Implementing Operations in Power-of-2 Cyclotomic Rings

Lattice Meeting

Martin R. Albrecht

2016-04-14

Outline

GGH-like Multilinear Maps

Multiplication

Computing Algebraic Norms

Primality

Inverting in $\mathbb{Q}[X]/(X^n+1)$

Small Remainders

Discrete Gaussians

Approximate Square Roots

Outline

GGH-like Multilinear Maps

- In 2013, Garg, Gentry and Halevi[1] proposed a construction, relying on ideal lattices, of a graded encoding scheme that approximates a cryptographic multilinear map.
- Shortly after, this construction was improved by Langlois, Stéhle and Steinfeld[2].
- Implementing GGH-like schemes naively would not allow instantiating it for non-trivial parameter sizes.

---

[1]Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate Multilinear Maps from Ideal Lattices. In: *EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. LNCS. Springer, Heidelberg, May 2013, pp. 1–17. DOI: `10.1007/978-3-642-38348-9_1`.

[2]Adeline Langlois, Damien Stehlé, and Ron Steinfeld. GGHLite: More Efficient Multilinear Maps from Ideal Lattices. In: *EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. LNCS. Springer, Heidelberg, May 2014, pp. 239–256. DOI: `10.1007/978-3-642-55220-5_14`.

Paper

Martin R. Albrecht, Catalin Cocis, Fabien Laguillaumie, and Adeline Langlois. Implementing Candidate Graded Encoding Schemes from Ideal Lattices. In: *ASIACRYPT 2015, Part II*. ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9453. LNCS. Springer, Heidelberg, 2015, pp. 752–775. DOI: `10.1007/978-3-662-48800-3_31`

Wait, Aren't Those All Broken?

http://malb.io/are-graded-encoding-schemes-broken-yet.html

ATTACKS I

Key Exchange

Yupu Hu and Huiwen Jia. Cryptanalysis of GGH Map. accepted at EUROCRYPT 2016. 2015

- Polynomial-time attack using low-level encodings of zero.[3]

[3] Sage implementation: `https://martinralbrecht.wordpress.com/2015/04/13/`

Attacks II

Attacks without Low-Level Encodings of Zero

Jung Hee Cheon, Jinhyuck Jeong, and Changmin Lee. An Algorithm for NTRU Problems and Cryptanalysis of the GGH Multilinear Map without an encoding of zero. In: *IACR Cryptology ePrint Archive* 2016 (2016). URL: `http://ia.cr/2016/139`

Martin Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched NTRU assumptions: Cryptanalysis of some FHE and Graded Encoding Schemes. In: *IACR Cryptology ePrint Archive* 2016 (2016). URL: `http://ia.cr/2016/127`

- Polynomial-time attack for large levels of multilinearity κ without low-level encodings of zero.
- Subexponential attack for large levels of multilinearity κ without low-level encodings of zero without using the zero-testing parameter.

Indistinguishability Obfuscation

Eric Miles, Amit Sahai, and Mark Zhandry. Annihilation Attacks for Multilinear Maps: Cryptanalysis of Indistinguishability Obfuscation over GGH13. In: *IACR Cryptology ePrint Archive* 2016 (2016). URL: `http://ia.cr/2016/147`

- Polynomial-time attack on several iO constructions[4][5][6]

---

[4] Sanjam Garg et al. Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits. In: *54th FOCS*. IEEE Computer Society Press, Oct. 2013, pp. 40–49.

[5] Boaz Barak et al. Protecting Obfuscation against Algebraic Attacks. In: *EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. LNCS. Springer, Heidelberg, May 2014, pp. 221–238. DOI: `10.1007/978-3-642-55220-5_13`.

[6] Prabhanjan Vijendra Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai. Optimizing Obfuscation: Avoiding Barrington's Theorem. In: *ACM CCS 14*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM Press, Nov. 2014, pp. 646–658.

What good is an implementation of GGH?

GGH-like graded encodings schemes might be broken, but designers of lattice-based schemes might still be tempted to write: "Sample $g \hookleftarrow D_{R,\sigma}$ until $\mathcal{I} = (g)$ is a prime ideal" or "Sample $f \hookleftarrow D_{(g)+c,\sigma}$."

- We work in the m-th cyclotomic ring for m a power of two.
- It has degree $n = m/2$ and we consider the representation $R \simeq \mathbb{Z}[X]/(x^n + 1)$.
- We also consider $R_q \simeq \mathbb{Z}_q[X]/(x^n + 1)$ and $R_g \simeq \mathbb{Z}[X]/(x^n + 1, g)$.

GGHLite: Instance Generation

- **Instance generation.** Given security parameter λ and multilinearity parameter κ, determine scheme parameters n, q, σ, σ', $\ell_{g^{-1}}$, ℓ_b, ℓ as in GGHLite[7]. 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 $z_i \hookleftarrow U(R_q)$ for all $0 < i \leq \kappa$.
  - Sample $h \hookleftarrow D_{R,\sqrt{q}}$ s.t. h and g are co-prime and define the zero-testing parameter $p_{zt} = \left[\frac{h}{g}\prod_{i=1}^{\kappa} z_i\right]_q$.
  - Return public parameters params $= (n, q, \ell)$ and p_{zt}.

[7] Adeline Langlois, Damien Stehlé, and Ron Steinfeld. GGHLite: More Efficient Multilinear Maps from Ideal Lattices. In: *EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. LNCS. Springer, Heidelberg, May 2014, pp. 239–256. DOI: 10.1007/978-3-642-55220-5_14.

GGHLite: Encoding

- **Encode at level-0.** Compute a small representative $e' = [e]_g$ and sample an element $e'' \hookleftarrow D_{e'+\mathcal{I},\sigma'}$. Output e''.
- **Encode in group.** Given parameters params, z_i and a level-0 encoding $e \in R$, output $[e/z_i]_q$.

GGHLite: Arithmetic & Zero-Testing

- **Adding encodings.** 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.** Let $S_1 \subset [\kappa]$, $S_2 \subset [\kappa]$ with $S_1 \cap S_2 = \emptyset$, 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 $u = [u_1 u_2]_q$, an encoding of $[c_1 c_2]_q$ in $S_1 \cup S_2$.
- **Zero testing.** Given parameters params, a zero-testing parameter p_{zt}, and an encoding $u = \left[c / \left(\prod_{i=0}^{\kappa-1} z_i\right)\right]_q$ in the group $[\kappa]$, return 1 if $\|[p_{zt} u]_q\|_\infty < q^{3/4}$ and 0 else.

LIBOZ ⊂ GGHLITE-FLINT

```
gghlite-flint
|- applications   # 0.9k benchmarks, high-level applications, …
|- dgs            # 1.2k discrete Gaussian sampling over the Integers
|- dgsl           # 0.7k discrete Gaussian sampling over lattices
|- flint          # 250k we rely on flint
|- gghlite        # 1.5k instance generation, zero testing …
|- oz             # 2.2k operations in Z[x]/(x^n+1)
|- tests          # 1.1k tests!
```

https://bitbucket.org/malb/gghlite-flint

https://bitbucket.org/malb/dgs

Outline

Options

- Naive multiplication takes $\mathcal{O}\left(n^2\right)$.
- Asymptotically fast multiplication:
  - Reduce to multiplication in $\mathbb{Z}[X]$
  - Schönehage-Strassen algorithm for multiplying large integers in $\mathcal{O}(n \log n \log\log n)$.
  - This is the strategy implemented in FLINT.
  - FLINT has highly optimised implementation of the Schönehage-Strassen algorithm.
- We can also achieve $\mathcal{O}(n \log n)$ by the Number-Theoretic Transform.

Negative Wrapped Convolution

Theorem (Negative Wrapped Convolution)

Let ω_n be an nth root of unity in $\mathbb{Z}_q$ and $\varphi^2 = \omega_n$. Let

$$a = \sum_{i=0}^{n-1} a_i X^i \text{ and } b = \sum_{i=0}^{n-1} b_i X^i \in \mathbb{Z}_q[X]/(X^n + 1).$$

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

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

NTT over machine words: Bit Reversal

```
void _nmod_vec_oz_ntt(mp_ptr rop, const mp_ptr op, const mp_ptr w,
                      const size_t n, const nmod_t q) {
  const size_t k = n_flog(n,2);

  mp_ptr a = _nmod_vec_init(n);

  for (unsigned int i = 0; i < n; i++) {
    unsigned int r;
    r = (bit_reverse_table_256[i & 0xff] << 16) | \
      (bit_reverse_table_256[(i >>  8) & 0xff] << 8) | \
      (bit_reverse_table_256[(i >> 16) & 0xff]);
    r >>= (24 - k);
    a[r] = op[i];
  }
```

NTT over machine words: main loop

```
  mp_ptr b = _nmod_vec_init(n);

  const double ninv = n_precompute_inverse(q.n);
  for(size_t i=0; i<k; i++) {
    const mp_limb_t tkm = ~(((1UL)<<(k-1-i)) - 1);
    for(size_t j=0; j<n/2; j++) {
      const size_t pij = j & tkm;
      mp_limb_t tmp = n_mulmod_precomp(a[2*j+1], w[pij], q.n, ninv);
      b[j]      = n_addmod(a[2*j], tmp, q.n);
      b[j+n/2]  = n_submod(a[2*j], tmp, q.n);
    }
    if(i!=k-1)
      _nmod_vec_set(a, b, n);
  }
  _nmod_vec_set(rop, b, n);
  _nmod_vec_clear(b);
  _nmod_vec_clear(a);
}
```

Avoiding Conversion

- If we do many operations in $\mathbb{Z}_q[X]/(X^n+1)$ we can avoid repeated conversions between coefficient and “evaluation” representation $\left(f(1), f(\omega_n), \ldots, f(\omega_n^{n-1})\right)$
- We convert encodings to their evaluation representation once on creation
- We convert back only when running extraction.
- This reduces the amortised cost from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$.

Outline

Computing Algebraic Norms: Resultants

- During instance generation we have to compute the norm of g.
- We can compute norms in $\mathbb{Z}[X]/(X^n+1)$ by observing that

$$\mathcal{N}(f) = \mathrm{res}(f, X^n+1).$$

Multi-modular Resultants

- The usual strategy for computing resultants over the integers is to use a multi-modular approach.
- 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 $\mathcal{O}\left(M(n)\log n\right)$ operations where $M(n)$ is the cost of one multiplication in $\mathbb{Z}_{q_i}[X]/(X^n+1)$.
- Overall cost $\mathcal{O}\left(n\log^2 n\right)$ without specialisation.

Faster Resultants

- $\mathrm{res}(f, X^n + 1) \bmod q_i$ can be rewritten as

$$\prod_{(X^n+1)(x)=0} f(x) \bmod q_i,$$

  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 $\mathcal{O}(M(n))$ saving a factor of $\log n$.

Source Code: Main Loop

```c
void _fmpz_poly_oz_ideal_norm(fmpz_t norm, const fmpz_poly_t f,
                              const long n) {
  …
#pragma omp parallel for
  for (i = 0; i<num_primes; i++) {
    nmod_t mod;
    nmod_init(&mod, parr[i]);

    const int id = omp_get_thread_num();
    /* reduce polynomials modulo p */
    _fmpz_vec_get_nmod_vec(a[id], F, n, mod);
    /* compute resultant over Z/pZ */
    rarr[i] = _nmod_vec_oz_resultant(a[id], n, mod);
    flint_cleanup();
  }
  …
}
```

SOUCE CODE: RESULTANTS MOD $q \equiv 1 \bmod 2n$

```c
mp_limb_t _nmod_vec_oz_resultant(const mp_ptr a, long n, nmod_t q) {
  const mp_limb_t w_ = _nmod_nth_root(2*n, q.n);
  mp_ptr w = _nmod_vec_init(2*n);
  mp_ptr t = _nmod_vec_init(2*n);

  _nmod_vec_oz_set_powers(w, 2*n, w_, q);
  _nmod_vec_oz_ntt(t, a, w, 2*n, q);

  mp_limb_t acc = 1;
  for(int i=1; i<2*n; i+=2)
    acc = n_mulmod2_preinv(acc, t[i], q.n, q.ninv);

  _nmod_vec_clear(w);
  _nmod_vec_clear(t);
  return acc;
}
```

Outline

GGH-like Multilinear Maps

Multiplication

Computing Algebraic Norms

Primality

Inverting in $\mathbb{Q}[X]/(X^n+1)$

Small Remainders

Discrete Gaussians

Approximate Square Roots

Checking Primality

To check if (g) is prime, compute the norm and check if prime. This is a sufficient but not necessary condition.

```
int fmpz_poly_oz_ideal_is_probaprime(const fmpz_poly_t f,
                                     const long n,
                                     const mp_limb_t *primes) {
    …
    fmpz_t norm;
    fmpz_init(norm);
    fmpz_poly_oz_ideal_norm(norm, f, n, 0);
    r = fmpz_is_probabprime(norm);
    fmpz_clear(norm);
    …
  return r;
}
```

Ruling out Common Factors Quickly

Before computing resultants, check if $\mathrm{res}(g, X^n + 1) \equiv 0 \bmod q_i$ for several "interesting" primes q_i.

```
int fmpz_poly_oz_ideal_is_probaprime(const fmpz_poly_t f,
                                     const long n,
                                     const mp_limb_t *primes) {
  int r = fmpz_poly_oz_ideal_not_prime_factors(f, n, primes);
  if (r) {
    fmpz_t norm;
    fmpz_init(norm);
    fmpz_poly_oz_ideal_norm(norm, f, n, 0);
    r = fmpz_is_probabprime(norm);
    fmpz_clear(norm);
  }
  return r;
}
```

Ruling out Common Factors Quickly

```
int fmpz_poly_oz_ideal_not_prime_factors(const fmpz_poly_t f, long n,
                                         const mp_limb_t *primes) {
  nmod_poly_t a[num_threads], b[num_threads];
  int r[num_threads];

  for(size_t i=0; i<k; i+=num_threads) {
    if (k-i < (unsigned long)num_threads)
      num_threads = k-i;
#pragma omp parallel for
    for (int j=0; j<num_threads; j++) {
      mp_limb_t p = primes[1+i+j];
      fmpz_poly_get_nmod_poly(a[j], f);
      r[j] = nmod_poly_oz_resultant(a[j], n);
    }
    for(int j=0; j<num_threads; j++)
      if (r[j] == 0)
        return r[0];
  }
  return r[0];
}
```

Common Factors

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.

```
int _gghlite_nsmall_primes(const gghlite_params_t self) {
  /* we try about 1% small primes first, where 1% relates to the total
     number of primes needed for multi-modular result */
  const long n = self->n;
  int nsp = ceil((log2(_gghlite_sigma(n)) + log2(n)/2.0)
                 * n/100.0/(FLINT_BITS -1));
  if (nsp < 20)
    nsp = 20;
  return nsp;
}
```

Timings

n	$\log \sigma$	wall time
1024	15.1	0.54s
2048	16.2	3.03s
4096	17.3	20.99s
32768	20.4	1834.99s

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.

Verifying Co-Primality

- When re-randomisation elements are required, then it is necessary that they generate all of (g), i.e.

$$(b_1^{(1)}, b_2^{(1)}) = (g).$$

- When $b_i^{(1)} = \tilde{b}_i^{(1)} g$ for $0 < i \leq 2$ then this is equivalent to

$$(\tilde{b}_1^{(1)}) + (\tilde{b}_2^{(1)}) = \mathbb{Z}[X]/(X^n + 1).$$

- We check the sufficient but not necessary condition

$$\gcd(\mathrm{res}(\tilde{b}_1^{(1)}, X^n + 1),\ \mathrm{res}(\tilde{b}_2^{(1)}, X^n + 1)) = 1,$$

i.e. if the respective ideal norms are co-prime.

Avoiding Resultants

- Perform this check for every candidate pair $(\tilde{b}_1^{(1)}, \tilde{b}_2^{(1)})$.
- Compute two resultants and their gcd: expensive.
- But

$$\gcd(\mathrm{res}(\tilde{b}_1^{(1)}, X^n + 1),\ \mathrm{res}(\tilde{b}_2^{(1)}, X^n + 1)) \neq 1$$

  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.

➡ Check this condition for several “interesting” primes and resample if this condition holds.

Avoiding Resultants

- After having ruled out small common prime factors it is quite unlikely that the gcd of the norms is not equal to one.
- With good probability we will perform this expensive step only once as a final verification.

Improvement

A possible strategy is to sample $m > 2$ re-randomisers $b_i^{(1)}$ and to apply bounds on the probability of m elements $\tilde{b}_i^{(1)}$ sharing a prime factor after excluding small prime factors.

Outline

GGH-like Multilinear Maps

Multiplication

Computing Algebraic Norms

Primality

Inverting in $\mathbb{Q}[X]/(X^n+1)$

Small Remainders

Discrete Gaussians

Approximate Square Roots

GGH Motivation

Instance generation relies on inversion in $\mathbb{Q}[X]/(X^n+1)$.

1. when sampling g we have to check that the norm of its inverse is bounded by ℓ_g.
2. To set up our discrete Gaussian samplers we need to run many inversions in an iterative process.

Inverting in $\mathbb{Q}[X]/(X^n+1)$

- The core idea[8] is similar to the FFT, i.e. to reduce the inversion of f to the inversion of an element of degree $n/2$.
- Since n is even, $f(X)$ is invertible modulo X^n+1 if and only if $f(-X)$ is also invertible.
- By setting

$$F(X^2) = f(X)f(-X) \bmod X^n+1,$$

the inverse $f^{-1}(X)$ of $f(X)$ satisfies

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

[8] Dario Bini, Gianna M. Del Corso, Giovanni Manzini, and Luciano Margara. Inversion of Circulant Matrices over $\mathbb{Z}_m$. In: *International Colloquium on Automata, Languages and Programming*. Vol. 1443. LNCS. Springer, 1998, pp. 719–730.

Inverting in $\mathbb{Q}[X]/(X^n+1)$

- Let
$$f^{-1}(X) = g(X) = G_e(X^2) + XG_o(X^2)$$
and
$$f(-X) = F_e(X^2) + XF_o(X^2).$$
- We obtain
$$F(X^2)(G_e(X^2) + XG_o(X^2)) = F_e(X^2) + XF_o(X^2) \bmod X^n + 1$$
or equivalently
$$\begin{aligned} F(X^2)G_e(X^2) &= F_e(X^2) \pmod{X^n+1}, \\ F(X^2)G_o(X^2) &= F_o(X^2) \pmod{X^n+1} \end{aligned}$$
- Invert $f(X)$ by inverting $F(X^2)$ and multiplying at degree $n/2$.
- Recursively call the inversion of $F(Y)$ modulo $(X^{n/2}+1)$ by setting $Y = X^2$.

GGH Motivation Revisited

Instance generation relies on inversion in $\mathbb{Q}[X]/(X^n+1)$.

1. when sampling g we have to check that the norm of its inverse is bounded by ℓ_g.
2. To set up our discrete Gaussian samplers we need to run many inversions in an iterative process.

GGH Motivation Revisited

Instance generation relies on inversion in $\mathbb{Q}[X]/(X^n+1)$.

1. when sampling g we have to check that the norm of its inverse is bounded by ℓ_g.
2. To set up our discrete Gaussian samplers we need to run many inversions in an iterative process.

Approximates Suffice

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.

Truncation

```c
void fmpq_poly_truncate_prec(fmpq_poly_t op, const mp_bitcnt_t prec) {
  mpq_t *tmp_q = (mpq_t*)calloc(fmpq_poly_length(op), sizeof(mpq_t));
  mpf_t tmp_f; mpf_init2(tmp_f, prec);

  for (int i=0; i<fmpq_poly_length(op); i ++) {
    mpq_init(tmp_q[i]);
    fmpq_poly_get_coeff_mpq(tmp_q[i], op, i);
    mpf_set_q(tmp_f, tmp_q[i]);
    mpq_set_f(tmp_q[i], tmp_f);
  }
  fmpq_poly_set_array_mpq(op, (const mpq_t*)tmp_q, fmpq_poly_length(op));
  …
}
```

Calling `fmpq_poly_set_array_mpq` instead of setting each coefficient one-by-one avoids repeated GCD computations.

Algorithm

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

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

Timings

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

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

- The Jigsaw Generator[9] takes as input elements a_i in $\mathbb{Z}_p$ where $p = \mathcal{N}(\mathcal{I})$ and produces encodings with respect to some S_i.
- This algorithm produces some small representative of the coset a_i modulo (g) from large integers of size $\approx (\sigma\sqrt{n})^n$.

[9] Sanjam Garg et al. Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits. In: *54th FOCS*. IEEE Computer Society Press, Oct. 2013, pp. 40–49.

Small Remainders: HD

- We can use Babai's trick and that g is small, i.e. compute

$$a_i - g \cdot \lfloor g^{-1} \cdot a_i \rceil \text{ in } \mathbb{Q}[X]/(X^n + 1)$$

- To produce sufficiently small elements, we need g^{-1} either exactly or with high precision.
- Computing such a high quality approximation of g^{-1} is prohibitively expensive.

Small Remainders: SD

1. 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.
2. Compute small representatives for all $2^{B \cdot j}$ and a_{ij} using an approximation of g^{-1} with precision B.
3. Multiply 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 approximation of g^{-1} with precision B until its size does not decrease any more.

Outline

Discrete Gaussian Sampling

- We need to sample from the discrete Gaussian $D_{(g),\sigma',c}$ where c is a small representative of a coset of (g).
- Fundamental building block is sampler over the Integers.

https://bitbucket.org/malb/dgs

- Discrete Gaussian sampler over the integers for arbitrary precision using `MPFR` and `double` precision.
- Implements rejection sampling from a uniform distribution with and without table ("online") lookups [10] and Ducas et al's sampler which samples from $D_{\mathbb{Z},k\sigma_2}$ where σ_2 is a constant[11].
- Implementation automatically chooses the best algorithm based on σ, c and τ (tail cut).

---

[10] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In: *40th ACM STOC*. ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, May 2008, pp. 197–206.

[11] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice Signatures and Bimodal Gaussians. In: *CRYPTO 2013, Part I*. ed. by Ran Canetti and Juan A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, Aug. 2013, pp. 40–56. DOI: 10.1007/978-3-642-40041-4_3.

Timings

algorithm	σ	c	prec	samp./s	prec	samp./s
tabulated	10000	1.0	53	660.000	160	310.000
tabulated	10000	0.5	53	650.000	160	260.000
online	10000	1.0	53	414.000	160	9.000
online	10000	0.5	53	414.000	160	9.000
Alg 12 [DDLL13]	10000	1.0	53	350.000	160	123.000

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

Sampling from $D_{(g),\sigma',0}$

- Implemented naively this takes $\mathcal{O}\left(n^3 \log n\right)$ operations even if we ignore issues of precision.
- Following Léo's thesis[12], we implemented a variant of Peikert's sampler[13].

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

[13] Chris Peikert. An Efficient and Parallel Gaussian Sampler for Lattices. In: *CRYPTO 2010*. Ed. by Tal Rabin. Vol. 6223. LNCS. Springer, Heidelberg, Aug. 2010, pp. 80–97.

Sampling from $D_{(g),\sigma',0}$

1. Observe that
$$D_{(g),\sigma',0} = g \cdot D_{R,\sigma' g^{-T}}$$
2. Compute approximate square-root $\sqrt[appr]{\Sigma_2}$ of
$$\Sigma_2 = \sigma'^2 \cdot g^{-T} \cdot g^{-1} - r^2 \text{ with } r = 2 \cdot \lceil \sqrt{\log n}\, \rceil$$
3. Sample a vector $x \leftarrow_{\$} \mathbb{R}^n$ from a standard normal distribution and interpret it as a polynomial in $\mathbb{Q}[X]/(X^n + 1)$.
4. Compute $y = \sqrt[appr]{\Sigma_2} \cdot x$ 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 $D_{\mathbb{Z},r,y_i}$.

Sampling from $D_{(g),\sigma',0}$: Sqrt

1. Compute an approximate square root of

$$\Sigma_2' = g^{-T} \cdot g^{-1}$$

   up to λ bits of precision.
   - Precision: $\log(n) + 4\,(\log(\sqrt{n}\sigma))$ bits.
   - If square root does not converge, double precision and start over.
2. Use this approximate square-root, scaled appropriately, as the initial value from which to start computing a square-root of

$$\Sigma_2 = \sigma'^2 \cdot g^{-T} \cdot g^{-1} - r^2 \text{ with } r = 2 \cdot \lceil \sqrt{\log n}\, \rceil$$

3. Terminate when the square is within distance $2^{-2\lambda}$ to Σ_2.
4. Converges quickly because initial candidate close to target.

Outline

Strategy

- For some input element Σ we want to compute some element $\sqrt[\mathit{appr}]{\Sigma} \in \mathbb{Q}[X]/(X^n+1)$ such that $\| \sqrt[\mathit{appr}]{\Sigma} \cdot \sqrt[\mathit{appr}]{\Sigma} - \Sigma \| < 2^{-2\lambda}$.
- We use iterative methods which iteratively refine the approximation of the square root similar to Newton's method.[14]
- Computing approximate square roots of matrices is a well studied research area with many algorithms known in the literature.[15]
- All algorithms with global convergence invoke approximate inversions in $\mathbb{Q}[X]/(X^n+1)$ for which we call our inversion algorithm.

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

[15] Nicholas J. Higham. Stable iterations for the matrix square root. In: *Numerical Algorithms* 15.2 (1997), pp. 227–242. ISSN: 1017-1398. DOI: `10.1023/A:1019150005407`. URL: `http://dx.doi.org/10.1023/A%3A1019150005407`.

Babylonian only one inversion, which allows lower precision.

Denman-Beavers converges faster in practice and can be parallelised on two cores.[16]

Padé iteration arbitrarily many cores, but workload on each core is greater than Denman-Beavers.[17] Only better for us when more than 8 cores were used.

---

[16] Eugene D. Denman and Alex N. Beavers Jr. The matrix sign function and computations in systems. In: *Applied Mathematics and Computation* 2.1 (1976), pp. 63–94.

[17] Nicholas J. Higham. Stable iterations for the matrix square root. In: *Numerical Algorithms* 15.2 (1997), pp. 227–242. ISSN: 1017-1398. DOI: 10.1023/A:1019150005407. URL: http://dx.doi.org/10.1023/A%3A1019150005407.

Rapid Convergence

- Quadratic convergence does not assure rapid convergence in practice because error can take many iterations to become small enough.
- Speed-up convergence by scaling the operands appropriately in each loop.[18]
- Common scaling scheme: scale by the determinant, i.e. $\operatorname{res}(f, X^n + 1)$ for some $f \in \mathbb{Q}[X]/(X^n + 1)$.
- Computing resultants in $\mathbb{Q}[X]/(X^n + 1)$ reduces to computing resultants in $\mathbb{Z}[X](X^n + 1)$.
- Computing resultants in $\mathbb{Z}[X]/(X^n + 1)$ can be expensive.

[18] Nicholas J. Higham. Stable iterations for the matrix square root. In: *Numerical Algorithms* 15.2 (1997), pp. 227–242. ISSN: 1017-1398. DOI: `10.1023/A:1019150005407`. URL: `http://dx.doi.org/10.1023/A%3A1019150005407`.

Approximate Resultants

- We are only interested in approximate determinant for scaling $\rightarrow$ compute with reduced precision.
- Clear all but the most significant bit for each coefficient's numerator and denominator of f to produce f' and compute $\mathrm{res}(f', X^n + 1)$.
- Reduces the size of the integer representation 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.

Sqrt Timing

prec	n	$\log \sigma'$	it.	wall time	$\log\left(\left(\sqrt[appr]{\Sigma_2}\right)^2 - \Sigma_2\right)$
160	1024	45.8	9	0.4s	-200
160	2048	49.6	9	0.9s	-221
160	4096	53.3	10	2.5s	-239
160	8192	57.0	10	8.6s	-253
160	16384	60.7	10	35.4s	-270

Approximate square roots of $\Sigma_2 = \sigma'^2 \cdot g^{-T} \cdot g - r^2$ 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.

Thank You

Code `https://bitbucket.org/malb/gghlite-flint`

Paper `http://ia.cr/2014/928`