Implementing Gentry's Fully-Homomorphic Encryption Scheme

Craig Gentry Shai Halevi

IBM Research

February 4, 2011

Abstract

We describe a working implementation of a variant of Gentry's fully homomorphic encryption scheme (STOC 2009), similar to the variant used in an earlier implementation effort by Smart and Vercauteren (PKC 2010). Smart and Vercauteren implemented the underlying "somewhat homomorphic" scheme, but were not able to implement the bootstrapping functionality that is needed to get the complete scheme to work. We show a number of optimizations that allow us to implement all aspects of the scheme, including the bootstrapping functionality.

Our main optimization is a key-generation method for the underlying somewhat homomorphic encryption, that does not require full polynomial inversion. This reduces the asymptotic complexity from $\tilde{O}(n^{2.5})$ to $\tilde{O}(n^{1.5})$ when working with dimension-n lattices (and practically reducing the time from many hours/days to a few seconds/minutes). Other optimizations include a batching technique for encryption, a careful analysis of the degree of the decryption polynomial, and some space/time trade-offs for the fully-homomorphic scheme.

We tested our implementation with lattices of several dimensions, corresponding to several security levels. From a "toy" setting in dimension 512, to "small," "medium," and "large" settings in dimensions 2048, 8192, and 32768, respectively. The public-key size ranges in size from 70 Megabytes for the "small" setting to 2.3 Gigabytes for the "large" setting. The time to run one bootstrapping operation (on a 1-CPU 64-bit machine with large memory) ranges from 30 seconds for the "small" setting to 30 minutes for the "large" setting.

1 Introduction

Encryption schemes that support operations on encrypted data (aka homomorphic encryption) have a very wide range of applications in cryptography. This concept was introduced by Rivest et al. shortly after the discovery of public key cryptography [14], and many known public-key cryptosystems support either addition or multiplication of encrypted data. However, supporting both at the same time seems harder, and until very recently all the attempts at constructing so-called "*fully homomorphic*" encryption turned out to be insecure.

In 2009, Gentry described the first plausible construction of a fully homomorphic cryptosystem [4]. Gentry's construction consists of several steps: He first constructed a "*somewhat homomorphic*" scheme that supports evaluating low-degree polynomials on the encrypted data, next he needed to "*squash*" the decryption procedure so that it can be expressed as a low-degree polynomial which is supported by the scheme, and finally he applied a "*bootstrapping*" transformation to obtain a fully homomorphic scheme. The crucial point in this process is to obtain a scheme that can evaluate polynomials of high-enough degree, and at the same time has decryption procedure that can be expressed as a polynomial of low-enough degree. Once the degree of polynomials that can be evaluated by the scheme exceeds the degree of the decryption polynomial (times two), the scheme is called "*bootstrappable*" and it can then be converted into a fully homomorphic scheme.

Toward a bootstrappable scheme, Gentry described in [4] a somewhat homomorphic scheme, which is roughly a GGH-type scheme [8, 10] over ideal lattices. Gentry later proved [5] that with an appropriate key-generation procedure, the security of that scheme can be (quantumly) reduced to the worst-case hardness of some lattice problems in ideal lattices.

This somewhat homomorphic scheme is not yet bootstrappable, so Gentry described in [4] a transformation to squash the decryption procedure, reducing the degree of the decryption polynomial. This is done by adding to the public key an additional hint about the secret key, in the form of a "*sparse subset-sum*" problem (SSSP). Namely the public key is augmented with a big set of vectors, such that there exists a very sparse subset of them that adds up to the secret key. A ciphertext of the underlying scheme can be "*post-processed*" using this additional hint, and the post-processed ciphertext can be decrypted with a low-degree polynomial, thus obtaining a bootstrappable scheme.

Stehlé and Steinfeld described in [18] two optimizations to Gentry's scheme, one that reduces the number of vectors in the SSSP instance, and another that can be used to reduce the degree of the decryption polynomial (at the expense of introducing a small probability of decryption errors). We mention that in our implementation we use the first optimization but not the second.[1] Some improvements to Gentry's key-generation procedure were discussed in [11].

1.1 The Smart-Vercauteren implementation

The first attempt to implement Gentry's scheme was made in 2010 by Smart and Vercauteren [17]. They chose to implement a variant of the scheme using "*principal-ideal lattices*" of prime determinant. Such lattices can be represented implicitly by just two integers (regardless of their dimension), and moreover Smart and Vercauteren described a decryption method where the secret key is represented by a single integer. Smart and Vercauteren were able to implement the underlying somewhat homomorphic scheme, but they were not able to support large enough parameters to make Gentry's squashing technique go through. As a result they could not obtain a bootstrappable scheme or a fully homomorphic scheme.

One obstacle in the Smart-Vercauteren implementation was the complexity of key generation for the somewhat homomorphic scheme: For one thing, they must generate very many candidates before they find one whose determinant is prime. (One may need to try as many as $n^{1.5}$ candidates when working with lattices in dimension n.) And even after finding one, the complexity of computing the secret key that corresponds to this lattice is at least $\tilde{\Theta}(n^{2.5})$ for lattices in dimension n. For both of these reasons, they were not able to generate keys in dimensions $n > 2048$.

Moreover, Smart and Vercauteren estimated that the squashed decryption polynomial will have degree of a few hundreds, and that to support this procedure with their parameters they need to use lattices of dimension at least $n = 2^{27} (\approx 1.3 \times 10^8)$, which is well beyond the capabilities of the key-generation procedure.

1.2 Our implementation

We continue in the same direction of the Smart-Vercauteren implementation and describe optimizations that allow us to implement also the squashing part, thereby obtaining a bootstrappable scheme and a fully homomorphic scheme.

For key-generation, we present a new faster algorithm for computing the secret key, and also eliminate the requirement that the determinant of the lattice be prime. We also present many sim-

[1] The reason we do not use the second optimization is that the decryption error probability is too high for our parameter settings.

plifications and optimizations for the squashed decryption procedure, and as a result our decryption polynomial has degree only fifteen. Finally, our choice of parameters is somewhat more aggressive than Smart and Vercauteren (which we complement by analyzing the complexity of known attacks).

Differently from [17], we decouple the dimension n from the size of the integers that we choose during key generation.[2] Decoupling these two parameters lets us decouple functionality from security. Namely, we can obtain bootstrappable schemes in any given dimension, but of course the schemes in low dimensions will not be secure. Our (rather crude) analysis suggests that the scheme may be practically secure at dimension $n = 2^{13}$ or $n = 2^{15}$, and we put this analysis to the test by publishing a few challenges in dimensions from 512 up to 2^{15}.

1.3 Organization

We give some background in Section 2, and then the report is organized in two parts. In Part I we describe our implementation of the underlying "somewhat homomorphic" encryption scheme, and in Part II we describe our optimizations that are specific to the bootstrapping functionality. To aid reading, we list here all the optimizations that are described in this report, with pointers to the sections where they are presented.

Somewhat-homomorphic scheme.

1. We replace the Smart-Vercauteren requirement [17] that the lattice has prime determinant, by the much weaker requirement that the Hermite normal form (HNF) of the lattice has a particular form, as explained in Step 3 of Section 3. We also provide a simple criterion for checking for this special form.

2. We decrypt using a single coefficient of the secret inverse polynomial (similarly to Smart-Vercauteren [17]), but for convenience we use modular arithmetic rather than rational division. See Section 6.1.

3. We use a highly optimized algorithm for computing the resultant and *one coefficient of the inverse* of a given polynomial $v(x)$ with respect to $f(x) = x^{2^m} \pm 1$ (without having to compute the entire inverse). This is probably the most algorithmically interesting part of this work. See Section 4.

4. We use batch techniques to speed-up encryption. Specifically, we use an efficient algorithm for batch evaluation of many polynomials with small coefficients on the same point. See Section 5. Our algorithm, when specialized to evaluating a single polynomial, is essentially the same as Avanzi's trick [1], which itself is similar to the algorithm of Paterson and Stockmeyer [12]. The time to evaluate k polynomials is only $O(\sqrt{k})$ more than evaluating a single polynomial.

Fully homomorphic scheme.

5. The secret key in our implementation is a binary vector of length $S \approx 1000$, with only $s = 15$ bits set to one, and the others set to zero. We get significant speedup by representing the secret key in s groups of S bits each, such that each group has a single 1-bit in it. See Section 8.1.

[2] The latter parameter is denoted t in this report. It is the logarithm of the parameter η in [17].

6. The public key of the bootstrappable scheme contains an instance of the sparse-subset-sum problem, and we use instances that have a very space-efficient representation. Specifically, we derive our instances from geometric progressions. See Section 9.1.

7. Similarly, the public key of the fully homomorphic scheme contains an encryption of all the secret-key bits, and we use a space-time tradeoff to optimize the space that it takes to store all these ciphertexts without paying too much in running time. See Section 9.2.

Finally, our choice of parameters is presented in Section 10, and some performance numbers are given in Section 11. Throughout the text we put more emphasis on concrete parameters than on asymptotics, asymptotic bounds can be found in [18].

2 Background

Notations. Throughout this report we use '$\cdot$' to denote scalar multiplication and '$\times$' to denote any other type of multiplication. For integers z, d, we denote the reduction of z modulo d by either $[z]_d$ or $\langle z \rangle_d$. We use $[z]_d$ when the operation maps integers to the interval $[-d/2,\ d/2)$, and use $\langle z \rangle_d$ when the operation maps integers to the interval $[0, d)$. We use the generic "z mod d" when the specific interval does not matter (e.g., mod 2). For example we have $[13]_5 = -2$ vs. $\langle 13 \rangle_5 = 3$, but $[9]_7 = \langle 9 \rangle_7 = 2$.

For a rational number q, we denote by $\lceil q \rfloor$ the rounding of q to the nearest integer, and by $[q]$ we denote the distance between q and the nearest integer. That is, if $q = \frac{a}{b}$ then $[q] \overset{\text{def}}{=} \frac{[a]_b}{b}$ and $\lceil q \rfloor \overset{\text{def}}{=} q - [q]$. For example, $\left\lceil \frac{13}{5} \right\rfloor = 3$ and $[\frac{13}{5}] = \frac{-2}{5}$. These notations are extended to vectors in the natural way: for example if $\vec{q} = \langle q_0, q_1, \ldots, q_{n-1} \rangle$ is a rational vector then rounding is done coordinate-wise, $\lceil \vec{q} \rfloor = \langle \lceil q_0 \rfloor, \lceil q_1 \rfloor, \ldots, \lceil q_{n-1} \rfloor \rangle$.

2.1 Lattices

A full-rank n-dimensional *lattice* is a discrete subgroup of $\mathbb{R}^n$, concretely represented as the set of all integer linear combinations of some *basis* $B = (\vec{b}_1, \ldots, \vec{b}_n) \in \mathbb{R}^n$ of linearly independent vectors. Viewing the vectors $\vec{b}_i$ as the rows of a matrix $B \in \mathbb{R}^{n \times n}$, we have:

$$L = \mathcal{L}(B) = \{\vec{y} \times B \ : \ \vec{y} \in \mathbb{Z}^n\}.$$

Every lattice (of dimension $n > 1$) has an infinite number of lattice bases. If B_1 and B_2 are two lattice bases of L, then there is some unimodular matrix U (i.e., U has integer entries and $\det(U) = \pm 1$) satisfying $B_1 = U \times B_2$. Since U is unimodular, $|\det(B_i)|$ is invariant for different bases of L, and we may refer to it as $\det(L)$. This value is precisely the size of the quotient group $\mathbb{Z}^n/L$ if L is an integer lattice. To basis B of lattice L we associate the half-open parallelepiped $\mathcal{P}(B) \leftarrow \{\sum_{i=1}^n x_i \vec{b}_i : x_i \in [-1/2, 1/2)\}$. The volume of $\mathcal{P}(B)$ is precisely $\det(L)$.

For $\vec{c} \in \mathbb{R}^n$ and basis B of L, we use $\vec{c} \bmod B$ to denote the unique vector $\vec{c'} \in \mathcal{P}(B)$ such that $\vec{c} - \vec{c'} \in L$. Given $\vec{c}$ and B, $\vec{c} \bmod B$ can be computed efficiently as $\vec{c} - \lfloor \vec{c} \times B^{-1} \rceil \times B = [\vec{c} \times B^{-1}] \times B$. (Recall that $\lfloor \cdot \rceil$ means rounding to the nearest integer and $[\cdot]$ is the fractional part.)

Every full-rank lattice has a unique Hermite normal form (HNF) basis where $b_{i,j} = 0$ for all $i < j$ (lower-triangular), $b_{j,j} > 0$ for all j, and for all $i > j$ $b_{i,j} \in [-b_{j,j}/2, +b_{j,j}/2)$. Given any basis B of L, one can compute $\text{HNF}(L)$ efficiently via Gaussian elimination. The HNF is in some sense the "least revealing" basis of L, and thus typically serves as the public key representation of the lattice [10].

Short vectors and Bounded Distance Decoding. The length of the shortest nonzero vector in a lattice L is denoted $\lambda_1(L)$, and Minkowski's theorem says that for any n-dimensional lattice L ($n > 1$) we have $\lambda_1(L) < \sqrt{n} \cdot \det(L)^{1/n}$. Heuristically, for random lattices the quantity $\det(L)^{1/n}$ serves as a threshold: for $t \ll \det(L)^{1/n}$ we don't expect to find any nonzero vectors in L of size t, but for $t \gg \det(L)^{1/n}$ we expect to find exponentially many vectors in L of size t.

In the "*bounded distance decoding*" problem (BDDP), one is given a basis B of some lattice L, and a vector $\vec{c}$ that is very close to some lattice point of L, and the goal is to find the point in L nearest to $\vec{c}$. In the promise problem γ-BDDP, we have a parameter $\gamma > 1$ and the promise that $\mathrm{dist}(L, \vec{c}) \overset{\text{def}}{=} \min_{\vec{v} \in L}\{\|\vec{c} - \vec{v}\|\} \leq \det(L)^{1/n}/\gamma$. (BDDP is often defined with respect to λ_1 rather than with respect to $\det(L)^{1/n}$, but the current definition is more convenient in our case.)

Gama and Nguyen conducted extensive experiments with lattices in dimensions 100-400 [3], and concluded that for those dimensions it is feasible to solve γ-BDDP when $\gamma > 1.01^n \approx 2^{n/70}$. More generally, the best algorithms for solving the γ-BDDP in n-dimensional lattices take time exponential in $n/\log\gamma$. Specifically, currently known algorithms can solve dimension-n γ-BDDP in time 2^k up to $\gamma = 2^{\frac{\mu n}{k/\log k}}$, where μ is a parameter that depends on the exact details of the algorithm. (Extrapolating from the Gama-Nguyen experiments, we expect something like $\mu \in [0.1, 0.2]$.)

2.2 Ideal lattices

Let $f(x)$ be an integer monic irreducible polynomial of degree n. In this paper, we use $f(x) = x^n + 1$, where n is a power of 2. Let R be the ring of integer polynomials modulo $f(x)$, $R \overset{\text{def}}{=} \mathbb{Z}[x]/(f(x))$. Each element of R is a polynomial of degree at most $n-1$, and thus is associated to a coefficient vector in $\mathbb{Z}^n$. This way, we can view each element of R as being both a polynomial and a vector. For $\vec{v}(x)$, we let $\|\vec{v}\|$ be the Euclidean norm of its coefficient vector. For every ring R, there is an associated expansion factor $\gamma_{\mathsf{Mult}}(R)$ such that $\|\vec{u} \times \vec{v}\| \leq \gamma_{\mathsf{Mult}}(R) \cdot \|\vec{u}\| \cdot \|\vec{v}\|$, where × denotes multiplication in the ring. When $f(x) = x^n + 1$, $\gamma_{\mathsf{Mult}}(R)$ is $\sqrt{n}$. However, for "random vectors" $\vec{u}, \vec{v}$ the expansion factor is typically much smaller, and our experiments suggest that we typically have $\|\vec{u} \times \vec{v}\| \approx \|\vec{u}\| \cdot \|\vec{v}\|$.

Let I be an ideal of R – that is, a subset of R that is closed under addition and multiplication by elements of R. Since I is additively closed, the coefficient vectors associated to elements of I form a *lattice*. We call I an *ideal lattice* to emphasize this object's dual nature as an algebraic ideal and a lattice.[3] Ideals have additive structure as lattices, but they also have multiplicative structure. The *product* IJ of two ideals I and J is the additive closure of the set $\{\vec{v} \times \vec{w} : \vec{v} \in I, \vec{w} \in J\}$, where '×' is ring multiplication. To simplify things, we will use *principal* ideals of R – i.e., ideals with a single generator. The ideal $(\vec{v})$ generated by $\vec{v} \in R$ corresponds to the lattice generated by the vectors $\{\vec{v}_i \overset{\text{def}}{=} \vec{v} \times x^i \bmod f(x) : i \in [0, n-1]\}$; we call this the *rotation basis* of the ideal lattice $(\vec{v})$.

Let K be a field containing the ring R (in our case $K = \mathbb{Q}[x]/(f(x))$). The *inverse* of an ideal $I \subseteq R$ is $I^{-1} = \{\vec{w} \in K : \forall \vec{v} \in I, \vec{v} \times \vec{w} \in R\}$. The inverse of a principal ideal $(\vec{v})$ is given by $(\vec{v}^{-1})$, where the inverse $\vec{v}^{-1}$ is taken in the field K.

2.3 GGH-type cryptosystems

We briefly recall Micciancio's "cleaned-up version" of GGH cryptosystems [8, 10]. The secret and public keys are "good" and "bad" bases of some lattice L. More specifically, the key-holder

[3] Alternative representations of an ideal lattice are possible – e.g., see [13, 9].

generates a good basis by choosing B_{sk} to be a basis of short, "nearly orthogonal" vectors. Then it sets the public key to be the Hermite normal form of the same lattice, $B_{pk} \stackrel{\text{def}}{=} \text{HNF}(\mathcal{L}(B_{sk}))$.

A ciphertext in a GGH-type cryptosystem is a vector $\vec{c}$ close to the lattice $\mathcal{L}(B_{pk})$, and the message which is encrypted in this ciphertext is somehow embedded in the distance from $\vec{c}$ to the nearest lattice vector. To encrypt a message m, the sender chooses a short "error vector" $\vec{e}$ that encodes m, and then computes the ciphertext as $\vec{c} \leftarrow \vec{e} \bmod B_{pk}$. Note that if $\vec{e}$ is short enough (i.e., less than $\lambda_1(L)/2$), then it is indeed the distance between $\vec{c}$ and the nearest lattice point.

To decrypt, the key-holder uses its "good" basis B_{sk} to recover $\vec{e}$ by setting $\vec{e} \leftarrow \vec{c} \bmod B_{sk}$, and then recovers m from $\vec{e}$. The reason decryption works is that, if the parameters are chosen correctly, then the parallelepiped $\mathcal{P}(B_{sk})$ of the secret key will be a "plump" parallelepiped that contains a sphere of radius bigger than $\|\vec{e}\|$, so that $\vec{e}$ is the point inside $\mathcal{P}(B_{sk})$ that equals $\vec{c}$ modulo L. On the other hand, the parallelepiped $\mathcal{P}(B_{pk})$ of the public key will be very skewed, and will not contain a sphere of large radius, making it useless for solving BDDP.

2.4 Gentry's somewhat-homomorphic cryptosystem

Gentry's somewhat homomorphic encryption scheme [4] can be seen as a GGH-type scheme over ideal lattices. The public key consists of a "bad" basis B_{pk} of an ideal lattice J, along with some basis B_I of a "small" ideal I (which is used to embed messages into the error vectors). For example, the small ideal I can be taken to be $I = (2)$, the set of vectors with all even coefficients.

A ciphertext in Gentry's scheme is a vector close to a J-point, with the message being embedded in the distance to the nearest lattice point. More specifically, the plaintext space is $\{0, 1\}$, which is embedded in $R/I = \{0, 1\}^n$ by encoding 0 as 0^n and 1 as $0^{n-1}1$. For an encoded bit $\vec{m} \in \{0, 1\}^n$ we set $\vec{e} = 2\vec{r} + \vec{m}$ for a random small vector $\vec{r}$, and then output the ciphertext $\vec{c} \leftarrow \vec{e} \bmod B_{pk}$.

The secret key in Gentry's scheme (that plays the role of the "good basis" of J) is just a short vector $\vec{w} \in J^{-1}$. Decryption involves computing the fractional part $[\vec{w} \times \vec{c}]$. Since $\vec{c} = \vec{j} + \vec{e}$ for some $j \in J$, then $\vec{w} \times \vec{c} = \vec{w} \times \vec{j} + \vec{w} \times \vec{e}$. But $\vec{w} \times \vec{j}$ is in R and thus an integer vector, so $\vec{w} \times \vec{c}$ and $\vec{w} \times \vec{e}$ have the same fractional part, $[\vec{w} \times \vec{c}] = [\vec{w} \times \vec{e}]$. If $\vec{w}$ and $\vec{e}$ are short enough – in particular, if we have the guarantee that all of the coefficients of $\vec{w} \times \vec{e}$ have magnitude less than 1/2 – then $[\vec{w} \times \vec{e}]$ equals $\vec{w} \times \vec{e}$ exactly. From $\vec{w} \times \vec{e}$, the decryptor can multiply by $\vec{w}^{-1}$ to recover $\vec{e}$, and then recover $\vec{m} \leftarrow \vec{e} \bmod 2$. The actual decryption procedure from [4] is slightly different, however. Specifically, $\vec{w}$ is "tweaked" so that decryption can be implemented as $\vec{m} \leftarrow \vec{c} - [\vec{w} \times \vec{c}] \bmod 2$ (when $I = (2)$).

The reason that this scheme is somewhat homomorphic is that for two ciphertexts $\vec{c}_1 = \vec{j}_1 + \vec{e}_1$ and $\vec{c}_2 = \vec{j}_2 + \vec{e}_2$, their sum is $\vec{j}_3 + \vec{e}_3$ where $\vec{j}_3 = \vec{j}_1 + \vec{j}_2 \in J$ and $\vec{e}_3 = \vec{e}_1 + \vec{e}_2$ is small. Similarly, their product is $\vec{j}_4 + \vec{e}_4$ where $\vec{j}_4 = \vec{j}_1 \times (\vec{j}_2 + \vec{e}_2) + \vec{e}_1 \times \vec{j}_2 \in J$ and $\vec{e}_4 = \vec{e}_1 \times \vec{e}_2$ is still small. If fresh encrypted ciphertexts are very very close to the lattice, then it is possible to add and multiply ciphertexts for a while before the error grows beyond the decryption radius of the secret key.

2.4.1 The Smart-Vercauteren Variant

Smart and Vercauteren [17] work over the ring $R = \mathbb{Z}[x]/f_n(x)$, where $f_n(x) = x^n + 1$ and n is a power of two. The ideal J is set as a principal ideal by choosing a vector $\vec{v}$ at random from some n-dimensional cube, subject to the condition that the determinant of $(\vec{v})$ is prime, and then setting $J = (\vec{v})$. It is known that such ideals can be implicitly represented by only two integers, namely the determinant $d = \det(J)$ and a root r of $f_n(x)$ modulo d. (An easy proof of this fact "from first principles" can be derived from our Lemma 1 below.) Specifically, the Hermite normal form of this ideal lattice is

$$\mathrm{HNF}(J) = \begin{bmatrix} d & 0 & 0 & 0 & 0 \\ -r & 1 & 0 & 0 & 0 \\ -[r^2]_d & 0 & 1 & 0 & 0 \\ -[r^3]_d & 0 & 0 & 1 & 0 \\ & & & \ddots & \\ -[r^{n-1}]_d & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

It is easy to see that reducing a vector $\vec{a}$ modulo HNF(J) consists of evaluating the associated polynomial $a(x)$ at the point r modulo d, then outputting the vector $\langle [a(r)]_d, 0, 0, \dots, 0\rangle$ (see Section 5). Hence encryption of a vector $\langle m, 0, 0, \dots, 0\rangle$ with $m \in \{0,1\}$ can be done by choosing a random small polynomial $u(x)$ and evaluating it at r, then outputting the integer $c \leftarrow [2u(r) + m]_d$.

Smart and Vercauteren also describe a decryption procedure that uses a single integer w as the secret key, setting $m \leftarrow (c - \lceil cw/d \rfloor) \bmod 2$. Jumping ahead, we note that our decryption procedure from Section 6 is very similar, except that for convenience we replace the rational division cw/d by modular multiplication $[cw]_d$.

2.5 Gentry's fully-homomorphic scheme

As explained above, Gentry's somewhat-homomorphic scheme can evaluate low-degree polynomials but not more. Once the degree (or the number of terms) is too large, the error vector $\vec{e}$ grows beyond the decryption capability of the private key. Gentry solved this problem using bootstrapping. He observed in [4] that a scheme that can homomorphically evaluate its own decryption circuit plus one additional operation, can be transformed into a fully-homomorphic encryption. In more details, fix two ciphertexts $\vec{c}_1, \vec{c}_2$ and consider the functions

$$\mathsf{DAdd}_{\vec{c}_1,\vec{c}_2}(sk) \overset{\text{def}}{=} \mathsf{Dec}_{sk}(\vec{c}_1) + \mathsf{Dec}_{sk}(\vec{c}_2) \quad \text{and} \quad \mathsf{DMul}_{\vec{c}_1,\vec{c}_2}(sk) \overset{\text{def}}{=} \mathsf{Dec}_{sk}(\vec{c}_1) \times \mathsf{Dec}_{sk}(\vec{c}_2).$$

A somewhat-homomorphic scheme is called "*bootstrappable*" if it is capable of homomorphically evaluating the functions $\mathsf{DAdd}_{\vec{c}_1,\vec{c}_2}$ and $\mathsf{DMul}_{\vec{c}_1,\vec{c}_2}$ for any two ciphertexts $\vec{c}_1, \vec{c}_2$. Given a bootstrappable scheme that is also circular secure, it can be transformed into a fully-homomorphic scheme by adding to the public key an encryption of the secret key, $\vec{c^*} \leftarrow \mathsf{Enc}_{pk}(sk)$. Then given any two ciphertexts $\vec{c_1}$, $\vec{c_2}$, the addition/multiplication of these two ciphertexts can be computed by homomorphically evaluating the functions $\mathsf{DAdd}_{\vec{c}_1,\vec{c}_2}(\vec{c^*})$ or $\mathsf{DMul}_{\vec{c}_1,\vec{c}_2}(\vec{c^*})$. Note that the error does not grow, since we always evaluate these functions on the fresh ciphertext $\vec{c^*}$ from the public key.

Unfortunately, the somewhat-homomorphic scheme from above is not bootstrappable. Although it is capable of evaluating low-degree polynomials, the degree of its decryption function, when expressed as a polynomial in the secret key bits, is too high. To overcome this problem Gentry shows how to "squash the decryption circuit", transforming the original somewhat-homomorphic scheme E into a scheme E^* that can correctly evaluate any circuit that E can, but where the complexity of E^*'s decryption circuit is much less than E's. In the original somewhat-homomorphic scheme E, the secret key is a vector $\vec{w}$. In the new scheme E^*, the public key includes an additional "hint" about $\vec{w}$ – namely, a big set of vectors $\mathcal{S} = \{\vec{x}_i : i = 1, 2, \dots, S\}$ that have a hidden sparse subset T that adds up to $\vec{w}$. The secret key of E^* is the characteristic vector of the sparse subset T, which is denoted $\vec{\sigma} = \langle \sigma_1, \sigma_2, \dots, \sigma_S \rangle$.

Whereas decryption in the original scheme involved computing $\vec{m} \leftarrow \vec{c} - [\vec{w} \times \vec{c}] \bmod 2$, in the new scheme the ciphertext $\vec{c}$ is "post-processed" by computing the products $\vec{y}_i = \vec{x}_i \times \vec{c}$ for all of the vectors $\vec{x}_i \in \mathcal{S}$. Obviously, then, the decryption in the new scheme can be done by computing $\vec{c} - [\sum_j \sigma_j \vec{y}_j] \bmod 2$. Using some additional tricks, this computation can be expressed as a polynomial in the σ_i's of degree roughly the size of the sparse subset T. (The underlying algorithm could be a simple grade-school addition – add up the least significant column, bring a carry bit over to the next column if necessary, and so on.) With appropriate setting of the parameters, the subset T can be made small enough to get a bootstrappable scheme.

Part I
The "Somewhat Homomorphic" Scheme

3 Key generation

We adopt the Smart-Vercauteren approach [17], in that we also use principal-ideal lattices in the ring of polynomials modulo $f_n(x) \stackrel{\text{def}}{=} x^n + 1$ with n a power of two. We do not require that these principal-ideal lattices have prime determinant, instead we only need the Hermite normal form to have the same form as in Equation (1). During key-generation we choose $\vec{v}$ at random in some cube, verify that the HNF has the right form, and work with the principal ideal $(\vec{v})$. We have two parameters: the dimension n, which must be a power of two, and the bit-size t of coefficients in the generating polynomial. Key-generation consists of the following steps:

1. Choose a random n-dimensional integer lattice $\vec{v}$, where each entry v_i is chosen at random as a t-bit (signed) integer. With this vector $\vec{v}$ we associate the formal polynomial $v(x) \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} v_i x^i$, as well as the rotation basis:

$$V = \begin{bmatrix} v_0 & v_1 & v_2 & & v_{n-1} \\ -v_{n-1} & v_0 & v_1 & & v_{n-2} \\ -v_{n-2} & -v_{n-1} & v_0 & & v_{n-3} \\ & & & \ddots & \\ -v_1 & -v_2 & -v_3 & & v_0 \end{bmatrix} \qquad (2)$$

The i'th row is a cyclic shift of $\vec{v}$ by i positions to the right, with the "overflow entries" negated. Note that the i'th row corresponds to the coefficients of the polynomial $v_i(x) = v(x) \times x^i \pmod{f_n(x)}$. Note that just like V itself, the entire lattice $\mathcal{L}(V)$ is also closed under "rotation": Namely, for any vector $\langle u_0, u_1, \ldots, u_{n-1}\rangle \in \mathcal{L}(V)$, also the vector $\langle -u_{n-1}, u_0, \ldots, u_{n-2}\rangle$ is in $\mathcal{L}(V)$.

2. Next we compute the scaled inverse of $v(x)$ modulo $f_n(x)$, namely an integer polynomial $w(x)$ of degree at most $n-1$, such that

$$w(x) \times v(x) = \mathsf{constant} \pmod{f_n(x)}.$$

Specifically, this constant is the determinant of the lattice $\mathcal{L}(V)$, which must be equal to the resultant of the polynomials $v(x)$ and $f_n(x)$ (since f_n is monic). Below we denote the resultant by d, and denote the coefficient-vector of $w(x)$ by $\vec{w} = \langle w_0, w_1, \ldots, w_{n-1}\rangle$. It is easy to check that the matrix

$$W = \begin{bmatrix} w_0 & w_1 & w_2 & & w_{n-1} \\ -w_{n-1} & w_0 & w_1 & & w_{n-2} \\ -w_{n-2} & -w_{n-1} & w_0 & & w_{n-3} \\ & & & \ddots & \\ -w_1 & -w_2 & -w_3 & & w_0 \end{bmatrix} \qquad (3)$$

is the scaled inverse of V, namely $W \times V = V \times W = d \cdot I$. One way to compute the polynomial $w(x)$ is by applying the extended Euclidean-GCD algorithm (for polynomials) to $v(x)$ and $f_n(x)$. See Section 4 for a more efficient method of computing $w(x)$.

3. We also check that this is a good generating polynomial. We consider $\vec{v}$ to be good if the Hermite-Normal-form of V has the same form as in Equation (1), namely all except the leftmost column equal to the identity matrix. See below for a simple check that the $\vec{v}$ is good, in our implementation we test this condition while computing the inverse.

   It was observed by Nigel Smart that the HNF has the correct form whenever the determinant is odd and square-free. Indeed, in our tests this condition was met with probability roughly 0.5, irrespective of the dimension and bit length, with the failure cases usually due to the determinant of V being even.

Checking the HNF. In Lemma 1 below we prove that the HNF of the lattice $\mathcal{L}(V)$ has the right form if and only if the lattice contains a vector of the form $\langle -r, 1, 0, \ldots, 0\rangle$. Namely, if and only if there exists an integer vector $\vec{y}$ and another integer r such that

$$\vec{y} \times V = \langle -r, 1, 0, \ldots, 0\rangle$$

Multiplying the last equation on the right by W, we get the equivalent condition

$$\begin{aligned} \vec{y} \times V \times W &= \langle -r, 1, 0 \ldots, 0\rangle \times W \\ \Leftrightarrow \quad \vec{y} \times (dI) \;=\; d \cdot \vec{y} &= -r \cdot \langle w_0, w_1, w_2, \ldots, w_{n-1}\rangle + \langle -w_{n-1}, w_0, w_1, \ldots, w_{n-2}\rangle \end{aligned} \tag{4}$$

In other words, there must exists an integer r such that the second row of W minus r times the first row yields a vector of integers that are all divisible by d:

$$\begin{aligned} -r \cdot \langle w_0, w_1, w_2, \ldots, w_{n-1}\rangle + \langle -w_{n-1}, w_0, w_1, \ldots, w_{n-2}\rangle &= 0 \pmod{d} \\ \Leftrightarrow \quad -r \cdot \langle w_0, w_1, w_2, \ldots, w_{n-1}\rangle &= \langle w_{n-1}, -w_0, -w_1, \ldots, -w_{n-2}\rangle \pmod{d} \end{aligned}$$

The last condition can be checked easily: We compute $r := w_0/w_1 \bmod d$ (assuming that w_1 has an inverse modulo d), then check that $r \cdot w_{i+1} = w_i \pmod{d}$ holds for all $i = 1, \ldots, n-2$ and also $-r \cdot w_0 = w_{n-1} \pmod{d}$. Note that this means in particular that $r^n = -1 \pmod{d}$. (In our implementation we actually test only that last condition, instead of testing all the equalities $r \cdot w_{i+1} = w_i \pmod{d}$.)

Lemma 1. *The Hermite normal form of the matrix V from Equation (2) is equal to the identity matrix in all but the leftmost column, if and only if the lattice spanned by the rows of V contains a vector of the form $\vec{r} = \langle -r, 1, 0 \ldots, 0\rangle$.*

Proof. Let B be the Hermite normal form of V. Namely, B is lower triangular matrix with non-negative diagonal entries, where the rows of B span the same lattice as the rows of V, and the absolute value of every entry under the diagonal in B is no more than half the diagonal entry above it. This matrix B can be obtained from V by a sequence of elementary row operations, and it is unique. It is easy to see that the existence of a vector $\vec{r}$ of this form is necessary: indeed the second row of B must be of this form (since B is equal the identity in all except the leftmost column). We now prove that this condition is also sufficient.

It is clear that the vector $d \cdot \vec{e_1} = \langle d, 0, \ldots, 0\rangle$ belongs to $\mathcal{L}(V)$: in particular we know that $\langle w_0, w_1, \ldots, w_{n-1}\rangle \times V = \langle d, 0, \ldots, 0\rangle$. Also, by assumption we have $\vec{r} = -r \cdot \vec{e_1} + \vec{e_2} \in \mathcal{L}(V)$, for some integer r. Note that we can assume without loss of generality that $-d/2 \le r < d/2$, since otherwise we could subtract from $\vec{r}$ multiples of the vector $d \cdot \vec{e_1}$ until this condition is satisfied:

$$\begin{array}{rl} & \langle\, -r \quad 1\ 0\ \ldots\ 0 \rangle \\ -\kappa\cdot & \langle\ \ d \quad\ 0\ 0\ \ldots\ 0 \rangle \\ \hline = & \langle [-r]_d\ 1\ 0\ \ldots\ 0 \rangle \end{array}$$

For $i = 1, 2, \ldots, n-1$, denote $r_i \overset{\text{def}}{=} [r^i]_d$. Below we will prove by induction that for all $i = 1, 2, \ldots, n-1$, the lattice $\mathcal{L}(V)$ contains the vector:

$$\vec{r_i} \overset{\text{def}}{=} -r_i \cdot \vec{e_1} + \vec{e}_{i+1} = \underbrace{\langle -r_i, 0 \ldots 0, 1, 0 \ldots 0 \rangle}_{1 \text{ in the } i{+}1'\text{st position}}.$$

Placing all these vectors $\vec{r_i}$ at the rows of a matrix, we got exactly the matrix B that we need:

$$B = \begin{bmatrix} d & 0 & 0 & 0 \\ -r_1 & 1 & 0 & 0 \\ -r_2 & 0 & 1 & 0 \\ & & \ddots & \\ -r_{n-1} & 0 & 0 & 1 \end{bmatrix}. \tag{5}$$

B is equal to the identity except in the leftmost column, its rows are all vectors in $\mathcal{L}(V)$ (so they span a sub-lattice), and since B has the same determinant as V then it cannot span a proper sub-lattice, it must therefore span $\mathcal{L}(V)$ itself.

It is left to prove the inductive claim. For $i = 1$ we set $\vec{r_1} \overset{\text{def}}{=} \vec{r}$ and the claim follow from our assumption that $\vec{r} \in \mathcal{L}(V)$. Assume now that it holds for some $i \in [1, n-2]$ and we prove for $i+1$. Recall that the lattice $\mathcal{L}(V)$ is closed under rotation, and since $\vec{r_i} = -r_i\vec{e_1} + \vec{e}_{i+1} \in \mathcal{L}(V)$ then the right-shifted vector $\vec{s}_{i+1} \overset{\text{def}}{=} -r_i\vec{e_2} + \vec{e}_{i+2}$ is also in $\mathcal{L}(V)$.[4] Hence $\mathcal{L}(V)$ contains also the vector

$$\vec{s}_{i+1} + r_i \cdot \vec{r} = (-r_i\vec{e_2} + e_{i\vec{+}2}) + r_i(-r\vec{e_1} + \vec{e_2}) = \ = -r_i r \cdot \vec{e_1} + \vec{e}_{i+2}$$

We can now reduce the first entry in this vector modulo d, by adding/subtracting the appropriate multiple of $d \cdot \vec{e_1}$ (while still keeping it in the lattice), thus getting the lattice vector

$$[-r \cdot r_i]_d \cdot \vec{e_1} + e_{i\vec{+}2} = -[r^{i+1}]_d \cdot \vec{e_1} + e_{i\vec{+}2} = \vec{r}_{i+1} \in \mathcal{L}(V)$$

This concludes the proof. □

Remark 1. *Note that the proof of Lemma 1 shows in particular that if the Hermite normal form of V is equal to the identity matrix in all but the leftmost column, then it must be of the form specified in Equation (5). Namely, the first column is $\langle d, -r_1, -r_2, \ldots, -r_{n-1} \rangle^t$, with $r_i = [r^i]_d$ for all i. Hence this matrix can be represented implicitly by the two integers d and r.*

3.1 The public and secret keys

In principle the public key is the Hermite normal form of V, but as we explain in Remark 1 and Section 5 it is enough to store for the public key only the two integers d, r. Similarly, in principle the secret key is the pair $(\vec{v}, \vec{w})$, but as we explain in Section 6.1 it is sufficient to store only a single (odd) coefficient of $\vec{w}$ and discard $\vec{v}$ altogether.

[4] This is a circular shift, since $i \le n-2$ and hence the rightmost entry in $\vec{r_i}$ is zero.

4 Inverting the polynomial $v(x)$

The fastest known methods for inverting the polynomial $v(x)$ modulo $f_n(x) = x^n + 1$ are based on FFT: We can evaluate $v(x)$ at all the roots of $f_n(x)$ (either over the complex field or over some finite field), then compute $w^*(\rho) = 1/v(\rho)$ (where inversion is done over the corresponding field), and then interpolate $w^* = v^{-1}$ from all these values. If the resultant of v and f_n has N bits, then this procedure will take $O(n \log n)$ operations over $O(N)$-bit numbers, for a total running time of $\tilde{O}(nN)$. This is close to optimal in general, since just writing out the coefficients of the polynomial w^* takes time $O(nN)$. However, in Section 6.1 we show that it is enough to use for the secret key only one of the coefficients of $w = d \cdot w^*$ (where $d = \mathsf{resultant}(v, f_n)$). This raises the possibility that we can compute this one coefficient in time quasi-linear in N rather than quasi-linear in nN. Although polynomial inversion is very well researched, as far as we know this question of computing just one coefficient of the inverse was not tackled before. Below we describe an algorithm for doing just that.

The approach for the procedure below is to begin with the polynomial v that has n small coefficients, and proceed in steps where in each step we halve the number of coefficients to offset the fact that the bit-length of the coefficients approximately doubles. Our method relies heavily on the special form of $f_n(x) = x^n + 1$, with n a power of two. Let $\rho_0, \rho_1, \ldots, \rho_{n-1}$ be roots of $f_n(x)$ over the complex field: That is, if ρ is some primitive $2n$'th root of unity then $\rho_i = \rho^{2i+1}$. Note that the roots r_i satisfy that $\rho_{i+\frac{n}{2}} = -\rho_i$ for all i, and more generally for every index i (with index arithmetic modulo n) and every $j = 0, 1, \ldots, \log n$, if we denote $n_j \stackrel{\text{def}}{=} n/2^j$ then it holds that

$$\left(\rho_{i+n_j/2}\right)^{2^j} \;=\; \left(\rho^{2i+n_j+1}\right)^{2^j} \;=\; \left(\rho^{2i+1}\right)^{2^j} \cdot \rho^n \;=\; -\left(\rho_i^{2^j}\right) \tag{6}$$

The method below takes advantage of Equation (6), as well as a connection between the coefficients of the scaled inverse w and those of the formal polynomial

$$g(z) \;\stackrel{\text{def}}{=}\; \prod_{i=0}^{n-1} \big(v(\rho_i) - z\big).$$

We invert $v(x) \bmod f_n(x)$ by computing the lower two coefficients of $g(z)$, then using them to recover both the resultant and (one coefficient of) the polynomial $w(x)$, as described next.

Step one: the polynomial $g(z)$. Note that although the polynomial $g(z)$ is defined via the complex numbers ρ_i, the coefficients of $g(z)$ are all integers. We begin by showing how to compute the lower two coefficients of $g(z)$, namely the polynomial $g(z) \bmod z^2$. We observe that since $\rho_{i+\frac{n}{2}} = -\rho_i$ then we can write $g(z)$ as

$$\begin{aligned} g(z) \;&=\; \prod_{i=0}^{\frac{n}{2}-1} (v(\rho_i) - z)(v(-\rho_i) - z) \\ &=\; \prod_{i=0}^{\frac{n}{2}-1} \Big(\underbrace{v(\rho_i)v(-\rho_i)}_{a(\rho_i)} - z(\underbrace{v(\rho_i) + v(-\rho_i)}_{b(\rho_i)}) + z^2 \Big) \;=\; \prod_{i=0}^{\frac{n}{2}-1} \Big(a(\rho_i) - zb(\rho_i) \Big) \pmod{z^2} \end{aligned}$$

We observe further that for both the polynomials $a(x) = v(x)v(-x)$ and $b(x) = v(x) + v(-x)$, all the odd powers of x have zero coefficients. Moreover, the same equalities as above hold if we use $A(x) = a(x) \bmod f_n(x)$ and $B(x) = b(x) \bmod f_n(x)$ instead of $a(x)$ and $b(x)$ themselves (since we only evaluate these polynomials in roots of f_n), and also for A, B all the odd powers of x have zero coefficients (since we reduce modulo $f_n(x) = x^n + 1$ with n even).

Thus we can consider the polynomials $\hat{v}, \tilde{v}$ that have half the degree and only use the nonzero coefficients of A, B, respectively. Namely they are defined via $\hat{v}(x^2) = A(x)$ and $\tilde{v}(x^2) = B(x)$. Thus we have reduced the task of computing the n-product involving the degree-n polynomial $v(x)$ to computing a product of only $n/2$ terms involving the degree-$n/2$ polynomials $\hat{v}(x), \tilde{v}(x)$. Repeating this process recursively, we obtain the polynomial $g(z) \bmod z^2$. The details of this process are described in Section 4.1 below.

Step two: recovering d and w_0. Recall that if $v(x)$ is square free then $d = \mathsf{resultant}(v, f_n) = \prod_{i=0}^{n-1} v(\rho_i)$, which is exactly the free term of $g(z)$, $g_0 = \prod_{i=0}^{n-1} v(\rho_i)$.

Recall also that the linear term in $g(z)$ has coefficient $g_1 = \sum_{i=0}^{n-1} \prod_{j \neq i} v(\rho_i)$. We next show that the free term of $w(x)$ is $w_0 = g_1/n$. First, we observe that g_1 equals the sum of w evaluated in all the roots of f_n, namely

$$g_1 \;=\; \sum_{i=0}^{n-1} \prod_{j \neq i} v(\rho_j) \;=\; \sum_{i=0}^{n-1} \frac{\prod_{j=0}^{n-1} v(\rho_j)}{v(\rho_i)} \;\overset{(a)}{=}\; \sum_{i=0}^{n-1} \frac{d}{v(\rho_i)} \;\overset{(b)}{=}\; \sum_{i=0}^{n-1} w(\rho_i)$$

where Equality (a) follows since $v(x)$ is square free and $d = \mathsf{resultant}(v, f_n)$, and Equality (b) follows since $v(\rho_i) = d/w(\rho_i)$ holds in all the roots of f_n. It is left to show that the constant term of $w(x)$ is $w_0 = n \sum_{i=0}^{n-1} w(\rho_i)$. To show this, we write

$$\sum_{i=0}^{n-1} w(\rho_i) \;=\; \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} w_j \rho_i^j \;=\; \sum_{j=0}^{n-1} w_j \sum_{i=0}^{n-1} \rho_i^j \;\overset{(\star)}{=}\; \sum_{j=0}^{n-1} w_j \sum_{i=0}^{n-1} (\rho^j)^{2i+1} \tag{7}$$

where the Equality $(\star)$ holds since the i'th root of f_n is $\rho_i = \rho^{2i+1}$ where ρ is a $2n$-th root of unity. Clearly, the term corresponding to $j = 0$ in Equation (7) is $w_0 \cdot n$, it is left to show that all the other terms are zero. This follows since ρ^j is a $2n$-th root of unity different from ± 1 for all $j = 1, 2, \ldots, n-1$, and summing over all odd powers of such root of unity yields zero.

Step three: recovering the rest of w. We can now use the same technique to recover all the other coefficients of w: Note that since we work modulo $f_n(x) = x^n + 1$, then the coefficient w_i is the free term of the scaled inverse of $x^i \times v \pmod{f_n}$.

In our case we only need to recover the first two coefficients, however, since we are only interested in the case where $w_1/w_0 = w_2/w_1 = \cdots = w_{n-1}/w_{n-2} = -w_0/w_{n-1} \pmod{d}$, where $d = \mathsf{resultant}(v, f_n)$. After recovering w_0, w_1 and $d = \mathsf{resultant}(v, f_n)$, we therefore compute the ratio $r = w_1/w_0 \bmod d$ and verify that $r^n = -1 \pmod{d}$. Then we recover as many coefficients of w as we need (via $w_{i+1} = [w_i \cdot r]_d$), until we find one coefficient which is an odd integer, and that coefficient is the secret key.

4.1 The gory details of step one

We denote $U_0(x) \equiv 1$ and $V_0(x) = v(x)$, and for $j = 0, 1, \ldots, \log n$ we denote $n_j = n/2^j$. We proceed in $m = \log n$ steps to compute the polynomials $U_j(x), V_j(x)$ ($j = 1, 2, \ldots, m$), such that the degrees of U_j, V_j are at most $n_j - 1$, and moreover the polynomial $g_j(z) = \prod_{i=0}^{n_j-1} (V_j(\rho_i^{2^j}) - z U_j(\rho_i^{2^j}))$ has the same first two coefficients as $g(z)$. Namely,

$$g_j(z) \;\overset{\text{def}}{=}\; \prod_{i=0}^{n_j-1} \Big(V_j(\rho_i^{2^j}) - z U_j(\rho_i^{2^j}) \Big) \;=\; g(z) \pmod{z^2}. \tag{8}$$

Equation (8) holds for $j = 0$ by definition. Assume that we computed U_j, V_j for some $j < m$ such that Equation (8) holds, and we show how to compute U_{j+1} and V_{j+1}. From Equation (6) we know that $\left(\rho_{i+n_j/2}\right)^{2^j} = -\rho_i^{2^j}$, so we can express g_j as

$$\begin{aligned} g_j(z) &= \prod_{i=0}^{n_j/2-1} \left(V_j(\rho_i^{2^j}) - zU_j(\rho_i^{2^j})\right)\left(V_j(-\rho_i^{2^j}) - zU_j(-\rho_i^{2^j})\right) \\ &= \prod_{i=0}^{n_j/2-1} \Big(\underbrace{V_j(\rho_i^{2^j})V_j(-\rho_i^{2^j})}_{=A_j(\rho_i^{2^j})} - z\big(\underbrace{U_j(\rho_i^{2^j})V_j(-\rho_i^{2^j}) + U_j(-\rho_i^{2^j})V_j(\rho_i^{2^j})}_{=B_j(\rho_i^{2^j})}\big)\Big) \pmod{z^2} \end{aligned}$$

Denoting $f_{n_j}(x) \stackrel{\text{def}}{=} x^{n_j} + 1$ and observing that $\rho_i^{2^j}$ is a root of f_{n_j} for all i, we next consider the polynomials:

$$\begin{aligned} A_j(x) &\stackrel{\text{def}}{=} V_j(x)V_j(-x) \bmod f_{n_j}(x) \quad \text{(with coefficients } a_0, \ldots, a_{n_j-1}) \\ B_j(x) &\stackrel{\text{def}}{=} U_j(x)V_j(-x) + U_j(-x)V_j(x) \bmod f_{n_j}(x) \quad \text{(with coefficients } b_0, \ldots, b_{n_j-1}) \end{aligned}$$

and observe the following:

- Since $\rho_i^{2^j}$ is a root of f_{n_j}, then the reduction modulo f_{n_j} makes no difference when evaluating A_j, B_j on $\rho_i^{2^j}$. Namely we have $A_j(\rho_i^{2^j}) = V_j(\rho_i^{2^j})V_j(-\rho_i^{2^j})$ and similarly $B_j(\rho_i^{2^j}) = U_j(\rho_i^{2^j})V_j(-\rho_i^{2^j}) + U_j(-\rho_i^{2^j})V_j(\rho_i^{2^j})$ (for all i).
- The odd coefficients of A_j, B_j are all zero. For A_j this is because it is obtained as $V_j(x)V_j(-x)$ and for B_j this is because it is obtained as $R_j(x) + R_j(-x)$ (with $R_j(x) = U_j(x)V_j(-x)$). The reduction modulo $f_{n_j}(x) = x^{n_j} + 1$ keeps the odd coefficients all zero, because n_j is even.

We therefore set

$$U_{j+1}(x) \stackrel{\text{def}}{=} \sum_{t=0}^{n_j/2-1} b_{2t} \cdot x^t, \quad \text{and} \quad V_{j+1}(x) \stackrel{\text{def}}{=} \sum_{t=0}^{n_j/2-1} a_{2t} \cdot x^t,$$

so the second bullet above implies that $U_{j+1}(x^2) = B_j(x)$ and $V_{j+1}(x^2) = A_j(x)$ for all x. Combined with the first bullet, we have that

$$\begin{aligned} g_{j+1}(z) &\stackrel{\text{def}}{=} \prod_{i=0}^{n_j/2-1} \left(V_{j+1}(\rho_i^{2^{j+1}}) - z \cdot U_{j+1}(\rho_i^{2^{j+1}})\right) \\ &= \prod_{i=0}^{n_j/2-1} \left(A_j(\rho_i^{2^j}) - z \cdot B_j(\rho_i^{2^j})\right) \;=\; g_j(z) \pmod{z^2}. \end{aligned}$$

By the induction hypothesis we also have $g_j(z) = g(z) \pmod{z^2}$, so we get $g_{j+1}(z) = g(z) \pmod{z^2}$, as needed.

5 Encryption

To encrypt a bit $b \in \{0, 1\}$ with the public key B (which is implicitly represented by the two integers d, r), we first choose a random $0, \pm 1$ "noise vector" $\vec{u} \stackrel{\text{def}}{=} \langle u_0, u_1, \ldots, u_{n-1} \rangle$, with each entry chosen as 0 with some probability q and as ± 1 with probability $(1-q)/2$ each. We then set $\vec{a} \overset{\mathrm{def}}{=} 2\vec{u} + b \cdot \vec{e}_1 = \langle 2u_0 + b, 2u_1, \ldots, 2u_{n-1} \rangle$, and the ciphertext is the vector

$$\vec{c} \;=\; \vec{a} \bmod B \;=\; \vec{a} - \left(\lceil \vec{a} \times B^{-1} \rfloor \times B \right) \;=\; \underbrace{[\vec{a} \times B^{-1}]}_{[\cdot] \text{ is fractional part}} \times B$$

We now show that also $\vec{c}$ can be represented implicitly by just one integer. Recall that B (and therefore also B^{-1}) are of a special form

$$B = \begin{bmatrix} d & 0 & 0 & 0 & & 0 \\ -r & 1 & 0 & 0 & & 0 \\ -[r^2]_d & 0 & 1 & 0 & & 0 \\ -[r^3]_d & 0 & 0 & 1 & & 0 \\ & & & & \ddots & \\ -[r^{n-1}]_d & 0 & 0 & 0 & & 1 \end{bmatrix}, \text{ and } B^{-1} = \frac{1}{d} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & & 0 \\ r & d & 0 & 0 & & 0 \\ [r^2]_d & 0 & d & 0 & & 0 \\ [r^3]_d & 0 & 0 & d & & 0 \\ & & & & \ddots & \\ [r^{n-1}]_d & 0 & 0 & 0 & & d \end{bmatrix}.$$

Denote $\vec{a} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, and also denote by $a(\cdot)$ the integer polynomial $a(x) \overset{\mathrm{def}}{=} \sum_{i=0}^{n-1} a_i x^i$. Then we have $\vec{a} \times B^{-1} = \left\langle \frac{s}{d},\ a_1,\ \ldots,\ a_{n-1} \right\rangle$ for some integer s that satisfies $s = a(r) \pmod d$. Hence the fractional part of $\vec{a} \times B^{-1}$ is $\left[\vec{a} \times B^{-1}\right] = \left\langle \frac{[a(r)]_d}{d},\ 0,\ \ldots,\ 0 \right\rangle$, and the ciphertext vector is $\vec{c} = \left\langle \frac{[a(r)]_d}{d},\ 0,\ \ldots,\ 0 \right\rangle \times B = \langle [a(r)]_d,\ 0,\ \ldots,\ 0 \rangle$. Clearly, this vector can be represented implicitly by the integer $c \overset{\mathrm{def}}{=} \left[a(r)\right]_d = \left[b + 2\sum_{i=1}^{n-1} u_i r^i\right]_d$. Hence, to encrypt the bit b, we only need to evaluate the noise-polynomial $u(\cdot)$ at the point r, then multiply by two and add the bit b (everything modulo d). We now describe an efficient procedure for doing that.

5.1 An efficient encryption procedure

The most expensive operation during encryption is evaluating the degree-$(n-1)$ polynomial u at the point r. Polynomial evaluation using Horner's rule takes $n-1$ multiplications, but it is known that for small coefficients we can reduce the number of multiplications to only $O(\sqrt{n})$, see [1, 12]. Moreover, we observe that it is possible to batch this fast evaluation algorithm, and evaluate k such polynomials in time $O(\sqrt{kn})$.

We begin by noting that evaluating many $0, \pm 1$ polynomials at the same point x can be done about as fast as a naive evaluation of a single polynomial. Indeed, once we compute all the powers $(1, x, x^2, \ldots, x^{n-1})$ then we can evaluate each polynomial just by taking a subset-sum of these powers. As addition is much faster than multiplication, the dominant term in the running time will be the computation of the powers of x, which we only need to do once for all the polynomials.

Next, we observe that evaluating a single degree-$(n-1)$ polynomial at a point x can be done quickly given a subroutine that evaluates two degree-$(n/2-1)$ polynomials at the same point x. Namely, given $u(x) = \sum_{i=0}^{n-1} u_i x^i$, we split it into a "bottom half" $u^{\mathsf{bot}}(x) = \sum_{i=0}^{n/2-1} u_i x^i$ and a "top half" $u^{\mathsf{top}}(x) = \sum_{i=0}^{n/2-1} u_{i+d/2} x^i$. Evaluating these two smaller polynomials we get $y^{\mathsf{bot}} = u^{\mathsf{bot}}(x)$ and $y^{\mathsf{top}} = u^{\mathsf{top}}(x)$, and then we can compute $y = u(x)$ by setting $y = x^{n/2} y^{\mathsf{top}} + y^{\mathsf{bot}}$. If the subroutine for evaluating the two smaller polynomials also returns the value of $x^{n/2}$, then we need just one more multiplication to get the value of $y = u(x)$.

These two observations suggest a recursive approach to evaluating the $0, \pm 1$ polynomial u of degree $n-1$. Namely, we repeatedly cut the degree in half at the price of doubling the number of polynomials, and once the degree is small enough we use the "trivial implementation" of just computing all the powers of x. Analyzing this approach, let us denote by $M(k,n)$ the number of multiplications that it takes to evaluate k polynomials of degree $(n-1)$. Then we have $M(k,n) \le \min(n-1,\ M(2k,n/2)+k+1)$. To see the bound $M(k,n) \le M(2k,n/2)+k+1$, note that once we evaluated the top- and bottom-halves of all the k polynomials, we need one multiplication per polynomial to put the two halves together, and one last multiplication to compute x^n (which is needed in the next level of the recursion) from $x^{n/2}$ (which was computed in the previous level). Obviously, making the recursive call takes less multiplications than the "trivial implementation" whenever $n-1 > (n/2-1)+k+1$. Also, an easy inductive argument shows that the "trivial implementation" is better when $n-1 < (n/2-1)+k+1$. We thus get the recursive formula

$$M(k,n) \; = \; \begin{cases} M(2k,n/2)+k+1 & \text{when } n/2 > k+1 \\ n-1 & \text{otherwise.} \end{cases}$$

Solving this formula we get $M(k,n) \le \min(n-1, \sqrt{2kn})$. In particular, the number of multiplications needed for evaluating a single degree-$(n-1)$ polynomial is $M(1,n) \le \sqrt{2n}$.

We comment that this "more efficient" batch procedure relies on the assumption that we have enough memory to keep all these partially evaluated polynomials at the same time. In our experiments we were only able to use it in dimensions up to $n = 2^{15}$, trying to use it in higher dimension resulted in the process being killed after it ran out of memory. A more sophisticated implementation could take the available amount of memory into account, and stop the recursion earlier to preserve space at the expense of more running time. An alternative approach, of course, is to store partial results to disk. More experiments are needed to determine what approach yields better performance for which parameters.

5.2 The Euclidean norm of fresh ciphertexts

When choosing the noise vector for a new ciphertext, we want to make it as sparse as possible, i.e., increase as much as possible the probability q of choosing each entry as zero. The only limitation is that we need q to be bounded sufficiently below 1 to make it hard to recover the original noise vector from c. There are two types of attacks that we need to consider: lattice-reduction attacks that try to find the closest lattice point to c, and exhaustive-search/birthday attacks that try to guess the coefficients of the original noise vector (and a combination thereof). Pure lattice-reduction attacks should be thwarted by working with lattices with high-enough dimension, so we concentrate here on exhaustive-search attacks.

Roughly, if the noise vector has ℓ bits of entropy, then we expect birthday-type attacks to be able to recover it in $2^{\ell/2}$ time, so we need to ensure that the noise has at least 2λ bits of entropy for security parameter λ. Namely, for dimension n we need to choose q sufficiently smaller than one so that $2^{(1-q)n} \cdot \binom{n}{qn} > 2^{2\lambda}$.

Another "hybrid" attack is to choose a small random subset of the powers of r (e.g., only 200 of them) and "hope" that they include all the noise coefficients. If this holds then we can now search for a small vector in this low-dimension lattice (e.g., dimension 200). For example, if we work in dimension $n = 2048$ and use only 16 nonzero entries for noise, then choosing 200 of the 2048 entries, we have probability of about $(200/2048)^{16} \approx 2^{54}$ of including all of them (hence we can recover the original noise by solving 2^{54} instances of SVP in dimension 200). The same attack will have success probability only $\approx 2^{-80}$ if we use 24 nonzero entries.

For our public challenges we chose a (somewhat aggressive) setting where the number of nonzero entries in the noise vector is between 15 and 20. We note that increasing the noise will have only moderate effect on the performance numbers of our fully-homomorphic scheme, for example using

30 nonzero entries is likely to increase the size of the key (and the running time) by only about 5-10%.

6 Decryption

The decryption procedure takes the ciphertext c (which implicitly represents the vector $\vec{c} = \langle c, 0, \ldots, 0 \rangle$) and in principle it also has the two matrices V, W. The vector $\vec{a} = 2\vec{u} + b \cdot \vec{e}_1$ that was used during encryption is recovered as

$$\vec{a} \leftarrow \vec{c} \bmod V \;=\; \vec{c} - \left(\lceil \vec{c} \times \underbrace{V^{-1}}_{=W/d} \rfloor \times V\right) \;=\; \underbrace{[\vec{c} \;\times\; W/d]}_{[\cdot]\text{ is fractional part}} \times V,$$

and then outputs the least significant bit of the first entry of $\vec{a}$, namely $b := a_0 \bmod 2$.

The reason that this decryption procedure works is that the rows of V (and therefore also of W) are close to being orthogonal to each other, and hence the operator l_∞-norm of W is small. Namely, for any vector $\vec{x}$, the largest entry in $\vec{x} \times W$ (in absolute value) is not much larger than the largest entry in $\vec{x}$ itself. Specifically, the procedure from above succeeds when all the entries of $\vec{a} \times W$ are smaller than $d/2$ in absolute value. To see that, note that $\vec{a}$ is the distance between $\vec{c}$ and some point in the lattice $\mathcal{L}(V)$, namely we can express $\vec{c}$ as $\vec{c} = \vec{y} \times V + \vec{a}$ for some integer vector $\vec{y}$. Hence we have

$$[\vec{c} \;\times\; W/d] \times V = [\vec{y} \times V \times W/d \;+\; \vec{a} \times W/d] \overset{(\star)}{=} [\vec{a} \;\times\; W/d] \times V$$

where the equality $(\star)$ follows since $\vec{y} \times V \times W/d$ is an integer vector. The vector $[\vec{a} \;\times\; W/d] \times V$ is supposed to be $\vec{a}$ itself, namely we need

$$[\vec{a} \;\times\; W/d] \times V = \vec{a} = (\vec{a} \;\times\; W/d) \times V.$$

But this last condition holds if and only if $[\vec{a} \times W/d] = (\vec{a} \times W/d)$, i.e., $\vec{a} \times W/d$ is equal to its fractional part, which means that every entry in $\vec{a} \times W/d$ must be less than $1/2$ in absolute value.

6.1 An optimized decryption procedure

We next show that the encrypted bit b can be recovered by a significantly cheaper procedure: Recall that the (implicitly represented) ciphertext vector $\vec{c}$ is decrypted to the bit b when the distance from $\vec{c}$ to the nearest vector in the lattice $\mathcal{L}(V)$ is of the form $\vec{a} = 2\vec{u} + b\vec{e_1}$, and moreover all the entries in $\vec{a} \times W$ are less than $d/2$ in absolute value. As we said above, in this case we have $[\vec{c} \times W/d] = [\vec{a} \times W/d] = \vec{a} \times W/d$, which is equivalent to the condition

$$[\vec{c} \times W]_d \;=\; [\vec{a} \times W]_d \;=\; \vec{a} \times W.$$

Recall now that $\vec{c} = \langle c, 0, \ldots, 0 \rangle$, hence

$$[\vec{c} \times W]_d \;=\; [c \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle]_d \;=\; \langle [cw_0]_d, [cw_1]_d, \ldots, [cw_{n-1}]_d \rangle\,.$$

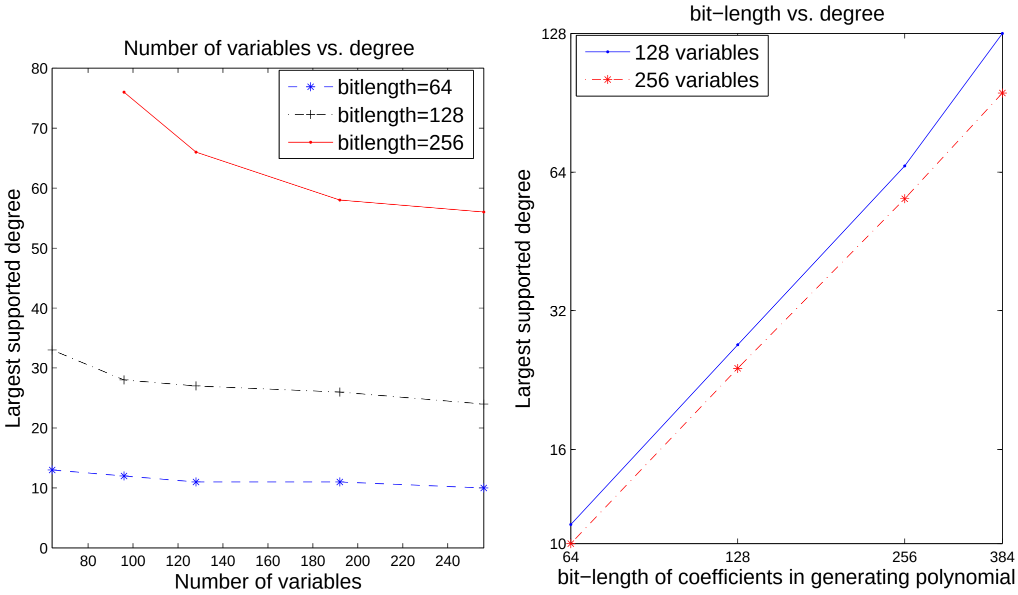
On the other hand, we have

$$[\vec{c} \times W]_d \;=\; \vec{a} \times W \;=\; 2\vec{u} \times W + b\vec{e_1} \times W \;=\; 2\vec{u} \times W + b \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle\,.$$

Putting these two equations together, we get that any decryptable ciphertext c must satisfy the relation

$$\langle [cw_0]_d, [cw_1]_d, \ldots, [cw_{n-1}]_d \rangle = b \cdot \langle w_0, w_1, \ldots, w_{n-1} \rangle \pmod 2$$

In other words, for every i we have $[c \cdot w_i]_d = b \cdot w_i \pmod 2$. It is therefore sufficient to keep only one of the w_i's (which must be odd), and then recover the bit b as $b := [c \cdot w_i]_d \bmod 2$.



m =#-of-variables t =bit-length	$m = 64$	$m = 96$	$m = 128$	$m = 192$	$m = 256$
$t =\ 64$	13	12	11	11	10
$t = 128$	33	28	27	26	24
$t = 256$	64	76	66	58	56
$t = 384$	64	96	128	100	95

Cells contain the largest supported degree for every m, t combination

Figure 1: Supported degree vs. number of variables and bit-length of the generating polynomial, all tests were run in dimension $n = 128$

7 How Homomorphic is This Scheme?

We ran some experiments to get a handle on the degree and number of monomials that the somewhat homomorphic scheme can handle, and to help us choose the parameters. In these experiments we generated key pairs for parameters n (dimension) and t (bit-length), and for each key pair we encrypted many bits, evaluated on the ciphertexts many elementary symmetric polynomials of various degrees and number of variables, decrypted the results, and checked whether or not we got back the same polynomials in the plaintext bits.

More specifically, for each key pair we tested polynomials on 64 to 256 variables. For every fixed number of variables m we ran 12 tests. In each test we encrypted m bits, evaluated all the elementary symmetric polynomials in these variables (of degree up to m), decrypted the results, and compared them to the results of applying the same polynomials to the plaintext bits. For each setting of m, we recorded the highest degree for which all 12 tests were decrypted to the correct value. We call this the "*largest supported degree*" for those parameters.

In these experiments we used fresh ciphertexts of expected Euclidean length roughly $2 \cdot \sqrt{20} \approx 9$, regardless of the dimension. This was done by choosing each entry of the noise vector $\vec{u}$ as 0 with probability $1 - \frac{20}{n}$, and as ± 1 with probability $\frac{10}{n}$ each. With that choice, the degree of polynomials that the somewhat-homomorphic scheme could evaluate did not depend on the dimension n: We tested various dimensions from 128 to 2048 with a few settings of t and m, and the largest supported degree was nearly the same in all these dimensions. Thereafter we tested all the other settings only in dimension $n = 128$.

The results are described in Figure 1. As expected, the largest supported degree grows linearly with the bit-length parameter t, and decreases slowly with the number of variables (since more variables means more terms in the polynomial).

These results can be more or less explained by the assumptions that the decryption radius of the secret key is roughly 2^t, and that the noise in an evaluated ciphertext is roughly $c^{\text{degree}} \times \sqrt{\text{\#-of-monomials}}$, where c is close to the Euclidean norm of fresh ciphertexts (i.e., $c \approx 9$). For elementary symmetric polynomials, the number of monomials is exactly $\binom{m}{\text{deg}}$. Hence to handle polynomials of degree deg with m variables, we need to set t large enough so that $2^t \geq c^{\text{deg}} \times \sqrt{\binom{m}{\text{deg}}}$, in order for the noise in the evaluated ciphertexts to still be inside the decryption radius of the secret key.

Trying to fit the data from Figure 1 to this expression, we observe that c is not really a constant, rather it gets slightly smaller when t gets larger. For $t = 64$ we have $c \in [9.14, 11.33]$, for $t = 128$ we have $c \in [7.36, 8.82]$, for $t = 256$ we get $c \in [7.34, 7.92]$, and for $t = 384$ we have $c \in [6.88, 7.45]$. We speculate that this small deviation stems from the fact that the norm of the individual monomials is not exactly c^{deg} but rather has some distribution around that size, and as a result the norm of the sum of all these monomials differs somewhat from $\sqrt{\text{\#-of-monomials}}$ times the expected c^{deg}.

Part II
A Fully Homomorphic Scheme

8 Squashing the Decryption Procedure

Recall that the decryption routine of our "somewhat homomorphic" scheme decrypts a ciphertext $c \in \mathbb{Z}_d$ using the secret key $w \in \mathbb{Z}_d$ by setting $b \leftarrow [wc]_d \bmod 2$. Unfortunately, viewing c, d as constants and considering the decryption function $D_{c,d}(w) = [wc]_d \bmod 2$, the degree of $D_{c,d}$ (as a polynomial in the secret key bits) is higher than what our somewhat-homomorphic scheme can handle. Hence that scheme is not yet bootstrappable. To achieve bootstrapping, we therefore change the secret-key format and add some information to the public key to get a decryption routine of lower degree, as done in [4].

On a high level, we add to the public key also a "big set" of elements $\{x_i \in \mathbb{Z}_d \ : \ i = 1, 2, \ldots, S\}$, such that there exists a very sparse subset of the x_i's that sums up to w modulo d. The secret key bits will be the characteristic vector of that sparse subset, namely a bit vector $\vec{\sigma} = \langle \sigma_1, \ldots, \sigma_S \rangle$ such that the Hamming weight of $\vec{\sigma}$ is $s \ll S$, and $\sum_i \sigma_i x_i = w \pmod{d}$.

Then, given a ciphertext $c \in \mathbb{Z}_d$, we post-process it by computing (in the clear) all the integers $y_i \stackrel{\text{def}}{=} \langle c x_i \rangle_d$ (i.e., c times x_i, reduced modulo d to the interval $[0, d)$). The decryption function $D_{c,d}(\vec{\sigma})$ can now be written as

$$D_{c,d}(\vec{\sigma}) \ \stackrel{\text{def}}{=} \ \left[\sum_{i=1}^{S} \sigma_i y_i \right]_d \bmod 2$$

We note that the y_i's are in the interval $[0, d)$ rather than $[-d/2, +d/2)$. This is done for implementation convenience, and correctness is not impacted since the sum of these y_i's is later reduced again modulo d to the internal $[-d/2, +d/2)$. We now show that (under some conditions), this function $D_{c,d}(\cdot)$ can be expressed as a low-degree polynomial in the bits σ_i. We have:

$$\left[\sum_{i=1}^{S} \sigma_i y_i\right]_d \;=\; \left(\sum_{i=1}^{S} \sigma_i y_i\right) - d \cdot \left\lceil \frac{\sum_i \sigma_i y_i}{d} \right\rfloor \;=\; \left(\sum_{i=1}^{S} \sigma_i y_i\right) - d \cdot \left\lceil \sum_{i=1}^{S} \sigma_i \frac{y_i}{d} \right\rfloor,$$

and therefore to compute $D_{c,d}(\vec{\sigma})$ we can reduce modulo 2 each term in the right-hand-side separately, and then XOR all these terms:

$$D_{c,d}(\vec{\sigma}) \;=\; \left(\bigoplus_{i=1}^{S} \sigma_i \langle y_i \rangle_2\right) \;\oplus\; \langle d \rangle_2 \cdot \left\langle \left\lceil \sum_{i=1}^{S} \sigma_i \frac{y_i}{d} \right\rfloor \right\rangle_2 \;=\; \bigoplus_{i=1}^{S} \sigma_i \langle y_i \rangle_2 \;\oplus\; \left\langle \left\lceil \sum_{i=1}^{S} \sigma_i \frac{y_i}{d} \right\rfloor \right\rangle_2$$

(where the last equality follows since d is odd and so $\langle d \rangle_2 = 1$). Note that the y_i's and d are constants that we have in the clear, and $D_{c,d}$ is a functions only of the σ_i's. Hence the first big XOR is just a linear functions of the σ_i's, and the only nonlinear term in the expression above is the rounding function $\left\langle \left\lceil \sum_{i=1}^{S} \sigma_i \frac{y_i}{d} \right\rfloor \right\rangle_2$.

We observe that if the ciphertext c of the underlying scheme is much closer to the lattice than the decryption capability of w, then wc is similarly much closer to a multiple of d than $d/2$. In the bootstrappable scheme we will therefore keep the noise small enough so that the distance from c to the lattice is below $1/(s+1)$ of the decryption radius, and thus the distance from wc to the nearest multiple of d is bounded below $d/2(s+1)$. (Recall that s is the the number of nonzero bits in the secret key.) Namely, we have

$$\mathsf{abs}([wc]_d) \;=\; \mathsf{abs}\left(\left[\sum_{i=1}^{S} \sigma_i y_i\right]_d\right) \;<\; \frac{d}{2(s+1)}$$

and therefore also

$$\mathsf{abs}\left(\left\lceil \sum_{i=1}^{S} \sigma_i \frac{y_i}{d} \right\rfloor\right) \;<\; \frac{1}{2(s+1)}$$

Recall now that the y_i's are all in $[0, d-1]$, and therefore y_i/d is a rational number in $[0, 1)$. Let p be our precision parameter, which we set to

$$p \;\stackrel{\text{def}}{=}\; \lceil \log_2(s+1) \rceil .$$

For every i, denote by z_i the approximation of y_i/d to within p bits after the binary point.[5] Formally, z_i is the closest number to y_i/d among all the numbers of the form $a/2^p$, with a an integer and $0 \le a \le 2^p$. Then $\mathsf{abs}(z_i - \frac{y_i}{d}) \le 2^{-(p+1)} \le 1/2(s+1)$. Consider now the effect of replacing one term of the form $\sigma_i \cdot \frac{y_i}{d}$ in the sum above by $\sigma_i \cdot z_i$: If $\sigma_i = 0$ then the sum remains unchanged, and if $\sigma_i = 1$ then the sum changes by at most $2^{p+1} \le 1/2(s+1)$. Since only s of the σ_i's are nonzero, it follows that the sum $\sum_i \sigma_i z_i$ is at most $s/2(s+1)$ away from the sum $\sum_i \sigma_i \frac{y_i}{d}$. And since the distance between the latter sum and the nearest integer is smaller than $1/2(s+1)$, then the distance between the former sum and *the same integer* is strictly smaller than $1/2(s+1) + s/2(s+1) = 1/2$. It follows that both sums will be rounded to the same integer, namely

$$\left\lceil \sum_{i=1}^{S} \sigma_i \frac{y_i}{d} \right\rfloor = \left\lceil \sum_{i=1}^{S} \sigma_i z_i \right\rfloor$$

[5] Note that z_i is in the interval $[0, 1]$, and in particular it could be equal to 1.

We conclude that for a ciphertext c which is close enough to the underlying lattice, the function $D_{c,d}$ can be computed as $D_{c,d}(\vec{\sigma}) = \langle \lceil \sum_i \sigma_i z_i \rfloor \rangle_2 \oplus \bigoplus_i \sigma_i \langle y_i \rangle_2$, and moreover the only nonlinear part in this computation is the addition and rounding (modulo two) of the z_i's, which all have only p bits of precision to the right of the binary point.

8.1 Adding the z_i's

Although it was shown in [4] that adding a sparse subset of the "low precision" numbers $\sigma_i z_i$'s can be done with a low-degree polynomial, a naive implementation (e.g., using a simple grade-school addition) would require computing about $s \cdot S$ multiplications to implement this operation. We now describe an alternative procedure that requires only about s^2 multiplications.

For this alternative procedure, we use a slightly different encoding of the sparse subset. Namely, instead of having a single vector $\vec{\sigma}$ of Hamming weight s, we instead keep s vectors $\vec{\sigma_1}, \ldots, \vec{\sigma_s}$, each of Hamming weight 1, whose bitwise sum is the original vector $\vec{\sigma}$. (In other words, we split the '1'-bits in $\vec{\sigma}$ between the s vectors $\vec{\sigma_k}$, putting a single '1' in each vector.)

In our implementation we also have s different big sets, $\mathcal{B}_1, \ldots, \mathcal{B}_s$, and each vector $\vec{\sigma_k}$ chooses one element from the corresponding $\mathcal{B}_k$, such that these s chosen elements sum up to w modulo d. We denote the elements of $\mathcal{B}_k$ by $\{x(k,i) : i = 1, 2, \ldots, S\}$, and the bits of $\vec{\sigma_k}$ by $\sigma_{k,i}$. We also denote $y(k,i) \overset{\text{def}}{=} \langle c \cdot x(k,i) \rangle_d$ and $z(k,i)$ is the approximation of $y(k,i)/d$ with p bits of precision to the right of the binary point. Using these notations, we can re-write the decryption function $D_{c,d}$ as

$$D_{c,d}(\vec{\sigma_1}, \ldots, \vec{\sigma_s}) = \left\langle \left\lceil \sum_{k=1}^{s} \underbrace{\left(\sum_{i=1}^{S} \sigma_{k,i} z(k,i) \right)}_{q_k} \right\rfloor \right\rangle_2 \oplus \bigoplus_{i,k} \sigma_{k,i} \langle y(k,i) \rangle_2 \qquad (9)$$

Denoting $q_k \overset{\text{def}}{=} \sum_i \sigma_{k,i} z(k,i)$ (for $k = 1, 2, \ldots, s$), we observe that each q_k is obtained by adding S numbers, at most one of which is nonzero. We can therefore compute the j'th bit of q_k by simply XOR-ing the j'th bits of all the numbers $\sigma_{k,i} z(k,i)$ (for $i = 1, 2, \ldots, S$), since we know a-priori that at most one of these bits in nonzero. When computing homomorphic decryption, this translate to just adding modulo d all the ciphertexts corresponding to these bits. The result is a set of s numbers u_j, each with the same precision as the z's (i.e., only $p = \lceil \log(s+1) \rceil$ bits to the right of the binary point).

Grade-school addition. Once we have only s numbers with $p = \lceil \log(s+1) \rceil$ bits of precision in binary representation, we can use the simple grade-school algorithm for adding them: We arrange these numbers in s rows and $p+1$ columns: one column for each bit-position to the right of the binary point, and one column for the bits to the left of the binary point. We denote these columns (from left to right) by indexes $0, -1, \ldots, -p$. For each column we keep a stack of bits, and we process the columns from right ($-p$) to left (0): for each column we compute the carry bits that it sends to the columns on its left, and then we push these carry bits on top of the stacks of these other columns before moving to process the next column.

In general, the carry bit that column $-j$ sends to column $-j+\Delta$ is computed as the elementary symmetric polynomial of degree 2^Δ in the bits of column $-j$. If column $-j$ has m bits, then we can compute all the elementary symmetric polynomials in these bits up to degree 2^Δ using less than $m2^\Delta$ multiplications. The Δ's that we need as we process the columns in order (column $-p$, then $1-p$, all the way through column -1) are $p-1, p-1, p-2, p-3, \ldots 1$, respectively. Also, the number of bits in these columns at the time that we process them are $s, s+1, s+2, \ldots, s+p-1$, respectively. Hence the total number of multiplications throughout this process is bounded by $s \cdot 2^{p-1} + \sum_{k=1}^{p-1}(s+k) \cdot 2^{p-k} = O(s^2)$.

Other addition algorithms. One can also use other algorithms to add these s numbers of precision p, which could could be done in less than $O(s^2)$ multiplications. (For example, using the 3-for-2 trick as proposed in [4] requires only $O(s \cdot p)$ multiplications.) In our implementation we nonetheless used grade-school addition since (a) it results in a slightly smaller polynomial degree (only 15 rather than 16, for our parameters); and (b) the addition algorithm takes only about 10% of the total running time, hence optimizing its performance had low priority for us.

9 Reducing the Public-Key Size

Two main factors contribute to the size of the public key of the fully-homomorphic scheme. One is the need to specify an instance of the sparse-subset-sum problem, and the other is the need to include in the public key also encryption of all the secret-key bits. In the next two subsections we show how we reduce the size of each of these two parts.

9.1 The Sparse-Subset-Sum Construction

Recall that with the optimization from Section 8.1, our instance of the Sparse-Subset-Sum problem consists of s "big sets" $\mathcal{B}_1, \ldots, \mathcal{B}_s$, each with S elements in $\mathbb{Z}_d$, such that there is a collection of elements, one from each $\mathcal{B}_k$, that add up to the secret key w modulo d.

Representing all of these big sets explicitly would require putting in the public key $s \cdot S$ elements from $\mathbb{Z}_d$. Instead, we keep only s elements in the public key, $x_1, \ldots, x_s$, and each of these elements implicitly defines one of the big sets. Specifically, the big sets are defined as geometric progressions in $\mathbb{Z}_d$: the k'th big set $\mathcal{B}_k$ consists of the elements $x(k,i) = \left\langle x_k \cdot R^i \right\rangle_d$ for $i = 0, 1, \ldots, S-1$, where R is some parameter. Our sparse subset is still one element from each progression, such that these s elements add up to the secret key w. Namely, there is a single index i_k in every big set such that $\sum_k x(k, i_k) = w \pmod{d}$. The parameter R is set to avoid some lattice-reduction attacks on this specific form of the sparse-subset-sum problem, see the bottom of Section 10.2 for more details.

9.2 Encrypting the Secret Key

As we discussed in Section 8.1, the secret key of the squashed scheme consists of s bit-vectors, each with S bits, such that only one bit in each vector is one, and the others are all zeros. If we encrypt each one of these bits individually, then we would need to include in the public key $s \cdot S$ ciphertexts, each of which is an element in $\mathbb{Z}_d$. Instead, we would like to include an implicit representation that takes less space but still allows us to compute encryptions of all these bits.

Since the underlying scheme is somewhat homomorphic, then in principle it is possible to store for each big set $\mathcal{B}_k$ an encrypted description of the function that on input i outputs 1 iff $i = i_k$. Such a function can be represented using only $\log S$ bits (i.e., the number of bits that it takes to represent i_k), and it can be expressed as a polynomial of total degree $\log S$ in these bits. Hence, in principle it is possible to represent the encryption of all the secret-key bits using only $s \log S$ ciphertexts, but there are two serious problems with this solution:

Recall the decryption function from Equation (9), $D_{c,d}(\ldots) = \left\langle \left\lceil \sum_{k=1}^{s} \left(\sum_{i=1}^{S} \sigma_{k,i} z(k,i) \right) \right\rfloor \right\rangle_2 \oplus \bigoplus_{i,k} \sigma_{k,i} \left\langle y(k,i) \right\rangle_2$. Since the encryption of each of the bits $\sigma_{k,i}$ is now a degree-$\log S$ polynomial in the ciphertexts that are kept in the public key, then we need the underlying somewhat-homomorphic scheme to support polynomials of degrees $\log S$ times higher than what would be needed if we store all the $\sigma_{k,i}$ themselves. Perhaps even more troubling is the increase in running time: Whereas before computing the bits of $q_k = \sum_{i=1}^{S} \sigma_{k,i} z(k,i)$ involved only additions, now we also need $S \log S$ multiplications to determine all the $\sigma_{k,i}$'s, thus negating the running-time advantage of the optimization from Section 8.1.

Instead, we use a different tradeoff that lets us store in the public key only $O(\sqrt{S})$ ciphertexts for each big set, and compute $p\sqrt{S}$ multiplications per each of the q_k's. Specifically, for every big set $\mathcal{B}_k$ we keep in the public key some $c > \left\lceil \sqrt{2S} \right\rceil$ ciphertexts, all but two of them are encryptions of zero. Then the encryption of every secret-key bit $\sigma_{k,i}$ is obtained by multiplying two of these ciphertexts. Specifically, let $\binom{c}{2}$ be the set of pairs of distinct numbers in $[1, c]$. For any $a \neq b \in [1, c]$, denote by $i(a, b)$ the index of the pair (a, b) (in the lexicographical order over $\binom{c}{2}$). That is,

$$i(a,b) \overset{\text{def}}{=} (a-1)\cdot c - \binom{a}{2} + (b-a).$$

In particular, if a_k, b_k are the indexes of the two 1-encryptions (in the group corresponding to the k'th big set $\mathcal{B}_k$), then $i_k = i(a_k, b_k)$.

A naive implementation of the homomorphic decryption with this representation will compute explicitly the encryption of every secret key bit (by multiplying two ciphertexts), and then add a subset of these ciphertexts. Here we use a better implementation, where we first add the ciphertexts in groups before multiplying. Specifically, let $\{\eta_m^{(k)} : k \in [s], m \in [c]\}$ be the bits whose encryption is stored in the public key (where for each k exactly two of the bits $\eta_m^{(k)}$ are '1' and the rest are '0', and each of the bits $\sigma_{k,i}$ is obtained as a product of two of the $\eta_m^{(k)}$'s). Then we compute each of the q_k's as:

$$q_k \;=\; \sum_{a,b} \underbrace{\eta_a^{(k)}\eta_b^{(k)}}_{\sigma(k,i(a,b))} z(k,i(a,b)) \;=\; \sum_a \eta_a^{(k)} \sum_b \eta_b^{(k)} z(k,i(a,b)) \qquad (10)$$

Since we have the bits of $z(k, i(a, b))$ in the clear, we can get the encryptions of the bits of $\eta_b^{(k)} z(k, i(a, b))$ by multiplying the ciphertext for $\eta_b^{(k)}$ by either zero or one. The only real $\mathbb{Z}_d$ multiplications that we need to implement are the multiplications by the $\eta_a^{(k)}$'s, and we only have $O(p\sqrt{S})$ such multiplications for each q_k.

Note that we get a space-time tradeoff by choosing different values of the parameter c (i.e., the number of ciphertexts that are stored in the public key for every big set). We must choose $c \geq \lceil\sqrt{2S}\,\rceil$ to be able to encode any index $i \in [S]$ by a pair $(a, b) \in \binom{c}{2}$, but we can choose it even larger. Increasing c will increase the size of the public key accordingly, but decrease the number of multiplications that need to be computed when evaluating Equation (10). In particular, setting $c = \lceil 2\sqrt{S}\,\rceil$ increases the space requirements (over $c = \lceil\sqrt{2S}\,\rceil$) only by a $\sqrt{2}$ factor, but cuts the number of multiplications in half. Accordingly, in our implementation we use the setting $c = \lceil 2\sqrt{S}\,\rceil$.

10 Setting the Parameters

10.1 The security parameters λ and μ

There are two main security parameters that drive the choice of all the others: one is a security parameter λ (that controls the complexity of exhaustive-search/birthday attacks on the scheme), and the other is a "BDDP-hardness parameter" μ. More specifically, the parameter μ it quantifies

Parameter	Meaning
$\lambda = 72$	security parameter (Section 10.1)
$\mu = 2.34,\ 0.58,\ 0.15$	BDD-hardness parameter (Section 10.1)
$s = 15$	size of the sparse subset
$p = 4$	precision parameter: number of bits for the $z(k,i)$'s
$d = 15$	the degree of the squashed decryption polynomial
$t = 380$	bit-size of the coefficients of the generator polynomial v
$n = 2^{11},\ 2^{13},\ 2^{15}$	the dimension of the lattice
$S = 1024, 1024, 4096$	size of the big sets
$R = 2^{51},\ 2^{204},\ 2^{850}$	ratio between elements in the big sets

Table 1: The various parameters of the fully homomorphic scheme. The specific numeric values correspond to our three challenges.

the exponential hardness of the Shortest-Vector-Problem (SVP) and Bounded-Distance Decoding problems (BDDP) in lattices. Specifically, we assume that for any k and (large enough) n, it takes time 2^k to approximate SVP or BDDP in n-dimensional lattices[6] to within a factor of $2^{\frac{\mu \cdot n}{k/\log k}}$. We use this specific form since it describes the asymptotic behavior of our best algorithms for approximating SVP and BDDP (i.e., the ones based on block reductions [15]).

We can make a guess as to the "true value" of μ by extrapolating from the results of Gama and Nguyen [3]: They reported achieving BDDP approximation factors of $1.01^n \approx 2^{n/70}$ for "unique shortest lattices" in dimension n in the range of 100-400. Assuming that their implementation took $\approx 2^{40}$ computation steps to compute, we have that $\mu \log(40)/40 \approx 1/70$, which gives $\mu \approx 0.11$.

For our challenges, however, we start from larger values of μ, corresponding to stronger (maybe false) hardness assumptions. Specifically, our three challenges correspond to the three values $\mu \approx 2.17$, $\mu \approx 0.54$, and $\mu \approx 0.14$. This makes it plausible that at least the smaller challenges could be solved (once our lattice-reduction technology is adapted to lattices in dimensions a few thousands). For the security parameter λ we chose the moderate value $\lambda = 72$. (This means that there may be birthday-type attacks on our scheme with complexity 2^{72}, at least in a model where each bignum arithmetic operation counts as a single step.)

10.2 The other parameters

Once we have the parameters λ and μ, we can compute all the other parameters of the system.

The sparse-subset size s and precision parameter p. The parameter that most influences our implementation is the size of the sparse subset. Asymptotically, this parameter can be made as small as $\Theta(\lambda/\log\lambda)$,[7] so we just set it to be $\lambda/\log\lambda$, rounded up to the next power of two minus one. For $\lambda = 72$ we have $\lambda/\log\lambda \approx 11.7$, so we set $s = 15$.

Next we determine the precision p that we need to keep of the $z(k,i)$'s. Recall that for any element in any of the big sets $x(k,i) \in \mathcal{B}_k$ we set $z(k,i)$ to be a p-bit-precision approximation of the rational number $\langle c \cdot x(k,i) \rangle_d / d$. To avoid rounding errors, we need p to be at least $\lceil \log(s+1) \rceil$, so for $s = 15$ we have $p = 4$. This means that we represent each $z(k,i)$ with four bits of precision

[6] What we are really assuming is that this hardness holds for the specific lattices that result from our scheme.

[7] See, e.g., [2] for some upper- and lower-bounds on the complexity of related problems.

columns:	$0_\bullet$	-1	-2	-3	-4
carry-degree from column -4:		8	4	2	
carry-degree from column -3:	9	5	3		
carry-degree from column -2:	9	7			
carry-degree from column -1:	15				
max degree:	15	8	4	2	1

Figure 2: Carry propagation for grade-school addition of 15 numbers with four bits of precision

to the right of the binary digit, and one bit to the left of the binary digit (since after rounding we may have $z(k, i) = 1$).

The degree of squashed decryption. We observe that using the grade-school algorithm for adding $s = 2^p - 1$ integers, each with p bits of precision, the degree of the polynomial that describes the carry bit to the $p + 1$'st position is less than 2^p. Specifically for the cases of $s = 15$ and $p = 4$, the degree of the carry bit is exactly 15. To see this, Figure 2 describes the carry bits that result from adding the bits in each of the four columns to the right of the binary point (where we ignore carry bits beyond the first position to the left of the point):

- The carry bit from column -4 to column -3 is a degree-2 polynomial in the bits of column -4, the carry bit to column -2 is a degree-4 polynomial, the carry bit to column -1 is a degree-8 polynomial, and there are no more carry bits (since we add only 15 bits).
- The carry bit from column -3 to column -2 is a degree-2 polynomial in the bits of column -3, including the carry bit from column -4. But since that carry bit is itself a degree-2 polynomial, then any term that includes that carry bit has degree 3. Hence the total degree of the carry bit from column -3 to column -2 is 3. Similarly, the total degrees of the carry bits from column -3 to columns $-1, 0$ are $5, 9$, respectively (since these are products of 4 and 8 bits, one of which has degree 2 and all the others have degree 1).
- By a similar argument every term in the carry from column -3 to -2 is a product of two bits, but since column -3 includes two carry bits of degrees 4 and 3, then their product has total degree 7. Similarly, the carry to column 0 has total degree 9 ($= 4 + 3 + 1 + 1$).
- Repeating the same argument, we get that the total degree of the carry bit from column -1 to columns 0 is 15 ($= 7 + 8$).

We conclude that the total degree of the grade-school addition algorithm for our case is 15, but since we are using the space/degree trade-off from Section 9.2 then every input to this algorithm is itself a degree-2 polynomial, so we get total degree of 30 for the squashed-decryption polynomial.

One can check that the number of degree-15 monomials in the polynomial representing our grade-school addition algorithm is $\binom{15}{8} \times \binom{15}{4} \times \binom{15}{2} \times 15 \approx 2^{34}$. Also, every bit in the input of the grade-school addition algorithm is itself a sum of S bits, each of which is a degree-2 monomial in the bits from the public key. Hence each degree-15 monomial in the grade-school addition polynomial corresponds to S^{15} degree-30 monomials in the bits from the public key, and the entire decryption polynomial has $2^{34} \times S^{15}$ degree-30 monomials.

The bit-size t of the generating polynomial. Since we need to support a product of two homomorphically-decrypted bits, then our scheme must support polynomials with $2^{68} \cdot S^{30}$ degree-60 monomials. Recall from Section 5.2 that we choose the noise in fresh ciphertexts with roughly 15-20 nonzero ± 1 coefficients, and we multiply the noise by 2, so fresh ciphertexts have Euclidean norm of roughly $2\sqrt{20} \approx 9$. Our experimental results from Section 7 suggest that for a degree-60 polynomial with M terms we need to set the bit-length parameter t large enough so that $2^t \geq c^{60} \times \sqrt{M}$ where c is slightly smaller than the norm of fresh ciphertexts (e.g., $c \approx 7$ for sufficiently large values of t).

We therefore expect to be able to handle homomorphic-decryption (plus one more multiplication) if we set t large enough so that $2^{t-p} \geq c^{60} \cdot \sqrt{2^{68} \cdot S^{30}}$. (We use 2^{t-p} rather than 2^t since we need the resulting ciphertext to be 2^p closer to the lattice than the decryption radium of the key, see Section 8.) For our concrete parameters ($p = 4, S \leq 2048$) we get the requirement $2^{t-p} \geq c^{60} \cdot 2^{(68+11\cdot 30)/2} = c^{60} \cdot 2^{199}$.

Using the experimental estimate $c \approx 7$ (so $c^{60} \approx 2^{170}$), this means that we expect to be able to handle bootstrapping for $t \approx 170 + 199 + 4 = 373$. Our experiments confirmed this expectation, in fact we were able to support homomorphic decryption of the product of two bits by setting the bit-length parameter to $t = 380$.

The dimension n. We need to choose the dimension n large enough so that the achievable approximation factor $2^{\mu n \log \lambda / \lambda}$ is larger than the Minkowski bound for the lattice (which is $\approx 2^t$), so we need $n = \lambda t / \mu \log \lambda$. In our case we have $t = 380$ and $\lambda / \log \lambda \approx 11.67$, so choosing the dimension as $n \in \{2^{11}, 2^{13}, 2^{15}\}$ corresponds to the settings $\mu \in \{2.17, 0.54, 0.14\}$, respectively.

Another way to look at the same numbers is to assume that the value $\mu \approx 0.11$ from the work of Gama and Nguyen [3] holds also in much higher dimensions, and deduce the complexity of breaking the scheme via lattice reduction. For $n = 2048$ we get $\lambda / \log \lambda = 2048 \cdot 0.11/380 < 1$, which means that our small challenge should be readily breakable. Repeating the computations with this value of $\mu = 0.11$ for the medium and large challenges yields $\lambda \approx 6$ and $\lambda \approx 55$, corresponding to complexity estimates of 2^6 and 2^{55}, respectively. Hence, if this estimate holds then even our large challenge may be feasibly breakable (albeit with significant effort).

This "optimistic" view should be taken with a grain of salt, however, since there are significant polynomial factors that need to be accounted for. We expect that once these additional factors are incorporated, our large challenge will turn out to be practically secure, perhaps as secure as RSA-1024. We hope that our challenges will spur additional research into the "true hardness" of lattice reduction in these high dimensions.

The big-set size S. One constraint on the size of the big sets is that birthday-type exhaustive search attacks on the resulting SSSP problem should be hard. Such attacks take time $S^{\lceil s/2 \rceil}$, so we need $S^{\lceil s/2 \rceil} \geq 2^\lambda$. For our setting with $\lambda = 72, s = 15$, we need $S^8 \geq 2^{72}$, which means $S \geq 512$.

Another constraint on S is that it has to be large enough to thwart lattice attacks on the SSSP instance. The basic lattice-based attack consists of putting all the $s \cdot S$ elements in all the big sets (denoted $\{x(k,i) : k = 1, \ldots, s,\ i = 1, \ldots, S\}$) in the following matrix:

$$B = \begin{pmatrix} 1 & & & & & x(1,1) \\ & 1 & & & & x(1,2) \\ & & \ddots & & & \vdots \\ & & & 1 & & x(s,S) \\ & & & & 1 & -w \\ & & & & & d \end{pmatrix}$$

Dimension n	bit-size t	determinant d	keyGen	Encrypt	Decrypt
512	380	$\log_2 d \approx 195764$	0.32 sec	0.19 sec	—
2048	380	$\log_2 d \approx 785006$	1.2 sec	1.8 sec	0.02 sec
8192	380	$\log_2 d \approx 3148249$	10.6 sec	19 sec	0.13 sec
32768	380	$\log_2 d \approx 12625500$	3.2 min	3 min	0.66 sec

Table 2: Parameters of the underlying somewhat-homomorphic scheme. The bit-length of the determinant is $|d| \approx \log_2 d$. Decryption time in dimension 512 is below the precision of our measurements.

with w being the secret key of the somewhat homomorphic scheme[8] and d being the determinant of the lattice (i.e., the modulus in the public key). Clearly, if $\sigma_{1,1}, \ldots, \sigma_{s,S}$ are the bits of the secret key, then the lattice spanned by the rows of B contains the vector $\langle \sigma_{1,1}, \ldots, \sigma_{s,S}, 1, 0 \rangle$, whose length is $\sqrt{sS+1}$. To hide that vector, we need to ensure that the BDDP approximation factor for this lattice is larger than the Minkowski bound for it, namely $2^{\mu(sS+2)\log\lambda/\lambda} \geq \sqrt[sS+2]{d} \approx 2^{tn/(sS+2)}$, which is roughly equivalent to $sS \geq \sqrt{tn\lambda/\mu \log \lambda}$. Using $s = 15, t = 380, \lambda = 72$ and the values of n and μ in the different dimensions, this gives the bounds $S \geq 137$ for the small challenge, $S \geq 547$ for the medium challenge, and $S \geq 2185$ for the large challenge.

Combining the two constraints, we set $S = 512$ for the small challenge, $S = 547$ for the medium challenge, and $S = 2185$ for the large challenge.

The ratio R between elements in the big sets. Since we use "big sets" of a special type (i.e., geometric progressions mod d), we need to consider also a lattice attack that uses this special form. Namely, we consider the lattice that includes only the first element in each progression

$$B = \begin{pmatrix} 1 & & & & & x(1,1) \\ & 1 & & & & x(2,1) \\ & & \ddots & & & \\ & & & 1 & & x(s,1) \\ & & & & 1 & -w \\ & & & & & d \end{pmatrix}$$

and use the fact that there is a combination of these $x(i,1)$'s with coefficients at most R^{S-1} that yields the element w modulo d. R must therefore be chosen large enough so that such combinations likely exist for many w's. This holds when $R^{s(S-1)} > d \approx 2^{nt}$. Namely, we need $\log R > nt/sS$. For our parameters in dimensions 2^{11}, 2^{13}, 2^{15}, we need $\log R \geq \frac{380}{15} \cdot \{\frac{2^{11}}{512}, \frac{2^{13}}{547}, \frac{2^{15}}{2185}\} \approx \{102, 381, 381\}$.

11 Performance

We used a strong contemporary machine to evaluate the performance of our implementation: We run it on an IBM System x3500 server, featuring a 64-bit quad-core Intel Xeon E5450 processor, running at 3GHz, with 12MB L2 cache and 24GB of RAM.

Our implementation uses Shoup's NTL library [16] version 5.5.2 for high-level numeric algorithms, and GNU's GMP library [7] version 5.0.1 for the underlying integer arithmetic operations. The code was compiled using the gcc compiler (version 4.4.1) with compilation flags `gcc -O2 -m64`.

[8] Recall that here we consider an attacker who knows w and tries to recover the sparse subset.

Dimension n	bit-size t	sparse-subset-size s	big-set size S	big-set ratio R
512	380	15	512	2^{26}
2048	380	15	512	2^{102}
8192	380	15	547	2^{381}
32768	380	15	2185	2^{381}

Dimension n	bit-size t	# of ctxts in PK ($s \cdot c$)	PK size $\approx s \cdot c \cdot \lvert d \rvert$	keyGen	Recrypt
512	380	690	17 MByte	2.5 sec	6 sec
2048	380	690	69 MByte	41 sec	32 sec
8192	380	705	284 MByte	8.4 min	2.8 min
32768	380	1410	2.25 GByte	2.2 hour	31 min

Table 3: Parameters of the fully homomorphic scheme, as used for the public challenges.

The main results of our experiments are summarized in Tables 2 and 3, for the parameter-setting that we used to generate the public challenges [6]. In Table 2 we summarize the main parameters of the underlying somewhat-homomorphic scheme. Recall that the public key of the underlying scheme consists of two $|d|$-bit integers and the secret key is one $|d|$-bit integer, so the size of these keys range from 50/25 KB for dimension 512 up to 3/1.5 MB for dimension 32768.

In Table 3 we summarize the main parameters of the fully homomorphic scheme. We note that most of the key-generation time is spent encrypting the secret-key bits: indeed one can check that key generation time for a public key with m ciphertexts takes roughly $\sqrt{m}$ longer than encryption of a single bit. (This is due to our batch encryption procedure from Section 5.1.)

We also note that 80-90% of the Recrypt time is spent adding the S numbers in each of the s big-sets, to come up with the final s numbers, and only 10-20% of the time is spent on the grade-school addition of these final s numbers. Even with the optimization from Section 9.2, the vast majority of that 80-90% is spent computing the multiplications from Equation (10). For example, in dimension 32768 we compute a single Recrypt operation in 31 minutes, of which 23 minutes are used to compute the multiplications from Equation (10), about 3.5 minutes are used to compute the arithmetic progressions (which we use for our big sets), two more minutes for the additions in from Equation (10), and the remaining 2.5 minutes are spent doing grade-school addition.

Acknowledgments. We thank Nigel Smart for many excellent comments. We also thank the CRYPTO reviewers for their helpful comments and Tal Rabin, John Gunnels, and Grzegorz Swirszcz for interesting discussions.

References

[1] R. M. Avanzi. Fast evaluation of polynomials with small coefficients modulo an integer. Web document, `http://caccioppoli.mac.rub.de/website/papers/trick.pdf`, 2005.

[2] A. Bhattacharyya, P. Indyk, D. P. Woodruff, and N. Xie. The complexity of linear dependence problems in vector spaces. In *Innovations in Computer Science - ICS 2011*, pages 496–508. Tsinghua University Press, 2011.

[3] N. Gama and P. Q. Nguyen. Predicting lattice reduction. In N. P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, 2008.

[4] 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.

[5] C. Gentry. Toward basing fully homomorphic encryption on worst-case hardness. In T. Rabin, editor, *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 116–137. Springer, 2010.

[6] C. Gentry and S. Halevi. Public Challenges for Fully-Homomorphic Encryption. `https://researcher.ibm.com/researcher/view_project.php?id=1548`, 2010.

[7] The GNU Multiple Precision Arithmetic Library. `http://gmplib.org/`, Version 5.0.1, 2010.

[8] O. Goldreich, S. Goldwasser, and S. Halevi. Public-key cryptosystems from lattice reduction problems. In B. S. K. Jr., editor, *Advances in Cryptology - CRYPTO 1997*, volume 1294 of *Lecture Notes in Computer Science*, pages 112–131. Springer, 1997.

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

[10] D. Micciancio. Improving lattice based cryptosystems using the hermite normal form. In *CaLC'01*, volume 2146 of *Lecture Notes in Computer Science*, pages 126–145. Springer, 2001.

[11] N. Ogura, G. Yamamoto, T. Kobayashi, and S. Uchiyama. An improvement of key generation algorithm for gentry's homomorphic encryption scheme. In *Advances in Information and Computer Security - IWSEC 2010*, volume 6434 of *Lecture Notes in Computer Science*, pages 70–83. Springer, 2010.

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

[13] C. Peikert and A. Rosen. Lattices that admit logarithmic worst-case to average-case connection factors. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing – STOC 2007*, pages 478–487. ACM, 2007.

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

[15] C.-P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53:201–224, 1987.

[16] V. Shoup. NTL: A Library for doing Number Theory. `http://shoup.net/ntl/`, Version 5.5.2, 2010.

[17] N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In P. Q. Nguyen and D. Pointcheval, editors, *Public Key Cryptography - PKC 2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.

[18] D. Stehlé and R. Steinfeld. Faster fully homomorphic encryption. In M. Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 377–394. Springer, 2010.