Algorithms in HElib

Shai Halevi (IBM) Victor Shoup[*](NYU)

Abstract

HElib is a software library that implements homomorphic encryption (HE), specifically the Brakerski-Gentry-Vaikuntanathan (BGV) scheme, focusing on effective use of the Smart-Vercauteren ciphertext packing techniques and the Gentry-Halevi-Smart optimizations. The underlying cryptosystem serves as the equivalent of a "hardware platform" for HElib, in that it defines a set of operations that can be applied homomorphically, and specifies their cost. This "platform" is a SIMD environment (somewhat similar Intel SSE and the like), but with a unique cost metrics and parameters. In this report we describe some of the algorithms and optimization techniques that are used in HElib for data movement and simple linear algebra over this "platform."

Keywords: Homomorphic Encryption, Implementation, SIMD

Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI/NBC) contract number D11PC20202. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: 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 IARPA, DoI/NBC, or the U.S. Government.

[*]Work partially done in IBM Research.

Contents

1 Introduction

Homomorphic encryption (HE) [17, 8] enables performing arithmetic operations on encrypted data even without knowing the secret decryption key. All HE schemes to date roughly follow the outline of Gentry's first candidate from 2009, in which fresh ciphertexts are "noisy" to ensure security and this noise grows with every operation until it becomes so large so as to cause decryption errors. This results in a "somewhat homomorphic" encryption scheme (SWHE) that can only evaluate low-depth circuits, which can then be converted to a "fully homomorphic" encryption scheme (FHE) using bootstrapping. Currently, the most asymptotically efficient SWHE schemes that we have are the RLWE-veriants of Brakerski-Gentry-Vaikuntanathan scheme [6] and Brakerski's scale-invariant scheme [4], and the NTRU-based scheme [12, 15]. All these schemes work in polynomial rings, and use rings of the form $R_p = \mathbb{Z}[X]/(F(X), p)$ as their native plaintext space, with F a cyclotomic polynomial and p an integer.

Smart and Vercauteren observed [18] that (for a prime p) an element in this native plaintext space can be used to encode a vector of values from a finite field $\mathbb{F}_{p^d}$, for some integer d that depends on F and p, and that homomorphic operations then induce the corresponding element-wise operation on the encrypted vectors. Gentry, Halevi, and Smart showed [10] how to use the SV "ciphertext packing" technique to perform asymptotically efficient computation, where a (wide enough) T-gate arithmetic circuit can be evaluated homomorphically in time $T \cdot \mathsf{polylog}(k)$, with k the security parameter. Crucial to obtaining this asymptotic efficiency is the use of automorphisms as a technique to move values between the different "slots" in a given plaintext vector, following [16, 6].

Turning to software implementations, HElib [11] is an open-source C++ library that implements the BGV scheme, focusing on effective use of ciphertext packing and the GHS optimizations. It includes an implementation of the BGV scheme itself with all its basic homomorphic operation, and also some higher-level procedures implementing the GHS data-movement procedures and simple linear algebra. This report is focused on these higher level procedures and the various optimizations that went into implementing them.

A useful analogy to keep in mind is to think of the lower-level of HElib as implementing an "assembly language" which is executed on a "hardware platform" given by the underlying HE scheme. The "platform" defines a set of operations that can be applied homomorphically and the cost of these operations; our goal in the current work is to provide efficient implementation of simple routing and linear-algebra procedures over that "platform." Since the homomorphic operations define element-wise operations on the vector of plaintext values, the "platform" defines for us a SIMD environment (somewhat similar to things like Intel's SSE, the Motorola/IBM AltiVec architecture, and the like). Hence the focus of this work is the design of efficient algorithms over this SIMD architecture.

We note that although SIMD hardware architectures are quite common in practice (cf. [19]), we were unable to find much algorithmic literature concerning asymptotic efficiency in such environments. This is perhaps related to the fact that common hardware architectures have vectors with only a handful of entries (for example an SSE register can hold at most 32 8-bit values). On the other hand, the plaintext arrays in HElib often hold a few hundred plaintext slots (sometimes even a few thousand), making asymptotic treatment of SIMD algorithms more relevant. Another difference between the "platform" provided by HE and the common hardware SIMD platforms is their cost metrics: in HElib we need to optimize for two parameters, namely time and noise-magnitude. These correspond roughly to size and depth of the corresponding SIMD circuits, but the correspondence is not quite one-to-one since different operations have different time and noise behavior.

Contents of this report. In Section 2 we describe some details of the "platform" that we get from the underlying HE scheme, and introduce our notations. Then in Section 3 we describe the implementation and optimizations of the GHS permutation techniques. In particular we describe there a generalization of Benes networks to handle networks of arbitrary width, extending earlier work of Chang and Melham [7], and also our approach for optimizing the GHS "hypercube networks."

In Section 4 we describe our procedures for computing running- and total-sums of a vector, for replicating the entries of a vector, and for performing a matrix-vector multiplication. We also describe there procedures for computing the norm and trace functions on the individual plaintext slots.

Finally in Section 5 we give another level of detail, showing how the vectors that we get from our HE "platform" may not simply be linear arrays, but rather, may be hypercubes for which some dimensions may be slightly defective in their functionality. We explain these complications in detail and show how they affect the permutation and linear-algebra procedures from the previous sections (and also how to implement linear arrays on top of these hypercubes).

2 Background and Notation

The characteristics that define the "hardware platform" for HElib are common to many contemporary HE schemes, including the ring-LWE variants of BGV [6] and Brakerski's scale-invariant scheme [4], the NTRU-based HE scheme [12, 15], and maybe even some LWE-based schemes [5]. Two salient characteristics of these cryptosystems are the following:

Growing noise. All contemporary SWHE schemes use *noisy* ciphertexts, where a fresh ciphertext includes a noise component that grows with each homomorphic operation, until it is so large that it causes decryption errors. However, different operations have very different noise-growth behavior. For example, multiplication increases the noise much more than addition.

Plaintext vectors. The plaintext space of these schemes can be viewed as a vector space over some finite field (or a module over a finite ring). This means that each native plaintext of the cryptosystem corresponds to a vector of plaintext values that the application cares about. The underlying field (or ring) and the dimension of the vector are both derived from some parameters of the cryptosystems; see, e.g., [10, Appendix c.2] (in the full version) for a description. When using such cryptosystems for specific homomorphic computation, we are typically faced with a 2-parameter optimization problem, trying to minimize both the noise-growth and the running time. In a typical scenario we would first choose the system parameters, which determine the maximum allowable level of noise, and then try to minimize the running time subject to this fixed bound on the noise. Consequently most of the optimization procedures that we describe in this work has the form of optimizing the running time subject to some depth constraints.

In Table 1 we summarize the available homomorphic operations, their effect on the noise, and their running time. For each parameter (noise and time) we divide the operations into expensive, moderate, and cheap. We often think of the cheap operations as essentially for free, the expensive operations as costing one unit (of either time or noise), and the moderate operations as having a cost of 1/2 unit. We remark that the cost in Table 1 (and even the operations themselves) are merely an approximation, see Section 5 for some more details.

We would like to draw the reader's attention to the "moderate" noise-growth of the multiply-by-constant operation, and stress that we have to pay this "moderate" cost even if we are multiplying by a constant zero-one vector. This is different than other (additively) homomorphic schemes where multiplication by zero or one is really "for free." In our implementation we extensively use multiplications by zero-one vectors to extract from a given vector only some of the entries but not others. We refer to this operation as multiplicative masking (or masking, for short). We also note that using ro-

Operation	Time	Noise	Comments
Addition	cheap	cheap	element-wise addition of vectors
Constant-add	cheap	cheap	element-wise addition of a constant vector
Multiplication	expensive	expensive	element-wise multiplication of vectors
Constant-multiply	cheap	moderate	element-wise multiplication by a constant vector
Rotation	expensive	cheap	cyclic rotation of vector by any amount
Frobenius	expensive	cheap	element-wise Frobenius map, $X \mapsto X^{p^n}$

Table 1: Homomorphic operations and their cost

tations and multiplicative masking we can implement shifts with zero-fill, which would be expensive in terms of running time and moderate in terms of noise.

Some notations. Throughout this report we use $[n]$ for the set $\{0 \mathinner{..} n-1\}$, and use zero-based indexing for vectors. For two vectors u, v, we use $u + v$ and $u \times v$ to denote entry-wise addition and multiplication.

3 Permutations and Shift-Networks

The core of the GHS homomorphic data-routing techniques [10] is the use of Benes-like networks to arbitrarily permute the slots in a ciphertext (which is needed to allow different slots to interact with each other). In this section we describe our implementation and optimizations of the GHS techniques. We begin by introducing the notion of a *shift network*, and the shift-network minimization problem.

3.1 Shift Networks

A shift network is a method to realize an arbitrary permutation in terms of rotations, multiplicative masking, and additions. We begin by describing an arbitrary permutation in terms of a single "shift column": for an arbitrary permutation $\pi : [n] \to [n]$, the shift-column corresponding to π is a vector sh_π that describes for each index i the distance that i needs to travel under π. In formula, we have $sh_\pi[i] = \pi(i) - i$ (subtraction over the integers).

We note that a shift-column gives us a simple way of applying π to an arbitrary vector v using shift operations, multiplicative masking, and additions. Namely, for every value δ that appears in sh_π we first construct a mask m_δ which is 1 in the entries where $sh_\pi[i] = \delta$ and 0 elsewhere. We then extract from v only these entries (by multiplying $m_\delta \times v$) and shift the result by δ positions, and finally add up all the resulting vectors. Namely the permuted vector is obtained by

$$w \leftarrow \sum_{\delta \in sh_\pi} (m_\delta \times v) \gg \delta$$

where $\times$ denote entry-wise multiplication and $\gg$ denotes shift. The running-time cost of this implementation of π is proportional to the number of distinct values in sh_π. Specifically if sh_π contains t distinct non-zero values then this implementation would perform t shift operations (and some other cheap operations that we ignore). Hence we define the *cost* of sh_π as the number of distinct non-zero values in it. The cost of this operation in terms of noise is roughly a single multiply-by-constant (since adding the resulting vector has almost no effect on the noise).

If we use rotations instead of shifts, then we can apply a similar procedure but this time use a mask m'_δ which is 1 in the entries where $sh_\pi[i] = \delta \pmod{n}$ and 0 elsewhere, then set

$$w \leftarrow \sum_\delta (m'_\delta \times v) \ggg \delta$$

where $\ggg$ denotes rotation. The running-time cost of the implementation would then be related to the number of distinct non-zero values in sh_π *modulo n*, and the cost in terms of noise will be a single multiply-by-constant (since rotations and additions are cheap). We thus also define the *reduced cost* of sh_π as the number of distinct non-zero values in it modulo n.

A shift network N is a sequence of shift-columns, namely, an $n \times d$ matrix (for some d), with each column representing a permutation. If the d columns represent the permutation $\pi_1, \ldots, \pi_d$ then the network as a whole represents the composed permutation $\pi = \pi_d \circ \cdots \circ \pi_1$. We say that d is the *depth* of the shift network, and the columns of N are the *levels* of the network. The (reduced) cost of the network N is just the sum of the (reduced) costs of all levels.

A shift network for π implies an algorithm for applying π to vectors, just by applying each π_i in turn using its shift vector. If the network has depth d and reduced cost c, then this implementation of π takes c multiplicative masks, c rotations, and $O(c)$ additions, and has depth of d multiplicative masking operations, d rotations, and $O(d)$ additions.

The Cheapest-shift-network (CSN) problem. Of course there are many different shift networks that implement the same permutation, and given a target permutation π we want to find the cheapest network for it. In our setting, we typically think of the depth as a constraint and the (reduced) cost as the quantity that we optimize for. Hence we get the following optimization problem:

Input: A permutation π over $[n]$ and a depth-bound B.

Output: A shift-network for π of depth at most B, minimizing the (reduced) cost.

We note that the bound parameter really does matter. For example, most permutations require a cost-$\Omega(n)$ depth-1 solution, but every permutation has a cost-$O(\sqrt{n})$ depth-2 solution (and more generally cost $O(d \cdot n^{1/d})$ depth-d solution). Even the unbounded version of this problem (with $B = \infty$) seems interesting, but in our case we are typically more interested in the bounded version. We do not know of an efficient procedure for finding the least-cost network for a given permutation and depth-bound, and speculate that it is a hard problem. Below we show, however, that when restricting ourselves to a certain natural class of solutions we can efficiently find the least-cost solution in this class.

3.2 Benes Networks

A Benes network for a permutation π is a special kind of shift network, which is rather cheap and can be constructed efficiently from any permutation. We begin by reviewing basic Benes network construction for $n = 2^r$, then describe the generalization of Chang and Melham [7] to arbitrary n and our optimization of the Chang-Melham construction for our setting.

For $n = 2^r$, a Benes network for a permutation π on $[n]$ is a shift network of depth $2r - 1$, where every level in the network has a cost at most 2. Such a network decomposes π into $2r - 1$ permutations: $\pi = \sigma_{r-1} \circ \cdots \sigma_1 \circ \sigma_0 \circ \tau_1 \cdots \circ \tau_{r-1}$, where the action of each σ_k and τ_k is to move any $i \in [n]$ to either i, $i + 2^k$, or $i - 2^k$. Moreover, each σ_k and τ_k consists of 2^{r-1-k} separate permutations on different 2^{k+1}-intervals of indexes. This is a network of depth $2r - 1 = O(\log n)$ and cost at most $4r - 2 = O(\log n)$, hence it corresponds to a fairly efficient permutation algorithm.

Decomposing a permutation into a Benes network can be done via a recursive procedure. In the first step, we decompose $\pi = \sigma \circ \rho \circ \tau$, with ρ consisting of two separate permutation over the top half and bottom half of the network, and then we recurse on two halves of ρ. Computing the decomposition $\pi = \sigma \circ \rho \circ \tau$ can be done using the greedy "looping algorithm." Denote $m = n/2 = 2^{r-1}$, $S_0 = \{0 \,..\, m-1\}$ and $S_1 = \{m \,..\, n-1\}$. We seek a decomposition as above such that:

(P1) σ and τ map each $i \in S_0$ to either i or $i+m$, and each $i \in S_1$ to either i or $i-m$;

(P2) ρ consists of two permutations on S_0 and S_1 separately.

We construct an undirected graph G with n nodes on each side, L_i, R_i for $i \in [n]$, then add an edge from L_i to $R_{\pi(i)}$ for each $i \in [n]$ (call these "permutation edges"), and also add edges from each L_i to L_{i+m} and each R_i to R_{i+m} when $i < m$ (called "conflict edges").

It is easy to see that G is 2-colorable. Indeed, a simple algorithm to 2-color the graph is to start at any node, trace out a path that must lead back to the starting node, alternates between permutation and conflict edges. This creates an even-size circle that we can color with two colors, then remove from G and repeat the procedure on the smaller graph.

Once we have a two coloring of G with each vertex ν colored by $C(\nu) \in \{0,1\}$, we define σ and τ as follows: For each left vertex L_i we interpret a color of 0 as τ sending i to the top half and color of 1 as τ sending i to the bottom half. So we have $\tau(i) \leftarrow i$ if $i \in S_0$ and $C(L_i) = 0$ or if $i \in S_1$ and $C(L_i) = 1$, and otherwise $\tau(i) \leftarrow i \pm m$.

Similarly for each right vertex R_i we interpret a color of 0 as σ receiving i from the bottom half and color of 1 as σ receiving i from the top half. Hence we have $\sigma^{-1}(i) \leftarrow i$ if $i \in S_0$ and $C(L_i) = 1$ or if $i \in S_1$ and $C(L_i) = 0$, and otherwise $\sigma^{-1}(i) \leftarrow i \pm m$.

$$\begin{array}{lll} \text{for } i \in S_0\text{:} & \tau(i) \leftarrow i + C(L_i)m, & \sigma^{-1}(i) \leftarrow i + (1 - C(R_i))m; \\ \text{for } i \in S_1\text{:} & \tau(i) \leftarrow i - (1 - C(L_i))m, & \sigma^{-1}(i) \leftarrow i - C(R_i)m. \end{array} \tag{1}$$

Setting the permutations τ and σ determines also the middle permutation ρ (which must satisfy property (P2)) and we can then recurse on the two halves of ρ.

We stress that in our setting it is crucial that the shift amounts for the permutations σ_k, τ_k are always exactly $\pm 2^k$ and 0, regardless of the permutation π. Indeed, in the above we recurse on two different halves of ρ, and subsequent steps recurse on a large number of different permutations. Had the shift amounts depended on the actual permutations, we would have had a higher cost for the shift-columns that implement ρ.

3.3 General Benes Networks

Chang and Melham [7] proposed a generalization of Benes networks that works for any n, not just a power of two. Below we describe this generalization and then optimize it for our setting.

Note that the procedure above for decomposing $\pi = \sigma \circ \rho \circ \tau$ work for any even n. When n is odd, we instead break the network into two "nearly equal" parts, namely one part of size $\lfloor n/2 \rfloor$ the other of size $\lceil n/2 \rceil$. Suppose that we let the top part be the smaller of the two, so we set $m = \lfloor n/2 \rfloor$, $S_0 = \{0 \,..\, m-1\}$ and $S_1 = \{m \,..\, n-1\}$. Chang and Melham observed that we can adapt the procedure from above for decomposing $\pi = \sigma \circ \rho \circ \tau$ with properties (P1) and (P2) simply by insisting that the last index, $n-1$, is mapped to itself by both σ and τ, and applying the procedure from above to all the other indexes. Formally, we construct a graph G as above but only put the conflict edges L_i—L_{i+m} and R_i—R_{i+m} for $i < m$ (which means that none of these edges touches L_{n-1} or R_{n-1}), and then add a special conflict edge between L_{n-1} and R_{n-1}. The rest of the algorithm works without any change, and correctness follows from the exact same arguments.

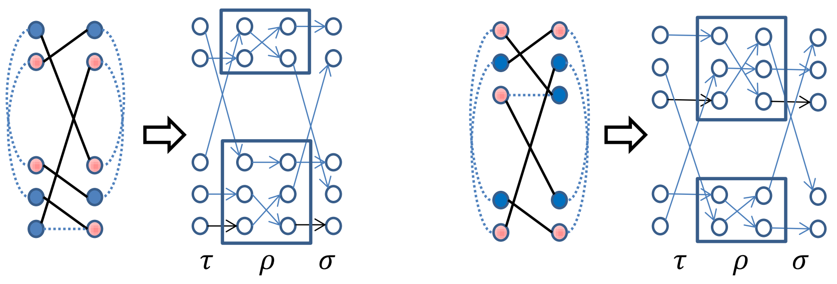


Figure 1: An illustration of the two different ways to decompose a size-5 permutation as $\pi = \tau \circ \rho \circ \sigma$.

Now that we can partition both even- and odd-size networks, we can again recurse and construct a "generalized Benes network" of depth $d = 2\lceil \log n \rceil - 1$ for any permutation. However, we no longer have the property that each level of the network only has shift amounts 0 and $\pm m$ for a single shift amount m, so we can no longer bound the cost of the network by $2d$.

Trying to bound the cost of the resulting network, we observe that all the sub-permutations at a certain level of the network are almost of the same size. specifically they have size either $\lceil n/2^k \rceil$ or $\lfloor n/2^k \rfloor$. It follows that each level has at most four non-zero shift amounts, namely $\pm\lceil n/2^{k+1} \rceil$ and $\pm\lfloor n/2^{k+1} \rfloor$, so we can bound the cost of the network by $4d$. Unfortunately this bound still implies a factor-of-2 slowdown when n is not a power of two. Below we describe another optimization that allows us to recover the original bound of $2d$.

Further optimizations. To reduce the cost further, we observe that there are two different options for how to split the network when n is odd, and that these two options result in different shift amounts in the shift-vectors for σ and τ. Specifically, above we made the bottom part larger, which meant setting the shift amount to $m = \lfloor n/2 \rfloor$ and fixing $\sigma(n-1) = \tau(n-1) = n-1$ by adding a conflict edge between $L_{n-1} = R_{n-1}$. However we can also make the top half larger, then we set the shift amount to $m = \lceil n/2 \rceil$ and fix $\sigma(m-1) = \tau(m-1) = m-1$ by adding a conflict edge between $L_{m-1} = R_{m-1}$. An illustration of the two bipartite graphs and the corresponding decompositions of π that we get for a size-5 permutation can be found in Figure 1.

This observation gives us the freedom to choose the shift amounts that are used in partitioning odd-size subnetworks to either $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$, as needed to be compatible with the even-size subnetworks in that level (if any). Thus we can recursively decompose any permutation π on $[n]$ for arbitrary n as $\pi = \sigma_{r-1} \circ \cdots \sigma_1 \circ \sigma_0 \circ \tau_1 \circ \cdots \tau_{r-1}$, where $r = \lceil \log_2 n \rceil$ and the action of each σ_k and τ_k is to move any $i \in [n]$ to either i or $i \pm \Delta_k$, with the "shift amount" $\Delta_k := \left\lceil \lfloor n/2^{r-1-k} \rfloor / 2 \right\rceil$. We get a shift network for π of depth $2\lceil \log_2 n \rceil - 1$ and a cost of 2 for each level, which means a $(4 \log n)$-approximation for the *unbounded* cheapest-shift-network problem. As we said above, however, in our application we typically care more about the *bounded* cheapest-shift-network problem. In the next section we describe a method for incorporating the bounded depth into the algorithm.

3.4 Balancing Depth and Cost in Benes Networks

In our application to HE we often need to consider trade-offs between depth and cost in constructing shift networks. One natural way to enforce a depth constraint is to start from a solution to the unbounded CSN problem (such as a Benes network), and then "collapse" several consecutive levels into one, thereby reducing the depth at the price of increasing the cost.

Given a general Benes network and a bound B, we seek the "optimal way" to collapse consecutive levels so as to get a depth-B network for the same permutation. Recall that the domain size n

determines the depth d of the generalized Benes network, as well as the set of possible shift amounts that may appear at each level of the network. Our approach is therefore to devise the level-collapse strategy based only on n and the bound B, rather than re-compute it for each permutation separately.

To compute the optimal level-collapse strategy for given n and B, we use a simple dynamic-programming approach. Let $d = 2\lceil \log_2 n \rceil - 1$ be the depth of a generalized Benes network for size-n permutations and let S_k be the set of shift amounts that can occur at level k in the network. (That is, $S_k = \{0, \pm\Delta_k\}$ for $k \le \lceil \log n \rceil$ and $S_k = \{0, \pm\Delta_{d-k}\}$ for $k > \lceil \log n \rceil$.)

For each pair of indexes $0 \le j_1 \le j_2 < d$, we let $L(j_1, j_2)$ denote the number of possible non-zero shift amounts that can occur when collapsing levels j_1 through j_2. (This number is certainly an upper bound for the cost of the corresponding shift-vector for any particular Benes network, and usually it is a fairly tight one.) Specifically, $L(j_1, j_2)$ is the number of distinct non-zero integers in the interval $(-n, n)$ that can be written as a sum $\epsilon_{j_1} + \epsilon_{j_1+1} + \cdots + \epsilon_{j_2}$, with $\epsilon_k \in S_k$ for all $k = j_1 \mathinner{..} j_2$. Clearly the $L(j_1, j_2)$ values can be computed efficiently (in time quasi-linear in n). Given these values, we can write a recursive formula for the optimal level-collapsing strategy for a given n, B. Specifically for each $0 \le d' \le d, 0 \le B' \le B$ let $\mathrm{Opt}(d', B')$ be the cost of the optimal way of collapsing some of the first d' columns of the depth-d network so as to get depth B'. Then we have $\mathrm{Opt}(d', B') = 0$ if $d' = 0$, $\mathrm{Opt}(d', B') = \infty$ if $d' > 0$ and $B' = 0$, and otherwise

$$\mathrm{Opt}(d', B') = \min_{\ell=1..d'} \left\{ L(d' - \ell, d' - 1) + \mathrm{Opt}(d' - \ell, B' - 1) \right\}.$$

In words, we consider collapsing the last ℓ levels into a single level of cost $L(d' - \ell, d' - 1)$, and then add to that the optimal cost for the first $d' - \ell$ levels, using the bound $B' - 1$ in place of B'.

Since there are only $O(d^2)$ values (d', B') as above, we can use standard dynamic programming techniques to compute $\mathrm{Opt}(d, B)$ and the collapsing strategy that achieves it.[1] We should note here that any $n \times d$ shift network can be collapsed to a network of depth 1 and cost at most $2n - 1$ (and reduced cost at most $n - 1$).

3.5 Hypercube networks

A different method of constructing shift networks, which is described in [10], is via "hypercube networks": If n can be factored as $n = ab$, then we can impose on $[n]$ a two-dimensional matrix structure of a rows and b columns, using some appropriate bijective map $M : [n] \to [a] \times [b]$. Some possible choices of the map M include:

CRT order (only when $\gcd(a, b) = 1$): M maps $i \in [n]$ to $(i \bmod a, i \bmod b) \in [a] \times [b]$;

Row-major order: M maps $0 \mathinner{..} b - 1$ to the first matrix row, $b \mathinner{..} 2b - 1$ to the second row, etc;

column-major order: M maps $0 \mathinner{..} a-1$ to the first matrix column, $a \mathinner{..} 2a-1$ to the second column, etc.

Row- and column-major orders may appear more natural, but CRT ordering (when applicable) has an advantage, because the map M is actually a ring homomorphism (viewing $[n], [a], [b]$ as the rings $\mathbb{Z}_n, \mathbb{Z}_a, \mathbb{Z}_b$, respectively). As done in [10], we will use the following decomposition lemma from [14]:

Lemma 1 *Let $S = [a] \times [b]$ be a set of ab positions, arranged as a rectangular matrix of a rows and b columns. For any permutation π over S, there are permutations σ, ρ, τ such that $\pi = \sigma \circ \rho \circ \tau$, where σ and τ permute positions within each column, and ρ permutes positions within each row. Moreover, there is a polynomial-time algorithm that given π outputs the permutations σ, ρ, τ.*

[1] This algorithm can be easily adapted to use reduced network costs in place of network costs, when that is the desired cost metric.

Of course, once we decompose π as above, we can apply the same lemma recursively to each row of ρ, thus imposing an r-dimensional hypercube structure on $[n]$ and decomposing π into $2r-1$ permutations $\pi = \pi_1 \circ \cdots \circ \pi_{2r-1}$, each of which acts along a single dimension.[2] We can then construct Benes networks for the π_i's, collapsing some of the levels within those networks so as to satisfy a bound B on the overall depth. Optimizing over this class of solutions requires finding the best splitting of n into factors, the best way to layout the hypercube, and the best strategy for collapsing the levels of the Benes networks.

So consider $n = ab$, and a map $M : [n] \to [a] \times [b]$, which induces a correspondence between a permutation π on $[a] \times [b]$ and its representation $\bar{\pi}$ as a permutation on $[n]$. Furthermore, consider the natural generalization of the notion of a shift network to an $a \times b$ matrix: the entries in such a network are now of the form $(\Delta i, \Delta j)$, and in determining reduced costs, we consider two entries $(\Delta i, \Delta j)$ and $(\Delta i', \Delta j')$ to be equivalent if $\Delta i \equiv \Delta i' \pmod a$ and $\Delta j \equiv \Delta j' \pmod b$.

Next, consider a decomposition $\pi = \sigma \circ \rho \circ \tau$, as in Lemma 1 and let $\bar{\sigma}, \bar{\rho}, \bar{\tau}$ be the corresponding permutations on $[n]$. We can easily translate shift networks for σ, ρ, τ into shift networks for $\bar{\sigma}, \bar{\rho}, \bar{\tau}$; however, the relationship between the (reduced) costs of the shift for σ, ρ, τ and the (reduced) costs of the shift networks for $\bar{\sigma}, \bar{\rho}, \bar{\tau}$ depends on the mapping M used to impose the matrix structure on $[n]$.

CRT order. Let λ_a, λ_b be the CRT coefficients of a, b, respectively. Then a shift amount of $(\Delta i, \Delta j)$ for a permutation on $[a] \times [b]$ translates to a shift amount that is congruent to $\lambda_a \Delta i + \lambda_b \Delta j$ modulo n for a permutation on $[n]$. Since $\lambda_a \equiv 0 \pmod b$ and $\lambda_b \equiv 0 \pmod a$, it follows that the reduced costs of the shift networks for $\bar{\sigma}, \bar{\rho}, \bar{\tau}$ are equal to the reduced costs for the networks for σ, ρ, τ. Thus, reduced costs are preserved in the translation; however, unreduced costs may not be preserved.

Row-major order. A shift amount of $(\Delta i, \Delta j)$ for a permutation on $[a] \times [b]$ translates to a shift amount of $b\Delta i + a\Delta_i$ for a permutation on $[n]$. It follows that the unreduced costs of the shift networks for $\bar{\sigma}, \bar{\rho}, \bar{\tau}$ are equal to the unreduced costs of the networks for σ, ρ, τ.

For reduced costs, the situation is a bit different. The shift networks for σ, τ have entries of the form $(\Delta i, 0)$, which translates to $b\Delta_i$; it follows that the reduced costs of the shift networks for $\bar{\sigma}, \bar{\tau}$ are the same as the reduced costs of the shift networks for σ, τ. In contrast, the shift network for ρ has entries of the form $(0, \Delta_j)$, which translates to Δ_j; it follows that the reduced cost of the shift network for $\bar{\rho}$ is equal to the *unreduced cost* of the shift network for ρ.

Column-major order. This situation is analogous to row-major order, except that now the reduced costs of the shift networks for $\bar{\sigma}, \bar{\tau}$ are equal to the *unreduced costs* of the shift networks for σ, τ, while the reduced cost of the shift network for $\bar{\rho}$ is equal to reduced cost of the shift network for ρ.

The above observations suggest a recursive formulation to obtain a network of optimal cost for domain size n satisfying a bound B on the depth of the network. Starting from an initial domain size n, bound B, and cost metric to optimize (reduced/unreduced cost), we compare using size-n generalized Benes network to all splits $n = ab$ and all possible ways of allocating our depth budget B to the three recursive subproblems. We use row/column ordering for the $a \times b$ matrix when trying to minimize the unreduced cost, and CRT ordering when trying to minimize the reduced cost and have $\gcd(a, b) = 1$. We then recursively solve the three subproblems, trying to optimize either the reduced or unreduced cost, as needed according to the rules from above.

Let $\mathsf{SplitRcost}(n, B), \mathsf{SplitUcost}(n, B)$ denote the best reduced/unreduced cost for a side-n network with depth-bound B, and similarly let $\mathsf{BenesRcost}(n, B), \mathsf{BenesUcost}(n, B)$ be the best (re-

[2]Clearly, a Benes network of width $n = 2^r$ is a special case of this construction. Unfortunately, we do not know of a generalization of Lemma 1 along the lines of the generalized Benes networks from [7].

Cyclotomic field	Vector size	Shift-network depth	Shift-network cost	Time
$m = 4369$	$n = 256$	3	60	4.1 sec
		7	35	2.6 sec
		10	31	2.8 sec$*$
$m = 8191$	$n = 630$	5	37	5.0 sec
		7	30	4.3 sec
		9	28	4.0 sec
$m = 21845$	$n = 1024$	5	66	21.2 sec
		7	45	18.3 sec$*$
		9	41	16.7 sec$*$

Table 2: Timing results for permutations in various vector sizes. The starred lines indicate that we had to choose larger parameters because of the larger depth.

duced/unreduced) cost of a generalized Benes for these parameters. Then we have:

$$\mathsf{SplitUcost}(n,B) = \min\left(\begin{array}{l} \mathsf{BenesUcost}(n,B), \\ \displaystyle\min_{\substack{ab=n \\ B_1+B_2+B_3=B}} \big(\mathsf{SplitUcost}(a,B_1) + \mathsf{SplitUcost}(b,B_2) + \mathsf{SplitUcost}(a,B_3)\big) \end{array}\right);$$

$$\mathsf{SplitRcost}(n,B) = \min\left(\begin{array}{l} \mathsf{BenesRcost}(n,B), \\ \displaystyle\min_{\substack{ab=n,\gcd(a,b)=1 \\ B_1+B_2+B_3=B}} \big(\mathsf{SplitRcost}(a,B_1) + \mathsf{SplitRcost}(b,B_2) + \mathsf{SplitRcost}(a,B_3)\big), \\ \displaystyle\min_{\substack{ab=n,\gcd(a,b)\neq1 \\ B_1+B_2+B_3=B}} \big(\mathsf{SplitRcost}(a,B_1) + \mathsf{SplitUcost}(b,B_2) + \mathsf{SplitRcost}(a,B_3)\big), \\ \displaystyle\min_{\substack{ab=n,\gcd(a,b)\neq1 \\ B_1+B_2+B_3=B}} \big(\mathsf{SplitUcost}(a,B_1) + \mathsf{SplitRcost}(b,B_2) + \mathsf{SplitUcost}(a,B_3)\big) \end{array}\right).$$

Since there are only polynomially many (n, B) pairs, we can again use dynamic programming to solve these recursions efficiently. We note that to count the total number of rotations required to implement a permutation on a domain of size n, the relevant quantity is the *reduced cost* of the network, i.e., $\mathsf{SplitRcost}(n, B)$. However, in calculating this reduced cost we need to know the unreduced cost of some of the subproblems that arise in the above calculation.

Performance results. An illustrative timing results for some settings of the parameters are given in Table 2.

4 Replication and Linear Algebra

Since our "platform" works natively on vectors of plaintext values, it seems natural to provide support for simple vector and linear algebra operations. In this section we describe algorithmic issues in our implementation of these operations. We begin with some basic operations for computing running sums and total sums, and then continue to replication and matrix-vector multiplication.

4.1 Running- and Total Sums

The "running sums" function $w \leftarrow \mathrm{RS}(v)$ outputs a vector w such that $w[i] = \sum_{k=0}^{i} v[k]$ for $i \in [n]$. The "total sums" function $w \leftarrow \mathrm{TS}(v)$ outputs a vector w such that $w[i] = \sum_{k=0}^{n-1} v[k]$ for $i \in [n]$. Both of these functions are implemented using a "repeated doubling" approach whose running time and depth is $O(\log n)$ additions and rotations/shifts.

Below is the code for these procedures, note that the running-sums procedure uses shifts with zero-fill (which can be implemented using rotations and multiplicative masking), while total-sums uses rotations. Below we denote by $\mathrm{numBits}(n)$ is the number of bits in n, and $\mathrm{bit}_j(n)$ is the jth bit of n (with bit 0 being the low-order bit). The invariant throughout the total-sums procedure is that $w[i] = \sum_{k=0}^{e-1} v[i - k \bmod n]$ for $i \in [n]$; moreover, at the end of loop iteration j, the binary representation of e consists of bits $j \mathinner{..} \mathrm{numBits}(n) - 1$ of n.

```
RS(v):
1  $w \leftarrow v$, $e \leftarrow 1$
2  while $e < n$ do
3          $w \leftarrow w + (w \gg e)$, $e \leftarrow 2 \cdot e$
4  return $w$
```

```
TS(v):
1  $w \leftarrow v$, $e \leftarrow 1$
2  for $j \leftarrow \mathrm{numBits}(n) - 2$ down to 0 do
3          $w \leftarrow w + (w \ggg e)$, $e \leftarrow 2 \cdot e$
4          if $\mathrm{bit}_j(n) = 1$ then
5                  $w \leftarrow v + (w \ggg 1)$, $e \leftarrow e + 1$
6  return $w$
```

We stress that although these two procedures are quite similar, the total-sum procedure uses only rotations and additions that are "cheap" in terms of noise, while the running-sums procedure uses shifts that induce "moderate" noise growth via the requisite masks.

4.2 Replication

Typical homomorphic computation has gates with large fan-out, which require that we replicate some plaintext values many times. We have not (yet) implemented a completely generic replication method (such as the ones from [10]), but we describe procedures that we did implement for efficient replication in a few interesting special cases.

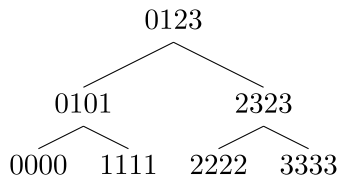
Replicating a single value. We begin with a procedure for replicating a single entry across the entire array. This procedure uses multiplicative masking to extract the entry, then total-sums to replicate it across the vector. It has both running time and depth of $O(\log n)$ additions and rotations and a single multiplicative masking.

4.2.1 Full replication

In full replication we take a vector v and produce vectors $w_1, \ldots, w_n$ such that for $i = 1 \mathinner{..} n$, w_i has $v[i]$ in all positions. A naive solution just repeats the single-element replication n times, resulting in running time of $O(n \log n)$ additions and rotations, and n masks; and depth $O(\log n)$ "cheap" additions and rotations and one "moderate" masking.

We now describe a faster simple recursive procedure that uses just $O(n)$ additions, rotations, and masks, but has depth $O(\log n)$ multiplicative masking operations. Later we present a hybrid algorithm with the same linear running time, but with masking depth of just $O(\log \log n)$.

A simple recursive procedure. Consider first the case of $n = 2^\ell$, in this case it is easy to apply a divide-and-conquer approach, where in each stage we double the number of vectors while halving the number of distinct values in each vector. The following diagram illustrates this approach on a vector of size 4:



Implementing this approach, we have a recursive procedure that takes as input a vector w and an integer $h = 0 \mathinner{..} \ell$ (and is invoked initially with $w = v$ and $h = \ell$). The input vector w consists of $2^{\ell-h}$ repetitions of the same size-2^h vector (which we call u). The procedure computes two vectors w_L, w_R, with w_L consisting of $2^{\ell-h+1}$ repetitions of the first half of u, and w_R consisting of $2^{\ell-h+1}$ repetitions of the second half of u, and then concatenates the lists obtained by processing w_L and w_R, with h decreased by 1. The recursion stops when $h = 0$ and the singleton list $\langle w \rangle$ is returned.

```
RecursiveReplicate($w, h$) :
1  if $h = 0$ then return $\langle w \rangle$
2  else
       set $mask[i] = \mathrm{bit}_{h-1}(i)$ for $i \in [n]$   // choose half the entries
3      $w_1 \leftarrow mask \times w$, $w_0 \leftarrow w - w_1$
4      $w_\mathrm{L} \leftarrow w_0 + (w_0 \ggg 2^{h-1})$, $w_\mathrm{R} \leftarrow w_1 + (w_1 \lll 2^{h-1})$
5      return RecursiveReplicate($w_\mathrm{L}, h-1$) || RecursiveReplicate($w_\mathrm{R}, h-1$).
```

It is easy to adapt this procedure for the case where n is not a power of 2. In this case, suppose 2^ℓ is the largest power of 2 not exceeding n. By multiplying by appropriate masks, we can construct vectors v_1 and v_2, so that v_1 equals v in the first 2^ℓ positions and is 0 everywhere else, and v_2 equals v in the last $n - 2^\ell$ positions and is 0 everywhere else. We apply RecursiveReplicate(v_1, ℓ), which gives us vectors $w_0, \ldots, w_{2^\ell - 1}$, where w_i is $v[i]$ is the first 2^ℓ positions, and 0 everywhere else. Since $2^\ell > n/2$, we can fill out the rest of each w_i as required at a cost of one mask, rotation, and addition per output vector. We apply the very same procedure to $v_2 \lll 2^\ell$, but we only need to process the first $n - 2^\ell$ vectors produced by RecursiveReplicate.

One easily verifies that the running time of this algorithm is $O(n)$ additions, rotations, and multiplicative masking; its depth is $O(\log n)$ additions, rotations, and masking.

4.2.2 A Shallower Full Replication Procedure

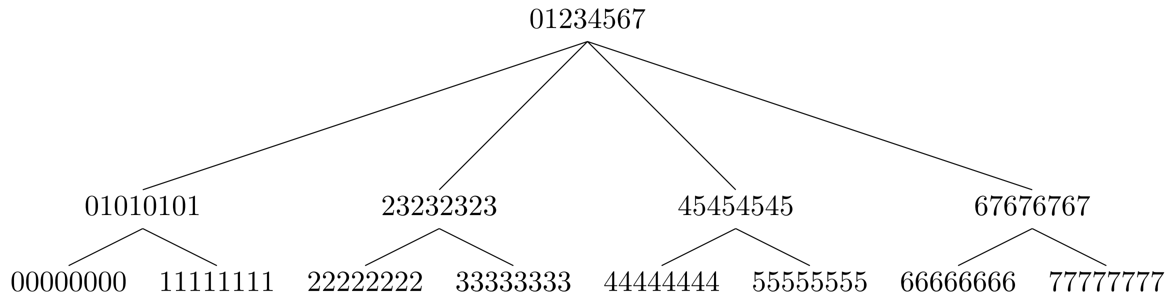
We now describe a modification of RecursiveReplicate that retains the same running time bound, while achieving a masking depth of $O(\log \log n)$, rather than $O(\log n)$. This is done by replacing the top levels of the recursive algorithm by flatter but more time-consuming procedure (similar to the naive solution from the beginning of Section 4.2.1), and only switch back to the recursive procedure for the bottom few levels. We show that with a judicious choice of the number of levels to flatten, the overall running time remains $O(n)$, while the masking depth decreases to $O(\log \log n)$. Again, assume for simplicity that $n = 2^\ell$ is a power of two, and let k be a parameter, whose value we will choose to be $\log_2 \log_2 n + O(1)$.

We partition the entries in the input vector v into $n/2^k$ blocks, each of size 2^k, with block i consisting of positions $i2^k \mathinner{..} (i{+}1)2^k{-}1$. In the first stage of the algorithm we use a "naive procedure," similar to the single-entry replication, to construct vectors v_i, $i = 0 \mathinner{..} n/2^k - 1$, where v_i consists of the entries in block i repeated $n/2^k$ times. (With our choice of the parameter $k \approx \log \log n$, this "naive part" does most of the replication work, giving us $n/\log n$ vectors with only $\log n$ distinct values in each.)

Each v_i is produced using the naive procedure, whose running time and depth are both $O(\log(n/2^k))$ additions and rotations, and a single multiplicative masking. Since we have to repeat this procedure for each v_i, the total running time of this first stage is $n/2^k \cdot \log(n/2^k)$ additions and rotations, and $O(n/2^k)$ masks. With our choice of $k \approx \log\log n$ we get running time of $n/\log n \cdot \log(n/\log n) = O(n)$.

For the second stage, we simply apply Algorithm RecursiveReplicate to (v_i, k) for $i = 0 \,..\, n/2^k - 1$. The running time of the second stage is $O(n)$ additions, rotations, and masks; its depth is $k = O(\log\log n)$ additions, rotations, and masks.

For example, if $n = 8$ and $k = 1$, the block size would be 2, and the first stage would produce 4 vectors. This is as illustrated in the following diagram:



4.3 Matrix/Vector Multiplication

We now proceed to describe our matrix-vector multiplication implementation, namely implementing the operation $w \leftarrow Av$ where we consider w, v as column vectors. The vectors are always encrypted, and the matrix that could either be encrypted or in plaintext. The main difference between the two cases is in the cost of moving the data-movement operations that are required to move the matrix entries around. When the matrix is encrypted, it would be important how it is represented because a change of representation requires expensive data movement techniques. If it is in the clear, then we consider changing its representation to be almost for free. We consider first the case of an encrypted matrix, which is given to us in either row-major order or column-major order, and later consider the case of a plaintext matrix that we can represent in any convenient order.

Matrix in column-order. Assume that we are given the columns of the matrix as vectors of our underlying "platform", $A = (c_0 \mid \cdots \mid c_{n-1})$, so we have $Av = \sum_{i=0}^{n-1} v[i]c_i$. This suggests that we apply an algorithm for full replication to v, obtaining the vectors $v_0, \ldots, v_{n-1}$, and then compute $w \leftarrow \sum_{i=0}^{n-1} v_i \times c_i$. Using the HybridReplicate algorithm in §4.2.2, the running time of this algorithm will be $O(n)$ additions, multiplications, multiplicative masking, and rotations, and its depth is $O(n)$ additions, $O(\log n)$ rotations, $O(\log\log n)$ multiplicative masking, and a single multiplication.

Matrix in row-order. Another natural layout of A is where the rows of A are stored as vectors of the underlying "platform". In this case we could try to transpose the matrix A so as to be able use the $O(n)$ algorithm from above (or otherwise rearrange the entries of A), but this seem to require $O(n \log n)$ complexity. However, we can still get a linear-time algorithm, as follows. Suppose the rows of A are stored as vectors $r_0, \ldots, r_{n-1}$. We first compute the vectors $p_i = v \times r_i$ for $i \in [n]$. To complete the calculation, it remains to compute the entries of w by the rule $w[i] = \sum_j p_i[j]$ for $i \in [n]$. Viewing this mapping from $p_0, \ldots, p_{n-1}$ to w as a linear map, we may consider the $n \times n^2$ matrix that represents it. But observe: the transpose of this matrix represents the linear map corresponding to the replication problem; by the "transposition principle" [2, 3], this immediately gives us an algorithm with the same complexity as any of our algorithms for replication: the algorithm for the transposed problem simply runs the original in reverse, with fan-out and fan-in of addition exchanging roles, and rotations having their direction reversed, and masking operations unchanged.

Matrix in diagonal order. If the matrix is given to us in plaintext (or if we can pre-process it arbitrarily before the multiplication), then the best solution is to put it in diagonal order, which would lets us use the parallel "systolic" multiplication algorithm, cf. [13, Figure 1-35]. (As far as we know, the first usage of this method in the context of SIMD computation was in the implementations of Slasa20/ChaCha, see [1, Section 3].) We thank Daniel Bernstein for pointing out to us this method.

In detail, we represent the matrix by n vectors of the underlying "platform" $d_0, \dots, d_{n-1}$ that contain the generalized diagonals of A. Namely, $d_i = (A_{0,i}, A_{1,i+1}, \dots, A_{n-1,i-1})$, so $d_i[j] = A_{j,j+i}$, index arithmetic modulo n). Then the product $w = Av$ can be computed as $w \leftarrow \sum_{i=0}^{n-1} d_i \times (v \lll i)$, which takes n rotations, multiplications, and additions, and has depth of one multiplication, one rotation, and n additions. To see that this gives the right answer, note that the j'th entry in the result is $w[j] = \sum_{i=0}^{n-1} d_i[j] \cdot (v \lll i)[j] = \sum_{i=0}^{n-1} A_{j,j+i} \cdot v[j+i] = \sum_{k=0}^{n-1} A_{j,k} \cdot v[k]$, as needed.

4.4 Computing norms and traces

Recall that the individual plaintext slots in a HE ciphertext can hold elements from some finite field $\mathbb{F}_{p^d}$, and that the underlying HE "platform" gives us the Frobenius operations $\sigma^i(X) = X^{p^i}$ for $0 < i < d$ which is applied to all the slots in a SIMD manner. These operations have the same cost as the rotation operations, namely they are "expensive" in terms of running time but "cheap" in terms of added noise.

Below we describe how to use the Frobenius operations to compute the norms and traces of the elements in the slots. Recall that the norm and trace maps are defined as follows:

$$\begin{array}{lll} \text{Norm: } N : \mathbb{F}_{p^d} \to \mathbb{F}_p, & N(\alpha) := \prod_{i=0}^{d-1} \sigma^i(\alpha) \; = \; \prod_{i=0}^{d-1} \alpha^{p^i} \; = \; \alpha^{(p^d-1)/(p-1)}; \\ \text{Trace: } T : \mathbb{F}_{p^d} \to \mathbb{F}_p, & T(\alpha) := \sum_{i=0}^{d-1} \sigma^i(\alpha) \; = \; \sum_{i=0}^{d-1} \alpha^{p^i}. \end{array}$$

Computing traces and norms is often useful. For example, the "field switching" procedure of Gentry, Halevi, Peikert and Smart [9] relies on computing the trace. Also, computing the norm is useful in the (quite common) case where we need to compute the "not-equal-to-zero" function. That is, to map each non-zero slot to 1 while keeping the zero slots as zero, we just need to compute the function $N(X)^{p-1}$ (and in the special case $p = 2$ this is just the norm function itself).

Computing the norm and trace is done directly by their definitions above, as described in the following code:

```
Norm($v$):
1  $w \leftarrow v$
2  $e \leftarrow 1$
3  for $j \leftarrow$ numBits($d$) $- 2$ down to 0 do
4      $w \leftarrow w \times \sigma^e(w)$
5      $e \leftarrow 2 \cdot e$
6      if bit$_j(d) = 1$ then
7          $w \leftarrow v \times \sigma(w)$
8          $e \leftarrow e + 1$
9  return $w$
```

```
Trace($v$):
1  $w \leftarrow v$
2  $e \leftarrow 1$
3  for $j \leftarrow$ numBits($d$) $- 2$ down to 0 do
4      $w \leftarrow w + \sigma^e(w)$
5      $e \leftarrow 2 \cdot e$
6      if bit$_j(d) = 1$ then
7          $w \leftarrow v + \sigma(w)$
8          $e \leftarrow e + 1$
9  return $w$
```

The running time and depth of the norm computation is $O(\log d)$ Frobenius powers and multiplications, and that of the trace computation is $O(\log d)$ Frobenius powers and additions.

Cyclotomic field	Vector size	Operation	Time
$m = 4369$	$n = 256$	One-Entry Replication	0.3 sec
		Full Replication	24.8 sec
		Matrix multiply	25.7 sec
$m = 8191$	$n = 630$	One-Entry Replication	0.9 sec
		Full Replication	192 sec
		Matrix multiply	84.3 sec
$m = 21845$	$n = 1024$	One-Entry Replication	3.2 sec
		Full Replication	800 sec
		Matrix multiply	473 sec

Table 3: Timing results for some operations in various vector sizes.

Performance results. An illustrative timing results for some settings of the parameters are given in Table 3.

5 Hypercubes in the Underlying "Platform"

So far in this report we assumed that the underlying HE "platform" provides us with operations that rotate (or shift) the plaintext slots as if they are in a linear array. This assumption is not accurate, however. As described in [10], in general the "platform" gives as a multi-dimensional hypercube with rotations along each dimension separately. For some parameters we could get a one-dimensional array, while for others there are more dimensions.[3] Moreover, not all these dimensions are created equal, for some of them we indeed get rotation operations while for others we only get shift operations (with "garbage-filling"). Of course we can always implement shift with zero-filling using multiplicative masking and then implement rotations using two zero-fill shifts and addition, but this incurs a moderate cost in terms of noise and a factor of two in running time. Below we call dimensions where we get true rotations *good dimensions* and the others are *bad dimensions*.

It is possible to implement linear-array rotations and shifts using the "native" rotations along the different dimensions, and then use the resulting linear rotations in the permutations and linear algebra routines from above. However this implementation incurs a slowdown factor equals to the number of dimensions, in terms of both noise and running time. Below we first show how to implement linear rotations and shifts using the native operations, and then describe more efficient ways of implementing permutations and linear algebra directly from the hypercube structure, without going through the linear-array implementation.

5.1 Implementing Linear Rotations and Shifts

Cyclic rotations. The cyclic right-rotation procedure moves the content of the j'th slot to slot $j + k \bmod n$. Hence we can view this operation as adding k to the index of each plaintext slot, all in parallel. To implement this operation we use a concurrent version of the grade-school addition-with-carry procedure, using the native rotation-along-a-single-dimension operations.

We think of the rotation amount k (and the indexes) as they are represented in the base corresponding to the sizes of the different dimensions. That is, we number the entries of the hypercube in lexicographic order, and identify the integer k with the k'th entry in this order. Let r be the

[3]If the native plaintext space of the HE scheme is $\mathbb{Z}[X]/(\Phi_m(X), p)$ then the number of dimensions that we get is the number of elements in a generating set of the quotient group $Z_m^*/(p)$.

number of dimensions and f_i be the size of the i'th dimension. Then the integer k is represented by the vector $\vec{e}^{(k)} = (e_1^{(k)}, \ldots, e_r^{(k)})$, where $e_i^{(k)} \in [0, f_i)$ is the index along the i'th dimension of the k'th lexicographic entry of the hypercube. (We also represent each index j in the same fashion.) We can now think of rotation by k as adding the multi-precision vector $\vec{e}^{(k)}$ to all the vectors $\vec{e}^{(j)}$, $j = 1, \ldots, r$ in parallel.

Beginning with the least-significant digit in these vectors, we use rotate-by-$e_r^{(k)}$ along the r'th dimension to implement the operation of $e_r^{(j)} \leftarrow e_r^{(j)} + e_r^{(k)} \bmod f_r$ for all j at once. Moving to the next digit, we now have to add to each $e_{r-1}^{(j)}$ either $e_{r-1}^{(k)}$ or $1 + e_{r-1}^{(k)}$, depending on whether or not there was a carry from the previous position. To do that, we compute two rotation amount along the $(r-1)$'th dimension, by $e_{r-1}^{(k)}$ and $1 + e_{r-1}^{(k)}$, then use a MUX operation to choose the right rotation amount for every slot. Namely, indexes j for which $e_r^{(j)} \geq f_r - e_r^{(k)}$ (so we have a carry) are taken from the copy that was rotated by $1 + e_{r-1}^{(k)}$, while other indexes j are taken from the copy that was rotated by $e_{r-1}^{(k)}$.

The MUX operation is implemented by preparing a constant mask (denoted `mask`) that has 1's in the slots corresponding to indexes $(e_1, \ldots, e_r)$ with $e_r \geq f_r - e_r^{(k)}$ and 0's in all the other slots, then computing $\vec{v}' = \vec{v}_1 \times \texttt{mask} + \vec{v}_2 \times (1 - \texttt{mask})$, where $\vec{v}_1, \vec{v}_2$ are the two cubes generated by rotation along dimension $r-1$ by $1 + e_{r-1}^{(k)}$ and $e_{r-1}^{(k)}$, respectively.

We then move to the next digit, preparing a mask for those j's for which we have a carry into that position, then rotating by $1 + e_{r-2}^{(k)}$ and $e_{r-2}^{(k)}$ along the $(r-2)$'nd dimension and using the mask to do the MUX between these two ciphertexts. We proceed in a similar manner until the most significant digit. To complete the description of the algorithm, note that the mask for processing the i'th digit is computed as follows: For each index j, which is represented by the vector $(e_1^{(j)} \ldots, e_i^{(j)}, \ldots e_r^{(j)})$, we have $\texttt{mask}_i[j] = 1$ if either $e_i^{(j)} \geq f_i - e_i^{(k)}$, or if $e_i^{(j)} = f_i - e_i^{(k)} - 1$ and $\texttt{mask}_{i-1}[j] = 1$ (i.e. we had a carry from position $i-1$ to position i). Hence the rotation procedure works as follows:

```
Rotate($\vec{v}, k$):
1   let $(e_1^{(k)}, \ldots, e_r^{(k)})$ be the $k$th vector in lexicographic order.
2   $M_n \leftarrow$ all-1 mask
3   rotate $\vec{v}$ by $e_r^{(k)}$ along the $r$th dimension
4   for $i \leftarrow r-1$ down to 1
5       $M_i' \leftarrow 1$ in the slots $j$ with $e_{i+1}^{(j)} \geq f_{i+1} - e_{i+1}^{(k)}$,  0 in all the other slots
6       $M_i'' \leftarrow 1$ in the slots $j$ with $e_{i+1}^{(j)} = f_{i+1} - e_{i+1}^{(k)} - 1$,  0 in all the outer slots
7       $M_i \leftarrow M_i' + M_i'' \times M_{i+1}$   // The $i$th mask
8       $\vec{v}\,' \leftarrow$ rotate $\vec{v}$ by $e_i^{(k)}$ along the $i$th dimension
9       $\vec{v}\,'' \leftarrow$ rotate $\vec{v}$ by $1 + e_i^{(k)}$ along the $i$th dimension
10      $\vec{v} \leftarrow \vec{v}\,'' \times M_i + \vec{v}\,' \times (1 - M_i)$
11  return $\vec{v}$
```

Zero-fill shifts. A non-cyclic zero-fill shift of the linear array by k positions, is implemented very similarly, except that some positions should be set to zero. It turns out that the only change from the rotation-by-k procedure is the last step, where we deal with the most significant digit. For a positive $k > 0$, every slot $j \geq k$ gets the content of slot $j - k$ and every slot $j < k$ gets zero. For a negative $k < 0$, every slot $j < n - |k|$ gets the content of slot $j + |k|$, and every slot $j \geq n - |k|$ gets zero (with n the number of slots).

For $k > 0$, this procedure is implemented very similarly to the `rotate` procedure above, except that in the last iteration (processing the most-significant digit) we replace the operation of rotate-by-$e_1^{(k)}$ along the 1'st dimension by shift-by-$e_1^{(k)}$ along the 1'st dimension (and similarly use shift-by-

$(1+e_1^{(k)})$ rather than rotate-by-$(1+e_1^{(k)})$). For a negative amount $-n < k < 0$, we use the same procedure upto the last iteration with amount $n+k$, and in the last iteration use shift-by-e' and shift-by-$(1+e')$ along the 1st dimension, for the negative number $e' = e_1^{(k)} - f_1$.

5.2 Permuting the Hypercube

Since the permutation implementation from Section 3 already includes hypercube networks, then implementing it on top of a hypercube "platform" entails almost no changes. The main difference from the optimization formula at the bottom of Section 3.5 is that we are forced to use the factorization of n as a hypercube $n = n_1 n_2 \cdots n_r$ (but we can further factorize each of the n_i's). Also, in the optimization problem we need to minimize the reduced cost for networks corresponding to the good dimensions, and minimize the unreduced cost for the bad dimensions.

5.3 Linear Algebra on the Hypercube

Sums. The total sums algorithm in Section 4.1 is better implemented on a hypercube by running that algorithm once for each dimension, with n replaced by the size of a given dimension, and the rotation being replace by a rotation in that dimension. The result is an algorithm whose running time and depth on a hypercube is essentially the same as that of the original algorithm on a linear array, without any slowdown factor related the dimension of the hypercube.

Unfortunately, the same cannot be said about the partial-sums procedure, over there there does not appear to be a better implementation than going through the linear shifts as in Section 5.1.

Replication. The algorithm in Section 4.2 for replicating a single value can also be easily adapted so that it works one dimension at a time, using one-dimensional rotations in each dimension. The same holds for the full replication algorithms discussed in that section; however, in the shallower full-replication procedure, the actual implementation in HElib uses a heuristic to choose the switch-over parameter from the naive to the recursive algorithm in each dimension.

Matrix-vector multiplication. The diagonal-based algorithm for matrix-vector multiplication can also be implemented directly using one-dimensional rotations on a hypercube. The idea is that instead of working with the n rotations of a vector viewed as a linear array, we work with the n permutations obtained by composing all possible rotations in each dimension. This set of permutations is "sharply transitive", meaning that for every i, j in the domain, there is a unique permutation π in the set such that $\pi(i) = j$. It turns out that this property is sufficient to guarantee correctness.

If the hypercube has dimensions $(n_1, \ldots, n_r)$, then running time of the resulting algorithm is n multiplications and additions, and $n + n/n_r + n/(n_r n_{r-1}) + \cdots$ rotations along the different dimensions. This gives $2n$ rotations in the worst case, but usually much less. Actually, the dimensions should be sorted to optimize the number of rotations, with all bad dimensions coming before good ones, and otherwise in ascending order of size. The depth is 1 constant multiplication, r rotations along the different dimensions, and n additions.

References

[1] D. J. Bernstein. ChaCha, a variant of Salsa20. Workshop Record of SASC '08: The State of the Art of Stream Ciphers, 2008. `http://cr.yp.to/papers.html#chacha`.

[2] J. L. Bordewijk. Inter-reciprocity applied to electrical networks. *Applied Scientific Research B: Electrophysics, Acoustics, Optics, Mathematical Methods*, 6:1–74, 1956.

[3] A. Bostan, G. Lecerf, and E. Schost. Tellegen's principle into practice. In *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, ISSAC '03, pages 37–44, New York, NY, USA, 2003. ACM.

[4] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

[5] Z. Brakerski, C. Gentry, and S. Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In K. Kurosawa and G. Hanaoka, editors, *Public Key Cryptography*, volume 7778 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2013.

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

[7] C. Chang and R. Melhem. Arbitrary size benes networks. *Parallel Processing Letters*, 07(03):279–284, 1997.

[8] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing – STOC 2009*, pages 169–178. ACM, 2009.

[9] C. Gentry, S. Halevi, C. Peikert, and N. P. Smart. Field switching in BGV-style homomorphic encryption. *Journal of Computer Security*, 21(5):663–684, 2013.

[10] C. Gentry, S. Halevi, and N. Smart. Fully homomorphic encryption with polylog overhead. In *"Advances in Cryptology - EUROCRYPT 2012"*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. Full version at `http://eprint.iacr.org/2011/566`.

[11] S. Halevi and V. Shoup. HElib - An Implementation of homomorphic encryption. `https://github.com/shaih/HElib/`, Accessed Feb 2014.

[12] J. Hoffstein, J. Pipher, and J. H. Silverman. Ntru: A ring-based public key cryptosystem. In J. Buhler, editor, *ANTS*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.

[13] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 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] A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*, pages 1219–1234, 2012.

[16] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In H. Gilbert, editor, *Advances in Cryptology - EUROCRYPT'10*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[17] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–177. Academic Press, 1978.

[18] N. P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. Manuscript at http://eprint.iacr.org/2011/133, 2011.

[19] SIMD. Wikipedia article. `http://en.wikipedia.org/wiki/SIMD`, accessed Feb 2014.