Algorithms in HElib

Shai Halevi[1] and Victor Shoup[1,2]

[1] IBM Research, Yorktown Heights, NY, USA
[2] Newyork University, Newyork, NY, USA

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 to Intel SSE and the like), but with 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, linear algebra, and other operations over this "platform."

1 Introduction

Homomorphic encryption (HE) [18,8] 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-variants of Brakerski-Gentry-Vaikuntanathan scheme [6] and Brakerski's scale-invariant scheme [4], and the NTRU-based scheme [13,16]. 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 [19] 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 entry-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 [17,6].

J.A. Garay and R. Gennaro (Eds.): CRYPTO 2014, Part I, LNCS 8616, pp. 554–571, 2014.

Turning to software implementations, HElib [12] 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 operations, and also some higher-level procedures implementing the GHS data-movement procedures, simple linear algebra, and some other procedures. 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 entry-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.

A word of caution: The view of the HE "platform" as a linear array is oversimplified, and presented here for sake of readability. In reality we have something closer to a multi-dimensional array (and even this view hides some details) — see the full version [11, Section 5].

Although SIMD hardware architectures are quite common in practice (cf. [20]), 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 introduce notations and describe some details of the HE "platform." In Section 3 we describe the implementation and optimizations of the GHS permutation techniques. In particular we describe 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, and in Section 5 we describe procedures for computing the norm and trace functions entry-wise on a vector

In the full version [11] we describe a procedure for evaluating a polynomial entry-wise on a vector. We also discuss how to adapt all of these procedures to work on a multi-dimensional array (which is what more naturally arises in the HE context), rather than a linear array.

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 [13,16], 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 the full version [11, 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

Table 1. Homomorphic operations and their cost

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

entries but not others. We refer to this operation as *multiplicative masking* (or *masking*, for short). We also note that using rotations 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.

Notations. Throughout this report we use $[n]$ for the set $\{0\,..\,n-1\}$, 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 $m_\delta \times v$), shift the result by δ positions, and 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

$$w \leftarrow \sum_{\delta \in sh_\pi \bmod n} (m'_\delta \times v) \ggg \delta$$

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 $\leq B$, 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 optimizations.

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 $i - 2^k$. 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 independent permutations on the top and bottom halves 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 i or $i + m$, and each $i \in S_1$ to i or $i - m$;
(P2) ρ consists of a permutation on S_0 and a permutation on S_1.

We construct an undirected graph G with $2n$ nodes, L_i, R_i for $i \in [n]$ (call these "left nodes" and "right nodes"); we add an edge from L_i to $R_{\pi(i)}$ for each $i \in [n]$ (call these "permutation edges"), and also an edge from L_i to L_{i+m} and R_i to R_{i+m} for each $i < m$ (call these "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, and trace out a path that must lead back to the starting node, alternating between permutation and conflict edges. This creates an even cycle that we can color with two colors, which we then remove from G; we then 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 a color of 1 as τ sending i to the bottom half. So we have $\tau(i) := i$ if $i \in S_0$ and $C(L_i) = 0$, or if $i \in S_1$ and $C(L_i) = 1$, and otherwise $\tau(i) := i \pm m$.

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

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

Setting the permutations τ and σ determines also the middle permutation ρ (which must satisfy property (P2)) and we 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

When n is not a power of two, we could, of course, round n to the next power of two and then apply a Benes network; however, this would effectively double the cost of implementing a permutation in our setting. 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 $\lfloor n/2 \rfloor$, and 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 using the same rules as above, except that we add a special conflict edge between L_{n-1} and R_{n-1} (note that none of the other conflict edges are incident to either of these two nodes). The rest of the algorithm works without any change, and correctness follows from the exact same argument.

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, setting the shift amount to $m = \lceil n/2 \rceil$ and fixing $\sigma(m-1) = \tau(m-1) = m-1$ 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 sub-networks 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" $\Delta_k := \left\lceil \lfloor n/2^{r-1-k} \rfloor / 2 \right\rceil$. Thus, 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;

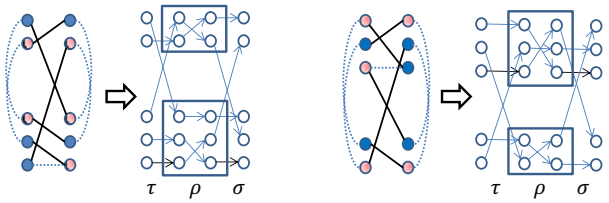


Fig. 1. Illustration of two ways to decompose a size-5 permutation as $\pi = \tau \circ \rho \circ \sigma$

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 $k = j_1 \,..\, j_2$. 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

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

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 (when $\gcd(a, b) = 1$**):** $M(i) \mapsto (i \bmod a, i \bmod b)$;
Row-major order: M maps $0 \,..\, b-1$ to the first row, $b \,..\, 2b-1$ to the second row, etc;
column-major order: M maps $0 \,..\, a-1$ to the first column, $a \,..\, 2a-1$ 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 [15]:

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

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 every permutation π over S, there exist 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.*

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 lay out 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 $b\Delta i + a\Delta j$ 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

[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].

form $(0, \Delta j)$, which translates to Δj; 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 reduced costs for $\bar{\sigma}, \bar{\tau}$ are equal to the *unreduced costs* for σ, τ, while for $\bar{\rho}$ we get the reduced cost of 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 size-n network with depth-bound B, and similarly let $\mathsf{BenesRcost}(n, B), \mathsf{BenesUcost}(n, B)$ be the best (reduced/unreduced) cost of a generalized Benes for these parameters. Then we have:

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

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

Since there are only polynomially many (n, B) pairs, we can again use dynamic programming to solve these recurrences 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.

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.

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.

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∗

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. Here, we denote by $\mathrm{NumBits}(n)$ 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 iteration j, the binary representation of e consists of bits $j \,..\, \mathrm{NumBits}(n) - 1$ of n.

RS(v):
1 $w \leftarrow v$, $e \leftarrow 1$
2 while $e < n$ do
3 $\quad w \leftarrow w + (w \gg e)$, $e \leftarrow 2 \cdot e$
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 $\quad w \leftarrow w + (w \ggg e)$, $e \leftarrow 2 \cdot e$
4 $\quad$ if $\mathrm{bit}_j(n) = 1$ then
5 $\quad\quad w \leftarrow v + (w \ggg 1)$, $e \leftarrow e + 1$
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 requires that we replicate some plaintext values many times. We have not (yet) implemented a completely generic replication method (such as the ones from [10]),

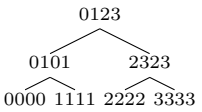
but we describe procedures that we did implement for efficient replication in a few interesting special cases.

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.

Full Replication. In full replication, we take a vector v and produce vectors $\{w_i\}_{i\in[n]}$ such that each 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 a depth of $O(\log n)$ "cheap" additions and rotations and one "moderate" masking.

We can do better than this. We begin by describing a faster, simple recursive procedure that uses just $O(n)$ additions, rotations, and masks, but has a depth of $O(\log n)$ multiplicative masking operations. We then we present a hybrid algorithm with the same linear running time, but with a masking depth of just $O(\log \log n)$.

A simple recursive procedure. Consider first the case of $n = 2^\ell$, where it is easy to apply a simple divide-and-conquer approach: 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 $h = 0\,..\,\ell$ (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 recursively, but 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 $mask[i] \leftarrow \mathrm{bit}_{h-1}(i)$ 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 not too difficult to adapt this procedure for the case where n is not a power of 2. 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.

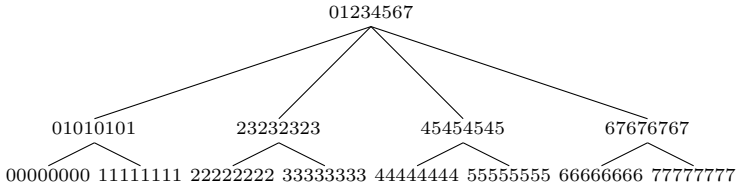
One easily verifies that the running time of this algorithm is $O(n)$ additions, rotations, and multiplicative masking; its depth is $O(\log n)$ additions, rotations, and masking.

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 a flatter but more time-consuming procedure (similar to the naive solution from the beginning of Section 4.2), 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 $i2^k \,..\, (i+1)2^k - 1$. In the first stage of the algorithm we use a "naive procedure," similar to the single-entry replication, to construct vectors v_i, $i = 0 \,..\, n/2^k - 1$, 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 could either be encrypted or in plaintext. The main difference between the two cases is the cost of the operations that are required to move the matrix entries around. When the matrix is encrypted, its representation (column-, row-, or diagonal-order) may have a significant impact on the cost of these data movement operations. If it is in the clear, we can use the most convenient representation.

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, 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. It turns out that the most convenient representation of the matrix is diagonal order, which lets us use the parallel "systolic" multiplication algorithm, cf. [14, Figure 1-35]. Certainly, if the matrix is given to us in the clear, this is the representation of choice. As far as we know, the first usage of this method in the context of SIMD computation was in the implementations of Salsa20/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" $d_0, \ldots, d_{n-1}$ that contain the generalized diagonals of A, namely, $d_i = (A_{0,i}, A_{1,i+1}, \ldots, A_{n-1,n+i-1})$, so $d_i[j] = A_{j,j+i}$ (where index arithmetic is 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 a 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 Performance Results

An illustrative timing results for some settings of the parameters are given in Table 3. These tests were run on a five-year-old IBM BladeCenter HS22/7870, with two Intel X5570 (4-core) processors, running at 2.93GHz. However, since HElib is (currently) not thread safe, these tests only utilized one of the eight cores available on that machine. As the operations in these procedures are "embarrassingly parallelizable" we expect that a thread-safe implementation would be about eight times faster on the same machine.

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

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

5 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 $i = 0 \,..\, d-1$, 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 ← v
2  e ← 1
3  for j ← NumBits(d) − 2 down to 0 do
4        w ← w × σ^e(w)
5        e ← 2 · e
6        if bit_j(d) = 1 then
7              w ← v × σ(w)
8              e ← e + 1
9  return w
```

```
Trace(v):
1  w ← v
2  e ← 1
3  for j ← NumBits(d) − 2 down to 0 do
4        w ← w + σ^e(w)
5        e ← 2 · e
6        if bit_j(d) = 1 then
7              w ← v + σ(w)
8              e ← 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.

Acknowledgments. 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.

References

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

2. Bordewijk, J.L.: Inter-reciprocity applied to electrical networks. Applied Scientific Research B: Electrophysics, Acoustics, Optics, Mathematical Methods 6, 1–74 (1956)
3. Bostan, A., Lecerf, G., Schost, E.: Tellegen's principle into practice. In: Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation, ISSAC 2003, pp. 37–44. ACM (2003)
4. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Heidelberg (2012)
5. Brakerski, Z., Gentry, C., Halevi, S.: Packed ciphertexts in LWE-based homomorphic encryption. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 1–13. Springer, Heidelberg (2013)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. In: Innovations in Theoretical Computer Science, ITCS 2012 (2012), http://eprint.iacr.org/2011/277
7. Chang, C., Melhem, R.: Arbitrary size benes networks. Parallel Processing Letters 07(03), 279–284 (1997)
8. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st ACM Symposium on Theory of Computing – STOC 2009, pp. 169–178. ACM (2009)
9. Gentry, C., Halevi, S., Peikert, C., Smart, N.P.: Field switching in BGV-style homomorphic encryption. Journal of Computer Security 21(5), 663–684 (2013)
10. Gentry, C., Halevi, S., Smart, N.: Fully homomorphic encryption with polylog overhead. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 465–482. Springer, Heidelberg (2012), Full version at http://eprint.iacr.org/2011/566
11. Halevi, S., Shoup, V.: Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106 (2014), http://eprint.iacr.org/
12. Halevi, S., Shoup, V.: HElib - An Implementation of homomorphic encryption (accessed February 2014), https://github.com/shaih/HElib/
13. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 267–288. Springer, Heidelberg (1998)
14. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann Publishers Inc., San Francisco (1992)
15. Lev, G., Pippenger, N., Valiant, L.: A fast parallel algorithm for routing in permutation networks. IEEE Transactions on Computers C-30, 93–100 (1981)
16. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: STOC, pp. 1219–1234 (2012)
17. 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)
18. Rivest, R., Adleman, L., Dertouzos, M.: On data banks and privacy homomorphisms. In: Foundations of Secure Computation, pp. 169–177. Academic Press (1978)
19. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. Designs, Codes and Cryptography 71(1), 57–81 (2014)
20. SIMD. Wikipedia article (accessed February 2014), http://en.wikipedia.org/wiki/SIMD