphertexts. The other two operations — key-switching and modulus-switching — are used for "maintenance": These operations do not change the plaintext at all, they only change the current key or modulus (respectively), and they are mainly used to control the complexity of the evaluation. Below we briefly describe each of these five operations on a high level. For the sake of self-containment, we also describe key generation and encryption in Appendix B. More detailed description can be found in [15, Appendix D].

Addition. Homomorphic addition of two ciphertext vectors with respect to the same secret key and modulus q is done just by adding the vectors over $\mathbb{A}_q$. If the two arguments were encrypting the plaintext polynomials $a_1, a_2 \in \mathbb{A}_2$ then the sum will be an encryption of $a_1 + a_2 \in \mathbb{A}_2$. This operation has no effect on the current modulus or key, and the norm of the noise is at most the sum of norms from the noise in the two arguments.

Multiplication. Homomorphic multiplication is done via tensor product over $\mathbb{A}_q$. In principle, if the two arguments have dimension n over $\mathbb{A}_q$ then the product ciphertext has dimension n^2, each entry in the output computed as the product of one entry from the first argument and one entry from the second.[2]

This operation does not change the current modulus, but it changes the current key: If the two input ciphertexts are valid with respect to the dimension-n secret key vector $\mathbf{s}$, encrypting the plaintext polynomials $a_1, a_2 \in \mathbb{A}_2$, then the output is valid with respect to the dimension-n^2 secret key $\mathbf{s}'$ which is the tensor product of $\mathbf{s}$ with itself, and it encrypts the polynomial $a_1 \cdot a_2 \in \mathbb{A}_2$. The norm of the noise in the product ciphertext can be bounded in terms of the product of norms of the noise in the two arguments. For our choice of norm function, the norm of the product is no larger than the product of the norms of the two arguments.

Key Switching. The public key of BGV-type cryptosystems includes additional components to enable converting a valid ciphertext with respect to one key into a valid ciphertext encrypting the same plaintext with respect to another key. For example, this is used to convert the product ciphertext which is valid with respect to a high-dimension key back to a ciphertext with respect to the original low-dimension key.

To allow conversion from dimension-n' key $\mathbf{s}'$ to dimension-n key $\mathbf{s}$ (both with respect to the same modulus q), we include in the public key a matrix $W = W[\mathbf{s}' \to \mathbf{s}]$ over $\mathbb{A}_q$, where the i'th column of W is roughly an encryption of the i'th entry of $\mathbf{s}'$ with respect to $\mathbf{s}$ (and the current modulus). Then given a valid ciphertext $\mathbf{c}'$ with respect to $\mathbf{s}'$, we roughly compute $\mathbf{c} = W \cdot \mathbf{c}'$ to get a valid ciphertext with respect to $\mathbf{s}$.

In some more detail, the BGV key switching transformation first ensures that the norm of the ciphertext $\mathbf{c}'$ itself is sufficiently low with respect to q. In [5] this was done by working with the binary encoding of $\mathbf{c}'$, and one of our main optimization in this work is a different method for achieving the same goal (cf. Section 3.1). Then, if the i'th entry in $\mathbf{s}'$ is $\mathfrak{s}'_i \in \mathbb{A}$ (with norm smaller than q), then the i'th column of $W[\mathbf{s}' \to \mathbf{s}]$ is an n-vector $\mathbf{w}_i$ such that $[\langle \mathbf{w}_i, \mathbf{s} \rangle \bmod \Phi_m(X)]_q = 2e_i + \mathfrak{s}'_i$ for a low-norm polynomial

[2] It was shown in [7] that over polynomial rings this operation can be implemented while increasing the dimension only to $2n-1$ rather than to n^2.

$e_i \in \mathbb{A}$. Denoting $\mathbf{e} = (e_1, \ldots, e_{n'})$, this means that we have $\mathbf{s}W = \mathbf{s}' + 2\mathbf{e}$ over $\mathbb{A}_q$. For any ciphertext vector $\mathbf{c}'$, setting $\mathbf{c} = W \cdot \mathbf{c}' \in \mathbb{A}_q$ we get the equation

$$[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q = [\mathbf{s}W\mathbf{c}' \bmod \Phi_m(X)]_q = [\langle \mathbf{c}', \mathbf{s}' \rangle + 2\langle \mathbf{c}', \mathbf{e} \rangle \bmod \Phi_m(X)]_q$$

Since $\mathbf{c}'$, $\mathbf{e}$, and $[\langle \mathbf{c}', \mathbf{s}' \rangle \bmod \Phi_m(X)]_q$ all have low norm relative to q, then the addition on the right-hand side does not cause a wrap around q, hence we get $[[\langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)]_q]_2 = [[\langle \mathbf{c}', \mathbf{s}' \rangle \bmod \Phi_m(X)]_q]_2$, as needed. The key-switching operation changes the current secret key from $\mathbf{s}'$ to $\mathbf{s}$, and does not change the current modulus. The norm of the noise is increased by at most an additive factor of $2\| \langle \mathbf{c}', \mathbf{e} \rangle \|$.

Modulus Switching. The modulus switching operation is intended to reduce the norm of the noise, to compensate for the noise increase that results from all the other operations. To convert a ciphertext $\mathbf{c}$ with respect to secret key $\mathbf{s}$ and modulus q into a ciphertext $\mathbf{c}'$ encrypting the same thing with respect to the same secret key but modulus q', we roughly just scale $\mathbf{c}$ by a factor q'/q (thus getting a fractional ciphertext), then round appropriately to get back an integer ciphertext. Specifically $\mathbf{c}'$ is a ciphertext vector satisfying (a) $\mathbf{c}' = \mathbf{c} \pmod 2$, and (b) the "rounding error term" $\tau \stackrel{\text{def}}{=} \mathbf{c}' - (q'/q)\mathbf{c}$ has low norm. Converting $\mathbf{c}$ to $\mathbf{c}'$ is easy in coefficient representation, and one of our optimizations is a method for doing the same in evaluation representation (cf. Section 3.2) This operation leaves the current key $\mathbf{s}$ unchanged, changes the current modulus from q to q', and the norm of the noise is changed as $\|n'\| \leq (q'/q)\|n\| + \|\tau \cdot s\|$. Note that if the key $\mathbf{s}$ has low norm and q' is sufficiently smaller than q, then the noise magnitude decreases by this operation.

A BGV-type cryptosystem has a chain of moduli, $q_0 < q_1 \cdots < q_{L-1}$, where fresh ciphertexts are with respect to the largest modulus q_{L-1}. During homomorphic evaluation every time the (estimated) noise grows too large we apply modulus switching from q_i to q_{i-1} in order to decrease it back. Eventually we get ciphertexts with respect to the smallest modulus q_0, and we cannot compute on them anymore (except by using bootstrapping).

Automorphisms. In addition to adding and multiplying polynomials, another useful operation is converting the polynomial $a(X) \in \mathbb{A}$ to $a^{(i)}(X) \stackrel{\text{def}}{=} a(X^i) \bmod \Phi_m(X)$. Denoting by κ_i the transformation $\kappa_i : a \mapsto a^{(i)}$, it is a standard fact that the set of transformations $\{\kappa_i : i \in (\mathbb{Z}/m\mathbb{Z})^*\}$ forms a group under composition (which is the Galois group $\mathcal{G}\mathsf{al}(\mathbb{Q}(\zeta_m)/\mathbb{Q})$), and this group is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$. In [5, 15] it was shown that applying the transformations κ_i to the plaintext polynomials is very useful, some more examples of its use can be found in our Section 4.

Denoting by $\mathbf{c}^{(i)}$, $\mathbf{s}^{(i)}$ the vector obtained by applying κ_i to each entry in $\mathbf{c}, \mathbf{s}$, respectively, it was shown in [5, 15] that if $\mathbf{s}$ is a valid ciphertext encrypting a with respect to key $\mathbf{s}$ and modulus q, then $\mathbf{c}^{(i)}$ is a valid ciphertext encrypting $a^{(i)}$ with respect to key $\mathbf{s}^{(i)}$ and the same modulus q. Moreover the norm of noise remains the same under this operation. We remark that we can apply key-switching to $\mathbf{c}^{(i)}$ in order to get an encryption of $a^{(i)}$ with respect to the original key $\mathbf{s}$.

2.3 Computing on Packed Ciphertexts

Smart and Vercauteren observed [28, 29] that the plaintext space $\mathbb{A}_2$ can be viewed as a vector of "plaintext slots", by an application the polynomial Chinese Remainder Theorem. Specifically, if the ring polynomial $\Phi_m(X)$ factors modulo 2 into a product of irreducible factors $\Phi_m(X) = \prod_{j=0}^{\ell-1} F_j(X) \pmod 2$, then a plaintext polynomial $a(X) \in \mathbb{A}_2$ can be viewed as encoding ℓ different small polynomials, $a_j = a \bmod F_j$. Just like for integer Chinese Remaindering, addition and multiplication in $\mathbb{A}_2$ correspond to element-wise addition and multiplication of the vectors of slots.

The effect of the automorphisms is a little more involved. When i is a power of two then the transformations $\kappa_i : a \mapsto a^{(i)}$ is just applied to each slot separately. When i is not a power of two the transformation κ_i has the effect of roughly shifting the values between the different slots. For example, for some parameters we could get a cyclic shift of the vector of slots: If a encodes the vector $(a_0, a_1, \ldots, a_{\ell-1})$, then $\kappa_i(a)$ (for some i) could encode the vector $(a_{\ell-1}, a_0, \ldots, a_{\ell-2})$. This was used in [15] to devise efficient procedures for applying arbitrary permutations to the plaintext slots.

We note that the values in the plaintext slots are not just bits, rather they are polynomials modulo the irreducible F_j's, so they can be used to represents elements in extension fields GF(2^d). In particular, in our AES implementations we used the plaintext slots to hold elements of GF(2^8), and encrypt one byte of the AES state in each slot. Then we can use an adaption of the techniques from [15] to permute the slots when performing the AES row-shift and column-mix.

3 General-Purpose Optimizations

Below we summarize our optimizations that are not tied directly to the AES circuit and can be used also in homomorphic evaluation of other circuits. Underlying many of these optimizations is our choice of keeping ciphertext and key-switching matrices in evaluation (double-CRT) representation. Roughly speaking, our chain of moduli is defined via a set of same-size primes, $p_0, p_1, p_2, \ldots$, chosen such that $\mathbb{Z}/p_i\mathbb{Z}$ has m'th roots of unity. (In other words, $m|p_i - 1$ for all i.) For $i = 0, \ldots, L-1$ we then define our i'th modulus as $q_i = \prod_{j=0}^{i} p_i$. To gain efficiency, we actually choose p_0 to be half the bit-size of the other p_i's, and so the odd indexed moduli in the chain are a product of the primes starting at p_0 ($q_i = \prod_{i=0}^{\lfloor i/2 \rfloor} p_i$) and the even-indexed moduli are products that do not include p_0 ($q_i = \prod_{i=1}^{i/2} p_i$). In our implementation the half-sized prime has 23-25 bits (and the full-sized primes therefore have 46-50 bits). For easy of exposition, however, in the rest of this report we ignore this "half-sized" prime and describe all our optimizations as if we were using only a chain of same-size primes.

In the t-th level of the scheme we have ciphertexts consisting of elements in $\mathbb{A}_{q_t}$ (i.e., polynomials modulo $(\Phi_m(X), q_t)$). We represent an element $c \in \mathbb{A}_{q_t}$ by a $\phi(m) \times (t+1)$ "matrix" of its evaluations at the primitive m-th roots of unity modulo the primes $p_0, \ldots, p_t$. Computing this representation from the coefficient representation of c involves reducing c modulo the p_i's and then $t+1$ invocations of the FFT algorithm, modulo each of the p_i (picking only the FFT coefficients corresponding to $(\mathbb{Z}/m\mathbb{Z})^*$). To convert back to coefficient representation we invoke the inverse FFT algorithm, each time padding the $\phi(m)$-vector of evaluation point with $m - \phi(m)$ zeros (for the evaluations at the non-primitive roots of unity). This yields the coefficients of the polynomials modulo $(X^m - 1, p_i)$ for $i = 0, \ldots, t$, we then reduce each of these polynomials modulo $(\Phi_m(X), p_i)$ and apply Chinese Remainder interpolation. We stress that we try to perform these transformations as rarely as we can.

3.1 A New Variant of Key Switching

As described in Section 2, the key-switching transformation introduces an additive factor of $2\langle \mathbf{c}', \mathbf{e} \rangle$ in the noise, where $\mathbf{c}'$ is the input ciphertext and $\mathbf{e}$ is the noise component in the key-switching matrix. To keep the noise magnitude below the modulus q, it seems that we need to ensure that the ciphertext $\mathbf{c}'$ itself has low norm. In BGV [5] this was done by representing $\mathbf{c}'$ as a fixed linear combination of small vectors, i.e. $\mathbf{c}' = \sum_i 2^i \mathbf{c}'_i$ with $\mathbf{c}'_i$ the vector of i'th bits in $\mathbf{c}'$. Considering the high-dimension ciphertext $\mathbf{c}^* = (\mathbf{c}'_0|\mathbf{c}'_1|\mathbf{c}'_2|\cdots)$ and secret key $\mathbf{s}^* = (\mathbf{s}'|2\mathbf{s}'|4\mathbf{s}'|\cdots)$, we note that we have $\langle \mathbf{c}^*, \mathbf{s}^* \rangle = \langle \mathbf{c}', \mathbf{s}' \rangle$, and $\mathbf{c}^*$ has low norm (since it consists of 0-1 polynomials). BGV therefore included in the public key the matrix

$W = W[\mathbf{s}^* \to \mathbf{s}]$ (rather than $W[\mathbf{s}' \to \mathbf{s}]$), and had the key-switching transformation computes $\mathbf{c}^*$ from $\mathbf{c}'$ and sets $\mathbf{c} = W \cdot \mathbf{c}^*$.

When implementing key-switching, there are two drawbacks to the above approach. First, this increases the dimension (and hence the size) of the key switching matrix. This drawback is fatal when evaluating deep circuits, since having enough memory to keep the key-switching matrices turns out to be a limiting factor in our ability to evaluate such circuits. In addition, for this key-switching we must first convert $\mathbf{c}'$ to coefficient representation (in order to compute the $\mathbf{c}'_i$'s), then convert each of the $\mathbf{c}'_i$'s back to evaluation representation before multiplying by the key-switching matrix. In level t of the circuit, this seem to require $\Omega(t \log q_t)$ FFTs.

In this work we propose a different variant: Rather than manipulating $\mathbf{c}'$ to decrease its norm, we instead temporarily increase the modulus q. We recall that for a valid ciphertext $\mathbf{c}'$, encrypting plaintext a with respect to $\mathbf{s}'$ and q, we have the equality $\langle \mathbf{c}', \mathbf{s}' \rangle = 2e' + a$ over A_q, for a low-norm polynomial e'. This equality, we note, implies that for every odd integer p we have the equality $\langle \mathbf{c}', p\mathbf{s}' \rangle = 2e'' + a$, *holding over* A_{pq}, for the "low-norm" polynomial e'' (namely $e'' = p \cdot e' + \frac{p-1}{2}a$). Clearly, when considered relative to secret key $p\mathbf{s}$ and modulus pq, the noise in $\mathbf{c}'$ is p times larger than it was relative to $\mathbf{s}$ and q. However, since the modulus is also p times larger, we maintain that the noise has norm sufficiently smaller than the modulus. In other words, $\mathbf{c}'$ is still a valid ciphertext that encrypts the same plaintext a with respect to secret key $p\mathbf{s}$ and modulus pq. By taking p large enough, we can ensure that the norm of $\mathbf{c}'$ (which is independent of p) is sufficiently small relative to the modulus pq.

We therefore include in the public key a matrix $W = W[p\mathbf{s}' \to \mathbf{s}]$ modulo pq for a large enough odd integer p. (Specifically we need $p \approx q\sqrt{m}$.) Given a ciphertext $\mathbf{c}'$, valid with respect to $\mathbf{s}$ and q, we apply the key-switching transformation simply by setting $\mathbf{c} = W \cdot \mathbf{c}'$ over $\mathbb{A}_{pq}$. The additive noise term $\langle \mathbf{c}', \mathbf{e} \rangle$ that we get is now small enough relative to our large modulus pq, thus the resulting ciphertext $\mathbf{c}$ is valid with respect to $\mathbf{s}$ and pq. We can now switch the modulus back to q (using our modulus switching routine), hence getting a valid ciphertext with respect to $\mathbf{s}$ and q.

We note that even though we no longer break $\mathbf{c}'$ into its binary encoding, it seems that we still need to recover it in coefficient representation in order to compute the evaluations of $\mathbf{c}' \bmod p$. However, since we do not increase the dimension of the ciphertext vector, this procedure requires only $O(t)$ FFTs in level t (vs. $O(t \log q_t) = O(t^2)$ for the original BGV variant). Also, the size of the key-switching matrix is reduced by roughly the same factor of $\log q_t$.

Our new variant comes with a price tag, however: We use key-switching matrices relative to a larger modulus, but still need the noise term in this matrix to be small. This means that the LWE problem underlying this key-switching matrix has larger ratio of modulus/noise, implying that we need a larger dimension to get the same level of security than with the original BGV variant. In fact, since our modulus is more than squared (from q to pq with $p > q$), the dimension is increased by more than a factor of two. This translates to more than doubling of the key-switching matrix, partly negating the size and running time advantage that we get from this variant.

Of course, one can also use a hybrid of the two approaches: we can decrease the norm of $\mathbf{c}'$ only somewhat by breaking it into a few digits (as opposed to binary bits as in [5]), and then increase the modulus somewhat until it is large enough relative to the smaller norm of $\mathbf{c}'$. The HElib implementation indeed let us break $\mathbf{c}$ to any number of digits, upto the number of primes in the chain, and in our experiments we used anywhere between 3 and 6 digits to get the right level of security for the different settings.

3.2 Modulus Switching in Evaluation Representation

Given an element $c \in \mathbb{A}_{q_t}$ in evaluation (double-CRT) representation relative to $q_t = \prod_{j=0}^{t} p_j$, we want to modulus-switch to q_{t-1} – i.e., scale down by a factor of p_t; we call this operation $\mathsf{Scale}(c, q_t, q_{t-1})$. The output should be $c' \in \mathbb{A}$, represented via the same double-CRT format (with respect to $p_0, \ldots, p_{t-1}$), such that (a) $c' \equiv c \pmod 2$, and (b) the "rounding error term" $\tau = c' - (c/p_t)$ has a very low norm. As p_t is odd, we can equivalently require that the element $c^\dagger \stackrel{\text{def}}{=} p_t \cdot c'$ satisfy

(i) $c^\dagger$ is divisible by p_t,

(ii) $c^\dagger \equiv c \pmod 2$, and

(iii) $c^\dagger - c$ (which is equal to $p_t \cdot \tau$) has low norm.

Rather than computing c' directly, we will first compute $c^\dagger$ and then set $c' \leftarrow c^\dagger/p_t$. Observe that once we compute $c^\dagger$ in double-CRT format, it is easy to output also c' in double-CRT format: given the evaluations for $c^\dagger$ modulo p_j ($j < t$), simply multiply them by $p_t^{-1} \bmod p_j$. The algorithm to output $c^\dagger$ in double-CRT format is as follows:

1. Set $\bar{c}$ to be the coefficient representation of $c \bmod p_t$. (Computing this requires a single "small FFT" modulo the prime p_t.)
2. Add or subtract p_t from every odd coefficient of $\bar{c}$, thus obtaining a polynomial δ with coefficients in $(-p_t, p_t]$ such that $\delta \equiv \bar{c} \equiv c \pmod{p_t}$ and $\delta \equiv 0 \pmod 2$.
3. Set $c^\dagger = c - \delta$, and output it in double-CRT representation.

   Since we already have c in double-CRT representation, we only need the double-CRT representation of δ, which requires t more "small FFTs" modulo the p_j's.

As all the coefficients of $c^\dagger$ are within p_t of those of c, the "rounding error term" $\tau = (c^\dagger - c)/p_t$ has coefficients of magnitude at most one, hence it has low norm.

The procedure above uses $t + 1$ small FFTs in total. This should be compared to the naive method of just converting everything to coefficient representation modulo the primes ($t + 1$ FFTs), CRT-interpolating the coefficients, dividing and rounding appropriately the large integers (of size $\approx q_t$), CRT-decomposing the coefficients, and then converting back to evaluation representation ($t + 1$ more FFTs). The above approach makes explicit use of the fact that we are working in a plaintext space modulo 2; in Appendix D we present a technique which works when the plaintext space is defined modulo a larger modulus.

3.3 Dynamic Noise Management

As described in the literature, BGV-type cryptosystems tacitly assume that each homomorphic operation operation is followed a modulus switch to reduce the noise magnitude. In our implementation, however, we attach to each ciphertext an estimate of the noise magnitude in that ciphertext, and use these estimates to decide dynamically when a modulus switch must be performed.

Each modulus switch consumes a level, and hence a goal is to reduce, over a computation, the number of levels consumed. By paying particular attention to the parameters of the scheme, and by carefully analyzing how various operations affect the noise, we are able to control the noise much more carefully than in prior work. In particular, we note that modulus-switching is really only necessary just prior to multiplication (when the noise magnitude is about to get squared), in other times it is acceptable to keep the ciphertexts at a higher level (with higher noise).

4 Homomorphic Evaluation of AES

Next we describe our homomorphic implementation of AES-128. Our main impelemntation is "packed", namely the entire AES state is packed in just one ciphertext. Two other possible implementations (of byte-slice and bit-slice AES) are described later in Section 4.2. We note that in our earlier work we implemented all htree versions, but in the newer work we only re-implemented the "packed" version.

A Brief Overview of AES. The AES-128 cipher consists of ten applications of the same keyed round function (with different round keys). The round function operates on a 4×4 matrix of bytes, which are sometimes considered as element of $\mathbb{F}_{2^8}$. The basic operations that are performed during the round function are AddKey, SubBytes, ShiftRows, MixColumns. The AddKey is simply an XOR operation of the current state with 16 bytes of key; the SubBytes operation consists of an inversion in the field $\mathbb{F}_{2^8}$ followed by a fixed $\mathbb{F}_2$-affine map on the bits of the element; the ShiftRows rotates the entries in the row i of the 4×4 matrix by $i-1$ places to the left; finally the MixColumns operations pre-multiplies the state matrix by a fixed 4×4 matrix.

Our Packed Representation of the AES state. For our implementation we chose the native plaintext space of our homomorphic encryption so as to support operations on the finite field $\mathbb{F}_{2^8}$. To this end we choose our ring polynomial as $\Phi_m(X)$ that factors modulo 2 into degree-d irreducible polynomials such that $8|d$. (In other words, the smallest integer d such that $m|(2^d-1)$ is divisible by 8.) This means that our plaintext slots can hold elements of $\mathbb{F}_{2^d}$, and in particular we can use them to hold elements of $\mathbb{F}_{2^8}$ which is a sub-field of $\mathbb{F}_{2^d}$. Since we have $\ell = \phi(m)/d$ plaintext slots in each ciphertext, we can represent upto $\lfloor \ell/16 \rfloor$ complete AES state matrices per ciphertext.

Moreover, we choose our parameter m so that there exists an element $g \in \mathbb{Z}_m^*$ that has order 16 in both $\mathbb{Z}_m^*$ and the quotient group $\mathbb{Z}_m^*/\langle 2 \rangle$. This condition means that if we put 16 plaintext bytes in slots $t, tg, tg^2, tg^3, \ldots$ (for some $t \in \mathbb{Z}_m^*$), then the conjugation operation $X \mapsto X^g$ implements a cyclic right shift over these sixteen plaintext bytes. Below we denote the vector of plaintext slots by $a = (\alpha_i)_{i=1}^{\ell}$, with each $\alpha_i \in \mathbb{F}_{2^8}$. We place the 16 bytes of the AES state in plaintext slots using column-first ordering, namely we have

$$a \approx [\alpha_{00}\alpha_{10}\alpha_{20}\alpha_{30}\alpha_{01}\alpha_{11}\alpha_{21}\alpha_{31}\alpha_{02}\alpha_{12}\alpha_{22}\alpha_{32}\alpha_{03}\alpha_{13}\alpha_{23}\alpha_{33}],$$

representing the input plaintext matrix

$$A = (\alpha_{ij})_{i,j} = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix}.$$

4.1 Homomorphic Evaluation of the Basic Operations

We now examine each AES operation in turn, and describe how it is implemented homomorphically.

4.1.1 AddKey and SubBytes

The AddKey is just a simple addition of ciphertexts, which yields a 4×4 matrix of bytes in the input to the SubBytes operation.

During S-box lookup, each plaintext byte α_{ij} should be replaced by $\beta_{ij} = S(\alpha_{ij})$, where $S(\cdot)$ is a fixed permutation on the bytes. Specifically, $S(x)$ is obtained by first computing $y = x^{-1}$ in $\mathbb{F}_{2^8}$ (with 0 mapped to 0), then applying a bitwise affine transformation $z = T(y)$ where elements in $\mathbb{F}_{2^8}$ are treated as bit strings with representation polynomial $G(X) = x^8 + x^4 + x^3 + x + 1$.

We implement $\mathbb{F}_{2^8}$ inversion followed by the $\mathbb{F}_2$ affine transformation using the Frobenius automorphisms, $X \longrightarrow X^{2^j}$. Recall that the transformation $\kappa_{2^j}(a(X)) = (a(X^{2^j}) \bmod \Phi_m(X))$ is applied separately to each slot, hence we can use it to transform the vector $(\alpha_i)_{i=1}^{\ell}$ into $(\alpha_i^{2^j})_{i=1}^{\ell}$. We note that applying the Frobenius automorphisms to ciphertexts has almost no influence on the noise magnitude, and hence it does not consume any levels.[3]

Inversion over $\mathbb{F}_{2^8}$ is done using essentially the same procedure as Algorithm 2 from [27] for computing $\beta = \alpha^{-1} = \alpha^{254}$. This procedure takes only three Frobenius automorphisms and four multiplications, arranged in a depth-3 circuit (see details below.) To apply the AES $\mathbb{F}_2$ affine transformation, we use the fact that any $\mathbb{F}_2$ affine transformation can be computed as a $\mathbb{F}_{2^8}$ affine transformation over the conjugates. Thus there are constants $\gamma_0, \gamma_1, \ldots, \gamma_7, \delta \in \mathbb{F}_{2^8}$ such that the AES affine transformation $T_{\mathrm{AES}}(\cdot)$ can be expressed as $T_{\mathrm{AES}}(\beta) = \delta + \sum_{j=0}^{7} \gamma_j \cdot \beta^{2^j}$ over $\mathbb{F}_{2^8}$. We therefore again apply the Frobenius automorphisms to compute eight ciphertexts encrypting the polynomials $\kappa_{2^j}(b)$ for $j = 0, 1, \ldots, 7$, and take the appropriate linear combination (with coefficients the γ_j's) to get an encryption of the vector $(T_{\mathrm{AES}}(\alpha_i^{-1}))_{i=1}^{\ell}$. For our parameters, a multiplication-by-constant operation consumes roughly half a level in terms of added noise.

One subtle implementation detail to note here, is that although our plaintext slots all hold elements of the same field $\mathbb{F}_{2^8}$, they hold these elements with respect to different polynomial encodings. The AES affine transformation, on the other hand, is defined with respect to one particular fixed polynomial encoding. This means that we must implement in the i'th slot not the affine transformation $T_{\mathrm{AES}}(\cdot)$ itself but rather the projection of this transformation onto the appropriate polynomial encoding: When we take the affine transformation of the eight ciphertexts encrypting $b_j = \kappa_{2^j}(b)$, we therefore multiply the encryption of b_j not by a constant that has γ_j in all the slots, but rather by a constant that has in slot i the projection of γ_j to the polynomial encoding of slot i.

Below we provide a pseudo-code description of our S-box lookup implementation, together with an approximation of the levels that are consumed by these operations.

	Level	
Input: ciphertext $\mathfrak{c}$	t	
// Compute $\mathfrak{c}_{254} = \mathfrak{c}^{-1}$		
1. $\mathfrak{c}_2 \leftarrow \mathfrak{c} \gg 2$	t	// Frobenius $X \mapsto X^2$
2. $\mathfrak{c}_3 \leftarrow \mathfrak{c} \times \mathfrak{c}_2$	$t-1$	// Multiplication
3. $\mathfrak{c}_{12} \leftarrow \mathfrak{c}_3 \gg 4$	$t-1$	// Frobenius $X \mapsto X^4$
4. $\mathfrak{c}_{14} \leftarrow \mathfrak{c}_{12} \times \mathfrak{c}_2$	$t-2$	// Multiplication
5. $\mathfrak{c}_{15} \leftarrow \mathfrak{c}_{12} \times \mathfrak{c}_3$	$t-2$	// Multiplication
6. $\mathfrak{c}_{240} \leftarrow \mathfrak{c}_{15} \gg 16$	$t-2$	// Frobenius $X \mapsto X^{16}$
7. $\mathfrak{c}_{254} \leftarrow \mathfrak{c}_{240} \times \mathfrak{c}_{14}$	$t-3$	// Multiplication
// Affine transformation over $\mathbb{F}_2$		
8. $\mathfrak{c}'_{2^j} \leftarrow \mathfrak{c}_{254} \gg 2^j$ for $j = 0, 1, 2, \ldots, 7$	$t-3$	// Frobenius $X \mapsto X^{2^j}$
9. $\mathfrak{c}'' \leftarrow \gamma + \sum_{j=0}^{7} \gamma_j \times \mathfrak{c}'_{2^j}$	$t-3.5$	// Linear combination over $\mathbb{F}_{2^8}$

[3] It does increase the noise magnitude somewhat, because we need to do key switching after these automorphisms. But this is only a small influence, and we will ignore it here.

4.1.2 ShiftRows **and** MixColumns

As commonly done, we lump together the ShiftRows/MixColumns operations, viewing both as a single linear transformation over vectors from $(\mathbb{F}_{2^8})^{16}$. As mentioned above, by a careful choice of the parameter m and the placement of the AES state bytes in our plaintext slots, we can implement a rotation-by-i of the rows of the AES matrix as a single automorphism operations $X \mapsto X^{g^i}$ (for some element $g \in (\mathbb{Z}/m\mathbb{Z})^*$). Given the ciphertext $\mathfrak{c}''$ after the SubBytes step, we use these operations in conjunction with ℓ-SELECT operations (as described in [15]) to compute four ciphertexts corresponding to the appropriate permutations of the 16 bytes (in each of the $\ell/16$ different input blocks). These four ciphertexts are combined via a linear operation (with coefficients $1, X$, and $(1+X)$) to obtain the final result of this round function.

Moreover, the multiply-by-constant operations implied by ℓ-SELECT can be folded into the multiply-by-constant operations of the linear transformations, hence the entire shift-row/mix-column operation consumes only 1/2 level in terms of noise. Finally, it is possible to implement the entire procedure using only six rotation operations, as described next. Recall our column-byte-ordering of the AES state:

$$a \approx [\alpha_{00}\alpha_{10}\alpha_{20}\alpha_{30}\alpha_{01}\alpha_{11}\alpha_{21}\alpha_{31}\alpha_{02}\alpha_{12}\alpha_{22}\alpha_{32}\alpha_{03}\alpha_{13}\alpha_{23}\alpha_{33}]$$
$$A = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{pmatrix}.$$

We apply to the state vector a three right-rotations by 11, 6, and 1 positions to get the three vectors a_{11}, a_6, a_1 representing the matrices A_{11}, A_6, A_1, respectively:

$$a_{11} \approx [\alpha_{11}\alpha_{21}\alpha_{31}\ldots\alpha_{30}\alpha_{01}] \qquad a_6 \approx [\alpha_{22}\alpha_{32}\alpha_{03}\ldots\alpha_{02}\alpha_{12}] \qquad a_1 \approx [\alpha_{33}\alpha_{00}\alpha_{10}\ldots\alpha_{13}\alpha_{23}]$$
$$A_{11} = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{10} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{20} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{30} \\ \alpha_{02} & \alpha_{03} & \alpha_{00} & \alpha_{01} \end{pmatrix} \quad A_6 = \begin{pmatrix} \alpha_{22} & \alpha_{23} & \alpha_{20} & \alpha_{21} \\ \alpha_{32} & \alpha_{33} & \alpha_{30} & \alpha_{31} \\ \alpha_{03} & \alpha_{00} & \alpha_{01} & \alpha_{02} \\ \alpha_{13} & \alpha_{10} & \alpha_{11} & \alpha_{12} \end{pmatrix} \quad A_1 = \begin{pmatrix} \alpha_{33} & \alpha_{30} & \alpha_{31} & \alpha_{32} \\ \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \end{pmatrix}$$

Considering the top row in the four matrices (consisting of the bytes in positions 0,4,8,12), we see that we get exactly the four rows of the matrix after the shift-row operations. Hence these four bytes in the four matrices are exactly aligned so we can use SIMD operations to compute the column-mix operations. We next multiply these matrices by constants that have 0's in all positions except 0,4,8,12, and in those selected positions they have either 1, X, or $X+1$. Below we denote these constants by $\mathsf{C_1}$, $\mathsf{C_X}$ and $\mathsf{C_{X+1}}$, respectively. Setting

$$B'_0 = A\cdot\mathsf{C_X} + (A_1+A_6)\cdot\mathsf{C_1} + A_{11}\cdot\mathsf{C_{X+1}}, \qquad B'_1 = (A+A_1)\cdot\mathsf{C_1} + A_6\cdot\mathsf{C_{X+1}} + A_{11}\cdot\mathsf{C_X},$$
$$B'_2 = (A+A_{11})\cdot\mathsf{C_1} + A_1\cdot\mathsf{C_{X+1}} + A_6\cdot\mathsf{C_X}, \qquad B'_3 = A\cdot\mathsf{C_{X+1}} + A_1\cdot\mathsf{C_X} + (A_6+A_{11})\cdot\mathsf{C_1}$$

we get that the top rows of the four B'_i's contain the four rows of the resulting matrix B after mix-column, and moreover all the other rows in the B'_i's are zero. Having computed all the rows of the result, we use three more rotations to move them to place, namely set $B = B'_0 + (B'_1 \gg 1) + (B'_2 \gg 2) + (B'_3 \gg 3)$. A pseudo-code of the combined shift-row/mix-column operation is given below:

		Level	
Input: ciphertext $\mathfrak{c}''$		$t-3.5$	
10.	$\mathfrak{c}''_j \leftarrow \mathfrak{c}'' \gg j$ for $j = 0, 1, 6, 11$	$t-3.5$	// Rotations
11.	$\mathfrak{c}^*_0 \leftarrow \mathfrak{c}''_0 \cdot \mathsf{C}_\mathsf{X} + (\mathfrak{c}''_1 + \mathfrak{c}''_6)\mathsf{C}_1 + \mathfrak{c}''_{11} \cdot \mathsf{C}_{\mathsf{X}+1}$ $\mathfrak{c}^*_1 \leftarrow (\mathfrak{c}''_0 + \mathfrak{c}''_1)\mathsf{C}_1 + \mathfrak{c}''_6 \cdot \mathsf{C}_{\mathsf{X}+1} + \mathfrak{c}''_{11} \cdot \mathsf{C}_\mathsf{X}$ $\mathfrak{c}^*_2 \leftarrow (\mathfrak{c}''_0 + \mathfrak{c}''_{11})\mathsf{C}_1 + \mathfrak{c}''_1 \cdot \mathsf{C}_{\mathsf{X}+1} + \mathfrak{c}''_6 \cdot \mathsf{C}_\mathsf{X}$ $\mathfrak{c}^*_3 \leftarrow \mathfrak{c}''_0 \cdot \mathsf{C}_{\mathsf{X}+1} + \mathfrak{c}''_1 \cdot \mathsf{C}_\mathsf{X} + (\mathfrak{c}''_6 + \mathfrak{c}''_{11})\mathsf{C}_1$	$t-4$	// Linear combinations
12.	Output $\mathfrak{c}^*_0 + (\mathfrak{c}^*_1 \gg 1) + (\mathfrak{c}^*_2 \gg 2) + (\mathfrak{c}^*_3 \gg 3)$	$t-4$	// Assembling the result

4.1.3 The Cost of One Round Function

The above description yields an estimate of 4 levels for implementing one round function, which is indeed what we get in our experiments. The time complexity is dominated by the number of key-switching operations, which we need to do for every multiplication and every automorphism. The byte-substitution takes three multiplications and four automorphisms for inversion, and seven more automorphisms for the affine transformation, for a total of 14 key-switches. The shift-row/mix-column operation adds six more automorphisms, for a grand total of 20 key-switches per round.

We mention that the byte-slice implementation in Section 4.2 below would consume the same number of levels but use less key-switching operations per round since the shift-row/column-mix operation no longer needs automorphisms. Hence we would get 14 rather than 20 key-switching operations per round, so we expect the amortized complexity of this implementation to be faster by a factor of $20/14 \approx 1.4$. However, since we need to manipulate 16 times as many ciphertexts, the implementation would take much more time per evaluation (by a factor of $16 \cdot 14/20 = 11.2$) and require more memory.

4.2 Byte- and Bit-Slice Implementations

In the byte sliced implementation we use sixteen distinct ciphertexts to represent a single state matrix. (But since each ciphertext can hold ℓ plaintext slots, then these 16 ciphertexts can hold the state of ℓ different AES blocks). In this representation there is no interaction between the slots, thus we operate with pure ℓ-fold SIMD operations. The AddKey and SubBytes steps are exactly as above (except applied to 16 ciphertexts rather than a single one). The permutations in the ShiftRows/MixColumns step are now "for free", but the scalar multiplication in MixColumns still consumes 1/2 level in the modulus chain.

For the bit sliced implementation we represent the entire round function as a binary circuit, and we use 128 distinct ciphertexts (one per bit of the state matrix). However each set of 128 ciphertexts is able to represent a total of ℓ distinct blocks. The main issue here is how to create a circuit for the round function which is as shallow, in terms of number of multiplication gates, as possible. Again the main issue is the SubBytes operation as all operations are essentially linear. To implement the SubBytes we used the "depth-16" circuit of Boyar and Peralta [3], which consumes four levels. The rest of the round function can be represented as a set of bit-additions, Thus, implementing this method means that we should again consume only four levels per level.

4.3 Using Bootstrapping

Without bootstrapping, implementing ten rounds requires over 40 levels in the modulus chain, which means that we need a very large dimension to get security. We could hope to use the "bootstrapping as optimization" technique from BGV [5] to get smaller dimension, and hence speed up the computation. As it turns

Test	m	$\phi(m)$	lvls	$\|Q\|$	security	params/key-gen	Encrypt	Decrypt	memory
no bootstrap	53261	46080	40	886	150-bit	26.45 / 73.03	245.1	394.3	3GB
bootstrap	28679	23040	23	493	123-bit	148.2 / 37.2	1049.9	1630.5	3.7GB

Table 1: Performence results of homomorphic AES. Time is in seconds, the modulus size $|Q|$ includes extra primes as in Section 3.1.

out, however, the reduction in dimension is not enough to compensate for the extra time spent in the recryption procedure itself, so this does not lead to faster process. Bootstrapping is still needed, however, in applications that further process the result of the AES encryption. Hence in our implementation we also tested incorporating recryption into the AES computation.

One avenue for optimization in this case is to recrypt several ciphertexts together: The implementation of recryption in HElib handles "fully packed ciphertexts" whose slots contain elements from $\mathbb{F}_{2^d}$ (for some d divisible by 8), but our AES implementation only uses $\mathbb{F}_{2^8}$ elements (i.e. bytes) in the slots. We can therefore recrypt several ciphertexts together, packing $d/8$ bytes in each slot. Since in this setting most of the AES computation time is spent on recryption, we can process $d/8$ ciphertexts at nearly the same time as we do a single ciphertext, yielding a nearly $d/8$ speedup in amortized time. In our experiments we used $d = 24$, so this yields roughly a $3\times$ improvement.

4.4 Performance Details

As remarked in the introduction, we tested our implementations on a two-year-old Lenovo X230 laptop with Intel Core i5-3320M running at 2.6GHz, on an Ubuntu 14.04 VM with 4GB of RAM, using the g++ compiler version 4.9.2. The results of these tests are summarized in Table 1.

Non-bootstrapping implementation. For the non-bootstrapping experiment we selected parameters large enough to cope with 40 levels of computation. Appendix C contains our old derivation of the parameters to use, in our newer implementation we used instead the HElib derivation (that takes into consideration also the hybrid approach from Section 3.1), and is described in the HElib design document [18, Sec 3.1.4]. A rule-of-thumb is that for an L-level computation we need the dimension to be roughly $1000 \cdot L$. Specifically here we worked with the m-th cyclotomic for $m = 53261$, which yields lattices of dimension $\phi(m) = 46080$. This setting has 1920 slots, so we can fit $1920/16 = 120$ AES blocks in each ciphertext.

For this setting, key-generation took about 1.5 minutes, of which roughly 30 seconds were spent computing key-independent tables and about one minute was spent generating the keys and key-switching matrices. The input to the actual computation consisted of 120 plaintext blocks (in cleartext), and the eleven AES round keys encrypted in eleven packed ciphertext using our homomorphic encryption scheme. Homomorphic AES-encryption operation took 252 seconds, yielding throughput of 2 seconds per block.

Implementation using bootstrapping. Since bootstrapping in HElib takes about 12 levels, we chose our parameters here to cope with more than 20 levels of computation, so that we can compute at least two AES rounds per recryption. Specifically we had 23 computation levels and worked with $m = 28679$ and $\phi(m) = 23040$, a setting that yields 123-bit security by our estimates (see Equation (8) in Appendix C). This setting features 960 slots per ciphertext, each holding an element of $\mathbb{F}_{2^{24}}$, which is enough to pack 60 AES blocks.

Key-generation for this setting took about four minutes, three of which were spent computing key-independent tables, and under one minute spent on generating the keys and key-switching matrices. The input to the actual computation consisted of 180 plaintext blocks (in cleartext), and the same 11 packed cipehrtext encrypting the AES round keys. During the computation we applied the AES operation to three ciphertexts in parallel, and packed them into a single cipehrtext before each recryption.

The AES-encryption operation took 1050 seconds, of which 823 seconds were spent during two recryption operations, and the other 227 seconds were spent on the AES computation of the three ciphertexts. With 180 blocks, this gives throughput of 5.8 seconds per block. The entire computation used 3.7GB of memory.

Implementing AES decryption. We also implemented the AES decryption operation, basically by just reversing all the operations of the AES-encryption circuit. The operations performed in both cases are nearly identical (except a few multiply-by-constant operations), and yet in our tests the decryption time was about 60% slower than encryption.

For the non-bootstrapping case, one reason is that the AES encryption operation begins with inversion that lowers the level of the ciphertext, whereas decryption begins with the linear operations that keep the level more or less the same. As a result, operations on decryption are performed 2-3 levels higher than on encryption, which means that they need to manipulate more primes in our chain of moduli. It is not clear to us why this causes such a large slowdown, we speculate that some of it is the result of memory swapping or some other low-level effects.

For the bootstrapping case, the reason for the large slowdown is that the last inversion operation on decryption happens quite low in the chain, which triggers one more recryption operation, three on decryption vs. two on encryption. (This artifactc can probably be removed by special-casing the last round, but we did not attempt to do it.)

Acknowledgments

We thank Jean-Sebastien Coron for pointing out to us the efficient implementation from [27] of the AES S-box lookup.

References

[1] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 595–618. Springer, 2009.

[2] Sanjeev Arora and Rong Ge. New algorithms for learning in the presence of errors. In *ICALP*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.

[3] Joan Boyar and René Peralta. A depth-16 circuit for the AES S-box. Manuscript, `http://eprint.iacr.org/2011/332`, 2011.

[4] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. Manuscript, `http://eprint.iacr.org/2012/078`, 2012.

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS'12)*, 2012. Available at `http://eprint.iacr.org/2011/277`.

[6] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS'11*. IEEE Computer Society, 2011.

[7] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[8] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Advances in Cryptology - CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 487–504. Springer, 2011.

[9] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 446–464. Springer, 2012.

[10] Ivan Damgård and Marcel Keller. Secure multiparty aes. In *Proc. of Financial Cryptography 2010*, volume 6052 of *LNCS*, pages 367–374, 2010.

[11] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. Manuscript, 2011.

[12] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2008.

[13] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[14] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.

[15] Craig Gentry, Shai Halevi, and Nigel Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. Full version at `http://eprint.iacr.org/2011/566`.

[16] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013, Part I*, pages 75–92. Springer, 2013.

[17] Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Robustness of the learning with errors assumption. In *Innovations in Computer Science - ICS '10*, pages 230–240. Tsinghua University Press, 2010.

[18] Shai Halevi and Victor Shoup. Design and implementation of a homomorphic-encryption library. manuscript, available at `http://people.csail.mit.edu/shaih/pubs/he-library.pdf`, Accessed January 2015.

[19] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

[20] C. Orlandi J.B. Nielsen, P.S. Nordholt and S. Sheshank. A new approach to practical active-secure two-party computation. Manuscript, 2011.

[21] Kristin Lauter, Michael Naehrig, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW*, pages 113–124. ACM, 2011.

[22] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2011.

[23] Adriana Lòpez-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*. ACM, 2012.

[24] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, 2010.

[25] Daniele Micciancio and Oded Regev. *Lattice-based cryptography*, pages 147–192. Springer, 2009.

[26] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Steven C. Williams. Secure two-party computation is practical. In *Proc. ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267, 2009.

[27] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

[28] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography - PKC'10*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.

[29] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. Manuscript at http://eprint.iacr.org/2011/133, 2011.

A More Details

Following [24, 5, 15, 29] we utilize rings defined by cyclotomic polynomials, $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$. We let $\mathbb{A}_q$ denote the set of elements of this ring reduced modulo various (possibly composite) moduli q. The ring $\mathbb{A}$ is the ring of integers of a the mth cyclotomic number field K.

A.1 Plaintext Slots

In our scheme plaintexts will be elements of $\mathbb{A}_2$, and the polynomial $\Phi_m(X)$ factors modulo 2 into ℓ irreducible factors, $\Phi_m(X) = F_1(X) \cdot F_2(X) \cdots F_\ell(X) \pmod 2$, all of degree $d = \phi(m)/\ell$. Just as in [5, 15, 29] each factor corresponds to a "plaintext slot". That is, we view a polynomial $a \in \mathbb{A}_2$ as representing an ℓ-vector $(a \bmod F_i)_{i=1}^{\ell}$.

It is standard fact that the Galois group $\mathcal{G}\mathsf{al} = \mathcal{G}\mathsf{al}(\mathbb{Q}(\zeta_m)/\mathbb{Q})$ consists of the mappings $\kappa_k : a(X) \mapsto a(x^k) \bmod \Phi_m(X)$ for all k co-prime with m, and that it is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$. As noted in [15], for each $i, j \in \{1, 2, \ldots, \ell\}$ there is an element $\kappa_k \in \mathcal{G}\mathsf{al}$ which sends an element in slot i to an element in slot j. Namely, if $b = \kappa_i(a)$ then the element in the j'th slot of b is the same as that in the i'th slot of a. In addition $\mathcal{G}\mathsf{al}$ contains the Frobenius elements, $X \longrightarrow X^{2^i}$, which also act as Frobenius on the individual slots separately.

For the purpose of implementing AES we will be specifically interested in arithmetic in $\mathbb{F}_{2^8}$ (represented as $\mathbb{F}_{2^8} = \mathbb{F}_2[X]/G(X)$ with $G(X) = X^8 + X^4 + X^3 + X + 1$). We choose the parameters so that d is divisible by 8, so $\mathbb{F}_{2^d}$ includes $\mathbb{F}_{2^8}$ as a subfield. This lets us think of the plaintext space as containing ℓ-vectors over $\mathbb{F}_{2^n}$.

A.2 Canonical Embedding Norm

Following [24], we use as the "size" of a polynomial $a \in \mathbb{A}$ the l_∞ norm of its canonical embedding. Recall that the canonical embedding of $a \in \mathbb{A}$ into $\mathbb{C}^{\phi(m)}$ is the $\phi(m)$-vector of complex numbers $\sigma(a) = (a(\zeta_m^i))_i$ where ζ_m is a complex primitive m-th root of unity and the indexes i range over all of $(\mathbb{Z}/m\mathbb{Z})^*$. We call the norm of $\sigma(a)$ the *canonical embedding norm* of a, and denote it by

$$\|a\|_\infty^{\mathsf{can}} = \|\sigma(a)\|_\infty.$$

We will make use of the following properties of $\|\cdot\|_\infty^{\mathsf{can}}$:

- For all $a, b \in \mathbb{A}$ we have $\|a \cdot b\|_\infty^{\mathsf{can}} \le \|a\|_\infty^{\mathsf{can}} \cdot \|b\|_\infty^{\mathsf{can}}$.
- For all $a \in \mathbb{A}$ we have $\|a\|_\infty^{\mathsf{can}} \le \|a\|_1$.
- There is a ring constant c_m (depending only on m) such that $\|a\|_\infty \le c_m \cdot \|a\|_\infty^{\mathsf{can}}$ for all $a \in \mathbb{A}$.

The ring constant c_m is defined by $c_m = \|\mathsf{CRT}_m^{-1}\|_\infty$ where CRT_m is the CRT matrix for m, i.e. the Vandermonde matrix over the complex primitive m-th roots of unity. Asymptotically the value c_m can grow super-polynomially with m, but for the "small" values of m one would use in practice values of c_m can be evaluated directly. See [11] for a discussion of c_m.

Canonical Reduction. When working with elements in $\mathbb{A}_q$ for some integer modulus q, we sometimes need a version of the canonical embedding norm that plays nice with reduction modulo q. Following [15], we define the *canonical embedding norm reduced modulo* q of an element $a \in \mathbb{A}$ as the smallest canonical embedding norm of any a' which is congruent to a modulo q. We denote it as

$$|a|_q^{can} \stackrel{\text{def}}{=} \min\{\, \|a'\|_\infty^{\mathsf{can}} : a' \in \mathbb{A},\ a' \equiv a \pmod q \,\}.$$

We sometimes also denote the polynomial where the minimum is obtained by $[a]_q^{\mathsf{can}}$, and call it the *canonical reduction* of a modulo q. Neither the canonical embedding norm nor the canonical reduction is used in the scheme itself, it is only in the analysis of it that we will need them. We note that (trivially) we have $|a|_q^{\mathsf{can}} \le \|a\|_\infty^{\mathsf{can}}$.

A.3 Double CRT Representation

As noted in Section 2, we usually represent an element $a \in A_q$ via double-CRT representation, with respect to both the polynomial factor of $\Phi_m(X)$ and the integer factors of q. Specifically, we assume that $\mathbb{Z}/q\mathbb{Z}$ contains a primitive m-th root of unity (call it ζ), so $\Phi_m(X)$ factors modulo q to linear terms $\Phi_m(X) = \prod_{i\in(\mathbb{Z}/m\mathbb{Z})^*}(X - \zeta^j) \pmod q$. We also denote q's prime factorization by $q = \prod_{i=0}^{t} p_i$. Then a polynomial $a \in \mathbb{A}_q$ is represented as the $(t+1) \times \phi(m)$ matrix of its evaluation at the roots of $\Phi_m(X)$ modulo p_i for $i = 0, \ldots, t$:

$$\mathsf{dble\text{-}CRT}^t(a) = \left(a\left(\zeta^j\right) \bmod p_i\right)_{0\le i\le t, j\in(\mathbb{Z}/m\mathbb{Z})^*}.$$

The double CRT representation can be computed using $t+1$ invocations of the FFT algorithm modulo the p_i, picking only the FFT coefficients which correspond to elements in $(\mathbb{Z}/m\mathbb{Z})^*$. To invert this representation we invoke the inverse FFT algorithm $t+1$ times on a vector of length m consisting of the thinned out values padded with zeros, then apply the Chinese Remainder Theorem, and then reduce modulo $\Phi_m(X)$ and q.

Addition and multiplication in $\mathbb{A}_q$ can be computed as component-wise addition and multiplication of the entries in the two tables (modulo the appropriate primes p_i),

$$\begin{aligned}\text{dble-CRT}^t(a+b) &= \text{dble-CRT}^t(a) + \text{dble-CRT}^t(b)\\ \text{dble-CRT}^t(a\cdot b) &= \text{dble-CRT}^t(a)\cdot \text{dble-CRT}^t(b).\end{aligned}$$

Also, for an element of the Galois group $\kappa_k \in \mathcal{G}\mathsf{al}$ (which maps $a(X) \in \mathbb{A}$ to $a(X^k) \bmod \Phi_m(X)$), we can evaluate $\kappa_k(a)$ on the double-CRT representation of a just by permuting the columns in the matrix, sending each column j to column $j \cdot k \bmod m$.

A.4 Sampling From $\mathbb{A}_q$

At various points we will need to sample from $\mathbb{A}_q$ with different distributions, as described below. We denote choosing the element $a \in \mathbb{A}$ according to distribution $\mathcal{D}$ by $a \leftarrow \mathcal{D}$. The distributions below are described as over $\phi(m)$-vectors, but we always consider them as distributions over the ring $\mathbb{A}$, by identifying a polynomial $a \in \mathbb{A}$ with its coefficient vector.

The uniform distribution $\mathcal{U}_q$: This is just the uniform distribution over $(\mathbb{Z}/q\mathbb{Z})^{\phi(m)}$, which we identify with $(\mathbb{Z} \cap (-q/2, q/2])^{\phi(m)}$. Note that it is easy to sample from $\mathcal{U}_q$ directly in double-CRT representation.

The "discrete Gaussian" $\mathcal{DG}_q(\sigma^2)$: Let $\mathcal{N}(0,\sigma^2)$ denote the normal (Gaussian) distribution on real numbers with zero-mean and variance σ^2, we use drawing from $\mathcal{N}(0,\sigma^2)$ and rounding to the nearest integer as an approximation to the discrete Gaussian distribution. Namely, the distribution $\mathcal{DG}_{q_t}(\sigma^2)$ draws a real ϕ-vector according to $\mathcal{N}(0,\sigma^2)^{\phi(m)}$, rounds it to the nearest integer vector, and outputs that integer vector reduced modulo q (into the interval $(-q/2, q/2]$).

Sampling small polynomials, $\mathcal{ZO}(p)$ and $\mathcal{HWT}(h)$: These distributions produce vectors in $\{0,\pm 1\}^{\phi(m)}$.

For a real parameter $\rho \in [0,1]$, $\mathcal{ZO}(p)$ draws each entry in the vector from $\{0,\pm 1\}$, with probability $\rho/2$ for each of -1 and $+1$, and probability of being zero $1-\rho$.

For an integer parameter $h \le \phi(m)$, the distribution $\mathcal{HWT}(h)$ chooses a vector uniformly at random from $\{0,\pm 1\}^{\phi(m)}$, subject to the conditions that it has exactly h nonzero entries.

A.5 Canonical embedding norm of random polynomials

In the coming sections we will need to bound the canonical embedding norm of polynomials that are produced by the distributions above, as well as products of such polynomials. In some cases it is possible to analyze the norm rigorously using Chernoff and Hoeffding bounds, but to set the parameters of our scheme we instead use a heuristic approach that yields better constants:

Let $a \in \mathbb{A}$ be a polynomial that was chosen by one of the distributions above, hence all the (nonzero) coefficients in a are IID (independently identically distributed). For a complex primitive m-th root of unity ζ_m, the evaluation $a(\zeta_m)$ is the inner product between the coefficient vector of a and the fixed vector $\mathbf{z}_m = (1, \zeta_m, \zeta_m^2, \ldots)$, which has Euclidean norm exactly $\sqrt{\phi(m)}$. Hence the random variable $a(\zeta_m)$ has variance $V = \sigma^2\phi(m)$, where σ^2 is the variance of each coefficient of a. Specifically, when $a \leftarrow \mathcal{U}_q$ then each coefficient has variance $q^2/12$, so we get variance $V_U = q^2\phi(m)/12$. When $a \leftarrow \mathcal{DG}_q(\sigma^2)$ we get variance $V_G \approx \sigma^2\phi(m)$, and when $a \leftarrow \mathcal{ZO}(\rho)$ we get variance $V_Z = \rho\phi(m)$. When choosing $a \leftarrow \mathcal{HWT}(h)$ we get a variance of $V_H = h$ (but not $\phi(m)$, since a has only h nonzero coefficients).

Moreover, the random variable $a(\zeta_m)$ is a sum of many IID random variables, hence by the law of large numbers it is distributed similarly to a complex Gaussian random variable of the specified variance.[4] We therefore use $6\sqrt{V}$ (i.e. six standard deviations) as a high-probability bound on the size of $a(\zeta_m)$. Since the evaluation of a at all the roots of unity obeys the same bound, we use six standard deviations as our bound on the canonical embedding norm of a. (We chose six standard deviations since $\text{erfc}(6) \approx 2^{-55}$, which is good enough for us even when using the union bound and multiplying it by $\phi(m) \approx 2^{16}$.)

In many cases we need to bound the canonical embedding norm of a product of two such "random polynomials". In this case our task is to bound the magnitude of the product of two random variables, both are distributed close to Gaussians, with variances σ_a^2, σ_b^2, respectively. For this case we use $16\sigma_a\sigma_b$ as our bound, since $\text{erfc}(4) \approx 2^{-25}$, so the probability that both variables exceed their standard deviation by more than a factor of four is roughly 2^{-50}.

B The Basic Scheme

We now define our leveled HE scheme on L levels; including the Modulus-Switching and Key-Switching operations and the procedures for KeyGen, Enc, Dec, and for Add, Mult, Scalar-Mult, and Automorphism.

Recall that a ciphertext vector $\mathbf{c}$ in the cryptosystem is a valid encryption of $a \in \mathbb{A}$ with respect to secret key $\mathbf{s}$ and modulus q if $[[\langle \mathbf{c}, \mathbf{s} \rangle]_q]_2 = a$, where the inner product is over $\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$, the operation $[\cdot]_q$ denotes modular reduction in coefficient representation into the interval $(-q/2, +q/2]$, and we require that the "noise" $[\langle \mathbf{c}, \mathbf{s} \rangle]_q$ is sufficiently small (in canonical embedding norm reduced mod q). In our implementation a "normal" ciphertext is a 2-vector $\mathbf{c} = (c_0, c_1)$, and a "normal" secret key is of the form $\mathbf{s} = (1, -\mathfrak{s})$, hence decryption takes the form

$$a \leftarrow [c_0 - c_1 \cdot \mathfrak{s}]_q \bmod 2. \tag{2}$$

B.1 Our Moduli Chain

We define the chain of moduli for our depth-L homomorphic evaluation by choosing L "small primes" $p_0, p_1, \ldots, p_{L-1}$ and the t'th modulus in our chain is defined as $q_t = \prod_{j=0}^{t} p_j$. (The sizes will be determined later.) The primes p_i's are chosen so that for all i, $\mathbb{Z}/p_i\mathbb{Z}$ contains a primitive m-th root of unity. Hence we can use our double-CRT representation for all $\mathbb{A}_{q_t}$.

This choice of moduli makes it easy to get a level-$(t-1)$ representation of $a \in \mathbb{A}$ from its level-t representation. Specifically, given the level-t double-CRT representation $\text{dble-CRT}^t(a)$ for some $a \in \mathbb{A}_{q_t}$, we can simply remove from the matrix the row corresponding to the last small prime p_t, thus obtaining a level-$(t-1)$ representation of $a \bmod q_{t-1} \in \mathbb{A}_{q_{t-1}}$. Similarly we can get the double-CRT representation for lower levels by removing more rows. By a slight abuse of notation we write $\text{dble-CRT}^{t'}(a) = \text{dble-CRT}^t(a) \bmod q_{t'}$ for $t' < t$.

Recall that encryption produces ciphertext vectors valid with respect to the largest modulus q_{L-1} in our chain, and we obtain ciphertext vectors valid with respect to smaller moduli whenever we apply modulus-switching to decrease the noise magnitude. As described in Section 3.3, our implementation *dynamically* adjust levels, performing modulus switching when the dynamically-computed noise estimate becomes too large. Hence each ciphertext in our scheme is tagged with both its level t (pinpointing the modulus q_t relative to which this ciphertext is valid), and an estimate ν on the noise magnitude in this ciphertext. In other words,

[4] The mean of $a(\zeta_m)$ is zero, since the coefficients of a are chosen from a zero-mean distribution.

a ciphertext is a triple $(\mathbf{c}, t, \nu)$ with $0 \le t \le L-1$, $\mathbf{c}$ a vector over $\mathbb{A}_{q_t}$, and ν a real number which is used as our noise estimate.

B.2 Modulus Switching

The operation SwitchModulus($\mathbf{c}$) takes the ciphertext $\mathbf{c} = ((c_0, c_1), t, \nu)$ defined modulo q_t and produces a ciphertext $\mathbf{c}' = ((c'_0, c'_1), t-1, \nu')$ defined modulo q_{t-1}, Such that $[c_0 - \mathfrak{s} \cdot c_1]_{q_t} \equiv [c'_0 - \mathfrak{s} \cdot c'_1]_{q_{t-1}} \pmod 2$, and ν' is smaller than ν. This procedure makes use of the function Scale(x, q, q') that takes an element $x \in \mathbb{A}_q$ and returns an element $y \in \mathbb{A}_{q'}$ such that in coefficient representation it holds that $y \equiv x \pmod 2$, and y is the closest element to $(q'/q) \cdot x$ that satisfies this mod-2 condition.

To maintain the noise estimate, the procedure uses the pre-set ring-constant c_m (cf. Appendix A.2) and also a pre-set constant B_{scale} which is meant to bound the magnitude of the added noise term from this operation. It works as follows:

SwitchModulus$((c_0, c_1), t, \nu)$:

1. If $t < 1$ then abort; // Sanity check
2. $\nu' \leftarrow \frac{q_{t-1}}{q_t} \cdot \nu + B_{\mathsf{scale}}$; // Scale down the noise estimate
3. If $\nu' > q_{t-1}/2c_m$ then abort; // Another sanity check
4. $c'_i \leftarrow$ Scale(c_i, q_t, q_{t-1}) for $i = 0, 1$; // Scale down the vector
5. Output $((c'_0, c'_1), t-1, \nu')$.

The constant B_{scale} is set as $B_{\mathsf{scale}} = 2\sqrt{\phi(m)/3} \cdot (8\sqrt{h} + 3)$, where h is the Hamming weight of the secret key. (In our implementation we use $h = 64$, so we get $B_{\mathsf{scale}} \approx 77\sqrt{\phi(m)}$.) To justify this choice, we apply to the proof of the modulus switching lemma from [15, Lemma 13] (in the full version), relative to the canonical embedding norm. In that proof it is shown that when the noise magnitude in the input ciphertext $\mathbf{c} = (c_0, c_1)$ is bounded by ν, then the noise magnitude in the output vector $\mathbf{c}' = (c'_0, c'_1)$ is bounded by $\nu' = \frac{q_{t-1}}{q_t} \cdot \nu + \| \langle \mathbf{s}, \tau \rangle \|^{\mathsf{can}}_{\infty}$, provided that the last quantity is smaller than $q_{t-1}/2$.

Above τ is the "rounding error" vector, namely $\tau \stackrel{\text{def}}{=} (\tau_0, \tau_1) = (c'_0, c'_1) - \frac{q_{t-1}}{q_t}(c_0, c_1)$. Heuristically assuming that τ behaves as if its coefficients are chosen uniformly in $[-1, +1]$, the evaluation $\tau_i(\zeta)$ at an m-th root of unity ζ_m is distributed close to a Gaussian complex with variance $\phi(m)/3$. Also, $\mathfrak{s}$ was drawn from $\mathcal{HWT}(h)$ so $\mathfrak{s}(\zeta_m)$ is distributed close to a Gaussian complex with variance h. Hence we expect $\tau_1(\zeta)\mathfrak{s}(\zeta)$ to have magnitude at most $16\sqrt{\phi(m)/3 \cdot h}$ (recall that we use $h = 64$). We can similarly bound $\tau_0(\zeta_m)$ by $6\sqrt{\phi(m)/3}$, and therefore the evaluation of $\langle \mathbf{s}, \tau \rangle$ at ζ_m is bounded in magnitude (whp) by:

$$16\sqrt{\phi(m)/3 \cdot h} + 6\sqrt{\phi(m)/3} = 2\sqrt{\phi(m)/3} \cdot \left(8\sqrt{h} + 3\right) \approx 77\sqrt{\phi(m)} = B_{\mathsf{scale}} \tag{3}$$

B.3 Key Switching

After some homomorphic evaluation operations we have on our hands not a "normal" ciphertext which is valid relative to "normal" secret key, but rather an "extended ciphertext" $((d_0, d_1, d_2), q_t, \nu)$ which is valid with respect to an "extended secret key" $\mathbf{s}' = (1, -\mathfrak{s}, -\mathfrak{s}')$. Namely, this ciphertext encrypts the plaintext $a \in \mathbb{A}$ via

$$a = \left[\left[d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}' \cdot d_2\right]_{q_t}\right]_2$$

and the magnitude of the noise $\left[d_0 - \mathfrak{s} \cdot d_1 - d_2 \cdot \mathfrak{s}'\right]_{q_t}$ is bounded by ν. In our implementation, the component $\mathfrak{s}$ is always the same element $\mathfrak{s} \in \mathbb{A}$ that was drawn from $\mathcal{HWT}(h)$ during key generation, but $\mathfrak{s}'$ can vary depending on the operation. (See the description of multiplication and automorphisms below.)

To enable that translation, we use some "key switching matrices" that are included in the public key. (In our implementation these "matrices" have dimension 2×1, i.e., the consist of only two elements from $\mathbb{A}$.) As explained in Section 3.1, we save on space and time by artificially "boosting" the modulus we use from q_t up to $P \cdot q_t$ for some "large" modulus P. We note that in order to represent elements in $\mathbb{A}_{Pq_t}$ using our dble-CRT representation we need to choose P so that $\mathbb{Z}/P\mathbb{Z}$ also has primitive m-th roots of unity. (In fact in our implementation we pick P to be a prime.)

The key-switching "matrix". Denote by $Q = P \cdot q_{L-2}$ the largest modulus relative to which we need to generate key-switching matrices. To generate the key-switching matrix from $\mathbf{s}' = (1, -\mathfrak{s}, -\mathfrak{s}')$ to $\mathbf{s} = (1, -\mathfrak{s})$ (note that both keys share the same element $\mathfrak{s}$), we choose two element, one uniform and the other from our "discrete Gaussian",

$$a_{\mathfrak{s},\mathfrak{s}'} \leftarrow \mathcal{U}_Q \text{ and } e_{\mathfrak{s},\mathfrak{s}'} \leftarrow \mathcal{DG}_Q(\sigma^2),$$

where the variance σ is a global parameter (that we later set as $\sigma = 3.2$). The "key switching matrix" then consists of the single column vector

$$W[\mathfrak{s}' \to \mathfrak{s}] = \begin{pmatrix} b_{\mathfrak{s},\mathfrak{s}'} \\ a_{\mathfrak{s},\mathfrak{s}'} \end{pmatrix}, \text{ where } b_{\mathfrak{s},\mathfrak{s}'} \stackrel{\text{def}}{=} \left[\mathfrak{s} \cdot a_{\mathfrak{s},\mathfrak{s}'} + 2e_{\mathfrak{s},\mathfrak{s}'} + P\mathfrak{s}'\right]_Q. \tag{4}$$

Note that W above is defined modulo $Q = Pq_{L-2}$, but we need to use it relative to $Q_t = Pq_t$ for whatever the current level t is. Hence before applying the key switching procedure at level t, we reduce W modulo Q_t to get $W_t \stackrel{\text{def}}{=} [W]_{Q_t}$. It is important to note that since Q_t divides Q then W_t is indeed a key-switching matrix. Namely it is of the form $(b, a)^T$ with $a \in \mathcal{U}_{Q_t}$ and $b = [\mathfrak{s} \cdot a + 2e_{\mathfrak{s},\mathfrak{s}'} + P\mathfrak{s}']_{Q_t}$ (with respect to the same element $e_{\mathfrak{s},\mathfrak{s}'} \in \mathbb{A}$ from above).

The SwitchKey **procedure.** Given the extended ciphertext $\mathfrak{c} = ((d_0, d_1, d_2), t, \nu)$ and the key-switching matrix $W_t = (b, a)^T$, the procedure $\mathsf{SwitchKey}_{W_t}(\mathfrak{c})$ proceeds as follows:[5]

$\mathsf{SwitchKey}_{(b,a)}((d_0, d_1, d_2), t, \nu)$:

1. Set $\begin{pmatrix} c'_0 \\ c'_1 \end{pmatrix} \leftarrow \left[\begin{pmatrix} Pd_0 & b \\ Pd_1 & a \end{pmatrix}\begin{pmatrix} 1 \\ d_2 \end{pmatrix}\right]_{Q_t}$; // The actual key-switching operation
2. $c''_i \leftarrow \mathsf{Scale}(c'_i, Q_t, q_t)$ for $i = 0, 1$; // Scale the vector back down to q_t
3. $\nu' \leftarrow \nu + B_{\mathsf{Ks}} \cdot q_t/P + B_{\mathsf{scale}}$; // The constant B_{Ks} is determined below
4. Output $((c''_0, c''_1), t, \nu')$.

To argue correctness, observe that although the "actual key switching operation" from above looks superficially different from the standard key-switching operation $\mathbf{c}' \leftarrow W \cdot \mathbf{c}$, it is merely an optimization that takes advantage of the fact that both vectors $\mathbf{s}'$ and $\mathbf{s}$ share the element $\mathfrak{s}$. Indeed, we have the equality over $\mathbb{A}_{Q_t}$:

$$\begin{aligned} c'_0 - \mathfrak{s} \cdot c'_1 &= [(P \cdot d_0) + d_2 \cdot b_{\mathfrak{s},\mathfrak{s}'} - \mathfrak{s} \cdot \left((P \cdot d_1) + d_2 \cdot a_{\mathfrak{s},\mathfrak{s}'}\right) \\ &= P \cdot (d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}' d_2) + 2 \cdot d_2 \cdot \epsilon_{\mathfrak{s},\mathfrak{s}'}, \end{aligned}$$

[5] For simplicity we describe the SwitchKey procedure as if it always switches back to mod-q_t, but in reality if the noise estimate is large enough then it can switch directly to q_{t-1} instead.

so as long as both sides are smaller than Q_t we have the same equality also over $\mathbb{A}$ (without the mod-Q_t reduction), which means that we get

$$[c'_0 - \mathfrak{s} \cdot c'_1]_{Q_t} \;=\; [P \cdot (d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}' d_2) + 2 \cdot d_2 \cdot \epsilon_{\mathfrak{s},\mathfrak{s}'}]_{Q_t} \;\equiv\; [d_0 - \mathfrak{s} \cdot d_1 - \mathfrak{s}' d_2]_{Q_t} \pmod 2.$$

To analyze the size of the added term $2d_2\epsilon_{\mathfrak{s},\mathfrak{s}'}$, we can assume heuristically that d_2 behaves like a uniform polynomial drawn from $\mathcal{U}_{q_t}$, hence $d_2(\zeta_m)$ for a complex root of unity ζ_m is distributed close to a complex Gaussian with variance $q_t^2\phi(m)/12$. Similarly $\epsilon_{\mathfrak{s},\mathfrak{s}'}(\zeta_m)$ is distributed close to a complex Gaussian with variance $\sigma^2\phi(m)$, so $2d_2(\zeta)\epsilon(\zeta)$ can be modeled as a product of two Gaussians, and we expect that with overwhelming probability it remains smaller than $2 \cdot 16 \cdot \sqrt{q_t^2\phi(m)/12 \cdot \sigma^2\phi(m)} \;=\; \frac{16}{\sqrt{3}} \cdot \sigma q_t \phi(m)$. This yields a heuristic bound $16/\sqrt{3} \cdot \sigma\phi(m) \cdot q_t \;=\; B_{\mathsf{Ks}} \cdot q_t$ on the canonical embedding norm of the added noise term, and if the total noise magnitude does not exceed $Q_t/2c_m$ then also in coefficient representation everything remains below $Q_t/2$. Thus our constant B_{Ks} is set as

$$\frac{16\sigma\phi(m)}{\sqrt{3}} \;\approx\; 9\sigma\phi(m) \;=\; B_{\mathsf{Ks}} \tag{5}$$

Finally, dividing by P (which is the effect of the Scale operation), we obtain the final ciphertext that we require, and the noise magnitude is divided by P (except for the added B_{scale} term).

B.4 Key-Generation, Encryption, and Decryption

The procedures below depend on many parameters, h, σ, m, the primes p_i and P, etc. These parameters will be determined later.

- $\mathsf{KeyGen}()$: Given the parameters, the key generation procedure chooses a low-weight secret key and then generates an LWE instance relative to that secret key. Namely, we choose

$$\mathfrak{s} \leftarrow \mathcal{HWT}(h), \quad a \leftarrow \mathcal{U}_{q_{L-1}}, \quad \text{and } e \leftarrow \mathcal{DG}_{q_{L-1}}(\sigma^2)$$

Then sets the secret key as $\mathfrak{s}$ and the public key as (a, b) where $b = [a \cdot s + 2e]_{q_{L-1}}$.

In addition, the key generation procedure adds to the public key some key-switching "matrices", as described in Appendix B.3. Specifically the matrix $W[\mathfrak{s}^2 \to \mathfrak{s}]$ for use in multiplication, and some matrices $W[\kappa_i(\mathfrak{s}) \to \mathfrak{s}]$ for use in automorphisms, for $\kappa_i \in \mathcal{G}\mathsf{al}$ whose indexes generates $(\mathbb{Z}/m\mathbb{Z})^*$ (including in particular κ_2).

- $\mathsf{Enc}_{\mathfrak{pt}}(\mathbf{m})$: To encrypt an element $m \in \mathbb{A}_2$, we choose one "small polynomial" (with $0, \pm 1$ coefficients) and two Gaussian polynomials (with variance σ^2),

$$v \leftarrow \mathcal{ZO}(0.5) \quad \text{and} \quad e_0, e_1 \leftarrow \mathcal{DG}_{q_{L-1}}(\sigma^2)$$

Then we set $c_0 = b \cdot v + 2 \cdot e_0 + m$, $c_1 = a \cdot v + 2 \cdot e_1$, and set the initial ciphertext as $\mathfrak{c}' = (c_0, c_1, L-1, B_{\mathsf{clean}})$, where B_{clean} is a parameter that we determine below.

The noise magnitude in this ciphertext (B_{clean}) is a little larger than what we would like, so before we start computing on it we do one modulus-switch. That is, the encryption procedure sets $\mathfrak{c} \leftarrow \mathsf{SwitchModulus}(\mathfrak{c}')$ and outputs $\mathfrak{c}$. We can deduce a value for B_{clean} as follows:

$$\begin{aligned}
\left|c_0 - \mathfrak{s} \cdot c_1\right|_{q_t}^{\mathsf{can}} &\le \|c_0 - \mathfrak{s} \cdot c_1\|_\infty^{\mathsf{can}} \\
&= \|((a \cdot s + 2 \cdot e) \cdot v + 2 \cdot e_0 + \mathbf{m} - (a \cdot v + 2 \cdot e_1) \cdot \mathfrak{s}\|_\infty^{\mathsf{can}} \\
&= \|\mathbf{m} + 2 \cdot (e \cdot v + e_0 - e_1 \cdot \mathfrak{s})\|_\infty^{\mathsf{can}} \\
&\le \|\mathbf{m}\|_\infty^{\mathsf{can}} + 2 \cdot (\|e \cdot v\|_\infty^{\mathsf{can}} + \|e_0\|_\infty^{\mathsf{can}} + \|e_1 \cdot \mathfrak{s}\|_\infty^{\mathsf{can}})
\end{aligned}$$

Using our complex Gaussian heuristic from Appendix A.5, we can bound the canonical embedding norm of the randomized terms above by

$$\|e \cdot v\|_\infty^{\mathsf{can}} \le 16\sigma\phi(m)/\sqrt{2}, \quad \|e_0\|_\infty^{\mathsf{can}} \le 6\sigma\sqrt{\phi(m)}, \quad \|e_1 \cdot \mathfrak{s}\|_\infty^{\mathsf{can}} \le 16\sigma\sqrt{h \cdot \phi(m)}$$

Also, the norm of the input message m is clearly bounded by $\phi(m)$, hence (when we substitute our parameters $h = 64$ and $\sigma = 3.2$) we get the bound

$$\phi(m) + 32\sigma\phi(m)/\sqrt{2} + 12\sigma\sqrt{\phi(m)} + 32\sigma\sqrt{h \cdot \phi(m)} \approx 74\phi(m) + 858\sqrt{\phi(m)} = B_{\mathsf{clean}} \quad (6)$$

Our goal in the initial modulus switching from q_{L-1} to q_{L-2} is to reduce the noise from its initial level of $B_{\mathsf{clean}} = \Theta(\phi(m))$ to our base-line bound of $B = \Theta(\sqrt{\phi(m)})$ which is determined in Equation (12) below.

$\mathsf{Dec}_{\mathfrak{sk}}(\mathfrak{c})$: Decryption of a ciphertext (c_0, c_1, t, ν) at level t is performed by setting $m' \leftarrow [c_0 - \mathfrak{s} \cdot c_1]_{q_t}$, then converting m' to coefficient representation and outputting $m' \bmod 2$. This procedure works when $c_m \cdot \nu < q_t/2$, so this procedure only applies when the constant c_m for the field $\mathbb{A}$ is known and relatively small (which as we mentioned above will be true for all practical parameters). Also, we must pick the smallest prime $q_0 = p_0$ large enough, as described in Appendix C.2.

B.5 Homomorphic Operations

$\mathsf{Add}(\mathfrak{c}, \mathfrak{c}')$: Given two ciphertexts $\mathfrak{c} = ((c_0, c_1), t, \nu)$ and $\mathfrak{c}' = ((c'_0, c'_1), t', \nu')$, representing messages $\mathbf{m}, \mathbf{m}' \in \mathbb{A}_2$, this algorithm forms a ciphertext $\mathfrak{c}_a = ((a_0, a_1), t_a, \nu_a)$ which encrypts the message $\mathbf{m}_a = \mathbf{m} + \mathbf{m}'$.

If the two ciphertexts do not belong to the same level then we reduce the larger one modulo the smaller of the two moduli, thus bringing them to the same level. (This simple modular reduction works as long as the noise magnitude is smaller than the smaller of the two moduli, if this condition does not hold then we need to do modulus switching rather than simple modular reduction.) Once the two ciphertexts are at the same level (call it t''), we just add the two ciphertext vectors and two noise estimates to get

$$\mathfrak{c}_a = \left(([c_0 + c'_0]_{q_{t''}}, [c_1 + c'_1]_{q_{t''}}),\ t'', \nu + \nu'\right).$$

$\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')$: Given two ciphertexts representing messages $\mathbf{m}, \mathbf{m}' \in \mathbb{A}_2$, this algorithm forms a ciphertext encrypts the message $\mathbf{m} \cdot \mathbf{m}'$.

We begin by ensuring that the noise magnitude in both ciphertexts is smaller than the pre-set constant B (which is our base-line bound and is determined inEquation (12) below), performing modulus-switching as needed to ensure this condition. Then we bring both ciphertexts to the same level by reducing modulo the smaller of the two moduli (if needed). Once both ciphertexts have small noise magnitude and the same level we form the extended ciphertext (essentially performing the tensor product of the two) and apply key-switching to get back a normal ciphertext. A pseudo-code description of this procedure is given below.

$\mathsf{Mult}(\mathfrak{c}, \mathfrak{c}')$:

1. While $\nu(\mathfrak{c}) > B$ do $\mathfrak{c} \leftarrow \mathsf{SwitchModulus}(\mathfrak{c})$; // $\nu(\mathfrak{c})$ is the noise estimate in $\mathfrak{c}$
2. While $\nu(\mathfrak{c}') > B$ do $\mathfrak{c}' \leftarrow \mathsf{SwitchModulus}(\mathfrak{c}')$; // $\nu(\mathfrak{c}')$ is the noise estimate in $\mathfrak{c}'$
3. Bring $\mathfrak{c}, \mathfrak{c}'$ to the same level t by reducing modulo the smaller of the two moduli
   Denote after modular reduction $\mathfrak{c} = ((c_0, c_1), t, \nu)$ and $\mathfrak{c}' = ((c'_0, c'_1), t, \nu')$

4. Set $(d_0, d_1, d_2) \leftarrow (c_0 \cdot c'_0\ ,\quad c_1 \cdot c'_0 + c_0 \cdot c'_1\ ,\quad -c_1 \cdot c'_1)$;
   Denote $\mathfrak{c}'' = ((d_0, d_1, d_2), t, \nu \cdot \nu')$
5. Output $\mathsf{SwitchKey}_{W[\mathfrak{s}^2 \to \mathfrak{s}]}(\mathfrak{c}'')$ // Convert to "normal" ciphertext

We stress that *the only place* where we force modulus switching is before the multiplication operation. In all other operations we allow the noise to grow, and it will be reduced back the first time it is input to a multiplication operation. We also note that we may need to apply modulus switching more than once before the noise is small enough.

$\mathsf{Scalar\text{-}Mult}(\mathfrak{c}, \alpha)$: Given a ciphertext $\mathfrak{c} = (c_0, c_1, t, \nu)$ representing the message $\mathbf{m}$, and an element $\alpha \in \mathbb{A}_2$ (represented as a polynomial modulo 2 with coefficients in $\{-1, 0, 1\}$), this algorithm forms a ciphertext $\mathfrak{c}_m = (a_0, a_1, t_m, \nu_m)$ which encrypts the message $\mathbf{m}_m = \alpha \cdot \mathbf{m}$. This procedure is needed in our implementation of homomorphic AES, and is of more general interest in general computation over finite fields.

The algorithm makes use of a procedure $\mathsf{Randomize}(\alpha)$ which takes α and replaces each non-zero coefficients with a coefficients chosen at random from $\{-1, 1\}$. To multiply by α, we set $\beta \leftarrow \mathsf{Randomize}(\alpha)$ and then just multiply both c_0 and c_1 by β. Using the same argument as we used in Appendix A.5 for the distribution $\mathcal{HWT}(h)$, here too we can bound the norm of β by $\|\beta\|_\infty^{\mathsf{can}} \le 6\sqrt{\mathsf{Wt}(\alpha)}$ where $\mathsf{Wt}(\alpha)$ is the number of nonzero coefficients of α. Hence we multiply the noise estimate by $6\sqrt{\mathsf{Wt}(\alpha)}$, and output the resulting ciphertext $\mathfrak{c}_m = (c_0 \cdot \beta,\ c_1 \cdot \beta,\ t,\ \nu \cdot 6\sqrt{\mathsf{Wt}(\alpha)})$.

$\mathsf{Automorphism}(\mathfrak{c}, \kappa)$: In the main body we explained how permutations on the plaintext slots can be realized via using elements $\kappa \in \mathcal{G}\mathsf{al}$; we also require the application of such automorphism to implement the Frobenius maps in our AES implementation.

For each κ that we want to use, we need to include in the public key the "matrix" $W[\kappa(\mathfrak{s}) \to \mathfrak{s}]$. Then, given a ciphertext $\mathfrak{c} = (c_0, c_1, t, \nu)$ representing the message $\mathbf{m}$, the function $\mathsf{Automorphism}(\mathfrak{c}, \kappa)$ produces a ciphertext $\mathfrak{c}' = (c'_0, c'_1, t, \nu')$ which represents the message $\kappa(\mathbf{m})$. We first set an "extended ciphertext" by setting

$$d_0 = \kappa(c_0), \quad d_1 \leftarrow 0, \quad \text{and } d_2 \leftarrow \kappa(c_1)$$

and then apply key switching to the extended ciphertext $((d_0, d_1, d_2), t, \nu)$ using the "matrix" $W[\kappa(\mathfrak{s}) \to \mathfrak{s}]$.

C Security Analysis and Parameter Settings

Below we derive the concrete parameters for use in our early implementation. This part of the report is outdated, we left it here for historical purpose.

We begin in Appendix C.1 by deriving a lower-bound on the dimension N of the LWE problem underlying our key-switching matrices, as a function of the modulus and the noise variance. (This will serve as a lower-bound on $\phi(m)$ for our choice of the ring polynomial $\Phi_m(X)$.) Then in Appendix C.2 we derive a lower bound on the size of the largest modulus Q in our implementation, in terms of the noise variance and the dimension N. Then in Appendix C.3 we choose a value for the noise variance (as small as possible subject to some nominal security concerns), solve the somewhat circular constraints on N and Q, and set all the other parameters.

C.1 Lower-Bounding the Dimension

Below we apply to the LWE-security analysis of Lindner and Peikert [22], together with a few (arguably justifiable) assumptions, to analyze the dimension needed for different security levels. The analysis below

assumes that we are given the modulus Q and noise variance σ^2 for the LWE problem (i.e., the noise is chosen from a discrete Gaussian distribution modulo Q with variance σ^2 in each coordinate). The goal is to derive a lower-bound on the dimension N required to get any given security level. The first assumption that we make, of course, is that the Lindner-Peikert analysis — which was done in the context of standard LWE — applies also for our ring-LWE case. We also make the following extra assumptions:

- We assume that (once σ is not too tiny), the security depends on the ratio Q/σ and not on Q and σ separately. Nearly all the attacks and hardness results in the literature support this assumption, with the exception of the Arora-Ge attack [2] (that works whenever σ is very small, regardless of Q).

- The analysis in [22] devised an experimental formula for the time that it takes to get a particular quality of reduced basis (i.e., the parameter δ of Gama and Nguyen [12]), then provided another formula for the advantage that the attack can derive from a reduced basis at a given quality, and finally used a computer program to solve these formulas for some given values of N and δ. This provides some time/advantage tradeoff, since obtaining a smaller value of δ (i.e., higher-quality basis) takes longer time and provides better advantage for the attacker.

  For our purposes we made the assumption that the best runtime/advantage ratio is achieved in the high-advantage regime. Namely we should spend basically all the attack running time doing lattice reduction, in order to get a good enough basis that will break security with advantage (say) 1/2. This assumption is consistent with the results that are reported in [22].

- Finally, we assume that to get advantage of close to 1/2 for an LWE instance with modulus Q and noise σ, we need to be able to reduce the basis well enough until the shortest vector is of size roughly Q/σ. Again, this is consistent with the results that are reported in [22].

Given these assumptions and the formulas from [22], we can now solve the dimension/security tradeoff analytically. Because of the first assumption we might as well simplify the equations and derive our lower bound on N for the case $\sigma = 1$, where the ratio Q/σ is equal to Q. (In reality we will use $\sigma \approx 4$ and increase the modulus by the same 2 bits).

Following Gama-Nguyen [12], recall that a reduced basis $B = (b_1|b_2|\ldots|b_m)$ for a dimension-M, determinant-D lattice (with $\|b_1\| \leq \|b_2\| \leq \cdots \|b_M\|$), has quality parameter δ if the shortest vector in that basis has norm $\|b_1\| = \delta^M \cdot D^{1/M}$. In other words, the quality of B is defined as $\delta = \|b_1\|^{1/M}/D^{1/M^2}$. The time (in seconds) that it takes to compute a reduced basis of quality δ for a random LWE instance was estimated in [22] to be at least

$$\log(\text{time}) \;\geq\; 1.8/\log(\delta) - 110. \tag{7}$$

For a random Q-ary lattice of rank N, the determinant is exactly Q^N whp, and therefore a quality-δ basis has $\|b_1\| = \delta^M \cdot Q^{N/M}$. By our second assumption, we should reduce the basis enough so that $\|b_1\| = Q$, so we need $Q = \delta^M \cdot Q^{N/M}$. The LWE attacker gets to choose the dimension M, and the best choice for this attack is obtained when the right-hand-side of the last equality is minimized, namely for $M = \sqrt{N \log Q/\log \delta}$. This yields the condition

$$\log Q = \log(\delta^M Q^{N/M}) = M \log \delta + (N/M) \log Q = 2\sqrt{N \log Q \log \delta},$$

which we can solve for N to get $N = \log Q/4 \log \delta$. Finally, we can use Equation (7) to express $\log \delta$ as a function of $\log(\text{time})$, thus getting $N = \log Q \cdot (\log(\text{time}) + 110)/7.2$. Recalling that in our case we used

$\sigma = 1$ (so $Q/\sigma = Q$), we get our lower-bound on N in terms of Q/σ. Namely, to ensure a time/advantage ratio of at least 2^k, we need to set the rank N to be at least

$$N \ \geq \ \frac{\log(Q/\sigma)(k+110)}{7.2} \tag{8}$$

For example, the above formula says that to get 80-bit security level we need to set $N \geq \log(Q/\sigma) \cdot 26.4$, for 100-bit security level we need $N \geq \log(Q/\sigma) \cdot 29.1$, and for 128-bit security level we need $N \geq \log(Q/\sigma) \cdot 33.1$. We comment that these values are indeed consistent with the values reported in [22].

C.1.1 LWE with Sparse Key

The analysis above applies to "generic" LWE instance, but in our case we use very sparse secret keys (with only $h = 64$ nonzero coefficients, all chosen as ± 1). This brings up the question of whether one can get better attacks against LWE instances with a very sparse secret (much smaller than even the noise). We note that Goldwasser et al. proved in [17] that LWE with low-entropy secret is as hard as standard LWE with weaker parameters (for large enough moduli). Although the specific parameters from that proof do not apply to our choice of parameter, it does indicate that weak-secret LWE is not "fundamentally weaker" than standard LWE. In terms of attacks, the only attack that we could find that takes advantage of this sparse key is by applying the reduction technique of Applebaum et al. [1] to switch the key with part of the error vector, thus getting a smaller LWE error.

In a sparse-secret LWE we are given a random N-by-M matrix A (modulo Q), and also an M-vector $\mathbf{y} = [\mathbf{s}A + \mathbf{e}]_Q$. Here the N-vector $\mathbf{s}$ is our very sparse secret, and $\mathbf{e}$ is the error M-vector (which is also short, but not sparse and not as short as $\mathbf{s}$).

Below let A_1 denotes the first N columns of A, A_2 the next N columns, then A_3, A_4, etc. Similarly $\mathbf{e}_1, \mathbf{e}_2, \ldots$ are the corresponding parts of the error vector and $\mathbf{y}_1, \mathbf{y}_2, \ldots$ the corresponding parts of $\mathbf{y}$. Assuming that A_1 is invertible (which happens with high probability), we can transform this into an LWE instance with respect to secret $\mathbf{e}_1$, as follows:

We have $\mathbf{y}_1 = \mathbf{s}A_1 + \mathbf{e}_1$, or alternatively $A_1^{-1}\mathbf{y}_1 = \mathbf{s} + A_1^{-1}\mathbf{e}_1$. Also, for $i > 1$ we have $\mathbf{y}_i = \mathbf{s}A_i + \mathbf{e}_i$, which together with the above gives $A_i A_1^{-1}\mathbf{y}_1 - \mathbf{y}_i = A_i A_1^{-1}\mathbf{e}_1 - \mathbf{e}_i$. Hence if we denote

$$B_1 \stackrel{\text{def}}{=} A_1^{-1}, \quad \text{and for } i > 1 \ B_i \stackrel{\text{def}}{=} A_i A 1^{-1},$$
$$\text{and similarly } \mathbf{z}_1 = A_1^{-1}\mathbf{y}_1, \quad \text{and for } i > 1 \ \ \mathbf{z}_i \stackrel{\text{def}}{=} A_i A_1^{-1}\mathbf{y}_i,$$

and then set $B \stackrel{\text{def}}{=} (B_1^t | B_2^t | B_3^t | \ldots)$ and $\mathbf{z} \stackrel{\text{def}}{=} (\mathbf{z}_1 | \mathbf{z}_2 | \mathbf{z}_3 | \ldots)$, and also $\mathbf{f} = (\mathbf{s} | \mathbf{e}_2 | \mathbf{e}_3 | \ldots)$ then we get the LWE instance

$$\mathbf{z} = \mathbf{e}_1^t B + \mathbf{f}$$

with secret $\mathbf{e}_1^t$. The thing that makes this LWE instance potentially easier than the original one is that the first part of the error vector f is our sparse/small vector s, so the transformed instance has smaller error than the original (which means that it is easier to solve).

Trying to quantify the effect of this attack, we note that the optimal M value in the attack from Appendix C.1 above is obtained at $M = 2N$, which means that the new error vector is $\mathbf{f} = (\mathbf{s}|\mathbf{e}_2)$, which has Euclidean norm smaller than $\mathbf{e} = (\mathbf{e}_1|\mathbf{e}_2)$ by roughly a factor of $\sqrt{2}$ (assuming that $\|\mathbf{s}\| \ll \|\mathbf{e}_1\| \approx \|\mathbf{e}_2\|$). Maybe some further improvement can be obtained by using a smaller value for M, where the shorter error may outweigh the "non optimal" value of M. However, we do not expect to get major improvement this way, so it seems that the very sparse secret should only add maybe one bit to the modulus/noise ratio.

C.2 The Modulus Size

In this section we assume that we are given the parameter $N = \phi(m)$ (for our polynomial ring modulo $\Phi_m(X)$). We also assume that we are given the noise variance σ^2, the number of levels in the modulus chain L, an additional "slackness parameter" ξ (whose purpose is explained below), and the number of nonzero coefficients in the secret key h. Our goal is to devise a lower bound on the size of the largest modulus Q used in the public key, so as to maintain the functionality of the scheme.

Controlling the Noise. Driving the analysis in this section is a bound on the noise magnitude right after modulus switching, which we denote below by B. We set our parameters so that starting from ciphertexts with noise magnitude B, we can perform one level of fan-in-two multiplications, then one level of fan-in-ξ additions, followed by key switching and modulus switching again, and get the noise magnitude back to the same B.

- Recall that in the "reduced canonical embedding norm", the noise magnitude is at most multiplied by modular multiplication and added by modular addition, hence after the multiplication and addition levels the noise magnitude grows from B to as much as ξB^2.
- As we've seen in Appendix B.3, performing key switching scales up the noise magnitude by a factor of P and adds another noise term of magnitude upto $B_{\mathsf{Ks}} \cdot q_t$ (before doing modulus switching to scale it back down). Hence starting from noise magnitude ξB^2, the noise grows to magnitude $P\xi B^2 + B_{\mathsf{Ks}} \cdot q_t$ (relative to the modulus Pq_t).

  Below we assume that after key-switching we do modulus switching directly to a smaller modulus.
- After key-switching we can switch to the next modulus q_{t-1} to decrease the noise back to our bound B. Following the analysis from Appendix B.2, switching moduli from Q_t to q_{t-1} decreases the noise magnitude by a factor of $q_{t-1}/Q_t = 1/(P \cdot p_t)$, and then add a noise term of magnitude B_{scale}.

  Starting from noise magnitude $P\xi B^2 + B_{\mathsf{Ks}} \cdot q_t$ before modulus switching, the noise magnitude after modulus switching is therefore bounded whp by

$$\frac{P \cdot \xi B^2 + B_{\mathsf{Ks}} \cdot q_t}{P \cdot p_t} + B_{\mathsf{scale}} \;=\; \frac{\xi B^2}{p_t} + \frac{B_{\mathsf{Ks}} \cdot q_{t-1}}{P} + B_{\mathsf{scale}}$$

Using the analysis above, our goal next is to set the parameters B, P and the p_t's (as functions of N, σ, L, ξ and h) so that in every level t we get $\frac{\xi B^2}{p_t} + \frac{B_{\mathsf{Ks}} \cdot q_{t-1}}{P} + B_{\mathsf{scale}} \le B$. Namely we need to satisfy at every level t the quadratic inequality (in B)

$$\frac{\xi}{p_t} B^2 \;-\; B \;+\; \Bigg(\underbrace{\frac{B_{\mathsf{Ks}} \cdot q_{t-1}}{P} + B_{\mathsf{scale}}}_{\text{denote this by } R_{t-1}} \Bigg) \;\le\; 0\,. \tag{9}$$

Observe that (assuming that all the primes p_t are roughly the same size), it suffices to satisfy this inequality for the largest modulus $t = L-2$, since R_{t-1} increases with larger t's. Noting that $R_{L-3} > B_{\mathsf{scale}}$, we want to get this term to be as close to B_{scale} as possible, which we can do by setting P large enough. Specifically, to make it as close as $R_{L-3} = (1 + 2^{-n})B_{\mathsf{scale}}$ it is sufficient to set

$$P \;\approx\; 2^n \frac{B_{\mathsf{Ks}} q_{L-3}}{B_{\mathsf{scale}}} \;\approx\; 2^n \frac{9\sigma N q_{L-3}}{77\sqrt{N}} \;\approx\; 2^{n-3} q_{L-3} \cdot \sigma\sqrt{N}, \tag{10}$$

Below we set (say) $n = 8$, which makes it close enough to use just $R_{L-3} \approx B_{\mathsf{scale}}$ for the derivation below.

Clearly to satisfy Inequality (9) we must have a positive discriminant, which means $1-4\frac{\xi}{p_{L-2}}R_{L-3} \geq 0$, or $p_{L-2} \geq 4\xi R_{L-3}$. Using the value $R_{L-3} \approx B_{\mathsf{scale}}$, this translates into setting

$$p_1 \approx p_2 \cdots \approx p_{L-2} \approx 4\xi \cdot B_{\mathsf{scale}} \approx 308\xi\sqrt{N} \tag{11}$$

Finally, with the discriminant positive and all the p_i's roughly the same size we can satisfy Inequality (9) by setting

$$B \approx \frac{1}{2\xi/p_{L-2}} = \frac{p_{L-2}}{2\xi} \;\approx\; 2B_{\mathsf{scale}} \approx 154\sqrt{N}. \tag{12}$$

The Smallest Modulus. After evaluating our L-level circuit, we arrive at the last modulus $q_0 = p_0$ with noise bounded by ξB^2. To be able to decrypt, we need this noise to be smaller than $q_0/2c_m$, where c_m is the ring constant for our polynomial ring modulo $\Phi_m(X)$. For our setting, that constant is always below 40, so a sufficient condition for being able to decrypt is to set

$$q_0 = p_0 \;\approx\; 80\xi B^2 \;\approx\; 2^{20.9}\xi N \tag{13}$$

The Encryption Modulus. Recall that freshly encrypted ciphertext have noise B_{clean} (as defined in Equation (6)), which is larger than our baseline bound B from above. To reduce the noise magnitude after the first modulus switching down to B, we therefore set the ratio $p_{L-1} = q_{L-1}/q_{L-2}$ so that $B_{\mathsf{clean}}/p_{L-1} + B_{\mathsf{scale}} \leq B$. This means that we set

$$p_{L-1} \;=\; \frac{B_{\mathsf{clean}}}{B - B_{\mathsf{scale}}} \approx \frac{74N + 858\sqrt{N}}{77\sqrt{N}} \approx \sqrt{N} + 11 \tag{14}$$

The Largest Modulus. Having set all the parameters, we are now ready to calculate the resulting bound on the largest modulus, namely $Q_{L-2} = q_{L-2} \cdot P$. Using Equations (11), and (13), we get

$$q_t \;=\; p_0 \cdot \prod_{i=1}^{t} p_i \;\approx\; (2^{20.9}\xi N) \cdot \left(308\xi\sqrt{N}\right)^t \;=\; 2^{20.9} \cdot 308^t \cdot \xi^{t+1} \cdot N^{t/2+1}. \tag{15}$$

Now using Equation (10) we have

$$\begin{aligned} P \;&\approx\; 2^5 q_{L-3}\sigma\sqrt{N} \;\approx\; 2^{25.9} \cdot 308^{L-3} \cdot \xi^{L-2} \cdot N^{(L-3)/2+1} \cdot \sigma\sqrt{N} \\ &\approx\; 2 \cdot 308^L \cdot \xi^{L-2}\sigma N^{L/2} \end{aligned}$$

and finally

$$\begin{aligned} Q_{L-2} \;=\; P \cdot q_{L-2} \;&\approx\; (2 \cdot 308^L \cdot \xi^{L-2}\sigma N^{L/2}) \cdot (2^{20.9} \cdot 308^{L-2} \cdot \xi^{L-1} \cdot N^{L/2}) \\ &\approx\; \sigma \cdot 2^{16.5L+5.4} \cdot \xi^{2L-3} \cdot N^L \end{aligned} \tag{16}$$

C.3 Putting It Together

We now have in Equation (8) a lower bound on N in terms of Q, σ and the security level k, and in Equation (16) a lower bound on Q with respect to N, σ and several other parameters. We note that σ is a free parameter, since it drops out when substituting Equation (16) in Equation (8). In our implementation we used $\sigma = 3.2$, which is the smallest value consistent with the analysis in [25].

For the other parameters, we set $\xi = 8$ (to get a small "wiggle room" without increasing the parameters much), and set the number of nonzero coefficients in the secret key at $h = 64$ (which is already included in the formulas from above, and should easily defeat exhaustive-search/birthday type of attacks). Substituting these values into the equations above we get

$$p_0 \approx 2^{23.9} N, \quad p_i \approx 2^{11.3}\sqrt{N} \text{ for } i = 1, \ldots, L-2$$
$$P \approx 2^{11.3L-5} N^{L/2}, \text{ and } \quad Q_{L-2} \approx 2^{22.5L-3.6} \sigma N^L.$$

Substituting the last value of Q_{L-2} into Equation (8) yields

$$N > \frac{(L(\log N + 23) - 8.5)(k + 110)}{7.2} \tag{17}$$

Targeting $k = 80$-bits of security and solving for several different depth parameters L, we get the results in the table below, which also lists approximate sizes for the primes p_i and P.

L	N	$\log_2(p_0)$	$\log_2(p_i)$	$\log_2(p_{L-1})$	$\log_2(P)$
10	9326	37.1	17.9	7.5	177.3
20	19434	38.1	18.4	8.1	368.8
30	29749	38.7	18.7	8.4	564.2
40	40199	39.2	18.9	8.6	762.2
50	50748	39.5	19.1	8.7	962.1
60	61376	39.8	19.2	8.9	1163.5
70	72071	40.0	19.3	9.0	1366.1
80	82823	40.2	19.4	9.1	1569.8
90	93623	40.4	19.5	9.2	1774.5

Choosing Concrete Values. Having obtained lower-bounds on $N = \phi(m)$ and other parameters, we now need to fix precise cyclotomic fields $\mathbb{Q}(\zeta_m)$ to support the algebraic operations we need. We have two situations we will be interested in for our experiments. The first corresponds to performing arithmetic on bytes in $\mathbb{F}_{2^8}$ (i.e. $n = 8$), whereas the latter corresponds to arithmetic on bits in $\mathbb{F}_2$ (i.e. $n = 1$). We therefore need to find an odd value of m, with $\phi(m) \approx N$ and m dividing $2^d - 1$, where we require that d is divisible by n. Values of m with a small number of prime factors are preferred as they give rise to smaller values of c_m. We also look for parameters which maximize the number of slots ℓ we can deal with in one go, and values for which $\phi(m)$ is close to the approximate value for N estimated above. When $n = 1$ we always select a set of parameters for which the ℓ value is at least as large as that obtained when $n = 8$.

	$n = 8$				$n = 1$			
L	m	$N = \phi(m)$	(d, ℓ)	c_K	m	$N = \phi(m)$	(d, ℓ)	c_K
10	11441	10752	(48,224)	3.60	11023	10800	(45,240)	5.13
20	34323	21504	(48,448)	6.93	34323	21504	(48,448)	6.93
30	31609	31104	(72,432)	5.15	32377	32376	(57,568)	1.27
40	54485	40960	(64,640)	12.40	42799	42336	(21,2016)	5.95
50	59527	51840	(72,720)	21.12	54161	52800	(60,880)	4.59
60	68561	62208	(72,864)	36.34	85865	63360	(60,1056)	12.61
70	82603	75264	(56,1344)	36.48	82603	75264	(56,1344)	36.48
80	92837	84672	(56,1512)	38.52	101437	85672	(42,2016)	19.13
90	124645	98304	(48,2048)	21.07	95281	94500	(45,2100)	6.22

D $\mathsf{Scale}(c, q_t, q_{t-1})$ in dble-CRT Representation

Let $q_i = \prod_{j=0}^{i} p_j$, where the p_j's are primes that split completely in our cyclotomic field $\mathbb{A}$. We are given a $c \in \mathbb{A}_{q_t}$ represented via double-CRT – that is, it is represented as a "matrix" of its evaluations at the primitive m-th roots of unity modulo the primes $p_0, \ldots, p_t$. We want to modulus switch to q_{t-1} – i.e., scale down by a factor of p_t. Let's recall what this means: we want to output $c' \in \mathbb{A}$, represented via double-CRT format (as its matrix of evaluations modulo the primes $p_0, \ldots, p_{t-1}$), such that

1. $c' = c \bmod 2$.
2. c' is very close (in terms of its coefficient vector) to c/p_t.

In the main body we explained how this could be performed in dble-CRT representation. This made explicit use of the fact that the two ciphertexts need to be equivalent modulo two. If we wished to replace two with a general prime p, then things are a bit more complicated. For completeness, although it is not required in our scheme, we present a methodology below. In this case, the conditions on $c^\dagger$ are as follows:

1. $c^\dagger = c \cdot p_t \bmod p$.
2. $c^\dagger$ is very close to c.
3. $c^\dagger$ is divisible by p_t.

As before, we set $c' \leftarrow c^\dagger / p_t$. (Note that for $p = 2$, we trivially have $c \cdot p_t = c \bmod p$, since p_t will be odd.)

This causes some complications, because we set $c^\dagger \leftarrow c + \delta$, where $\delta = -\bar{c} \bmod p_t$ (as before) but now $\delta = (p_t - 1) \cdot c \bmod p$. To compute such a δ, we need to know $c \bmod p$. Unfortunately, we don't have $c \bmod p$. One not-very-satisfying way of dealing with this problem is the following. Set $\hat{c} \leftarrow [p_t]_p \cdot c \bmod q_t$. Now, if c encrypted m, then $\hat{c}$ encrypts $[p_t]_p \cdot m$, and $\hat{c}$'s noise is $[p_t]_p < p/2$ times as large. It is obviously easy to compute $\hat{c}$'s double-CRT format from c's. Now, we set $c^\dagger$ so that the following is true:

1. $c^\dagger = \hat{c} \bmod p$.
2. $c^\dagger$ is very close to $\hat{c}$.
3. $c^\dagger$ is divisible by p_t.

This is easy to do. The algorithm to output $c^\dagger$ in double-CRT format is as follows:

1. Set $\bar{c}$ to be the coefficient representation of $\hat{c} \bmod p_t$. (Computing this requires a single "small FFT" modulo the prime p_t.)
2. Set δ to be the polynomial with coefficients in $(-p_t \cdot p/2, p_t \cdot p/2]$ such that $\delta = 0 \bmod p$ and $\delta = -\bar{c} \bmod p_t$.
3. Set $c^\dagger = \hat{c} + \delta$, and output $c^\dagger$'s double-CRT representation.
   (a) We already have $\hat{c}$'s double-CRT representation.
   (b) Computing δ's double-CRT representation requires t "small FFTs" modulo the p_j's.

E Other Optimizations

Some other optimizations that we encountered during our implementation work are discussed next. Not all of these optimizations are useful for our current implementation, but they may be useful in other contexts.

Three-way Multiplications. Sometime we need to multiply several ciphertexts together, and if their number is not a power of two then we do not have a complete binary tree of multiplications, which means that at some point in the process we will have three ciphertexts that we need to multiply together.

The standard way of implementing this 3-way multiplication is via two 2-argument multiplications, e.g., $x \cdot (y \cdot z)$. But it turns out that here it is better to use "raw multiplication" to multiply these three ciphertexts (as done in [7]), thus getting an "extended" ciphertext with four elements, then apply key-switching (and later modulus switching) to this ciphertext. This takes only six ring-multiplication operations (as opposed to eight according to the standard approach), three modulus switching (as opposed to four), and only one key switching (applied to this 4-element ciphertext) rather than two (which are applied to 3-element extended ciphertexts). All in all, this three-way multiplication takes roughly 1.5 times a standard two-element multiplication.

We stress that this technique is not useful for larger products, since for more than three multiplicands the noise begins to grow too large. But with only three multiplicands we get noise of roughly B^3 after the multiplication, which can be reduced to noise $\approx B$ by dropping two levels, and this is also what we get by using two standard two-element multiplications.

Commuting Automorphisms and Multiplications. Recalling that the automorphisms $X \mapsto X^i$ commute with the arithmetic operations, we note that some ordering of these operations can sometimes be better than others. For example, it may be better perform the multiplication-by-constant before the automorphism operation whenever possible. The reason is that if we perform the multiply-by-constant after the key-switching that follows the automorphism, then added noise term due to that key-switching is multiplied by the same constant, thereby making the noise slightly larger. We note that to move the multiplication-by-constant before the automorphism, we need to multiply by a different constant.

Switching to higher-level moduli. We note that it may be better to perform automorphisms at a higher level, in order to make the added noise term due to key-switching small with respect to the modulus. On the other hand operations at high levels are more expensive than the same operations at a lower level. A good rule of thumb is to perform the automorphism operations one level above the lowest one. Namely, if the naive strategy that never switches to higher-level moduli would perform some Frobenius operation at level q_i, then we perform the key-switching following this Frobenius operation at level Q_{i+1}, and then switch back to level q_{i+1} (rather then using Q_i and q_i).

Commuting Addition and Modulus-switching. When we need to add many terms that were obtained from earlier operations (and their subsequent key-switching), it may be better to first add all of these terms relative to the large modulus Q_i before switching the sum down to the smaller q_i (as opposed to switching all the terms individually to q_i and then adding).

Reducing the number of key-switching matrices. When using many different automorphisms $\kappa_i : X \mapsto X^i$ we need to keep many different key-switching matrices in the public key, one for every value of i that

we use. We can reduces this memory requirement, at the expense of taking longer to perform the automorphisms. We use the fact that the Galois group $\mathcal{G}\mathsf{al}$ that contains all the maps κ_i (which is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$) is generated by a relatively small number of generators. (Specifically, for our choice of parameters the group $(\mathbb{Z}/m\mathbb{Z})^*$ has two or three generators.) It is therefore enough to store in the public key only the key-switching matrices corresponding to κ_{g_j}'s for these generators g_j of the group $\mathcal{G}\mathsf{al}$. Then in order to apply a map κ_i we express it as a product of the generators and apply these generators to get the effect of κ_i. (For example, if $i = g_1^2 \cdot g_2$ then we need to apply κ_{g_1} twice followed by a single application of κ_{g_2}.)