Fully Homomorphic Encryption with Polylog Overhead

Craig Gentry[1], Shai Halevi[1], and Nigel 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 $t \cdot \mathrm{polylog}(\lambda)$.

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.

1 Introduction

Fully homomorphic encryption (FHE) [1–3] 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 – $\mathsf{Add}, \mathsf{Mult}$ and $\mathsf{Permute}$ – that allows us to evaluate arbitrary circuits *while keeping the ciphertexts packed*. Batch Add and Mult have been done before [4], and follow easily from the Chinese Remainder Theorem within our underlying polynomial ring. Here we introduce the operation $\mathsf{Permute}$, that allows us to

D. Pointcheval and T. Johansson (Eds.): EUROCRYPT 2012, LNCS 7237, pp. 465–482, 2012.
© International Association for Cryptologic Research 2012

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 [4, 5] 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. [6] 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 $\mathsf{Add}, \mathsf{Mult}$ and $\mathsf{Permute}$ operations can all be accomplished with $\tilde{O}(\lambda)$ computation by building on the recent Brakerski-Gentry-Vaikuntanathan (BGV) "FHE without bootstrapping" scheme [5], which builds on prior work by Brakerski and Vaikuntanathan and others [7–9]. 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 $\tilde{O}(\lambda^{3.5})$ [10] until BGV recently reduced it to $\tilde{O}(\lambda)$ [5].[1]

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 the full version of this paper [12].

1.1 Packing Plaintexts and Batched Homomorphic Computation

Smart and Vercauteren [4, 13] 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 [14]) homomorphic operations. That is, they show how to evaluate a function f

[1] 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, 11].

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 [5]) 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[2] [2, 3]), 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 [5] observed that for certain parameter settings, one can use *automorphisms* associated with the algebraic ring $\mathbb{A}$

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

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, 7, 8]. This lets us devise low-overhead homomorphic schemes for arithmetic circuits over essentially any small finite field $\mathbb{F}_{p^n}$.

Our efficient implementation of Permute, described in Section 3, uses the Beneš/Waksman permutation network [15, 16]. 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 [5], 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 [17] 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 [5] depend polynomially on the depth of the circuit being evaluated, because the modulus size must grow with the depth of the circuit (unless bootstrapping [2, 3] 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 [5], 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 [4, 5] how to homomorphically evaluate batch addition and multiplication operations on ℓ-vectors:

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

on packed ciphertexts in time $\tilde{O}((\ell+\lambda)(\log|\mathbb{K}_n|)$ where λ is the security parameter (with addition and multiplication in $\mathbb{K}_n$).[3] 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-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-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.

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

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-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-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 [5], then the total overhead is polylogarithmic in the security parameter.

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. [18]).

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.*

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 $I_{n_i} = \{0, \ldots, n_i - 1\}$. (Each element in S has k coordinates.) For any permutation π over S, there are permutations $\pi_1, \ldots, \pi_{2k-1}$ such that $\pi = \pi_{2k-1} \circ \cdots \circ \pi_1$ and such that π_i affects only the i-th coordinate for $i \leq k$ and only the $(2k-i)$-th coordinate for $i \geq k$.*

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 [15, 16]. 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 $i = 1, 2, \ldots, 2r-1$, 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. [19, 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

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

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 lemma, whose proof is in the full version [12].

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 $O(k \cdot \ell \cdot \log(k \cdot \ell))$.*

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-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 $w' \stackrel{\text{def}}{=} m_1 + m_2 + \cdots + m_w \geq w$. 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 the full version [12].

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 the full version, 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 the full version for details [12].

Input: k ℓ-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' ℓ-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

0. Set $M \leftarrow \max_i\{m_i\}$
1. For $j = 1$ to $\lceil \log M \rceil$ // The j'th phase
2. Make another copy of all the arrays // Duplicate everything
3. While there are values v_i with multiplicity more than m_i:
4. Replace the excess occurrences of v_i by $\perp$ // Remove redundant entries
5. While there exist pairs of arrays that have between them ℓ or more slots with $\perp$:
6. Pick one such pair and merge the two arrays //Merge sparse arrays
7. Output the remaining arrays

Fig. 1. The cloning procedure

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 the full version of this paper.

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 Section 4)

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, it is 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 [5] that we use.

Smart and Vercauteren [4] 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 the full version of this paper. 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 the full version 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 the full version of this paper [12].

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 $\mathbf{s}^{(i)}$! Then using key-switching we can get an encryption of $a^{(i)}$ back under $\mathbf{s}$ (or any other key). See the full version for more details [12].

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 the full version [12].

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 $\mathbf{s}$, then $\mathbf{c}^{(i)}$ decrypts to $a^{(i)}$ under $\mathbf{s}^{(i)}$. 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[4] 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 the full version 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 the full version we also show how to emulate the permutations in $\mathcal{H}$, via use of coset representatives in $\mathcal{G}\mathsf{al}$.

4.3 Low-Overhead FHE

Given the background from above (and the modification of the BGV cryptosystem [7] described in the full version), we explain in the full version how to set the parameters for our variant of the BGV scheme so as to get low-overhead FHE scheme. This gives us:

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 $p^n \leq poly(\lambda)$) can be evaluated homomorphically in time $t \cdot \tilde{O}(L) \cdot polylog(\lambda)$.*

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 [5], the moduli get smaller as we go

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

up the circuit, which means that for the first layers of the circuit we must choose moduli of bitsize $\Omega(L)$.

As explained in [5], 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 [5], 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 $p^n \leq poly(\lambda)$) can be evaluated homomorphically in time $t \cdot polylog(\lambda)$.*

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.

References

1. Rivest, R., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. In: Foundations of Secure Computation, pp. 169–180 (1978)
2. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) STOC, pp. 169–178. ACM (2009)
3. Gentry, C.: A fully homomorphic encryption scheme. PhD thesis, Stanford University (2009), `http://crypto.stanford.edu/craig`

4. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations (2011) (manuscript), http://eprint.iacr.org/2011/133
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. In: The 3rd Innovations in Theoretical Computer Science Conference, ITCS (2012), Full version at, http://eprint.iacr.org/2011/277
6. Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 1–23. Springer, Heidelberg (2010)
7. Brakerski, Z., Vaikuntanathan, V.: Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 505–524. Springer, Heidelberg (2011)
8. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. In: FOCS. IEEE Computer Society (2011)
9. Lauter, K., Naehrig, M., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: ACM Workshop on Cloud Computing Security, pp. 113–124 (2011)
10. Stehlé, D., Steinfeld, R.: Faster Fully Homomorphic Encryption. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 377–394. Springer, Heidelberg (2010)
11. Gentry, C., Halevi, S.: Implementing Gentry's Fully-Homomorphic Encryption Scheme. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 129–148. Springer, Heidelberg (2011)
12. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead (2011), Full version at http://eprint.iacr.org/2011/566
13. Smart, N.P., Vercauteren, F.: Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 420–443. Springer, Heidelberg (2010)
14. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 4th edn. Morgan Kaufmann (2006)
15. Beneš, V.E.: Optimal rearrangeable multistage connecting networks. Bell System Technical Journal 43, 1641–1656 (1964)
16. Waksman, A.: A permutation network. J. ACM 15, 159–163 (1968)
17. Damgård, I., Ishai, Y., Krøigaard, M.: Perfectly Secure Multiparty Computation and the Computational Overhead of Cryptography. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 445–465. Springer, Heidelberg (2010)
18. Lev, G., Pippenger, N., Valiant, L.: A fast parallel algorithm for routing in permutation networks. IEEE Transactions on Computers C-30, 93–100 (1981)
19. Leighton, F.T.: Introduction to parallel algorithms and architectures: arrays, trees, hypercubes, 2nd edn. M. Kaufmann Publishers (1992)