Fully Homomorphic Encryption with Polylog Overhead

C. Gentry[1], S. Halevi[1], and N.P. Smart[2]

[1] IBM T.J. Watson Research Center,
Yorktown Heights, New York, U.S.A.
[2] Dept. Computer Science, University of Bristol,
Bristol, United Kingdom.

Abstract. We show that homomorphic evaluation of (wide enough) arithmetic circuits can be accomplished with only polylogarithmic overhead. Namely, we present a construction of fully homomorphic encryption (FHE) schemes that for security parameter λ can evaluate any width-$\Omega(\lambda)$ circuit with t gates in time $t \cdot \text{polylog}(\lambda)$.
To get low overhead, we use the recent batch homomorphic evaluation techniques of Smart-Vercauteren and Brakerski-Gentry-Vaikuntanathan, who showed that homomorphic operations can be applied to "packed" ciphertexts that encrypt vectors of plaintext elements. In this work, we introduce permuting/routing techniques to move plaintext elements across these vectors efficiently. Hence, we are able to implement general arithmetic circuit in a batched fashion without ever needing to "unpack" the plaintext vectors.
We also introduce some other optimizations that can speed up homomorphic evaluation in certain cases. For example, we show how to use the Frobenius map to raise plaintext elements to powers of p at the "cost" of a linear operation.

Keywords. Homomorphic encryption, Bootstrapping, Batching, Automorphism, Galois group, Permutation network.

Acknowledgments. The first and second authors are sponsored by DARPA and ONR under agreement number N00014-11C-0390. The U.S. Government is authorized to reproduce and distribute reprints 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).

The third author is 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.

Table of Contents

1 Introduction

Fully homomorphic encryption (FHE) [16, 9, 8] allows a worker to perform arbitrarily-complex dynamically-chosen computations on encrypted data, despite not having the secret decryption key. Processing encrypted data homomorphically requires more computation than processing the data unencrypted. But how much more? What is the *overhead*, the ratio of encrypted computation complexity to unencrypted computation complexity (using a circuit model of computation)? Here, under the ring-LWE assumption, we show that the overhead can be made as low as *polylogarithmic* in the security parameter.

We accomplish this by *packing* many plaintexts into each ciphertext; each ciphertext has $\tilde{\Omega}(\lambda)$ "plaintext slots". Then, we describe a complete set of operations – Add, Mult and Permute – that allows us to evaluate arbitrary circuits *while keeping the ciphertexts packed*. Batch Add and Mult have been done before [18], and follow easily from the Chinese Remainder Theorem within our underlying polynomial ring. Here we introduce the operation Permute, that allows us to homomorphically move data between the plaintext slots, show how to realize it from our underlying algebra, and how to use it to evaluate arbitrary circuits.

Our approach begins with the observation [3, 18] that we can use an automorphism group $\mathcal{H}$ associated to our underlying ring to "rotate" or "re-align" the contents of the plaintext slots. (These automorphisms were used in a somewhat similar manner by Lyubashevsky et al. [15] in their proof of the pseudorandomness of RLWE.) While $\mathcal{H}$ alone enables only a few permutations (e.g., "rotations"), we show that any permutation can be constructed as a log-depth permutation network, where each level consists of a constant number of "rotations", batch-additions and batch-multiplications. Our method works when the underlying ring has an associated automorphism group $\mathcal{H}$ which is abelian and sharply transitive, a condition that we prove always holds for our scheme's parameters.

Ultimately, the Add, Mult and Permute operations can all be accomplished with $\tilde{O}(\lambda)$ computation by building on the recent Brakerski-Gentry-Vaikuntanathan (BGV) "FHE without bootstrapping" scheme [3], which builds on prior work by Brakerski and Vaikuntanathan and others [5, 4, 12]. Thus, we obtain an FHE scheme that can evaluate any circuit that has $\Omega(\lambda)$ average width with only polylog(λ) overhead. For comparison, the smallest overhead for FHE was $\tilde{O}(\lambda^{3.5})$ [19] until BGV recently reduced it to $\tilde{O}(\lambda)$ [3].[3]

In addition to their essential role in letting us move data across plaintext slots, ring automorphisms turn out to have interesting secondary consequences: they also enable more nimble manipulation of data *within* plaintext slots. Specifically, in some cases we can use them to raise the packed plaintext elements to a high power with hardly any increase in the noise magnitude of the ciphertext! In practice, this could permit evaluation of high-degree circuits without resorting to bootstrapping, in applications such as computing AES. See Appendix A.3.

1.1 Packing Plaintexts and Batched Homomorphic Computation

Smart and Vercauteren [17, 18] were the first to observe that, by an application the Chinese Remainder Theorem to number fields, the plaintext space of some previous FHE schemes can be partitioned into a vector of "plaintext slots", and that a single homomorphic Add or Mult of a pair of ciphertexts implicitly adds or multiplies (component-wise) the entire plaintext vectors. Each plaintext slot is defined to hold an element in some finite field $\mathbb{K}_n = \mathbb{F}_{p^n}$, and, abstractly, if one has two ciphertexts that hold (encrypt) messages $m_0, \ldots, m_{\ell-1} \in \mathbb{K}_n^\ell$ and $m'_0, \ldots, m'_{\ell-1} \in \mathbb{K}_n^\ell$ respectively in plaintext slots $0, \ldots, \ell - 1$, applying ℓ-Add to the two ciphertexts gives a new ciphertext that holds $m_0 + m'_0, \ldots, m_{\ell-1} + m'_{\ell-1}$ and applying ℓ-Mult gives a new ciphertext that holds $m_0 \cdot m'_0, \ldots, m_{\ell-1} \cdot m'_{\ell-1}$. Smart and Vercauteren used this observation for *batch* (or SIMD [11]) homomorphic

[3] However, the polylog factors in our new scheme are rather large. It remains to be seen how much of an improvement this approach yields in practice, as compared to the $\tilde{O}(\lambda^{3.5})$ approach implemented in [10, 19].

operations. That is, they show how to evaluate a function f homomorphically ℓ times in parallel on ℓ different inputs, with approximately the same cost that it takes to evaluate the function once without batching.

Here is a taste of how these separate plaintext slots are constructed algebraically. As an example, for the ring-LWE-based scheme, suppose we use the polynomial ring $\mathbb{A} = \mathbb{Z}[x]/(x^\ell + 1)$ where ℓ is a power of 2. Ciphertexts are elements of $\mathbb{A}_q^2$ where (as in in [3]) q has only polylog(λ) bits. The "aggregate" plaintext space is $\mathbb{A}_p$ (that is, ring elements taken modulo p) for some small prime $p = 1 \bmod 2\ell$. Any prime $p = 1 \bmod 2\ell$ *splits* over the field associated to this ring – that is, in $\mathbb{A}$, the ideal generated by p is the product of ℓ ideals $\{\mathfrak{p}_i\}$ each of norm p – and therefore $\mathbb{A}_p \equiv \mathbb{A}_{\mathfrak{p}_0} \times \cdots \times \mathbb{A}_{\mathfrak{p}_{\ell-1}}$. Consequently, using the Chinese remainder theorem, we can encode ℓ independent mod-p plaintexts $m_0, \ldots, m_{\ell-1} \in \{0, \ldots, p-1\}$ as the unique element in $\mathbb{A}_p$ that is in all of the cosets $m_i + \mathfrak{p}_i$. Thus, in a single ciphertext, we may have ℓ independent plaintext "slots".

In this work, we often use ℓ-Add and ℓ-Mult to efficiently implement a Select operation: Given an index set I we can construct a vector $\mathbf{v}_I$ of "select bits" $(v_0, \ldots, v_{\ell-1})$, such that $v_i = 1$ if $i \in I$ and $v_i = 0$ otherwise. Then element-wise multiplication of a packed ciphertext $\mathbf{c}$ with the select vector $\mathbf{v}$ results in a new ciphertext that contains only the plaintext element in the slots corresponding to I, and zero elsewhere. Moreover, by generating two complementing select vectors $\mathbf{v}_I$ and $\mathbf{v}_{\bar{I}}$ we can mix-and-match the slots from two packed ciphertexts $\mathbf{c}_1$ and $\mathbf{c}_2$: Setting $\mathbf{c} = (\mathbf{v}_I \times \mathbf{c}_1) + (\mathbf{v}_{\bar{I}} \times \mathbf{c}_2)$, we pack into $\mathbf{c}$ the slots from $\mathbf{c}_1$ at indexes from I and the slots from $\mathbf{c}_2$ elsewhere.

While batching is useful in many setting, it does not, by itself, yield low-overhead homomorphic computation in general, as it does not help us to reduce the overhead of computing a complicated function just once. Just as in normal program execution of SIMD instructions (e.g., the SSE instructions on x86), one needs a method of moving data between slots in each SIMD word.

1.2 Permuting Plaintexts Within the Plaintext Slots

To reduce the overhead of homomorphic computation *in general*, we need a *complete* set of operations over *packed vectors of plaintexts*. The approach above allows us to add or multiply messages that are in the same plaintext slot, but what if we want to add the content of the i-th slot in one ciphertext to the content of the j-th slot of another ciphertext, for $i \neq j$? We can "unpack" the slots into separate ciphertexts (say, using homomorphic decryption[4] [8, 9]), but there is little hope that this approach could yield very efficient FHE. Instead, we complement ℓ-Add and ℓ-Mult with an operation ℓ-Permute to move data efficiently across slots within a a given ciphertext, and efficient procedures to clone slots from a packed ciphertext and move them around to other packed ciphertexts.

Brakerski, Gentry, and Vaikuntanathan [3] observed that for certain parameter settings, one can use *automorphisms* associated with the algebraic ring $\mathbb{A}$ to "rotate" all of plaintext spaces simultaneously, sort of like turning a dial on a safe. That is, one can transform a ciphertext that holds $m_0, m_1, \ldots, m_{\ell-1}$ in its ℓ slots into another ciphertext that holds $m_i, m_{i+1}, \ldots, m_{i+\ell-1}$ (for an arbitrary given i, index arithmetic mod ℓ), and this rotation operation takes time quasi-linear in the ciphertext size, which is quasi-linear in the security parameter. They used this tool to construct Pack and Unpack algorithms whereby separate ciphertexts could be aggregated (packed) into a single ciphertext with packed plaintexts before applying bootstrapping (and then the refreshed ciphertext would be unpacked), thereby lowering the amortized cost of bootstrapping.

We exploit these automorphisms more fully, using the basic rotations that the automorphisms give us to construct *permutation networks* that can permute data in the plaintext slots arbitrarily. We also extend the application of the automorphisms to more general underlying rings, beyond the specific parameter settings considered in prior work [5, 4, 3]. This lets us devise low-overhead homomorphic schemes for arithmetic circuits over essentially any small finite field $\mathbb{F}_{p^n}$.

[4] This is the approach suggested in [18] for Gentry's original FHE scheme.

Our efficient implementation of $\mathsf{Permute}$, described in Section 3, uses the Beneš/Waksman permutation network [2, 20]. This network consists of two back-to-back butterfly network of width 2^k, where each level in the network has 2^{k-1} "switch gates" and each switch gate swaps (or not) its two inputs, depending on a control bit. It is possible to realize any permutation of $\ell = 2^k$ items by appropriately setting the control bits of all the switch gates. Viewing this network as acting on k-bit addresses, the i-th level of the network partitions the 2^k addresses into 2^{k-1} pairs, where each pair of addresses differs only in the $|i-k|$-th bit, and then it swaps (or not) those pairs. The fact that the pairs in the i-th level always consist of addresses that differ by exactly $2^{|i-k|}$, makes it easy to implement each level using rotations: All we need is one rotation by $2^{|i-k|}$ and another by $-2^{|i-k|}$, followed by two batched Select operations.

For general rings $\mathbb{A}$, the automorphisms do not always exactly "rotate" the plaintext slots. Instead, they act on the slots in a way that depends on a quotient group $\mathcal{H}$ of the appropriate Galois group. Nonetheless, we use basic theorems from Galois theory, in conjunction with appropriate generalizations of the Beneš/Waksman procedure, to construct a permutation network of depth $O(\log \ell)$ that can realize any permutation over the ℓ plaintext slots, where each level of the network consists of a constant number of permutations from $\mathcal{H}$ and Select operations. As with the rotations considered in [3], applying permutations from $\mathcal{H}$ can be done in time quasi-linear in ciphertext size, which is only quasi-linear in the security parameter. Overall, we find that permutation networks and Galois theory are a surprisingly fruitful combination.

We note that Damgård, Ishai and Krøigaard [7] used permutation networks in a somewhat analogous fashion to perform secure multiparty computation with *packed secret shares*. In their setting, which permits interaction between the parties, the permutations can be evaluated using much simpler mathematical machinery.

1.3 FHE with Polylog Overhead

In our discussion above, we glossed over the fact that ciphertext sizes in a BGV-like cryptosystem [3] depend polynomially on the depth of the circuit being evaluated, because the modulus size must grow with the depth of the circuit (unless bootstrapping [8, 9] is used). So, without bootstrapping, the "polylog overhead" result only applies to circuits of polylog depth. However, decryption itself can be accomplished in log-depth [3], and moreover the parameters can be set so that a ciphertext with $\tilde{\Omega}(\lambda)$ slots can be decrypted using a circuit of size $\tilde{O}(\lambda)$. Therefore, "recryption" can be accomplished with polylog overhead, and we obtain FHE with polylog overhead for arbitrary (wide enough) circuits.

2 Computing on (Encrypted) Arrays

As we explained above, our main tool for low-overhead homomorphic computation is to compute on "packed ciphertexts", namely make each ciphertext hold a vector of plaintext values rather than a single value. Throughout this section we let ℓ be a parameter specifying the number of plaintext values that are packed inside each ciphertext, namely we always work with ℓ-vectors of plaintext values. Let $\mathbb{K}_n = \mathbb{F}_{p^n}$ denote the plaintext space (e.g., $\mathbb{K}_n = \mathbb{F}_2$ if we are dealing with binary circuits directly). It was shown in [3, 18] how to homomorphically evaluate batch addition and multiplication operations on ℓ-vectors:

$$\ell\text{-}\mathsf{Add}\big(\langle u_0,\dots,u_{\ell-1}\rangle, \langle v_0,\dots,v_{\ell-1}\rangle\big) \overset{\text{def}}{=} \langle u_0+v_0,\dots,u_{\ell-1}+v_{\ell-1}\rangle$$
$$\ell\text{-}\mathsf{Mult}\big(\langle u_0,\dots,u_{\ell-1}\rangle, \langle v_0,\dots,v_{\ell-1}\rangle\big) \overset{\text{def}}{=} \langle u_0\times v_0,\dots,u_{\ell-1}\times v_{\ell-1}\rangle$$

on packed ciphertexts in time $\tilde{O}((\ell+\lambda)(\log|\mathbb{K}_n|)$ where λ is the security parameter (with addition and multiplication in $\mathbb{K}_n$).[5] Specifically, if the size of our plaintext space is polynomially bounded and we set $\ell = \Theta(\lambda)$, then we can evaluate the above operations homomorphically in time $\tilde{O}(\lambda)$.

Unfortunately, component-wise ℓ-Add and ℓ-Mult are not sufficient to perform arbitrary computations on encrypted arrays, since data at different indexes within the arrays can never interact. To get a *complete set of operations for arrays*, we introduce the ℓ-$\mathsf{Permute}$ operation that can arbitrarily permute the data within the ℓ-element arrays. Namely, for any permutation π over the indexes $I_\ell = \{0, 1, \ldots, \ell-1\}$, we want to homomorphically evaluate the function

$$\ell\text{-}\mathsf{Permute}_\pi\big(\langle u_0, \ldots, u_{\ell-1}\rangle\big) = \big\langle u_{\pi(0)}, \ldots, u_{\pi(\ell-1)}\big\rangle .$$

on a packed ciphertext, with complexity similar to the above. We will show how to implement ℓ-$\mathsf{Permute}$ homomorphically in Sections 3 and 4 below. For now, we just assume that such an implementation is available and show how to use it to obtain low-overhead implementation of general circuits.

2.1 Computing with ℓ-Fold Gates

We are interested in computing arbitrary functions using "ℓ-fold gates" that operate on ℓ-element arrays as above. We assume that the function $f(\cdot)$ to be computed is specified using a fan-in-2 arithmetic circuit with t "normal" arithmetic gates (that operate on singletons). Our goal is to implement f using as few ℓ-fold gates as possible, hopefully not much more than t/ℓ of them.

We assume that the input to f is presented in a packed form, namely when computing an r-variate function $f(x_1, \ldots, x_r)$ we get as input $\lceil r/\ell \rceil$ arrays (indexed $A_0, \ldots, A_{\lceil r/\ell \rceil}$) with the j'th array containing the input elements $x_{j\ell}$ through $x_{j\ell+\ell-1}$. The last array may contain less than ℓ elements, and the unused entries contain "don't care" elements. In fact, throughout the computation we allow all of the arrays to contain "don't care" entries. We say that an array is *sparse* if it contains $\ell/2$ or more "don't care" entries. We maintain the invariant that our collection of arrays is always at least half full, i.e., we hold r values using at most $\lceil 2r/\ell \rceil$ ℓ-element arrays.

The gates that we use in the computation are the ℓ-Add, ℓ-Mult, and ℓ-$\mathsf{Permute}$ gates from above. The rest of this section is devoted to establishing the following theorem:

Theorem 1. *Let ℓ, t, w and W be parameters. Then any t-gate fan-in-2 arithmetic circuit C with average width w and maximum width W, can be evaluated using a network of $O\big(\lceil t/\ell \rceil \cdot \lceil \ell/w \rceil \cdot \log W \cdot \mathrm{polylog}(\ell)\big)$ ℓ-fold gates of types ℓ-Add, ℓ-Mult, and ℓ-$\mathsf{Permute}$. The depth of this network of ℓ-fold gates is at most $O(\log W)$ times that of the original circuit C, and the description of the network can be computed in time $\tilde{O}(t)$ given the description of C.*

Before turning to proving Theorem 1, we point out that Theorem 1 implies that if the original circuit C has size $t = \mathrm{poly}(\lambda)$, depth L, and average width $w = \Omega(\lambda)$, and if we set the packing parameter as $\ell = \Theta(\lambda)$, then we get an $O(L \cdot \log \lambda)$-depth implementation of C using $O(t/\lambda \cdot \mathrm{polylog}(\lambda))$ ℓ-fold gates. If implementing each ℓ-fold gate takes $\tilde{O}(L\lambda)$ time, then the total time to evaluate C is no more than

$$O\big(\frac{t}{\lambda}\mathrm{polylog}(\lambda) \cdot L \cdot \lambda \cdot \mathrm{polylog}(\lambda)\big) = O(t \cdot L \cdot \mathrm{polylog}(\lambda)).$$

Therefore, with this choice of parameter (and for "wide enough" circuits of average width $\Omega(\lambda)$), our overhead for evaluating depth-L circuits is only $O(L \cdot \mathrm{polylog}(\lambda))$. And if L is also polylogarithmic, as in BGV with bootstrapping [3], then the total overhead is polylogarithmic in the security parameter.

[5] To compute L levels of such operations, the complexity expression becomes $\tilde{O}((\ell+\lambda)(L + \log|\mathbb{K}_n|))$.

The high-level idea of the proof of Theorem 1 is what one would expect. Consider an arbitrary fan-in two arithmetic circuit C. Suppose that we have $\approx w$ output wire values of level $i-1$ packed into roughly w/ℓ arrays. We need to route these output values to their correct input positions at level i. It should be obvious that the ℓ-Permute gates facilitate this routing, except for two complications:

1. The mapping from outputs of level $i-1$ to inputs of level i is not a permutation. Specifically, level-$(i-1)$ gates may have high fan-out, and so some of the output values may need to be *cloned*.
2. Once the output values are cloned sufficiently (for a total of, say, w' values), routing to level i apparently calls for a *big permutation* over w' elements, not just a small permutation within arrays of ℓ elements.

Below we show that these complications can be handled efficiently.

2.2 Permutations over Hyper-Rectangles

First, consider the second complication from above – namely, that we need to perform a permutation over some w elements (possibly $w \gg \ell$) using ℓ-Add, ℓ-Mult, and ℓ-Permute operations that only work on ℓ-element arrays. We use the following basic fact (cf. [14]), for completeness we provide a proof in Appendix B.

Lemma 1. *Let $S = \{0,\dots,a-1\} \times \{0,\dots,b-1\}$ be a set of ab positions, arranged as a matrix of a rows and b columns. For any permutation π over S, there are permutations π_1, π_2, π_3 such that $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ (that is, π is the composition of the three permutations) and such that π_1 and π_3 only permute positions within each column (these permutations only change the row, not the column, of each element) and π_2 only permutes positions within each row. Moreover, there is a polynomial-time algorithm that given π outputs the decomposition permutations π_1, π_2, π_3.*

In our context, Lemma 1 says that if we have w elements packed into $k = \lceil w/\ell \rceil$ ℓ-element arrays, we can express any permutation π of these elements as $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ where π_2 invokes ℓ-Permute (k times in parallel) to permute data within the respective arrays, and π_1, π_3 only permute (ℓ times in parallel) elements that share the same index within their respective arrays. In Section 2.3, we describe how to implement π_1, π_3 using ℓ-Add and ℓ-Mult, and analyze the overall efficiency of implementing π. The following generalization of Lemma 1 to higher dimensions will be used later in this work. It is proved by invoking Lemma 1 recursively.

Lemma 2. *Let $S = I_{n_1} \times \cdots \times I_{n_k}$ where $I_{n_i} = \{0,\dots,n_i - 1\}$. (Each element in S has k coordinates.) For any permutation π over S, there are permutations $\pi_1,\dots,\pi_{2k-1}$ such that $\pi = \pi_{2k-1} \circ \cdots \circ \pi_1$ and such that π_i affects only the i-th coordinate for $i \le k$ and only the $(2k-i)$-th coordinate for $i \ge k$.*

2.3 Batch Selections, Swaps, and Permutation Networks

We now describe how to use ℓ-Add and ℓ-Mult to realize the outer permutations π_1, π_3, which permute (ℓ times in parallel) elements that share the same index within their respective arrays. To perform these permutations, we can apply a *permutation network* à la Beneš/Waksman [2, 20]. Recall that a r-dimensional Beneš network consists of two back-to-back butterfly networks. Namely it is a $(2r-1)$-level network with 2^r nodes in each level, where for $i = 1, 2, \dots, 2r-1$, we have an edge connecting node j in level $i-1$ to node j' in level i if the indexes j, j' are either equal (a "straight edge") or they differ in only in the $|r-i|$'th bit (a "cross edge"). The following lemma is an easy corollary of Lemma 2.

Lemma 3. **[13, Thm 3.11]** *Given any one-to-one mapping π of 2^r inputs to 2^r outputs in an r-dimensional Beneš network (one input per level-0 node and one output per level-$(2r-1)$ node), there is a set of node-disjoint paths from the inputs to the outputs connecting input i to output $\pi(i)$ for all i.*

In our setting, to implement our π_1 and π_3 from Lemma 1 we need to evaluate ℓ of these permutation networks in parallel, one for each index in our ℓ-fold arrays. Assume for simplicity that the number of ℓ-fold arrays is a power of two, say 2^r, and denote these arrays by $A_0, \ldots, A_{2^r-1}$, we would have a $(2r-1)$-level network, where the i'th level in the network consists of operating on pairs of arrays $(A_j, A_{j'})$, such that the indexes j, j' differ only in the $|r-i|$'th bit.

The operation applied to two such arrays $A_j, A_{j'}$ works separately on the different indexes of these arrays. For each $k = 0, 1, \ldots, \ell-1$ the operation will either swap $A_j[k] \leftrightarrow A_{j'}[k]$ or will leave these two entries unchanged, depending on whether the paths in the k'th permutation network uses the cross edges or the straight edges between nodes j and j' in levels $i-1, i$ of the permutation network.

Thus, evaluating ℓ such permutation networks in parallel reduces to the following Select function: Given two arrays $A = [m_0, \ldots, m_{\ell-1}]$ and $A' = [m'_0, \ldots, m'_{\ell-1}]$ and a string $S = s_0 \cdots s_{\ell-1} \in \{0,1\}^\ell$, the operation $\mathsf{Select}_S(A, A')$ outputs an array $A'' = [m''_0, \ldots, m''_{\ell-1}]$ where, for each k, $m''_k = m_k$ if $s_k = 1$ and $m''_k = m'_k$ otherwise. It is easy to implement $\mathsf{Select}_S(A, A')$ using just the ℓ-Add and ℓ-Mult operations – in particular

$$\mathsf{Select}_S(A, A') = \ell\text{-}\mathsf{Add}\left(\ \ell\text{-}\mathsf{Mult}(A, S),\ \ell\text{-}\mathsf{Mult}(A', \bar{S})\ \right)$$

where $\bar{S}$ is the bitwise complement of S. Note that $\mathsf{Select}_{\bar{S}}(A, A')$ outputs precisely the elements that are discarded by $\mathsf{Select}_S(A, A')$. So, $\mathsf{Select}_S(A, A')$ and $\mathsf{Select}_{\bar{S}}(A, A')$ are exactly like the arrays A' and A', except that some pairs of elements with identical indexes have been *swapped* – namely, those pairs at index k where $S_k = 0$. Hence we obtain the following, again the proof is deferred to Appendix B.

Lemma 4. *Evaluating ℓ permutation networks in parallel, each permuting k items, can be accomplished using $O(k \cdot \log k)$ gates of ℓ-Add and ℓ-Mult, and depth $O(\log k)$. Also, evaluating a permutation π over $k \cdot \ell$ elements that are packed into k ℓ-element arrays, can be accomplished using k ℓ-$\mathsf{Permute}$ gates and $O(k \log k)$ gates of ℓ-Add and ℓ-Mult, in depth $O(\log k)$. Moreover, there is an efficient algorithm that given π computes the circuit of ℓ-$\mathsf{Permute}$, ℓ-Add, and ℓ-Mult gates that evaluates it, specifically we can do it in time $O(k \cdot \ell \cdot \log(k \cdot \ell))$.*

2.4 Cloning: Handling High Fan-out in the Circuit

We have described how to efficiently realize a permutation over $w > \ell$ items using ℓ-Add, ℓ-Mult and ℓ-$\mathsf{Permute}$ gates that operate on ℓ-element arrays. However, the wiring between adjacent levels of a fan-in-two circuit are typically not permutations, since we typically have gates with high fan-out. We therefore need to clone the output values of these high-fan-out gates before performing a permutation that maps them to their input positions at the next level. We describe an efficient procedure for this "cloning" step.

A cloning procedure. The input to the cloning procedure consists of a collection of k arrays, each with ℓ slots, where each slot is either "full" (i.e., contains a value that we want to use) or "empty" (i.e., contains a don't-care value). We assume that initially more than $k \cdot \ell/2$ of the available slots are full, and will maintain a similar invariant throughout the procedure. Denote the number of full slots in the input arrays by w (with $k \cdot \ell/2 < w \leq k \cdot \ell$), and denote the i'th input value by v_i. The ordering of input values is arbitrary – e.g., we concatenate all the arrays and order input values by their index in the concatenated multi-array.

We are also given a set of positive integers $m_1, \ldots, m_w \geq 1$, such that v_1 should be duplicated m_1 times, v_2 should be duplicated m_2 times, etc. We say that m_i is the *intended multiplicity* of v_i. The total number of full slots in the output arrays will therefore be $w' \overset{\text{def}}{=} m_1 + m_2 + \cdots + m_w \geq w$. In more detail, the output of the cloning procedure must consist of some number k' of ℓ-slot arrays, where $k'\ell/2 < w' \leq k'\ell$, such that v_1 appears in at least m_1 of the output slots, v_2 appears in at least m_2 of the output slots, etc.

Denote the largest intended multiplicity of any value by $M = \max_i\{m_i\}$. The cloning procedure works in $\lceil \log M \rceil$ phases, such that after the j'th phase each value v_i is duplicated $\min(m_i, 2^j)$ times. Each phase consists of making a copy of all the arrays, then for values that occur too many times marking the excess slots as empty (i.e., marking the extra occurrences as don't-care values), and finally merging arrays that are "sparse" until the remaining arrays are at least half full. A simple way to merge two sparse arrays is to permute them so that the full slots appear in the left half in one array and the right half in the other, and then apply Select in the obvious way. A pseudo-code description of this procedure is given in Figure 1, whilst the proof of the following lemma is in Appendix B.

```
Input: $k$ $\ell$-slot arrays, $A_1, \ldots, A_k$, each of the $k \cdot \ell$ slots containing either a value or the special symbol '$\perp$',
        $w$ positive integers $m_1, \ldots, m_w \geq 1$, where $w$ is the number of full slots in the input arrays.
Output: $k'$ $\ell$-slot arrays, $A'_1, \ldots, A'_{k'}$, with each slot containing either a value or the special symbol '$\perp$',
        where $k'/2 \leq (\sum_i m_i)/\ell \leq k'$ and each input value $v_i$ is replicated $m_i$ times in the output arrays
Set $M \leftarrow \max_i\{m_i\}$
For $j = 1$ to $\lceil \log M \rceil$                          // The $j$'th phase
    Make another copy of all the arrays                         // Duplicate everything
    While there are values $v_i$ with multiplicity more than $m_i$:
        Replace the excess occurrences of $v_i$ by $\perp$       // Remove redundant entries
    While there exist pairs of arrays that have between them $\ell$ or more slots with $\perp$:
        Pick one such pair and merge the two arrays              //Merge sparse arrays
Output the remaining arrays
```

Fig. 1. The cloning procedure

Lemma 5. *(i) The cloning procedure from Figure 1 is correct.*

***(ii)** Assuming that at least half the slots in the input arrays are full, this procedure can be implemented by a network of $O(w'/\ell \cdot \log(w'))$ ℓ-fold gates of type ℓ-Add, ℓ-Mult and ℓ-Permute, where w' is the total number of full slots in the output, $w' = \sum m_i$. The depth of the network is bounded by $O(\log w')$.*

***(iii)** This network can be constructed in time $\tilde{O}(w')$, given the input arrays and the m_i's.*

We also describe some more optimizations in Appendix A, including a different cloning procedure that improves on the complexity bound in Lemma 5. Putting all the above together we can efficiently evaluate a circuit using ℓ-Permute, ℓ-Add and ℓ-Mult, yielding a proof of Theorem 1, see Appendix B.

3 Permutation Networks from Abelian Group Actions

As we will show in Section 4, the algebra underlying our FHE scheme makes it possible to perform inexpensive operations on packed ciphertexts, that have the effect of permuting the ℓ plaintext slots inside this packed ciphertext. However, not every permutation can be realized this way; the algebra only gives us a small set of "simple" permutations. For example, in some cases, the given automorphisms "rotate" the plaintext slots, transforming a ciphertext that encrypts the vector $\langle v_0, \ldots, v_{\ell-1} \rangle$ into one that encrypts $\langle v_k, \ldots, v_{\ell-1}, v_0, \ldots, v_{k-1} \rangle$, for any value of k of our choosing. (See Section 3.2 for the general case.)

Our goal in this section is therefore to efficiently implement an ℓ-$\mathsf{Permute}_\pi$ operation for an arbitrary permutation π using only the simple permutations that the algebra gives us (and also the ℓ-Add and ℓ-Mult operations that we have available). We begin in Section 3.1 by showing how to efficiently realize arbitrary permutations when the small set of "simple permutations" is the set of rotations. In Section 3.2 we generalize this construction to a more general set of simple permutations.

3.1 Permutation Networks from Cyclic Rotations and Swaps

Consider the Beneš permutation network discussed in Lemma 3. It has the interesting property that when the 2^r items being permuted are labeled with r-bit strings, then the i-th level only swaps (or not) pairs whose index differs in the $|r-i|$-th bit. In other words, the i-th level swaps only disjoint pairs that have offset $2^{|r-i|}$ from each other. We call this operation an "offset-swap", since all pairs of elements that might be swapped have the same mutual offset.

Definition 1 (Offset Swap). *Let $I_\ell = \{0, \ldots, \ell-1\}$. We say that a permutation π over I_ℓ is an i-offset swap if it consists only of 1-cycles and 2-cycles (i.e., $\pi = \pi^{-1}$), and moreover all the 2-cycles in π are of the form $(k, k+i \bmod \ell)$ for different values $k \in I_\ell$.*

Offset swaps modulo ℓ are easy to implement by combining two rotations with the Select operation defined in Section 2.3. Specifically, for an i-offset swap, we need rotations by i and $-i$ mod ℓ and two Select operations. By Lemma 3, a Beneš network can realize any permutation over 2^r elements using $2r-1$ levels where the i-th level is a $2^{|k-i|}$-offset swap modulo 2^r. An i-offset modulo 2^r, $\ell < 2^r < 2\ell$ can be cobbled together using a constant number of offset swaps modulo ℓ and Select operations, with offsets i and $2\ell - i$. Therefore, given a cyclic group of "simple" permutations $\mathcal{H}$ and Select operations, we can implement any permutation using a Beneš network with low overhead. Specifically, we prove the following lemma in Appendix B.

Lemma 6. *Fix an integer ℓ and let $k = \lceil \log \ell \rceil$. Any permutation π over $I_\ell = \{0, \ldots, \ell-1\}$ can be implemented by a $(2k-1)$-level network, with each level consisting of a constant number of rotations and* Select *operations on ℓ-arrays.*

Moreover, regardless of the permutation π, the rotations that are used in level i $(i = 1, \ldots, 2k-1)$ are always exactly $2^{|k-i|}$ and $\ell - 2^{|k-i|}$ positions, and the network depends on π only via the bits that control the Select *operations. Finally, this network can be constructed in time $\tilde{O}(\ell)$ given the description of π.*

3.2 Generalizing to Sharply-Transitive Abelian Groups

Below, we extend our techniques above to deal with a more general set of "simple permutations" that we get from our ring automorphisms. (See Sections 4 and C.3.)

Definition 2 (Sharply Transitive Permutation Groups). *Denote the ℓ-element symmetric group by $\mathcal{S}_\ell$ (i.e., the group of all permutations over $I_\ell = \{0, \ldots, \ell-1\}$), and let $\mathcal{H}$ be a subgroup of $\mathcal{S}_\ell$. The subgroup $\mathcal{H}$ is* sharply transitive *if for every two indexes $i, j \in I_\ell$ there exists a unique permutation $h \in \mathcal{H}$ such that $h(i) = j$.*

Of course, the group of rotations is an example of an abelian and sharply transitive permutation group. It is abelian: rotating by k_1 positions and then by k_2 positions is the same as rotating by k_2 positions and then by k_1 positions. It is also sharply transitive: for all i, j there is a single rotation amount that maps index i to index j, namely rotation by $j-i$. However, rotations are certainly not the only example. We now explain how to efficiently realize arbitrary permutations using as building blocks the permutations from any sharply-transitive abelian group.

Recall that any abelian group is isomorphic to a direct product of cyclic groups, hence $\mathcal{H} \cong C_{\ell_1} \times \cdots \times C_{\ell_k}$ (where C_{ℓ_i} is a cyclic group with ℓ_i elements for some integers $\ell_i \geq 2$ where ℓ_i divides ℓ_{i+1} for all i). As any cyclic group with ℓ_i elements is isomorphic to $I_{\ell_i} = \{0, 1, \ldots, \ell_i - 1\}$ with the operation of addition mod ℓ_i, we will identify elements in $\mathcal{H}$ with vectors in the box $\mathcal{B} = I_{\ell_1} \times \cdots \times I_{\ell_k}$, where composing two group elements corresponds to adding their associated vectors (modulo the box). The group $\mathcal{H}$ is generated by the k unit vectors $\{\boldsymbol{e}_r\}_{r=1}^k$ (where $\boldsymbol{e}_r = \langle 0, \ldots, 0, 1, 0, \ldots, 0\rangle$ with 1 in the r-th position). We stress that our group $\mathcal{H}$ has polynomial size, so we can efficiently compute the representation of elements in $\mathcal{H}$ as vectors in $\mathcal{B}$.

Since $\mathcal{H}$ is a sharply transitive group of permutations over the indexes $I_\ell = \{0, \ldots, \ell - 1\}$, we can similarly label the indexes in I_ℓ by vectors in $\mathcal{B}$: Pick an arbitrary index $i_0 \in I_\ell$, then for all $h \in \mathcal{H}$ label the index $h(i_0) \in I_\ell$ with the vector associated with h. This procedure labels every element in I_ℓ with exactly one vector from $\mathcal{B}$, since for every $i \in I_\ell$ there is a unique $h \in \mathcal{H}$ such that $h(i_0) = i$. Also, since $\mathcal{H} \cong \mathcal{B}$, we use all the vectors in $\mathcal{B}$ for this labeling ($|\mathcal{H}| = |\mathcal{B}| = \ell$). Note that with this labeling, applying the generator $\boldsymbol{e}_r$ to an index labeled with vector $\boldsymbol{v} \in \mathcal{B}$, yields an index labeled with $\boldsymbol{v'} = \boldsymbol{v} + \boldsymbol{e}_r \bmod \mathcal{B}$. Namely we increment by one the r'th entry in $\boldsymbol{v}$ $\pmod{\ell_r}$, leaving the other entries unchanged.

In other words, rather than a one-dimensional array, we view I_ℓ as a k-dimensional matrix (by identifying it with $\mathcal{B}$). The action of the generator $\boldsymbol{e}_r$ on this matrix is to rotate it by one along the r-th dimension, and similarly applying the permutation $\boldsymbol{e}_r^k \in \mathcal{H}$ to this matrix rotates it by k positions along the r-th dimension. For example, when $k = 2$, we view I_ℓ as an $\ell_1 \times \ell_2$ matrix, and the group $\mathcal{H}$ includes permutations of the form $\boldsymbol{e}_1^k$ that rotate all the columns of this matrix by k positions and also permutations of the form $\boldsymbol{e}_2^k$ that rotate all the rows of this matrix by k positions.

Using Lemma 6, we can now implement arbitrary permutations along the r'th dimension using a permutation network built from offset-swaps along the r'th dimension. Moreover, since the offset amounts used in the network do not depend on the specific permutation that we want to implement, we can use just one such network to implement in parallel different arbitrary permutations on different r'th-dimension sub-matrices. For example, in the 2-dimensional case, we can effect a different permutation on every column, yet realize all these different permutations using just one network of rotations and Selects, by using the same offset amounts but different Select bits for the different columns. More generally we can realize arbitrary (different) ℓ/ℓ_r permutations along all the different "generalized columns" in dimension-r, using a network of depth $O(\log \ell_r)$ consisting of permutations $h \in \mathcal{H}$ and ℓ-fold Select operations (and we can construct that network in time $\ell/\ell_r \cdot \tilde{O}(\ell_r) = \tilde{O}(\ell)$).

Once we are able to realize different arbitrary permutations along the different "generalized columns" in all the dimensions, we can apply Lemma 2. That lemma allows us to decompose any permutation π on I_ℓ into $2k-1$ permutations $\pi = \pi_i \circ \cdots \circ \pi_{2k-1}$ where each π_i consists only of permuting the generalized columns in dimension $r = |k - i|$. Hence we can realize an arbitrary permutation on I_ℓ as a network of permutations $h \in \mathcal{H}$ and ℓ-fold Select operations, of total depth bounded by $2\sum_{i=0}^{k-1} O(\log \ell_i) = O(\log \ell)$ (the last bound follows since $\ell = \prod_{i=0}^{k-1} \ell_i$). Also we can construct that network in time bounded by $2\sum_{i=0}^{k-1} \tilde{O}(\ell_i) = \tilde{O}(\ell)$ (the bound follows since $k \leq \log \ell$). Concluding this discussion, we have:

Lemma 7. *Fix any integer ℓ and any abelian sharply-transitive group of permutations over I_ℓ, $\mathcal{H} \subset \mathcal{S}_\ell$. Then for every permutation $\pi \in \mathcal{S}_\ell$, there is a permutation network of depth $O(\log \ell)$ that realizes π, where each level of the network consists of a constant number of permutations from $\mathcal{H}$ and* Select *operations on ℓ-arrays.*

Moreover, the permutations used in each level do not depend on the particular permutation π, the network depends on π only via the bits that control the Select *operations. Finally, this network can be constructed in time $\tilde{O}(\ell)$ given the description of π and the labeling of elements in $\mathcal{H}, I_\ell$ as vectors in $\mathcal{B}$.* □

Lemma 7 tells us that we can implement an arbitrary ℓ-Permute operation using a log-depth network of permutations $h \in \mathcal{H}$ (in conjunction with ℓ-Add and ℓ-Mult). Plugging this into Theorem 1 we therefore obtain:

Theorem 2. *Let ℓ, t, w and W be parameters, and let $\mathcal{H}$ be an abelian, sharply-transitive group of permutations over I_ℓ.*

Then any t-gate fan-in-2 arithmetic circuit C with average width w and maximum width W, can be evaluated using a network of $O\big(\lceil t/\ell \rceil \cdot \lceil \ell/w \rceil \cdot \log W \cdot \mathrm{polylog}(\ell)\big)$ ℓ-fold gates of types ℓ-Add, ℓ-Mult, and $h \in \mathcal{H}$. The depth of this network of ℓ-fold gates is at most $O(\log W \cdot \log \ell)$ times that of the original circuit C, and the description of the network can be computed in time $\tilde{O}(t \cdot \log \ell)$ given the description of C. □

4 FHE With Polylog Overhead

Theorem 2 implies that if we could efficiently realize ℓ-Add, ℓ-Mult, and $\mathcal{H}$-actions on packed ciphertexts (where $\mathcal{H}$ is a sharply transitive abelian group of permutations on ℓ-slot arrays), then we can evaluate arbitrary (wide enough) circuits with low overhead. Specifically, if we could set $\ell = \Theta(\lambda)$ and realize ℓ-Add, ℓ-Mult, and $\mathcal{H}$-actions in time $\tilde{O}(\lambda)$, then we can realize any circuit of average width $\Omega(\lambda)$ with just $\mathrm{polylog}(\lambda)$ overhead. It remains only to describe an FHE system that has the required complexity for these basic homomorphic operations.

4.1 The Basic Setting of FHE Schemes Based on Ideal Lattices and Ring LWE

Many of the known FHE schemes work over a polynomial ring $\mathbb{A} = \mathbb{Z}[X]/F(X)$, where $F(X)$ is irreducible monic polynomial, typically a cyclotomic polynomial. Ciphertexts are typically vectors (consisting of one or two elements) over $\mathbb{A}_q = \mathbb{A}/q\mathbb{A}$ where q is an integer modulus, and the plaintext space of the scheme is $\mathbb{A}_p = \mathbb{A}/p\mathbb{A}$ for some integer modulus $p \ll q$ with $\gcd(p, q) = 1$, for example $p = 2$. (Namely, the plaintext is represented as an integer polynomial with coefficients mod p.) Secret keys are also vectors over $\mathbb{A}_q$, and decryption works by taking the inner product $b \leftarrow \langle \mathbf{c}, \mathbf{s} \rangle$ in $\mathbb{A}_q$ (so b is an integer polynomial with coefficients in $(-q/2, q/2]$) then recovering the message as $b \bmod p$. Namely, the decryption formula is $[[\langle \mathbf{c}, \mathbf{s} \rangle \bmod F(X)]_q]_p$ where $[\cdot]_q$ denotes modular reduction into the range $(-q/2, q/2]$. Below we consider ciphertext vectors and secret-key vectors with two entries, since this is indeed the case for the variant of the BGV scheme [3] that we use.

Smart and Vercauteren [18] observed that the underlying ring structure of these schemes makes it possible to realize homomorphic (batch) Add and Mult operations, i.e. our ℓ-Add and ℓ-Mult. Specifically, though $F(X)$ is typically irreducible over $\mathbb{Q}$, it may nonetheless factor modulo p; $F(X) = \prod_{i=0}^{\ell-1} F_i(X) \bmod p$. In this case, the plaintext space of the scheme also factors: $\mathbb{A}_p = \otimes_{j=0}^{\ell-1} \mathbb{A}_{\mathfrak{p}_j}$ where $\mathfrak{p}_i$ is the ideal in $\mathbb{A}$ generated by p and $F_i(X)$. In particular, the Chinese Remainder Theorem applies, and the plaintext space is partitioned into ℓ independent non-interacting "plaintext slots", which is precisely what we need for component-wise ℓ-Add and ℓ-Mult. The decryption formula recovers the "aggregate plaintext" $a \leftarrow [[\langle \mathbf{c}, \mathbf{s} \rangle \bmod F(X)]_q]_p$, and this aggregate plaintext is decoded to get the individual plaintext elements, roughly via $z_j \leftarrow a \bmod (F_i(x), p) \in \mathbb{A}_{\mathfrak{p}_j}$.

4.2 Implementing Group Actions on FHE Plaintext Slots

While component-wise Add and Mult are straightforward, getting different plaintext slots to interact is more challenging. For ease of exposition, suppose at first that $F(X)$ is the degree-$(m-1)$ polynomial $\Phi_m(X) = (X^m - 1)/(X - 1)$ for m prime, and that $p \equiv 1 \pmod{m}$. Thus our ring $\mathbb{A}$ above is the mth cyclotomic number field. In this case $F(X)$ factors to linear terms modulo p, $F(X) = \prod_{i=0}^{\ell-1}(X - \rho_i) \pmod{p}$ with $\rho_i \in \mathbb{F}_p$. Hence we obtain $\ell = m - 1$ plaintext slots, each slot holding an element of the finite field $\mathbb{F}_p$ (i.e. in this case $\mathbb{A}_{\mathfrak{p}_i}$ above is equal to $\mathbb{F}_p$).

To get Φ_m to factor modulo p into linear terms we must have $p \equiv 1 \pmod{m}$, so $p > m$. Also we need $m = \Omega(\lambda)$ to get security (since m is roughly the dimension of the underlying lattice). This means that to get Φ_m to factor into linear terms we must use plaintext spaces that are somewhat large (in particular we cannot directly use $\mathbb{F}_2$). Later in this section we sketch the more elaborate algebra needed to handle the general (and practical) case of non-prime m and $p \ll m$, where Φ_m may not factor into linear terms. This is covered in more detail in Appendix C. For now, however, we concentrate on the simple case where Φ_m factors into linear terms modulo p.

Recall that ciphertexts are vectors over $\mathbb{Z}_q[X]/\Phi_m(X)$, so each entry in these vectors corresponds to an integer polynomial. Consider now what happens if we simply replace X with X^i inside all these polynomials, for some exponent $i \in Z_m^*, i > 1$. Namely, for each polynomial $f(X)$, we consider $f^{(i)}(X) = f(X^i) \bmod \Phi_m(X)$. Notice that if we were using polynomial arithmetic modulo $X^m - 1$ (rather then modulo $\Phi_m(X)$) then this transformation would just permutes the coefficients of the polynomials. Namely $f^{(i)}$ has the same coefficients as f but in a different order, which means that if the coefficient vector of f has small norm then the same holds for the coefficient vector of $f^{(i)}$. In Appendix D we show that using a different notion of "size" of a polynomial (namely, the norm of the canonical embedding of a polynomial rather than the norm of its coefficient vector), we can conclude the same also for mod-Φ_m polynomial arithmetic. Namely, the mapping $f(X) \mapsto f(X^i) \bmod \Phi_m(X)$ does not change the "size" of the polynomial. To simplify presentation, below we describe everything in terms of coefficient vectors and arithmetic modulo $X^m - 1$. The actual mod-Φ_m implementation that we use is described in Appendix D (and a slightly different implementation is described in Appendix E).

Let us now consider the effect of the transformation $X \mapsto X^i$ on decryption. Let $\mathbf{c} = (c_0(X), c_1(X))$ and $\mathbf{s} = (s_0(X), s_1(X))$ be ciphertext and secret-key vectors, and let $b = \langle \mathbf{c}, \mathbf{s} \rangle \bmod (X^m - 1, q)$ and $a = b \bmod p$. Denote $\mathbf{c}^{(i)} = (c_0(X^i), c_1(X^i)) \bmod (X^m - 1)$, and define $\mathbf{s}^{(i)}$, $b^{(i)}$ and $a^{(i)}$ similarly. Since $\langle \mathbf{c}, \mathbf{s} \rangle = b \pmod{X^m - 1, q}$, we have that

$$c_0(X)s_0(X) + c_1(X)s_1(X) \;=\; b(X) + q \cdot r(X) + (X^m - 1)s(X) \;\; (\text{over } \mathbb{Z}[X])$$

for some integer polynomials $r(X), s(X)$, and therefore also

$$c_0(X^i)s_0(X^i) + c_1(X^i)s_1(X^i) \;=\; b(X^i) + q \cdot r(X^i) + (X^{mi} - 1)s(X^i) \;\; (\text{over } \mathbb{Z}[X]).$$

Since $X^m - 1$ divides $X^{mi} - 1$, then we also have

$$\left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle \;=\; b^{(i)} + q \cdot r(X^i) + (X^m - 1)S(X) \;\; (\text{over } \mathbb{Z}[X])$$

for some $r(X), S(X)$. That is, $b^{(i)} = \left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle \bmod (X^m - 1, q)$. Clearly, we also have $a^{(i)} = b^{(i)} \pmod{p}$. This means that if $\mathbf{c}$ decrypts to the aggregate plaintext a under $\mathbf{s}$, then $\mathbf{c}^{(i)}$ decrypts to $a^{(i)}$ under $\mathbf{s}^{(i)}$!

The cryptosystem from [3, 4] have a mechanism for "key switching" (which is also applicable to the scheme from [5]), transforming a ciphertext $\mathbf{c}$ that decrypts to a under $\mathbf{s}$ to a new ciphertext $\mathbf{c}'$ that decrypts to the same a under some other secret key $\mathbf{s}'$. Using the same mechanism, we can translate the transformed ciphertext $\mathbf{c}^{(i)}$ into one that decrypts to $a^{(i)}$ under another $\mathbf{s}'$ of our choice. We can even translate it back to a ciphertext decryptable under the original $\mathbf{s}$ is we are willing to assume circular security. Using the BGV cryptosystem [5, 4, 3] with appropriate parameters, key switching can be accomplished in time $\tilde{O}(\lambda)$. (See Appendices D and E for details on our variants of the BGV scheme [5].)

But how does this new aggregate plaintext $a^{(i)}$ relate to the original a? Here we apply to Galois theory, which tells us that decoding the aggregate $a^{(i)}$ (which we do roughly by setting $z_j \leftarrow a^{(i)} \bmod (F_j, p)$), the set of z_j's that we get is exactly the same as when decoding the original aggregate a, albeit in different order. Roughly, this is because each of our plaintext slots corresponds to a root of the polynomial $F(X)$, and the transformations $X \mapsto X^i$, which are precisely the elements of the Galois group, permute these roots. In other words by transforming $\mathbf{c} \to \mathbf{c}^{(i)}$ (followed by key switching), we can permute the plaintext slots inside the packed ciphertext. Moreover, in our simplified case, the permutations have a single cycle – i.e., they are rotations of the slots. Arranging the slots appropriately we can get that the transformation $\mathbf{c} \to \mathbf{c}^{(i)}$ rotates the slots by exactly i positions, thus we get the group of rotations that we were using in Section 3.1. In general the situation is a little more complicated, but the above intuition still can be made to hold; for more details see Appendix C.

The general case. In the general case, when m is not a prime, the polynomial $\Phi_m(X)$ has degree $\phi(m)$ (where $\phi(\cdot)$ is Euler's totient function), and it factors mod p into a number of same-degree irreducible factors. Specifically, the degree of the factors is the smallest integer d such that $p^d = 1 \pmod{m}$, and the number of factors is $\ell = \phi(m)/d$ (which is of course an integer), $\Phi_m(X) = \prod_{j=0}^{\ell-1} F_j(X)$. For us, it means that we have ℓ plaintext slots, each isomorphic to the finite field $\mathbb{F}_{p^d}$, and an aggregate plaintext is a degree-$(\phi(m)-1)$ polynomial over $\mathbb{F}_p$.

Suppose that we want to evaluate homomorphically a circuit over some underlying field $\mathbb{K}_n = \mathbb{F}_{p^n}$, then we need to find an integer m such that $\Phi_m(X)$ factors mod p into degree-d factors, where d is divisible by n. This way we could directly embed elements of the underlying plaintext space $\mathbb{K}_n$ inside our plaintext slots that hold elements of $\mathbb{F}_{p^d}$, and addition and multiplication of plaintext slots will directly correspond to additions and multiplications of elements in $\mathbb{K}_n$. (This follows since $\mathbb{K}_n = \mathbb{F}_{p^n}$ is a subfield of $\mathbb{F}_{p^d}$ when n divides d.)

Note that each plaintext slot will only have $n \log p$ bits of relevant information, i.e., the underlying element of $\mathbb{F}_{p^n}$, but it takes $d \log p$ bits to specify. We thus get an "embedding overhead" factor of d/n even before we encrypt anything. We therefore need to choose our parameter m so as to keep this overhead to a minimum.

Even for a non-prime m, the Galois group $\mathcal{G}\mathsf{al}(\mathbb{Q}[X]/\Phi_m(X))$ consists of all the transformations $X \mapsto X^i$ for $i \in \mathbb{Z}_m^*$, hence there are exactly $\phi(m)$ of them. As in the simplified case above, if we have a ciphertext $\mathbf{c}$ that decrypts to an aggregate plaintext a under $\mathfrak{s}$, then $\mathbf{c}^{(i)}$ decrypts to $a^{(i)}$ under $\mathfrak{s}^{(i)}$. Differently from the simple case, however, not all members of the Galois group induce permutations on the plaintext slots, i.e., decoding the aggregate plaintext $a^{(i)}$ does not necessarily give us the same set of (permuted) plaintext elements as decoding the original a. Instead $\mathcal{G}\mathsf{al}(\mathbb{Q}[X]/\Phi_m(X))$ contains a subgroup $\mathcal{G} = \{(X \mapsto X^{p^j}) : j = 0, 1, \ldots, d-1\}$ corresponding to the Frobenius automorphisms[6] modulo p. This subgroup does not permute the slots at all, but the quotient group $\mathcal{H} = \mathcal{G}\mathsf{al}/\mathcal{G}$ does. Clearly, $\mathcal{G}$ has order d and $\mathcal{H}$ has order $\phi(m)/d = \ell$. In Appendix C we show that the quotient group $\mathcal{H}$ acts as a transitive permutation group on our ℓ plaintext slots, and since it has order ℓ then it must be sharply transitive. In the general case we therefore use this group $\mathcal{H}$ as our permutation group for the purpose of Lemma 7. Another complication is that the automorphism that we can compute are elements of $\mathcal{G}\mathsf{al}$ and not elements in the quotient group $\mathcal{H}$. In Appendix C we also show how to emulate the permutations in $\mathcal{H}$, via use of coset representatives in $\mathcal{G}\mathsf{al}$.

4.3 Parameter Setting for Low-Overhead FHE

Given the background from above (and the modification of the BGV cryptosystem [5] in Appendices D or E), we explain how to set the parameters for our variant of the BGV scheme so as to get low-overhead FHE scheme. Below we first show how to evaluate depth-L circuits with average-width $\Omega(\lambda)$ with overhead of only $\tilde{O}(L)\cdot\mathrm{polylog}(\lambda)$, and then use bootstrapping to get overhead of $\mathrm{polylog}(\lambda)$ regardless of depth.

Plaintext-Space Terminology and Notations The discussion below refers to three different "plaintext spaces":

[6] The group G is called the *decomposition group* at p in the literature.

- The *"underlying plaintext space"*: The circuit that we want to evaluate homomorphically is an arithmetic circuit over some (finite) ring, and that finite ring is the "underlying plaintext space". We typically think of the underlying plaintext space as being just $\mathbb{F}_2$, but it is sometimes convenient to use other spaces (e.g., $\mathbb{F}_{2^8}$ when computing AES, or perhaps $\mathbb{F}_p$ for some 32-bit prime p in other applications).
  In this work we always assume that the underlying plaintext space is small, either of constant size or at most of size polynomial in λ. Moreover, we assume that it is a field, namely $\mathbb{K}_n = \mathbb{F}_{p^n}$ for some prime p and integer $n \geq 1$.
- The *"embedded plaintext space"*. This is what is held in each of our plaintext slots. For example, we could have underlying space $\mathbb{F}_2$, but embed our bits in elements of $\mathbb{F}_p$ for some larger integer p, or maybe in elements of $\mathbb{F}_{2^d}$ for some $d > 1$. (In the former case we need to emulate binary XOR using a degree-2 polynomial mod p, in the latter case multiplication and addition work as expected.)
- The *"aggregate plaintext space"*. This is the plaintext space that is natively encrypted in the cryptosystem: An element in the aggregate plaintext space is a polynomial in some $\mathbb{F}_p[X]$, and as explained above it encodes (via CRT) an ℓ-vector over the embedded plaintext space.

When choosing parameters for our FHE construction, we are given the depth and width of the circuits that we need to evaluate homomorphically, as well as the underlying plaintext space and the security parameter. We then want to choose the "embedded" and "aggregate" plaintext spaces and all the other parameters so as to minimize the overhead. Namely, minimize the ratio between the number of gates in the underlying circuits and the time that it takes to evaluate them homomorphically. We describe two methods for choosing the parameters: One is likely to be more efficient in practice, but we can only prove that it yields low overhead for either small underlying plaintext spaces (of size polylog(λ)) or very wide circuits (of width $\Omega(\lambda \cdot p^n)$). The other (simpler) method can be shown to work for any poly-size underlying plaintext space and circuits of width $\Omega(\lambda)$, but is almost certain to yield worst performance in practice.

In either approach, we begin by lower-bounding the dimension of the lattice that we need (in order to get security), thus getting a lower-bound on our parameter m (recall that we will eventually get a dimension-$\phi(m)$ lattice). Once we have this lower-bound M, we either pick $m = p^{ns} - 1 \geq M$ for some integer s, or just choose m as $p' - 1$ for some prime number p' sufficiently larger than M. In the former case we have "embedded plaintext space" $\mathbb{F}_{p^{ns}}$ into which we can directly embed the underlying space $\mathbb{F}_{p^n}$, and in the latter case we need to emulate $\mathbb{F}_{p^n}$ arithmetic using polynomials over $\mathbb{F}_{p'}$.

Once we set the parameter m and get the corresponding "embedded plaintext space", we can easily compute the packing parameter ℓ and all the other parameters.

Step 1. Lower-Bounding the Dimension Suppose that we want to evaluate homomorphically circuits of depth L over some small finite field $\mathbb{F}_{p^n}$, with average depth w and maximum depth $W = \text{poly}(\lambda)$, where λ is the security parameter. Clearly, for security parameter λ we need ciphertexts of size at least $\Omega(\lambda)$, so we cannot hope to evaluate any homomorphic operation faster than $\tilde{O}(\lambda)$. To get low overhead, we therefore must be able to pack at least $\ell = \tilde{\Omega}(\lambda)$ plaintext slots (from our "embedded" space) into one ciphertext. This means that we only get low-overhead implementation when the width of the underlying circuits is at least $\tilde{\Omega}(\lambda)$.

From Theorem 2 we know that for any packing parameter ℓ we can evaluate depth-L circuits using a network of ℓ-fold gates of depth $L' = O(L \cdot \log W \cdot \log \ell)$. (If we use the second approach below for choosing the parameter m then we need another additive term of $L \cdot \log(p^n) = O(L \cdot \log \lambda)$ to emulate $\mathbb{F}_{p^n}$ arithmetic using mod-m polynomials.) We will show below that it is sufficient to choose either $\ell = \Theta(\lambda)$ or $\ell = \Theta(p^n \cdot \lambda) \leq$poly$(\lambda)$ (depending on which of the two approaches we use), but in either case we have $L' \leq c \cdot L \cdot \log W \cdot \log \lambda$ for some constant c that we can compute from the given parameters.

Recall that the BGV cryptosystem needs L' different moduli q_i when evaluating a depth-L' network. When implementing arithmetic operations over a characteristic-p field and working with dimension-M lattices, the largest modulus needs to be $q_0 = (M \cdot p)^{c' \cdot L'}$ (for some constant $c' < 2$) to get the homomorphic evaluation functionality, and $M \geq \lambda \cdot \log q_0$ to get security. Plugging in all these constraints, we get a lower-bound on the dimension of the lattice $M \geq c'' \cdot L \cdot \lambda \log \lambda \cdot \log W \cdot \log p$ for some constant c'' that we can compute from the given parameters (note that $M = \tilde{\Theta}(L \cdot \lambda)$).

Step 2. Choosing the parameter m Below we will choose our parameter m so as to get $\phi(m) \geq M$. We use the following lemma, whose proof is in Appendix B.

Lemma 8. *For all positive integers m we have $m/\phi(m) = O(\log \log m)$.*

We will then choose our parameter m larger than $c^* M$ for some $c^* = O(\log \log M)$, to ensure that $\phi(m) \geq M$.

Approach 1: Using Extension Fields. Setting $s = \lceil \log_{p^n}(c^* M + 1) \rceil$, we see that the integer $m = p^{ns} - 1$ satisfies all our requirements. On one hand it is large enough, $m \geq c^* M$ by construction. On the other hand for $d = n \cdot s$ we clearly have that $p^d = 1 \pmod{m}$, which is what we need in order to use the "embedded plaintext space" $\mathbb{F}_{p^d}$ with the "aggregate plaintext space" $\mathbb{F}_p[X]/\Phi_m(X)$.

Moreover, the "embedding overhead" $d/n = s$ is small: since $M = \tilde{O}(L \cdot \lambda)$ and $s \leq \log_2(c^* M + 1)$ then clearly $s = O(\log(L \cdot \lambda))$. Thus the number of bits that it takes to specify an "aggregate plaintext" is only a factor of $O(\log(L \cdot \lambda))$ larger than what you need to specify all the elements of the "underlying plaintext space" that are embedded in this aggregate plaintext.

However, in some cases the parameter m itself (and therefore the lattice dimension) could be large: Note that we have $M = \tilde{O}(L \cdot \lambda)$ and since $s = \lceil \log_{p^n}(c^* M + 1) \rceil$ then $p^{ns} < (c^* M + 1) \cdot p^n$. If the size of the underlying plaintext space (i.e., p^n) is polylogarithmic, then we have $m = \tilde{O}(L \cdot \lambda)$ which is what we need. However, if the underlying plaintext size is larger, say $p^n \approx \lambda$, then we could have $m = \tilde{\Theta}(L \cdot \lambda^2)$. In this case we can no longer hope to evaluate homomorphic operations in time $\tilde{O}(L \cdot \lambda)$ (since the ciphertext size is too large).

If the circuits that we want to evaluate are very wide (i.e., of width $\tilde{\Omega}(\lambda \cdot p^n)$) then we can just pack sufficiently many plaintext slots inside each ciphertext to get the overhead down. We can do this since the "embedding overhead" is logarithmic. But for narrower circuits, say of width $\Theta(\lambda + p^n)$, we just don't have enough plaintext to put in all these slots, hence our overhead increases.

We point out that we may be able to do better than $m = p^{ns} - 1$, for example we can use any m' such that $\phi(m') > M$ and m' divides $p^{ns} - 1$. But it is not clear that such $m' < m$ exists (for example when $p = 2$ then $p^{ns} - 1$ could be a prime number). It is also permissible to choose some $s' > s$ and then choose m' that divides $p^{ns'} - 1$ with $\phi(m') \geq M$. As long as $s' \leq$polylog$(L \cdot \lambda)$ then we still have only a polylog "embedding overhead", and m' may be much smaller than $m = p^{ns} - 1$. Unfortunately we were not able to prove that such $s' \leq$polylog$(L \cdot \lambda)$ and $m' \leq \tilde{O}(L \cdot \lambda)$ always exist, we consider this an interesting open problem.

Approach 2: Using Prime Fields. An alternative, simpler, approach is to just pick $m = p' - 1$ for a prime number p' sufficiently larger than M, (so as to get $\phi(m) \geq M$), and set our "embedded plaintext space" to be $\mathbb{F}_{p'}$. This will give us the "simple case" that we discussed earlier in this section, where Φ_m factors into linear terms mod p'. Note that in this case we clearly have $m = \tilde{O}(M)$, so (a) the "embedding overhead" is at most $O(\log M) = \tilde{O}(\log(L\lambda))$, and (b) as long as we work with circuits of width $\tilde{\Omega}(\lambda)$ we can pack enough plaintext elements into each ciphertext to get low overhead.

This solutions has a few drawbacks, however. One relatively minor drawback is that the native operations of the scheme are now over a characteristic-p' field, and if $p' > p$ then the bound M on the dimension will be slightly larger than before (since the noise in fresh ciphertexts is now of the form $p' \cdot e$ rather that $p \cdot e$). A more serious problem is that each gate of the underlying circuit must now be emulated using a polynomial mod p'. We note, however, that this only results in a logarithmic slowdown: It is not hard to see that arithmetic over $\mathbb{F}_{p^n}$ can be emulated by mod-p' circuits of depth and size $O(n \cdot \log p)$ (e.g., express these operations as binary circuits and emulate that binary circuit mod-p').

Once we determined the parameter m and the "embedded plaintext space", all the other parameters of the scheme easily follow, and we obtain the following theorem:

Theorem 3. *For security parameter λ, any t-gate, depth-L arithmetic circuit of average width $\Omega(\lambda)$ over underlying plaintext space $\mathbb{F}_{p^n}$ (with $p^n \leq$poly(λ)) can be evaluated homomorphically in time $t \cdot \tilde{O}(L) \cdot$polylog$(\lambda)$.*

4.4 Achieving Depth-Independent Overhead

Theorem 3 implies that we can implement shallow arithmetic circuit with low overhead, but when the circuit gets deeper the dependence of the overhead on L causes the overhead to increase. Recall that the reason for this dependence on the depth is that in the BGV cryptosystem [3], the moduli get smaller as we go up the circuit, which means that for the first layers of the circuit we must choose moduli of bitsize $\Omega(L)$.

As explained in [3], the dependence on the depth can be circumvented by using bootstrapping. Namely, we can start with a modulus which is not too large, then reduce it as we go up the circuit, and once the modulus become too small to do further computation we can bootstrap back into the larger-modulus ciphertexts, then continue with the computation.

For our purposes, we need to ensure that we bootstrap often enough to keep the moduli small, and yet that the time we spend on bootstrapping does not significantly impact the overhead. Here we apply to the analysis from [3], that shows that a packed ciphertext with $\tilde{\Omega}(\lambda)$ slots can be decrypted using a circuit of size $\tilde{O}(\lambda)$ and depth polylog(λ). Hence we can even bootstrap after every layer of the circuit and still keep the overhead polylogarithmic, and the moduli never grow beyond polylogarithmic bitsize. We thus get:

Theorem 4. *For security parameter λ, any t-gate arithmetic circuit of average width $\Omega(\lambda)$ over underlying plaintext space $\mathbb{F}_{p^n}$ (with $p^n \leq$poly(λ)) can be evaluated homomorphically in time $t\cdot$polylog(λ).*

References

1. Paul T. Bateman, Carl Pomerance, and Robert C. Vaughan. On the size of the coefficients of the cyclotomic polynomial. In *Topics in Classical Number Theory, Vol. I*, pages 171–202, 1984.
2. Václav E. Beneš. Optimal rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43:16411656, 1964.
3. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Manuscript at http://eprint.iacr.org/2011/277, 2011.
4. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE, 2011.
5. 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.
6. I. Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarais. Multiparty computation from somewhat homomorphic encryption. Manuscript at http://eprint.iacr.org/2011/535, 2011.
7. Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465. Springer, 2010.

8. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `http://crypto.stanford.edu/craig`.
9. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
10. 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.
11. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 2006.
12. Kristin Lauter, Michael Naehrig, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? Manuscript at http://www.codeproject.com/News/15443/Can-Homomorphic-Encryption-be-Practical.aspx, 2011.
13. Frank Thomson Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. M. Kaufmann Publishers, 2 edition, 1992.
14. G. Lev, N. Pippenger, and L. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C-30:93–100, 1981.
15. 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.
16. Ron Rivest, Leonard Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–180, 1978.
17. 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.
18. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. Manuscript at http://eprint.iacr.org/2011/133, 2011.
19. Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 377–394. Springer, 2010.
20. Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968.
21. Lawrence C. Washington. *Introduction to Cyclotomic Fields*, volume 83 of *Graduate Texts in Mathematics*. Springer, 1996.

A Additional Optimizations

A.1 Faster Cloning

In Lemma 5 we establish that we can clone w' values using ℓ-fold operations in time $O((w' \log w')/\ell)$. Below we show how to remove the $\log w'$ term, which would allow us to clone values between levels in the circuit using asymptotically optimal $O(w'/\ell)$ time.

Recall that for the cloning procedure we are given a "multi-array" $\mathbf{A}'$ consisting of several ℓ-element arrays, and also the intended multiplicities of the values in these arrays $m_1, \ldots, m_w$. As before, denote the maximum intended multiplicity by $M = \max_i\{m_i\}$. The new procedure consists of two main parts:

Decomposition: For $i = 0, 1 \ldots, M$, construct a "multi-array" $\mathbf{A}'_i$ that contains the elements whose intended multiplicity is at least 2^i, as follows:

Set $\mathbf{A}'_0 = \mathbf{A}'$. Then for $i > 0$ we compute $\mathbf{A}'_i$ from $\mathbf{A}'_{i-1}$ by marking the slots of all the elements with intended multiplicity smaller than 2^i as empty, and then merging sparse arrays until the multi-array is at least half-full (or contains only one array). Note that when computing $\mathbf{A}'_i$ from $\mathbf{A}'_{i-1}$, we also keep a copy of $\mathbf{A}'_{i-1}$ for use in the aggregation part below.

Aggregation: For $i = M, \ldots, 1, 0$, construct a multi-array $\mathbf{A}_i$ as follows. Set $\mathbf{A}_M = \mathbf{A}'_M$, then for all $i < M$ concatenate two copies of $\mathbf{A}_{i+1}$ with one copy of $\mathbf{A}'_i$, and if the result is not half full them merge sparse arrays until it is half full again. The result is $\mathbf{A}_i$.

Note since each of $\mathbf{A}_{i+1}, \mathbf{A}'_i$ is either half full or contains a single array, then at most two merge operations are needed in each aggregation step. The output of the cloning procedure is $\mathbf{A}_0$.

Lemma 9. *The procedure above is correct, and it uses only $O(\frac{w'}{\ell} + \log w')$ copy and merge operations on ℓ-element arrays, where $w' = \sum_i m_i$*

Proof. Consider an arbitrary element of the input multi-array $\mathbf{A}'$, with intended multiplicity $m_i \in [2^j, 2^{j+1}-1]$ for some j. The decomposition part will output multi-arrays such that this element is in each of $\mathbf{A}'_0, \ldots, \mathbf{A}'_j$. Then, during the aggregation part, $\mathbf{A}_j$ will include one copy of this element, $\mathbf{A}_{j-1}$ three copies, $\mathbf{A}_{j-2}$ seven copies, and in general $\mathbf{A}_{j-k}$ contains $2^{k+1} - 1$ copies. Hence at the end of the aggregation part, $\mathbf{A}_0$ includes $2^{j+1} - 1$ occurrences of this element (which is at least as much as m_i but less than $2m_i$).

To analyze complexity, notice that the number of arrays in every multi-array $\mathbf{A}'_j$ equals the number of arrays in $\mathbf{A}'_{j-1}$ minus the number of merge operations that were used when computing $\mathbf{A}'_j$. Since $\mathbf{A}'_M$ cannot have less than zero arrays, it follows that the total number of merge operations throughout the decomposition part cannot be more than the initial number of arrays, namely $\lceil 2w/\ell \rceil \le \lceil 2w'/\ell \rceil$. We observed above that the aggregation part does at most two merges for each $\mathbf{A}_j$, so the total number of merges during this part is at most $2\lceil \log M \rceil \le 2\lceil \log w' \rceil$. Thus the total number of merge operations is bounded by $N = \lceil 2w'/\ell \rceil + 2\lceil \log M \rceil = O(\frac{w'}{\ell} + \log w')$.

Finally, the output multi-array $\mathbf{A}'$ contains at most twice as many occurrences of each element as needed, and it is at least half full. Hence it contains at most $\lceil \frac{4w'}{\ell} \rceil$ arrays, which means that the entire procedure duplicated arrays at most $\lceil \frac{4w'}{\ell} \rceil + N = O(\frac{w'}{\ell} + \log w')$ times. □

The procedure above can be made particularly efficient in our case, when used in conjunction with the following optimization: When considering a circuit, we sort the gates in each level according to their fan-out, thus making the input to the cloning procedure sorted by the intended multiplicity. Note that the decomposition part now becomes unnecessary, we just define $\mathbf{A}'_j$ to be the collection of the first few arrays, all the ones that contain elements of intended multiplicity at least 2^j.

Also important is that once the inputs are sorted, merging arrays do not need the full power of the Permute operation. As long as we keep the full slots in the arrays continuous, we can use the simple rotation operation to align the two arrays before we merge them. (The same can be done with the "higher-dimensional rotations" that we get in the general case in Section 4.) Hence the entire cloning network can be implemented using only $O(\frac{w'}{\ell} + \log w')$ basic operations of ℓ-Add, ℓ-Mult, and $\mathcal{H}$-actions.

A.2 Faster Routing

Tracing through the proofs in Section 2, in conjunction with the more efficient cloning technique from above, one can verify that the $\log W$ term in the statement of Theorem 1 can be made to multiply only the number of ℓ-Add and ℓ-Mult gates, not ℓ-Permute, which can make a big difference in practice. Roughly, the $\log W$ term arises from the fact that we seem to need $\Omega(W \cdot \log W)$ computation (in the worst-case) to route the inter-level wires. Note that such a $\log W$ term does not appear in the overhead of non-batched FHE schemes that operate on singletons rather than arrays. It seems plausible that this term could be eliminated somehow, and we consider this an interesting open problem.

A.3 Powering (Almost) for Free

In some applications, plaintext elements are not bits or integers, but rather elements in a finite extension field. For example, when implementing homomorphic AES, it may be convenient to use $\mathbb{F}_{2^8}$ as the underlying plaintext space [12, 18]. In these cases, the corresponding Galois group (whose automorphisms we use to permute the slots) includes also the Frobenius automorphism. (This is $x \to x^{2^j}$ in the AES example, and more generally $x \to x^{p^j}$

when using a characteristic-p field.) We show in Section 4 that applying the Galois group transformations to packed ciphertexts results in almost no additional noise. Thus we get a new function, ℓ-Frobenius, that raises the ℓ slots in parallel to a power of p, while adding almost no additional noise. This may not be surprising, since the Frobenius map is a linear operation on $\mathbb{F}_{p^n}$.

In practice this turns out to be a useful optimization for particular functions of interest: For the case of AES, the only non-linear part of AES is inversion in $\mathbb{F}_{2^8}$, which is equivalent to exponentiation to the 254-th power. While this may seem to be high-degree, the Frobenius automorphism allows us to evaluate this power relatively cheaply on ℓ elements in parallel. For an $a \in \mathbb{F}_{2^8}$ sitting in a plaintext slot, we use the Frobenius map to compute $a_j = a^{2^j}$ for $j = 1, 2, \ldots, 7$ (these are the '1's in the binary representation of 254), then multiply all the a_j to get $a^{254} = a^{-1}$. Thus, we can evaluate a^{254} at a price of only seven products (in terms of noise), and this 7-fold product can be computed by a depth-3 circuit. The binary affine transformation of the AES S-box is not linear over $\mathbb{F}_{2^8}$, but it *is* linear over the outputs of the Frobenius automorphisms, and so it *is* linear in terms of its effect on ciphertext noise (although to extract and pack the bits uses up two more levels in the circuit). The ShiftRows and MixColumns operation take four more levels using our permutation networks, and the matrix multiplication in the MixColumns uses another level. An AES round can therefore be accomplished using only a depth-10 circuit (in terms of noise), so homomorphic implementation of the full AES-128 will take a circuit of depth less than 100. It is therefore plausible that we could implement AES-128 homomorphically without resorting to bootstrapping at all!!! (We note, however, that many other optimizations are possible, and it is not clear if the approach sketched above is really the most efficient one for implementing AES-128.)

B Proofs

Lemma 1. *Let $S = \{0, \ldots, a-1\} \times \{0, \ldots, b-1\}$ be a set of ab positions, arranged as a matrix of a rows and b columns. For any permutation π over S, there are permutations π_1, π_2, π_3 such that $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ (that is, π is the composition of the three permutations) and such that π_1 and π_3 only permute positions within each column (these permutations only change the row, not the column, of each element) and π_2 only permutes positions within each row. Moreover, there is a polynomial-time algorithm that given π outputs the decomposition permutations π_1, π_2, π_3.*

Proof. The basic strategy of the decomposition is that π_2 will send each element to some address with the same y-coordinate as its target destination, and similarly π_3 will correct all of the x-coordinates. The permutation π_1, on the other hand, serves as a strategic indirection. The reason this indirection is needed – i.e., the reason we cannot decompose π just as $\pi_3 \circ \pi_2$ with the properties above – is that several elements in the same row could have the same target y-coordinate (and thus π_2 cannot achieve its goal). Thus, π_1 is used to ensure that, when π_2 receives its input, no two elements in the same row have the same target column. The only nontrivial part of the proof is showing that a suitable π_1 always exists.

For $s \in S$, let s_x and s_y denote its x and y coordinates, namely $s = (s_x, s_y)$. Consider a bipartite graph $G = (V_1, V_2, E)$ where V_1 and V_2 each have b vertexes with labels $\{0, \ldots, b-1\}$. For every $s \in S$, we draw an edge from the V_1-vertex labeled s_y to the V_2-vertex labeled $\pi(s)_y$, and we label the edge 's'. (We may have more than one edge between the same pair of vertices's.) Clearly, this is a bipartite, a-regular graph. Therefore G's edges can be partitioned into a perfect matches, and this partition can be computed efficiently (e.g., using network-flow algorithms). In other words, one can compute in polynomial time a coloring of the edges of G using the colors $\{0, \ldots, a-1\}$, such that for all i the i-colored subgraph G_i of G is a perfect matching.

Let $\rho(s)$ denote the color of the edge labeled 's'. Now, define π_1, π_2, π_3 as follows: for all $s = (s_x, s_y) \in S$:

$$\pi_1(s) = (\rho(s), s_y), \quad \pi_2 \circ \pi_1(s) = (\rho(s), \pi(s)_y), \quad \pi_3 \circ \pi_2 \circ \pi_1(s) = (\pi(s)_x, \pi(s)_y)$$

Clearly, π_1, π_3 have the claimed property of only permuting within columns and π_2 only permutes within rows. All that remains is to establish that they are all well-defined permutations – i.e., that no "collisions" occur. π_1 is a permutation because no two edges emanating from the V_1-vertex labeled 's_y' have the same color. π_2 is a permutation, in particular it permutes elements in row i, because the subgraph G_i is a perfect matching. Finally, π_3 is a permutation since both $\pi_2 \circ \pi_1$ and π are permutations and since $\pi = \pi_3 \circ \pi_2 \circ \pi_1$. □

Lemma 4. *Evaluating ℓ permutation networks in parallel, each permuting k items, can be accomplished using $O(k \cdot \log k)$ gates of ℓ-*Add *and ℓ-*Mult*, and depth $O(\log k)$. Also, evaluating a permutation π over $k \cdot \ell$ elements that are packed into k ℓ-element arrays, can be accomplished using k ℓ-*Permute *gates and $O(k \log k)$ gates of ℓ-*Add *and ℓ-*Mult*, in depth $O(\log k)$. Moreover, there is an efficient algorithm that given π computes the circuit of ℓ-*Permute*, ℓ-*Add*, and ℓ-*Mult *gates that evaluates it, specifically we can do it in time $O(k \cdot \ell \cdot \log(k \cdot \ell))$.*

Proof. The first statement follows directly from Lemma 3 and the discussion above. The second statement follows from Lemma 1, which says that the permutation π can be decomposed as $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ where π_1 and π_3 each involve evaluating n permutation networks in parallel across the ℓ indexes, and π_2 only permutes elements within each ℓ-element array, and therefore can be done using k gates of ℓ-Permute and just one level.

The efficiency of computing the circuit that realizes π follows from the fact that the decomposition π_1, π_2, π_3 can be computed efficiently, as per Lemma 1. In fact, it was shown by Lev et al. [14] that this decomposition can be computed in time $O(k \cdot \ell \cdot \log(k \cdot \ell))$. □

Lemma 5. ***(i)** The cloning procedure from Figure 1 is correct.*

***(ii)** Assuming that at least half the slots in the input arrays are full, this procedure can be implemented by a network of $O(w'/\ell \cdot \log(w'))$ ℓ-fold gates of type ℓ-*Add, *ℓ-*Mult *and ℓ-*Permute*, where w' is the total number of full slots in the output, $w' = \sum m_i$. The depth of the network is bounded by $O(\log w')$.*

***(iii)** This network can be constructed in time $\tilde{O}(w')$, given the input arrays and the m_i's.*

Proof. In each phase j, first the number of occurrences of every value is doubled, and next if a value v_i occurs more than m_i times then the excess occurrences are removed. Therefore after the j'th phase each value v_i is duplicated $\min(m_i, 2^j)$ times. Denoting the number of full slots after the j'th phase by $w_j \stackrel{\text{def}}{=} \sum_i \min(m_i, 2^j)$, we have at the end of phase j some number k_j of ℓ-slot arrays, where $(k_j - 1)\ell/2 < w_j \leq k_j \cdot \ell$, since once the merging part is over we must have at least half the slots full. Correctness now follows easily just by looking at $j = \lceil \log M \rceil$.

Regarding complexity (part (ii)), we note that if the input arrays are at least half full then at the beginning of every iteration we have $k_{j-1} \leq 2w_{j-1}/\ell =< 2w'/\ell = O(w'/\ell)$ arrays (clearly $w_j < w'$ for all j by definition.) After the duplication step (Line 2) we have $2k_{j-1}$ arrays, and then each merging step (Line 6) removes one array, so we can have at most $2k_{j-1} = O(w'/\ell)$ such steps. Observing that every merge takes a constant number of gates (two ℓ-Permute gates and one Select operation), we conclude that each phase takes at most $O(w'/\ell)$ ℓ-fold gates.[7] The number of phases is $\lceil \log M \rceil \leq \lceil \log w' \rceil$, and the claimed complexity follows.

Part (iii) follows easily by noting that the network implementing each phase can be constructed in time quasi-linear in the number of slots that are available at the beginning of that phase, just by using greedy algorithms to make all the decisions. (The most time-consuming operation is marking entries as "don't-care"s in Line 4, everything else can be done in time $\tilde{O}(w'/\ell)$.) □

Theorem 1. *Let ℓ, t, w and W be parameters. Then any t-gate fan-in-2 arithmetic circuit C with average width w and maximum width W, can be evaluated using a network of $O\big(\lceil t/\ell \rceil \cdot \lceil \ell/w \rceil \cdot \log W \cdot \text{polylog}(\ell)\big)$ ℓ-fold gates*

[7] Note that removing redundant values (Line 4) does not take any gates, we leave the arrays unchanged and just mark the redundant values as "don't-care"s.

*of types ℓ-*Add*, ℓ-*Mult*, and ℓ-*Permute*. The depth of this network of ℓ-fold gates is at most $O(\log W)$ times that of the original circuit C, and the description of the network can be computed in time $\tilde{O}(t)$ given the description of C.*

Proof. Consider one level of the circuit with w' gates, where in the previous level we computed $w \leq 2w'$ input values, packed into $O(\lceil w/\ell \rceil)$ ℓ-element arrays. Our approach is to first clone and then permute these values so that the $2w'$ input slots of the w' gates are filled correctly. More precisely, these $2w'$ input slots will be arranged in two sets of ℓ-slot array, one set for the left inputs and the other for the right inputs to all the gates. Concatenating these two sets of arrays into two multi-arrays, we arrange the slots such that the left and right inputs to each gate are aligned in the same index in the two multi-arrays. Once all the values are routed to their correct locations in the multi-arrays, the actual computation of the gates in this layer can obviously be evaluated only $O(\lceil w'/\ell \rceil)$ ℓ-fold gates of ℓ-Adds or ℓ-Mults.

By Lemma 5, we can compute the multi-arrays of $O(w'/\ell)$ ℓ-element arrays that contains the inputs with sufficient multiplicity using $O(\lceil w'/\ell \rceil \cdot \log(w'))$ ℓ-fold gates. The resulting multi-arrays have $O(w)$ slots (more than either the source or target multi-arrays), at least half of which contain "real values" while the other slots contain "don't-care"s. Let π be a permutation over these $O(w)$ slots that maps the slots that contain the real values to the appropriate positions in the target multi-arrays. By Lemma 4 we can evaluate π with a network of $O(w'/\ell \text{polylog}\lceil w'/\ell \rceil)$ n-fold gates, and can compute the structure of that network in time $\tilde{O}(w')$.

The result for the whole circuit follows easily, using as our inductive hypothesis that the w' outputs are indeed packed into $O(\lceil w'/\ell \rceil)$ ℓ-element arrays for input to the next level. □

Lemma 6. *Fix an integer ℓ and let $k = \lceil \log \ell \rceil$. Any permutation π over $I_\ell = \{0, \ldots, \ell - 1\}$ can be implemented by a $(2k-1)$-level network, with each level consisting of a constant number of rotations and* Select *operations on ℓ-arrays.*

Moreover, regardless of the permutation π, the rotations that are used in level i $(i = 1, \ldots, 2k-1)$ are always exactly $2^{|k-i|}$ and $\ell - 2^{|k-i|}$ positions, and the network depends on π only via the bits that control the Select *operations. Finally, this network can be constructed in time $\tilde{O}(\ell)$ given the description of π.*

Proof. If ℓ is a power of two then the network is just a Beneš network. Otherwise (i.e., $2^{k-1} < \ell < 2^k$ for some k) the basic strategy is to realize a permutation over I_ℓ by using two k-element arrays to realize a Beneš permutation network over the first 2^k of the 2ℓ positions. We realize each level of the Beneš network using a constant number of rotations and Select operations. Since $2^k > \ell$ then clearly any permutation on I_ℓ can be expressed as a permutation over the first 2^k positions (e.g., where the last $2^k - \ell$ elements remain fixed).

It remains only to show how to realize an i-offset-swap over the first 2^k elements using just a constant number of operations on the two ℓ-slot arrays. Clearly, we can handle all the pairs $(v, v+j)$ where both indexes are in the same array using the rotations j and $\ell - j$ and two Select operations, applied to the each of the arrays. To handle the pairs where v is in the first array and $v + j$ is in the second (at index $v + j - \ell$), we shift the first array by $\ell - j$ and the second array by j, then again use two Select operations (one Select on the first array and the shifted version of the second, the other Select on the second array and the shifted version of the first). All in all we have four rotation operations (two for each array) and six Select's. The "Finally" part follows directly from Lemma 3. □

Lemma 8. *For all positive integers m we have $m/\phi(m) = O(\log \log m)$.*

Proof. The "worst-case" that maximizes $m/\phi(m)$ is when m is a product of distinct primes $m = p_1 \cdots p_t$, in which case we have $m/\phi(m) = p_1/(p_1 - 1) \cdots p_t/(p_t - 1)$. Clearly, the worst-case is when the p_i's are the *first* t primes. In this case, we can use the prime number theorem to argue that $p_t = \text{polylog}(m)$ (actually, something like $\log m$). By Merten's theorem the product over primes $\prod_{p < \text{polylog} m} p/(p-1)$ is $\theta(\log \log m)$.

C Basic Algebra

To understand our techniques it is first necessary to recap on the underlying algebra of cyclotomic fields. We have tried to cover as much detail as needed, but the reader should be aware a self contained treatment will be hard to come by in such a short space. We therefore refer the interested reader to [21] for details on cyclotomic fields.

C.1 Reductions of Cyclotomic Fields

We let $\mathit{\Phi}_m(X)$ be the m-th cyclotomic polynomial, and let $K = \mathbb{Q}(\zeta_m)$ denote the associated number field. The degree of $\mathit{\Phi}_m$ is $\phi(m)$, where $\phi(\cdot)$ is Euler's phi-function. Note that asymptotically m is of the same size as $\phi(m)$, but for the small values of m that we will use in practice, $\phi(m)$ is roughly 10%-50% smaller than m. We associate K with the set of rational polynomials in X of degree less than N, with multiplication and addition defined modulo $\mathit{\Phi}_m$. We let the ring of integers of K be denoted by $\mathcal{O}_K = \mathbb{Z}[\zeta_m]$.

We now fix a prime p, which is neither ramified in K, nor an index divisor (i.e. p does not divide m). Consider the reduction of K at p; we define

$$\mathbb{A}_p := \mathbb{Z}_p[X]/\mathit{\Phi}_m(X)$$

to be the ring of polynomials over $\mathbb{Z}_p$ where multiplication and addition are defined modulo $\mathit{\Phi}_m$ and p. Note, we assume that the representation of $\mathbb{A}_p$ is such that the coefficients are given in the range $(-p/2, p/2]$. In general $\mathbb{A}_p$ is not a field but is an *algebra*, since $\mathit{\Phi}_m$ is generally not irreducible mod p.

Since p is neither an index divisor nor ramified, and because $K/\mathbb{Q}$ is Galois, we have that the polynomial $\mathit{\Phi}_m$ splits mod p into ℓ distinct factors $F_i(X)$, each of degree d, where $\ell \cdot d = \phi(m)$. We then have that

$$\begin{aligned}\mathbb{A}_p &\cong \mathbb{Z}_p[X]/F_0(X) \times \ldots \times \mathbb{Z}_p[X]/F_{\ell-1}(X)\\ &= \mathbb{L}_0 \times \ldots \times \mathbb{L}_{\ell-1} =: A_p.\end{aligned}$$

i.e. the reduction of K modulo p is isomorphic to ℓ copies $\mathbb{L}_i = \mathbb{Z}_p[X]/F_i(X)$ of $\mathbb{F}_{p^d}$. Since all finite fields of a given degree are isomorphic, each of these copies of $\mathbb{F}_{p^d}$ is isomorphic to each other. Note we let $\mathbb{A}_p$ denote the representation of the algebra by polynomials modulo $\mathit{\Phi}_m$ and A_p denote the algebra by a set of l copies of the fields defined by the polynomials $F_i(X)$.

We note there is a natural homomorphic inclusion maps $\mathbb{A}_p \longrightarrow \mathcal{O}_K$ defined by mapping $\mathbb{A}_p$ to the coset representative with coefficients in $(-p/2, p/2]$. If $\alpha \in \mathcal{O}_K$ then we let $\alpha \mod p$ denote the inverse in $\mathbb{A}_p$ under this inclusion. If q is a prime greater than p then we can also consider elements of $\mathbb{A}_p$ as elements in $\mathbb{A}_q$ but this inclusion is not a homomorphism (since it only preserve the arithmetic operations "as long as there is no wraparound").

We will use $\mathbb{A}_p$ (resp. A_p) in two distinct ways. In the first way we use $\mathbb{A}_p$ and A_p to describe the message space of our scheme; in this case we take p to be small (think $p = 2$, or a 32-bit prime). In the second way, we use $\mathbb{A}_q$ (for a large prime q) as an approximation of the global object $\mathbb{A}$. Looking ahead the basic construction is that we take an element $\alpha \in \mathbb{A}_p$, then form the element in $\mathbb{A}_q$ given by $\alpha + p^t \cdot \tau$, where τ is referred to as the *noise*. Public operations are then performed, and these will correspond to valid operations in $\mathbb{A}_p$ only if the noise term does not become too large (in the sense of the ∞-norm of the noise becoming bigger than $q/2$). If the operation is does not result in wrap-around then we can (upon decrypting) obtain the plaintext in $\mathbb{A}_p$.

C.2 Underlying Plaintext Algebra

Each message in $\mathbb{A}_p$ actually corresponds to ℓ messages in $\mathbb{F}_{p^d} \cong \mathbb{Z}_p[X]/F_i(X)$. We call each of these components a "slot". By the Chinese Remainder Theorem, additive and multiplicative operations in $\mathbb{A}_p$ correspond to SIMD operations on the slots. However, in many applications we will be interested in plaintexts where each slot lies in $\mathbb{F}_{p^n}$, for some n dividing d. (In particular this includes the important case of $n = 1$.) In addition an application may have a preferred representation (i.e. preferred polynomial basis) for the underlying field, $\mathbb{F}_{p^n}$.

We therefore fix (or are given) an irreducible polynomial $G(X) \in \mathbb{Z}_p[X]$, of degree n, which defines the specific polynomial basis we are interested in; we take $G(X) = X - 1$ when $n = 1$. To fix notation we define $\mathbb{K}_n = \mathbb{Z}_p[X]/G(X)$ to denote one copy of this degree n field, with the given polynomial representation.

Note, in applications one is given p and n, and then one needs to find values of m which enable the above representation. Basic algebra shows us that $\Phi_m(X)$ will have a degree d factor if and only if m divides $p^d - 1$. Thus, given p and n, we need to select m such that for some value $d = s \cdot n$, we have m divides $p^d - 1$. The value ℓ is given by $\phi(m)/d$.

For each of our fields $\mathbb{L}_i = \mathbb{Z}_p[X]/F_i(X)$ there will be a distinct homomorphic embedding of $\mathbb{K}_n$ into $\mathbb{L}_i$ which we will denote by $\Psi_{n,i}$, which will be an isomorphism in the case when $n = d$. Our basic plaintext space will now be defined as ℓ copies of $\mathbb{K}_n$, i.e. $\mathcal{M} = (\mathbb{K}_n)^\ell$, where addition and multiplication will be defined component-wise. We therefore can define a map

$$\Psi_n : \begin{cases} \quad\mathcal{M} \quad\longrightarrow \quad A_p \\ (m_0, \ldots, m_{\ell-1}) \longmapsto (\Psi_{n,0}(m_0), \ldots, \Psi_{n,\ell-1}(m_{\ell-1})). \end{cases}$$

By applying the Chinese Remainder Theorem given an element $\mathbf{a} \in A_p$ we can obtain a value $\alpha \in \mathbb{A}_p$; we write $\alpha = \mathsf{CRT}_p(\mathbf{a})$. Note, our use of notations: Elements in $\mathbb{A}_p$ and $\mathbb{B}_p$ will be represented by lower case Greek letters; elements in A_p and $\mathcal{M}$ will be represented by bold face roman letters (since they are vectors); and elements in $\mathbb{K}_n$ and $\mathbb{L}_i$ will be represented by standard lower case roman letters.

We end this discussion of the plaintext space by noting that there is a simple operation that produces the projection map. If we consider the element $\pi_i \in \mathbb{A}_p$ which is defined by the element in A_p given by the i unit vector $\mathbf{e}_i$. Then if $\mathbf{m} = (m_0, \ldots, m_{\ell-1}) \in A_p$ that $\pi_i \cdot \mathsf{CRT}_p(\mathbf{m}) = \mathsf{CRT}_p(0, \ldots, 0, m_i, 0, \ldots, 0)$. From π_i we can also define a projection on an arbitrary subset $I \subset \{0, \ldots, \ell - 1\}$ in the obvious way; by defining π_I to be the element $\sum_{i \in I} \mathsf{CRT}_p(\mathbf{e}_i)$.

C.3 Galois Theory of Cyclotomic Fields

The field $K = \mathbb{Q}(\zeta_m)$ is abelian (i.e. has abelian Galois group) and has Galois group given by $\mathcal{G}\mathsf{al}(K/\mathbb{Q}) \cong (\mathbb{Z}/m\mathbb{Z})^*$. If we think of X in the representation of K as denoting a generic mth root of unity ζ_m, then given an element $i \in (\mathbb{Z}/m\mathbb{Z})^*$ the associated element of the Galois group is given by the mapping $\kappa_i : X \mapsto X^i$.

We now need to consider how the Galois group $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ works when we consider K modulo p, to A_p and $\mathbb{A}_p$. Notice, that since $\mathbb{A}_p$ is not a field the usual theorems of Galois Theory do not apply (an obvious fact but worth stating). The maps defined by the Galois group commute with our functions Ψ_n, and CRT_p etc. Thus, to fix ideas, consider an element $\mathbf{m} = (m_0, \ldots, m_{\ell-1}) \in \mathcal{M} = \mathbb{K}^\ell$. We obtain the corresponding element in $\mathbb{A}_p$ by applying $\alpha = \mathsf{CRT}_p(\Psi_n(\mathbf{m})) \in \mathbb{A}_p$. Now if we apply the element κ_i from $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ to the element α we obtain an element β such that $\beta = \mathsf{CRT}_p(\Psi_n(\kappa_i(m_1), \ldots, \kappa_i(m_\ell)))$, where $\kappa_i(m_j(X)) = m_j(X^i) \pmod{G(X)}$.

Considering how automorphisms work on $\mathbb{A}_p$, it is well known that any field $\mathbb{F}_{p^k}$ has Galois group over $\mathbb{Z}_p$ given by the cyclic group C_k of order k. Now since $\mathbb{A}_p$ contains the subfield $\mathbb{F}_{p^d}$ we have that $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ contains the cyclic subgroup $C_d \lhd (\mathbb{Z}/m\mathbb{Z})^*$. The group C_d is called the *decomposition group* of a prime ideal lying above p in K. The group C_d is generated by the element $p \in (\mathbb{Z}/m\mathbb{Z})^*$, which corresponds to the Frobenius map $\kappa_p : X \mapsto X^p$. In what follows we let $\mathcal{G}$ denote this subgroup C_d of $(\mathbb{Z}/m\mathbb{Z})^*$

Considering how $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ acts on $\mathbb{K}_n$, we notice that the Galois group of $\mathbb{K}_n$ over $\mathbb{Z}_p$ is given by $C_n \cong C_d/C_{d/n}$ and generated by the Frobenius map. The key difference, between $\mathbb{K}_n$ and $\mathbb{K}_d$, being that the map κ_{p^n}

is the identity on the subfields $\mathbb{K}_n$. If we want to restrict to the Galois group of $\mathbb{K}_n$ we let $\hat{\mathcal{G}}$ denote the subset $\{1, p, p^2, \ldots, p^{n-1}\}$ consisting of a set of representatives for the Galois group of $\mathbb{K}_n$.

Since $(\mathbb{Z}/m\mathbb{Z})^*$ is abelian all subgroups are normal, and hence we can define quotient groups, and so we define $\mathcal{H}$ to be the quotient group $(\mathbb{Z}/m\mathbb{Z})^*/\mathcal{G}$, note $\mathcal{H}$ has order ℓ. We write $\mathcal{H}$ as a product of cyclic groups $C_{n_1} \times C_{n_t}$ with n_i dividing n_{i+1}. As a set of coset representatives for $\mathcal{H}$ we first pick a coset representative h_i for C_{n_i}, and then as the coset representatives of all other elements we take those elements in $(\mathbb{Z}/m\mathbb{Z})^*$ given by

$$\prod_{i=1}^{t} h_i^{e_i} \text{ for } 0 \leq e_i < n_i.$$

Thus we can identify $\mathcal{H}$ with a *subset* of $(\mathbb{Z}/m\mathbb{Z})^*$.

If we label the roots of Φ_m in K by $\zeta_m^{(0)}$ to $\zeta_m^{(\phi(m)-1)}$ then it is a standard fact that the Galois group acts transitively on these roots. The subgroup $\mathcal{G}$ acts on these roots, and we can partition the set of roots into disjoint sets with respect to the group action of $\mathcal{G}$. That is we create $\ell = \phi(m)/d$ subsets each of d elements, we label these subsets $X_0, \ldots, X_{\ell-1}$. Since $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ acts transitively on the set $\{\zeta_m^{(0)}, \ldots, \zeta_m^{(\phi(m)-1)}\}$, the quotient group $\mathcal{H} = \mathcal{G}\mathsf{al}(K/\mathbb{Q})/\mathcal{G}$ acts transitively on the set $X_0, \ldots, X_{\ell-1}$.

Since $\mathcal{G}$ was the decomposition group of p the sets X_i, each containing d complex roots, when reduced modulo p can be placed in correspondence with the roots of $F_i(X)$, i.e. one of the factors of Φ_m modulo p. We need to fix a representative for for each set $X_i \mod p$. Fixing a representative for $X_i \mod p$ means essentially fixing a root of $F_i(X)$ modulo p; and one can think of the symbolic root X being such a root with all other roots being given by a polynomial in X modulo p of degree less than $d-1$ (when reduced arithmetic is considered modulo $F_i(X)$). Since $\mathcal{H}$ has order ℓ and acts transitively on $\{X_0, \ldots, X_{\ell-1}\}$, for each $i \in \{0, \ldots, \ell-1\}$ there is exactly one element σ_i in $\mathcal{H}$ which sends 0 to i. If we fix the representative of the set X_0 to be $\zeta_m^{(0)}$ then to define the representative of the set X_i we take $\sigma_i \in \mathcal{H}$ and set the representative of X_i to be $\sigma_i(\zeta_m^{(0)})$. Since, defining a representative of X_i essentially means fixing a representation of the field $\mathbb{Z}_p[X]/F_i(X)$ this then means that our set of representatives for $\mathcal{H}$ act "transitively on the plaintext slots" in the following sense: For each pair $i, j \in \{0, \ldots, \ell-1\}$ we have that

$$\sigma_j(\sigma_i^{-1}(\mathsf{CRT}_p(\Psi_n(0, \ldots, 0, m_i, 0, \ldots, 0)))) = \mathsf{CRT}_p(\Psi_n(0, \ldots, 0, m_j^{p^t}, 0, \ldots, 0)).$$

for some integer t. In the case $n = 1$ we have $m_j^{p^t} = m_j$ and so our set of representatives for $\mathcal{H}$ act directly as permutations on the slots.

Our main technical contribution in both practical and theoretical terms to FHE is based on the properties of the group $\mathcal{H}$ and how it acts on the plain text slots. It is clear, since $\mathcal{H}$ acts transitively as above and we have projection maps, that we can, given a vector of slots $(m_0, \ldots, m_{\ell-1}) \in \mathbb{K}_n^{\ell}$ map it to an arbitrary permutation of the slots. The naive algorithm for this, consisting of projecting each element, mapping via $\mathcal{H}$ as above, making sure we cope with the possibility of powering by Frobenius, and then recombining via addition, has complexity $O(\ell)$. In Section 3 we showed that an arbitrary permutation on the slots can be realized in $O(t \cdot \log \ell)$ operations, where t is the number of cyclic components of the group $\mathcal{H}$, note $t = O(\log \ell)$. That this algorithm can be applied in our case should be immediate, but to fix ideas, we examine how $\mathcal{H}$ acts on the slots when $\mathcal{H}$ is cyclic; and how to construct our offset swaps in this case.

When $\mathcal{H}$ is cyclic If $\mathcal{H} = \langle h \rangle$ is cyclic we can, by fixing on a given value of $F_0(X)$, reorder the factors $F_i(X)$ so that the factors are precisely those factors corresponding to $\sigma_h^i(1)$. Thus we can consider $\mathcal{H}$ as defining permutations on the factors of Φ_m modulo p. Although $\mathcal{H}$ is rarely cyclic this case is illustrative of what is occurring, and in practice we can often restrict the number of slots to correspond to the largest cyclic subgroup of $\mathcal{H}$. [8] We consider three examples of increasing complexity:

Example 1: The simplest case to understand is when the decomposition group is trivial, i.e. $d = 1$. Consider the case of $m = 11$ and $p = 23$, we have that the polynomial $\Phi_m(X)$ factors into ten linear factors modulo 23, and the Galois group $(\mathbb{Z}/m\mathbb{Z})^*$ is cyclic of order 10 and generated by the element 2. Since $\mathcal{G} = \langle 1 \rangle$ we take using the procedure above $\mathcal{H} \cong (\mathbb{Z}/m\mathbb{Z})^* = \langle 2 \rangle$. Thus we have ten slots and we order them such that we have

$$\kappa_2(\mathsf{CRT}_p(\Psi_n(m_0, m_2, \ldots, m_9))) = \mathsf{CRT}_p(\Psi_n(m_9, m_0, m_2, \ldots, m_8)).$$

Hence κ_2 produces a cyclic shift of the slots. If we wish to switch elements in positions i and j, for $i < j$, the we only need to apply the following operation

$$\mathsf{swap}_{i,j}(\alpha) = \kappa_{2^{j-i}}(\pi_i \cdot \alpha) + \kappa_{2^{i-j}}(\pi_j \cdot \alpha) + \pi_{\{0,\ldots,9\}\setminus\{i,j\}} \cdot \alpha.$$

Example 2: To see what happens for non-trivial decomposition groups we consider the case of $m = 31$ and $p = 2$. We have since $2^5 \equiv 1 \pmod{31}$ that the decomposition group at p is cyclic of order 5, i.e. $d = 5$. In this example we find that by $\mathcal{G}\mathsf{al}$ factors directly into the product of $\mathcal{G} = \langle 2 \rangle$ and the cyclic subgroup $\langle 6 \rangle$. The set of coset representatives for $\mathcal{H}$ we can take to be this subgroup $\langle 6 \rangle$, thus we can identify $\mathcal{H}$ with a subgroup of $\mathcal{G}\mathsf{al}$. This implies that the elements in $\mathcal{H}$ act as direct permutations on the slots, and we do not need to worry about the action of Frobenius. In particular we can define the six slots so that we have, for a specific representation of $\mathbb{K}_n = \mathbb{F}_{2^5}$,

$$\kappa_6(\mathsf{CRT}_2(\Psi_n(m_0, m_1, m_2, m_3, m_4, m_5))) = \mathsf{CRT}_2(\Psi_n(m_5, m_0, m_1, m_2, m_3, m_4)).$$

If we wish to shift to the left we take the elements in $\mathcal{G}\mathsf{al}(K/\mathbb{Q})$ given by $1/6^i \pmod{m}$, so for example since $1/6 = 26 \pmod{31}$ we have

$$\kappa_{26}(\mathsf{CRT}_2(\Psi_n(m_0, m_1, m_2, m_3, m_4, m_5))) = \mathsf{CRT}_2(\Psi_n(m_1, m_2, m_3, m_4, m_5, m_0)).$$

If we wish to switch elements, for an element $\alpha \in \mathbb{A}_p$, in positions i and j, with $i < j$, then we apply the following operation

$$\mathsf{swap}_{i,j}(\alpha) = \kappa_{6^{j-i}}(\pi_i \cdot \alpha) + \kappa_{6^{i-j}}(\pi_j \cdot \alpha) + \pi_{\{0,\ldots,5\}\setminus\{i,j\}} \cdot \alpha.$$

Example 3: The above example, in which $\mathcal{H}$ could be identified with a subgroup of $\mathcal{G}\mathsf{al}$ is not typical. In the general case we have the added complication of dealing with actions of Frobenius on applying automorphism corresponding to elements in $\mathcal{H}$. We examine this more general situation via means of an example. We make $m = 257$ and $p = 2$. In this case we find that 2 has order 16 modulo m, and that the quotient group $\mathcal{H} = \mathcal{G}\mathsf{al}/\langle 2 \rangle$ is cyclic of order 16. We also find that there is no cyclic subgroup of order 16 of $\mathcal{G}\mathsf{al}$ which is not equal to $\langle 2 \rangle$. Thus $\mathcal{H}$ cannot be represented as a subgroup of $\mathcal{G}\mathsf{al}$.

We instead represent $\mathcal{H}$ by the set of coset representatives given by $3^i \bmod m$, for $i = 0, \ldots, 15$. Since $3^8 \bmod m = 136 \notin \langle 2 \rangle$, whilst $3^{16} \bmod m = 249 = 2^{11} \bmod m$. We therefore have 16 slots, each consisting of an element in $\mathbb{K}_n = \mathbb{F}_{2^{16}}$. We fix a specific representation of each slot so that

$$\kappa_3(\mathsf{CRT}_2(\Psi_n(m_0, m_1, \ldots, m_{14}, m_{15}))) = \mathsf{CRT}_2(\Psi_n(m_{15}^{2^{11}}, m_0, m_1, \ldots, m_{13}, m_{14})).$$

[8] For implementation purposes restricting the slots in this way is simpler, although for our asymptotic result on FHE with polylog overhead we will require to consider the whole of $\mathcal{H}$.

However, we also have

$$\kappa_8 6(\mathsf{CRT}_2(\mathit{\Psi}_n(m_0, m_1, \ldots, m_{14}, m_{15}))) = \mathsf{CRT}_2(\mathit{\Psi}_n(m_1, \ldots, m_{13}, m_{14}, m_{15}, m_0^{2^5})).$$

Note that $(1/3) \bmod m = 86$, but that 86 is not one of our coset representatives for $\mathcal{H}$.

In other words to move elements to the right (without wrap around) by i places we apply the map $\kappa_{3^i \bmod m}$, but to move elements to the left (without wrap around) by i places we need to apply the map $\kappa_{3^{-i} \bmod m}$. Hence if we wish to switch elements, for an element $\alpha \in \mathbb{A}_p$, in positions i and j, with $i < j$, then we apply the following operation

$$\mathsf{swap}_{i,j}(\alpha) = \kappa_{3^{j-i}}(\pi_i \cdot \alpha) + \kappa_{(1/3)^{j-i} \bmod m}(\pi_j \cdot \alpha) + \pi_{\{0,\ldots,5\}\setminus\{i,j\}} \cdot \alpha.$$

Hence, although the underlying algebra is different when $\mathcal{H}$ cannot be identified with a subgroup of $\mathcal{G}\mathsf{al}$, the method to obtain a swap is exactly the same.

These examples show that for cyclic groups we can realize any transposition via the use of scalar multiplication by the π_I and application of maps κ_i. The above technique also allows us to realize the offset swaps from Definition 1 for any subset $T \subset S = \{0, \ldots, \ell - 1\}$ and any i. The following technique works for when $\mathcal{H} = \langle h \rangle$ is a cyclic group generated by h, generalizing to other groups follows from our methods but leads to more complex formulas. Recall that a permutation π over S is an i-offset swap over S if there exists a subset $T \subset S$ such that the pairs $\{(t, t + i \bmod \ell) : t \in T\}$ are disjoint and π simply swaps each pair (leaving the other elements fixed).

For a set A we let $A + i = \{j + i \bmod \ell : j \in A\}$ and $\overline{A} = S \setminus A$. We also split T into two sets T_L and T_R such that $t \in T_L$ if and only if $t \in T$ and $t + i < \ell$, i.e. T_L is the set of elements in T which can be shifted to the left by i, without wrap around. Algebraically an offset swap on an element α is then defined in terms of our isomorphisms κ_i etc as

$$\pi_{\overline{T \cup (T+i)}} \cdot \alpha + \kappa_{h^i}(\pi_{T_L} \cdot \alpha) + \kappa_{(1/h)^i}(\pi_{T_L+i} \cdot \alpha) + \kappa_{(1/h)^{\ell-i}}(\pi_{T_R} \cdot \alpha) + \kappa_{h^{\ell-i}}(\pi_{T_R+i} \cdot \alpha)$$

The first term corresponds to those elements which are kept fixed by the offset swap, i.e. those elements neither in T nor $T + i$. The second term corresponds to those elements shifted to the left by i without wrap around, the third corresponds to elements shifted to the right by i without wrap around by i without wraparound, the final two terms deal with the case of wraparound.

D Using mod-$\mathit{\Phi}_m$ Polynomial Arithmetic

Part of our goal in this paper is to allow implementations of BGV-type cryptosystems over rings of the form $\mathbb{Z}[X]/\mathit{\Phi}_m(X)$ for arbitrary integers m, not only when m is a prime. Although most of the underlying algebra works the same way regardless of what m is, we do not have a good bound on the increase in the size of coefficient vectors when using mod-$\mathit{\Phi}_m$ arithmetic.

Recall that for every ring $\mathcal{R} = \mathbb{Z}[X]/F(X)$ there is a "ring-constant" $\gamma_{\mathcal{R}}$, such that for all $a, b \in \mathcal{R}$ it holds that $\|ab\| \le \gamma_{\mathcal{R}} \cdot \|a\| \cdot \|b\|$, where $\|x\|$ is the norm of the coefficient-vector of x (say, the l_∞ norm). However, we do not have a good bound on the "ring-constant" for rings of the for $\mathcal{R}_m = \mathbb{Z}[X]/\mathit{\Phi}_m(X)$, and in particular $\gamma_{\mathcal{R}_m}$ can be super-polynomial in m. In particular $\gamma_{\mathcal{R}_m}$ is related to the sizes of the coefficients of $\mathit{\Phi}_m(X)$ which are known to get rather large [1]. In our context, this means that when multiplying two "short" ciphertexts, the result can be "longer" than the product of the two by this factor $\gamma_{\mathcal{R}_m}$ for which we do not have a good bound.

D.1 Canonical Embeddings and Norms

To analyze a cryptosystem that works mod-Φ_m, we therefore use a different measure of "size" of polynomials: Rather than considering the norm of the coefficient vector of a polynomial, we consider the norm of the "canonical embedding" of that polynomial: For an integer m, let $\mathcal{P}_m$ be the set of complex primitive m-th roots of unity. Then for a polynomial $a \in \mathbb{Q}[X]/\Phi_m(X)$, the "canonical embedding" of a is the vector of values that a assumes in all the roots in $\mathcal{P}_m$,

$$\mathcal{E}(a) \stackrel{\text{def}}{=} \left\langle a(\rho^k) \ : \ k \in \mathbb{Z}_m^* \right\rangle, \text{ where } \rho \text{ is a fixed complex primitive } m\text{-th root of unity (e.g., } \rho = e^{-2\pi i/m}).$$

More generally, the canonical embedding of an element $a \in \mathbb{Q}[X]/F(X)$ consists of the evaluations of a in all the complex roots of F. Below we only use the canonical embeddings for the cases $F(X) = \Phi_m(X)$ and $F(X) = X^m - 1$. Note that $\mathcal{E}(a)$ is in general a vector of complex numbers, and the size of each entry in that vector is the norm (absolute value) of that complex number.

Below we refer to the norm of $\mathcal{E}(a)$ as the "canonical embedding norm" of a, and denote it by $\|a\|^{can}$. Although it is possible to define the "canonical embedding l_p norm" for any l_p, below we always refer to the canonical embedding l_∞ norm. Namely,

$$\|a\|^{can} \stackrel{\text{def}}{=} \|\mathcal{E}(a)\| = \max_{k \in \mathbb{Z}_m^*} |a(\rho^k)|.$$

(Note again that in this section we consistently use $\|\cdot\|$ to refer to the l_∞ norm of a vector and not the l_2 norm.) We extend the canonical embedding norm to vectors over $\mathbb{Q}[X]/\Phi_m(X)$ in the natural way, namely if $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ is an n-vector over $\mathbb{Q}[X]/\Phi_m(X)$, then $\|\mathbf{a}\|^{can} = \max_{i<n} \|a_i\|^{can}$.

It is easy to see that for any element $a \in \mathbb{Q}[X]/\Phi_m(X)$, the canonical embedding norm is not much more than the coefficient norm, namely $\|a\|^{can} < \phi(m) \cdot \|a\|$ (where $\|a\|$ is the norm of a's coefficient vector). This follows since each of the m-th roots of unity has norm one, and we are adding $\phi(m)$ of them with coefficients bounded by $\|a\|$. Clearly, for any two elements $a, b \in \mathbb{Z}[X]/\Phi_m(X)$ we have $\|a+b\|^{can} \le \|a\|^{can} + \|b\|^{can}$, and since the primitive m-th roots of unity are all roots of $\Phi_m(X)$ then $\|ab \bmod \Phi_m(X)\|^{can} = \|ab\|^{can} \le \|a\|^{can} \cdot \|b\|^{can}$. Similarly for n-vectors $\mathbf{a}, \mathbf{b} \in (\mathbb{Q}[X]/\Phi_m(X))^n$ we get $\| \langle \mathbf{a}, \mathbf{b} \rangle \bmod \Phi_m(X)\|^{can} \le n \cdot \|\mathbf{a}\|^{can} \cdot \|\mathbf{b}\|^{can}$.

Also, for every m there exists a "ring constant" c_m (which is a real number) such that for all $a \in \mathbb{Z}[X]/\Phi_m(X)$ it holds that $\|a\| \le c_m \cdot \|a\|^{can}$; see [6] for a discussion of c_m. Another property of the canonical embedding norm that we use below, is that a nonzero integer polynomial must have norm at least one:

Lemma 10. *Let $a \in \mathbb{Z}[X]/\Phi_m(X)$ for some integer m, then $\|a\|^{can} \ge 1$.*

Proof. Since a is a nonzero integer polynomial, then the result of the complex product $\prod_{k \in \mathbb{Z}_m^*} a(\rho^k)$ must be a nonzero integer, and therefore it has magnitude at least 1. It follows that some of the terms in the product must have magnitude 1 or more, hence the l_∞ norm of $\mathcal{E}(a)$ is at least 1. □

Modular Reduction in Canonical Embedding. To talk about the canonical norm of elements in $\mathbb{Z}_q[X]/\Phi_m(X)$ (i.e., polynomials reduced both mod $\Phi_m(X)$ and mod q), we define the "canonical embedding norm reduced mod q", denoted $|a|_q^{can}$, as the smallest norm $\|b\|^{can}$ among all the polynomials that are congruent to a modulo q. Namely, for $a \in \mathbb{Z}[X]/\Phi_m(X)$ we denote

$$|a|_q^{can} \stackrel{\text{def}}{=} \min\{ \|b\|^{can} : \ b \in \mathbb{Z}[X]/\Phi_m(X), \ b \equiv a \pmod{q} \}.$$

(We note that the minimum exists, even though we take it over an infinite set, since the set $\{\mathcal{E}(b) : b \equiv a \pmod{q}\}$ is a coset of a lattice.) Sometimes we may want to talk about the specific polynomial where the minimum is obtained, namely the polynomial b satisfying $b \equiv a \pmod{q}$ and $\|b\|^{can} = |a|_q^{can}$. If this polynomial is unique, then we call it the "canonical reduction mod q of a" and denote it by

$$\overset{\text{can}}{[a]}_q \ \stackrel{\text{def}}{=} \ \text{argmin}\{\ \|b\|^{can} :\ b \in \mathbb{Z}[X]/\Phi_m(X),\ b \equiv a \pmod{q}\ \}.$$

We stress that our cryptosystem never needs to compute the canonical embedding (or the canonical reduction, or the canonical norm) of polynomials, it is only in the analysis of this scheme that we use these terms.

Obviously, for any element $a \in \mathbb{Z}[X]/\Phi_m(X)$ and any modulus q, the reduced canonical embedding norm is not more than the canonical embedding norm, namely $|a|_p^{can} \le \|a\|^{can}$. Similarly, it is easy to check that if $c \equiv ab \bmod (\Phi_m(X), q)$ then $|c|_q^{can} \le |a|_q^{can} \cdot |b|_q^{can}$. A corollary of Lemma 10 (that we use in our analysis of modulus switching) is that an element with small enough canonical embedding norm must be the unique canonical reduction mod q of its coset:

Lemma 11. *Let m, q be integers, and let $a \in \mathbb{Z}[X]/\Phi(m)$ be such that $\|a\|^{can} < q/2$. Then for any $b \in \mathbb{Z}[X]/\Phi_m$ such that $b \ne a$ but $b \equiv a \bmod q$, it holds that $\|b\|^{can} \gneq \|a\|^{can} = |b|_q^{can}$. Hence for all $b \equiv a \bmod q$ we have $a = \overset{\text{can}}{[b]}_q$.*

Proof. Fix any $b \in \mathbb{Z}[X]/\Phi_m$ such that $b \ne a$ but $b \equiv a \bmod q$. Then $\frac{b-a}{q}$ is a nonzero integer polynomial, and by Lemma 10 its canonical embedding has an entry of magnitude ≥ 1. This implies that $\mathcal{E}(b)$ has an entry of distance at least q from the corresponding entry in $\mathcal{E}(a)$. Since that entry in $\mathcal{E}(a)$ has magnitude $< q/2$, then the one in $\mathcal{E}(b)$ must have magnitude $> q/2$, and therefore $\|b\|^{can} > q/2 > \|a\|^{can}$. It follows that a has the unique smallest canonical embedding norm among all the polynomials in its coset mod q. □

D.2 Our Cryptosystem

In terms of operations, our cryptosystem is almost identical to the BGV cryptosystem [3], where all the operations are done modulo $\Phi_m(X)$. However, our analysis of (the functionality of) this cryptosystem is somewhat different, in that we keep track of the canonical norm of "the noise" rather than the norm of its coefficient vector. Specifically, we maintain the invariant that if $\mathbf{c}$ is a ciphertext encrypting the aggregate plaintext $a \in \mathbb{Z}_p[X]/\Phi_m(X)$ relative to secret key $\mathbf{s}$ and modulus q, then in the ring $\mathbb{Z}_q[X]/\Phi_m(X)$ we have the equality

$$\langle \mathbf{c}, \mathbf{s} \rangle \ = \ p \cdot u \ + \ a \pmod{\Phi_m(X), q}, \tag{1}$$

where $u \in \mathbb{Z}[X]/\Phi_m(X)$ has small canonical norm mod q, $|u|_q^{can} \ll q$.

Decryption. We claim that as long as this invariant holds, we can use $\mathbf{s}$ to decrypt $\mathbf{c}$. This can be done in one of two ways:

- If the "ring constant" c_m happens to be small enough (i.e., much smaller than q), then from $\|u\|^{can} \ll q$ and $p \ll q$ and $c_m \ll q$ we conclude that also $\|p \cdot u\| \le c_m \cdot p \cdot \|u\|^{can} \ll q$, which means that the coefficient vector of the noise has small norm and decryption works just as in standard BGV cryptosystems. For example for prime values of m the constant c_m is equal to approximately $4/\pi$, [6].
- Otherwise, we "lift" decryption to work modulo $X^m - 1$ rather than modulo $\Phi_m(X)$, and use the fact that the "ring constant" of $\mathbb{Z}[X]/(X^m - 1)$ is small (namely, it is $\sqrt{m}$).

Describing the second option in more detail, Lemma 12 below tells us that there exists an integer polynomial $G \in \mathbb{Z}[X]/(X^m - 1)$ such that $G(\alpha) = m$ for every complex primitive m-th root of unity α, and $G(\beta) = 0$ for every complex non-primitive m-th root of unity β. This means in particular that $G \equiv m \pmod{\Phi_m(X)}$ (in words, the polynomial G reduces to the constant m modulo Φ_m).

Computing $b \leftarrow G \cdot \langle \mathbf{c}, \mathbf{s} \rangle \bmod (X^m - 1, q)$, we get $b = p \cdot Gu + Ga \pmod{X^m - 1, q}$, due to Equation (1). We now observe that the evaluation of the polynomial Gu in all the m-th roots of unity must be small: For the primitive roots this evaluation is only m times that of u (which is small by our invariant), and for the non-primitive roots this evaluation is zero (since G evaluates to zero in these roots). Therefore the canonical norm of Gu in $\mathbb{Z}[X]/(X^m - 1)$ is small and therefore also the norm of its coefficient vector is small, so it can be decrypted as in standard BGV cryptosystems. Namely, we have no wraparound so setting $b' \leftarrow b \bmod p$ we have $b' = Ga \in \mathbb{Z}[X](X^m - 1)$. If we now further reduce modulo $\Phi_m(X)$, $b'' \leftarrow b' \bmod \Phi_m$, we get $b = m \cdot a \in \mathbb{Z}[X]/\Phi_m(X)$ (because $G \equiv m \pmod{\Phi_m(X)}$). Finally we can multiply by $(m^{-1} \bmod p)$ to get $a = m^{-1} \cdot b'' \bmod p$.

Lemma 12. *For any integer m there is an integer polynomial G_m of degree $\leq m - 1$, such that $G_m(\alpha) = m$ for every complex primitive m-th root of unity α, and $G_m(\beta) = 0$ for every complex non-primitive m-th root of unity β. Moreover the Euclidean norm of G_m's coefficient vector is $\sqrt{m \cdot \phi(m)}$.*

Proof. Clearly there exists a complex polynomial of degree $\leq m - 1$ which evaluates to m in the primitive m-th roots of unity and to zero in the non-primitive m-th roots of unity. We only need to show that this polynomial has integer coefficients, and that it has a low-norm coefficient vector.

To show that, let D be the $m \times m$ DFT matrix (i.e., the Vandemonde matrix on complex m-th roots of unity, $D_{ij} = \rho^{ij}$ for some fixed primitive m-th root of unity ρ). Denote the coefficient vector of G by $\boldsymbol{g}$, and the vector of values that it assumes in all the m-th roots of unity by $\boldsymbol{v}$ (so $\boldsymbol{v}$ is a vector of m's and 0's), and we have $\boldsymbol{v} = D\boldsymbol{g}$. Recalling that the inverse of D is $D^{-1} = D^*/m$ (with D^* the conjugate transpose of D), and considering the 0-1 vector $\boldsymbol{v'} = \boldsymbol{v}/m$, we have that $\boldsymbol{g} = D^*\boldsymbol{v'}$. Each coefficient in G is therefore a 0-1 combination of the entries in one row of D^*, with the 1's in the positions corresponding to the primitive roots of unity. Specifically, the coefficient of x^j in G is $g_j = \sum_i (\rho^{-j})^i$, where the sum goes over all indexes $i \in \mathbb{Z}_m^*$. Since the sum is symmetric over the primitive roots of unity, then it must sum to an integer. Hence G must be an integer polynomial.

Finally, recall that the matrix D^* is orthogonal with rows of norm $\sqrt{m}$, hence the l_2 norm of $\boldsymbol{g}$ is $\sqrt{m}$ times the l_2-norm of $\boldsymbol{v'}$. Since the number of 1's in $\boldsymbol{v'}$ is exactly $\phi(m)$, then the l_2 norm of $\boldsymbol{v'}$ is $\sqrt{\phi(m)}$, and therefore the l_2 norm of $\boldsymbol{g}$ is $\sqrt{m\phi(m)}$. □

Having described decryption, we now proceed to describe all the other elements of our cryptosystem, namely key-generation, encryption, addition, "raw multiplication", key-switching, modulus switching, and Galois group actions. All these components (bar the last) are very similar to their counterpart in the BGV cryptosystem [3], but their analysis is slightly different.

Key Generation. The parameters of the scheme include the integer m (that defines the polynomial Φ_m), the integer p (that defines the aggregate plaintext space $\mathbb{Z}_p[X]/\Phi_m$), and the sequence of moduli $q_0 > q_1 > \cdots > q_L$.

Key generation is as in the ring-LWE-based version BGV [3] over the ring $\mathbb{Z}[x]/\Phi_m$. That is, for appropriate $N = \text{polylog}(q_0, m)$, one chooses $\mathfrak{s}_0, \epsilon_{0,1}, \ldots \epsilon_{0,N} \in \mathbb{Z}[X]/\Phi_m$ (with l_∞ coefficient norm $\ll q_0$) as well as a random elements $\alpha_{0,1}, \ldots, \alpha_{0,N} \in_R \mathbb{Z}_{q_0}[X]/\Phi_m$, and computes $\beta_{0,i} \leftarrow \alpha_{0,i}\mathfrak{s}_0 + p \cdot \epsilon_{0,i} \bmod (\Phi_m(X), q_0)$. The level-0 secret key is $\mathbf{s}_0 = [1, \mathfrak{s}_0]$, and the corresponding public encryption key includes the vectors $\mathbf{b}_i = [\beta_{0,i}, -\alpha_{0,i}]$.

In addition to these keys, the key-generation procedure chooses other secret key vectors for the other levels, and generates the key-switching matrices between them, as described in Section D.2 below.

Encryption. Encryption is as in BGV. An aggregate plaintext $a \in \mathbb{Z}_p[X]/\Phi_m(X)$ is encrypted by choosing random short elements $\tau_1, \ldots \tau_N \in \mathbb{Z}[X]/\Phi_m$ (with l_∞ coefficient norm $\ll q_0$) and setting

$$\mathbf{c} = [c_0, c_1] \leftarrow [a, 0] + \sum_{i=1}^{N} \tau_i \cdot \mathbf{b}_i \bmod (\Phi_m(X), q_0). \tag{2}$$

(Actually, the τ_i's can be chosen as elements of $\mathbb{Z}[x]/\Phi_m$ with 0/1 coefficients, versus merely being short.)

It is easy to show that semantic security reduces to the hardness of the decision ring-LWE problem for the ring $\mathbb{Z}_q[X]/\Phi_m$ and the distributions used to sample the short elements.

To see that our invariant holds with respect to the level-0 secret key $\mathbf{s}_0$ and freshly encrypted ciphertexts, note that Equation (2) implies that $\mathbf{c} = [a, 0] + \sum_{i=1}^{N} \tau_i \cdot \mathbf{b}_i \pmod{\Phi_m(X), q_0}$, and therefore

$$\begin{aligned} \langle \mathbf{c}, \mathbf{s}_0 \rangle = a + \sum_{i=1}^{N} \tau_i \langle \mathbf{s}_0, \mathbf{b}_i \rangle \ &= \ a + p \cdot \sum_{i=1}^{N} \tau_i \cdot \epsilon_i \\ &= a + p \cdot \sum_{i=1}^{N} \tau_i \cdot \epsilon_i \pmod{\Phi_m(X), q_0} \end{aligned}$$

and the since all the τ_i's and ϵ_i's are small (and therefore also have small canonical embedding norm), then the canonical embedding norm of the polynomial $u = \sum_{i=1}^{N} \tau_i \cdot \epsilon_i \bmod (\Phi_m(X), q_0)$ is small.

Addition. Adding two ciphertext vectors that are defined with respect to the same secret key and modulus is just standard addition in $\mathbb{Z}_q[X]/\Phi_m(X)$. Clearly, if $\langle \mathbf{c}, \mathbf{s} \rangle = p \cdot u + a$ and $\langle \mathbf{c}', \mathbf{s} \rangle = p \cdot u' + a'$ then also $\langle \mathbf{c} + \mathbf{c}', \mathbf{s} \rangle = p \cdot (u + u') + (a + a')$, and the canonical embedding norm of $u + u'$ is still small.

"Raw Multiplication". As in the BV/BGV family of cryptosystems [5, 4, 3], "raw multiplication" of two ciphertext vectors (defined with respect to the same modulus) is done using tensor product. Namely, if we have ciphertext vector $\mathbf{c}$ which is decrypted to a under $\mathbf{s}$ and q, and another vector $\mathbf{c}'$ which is decrypted to a' under $\mathbf{s}'$ and q, then we set $\tilde{\mathbf{c}} = \text{vector}(\mathbf{c} \otimes \mathbf{c}') \bmod (\Phi_m(X), q)$ (where vector$(\cdot)$ opens the matrix into a vector using some appropriate ordering). Denoting $\tilde{\mathbf{s}} = \text{vector}(\mathbf{s} \otimes \mathbf{s}') \bmod (\Phi_m(X), q)$, we thus have

$$\begin{aligned} \langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}} \rangle &= \mathbf{s}^t (\mathbf{c} \otimes \mathbf{c}') \mathbf{s}' \ = \ \langle \mathbf{c}, \mathbf{s} \rangle \cdot \langle \mathbf{c}', \mathbf{s}' \rangle \\ &= (p \cdot u + a) \cdot (p \cdot u' + a') \ = \ p \cdot (puu' + ua' + au') + aa' \pmod{\Phi_m(X), q}. \end{aligned}$$

Since the canonical embedding norm of $\tilde{u} = puu' + ua' + au' \bmod (\Phi_m(X), q)$ is still small, it means that $\tilde{\mathbf{c}}$ is a valid ciphertext with respect to $\tilde{\mathbf{s}}$ and q, which is decrypted to aa'.

Key Switching. A crucial component of the BV/BGV cryptosystems is the ability to translate a ciphertext with respect to one secret key into a ciphertext that decrypts to the same thing under another secret key. This is used, for example, to translate the "extended ciphertext" that we get from raw multiplication back to a normal ciphertext, or to translate two ciphertext vectors with respect to different keys into ciphertexts with respect to the same key, so that they can be added or raw-multiplied.

Let $\mathbf{s}$ be a secret-key vector over $\mathbb{Z}_q[X]/\Phi_m(X)$, and consider another 2-element secret-key vector $\mathbf{t} \in (\mathbb{Z}_q[X]/\Phi_m(X))^2$ whose first entry is 1. To allow translation from $\mathbf{s}$-ciphertexts to $\mathbf{t}$-ciphertexts, we first encode $\mathbf{s}$ in a redundant manner by computing $2^i \mathbf{s} \bmod q$ for $i = 0, 1, \ldots, l = \lceil \log q \rceil$ and concatenating all these vectors to form

$$\hat{\mathbf{s}} \;=\; \mathsf{Powersof2}_q(\mathbf{s}) \;\overset{\text{def}}{=}\; [\mathbf{s} \mid 2\mathbf{s} \mid 4\mathbf{s} \mid \;\ldots\; \mid 2^l\mathbf{s}] \bmod q.$$

Then we choose a random low coefficient norm vector $\mathbf{v}$ over $\mathbb{Z}_q[X]/\mathit{\Phi}_m(X)$ of the same dimension as $\hat{\mathbf{s}}$ (call this dimension d), and a matrix $R \in (\mathbb{Z}_q[X]/\phi_m)^{2\times d}$ which is chosen at random from the orthogonal space to $\mathbf{t}$, namely $\mathbf{t}R = \mathbf{0} \pmod{\mathit{\Phi}_m(X), q}$. The key-switching matrix from $\mathbf{s}$ to $\mathbf{t}$ is then set as

$$W = W[\mathbf{s} \to \mathbf{t}] = \begin{bmatrix} \hat{\mathbf{s}} + p\mathbf{v} \\ -\;0\;- \end{bmatrix} \;+\; R \mod (\mathit{\Phi}_m(X), q)$$

Again it is easy to show that if decision ring-LWE is hard for the ring $\mathbb{Z}_q[X]/\mathit{\Phi}_m(X)$ and the distributions used to sample $\mathbf{t}$ and $\mathbf{v}$, then the matrix W above is pseudo-random, even for someone who knows $\mathbf{s}$.

Given a ciphertext vector $\mathbf{c}$ (over $\mathbb{Z}_q[X]/\mathit{\Phi}_m(X)$) that satisfies our invariant with respect to $\mathbf{s}$ and q, we use W to translate it into another vector $\mathbf{c}'$ that satisfies our invariant with respect to $\mathbf{t}$ and q, as follows: First, for $i = 0, 1, \ldots, l = \lceil \log q \rceil$ we denote by $\mathbf{c}_i$ the vector over $\mathbb{Z}_2[X]/\mathit{\Phi}_m(X)$ containing the i'th bits from all the coefficients of all the entries of $\mathbf{c}$. Namely:

$$\mathbf{c}_0 = \mathbf{c} \bmod 2, \quad \text{and } \mathbf{c}_i = 2^{-i} \cdot \big((\mathbf{c} \bmod 2^{i+1}) - \sum_{j<i} 2^j \mathbf{c}_j\big) \text{ for } i > 0.$$

Then the bit-decomposition of $\mathbf{c}$ is the concatenation of all these vectors,

$$\hat{\mathbf{c}} = \mathsf{BitDecomp}(\mathbf{c}) \;\overset{\text{def}}{=}\; [\mathbf{c}_0 \mid \mathbf{c}_1 \mid \;\ldots\; \mid \mathbf{c}_l].$$

Clearly $\hat{\mathbf{c}}$ has low norm coefficient vectors, since they are all 0-1 vectors, and we have $\langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle = \langle \mathbf{c}, \mathbf{s} \rangle$ over $\mathbb{Z}_q[X]$ (and therefore also over $\mathbb{Z}_q[X]/\mathit{\Phi}_m(X)$). Switching keys from $\mathbf{s}$ to $\mathbf{t}$ is done simply by setting $\mathbf{c}' \leftarrow W\hat{\mathbf{c}} \bmod (\mathit{\Phi}_m(X), q)$. To see that this maintains our invariant, assume that for some $a \in \mathbb{Z}_p[X]/\mathit{\Phi}_m(X)$ we have $\langle \mathbf{c}, \mathbf{s} \rangle = p \cdot u + a \pmod{\mathit{\Phi}_m(X), q}$, where u has low canonical embedding norm. Then:

$$\begin{aligned} \big\langle \mathbf{c}', \mathbf{t} \big\rangle = \mathbf{t}W\hat{\mathbf{c}} \;&=\; \mathbf{t} \begin{bmatrix} \hat{s} + p\mathbf{v} \\ -\;0\;- \end{bmatrix} \hat{\mathbf{c}} \;\overset{(a)}{=}\; \langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle + p \cdot \langle \hat{\mathbf{c}}, \mathbf{v} \rangle \\ &\overset{(b)}{=}\; \langle \mathbf{c}, \mathbf{s} \rangle + p \cdot \langle \hat{\mathbf{c}}, \mathbf{v} \rangle \;=\; p \cdot \underbrace{(u + \langle \hat{\mathbf{c}}, \mathbf{v} \rangle)}_{u'} + a \pmod{\mathit{\Phi}_m(X), q}, \end{aligned}$$

where Equality (a) holds since the first entry of $\mathbf{t}$ is 1, and Equality (b) follows from $\langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle = \langle \mathbf{c}, \mathbf{s} \rangle$. Finally, since both $\mathbf{v}$ and $\hat{\mathbf{c}}$ have low canonical embedding norm (because they have low coefficient norm), then so has $\langle \hat{\mathbf{c}}, \mathbf{v} \rangle$ and therefore also $u' = \langle \hat{\mathbf{c}}, \mathbf{v} \rangle + u \bmod (\mathit{\Phi}_m(X), q)$.

Galois Group Actions. Recall that a Galois group action is obtained by applying the transformation $f(X) \mapsto f(X^i) \bmod (\mathit{\Phi}_m(X), q)$ for some $i \in \mathbb{Z}_m^*$ to all the polynomials in our ciphertext vectors, secret keys, etc. Assume that we have $\langle \mathbf{c}, \mathbf{s} \rangle = p \cdot u + a \pmod{\mathit{\Phi}_m(X), q}$, and define $\mathbf{c}^{(i)}, \mathbf{s}^{(i)}, u^{(i)}, a^{(i)}$ as what you get by applying the above Galois group action to $\mathbf{c}, \mathbf{s}, u, a$, respectively. Our invariant means that for some polynomial $k \in \mathbb{Z}_q[X]$ we have

$$\sum_j \mathbf{c}_j(X)\mathbf{s}_j(X) = p \cdot u(X) + a(X) + k(X)\mathit{\Phi}_m(X) \;\; (\text{equality in } \mathbb{Z}_q[X]), \tag{3}$$

and therefore also for every i

$$\sum_j \mathbf{c}_j(X^i)\mathbf{s}_j(X^i) = p \cdot u(X^i) + a(X^i) + k(X^i)\mathit{\Phi}_m(X^i) \;\; (\text{equality in } \mathbb{Z}_q[X]). \tag{4}$$

Equation (4) follows since the two sides of Equation (3) are identical as formal polynomials over $\mathbb{Z}_q$, and therefore they must coincide also as functions over any characteristic-q field. It follows that the functions on both sides of Equation (4) must also coincide over any characteristic-q field, and therefore the two sides must be identical as formal polynomials over $\mathbb{Z}_q$.

Recalling that if $i \in \mathbb{Z}_m^*$ then $\Phi(X)$ divides $\Phi(X^i)$, we obtain

$$\left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle = \sum_j \mathbf{c}_j(X^i)\mathbf{s}_j(X^i) = p \cdot u(X^i) + a(X^i) = p \cdot u^{(i)} + a^{(i)} \pmod{\Phi_m(X), q},$$

as needed. Observing that for $i \in \mathbb{Z}_m^*$ the canonical embeddings of u and $u^{(i)}$ are just a permutation of each other (and hence have the same norm) we deduce that our invariant in maintained under the transformation $X \mapsto X^i$ whenever $i \in \mathbb{Z}_m^*$.

Modulus Switching. Our modulus switching procedure works exactly as in the BGV cryptosystem. Namely, to switch a ciphertext $\mathbf{c}$ (in coefficient representation) from q_i to q_{i+1}, we just scale the coefficient vectors in $\mathbf{c}$ by a q_{i+1}/q_i factor, and then round the result to get an integer polynomial vector $\mathbf{c}'$ such that $\mathbf{c}' \equiv \mathbf{c} \pmod{p}$.

Definition 3 (Scale)**.** *For a vector* $\mathbf{c}$ *over* $\mathbb{Z}[X]/\Phi_m(X)$ *and integers* $q_i > q_{i+1} > p$*, define* $\mathbf{c}' \leftarrow \mathsf{Scale}(\mathbf{c}, q_i, q_{i+1}, p)$ *to be the vector over* $\mathbb{Z}[X]/\Phi_m(X)$ *closest to* $(p/q) \cdot \mathbf{c}$ *(in coefficient representation) that satisfies* $\mathbf{c}' \equiv \mathbf{c} \pmod{p}$*.*

Our analysis, however, is a little different than in [3]. The proof from [3, Lemma 4] relies on the fact that the coefficient vector of $[\langle \mathbf{c}, \mathbf{s} \rangle]_{q_i}$ has low norm, whereas in out case we instead have that this polynomials has low canonical embedding norm mod q_i. We therefore re-prove this lemma under our new condition.

Lemma 13. *Let* $q_i > q_{i+1} > p$ *be positive integers satisfying* $q_i = q_{i+1} = 1 \pmod{p}$*. Let* $\mathbf{c}, \mathbf{s}$ *be two* n*-vectors over* $\mathbb{Z}[X]/\Phi_m(X)$ *such that* $|\langle \mathbf{c}, \mathbf{s} \rangle|_{q_i}^{can} < q_i/2 - \frac{q_i}{q_{i+1}} \cdot pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can}$*, and let* $\mathbf{c}' = \mathsf{Scale}(\mathbf{c}, q_i, q_{i+1}, p)$*. Denoting* $e = \langle \mathbf{c}, \mathbf{s} \rangle \bmod \Phi_m(X)$ *and* $e' = \langle \mathbf{c}', \mathbf{s} \rangle \bmod \Phi_m(X)$ *(arithmetic in* $\mathbb{Z}[X]/\Phi_m(X)$*), it holds that*

$$\begin{aligned} \overset{\text{can}}{[e']}_{q_{i+1}} &\equiv \overset{\text{can}}{[e]}_{q_i} \pmod{p} \text{ (in coefficient representation), and} \\ |e'|_{q_{i+1}}^{can} &< \frac{q_{i+1}}{q_i} \cdot |e|_{q_i}^{can} + pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can} \end{aligned}$$

Proof. For some $k \in \mathbb{Z}[X]/\Phi_m(X)$, we have $\overset{\text{can}}{[e]}_{q_i} = \langle \mathbf{c}, \mathbf{s} \rangle - q_i k$, where the equality is over $\mathbb{Z}[X]/\Phi_m(X)$. For the same k, let $e'' = e' - q_{i+1}k \in \mathbb{Z}[X]/\Phi_m(X)$. Since $\mathbf{c}' \equiv \mathbf{c} \pmod{p}$ and $q_i \equiv q_{i+1} \pmod{p}$, then also

$$e'' = \langle \mathbf{c}', \mathbf{s} \rangle - q_{i+1}k \equiv \langle \mathbf{c}, \mathbf{s} \rangle - q_i k = \overset{\text{can}}{[e]}_{q_i} \pmod{\Phi_m(X), p}.$$

It therefore suffices to prove that $e'' = \overset{\text{can}}{[e']}_{q_{i+1}}$ (equality over $\mathbb{Z}[X]/\Phi_m(X)$) and that it has small enough norm.

Denote the distance between $\frac{q_{i+1}}{q_i} \cdot \mathbf{c}$ and its rounded version $\mathbf{c}'$ by $\delta \overset{\text{def}}{=} \mathbf{c}' - \frac{q_{i+1}}{q_i}\mathbf{c}$. Then δ is a vector over $\mathbb{Q}[X]/\Phi_m(X)$, and the coefficient-vectors in δ all have entries in $[-p/2, p/2)$. Moreover, we have

$$\begin{aligned} e'' = \langle \mathbf{c}', \mathbf{s} \rangle - q_{i+1}k &= \frac{q_{i+1}}{q_i} \langle \mathbf{c}, \mathbf{s} \rangle + \langle \delta, \mathbf{s} \rangle - q_{i+1}k \\ &= \frac{q_{i+1}}{q_i} \big(\langle \mathbf{c}, \mathbf{s} \rangle - q_i k \big) + \langle \delta, \mathbf{s} \rangle = \frac{q_{i+1}}{q_i} \cdot \overset{\text{can}}{[e]}_{q_i} + \langle \delta, \mathbf{s} \rangle . \end{aligned} \tag{5}$$

Considering the polynomial $\langle \delta, \mathbf{s} \rangle \in \mathbb{Q}[X]/\Phi_m(X)$, we can bound its canonical embedding norm by:

$$\|\, \langle \delta, \mathbf{s} \rangle \,\|^{can} \;\le\; n \cdot \|\delta\|^{can} \cdot \|\mathbf{s}\|^{can} \;\le\; n \cdot \phi(m) \cdot \|\delta\| \cdot \|\mathbf{s}\|^{can} \;\le\; pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can}.$$

From Equation (5) we now get:

$$\begin{aligned} \|e''\|^{can} \;&\le\; \frac{q_{i+1}}{q_i} \cdot |e|^{can}_{q_i} + \|\, \langle \delta, \mathbf{s} \rangle \,\|^{can} \;\le\; \frac{q_{i+1}}{q_i} \cdot |e|^{can}_{q_i} + pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can} \\ &<\; \left(\frac{q_{i+1}}{2} - pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can} \right) + pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can} \;=\; \frac{q_{i+1}}{2} \end{aligned} \tag{6}$$

Finally, Lemma 11 implies that $e'' \overset{\text{can}}{=} [e']_{q_{i+1}}$, completing the proof. □

It follows immediately from Lemma 13 that if $\mathbf{c}$ satisfies our invariant with respect to $\mathbf{s}$ and q_i, and if the canonical embedding norm of $\mathbf{s}$ is small enough so that we have $|\, \langle \mathbf{c}, \mathbf{s} \rangle \,|^{can}_{q_i} < q_i/2 - \frac{q_i}{q_{i+1}} \cdot pn \cdot \phi(m) \cdot \|\mathbf{s}\|^{can}$, then the scaled vector $\mathbf{c}' = \mathsf{Scale}(\mathbf{c}, q_i, q_{i+1}, p)$ satisfies our invariant with respect to the same $\mathbf{s}$ and the new modulus q_{i+1}.

Variants. We note that one can optimize BGV key generation and encryption using a cute trick by Brakerski and Vaikuntanathan [5] (following [15]). This reduces the public key size and encryption time, without changing the scheme in an any way that affects the applicability of our techniques; we still obtain FHE with polylog overhead using BGV with BV's optimizations. (We note that our techniques can be applied to the cryptosystem of BV [5] as well, but one needs to use BGV's noise management technique to reduce the overhead to polylog.)

In BV key generation [5], for level-0, one only needs to choose low-norm elements $\mathfrak{s}_0, \epsilon_0 \in \mathbb{Z}[X]/\Phi_m(X)$ (with coefficient norm $\ll q_L$) as well as a random element $\alpha_0 \in_R \mathbb{Z}_{q_0}[X]/\Phi_m(X)$, and computing $\beta_0 \leftarrow -\alpha_0 \mathfrak{s}_0 + p \cdot \epsilon_0 \bmod (\Phi_m(X), q_0)$. The level-0 secret key is $\mathbf{s}_0 = [1, \mathfrak{s}_0]$, and the corresponding public encryption key is $\mathbf{b} = [\beta_0, \alpha_0]$. This approach reduces level-0 key size by factor of $O(\log q_0)$. One generates keys for the other levels similarly.

In BV encryption, an aggregate plaintext $a \in \mathbb{Z}_p[X]/\Phi_m(X)$ is encrypted by choosing three random short elements $\tau, \epsilon_1, \epsilon_2 \in \mathbb{Z}_{q_0}[X]/\Phi_m(X)$ and setting

$$\mathbf{c} = [c_0, c_1] \leftarrow [\tau\beta_0, \tau\alpha_0] + p \cdot [\epsilon_1, \epsilon_2] + [a, 0] \bmod (\Phi_m(X), q_0). \tag{7}$$

It is easy to show that semantic security reduces to the hardness of the decision ring-LWE problem for the ring $\mathbb{Z}_q[X]/\Phi_m(X)$ and the distributions used to sample $\mathfrak{s}_0, \tau$, and $\epsilon, \epsilon_1, \epsilon_2$.

To see that our invariant holds with respect to the level-0 secret key $\mathbf{s}_0$ and freshly encrypted ciphertexts, note that Equation (7) implies that $\mathbf{c} = [\tau\beta_0, \tau\alpha_0] + p \cdot [\epsilon_1, \epsilon_2] + [a, 0] \pmod{\Phi_m(X), q_0}$, and therefore

$$\begin{aligned} \langle \mathbf{c}, \mathbf{s}_0 \rangle &= \tau\beta_0 + p\epsilon_1 + a + \mathfrak{s}(\tau\alpha_0 + p\epsilon_2) \;=\; -\tau\mathfrak{s}\alpha_0 + p\tau\epsilon_0 + p\epsilon_1 + a + \mathfrak{s}(\tau\alpha_0 + p\epsilon_2) \\ &= p \cdot (\tau\epsilon_0 + \epsilon_1 + \mathfrak{s}\epsilon_2) + a \pmod{\Phi_m(X), q_0} \end{aligned}$$

and the polynomial $u = (\tau\epsilon_0 + \epsilon_1 + \mathfrak{s}\epsilon_2) \bmod (X^m - 1, q_0)$ has low coefficient norm, and therefore also low canonical embedding norm. When using BV encryption and key generation, the other aspects of the scheme remain the same.

E A Delayed-Reduction Technique

We describe here another variant, where we work with polynomials modulo $X^m - 1$ rather than polynomials modulo Φ_m, and reduce back mod Φ_m only upon decryption. Importantly, we still want to base our security on the hardness of ring-LWE with respect to the ring $\mathbb{Z}_q[X]/\Phi_m(X)$ (recall that decision ring-LWE is easy modulo $X^m - 1$, since it can be reduced to the one-dimensional problem modulo $X - 1$).

We can use Lemma 12 to "lift" the mod-$\Phi_m(x)$ polynomials in the cryptosystem into mod-$(X^m - 1)$ polynomials, simply by multiplying by the polynomial $G(X)$ from that lemma. (This has the effect of introducing an extra multiplicative factor of m, which we can correct upon decryption.) Note that since $G = 0 \pmod{\frac{X^m-1}{\Phi_m(x)}}$, then we can write $G(X) = \frac{X^m-1}{\Phi_m(x)} \cdot \mu(X)$ (equality over $\mathbb{Z}[X]$) for some integer polynomial μ. It follows that if we have two polynomials satisfying $u = v \pmod{\Phi_m}$ then $Gu = Gv \pmod{X^m - 1}$. This is because over $\mathbb{Z}[X]/(X^m - 1)$ we have $u = v + \tau\Phi_m$ for some integer polynomial τ, and so

$$Gu \;=\; G(v + \tau\Phi_m) \;=\; Gu + (\frac{X^m - 1}{\Phi_m}\mu) \cdot \tau\Phi_m \;=\; Gu + (X^m - 1) \cdot \mu\tau \;=\; Gu \pmod{X^m - 1}$$

In our variant of the BGV cryptosystem, ciphertexts are vectors over the ring $\mathbb{Z}[X]/(X^m - 1)$, secret keys are vectors over the sub-ring $\mathbb{Z}[X]/\Phi_m$, and aggregate plaintexts are elements in $\mathbb{Z}_p[X]/\Phi_m$. We maintain the invariant that if $\mathbf{c}$ is a ciphertext encrypting the aggregate plaintext a relative to secret key $\mathbf{s}$ and modulus q, then in the ring $\mathbb{Z}_q[X]/(X^m - 1)$ we have the equality

$$G \cdot \langle \mathbf{c}, \mathbf{s} \rangle \;=\; p \cdot G \cdot u \;+\; G \cdot a \pmod{X^m - 1, q}, \tag{8}$$

where $u \in \mathbb{Z}[X]/(X^m - 1)$ has coefficient vector with small l_2-norm, $\|u\|_2 \ll q$. Note that we can use $\mathbf{s}$ to decrypt $\mathbf{c}$ by setting $b \leftarrow G \cdot \langle \mathbf{c}, \mathbf{s} \rangle \bmod (X^m - 1, q)$, then recovering $a = m^{-1} \cdot b \bmod (\Phi_m, p)$. Since both b and $p \cdot Gu + Ga \pmod{X^m - 1}$ have coefficients smaller than $q/2$ in absolute value, then we have the equality $b = p \cdot Gu + Ga$ holding over $\mathbb{Z}[X]/(X^m - 1)$, without reduction modulo q. We thus have $b = Ga \pmod{X^m - 1, p}$, so also $b = Ga = m \cdot a \pmod{\Phi_m, p}$, so indeed $a = b \cdot m^{-1} \pmod{\Phi_m, p}$.

Having described decryption, we now proceed to describe all the other elements of our cryptosystem, namely key-generation, encryption, addition, "raw multiplication", key-switching, modulus switching, and Galois group actions. All these components (bar the last) are very similar to their counterpart in the BGV cryptosystem [3], except that we use some mix of mod-Φ_m and mod-$(X^m{-}1)$ arithmetic, using multiplication-by-G and Equation (8) to move between them.

E.1 Key generation

The parameters of the scheme include the integer m (that defines the polynomials Φ_m and $X^m - 1$), the integer p (that defines the aggregate plaintext space $\mathbb{Z}_p[X]/\Phi_m$), and the sequence of moduli $q_0 > q_1 > \cdots > q_L$.

Key generation is as in the ring-LWE-based version BGV [3] over the ring $\mathbb{Z}[x]/\Phi_m$. That is, for appropriate $N = \text{polylog}(q_0, m)$, one chooses low-norm elements $\mathfrak{s}_0, \epsilon_{0,1}, \ldots \epsilon_{0,N} \in \mathbb{Z}[X]/\Phi_m$ (with l_2 norm $\ll q_0$) as well as a random elements $\alpha_{0,1}, \ldots, \alpha_{0,N} \in_R \mathbb{Z}_{q_0}[X]/\Phi_m$, and computes $\beta_{0,i} \leftarrow \alpha_{0,i}\mathfrak{s}_0 + p \cdot \epsilon_{0,i} \bmod (\Phi_m, q_0)$. The level-0 secret key is $\mathbf{s}_0 = [1, \mathfrak{s}_0]$, and the corresponding public encryption key includes the vectors $\mathbf{b}_i = [\beta_{0,i}, -\alpha_{0,i}]$.

In addition to these keys, the key-generation procedure chooses other secret key vectors for the other levels, and generates the key-switching matrices between them, as described in Section E.5 below.

E.2 Encryption

Encryption is as in BGV. An aggregate plaintext $a \in \mathbb{Z}_p[X]/\Phi_m(X)$ is encrypted by choosing random short elements $\tau_1, \dots \tau_N \in \mathbb{Z}[X]/\Phi_m$ and setting

$$\mathbf{c} = [c_0, c_1] \leftarrow [a, 0] + \sum_{i=1}^{N} \tau_i \cdot \mathbf{b}_i \bmod (\Phi_m, q_0). \tag{9}$$

(Actually, the τ_i's can be chosen as elements of $\mathbb{Z}[x]/\Phi_m$ with 0/1 coefficients, versus merely being short.)

Note that freshly encrypted ciphertexts are vectors over the sub-ring $\mathbb{Z}[X]/\Phi_m(X)$, but later we allow evaluated ciphertexts to be in the larger ring $\mathbb{Z}[X]/(X^m - 1)$. It is easy to show that semantic security reduces to the hardness of the decision ring-LWE problem for the ring $\mathbb{Z}_q[X]/\Phi_m$ and the distributions used to sample the short elements.

To see that our invariant holds with respect to the level-0 secret key $\mathbf{s}_0$ and freshly encrypted ciphertexts, note that Equation (9) implies that $G \cdot \mathbf{c} = G([a, 0] + \sum_{i=1}^{N} \tau_i \cdot \mathbf{b}_i) \pmod{X^m - 1, q_0}$, and therefore

$$\begin{aligned} G \cdot \langle \mathbf{c}, \mathbf{s}_0 \rangle &= G(a + \sum_{i=1}^{N} \tau_i \langle \mathbf{s}_0, \mathbf{b}_i \rangle) \\ &= G(a + p \cdot \sum_{i=1}^{N} \tau_i \cdot \epsilon_i) \\ &= Ga + p \cdot G(\sum_{i=1}^{N} \tau_i \cdot \epsilon_i) \pmod{X^m - 1, q_0} \end{aligned}$$

and the coefficient vector of the polynomial $u = \sum_{i=1}^{N} \tau_i \cdot \epsilon_i \bmod (X^m - 1, q_0)$ has low l_2 norm.

We stress that the low l_2 norm of u depends crucially on our delayed reduction. Indeed, each of the polynomials $\{\tau_i\}, \{\epsilon_i\}, G$ has low l_2 norm, hence their products and sums over $\mathbb{Z}[X]$ would still have low norms. However, we do not know how to prove that the norm remains low when we reduce them modulo Φ_m, it is only because we reduce modulo $X^m - 1$ that we can argue that the norm remains low.

E.3 Addition

Adding two ciphertext vectors that are defined with respect to the same secret key and modulus is just standard addition in $\mathbb{Z}_q[X]/(X^m - 1)$. Indeed, if we have $G \cdot \langle \mathbf{c}, \mathbf{s} \rangle = p \cdot Gu + Ga$ and $G \cdot \langle \mathbf{c}', \mathbf{s} \rangle = p \cdot Gu' + Ga'$ (both over $\mathbb{Z}_q[X]/(X^m - 1)$) then also $G \cdot \langle \mathbf{c} + \mathbf{c}', \mathbf{s} \rangle = p \cdot G(u + u') + G(a + a')$, and the l_2 norm of the coefficient vector of $u + u'$ is still small.

E.4 "Raw multiplication"

As in the BV/BGV family of cryptosystems [5, 4, 3], "raw multiplication" of two ciphertext vectors (defined with respect to the same secret key and modulus) is done using tensor product. Namely, if we have ciphertext vector $\mathbf{c}$ which is decrypted to a under $\mathbf{s}$ and q, and another vector $\mathbf{c}'$ which is decrypted to a' under $\mathbf{s}$ and q, then we set $\tilde{\mathbf{c}} = \text{vector}(\mathbf{c} \otimes \mathbf{c}') \bmod (X^m - 1, q)$ (where vector$(\cdot)$ opens the matrix into a vector using some appropriate ordering). Denoting $\tilde{\mathbf{s}} = \text{vector}(\mathbf{s} \otimes \mathbf{s}) \bmod (\Phi_m, q)$, we thus have

$$\begin{aligned} G \cdot \langle \tilde{\mathbf{c}}, \tilde{\mathbf{s}} \rangle &= G \cdot \mathbf{s}^t (\mathbf{c} \otimes \mathbf{c}') \mathbf{s} \;=\; G \cdot \langle \mathbf{c}, \mathbf{s} \rangle \cdot \big\langle \mathbf{c}', \mathbf{s} \big\rangle \\ &= (p \cdot Gu + Ga) \cdot \big\langle \mathbf{c}', \mathbf{s} \big\rangle \;=\; (p \cdot u + a) \cdot G \cdot \big\langle \mathbf{c}', \mathbf{s} \big\rangle \;=\; (p \cdot u + a) \cdot (p \cdot Gu' + Ga') \\ &= p \cdot G(puu' + ua' + au') + Gaa' \quad (\bmod\ X^m - 1, q). \end{aligned}$$

Since the coefficient vector of $\tilde{u} = puu' + ua' + au' \bmod (X^m - 1, q)$ still has small l_2 norm, it means that $\tilde{\mathbf{c}}$ is a valid ciphertext with respect to $\tilde{\mathbf{s}}$ and q, which is decrypted to aa'. Note that above we used mod-$(X^m - 1)$ arithmetic for the ciphertext and mod-Φ_m arithmetic for the secret key. This choice was made for convenience in other operations.

E.5 Key switching

A crucial component of the BV/BGV cryptosystems is the ability to translate a ciphertext with respect to one secret key into a ciphertext that decrypts to the same thing under another secret key. This is used, for example, to translate the "extended ciphertext" that we get from raw-multiplication back to a normal ciphertext, or to translate two ciphertext vectors with respect to different keys into ciphertexts with respect to the same key, so that they can be added or raw-multiplied.

Let $\mathbf{s}$ be a secret-key vector over $\mathbb{Z}_q[X]/\Phi_m$, and consider another 2-element secret-key vector $\mathbf{t} \in (\mathbb{Z}_q[X]/\Phi_m)^2$ whose first entry is 1. To allow translation from $\mathbf{s}$-ciphertexts to $\mathbf{t}$-ciphertexts, we first encode $\mathbf{s}$ in a redundant manner by computing $2^i \mathbf{s} \bmod q$ for $i = 0, 1, \ldots, l = \lceil \log q \rceil$ and concatenating all these vectors to form

$$\hat{\mathbf{s}} \;=\; \mathsf{Powersof2}_q(\mathbf{s}) \;\stackrel{\text{def}}{=}\; [\mathbf{s} \mid 2\mathbf{s} \mid 4\mathbf{s} \mid \ \ldots\ \mid 2^l \mathbf{s}] \bmod q.$$

Then we choose a random low l_2 norm vector $\mathbf{v}$ over $\mathbb{Z}_q[X]/\Phi_m$ of the same dimension as $\hat{\mathbf{s}}$ (call this dimension d), and a matrix $R \in (\mathbb{Z}_q[X]/\phi_m)^{2 \times d}$ which is chosen at random from the orthogonal space to $\mathbf{t}$, namely $\mathbf{t}R = \mathbf{0}$ $(\bmod\ \Phi_m, q)$. The key-switching matrix from $\mathbf{s}$ to $\mathbf{t}$ is then set as

$$W = W[\mathbf{s} \to \mathbf{t}] = \begin{bmatrix} \hat{\mathbf{s}} + p\mathbf{v} \\ -\ 0\ - \end{bmatrix} \;+\; R \quad \bmod (\Phi_m, q)$$

Again it is easy to show that if decision ring-LWE is hard for the ring $\mathbb{Z}_q[X]/\Phi_m(X)$ and the distributions used to sample $\mathbf{t}$ and $\mathbf{v}$, then the matrix W above is pseudo-random, even for someone who knows $\mathbf{s}$.

Given a ciphertext vector $\mathbf{c}$ (over $\mathbb{Z}_q[X]/(X^m - 1)$) that satisfies our invariant with respect to $\mathbf{s}$ and q, we use W to translate it into another vector $\mathbf{c}'$ that satisfies our invariant with respect to $\mathbf{t}$ and q, as follows: First, for $i = 0, 1, \ldots, l = \lceil \log q \rceil$ we denote by $\mathbf{c}_i$ the vector over $\mathbb{Z}_2[X]/(X^m - 1)$ containing the i'th bits from all the coefficients of all the entries of $\mathbf{c}$. Namely:

$$\mathbf{c}_0 = \mathbf{c} \bmod 2, \quad \text{and } \mathbf{c}_i = 2^{-i} \cdot \big((\mathbf{c} \bmod 2^{i+1}) - \sum_{j<i} 2^j \mathbf{c}_j \big) \text{ for } i > 0.$$

Then the bit-decomposition of $\mathbf{c}$ is the concatenation of all these vectors,

$$\hat{\mathbf{c}} = \mathsf{BitDecomp}(\mathbf{c}) \;\stackrel{\text{def}}{=}\; [\mathbf{c}_0 \mid \mathbf{c}_1 \mid \ \ldots\ \mid \mathbf{c}_l].$$

Clearly $\hat{\mathbf{c}}$ has low l_2 norm, since it is represented by a 0-1 vector, and we have $\langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle = \langle \mathbf{c}, \mathbf{s} \rangle$ over $\mathbb{Z}_q[X]$ (and therefore also over $\mathbb{Z}_q[X]/(X^m - 1)$). Switching keys from $\mathbf{s}$ to $\mathbf{t}$ is done simply by setting $\mathbf{c}' \leftarrow W\hat{\mathbf{c}} \bmod (X^m -$

$1, q)$. To see that this maintains our invariant, assume that for some $a \in \mathbb{Z}_p[X]/\varPhi_m$ we have $G \cdot \langle \mathbf{c}, \mathbf{s} \rangle = p \cdot Gu + Ga$ $\pmod{X^m - 1, q}$, where the coefficient vector of u has low l_2 norm. Then:

$$\begin{aligned} G \cdot \left\langle \mathbf{c}', \mathbf{t} \right\rangle \;=\; G \cdot \mathbf{t} W \hat{\mathbf{c}} \;&\overset{(a)}{=}\; G \cdot \mathbf{t} \begin{bmatrix} \hat{s} + p\mathbf{v} \\ -\ 0\ - \end{bmatrix} \hat{\mathbf{c}} \;\overset{(b)}{=}\; G \cdot \langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle + p \cdot G \cdot \langle \hat{\mathbf{c}}, \mathbf{v} \rangle \\ &\overset{(c)}{=}\; G \cdot \langle \mathbf{c}, \mathbf{s} \rangle + p \cdot G \cdot \langle \hat{\mathbf{c}}, \mathbf{v} \rangle \;=\; p \cdot G \underbrace{(u + \langle \hat{\mathbf{c}}, \mathbf{v} \rangle)}_{u'} + Ga \pmod{X^m - 1, q}, \end{aligned}$$

where Equality (a) follows since $\mathbf{t}R = \mathbf{0} \pmod{\varPhi_m, q}$ and therefore $G \cdot \mathbf{t}R = \mathbf{0} \pmod{X^m - 1, q}$, Equality (b) holds since the first entry of $\mathbf{t}$ is 1, and Equality (c) follows from $\langle \hat{\mathbf{c}}, \hat{\mathbf{s}} \rangle = \langle \mathbf{c}, \mathbf{s} \rangle$. Finally, since both $\mathbf{v}$ and $\hat{\mathbf{c}}$ have low l_2 norm, then over $\mathbb{Z}_q[X]/(X^m - 1)$ so has $\langle \hat{\mathbf{c}}, \mathbf{v} \rangle$ and therefore also $u' = \langle \hat{\mathbf{c}}, \mathbf{v} \rangle + u \bmod (X^m - 1, q)$.

E.6 Modulus switching

The modulus-switching procedure is exactly as in the BGV cryptosystem. Note that this procedure does not involve any mod-$\varPhi_m$ or mod-$(X^m - 1)$ arithmetic: All we do is take a ciphertext vector $\mathbf{c}$ over $\mathbb{Z}_{q_i}[X]/(X^m - 1)$, scale it down by a factor q_{i+1}/q_i and round to get $\mathbf{c}' = \text{round}_{\mathbf{c}}(\frac{q_{i+1}}{q_i} \cdot \mathbf{c})$ such that $\mathbf{c}' \equiv \mathbf{c} \pmod{p}$. The reason that this works in our case is exactly as in BGV, our delayed reduction has no effect here.

E.7 Galois group actions

As described in Section 4.2, applying the action $X \to X^i$ on a ciphertext vector $\mathbf{c}$ over $\mathbb{Z}_q[X]/(X^m - 1)$ requires only a permutation of the coefficients in each of the elements of $\mathbf{c}$ (all which are degree-$(m - 1)$ polynomials over $\mathbb{Z}_q$).

Assume that we have $G \cdot \langle \mathbf{c}, \mathbf{s} \rangle = p \cdot Gu + Ga \pmod{X^m - 1, q}$, and define $\mathbf{c}^{(i)}, u^{(i)}$ as what you get by applying the transformation $X \to X^i$ to $\mathbf{c}, u$, respectively, over $\mathbb{Z}_q[X]/(X^m - 1)$, and $\mathbf{s}^{(i)}, a^{(i)}$ as what you get by applying the transformation $X \to X^i$ to $\mathbf{s}, a$, respectively over $\mathbb{Z}_q[X]/\varPhi_m$. Below we prove that if i, m are co-prime and also q, m are co-prime, then we have $G \cdot \left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle = p \cdot Gu^{(i)} + Ga^{(i)} \pmod{X^m - 1, q}$.

Using $G = m \pmod{\varPhi_m}$, and reducing modulo $\varPhi_m$ the equality $G \cdot \langle \mathbf{c}, \mathbf{s} \rangle = p \cdot Gu + Ga$, we have $m \cdot \langle \mathbf{c}, \mathbf{s} \rangle = pm \cdot u + ma \pmod{\varPhi_m, q}$. Since m, q are co-prime then multiplying by $m^{-1} \pmod{q}$ we get $\langle \mathbf{c}, \mathbf{s} \rangle = p \cdot u + a$ $\pmod{\varPhi_m, q}$. Namely, for some polynomial $k \in \mathbb{Z}_q[X]$ we have

$$\sum_j \mathbf{c}_j(X) \mathbf{s}_j(X) = p \cdot u(X) + a(X) + k(X) \varPhi_m(X) \ \ \text{(equality in } \mathbb{Z}_q[X]\text{)}, \tag{10}$$

and therefore also for every i

$$\sum_j \mathbf{c}_j(X^i) \mathbf{s}_j(X^i) = p \cdot u(X^i) + a(X^i) + k(X^i) \varPhi_m(X^i) \ \ \text{(equality in } \mathbb{Z}_q[X]\text{)}. \tag{11}$$

Equation (11) follows since the two sides of Equation (10) are identical as formal polynomials over $\mathbb{Z}_q$, and therefore they must coincide also as functions over any characteristic-q field. It follows that the functions on both sides of Equation (11) must also coincide over any characteristic-q field, and therefore the two sides must be identical as formal polynomials over $\mathbb{Z}_q$.

Recalling that if $i \in Z_m^*$ then $\varPhi(X)$ divides $\varPhi(X^i)$, we obtain

$$\left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle = \sum_j \mathbf{c}_j(X^i) \mathbf{s}_j(X^i) = p \cdot u(X^i) + a(X^i) = p \cdot u^{(i)} + a^{(i)} \pmod{\varPhi_m(X), q}.$$

Now we can multiply by G to "lift" the equality over to $\mathbb{Z}_q[X]/(X^m - 1)$ and we get

$$G \cdot \left\langle \mathbf{c}^{(i)}, \mathbf{s}^{(i)} \right\rangle = p \cdot Gu^{(i)} + Ga^{(i)} \pmod{X^m - 1, q},$$

as needed. Observing that over $\mathbb{Z}_q[X]/(X^m - 1)$ the coefficient vectors of u and $u^{(i)}$ are just a permutation of each other (and hence have the same l_2 norm) we deduce that our invariant in maintained under the transformation $X \mapsto X^i$ whenever $i \in \mathbb{Z}_m^*$ and $m \in \mathbb{Z}_q^*$.