Implementing Candidate Graded Encoding Schemes from Ideal Lattices

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 popular 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. Implementing GGH-like schemes naively would not allow instantiating 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. Due to our improvements we were able to compute a multilinear jigsaw puzzle for $\kappa = 52$ (resp. $\kappa = 38$) and $\lambda = 52$ (resp. $\lambda = 80$).

1 Introduction

Multilinear maps, starting with bilinear ones, are popular 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 constructing 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 a variant of multilinear maps and some additional assumptions.

The GGH scheme. For a multilinearity parameter κ, the principle of the symmetric GGH graded encoding scheme is as follows: given a ring R and a principal ideal $\mathcal{I}$ generated by a small secret element $g \in R$, a plaintext is a small element of $R/\mathcal{I}$ and is viewed as a level-0 encoding. Given a level-0 encoding, it is easy increase the level to 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 encoding gives a level-$(i + j)$ encoding. Additionally, a zero-testing parameter p_{zt} allows testing if a level-κ element is an encoding of 0, and hence also allows testing 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}. We will see later that the encodings of zero made public for this step are a problem for the security of the scheme.

The asymmetric variant of this scheme replaces levels by "groups" which are identified with subsets of $\{1, \ldots, \kappa\}$. Addition of two elements in the same group stays within the group, multiplying two elements of different groups with disjoint index sets produces an element in the group defined by the union of their index sets. These groups are realised by defining one z_i for each index $1 \leqslant i \leqslant \kappa$ and then dividing by the appropriate product of z_i. Given a group characterised by $S \subseteq \{1, \ldots, \kappa\}$ we call the cardinality of S its level.

We can distinguish between GGH instances where encodings of zero are made publicly available to allow anyone to encode elements and those where this is not the case. The latter are also called "Multilinear Jigsaw Puzzles" and were introduced in [GGH+13b] as a building block for indistinguishability obfuscation. Such instances can be thought of as secret-key graded encoding schemes. To distinguish the two cases, we denote those instances where no encodings of zero $x_j^{(i)}$ are published as GGH_s. In such instances the secret elements g and z_i are required to encode elements at levels above zero.

Security. Already in [GGH13a] it was shown that an attacker can recover the ideal (g) and the coset of (g) for any encoding at level $\leqslant \kappa$ if encodings of zero are made available. However, since these representatives of either (g) or the cosets are not small, it was believed that these "weak discrete log" attacks would not undermine the central security goal of GGH – the analogue of the BDDH assumption. However, in [HJ15] it was shown that these attacks can be extended to recover short representatives of the cosets. As a consequence, if encodings of zero are published, then [HJ15] breaks the GGH security goals in many scenarios and it is not clear, at present, if and how GGH-like graded encoding schemes can be defended against such attacks. A candidate proposal to prevent weak discrete logarithm attacks was proposed in [CLT15, Appendix G], where the strategy is to change zero testing to make it non-linear in the encodings such that the attack does not work anymore. However, no security analyses was provided in [CLT15] and revision `20150516:083005` of [CLT15] drops any mention of this candidate fix. Hence, the status of GGH-like schemes where encodings of zero are published is currently unclear. However, we note that GGH_s, where no encodings of zero are made available, does not appear to be vulnerable to weak discrete log attacks. This version is a central building block of indistinguishability obfuscation, i.e. this case has important applications despite being more limited in functionality.

Alternative Constructions. An alternative instantiation of graded encoding schemes over the integers promising practicality was proposed by Coron, Lepoint and Tibouchi [CLT13]. This first proposal was also broken in polynomial time using public encodings of zero in [CHL+15]. The attack was later generalised in [GHMS14] and a candidate defence against these attacks was proposed in [CLT15]. The authors of [CLT15] also provided a C++ implementation of a heuristic variant of this scheme. They report that the **Setup** phase of an 7-partite Diffie-Hellman key exchange takes 4528s (parallelised on 16 cores), publishing a share (**Publish**) takes 7.8s per party (single core) and the final key derivation (**KeyGen**) takes 23.9s per party (single core) for a level of security $\lambda = 80$.

Instantiation. The implementation reported in [CLT15] is to date the only implementation of a candidate graded encoding scheme. This is partly because instantiating the original GGH construction is too costly in practice for anything but toy instances. In 2014, Langlois, Stehlé and Steinfeld [LSS14a] proposed a variant of GGH called GGHLite, improving the re-randomisation process of the original scheme. It reduces the number m of re-randomisers, public encodings of zero, 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 $\tilde{\mathcal{O}}(2^\lambda \lambda n^{4.5}\kappa)$ to $\tilde{\mathcal{O}}(n^{5.5}\sqrt{\kappa})$. These improvements allow reducing the size of the public parameters and improving the overall efficiency of the scheme. But even though [LSS14a] made a step forward towards efficiency and in some cases no public re-randomisation is required at all (GGH_s), GGH-like schemes are still far from being practical.

Our contribution. Our main contribution is a first and efficient implementation of improved GGH-like schemes which we make publicly available under an open-source license. This implementation covers symmetric and asymmetric flavours and we allow encodings of zero to be published or not. Since the security of GGH-like constructions is unclear when encodings of zero are published, we do not discuss this variant here. We note, however, that our implementation provides a good basis for implementing any future fixes and improvements for GGH-based graded encoding schemes.

Implementing GGH-like schemes efficiently such that non-trivial levels of multilinearity and security can be achieved is not straight forward and to obtain an implementation we had to address several issues. In particular, we contribute the following improvements to make GGH-like multilinear maps more 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, which in turn reduces the required dimension n.
- We apply the analyses from [CS97] to pick parameters to defend against lattice attacks.
- 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 inverting in some Cyclotomic number fields, for computing norms of ideals in some Cyclotomic number rings, for producing short representatives of elements modulo (g) and for sampling from discrete Gaussians in $\tilde{\mathcal{O}}(n)$. For the latter we use Ducas and Nguyen's strategy [Duc13] Our implementation of these operations might be of independent interest (cf. [LP15] for recent work on efficient sampling from a discrete Gaussian distribution), which is why they are available as a separate module in our code.
- 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 $\kappa = 52$ (resp. $\kappa = 38$) at security level $\kappa = 52$ (resp. $\lambda = 80$) in the asymmetric GGH_s case. 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). We note that the results in Table 1, where no encodings of zero are made available, are not directly comparable with those reported in [CLT15].

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 degree is a power of two and related rings as listed above.

Our variant of GGH foregoes checking if g generates a prime ideal. During instance generation [GGH13a,LSS14a] specify to sample g such that (g) is a prime ideal. This condition is needed

λ	κ	λ'	n	$\log q$	Setup	Encode	Mul	$\|\text{enc}\|$
52	6	64.4	2^{15}	2117	114s	26s	0.05s	8.3MB
52	9	53.5	2^{15}	3086	133s	25s	0.12s	12.1MB
52	14	56.6	2^{16}	4966	634s	84s	0.62s	38.8MB
52	19	56.6	2^{16}	6675	762s	75s	1.38s	52.2MB
52	25	59.6	2^{17}	9196	2781s	243s	5.78s	143.7MB
52	52	62.7	2^{18}	19898	26695s	1016s	84.1s	621.8MB
80	6	155.2	2^{16}	2289	415s	74s	0.13s	17.9MB
80	9	86.7	2^{16}	3314	445s	72s	0.27s	25.9MB
80	14	120.9	2^{17}	5288	1525s	252s	1.38s	82.6MB
80	19	80.4	2^{17}	7089	1821s	268s	3.07s	110.8MB
80	25	138.8	2^{18}	9721	9595s	967s	13.52s	303.8MB
80	38	80.3	2^{18}	14649	20381s	947s	16.21s	457.8MB

Table 1. Computing a κ-level asymmetric multilinear maps with our implementation without encodings of zero. 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. Setup gives the time for generating the GGH instance. Encode lists the time it takes to reduce an element $\in \mathbb{Z}_p$ with $p = \mathcal{N}(\mathcal{I})$ to a small element in $\mathbb{Z}[X]/(X^n+1)$ modulo (g). Mult lists the time to multiply κ elements. All times are wall times.

in [GGH13a,LSS14a] 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 speeding-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).

We also improve the size of the two parameters q and ℓ compared to [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 reducing the dimension n. When no encodings of zero are published we simply set $\sigma_1^\star = 1$ and apply the same analysis.

Open problems. The most pressing question at this point is whether GGH-like constructions are secure. There exist no security proofs for any variant and recent cryptanalysis results recommend caution. Even speculating that secure variants of GGH-like multilinear maps can be found, performance is still an issue. 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. Finally, establishing better estimates for lattice reduction and tuning the parameter choices of our schemes are areas of future work. For example, it is clear that our analysis, where we simply set $\sigma_1^\star = 1$ when no encodings of zero are published, can be refined further.

Roadmap. We give some preliminaries in Section 2. In Section 3 we describe the GGH-like asymmetric graded encoding schemes and the multilinear jigsaw puzzles used for iO. In Section 4, we explain our modifications to GGH-like schemes, especially concerning the parameter q. We also recall a lattice attack 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 $D_{\mathcal{I}, sY^T}$.

3 GGH-like Asymmetric Graded Encoding Scheme

We now recall the definitions given in [GGH+13b, Section 2.2] for the notions of Jigsaw specifier, Multilinear Form and Multilinear Jigsaw puzzle. We refer to the original paper for the security of this construction.

Definition 1 ([GGH+13b, Def. 5]). *A* Jigsaw specifier *is a tuple (κ, ℓ, A) where $\kappa, \ell \in \mathbb{Z}^+$ are parameters and A is a probabilistic circuit with the following behavior: On input a prime number q, A outputs the prime q and an ordered set of ℓ pairs $(S_1, a_1), \ldots, (S_\ell, a_\ell)$ where each $a_i \in \mathbb{Z}_q$ and each $S_i \subseteq [\kappa]$.*

Definition 2 ([GGH+13b, Def. 6 and 7]). *A* Multilinear Form *is a tuple* $\mathcal{F} = (\kappa, \ell, \Pi, F)$ *where* $\kappa, \ell \in \mathbb{Z}^+$ *are parameters and* Π *is a circuit with* ℓ *imput wires, made out of binary and unary gates.* F *is an assignment of an index set* $I \subseteq [\kappa]$ *to every wire of* Π*. A multilinear form must satisfies constraints given in the original definition (on gates, and the output wire is assigned to* $[\kappa]$*).*

We say that a Multilinear Form $\mathcal{F} = (\kappa', \ell', \Pi, F)$ *is compatible with* $X = ((S_1, a_1), \ldots, (S_\ell, a_\ell))$ *if* $\kappa = \kappa'$*,* $\ell = \ell'$ *and the input wires of* Π *are assigned to the sets* $S_1, \ldots, S_\ell$*. The evaluation of* $\mathcal{F}$ *on* X *is then doing arithmetic operations on the inputs depending on the gates. We say that the evaluation succeeds if the final output is* $([\kappa], 0)$*.*

We now define the Multilinear Jigsaw Puzzles.

Jigsaw Generator: $\mathsf{JGen}(\lambda, \kappa, \ell, A) \to (q, X, \mathsf{puzzle})$**.** This algorithm takes as input λ, and a Jigsaw specifier (κ, ℓ, A). It outputs a prime q, a private output X and a public output puzzle. The generator is using a pair of PPT algorithms $\mathsf{JGen} = (\mathsf{InstGen}, \mathsf{Encode})$.

$\mathsf{InstGen}(\lambda, \kappa) \to (q, \mathsf{params}, s)$**.** This algorithm takes λ and κ as inputs and outputs (q, params, s), where q is a prime of size at least 2^λ params is a description of public parameters, and s is a secret state to pass to the encoding algorithm.

$\mathsf{Encode}(p, \mathsf{params}, s, (S, a)) \to (S, u)$**.** The encoding algorithm takes as inputs the prime q, the parameters params, the secret state s, and a pair (S, a) with $S \subseteq [\kappa]$ and $a \in \mathbb{Z}_q$ and outputs u, and encoding of a relative to S.

More precisely, the algorithm runs the Jigsaw specifier on input p to get ℓ pairs $(S_1, a_1), \ldots, (S_\ell, a_\ell)$. Then encodes all the plaintext elements by using the Encode algorithm on each (S_i, a_i) which return (S_i, u_i). We have:

$$X = (q, (S_1, a_1), \ldots, (S_\ell, a_\ell)) \text{ and } \mathsf{puzzle} = (\mathsf{params}, (S_1, u_1), \ldots, (S_\ell, u_\ell)).$$

Jigsaw Verifier: $\mathsf{JVer}(\mathsf{puzzle}, \mathcal{F}) \to \{0, 1\}$**.** This algorithm takes as input the public output of a Jigsaw Generator puzzle, and a multilinear form $\mathcal{F}$. It outputs either accept (1) or reject (0).

Correctness. For an output (q, X, puzzle) and a form $\mathcal{F}$ compatible with X, we say that the verifier JVer is correct if either the evaluation of $\mathcal{F}$ on X succeeds and $\mathsf{JVer}(\mathsf{puzzle}, \mathcal{F}) = 1$ either the evaluation fails and $\mathsf{JVer}(\mathsf{puzzle}, \mathcal{F}) = 0$. We require that with high probability over the randomness of the generator, the verifier will be correct on all form.

Security. The hardness assumptions for the Multilinear Jigsaw puzzle requires that for two different polynomial-size families of Jigsaw Specifier $\{(\kappa_\lambda, \ell_\lambda, A_\lambda)\}_{\lambda \in \mathbb{Z}^+}$ and $\{(\kappa_\lambda, \ell_\lambda, A'_\lambda)\}_{\lambda \in \mathbb{Z}^+}$ the public output of the Jigsaw Generator on $(\kappa_\lambda, \ell_\lambda, A_\lambda)$ will be computationally indistinguishable from the public output of the Jigsaw Generator on $(\kappa_\lambda, \ell_\lambda, A'_\lambda)$.

3.1 Using GGH to construct Jigsaw puzzles

In Figure 1, we describe a GGH-like asymmetric graded encoding scheme without encodings of zero based on the definition of GGHLite from [LSS14a].

We now explain how to use those procedure to construct the Jigsaw Generator, described in [GGH+13b, Appendix A].

Jigsaw Generator. The Jigsaw Generator uses InstGen to generate all the public (params and p_{zt}) and secret parameters of the multilinear map. Each level of the multilinear map will be associated with a subset of the set $[\kappa]$. To create the puzzle pieces, which are encodings of some elements of R at different level, the Generator simply encodes some random elements at level $S \subset [1, \kappa]$, those are given as puzzle.

- **Instance generation.** $\mathsf{InstGen}(1^\lambda, 1^\kappa)$: Given security parameter λ and multilinearity parameter κ, determine scheme parameters n, q, σ, σ', $\ell_{g^{-1}}$, ℓ_b, ℓ as in [LSS14a]. 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 $z_i \hookleftarrow U(R_q)$ for all $0 < z_i \leqslant k$.
  - Sample $h \hookleftarrow D_{R,\sqrt{q}}$ and define the zero-testing parameter $p_{zt} = \left[\frac{h}{g}\prod_{i=1}^{\kappa} z_i\right]_q$.
  - Return public parameters $\mathsf{params} = (n, q, \ell)$ and p_{zt}.
- **Encode at level-0** $\mathsf{Enc0}(\mathsf{params}, g, e)$: Compute a small representative $e' = [e]_g$ and sample an element $e'' \hookleftarrow D_{e'+\mathcal{I},\sigma'}$. Output e''.
- **Encode in group** $\{i\}$. $\mathsf{Enc}(\mathsf{params}, z_i, e)$: Given level-0 encoding $e \in R$, parameters params and z_i, output $[e/z_i]_q$.
- **Adding encodings.** Add: Given encodings $u_1 = \left[c_1/\left(\prod_{i\in S} z_i\right)\right]_q$ and $u_2 = \left[c_2/\left(\prod_{i\in S} z_i\right)\right]_q$ with $S \subseteq \{1, \ldots, \kappa\}$:
  - Return $u = [u_1 + u_2]_q$, an encoding of $[c_1 + c_2]_q$ in the group S.
- **Multiplying encodings.** Mult: Let $S_1 \subset [\kappa]$, $S_2 \subset [\kappa]$ with $S_1 \cap S_2 = \varnothing$, given an encoding $u_1 = \left[c_1/\left(\prod_{i\in S_1} z_i\right)\right]_q$ and an encoding $u_2 = \left[c_2/\left(\prod_{i\in S_2} z_i\right)\right]_q$:
  - Return $u = [u_1 \cdot u_2]_q$, an encoding of $[c_1 \cdot c_2]_q$ in $S_1 \cup S_2$.
- **Zero testing at level** κ. $\mathsf{isZero}(\mathsf{params}, p_{zt}, u)$: Given an encoding $u = \left[c/\left(\prod_{i=0}^{\kappa-1} z_i\right)\right]_q$, return 1 if $\|[p_{zt}u]_q\|_\infty < q^{3/4}$ and 0 else.

Fig. 1. GGH-like asymmetric graded encoding scheme adapted from [LSS14a].

Jigsaw Verifier. The verifier is given the public parameters params and p_{zt}, a valid form Π (which is defined [GGH+13b, Def. 6] in as a circuit made of binary and unary gates) and puzzle, an input for Π (which are some encodings). The verifier is then evaluating Π on these input using Add for addition gates and Mult for multiplication gates. The verifier must succeeds if the evaluation of $\mathcal{F}$ on X succeeds, which means that the final output of the evaluation is an encoding of zero at level κ. The verifier is invoking the zero-testing procedure, and outputs 1 if the test passes, 0 otherwise.

4 Modifications to and parameters for GGH-like schemes

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 then we describe the lattice-attack against the scheme which we use to pick the dimension n. Finally, we describe our strategy to choose parameters that satisfy all these constraints.

4.1 Non-prime (g)

Both GGHLite and GGH-like jigsaw puzzles as specified in Figure 1 require 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 $\kappa + 1$ elements in R_g with overwhelming probability. 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, with overwhelming probability, the product of $\kappa + 1$ 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, following [CDKD14] we define $p_s = \prod_{i \in \mathcal{S}} p_i$ as the square-free part of p. Asymptotically, it holds that $\#\{p \leqslant x : p/p_s > p_s\}$ is $cx^{3/4}$ for some computable constant c (cf. [CDKD14]). Since in our case we have $x \geqslant 2^n$, this implies that with overwhelming probability it holds that $p_s \geqslant \sqrt{p}$ and hence $\log(p_s) \geqslant n/2$.

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 element 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", we can compute $h \bmod g$ as $h - g \cdot \lfloor g^{-1} \cdot h \rceil$, which produces an element of size $\approx \sqrt{n} \cdot \|g\|$. 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 $\|g\|$ and $\approx \sqrt{n} \cdot \|g\|$ 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.

Finally, we note that recent proposals of indistinguishability obfuscation from multilinear maps [Zim15,AB15] requires composite order maps. These are not the maps we are concerned with here as in [Zim15,AB15] it is assumed that the factorisation of (g) is known. However, we note that our techniques and implementation easily extend to this case by considering $g = g_1 \cdot g_2$ for known co-prime g_1 and g_2.

4.2 Reducing the size of q

In this section show how to reduce q for which we consider 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 but not the Jigsaw puzzle case. However, when no re-randomisers are published we may simply set $\sigma_1^\star = 1$ and apply the same analysis. Hence, assuming that re-randomisers are published fits our framework in all cases and makes our analysis compatible with previous work. We note that the analysis can be easily generalised to accommodate re-randomisers at higher levels than one by increasing q to accommodate "numerator growth".

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 $t \geqslant 1$ and any $0 < \xi \leqslant 1/4$. Then $\|[c \cdot h/g]_q\| > t \cdot q^{1-\xi}$.*

Proof. From [GGH13a, Lemma 3] and the discussion in Section 4.1 we know that since $\|c \cdot h\| < q/2$ we must have $\left\|g \cdot [c \cdot h/g]_q\right\| > q/2$ if (g) and (h) are co-prime (note that $c \cdot h \neq g \cdot [c \cdot h/g]_q$ in $R/(X^n+1)$). So we have $\left\|g \cdot [c \cdot h/g]_q\right\| > q/2 \implies \sqrt{n}\,\|g\| \cdot \left\|[c \cdot h/g]_q\right\| > q/2 \implies \left\|[c \cdot h/g]_q\right\| > q/(2n\sigma)$. 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 $\left\|g^{-1}\right\| \leqslant l_{g^{-1}}$ 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

$$\|e \cdot a + \rho_1 \cdot b_1^{(1)} + \rho_2 \cdot b_2^{(1)}\|^\kappa \leqslant (\|e\| \cdot \|a\|\sqrt{n} + \|\rho_1\| \cdot \|b_1^{(1)}\|\sqrt{n} + \|\rho_2\| \cdot \|b_2^{(1)}\|\sqrt{n})^\kappa,$$

$e \leftarrow D_{R,\sigma'}, a \leftarrow D_{1+I,\sigma'}, b_1^{(1)}, b_2^{(1)} \leftarrow D_{I,\sigma'}$ 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. For practical parameter sizes we reduce the size of q by a factor of almost 4 because ξ tends towards zero as 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 Lattice attacks

To pick a dimension n we rely on lattice attacks. The most efficient lattice attacks described [GGH13a] rely on computing weak discrete logarithms and hence do not seem to be applicable to either the case where no encodings of zero are published or the case where such attacks are ruled out in some other way. However, we may mount the attack from [CS97] against GGH-like graded encoding schemes. We explain it in the symmetric setting. Assume two encodings of random elements: $u_1 = [e_1/z]_q$ and $u_2 = [e_2/z]_q$. We have

$$\left[\frac{u_1}{u_2}\right]_q = \left[\frac{e_1/z}{e_2/z}\right]_q = \left[\frac{e_1}{e_2}\right]_q$$

with e_1 and e_2 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 U a rotational basis for $[u_1/u_2]_q$. By construction Λ contains the vector (e_1, e_2) which is short. We have $\det(\Lambda) = q^n$ and $\|(e_1, e_2)\| \approx \sqrt{2n}\sigma'$. 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 $\|(e_1, e_2)\|$. While Λ does not constitute a Unique-SVP instance because there are many short elements of norm roughly $\sqrt{2n}\sigma'$ 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. 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. We assume that the complexity of finding a short element in Λ depends on the gap between (e_1, e_2) and $\sqrt{q}$ in a similar way.

In order to succeed, an attacker needs to solve something akin of a Unique-SVP instance with gap $\lambda_2(\Lambda)/\lambda_1(\Lambda)$. We need to pick parameters such that this 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 $\|v\| = \delta_0^n \operatorname{vol}(L)^{1/n}$. Hence, in our case we require $\tau \cdot \delta_0^{2n} \leqslant \lambda_2(\Lambda)/\lambda_1(\Lambda)$ and thus

$$\delta_0 \leqslant \left(\frac{\sqrt{q}}{\sqrt{2n} \cdot \sigma' \cdot \tau}\right)^{1/(2n)}, \tag{5}$$

where τ is a constant which depends on the lattice structure and on the reduction algorithm used. Typically $\tau \approx 0.3$ [APS15], 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: 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 in [APS15].

We stress, though, that these assumptions requires further scrutiny. Firstly, this attack does not use p_{zt} which means we expect that better lattice attacks can be found eventually. Secondly, we are assuming that the lattice reduction estimates in [APS15] are accurate. However, should these assumptions be falsified, then this part of the analysis can simply be replaced by refined estimates.

4.4 Putting everything together

Our overall strategy is as follows. Pick an n and compute parameters σ, σ', $\sigma_1^\star$ as in [LSS14a] and ℓ_g and q as in Section 4.2. Now, establish the root-Hermite factor required to carry out the attack in Section 4.3 using Equation (5). If this δ_0 is small enough to satisfy security level λ terminate, otherwise double n and restart the procedure.

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 degree 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 for computing multilinear maps using our implementation in Table 1.

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 1 is $n = 2^{15}$ and $\log q \approx 2117$ 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 $\mathbb{Z}_q[X]/(X^n+1)$

During the evaluation of a GGH-style graded encoding scheme multiplications of polynomials in $\mathbb{Z}_q[X]/(X^n+1)$ are performed. Asymptotically fast multiplication in this ring can be realised by first reducing to multiplication in $\mathbb{Z}[X]$ and then to the Schönehage-Strassen algorithm for multiplying large integers in $O(n \log n \log\log n)$. This is the strategy implemented in FLINT, which has a highly optimised implementation of the Schönehage-Strassen algorithm. Alternatively, we can get an $O(n \log n)$ algorithm by using the *Number-Theoretic Transform* (NTT). Furthermore, using a negative wrapped convolution we can avoid reductions modulo (X^n+1):

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

The NTT with a negative wrapped convolution has been used in lattice-based cryptography before, e.g. [LMPR08]. We note that 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, which 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. We observe a considerable overall speed-up with the strategy of avoiding the conversions where possible. We also 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 $\log n$, which in our case is typically > 15.

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 2.

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

If re-randomisation elements are required, then we require that they generate all of (g), i.e. $(b_1^{(1)}, b_2^{(1)}) = (g)$. 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 $\gcd(\mathrm{res}(\tilde{b}_1^{(1)}, X^n+1), \mathrm{res}(\tilde{b}_2^{(1)}, X^n+1)) = 1$,

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 2. 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

i.e. if the respective ideal norms are co-prime. 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 several "interesting" primes and resample if this condition holds. These "interesting" 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. 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 some 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 $X^n + 1$

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

$$F(X^2)\, f^{-1}(X) = f(-X) \pmod{X^n+1}. \tag{6}$$

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. 6, 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.

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 $\|\tilde{f}^{-1}(X)\cdot f(X) - 1\| < 2^{-\mathtt{prec}}$ to ensure the accuracy of our result.

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

We give experimental results comparing Algorithm 1 with FLINT's extended GCD algorithm in Table 3 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 3. Inverting $g \hookleftarrow D_{\mathbb{Z}^n,\sigma}$ with FLINT's extended Euclidean algorithm ("xgcd"), our implementation with precision 160 ("160"), iterating our implementation until $\|\tilde{f}^{-1}(X) \cdot f(X)\| < 2^{-160}$ ("160iter") and our implementation without truncation ("∞") on Intel Core i7–4850HQ CPU at 2.30GHz, single core.

5.6 Small remainders

The Jigsaw Generator as defined in [GGH+13b, Definition 8] takes as input elements a_i in $\mathbb{Z}_p$ where $p = \mathcal{N}(\mathcal{I})$ and produces level encodings with respect to some source group S_i. In particular, this algorithm produces some small representative of the coset a_i modulo (g) from large integers of size $\approx (\sigma\sqrt{n})^n$ if we represents elements in $\mathbb{Z}_p$ as integers $0 \leqslant a_i < p$. This can be accomplished by using Babai's trick and that g is small, i.e. by computing $a_i - g \cdot \lfloor g^{-1} \cdot a_i \rceil$ in $\mathbb{Q}[X]/(X^n + 1)$. However, in order for this operation to produce sufficiently small elements, we need g^{-1} either exactly or with high precision. Computing such a high quality approximation of g^{-1} can be prohibitively expensive in terms of memory and time. Our strategy for computing with a lower precision is to rewrite a_i as

$$a_i = \sum_{j=0}^{\lceil \log_2(a_i)/B \rceil} 2^{B \cdot j} \cdot a_{ij}$$

where $a_{ij} < 2^B$ for some B. Then, we compute small representatives for all $2^{B \cdot j}$ and a_{ij} using an approximation of g^{-1} with precision B. Finally, we multiply the small representatives for $2^{B \cdot j}$ and a_{ij} and add up their products. This produces a somewhat short element which we then reduce using our approximation of g^{-1} with precision B until its size does not decrease any more.

5.7 Sampling from a Discrete Gaussian

While the strategy in Section 5.6 produces short elements it does not necessarily produce elements which follow a spherical Gaussian distribution and hence do not leak geometric information about g. To produce such samples we need to sample from the discrete Gaussian $D_{(g),\sigma',c}$ where c is a small representative of a coset of (g). Furthermore, if encodings of zero are published, we are

required to sample from $D_{(g),\sigma',0}$ and $D_{(g),\sigma',1}$. 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 4. We note that in our implementation 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, SampleZ]	10000	1.0	53	660.000	160	310.000
tabulated [GPV08, SampleZ]	10000	0.5	53	650.000	160	260.000
online [GPV08, SampleZ]	10000	1.0	53	414.000	160	9.000
online [GPV08, SampleZ]	10000	0.5	53	414.000	160	9.000
[DDLL13, Algorithm 12]	10000	1.0	53	350.000	160	123.000

Table 4. 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 a variant of [Pei10] which we reproduce in Algorithm 2. Namely, we first observe that $D_{(g),\sigma',0} = g \cdot D_{R,\sigma'\cdot g^{-T}}$ 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 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 (see below) of $\Sigma_2' = g^{-T} \cdot g^{-1}$ up to λ bits of precision. We perform operations 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 computing a square-root of $\Sigma_2 = \sigma'^2 \cdot g^{-T} \cdot g^{-1} - r^2$ with $r = 2 \cdot \lceil \sqrt{\log n} \rceil$. We terminate when the square of the approximation is within distance $2^{-2\lambda}$ to Σ_2. This typically happens quickly because our initial candidate is already very close to the target value.

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 prec. 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

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 $D_{\mathbb{Z},r,y_i}$. This algorithm is then easily extended to sample from arbitrary centres c. The whole algorithm is summarised in Algorithm 3 and we give experimental results in Table 5.

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 $g \cdot ([y]_r)$

			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 5. 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, 2 cores for Denman-Beavers, 4 cores for estimating the scaling factor, one core for sampling. The last column lists the rate (samples per second) of sampling from $D_{(g),\sigma'}$.

5.8 Approximate square roots

Our Gaussian sampler requires an (approximate) square root in $\mathbb{Q}[X]/(X^n+1)$. That is, for some input element Σ we want to compute some element $\sqrt{\Sigma}' \in \mathbb{Q}[X]/(X^n+1)$ 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 our inversion algorithm.

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 above, 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 5 for timings.

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

AB15. Benny Applebaum and Zvika Brakerski. Obfuscating circuits via composite-order graded encoding. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 528–556. Springer, March 2015.

APS15. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. `http://eprint.iacr.org/2015/046`.

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.

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.

CHL+15. Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In Oswald and Fischlin [OF15], pages 3–12.

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

CLT15. Jean-Sebastien Coron, Tancrede Lepoint, and Mehdi Tibouchi. New multilinear maps over the integers. Cryptology ePrint Archive, Report 2015/162, 2015. `http://eprint.iacr.org/2015/162`.

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.

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.

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.

HJ15. Yupu Hu and Huiwen Jia. Cryptanalysis of GGH map. Cryptology ePrint Archive, Report 2015/301, 2015. http://eprint.iacr.org/.

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

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.

LP15. Vadim Lyubashevsky and Thomas Prest. Quadratic time, linear space algorithms for Gram-Schmidt orthogonalization and gaussian sampling in structured lattices. In Oswald and Fischlin [OF15], pages 789–815.

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.

OF15. Elisabeth Oswald and Marc Fischlin, editors. *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*. Springer, April 2015.

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.

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.

Zim15. Joe Zimmerman. How to obfuscate programs directly. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 439–467. Springer, April 2015.

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 multilinear jigsaw puzzles at various levels of security and multilinearity. These tables were produced using the Sage [S+13] module from Appendix C.

λ	κ	n	q	$\|\text{enc}\|$	$\|\text{params}\|$	δ_0	BKZ Enum	BKZ Sieve
52	2	2^{14}	$\approx 2^{781.5}$	$\approx 2^{23.6}$	$\approx 2^{23.6}$	1.006855	$\approx 2^{112.2}$	$\approx 2^{101.8}$
52	4	2^{15}	$\approx 2^{1469.0}$	$\approx 2^{25.5}$	$\approx 2^{25.5}$	1.007031	$\approx 2^{110.4}$	$\approx 2^{102.3}$
52	6	2^{15}	$\approx 2^{2114.9}$	$\approx 2^{26.0}$	$\approx 2^{26.0}$	1.010477	$\approx 2^{64.4}$	$\approx 2^{83.3}$
52	10	2^{15}	$\approx 2^{3406.8}$	$\approx 2^{26.7}$	$\approx 2^{26.7}$	1.017404	$\approx 2^{53.5}$	$\approx 2^{68.6}$
52	20	2^{16}	$\approx 2^{7014.8}$	$\approx 2^{28.8}$	$\approx 2^{28.8}$	1.018311	$\approx 2^{56.6}$	$\approx 2^{71.7}$
52	40	2^{17}	$\approx 2^{14599.3}$	$\approx 2^{30.8}$	$\approx 2^{30.8}$	1.019272	$\approx 2^{59.6}$	$\approx 2^{74.8}$
52	80	2^{18}	$\approx 2^{30508.4}$	$\approx 2^{32.9}$	$\approx 2^{32.9}$	1.020258	$\approx 2^{62.7}$	$\approx 2^{77.8}$
52	160	2^{18}	$\approx 2^{60827.8}$	$\approx 2^{33.9}$	$\approx 2^{33.9}$	1.040912	$\approx 2^{54.0}$	$\approx 2^{54.0}$
80	2	2^{14}	$\approx 2^{837.5}$	$\approx 2^{23.7}$	$\approx 2^{23.7}$	1.007451	$\approx 2^{98.2}$	$\approx 2^{94.5}$
80	4	2^{15}	$\approx 2^{1525.0}$	$\approx 2^{25.6}$	$\approx 2^{25.6}$	1.007330	$\approx 2^{103.7}$	$\approx 2^{98.8}$
80	6	2^{16}	$\approx 2^{2287.2}$	$\approx 2^{27.2}$	$\approx 2^{27.2}$	1.005661	$\approx 2^{160.9}$	$\approx 2^{128.3}$
80	10	2^{17}	$\approx 2^{3844.7}$	$\approx 2^{28.9}$	$\approx 2^{28.9}$	1.004882	$\approx 2^{209.0}$	$\approx 2^{150.9}$
80	20	2^{18}	$\approx 2^{7824.9}$	$\approx 2^{30.9}$	$\approx 2^{30.9}$	1.005074	$\approx 2^{198.9}$	$\approx 2^{148.5}$
80	40	2^{19}	$\approx 2^{16152.9}$	$\approx 2^{33.0}$	$\approx 2^{33.0}$	1.005294	$\approx 2^{188.4}$	$\approx 2^{145.7}$
80	80	2^{20}	$\approx 2^{33546.4}$	$\approx 2^{35.0}$	$\approx 2^{35.0}$	1.005528	$\approx 2^{179.7}$	$\approx 2^{143.6}$
80	160	2^{21}	$\approx 2^{69810.9}$	$\approx 2^{37.1}$	$\approx 2^{37.1}$	1.005769	$\approx 2^{171.3}$	$\approx 2^{141.4}$

Table 6. Additional Parameters.

C 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.

    For small ‘k‘ we use estimates based on experiments. For ‘k ⩾ 90‘ we use the asymptotics.

    """
    repeat = 3*log(n, 2) - 2*log(k, 2) + log(log(n, 2), 2)
    if k < 90:
        return RR(0.45*k + 12.31) + repeat
    else:
        # we simply pick the same additive constant 12.31 as above
        return RR(0.3366*k + 12.31) + repeat

def bkz_runtime_k_bkz2(k, n):
    """
    Runtime estimation given ‘k‘ and assuming [CheNgu12]_ estimates are correct.

    The constants in this function were derived as follows based on Table 4 in [CheNgu12]_::

        sage: dim = [100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240,
              250]
        sage: nodes = [39.0, 44.0, 49.0, 54.0, 60.0, 66.0, 72.0, 78.0, 84.0, 96.0, 99.0, 105.0,
              111.0, 120.0, 127.0, 134.0]
        sage: times = [c + log(200,2).n() for c in nodes]
        sage: T = zip(dim, nodes)
        sage: var("a,b,c,k")
        sage: f = a*k*log(k, 2.0) + b*k + c
        sage: f = f.function(k)
        sage: f.subs(find_fit(T, f, solution_dict=True))
        k |--> 0.270188776350190*k*log(k) - 1.0192050451318417*k + 16.10253135200765

    .. [CheNgu12] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates (
         Full Version).
                  2012. http://www.di.ens.fr/~ychen/research/Full_BKZ.pdf


    """
    repeat = 3*log(n, 2) - 2*log(k, 2) + log(log(n, 2), 2)
    return RR(0.270188776350190*k*log(k) - 1.0192050451318417*k + 16.10253135200765 + 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=False, 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
    else:
        sigma_s = 1
    normk = sqrt(n)**(kappa-1) * ((sigma_p)**2 * n**RR(1.5) + 2*sigma_s * sigma_p * n**RR(1.5))
         **kappa
    q_base = RR(n * ell_g * normk)

    if xi is None:
        log_negl = target_lambda
        xivar = var('xivar')
        f = (ell + log_negl) == (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) - log_negl - 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, rerand=False):
    """
    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"]
    sigma_p = params[u"σ'"]

    # NTRU attack
    nt = OrderedDict()
    nt["n"] = n
    nt[u"τ"] = RR(0.3)
    base = (sqrt(q)/(sqrt(2) * sqrt(n)* sigma_p * nt[u"τ"]))
    if rerand:
        base = base/sigma
    nt[u"δ_0"] = RR(base**(1/(2*n)))
    nt = complete_lattice_attack(nt)

    return 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)
        best = gghlite_attacks(params, rerand=kwds.get('rerand', False))

        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"]
        current[u"k"] = best[u"k"]

        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)
```