`safeRNSgcd`: Accelerating Falcon key generation with RNS in SafeGCD

Cesare Huang and Bo-Yin Yang

Abstract. The deepest level of Falcon's NTRU solver needs only one Bézout coefficient from two large integer resultants already represented in a residue number system (RNS). We give `safeRNSgcd`, which computes this coefficient while keeping the large states in RNS. Its central insight is that centered explicit CRT can recover the exact low bits needed by SafeGCD batches using single-word fixed-point arithmetic. Integrated into `ntrugen`, our C implementation reduces inversion and private-key basis-generation costs on a Raspberry Pi 5 and two x86-64 processors, compared with original `ntrugen` and a variant omitting the unused coefficient.

Keywords: Falcon · SafeGCD · residue number system · explicit CRT · fixed-point arithmetic

1 Introduction

Falcon's recursive NTRU equation solver repeatedly computes field norms, ending with two large integer resultants f_0 and g_0 [PP19, FHK+20]. In the original `ntrugen` implementation, these resultants already arrive in residue number system (RNS) representation [Por25]: each is stored by its residues modulo small, pairwise-coprime primes. The next operation is a Bézout calculation that supplies the starting coefficient for polynomial lifting. The question we study is how to perform this operation while keeping the large states in RNS.

Two observations allow the batch loop to stay in RNS. First, SafeGCD selects k consecutive divsteps from only the low k bits of its operands and an auxiliary scalar δ [BY19, BCH+26]. Its batch selection requires no comparison of the full integers. The selected matrix acts in each RNS coordinate. Second, explicit CRT (ECRT) [BS07] can extract the low bits directly from those coordinates, without rebuilding the large integer. Together, they replace the original high-bit and sign-dependent batch interface by a low-bit interface supplied directly by RNS.

We apply centered ECRT to recover these low bits with single-word fixed-point precision. Bounds on the inversion states provide enough RNS range margin to make this recovery exact; Section 2.3 explains the mechanism.

Implementation and results. Our C implementation `safeRNSgcd` replaces CRT reconstruction and binary-GCD inversion at `ntrugen`'s deepest solver.

The original routine computes both Bézout coefficients, but the first lift needs only $v_\star = -g_0^{-1} \bmod f_0$ [Por25]. Our *half-ntrugen* baseline retains the original CRT reconstruction and binary-GCD arithmetic while removing updates used solely for the unused coefficient. It supplies the same required output as `safeRNSgcd`. Table 1 compares both baselines with `safeRNSgcd` on Raspberry Pi 5 and two x86-64 processors, showing mean cycles and relative cycle changes for inversion and key generation.

Table 1: Inversion and key generation on three platforms. Implementation rows report mean user-mode hardware cycles in millions, to three decimal places. Change rows report percentage changes relative to the named baseline using unrounded means; negative values indicate fewer cycles. Keygen denotes the private-key basis-generation API returning (f, g, F, G).

	Inverse endpoint		Keygen (private-key basis)	
Implementation	Falcon-512	Falcon-1024	Falcon-512	Falcon-1024
Raspberry Pi 5 (Cortex-A76)				
Original ntrugen	2.205	8.644	16.304	73.117
Half-ntrugen	1.557	6.048	15.539	68.509
`safeRNSgcd`	1.027	3.695	14.874	63.989
Change vs. original	−53.42%	−57.25%	−8.77%	−12.48%
Change vs. half	−34.06%	−38.91%	−4.28%	−6.60%
AMD EPYC 9754				
Original ntrugen	1.708	6.651	7.731	40.120
Half-ntrugen	1.244	4.813	7.203	36.697
`safeRNSgcd`	0.576	1.968	6.339	31.044
Change vs. original	−66.28%	−70.40%	−18.00%	−22.62%
Change vs. half	−53.69%	−59.10%	−11.99%	−15.40%
Intel Xeon Silver 4509Y				
Original ntrugen	1.555	6.187	8.020	41.489
Half-ntrugen	1.144	4.503	7.536	38.362
`safeRNSgcd`	0.679	2.358	6.890	33.943
Change vs. original	−56.33%	−61.88%	−14.08%	−18.19%
Change vs. half	−40.62%	−47.63%	−8.57%	−11.52%

Why the original implementation rebuilds the integers. The original Bézout routine uses Pornin's optimized binary GCD [Por20, Algorithm 2]. Its subtraction decision depends on operand ordering, equivalently the sign of $f - g$. For batching, the routine obtains these decisions from high-bit approximations combined with exact low bits. It applies the matrix to the full integers and corrects its rows using the actual output signs. `ntrugen` thus CRT-rebuilds the resultants as positional multiword integers before calling the routine [Por25]. The following schematic abbreviates the coefficient updates.

$(f, g) \leftarrow \mathrm{RebuildCRT}(f_{0,\mathsf{RNS}}, g_{0,\mathsf{RNS}})$.
for each public 31-step batch **do**
  $(\widehat{f}, \widehat{g}) \leftarrow \mathrm{HighLow}(f, g)$.
  $T \leftarrow \mathrm{BinaryBatch}_{31}(\widehat{f}, \widehat{g})$.
  Odd-operand swaps test the sign of $\widehat{f} - \widehat{g}$.
  $(f', g') \leftarrow T(f, g)/2^{31}$ in multiword arithmetic.
  Correct rows of T using the exact signs of f', g'.
  $(f, g) \leftarrow (|f'|, |g'|)$; update Bézout coefficients with T.
end for

Integration into ntrugen. In the original path, `zint_rebuild_CRT` reconstructs f_0, g_0 as positional integers. Binary GCD in `zint_bezout` then computes their Bézout coefficients [Por25]. `safeRNSgcd` instead reads the norm recursion's RNS arrays directly and computes $v_\star = -g_0^{-1} \bmod f_0$ in RNS. Its adapter multiplies by $q = 12289$ and extends $F_0 = qv_\star$ to the RNS basis used by the first polynomial lift. The lift consumes these residues directly, replacing its original conversion of positional F_0 back to RNS. The norm

computation and subsequent polynomial lifting and reduction use the existing `ntrugen` routines.

The inverse columns include each implementation's input conversions, inversion, output normalization, and workspace clearing, starting from the RNS resultants. The keygen columns time private-key basis generation: the `ntrugen` API returning (f, g, F, G), including sampling and rejection retries. Computation of the public key h and key encoding are outside this API. The inverse measurements exclude the adapter's multiplication by q and extension to the first-lift basis; the full keygen API includes these operations. Section 4 gives the measurement protocols and storage costs.

SafeGCD, its stopping bound, and explicit CRT are inherited tools [BY19, BCH+26, saf, BS07]. Kawamura et al. use finite-precision CRT sums and a range margin to recover an exact correction factor for RNS base extension [KKSS00, Section 3]. Bigou and Tisserand apply finite-precision CRT recovery to obtain residues modulo 4 for plus-minus RNS inversion [BT13, Section 4], with a later binary-ternary variant [BT16]. Our contribution combines single-word centered low-bit recovery with full SafeGCD batches and coefficient bounds that maintain its range conditions. Correctness assumes that the incoming residues faithfully represent the resultants; our capacity proof covers the new inversion states and adapter.

Section 2 states the endpoint. Section 3 develops `safeRNSgcd`, its single-word centered recovery, range bounds, and Falcon integration. Section 4 presents measurements.

2 Preliminaries

The construction below uses three established ingredients: Falcon's deepest NTRU-solver boundary, the low-bit batch rule of SafeGCD, and explicit CRT for recovering a selected integer from its residues. We state the interface and the properties needed later, separating them from our batch coefficient construction in Section 3.

2.1 Falcon's deepest solver boundary

From norms to integer resultants. Falcon works in $\mathcal{R}_n = \mathbb{Z}[X]/(X^n + 1)$, where n is 512 or 1024 and $q = 12289$. Key generation samples short f, g and seeks short F, G satisfying

$$fG - gF = q \qquad \text{in } \mathcal{R}_n. \tag{1}$$

Its recursive NTRU solver repeatedly applies a relative field norm [PP19]. Writing $f(X) = f_{\mathrm{e}}(X^2) + X f_{\mathrm{o}}(X^2)$ at degree d, one step produces

$$f'(Y) = f_{\mathrm{e}}(Y)^2 - Y f_{\mathrm{o}}(Y)^2 \pmod{Y^{d/2} + 1}, \qquad f'(X^2) = f(X)f(-X), \tag{2}$$

and similarly for g. The degree halves at each step, ending with the integer resultants $f_0 = \mathrm{Res}(f, X^n + 1)$ and $g_0 = \mathrm{Res}(g, X^n + 1)$. In `ntrugen`, this norm recursion produces the resultants in RNS representation [Por25]. Falcon's RNS approach is described in its specification [FHK+20, Section 4.4.3]. The original deepest routine rebuilds them as positional integers before calling its Bézout routine; our replacement takes the RNS arrays directly. At that boundary, coprimality permits integers $u_\star, v_\star$ with

$$f_0 u_\star - g_0 v_\star = 1. \tag{3}$$

The attempt is rejected if the gcd is not one. Otherwise $(F_0, G_0) = (q v_\star, q u_\star)$ solves the degree-one NTRU equation. Falcon's sampling, norm checks, and coefficient reduction lie outside this deepest integer operation [FHK+20, Section 3.8.2, Algorithm 6].

Why the first lift consumes only F_0. Given $f'G' - g'F' = q$ one level below, the norm identity yields a solution at the current level through

$$F(X) = F'(X^2)g(-X), \qquad G(X) = G'(X^2)f(-X). \tag{4}$$

In particular, the lift of F does not consume G'. Once F is known, a valid solution determines its companion by

$$G = \frac{q + gF}{f} \quad \text{in } \mathbb{Q}[X]/(X^d + 1). \tag{5}$$

The intermediate calls in `ntrugen` retain F alone; their reduction uses the retained states, and final reconstruction supplies G [Por25, `solve_NTRU_intermediate`]. Its original deepest routine computes both coefficients in Equation (3) and discards $u_\star$ [Por25, `solve_NTRU_deepest`]. The first lift needs only

$$v_\star = -g_0^{-1} \bmod f_0, \qquad 0 \le v_\star < f_0, \qquad F_0 = qv_\star. \tag{6}$$

We formulate this operation as negative modular inversion: set $(m, x) = (f_0, g_0)$ for Falcon. The general endpoint takes residues for positive odd integers m, x, with $m, x < P_{\text{in}} < 2^B$, where P_{in} is the input-basis product and B is a public bit bound. These residues must faithfully denote those integers; no assumption $x < m$ is needed. It returns a success flag and, on success, the unique $y \in [0, m)$ such that $xy \equiv -1 \pmod m$. It returns failure and zero if $\gcd(m, x) \neq 1$; for $m = 1$ the successful canonical output is zero. The Falcon adapter must also check that $F_0 = qy$ fits the signed range accepted by the first lift; successful inversion alone does not establish that range. The studied `ntrugen` profiles allot 104 and 207 31-bit coordinates to the resultants, giving positional capacities $B = 3224$ and $B = 6417$ bits. Here B is an integer bound, while n is the polynomial degree. Section 3.4 specifies the adapter and its faithful-input boundary.

2.2 SafeGCD batches

Divsteps and the public stopping bound. SafeGCD uses parity and an auxiliary scalar rather than comparisons of full operands to choose its steps [BY19]. We use the half-delta variant from prior work [BCH+26]. Starting at $(\delta_0, a_0, b_0) = (1/2, m, x)$, one divstep is

$$\text{divstep}(\delta, a, b) = \begin{cases} (1 - \delta,\ b,\ (b - a)/2), & \delta > 0 \text{ and } b \text{ odd}, \\ (1 + \delta,\ a,\ (b + (b \bmod 2)a)/2), & \text{otherwise}. \end{cases} \tag{7}$$

The first operand remains odd, so both displayed divisions are exact. Each step preserves the gcd. If t steps have been run, prior bounds give a public t at which $b_t = 0$ and $a_t = \pm \gcd(m, x)$; the bound is approximately $2.304B$ steps for B-bit inputs [BCH+26, saf]. Further steps leave a, b unchanged. The implemented public counts are derived from the precise prior bound in Section 3.4; neither the divstep recurrence nor that stopping theorem is new here.

Decisions from low bits. The linear action on operands is

$$\begin{pmatrix} a_{t+1} \\ b_{t+1} \end{pmatrix} = U_t \begin{pmatrix} a_t \\ b_t \end{pmatrix}, \qquad U_t = \begin{cases} \begin{pmatrix} 0 & 1 \\ -1/2 & 1/2 \end{pmatrix}, & \delta_t > 0 \text{ and } b_t \text{ odd}, \\ \begin{pmatrix} 1 & 0 \\ (b_t \bmod 2)/2 & 1/2 \end{pmatrix}, & \text{otherwise}. \end{cases} \tag{8}$$

For k consecutive steps, write

$$\begin{pmatrix} a_{t+k} \\ b_{t+k} \end{pmatrix} = M \begin{pmatrix} a_t \\ b_t \end{pmatrix}, \qquad T = 2^k M = \begin{pmatrix} u & v \\ r & s \end{pmatrix} \in \mathbb{Z}^{2\times 2}. \tag{9}$$

Given exact δ_t, the low k bits of a_t, b_t determine every branch in this batch [BY19, Theorem 9.4]. Indeed, two states congruent modulo 2^k make the same first parity test. Addition or subtraction preserves congruence, and exact halving loses at most one bit, so after $j < k$ equal steps the states still agree modulo 2^{k-j} and have the same δ. Induction gives all k decisions. The small-word simulation therefore selects the matrix that acts on the full integers. This is the information an RNS implementation must recover between batches: low bits and exact δ suffice, without magnitude comparisons or sign tests on the large operands.

The scaled matrix and its row bound. Each U_t introduces at most one factor of two in a denominator, which explains why T is integral [BY19, Theorems 9.1–9.2]. A step replaces a row of the unscaled product by an old row or by half a sum or difference of two old rows. Starting from the identity, both absolute row sums remain at most one. Consequently,

$$|u| + |v| \le 2^k, \qquad |r| + |s| \le 2^k. \tag{10}$$

The positive endpoint $+2^k$ is attainable: if $b = 0$, the first operand remains unchanged for the entire batch. This is an inherited property of the divstep matrices.

The same row bound will control the coefficient sizes in our construction. Section 3 initializes two coefficient representatives $(V, S) = (0, 1)$ and adds a correction cm with $0 \le c < 2^k$ to each batch numerator before division by 2^k. The matrix part cannot increase $\max(|V|, |S|)$, and the divided correction contributes less than m. Consequently, after a public number K of batches,

$$|V_{\text{final}}| \le \max\big(|V_{\text{final}}|, |S_{\text{final}}|\big) \le 1 + Km. \tag{11}$$

Since $m < 2^B$, the corresponding public bound $1 + K2^B$ lets us size the RNS basis to leave enough range margin for exact centered low-bit extraction. Section 3.3 gives the coefficient recurrence and the full growth and normalization argument.

2.3 Extracting low bits of a large integer from its RNS representation

SafeGCD asks for the low bits of an integer, including a negative integer. An RNS vector gives only its residues modulo a basis product P. Choosing between congruent representatives matters because P is odd: adding P can change even the parity. We first specify the centered representative, then show how explicit CRT supplies its low bits. The fixed-point calculation in Section 3.1 will recover the required rank exactly.

The CRT sum and the representative. Let the basis be $\mathcal{B} = (p_1, \ldots, p_\nu)$, where the primes are odd and distinct. Write $P = \prod_i p_i$, $P_i = P/p_i$, and let $z_i \in [0, p_i)$ be the coordinates. Set

$$\alpha_i = z_i(P_i^{-1} \bmod p_i) \bmod p_i, \qquad A = \sum_i \alpha_i P_i, \qquad \sigma = \frac{A}{P} = \sum_i \frac{\alpha_i}{p_i}. \tag{12}$$

Each term vanishes modulo every prime except its own, where its residue is z_i. Thus A has the right coordinates. To select the integer closest to zero, subtract the nearest integer number of copies of P [BS07, Theorem 2.1]:

$$\rho = \lfloor \sigma + 1/2 \rfloor, \qquad z = A - \rho P, \qquad -P/2 < z < P/2. \tag{13}$$

There is no tie because P is odd and A is an integer. We call ρ the *centered rank*: it identifies which multiple of P separates the weighted sum from the intended centered integer.

A running example. Take $(p_1, p_2, p_3) = (3, 5, 7)$, so $P = 105$, and represent $z = -2$. Its coordinates are $(1, 3, 5)$, the P_i are $(35, 21, 15)$, and their inverses modulo the respective primes are $(2, 1, 1)$. Therefore

$$(\alpha_1, \alpha_2, \alpha_3) = (2, 3, 5), \qquad A = 2 \cdot 35 + 3 \cdot 21 + 5 \cdot 15 = 208.$$

Since $\sigma = 208/105$, its nearest integer is $\rho = 2$, giving $z = 208 - 2 \cdot 105 = -2$. Subtracting only one copy of P would give 103, which has the same coordinates but different low bits.

Evaluating only the required bits. Once ρ is known, evaluate the centered identity directly modulo a small target h [BS07, Theorem 3.1]:

$$z \bmod h = \left(\sum_i \alpha_i (P_i \bmod h) - \rho (P \bmod h) \right) \bmod h. \tag{14}$$

The weights $P_i \bmod h$ and $P \bmod h$ are public constants. Taking $h = 2^k$ gives the low k bits without constructing A or z. For the example, $(-2) \bmod 16 = 14$; the representative 103 would give 7. Thus low-bit extraction must select the correct rank before cancellation.

The precision needed for the centered rank. In low-precision centered CRT, a known range bound leaves room for approximation error [BS07, Theorem 2.2]. The centered identity gives $\sigma = A/P = \rho + z/P$. If $|z| < P/4$, then σ lies within one quarter of ρ. An approximation with error below another quarter still rounds to the *exact* integer rank. ECRT then subtracts ρP modulo 2^k, giving exact low bits. This range margin permits a single-word fixed-point calculation even when z itself is thousands of bits long.

For our Falcon profiles, 31 fractional bits suffice to select the rank, and 64-bit word arithmetic supplies 62 exact low bits. Recovery visits every RNS coordinate, but each arithmetic operation uses single-word precision. Section 3.1 constructs this approximation and proves both exact rank selection and the machine-word bounds.

3 safeRNSgcd

Our construction keeps the inversion states in RNS throughout each SafeGCD batch. The batch matrix comes from exact low bits recovered by centered ECRT. One multiple of m per coefficient row makes the batch division exact; a linear growth bound keeps every subsequent state within the range required by the same recovery operation. We give the overall procedure first, then explain low-bit recovery, batch updates, range bounds, and integration into Falcon.

States and overall procedure. Let m, x be the positive odd integers specified in Section 2.1. We initialize four RNS states by

$$(a, b, V, S) = (m, x, 0, 1), \qquad a \equiv Vx \pmod{m}, \quad b \equiv Sx \pmod{m}. \tag{15}$$

Only the coefficient multiplying x is needed: at termination, $a = \epsilon = \pm 1$ gives the negative inverse $-\epsilon V \bmod m$. Each coefficient update adds a suitable multiple of m to make its numerator divisible by 2^k. This preserves the congruences above; Section 3.3 shows that the larger coefficient magnitude increases by at most m per batch. Let $\beta = 2^k$, where

Algorithm 1 `safeRNSgcd`: batched inversion with delayed normalization

Require: Faithful residues of positive odd m, x; a public k, K, J and working basis satisfying Section 3.3.

Ensure: $(1, y)$ with $0 \le y < m$ and $xy \equiv -1 \pmod{m}$, or $(0, 0)$.

Initialize

Extend m, x to the working basis by unsigned explicit CRT.
$(a, b, V, S) \leftarrow (m, x, 0, 1)$; $\delta \leftarrow 1/2$; $\beta \leftarrow 2^k$.
$\mu \leftarrow (\mathrm{Low}_k(m))^{-1} \bmod \beta$.

Process K batches

for $j = 0, \ldots, K-1$ **do**
  $(a_L, b_L, V_L, S_L) \leftarrow (\mathrm{Low}_k(a), \mathrm{Low}_k(b), \mathrm{Low}_k(V), \mathrm{Low}_k(S))$.
  $(\delta, T) \leftarrow \mathrm{Batch}_k(\delta, a_L, b_L)$; $T = \left(\begin{smallmatrix} u & v \\ r & s \end{smallmatrix}\right)$.
  $c_V \leftarrow -\mu(uV_L + vS_L) \bmod \beta$.
  $c_S \leftarrow -\mu(rV_L + sS_L) \bmod \beta$.
  for each working prime p_i **do**
    $a_i' \leftarrow (ua_i + vb_i)\beta^{-1} \bmod p_i$.
    $b_i' \leftarrow (ra_i + sb_i)\beta^{-1} \bmod p_i$.
    $V_i' \leftarrow (uV_i + vS_i + c_V m_i)\beta^{-1} \bmod p_i$.
    $S_i' \leftarrow (rV_i + sS_i + c_S m_i)\beta^{-1} \bmod p_i$.
    $(a_i, b_i, V_i, S_i) \leftarrow (a_i', b_i', V_i', S_i')$.
  end for
end for

Check termination and normalize

$\mathrm{ok} \leftarrow [b = 0] \wedge ([a = 1] \vee [a = -1])$.
$\epsilon \leftarrow \mathrm{Select}(-1, 1, [a = -1])$.
$X \leftarrow (K+1)m - \epsilon V$ in RNS.
for $\ell = J-1, \ldots, 0$ **do**
  $Z \leftarrow X - 2^\ell m$ in RNS.
  $X \leftarrow \mathrm{Select}(Z, X, \mathrm{Nonnegative}(Z))$.
end for
$y \leftarrow \mathrm{Select}(X, 0, \mathrm{ok})$.
Clear the workspace and return (ok, y).

$k \ge 1$ is the public batch width. The public counts K, J specify the number of batches and final normalization trials.

Algorithm 1 gives the complete procedure. $\mathrm{Low}_k(z)$ returns the low k bits of the intended integer represented by an RNS state. Batch_k runs k small-word divsteps on those bits, returning the new δ and scaled matrix T. $\mathrm{Select}(A, B, c)$ chooses A when $c = 1$ and B otherwise, coordinatewise for an RNS state. Input extension preserves the values of m, x; $\mathrm{Nonnegative}(z)$ tests whether the represented integer is nonnegative. All primed states in a batch are computed from the old states before being committed. Failure paths use the same public schedule and select zero output.

3.1 Exact low bits from single-word centered ECRT

The operation Low_k must recover the bits of the intended integer, rather than those of another representative with the same residues. We instantiate the low-precision centered CRT approach of Bernstein and Sorenson [BS07, Theorems 2.1–2.2] with public reciprocals and an error bound that permits nearest-integer rounding. The condition $|z| < P/4$ provides the rounding margin; Section 3.3 will ensure it for every loop state.

What the stored integers represent. The quantity to approximate is σ, not the large integer z. Each CRT fraction α_i/p_i lies in $[0, 1)$, so $0 \le \sigma < \nu$. To store an approximation with λ fractional bits, we use an integer F whose interpreted value is $F/2^\lambda$. For example,

the integer 62 at scale 2^5 represents $62/32$, not 62. Keeping fractional information in this integer form lets us perform the calculation with multiplication, addition, and shifts. No floating-point operation is involved.

Replacing division by a public reciprocal. Choose a power-of-two scale $W = 2^t$ and precompute $H_i = \lfloor W/p_i \rfloor$. At runtime the numerator $M_i = \alpha_i H_i$ represents the approximation M_i/W to α_i/p_i. Since $0 \le W/p_i - H_i < 1$, its error is one-sided:

$$0 \le \frac{\alpha_i}{p_i} - \frac{\alpha_i H_i}{W} \le \frac{\alpha_i}{W}. \tag{16}$$

The upper inequality is strict when $\alpha_i > 0$; a zero weight has zero error. Summing at this common scale gives

$$0 \le \sigma - \frac{\sum_i M_i}{W} < \frac{E}{W}, \qquad E = \sum_i (p_i - 1). \tag{17}$$

Thus a larger W reduces the error from the precomputed reciprocals. The division by p_i used to compute H_i is performed when generating the public constants. During inversion, we approximate α_i/p_i by multiplying α_i by this stored reciprocal; the scale adjustment below uses a bit shift.

Retaining a shorter fixed-point value. For repeated low-bit extraction we shorten each term to λ fractional bits, with $\lambda \le t$:

$$f_i = \left\lfloor \frac{M_i}{2^{t-\lambda}} \right\rfloor, \qquad F = \sum_i f_i, \qquad \widetilde{\sigma} = \frac{F}{2^\lambda}. \tag{18}$$

Here f_i is still a *scaled* fraction. Discarding the bottom $t - \lambda$ bits of M_i loses less than $2^{-\lambda}$ in its interpreted value. Together with the reciprocal error, this gives

$$0 \le \sigma - \widetilde{\sigma} < \frac{E}{2^t} + \frac{\nu}{2^\lambda}. \tag{19}$$

There are two sources of error: approximating the reciprocal, and shortening each scaled product. The additions forming F are exact. Only after adding these scaled fractions do we extract an integer rank. Taking the floor of each unscaled fraction first would return zero for every term, since every α_i/p_i is smaller than one.

Why a quarter-unit error is enough. For a loop state with $|z| < P/4$, the centered CRT identity says

$$\sigma = \rho + \frac{z}{P}, \qquad \left|\frac{z}{P}\right| < \frac{1}{4}.$$

Thus σ lies within a quarter of its nearest integer ρ. An approximation with error below another quarter still lies strictly within the half-unit interval that rounds to ρ:

$$|\widetilde{\sigma} - \rho| \le |\widetilde{\sigma} - \sigma| + |z|/P < \frac{1}{2}.$$

The approximation may change the fractional part; it cannot change the nearest integer. For a stored value $F/2^\lambda$, we obtain the *exact* centered rank by

$$\rho = \left\lfloor \frac{F + 2^{\lambda-1}}{2^\lambda} \right\rfloor. \tag{20}$$

The offset is half a unit at the stored scale, and the division by 2^λ is a right shift. For a fixed size of z, increasing P makes $\sigma = \rho + z/P$ closer to ρ. Since rounding changes only at $\rho \pm 1/2$, this leaves more room for approximation error. The condition $|z| < P/4$ leaves room for an error below $1/4$; the fixed-point sum need only meet this error bound.

The 31-fractional-bit instance. Our C implementation chooses $t = 62$ and $\lambda = 31$. Since the primes and weights are smaller than 2^{31}, each of the two errors per term is less than 2^{-31}. With $\gamma_i = \lfloor 2^{62}/p_i \rfloor$, the runtime calculation is

$$F_z = \sum_i \left\lfloor \frac{\alpha_i \gamma_i}{2^{31}} \right\rfloor, \quad \rho = \left\lfloor \frac{F_z + 2^{30}}{2^{31}} \right\rfloor, \\ 0 \le \sigma - F_z/2^{31} < \nu 2^{-30} < 1/4. \tag{21}$$

For the largest working basis, $\nu = 208$, the error bound is below 2^{-22}.

Why single-word arithmetic suffices. For a basis of at most 208 distinct primes with $2^{30} < p_i < 2^{31}$, the error bound above and $|z| < P/4$ make the rank ρ exact. The same calculation fits a 64-bit word: since $\alpha_i < p_i$ and $\gamma_i \le 2^{62}/p_i$, each product is below 2^{62}. Each shifted term is below 2^{31}, and $F_z + 2^{30} < 2^{39}$ for at most 208 coordinates. Thus the rank products, sum, and rounding offset fit a 64-bit word exactly.

For the target-modulus calculation, unsigned 64-bit multiplication, addition, and subtraction give the CRT expression modulo 2^{64}. For $1 \le k \le 62$, reducing that result modulo 2^k preserves the desired bits because 2^k divides 2^{64}. Thus Equation (14) returns the exact low k bits using 64-bit arithmetic throughout. No multiword fixed-point precision is needed.

Continuing the example. For $z = -2$ in the $(3, 5, 7)$ basis, use the smaller illustrative scales $t = 10$, $\lambda = 5$. The weights are still $(2, 3, 5)$, so

$$(\gamma_1, \gamma_2, \gamma_3) = (\lfloor 1024/3 \rfloor, \lfloor 1024/5 \rfloor, \lfloor 1024/7 \rfloor) = (341, 204, 146).$$

Shortening the products to five fractional bits gives

$$(f_1, f_2, f_3) = (\lfloor 682/32 \rfloor, \lfloor 612/32 \rfloor, \lfloor 730/32 \rfloor) = (21, 19, 22).$$

Hence $F = 62$ and $\widetilde{\sigma} = 62/32$ instead of the exact $\sigma = 208/105$. Adding half a scaled unit and shifting gives $\rho = \lfloor (62 + 16)/32 \rfloor = 2$, the same rank as exact CRT. Here the error bound is $12/1024 + 3/32 = 27/256 < 1/4$, and $|z|/P = 2/105 < 1/4$, so both premises of the rounding argument hold. Substitution into Equation (14) still gives the exact low four bits, 14. The fractional approximation has not approximated the output bits.

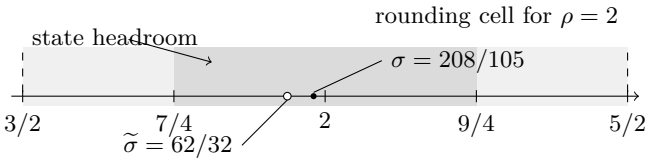

Figure 1: State headroom makes centered rounding robust. The darker interval is $\rho \pm 1/4$, where $|z| < P/4$ places the exact CRT fraction sum. An approximation error below $1/4$ keeps the estimate inside the wider rounding cell $\rho \pm 1/2$. In the running example, both the exact sum and its five-fractional-bit approximation therefore select rank 2, giving the same exact low bits.

Supplying the batch interface. For each $z \in \{a, b, V, S\}$ with $|z| < P/4$, compute the exact centered rank by Equation (21) and use

$$\mathrm{Low}_k(z) = \left(\sum_i \alpha_i (P_i \bmod \beta) - \rho (P \bmod \beta) \right) \bmod \beta, \qquad \beta = 2^k. \tag{22}$$

This returns the integer's low bits, including $\mathrm{Low}_k(-1) = \beta - 1$. Section 3.2 uses the bits of a, b to select the divstep matrix and those of V, S to select the numerator corrections. Only the calculation that selects ρ uses an approximate fractional sum. The products and subtraction in Equation (22) are exact modulo β. Thus 31 fractional bits for the rank and 62 exact output bits serve different purposes: first select the multiple of P, then remove that multiple in the target modulus.

3.2 Exact SafeGCD batches in RNS

The recovered bits now supply the matrix selection and the two coefficient corrections in Algorithm 1. We first identify the intended integer updates, then carry them out in each RNS coordinate.

Selecting the batch matrix. At the start of a batch, a_L, b_L and exact δ select the same k branches as full-integer divsteps by the inherited low-bit property of Section 2.2. The small-word operand values need only retain enough low bits for subsequent decisions; they need not approximate the full magnitudes. Multiplying the inherited small-word divstep matrices gives the scaled T of Equation (9); the full operand updates are

$$a' = \frac{ua + vb}{\beta}, \qquad b' = \frac{ra + sb}{\beta}. \tag{23}$$

Both numerators are divisible by β in the integers because they compose k exact divsteps. The matrix row bounds in Equation (10) will also control the coefficient representatives.

Correcting the coefficient numerators. Although the operand divisions in Equation (23) are exact, $uV + vS$ and $rV + sS$ need not be divisible by β as integers. Multiplying by $\beta^{-1} \bmod p_i$ gives a residue even when its numerator is not divisible by β. To obtain an integer quotient, we first add a multiple of m that makes the numerator divisible by β. Let $\mu = m^{-1} \bmod \beta$ and choose

$$c_V = -\mu(uV_L + vS_L) \bmod \beta, \qquad c_S = -\mu(rV_L + sS_L) \bmod \beta, \qquad 0 \le c_V, c_S < \beta. \tag{24}$$

Since $V_L \equiv V$ and $S_L \equiv S \pmod{\beta}$, the numerator in each row plus its chosen multiple of m is divisible by β. The integer coefficient updates are therefore

$$V' = \frac{uV + vS + c_V m}{\beta}, \qquad S' = \frac{rV + sS + c_S m}{\beta}. \tag{25}$$

The values c_V, c_S are the unique corrections in $[0, \beta)$ making the respective divisions exact. One correction per row is enough for the entire k-divstep batch, independent of the number of RNS coordinates.

For example, set $(m, x, \delta) = (13, 5, 1/2)$, $(V, S) = (0, 1)$, and $k = 2$. The two divsteps produce

$$T = \begin{pmatrix} 0 & 4 \\ -1 & 1 \end{pmatrix}, \qquad (a', b') = (5, -2).$$

The uncorrected second coefficient numerator is 1, not a multiple of four. As $\mu = 1 \bmod 4$, Equation (24) gives $(c_V, c_S) = (0, 3)$, and Equation (25) gives $(V', S') = (1, 10)$. Indeed $5 \equiv 1 \cdot 5$ and $-2 \equiv 10 \cdot 5 \pmod{13}$. Operand batching alone therefore does not produce valid integer representatives for the coefficient state.

Preserving the congruences. Assume Equation (15) holds before the batch. Then $ua + vb \equiv (uV + vS)x \pmod m$, and adding $c_V m$ to the coefficient numerator does not change its residue modulo m. Because β is invertible modulo odd m, division in Equations (23) and (25) yields $a' \equiv V'x \pmod m$. The other row similarly yields $b' \equiv S'x \pmod m$. The invariant is therefore maintained without the coefficients multiplying m in a full Bézout matrix.

Executing the integer quotient in each coordinate. The integer batch makes every update numerator divisible by β. Since every working prime is odd, multiplying by $\beta^{-1} \bmod p_i$ gives the residue of that *integer* quotient. For example,

$$V_i' = (uV_i + vS_i + c_V m_i)\beta^{-1} \bmod p_i = \left(\frac{uV + vS + c_V m}{\beta}\right) \bmod p_i. \tag{26}$$

Canonical reduction modulo each public p_i keeps coordinates bounded; it does not reduce the integer coefficient modulo m. Section 3.3 bounds these quotients so that the next centered extraction selects the same intended integers.

3.3 Maintaining the recovery range

The low-bit construction remains usable if every updated state stays within $(-P/4, P/4)$. The matrix row bound and the chosen coefficient corrections give a public bound on that growth, allowing us to choose the working basis before seeing the inputs.

Linear growth of representatives. Let L_j be the larger absolute coefficient representative after j batches. The first row of Equation (25), together with Equation (10), gives

$$\left|\frac{uV + vS}{\beta}\right| \le \frac{|u| + |v|}{\beta} L_j \le L_j.$$

The correction contributes $c_V m/\beta < m$, and the same argument applies to the other row. Since $L_0 = 1$,

$$L_{j+1} \le L_j + m, \qquad L_j \le 1 + jm. \tag{27}$$

This bounds the selected *integer representatives*, not merely their residues modulo m. The operand updates have the parallel bound $\max(|a'|, |b'|) \le \max(|a|, |b|)$. Hence, for $m, x < 2^B$, all loop operands have magnitude below 2^B and both coefficient representatives have magnitude at most $1 + K2^B$. A working-basis product satisfying

$$4(1 + K2^B) < P \tag{28}$$

therefore keeps all four states in $(-P/4, P/4)$, where Section 3.1 recovers their exact low bits. No secret-modulus reduction is needed during the loop.

Fixed termination and failure. Choose K so that kK reaches the prior half-delta stopping bound for the public input size [BCH+26, saf]. The integer divsteps then give $b = 0$ and $a = \pm\gcd(m, x)$. Extra divsteps after $b = 0$ leave the operands unchanged; they may change V, S, but preserve the congruences and the growth bound. At line 17, equality with $0, 1, -1$ can be tested from complete residue vectors because the operand range excludes a different integer congruent modulo the working-basis product. Thus ok $= 1$ exactly when $\gcd(m, x) = 1$. On success, take $\epsilon = a \in \{-1, 1\}$; Equation (15) gives $Vx \equiv \epsilon \pmod m$ and therefore $-\epsilon Vx \equiv -1 \pmod m$. On failure, the algorithm uses $\epsilon = 1$ to run the same remaining schedule and finally selects zero output.

Fixed canonical normalization. The coefficient bound allows a public offset to make the candidate nonnegative, without first determining the sign or quotient of V:

$$X = (K+1)m - \epsilon V, \qquad 0 \le X \le (2K+1)m + 1, \qquad X \equiv -\epsilon V \pmod{m}. \tag{29}$$

The lower bound is at least $m-1$ because $|V| \le 1 + Km$; the upper bound follows from the same inequality. Pick J with $2K + 2 < 2^J$. Since $m \ge 1$, this implies $X < 2^J m$. For $\ell = J-1, \ldots, 0$, form $Z = X - 2^\ell m$ and replace X by Z exactly when $Z \ge 0$. If $0 \le X < 2^{\ell+1} m$ before a trial, the retained or subtracted value lies in $[0, 2^\ell m)$. Induction leaves $0 \le X < m$ after the last trial. These are exactly J comparisons, with no division by the secret m and no data-dependent number of trials. On success, this X is the unique canonical negative inverse. For $m = 1$, the same reduction yields zero. On a noncoprime input the bound and schedule still apply before the final zero selection.

Capacity for the complete procedure. Every threshold subtracted in normalization is at most $2^{J-1}m$, and subsequent X values only decrease. Thus every trial difference has absolute value bounded by

$$C_{\text{inv}} = \max\big((2K+1)2^B + 1,\ 2^{J-1+B}\big). \tag{30}$$

The public conditions

$$2K + 2 < 2^J, \qquad 4C_{\text{inv}} < P \tag{31}$$

guarantee enough normalization trials and sufficient RNS capacity. The second inequality implies Equation (28) and also puts every X and trial difference within $(-P/4, P/4)$. These capacity conditions depend only on public parameters.

With exact input extension and final comparisons, these bounds ensure that the RNS procedure follows the integer updates throughout. The stopping bound and coefficient invariant give the negative inverse, and the final subtractions select its value in $[0, m)$. Noncoprime inputs follow the same schedule and return zero.

3.4 Falcon integration and public parameters

We instantiate this procedure for the two Falcon input capacities and connect its output to the existing polynomial-lifting code.

Integration point. The replacement begins at `solve_NTRU_deepest` in `ntrugen` [Por25], where norm computation has supplied f_0, g_0 in RNS. Instead of `zint_rebuild_CRT` followed by `zint_bezout`, this routine calls our RNS inverter and supplies its scaled output directly to the first lift. The sampler, norm recursion, and their allocation schedule are retained. For this interface, we assume that those upstream calculations fit the allocated arrays, so their residues faithfully represent the true resultants $m = f_0$ and $x = g_0$. This is a stated input premise, not a global capacity proof of the upstream recursion. The capacity arguments in Section 3.3 establish what our inversion and normalization require after this boundary.

The true resultants are positive and odd under the sampler's input conditions. For $h = f, g$, the sampler enforces $h(1)$ odd. Since $X^n + 1 = (X+1)^n$ over $\mathbb{F}_2$ for the Falcon powers of two, $\text{Res}(h, X^n + 1) \equiv h(1)^n = 1 \pmod 2$. Conjugate roots pair into positive factors $|h(\zeta)|^2$. Thus the true f_0, g_0 meet the positive-odd input contract, provided their upstream RNS representations are faithful.

First-lifting value. The endpoint returns $y \in [0, m)$ with $xy \equiv -1 \pmod{m}$. The adapter multiplies by $q = 12289$ in the working basis and extends qy to the first-lift RNS basis. The public headroom bound below puts qy in the working basis's centered CRT range, so evaluating that centered reconstruction in the added coordinates preserves the nonnegative integer qy. Mathematically, the first lifting value is $F_0 = qy$: the congruence $xy \equiv -1 \pmod{m}$ makes $q + xF_0$ divisible by m, with companion $G_0 = (q + xF_0)/m$. The adapter need not compute G_0 because the subsequent solver recovers it through its existing lifting path.

Public parameters. We use 31-bit RNS primes and set $k = 62$, so $\beta = 2^{62}$. The batch-matrix row bounds allow an entry to equal $+2^k$. Signed $(k+2)$-bit storage is sufficient, and 62 is the largest batch length supported by our signed 64-bit matrix representation. The batch width follows from this matrix-storage bound.

For public input capacity B, the existing general half-delta bound [saf, Formula and Theorem 6] gives

$$N(B) = \left\lfloor \frac{45907B + 30179}{19929} \right\rfloor, \qquad K = \left\lceil \frac{N(B)}{62} \right\rceil. \tag{32}$$

Each batch contains 62 divsteps. The batch and normalization counts are public; Table 2 gives their values.

Table 2: Public parameters of the two C profiles. B is an allocated integer capacity; Falcon size is the polynomial degree. Product and comparison entries are bit lengths of public bounds.

Parameter	Falcon-512	Falcon-1024
Input coordinates	104	207
Working coordinates	105	208
First-lifting output coordinates	155	308
Input capacity bound B	3224	6417
Working product bit length	3255	6448
Largest comparison bound bit length	3238	6431
Existing bound $N(B)$	7428	14783
Batches K	120	239
Total divsteps $62K$	7440	14818
Normalization trials J	8	9

Working basis. The working basis adds one distinct 31-bit prime to the input basis. Both profiles satisfy $P_{\text{in}} < 2^B$, the capacity conditions of Equation (31), and the adapter bound below.

Output range. The adapter requires the public headroom condition

$$4q2^B < P. \tag{33}$$

Because $0 \le y < m < 2^B$, this puts both qy and $qy - 2^{B-1}$ within the working basis's centered sign-recovery range. Comparing the latter difference with zero enforces the solver's positive signed B-bit capacity:

$$F_{\text{adapt}} = \begin{cases} qy, & 0 \le qy < 2^{B-1}, \\ \text{range failure}, & qy \ge 2^{B-1}. \end{cases}$$

On acceptance, $F_{\text{adapt}} = F_0$, so the deepest NTRU relation follows from the inverse calculation. For $m = 1$, the convention $y = 0$ supplies $(F_0, G_0) = (0, q)$.

Key generation rejects attempts with noncoprime resultants or an out-of-range first-lift value.

4 Evaluation

4.1 Performance

Table 1 reports inversion and private-key basis-generation costs for original ntrugen, half-ntrugen, and `safeRNSgcd` on all three target processors.

Measurement boundaries. The inverse endpoint has the same timing boundary on all three platforms. Timing starts from captured RNS resultants and includes the required conversions, inversion, output normalization, and workspace clearing. Input loading and output validation are outside timing. Keygen times the complete private-key basis-generation API returning (f, g, F, G), including sampling, rejection retries, the recursive solver, lifting and reduction, final reconstruction, and internal clearing. Public-key computation, key encoding, and signing are outside this API.

Platforms. The Raspberry Pi 5 uses a Cortex-A76 core and Clang 19.1.7 with `-O3 -mcpu=cortex-a76`. The timed process is pinned to core 3; the `performance` governor maintains 2.4 GHz with no recorded throttling or undervoltage events. The AMD EPYC 9754 and Intel Xeon Silver 4509Y use Clang 18.1.3 with `-O3 -march=x86-64-v3 -mtune=generic`, with AVX2 enabled. Both x86 processes are pinned to CPU 2. Their governors are `schedutil` and `powersave`, respectively. The three implementations use the same build options on each platform. Automatic vectorization is enabled and LTO is disabled on all three platforms. The PMU counts user-mode cycles without multiplexing or measurement-overhead subtraction.

Sample counts. All three platforms use the same captured inverse inputs. Table 3 reports the input and timed call counts for each implementation. Repeated measurements reuse the input corpus; averages include noncoprime failures. Initialization and external output checks are outside timing. Mean costs are total cycles divided by timed inverse calls or completed key generations; percentage changes use the unrounded means.

Table 3: Sample counts for each implementation. Paired counts give Falcon-512 / Falcon-1024.

	Raspberry Pi 5	Each x86 host
Distinct inverse inputs	4,212 / 1,657	4,212 / 1,657
Timed inverse calls	12,636 / 4,971	16,848 / 6,628
Timed keygen calls	3,840 / 960	56 / 28

Reading the results. Table 1 shows gains over half-ntrugen on all three platforms, beyond omitting the unused Bézout coefficient. Basis-generation gains are smaller because the workflow also includes sampling, rejection retries, and polynomial operations.

4.2 Space

Table 4 reports the endpoint workspace and additional public constants linked into the measured executable. The storage figures cover an implementation supporting both Falcon sizes.

Table 4: Workspace and additional public constant storage in bytes. The shared ntrugen prime table is excluded from the constants.

		Additional constants			
Implementation	Workspace	512	1024	Shared	Total
ntrugen	6,624	0	0	0	0
half-ntrugen	4,968	0	0	0	0
`safeRNSgcd`	6,752	142,028	545,108	4,816	691,952

The additional storage is primarily public CRT weights and reciprocal tables shared across calls. Our implementation targets high-end processors, where we consider these storage requirements an acceptable tradeoff for the measured improvements in inversion and key generation. Workspace denotes the endpoint's allocated workspace, excluding compiler temporaries and the full key generator's peak stack.

References

[BCH+26] Daniel J. Bernstein, Han-Ting Chen, John R. Harrison, Cesare Huang, Gregory Maxwell, Bow-Yaw Wang, Pieter Wuille, and Bo-Yin Yang. Accelerating and verifying constant-time modular inversion. In Joan Daemen and Emmanuel Thomé, editors, *Advances in Cryptology – EUROCRYPT 2026, Part VII*, volume 16547 of *Lecture Notes in Computer Science*, pages 598–627. Springer, 2026.

[BS07] Daniel J. Bernstein and Jonathan P. Sorenson. Modular exponentiation via the explicit Chinese remainder theorem. *Mathematics of Computation*, 76(257):443–454, 2007.

[BT13] Karim Bigou and Arnaud Tisserand. Improving modular inversion in RNS using the plus-minus method. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 233–249. Springer, 2013.

[BT16] Karim Bigou and Arnaud Tisserand. Binary-ternary plus-minus modular inversion in RNS. *IEEE Transactions on Computers*, 65(11):3495–3501, 2016.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, 2019.

[FHK+20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier lattice-based compact signatures over NTRU. Specification v1.2, 2020. https://falcon-sign.info/falcon.pdf.

[KKSS00] Shinichi Kawamura, Masanobu Koike, Fumihiko Sano, and Atsushi Shimbo. Cox-Rower architecture for fast parallel Montgomery multiplication. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 523–538. Springer, 2000.

[Por20] Thomas Pornin. Optimized binary GCD for modular inversion. Cryptology ePrint Archive, Paper 2020/972, 2020.

[Por25] Thomas Pornin. ntrugen: NTRU key pair generation. https://github.com/pornin/ntrugen, 2025. Software revision `1754115`, 3 July 2025.

[PP19] Thomas Pornin and Thomas Prest. More efficient algorithms for the NTRU key generation using the field norm. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography – PKC 2019, Part II*, volume 11443 of *Lecture Notes in Computer Science*, pages 504–533. Springer, 2019.

[saf] safegcd-bounds project. Generic formula for hddivsteps. https://github.com/sipa/safegcd-bounds. Proof document `genproofhd.md`, revision `06abb7f`; accessed 21 September 2026.