keygen(f, g, F, G, h, seed)

poly_small_mkgauss(rc, f, logn)
[4, 0, 10083, 13098]
poly_small_mkgauss(rc, g, logn)
poly_small_sqnorm(f, logn)
poly_small_sqnorm(g, logn)
poly_small_to_fp(rt1, f, logn)
poly_small_to_fp(rt2, g, logn)
FFT(rt1, logn)
[8, 25, 3245, 3440]
FFT(rt2, logn)
poly_invnorm2_fft(rt3, rt1, rt2, logn)
poly_adj_fft(rt1, logn)
poly_adj_fft(rt2, logn)
poly_mulconst(rt1, fpr_q, logn)
poly_mulconst(rt2, fpr_q, logn)
poly_mul_autoadj_fft(rt1, rt3, logn)
poly_mul_autoadj_fft(rt2, rt3, logn)
iFFT(rt1, logn)
[8, 25, 2872, 3642]
iFFT(rt2, logn)

compute_public.(h2, f, g, logn, tmp2)
[2, 43, 4490, 4514]
solve_NTRU(logn, F, G, f, g, lim, solve_NTRU_break)
[49.680, 133974, 156460] →不夠.

compute_public(h, f, g, logn, tmp)
tt[u] = mq_conv_small(f[u])
h[u] = mq_conv_small(g[u])
mq_NTT(h, logn)
mq_NTT(tt, logn)
h[u] = mq_div_12289(h[u], tt[u])
mq_INTT(h[u], logn)
✓

solve NTRU(*F_upper, *G_upper, *tmp2, logn, *f, *g, lim,)
solve_NTRU_all(logn, f, g, 10, tmp, tmp2)
if logn ≤ 2
loop from depth = logn to depth = 1 or output = 0
solve_NTRU_all(logn, f, g, depth, tmp, tmp2)
else
loop from depth = logn to depth = 3 or output = 0
solve_NTRU_all(logn, f, g, depth, tmp, tmp2)
if loop end
solve_NTRU_all(logn, f, g, 1, tmp, tmp2)
if output = 1
solve_NTRU_all(logn, f, g, 0, tmp, tmp2)
poly_big_to_small(F_upper, tmp, lim, logn)
poly_big_to_small(G_upper, tmp, lim, logn)
modp_ninv(p) return value to p0i
↓
modp_mkgm2(gm, tmp, g, g, logn, primes[0], p, p0i)

loop from 0 to n-1
modp_set(G_upper[u], p) return value to Gt[u]

loop from 0 to n-1
modp_set(f[u], p) return value to ft[u]
modp_set(g[u], p) return value to gt[u]
modp_set(F_upper[u], p) return value to Ft[u]

modp_NTT2_ext(ft, 1, gm, logn, p, p0i)
modp_NTT2_ext(gt, 1, gm, logn, p, p0i)
modp_NTT2_ext(Ft, 1, gm, logn, p, p0i)
modp_NTT2_ext(Gt, 1, gm, logn, p, p0i)

loop from 0 to n-1
① modp_montymul(ft[u], Gt[u], p, p0i)
② modp_montymul(gt[u], Ft[u], p, p0i)
return value to
modp_sub(①, ②, p)

Solve NTRU all (else if depth=0)

modp_mov31(p) return value to p0i

modp_R2(p, p0i) return value to R2

modp_mkgm2(ngm, nigm, logn, primes[u].g, p, p0i) change value of ngm, nigm, pr

loop from u=0 to hn:
- modp_set(zint_one_to_plain(Fp+u), p) return value to Fp[u]
- modp_set(zint_one_to_plain(Gp+u), p) return value to Gp[u]

modp_NTT_ext(Fp, 1, gm, logn-1, p, p0i) change value Fp pont at

modp_NTT_ext(Gp, 1, gm, logn-1, p, p0i) change value Gp pont at

loop from u=0 to n:
- modp_set(zint_one_to_plain(ft+u), p) return value to ft[u]
- modp_set(zint_one_to_plain(gt+u), p) return value to gt[u]

modp_NTT_ext(ft, 1, gm, logn, p, p0i) change value ft pont at

modp_NTT_ext(gt, 1, gm, logn, p, p0i) change value gt pont at

loop from u=0 to u=n (u+=2):
- modp_montymul(Fp[u+1], R2, p, p0i) return value to wFp
- modp_montymul(Gp[u+1], R2, p, p0i) return value to wGp
- modp_montymul(ft[u], wFp, p, p0i) return value to ft[u+1]
- modp_montymul(ft[u], wFp, p, p0i) return value to ft[u+1]
- modp_montymul(ft[u], wGp, p, p0i) return value to gt[u+1]
- modp_montymul(ft[u], wGp, p, p0i) return value to gt[u+1]

modp_iNTT2_ext(ft, 1, igm, logn, p, p0i) change the value ft pont to

modp_iNTT2_ext(gt, 1, igm, logn, p, p0i) change the value gt pont to

memmove(Fp, ft, 2*n*sizeof *ft)

modp_mkgm2(t1, t2, logn, primes[0].g, p, p0i) change value t1, t2 pont to, primes[0].g

modp_NTT2_ext(Fp, 1, t1, logn, p, p0i) change value Fp pont to

modp_NTT2_ext(Gp, 1, t1, logn, p, p0i) change value Gp pont to

loop from u=1 to u=n-1:
- modp_set(f[u], p) return value to t1[u]
- modp_set(-f[u], p) return value to t1[n-u]

modp_NTT2_ext(t1, 1, t1, logn, p, p0i) change value t1 pont at

modp_NTT2_ext(t2, 1, t2, logn, p, p0i) change value t2 pont at

modp_set(f[0], p) return value to t1[0]. t2[0]

loop from u=0 to u=hn-1:
- modp_set(f[u], p) return value to t1[u]
- modp_set(-f[u], p) return value to t1[n-u]

modp_NTT2_ext(t1, 1, t1, logn, p, p0i) change value t1 pont at

modp_NTT2_ext(t2, 1, t2, logn, p, p0i) change value t2 pont at

loop from u=0 to u=hn-1:
- modp_montymul(t2[u], R2, p, p0i) return value to w
- modp_montymul(w, Fp[u], p, p0i) return value to t2[u]
- modp_montymul(w, t1[u], p, p0i) return value to t3[u]

modp_set(g[0], p) return value to t1[0]. t2[0]

loop from u=0 to u=hn-1:
- modp_set(g[u], p) return value to t1[u]
- modp_set(-g[u], p) return value to t1[n-u]

modp_NTT2_ext(t1, 1, t1, logn, p, p0i) change value t1 pont at

modp_NTT2_ext(t2, 1, t2, logn, p, p0i) change value t2 pont at

loop from u=0 to u=hn-1:
- modp_montymul(t2[u], R2, p, p0i) return value to w
- modp_montymul(w, Gp[u], p, p0i)
- modp_montyadd(t2[u], , p) return value to t2[u]
- modp_montymul(w, t1[u], p, p0i)
- modp_montysub(t3[u], , p) return value to t3[u]

modp_mkgm2(t1, t4, logn, PRIMES[0].g, p, p0i) change value t1, t4 pont to, primes[0].g

modp_iNTT2_ext(t2, 1, t4, logn, p, p0i) change value t2 pont to

modp_iNTT2_ext(t3, 1, t4, logn, p, p0i) change value t3 pont to

loop from u=0 to u=n-1:
- modp_norm(t2[u], p) return value to t1[u]
- modp_norm(t3[u], p) return value to t2[u]

align_fpr(tmp, t2) return value to rt3

loop from u=0 to u=n-1:
- fpr_of(mt32_t * t2[u]) return value to rt3[u]

FFT(rt3, logn) change value rt3 pont to

poly_div_autoadj_fft(rt3, rt2, logn) change value rt3 pont to

iFFT(rt3, logn) change value rt3 pont to

loop from u=0 to u=n-1:
- modp_set((mt32_t)fpr_rint(rt3[u]), p) return value to t1[u]

modp_mkgm2(t2, t3, logn, primes[0].g, p, p0i) change value t2, t3 pont to, primes[0].g

loop from u=0 to u=n-1:
- modp_set(f[u], p) return value to t2[u]
- modp_set(g[u], p) return value to t3[u]
- modp_NTT2_ext(t1, 1, t2, logn, p, p0i) change value t1 pont to
- modp_NTT2_ext(t1, 1, t2, logn, p, p0i) change value t1 pont to
- modp_NTT2_ext(t1, 1, t2, logn, p, p0i) change value t1 pont to
- loop from u=0 to u=n-1:
  - modp_montymul(t1[u], R2, p, p0i) return value to kw
  - modp_montymul(kw, t4[u], p, p0i) return to
  - modp_sub(Fp[u], , p) return value to Fp[u]
  - modp_montymul(kw, t4[u], p, p0i) return to
  - modp_sub(Gp[u], , p) return value to Gp[u]
- modp_iNTT2_ext(Fp, 1, t3, logn, p, p0i) change value Fp pont to
- modp_iNTT2_ext(Fp, 1, t3, logn, p, p0i) change value Gp pont to

loop from u=0 to u=n-1:
- modp_norm(Fp[u], p) return value to Fp[u]
- modp_norm(Gp[u], p) return value to Gp[u]

Signature

Top function(output, input) function(output, input)

test_sign_self(f, g, F, G, logn, h_src, tmp)

to_ntt_monty(h, logn)

? { i_shake256_init(rng)
i_shake256_inject(rng, buf, len) (inner_shake256_context *)
i_shake256_flip(rng) }

expand_privkey(expanded_key, f, g, F, G)

"for loop" just one time

i_shake256_init(sc)
i_shake256_inject(sc, msg, sizeof(msg))
i_shake256_flip(sc)
hash_to_point_vartime(sc, hm, logn)
sign_tree(sig, expanded_key, hmtest, msg, seed_nist)
verify_raw(hmtest, sig, h, logn, tt)

hmtest

expand_privkey(expanded_key, f, g, F, G)

smallints_to_fpr(rf, f, logn)
smallints_to_fpr(rg, g, logn)
smallints_to_fpr(rF, F, logn)
smallints_to_fpr(rG, G, logn)
FFT(rf, logn)
FFT(rg, logn)
FFT(rF, logn)
FFT(rG, logn)
poly_neg(rf, logn)
poly_neg(rF, logn)
poly_mulselfadj_fft(g00, logn)
poly_mulselfadj_fft(gxx, logn)
poly_add(g00, gxx, logn)
poly_muladj_fft(g01, b10, logn)
poly_muladj_fft(gxx, b11, logn)
poly_add(g01, gxx, logn)
poly_mulselfadj_fft(g11, logn)
poly_mulselfadj_fft(gxx, logn)
poly_add(g11, gxx, logn)
ffLDL_fft(tree, g00, g01, g11, logn, gxx)
ffLDL_binary_normalize_it(tree, logn, logn)

rf rg rF rG rf rF g00 gxx g00 g01 gxx gxx g01 gxx g11 g11 tree

sign_tree(sig, expanded_key, hm, msg, seed)

i_shake256_init(sc)
i_shake256_inject(sc, msg, sizeof(msg))
i_shake256_flip(sc)
hash_to_point_vartime(sc, x, logn)

? { i_shake256_init(rng)
i_shake256_inject(rng, seed, 48)
i_shake256_flip(rng) }

prng_init(&spc.p, rng)
samp_ctx = &spc
forever loop do_sign_tree(samp_ctx, sig, expanded_key, hm, logn)

rng spc samp_ctx

ffLDL_fft(tree, g00, g01, g11, logn, tmp)

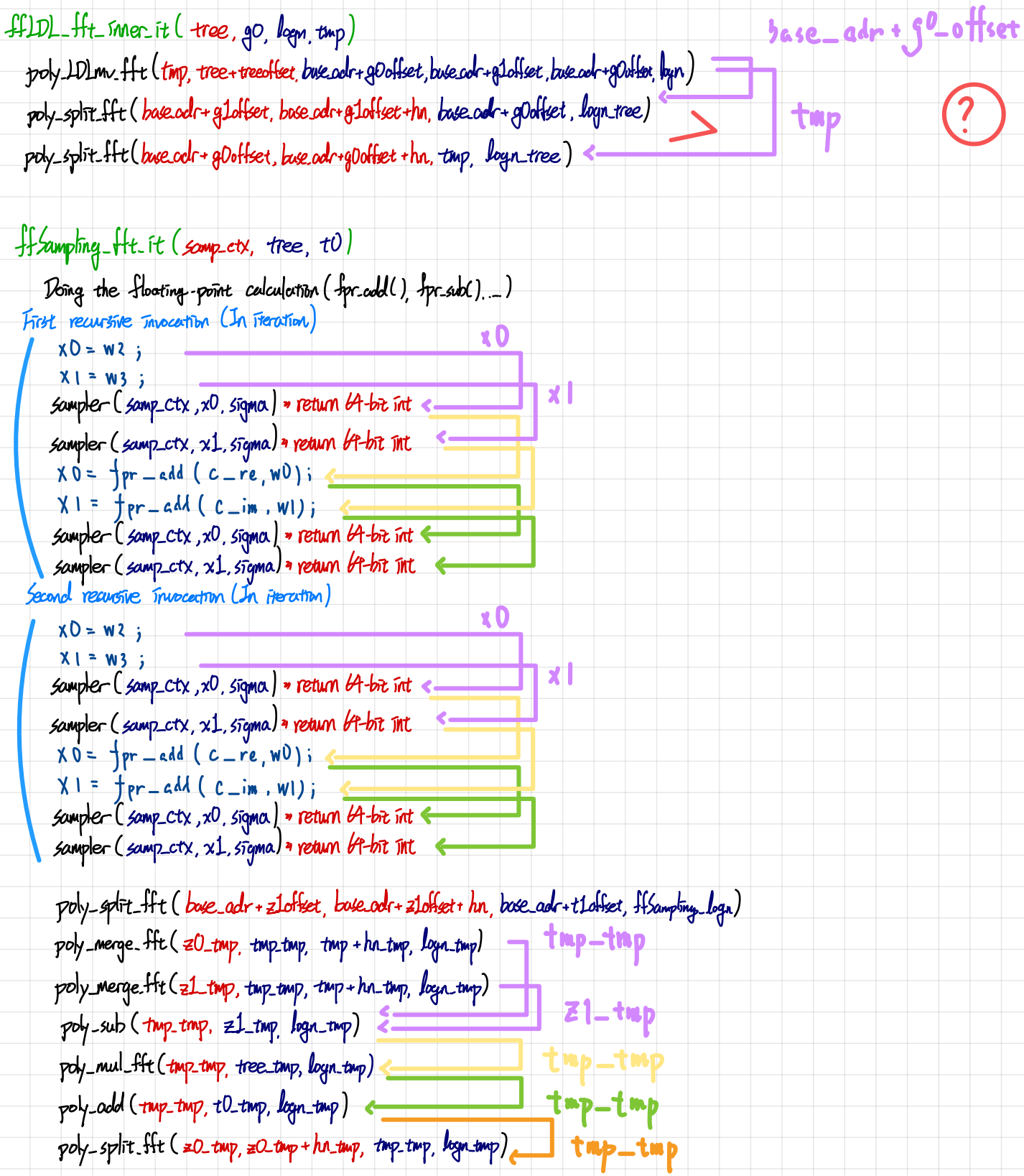
poly_LDLmv_fft(d11, tree, g00, g01, g11, logn)
poly_split_fft(tmp, tmp+hn, d00, logn)
poly_split_fft(d00, d00+hn, d11, logn)
ffLDL_fft_inner_it(tree+n, d00, logn-1, tmp)

d11 d00 d00

do_sign_tree(samp_ctx, s2, expanded_key, hm, logn)

FFT(t0, logn)
poly_mul_fft(t1, b01, logn)
poly_mulconst(t1, fpr_neg(ni), logn)
poly_mul_fft(t0, b11, logn)
poly_mulconst(t0, ni, logn)
ffSampling_fft_it(samp_ctx, tree, t0)
tx = t1 + n
ty = tx + n
poly_mul_fft(tx, b00, logn)
poly_mul_fft(ty, b10, logn)
poly_add(tx, ty, logn)
poly_mul_fft(ty, b01, logn)
poly_mul_fft(t1, b11, logn)
poly_add(t1, ty, logn)
iFFT(t0, logn)
iFFT(t1, logn)
is_short_half(sqn, s2tmp, logn)

t0 t1 t1 t0 t0 tx tx ty ty t1 t1


ffLDL_fft_inner_it(tree, g0, logn, tmp)
poly_LDLmv_fft(tmp, tree+treeoffset, base_adr+g0offset, base_adr+g1offset, base_adr+g0offset, logn)
poly_split_fft(base_adr+g1offset, base_adr+g1offset+hn, base_adr+g0offset, logn_tree)
poly_split_fft(base_adr+g0offset, base_adr+g0offset+hn, tmp, logn_tree)
base_adr + g0_offset
tmp
?
ffSampling_fft_it(samp_ctx, tree, t0)
Doing the floating-point calculation (fpr_add(), fpr_sub()...)
First recursive invocation (In iteration)
x0 = w2;
x1 = w3;
sampler(samp_ctx, x0, sigma) → return 64-bit int
sampler(samp_ctx, x1, sigma) → return 64-bit int
x0 = fpr_add(c_re, w0);
x1 = fpr_add(c_im, w1);
sampler(samp_ctx, x0, sigma) → return 64-bit int
sampler(samp_ctx, x1, sigma) → return 64-bit int
x0
x1
Second recursive invocation (In iteration)
x0 = w2;
x1 = w3;
sampler(samp_ctx, x0, sigma) → return 64-bit int
sampler(samp_ctx, x1, sigma) → return 64-bit int
x0 = fpr_add(c_re, w0);
x1 = fpr_add(c_im, w1);
sampler(samp_ctx, x0, sigma) → return 64-bit int
sampler(samp_ctx, x1, sigma) → return 64-bit int
x0
x1
poly_split_fft(base_adr+z1offset, base_adr+z1offset+hn, base_adr+t1offset, ffSampling_logn)
poly_merge_fft(z0_tmp, tmp_tmp, tmp+hn_tmp, logn_tmp)
poly_merge_fft(z1_tmp, tmp_tmp, tmp+hn_tmp, logn_tmp)
poly_sub(tmp_tmp, z1_tmp, logn_tmp)
poly_mul_fft(tmp_tmp, tree_tmp, logn_tmp)
poly_add(tmp_tmp, t0_tmp, logn_tmp)
poly_split_fft(z0_tmp, z0_tmp+hn_tmp, tmp_tmp, logn_tmp)
tmp-tmp
z1_tmp
tmp_tmp
tmp_tmp
tmp_tmp
sampler(ctx, mu, sigma)
gaussian0_sampler(&spc→p) return 64-bit int
prng_get_u8(&spc→p) return unsigned 8-bit value
BerExp(&spc→p, x, ccs) return integer

test_vrfy_inner (h , sig_in , msg_in , nonce_in) ⇒ Cosim "pass"
[12, 16, 8189, 20630]

h ← ntru_h_1024
sig_in ← KAT_SIG_1024_sig
msg_in ← KAT_SIG_1024_msg
nonce_in ← KAT_SIG_1024_nonce

nonce = temp

nonce_len = hextobin (nonce , tlen , nonce_in)

- i_shake256_init (sc)
- i_shake256_inject (sc , nonce , nonce_len)
- i_shake256_inject (sc , msg_in , 8)
- i_shake256_flip (sc)

sig = tmp

len1 = hextobin (sig , tlen , sig_in)

len1 --

memmove (sig , sig+1 , len1)

s2 = sig + len1

len2 = trim_i16_decode (s2 , logn , 16 , sig , len1)

memmove (tmp , s2 , n * sizeof (s2))

s2 = tmp

h2 = s2 + n

memcpy (h2 , h , n * sizeof (h))

to_ntt_monty (h2 , logn) →
- mq_NTT (h2 , logn)
- mq_poly_tomonty (h2 , logn)

c0 = h2 + n

hash_to_point_vartime (sc , c0 , logn)

return = verify_raw (c0 , s2 , h2 , logn , c0 + n)

verify_raw (c0 , s2 , h , logn , tmp) = "cosim pass"

generate tt

mq_NTT (tt , logn)

mq_poly_montymul_ntt (tt , h , logn)

mq_iNTT (tt , logn)

mq_poly_sub (tt , c0 , logn)

normalize tt

return = is_short (tt , s2 , logn)