The whole alphabet (and then some) agree on a key in one round: making multilinear maps practical

Martin R. Albrecht[1], Catalin Cocis[2], Fabien Laguillaumie[3], Adeline Langlois[4]

[1] Information Security Group, Royal Holloway, University of London
[2] Technical University of Cluj-Napoca
[3] Université Claude Bernard Lyon 1, LIP (U. Lyon, CNRS, ENS Lyon, INRIA, UCBL)
[4] EPFL, Lausanne, Switzerland

Abstract. Multilinear maps have become versatile tools for designing cryptographic schemes since a first approximate realisation candidate was proposed by Garg, Gentry and Halevi (GGH). This construction was later improved by Langlois, Stehlé and Steinfeld who proposed GGHLite which offers smaller parameter sizes. In this work, we provide the first implementation of such approximate multilinear maps based on ideal lattices. While GGHLite represents an asymptotic improvement in parameter sizes compared to GGH, implementing it naively would still not allow to instantiate it for non-trivial parameter sizes. We hence propose a strategy which reduces parameter sizes further and several technical improvements to allow for an efficient implementation. In particular, since finding a prime generator for some ideal when generating instances is an expensive operation, we show how we can drop this requirement. We also propose algorithms and implementations for sampling from discrete Gaussians, for inverting in some Cyclotomic number fields and for computing norms of ideals in some Cyclotomic number rings. To demonstrate the performance of our results, we also provide an implementation of a non-interactive N-partite Diffie-Hellman key exchange and report on experimental results. For example, due to our improvements we were able to compute a multilinear map of multilinearity level $\kappa = 47$ at security level $\lambda = 80$.

1 Introduction

Multilinear maps, starting with bilinear ones, are versatile tools for designing cryptosystems. When pairings were introduced to cryptography [Jou04], many previously unreachable cryptographic primitives, such as identity-based encryption [BF03], became possible to construct. Maps of higher degree of linearity were conjectured to be hard to find – at least in the "realm of algebraic geometry" [BS03]. But in 2013, Garg, Gentry and Halevi [GGH13a] proposed a construction, relying on ideal lattices, of a so-called "graded encoding scheme" that approximates the concept of a cryptographic multilinear map.

As expected, graded encoding schemes quickly found many applications in cryptography. Already in [GGH13a] the authors showed how to generalise the 3-partite Diffie-Hellman key exchange first constructed with cryptographic bilinear maps [BS03] to N parties: the protocol allows N users to share a secret key with only one broadcast message each. Furthermore, a graded encoding scheme also allows to construct very efficient broadcast encryption [BS03,BWZ14]: a broadcaster can encrypt a message and send it to a group where only a part of it (decided by the broadcaster before encrypting) will be able to read it. Moreover, [GGH+13b] introduced indistinguishability obfuscation (iO) and functional encryption based on multilinear maps and some additional assumptions. Program obfuscation is one of the most intriguing and powerful concept in computer security, and, for now, any instantiation would require an efficiently computable multilinear map.

The GGH scheme. For a multilinearity parameter κ, the principle of a graded encoding scheme is as follows: given a ring R and a principal ideal $\mathcal{I}$ generated by a small secret element g of R, a plaintext is a small element of $R/\mathcal{I}$ and is viewed as a level-0 encoding. Given a level-0 encoding, anyone can encode an element at a higher level $i \leqslant \kappa$, but it is assumed hard to come back to an inferior level.

The encodings are additively homomorphic at the same level, and multiplicatively homomorphic up to κ operations. The multiplication of a level-i and a level-j encodings gives a level-$(i+j)$ encoding. Additionally, a zero-testing parameter p_{zt} allows to test if a level-κ element is an encoding of 0, and hence also allows to test if two level-κ encodings are encoding the same elements. Finally, the extraction procedure uses p_{zt} to extract ℓ bits which are a "canonical" representation of a ring element given its level-κ encoding.

More precisely, in GGH we are given $R = \mathbb{Z}[X]/(X^n + 1)$,where n is a power of 2, a secret element z uniformly sampled in $R_q = R/qR$ (for a certain prime number q), and a public element y which is a level-1 encoding of 1 of the form $[a/z]_q$ for some small a in the coset $1 + \mathcal{I}$. We are also given m level-i encodings of 0 named $x_j^{(i)}$, for all $1 \leqslant i \leqslant \kappa$, and a zero-testing parameter p_{zt}. To encode an element of $R/\mathcal{I}$ at level-i (for $i \leqslant \kappa$), we multiply it by y^i in R_q (which give an element of the form $[c/z^i]_q$, where c is an arbitrary small coset representative). Then we add a linear combination of encodings of 0 at level-i of the form $\sum_j \rho_j x_j^{(i)}$ to it where the ρ_j are sampled from a certain discrete Gaussian. This last step is the re-randomisation process and ought to ensure that the analogue of the discrete logarithm problem is hard: going from level-i to level-0, for example by multiplying the encoding by y^{-i}.

N-partite Diffie-Hellman key exchange. Now, given this construction for $\kappa = N - 1$, the principle of the N-partite Diffie-Hellman key exchange is as follows: during the **Setup** phase, all parameters of the system are chosen. In the **Publish** phase the ith party chooses a (secret) random level-0 encoding e_i, encodes it at level-1 and publishes this level-1 encoding. Then, in the **KeyGen** phase, each party multiplies his level-0 encoding with $(N - 1)$ level-1 encodings (of the other users) to obtain a level-κ encoding of $e = \Pi_{i=1}^N e_i$. Finally, each party runs the extraction procedure on this element to obtain the same shared secret key. Note that it is easy to find a "level-$(\kappa + 1)$" encoding of e by multiplying all the public level-1 encodings, but hard to find a level-κ encoding of e without knowing one of the e_i (this problem is formalised in the Graded Computational Diffie Hellman (GCDH) problem).

Instantiation. However, instantiating and running this protocol based on the GGH construction is too costly in practice for anything but toy instances. An alternative instantiation of a graded encoding scheme promising practicality was proposed by Coron, Lepoint and Tibouchi [CLT13]. The construction is over the integers and the authors also provided a C++ implementation of a heuristic variant of this scheme. This implementation was the only known implementation of graded encoding schemes and of a multipartite Diffie-Hellman key exchange. In [CLT13] the authors report that the **Setup** phase takes 27295s (parallelised on 16 cores), the **Publish** phase takes 17.8s per party (single core) and the **KeyGen** phase takes 20.2s per party (single core) for a level of security $\lambda = 80$ and $\kappa = 6$ (i.e. $N = 7$). More timings are available in [Lep14]. However, recently [CHL+14] demonstrated that the CLT construction can be broken in polynomial time using the encodings of zero published. The attack was later generalised to the case when no encodings of zero are published [GHMS14]. Hence, for now, we have to rule out the CLT construction as a possibility for instantiating a graded encoding scheme.

In 2014, Langlois, Stehlé and Steinfeld [LSS14a] proposed a new variant of GGH called GGHLite, improving the re-randomisation process of the original scheme. It reduces the number m of re-randomisers needed from $\Omega(n \log n)$ to 2 and also the size of the parameter $\sigma_i^\star$ of the Gaussian used to sample multipliers ρ_j during the re-randomisation phase from $\widetilde{\mathcal{O}}(2^\lambda \lambda n^{4.5}\kappa)$ to $\widetilde{\mathcal{O}}(n^{5.5}\sqrt{\kappa})$. These improvements allow to reduce the size of the public parameters and to improve the overall efficiency of the scheme. But even though [LSS14a] made a step forward towards efficiency, the scheme they proposed is still far from being practical.

Our contribution. Our main contribution is a first and efficient implementation of an improved variant of GGHLite and of the N-partite Diffie-Hellman key exchange, both of which we make publicly available under an open-source license.

While GGHLite dramatically reduces the parameter sizes compared to GGH, implementing it efficiently such that non-trivial levels of multilinearity and security can be achieved is not straight forward and to obtain a practical implementation we had to address several issues. In particular, we contribute the following improvements to make GGHLite-like multilinear maps practical:

- We show that we do not require (g) to be a prime ideal for the existing proofs to go through. Indeed, sampling an element $g \in \mathbb{Z}[X]/(X^n + 1)$ such that the ideal it generates is prime, as required by GGH and GGHLite, is a prohibitively expensive operation. Avoiding this check is then a key step to allow us to go beyond toy instances.
- We give a strategy to choose practical parameters for the scheme and extend the analysis of [LSS14a] to ensure the correctness of all the procedures of the scheme. Our refined analysis reduces the *bitsize* of q by a factor of about 4.
- We apply the analyses from [Gar13,CS97] to thwart the best known attacks. Due to our parameter modifications, recovering a short multiple of g is typically less efficient than attacking level-1 encodings of zero using a lattice as in [CS97], changing what is currently the best known attack.
- For all steps during the instance generation we provide implementations and algorithms which work in quasi-linear time and efficiently in practice. In particular, we provide algorithms and implementations for sampling from discrete Gaussians, for inverting in some Cyclotomic number fields and for computing norms of ideals in some Cyclotomic number rings.
- We discuss our implementation and report on experimental results.

Our results (cf. Table 1) are promising, as we manage to compute up to multilinearity level $\kappa = 47$ at an estimated security level of $\lambda' = 83$ and up to $\kappa = 25$ at an estimated security level of $\lambda' = 165$. We note that much smaller levels of multilinearity have been used to realise non-trivial functionality in the literature. For example, [BLR+14] reports on comparisons between 16-bit encrypted values using a 9-linear map (however, this result holds in a generic multilinear map model). For completeness, we also compare our results with those obtained in [CLT13] in Appendix C.

λ	κ	λ'	n	$\log q$	Setup	Publish	KeyGen	$\|$params$\|$
52	6	61.8	2^{15}	3079	187s	1.59s	0.38s	48.1MB
52	9	65.3	2^{16}	4729	841s	3.64s	1.28s	147.8MB
52	14	64.9	2^{16}	7273	1124s	5.01s	2.55s	227.3MB
52	19	68.0	2^{17}	10219	3783s	14.42s	9.66s	683.7MB
52	25	68.0	2^{17}	13407	6524s	19.11s	16.70s	837.9MB
52	47	71.1	2^{18}	26100	52183s	88.39s	141.35s	3.5GB
80	6	173.7	2^{17}	3733	2532s	7.31s	1.75s	233.3MB
80	9	121.2	2^{17}	5489	2644s	9.35s	3.21s	343.0MB
80	14	150.7	2^{18}	8715	15132s	26.93s	13.21s	1.1GB
80	19	108.7	2^{18}	11756	17158s	35.06s	22.91s	1.4GB
80	25	165.9	2^{19}	15929	93561s	103.19s	85.68s	3.9GB
80	47	83.8	2^{19}	29812	152052s	221.91s	343.94s	7.3GB

Table 1. Running the one-round $(\kappa+1)$-partite Diffie-Hellman key exchange. Column λ gives the minimum security level we accepted, column λ' gives the actually expected security level based on the best known attacks for the given parameter sizes. Timings produced on Intel Xeon CPU E5–2667 v2 3.30GHz with 256GB of RAM, parallelised on 16 cores, but not all operations took full advantage of all cores. Times for Publish and KeyGen are given per participant. All times are wall times.

Technical overview. Our implementation relies on FLINT [HJP14]. However, we provide our own specialised implementations for operations in the ring of integers of Cyclotomic number fields where the order is a power of two and related rings. For example, we provide our own

implementation for multiplication in $\mathbb{Z}_q[X]/(X^n+1)$, for taking approximate square roots and inverses in $\mathbb{Q}[X]/(X^n+1)$, for computing the norm of ideals in $\mathbb{Z}[X]/(X^n+1)$ (which reduces the cost by a factor of $\log n$), for checking if two ideals are co-prime and for sampling from a discrete Gaussian distribution over arbitrary ideals and cosets. For the latter we use Ducas and Nguyen's strategy [DN12]. Our implementation of these operations might be of independent interest, which is why they are available as a separate module in our code.

Our variant of GGHLite foregoes checking if g generates a prime ideal. During instance generation [GGH13a,LSS14a] specifies to sample g such that (g) is a prime ideal. This condition is needed in [GGH13a,GGH13a] to ensure that no non-zero encoding passes the zero-testing test and to argue that the non-interactive N-partite key exchange produces a shared key with sufficient entropy. We show that for both arguments we can drop the requirement that g generates a prime ideal. This was already mentioned as a potential improvement in [Gar13, Section 6.3] but not shown there. As rejection sampling until a prime ideal (g) is found is prohibitively expensive due to the low density of prime ideals in $\mathbb{Z}[X]/(X^n+1)$, this allows to speed-up instance generation such that non-trivial instances are possible. We also provide fast algorithms and implementations for checking if $(g) \subset \mathbb{Z}[X]/(X^n+1)$ is prime for applications which still require prime (g).

Another of our contribution is to improve the size of the two parameters q and ℓ compared to the one proposed in [LSS14a]. We first perform a finer analysis than [LSS14a], which allows us to reduce the *size* of the parameter q by a factor 2. Then, we introduce a new parameter ξ, which controls what fraction of q is considered "small", i.e. passes the zero-testing test, which reduces the size of q further. This also reduces the number of bits extracted from each coefficient ℓ. Indeed, instead of setting $\ell = 1/4 \log q - \lambda$ where λ is the security parameter, we set $\ell = \xi \log q - \lambda$ with $0 < \xi \leqslant 1/4$. We then show that for a good choice of ξ this is enough to ensure the correctness of the extraction procedure and the security of the scheme. Overall, our refined analysis allows us to reduce the size of $q \approx (3n^{\frac{3}{2}}\sigma_1^\star\sigma')^{8\kappa}$ in [LSS14a] to $q \approx (3n^{\frac{3}{2}}\sigma_1^\star\sigma')^{(2+\varepsilon)\kappa}$ which, in turn, allows to reduce the dimension n.

Finally, we observe that under some conditions it can be beneficial not to reduce the size of $\sigma_i^\star$ as suggested in [LSS14a]. For example, when considering a non-interactive N-partite key exchange reducing $\sigma_i^\star$ to $\mathrm{Poly}(\lambda)$ means that we need to instantiate our graded encoding scheme at security level $2\lambda+1$ in order to guarantee security level λ for the key exchange. Doubling the security level increases parameters considerably. On the other hand, since n must be a power of two, our parameter choices might provide stronger security than required. In some cases this allows us to increase q (by increasing $\sigma_i^\star$) without increasing the dimension n. We give example parameters for both choices, $\sigma_1^\star = \mathrm{Poly}(\lambda)$ and $\sigma_1^\star \approx 2^\lambda$, in Appendix B.

Open problems. We recall that there is no formal proof that the algorithmic problems underlying the security of the N-partite Diffie-Hellman key exchange using GGHLite are actually hard. It is hence a natural and important question to provide reductions to hard lattice problems or to find an alternative graded encoding construction which would allow a direct reduction to a natural problem like the shortest vector problem. While we manage to compute approximate multilinear maps for relatively high levels of κ in this work, all known schemes are still at least quadratic in κ which presents a major obstacle to efficiency. Any improvement which would reduce this to something linear in κ would mean a significant step forward. Furthermore, some applications such as indistinguishability obfuscation have different requirements on the platform graded encoding scheme which allow to reduce parameters further. Deriving concrete parameters for these applications and evaluating the performance of the resulting schemes is a natural open question. Finally, establishing better estimates for lattice reduction and tuning the parameter choices of our scheme are areas of future work.

Roadmap. We give some preliminaries in Section 2 and recall the GGHLite scheme, the one-round Diffie-Hellman key exchange and some associated security assumptions in Section 3. In Section 4, we explain our choice of parameters for the scheme, especially concerning the parameter q. We also recall some lattice attacks to derive the parameter n and show that we do not require (g) to be prime. In Section 5, we give the details of our implementation.

2 Preliminaries

Lattices and ideal lattices. An m-dimensional *lattice* L is an additive subgroup of $\mathbb{R}^m$. A lattice L can be described by its basis $B = \{b_1, b_2, \ldots, b_k\}$, with $b_i \in \mathbb{R}^m$, consisting in k linearly independent vectors, for some $k \leqslant m$, called the *rank* of the lattice. If $k = m$, we say that the lattice has *full-rank*. The lattice L spanned by B is given by $L = \{\sum_{i=1}^{k} c_i \cdot b_i, c_i \in \mathbb{Z}\}$. The volume of the lattice L, denoted by $\mathrm{vol}(L)$, is the volume of the parallelepiped defined by its basis vectors. We have $\mathrm{vol}(L) = \sqrt{\det(B^T B)}$, where B is any basis of L.

For n a power of two, let $f(X) \in \mathbb{Z}[X]$ be a monic polynomial of degree n (in our case, $f(X) = X^n + 1$). Then, the polynomial ring $R = \mathbb{Z}[X]/f(X)$ is isomorphic to the integer lattice $\mathbb{Z}^n$, i.e. we can identify an element $u(X) = \sum_{i=0}^{n-1} u_i \cdot X^i \in R$ with its corresponding coefficient vector $(u_0, u_1, \ldots, u_{n-1})$. We also define $R_q = R/qR = \mathbb{Z}_q[X]/(X^n + 1)$ (isomorphic to $\mathbb{Z}_q^n$) for a large prime q and $K = \mathbb{Q}[X]/(X^n + 1)$ (isomorphic to $\mathbb{Q}^n$).

Given an element $g \in R$, we denote by $\mathcal{I}$ the principal ideal in R generated by g: $(g) = \{g \cdot u : u \in R\}$. The ideal (g) is also called an *ideal lattice* and can be represented by its $\mathbb{Z}$-basis $(g, X \cdot g, \ldots, X^{n-1} \cdot g)$. We denote by $\mathcal{N}(g)$ its norm. For any $y \in R$, let $[y]_g$ be the reduction of y modulo $\mathcal{I}$. That is, $[y]_g$ is the unique element in R such that $y - [y]_g \in (g)$ and $[y]_g = \sum_{i=0}^{n-1} y_i X^i g$, with $y_i \in [-1/2, 1/2), \forall i, 0 \leqslant i \leqslant n-1$. Following [LSS14a] we abuse notation and let $\sigma_n(b)$ denotes the last singular value of the matrix $\mathrm{rot}(b) \in \mathbb{Z}^{n \times n}$, for any $b \in \mathcal{I}$. For $z \in R$, we denote by $\mathrm{MSB}_\ell \in \{0,1\}^{\ell \cdot n}$ the ℓ most significant bits of each of the n coefficients of z in R.

Gaussian distributions. For a vector $c \in \mathbb{R}^n$ and a positive parameter $\sigma \in \mathbb{R}$, we define the Gaussian distribution of centre c and width parameter σ as $\rho_{\sigma,c}(x) = \exp(-\pi \frac{||x-c||^2}{\sigma^2})$, for all $x \in \mathbb{R}^n$. This notion can be extended to ellipsoid Gaussian distribution by replacing the parameter σ with the square root of the covariance matrix $\Sigma = BB^t \in \mathbb{R}^{n \times n}$ with $\det(B) \neq 0$. We define it by $\rho_{\sqrt{\Sigma},c}(x) = \exp(-\pi \cdot (x-c)^t (B^t B)^{-1} (x-c))$, for all $x \in \mathbb{R}^n$. For L a subset of $\mathbb{Z}^n$, let $\rho_{\sigma,c}(L) = \sum_{x \in L} \rho_{\sigma,c}(x)$. Then, the *discrete Gaussian distribution* over L with centre c and standard deviation σ (resp. $\sqrt{\Sigma}$) is defined as $D_{L,\sigma,c}(y) = \frac{\rho_{\sigma,c}(y)}{\rho_{\sigma,c}(L)}$, for all $y \in L$. We use the notations ρ_σ (resp. $\rho_{\sqrt{\Sigma}}$) and $D_{L,\sigma}$ (resp. $D_{L,\sqrt{\Sigma}}$) when c is 0.

Finally, for a fixed $Y = (y_1, y_2) \in R^2$, we define: $\widetilde{\mathcal{E}}_{Y,s} = y_1 D_{R,s} + y_2 D_{R,s}$ as the distribution induced by sampling $\boldsymbol{u} = (u_1, u_2) \in R^2$ from a discrete spherical Gaussian with parameter s, and outputting $y = y_1 u_1 + y_2 u_2$. It is shown in [LSS14a, Th. 5.1] that if $Y \cdot R^2 = \mathcal{I}$ and $s \geqslant \max(\|g^{-1} y_1\|_\infty, \|g^{-1} y_2\|_\infty) \cdot n \cdot \sqrt{2\log(2n(1+1/\varepsilon))/\pi}$ for $\varepsilon \in (0, 1/2)$, this distribution is statistically close to the Gaussian distribution $D_{\mathcal{I}, sY^T}$.

2.1 Graded encoding scheme

The first candidate realisation of multilinear maps from [GGH13a] does not completely fit the formal definition from [BS03]. Instead, it realises the notion of graded encoding scheme. A graded encoding scheme manipulates *encoding levels*: the "plaintext" is a level-0 encoding. From a level-0 encoding one can construct a level-i encoding of the same element until κ, where κ is called the multilinearity parameter. It is assumed that given a level-i encoding it is hard to find a level-j encoding for $j < i$ of the same element. Encodings are both additively and multiplicatively homomorphic (up to κ operations for the multiplication). We define the procedures formally in Appendix A.

3 GGHLite and N-partite Diffie-Hellman Key exchange

In this section, we describe the GGHLite encoding scheme, the N-partite Diffie-Hellman key exchange using GGHLite and the Graded Decisional Diffie-Hellman assumption (GDDH) on which the security of the key exchange relies.

3.1 The GGHLite graded encoding scheme

In Figure 1 we recall the GGHLite graded encoding scheme from [LSS14a]. This scheme is adapted from the original [GGH13a] scheme. The principle of this scheme is the following: we are given a public principal ideal $\mathcal{I}$, generated by a small secret element g of a polynomial ring $R = \mathbb{Z}[X]/(X^n + 1)$, a secret element z uniformly sampled in R_q, and a public element y which is a level-1 encoding of 1 of the form $[a/z]_q$ with $a \hookleftarrow D_{1+\mathcal{I},\sigma'}$. Then, to encode an element of $R/\mathcal{I}$ at level-i, we multiply it by y^i in R_q, and we add a random linear combination of encodings of 0 at level-i to randomise this encoding.

- **Instance generation.** $\mathsf{InstGen}(1^\lambda, 1^\kappa)$: Given security parameter λ and multilinearity parameter κ, determine scheme parameters n, q, σ, σ', $\sigma_k^\star$, $\ell_{g^{-1}}$, ℓ_b, ℓ as described below. Then proceed as follows:
  - Sample $g \hookleftarrow D_{R,\sigma}$ until $\|g^{-1}\| \leqslant \ell_{g^{-1}}$ and $\mathcal{I} = (g)$ is a prime ideal. Define encoding domain $R_g = R/(g)$.
  - Sample $z \hookleftarrow U(R_q)$.
  - Sample a level-1 encoding of 1: set $y = [a \cdot z^{-1}]_q$ with $a \hookleftarrow D_{1+\mathcal{I},\sigma'}$.
  - For $k \leqslant \kappa$:
    - Sample $B^{(k)} = (b_1^{(k)}, b_2^{(k)})$ from $(D_{\mathcal{I},\sigma'})^2$. If $(b_1^{(k)}, b_2^{(k)}) \neq \mathcal{I}$, or $\sigma_n(\mathrm{rot}(B^{(k)})) < \ell_b$ or $\|B^{(k)}\| > \sqrt{n} \cdot \sigma'$, then re-sample.
    - Define level-k encodings of 0: $x_1^{(k)} = [b_1^{(k)} \cdot z^{-k}]_q$, $x_2^{(k)} = [b_2^{(k)} \cdot z^{-k}]_q$.
  - Sample $h \hookleftarrow D_{R,\sqrt{q}}$ and define the zero-testing parameter $p_{zt} = [\frac{h}{g} z^\kappa]_q \in R_q$.
  - Return public parameters $\mathsf{params} = (n, q, y, \sigma', \{\sigma_k^\star\}_{k\leqslant\kappa}, \ell, \{(x_1^{(k)}, x_2^{(k)})\}_{k\leqslant\kappa})$ and p_{zt}.
- **Level-0 sampler.** $\mathsf{Samp}(\mathsf{params})$: Sample $e \hookleftarrow D_{R,\sigma'}$ and return e.
- **Level-k encoding.** $\mathsf{Enc}_k(\mathsf{params}, e)$: Given level-0 encoding $e \in R$ and parameters params:
  - Encode e at level k: Compute $u' = [e \cdot y^k]_q$.
  - Return $u = [u' + \rho_1 \cdot x_1^{(k)} + \rho_2 \cdot x_2^{(k)}]_q$, with $\rho_1, \rho_2 \hookleftarrow D_{R,\sigma_k^\star}$.
- **Adding encodings.** Add: Given level-k encodings $u_1 = [c_1/z^k]_q$ and $u_2 = [c_2/z^k]_q$:
  - Return $u = [u_1 + u_2]_q$, a level-k encoding of $[c_1 + c_2]_g$.
- **Multiplying encodings.** Mult: Given level-k_1 encoding $u_1 = [c_1/z^{k_1}]_q$ and a level-k_2 encoding $u_2 = [c_2/z^{k_2}]_q$:
  - Return $u = [u_1 \cdot u_2]_q$, a level-$(k_1 + k_2)$ encoding of $[c_1 \cdot c_2]_g$.
- **Zero testing at level κ.** $\mathsf{isZero}(\mathsf{params}, p_{zt}, u)$: Given a level-$\kappa$ encoding $u = [c/z^\kappa]_q$, return 1 if $\|[p_{zt} u]_q\|_\infty < q^{3/4}$ and 0 else.
- **Extraction at level κ.** $\mathsf{Ext}(\mathsf{params}, p_{zt}, u)$: Given a level-$\kappa$ encoding $u = [c/z^\kappa]_q$, return $v = \mathrm{MSB}_\ell([p_{zt} \cdot u]_q)$.

Fig. 1. The GGHLite graded encoding scheme as described in [LSS14a].

The following asymptotic choice of parameters is detailed in [LSS14a]:

- The dimension n is $O(\kappa\lambda \log \lambda)$, the modulus q is $\widetilde{O}(n^{10.5}\sqrt{\kappa})^{8\kappa}$.
- Generating g:
  - the parameter σ is $O(n \log n)$, more precisely, for some constant $p_g < 1$:
$$\sigma \geqslant 4\pi n \sqrt{e \ln(8n)/\pi}/p_g,$$
  - the upper bound $\ell_{g^{-1}}$ is $O(1/\sqrt{n \log n})$, more precisely $\ell_{g^{-1}} = \frac{4\sqrt{\pi e n}}{p_g \sigma}$.
- Generating $(b_i^{(k)})_{i,k}$ and a:
  - the parameter σ' is $\widetilde{O}(n^{3.5})$, more precisely, $\sigma' \geqslant 7n^{\frac{5}{2}} \ln^{\frac{3}{2}}(n)\sigma$,
  - the lower bound ℓ_b is $O(n^3)$, more precisely, for some constant $p_b < 1$, $\ell_b = \frac{p_b}{2\sqrt{\pi e n}}\sigma'$.
- Re-randomisation:
  - the parameter $\sigma_1^\star$ is $\widetilde{O}(n^{5.5}\kappa)$, more precisely, $\sigma_1^\star = n^{3/2} \cdot (\sigma')^2 \sqrt{8\pi/\varepsilon_d}/\ell_b$ with $\varepsilon_d = \log(\lambda/\kappa)$,

- more generally, for $1 \leqslant k \leqslant \kappa$, let $u = [c/z^k]_q$ be the level-k encoding to re-randomize with $\|c\| \leqslant \gamma_k$, then the parameter $\sigma_k^\star$ is $\left(\sqrt{8\pi/\varepsilon_d}/\ell_b\right) \cdot \gamma_k$.

- The number of extracted bit ℓ is such that: $\log_2(8n\sigma) < \ell \leqslant 1/4 \log_2 q - \log_2(2n/\varepsilon_{ext})$, where ε_{ext} is the negligible probability that the extraction are the same for two different elements.

Those parameters ensure the correctness of the zero-testing and extraction procedures.

3.2 One-round N-partite Diffie-Hellman key exchange

In Figure 2 we recall the construction given by [GGH13a] to adapt the N-partite Diffie-Hellman key exchange using an encoding scheme with $\kappa = N - 1$. The principle of the key exchange is that each party shares some public parameters and starts by sampling a secret key. Then each party publishes a public element (computed with its secret key), and given all the public elements and her secret, each party must be able to compute a shared secret key. The consistency requirement is that all parties must generate the same shared secret key. It follows from the agreement property

---

- **Setup**. $\mathsf{Setup}(1^\lambda, 1^N)$: Given security parameter λ and number of parties N, run $\mathsf{InstGen}(1^\lambda, 1^{N-1})$ for the graded encoding scheme to get $(\mathsf{params}, p_{zt})$ and output protocol public parameters $(\mathsf{params}, p_{zt})$.
- **Publish**. $\mathsf{Publish}(\mathsf{params}, p_{zt}, i)$: The ith party runs the level-0 encoding sampler to generate a random secret key $e_i = \mathsf{Samp}(\mathsf{params})$, and publishes a corresponding level-1 public key $u_i = \mathsf{enc}_1(\mathsf{params}, e_i)$.
- **KeyGen**. $\mathsf{KeyGen}(\mathsf{params}, p_{zt}, j, e_j, \{u_i\}_{i \neq j})$: The jth party computes a level-$(N-1)$ encoding $v_j = e_j \cdot \prod_{i \neq j} u_i$ of the product $\prod_i e_i$, and outputs the extracted string $s_j = \mathsf{ext}(\mathsf{params}, p_{zt}, v_j)$ as the common key.

Fig. 2. The adapted N-partite Diffie-Hellman key exchange protocol from [GGH13a].

of the extraction procedure. As proved in [GGH13a], the security comes from the randomness property of the extraction procedure and from the Graded Decisional Diffie-Hellman assumption (see Def. 1) that we define in the next section.

3.3 Graded DH assumptions and setting parameters

In [LSS14a], the key exchange is modified so that the final common key is the hashed value of the final product modelled as a random oracle. In this case, the protocol can be proved secure under the *Extraction Graded Computational Diffie-Hellman assumption* (see Def. 1). However, there is a loss of security in the reduction from [LSS14a, Lemma 8.1] which has a dramatic impact on the efficiency of the scheme. That is, the reduction from GDDH – which is a decisional problem to Ext-GCDH which is a computational one – has to take into account guessing a good query to the random oracle, which reduces the advantage by a factor of 2^λ. As a consequence, in [LSS14a, Figure 7], the setup phase starts with a run of $\mathsf{InstGen}(1^{2\lambda+1}, 1^{N-1})$. This requires to (at least) double the dimension n and has severe performance penalties.

To overcome this issue, we implement the key-exchange without modelling the final step as a random oracle, so that its security relies on the Graded Decisional Diffie-Hellman assumption (see Def. 1, previously defined in [LSS14a] and adapted from the original GDDH/GCDH assumptions of [GGH13a]). The consequence is that we cannot benefit from the powerful analysis provided in [LSS14a] using the Rényi divergence, which allows to save an exponential factor in the re-randomization parameter $\sigma_1^\star$. It appears, though, that using this enlarged parameter has less of a damaging effect for most parameter choices we considered than generating parameters with twice the security level, as shown in Tables 7 and 8 in Appendix B.

Definition 1 ([LSS14a, Definition 3.1] GDDH/Ext-GCDH). *Define elements as in Figure 3. The* ***κ-graded DDH problem (GDDH)*** *is defined as follows: Distinguish between the distributions* $\{\mathsf{params}, p_{zt}, (u_i)_{i=1,\ldots,\kappa}, v_C\}$ *and* $\{\mathsf{params}, p_{zt}, (u_i)_{i=1,\ldots,\kappa}, v_R\}$.

Given parameters λ, κ, proceed as follows:

1. Run $\mathsf{InstGen}(1^\lambda, 1^\kappa)$ to get $\mathsf{params} = (n, q, y, \sigma', \{\sigma_k^\star\}_{k\leqslant\kappa}, \ell, \{(x_1^{(k)}, x_2^{(k)})\}_{k\leqslant\kappa})$ and p_{zt}.
2. Sample $f \hookleftarrow D_{R,\sigma'}$
3. For $i = 0, \ldots, \kappa$:
   - Sample $e_i \hookleftarrow D_{R,\sigma'}$.
   - Set $u_i = [e_i \cdot y + \rho_{i,1} x_1^{(1)} + \rho_{i,2} x_2^{(1)}]_q$ with $\rho_{i,j} \hookleftarrow D_{R,\sigma_1^\star}$ for $j \in \{1, 2\}$.
4. Set $u^* = \left[\prod_{i=1}^{\kappa} u_i\right]_q$.
5. Set $v_C = [e_0 u^*]_q$.
6. Set $v_R = [f u^*]_q$.

Fig. 3. GGH challenge generation (only level-1 encodings of 0 and $\sigma_1^\star$ are necessary).

*The **Extraction κ-graded CDH problem (Ext-GCDH)** is defined as follows: On inputs* params, p_{zt} *and the u_i's, output the extracted string for a level-κ encoding of $\prod_{i\geqslant 0} e_i + \mathcal{I}$, i.e., $w = \mathsf{Ext}(\mathsf{params}, p_{zt}, v_C) = MSB_\ell([p_{zt} \cdot v_C]_q)$.*

4 Parameter selection

We choose the parameters σ, σ', $\sigma_1^\star$, $\ell_{g^{-1}}$ and ℓ_b according to the conditions described in Section 3.1. In this section, we first show that we do not require a prime (g) and then describe a method which allows to reduce the size of two parameters: the modulus q and the number ℓ of extracted bits. In Section 4.3 we describe the best known attacks against the scheme. We need to choose parameters n and q to resist these attacks. Finally, we describe our strategy to choose parameters that satisfy all these constraints.

We note that in this work we focus exclusively on the case where re-randomisers are published for level-1 but no other levels. This matches the requirements of the N-partite Diffie-Hellman key exchange. The general case can be generalised by increasing q to accommodate "numerator growth" due to re-randomisation at higher levels.

4.1 Non-prime (g)

GGHLite as specified in Figure 1 requires to sample a g such that (g) is a prime ideal. However, finding such a g is prohibitively expensive. While checking each individual g whether (g) is a prime ideal is asymptotically not slower than polynomial multiplication, finding such a g requires to run this check often. The probability that an element generates a prime ideal is assumed to be roughly $1/(n^c)$ for some constant $c > 1$ [Gar13, Conjecture 5.18], so we expect to run this check n^c times. Hence, the overall complexity is at least quadratic in n which is too expensive for anything but toy instances.

Primality of (g) is used in two proofs. Firstly, to ensure that after multiplying $\kappa + 1$ elements in R_g the product contains enough entropy. This is used to argue security of the N-partite non-interactive key exchange. Secondly, to prove that $c \cdot h/g$ is big if $c, h \notin g$ (cf. Lemma 2). Below, we show that we can relax the conditions on g for these two arguments to still go through, which then allows us to drop the condition that (g) should be prime. We note, though, that some other applications might still require g to be prime and that future attacks might find a way to exploit non-prime (g).

Entropy of the product. The next lemma shows that excluding prime factors $\leqslant 2N$ and guaranteeing $\mathcal{N}(g) \geqslant 2^n$ is sufficient to ensure 2λ bits of entropy in a product of N elements in R_g. We note that both conditions hold with high probability, are easy to check and are indeed checked in our implementation.

Lemma 1. *Let $\kappa \geqslant 2$, λ be the security parameter and $g \in \mathbb{Z}[X]/(X^n+1)$ with norm $p = \mathcal{N}(g) \geqslant 2^n$ such that p has no prime factors $\leqslant 2\kappa+2$, and such that $n \geqslant \kappa \cdot \lambda \cdot \log(\lambda)$. Then the product of $N = \kappa + 1$ uniformly random elements in R_g has at least $\kappa \cdot \lambda \cdot \log(\lambda)/4$ bits of entropy.*

Proof. Write $p = \prod_{i=1}^{r} p_i^{e_i}$ where p_i are distinct primes and $e_i \geqslant 1$ for all i. Let us consider the set $\mathcal{S} = \{i \in \{1, \ldots, r\} : e_i = 1\}$. Then, $p_s = \prod_{i \in \mathcal{S}} p_i$ is the square-free part of p. It is known (see for instance [CDKD14]) that the square-free part p_s of p asymptotically dominates its powerful part p/p_s, which means that $p_s \geqslant \sqrt{p}$ and then $\log(p_s) \geqslant n/2$.

By the Chinese Remainder Theorem, R_g is isomorphic to $F_1 \times \cdots \times F_r$ where each "slot" $F_i = \mathbb{Z}_{p_i^{e_i}}$. The set of F_i, for $i \in \mathcal{S}$ corresponds to the square-free part of p. Those F_i are fields, and each of them has order $p_i \geqslant 2N$ which means that a random element in such F_i is zero with probability $1/p_i$. In those slots, the product of N elements has E_s bits of entropy, where

$$E_s = \sum_{i \in \mathcal{S}} \left(1 - \frac{N}{p_i}\right) \log(p_i).$$

First, as $p_i \geqslant 2N$ for all $i \in \mathcal{S}$, the quotient $N/p_i \leqslant 1/2$ and then $\left(1 - \frac{N}{p_i}\right) \geqslant 1/2$ for all $i \in \mathcal{S}$. This implies that

$$E_s \geqslant 1/2 \sum_{i \in \mathcal{S}} \log(p_i) = 1/2 \log\left(\prod_{i \in \mathcal{S}} p_i\right) = 1/2 \log(p_s).$$

Because $\log(p_s) \geqslant n/2$, we conclude that $E_s \geqslant \frac{n}{4} \geqslant \frac{\kappa \cdot \lambda \cdot \log(\lambda)}{4}$. □

Probability of false positive. It remains to be shown that we can ensure that there are no false positives even if (g) is not prime. In [GGH13a, Lemma 3] false positives are ruled out as follows. Let $u = [c/z^\kappa]_q$ where c is a short vector in some coset of $\mathcal{I}$, and let $w = [p_{zt} \cdot u]_q$, then we have $w = [c \cdot h/g]_q$. The first step in [GGH13a] is to suppose that $\|g \cdot w\|$ and $\|c \cdot h\|$ are each at most $q/2$, then, since $g \cdot w = c \cdot h \bmod q$ we have that $g \cdot w = c \cdot h$ exactly. We also have an equality of ideals: $(g) \cdot (w) = (c) \cdot (h)$, and then several cases are possible. If (g) is prime as in [GGH13a, Lemma 3], then (g) divides either (c) or (h) and either c or h is in (g). As, by construction, none of them is in (g) if c is not in $\mathcal{I}$, either $\|g \cdot w\|$ or $\|c \cdot h\|$ is more than $q/2$. Using this, they conclude that there is no small c (not in $\mathcal{I}$) such that w is small enough to be accepted by the zero-test.

Our approach is to simply notice that all we require is that (g) and (h) are co-prime. Checking if (g) and (h) are co-prime can be done by checking $\gcd(\mathcal{N}(g), \mathcal{N}(h)) = 1$. However, computing $\mathcal{N}(h)$ is rather costly because h is sampled from $D_{\mathbb{Z}^n, \sqrt{q}}$ and hence has a large norm $\mathcal{N}(h)$. To deal with this issue we notice that if $\gcd(\mathcal{N}(g), \mathcal{N}(h)) \neq 1$ then we also have $\gcd(\mathcal{N}(g), \mathcal{N}(h \bmod g)) \neq 1$ which can be verified with a simple calculation. Now, interpreting $h \bmod g$ as "a small representative of h modulo g", which we can compute as $h - g \cdot \lfloor g^{-1} \cdot h \rceil$, we can use this observation to reduce the complexity of checking if (g) and (h) are co-prime to computing two norms for elements of size $\approx \|g\|$ and taking their gcd. Furthermore, this condition holds with high probability, i.e. we only have to perform this test $O(1)$ times. Indeed, by ruling out likely common prime factors first, we expect to run this test exactly once. Hence, checking co-primality of (g) and (h) is much cheaper than finding a prime (g) but still rules out false positives.

4.2 Reducing the size of q

The size of q is driven from both correctness and security considerations. To ensure the correctness of the zero-testing procedure, [LSS14a] showed the two following lower bounds on q. Eq. 1 implies that false negatives do not exist, and Eq. 2 implies that the probability of false positive occurrence is negligible:

$$q > \max\left((n\ell_{g^{-1}})^8, (3n^{\frac{3}{2}}\sigma_1^\star\sigma')^{8\kappa}\right), \tag{1}$$

$$q > (2n\sigma)^4. \tag{2}$$

The strongest constraint for q is given by the inequality $q > (3n^{\frac{3}{2}}\sigma_1^\star\sigma')^{8\kappa}$. It comes from the fact that for any level-κ encoding u of 0 (*i.e.*, $u \in S_\kappa^{(0)}$), the inequality $\|p_{zt}u\|_\infty < q^{3/4}$ has to hold. The condition is needed for the correctness of zero-testing and extraction.

New parameter ξ. The choice suggested in [LSS14a] is to extract $\ell = \log(q)/4 - \lambda$ bits from each element of the level-κ encoding. We show that this supplies much more entropy than needed and that we can sample a smaller fraction, $\ell = \xi\log(q) - \lambda$ bits. The equation for q can be rewritten in terms of the variable ξ, by setting the initial condition $\|p_{zt}\,u\|_\infty < q^{1-\xi}$.

Lemma 2 (Adapted from Lemma A.1 in [LSS14b]). *Let $g \in R$ and $\mathcal{I} = (g)$, let $c, h \in R$ such that $c \notin \mathcal{I}$, (g) and (h) are co-prime, $\|c \cdot h\| < q/2$ and $q > (2tn\sigma)^{1/\xi}$ for some $t \geqslant 1$. Then $\|[c \cdot h/g]_q\| > t \cdot q^{1-\xi}$ for any $0 < \xi \leqslant 1/4$.*

Proof. We know that since $\|c \cdot h\| < q/2$ we must have $\|g \cdot [c \cdot h/g]_q\| > q/2$ if (g) and (h) are co-prime. So we have $\|g \cdot [c \cdot h/g]_q\| > q/2 \Rightarrow \sqrt{n}\|g\| \cdot \|[c \cdot h/g]_q\| > q/2 \Rightarrow \|[c \cdot h/g]_q\| > q/(2n\sigma)$. We have $t \cdot q^{1-\xi} = t \cdot q/q^\xi < t \cdot q/(2tn\sigma) = q/(2n\sigma)$ and the claim follows. □

Correctness of zero-testing. We can obtain a tighter bound on q by refining the analysis in [LSS14a]. Recall that $\|[p_{zt}\,u]_q\|_\infty = \|[hc/g]_q\|_\infty = \|h \;\cdot\; c/g\|_\infty \leqslant \|h\| \cdot \|c/g\| \leqslant \|h\| \cdot \|c\| \cdot \|g^{-1}\|\sqrt{n}$. The first inequality is a direct application of the inequalities between the infinity norm of a product and the product of the Euclidean norms, the second comes from [Gar13, Lemma 5.9].

Since $h \leftarrow D_{R,\sqrt{q}}$, we have $\|h\| \leqslant \sqrt{n}q^{1/2}$. Moreover, c can be written as a product of κ level-1 encodings u_i, for $i = 1, \ldots, \kappa$, *i.e.*, $c = \prod_{i=1}^{\kappa} u_i$. Thus, $\|c\| \leqslant (\sqrt{n})^{\kappa-1}(\max_{i=1,\ldots,\kappa}\|u_i\|)^\kappa$ since each of the $\kappa - 1$ multiplications brings an extra $\sqrt{n}$ factor. Let u_{max} be one of the u_i of largest norm. It can be written as $u_{max} = e \cdot a + \rho_1 \cdot b_1^{(1)} + \rho_2 \cdot b_2^{(1)}$. As we sampled the polynomial g such that $\|g^{-1}\| \leqslant l_{g^{-1}}$ the inequality $\|[p_{zt}\,u]_q\|_\infty < q^{1-\xi}$ holds if:

$$nl_{g^{-1}}(\sqrt{n})^{\kappa-1}\|(e \cdot a + \rho_1 \cdot b_1^{(1)} + \rho_2 \cdot b_2^{(1)})\|^\kappa < q^{1/2-\xi}. \tag{3}$$

Then, since $\|e \cdot a + \rho_1 \cdot b_1^{(1)} + \rho_2 \cdot b_2^{(1)}\|^\kappa \leqslant (\|e\| \cdot \|a\|\sqrt{n} + \|\rho_1\| \cdot \|b_1^{(1)}\|\sqrt{n} + \|\rho_2\| \cdot \|b_2^{(1)}\|\sqrt{n})^\kappa$, $e \leftarrow D_{R,\sigma'}, a \leftarrow D_{1+I,\sigma'}, b_1^{(1)}, b_2^{(1)} \leftarrow D_{I,\sigma'}$ and $\rho_1, \rho_2 \leftarrow D_{R,\sigma_1^\star}$, we can bound each of these values as $\|e\|, \|a\|, \|b_1^{(1)}\|, \|b_2^{(1)}\| \leqslant \sigma'\sqrt{n}, \|\rho_1\|, \|\rho_2\| \leqslant \sigma_1^\star\sqrt{n}$ to get:

$$nl_{g^{-1}}(\sqrt{n})^{\kappa-1}(\sigma'\sqrt{n} \cdot \sigma'\sqrt{n} \cdot \sqrt{n} + 2 \cdot \sigma_1^\star\sqrt{n} \cdot \sigma'\sqrt{n} \cdot \sqrt{n})^\kappa < q^{1/2-\xi},$$

$$\left(nl_{g^{-1}}(\sqrt{n})^{\kappa-1}((\sigma')^2 n^{\frac{3}{2}} + 2\sigma_1^\star\sigma' n^{\frac{3}{2}})^\kappa\right)^{\frac{2}{1-2\xi}} < q. \tag{4}$$

In [LSS14a], we had $\xi = 1/4$ (which give $2/(1-2\xi) = 4$), we hence have that this analysis allows to save a factor of 2 in the size of q even for the same ξ. If we additionally consider $\xi < 1/4$ bigger improvements are possible. Consulting Table 2 in Appendix B shows that for practical parameter sizes we reduce the size of q by a factor of almost 4 because ξ tends towards zero a κ and λ grow.

Correctness of extraction. As in [LSS14a], we need that two level-κ encodings u and u' of different elements have different extracted elements, which implies that we need: $\|[p_{zt}(u-u')]_q\|_\infty > 2^{L-\ell+1}$ with $L = \lfloor\log q\rfloor$. This condition follows from Lemma 2 with t satisfying $t \cdot q^{1-\xi} > 2^{L-\ell+1}$, which holds for $t = q^\xi \cdot 2^{-\ell+1}$. As a consequence, the condition $q > (2tn\sigma)^{1/x}$ is still satisfied if we have $\ell > \log_2(8n\sigma)$, and to ensure that $t > 1$ we need that $\ell < \xi\log q + 2$. Finally, to ensure that ε_{ext}, the probability of the extraction to be the same for two different elements, is negligible, we need that $\ell \leqslant \xi\log_2 q - \log_2(2n/\varepsilon_{ext})$.

Picking ξ and q. Putting all constraints together, we let $\ell = \log(8n\sigma)$ and

$$\tilde{q} = nl_{g^{-1}}(\sqrt{n})^{\kappa-1}\left((\sigma')^2 n^{\frac{3}{2}} + 2\sigma_1^\star\sigma' n^{\frac{3}{2}}\right)^\kappa.$$

To find ξ we solve $\ell + \lambda = \frac{2\xi}{1-2\xi} \cdot \log\tilde{q}$ for ξ and set $q = \tilde{q}^{\frac{2}{1-2\xi}}$.

4.3 Known attacks

Recovering a short $d \cdot g$. In this section, we described the attack of [Gar13, Section 7.3.3] except that we have no challenge elements and we are targeting GCDH instead of GDDH.

The attack consists in recovering a multiple of g of the form $d \cdot g$ and multiplying it by the zero-test parameter p_{zt}, obtaining a modified parameter $p'_{zt} = [d \cdot h \cdot z^\kappa]_q$. This parameter is then multiplied by two well chosen elements: v_0 a product of κ level-one encodings u_j, and v_1 a product of $\kappa - 1$ level-one encodings u_j multiplied by y. The attack then relies on the fact that the elements are small enough, such that $[p'_{zt} \cdot v_0]_q = p'_{zt} \cdot v_0$ and $[p'_{zt} \cdot v_1]_q = p'_{zt} \cdot v_1$.

To recover such a multiple of g, we first recover a basis for (g) by constructing elements of the following form: $v = \left[\left(\prod_{\ell=1}^{r} x_{i_\ell}^{(1)}\right) \cdot \left(\prod_{\ell=1}^{s} u_{j_\ell}\right) \cdot y^{\kappa-r-s} \cdot p_{zt}\right]_q$, for some r and s. Now assume we are sampling many v_i of the above form. All these elements share h but if we pick at least one v_i with $r = 1$ this is, with high probability, the only common factor. We hence get many elements in (h) from which we can compute a basis for (h). If we play the same game but this time all v_i have $r > 1$ we get many elements in $(h \cdot g)$. From bases for $(h \cdot g)$ and (h) we can compute a basis for (g). We note that it is typically simply assumed that a basis for (g) is public because an attacker can always run these steps.

Now, run lattice reduction on the public basis for (g). Lattice reduction produces a short element $d \cdot g$ where we know that d is also short because g^{-1} is short in $\mathbb{Q}[X]/(X^n + 1)$. In particular we have $d = d \cdot g \cdot g^{-1}$ so $\|d\| \leqslant \sqrt{n} \cdot \|d \cdot g\| \cdot \ell_{g^{-1}}$. However, this is an upper bound and indeed it is quite likely that $\|d\| < \|d \cdot g\|$. Indeed, based on experiments, we may expect $\|d\| \approx \|d \cdot g\|/\|g\|$.

Thus, as attackers we want to find an element of form $d \cdot g$ such that $\|p'_{zt} \cdot \prod_{j=0}^{\kappa-1} u_j\|_\infty < q$. We want each component of the product to be less than $q/2$ and hence the overall euclidean norm less than $q\sqrt{n}/2$. We then require:

$$\left\| d \cdot h \cdot \prod_{j=0}^{\kappa-1} (e_j \cdot a + \rho_{j,1}^{(1)} \cdot b_1^{(1)} + \rho_{j,2}^{(1)} \cdot b_2^{(1)}) \right\| \leqslant q\sqrt{n}/2 \quad \text{and} \tag{5}$$

$$\left\| d \cdot h \cdot a \cdot \prod_{j=0}^{\kappa-1} (e_j \cdot a + \rho_{j,1}^{(1)} \cdot b_1^{(1)} + \rho_{j,2}^{(1)} \cdot b_2^{(1)}) \right\| \leqslant q\sqrt{n}/2 \tag{6}$$

Assume, our lattice reduction manages to find an element $d \cdot g$ such that this inequality is satisfied. We still have to use this short element to compute w. For ease of exposure assume that $\mathcal{N}(\mathcal{I})$ is a prime. Then the Hermite Normal Form of g is a diagonal matrix with $\mathcal{N}(\mathcal{I})$ at index $(n-1) \times (n-1)$. In other words, reducing any element in R by $\mathcal{I}$ results in an integer mod $\mathcal{N}(\mathcal{I})$. We compute $\nu_0 = [v_0]_{\mathrm{HNF}(\mathcal{I})} \equiv \left(d \cdot h \cdot y \cdot \prod_{j=0}^{\kappa-1} e_j\right) \bmod \mathcal{I}$, and $\nu_1 = [v_1]_{\mathrm{HNF}(\mathcal{I})} \equiv \left(d \cdot h \cdot y \cdot \prod_{j=1}^{\kappa-1} e_j\right) \bmod \mathcal{I}$.

We can now compute $\eta = \nu_0 \cdot \nu_1^{-1}$ and observe $\eta \cdot \nu_1 = \nu_0 + \tau \cdot \mathcal{N}(\mathcal{I}) \equiv \nu_0 \bmod \mathcal{I}$. At the same time $e_0 \cdot \nu_1 \equiv e_0 \cdot v_1 \equiv v_0 \equiv \nu_0 \bmod \mathcal{I}$. Now, reducing η modulo $d \cdot g$ yields a short element which is functionally equivalent to e_0: a short representative of its coset. Here an element is short enough if it does not overflow mod q when we compute $p_{zt} \cdot w = p_{zt} \cdot \eta \cdot \prod_{j=1}^{\kappa} u_j$.

In this attack, we used the norm of $d \cdot g$ twice. Firstly, to produce v_0 and v_1 and, secondly, to produce a short representative of e_0. We now come back to the conditions of Eq. 5. Following the computations of Section 4.2 and using that $\sigma_1^\star > \sigma'$, we get:

$$n \cdot \sqrt{n}^{\kappa-1} \|d\| \cdot \left(3\sigma_1^\star \sigma' n^{\frac{3}{2}}\right)^\kappa \leqslant \sqrt{q}/2,$$

which gives

$$\|d\| \leqslant \frac{\sqrt{q}}{2 \cdot n^{2\kappa+1/2}(3\sigma_1^\star\sigma')^\kappa} \text{ and then } \|d \cdot g\| \leqslant \frac{\sigma\sqrt{q}}{2 \cdot n^{2\kappa}(3\sigma_1^\star\sigma')^\kappa}. \tag{7}$$

Assuming that $\|[\eta]_{d \cdot g}\| \approx \|d \cdot g\|$ we get that the same condition applies. We note that under our parameter choices this attack does not get easier as κ grows. To see why first observe that we

have

$$\frac{\sigma\sqrt{q}}{2n^{2\kappa}(3\sigma_1^\star\sigma')^\kappa} \approx \frac{\sigma(nl_{g^{-1}})^{1/(1-2\xi)}}{2} \cdot \frac{\left(n^{2\kappa}\left(3\sigma_1^\star\sigma'\right)^\kappa\right)^{1/(1-2\xi)}}{n^{2\kappa}(3\sigma_1^\star\sigma')^\kappa}.$$

First, note that the first multiplicand does not grow as κ grows because σ is defined independently of κ and ξ drops as κ grows. Second, note that that ξ is chosen such that the square of the numerator of the second multiplicand is $\log(8n\sigma) + \lambda$ bits larger than the denominator. Hence, regardless of choice for κ this fraction will be bounded by same magnitude in $O(2^\lambda \cdot \mathrm{poly}(\lambda))$.

Recovering $\tilde{b}_1^{(1)}, \tilde{b}_2^{(1)}$ from $x_1^{(1)}/x_2^{(1)}$ We can also mount the attack from [CS97] against GGHLite. We have

$$\left[\frac{x_1^{(1)}}{x_2^{(1)}}\right]_q = \left[\frac{b_1^{(1)}/z}{b_2^{(1)}/z}\right]_q = \left[\frac{\tilde{b}_1^{(1)} \cdot g}{\tilde{b}_2^{(1)} \cdot g}\right]_q = \left[\frac{\tilde{b}_1^{(1)}}{\tilde{b}_2^{(1)}}\right]_q$$

where we use that $b_i^{(1)} = \tilde{b}_i^{(1)} \cdot g$ with $\tilde{b}_i^{(1)}$ small. We set up the lattice $\Lambda = \begin{pmatrix} qI & 0 \\ X & I \end{pmatrix}$ where I is the $n \times n$ identity matrix, 0 is the $n \times n$ zero matrix, and X a rotational basis for $[x_1^{(1)}/x_2^{(1)}]_q$. By construction Λ contains the vector $(\tilde{b}_1^{(1)}, \tilde{b}_2^{(1)})$ which is short. We have $\det(\Lambda) = q^n$ and $\|(\tilde{b}_1^{(1)}, \tilde{b}_2^{(1)})\| \approx \sqrt{2n}\sigma'/\sigma$. In contrast, a random lattice with determinant q^n and dimension $2n$ is expected to have a shortest vector of norm $\approx q^{n/2n} = \sqrt{q}$ which is much longer than $\|(\tilde{b}_1^{(1)}, \tilde{b}_2^{(1)})\|$. While Λ does not constitute a Unique-SVP instance because there are many short elements of norm roughly $\sqrt{2n}\sigma'/\sigma$ we may consider all of these "interesting". Clearly, there is a gap between those "interesting" vectors and the expected length of short vectors for random lattices. To hedge against potential attacks exploiting this gap, we may hence want to ensure that finding those "interesting" short vectors is hard. We note that this attack does not use p_{zt} which means we would expect it to be less efficient that the previous attack. However, this is not the case. The hardness of Unique-SVP instances is determined by the ratio of the second shortest $\lambda_2(\Lambda)$ and the shortest vector $\lambda_1(\Lambda)$ of the lattice. Assuming that the complexity of finding a short element in Λ depends on the gap between $(\tilde{b}_1^{(1)}, \tilde{b}_2^{(1)})$ and $\sqrt{q}$ in a similar way, for a given n this attack becomes more efficient as we increase κ (and hence q) in contrast to the previous attack.

4.4 Lattice reduction

In order to succeed, an attacker needs to obtain $d \cdot g$ short enough to satisfy Equation 7 or to solve a Unique-SVP instance with gap $\lambda_2(\Lambda)/\lambda_1(\Lambda)$. We need to pick parameters such that solving either problem takes at least 2^λ operations.

The most efficient technique known in the literature to produce short lattice vectors is to run lattice reduction. The quality of lattice reduction is typically expressed as the root-Hermite factor δ_0. An algorithm with root-Hermite factor δ_0 is expected to output a vector v in a lattice L such that $\|v\| = \delta_0^n \,\mathrm{vol}(L)^{1/n}$.

Hence, in order to find a short enough $d \cdot g$, we need lattice reduction with root-Hermite factor δ_0 of at most:

$$\frac{\sigma\sqrt{q}}{2n^{2\kappa}\left(3\sigma_1^\star\sigma'\right)^\kappa} = \delta_0^n \cdot \mathrm{vol}(g)^{1/n} \leqslant \delta_0^n\sqrt{n}\sigma, \tag{8}$$

where the final inequality follows from $\|g\|_\infty \leqslant \sigma\sqrt{n}$. Similarly, for the NTRU-style attack, we require $\tau \cdot \delta_0^{2n} \leqslant \lambda_2(\Lambda)/\lambda_1(\Lambda)$ and thus

$$\delta_0 \leqslant \left(\frac{\sqrt{q}}{\sqrt{2n} \cdot \sigma'/\sigma \cdot \tau}\right)^{1/(2n)}, \tag{9}$$

where τ is a constant which depends on the lattice structure and on the reduction algorithm used. Typically $\tau \approx 0.3$ [GN08], which we will use as an approximation.

Currently, the most efficient algorithm for lattice reduction is a variant of the BKZ algorithm [SE94] referred to as BKZ 2.0 [CN11]. However, its running time and behaviour, especially in high dimensions, is not very well understood. Hence, there is no consensus in the literature as to how to relate a given δ_0 to computational cost.

We estimate the cost of lattice reduction as follows. First, we estimate the required BKZ block size required for a certain δ_0 by assuming that $\lim_{n\to\infty} \delta_0 = (\frac{k}{2\pi e}(\pi k)^{\frac{1}{k}})^{\frac{1}{2(k-1)}}$ [Che13] also approximates k for our finite n. Then the running time of BKZ is mainly determined by two factors: firstly, the time t_k it takes to find shortest or short enough vectors in lattices of dimension k, and secondly, the number of BKZ rounds needed ρ. If t_k is the number of clock cycles it takes to solve SVP in dimension k we expect BKZ to take $\rho \cdot n \cdot t_k$ clock cycles. No closed formula for the expected number of BKZ rounds is known. The best upper bound is exponential, but after $\rho \approx \frac{n^2}{k^2} \log n$ many rounds, the quality of the basis is already very close to the final output [HPS11]. For estimating t_k we take the minimum of $0.002898\,k^2 - 0.1227\,k + 23.83$ and $0.3774\,k + 20$. The first was extrapolated from data points made available in [CN11] and the second was extrapolated from data points made available in [BGJ13].

We stress, though, that these assumptions requires further scrutiny. They are assuming that the attacks in Section 4.3 are the most efficient attack and that our lattice reduction estimate are accurate.

4.5 Putting everything together

Our overall strategy is as follows. Pick an n and compute parameters σ, σ', $\sigma_1^\star$, ℓ_g and q. Now, establish the root-Hermite factor required to carry out the attack in Section 4.3 using Equations (8) and (9). If this δ_0 is small enough to satisfy security level λ terminate, otherwise double n and restart the procedure. For example parameters, see Table 2 as well as Tables 7 and 8 in Appendix B.

5 Implementation

Our implementation relies on FLINT [HJP14]. We use its data types to encode elements in $\mathbb{Z}[X]$, $\mathbb{Q}[X]$, and $\mathbb{Z}_q[X]$ but re-implement most non-trivial operations for the ring of integers of a Cyclotomic number field where the order is a power of two. Other operations – such as Gaussian sampling or taking approximate inverses – are not readily available in FLINT and are hence provided by our implementation. For computation with elements in $\mathbb{R}$ we use MPFR's `mpfr_t` [The13] with precision 2λ if not stated otherwise. Our implementation is available under the GPLv2+ license at `https://bitbucket.org/malb/gghlite-flint`. We give experimental results from running an N-partite Diffie-Hellman key exchange using our implementation in Table 2.

For many operations considered in this section straight-forward algorithms are available in $\mathcal{O}\left(n^3 \log q\right)$ bit operations. However, the smallest set of parameters we consider in Table 2 is $n = 2^{15}$ and $\log q \approx 3079$ which implies that if implemented naively each operation would take 2^{56} bit operations for the smallest set of parameters we consider. Even quadratic algorithms can be prohibitively expensive. Hence, in order to be feasible, all algorithms should run in quasi-linear time in n, or more precisely in $O(n \log n)$. All algorithms discussed in this section run in quasi-linear time.

5.1 Polynomial Multiplication in $\mathbb{Z}_q[X]/(X^n + 1)$

The most time-critical operation in the online phase of a GGH-style graded encoding scheme is the multiplication of polynomials in $\mathbb{Z}_q[X]/(X^n + 1)$. Asymptotically fast multiplication in this ring can be realised using the *Fast Fourier Transform* (FFT) over $\mathbb{Z}[X]$ and reducing the result modulo q which is the strategy implemented in FLINT, which has a highly optimised FFT implementation. The FFT is known to provide an $O(n \log n)$-time algorithm to compute the product of two polynomials of degree $< n$ using $2n$ evaluation points if a $2n$-th root of unity exists in the ring. Specialising to $\mathbb{Z}_q[X]/(X^n + 1)$ this can be reduced to n evaluation points and no modular reduction using the *Number-Theoretic Transform* (NTT) with a negative wrapped convolution:

λ	κ	δ_0	t_{en}	t_{sv}	n	$\log q$	ξ	$\log \sigma$	$\log \sigma_1^\star$	Setup	Publish	KeyGen	$\|$params$\|$
52	6	1.0159	61.8	73.4	2^{15}	3079	0.030	20.4	144.9	187s	1.59s	0.38s	48.1MB
52	9	1.0123	65.3	78.3	2^{16}	4729	0.020	21.4	150.2	841s	3.64s	1.28s	147.8MB
52	14	1.0191	64.9	76.5	2^{16}	7273	0.013	21.4	150.8	1124s	5.01s	2.55s	227.3MB
52	19	1.0134	68.0	79.5	2^{17}	10219	0.009	22.4	155.9	3783s	14.42s	9.66s	683.7MB
52	25	1.0177	68.0	79.5	2^{17}	13407	0.007	22.4	156.3	6524s	19.11s	16.70s	837.9MB
52	47	1.0192	71.1	82.6	2^{18}	26100	0.004	23.5	161.9	52183s	88.39s	141.35s	3.5GB
80	6	1.0093	294.6	173.7	2^{17}	3733	0.033	22.4	182.8	2532s	7.31s	1.75s	233.3MB
80	9	1.0071	121.2	121.9	2^{17}	5489	0.022	22.4	183.4	2644s	9.35s	3.21s	343.0MB
80	14	1.0057	196.2	150.7	2^{18}	8715	0.014	23.5	188.7	15132s	26.93s	13.21s	1.1GB
80	19	1.0077	108.7	117.5	2^{18}	11756	0.011	23.5	189.1	17158s	35.06s	22.91s	1.4GB
80	25	1.0052	242.8	165.9	2^{19}	15929	0.008	24.5	194.2	93561s	103.19s	85.68s	3.9GB
80	47	1.0098	101.5	83.8	2^{19}	29812	0.004	24.5	195.1	152052s	221.91s	343.94s	7.3GB

Table 2. Experimental results for running the one-round $(\kappa+1)$-partite Diffie-Hellman key exchange. The column λ gives the minimum security level we accepted. The column δ_0 gives the expected root-Hermite factor for the currently best known attack. The column t_{en} (resp. t_{sv}) gives the expected number of bit operations for running the best known attack with enumeration (resp. sieving) implementing the SVP oracle. The minimum of the two gives the expected security level ($\geqslant \lambda$) of a given row. Timings were produced on Intel Xeon CPU E5–2667 v2 3.30GHz with 256GB of RAM, all operations were parallelised on 16 cores, but not all operations took full advantage of all cores. Times for Publish and KeyGen are given per participant ("pp"). All times are wall times. See Section 3 for details on σ, $\sigma_1^\star$, and ξ.

Theorem 1 (Adapted from [Win96]). *Let ω_n be a nth root of unity in $\mathbb{Z}_q$ and $\varphi^2 = \omega_n$. Let $a = \sum_{i=0}^{n-1} a_i X^i$ and $b = \sum_{i=0}^{n-1} b_i X^i \in \mathbb{Z}_q[X]/(X^n+1)$. Let $c = a \cdot b \in \mathbb{Z}_q[X]/(X^n+1)$ and let $\overline{a} = (a_0, \varphi a_1, \ldots, \varphi^{n-1} a_{n-1})$ and define $\overline{b}$ and $\overline{c}$ analogously. Then $\overline{c} = 1/n \cdot NTT_{\omega_n}^{-1}(NTT_{\omega_n}(\overline{a}) \odot NTT_{\omega_n}(\overline{b}))$.*

This NTT with a negative wrapped convolution has been used in lattice-based cryptography before, cf. [LMPR08,PG12]. As noted above, FLINT does not take advantage of $q-1$ being a multiple of $2n$ in which case Theorem 1 applies because ω_n and φ exist. While both strategies have roughly the same asymptotic complexity and FLINT's FFT is very optimised, if we are doing many operations in $\mathbb{Z}_q[X]/(X^n+1)$ we can avoid repeated conversions between coefficient and "evaluation representations", $\big(f(1), f(\omega_n), \ldots, f(\omega_n^{n-1})\big)$, of our elements by relying on the negative wrapped convolution NTT instead of the FFT.This reduces the amortised cost from $O(n \log n)$ to $O(n)$. That is, we can convert encodings to their evaluation representation once on creation and back only when running extraction. We implemented this strategy. While our own NTT is slower than FLINT's optimised implementation over the integers mainly due to modular reductions, we observe a considerable overall speed-up with this strategy. We note that operations on elements in their evaluation representation are embarrassingly parallel.

5.2 Computing norms in $\mathbb{Z}[X]/(X^n+1)$

During instance generation we have to compute several norms of elements in $\mathbb{Z}[X]/(X^n+1)$. The norm $\mathcal{N}(f)$ of an element f in $\mathbb{Z}[X]/(X^n+1)$ is equal to the resultant $\mathrm{res}(f, X^n+1)$. The usual strategy for computing resultants over the integers is to use a multi-modular approach. That is, we compute resultants modulo many small primes q_i and then combine the results using the Chinese Remainder Theorem. Resultants modulo a prime q_i can be computed in $O(M(n)\log n)$ operations where $M(n)$ is the cost of one multiplication in $\mathbb{Z}_{q_i}[X]/(X^n+1)$. Hence, in our setting computing the norm costs $O(n \log^2 n)$ operations without specialisation.

However, we can observe that $\mathrm{res}(f, X^n+1) \bmod q_i$ can be rewritten as $\prod_{(X^n+1)(x)=0} f(x) \bmod q_i$ as X^n+1 is monic, i.e. as evaluating f on all roots of X^n+1. Picking q_i such that $q_i \equiv 1 \bmod 2n$ this can be accomplished using the NTT reducing the cost mod q_i to $O(M(n))$ saving a factor of $\log n$, e.g. a factor of 19 in our biggest parameter set (cf. Table 2).

5.3 Checking if (g) is a prime ideal

While we show in Section 4.1 that we do not require prime (g) for a non-interactive N-partite key exchange, some applications might still rely on this property. We hence provide an implementation for sampling such g.

To check whether the ideal generated by g is prime in $\mathbb{Z}[X]/(X^n+1)$ we compute the norm $\mathcal{N}(g)$ and check if it is prime which is a sufficient but not necessary condition. However, before computing full resultants, we first check if $\mathrm{res}(g, X^n+1) = 0 \bmod q_i$ for several "interesting" primes q_i. These primes are 2 and then all primes up to some bound with $q_i \equiv 1 \bmod n$ because these occur with good probability as factors. We list timings in Table 3.

n	$\log\sigma$	wall time	n	$\log\sigma$	wall time	n	$\log\sigma$	wall time
1024	15.1	0.54s	2048	16.2	3.03s	4096	17.3	20.99s

Table 3. Average time of checking primality of a single (g) on Intel Xeon CPU E5–2667 v2 3.30GHz with 256GB of RAM using 16 cores

5.4 Verifying that $(\boldsymbol{b_1^{(1)}, b_2^{(1)}}) = (\boldsymbol{g})$

When computing re-randomisation elements we require that $(b_1^{(1)}, b_2^{(1)}) = (g)$ holds, i.e. that our re-randomisers generate the whole ideal. If $b_i^{(1)} = \tilde{b}_i^{(1)} \cdot g$ for $0 < i \leqslant 2$ then this condition is equivalent to $\tilde{b}_1^{(1)} + \tilde{b}_2^{(1)} = R$. We check the sufficient but not necessary condition $\gcd(\mathrm{res}(\tilde{b}_1^{(1)}, X^n + 1), \mathrm{res}(\tilde{b}_2^{(1)}, X^n+1)) = 1$. This check which we have to perform for every candidate pair $(\tilde{b}_1^{(1)}, \tilde{b}_2^{(1)})$ involves computing two resultants and their gcd which is quite expensive. However, we observe that $\gcd(\mathrm{res}(\tilde{b}_1^{(1)}, X^n+1), \mathrm{res}(\tilde{b}_2^{(1)}, X^n+1)) \neq 1$ when $\mathrm{res}(\tilde{b}_1^{(1)}, X^n+1) = 0 = \mathrm{res}(\tilde{b}_2^{(1)}, X^n+1) \bmod q_i$ for any modulus q_i. Hence, we first check this condition for the same "interesting" primes as in the previous subsection and resample if it holds. Only if these tests pass, we compute two full resultants and their gcd. Indeed, after having ruled out small common prime factors it is quite unlikely that the gcd of the norms is not equal to one which means that with good probability we will perform this expensive step only once as a final verification. However, this step is still by far the most time consuming step during setup even with our optimisations applied. We note that a possible strategy for reducing setup time is to sample $m > 2$ re-randomisers $b_i^{(1)}$ and to apply standard bounds on the probability of m elements $\tilde{b}_i^{(1)}$ sharing a prime factor (after excluding small prime factors).

5.5 Computing the inverse of a polynomial modulo $\boldsymbol{X^n + 1}$

Instance generation relies on inversion in $\mathbb{Q}[X]/(X^n+1)$ in two places. Firstly, when sampling g we have to check that the norm of its inverse is bounded by ℓ_g. Secondly, to set up our discrete Gaussian samplers we need to run many inversions in an iterative process. We note that for computing the zero-testing parameter we only need to invert g in $\mathbb{Z}_q[X]/(X^n+1)$ which can be realised in n inversions in $\mathbb{Z}_q$ in the NTT representation.

In both cases where inversion in $\mathbb{Q}[X]/(X^n+1)$ is required approximate solutions are sufficient. In the first case we only need to estimate the size of g^{-1} and in the second case inversion is a subroutine of an approximation algorithm (see below). Hence, we implemented a variant of [BCMM98] to compute the approximate inverse of a polynomial in $\mathbb{Q}[X]/(X^n+1)$, with n a power of two.

The core idea is similar to the FFT, i.e. to reduce the inversion of f to the inversion of an element of degree $n/2$. Indeed, since n is even, $f(X)$ is invertible modulo X^n+1 if and only if $f(-X)$ is also invertible. By setting $F(X^2) = f(X)f(-X) \mod X^n+1$, the inverse $f^{-1}(X)$ of $f(X)$ satisfies

$$F(X^2)\, f^{-1}(X) = f(-X) \pmod{X^n+1}. \tag{10}$$

Let $f^{-1}(X) = g(X) = G_e(X^2) + XG_o(X^2)$ and $f(-X) = F_e(X^2) + XF_o(X^2)$ be split into their even and odd parts respectively. From Eq. 10, we obtain $F(X^2)(G_e(X^2) + XG_o(X^2)) = F_e(X^2) + XF_o(X^2) \pmod{X^n + 1}$ which is equivalent to

$$\begin{cases} F(X^2)G_e(X^2) = F_e(X^2) \pmod{X^n + 1} \\ F(X^2)G_o(X^2) = F_o(X^2) \pmod{X^n + 1}. \end{cases}$$

From this, inverting $f(X)$ can be done by inverting $F(X^2)$ and multiplying polynomials of degree $n/2$. It remains to recursively call the inversion of $F(Y)$ modulo $(X^{n/2} + 1)$ (by setting $Y = X^2$). This leads to Algorithm 1 for approximately inverting elements of $\mathbb{Q}[X]/(X^n + 1)$ when n is a power of 2, where we truncate the result of each recursive call to `prec` bits of precision.

Algorithm 1 Approximate inverse of $f(X) \mod X^n + 1$ using `prec` bits of precision

if $n = 1$ **then**
  $g_0 \leftarrow f_0^{-1}$
else
  $F(X^2) \leftarrow f(X)f(-X) \mod X^n + 1$
  $\tilde{F}(Y) = F(Y)$ truncated to `prec` bits of precision
  $G(Y) \leftarrow \text{InverseMod}(\tilde{F}(Y), q, n/2)$
  Set $F_e(X^2), F_o(X^2)$ such that $f(-X) = F_e(X^2) + XF_o(X^2)$
  $T_e(Y),\ T_o(Y) \leftarrow G(Y) \cdot F_e(Y),\ G(Y) \cdot F_o(Y)$
  $f^{-1}(X) \leftarrow T_e(X^2) + XT_o(X^2)$
  $\tilde{f}^{-1}(X) = f^{-1}(X)$ truncated to `prec` bits of precision
  return $\tilde{f}^{-1}(X)$
end if

Since Algorithm 1 reduces to polynomial multiplication and has $\log n$ iterations, it is easy to see that it can be performed in $O(n \log^2(n))$ operations in $\mathbb{Q}$. Yet, since we truncate out operands in each recursive call, it is not immediately obvious that this algorithm indeed produces an answer that is even close to $f^{-1}(X)$. Hence, we call Algorithm 1 in a loop, each time doubling the precision, until $\|\tilde{f}^{-1}(X) \cdot f(X) - 1\| < 2^{-\texttt{prec}}$ to ensure the accuracy of our result. We give experimental results comparing Algorithm 1 with FLINT's extended GCD algorithm in Table 4 which highlights that computing approximate inverses instead of exact inverses is necessary for anything but toy instances.

n	$\log \sigma$	xgcd	160	160iter	∞
4096	17.2	234.1s	0.067s	0.073s	121.8s
8192	18.3	1476.8s	0.195s	0.200s	755.8s

Table 4. Inverting $g \hookleftarrow D_{\mathbb{Z}^n,\sigma}$ with FLINT's extended Euclidean algorithm ("xgcd"), Algorithm 1 with precision 160 ("160"), iterating Algorithm 1 until $\|\tilde{f}^{-1}(X) \cdot f(X)\| < 2^{-160}$ ("160iter") and Algorithm 1 without truncation ("∞") on Intel Core i7–4850HQ CPU at 2.30GHz, single core.

5.6 Approximate Square Roots

During the Setup phase we need to sample from a discrete Gaussian $D_{(g),\sigma',c}$ with support (g). For this we need to compute an (approximate) square root in $\mathbb{Q}[X]/(X^n + 1)$. That is, for some input element Σ we want to compute some element $\sqrt{\Sigma}' \in \mathbb{Z}[X]/(X^n + 1)$ such that $\|\sqrt{\Sigma}' \cdot \sqrt{\Sigma}' - \Sigma\| < 2^{-2\lambda}$. We use iterative methods as suggested in [Duc13, Section 6.5] which iteratively refine the approximation of the square root similar to Newton's method. Computing approximate square roots of matrices is a well studied research area with many algorithms known in the literature (cf. [Hig97]). All algorithms with global convergence invoke approximate inversions in $\mathbb{Q}[X]/(X^n + 1)$ for which we call Algorithm 1.

We implemented the Babylonian method, the Denman-Beavers iteration [DJ76] and the Padé iteration [Hig97]. Although the Babylonian method only involves one inversion which allows us to compute with lower precision, we used Denman-Beavers, since it converges faster in practice and can be parallelised on two cores. While the Padé iteration can be parallelised on arbitrarily many cores, the workload on each core is much greater than in the Denman-Beavers iteration and in our experiments only improved on the latter when more than 8 cores were used.

Most algorithms have quadratic convergence but in practice this does not assure rapid convergence as error can take many iterations to become small enough for quadratic convergence to be observed. This effect can be mitigated, i.e. convergence improved, by scaling the operands appropriately in each loop iteration of the approximation [Hig97, Section 3]. A common scaling scheme is to scale by the determinant which in our case means computing $\mathrm{res}(f, X^n + 1)$ for some $f \in \mathbb{Q}[X]/(X^n + 1)$. Computing resultants in $\mathbb{Q}[X]/(X^n + 1)$ reduces to computing resultants in $\mathbb{Z}[X](X^n + 1)$. As discussed in Section 5.3 computing resultants in $\mathbb{Z}[X]/(X^n + 1)$ can be expensive. However, since we are only interested in an approximation of the determinant for scaling, we can compute with reduced precision. For this, we clear all but the most significant bit for each coefficient's numerator and denominator of f to produce f' and compute $\mathrm{res}(f', X^n + 1)$. The effect of clearing out the lower order bits of f is to reduce the size of the integer representation in order to speed up the resultant computation. With this optimisation scaling by an approximation of the determinant is both fast and precise enough to produce fast convergence. See Table 6 for example timings.

5.7 Sampling from a Discrete Gaussian

Our implementation needs to sample from discrete Gaussians over arbitrary integer lattices. For this, a fundamental building block is to sample from the integer lattice. We implemented a discrete Gaussian sampler over the integers both in arbitrary precision – using MPFR – and in double precision – using machine `doubles`. For both cases we implemented rejection sampling from a uniform distribution with and without table ("online") lookups [GPV08] and Ducas et al's sampler which samples from $D_{\mathbb{Z},k\sigma_2}$ where σ_2 is a constant [DDLL13, Algorithm 12]. Our implementation automatically chooses the best algorithm based on σ, c and τ (the tail cut). In our case σ is typically relatively large, so we call the latter whenever sampling with a centre $c \in \mathbb{Z}$ and the former when $c \notin \mathbb{Z}$. We list example timings of our discrete Gaussian sampler in Table 5. We note that in our implementation of GGHLite we – conservatively – only make use of the arbitrary precision implementation of this sampler with precision 2λ.

algorithm	σ	c	`double`		`mpfr_t`	
			prec	samp./s	prec	samp./s
tabulated [GPV08, Sample$\mathbb{Z}$]	10000	1.0	53	660.000	160	310.000
tabulated [GPV08, Sample$\mathbb{Z}$]	10000	0.5	53	650.000	160	260.000
online [GPV08, Sample$\mathbb{Z}$]	10000	1.0	53	414.000	160	9.000
online [GPV08, Sample$\mathbb{Z}$]	10000	0.5	53	414.000	160	9.000
[DDLL13, Algorithm 12]	10000	1.0	53	350.000	160	123.000

Table 5. Example timings for discrete Gaussian sampling over $\mathbb{Z}$ on Intel Core i7–4850HQ CPU at 2.30GHz, single core.

Using our discrete Gaussian sampler over the integers we implemented discrete Gaussian samplers over lattices. We implemented both [GPV08, SampleD] as well as a variant of [Pei10]. The former is not applicable for anything but toy instances as it requires to compute the Gram-Schmidt matrix of our lattice basis which costs $O(n^5)$ if computations are performed over $\mathbb{Q}$ and $O(n^3)$ if performed with fixed precision. Neither is feasible for large n. Instead, we utilise a variant of [Pei10] which we reproduce in Algorithm 2 . Namely, we first observe that $D_{(g),\sigma'} = g \cdot D_{R,\sigma' \cdot g^{-T}}$ and then use [Pei10, Algorithm 1] to sample from $D_{R,\sigma' \cdot g^{-T}}$ where g^{-T} is the conjugate of g^{-1}. That

Algorithm 2 Computing an approximate square root of $\sigma'^2 \cdot g^{-T} \cdot g^{-1} - r^2$.

$p,\ s' \leftarrow \log n + 4\ \log(\sqrt{n}\|\sigma\|)),\ 1$
$\Sigma_2' \leftarrow g^{-T} \cdot g^{-1}$
while $\|s'^2 - \Sigma_2'\| > 2^{-\lambda}$ **do**
  $s' \leftarrow\approx \sqrt{\Sigma_2'}$ computed at precision p until $\|s'^2 - \Sigma_2\| < 2^{-\lambda}$ or no more convergence
  $p \leftarrow 2p$
end while
$p,\ r \leftarrow p + 2\log\sigma',\ 2 \cdot \lceil\sqrt{\log n}\rceil$
$\Sigma_2 \leftarrow \sigma \cdot g^{-T} \cdot g^{-1} - r^2$
$s \leftarrow\approx \sqrt{\Sigma_2}$ computed at precision p using s' as initial approximation until $\|s^2 - \Sigma_2\| < 2^{-2\lambda}$
return s

Algorithm 3 Sampling from $D_{(g),\sigma'}$

$\sqrt{\Sigma_2}' \leftarrow\approx \sqrt{\sigma'^2 \cdot g^{-T} \cdot g^{-1} - r^2}$
$x' \in \mathbb{R}^n \hookleftarrow \rho_{1,0}$
$x \leftarrow x'$ considered as an element $\in \mathbb{Q}[X]/(X^n + 1)$
$y \leftarrow \sqrt{\Sigma_2}' \cdot x$
return $g \cdot (\lfloor y \rceil_r)$

is, $g_0^T = g_0$ and $g_{n-i}^T = -g_i$ for $1 \leqslant i < n$ for $\deg(g) = n - 1$. We then proceed as follows. We first compute an approximate square root of $\Sigma_2' = g^{-T} \cdot g^{-1}$ up to λ bits of precision using the Denman-Beavers iteration. Operations are performed with $\log(n) + 4\,(\log(\sqrt{n}\|\sigma\|))$ bits of precision. If the square root does not converge for this precision, we double it and start over. We then use this value, scaled appropriately, as the initial value from which to start the Babylonian method for computing a square-root of $\Sigma_2 = \sigma'^2 \cdot g^{-T} \cdot g^{-1} - r^2$ where $r = 2 \cdot \lceil\sqrt{\log n}\,\rceil$. We terminate when the square of the approximation is within distance $2^{-2\lambda}$ to Σ_2. This typically happens after just one iteration because our initial candidate is already very close to the target value which is also why we call the cheaper Babylonian method.

Given an approximation $\sqrt{\Sigma_2}'$ of $\sqrt{\Sigma_2}$ we then sample a vector $x \hookleftarrow \mathbb{R}^n$ from a standard normal distribution and interpret it as a polynomial in $\mathbb{Q}[X]/(X^n + 1)$. We then compute $y = \sqrt{\Sigma_2}' \cdot x$ in $\mathbb{Q}[X]/(X^n + 1)$ and return $g \cdot (\lfloor y \rceil_r)$, where $\lfloor y \rceil_r$ denotes sampling a vector in $\mathbb{Z}^n$ where the i-th component follows $D_{\mathbb{Z},r,y_i}$. The whole algorithm is summarised in Algorithm 3 and we give experimental results in Table 6 .

			square root			
prec	n	$\log\sigma'$	iterations	wall time	$\log\|(\sqrt{\Sigma_2}')^2 - \Sigma_2\|$	$D_{(g),\sigma'}/s$
160	1024	45.8	9	0.4s	-200	26.0
160	2048	49.6	9	0.9s	-221	12.0
160	4096	53.3	10	2.5s	-239	5.1
160	8192	57.0	10	8.6s	-253	2.0
160	16384	60.7	10	35.4s	-270	0.8

Table 6. Example timings for taking approximate square roots of $\Sigma_2 = \sigma'^2 \cdot g^{-T} \cdot g - r^2 \cdot I$ for discrete Gaussian sampling over g with parameter σ' on Intel Core i7–4850HQ CPU at 2.30GHz. Computing square roots uses 2 cores for Denman-Beavers and 4 cores for estimating the scaling factor, sampling uses one core.

Acknowledgement: We would like to thank Guilhem Castagnos, Guillaume Hanrot, Bill Hart, Claude-Pierre Jeannerod, Clément Pernet, Damien Stehlé, Gilles Villard and Martin Widmer for helpful discussions. We would like to thank Steven Galbraith for pointing out the NTRU-style attack to us and for helpful discussions. This work has been supported in part by ERC Starting Grant ERC-2013-StG-335086-LATTAC. The work of Albrecht was supported by EPSRC grant EP/L018543/1 "Multilinear Maps in Cryptography".

References

BCMM98. Dario Bini, Gianna M. Del Corso, Giovanni Manzini, and Luciano Margara. Inversion of circulant matrices over $\mathbb{Z}_m$. In *Proc. of ICALP 1998*, volume 1443 of *LNCS*, pages 719–730. Springer, 1998.

BF03. Dan Boneh and Matthew Franklin. Identity-based encryption from the Weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003.

BGJ13. Anja Becker, Nicolas Gama, and Antoine Joux. Solving shortest and closest vector problems: The decomposition approach. Cryptology ePrint Archive, Report 2013/685, 2013. `http://eprint.iacr.org/2013/685`.

BLR+14. Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. Cryptology ePrint Archive, Report 2014/834, 2014. `http://eprint.iacr.org/2014/834`.

BS03. Dan Boneh and Alice Silverberg. Applications of multilinear forms to cryptography. *Contemporary Mathematics*, 324:71–90, 2003.

BWZ14. Dan Boneh, Brent Waters, and Mark Zhandry. Low overhead broadcast encryption from multilinear maps. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 206–223. Springer, August 2014.

CDKD14. Maurice-Étienne Cloutier, Jean-Marie De Koninck, and Nicolas Doyon. On the powerful and squarefree parts of an integer. *Journal of Integer Sequences*, 17(2):28, 2014.

CG13. Ran Canetti and Juan A. Garay, editors. *CRYPTO 2013, Part I*, volume 8042 of *LNCS*. Springer, August 2013.

Che13. Yuanmi Chen. *Réduction de réseau et sécurité concrète du chiffrement complètement homomorphe*. PhD thesis, Paris 7, 2013.

CHL+14. Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. Cryptology ePrint Archive, Report 2014/906, 2014. `http://eprint.iacr.org/`.

CLT13. Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Practical multilinear maps over the integers. In Canetti and Garay [CG13], pages 476–493.

CN11. Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 1–20. Springer, December 2011.

CS97. Don Coppersmith and Adi Shamir. Lattice attacks on NTRU. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 52–61. Springer, May 1997.

DDLL13. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Canetti and Garay [CG13], pages 40–56.

DJ76. Eugene D. Denman and Alex N. Beavers Jr. The matrix sign function and computations in systems. *Applied Mathematics and Computation*, 2:63–94, 1976.

DN12. Léo Ducas and Phong Q. Nguyen. Faster gaussian lattice sampling using lazy floating-point arithmetic. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 415–432. Springer, December 2012.

Duc13. Léo Ducas. *Signatures Fondées sur les Réseaux Euclidiens: Attaques, Analyse et Optimisations* . PhD thesis, Université Paris Diderot, 2013.

Gar13. Sanjam Garg. *Candidate Multilinear Maps*. PhD thesis, University of California, Los Angeles, 2013.

GGH13a. Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 1–17. Springer, May 2013.

GGH+13b. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

GHMS14. Craig Gentry, Shai Halevi, Hemanta K. Maji, and Amit Sahai. Zeroizing without zeroes: Cryptanalyzing multilinear maps without encodings of zero. Cryptology ePrint Archive, Report 2014/929, 2014. `http://eprint.iacr.org/2014/929`.

GN08. Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 31–51. Springer, April 2008.

GPV08. Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 197–206. ACM Press, May 2008.

Hig97. Nicholas J. Higham. Stable iterations for the matrix square root. *Numerical Algorithms*, 15(2):227–242, 1997.

HJP14. William Hart, Fredrik Johansson, and Sebastian Pancratz. FLINT: Fast Library for Number Theory, 2014. Version 2.4.4, `http://flintlib.org`.

HPS11. Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Analyzing blockwise lattice algorithms using dynamical systems. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 447–464. Springer, August 2011.

Jou04. Antoine Joux. A one round protocol for tripartite Diffie-Hellman. *Journal of Cryptology*, 17(4):263–276, September 2004.

Lep14. Tancrède Lepoint. *Design and Implementation of Lattice-based Crypography*. PhD thesis, ENS Paris & Université du Luxembourg, 2014.

LMPR08. Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A modest proposal for FFT hashing. In Kaisa Nyberg, editor, *FSE 2008*, volume 5086 of *LNCS*, pages 54–72. Springer, February 2008.

LSS14a. Adeline Langlois, Damien Stehlé, and Ron Steinfeld. GGHLite: More efficient multilinear maps from ideal lattices. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 239–256. Springer, May 2014.

LSS14b. Adeline Langlois, Damien Stehlé, and Ron Steinfeld. GGHLite: More efficient multilinear maps from ideal lattices. Cryptology ePrint Archive, Report 2014/487, 2014. `http://eprint.iacr.org/2014/487`.

Pei10. Chris Peikert. An efficient and parallel gaussian sampler for lattices. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 80–97. Springer, August 2010.

PG12. Thomas Pöppelmann and Tim Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In *Proc. of Latincrypt 2012*, volume 7533 of *LNCS*, pages 139–158. Springer, 2012.

S+13. William Stein et al. *Sage Mathematics Software Version 6.2*. The Sage Development Team, 2013. Available at `http://www.sagemath.org`.

SE94. C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(1-3):181–199, 1994.

The13. The MPFR team. *GNU MPFR: The Multiple Precision Floating-Point Reliable Library*, 3.1.2 edition, 2013. `http://www.mpfr.org/`.

Win96. Franz Winkler. *Polynomial Algorithms in Computer Algebra*. Texts and Monographs in Symbolic Computation. Springer, 1996.

A Graded Encoding Scheme

$\mathsf{InstGen}(1^\lambda, 1^\kappa) \to (\mathsf{params}, p_{zt})$**.** This algorithm takes λ and κ as inputs and outputs $(\mathsf{params}, p_{zt})$, where params is a description of the graded encoding system as above, and p_{zt} is a zero-testing parameter at level κ.

$\mathsf{Samp}(\mathsf{params}) \to a$**.** The ring sampler algorithm takes as input the parameters params and outputs a level-0 encoding $a \in S_0^{(\alpha)}$ for a nearly uniform element $\alpha \in R$.

$\mathsf{Enc}_i(\mathsf{params}, a) \to u$**.** The encoding algorithm takes as inputs the parameters params, a level i and a level-0 encoding $a \in S_0^{(\alpha)}$ of an element $\alpha \in R$. It outputs the level-i encoding $u \in S_i^{(\alpha)}$ for α.

$\mathsf{Add}(\mathsf{params}, i, u_1, u_2) \to u$**.** The addition algorithm takes as inputs the parameters params, a level i, and two level-i encodings $u_1 \in S_i^{(\alpha_1)}$ and $u_2 \in S_i^{(\alpha_2)}$. It outputs a level-i encoding $u_1 + u_2 \in S_i^{(\alpha_1+\alpha_2)}$.

$\mathsf{Neg}(\mathsf{params}, i, u_1) \to u$**.** The negation algorithm takes as inputs the parameters params, a level i, and a level-i encoding $u_1 \in S_i^{(\alpha_1)}$. It outputs a level-i encoding $-u_1 \in S_i^{(-\alpha_1)}$.

$\mathsf{Mult}(\mathsf{params}, i_1, i_2, u_1, u_2) \to u$**.** The multiplication algorithm takes as inputs the parameters params, two levels i_1 and i_2 such that $i_1 + i_2 \leqslant \kappa$, and a level-$i_1$ (resp. i_2) encoding $u_1 \in S_{i_1}^{(\alpha_1)}$ and $u_2 \in S_{i_2}^{(\alpha_2)}$. It outputs a level-$(i_1 + i_2)$ encoding $u_1 \times u_2 \in S_{i_1+i_2}^{(\alpha_1 \cdot \alpha_2)}$.

$\mathsf{isZero}(\mathsf{params}, p_{zt}, u) \to \{0, 1\}$**.** The zero-test algorithm takes as inputs the parameters params, the zero-testing parameter p_{zt} and a level-κ encodings $u \in S_\kappa^{(\alpha)}$. It outputs 1 for every $u \in S_\kappa^{(0)}$, and 0 otherwise, except with negligible probability:

$$\Pr_{\alpha \in R}\left[\exists u \in S_\kappa^{(\alpha)} \text{ s.t. } \mathsf{isZero}(\mathsf{params}, p_{zt}, u) = 1\right] = \text{negligible}(\lambda).$$

$\mathsf{Ext}(\mathsf{params}, p_{zt}, u) \to s$. The extraction algorithm takes as inputs the parameters params, the zero-testing parameter p_{zt} and a level-κ encodings $u \in S_\kappa^{(\alpha)}$. It outputs s such that:

1. For a randomly chosen $a \leftarrow \mathsf{Samp}(\mathsf{params})$, and two encodings of a: $u_1 \leftarrow \mathsf{Enc}_\kappa(\mathsf{params}, a)$ and $u_2 \leftarrow \mathsf{Enc}_\kappa(\mathsf{params}, a)$ then

$$\Pr\left[\mathsf{Ext}(\mathsf{params}, p_{zt}, u_1) = \mathsf{Ext}(\mathsf{params}, p_{zt}, u_2)\right] \geqslant 1 - \text{negligible}(\lambda).$$

2. The distribution

$$\{\mathsf{Ext}(\mathsf{params}, p_{zt}, u) : a \leftarrow \mathsf{Samp}(\mathsf{params}), u \leftarrow \mathsf{Enc}_\kappa(\mathsf{params}, a)\}$$

is nearly uniform over $\{0,1\}^\lambda$.

B More Parameters

In this section we give additional parameter sets for $\sigma_i^\star = \text{Poly}(\lambda)$ in Table 7 and $\sigma_i^\star = \widetilde{\mathcal{O}}(2^\lambda\, \lambda\, n^{4.5} \kappa)$ in Table 8. These tables were produced using the Sage [S+13] module from Appendix D.

λ	κ	n	q	$\|\text{enc}\|$	$\|\text{params}\|$	δ_0	BKZ Enum	BKZ Sieve
52	2	2^{13}	$\approx 2^{865.4}$	$\approx 2^{22.8}$	$\approx 2^{24.8}$	1.017486	$\approx 2^{54.5}$	$\approx 2^{66.0}$
52	4	2^{14}	$\approx 2^{1661.0}$	$\approx 2^{24.7}$	$\approx 2^{26.7}$	1.017195	$\approx 2^{57.6}$	$\approx 2^{69.1}$
52	6	2^{15}	$\approx 2^{2549.0}$	$\approx 2^{26.3}$	$\approx 2^{28.3}$	1.013292	$\approx 2^{60.6}$	$\approx 2^{72.2}$
52	10	2^{15}	$\approx 2^{4137.6}$	$\approx 2^{27.0}$	$\approx 2^{29.0}$	1.021841	$\approx 2^{60.6}$	$\approx 2^{72.2}$
52	20	2^{17}	$\approx 2^{9033.9}$	$\approx 2^{30.1}$	$\approx 2^{32.1}$	1.011937	$\approx 2^{67.8}$	$\approx 2^{82.4}$
52	40	2^{18}	$\approx 2^{18832.2}$	$\approx 2^{32.2}$	$\approx 2^{34.2}$	1.012486	$\approx 2^{69.9}$	$\approx 2^{82.0}$
52	80	2^{18}	$\approx 2^{37555.3}$	$\approx 2^{33.2}$	$\approx 2^{35.2}$	1.025095	$\approx 2^{54.0}$	$\approx 2^{54.0}$
105	2	2^{15}	$\approx 2^{1070.1}$	$\approx 2^{25.1}$	$\approx 2^{27.1}$	1.005398	$\approx 2^{212.7}$	$\approx 2^{148.0}$
105	4	2^{16}	$\approx 2^{1955.9}$	$\approx 2^{26.9}$	$\approx 2^{28.9}$	1.005039	$\approx 2^{255.7}$	$\approx 2^{161.3}$
105	6	2^{16}	$\approx 2^{2793.9}$	$\approx 2^{27.4}$	$\approx 2^{29.4}$	1.007268	$\approx 2^{112.2}$	$\approx 2^{115.4}$
105	10	2^{17}	$\approx 2^{4703.1}$	$\approx 2^{29.2}$	$\approx 2^{31.2}$	1.006160	$\approx 2^{161.1}$	$\approx 2^{136.6}$
105	20	2^{18}	$\approx 2^{9591.9}$	$\approx 2^{31.2}$	$\approx 2^{33.2}$	1.006320	$\approx 2^{155.8}$	$\approx 2^{136.7}$
105	40	2^{19}	$\approx 2^{19836.8}$	$\approx 2^{33.3}$	$\approx 2^{35.3}$	1.006557	$\approx 2^{147.1}$	$\approx 2^{135.4}$
105	80	2^{20}	$\approx 2^{41264.3}$	$\approx 2^{35.3}$	$\approx 2^{37.3}$	1.006831	$\approx 2^{139.2}$	$\approx 2^{134.1}$
80	2	2^{15}	$\approx 2^{1020.3}$	$\approx 2^{25.0}$	$\approx 2^{27.0}$	1.005133	$\approx 2^{240.7}$	$\approx 2^{155.3}$
80	4	2^{16}	$\approx 2^{1906.3}$	$\approx 2^{26.9}$	$\approx 2^{28.9}$	1.004907	$\approx 2^{274.2}$	$\approx 2^{165.7}$
80	6	2^{16}	$\approx 2^{2744.4}$	$\approx 2^{27.4}$	$\approx 2^{29.4}$	1.007136	$\approx 2^{116.1}$	$\approx 2^{117.2}$
80	10	2^{17}	$\approx 2^{4654.0}$	$\approx 2^{29.2}$	$\approx 2^{31.2}$	1.006094	$\approx 2^{165.4}$	$\approx 2^{138.0}$
80	20	2^{18}	$\approx 2^{9543.7}$	$\approx 2^{31.2}$	$\approx 2^{33.2}$	1.006288	$\approx 2^{156.8}$	$\approx 2^{137.1}$
80	40	2^{19}	$\approx 2^{19790.3}$	$\approx 2^{33.3}$	$\approx 2^{35.3}$	1.006541	$\approx 2^{148.0}$	$\approx 2^{135.8}$
80	80	2^{20}	$\approx 2^{41221.3}$	$\approx 2^{35.3}$	$\approx 2^{37.3}$	1.006824	$\approx 2^{139.2}$	$\approx 2^{134.1}$
161	2	2^{16}	$\approx 2^{1231.3}$	$\approx 2^{26.3}$	$\approx 2^{28.3}$	1.003115	$\approx 2^{915.2}$	$\approx 2^{266.1}$
161	4	2^{17}	$\approx 2^{2162.1}$	$\approx 2^{28.1}$	$\approx 2^{30.1}$	1.002785	$\approx 2^{1240.2}$	$\approx 2^{304.2}$
161	6	2^{17}	$\approx 2^{3045.1}$	$\approx 2^{28.6}$	$\approx 2^{30.6}$	1.003957	$\approx 2^{483.8}$	$\approx 2^{209.0}$
161	10	2^{18}	$\approx 2^{5044.2}$	$\approx 2^{30.3}$	$\approx 2^{32.3}$	1.003299	$\approx 2^{791.1}$	$\approx 2^{256.2}$
161	20	2^{19}	$\approx 2^{10157.4}$	$\approx 2^{32.3}$	$\approx 2^{34.3}$	1.003342	$\approx 2^{768.3}$	$\approx 2^{255.9}$
161	40	2^{20}	$\approx 2^{20850.6}$	$\approx 2^{34.3}$	$\approx 2^{36.3}$	1.003441	$\approx 2^{712.8}$	$\approx 2^{251.2}$
161	80	2^{21}	$\approx 2^{43173.8}$	$\approx 2^{36.4}$	$\approx 2^{38.4}$	1.003568	$\approx 2^{649.4}$	$\approx 2^{244.9}$

Table 7. Parameters for $\sigma' = \text{Poly}(n)$

λ	κ	n	q	$\|\text{enc}\|$	$\|\text{params}\|$	δ_0	BKZ Enum	BKZ Sieve
52	2	2^{13}	$\approx 2^{1046.2}$	$\approx 2^{23.0}$	$\approx 2^{25.0}$	1.021385	$\approx 2^{54.5}$	$\approx 2^{66.0}$
52	4	2^{14}	$\approx 2^{2018.6}$	$\approx 2^{25.0}$	$\approx 2^{27.0}$	1.021050	$\approx 2^{57.6}$	$\approx 2^{69.1}$
52	6	2^{15}	$\approx 2^{3076.9}$	$\approx 2^{26.6}$	$\approx 2^{28.6}$	1.016125	$\approx 2^{60.6}$	$\approx 2^{72.2}$
52	10	2^{16}	$\approx 2^{5236.0}$	$\approx 2^{28.4}$	$\approx 2^{30.4}$	1.013793	$\approx 2^{63.7}$	$\approx 2^{75.2}$
52	20	2^{17}	$\approx 2^{10748.4}$	$\approx 2^{30.4}$	$\approx 2^{32.4}$	1.014233	$\approx 2^{66.8}$	$\approx 2^{78.3}$
52	40	2^{18}	$\approx 2^{22221.1}$	$\approx 2^{32.4}$	$\approx 2^{34.4}$	1.014756	$\approx 2^{69.8}$	$\approx 2^{81.3}$
52	80	2^{18}	$\approx 2^{44413.1}$	$\approx 2^{33.4}$	$\approx 2^{35.4}$	1.029752	$\approx 2^{54.0}$	$\approx 2^{54.0}$
105	2	2^{16}	$\approx 2^{1504.4}$	$\approx 2^{26.6}$	$\approx 2^{28.6}$	1.003839	$\approx 2^{521.4}$	$\approx 2^{212.6}$
105	4	2^{16}	$\approx 2^{2729.7}$	$\approx 2^{27.4}$	$\approx 2^{29.4}$	1.007097	$\approx 2^{117.7}$	$\approx 2^{118.0}$
105	6	2^{17}	$\approx 2^{4086.0}$	$\approx 2^{29.0}$	$\approx 2^{31.0}$	1.005339	$\approx 2^{224.2}$	$\approx 2^{155.5}$
105	10	2^{18}	$\approx 2^{6841.0}$	$\approx 2^{30.7}$	$\approx 2^{32.7}$	1.004492	$\approx 2^{349.9}$	$\approx 2^{186.9}$
105	20	2^{19}	$\approx 2^{13843.0}$	$\approx 2^{32.8}$	$\approx 2^{34.8}$	1.004564	$\approx 2^{338.6}$	$\approx 2^{187.0}$
105	40	2^{20}	$\approx 2^{28293.7}$	$\approx 2^{34.8}$	$\approx 2^{36.8}$	1.004676	$\approx 2^{322.5}$	$\approx 2^{186.0}$
105	80	2^{21}	$\approx 2^{58092.0}$	$\approx 2^{36.8}$	$\approx 2^{38.8}$	1.004806	$\approx 2^{303.9}$	$\approx 2^{184.3}$
80	2	2^{15}	$\approx 2^{1307.5}$	$\approx 2^{25.4}$	$\approx 2^{27.4}$	1.006661	$\approx 2^{130.2}$	$\approx 2^{121.4}$
80	4	2^{16}	$\approx 2^{2476.9}$	$\approx 2^{27.3}$	$\approx 2^{29.3}$	1.006424	$\approx 2^{143.7}$	$\approx 2^{128.4}$
80	6	2^{17}	$\approx 2^{3731.8}$	$\approx 2^{28.9}$	$\approx 2^{30.9}$	1.004869	$\approx 2^{282.0}$	$\approx 2^{169.9}$
80	10	2^{17}	$\approx 2^{6073.7}$	$\approx 2^{29.6}$	$\approx 2^{31.6}$	1.007985	$\approx 2^{98.5}$	$\approx 2^{109.9}$
80	20	2^{18}	$\approx 2^{12363.0}$	$\approx 2^{31.6}$	$\approx 2^{33.6}$	1.008165	$\approx 2^{98.5}$	$\approx 2^{111.2}$
80	40	2^{19}	$\approx 2^{25388.9}$	$\approx 2^{33.6}$	$\approx 2^{35.6}$	1.008405	$\approx 2^{97.5}$	$\approx 2^{111.8}$
80	80	2^{20}	$\approx 2^{52338.5}$	$\approx 2^{35.7}$	$\approx 2^{37.7}$	1.008676	$\approx 2^{96.8}$	$\approx 2^{112.4}$
161	2	2^{16}	$\approx 2^{1842.6}$	$\approx 2^{26.8}$	$\approx 2^{28.8}$	1.004737	$\approx 2^{300.2}$	$\approx 2^{171.6}$
161	4	2^{17}	$\approx 2^{3380.8}$	$\approx 2^{28.7}$	$\approx 2^{30.7}$	1.004402	$\approx 2^{365.3}$	$\approx 2^{187.5}$
161	6	2^{18}	$\approx 2^{5004.4}$	$\approx 2^{30.3}$	$\approx 2^{32.3}$	1.003273	$\approx 2^{808.7}$	$\approx 2^{258.4}$
161	10	2^{19}	$\approx 2^{8294.1}$	$\approx 2^{32.0}$	$\approx 2^{34.0}$	1.002724	$\approx 2^{1325.3}$	$\approx 2^{318.1}$
161	20	2^{19}	$\approx 2^{16217.1}$	$\approx 2^{33.0}$	$\approx 2^{35.0}$	1.005353	$\approx 2^{229.0}$	$\approx 2^{161.3}$
161	40	2^{20}	$\approx 2^{32930.0}$	$\approx 2^{35.0}$	$\approx 2^{37.0}$	1.005446	$\approx 2^{222.7}$	$\approx 2^{161.8}$
161	80	2^{21}	$\approx 2^{67252.5}$	$\approx 2^{37.0}$	$\approx 2^{39.0}$	1.005567	$\approx 2^{215.3}$	$\approx 2^{161.9}$

Table 8. Parameters for $\sigma_i^\star = \widetilde{\mathcal{O}}(2^\lambda \, \lambda \, n^{4.5} \kappa)$

C Comparison with the CLT construction

In Table 9 we compare our results (cf. Table 2) with those reported in [CLT13].

	λ	κ	Setup	Publish (pp)	KeyGen (pp)	pk size
[Lep14]	52	6	7s	0.2s	0.2s	26.0MB
this work	52	6	187s	1.6s	0.4s	48.1MB
[Lep14]	52	25	10s	0.5s	2.2s	72.0MB
this work	52	25	6524s	19.1s	16.7s	837.9MB
[Lep14]	80	6	27295s	17.8s	20.2s	2.6GB
this work	80	6	2644s	7.3s	1.7s	233.0MB
[Lep14]	80	25	123633s	59.1s	254.5s	8.3GB
this work	80	25	93561s	103.2s	85.7s	3.9GB

Table 9. Comparison with timings reported in [CLT13]. Our experiments were run on Intel Xeon CPU E5–2667 v2 3.30GHz with 256GB of RAM using 16 cores. The experiments in [CLT13] were run on Intel Xeon E7-8837 2.67GHz, utilising 16 cores during the Setup phase and 1 core otherwise. All times are wall times.

D Parameter Estimation Source Code

```
# -*- coding: utf-8 -*-

from collections import OrderedDict
from copy import copy

from sage.calculus.var import var
from sage.functions.log import log
from sage.functions.other import sqrt
from sage.misc.misc import get_verbose
from sage.rings.all import ZZ, RR, RealField
from sage.symbolic.all import pi, e

# Utility Functions


def params_str(d, keyword_width=None):
    """
    Return string of key,value pairs as a string "key0: value0, key1: value1"

    :param d:        report dictionary
    :keyword_width:  keys are printed with this width
    """
    if d is None:
        return
    s = []
    for k in d:
        v = d[k]
        if keyword_width:
            fmt = u"%%%ds" % keyword_width
            k = fmt % k
        if ZZ(1)/2048 < v < 2048 or v == 0:
            try:
                s.append(u"%s: %9d" % (k, ZZ(v)))
            except TypeError:
                if v < 2.0 and v >= 0.0:
                    s.append(u"%s: %9.7f" % (k, v))
                else:
                    s.append(u"%s: %9.4f" % (k, v))
        else:
            t = u"≈2^%.1f" % log(v, 2).n()
            s.append(u"%s: %9s" % (k, t))
    return u",  ".join(s)


def params_reorder(d, ordering):
    """
```

```
    Return a new ordered dict from the key:value pairs in ‘d‘ but reordered such that the
    keys in ordering come first.

    :param d:        input dictionary
    :param ordering: keys which should come first (in order)

    """
    keys = list(d)
    for key in ordering:
        keys.pop(keys.index(key))
    keys = list(ordering) + keys
    r = OrderedDict()
    for key in keys:
        r[key] = d[key]
    return r


# Lattice Reduction Estimates


def k_chen(delta):
    """
    Estimate required blocksize ‘k‘ for a given root-hermite factor δ_0.

    :param delta: root-hermite factor δ_0
    """
    k = ZZ(40)
    RR = delta.parent()
    pi_r = RR(pi)
    e_r = RR(e)

    f = lambda k: (k/(2*pi_r*e_r) * (pi_r*k)**(1/k))**(1/(2*(k-1)))

    while f(2*k) > delta:
        k *= 2
    while f(k+10) > delta:
        k += 10
    while True:
        if f(k) < delta:
            break
        k += 1

    return k


def bkz_runtime_k_sieve(k, n):
    """
    Runtime estimation given ‘k‘ and assuming sieving is used to realise the SVP oracle.

    :param k: block size
    :param n: lattice dimension
    :returns: log to base two of the expected runtime

    """
    return RR(0.3774*k + 20  + 3*log(n, 2) - 2*log(k, 2) + log(log(n, 2, 2)))


def bkz_runtime_k_bkz2(k, n):
    """
    Runtime estimation given ‘k‘ and assuming BKZ 2.0 estimates are correct.

    :param k: block size
    :param n: lattice dimension
    :returns: log to base two of the expected runtime

    """
    repeat = 3*log(n, 2) - 2*log(k, 2) + log(log(n, 2, 2))
    return RR(0.002897773577138052*k**2 - 0.12266248055336093*k + 23.831116173986075 + repeat)


def complete_lattice_attack(d):
    """
    Fill in missing pieces for lattice attack estimates

    :param d: a cost estimate for lattice attacks
    :returns: a cost estimate for lattice attacks

    """
    r = copy(d)
    if r[u"δ_0"] >= 1.0219:
```

```
            r["k"] = 2
            r["bkz2"]  = r["n"]**3
            r["sieve"] = r["n"]**3
        else:
            r["k"] = k_chen(r[u"δ_0"])
            r["bkz2"]  = ZZ(2)**bkz_runtime_k_bkz2(r["k"], r["n"])
            r["sieve"] = ZZ(2)**bkz_runtime_k_sieve(r["k"], r["n"])
        r = params_reorder(r, [u"δ_0", "k", "bkz2", "sieve"])
        return r


def gghlite_params(n, kappa, target_lambda=80, xi=None, rerand=True, gddh_hard=False):
    """
    Return GGHLite parameter estimates for a given dimension `n` and
    multilinearity level `κ`.

    :param n:             lattice dimension, must be power of two
    :param kappa:         multilinearity level `κ>1`
    :param target_lambda: target security level
    :param xi:            pick `ξ` manually
    :param rerand:        is the instance supposed to support re-randomisation
                          This should be true for `N`-partite DH key
                          exchange and false for iO and friends.
    :param gddh_hard:     should the GDDH problem be hard
    :returns:             parameter choices for a GGHLite-like graded-encoding scheme
    """
    n = ZZ(n)
    kappa = ZZ(kappa)
    RR = RealField(2*target_lambda)
    sigma   = RR(4*pi*n * sqrt(e*log(8*n)/pi))
    ell_g = RR(4*sqrt(pi*e*n)/(sigma))
    sigma_p = RR(7 * n**(2.5) * log(n)**(1.5) * sigma)
    ell_b = RR(1.0/(2.0*sqrt(pi*e*n)) * sigma_p)
    eps = RR(log(target_lambda)/kappa)
    ell = RR(log(8*n*sigma, 2))
    m = RR(2)

    if rerand:
        sigma_s = RR(n**(1.5) * sigma_p**2 * sqrt(8*pi/eps)/ell_b)
        if gddh_hard:
            sigma_s *= 2**target_lambda * sqrt(kappa) * target_lambda / n
        normk = ((sigma_p)**2 * n**RR(1.5) + 2*sigma_s * sigma_p * n**RR(1.5))**kappa
    else:
        normk = ((sigma_p) * sqrt(n))**kappa
    normk = sqrt(n)**(kappa-1) * normk
    q_base = RR(n * ell_g * normk)

    if xi is None:
        xivar = var('xivar')
        f = (ell + target_lambda) == (2*xivar/(1-2*xivar))*log(q_base, 2)
        xi = RR(f.solve(xivar)[0].rhs())
        q = q_base**(ZZ(2)/(1-2*xi))
        t = q**xi * 2**(-ell + 2)
        assert(q > 2*t*n*sigma**(1/xi))
        assert(abs(xi*log(q, 2) - target_lambda - ell) <= 0.1)
    else:
        q = q_base**(ZZ(2)/(1-2*xi))
        t = q**xi * 2**(-ell + 2)

    params = OrderedDict()
    params[u"κ"] = kappa
    params["n"] = n
    params[u"σ"] = sigma
    params[u"σ'"] = sigma_p
    if rerand:
        params[u"σ^*"] = sigma_s
    params[u"lnorm_κ"] = normk
    params[u"unorm_κ"] = normk  # if we had re-rand at higher levels this could be bigger
    params[u"ℓ_g"] = ell_g
    params[u"ℓ_b"] = ell_b
    params[u"ϵ"] = eps
    params[u"m"] = m
    params[u"ξ"] = xi
    params["q"] = q
    params["|enc|"] = RR(log(q, 2) * n)
    if rerand:
        params["|par|"] = (2 + 1 + 1)*RR(log(q, 2) * n)
    else:
        params["|par|"] = RR(log(q, 2) * n)
```

```
    return params


def gghlite_attacks(params):
    """
    Given parameters for a GGHLite-like problem instance estimate how
    long two lattice attacks would take.

    The two attacks are:

    - finding a short multiple of ‘g‘.
    - finding short ‘b_0/b_1‘ from ‘x_0/x_1‘

    :param params: parameters for a GGHLite-like graded encoding scheme
    :returns: cost estimate for lattice attacks

    """
    n = params["n"]
    q = params["q"]
    sigma = params[u"σ"]
    xi = params[u"ξ"]

    # small generator attack
    sg = OrderedDict()
    sg["n"] = n
    normk = params[u"lnorm_κ"]
    sg["norm"] = RR(sqrt(q)/(2*n*normk))
    assert(sg["norm"] < q**xi)
    sg[u"δ_0"] = sg["norm"]**(1/n)
    sg = complete_lattice_attack(sg)

    # NTRU attack
    nt = OrderedDict()
    nt["n"] = n
    nt[u"τ"] = RR(0.2)
    nt[u"δ_0"] = RR((sqrt(q)/(sqrt(ZZ(2)*n)* sigma * nt[u"τ"]))**(1/(2*n)))
    nt = complete_lattice_attack(nt)

    return sg, nt


def gghlite_brief(l, kappa, **kwds):
    """

    Return parameter choics for a GGHLite-like graded encoding scheme
    instance with security level at least ‘λ‘ and multilinearity level
    ‘κ‘

    :param l:     security parameter ‘λ‘
    :param kappa: multilinearity level ‘k‘
    :returns:     parameter choices for a GGHLite-like graded-encoding scheme

    .. note::

       ‘‘lambda‘‘ is a reserved key word in Python.
    """
    n = 1024
    while True:
        params = gghlite_params(n, kappa, target_lambda=l, **kwds)
        sg, nt = gghlite_attacks(params)
        best = max(sg, nt, key=lambda d: d[u"δ_0"])

        current = OrderedDict()
        current[u"λ"] = l
        current[u"κ"] = kappa
        current["n"] = n
        current["q"] = params["q"]
        current["|enc|"] = params["|enc|"]
        current["|par|"] = params["|par|"]

        current[u"δ_0"]   = best[u"δ_0"]
        current[u"bkz2"]  = best[u"bkz2"]
        current[u"sieve"] = best[u"sieve"]

        if get_verbose() >= 1:
            print params_str(current)

        if best["bkz2"] >= ZZ(2)**l and best["sieve"] >= ZZ(2)**l:
            break
        n = 2*n
```

```
    return current


def gghlite_latex_table(L, K, **kwds):
    """
    Generate a table with parameter estimates for `λ ∈ L` and `κ ∈ K`.

    :param L: a list of `λ`
    :param K: a list of `κ`
    :returns: a string, ready to be pasted into TeX

    """
    ret = []
    for l in L:
        for k in K:
            line = []
            current = gghlite_brief(l, k, **kwds)
            line.append("%3d" % current[u"λ"])
            line.append("%3d" % current[u"κ"])
            line.append("$2^{%2d}$" % log(current["n"], 2))
            t = u"$≈2^{%7.1f}$" % log(current["q"], 2).n()
            line.append(u"%9s" % (t,))
            t = u"$≈2^{%4.1f}$" % log(current["|enc|"], 2).n()
            line.append(u"%9s" % (t,))
            t = u"$≈2^{%4.1f}$" % log(current["|par|"], 2).n()
            line.append(u"%9s" % (t,))
            line.append("%8.6f" % current[u"δ_0"])
            t = u"$≈2^{%5.1f}$" % log(current[u"bkz2"], 2)
            line.append(u"%9s" % (t,))
            t = u"$≈2^{%5.1f}$" % log(current[u"sieve"], 2)
            line.append(u"%9s" % (t,))
            ret.append(u" & ".join(line) + "\\\\")
        ret.append(r"\midrule")

    header = []
    header.append(r"\begin{tabular*}{0.75\textwidth}{@{\extracolsep{\fill}} "
                  + ("r" * 9) + "}")
    header.append(r"\toprule")
    line = u"$λ$ & $κ$ & $n$ & $q$ & \\encs & \\pars & $δ_0$ & BKZ Enum & BKZ Sieve\\\\"
    header.append(line)
    header.append(r"\midrule")

    ret = header + ret
    ret.append(r"\end{tabular*}")

    return "\n".join(ret)
```