Bootstrapping for HElib

Shai Halevi (IBM) Victor Shoup*(NYU)

October 22, 2014

Abstract

Gentry's bootstrapping technique is still the only known method of obtaining fully homomorphic encryption where the system's parameters do not depend on the complexity of the evaluated functions. Bootstrapping involves a *recryption* procedure where the scheme decryption is evaluated homomorphically. So far, there have been precious few implementations of recryption, and none that could handle "packed ciphertexts" that encrypt vectors of elements.

In the current work we implemented recryption of fully-packed ciphertexts using the HElib library for somewhat-homomorphic encryption. This required extending the recryption algorithms from the literature, as well as many aspects of the HElib library.

Our implementation supports bootstrapping of packed ciphertexts over many extension fields/rings. One example that we tested involves ciphertexts that encrypt vectors of 1024 elements from GF(2^{16}). In that setting, the recryption procedure takes under 5.5 minutes (at security-level ≈ 76) on a single core, and allows a depth-9 computation before the next recryption is needed.

Keywords: Bootstrapping, Homomorphic Encryption, Implementation

*Work partially done in IBM Research.

Contents

1 Introduction

Homomorphic Encryption (HE) [23, 7] enables computation of arbitrary functions on encrypted data without knowing the secret key. All current HE schemes follow Gentry's outline from [7], where fresh ciphertexts are "noisy" to ensure security and this noise grows with every operation until it overwhelms the signal and causes decryption errors. This yields a "somewhat homomorphic" scheme (SWHE) that can only evaluate low-depth circuits, which can then be converted to a "fully homomorphic" scheme (FHE) using bootstrapping. Gentry described a *recryption* operation, where the decryption procedure of the scheme is run homomorphically, using an encryption of the secret key that can be found in the public key, yielding in a new ciphertext that encrypts the same plaintext but has smaller noise.

The last few years saw a large body of work improving many aspects of homomorphic encryption in general (e.g., [4, 10, 17, 3, 13, 5]) and recryption in particular [12, 1, 2, 20, 6]. However, so far, only a few implementations of SWHE have been reported, and almost none of them supports recryption. Prior to the current work, we are aware of only two reported implementations of recryption: the implementation due to Gentry and Halevi [8] of Gentry's original cryptosystem [7], and the implementation due to Rohloff and Cousins [24] of the NTRU-based cryptosystem [16, 17].

In this paper we report on our new implementation of recryption for the cryptosystem of Brakerski, Gentry and Vaikuntanathan (BGV) [4]. We implemented recryption on top of the open-source library `HElib` [15, 14], which implements the ring-LWE variant of BGV. Our implementation includes both new algorithmic designs as well as re-engineering of some aspects of `HElib`. As noted in [14], the choice of homomorphic primitives in `HElib` was guided to a large extent by the desire to support recryption, but nonetheless in the course of our implementation we had to extend the implementation of some of these primitives (e.g., matrix-vector multiplication), and also implement a few new ones (e.g., polynomial evaluation).

The `HElib` library is "focused on effective use of the Smart-Vercauteren ciphertext packing techniques [26] and the Gentry-Halevi-Smart optimizations [10]," so in particular we implemented recryption for "fully-packed" ciphertexts. Namely, our implementation supports recryption of ciphertexts that encrypt vectors of elements from extension fields (or rings). Importantly, our recryption procedure itself is of very low depth, so as to allow significant processing between recryptions while keeping the lattice dimension reasonable to maintain efficiency.

For example, one parameter-setting that we tested has ciphertexts that encrypt vectors of 1024 elements from $\mathrm{GF}(2^{16})$. In our tests (for effective security level of 76 bits), recryption of such a ciphertext takes 320 seconds (under 5.5 minutes) on a single CPU core, and allows additional computations of depth 9 between recryptions. That setting can be conducive for implementing homomorphic AES as in [11], since it allows embedding $\mathrm{GF}(2^8)$ elements in the slots and using a byte-oriented implementation (thus embedding 64 AES blocks in each ciphertext). We estimate that one would have to recrypt five times during such AES computation (but we have not yet attempted this homomorphic AES implementation).

Compared to the two previous recrypt implementations, ours offers many advantages in both flexibility and speed. For flexibility, the previous recryption implementations only apply to non-packed ciphertexts that encrypt one bit per ciphertext, whereas ours supports packed ciphertexts that encrypt vectors from "any extension field/ring."[1] Some examples that we tested include vectors over the fields $\mathrm{GF}(2^{16})$, $\mathrm{GF}(2^{25})$, $\mathrm{GF}(2^{24})$, $\mathrm{GF}(2^{36})$, $\mathrm{GF}(17^{40})$, and $\mathrm{GF}(127^{36})$, as well as a degree-21 extension and a degree-30 extension of the ring $\mathbb{Z}_{256}$.

In terms of speed, the Gentry-Halevi implementation reportedly took 1/2-hour on a single core

[1] Some internals of `HElib` bound the characteristic of the supported fields/rings below one million or so. And even before hitting these limits, trying to recrypt ciphertexts relative to high characteristic fields/rings or with very high extension degree could be very slow.

to recrypt a single-bit ciphertext, and the Rohloff-Cousins implementation reported recryption time of 275 seconds for a single-bit ciphertext with 64-bit security on 20 cores. For our implementation, the tests that we ran clocked a single-core time of 320 seconds and up for packed ciphertexts with security levels of 76 bits and up (see Section 6). We note that the same parallelism that was used by Rohloff and Cousins could in principle be applied to our implementation too, but doing so involves several software-engineering challenges, and we have not yet attempted it.

Concurrent work. Concurrently with out work, Ducas and Micciancio described a new bootstrapping procedure [6]. This procedure is applied to Regev-like ciphertexts [22] that encrypt a single bit, using a secret-key encrypted similarly to the new cryptosystem of Gentry at al. [13]. They reported on an implementation of their scheme, where they can perform a NAND operation followed by recryption in less than a second. Compared to our scheme, theirs has the advantage of a much faster wall-clock time for recryption, but they do not support batching or large plaintext spaces (hence our implementation has much better amortized per-bit timing). It is a very interesting open problem to combine their techniques with ours, achieving a "best of both worlds" implementation.

1.1 Algorithmic Aspects

Our recryption procedure follows the high-level structure introduced by Gentry et al. [12], and for the required linear transformations is makes use of the tensor decomposition of Alperin-Sheriff and Peikert [1]. However those two works only dealt with characteristic-2 plaintext spaces so we had to extend some of their algorithmic components to deal with characteristics $p > 2$ (see Section 5).

Also, to get an efficient implementation, we had to make the decomposition from [1] explicit, specialize it to cases that support very-small-depth circuits, and align the different representations to reduce the required data-movement and multiplication-by-constant operations. These aspects are described in Section 4. One significant difference between our implementation and the procedure of Alperin-Sheriff and Peikert [1] is that we *do not use* the ring-switching techniques of Gentry et al. [9] (see discussion in Appendix C).

Organization. We describe our notations and give some background information on the BGV cryptosystem and the `HElib` library in Section 2. In Section 3 we provide an overview of the high-level recryption procedure from [12] and our variant of it. We then describe in detail our implementation of the linear transformations in Section 4 and the non-linear parts in Section 5. In Section 5.4 we explain how all these parts are put together in our implementation, and in Section 6 we discuss our performance results. We conclude with directions for future work in Section 7. In Appendix A we describe our choice of parameters, and in Appendix B we describe our implementation of homomorphic polynomial evaluation, which is used in the non-linear part of the recryption procedure for $p > 2$.

Disclaimer. Our recryption implementation is not yet available in open-source. We plan to clean up the code and then make it available as part of `HElib` within the next month or two.

2 Notations and Background

For an integer z, we denote by $[z]_q$ the reduction of z modulo q into the interval $[-q/2, q/2)$, except that for $q = 2$ we reduce to $(-1, 1]$. This notation extends to vectors and matrices coordinate-wise, and to elements of other algebraic groups/rings by reducing their coefficients in some convenient basis.

For an integer z (positive or negative) we consider the base-p representation of z and denote its digits by $z\langle 0\rangle_p,\ z\langle 1\rangle_p, \cdots$. When p is clear from the context we omit the subscript and just write

$z\langle 0\rangle, z\langle 1\rangle, \cdots$. When $p = 2$ we consider a 2's-complement representation of signed integers (i.e., the top bit represent a large negative number). For an odd p we consider balanced mod-p representation where all the digits are in $[-\frac{p-1}{2}, \frac{p-1}{2}]$.

For indexes $0 \leq i \leq j$ we also denote by $z\langle j,\ldots,i\rangle_p$ the integer whose base-p expansion is $z\langle j\rangle \cdots z\langle i\rangle$ (with $z\langle i\rangle$ the least significant digit). Namely, for odd p we have $z\langle j,\ldots,i\rangle_p = \sum_{k=i}^{j} z\langle k\rangle p^{k-i}$, and for $p = 2$ we have $z\langle j,\ldots,i\rangle_2 = (\sum_{k=i}^{j-1} z\langle k\rangle 2^{k-i}) - z\langle j\rangle 2^{j-i}$. The properties of these representations that we use in our procedures are the following:

- For any $r \geq 1$ and any integer z we have $z = z\langle r-1,\ldots,0\rangle \pmod{p^r}$.
- If the representation of z is $d_{r-1},\ldots,d_0$ then the representation of $z \cdot p^r$ is $d_{r-1},\ldots,d_0,\overbrace{0,\cdots,0}^{r \text{ zeros}}$.
- If p is odd and $|z| < p^e/2$ then the digits in positions e and up in the representation of z are all zero.
- If $p = 2$ and $|z| < 2^{e-1}$, then the bits in positions $e-1$ and up in the representation of z, are either all zero if $z \geq 0$ or all one if $z < 0$.

2.1 The BGV Cryptosystem

The BGV ring-LWE-based somewhat-homomorphic scheme [4] is defined over a ring $R \overset{\text{def}}{=} \mathbb{Z}[X]/(\Phi_m(X))$, where $\Phi_m(X)$ is the mth cyclotomic polynomial. For an arbitrary integer modulus N (not necessarily prime) we denote the ring $R_N \overset{\text{def}}{=} R/NR$. We often identify elements in R (or R_N) with their representation is some convenient basis, e.g., their coefficient vectors as polynomials. When dealing with R_N, we assume that the coefficients are in $[-N/2, N/2)$ (except for R_2 where the coefficients are in $\{0,1\}$). We discuss these representations in some more detail in Section 4.1. The norm of an element $\|a\|$ is defined as its norm in some convenient basis.[2]

As implemented in HElib, the native plaintext space of the BGV cryptosystem is R_{p^r} for a prime power p^r. The scheme is parametrized by a sequence of decreasing moduli $q_L \gg q_{L-1} \gg \cdots \gg q_0$, and an "$i$th level ciphertext" in the scheme is a vector $\mathsf{ct} \in (R_{q_i})^2$. Secret keys are elements $\mathfrak{s} \in R$ with "small" coefficients (chosen in $\{0, \pm 1\}$ in HElib), and we view $\mathfrak{s}$ as the second element of the 2-vector $\mathsf{sk} = (1, \mathfrak{s}) \in R^2$. A level-$i$ ciphertext $\mathsf{ct} = (\mathfrak{c}_0, \mathfrak{c}_1)$ encrypts a plaintext element $\mathfrak{m} \in R_{p^r}$ with respect to $\mathsf{sk} = (1, \mathfrak{s})$ if we have $[\langle \mathsf{sk}, \mathsf{ct}\rangle]_{q_i} = [\mathfrak{c}_0 + \mathfrak{s} \cdot \mathfrak{c}_1]_{q_i} = \mathfrak{m} + p^r \cdot \mathfrak{e}$ (in R) for some "small" error term, $p^r \cdot \|\mathfrak{e}\| \ll q_i$.

The error term grows with homomorphic operations of the cryptosystem, and switching from q_{i+1} to q_i is used to decrease the error term roughly by the ratio q_{i+1}/q_i. Once we have a level-0 ciphertext ct, we can no longer use that technique to reduce the noise. To enable further computation, we need to use Gentry's bootstrapping technique [7], whereby we "recrypt" the ciphertext ct, to obtain a new ciphertext ct^* that encrypts the same element of R_{p^r} with respect to some level $i > 0$.

2.2 Encoding Vectors in Plaintext Slots

As observed by Smart and Vercauteren [26], an element of the native plaintext space $\alpha \in R_{p^r}$ can be viewed as encoding a vector of "plaintext slots" containing elements from some smaller ring extension of $\mathbb{Z}/(p^r)$ via Chinese remaindering. In this way, a single arithmetic operation on α corresponds to the same operation applied component-wise to all the slots.

Specifically, suppose the factorization of $\Phi_m(X)$ modulo p^r is $\Phi_m(X) \equiv F_1(X) \cdots F_k(X) \pmod{p^r}$, where each F_i has the same degree d, which is equal to the order of p modulo m. (This

[2] The difference between the norm in the different bases is not very important for the current work.

factorization can be obtained by factoring $\Phi_m(X)$ modulo p and then Hensel lifting.) From the CRT for polynomials, we have the isomorphism

$$R_{p^r} \simeq \bigoplus_{i=1}^{k} (\mathbb{Z}[X]/(p^r, F_i(X)).$$

Let us now define $E \overset{\text{def}}{=} \mathbb{Z}[X]/(p^r, F_1(X))$, and let ζ be the residue class of X in E, which is a principal mth root of unity, so that $E = \mathbb{Z}/(p^r)[\zeta]$. The rings $\mathbb{Z}[X]/(p^r, F_i(X))$ for $i = 1, \ldots, k$ are all isomorphic to E, and their direct product is isomorphic to R_{p^r}, so we get an isomorphism between R_{p^r} and E^k. HElib makes extensive use of this isomorphism, representing it explicitly as follows. It maintains a set $S \subset \mathbb{Z}$ that forms a complete system of representatives for the quotient group $\mathbb{Z}_m^*/\langle p \rangle$, i.e., it contains exactly one element from every residue class. Then we use a ring isomorphism

$$\begin{aligned} R_{p^r} &\to \bigoplus_{h \in S} E \\ \alpha &\mapsto \{\alpha(\zeta^h)\}_{h \in S}. \end{aligned} \tag{1}$$

Here, if α is the residue class $a(X) + (p^r, \Phi_m(X))$ for some $a(X) \in \mathbb{Z}[X]$, then $\alpha(\zeta^h) = a(\zeta^h) \in E$, which is independent of the representative $a(X)$.

This representation allows HElib to effectively pack $k \overset{\text{def}}{=} |S| = |\mathbb{Z}_m^*/\langle p \rangle|$ elements of E into different "slots" of a single plaintext. Addition and multiplication of ciphertexts act on the slots of the corresponding plaintext in parallel.

2.3 Hypercube structure and one-dimensional rotations

Beyond addition and multiplications, we can also manipulate elements in R_{p^r} using a set of automorphisms on R_{p^r}, of the form

$$\begin{aligned} \tau_j : R_{p^r} &\to R_{p^r} \\ a(x) + (p^r, \Phi_m(X)) &\mapsto a(X^j) + (p^r, \Phi_m(X)). \end{aligned} \qquad (j \in \mathbb{Z}_m^*)$$

We can homomorphically apply these automorphisms by applying them to the ciphertext elements and then preforming "key switching" (see [4, 10]). As discussed in [10], these automorphisms induce a hypercube structure on the plaintext slots, where the hypercube structure depends on the structure of the group $\mathbb{Z}_m^*/\langle p \rangle$. Specifically, HElib keeps a hypercube basis $g_1, \ldots, g_n \in \mathbb{Z}_m$ with orders $\ell_1, \ldots, \ell_n \in \mathbb{Z}_{>0}$, and then defines the set S of representatives for $\mathbb{Z}_m^*/\langle p \rangle$ (which is used for slot mapping Eqn. (1)) as

$$S \overset{\text{def}}{=} \{g_1^{e_1} \cdots g_n^{e_n} \bmod m : 0 \le e_i < \ell_i,\ i = 1, \ldots, n\}. \tag{2}$$

This basis defines an n-dimensional hypercube structure on the plaintext slots, where slots are indexed by tuples $(e_1, \ldots, e_n)$ with $0 \le e_i < \ell_i$. If we fix $e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n$, and let e_i range over $0, \ldots, \ell_i - 1$, we get a set of ℓ_i slots, indexed by $(e_1, \ldots, e_n)$, which we refer to as a *hypercolumn in dimension i* (and there are k/ℓ_i such hypercolumns). Using automorphisms, we can efficiently perform rotations in any dimension; a rotation by v in dimension i maps a slot indexed by $(e_1, \ldots, e_i, \ldots, e_n)$ to the slot indexed by $(e_1, \ldots, e_i + v \bmod \ell_i, \ldots, e_n)$. Below we denote this operation by ρ_i^v.

We can implement ρ_i^v by applying either one automorphism or two: if the order of g_i in $\mathbb{Z}_m^*$ is ℓ_i, then we get by with just a single automorphism, $\rho_i^v(\alpha) = \tau_{g_i^v}(\alpha)$. If the order of g_i in $\mathbb{Z}_m^*$ is different from ℓ_i then we need to implement this rotation using two shifts: specifically, we use a constant "0-1 mask value" mask that selects some slots and zeros-out the others, and use two automorphisms with exponents $e = g_i^v \bmod m$ and $e' = g_i^{v-\ell_i} \bmod m$, setting

$$\rho_i^v(\alpha) = \tau_e(\mathsf{mask} \cdot \alpha) + \tau_{e'}((1 - \mathsf{mask}) \cdot \alpha).$$

In the first case, we call i a "good dimension", and in the latter, we call i a "bad dimension".

2.4 Frobenius and linearized polynomials

We define $\sigma \overset{\text{def}}{=} \tau_p$, which is the Frobenius map on R_{p^r}, and acts on each slot independently as the Frobenius map σ_E on E, which sends ζ to ζ^p and leaves elements of $\mathbb{Z}/(p^r)$ fixed (this is the same as the pth power map on E when $r = 1$). If M is a $\mathbb{Z}/(p^r)$-linear transformation on E, then there exist unique constants $\theta_0, \ldots, \theta_{d-1}$ such that for all $\eta \in E$, we have $M(\eta) = \sum_{f=0}^{d-1} \theta_f \sigma_E^f(\eta)$. When $r = 1$, this follows from the general theory of linearized polynomials (see, e.g., Theorem 10.4.4 on p. 237 of [25]), and these constants are readily computable by solving a system of equations mod p; when $r > 1$, we may lift these solutions via Hensel to a solution mod p^r. In the special case where the image of M is the sub-ring $\mathbb{Z}/(p^r)$ of E, then $\theta_f = \sigma_E^f(\theta_0)$ for $f = 1, \ldots, d-1$; again, this is standard field theory if $r = 1$, and is easily established for $r > 1$ as well.

Using linearized polynomials, we may effectively apply a fixed linear map to each slot of a plaintext element $\alpha \in R_{p^r}$ (either the same or different maps in each slot) by computing $\sum_{f=0}^{d-1} \kappa_f \sigma^f(\alpha)$, where the κ_f's constants in R_{p^r} obtained by embedding appropriate constants in E in each slot. We may perform the same computation homomorphically on an encryption of α in the time of $d - 1$ automorphisms and d constant-ciphertext multiplications, and in the depth of one constant-ciphertext multiplication (since automorphisms consume almost no depth).

3 Overview of the Recryption Procedure

Recall that the recryption procedure is given a BGV ciphertext $\mathsf{ct} = (\mathfrak{c}_0, \mathfrak{c}_1)$, defined relative to secret-key $\mathsf{sk} = (1, \mathfrak{s})$, modulus q, and plaintext space p^r, namely, we have $[\langle \mathsf{sk}, \mathsf{ct} \rangle]_q \equiv \mathfrak{m} \pmod{p^r}$ with $\mathfrak{m}$ being the plaintext. Also we have the guarantee that the noise is ct is still rather small, say $\|[\langle \mathsf{sk}, \mathsf{ct} \rangle]_q\| < q/100$.

The goal of the recryption procedure is to produce another ciphertext ct^* that encrypts the same plaintext element $\mathfrak{m}$ relative to the same secret key, but relative to a much larger modulus $Q \gg q$ and with a much smaller relative noise. That is, we still want to get $[\langle \mathsf{sk}, \mathsf{ct}^* \rangle]_Q = m \pmod{p^r}$, but with $\|[\langle \mathsf{sk}, \mathsf{ct}^* \rangle]_Q\| \ll Q/100$.[3]

Our implementation uses roughly the same high-level structure for the recryption procedure as in [12, 1], below we briefly recall the structure from [12] and then describe our variant of it.

3.1 The GHS Recryption Procedure

The recryption procedure from [12] (for plaintext space $p = 2$) begins by using modulus-switching to compute another ciphertext that encrypts the same plaintext as ct, but relative to a specially chosen modulus $\tilde{q} = 2^e + 1$ (for some integer e).

Denote the resulting ciphertext by ct', the rest of the recryption procedure consists of homomorphic implementation of the decryption formula $\mathfrak{m} \leftarrow [[\langle \mathsf{sk}, \mathsf{ct}' \rangle]_{\tilde{q}}]_2$, applied to an encryption of sk that can be found in the public key. Note that in this formula we know $\mathsf{ct}' = (\mathfrak{c}_0', \mathfrak{c}_1')$ explicitly, and it is sk

[3] The relative noise after recryption is a design parameter. In our implementation we tried to get the noise below $Q/2^{250}$, to allow significant additional processing before another recryption is needed.

that we process homomorphically. It was shown in [12] that for the special modulus $\tilde{q}$, the decryption procedure can be evaluated (roughly) by computing $\mathfrak{u} \leftarrow [\langle \mathsf{sk}, \mathsf{ct}' \rangle]_{2^{e+1}}$ and then $\mathfrak{m} \leftarrow \mathfrak{u}\langle e \rangle \oplus \mathfrak{u}\langle 0 \rangle$.[4]

To enable recryption, the public key is augmented with an encryption of the secret key $\mathfrak{s}$, relative to a (much) larger modulus $Q \gg \tilde{q}$, *and also relative to a larger plaintext space* 2^{e+1}. Namely this is a ciphertext $\tilde{\mathsf{ct}}$ such that $[\langle \mathsf{sk}, \tilde{\mathsf{ct}} \rangle]_Q = \mathfrak{s} \pmod{2^{e+1}}$. Recalling that all the coefficients in $\mathsf{ct}' = (\mathfrak{c}'_0, \mathfrak{c}'_1)$ are smaller than $\tilde{q}/2 < 2^{e+1}/2$, we consider $\mathfrak{c}'_0, \mathfrak{c}'_1$ as plaintext elements modulo 2^{e+1}, and compute homomorphically the inner-product $\mathfrak{u} \leftarrow \mathfrak{c}'_1 \cdot \mathfrak{s} + \mathfrak{c}'_0 \pmod{2^{e+1}}$ by setting

$$\tilde{\mathsf{ct}}' \leftarrow \mathfrak{c}'_1 \cdot \tilde{\mathsf{ct}} + (\mathfrak{c}'_0, 0).$$

This means that $\tilde{\mathsf{ct}}'$ encrypts the desired $\mathfrak{u}$, and to complete the recryption procedure we just need to extract and XOR the top and bottom bits from all the coefficients in $\mathfrak{u}$, thus getting an encryption of (the coefficients of) the plaintext $\mathfrak{m}$. This calculation is the most expensive part of recryption, and it is done in three steps:

Linear transformation. First apply homomorphically a $Z_{2^{e+1}}$-linear transformation to $\tilde{\mathsf{ct}}'$, converting it into ciphertexts that have the coefficients of $\mathfrak{u}$ in the plaintext slots.

Bit extraction. Next apply a homomorphic (non-linear) bit-extraction procedure, computing two ciphertexts that contain the top and bottom bits (respectively) of the integers stored in the slots. A side-effect of the bit-extraction computation is that the plaintext space is reduced from mod-2^{e+1} to mod-2, so adding the two ciphertexts we get a ciphertext whose slots contain the coefficients of $\mathfrak{m}$ relative to a mod-2 plaintext space.

Inverse linear transformation. Finally apply homomorphically the inverse linear transformation (this time over Z_2), obtaining a ciphertext ct^* that encrypts the plaintext element $\mathfrak{m}$.

An optimization. The deepest part of recryption is bit-extraction, and its complexity — both time and depth — increases with the most-significant extracted bit (i.e., with e). The parameter e can be made somewhat smaller by choosing a smaller $\tilde{q} = 2^e + 1$, but for various reasons $\tilde{q}$ cannot be too small, so Gentry et al. described in [12] an optimization for reducing the top extracted bit without reducing $\tilde{q}$.

After modulus-switching to the ciphertext ct, we can add multiples of $\tilde{q}$ to the coefficients of $\mathfrak{c}'_0, \mathfrak{c}'_1$ to make them divisible by $2^{e'}$ for some moderate-size $e' < e$. Let $\mathsf{ct}'' = (\mathfrak{c}''_0, \mathfrak{c}''_1)$ be the resulting ciphertext, clearly $[\langle \mathsf{sk}, \mathsf{ct}' \rangle]_{\tilde{q}} = [\langle \mathsf{sk}, \mathsf{ct}'' \rangle]_{\tilde{q}}$ so ct'' still encrypts the same plaintext $\mathfrak{m}$. Moreover, as long as the coefficients of ct'' are sufficiently smaller than $\tilde{q}^2$, we can still use the same simplified decryption formula $\mathfrak{u}' \leftarrow [\langle \mathsf{sk}, \mathsf{ct}'' \rangle]_{2^{e+1}}$ and $\mathfrak{m} \leftarrow \mathfrak{u}'\langle e \rangle \oplus \mathfrak{u}'\langle 0 \rangle$.

However, since ct'' is divisible by $2^{e'}$ then so is $\mathfrak{u}$. For one thing this means that $\mathfrak{u}'\langle 0 \rangle = 0$ so the decryption procedure can be simplified to $\mathfrak{m} \leftarrow \mathfrak{u}'\langle e \rangle$. But more importantly, we can divide ct'' by $2^{e'}$ and compute instead $\mathfrak{u}'' \leftarrow [\langle \mathsf{sk}, \mathsf{ct}''/2^{e'} \rangle]_{2^{e-e'+1}}$ and $\mathfrak{m} \leftarrow \mathfrak{u}'\langle e - e' \rangle$. This means that the encryption of $\mathfrak{s}$ in the public key can be done relative to plaintext space $2^{e-e'}$ and we only need to extract $e - e'$ bits rather than e.

In this work we observe that we can do even slightly better by adding to ct' multiples of $\tilde{q}$ and also multiples of 2 (or more generally multiples of $\tilde{q}$ and p when recrypting a ciphertext with mod-p plaintext space). This lets us get a value of e' which is one larger than what we can get by adding only multiples of $\tilde{q}$, so we can extract one less digit. See details in Section 5.2.

[4] This is a slight simplification, the actual formula for $p = 2$ is $\mathfrak{m} \leftarrow \mathfrak{u}\langle e \rangle \oplus \mathfrak{u}\langle e - 1 \rangle \oplus \mathfrak{u}\langle 0 \rangle$, see Lemma 5.1.

3.2 Our Recryption Procedure

We optimize the GHS recryption procedure and extend it to handle plaintext spaces modulo arbitrary prime powers p^r rather than just $p = 2$, $r = 1$. The high-level structure of the procedure remains roughly the same.

To reduce the complexity as much as we can, we use a special recryption key $\tilde{\mathsf{sk}} = (1, \tilde{\mathfrak{s}})$, which is chosen as sparse as possible (subject to security requirements). As we elaborate in Appendix A, the number of nonzero coefficients in $\tilde{\mathfrak{s}}$ plays an extremely important role in the complexity of recryption.[5]

To enable recryption of mod-p^r ciphertexts, we include in the public key a ciphertext $\tilde{\mathsf{ct}}$ that encrypts the secret key $\tilde{\mathfrak{s}}$ relative to a large modulus Q and plaintext space mod-p^{e+r} for some $e > r$. Then given a mod-p^r ciphertext ct to recrypt, we perform the following steps:

Modulus-switching. Convert ct into another ct' relative to the special modulus $\tilde{q} = p^e + 1$. We prove in Lemma 5.1 that for the special modulus $\tilde{q}$, the decryption procedure can be evaluated by computing $\mathfrak{u} \leftarrow [\langle \mathsf{sk}, \mathsf{ct}' \rangle]_{p^{e+r}}$ and then $\mathfrak{m} \leftarrow \mathfrak{u}\langle r-1, \ldots, 0 \rangle_p - \mathfrak{u}\langle e+r-1, \ldots, e \rangle_p \pmod{p^r}$.

Optimization. Add multiples of $\tilde{q}$ and multiples of p^r to the coefficients of ct', making them divisible by $p^{e'}$ for some $r \le e' < e$ without increasing them too much and also without increasing the noise too much. This is described in Section 5.2. The resulting ciphertext, which is divisible by $p^{e'}$, is denoted $\mathsf{ct}'' = (\mathfrak{c}_0'', \mathfrak{c}_1'')$. It follows from the same reasoning as above that we can now compute $\mathfrak{u}' \leftarrow [\langle \mathsf{sk}, \mathsf{ct}''/p^{e'} \rangle]_{p^{e-e'+r}}$ and then $\mathfrak{m} \leftarrow -\mathfrak{u}\langle e-e'+r-1, \ldots, e-e' \rangle_p \pmod{p^r}$.

Multiply by encrypted key. Evaluate homomorphically the inner product (divided by $p^{e'}$), $\mathfrak{u}' \leftarrow (\mathfrak{c}_1' \cdot \mathfrak{s} + \mathfrak{c}_0')/p^{e'} \pmod{p^{e-e'+r}}$, by setting $\tilde{\mathsf{ct}}' \leftarrow (\mathfrak{c}_1'/p^{e'}) \cdot \tilde{\mathsf{ct}} + (\mathfrak{c}_0'/p^{e'}, 0)$. The plaintext space of the resulting $\tilde{\mathsf{ct}}'$ is modulo $p^{e-e'+r}$.

Note that since we only use plaintext space modulo $p^{e-e'+r}$, then we might as well use the same plaintext space also for $\tilde{\mathsf{ct}}$, rather than encrypting it relative to plaintext space modulo p^{e+r} as described above. This yields somewhat smaller noise, see more details in Appendix A.

Linear transformation. Apply homomorphically a $Z_{p^{e-e'+r}}$-linear transformation to $\tilde{\mathsf{ct}}'$, converting it into ciphertexts that have the coefficients of $\mathfrak{u}'$ in the plaintext slots. This linear transformation, which is the most intricate part of the implementation, is described in Section 4. It uses a tensor decomposition similar to [1] to reduce complexity, but pays much closer attention to details such as the mult-by-constant depth and data movements.

Digit extraction. Apply a homomorphic (non-linear) digit-extraction procedure, computing r ciphertexts that contain the digits $e-e'+r-1$ through $e-e'$ of the integers in the slots, respectively, relative to plaintext space mod-p^r. This requires that we generalize the bit-extraction procedure from [12] to a digit-extraction procedure for any prime power $p^r \ge 2$, this is done in Section 5.3. Once we extracted all these digits, we can combine them to get an encryption of the coefficients of m in the slots (relative to plaintext space modulo p^r).

Inverse linear transformation. Finally apply homomorphically the inverse linear transformation, this time over Z_{p^r}, converting the ciphertext into an encryption ct^* of the plaintext element $\mathfrak{m}$ itself. This too is described in Section 4.

[5] In our implementation we use a Hamming-weight-56 key, which is slightly smaller than the default Hamming-weight-64-keys that are used elsewhere in `HElib`.

4 The Linear Transformations

In this section we describe the linear transformations that we apply during the recryption procedure to map the plaintext coefficients into the slots and back. We begin with some additional background.

4.1 Algebraic Background

Throughout this section, we write $m = m_1 \cdots m_t$, where the m_i's are pairwise relatively prime positive integers. We write $\mathrm{CRT}(h_1, \ldots, h_t)$ for the unique solution $h \in \{0, \ldots, m-1\}$ to the system of congruences $h \equiv h_i \pmod{m_i}$ $(i = 1, \ldots, t)$, where $h_i \in \{0, \ldots, m_i - 1\}$ for all $i = 1, \ldots, t$.

Lemma 4.1 *Let p be a prime not dividing any of the m_i's. Let $d_1, \ldots, d_t$ be positive integers, where d_i is the order of $p^{d_1 \cdots d_{i-1}}$ modulo m_i. Then the order of p modulo m is $d \overset{\text{def}}{=} d_1 \cdots d_t$. Moreover, suppose that $S_1, \ldots, S_t$ are sets of integers such that the set $S_i \subseteq \{0, \ldots, m_i - 1\}$ forms a complete system of representatives for $\mathbb{Z}^*_{m_i}/\langle p^{d_1 \cdots d_{i-1}} \rangle$ for each $i = 1, \ldots, t$. Then the set $S \overset{\text{def}}{=} \mathrm{CRT}(S_1, \ldots, S_t)$ forms a complete system of representatives for $\mathbb{Z}^*_m/\langle p \rangle$.*

Proof. It suffices to prove the lemma for $t = 2$. The general case follows by induction on t.

The fact that the order of p modulo $m \overset{\text{def}}{=} m_1 m_2$ is $d \overset{\text{def}}{=} d_1 d_2$ is clear by definition. The cardinality of S_1 is $\phi(m_1)/d_1$ and of S_2 is $\phi(m_2)/d_2$, and so the cardinality of S is $\phi(m_1)\phi(m_2)/d_1 d_2 = \phi(m)/d$. So it suffices to show that distinct elements of S belong to distinct cosets of $\langle p \rangle$ in $\mathbb{Z}^*_m$.

To this end, let $a, b \in S$, and assume that $p^f a \equiv b \pmod{m}$ for some nonnegative integer f. We want to show that $a = b$. Now, since the congruence $p^f a \equiv b$ holds modulo m, it holds modulo m_1 as well, and by the defining property of S_1 and the construction of S, we must have $a \equiv b \pmod{m_1}$. So we may cancel a and b from both sides of the congruence $p^f a \equiv b \pmod{m_1}$, obtaining $p^f \equiv 1 \pmod{m_1}$, and from the defining property of d_1, we must have $d_1 \mid f$. Again, since the congruence $p^f a \equiv b$ holds modulo m, it holds modulo m_2 as well, and since $d_1 \mid f$, by the defining property of S_2 and the construction of S, we must have $a \equiv b \pmod{m_2}$. It follows that $a \equiv b \pmod{m}$, and hence $a = b$. □

The powerful basis. The linear transformations in our recryption procedure make use of the same tensor decomposition that was used by Alperin-Sheriff and Peikert in [1], which in turn relies on the "powerful basis" representation of the plaintext space, due to Lyubashevsky et al. [19, 18]. The "powerful basis" representation is an isomorphism

$$R_{p^r} = \mathbb{Z}[X]/(p^r, \Phi_m(X)) \longleftrightarrow R'_{p^r} \overset{\text{def}}{=} \mathbb{Z}[X_1, \ldots, X_t]/(p^r, \Phi_{m_1}(X_1), \ldots, \Phi_{m_t}(X_t)),$$

defined explicitly by the map $\mathsf{PowToPoly} : R'_{p^r} \to R_{p^r}$ that sends (the residue class of) X_i to (the residue class of) X^{m/m_i}.

Recall that we view an element in R_{p^r} as encoding a vector over a ring E, where E is an extension ring of $\mathbb{Z}/(p^r)$ that contains a principal m'th root of unity ζ, and let us define $\zeta_i \overset{\text{def}}{=} \zeta^{m/m_i}$ for $i = 1, \ldots, t$. It follows from the definitions above that for $h = \mathrm{CRT}(h_1, \ldots, h_t)$ and $\alpha = \mathsf{PowToPoly}(\alpha')$, we have $\alpha(\zeta^h) = \alpha'(\zeta_1^{h_1}, \ldots, \zeta_t^{h_t})$.

Using Lemma 4.1, we can generalize the above to multi-point evaluation. Let $S_1, \ldots, S_t$ and S be sets as defined in the lemma. Then evaluating an element $\alpha' \in R'_{p^r}$ at all points $(\zeta_1^{h_1}, \ldots, \zeta_t^{h_t})$, where $(h_1, \ldots, h_t)$ ranges over $S_1 \times \cdots \times S_t$, is equivalent to evaluating the corresponding element in $\alpha \in R_{p^r}$ at all points ζ^h, where h ranges over S.

4.2 The Evaluation Map

With the background above, we can now describe our implementation of the linear transformations. Recall that these transformations are needed to map the coefficients of the plaintext into the slots and back. Importantly, it is the powerful basis coefficients that we put in the slots during the first linear transformation, and take from the slots in the second transformation.

Note that since the two linear transformations are inverse of each other (except modulo different powers of p), then once we have an implementation of one we also get an implementation of the other. For didactic reasons we begin by describing in detail the second transformation, and later we explain how to get from it also the implementation of the first transformation.

The second transformation begins with a plaintext element β that contains in its slots the powerful-basis coefficients of some other element α, and ends with the element α itself. Important to our implementation is the view of this transformation as *multi-point evaluation* of the polynomial whose coefficients are found in the slots of β. This view depends on the set of representatives S from Eqn. (2) that determines the plaintext slots to also be a CRT combination of sets of representatives S_i for the m_i's as in Lemma 4.1. If that is the case, then the second transformation begins with an element β whose slots contain the coefficients of the powerful basis $\alpha' = \mathsf{PowToPoly}(\alpha)$, and ends with the element α that holds in the slots the values

$$\alpha(\zeta^h) = \alpha'(\zeta_1^{h_1}, \ldots, \zeta_t^{h_t})$$

where the h_i's range over the S_i's and correspondingly h range over S.

Choosing the representatives. Our first order of business is therefore to ensure that the set S of representatives defined in Eqn. (2) is the same as the set S defined in Lemma 4.1. To facilitate this (and also other aspects of our implementation), we make some additional requirements on the value of m and its factorization.[6]

I. In the terminology of Lemma 4.1, we restrict ourselves to the case where each group $\mathbb{Z}_{m_i}^*/\langle p^{d_1\cdots d_{i-1}}\rangle$ is cyclic of order k_i, and let its generator be denoted by (the residue class of) $\tilde{g}_i \in \{0, \ldots, m_i - 1\}$. Then for $i = 1, \ldots, t$, we set $S_i \overset{\text{def}}{=} \{\tilde{g}_i^e \bmod m_i : 0 \le e < k_i\}$.

   We define $g_i \overset{\text{def}}{=} \mathrm{CRT}(1, \ldots, 1, \tilde{g}_i, 1, \ldots, 1)$ (with $\tilde{g}_i$ in the ith position), and use the g_i's as our hypercube basis with the order of g_i set to k_i. In this setting, the set S from Lemma 4.1 coincides with the set S in Eqn. (2); that is, we have

$$S \;=\; \left\{\textstyle\prod_{i=1}^t g_i^{e_i} \bmod m : 0 \le e_i < k_i\right\} \;=\; \mathrm{CRT}(S_1, \ldots, S_t).$$

II. In the terminology of Lemma Lemma 4.1, we further restrict ourselves to only use factorizations $m = m_1 \cdots m_t$ for which $d_1 = d$. (That is, the order of p is the same in $\mathbb{Z}_{m_1}^*$ as in $\mathbb{Z}_m^*$.) With this assumption, we have $d_2 = \cdots = d_t = 1$, and moreover $k_1 = \phi(m_1)/d$ and $k_i = \phi(m_i)$ for $i = 2, \ldots, t$.

Note that with the above assumptions, the first dimension could be either good or bad, but the other dimensions $2, \ldots, t$ are always good. This is because $p^{d_1\cdots d_{i-1}} \equiv 1 \pmod{m}$, so also $p^{d_1\cdots d_{i-1}} \equiv 1 \pmod{m_i}$, and therefore $\mathbb{Z}_{m_i}^*/\langle p^{d_1\cdots d_{i-1}}\rangle = \mathbb{Z}_{m_i}^*$, which means that the order of g_i in $\mathbb{Z}_m^*$ (which is the same as the order of $\tilde{g}_i$ in $\mathbb{Z}_{m_i}^*$) equals k_i.

[6] As we discuss in Section 6, there are still sufficiently many settings that satisfy these requirements.

Packing the coefficients. In designing the linear transformation, we have the freedom to choose how we want the coefficients of α' to be packed in the slots of β. Let us denote these coefficients by $c_{j_1,\ldots,j_t}$ where each index j_i runs over $\{0,\ldots,\phi(m_i)-1\}$, and each $c_{j_1,\ldots,j_t}$ is in $\mathbb{Z}/(p^r)$. That is, we have

$$\alpha'(X_1,\ldots,X_t) = \sum_{j_1,j_2,\ldots,j_t} c_{j_1,\ldots,j_t} X_1^{j_1} X_2^{j_2} \cdots X_t^{j_t} = \sum_{j_2,\ldots,j_t} \Big(\sum_{j_1} c_{j_1,\ldots,j_t} X_1^{j_1} \Big) X_2^{j_2} \cdots X_t^{j_t}.$$

Recall that we can pack d coefficients into a slot, so for fixed $j_2,\ldots,j_t$, we can pack the $\phi(m_1)$ coefficients of the polynomial $\sum_{j_1} c_{j_1,\ldots,j_t} X_1^{j_1}$ into $k_1 = \phi(m_1)/d$ slots. In our implementation we pack these coefficients into the slots indexed by $(e_1, j_2, \ldots, j_t)$, for $e_1 = 0,\ldots,k_1-1$. That is, we pack them into a single hypercolumn in dimension 1.

4.2.1 The Eval Transformation

The second (inverse) linear transformation of the recryption procedure beings with the element β whose slots pack the coefficients $c_{j_1,\ldots,j_t}$ as above. The desired output from this transformation is the element whose slots contain $\alpha(\zeta^h)$ for all $h \in S$ (namely the element α itself). Specifically, we need each slot of α with hypercube index $(e_1,\ldots,e_t)$ to hold the value

$$\alpha'\big(\zeta_1^{g_1^{e_1}}, \ldots, \zeta_t^{g_t^{e_t}}\big) \;=\; \alpha\big(\zeta^{g_1^{e_1}\cdots g_t^{e_t}}\big).$$

Below we denote $\zeta_{i,e_i} \overset{\text{def}}{=} \zeta_i^{g_i^{e_i}}$. We transform β into α in t stages, each of which can be viewed as multi-point evaluation of polynomials along one dimension of the hypercube.

Stage 1. This stage beings with the element β, in which each dimension-1 hypercolumn with index $(\star, j_2, \ldots, j_t)$ contains the coefficients of the univariate polynomial $P_{j_2,\ldots,j_t}(X_1) \overset{\text{def}}{=} \sum_{j_1} c_{j_1,\ldots,j_t} X_1^{j_1}$. We transform β into β_1 where that hypercolumn contains the evaluation of the same polynomial in many points. Specifically, the slot of β_1 indexed by $(e_1, j_2, \ldots, j_t)$ contains the value $P_{j_2,\ldots,j_t}(\zeta_{1,e_1})$.

By definition, this stage consists of parallel application of a particular $\mathbb{Z}/(p^r)$-linear transformation M_1 (namely a multi-point polynomial evaluation map) to each of the k/k_1 hypercolumns in dimension 1. In other words, M_1 maps $(k_1 \cdot d)$-dimensional vectors over $\mathbb{Z}/(p^r)$ (each packed into k_1 slots) to k_1-dimensional vectors over E. We elaborate on the efficient implementation of this stage later in this section.

Stages $2,\ldots,t$**.** The element β_1 from the previous stage holds in its slots the coefficients of the k_1 multivariate polynomials

$$\begin{aligned} A_{e_1}(X_2,\ldots,X_t) &\overset{\text{def}}{=} \alpha'(\zeta_{1,e_1}, X_2, \ldots, X_t) \\ &= \sum_{j_2,\ldots,j_t} \underbrace{\big(\sum_{j_1} c_{j_1,\ldots,j_t} \zeta_{1,e_1}^{j_1}\big)}_{\text{slot } (e_1,j_2,\ldots,j_t) = P_{j_2,\ldots,j_t}(\zeta_{1,e_1})} \cdot X_2^{j_2} \cdots X_t^{j_t} \qquad (e_1 = 0,\ldots,k_1-1). \end{aligned}$$

The goal in the remaining stages is to implement multi-point evaluation of these polynomials at all the points $X_i = \zeta_{i,e_i}$ for $0 \le e_i < k_i$. Note that differently from the polynomial α' that we started with, the polynomials A_{e_1} have coefficients from E (rather than from $\mathbb{Z}_{p^r}$), and these coefficients are encoded one per slot (rather than d per slot). As we explain later, this makes it easier to implement the desired multi-point evaluation. Separating out the second dimension we can write

$$A_{e_1}(X_2,\ldots,X_t) = \sum_{j_3,\ldots,j_t} \Big(\sum_{j_2} P_{j_2,\ldots,j_t}(\zeta_{1,e_1}) X_2^{j_2} \Big) X_3^{j_3} \cdots X_t^{j_t}.$$

We note that each dimension-2 hypercolumn in β_1 with index $(e_1, \star, j_3, \dots, j_t)$ contains the E-coefficients of the univariate polynomial $Q_{e_1,j_3,\dots,j_t}(X_2) \overset{\text{def}}{=} \sum_{j_2} P_{j_2,\dots,j_t}(\zeta_{1,e_1}) X_2^{j_2}$. In Stage 2, we transform β_1 into β_2 where that hypercolumn contains the evaluation of the same polynomial in many points. Specifically, the slot of β_2 indexed by $(e_1, e_2, j_3 \dots, j_t)$ contains the value

$$Q_{e_1,j_3,\dots,j_t}(\zeta_{2,e_2}) \;=\; \sum_{j_2} P_{j_2,\dots,j_t}(\zeta_{1,e_1}) \cdot \zeta_{2,e_2}^{j_2} \;=\; \sum_{j_1,j_2} c_{j_1,\dots,j_t} \zeta_{1,e_1}^{j_1} \zeta_{2,e_2}^{j_2},$$

and the following stages implement the multi-point evaluation of these polynomials at all the points $X_i = \zeta_{i,e_i}$ for $0 \le e_i < k_i$.

Stages $s = 3, \dots, t$ proceed analogously to Stage 2, each time eliminating a single variable X_s via the parallel application of an E-linear map M_s to each of the k/k_s hypercolumns in dimension s. When all of these stages are completed, we have in every slot with index $(e_1, \dots, e_t)$ the value $\alpha'(\zeta_{1,e_1}, \dots, \zeta_{t,e_t})$, as needed.

Implementing stages $2, \dots, t$. For $s = 2, \dots, t$, we obtain β_s from β_{s-1} by applying the linear transformation M_s in parallel to each hypercolumn in dimension s. We adapt for that purpose the HElib matrix-multiplication procedure [14], using only rotations along dimension s. The procedure from [14] multiplies an $n \times n$ matrix M by a $n \times 1$ column vector v by computing

$$Mv = D_0 v_0 + \cdots + D_{n-1} v_{n-1}, \tag{3}$$

where each v_i is the vector obtained by rotating the entries of v by i positions, and each D_i is a diagonal matrix whose entries are taken from M. In our case, we perform k/k_s such computations in parallel, one on every hypercolumn along the s dimension, by setting

$$\beta_s = \sum_{e=0}^{k_s-1} \kappa_{s,e} \cdot \rho_s^e(\beta_{s-1}), \tag{4}$$

where the $\kappa_{s,e}$'s are constants in R_{p^r} obtained by embedding appropriate constants in E in each slot. Eqn. (4) translates directly into a simple homomorphic evaluation algorithm, just by applying the same operations to the ciphertexts. The cost in time for stage s is $k_s - 1$ automorphisms and k_s constant-ciphertext multiplications; the cost in depth is a single constant-ciphertext multiplication.

Implementing stage 1. Stage 1 is more challenging, because the map M_1 is a $\mathbb{Z}/(p^r)$-linear map, rather than an E-linear map. Nevertheless, we can still use the same diagonal decomposition as in Eqn. (3), except that the entries in the diagonal matrices are no longer elements of E, but rather, $\mathbb{Z}/(p^r)$-linear maps on E. These maps may be encoded using linearized polynomials, as in Section 2.4, allowing us to write

$$\beta_1 = \sum_{e=0}^{k_1-1} \sum_{f=0}^{d-1} \lambda_{e,f} \cdot \sigma^f\big(\rho_1^e(\beta)\big), \tag{5}$$

where the $\lambda_{e,f}$'s are constants in R_{p^r}.

A naive homomorphic implementation of the formula from Eqn. (5) takes $O(dk_1)$ automorphisms, but we can reduce this to $O(d + k_1)$ as follows. Since σ^f is a ring automorphism, it commutes with addition and multiplication, so we can rewrite Eqn. (5) as follows:

$$\beta_1 \;=\; \sum_{f=0}^{d-1} \sum_{e=0}^{k_1-1} \sigma^f\Big(\; \sigma^{-f}(\lambda_{e,f}) \cdot \rho_1^e(\beta) \;\Big) \;=\; \sum_{f=0}^{d-1} \sigma^f\left(\sum_{e=0}^{k_1-1} \; \sigma^{-f}(\lambda_{e,f}) \cdot \rho_1^e(\beta) \;\right). \tag{6}$$

To evaluate Eqn. (6) homomorphically, we compute encryptions of $\rho_1^e(\beta)$ for $e = 0, \ldots, k_1 - 1$, then take d different linear combinations of these values, homomorphically computing

$$\gamma_f = \sum_{e=0}^{k_1-1} \sigma^{-f}(\lambda_{e,f}) \cdot \rho_1^e(\beta) \qquad (f = 0, \ldots, d-1).$$

Finally, we can compute an encryption of $\beta_1 = \sum_{f=0}^{d-1} \sigma^f(\gamma_f)$ by applying Frobenius maps to the ciphertexts encrypting the γ_f's, and summing.

If dimension 1 is good, the homomorphic computation of the γ_f's takes the time of $k_1 - 1$ automorphisms and $k_1 d$ constant-ciphertext multiplications, and the depth of one constant-ciphertext multiplication. If dimension 1 is bad, we can maintain the same depth by folding the multiplication by the constants $\sigma^{-f}(\lambda_{e,f})$ into the masks used for rotation (see Section 2.3); the time increases to $2(k_1 - 1)$ automorphisms and $(2k_1 - 1)d$ constant-ciphertext multiplications.

The entire procedure to compute an encryption of β_1 has depth of one constant-ciphertext multiplication, and it takes time $k_1 + d - 2 + B(k_1 - 1)$ automorphisms and $k_1 d + B(k_1 - 1)d$ constant-ciphertext multiplications, where B is a flag which is 1 if dimension 1 is bad and 0 if it is good.

Complexity of Eval. From the above, we get the following cost estimates for computing the Eval map homomorphically. The depth is t constant-ciphertext multiplications, and the time is at most

- $(B+1)\phi(m_1)/d + d + \phi(m_1) + \cdots \phi(m_t)$ automorphisms, and
- $(B+1)\phi(m_1) + \phi(m_2) + \cdots + \phi(m_t)$ constant-ciphertext multiplications.

4.2.2 The Transformation Eval^{-1}

The first linear transformation in the recryption procedure is the inverse of Eval. This transformation can be implemented by simply running the above stages in reverse order and using the inverse linear maps M_s^{-1} in place of M_s. The complexity estimates are identical.

4.3 Unpacking and Repacking the Slots

In our recryption procedure we have the non-linear digit extraction routine "sandwiched" between the linear evaluation map and its inverse. However the evaluation map transformations from above maintain fully-packed ciphertexts, where each slot contains an element of the extension ring E (of degree d), while our digit extraction routine needs "sparsely packed" slots containing only integers from $\mathbb{Z}/(p^r)$.

Therefore, before we can use the digit extraction procedure we need to "unpack" the slots, so as to get d ciphertexts in which each slot contains a single coefficient in the constant term. Similarly, after digit extraction we have to "repack" the slots, before running the second transformation.

Unpacking. Let us consider the unpacking procedure in terms of the element $\beta \in R_{p^r}$. Each slot of β contains an element of E which we can write as $\sum_{i=0}^{d-1} a_i \zeta^i$, where the a_i's are in $\mathbb{Z}/(p^r)$. We want to compute $\beta^{(0)}, \ldots, \beta^{(d-1)}$, so that the corresponding slot of each $\beta^{(i)}$ contains a_i. To obtain $\beta^{(i)}$, we need to apply to each slot of β the $\mathbb{Z}/(p^r)$-linear map $L_i : E \to \mathbb{Z}/(p^r)$ that maps $\sum_{i=0}^{d-1} a_i \zeta^i$ to a_i.

Using linearized polynomials, as discussed in Section 2.4, we may write $\beta^{(i)} = \sum_{f=0}^{d-1} \kappa_{i,f} \sigma^f(\beta)$, for constants $\kappa_{i,f} \in R/(p^r)$. Given an encryption of β, we can compute encryptions of all of the $\sigma^f(\beta)$'s and then take linear combinations of these to get encryptions of all of the $\beta^{(i)}$'s. This takes the time of $d-1$ automorphisms and d^2 constant-ciphertext multiplications, and a depth of one constant-ciphertext multiplication.

While the cost in time of constant-ciphertext multiplications is relatively cheap, it cannot be ignored, especially as we have to compute d^2 of them. In our implementation, the cost is dominated the time it takes to convert an element in R_{p^r} to its corresponding DoubleCRT representation [11]. It is possible, of course, to precompute and store all d^2 of these constants in DoubleCRT format, but the space requirement is significant: for typical parameters, our implementation takes about 4MB to store a single constant in DoubleCRT format, so for example with $d = 24$, these constants take up almost 2.5GB of space.

This unappealing space/time trade-off can be improved considerably using somewhat more sophisticated implementations. Suppose that in the first linear transformation Eval^{-1}, instead of packing the coefficients $a_0, \ldots, a_{d-1}$ into a slot as $\sum_i a_i \zeta^i$, we pack them as $\sum_i a_i \sigma_E^i(\theta)$, where $\theta \in E$ is a normal element. Further, let $L_0' : E \to \mathbb{Z}/(p^r)$ be the $\mathbb{Z}/(p^r)$-linear map that sends $\eta = \sum_i a_i \sigma_E^i(\theta)$ to a_0. Then we have $L_0'(\sigma^{-j}(\eta)) = a_j$ for $j = 0, \ldots, d-1$. If we realize the map L_0' with linearized polynomials, and if the plaintext γ has the coefficients packed into slots via a normal element as above, then we have

$$\beta^{(i)} = \sum_{f=0}^{d-1} \kappa_f \cdot \sigma^{f-i}(\gamma),$$

where the κ_f's are constants in R_{p^r}. So we have only d constants rather than d^2.

To use this strategy, however, we must address the issue of how to modify the Eval transformation so that Eval^{-1} will give us the plaintext element γ that packs coefficients as $\sum_i a_i \sigma_E^i(\theta)$. As it turns out, in our implementation this modification is for free: recall that the unpacking transformation immediately follows the last stage of the inverse evaluation map Eval^{-1}, and that last stage applies $\mathbb{Z}/(p^r)$-linear maps to the slots; therefore, we simply fold into these maps the $\mathbb{Z}/(p^r)$-linear map that takes $\sum_i a_i \zeta^i$ to $\sum_i a_i \sigma_E^i(\theta)$ in each slot.

We note that we can reduce the number of stored constants even further: since L_0' is a map from E to the base ring $\mathbb{Z}/(p^r)$, then the κ_f's are related via $\kappa_f = \sigma^f(\kappa_0)$. Therefore, we can obtain all of the DoubleCRTs for the κ_f's by computing just one for κ_0 and then applying the Frobenius automorphisms directly to the DoubleCRT for κ_0. We note, however, that applying these automorphisms directly to DoubleCRTs leads to a slight increase in the noise of the homomorphic computation. We did not use this last optimization in our implementation.

Repacking. Finally, we discuss the reverse transformation, which repacks the slots, taking $\beta^{(0)}, \ldots, \beta^{(d-1)}$ to β. This is quite straightforward: if $\bar{\zeta}$ is the plaintext element with ζ in each slot, then $\beta = \sum_{i=0}^{d-1} \bar{\zeta}^i \beta^{(i)}$. This formula can be evaluated homomorphically with a cost in time of d constant-ciphertext multiplications, and a cost in depth one constant-ciphertext multiplication.

5 Recryption with Plaintext Space Modulo $p > 2$

Below we extend the treatment from [12, 1] to handle plaintext spaces modulo $p > 2$. In Sections 5.1 through 5.3 we generalize the various lemmas to $p > 2$, in Appendix A we discuss the choice of parameters, and then in Section 5.4 we explain how these lemmas are put together in the recryption procedure.

5.1 Simpler Decryption Formula

We begin by extending the simplified decryption formula [12, Lemma 1] from plaintext space mod-2 to any prime-power p^r. Recall that we denote by $[z]_q$ the mod-q reduction into $[-q/2, q/2)$ (except when $q = 2$ we reduce to $(-1, 1]$). Also $z\langle j, \ldots, i\rangle_p$ denotes the integer whose mod-p expansion consists of digits i through j in the mod-p expansion of z (and we omit the p subscript if it is clear from the context).

Lemma 5.1 *Let $p > 1$ be an integer, and let $r \geq 1$, $e \geq r + 2$ and $q = p^e + 1$. Finally, let z be an integer such that $|z| \leq \frac{q^2}{4} - q$ and $|[z]_q| \leq \frac{q}{4}$.*

- *If p is odd then $[z]_q = z\langle r-1, \ldots, 0\rangle - z\langle e+r-1, \ldots, e\rangle \pmod{p^r}$.*
- *If $p = 2$ then $[z]_q = z\langle r-1, \ldots, 0\rangle - z\langle e+r-1, \ldots, e\rangle - z\langle e-1\rangle \pmod{2^r}$.*

Proof. We begin with the odd-p case. Denote $z_0 = [z]_q$, then $z = z_0 + kq$ for some $|k| \leq \frac{q}{4} - 1$, and hence $|z_0 + k| \leq \frac{q}{4} + \frac{q}{4} - 1 = (q-2)/2 = (p^e - 1)/2$. We can write

$$z \;=\; z_0 + kq \;=\; z_0 + k(p^e + 1) \;=\; z_0 + k + p^e k. \tag{7}$$

This means in particular that $z = z_0 + k \pmod{p^r}$, and also since the mod-p representation of the sum $w = z_0 + k$ has only 0's in positions e and up then $k\langle r-1, \ldots, 0\rangle = z\langle e+r-1, \ldots, e\rangle$. It follows that

$$z_0\langle r-1, \ldots, 0\rangle = z\langle r-1, \ldots, 0\rangle - k\langle r-1, \ldots, 0\rangle = z\langle r-1, \ldots, 0\rangle - z\langle e+r-1, \ldots, e\rangle \pmod{p^r}.$$

The proof for the $p = 2$ case is similar, but we no longer have the guarantee that the high-order bits of the sum $w = z_0 + k$ are all zero. Hence from Eqn. (7) we can only deduce that

$$z\langle e+r-1, \ldots, e\rangle = w\langle e+r-1, \ldots, e\rangle + k\langle r-1, \ldots, 0\rangle \pmod{2^r},$$

and also that $z\langle e-1\rangle = w\langle e-1\rangle$.

Since $|w| \leq |z_0| + |k| < \lfloor q/2 \rfloor = 2^{e-1}$, then the bits in positions $e-1$ and up in the representation of w are either all zero if $w \geq 0$, or all one if $w < 0$. In particular, this means that

$$w\langle e+r-1, \ldots, e\rangle \;=\; \left\{ \begin{array}{rl} 0 & \text{if } w \geq 0 \\ -1 & \text{if } w < 0 \end{array} \right\} \;=\; -\,w\langle e-1\rangle \;=\; -\,z\langle e-1\rangle \pmod{2^r}.$$

Concluding, we therefore have

$$\begin{aligned} z_0\langle r-1, \ldots, 0\rangle &= z\langle r-1, \ldots, 0\rangle - k\langle r-1, \ldots, 0\rangle \\ &= z\langle r-1, \ldots, 0\rangle - \big(z\langle e+r-1, \ldots, e\rangle - w\langle e+r-1, \ldots, e\rangle\big) \\ &= z\langle r-1, \ldots, 0\rangle - z\langle e+r-1, \ldots, e\rangle - z\langle e-1\rangle \pmod{2^r}. \quad \square \end{aligned}$$

5.2 Making an Integer Divisible By $p^{e'}$

As sketched in Section 3, we use the following lemma to reduce to number of digits that needs to be extracted, hence reducing the time and depth of the digit-extraction step.

Lemma 5.2 *Let z be an integer, and let p, q, r, e' be positive integers s.t. $e' \geq r$ and $q = 1 \pmod{p^{e'}}$. Also let α be an arbitrary real number in $[0, 1]$. Then there are integer coefficients u, v such that*

$$z + u \cdot p^r + v \cdot q = 0 \pmod{p^{e'}}$$

and moreover u, v are small. Specifically $|u| \leq \lceil \alpha p^{e'-1}/2 \rceil$ and $|v| \leq p^r(\frac{1}{2} + \lfloor (1-\alpha)p^{e'-1}/2 \rfloor)$.

Proof. Since q, p are co-prime then there exists $v' \in (-p^r/2, p^r/2]$ s.t. $z' = z + v'q = 0 \pmod{p^r}$. Let $\delta = -z' \cdot p^{-r} \bmod p^{e'-1}$, reduced into the interval $[\frac{-p^{e'-1}}{2}, \frac{p^{e'-1}}{2}]$, so we have $|\delta| \leq p^{e'-1}/2$ and $z' + p^r\delta = 0 \pmod{p^{e'}}$. Denote $\beta = 1 - \alpha$ and consider the integer

$$z'' \stackrel{\text{def}}{=} z' + \lceil \alpha\delta \rceil \cdot p^r + \lfloor \beta\delta \rfloor p^r \cdot q.$$

On one hand, we have that $z'' = z + u \cdot p^r + v \cdot q$ with $|u| = |\lceil \alpha\delta \rceil| \leq |\lceil \alpha p^{e'-1}/2 \rceil$ and $|v| = |v' + p^r \lfloor \beta\delta \rfloor| \leq p^r(\frac{1}{2} + \lfloor \beta p^{e'-1}/2 \rfloor)$. On the other hand since $q = 1 \pmod{p^{e'}}$ then we also have

$$z'' \;=\; z' + p^r(\lceil \alpha\delta \rceil + \lfloor \beta\delta \rfloor) \;=\; z' + p^r\delta \;=\; 0 \pmod{p^{e'}}. \quad \square$$

Discussion. Recall that in our recryption procedure we have a ciphertext ct that encrypts some $\mathfrak{m}$ with respect to modulus q and plaintext space mod-p^r, and use the lemma above to convert it into another ciphertext ct' that encrypts the same thing but is divisible by $p^{e'}$, and by doing so we need to extract e' fewer digits in the digit-extraction step.

Considering the elements $\mathfrak{u} \leftarrow \langle \mathsf{sk}, \mathsf{ct} \rangle$ and $\mathfrak{u}' \leftarrow \langle \mathsf{sk}, \mathsf{ct}' \rangle$ (without any modular reduction), since sk is integral then adding multiples of q to the coefficients of ct does not change $[\mathfrak{u}]_q$, and also as long as we do not wrap around q then adding multiples of p^r does not change $[[\mathfrak{u}]_q]_{p^r}$. Hence as long as we only add small multiples of p^r then we have $[[\mathfrak{u}]_q]_{p^r} = [[\mathfrak{u}']_q]_{p^r}$, so ct and ct' still encrypt the same plaintext. However in our recryption procedure we need more: to use our simpler decryption formula from Lemma 5.1 we not only need the noise magnitude $\|[\mathfrak{u}']_q\|$ to be smaller than $\frac{q}{4}$, but the magnitude of $\mathfrak{u}'$ itself (before mod-q reduction) must be smaller than $\frac{q^2}{4} - q$.

In essence, the two types of additive terms consume two types of "resources:" adding multiples of q increases the magnitude of $\mathfrak{u}'$, and adding multiple of p^r increases the magnitude of $[\mathfrak{u}']_q$. The parameter α from Lemma 5.2 above lets us trade-off these two resources: smaller α means slower increase in $\|[\mathfrak{u}']_q\|$ but faster increase in $\|\mathfrak{u}'\|$, and vice versa for larger α. As we discuss in Appendix A, the best trade-off is often obtained when α is just under $\frac{1}{2}$; our implementation tries to optimize this parameter, and for many settings it uses $\alpha \approx 0.45$.

5.3 Digit-Extraction for Plaintext Space Modulo p^r

The bit-extraction procedure that was described by Gentry et al. in [12] and further optimized by Alperin-Sheriff and Peikert in [1] is specific for the case $p = 2^e$. Namely, for an input ciphertext relative to mod-2^e plaintext space that contains an integer z in some slot, then the procedure can be used to compute in the same slot the i'th top bit of z, relative to a plaintext space mod-2^{e-i+1}. Below we show how to extend this bit-extraction procedure to a digit-extraction also when p is an odd prime.

The main observation underlying the original bit-extraction procedure, is that squaring an integer keeps the least-significant bit unchanged but inserts zeros in the higher-order bits. Namely, if b is the least significant bit of the integer z and moreover $z = b \pmod{2^e}$, $e \geq 1$, then squaring z we get $z^2 = b \pmod{2^{e+1}}$. Therefore, $z - z^2$ is divisible by 2^e, and the LSB of $(z - z^2)/2^e$ is the e'th bit of z.

Unfortunately the same does not hold when using a base $p > 2$. Instead, we show below that for any exponent e *there exists some degree-p polynomial* $F_e(\cdot)$ (but not necessarily $F_e(X) = X^p$) such that when $z = z_0 \pmod{p^e}$ then $F_e(z) = z_0 \pmod{p^{e+1}}$. Hence $z - F_e(z)$ is divisible by p^e, and the least-significant digit of $(z - F_e(z))/p^e$ is the e'th digit of z. The existence of such polynomial $F_e(X)$ follows from the simple derivation below.

Lemma 5.3 *For every prime p and exponent $e \geq 1$, and every integer z of the form $z = z_0 + p^e z_1$ (with z_0, z_1 integers, $z_0 \in [p]$), it holds that $z^p = z_0 \pmod{p}$, and $z^p = z_0^p \pmod{p^{e+1}}$.*

```
Digit-Extraction$_p(z, e)$:      // Extract $e$'th digit in base-$p$ representation of $z$
 1. $w_{0,0} \leftarrow z$
 2. For $k = 0$ to $e - 1$
 3.     $y \leftarrow z$
 4.     For $j = 0$ to $k$
 5.         $w_{j,k+1} \leftarrow F_e(w_{j,k})$   // $F_e$ from Corollary 5.5, for $p = 2$ we have $F_e(X) = X^2$
 6.         $y \leftarrow (y - w_{j,k+1})/p$
 7.     $w_{k+1,k+1} \leftarrow y$
 8. Return $w_{e,e}$
```

Figure 1: The digit extraction procedure

Proof. The first equality is obvious, and the proof of the second equality is just by the binomial expansion of $(z_0 + p^e z_1)^p$. □

Corollary 5.4 *For every prime p there exist a sequence of integer polynomials $f_1, f_2, \ldots,$ all of degree $\leq p - 1$, such that for every exponent $e \geq 1$ and every integer $z = z_0 + p^e z_1$ (with z_0, z_1 integers, $z_0 \in [p]$), we have*

$$z^p = z_0 + \sum_{i=1}^{e} f_i(z_0) p^i \pmod{p^{e+1}}.$$

Proof. From Lemma 5.3 we know that the mod-p digits of z^p modulo-p^{e+1} depend only on z_0, so there exist some polynomials in z_0 that describe them, $f_i(z_0) = z^p\langle i\rangle_p$. Since these f_i's are polynomials from Z_p to itself, then they have degree at most $p - 1$. Moreover, by the 1st equality in Lemma 5.3 we have that the first digit is exactly z_0. □

Corollary 5.5 *For every prime p and every $e \geq 1$ there exist a degree-p polynomial F_e, such that for every integers z_0, z_1 with $z_0 \in [p]$ and every $1 \leq e' \leq e$ we have $F_e(z_0 + p^{e'} z_1) = z_0 \pmod{p^{e'+1}}$.*

Proof. Denote $z = z_0 + p^{e'} z_1$. Since $z = z_0 \pmod{p^{e'}}$ then $f_i(z_0) = f_i(z) \pmod{p^{e'}}$. This implies that for all $i \geq 1$ we have $f_i(z_0)p^i = f_i(z)p^i \pmod{p^{e'+1}}$, and of course also for $i \geq e' + 1$ we have $f_i(z)p^i = 0 \pmod{p^{e'+1}}$. Therefore, setting $F_e(X) = X^p - \sum_{i=1}^{e} f_i(X)p^i$ we get

$$F_e(z) = z^p - \sum_{i=1}^{e} f_i(z)p^i = z^p - \sum_{i=1}^{e'} f_i(z_0)p^i = z_0 \pmod{p^{e'+1}}. \quad \square$$

We know that for $p = 2$ we have $F_e(X) = X^2$ for all e, and one can verify that also for $p = 3$ we have $F_e(X) = X^3$ for all e (when considering the balanced mod-3 representation), but for larger primes we no longer have $F_e(X) = X^p$.

The digit-extraction procedure. Just like in the base-2 case, in the procedure for extracting the e'th base-p digit from the integer $z = \sum_i z_i p^i$ proceeds by computing integers $w_{j,k}$ ($k \geq j$) such that the lowest digit in $w_{j,k}$ is z_j, and the next $k - j$ digits are zeros. The code in Figure 1 is purposely written to be similar to the code from [1, Appendix B], with the only difference being in Line 5 where we use $F_e(X)$ rather than X^2.

In our implementation we compute the coefficients of the polynomial F_e during the first call to the digit-extraction procedure for plaintext space mod-p^e, and then store it for future use. In the procedure itself, we apply the polynomial-evaluation procedure from Appendix B to compute $F_e(w_{j,k})$ in Line 5. We note that just as in [12, 1], the homomorphic division-by-p operation is done by multiplying the ciphertext by the constant $p^{-1} \bmod q$, where q is the current modulus. Since the encrypted values are guaranteed to be divisible by p, then this has the desired effect and also it reduces the noise magnitude by a factor of p. Correctness of the procedure from Figure 1 is proved exactly the same way as in [12, 1], the proof is omitted here.

5.3.1 An optimization for $p = 2$, $r \geq 2$.

As it turns out, for $p = 2$ we can sometimes extract several consecutive bits a little cheaper than what the procedure above implies. Specifically, it turns out that for $p = 2, e \geq 0$ and $r \geq 2$ we can compute the integer $z\langle e + r, \ldots, e\rangle$ by extracting only $e + r - 1$ bits (rather than $e + r$ of them). Specifically, when applying the procedure from Figure 1 (which for $p = 2$ is identical to the one from [1, Appendix B]), it turns out that we get

$$z\langle e+r, \ldots, e\rangle = \sum_{j=r}^{e+r-1} 2^{j-r} w_{j,e+r-1} \pmod{2^{e+r+1}}.$$

Note: the above would have been an immediate corollary from the correctness of the bit-extraction procedure if we added the terms $2^{j-r} w_{j,e+r}$ and let the index j go up to $e + r$, but in this case we can stop one step earlier and the result still holds.

To see why this works, observe that (by correctness), when we assign $w_{k+1,k+1} \leftarrow y$ in line 7 then it must be the case that $LSB(y) = z\langle k + 1\rangle$, and in subsequent iterations we just square w_{k+1} so as to get more zeros in higher-order bits, without changing the LSB. Recall also that squaring indeed has the desired effect since for any $i \geq 1$ and any bit b and integer n we have $(b + 2^i n)^2 = b \pmod{2^{i+1}}$. To prove the optimization, we need two additional observations:

Observation 1. *For any bit b and integer n we have $(b + 2n)^4 = b \pmod{16}$.*

Note that this is *not a corollary* of the squaring property above — that property only gives $b \pmod 8$, but in fact for this particular case we get one extra zero. (This holds only in this particular step, for later steps we only get one additional zero per squaring.)

Observation 2. *After line 7 in Figure 1, we always have $z = \sum_{j=0}^{k+1} 2^j w_{j,k+1}$.*

This can be verified by inspection: we start in line 3 from $y = z$, and at every step we subtract one w_j and divide by two, so adding them back with their respective powers of two gives back z.

Correctness now follows: Let us denote $w_j \overset{\text{def}}{=} w_{j,e+r-1}$ so we will not have to carry this extra index everywhere. Because of the first observation, the w_j's for $j = 0, 1, \ldots, e + r - 3$ have an extra zero bit, so for these w_j's we have $w_j = z\langle j\rangle \pmod{2^{e+r-j+1}}$, not just $\pmod{2^{e+r-j}}$. Denoting $v_j = 2^j w_j$, this means that the only v_j's that potentially have a nonzero bit in position $e + r$ are v_{e+r-2} and v_{e+r-1}. Also by correctness, for lower bit positions $j < e + r$, only v_j potentially has nonzero bit in position j, and all the other v_j's have zero in that position. Namely, we have

bit position:	$\star$	$e+r$	$e+r-1$	$e+r-2$	$e+r-3$	$\ldots$	1	0
$v_0 = w_0 =$	$\star$	0	0	0	0		0	$z\langle 0\rangle$
$v_1 = 2w_1 =$	$\star$	0	0	0	0		$z\langle 1\rangle$	0
	$\vdots$					$\vdots$		
$v_{e+r-3} = 2^{e+r-3} w_{e+r-3} =$	$\star$	0	0	0	$z\langle e+r-3\rangle$		0	0
$v_{e+r-2} = 2^{e+r-2} w_{e+r-2} =$	$\star$	σ	0	$z\langle e+r-2\rangle$	0		0	0
$v_{e+r-1} = 2^{e+r-1} w_{e+r-1} =$	$\star$	τ	$z\langle e+r-1\rangle$	0	0		0	0

for some two bits σ, τ (where the $\star$'s are bits above position $e + r$, which we do not care about).

This means that when adding $\sum_{j=0}^{e+r-1} v_j$, we have no carry bits upto position $e+r$. But by the second observation the sum of all these v_j's is z, so the two top bits σ, τ must satisfy $\sigma \oplus \tau = z\langle e+r \rangle$. We conclude that when adding $\sum_{j=e}^{e+r-1} v_j$, we get all the bits $z\langle e+r, \ldots, e\rangle$ which is what we needed to prove.

5.4 Putting Everything Together

Having described all separate parts of our recryption procedure, we now explain how they are combined in our implementation.

Initialization and parameters. Given the ring parameter m (that specifies the mth cyclotomic ring of integers $R = \mathbb{Z}[X]/\Phi_m(X)$) and the plaintext space p^r, we compute the recryption parameters as explained in Appendix A. That is, we use compute the Hamming weight of recryption secret key $t \geq 56$, some value of α (which is often $\alpha \approx 0.45$), and some values for e, e' where $e - e' - r \in \{\lceil \log_p(t+2) \rceil - 1, \lceil \log_p(t+2) \rceil\}$.

We also pre-compute some key-independent tables for use in the linear transformations, with the first transformation using plaintext space $p^{e-e'+r}$ and the second transformation using plaintext space p^r.

Key generation. During key generation we choose in addition to the "standard" secret key sk also a separate secret recryption key $\tilde{\mathsf{sk}} = (1, \tilde{\mathfrak{s}})$, with $\tilde{\mathfrak{s}}$ having Hamming weight t. We include in the secret key both a key-switching matrix from sk to $\tilde{\mathsf{sk}}$, and a ciphertext $\tilde{\mathsf{ct}}$ that encrypts $\tilde{\mathfrak{s}}$ under key sk, relative to plaintext space $p^{e-e'+r}$.

The recryption procedure itself. When we want to recrypt a mod-p^r ciphertext ct relative to the "standard" key sk, we first key-switch it to $\tilde{\mathsf{sk}}$ and modulus-switch it to $\tilde{q} = p^e + 1$, then make its coefficients divisible by $p^{e'}$ using the procedure from Lemma 5.2, thus getting a new ciphertext $\mathsf{ct}' = (\mathfrak{c}'_0, \mathfrak{c}'_1)$. We then compute the homomorphic inner-product divided by $p^{e'}$, by setting $\mathsf{ct}'' = (\mathfrak{c}'_1/p^{e'}) \cdot \tilde{\mathsf{ct}} + (0, \mathfrak{c}'_0/p^{e'})$.

Next we apply the first linear transformation (the map Eval^{-1} from Section 4.2), moving to the slots the coefficients of the plaintext $\mathfrak{u}'$ that is encrypted in ct''. The result is a single ciphertext with *fully packed slots*, where each slot holds d of the coefficients from $\mathfrak{u}'$. Before we can apply the digit-extraction procedure from Section 5.3, we therefore need to *unpack* the slots, so as to put each coefficient in its own slot, which results in d "sparsely packed" ciphertexts (as described in Section 4.3).

Next we apply the digit-extraction procedure from Section 5.3 to each one of these d "sparsely packed" ciphertexts. For each one we extract the digits upto $e+r-e'$ (or upto $e+r-e'-1$ if $p=2$ and $r>2$), and combine the top digits as per Lemma 5.1 to get in the slots the coefficients of the plaintext polynomial $\mathfrak{m}$ (one coefficient per slot). The resulting ciphertexts all have plaintext space mod-p^r.

Next we re-combine the d ciphertext into a single fully-packed ciphertext (as described in Section 4.3) and finally apply the second linear transformation (the map Eval described in Section 4.2). This completes the recryption procedure.

6 Implementation and Performance

As discussed in Section 4.2, our algorithms for the linear transformations rely on the parameter m having a fairly special form. Luckily, there are quite a few such m's, which we found by brute-force

cyclotomic ring m	21845 $=257{\cdot}5{\cdot}17$	18631 $=601{\cdot}31$	28679 $=241{\cdot}17{\cdot}7$	35113 $=(73{\cdot}13){\cdot}37$
lattice dim. $\phi(m)$	16384	18000	23040	31104
plaintext space	GF(2^{16})	GF(2^{25})	GF(2^{24})	GF(2^{36})
number of slots	1024	720	960	864
security level	76	110	96	159
before/after levels	22/10	20/10	24/11	24/12
initialization (sec)	177	248	224	694
linear transforms (sec)	127	131	123	325
digit extraction (sec)	193	293	342	1206
total recrypt (sec)	**320**	**424**	**465**	**1531**
space usage (GB)	3.4	3.5	3.5	8.2

Table 1: Experimental results with plaintext space GF(2^d)

cyclotomic ring m	45551 $=(41{\cdot}11){\cdot}101$	51319 $=(19{\cdot}73){\cdot}37$	42799 $= 337{\cdot}127$	49981 $=331{\cdot}151$
lattice dim. $\phi(m)$	40000	46656	42336	49981
plaintext space	GF(17^{40})	GF(127^{36})	$R(256, 21)$	$R(256, 30)$
number of slots	1000	1296	2016	1650
security level	106	161	79	91
before/after levels	38/10	32/11	52/6	56/10
initialization (sec)	1148	2787	1202	1533
linear transforms (sec)	735	774	2265	2834
digit extraction (sec)	3135	1861	8542	14616
total recrypt (sec)	**3870**	**2635**	**10807**	**17448**
space usage (GB)	14.8	39.9	15.6	21.6

Table 2: Experimental results with other plaintext spaces

search. We ran a simple program that searches through a range of possible m's (odd, and not divisible by p, and not prime). For each such m, we first compute the order d of p mod m. If this exceeds a threshold (we chose a threshold of 100), we skip this m. Next, we compute the factorization of m into prime powers as $m = m_1 \cdots m_t$. We then find all indexes i such that p has order d mod m_i and all pairs of indexes i, j such that p has order d mod $m_i m_j$. If we find none, we skip this m; otherwise, we choose one such index, or pair of indexes, in such a way to balance the time and depth complexity of the linear transformations (so m_i or $m_i m_j$ becomes the new m_1, and the other prime power factors are ordered arbitrarily).

For example, with $p = 2$, we processed all potential m's between 16,000 and 64,000. Among these, there were a total of 377 useful m's with $15,000 \leq \phi(m) \leq 60,016$, with a fairly even spread (the largest gap between successive $\phi(m)$'s was less than 2,500, and there were no other gaps that exceeded 2,000). So while such useful m's are relatively rare, there are still plenty to choose from. We ran this parameter-generation program to find potential settings for plaintext-space modulo $p = 2, p = 17$, $p = 127$, and $p^r = 2^8$, and manually chose a few of the suggested values of m for our tests.

For each of these values of m, p, r, we then run a test in which we chose three random keys, and performed recryption three times per key (for each key recrypting the same ciphertext over and over). These tests were run on a five-year-old IBM BladeCenter HS22/7870, with two Intel X5570 (4-core) processors, running at 2.93GHz. All of our programs are single-threaded, so only one core was used in the computations. Tables 1 and 2 summarize the results form our experiments.

In each table, the first row gives m and its factorization into prime powers. The first factor (or pair of factors, if grouped by parentheses) shows the value that was used in the role of m_1 (as in

Section 4.2). The second row gives $\phi(m)$. The third row gives the plaintext space, i.e., the field/ring that is embedded in each slot (here, $R(p^r, d)$ means a ring extension of degree d over $\mathbb{Z}/(p^r)$). The fourth row gives the number of slots packed into a single ciphertext. The fifth row gives the effective security level, computed using the formula that is used in HElib, taken from [11, Eqn.(8)]. The sixth row gives the levels of ciphertext just before recryption (i.e., the ciphertext which is included in the public key) and just after the end of the recryption procedure. The difference accounts for the depth of recryption, and the after-levels is roughly the circuit-depth that an application can compute on the resulting ciphertext before having to recrypt again. We tried to target 10 remaining levels, to allow nontrivial processing between recryptions.

The remaining rows show the resources used in performing a recryption. The timing results reflect the average of the 9 runs for each setting, and the memory usage is the top usage among all these runs. Row 7 gives a one-time initialization cost (all times in seconds). Row 10 (in boldface) gives the total time for a single recryption, while the previous two rows give a breakdown of that time (note that the time for the linear transforms includes some trivial preprocessing time, as well as the less trivial unpacking/repacking time). The last row gives the memory used (in gigabytes).

7 Future work

Eventually, we would like to enhance our implementations to take advantage of multicore computing environments. There are at least two levels at which the implementation could be easily parallelized. At a low level, the conversions between DoubleCRT and polynomial representation in HElib could be easily parallelized, as the FFT's for the different primes can be done in parallel (as already observed in [24]). Our bootstrapping procedure could also be parallelized at a higher level: the rotations in each stage of the linear transformation step can be done in parallel, as can the d different digit extraction steps. Doing the parallel steps at a higher level could possibly yield a better work/overhead ratio, but this would have to be confirmed experimentally.

Another direction to explore is the possibility of speeding up the digit extraction procedure in the special case where the values in the ciphertext slots are constants in the base ring $\mathbb{Z}/(p^r)$ (or, more generally, lie in some sub-ring of the ring E contained in each slot). Right now, our bootstrapping algorithm does not exploit this: even in this special case, our digit extraction algorithm still has to be applied to d ciphertexts. In principle, we should be able to reduce the number of applications of the digit extraction significantly (from d to 1, if the values in the slots are constants); however, it is not clear how to do this while maintaining the structure (and therefore efficiency) of the linear transformations.

Another direction to explore is to try to find a better way to represent constants. In HElib, the most compact way to store constants in R_{p^r} is also the most natural: as coefficient vectors of polynomials over $\mathbb{Z}/(p^r)$. However, in this representation, a surprisingly significant amount of time may be spent in homomorphic computations converting these constants to DoubleCRT format. One could precompute and store these DoubleCRT representations, but this can be quite wasteful of space, as DoubleCRT's occupy much more space than the corresponding polynomials over $\mathbb{Z}/(p^r)$. We may state as an open question: is there a more compact representation of elements of $\mathbb{Z}/(p^r)[X]$ that can be converted to DoubleCRT format in linear time?

References

[1] J. Alperin-Sheriff and C. Peikert. Practical bootstrapping in quasilinear time. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO'13*, volume 8042 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2013.

[2] J. Alperin-Sheriff and C. Peikert. Faster bootstrapping with polynomial error. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology - CRYPTO 2014, Part I*, pages 297–314. Springer, 2014.

[3] 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.

[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory*, 6(3):13, 2014.

[5] Z. Brakerski and V. Vaikuntanathan. Lattice-based FHE as secure as PKE. In M. Naor, editor, *Innovations in Theoretical Computer Science, ITCS'14*, pages 1–12. ACM, 2014.

[6] L. Ducas and D. Micciancio. FHE Bootstrapping in less than a second. Cryptology ePrint Archive, Report 2014/816, 2014. `http://eprint.iacr.org/`.

[7] 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.

[8] C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology - EUROCRYPT'11*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.

[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] C. Gentry, S. Halevi, and N. Smart. Homomorphic evaluation of the AES circuit. In *"Advances in Cryptology - CRYPTO 2012"*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012. Full version at `http://eprint.iacr.org/2012/099`.

[12] C. Gentry, S. Halevi, and N. P. Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography - PKC 2012*, volume 7293 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[13] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013, Part I*, pages 75–92. Springer, 2013.

[14] S. Halevi and V. Shoup. Algorithms in HElib. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology - CRYPTO 2014, Part I*, pages 554–571. Springer, 2014. Long version at `http://eprint.iacr.org/2014/106`.

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

[16] 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.

[17] 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.

[18] V. Lyubashevsky, C. Peikert, and O. Regev. ”a toolkit for ring-LWE cryptography”. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013*, pages 35–54. Springer, 2013.

[19] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43, 2013. Early version in EUROCRYPT 2010.

[20] E. Orsini, J. van de Pol, and N. P. Smart. Bootstrapping BGV ciphertexts with a wider choice of p and q. Cryptology ePrint Archive, Report 2014/408, 2014. `http://eprint.iacr.org/`.

[21] M. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM J. Comput.*, 2(1):60–66, 1973.

[22] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), 2009.

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

[24] K. Rohloff and D. B. Cousins. A scalable implementation of fully homomorphic encryption built on NTRU. 2nd Workshop on Applied Homomorphic Cryptography and Encrypted Computing, WAHC’14, 2014. Available at `https://www.dcsec.uni-hannover.de/fileadmin/ful/mitarbeiter/brenner/wahc14_RC.pdf`, accessed September 2014.

[25] S. Roman. *Field Theory.* Springer, 2nd edition, 2005.

[26] N. P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptography*, 71(1):57–81, 2014. Early verion at `http://eprint.iacr.org/2011/133`.

A Parameters for Digit Extraction

Here we explain our choice of parameters for the recryption procedure (e, e', α, etc.). These parameters depend on the cyclotomic ring R_m, plaintext space p^r, and the l_1-norm of the recryption secret key $\tilde{\mathsf{sk}}$ (which we denote t).

We begin the recryption procedure with a noise-n ciphertext $(\mathfrak{c}_0, \mathfrak{c}_1)$, relative to plaintext space p^r, a secret key $\tilde{\mathsf{sk}} = (1, \tilde{\mathfrak{s}})$ with $\|\mathfrak{s}\|_1 \leq t$, and modulus $\tilde{q} = p^e + 1$. This means that for the element $\mathfrak{u} \leftarrow \langle \tilde{\mathsf{sk}}, \mathsf{ct} \rangle$ (without modular reduction) we have $\|[\mathfrak{u}]_q\|_\infty < n$ and $\|\mathfrak{u}\|_\infty < (t+1)q/2$, and that the plaintext element encrypted in ct is $m \leftarrow [[\mathfrak{u}]_q]_{p^r}$. [7] We then make the coefficients of ct divisible by $p^{e'}$ using Lemma 5.2, thus getting another ciphertext

$$\mathsf{ct}' = (\mathfrak{c}_0', \mathfrak{c}_1') \;=\; (\mathfrak{c}_0 + p^r \mathfrak{u}_0 + q\mathfrak{v}_0,\ \mathfrak{c}_1 + p^r \mathfrak{u}_1 + q\mathfrak{v}_1).$$

Consider the effect of this modification on the coefficients of $\mathfrak{u}' \leftarrow \langle \tilde{\mathsf{sk}}, \mathsf{ct}' \rangle = \mathfrak{c}_0' + \tilde{\mathfrak{s}} \cdot \mathfrak{c}_1$. Clearly we have increased both the noise (due to the added p^r terms) and the magnitude of the coefficients

[7] The term $(t+1)q/2$ assumes no “ring constant”, i.e. $\|\mathfrak{s} \cdot \mathfrak{c}_1\| \leq \|\mathfrak{s}\|_1 \cdot \|\mathfrak{c}_1\|$. This is not always true but it makes a reasonable heuristic, and we use it for most of this section.

(mostly due to the added q terms). Specifically, we now have

$$\begin{aligned}
\|\mathfrak{u}'\| &\leq \|\mathfrak{u}\| + (\|\mathfrak{u}_0\| + t\|\mathfrak{u}_1\|)p^r + (\|\mathfrak{v}_0\| + t\|\mathfrak{v}_1\|)q \\
&\leq (t+1)\Big(\underbrace{\frac{q}{2} + \left\lceil\frac{\alpha p^{r+e'-1}}{2}\right\rceil}_{<q} + qp^r\big(\frac{1}{2} + \left\lfloor\frac{(1-\alpha)p^{e'-1}}{2}\right\rfloor\big)\Big) \\
&\leq (t+1)q\big(1 + (1-\alpha)p^{r+e'-1}/2 + p^r/2\big), \\
\|[\mathfrak{u}']_q\| &\leq \|[\mathfrak{u}]_q\| + (\|\mathfrak{u}_0\| + t\|\mathfrak{u}_1\|)p^r \;\leq\; n + (t+1)\lceil \alpha p^{r+e'-1}/2\rceil \;\leq\; n + (t+1)(1 + \alpha p^{r+e'-1}/2).
\end{aligned}$$

To be able to still use Lemma 5.1 we need to have $\|[\mathfrak{u}']_q\| < q/4$ and $\|\mathfrak{u}'\| < q^2/4 - q$. Namely we need both

$$n + (t+1)(1 + \alpha p^{r+e'-1}/2) < q/4 \quad \text{and} \quad (t+1)\big(1 + (1-\alpha)p^{r+e'-1}/2 + p^r/2\big) < q/4 - 1,$$

or in other words

$$q/4 \geq \max\left\{\begin{array}{ll} (t+1)\big(1 + \frac{\alpha p^{r+e'-1}}{2}\big) & +\; n, \\ (t+1)\big(1 + \frac{(1-\alpha)p^{r+e'-1}}{2}\big) & + \frac{(t+1)p^r}{2} + 1 \end{array}\right\}. \tag{8}$$

To get good parameters we would like to set α, e' such that these two constraints are roughly equivalent. Ignoring for simplicity the $+1$ at the end of the bottom constraint, we would want to set the parameters so that

$$(t+1)\big(1 + \frac{\alpha p^{r+e'-1}}{2}\big) + n = (t+1)\big(1 + \frac{(1-\alpha)p^{r+e'-1}}{2}\big) + \frac{(t+1)p^r}{2} \;\Leftrightarrow\; \alpha = \frac{1}{2} - \frac{n - (t+1)p^r/2}{(t+1)p^{r+e'-1}}.$$

Note that with out parameters the noise n is much larger than $(t+1)p^r/2$: The noise after modulus-switching is at least as large as the modulus-switching added factor (cf. [4, Lemma 4]), and the heuristic estimate for that added factor (taken from the `HElib` design document) is $p^r \cdot \sqrt{(t+1)\phi(m)/12}$. Since we use Hamming weight $t \ll \phi(m)$ for the secret key $\tilde{\mathfrak{s}}$, then $\sqrt{(t+1)\phi(m)} \gg (t+1)$, which means that $n \approx p^r \cdot \sqrt{(t+1)\phi(m)} \gg p^r \cdot (t+1)$. Hence to get good parameters we need $\alpha \approx \frac{1}{2} - n/((t+1)p^{r+e'-1}$, and since we can only use $\alpha \in [0,1]$ then it means that we need to set e' large enough in order to get $\alpha > 0$, and α tends to $1/2$ as e' grows.

To get a first estimate, we assume that we have e, e' large enough to get $\alpha \approx 1/2$, and we analyze how large must we make $e - e'$. With $\alpha \approx 1/2$ and the two terms in Eqn. (8) roughly equal, we can simplify that equation to get

$$q/4 \;=\; (p^e+1)/4 \;>\; (t+1)\big(1 + \frac{p^{r+e'-1}}{4} + \frac{p^r}{2}\big) + 1.$$

With $e' \gg 1$ the most significant term on the right-hand side is $(t+1)p^{r+e'-1}/4$, so we can simplify further to get $p^e/4 > (t+1+\epsilon)p^{r+e'-1}/4$ (with ϵ a small quantity that captures all the low-order terms), or $e - e' > r - 1 + \log_p(t+1+\epsilon)$. In our implementation we therefore try to use the setting $e - e' = r - 1 + \lceil\log_p(t+2)\rceil$, and failing that we use $e - e' = r + \lceil\log_p(t+2)\rceil$.

In more detail, on input m, p, r we set an initial value of $t = 56$, then set $\gamma \overset{\text{def}}{=} (t+1)/p^{\lceil\log_p(t+2)\rceil}$. Plugging $e - e' = r - 1 + \lceil\log_p(t+2)\rceil$ in Eqn. (8) and ignoring some '+1' terms, we get

$$p^e > \max\left\{\frac{4(t+n)}{1-2\alpha\gamma}\;,\;\; \frac{2(t+1)p^r}{1-2(1-\alpha)\gamma}\right\}. \tag{9}$$

m	p^r	e	e'	α	t	B	c_m	Comments
21854	2	15	9	0.453112	56	23	16.0	
18631	2	15	9	0.45291	56	23	0.5	
28679	2	15	9	0.452414	56	23	10.0	
35115	2	13	6	0	59	25	4.0	"conservative" flag
45551	17	4	2	0	134	25	20.0	
51319	127	3	2	0	56	25	2.0	forced $t = 56$
42799	2^8	23	10	0.451488	57	25	0.2	frequent mod-switching
49981	2^8	23	10	0.451254	57	25	1.0	frequent mod-switching

Table 3: Parameters in our different tests. B is the width (in bits) of levels in the modulus-chain, and c_m is the experimental "ring constant" that we used.

For the noise n we substitute twice the modulus-switching added noise term, $n \overset{\text{def}}{=} p^r\sqrt{(t+1)\phi(m)/3}$, and then we solve for the value $\alpha \in [0,1]$ that minimizes the right-hand side of Eqn. (9). This gives us a lower-bond on p^e.

Next we check that this lower-bound is not too big: recall that at the beginning of the recryption process we multiply the ciphertext $\tilde{\mathfrak{ct}}$ from the public key by the "constant" $\mathfrak{c}'_1/p^{e'}$, whose entries can be as large as $q^2/(4p^{e'}) \approx p^{2e-e'-2}$. Hence as we increase e we need to multiply by a larger constant, and the noise grows accordingly. In the implementation we define "too big" (somewhat arbitrarily) to be anything more than half the ratio between two successive moduli in our chain. If p^e is "too big" then we reset $e - e'$ to be one larger, which means re-solving the same system but this time using $\gamma' = \gamma/p$ instead of γ.

Once we computed the values e, e', α, we finally check if it is possible to increase our initial $t = 56$ (i.e., the recryption key weight) without violating Eqn. (9). This gives us the final values for all of our constants. We summarize the parameters that we used in our tests in Table 3.

Caveats. The BGV implementation in `HElib` relies on a myriad of parameters, some of which are heuristically chosen, and so it takes some experimentation to set them all so as to get a working implementation with good performance. Some of the adjustments that we made in the course of our testing include the following:

- `HElib` relies on a heuristic noise estimate in order to decide when to perform modulus-switching. One inaccuracy of that estimate is that it assumes that $\|xy\| \le \|x\| \cdot \|y\|$, which does not quite hold for the bases that are used in `HElib` for representing elements in the ring $R = \mathbb{Z}[X]/(\Phi_m(X))$. To compensate, the library contains a "ring constant" c_m which is set by default to 1 but can be adjusted by the calling application, and then it sets $\mathsf{estimate}(\|xy\|) := \mathsf{estimate}(\|x\|) \cdot \mathsf{estimate}(\|y\|) \cdot c_m$. In our tests we often had to set that constant to a larger value to get accurate noise estimation — we set the value experimentally so as to get good estimate for the noise at the output of the recryption procedure.

- The same "ring constant" might also affect the setting of the parameters e, e', α from above. Rather than trying to incorporate it into the calculation, our implementation just provides a flag that forces us to forgo the more aggressive setting of $e - e' = r - 1 + \lceil \log_p(t+2) \rceil$, and instead always use the more conservative $e - e' = r + \lceil \log_p(t+2) \rceil$. The effect is that we have to extract one more digit during the digit extraction part, but it ensures that we do not get recryption errors from the use of our simplified decryption formula. In our tests we had to use this "conservative" flag for the tests at $m = 35113$.

- Also, we sometimes had to manually set the Hamming weight of the recryption key to a lower value than what our automatic procedure suggests, to avoid recryption errors. This happened for the setting $p = 127, m = 51319$, where the automated procedure suggested to use $t = 59$ but we had to revert back to $t = 56$ to avoid errors.

- The "width" of each level (i.e., the ratio q_{i+1}/q_i in the modulus chain) can be adjusted in HElib. The trade-off is that wider levels give better noise reduction, but also larger overall moduli (and hence lower levels of security). The HElib implementation uses by default 23 bits per level, which seems to work well for values of $m < 30000$ and $p^r = 2$. For our tests, however, this was sometime not enough, and we had to increase it to 25 bits per level.

  For the tests with plaintext space modulo 2^8, even 25 bits per level were not quite enough. However for various low-level reasons (having to do with the NTL single-precision bounds), setting the bit length to 26 bits or more is not a good option. Instead we changed some of the internals of HElib, making it use modulus-switching a little more often than its default setting, while keeping the level width at 25 bits. As a result, for that setting we used many more levels than for all the other settings (an average of 1.5 levels per squaring).

B Homomorphic Polynomial Evaluation

Our polynomial-evaluation modules implements homomorphic evaluation of a cleartext polynomial at an encrypted point. We roughly implement the Paterson-Stockmeyer algorithm [21], but with several tweaks to minimize the multiplication depth of the resulting circuit.

B.1 The Paterson-Stockmeyer Algorithm

This algorithm is a variant of the baby-step/giant-step approach for computing a degree-n polynomial using $O(\sqrt{n})$ products, but it achieves a better constant in the $O(\cdot)$ than the naive approach.

Recall the basic approach: for a cleartext polynomial $f(X) = \sum_{i=0}^{n-1} f_i X^i$ and an encryption of the indeterminate X, we set some parameter $s = O(\sqrt{n})$ and denote $t = \lfloor n/s \rfloor$. We compute homomorphically the "baby" powers of $X, X^2, X^3 \ldots, X^s$, and the "giant" powers $X^{2s}, X^{3s}, \ldots, X^{t \cdot s}$. Then for $i = 0, 1, \ldots, t-1$ we can compute $Y_i = \sum_{j=0} s - 1 f_{i \cdot s + j} X^j$ using only multiply-by-constant and addition operations, and complete the evaluation of f as $Y = \sum_{i=0}^{s-1} X^{i \cdot s} \cdot Y_i$. This simple procedure takes $s + t - 2$ multiplications for computing the different powers and $t - 1$ more for completing the evaluation, for a total of $s + 2t - 3$. Setting $s = \lceil \sqrt{2n} \rceil$ we get roughly $2\sqrt{2} \cdot \sqrt{n}$ multiplications.

The Paterson-Stockmeyer algorithm reduces the number of multiplies by roughly a factor of two, computing the same polynomial using only $\sqrt{2n} + O(\log n)$ multiplications. Differently than the naive approach, they multiply not just powers of X by the polynomials $f_i(X)$ but different polynomials by each other. The simplest variant of this algorithm applies to monic polynomials whose degree is of the form $n = (2m - 1)s$ with m a power of two.

Here we compute the "baby step" powers $X^2, \ldots, X^s$, and "giant step" powers $X^{2s}, X^{4s}, X^{8s}, \ldots X^{ms}$. Then we break f into a monic top half of degree $(m-1)s$ and a bottom half of degree $\leq ms - 1$:

$$f(X) = X^{ms} \underbrace{\Big(\sum_{i=0}^{(m-1)s} f_{ms+i} X^i \Big)}_{q(X)} + \underbrace{\Big(\sum_{i=0}^{ms-1} f_i X^i \Big)}_{r(X)} .$$

Let us now denote $r'(X) = X^{(m-1)s} + r(X)$. Since $q(X)$ is monic and $\deg(r') < \deg(q) + s$, we can use polynomial division with remainder to find two polynomials $c(X), d(X)$ such that

$$r'(X) = c(X) \cdot q(X) + d(X),$$

with $\deg(c) < s$ and $\deg(d) < \deg(q) = (m-1)s$.

Now we can add $X^{(m-1)s}$ to $d(X)$ to get a monic degree-$(m-1)s$ polynomial $d'(X) = X^{(m-1)s} + d(X)$. Now we can apply the same procedure recursively to the two monic polynomials $q(X), d'(X)$, compute $c(X)$ from the baby-step powers using only additions and multiply-by-constant operations, and complete the evaluation by setting

$$\begin{aligned} Y &= \big(X^{ms} + C(X)\big) \cdot q(X) + d'(X) \;=\; X^{ms} \cdot q(X) + C(X) \cdot q(X) + \big(d(X) - X^{(m-1)s}\big) \\ &= X^{ms} \cdot q(X) + r'(X) - X^{(m-1)s} \;=\; X^{ms} \cdot q(X) + r(X) \;=\; f(X). \end{aligned}$$

Complexity. For a polynomial of degree $n = (2^t - 1)s$, the algorithm above takes $s + t - 2$ multiplications to compute the baby-step and giant-step powers. Once they are computed the complexity of the recursive procedure is captured by the recursion formula $T(2^t - 1) = 2T(2^{t-1} - 1) + 1$. Solving this recursion with base-case $T(2^1 - 1) = T(1) = 0$ we get $T(2^t - 1) = 2^{t-1} - 1$. Hence the total number of multiplications that is needed is $s + t + 2^{t-1} - 3 \leq s + \lceil n/2s \rceil + \lceil \log(n/s) \rceil - 3$. Setting $s = \lceil \sqrt{n/2} \rceil$ yields total number of at most $\lceil \sqrt{2n} + \log(n)/2 \rceil$ multiplications.

B.2 Our Modifications

In our context we need to modify the procedure above to handle non-monic polynomials or arbitrary degree, and also to account for the depth of the resulting multiplication circuit.

Minimizing depth. To see why the depth may matter, consider running the "naive" baby-step/giant-step procedure to evaluate a degree-14 polynomial:

- It is not hard to check that without depth consideration it is best to set the baby-step parameter to $s = 5$, yielding a total of 7 multiplications: five for the powers X^2, X^3, X^4, X^5 and X^{10} and two more for evaluating $f(X) = f_0(X) + X^5 \cdot f_1(X) + X^{10} \cdot f_2(X)$, with each of the f_i's having degree 4.

  However this setting yields a depth-5 circuit: Computing X^5 takes a depth-3 circuit, squaring it to get X^{10} makes depth-4 and then multiplying $X^{10} \cdot f_2(X)$ makes depth-5.

- On the other hand, setting $s = 4$ yields 8 multiplications: five for the powers X^2, X^3, X^4 and X^8, X^{12}, and three more for $f(X) = f_0(X) + X^4 \cdot f_1(X) + X^8 \cdot f_2(X) + X^{12} \cdot f_3(X)$, with f_0, f_1, f_2's having degree 3 and f_3 having degree 2.

  At the same time, this setting can be computed by a depth-4 circuit (which is optimal in this case): we have X^8 at depth-3, and then $X^8 \cdot f_3(X)$ at depth-4.

(Similar phenomena arises also for the Paterson-Stockmeyer procedure above, but the examples become longer.) Since for homomorphic operations circuit-depth is at a premium, we are willing to pay for a smaller depth by somewhat increasing the number of multiplications.

We note that the recursive Paterson-Stockmeyer procedure itself yields near-optimal depth for each given value of the baby-step parameter s: If computing x^s takes depth-d, then using the Paterson-Stockmeyer procedure for computing a polynomial of degree upto $s(2^t - 1)$ takes depth $d + t$ (whereas the naive baby-step/giant-step procedure with depth $d + t$ can evaluate polynomials of degree upto $s2^t$).

We also note that one can always ensure optimal depth by setting the baby-step parameter s as a power of two (since this is the largest power than can be computed in a given depth). In our implementation, we therefore always choose s as a power of two; namely, we use $\sqrt{\deg(P)/2}$ rounded either up or down to a power of two, and use a special case for $s(2^t-1) < \deg(P) \le s(2^t-1)$. [8]

Non-monic polynomials. Handling non-monic polynomials whose leading coefficient is invertible (modulo the plaintext space p) is simple: we multiply by the inverse, then evaluate the resulting monic polynomial, and finally multiply the result by the top coefficient.

If the leading coefficient is not invertible mod p then we either add one to it, or add one to a higher power of X. In either case we then evaluate the polynomial and subtract the appropriate power X^m from the result. The decision of which power of X to add and subtract is done so as to minimize the number of extra multiplications that are needed for computing X^m.

Monic polynomials of arbitrary degree. Once we have a monic polynomial $P(X)$ and we decided on the baby-step parameter s to use, we consider four cases. In all the cases below let t be the smallest integer such that $\deg(P) \le s \cdot 2^t$.

- If P is of degree $s(2^t-1)$ then we directly apply the Paterson-Stockmeyer procedure.
- If $s(2^t-2) < \deg(P) < s(2^t-1)$ then we add to it $X^{s(2^t+1)}$, evaluate the result using the Paterson-Stockmeyer procedure, then subtract the power $X^{s(2^t+1)}$. [9]
- If $s(2^t-1) < \deg(P) \le s2^t$ then use Paterson-Stockmeyer for the bottom $s(2^t-1)$ part and compute the top $\le s$ terms separately.

  Specifically let $P_{lo}(X)$ be the bottom $s(2^t-1)$ coefficients and $P_{lo}(X)$ be the top ones, we set $Q_{lo}(X) = X^{s(2^t-1)} + P_{lo}(X)$ and evaluate it using the Paterson-Stockmeyer procedure. Also let $Q_{hi}(X) = P_{hi}(X) - 1$ and evaluate it using the baby-step powers that we computed. Then we also compute the power $X^{s(2^t-1)}$ (in minimum depth by multiplying t of the giant-step powers) and finally set $P(X) = Q_{hi}(X) \cdot X^{s(2^t-1)} + Q_{lo}(X)$.

  It is easy to verify that this procedure takes depth exactly $t + \log s$, which is optimal.
- In any other case, $\deg(P) \le s(2^t-2)$, we use Paterson-Stockmeyer for the bottom $s(2^t-1)$ part and compute the top $\le s$ terms recursively. This case is very similar to the case above for $s(2^t-1) < \deg(P) \le s2^t$, except that we cannot compute $Q_{hi}(X)$ using only the baby-step powers. Instead simply make a recursive call to evaluate Q_{hi}, noting that it contains less than half the coefficients of P.

C Why We Didn't Use Ring Switching

One difference between our implementation and the procedure described by Alperin-Sheriff and Peikert [1] is that we do not use the ring-switching techniques of Gentry et al. [9] to implement the tensor decomposition of our Eval transformation and its inverse. There are several reasons why we believe that an implementation based on ring switching is less appealing in our context, especially for the smaller parameter settings (say, $\phi(m) < 30000$). The reasoning behind this is as follows:

[8] A better option, which we did not implement yet, would be to search for all the values of s between the two enclosing powers of two, and use the value that minimized the number of multiplications subject to keeping the smallest depth.

[9] Of course we handle this case together with the non-invertible top coefficient from above, so that we never add two power of X.

Rough factorization of m. Since the non-linear part of our recryption procedure takes at least seven levels, and we target having around 10 levels left at the end of recryption, it means that for our smaller examples we cannot afford to spend too many levels for the linear transformations. Since every stage of the linear transformation consumes at least half a level,[10] then for such small parameters we need very few stages. In other words, we have to consider fairly coarse-grained factorization of m, where the factors have sizes m^{ϵ} for a significant ϵ (as large as $\sqrt{m}$ in some cases).

Using large rings. Recall that the first linear transformation during recryption begins with the fresh ciphertext in the public key (after multiplying by a constant). That ciphertext has very low noise, so we have to process it in a large ring to ensure security.[11] This means that we must *switch up* to a much larger ring before we can afford to drop these rough factors of m. Hence we will be spending most of our time on operations in very large rings, which defeats the purpose of targeting these smaller sub-30000 rings in the first place.

We also note that in our tests, the recryption time is dominated by the non-linear part, so our implantation seems close to optimal there. It is plausible that some gains can be made by using ring switching for the second linear transformation, after the non-linear part, but we did not explore this option in our implementation. And as we said above, there is not much to be gained by optimizing the linear transformations.

[10] Whether or not we use ring-switching, each stage of the linear transformation has depth of at least one multiply-by-constant, which consumes at least half a level in terms of added noise.

[11] More specifically, the key-switching matrices that allow us to process it must be defined in a large ring.