Improved Parameters and an Implementation of Graded Encoding Schemes from Ideal Lattices

Martin R. Albrecht[1], Catalin Cocis[2], Fabien Laguillaumie[3], and 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, Switzerland

Abstract. We discuss how to set parameters for GGH-like graded encoding schemes approximating cryptographic multilinear maps from ideal lattices and propose a strategy which reduces parameter sizes for concrete instances. Secondly, we discuss a first software implementation of a graded encoding scheme based on GGHLite, an improved variant of Garg, Gentry and Halevi's construction (GGH) due to Langlois, Stehlé and Steinfeld. Thirdly, we provide an implementation of non-interactive N-partite Diffie-Hellman key exchange. We discuss our implementation strategies and show that our implementation outperforms previous work.

1 Introduction

Multilinear maps, starting with bilinear ones, are crucial 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.

Graded encoding schemes quickly found many applications in cryptography. Already in [GGH13a] the authors showed how to adapt the generalisation of the Diffie-Hellman key exchange first constructed with cryptographic bilinear maps [BS03]: the protocol allows N users to share a secret key with only one broadcast message each. 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 new constructions based on graded encoding schemes that go beyond what was previously considered feasible: they proposed indistinguishability obfuscation (iO) and functional encryption based on multilinear maps and some additional assumptions.

The principle of the GGH scheme is as follows: 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 $a \hookleftarrow D_{1+\mathcal{I},\sigma'}$ (discrete Gaussian of support $1 + \mathcal{I}$ and parameter σ'). We are also given m level-i encodings of 0 named $x_j^{(i)}$, for all $1 \leq i \leq \kappa$, and a zero-testing parameter p_{zt}. To encode an element of $R/\mathcal{I}$ at level-i, 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 $\sum_i \rho_j x_j^{(i)}$ to it where the ρ_j are sampled from a discrete Gaussian with parameter $\sigma^\star$. 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}. Hence, anyone can encode at a superior level, given

a level-0 encoding for example, 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. 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 the level-κ encoding of this element.

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 each ith party chooses a (secret) random level-0 encoding e_i, encodes it at level-1 and publish this level-1 encoding. Then, in the **KeyGen** phase, each party multiplies her 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.

However, instantiating and running this protocol based on the GGH construction is too costly in practice for anything but toy instances. Coron, Lepoint and Tibouchi [CLT13] introduced an alternative instantiation of a graded encoding scheme over the Integers. They also discuss a C++ implementation of a heuristic variant of this scheme, which was the only known implementation of graded encoding schemes and of the multipartite Diffie-Hellman key exchange. They report that the **Setup** phase ran in 27295s (parallelised on 16 cores), the **Publish** phase in 17.8s per party (single core) and the **KeyGen** phase in 20.2s per party (single core) for a level of security $\lambda = 80$ and $\kappa = 6$ (i.e. $N = 7$). The the choices of $\lambda = 80$ and $\kappa = 6$ are the largest set of parameters reported on. However, very recently [CHL+14] demonstrated that the CLT construction can be broken in polynomial time if encodings of zero are published.

In 2014, Langlois, Stehlé and Steinfeld [LSS14b] 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 $\sigma^\star$ used to sample multipliers ρ_j during the re-randomisation phase from $\widetilde{\mathcal{O}}(2^\lambda \lambda n^{4.5} \kappa)$ to $\widetilde{\mathcal{O}}(n^{5.5}\sqrt{\kappa})$ for a security parameter λ. These improvements allow to reduce the size of the public parameters and to improve the overall efficiency of the scheme. Indeed, with these improvements an implementation of GGH-style graded encoding schemes from ideal lattices is feasible.

Our contribution. In this work, we first discuss how to choose practical parameters for the GGHLite scheme. We rely on the analysis of [LSS14b] to ensure the correctness of all the procedures of the scheme and on the analysis in [Gar13,CS97] to thwart the best known attacks. Our refined analysis allows us to reduce the size of the parameters q and ℓ further compared to [LSS14b]. Due to our modifications recovering a short multiple of g is typically less efficient than attacking level-1 encodings of zero using a lattice as in [CS97].

Our main contribution is an implementation of a variant of the GGHLite scheme and of the N-partite Diffie-Hellman key exchange. Our implementation is in C99 and is using the free software libraries FLINT [HJP14], MPFR [The13] and GMP [Gt14]. We also make use of OpenMP to parallelise our computations. Our implementation is made available under the GPLv2+ license.

We discuss our implementation choices and also compare our implementation with results reported in [CLT13]. While [CLT13] has since been shown to be susceptible to attacks if encodings of zero are published, the timings reported in [CLT13] are still the only reference to compare our results to. Furthermore, some applications such as iO do not require public encodings of zero. For those applications the CLT construction can still be used which gives another motivation to

compare with it. Our results are encouraging, as all the phases of the scheme are considerably more efficient than in the implementation of the graded encoding scheme over the Integers at security level $\lambda = 80$:

	λ	κ	Setup	Publish (pp)	KeyGen (pp)	pk size
[CLT13]	52	6	7s	0.18s	0.2s	26.0MB
This work	52	6	549s	3.66s	1.7s	139.5MB
[CLT13]	80	6	27295s	17.8s	20.2s	2.6GB
This work	80	6	2273s	9.0s	4.3s	300MB

Technical overview. Our implementation relies on FLINT [HJP14] but we provide our own specialised implementations for operations in the ring of integers of Cyclotomic number fields of order 2^k such multiplication in $\mathbb{Z}_q[X]/(X^n+1)$, taking approximate square roots and inverses in $\mathbb{Q}[X]/(X^n+1)$ and for discrete Gaussian samplers over the Integers and over arbitrary lattices. For the latter we use Ducas and Nguyen's framework [DN12]. Our implementation of these operations might be of independent interest.

Our variant of GGHLite foregos checking if g generates a prime ideal. That is, during instance generation [LSS14b] specifies to sample g such that (g) is a prime ideal. This condition is needed in [GGH13a] to ensure that no non-zero encoding passes the zero-testing test. Hence, we first discuss how to speed up checking if $(g) \subset \mathbb{Q}[X]/(X^n+1)$ is prime. However, rejection sampling until an element is found which generates a prime ideal is still prohibitively expensive with these improvements. Hence, in our implementation we assume that no non-zero encoding passes the zero-testing test except perhaps with very small probability even when (g) is not prime and forego this check. We ran two million tests on a given q to verify this hypothesis, but stress that this does not provide sufficient assurances to rule out the possibility of this event happening. We also note that while some applications rely on (g) being prime, they can often be adapted. We note that picking g which is not prime was already mentioned in [Gar13, Section 6.3]. However, in [Gar13] it is still required that g does not have any small prime factors which we do not ensure.

Our second contribution is to improve the size of the two parameters q and ℓ compared to the one proposed in [LSS14b]. We first perform a finer analysis than [LSS14b], 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. 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 $0 < \xi \leq 1/4$. We then argue that for a good choice of ξ this is enough to ensure the correctness of the extraction procedure and the security of the scheme.

Open problems. Since we forego the requirement that (g) is prime, our implementation, is heuristic. It is hence a natural question to either speed-up primality testing for principal ideals in cyclotomic number fields of order 2^k or investigate if this condition can be dropped. 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 situations and evaluating the performance of the resulting schemes is, hence, also 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. After some preliminaries, we recall the GGHLite scheme, the one-round Diffie-Hellman key exchange and the Ext-GCDH security assumption in Section 3. In Section 4, we explain our choice of parameters for the scheme, especially concerning the parameter q. We also recall the best known lattice attacks to explain this choice. In Section 5, we give the details of our

implementation. Finally, in Section 6, we provide the timings for the full scheme and compare them with the existing one of [CLT13].

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 $k \leq m$, 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 $(g, X \cdot g, \ldots, X^{n-1} \cdot g)$. 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 $y_i \in [-1/2, 1/2), \forall i, 0 \leq i \leq n-1$. We also use the same abuse of notation as [LSS14b] 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 $c \in \mathbb{R}^m$ and a positive parameter $\sigma \in \mathbb{R}$, we define the Gaussian distribution of center c and standard deviation σ as $\rho_{\sigma,c}(x) = \exp(-\pi \frac{||x-c||^2}{\sigma^2})$, for all $x \in \mathbb{R}^m$. This notion can be extended to ellipsoid Gaussian distribution by replacing the parameter σ with the square root of the covariance matrix $\Sigma = BB^t \in \mathbb{R}^{m \times m}$ 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 $x \in \mathbb{R}^m$. For L a subset of $\mathbb{Z}^m$, let $\rho_{\sigma,c}(L) = \sum_{x \in L} \rho_{\sigma,c}(x)$. Then, the *discrete Gaussian distribution* over L with center 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 [LSS14b, Th. 5.1] that if $Y \cdot R^2 = \mathcal{I}$ and $s \geq \max(\|g^{-1} y_1\|_\infty, \|g^{-1} y_2\|_\infty) \cdot n \cdot \sqrt{2\log(2n(1+1/\varepsilon))/\pi}$ for $\varepsilon \in (0, 1/2)$, this distribution is statistically close from 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 we use the notion of graded encoding scheme. A graded encoding scheme manipulates *encoding levels*: the "plaintext" is a level-0 encoding, and from the level-0 encoding one can construct a level-i encoding of the same element until κ, where κ is called the multilinearity parameter. But given a level-i encoding, we cannot come back and find a level-j encoding for $j < i$ for the same element. The encodings are both additively and multiplicatively homomorphic (up to κ operations for the multiplication). We more formally define the procedures in Appendix A.

3 GGHLite

3.1 The GGHLite graded encoding scheme

We recall the GGHLite graded encoding scheme from [LSS14b] in Figure 1. 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. The scheme is fully described in Figure 1.

- **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 $\|g^{-1}\| \leq \ell_{g^{-1}}$ 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 $k \leq \kappa$:
    - 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 $\mathsf{params} = (n, q, y, \{(x_1^{(k)}, x_2^{(k)})\}_{k \leq \kappa})$ 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 $\rho_1, \rho_2 \hookleftarrow D_{R,\sigma_k^*}$.
- **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 [LSS14b].

The following asymptotic choice of parameters is detailed in [LSS14b]:

- the dimension n is $O(\kappa\lambda \log \lambda)$,
- the module q is $\widetilde{O}(n^{10.5}\sqrt{\kappa})^{8\kappa}$.
- Generation of g:
  - the parameter σ is $O(n \log n)$, more precisely, for some constant $p_g < 1$:

$$\sigma \geq 4\pi n \sqrt{e \ln(8n)/\pi}/p_g,$$

  - the upper bound $\ell_{g^{-1}}$ is $O(1/\sqrt{n \log n})$, more precisely $\ell_{g^{-1}} = \frac{4\sqrt{\pi e n}}{p_g \sigma}$.
- Generation of the $(b_i^{(k)})_{i,k}$ and a:

  - the parameter σ' is $\widetilde{O}(n^{3.5})$, more precisely, $\sigma' \geq 7n^{\frac{5}{2}} \ln^{\frac{3}{2}}(n)\sigma$,
  - 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 $\sigma^\star$ is $\widetilde{O}(n^{5.5}\kappa)$, more precisely, $\sigma^\star = n^{3/2} \cdot (\sigma')^2\sqrt{8\pi/\varepsilon_d}/\ell_b$ with $\varepsilon_d = \log(\lambda/\kappa)$.
- the number of extracted bit ℓ is such that: $\log_2(8n\sigma) < \ell \leq 1/4 \log_2 q - \log_2(2n/\varepsilon_{ext})$, 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. This analysis is detailed in [LSS14b].

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

In Figure 2, we recall the construction given by [GGH13a] to adapt the N-party 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 his 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. The function H denotes a hash function modeled as a random oracle.

- **Setup**. $\mathsf{Setup}(1^\lambda, 1^N)$: Given security parameter λ and number of parties N, run $\mathsf{InstGen}(1^{2\lambda+1}, 1^{N-1})$ 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 computes the key $K_j = H(s_j)$ where $s_j = \mathsf{ext}(\mathsf{params}, p_{zt}, v_j)$ is the extracted string for v_j.

Fig. 2. The adapted N-party Diffie-Hellman key exchange protocol.

The consistency requirement follows from the agreement property of the extraction procedure. As proven in [GGH13a,LSS14b], the security follows from the randomness property of the extraction procedure and from the Ext-GCDH assumption than we define in the next section.

3.3 Extraction Graded Computational Diffie-Hellman assumption

The security of the adapted N-party Diffie-Hellman key exchange protocol relies on the Extraction Graded Computational Diffie-Hellman assumption, defined in [LSS14b] and adapted from the original GDDH/GCDH assumptions of [GGH13a].

Definition 1 ([LSS14b, Definition 3.1] Ext-GCDH). *The problems Ext-GCDH is defined as follows with respect to experiment of Figure 3:*

– ***Extraction κ-graded CDH problem (Ext-GCDH):*** *On inputs* params, p_{zt} *and the u_i's of Step 2, output the extracted string for a level-κ encoding of $\prod_{i\geq 0} e_i + \mathcal{I}$, i.e., $w = \mathsf{Ext}(\mathsf{params}, p_{zt}, v_C) = MSB_\ell([p_{zt} \cdot v_C]_q)$.*

Given parameters $\lambda, n, q, \kappa, \sigma', \sigma^\star$, proceed as follows:

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

Fig. 3. The GGH security experiment.

4 Our parameter selection

We choose most of the parameters according to the conditions described in Section 3.1. In this section, we describe a method which allows to reduce the size of two parameters: the module q and the number ℓ of extracted bits. In a second part we describe the best known attack against the scheme needed to choose n and q. Finally, we described our strategy to satisfy all 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 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, [LSS14b] 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:

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

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

The strongest constraint for q is given by the inequality $q > (3n^{\frac{3}{2}}\sigma^\star\sigma')^{8\kappa}$. 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 [LSS14b] is to extract $\ell = \log q/4 - \lambda$ bits from each element of the level-κ encoding. We show that this may supply 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 $\|p_{zt}\, u\|_\infty < q^{1-\xi}$. We will make use of the following lemma below.

Lemma 1 (Adapted from Lemma A.1 in [LSS14a]). *Let $g \in R$ such that $\mathcal{I} = (g)$ is a prime ideal in R, let $c, h \in R$ such that $\|c \cdot h\| < q/2$ and $c, h \notin \mathcal{I}$ and $q > (2tn\sigma)^{1/\xi}$ for some $t \geq 1$. Then $\|[c \cdot h/g]_q\| > t \cdot q^{1-\xi}$ for any $0 < \xi \leq 1/4$.*

Proof. We know that since $\|c \cdot h\| < q/2$ we must have $\|g \cdot [c \cdot h/g]_q\| > q/2$ by [GGH13a, Lemma 3]. 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 of q with a more precise analysis compared to [LSS14b]. Recall that $\|[p_{zt}\, u]_q\|_\infty = \|[hc/g]_q\|_\infty = \|h \ \cdot \ c/g\|_\infty \leq \|h\| \cdot \|c/g\| \leq \|h\| \cdot \|c\| \cdot \|g^{-1}\|\sqrt{n}$. The first inequality is a direct application of the inequalities between the infinite 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 $\|h\| \leq \sqrt{n}q^{1/2}$. 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, $\|c\| \leq (\sqrt{n})^{\kappa-1}(\max_{i=1,\ldots,\kappa} \|u_i\|)^\kappa$ 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 $\|g^{-1}\| \leq l_{g^{-1}}$ the inequality $\|[p_{zt}\, u]_q\|_\infty < q^{1-\xi}$ holds if:

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

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

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

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

In [LSS14b], we had $\xi = 1/4$ (which give $2/(1-2\xi) = 4$), we then have that this analysis allows to save a factor of 2 in the size of q.

Correctness of extraction. As in [LSS14b], 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 1 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 $\ell \leq \xi \log_2 q - \log_2(2n/\varepsilon_{ext})$.

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

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

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

4.2 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 such a multiple of g of the form $d \cdot g$ and multiplying it by the zero-test parameter p_{zt}, thus 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 $\|d\| \leq \sqrt{n} \cdot \|d \cdot g\| \cdot \ell_{g^{-1}}$. 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, we have to make sure that no element of form $d \cdot g$ can be found in polynomial time 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 $q/2$, then the overall euclidean norm would be less than $q\sqrt{n}/2$. We then require:

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

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

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 this, recall 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

$$\nu_0 = [v_0]_{\mathrm{HNF}(\mathcal{I})} \equiv \left(d \cdot h \cdot y \cdot \prod_{j=0}^{\kappa-1} e_j \right) \bmod \mathcal{I}, \text{ and } \nu_1 = [v_1]_{\mathrm{HNF}(\mathcal{I})} \equiv \left(d \cdot h \cdot y \cdot \prod_{j=1}^{\kappa-1} e_j \right) \bmod \mathcal{I}.$$

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 4.1 and using that $\sigma^\star > \sigma'$, we get:

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

which gives

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

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

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

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 setup 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 $q^{n/2n} = \sqrt{q}$ 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, 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. This ratio grows with κ. Hence in contrast to the previous attack for a given n this attack becomes more efficient as we increase κ.

4.3 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:

$$\frac{\sigma\sqrt{q}}{2n^{2\kappa}(3\sigma^\star\sigma')^\kappa} = \delta_0^n \cdot \mathrm{vol}(g)^{1/n} \leq \delta_0^n\sqrt{n}\sigma. \tag{8}$$

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

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

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 behavior, 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 $0.00119\,k^2 + 0.2275\,k + 21.59$ and $0.3774\,k + 20$. The first was extrapolated from data points made available in [LN13] 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 4.2 are the most efficient attack and that our lattice reduction estimate are accurate.

Putting everything together. Our overall strategy is as follows. Pick an n and compute parameters σ, σ', $\sigma^\star$, ℓ_g and q. Now, establish the root-Hermite factor required to carry out the attack in Section 4.2 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 5 in Section 6.

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 specialised most non-trivial operations to the ring of integers of a Cyclotomic number field of order 2^k. 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`.

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

The most time-critical operation in the online phase of a 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 $\mathbb{Z}[X]$ directly 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):

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 $\overline{c} = NTT^{-1}_{\omega_n}(NTT_{\omega_n}(\overline{a}) \odot NTT_{\omega_n}(\overline{b}))$.*

This negative wrapped convolution was already 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 NTT representations 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 "NTT representation" $\big(f(1), f(\omega_n), \ldots, f(\omega_n^{n-1})\big)$ once on creation and back only when running extraction. We implemented this strategy. While our own NTT is about 3-4 times slower than FLINT's optimised implementation over the Integers mainly due to modular reductions, we observe a considerable overall speed-up with this strategy.

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

During instance generation we have 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 $\mathrm{res}(f, X^n+1)$. 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* (CRT). 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 $\mathbb{Z}[X]/(X^n+1)$. 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 $q \equiv 1 \bmod 2n$ this can be accomplished using the NTT reducing the cost mod q_i to $O(M(d))$ saving a factor of $\log n$.

5.3 Verifying that (g) is a prime ideal

When running the full setup phase, by far the most time consuming step is finding a g such that (g) is a prime ideal. 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. 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.

While checking each individual g is asymptotically not much slower than polynomial multiplication, finding a g such that (g) is prime 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, pushing us way above the $O(n \log n)$ threshold. We list timings in Table 1.

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 1. Average time of checking primality of a single (g) on Intel Xeon CPU E5-2667 v2 3.30GHz using 16 cores

In our experiments below we hence forego this check. While we always check if $\mathrm{res}(g, X^n+1) = 0 \bmod q_i$ for our "interesting" primes to rule out common small prime factors, we do not verify that the g we generate is indeed prime. We call this variant *sloppy*.

Primality is used only in Lemma 1 to prove that $c \cdot h/g$ is big if $c, h \notin g$. We experimentally verified on 2 million samples that this condition still holds if (g) is not prime. Yet, of course, as Lemma 1 is not just used to ensure correctness but also security, this experimental verification cannot give sufficient assurances. We also note that some applications assume that g is prime.

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

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 $0 < i \leq 2$ then this condition is equivalent to $\tilde{b}_1^{(1)} + \tilde{b}_2^{(1)} = R$. We know of no faster way of verifying this condition than to 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 $\mathrm{res}(\tilde{b}_i^{(1)} 1, X^n + 1) = 0 = \mathrm{res}(\tilde{b}_i^{(2)} 1, X^n + 1) \bmod q_i$ 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.

5.5 Computing the inverse of a polynomial modulo $X^n + 1$

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 setup 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 $n = 2^k$.

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 $g(X) = G_e(X^2) + XG_o(X^2)$ 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
  $F(X^2) \leftarrow f(X)f(-X) \mod X^n+1$
  $G(Y) \leftarrow \text{InverseMod}(F(Y), q, n/2)$
  Set $S_e(X^2), S_o(X^2)$ such that $f(-X) = S_e(X^2) + XS_o(X^2)$
  $T_e(Y) \leftarrow G(Y) \cdot S_e(Y)$
  $T_o(Y) \leftarrow G(Y) \cdot S_o(Y)$
  $f^{-1}(X) \leftarrow T_e(X^2) + XT_o(X^2)$
  $\tilde{f}^{-1}(X) = f^{-1}(X)$ 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 2.

n	$\log\sigma$	xgcd	prec=160	iter, prec=160	prec=∞	n	$\log\sigma$	xgcd	prec=160	iter, prec=160	prec=∞
4096	17.2	334.9s	3.5s	3.9s	104.9s	8192	18.2	2055.5s	14.2s	16.7s	619.7s

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

5.6 Approximate Square Roots

During setup, for sampling from a discrete Gaussian $D_{(g),\sigma',c}$ with support (g) we need to compute an approximate square root of an element 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 applying scaling. 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 $\mathbb{Z}[X](X^n+1)$ after some scaling. 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.

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 3. 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	rate per s	prec	rate per 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 3. 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]. 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 $D_{R,\sigma' \cdot g^{-T}}$. Here, g^{-T} means the conjugate of g^{-1}. That is, $g_0^T = g_0$ and $g_{n-i}^T = -g_i$ for $1 \leq i < n$ if $\deg(g) = n - 1$. 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 where operations are performed with precision $\frac{1}{2}\log_2(n) \cdot (\log_2(n\|\sigma\|))$ using the Denman-Beavers iteration. We then use this value, scaled appropriately, as the initial value from which to start the Babylonian method for computing a square-root of $\Sigma_2 = \sigma'^2 \cdot g^{-T} \cdot g^{-1} - r^2 \cdot I$ where $r = 2 \cdot \lceil \sqrt{\log n} \rceil$. Here we perform operations with $\frac{1}{2}\log_2(n) \cdot (\log_2(n\|\sigma\|) + 2\log_2 \sigma'$ precision and 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 $D_{\mathbb{Z},r,y_i}$. We give experimental results in Table 4.

prec	n	$\log \sigma'$	sqrt iter.	sqrt wall time	$\log \|(\sqrt{\Sigma_2}')^2 - \Sigma_2\|$	$\hookleftarrow D_{g,\sigma'}$ per second
160	1024	45.8	9	2.5s	-200	26.0
160	2048	49.6	9	7.8s	-221	12.0
160	4096	53.3	10	23.2s	-239	4.8
160	8192	57.0	10	61.7s	-253	2.0
200	16384	60.7	11	203.0s	-270	0.8

Table 4. 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 16 cores for estimating the scaling factor, sampling uses one core.

6 NIKE Timings

In Table 5 we give experimental results on running a non-interactive key exchange in the common reference string model based on our implementation of GGHLite. In Table 6 we compare these results with those reported in [CLT13]

Acknowledgement: We would like to thank Guilhem Castagnos, Steven Galbraith, Bill Hart, Claude-Pierre Jeannerod, Clément Pernet and Damien Stehlé 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".

λ	κ	n	$\log q$	ξ	$\log \sigma$	$\log \sigma'$	δ_0	Setup	Publish (pp)	KeyGen (pp)	$\|\|\|$params$\|\|$
52	6	32768	2551	0.036	20.4	65.7	1.0128	549s	3.66s	1.7s	139.5MB
52	9	32768	3768	0.024	20.4	65.7	1.0192	607s	5.89s	3.6s	292.3MB
52	14	65536	6052	0.015	21.4	69.4	1.0158	1982s	27.24s	21.31s	1.4GB
80	6	65536	2746	0.044	21.4	69.4	1.0069	2232s	8.97s	4.3s	300.0MB
80	9	65536	4005	0.030	21.4	69.4	1.0102	2315s	13.93s	8.30s	625.7MB
80	14	131072	6428	0.019	22.4	73.1	1.0084	7679s	64.09s	48.45s	2.9GB

Table 5. Experimental results for running the $(\kappa + 1)$-partite Non-Interactive Key Exchange. Timings were produced on Intel Xeon CPU E5-2667 v2 3.30GHz, Setup was parallelised on 16 cores, times for Publish and KeyGen are given per participant ("pp"). All instances are sloppy, i.e. (g) is not verified to be prime. The column "Setup$^-$" gives the setup time excluding the time to verify that $(b_1^{(1)}, b_2^{(1)}) = (g)$. All times are wall times.

	λ	κ	Setup	Publish (pp)	KeyGen (pp)	pk size
[CLT13]	52	6	7s	0.18s	0.2s	26.0MB
This work	52	6	549s	3.66s	1.7s	139.5MB
[CLT13]	80	6	27295s	17.8s	20.2s	2.6GB
This work	80	6	2273s	9.0s	4.3s	300MB

Table 6. Comparison with timings reported in [CLT13]. Our experiments were run on Intel Xeon CPU E5-2667 v2 3.30GHz, utilising 16 cores during the Setup phase and 1 core otherwise. 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 our instances are sloppy, i.e. (g) is not verified to be prime. All CLT instances rely on heuristic assumptions [CLT13, Section 6]. All times are wall times.

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`.

[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.

[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.

[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.

[Gt14] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.0.0 edition, 2014. `http://gmplib.org/`.

[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.

[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.

[LN13] Mingjie Liu and Phong Q. Nguyen. Solving BDD by enumeration: An update. In Ed Dawson, editor, *CT-RSA 2013*, volume 7779 of *LNCS*, pages 293–309. Springer, February / March 2013.

[LSS14a] A. Langlois, D. Stehlé, and R. Steinfeld. GGHLite: More Efficient Multilinear Maps from Ideal Lattices. Cryptology ePrint Archive, Report 2014/487, 2014. Full version of [LSS14b].

[LSS14b] 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.

[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.

[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 $i_1 + i_2 \le \kappa$, 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 $\Pr\left[\mathsf{Ext}(\mathsf{params}, p_{zt}, u_1) = \mathsf{Ext}(\mathsf{params}, p_{zt}, u_2)\right] \ge 1 - \text{negligible}(\lambda)$.
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 $\{0, 1\}^\lambda$.