PQ SHIELD

A Secure Fixed-Point Implementation of Falcon

Thomas Prest
Joint work with Daniel de Almeida Braga, Pierre-Alain Fouque & Bachir Lachguel (CRYPTO 2026)

ASIAPQC 2026

Why This Talk?

> NIST understands that some applications will not work as they are currently designed if the signature and the data being signed cannot fit in a single internet packet. [...] For this reason, NIST decided to standardize FALCON as well. Given FALCON's overall better performance when signature generation does not need to be performed on constrained devices, many applications may prefer to use FALCON over Dilithium [...].
>
> *NIST IR 8413 · Status Report on the Third Round of the PQC Standardization Process (2022)*

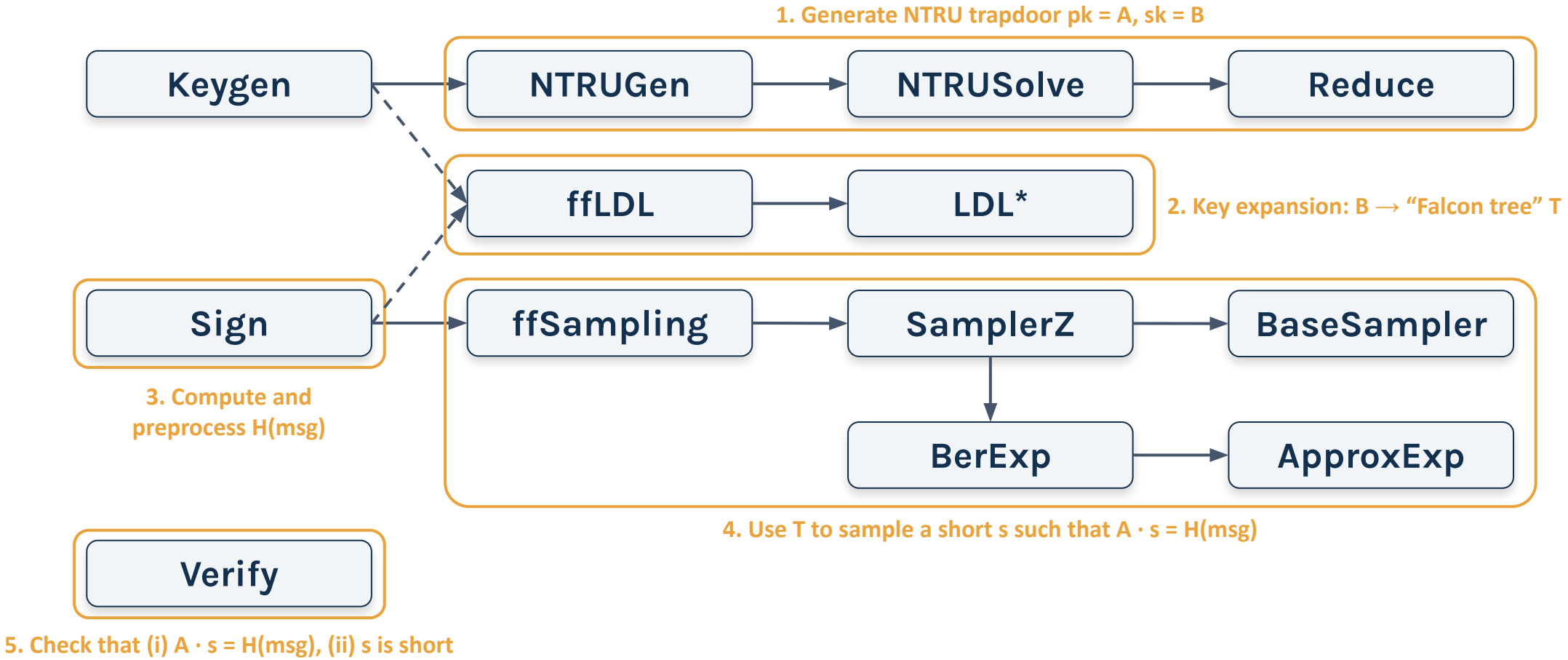
> Falcon, crucially, is the first big cryptographic algorithm to use double-precision floating-point arithmetic. [...] Falcon's constant-timeness is built on shaky grounds. The next generation of Intel CPUs might add an optimization that breaks Falcon's constant-timeness. Also, many CPUs today do not even have fast constant-time double-precision operations. [...] In time, it might be figured out how to do constant-time arithmetic on the FPU robustly, but we feel it's too early to deploy Falcon where the timing of signature minting can be measured. Notwithstanding, Falcon is a great choice for offline signatures such as those in certificates.
>
> *Bas Westerbaan (Cloudflare) · "NIST's pleasant post-quantum surprise"*

01

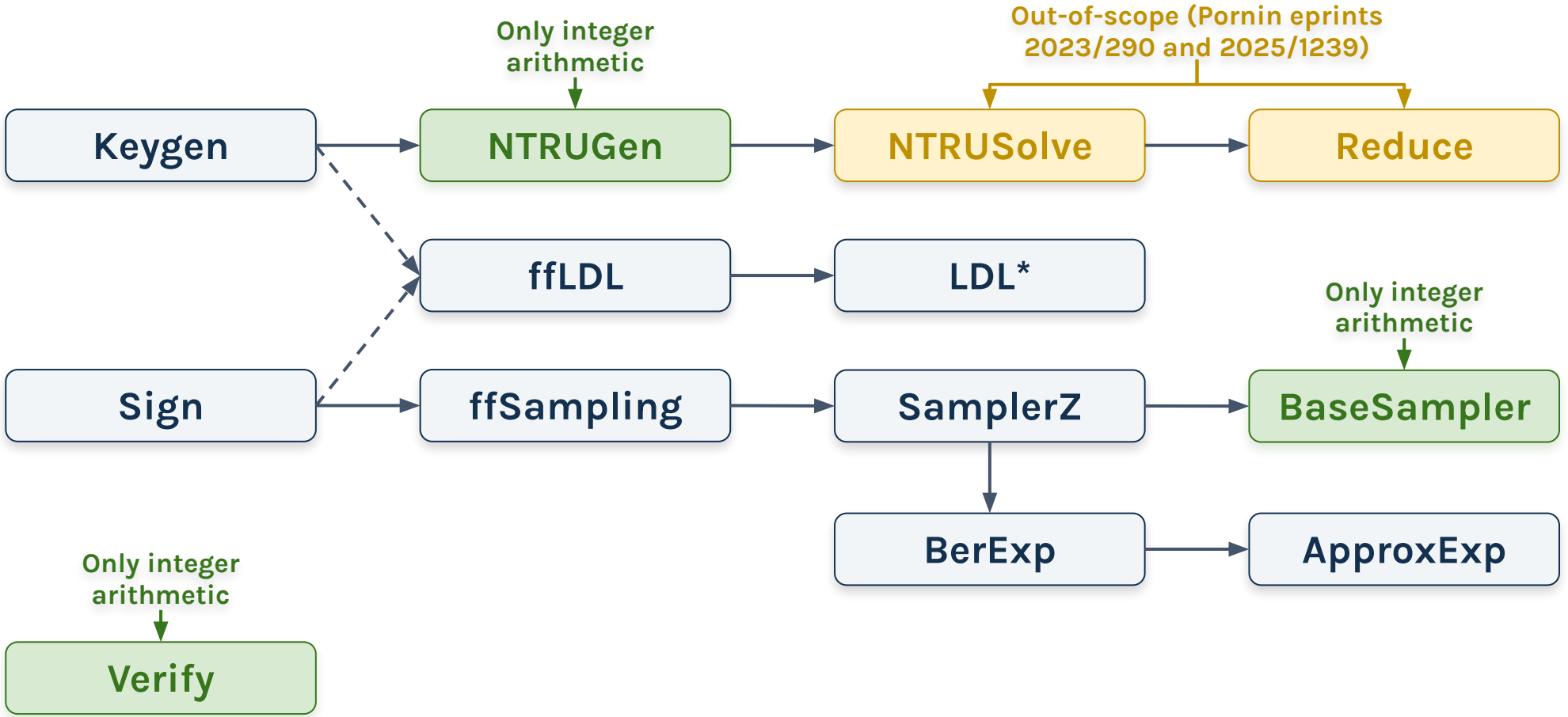
Falcon

What it does and where it uses floating-point numbers

Falcon: a Bird's Eye View

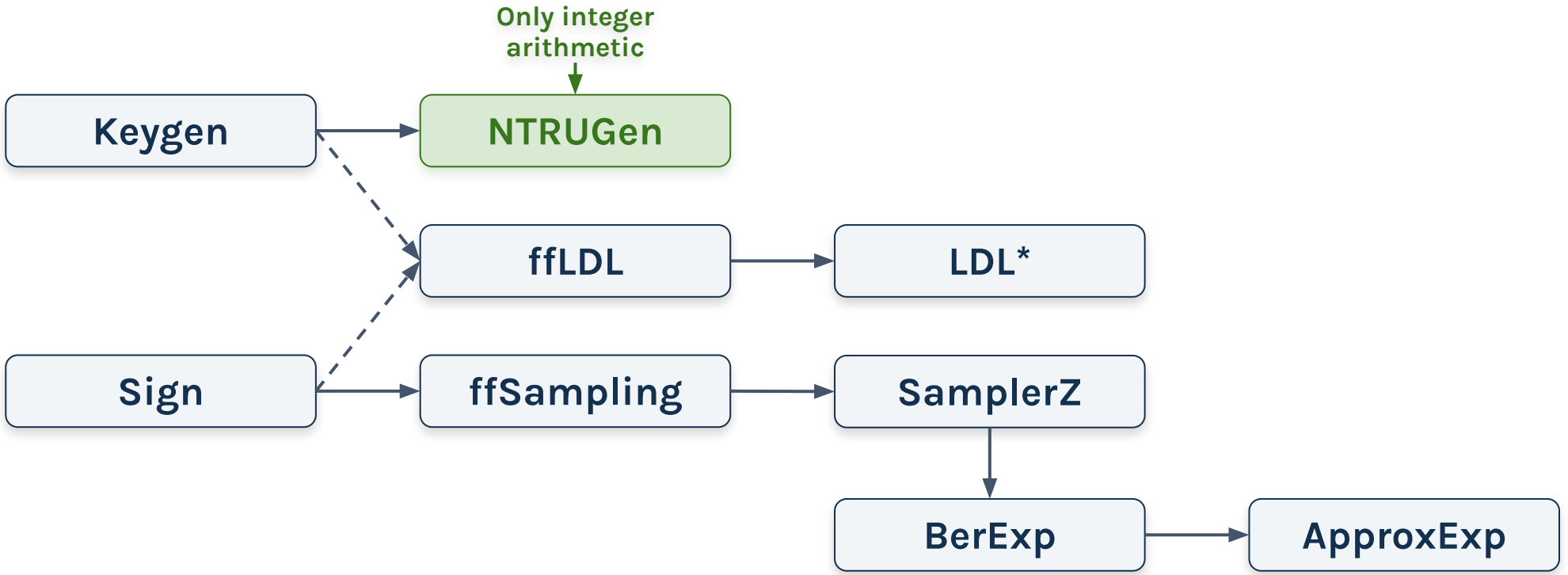


Falcon: Usage of Real Numbers



Color Code: No real number | Out of scope | FxP/FP add, mul, inv, sqrt

Falcon: Floating-Point Usage



We only focus on a subset of the algorithms.
Next steps: which floating-point operations are performed exactly?

Representing Real Numbers

Floating-point arithmetic (IEEE-754)

Values are approximated by $x = (-1)^{S} \cdot 2^{E-1023} \cdot (1 + 2^{-52} \cdot M)$:

- S is the sign bit (1 bit)
- E is the exponent (11 bits)
- M is the mantissa (52 bits)

A few special cases (sub-normals, NaN, etc.) but these are not used by Falcon.

Fixed-point arithmetic

Values are approximated by $x = 2^{-f} \cdot M$, where:

- M is the raw value (signed, p-bit integer)
- f is the scaling factor (known at compile time)
- p is the precision (global constant)

In this talk, p = 64.

Why fixed-point rather than floating-point arithmetic?

Simpler to implement / Faster and more portable / Easier to protect (e.g. constant-time and masking)
FxP is essentially integer arithmetic with extra steps

Example: Representing $\pi = 3.14159265359\ldots$

Using decimal basis for educational purposes

Optimal	Memory usage is optimal, no bit is wasted. High precision.
Underflow	Leading zeroes, memory is wasted on trivial information, precision is degraded.
Overflow	Arithmetic overflow. Leading bits are lost, incorrect.

f=7

f=3

f=9

Easy Arithmetic Operations

Addition

We only add numbers with the same f.

- FxP(x, f) + FxP(y, f) = FxP(x + y, f)

Example with π + e:

	3	1	4	1	5	9	2	6	f=7
+	2	7	1	8	2	8	1	8	f=7
=	5	8	5	9	8	7	4	7	f=7

Multiplication

(i) Multiply raw integers, (ii) drop LSBs, (iii) shift the point.

- FxP(x, fx) · FxP(y, fy) = FxP((x · y) >> p, fx + fy - p)

Example with π · e:

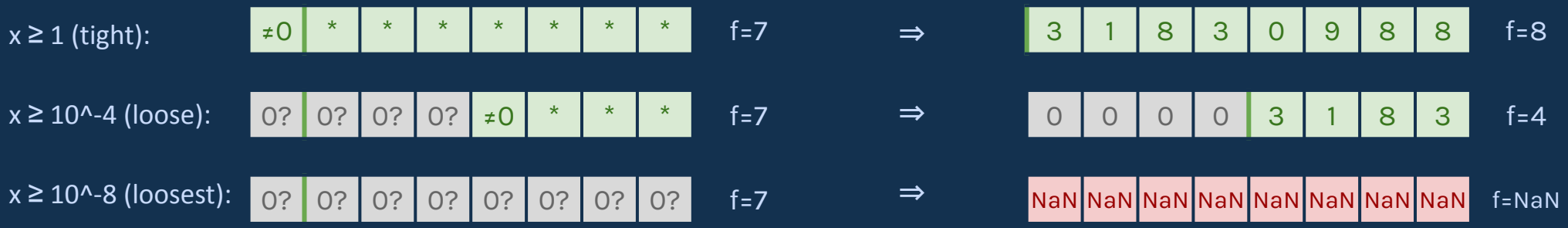
	3	1	4	1	5	9	2	6	f=7
·	2	7	1	8	2	8	1	8	f=7
=	0	8	5	3	9	7	3	4	f=6

Hard Arithmetic Operations

Two hard operations: Inversion & Square Root. For concision, we only present Inversion.

Inversion

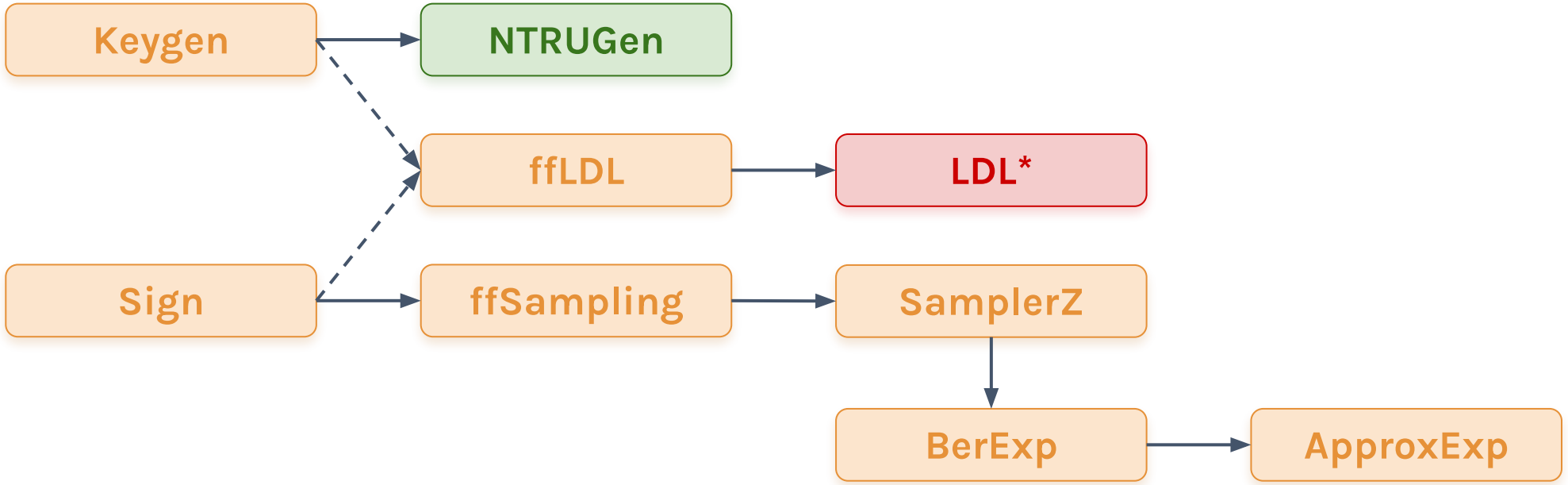
When inverting an (unknown at compile-time) value x, correctness requires to upper-bound and lower-bound x.
Example with x = π = 3.14159265359…, so that 1/x = 0.31830988618… We know that x < 10 and:



The computation itself can be tricky. Options:

- **Integer inversion** ⇒ difficult to protect against SCA
- **Newton-Raphson:** Iterate a few times $y_{n+1} = y_n \cdot (2 - x \cdot y_n)$.
  - **Caveat 1:** y_0 needs to be inside the interval of convergence $(0, 2/x)$
  - **Caveat 2:** renormalization helps with convergence, but is costly when protected against SCA

Falcon: Floating-Point Operations



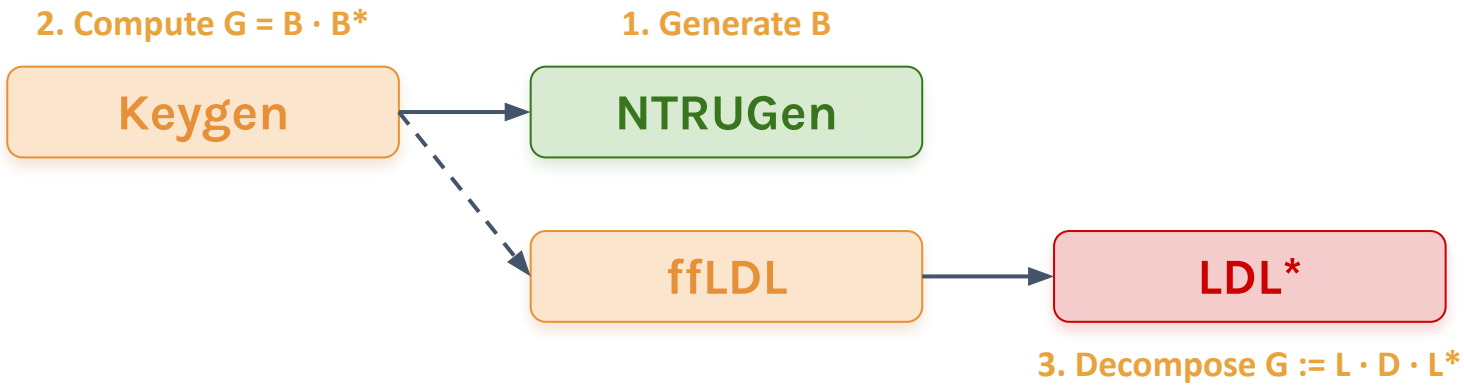
Color Code: No FxP/FP | FxP/FP add and mul | FxP/FP add, mul, inv, sqrt

02

Key Generation & Key Expansion

Rejection sampling & symplecticity save the day

Key Generation & Key Expansion



This section

- Handle fixed-point operations in ffLDL and LDL*, especially inversion and square root.
- We still care about NTRUGen, as changes in NTRUgen will benefit ffLDL and LDL*.

ffLDL and LDL*

- ffLDL is recursive: at each level it calls LDL*, then does two recursive calls to itself
- LDL* performs an inversion & (at bottom) a square root.

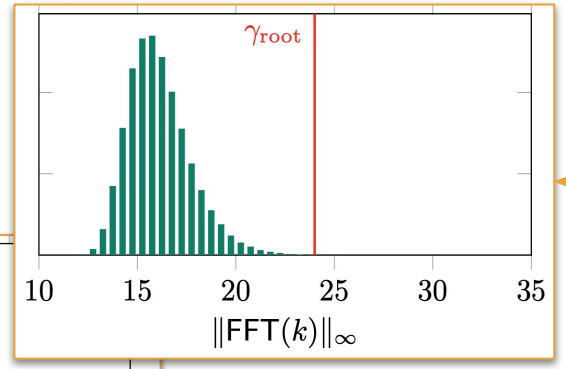
LDL*

Algorithm 3 $\mathrm{LDL}^{\star}(\hat{\mathbf{G}})$

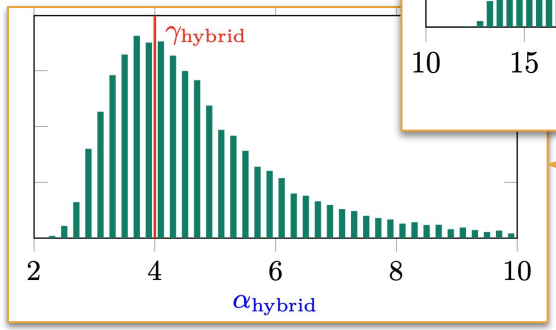
Require: A full-rank self-adjoint matrix $\hat{\mathbf{G}} = (\hat{G}_{ij}) \in \mathsf{FFT}(\mathbb{Q}[x]/(\phi))^{2\times 2}$
Ensure: The $\mathrm{LDL}^{\star}$ decomposition $\hat{\mathbf{G}} = \hat{\mathbf{L}} \odot \hat{\mathbf{D}} \odot \hat{\mathbf{L}}^{\star}$ over $\mathsf{FFT}(\mathbb{Q}[x]/(\phi))$
Format: All polynomials are in FFT() representation.

1: $\hat{D}_{00} \leftarrow \hat{G}_{00}$
2: $\hat{L}_{10} \leftarrow \hat{G}_{10}/\hat{G}_{00}$
3: $\hat{D}_{11} \leftarrow \hat{G}_{11} - \hat{L}_{10} \odot \hat{L}_{10}^{\star} \odot \hat{G}_{00}$
4: $\hat{\mathbf{L}} \leftarrow \left[\begin{array}{c|c} 1 & \hat{0} \\ \hline \hat{L}_{10} & \hat{1} \end{array}\right], \hat{\mathbf{D}} \leftarrow \left[\begin{array}{c|c} \hat{D}_{00} & \hat{0} \\ \hline \hat{0} & \hat{D}_{11} \end{array}\right]$
5: **return** $(\hat{\mathbf{L}}, \hat{\mathbf{D}})$

Check D



Check B



Critical step (highlighted): Line 2 of LDL*

We need to ensure that: (i) G00, G10, G11, L10 are upper-bounded, (ii) G00 is lower-bounded. These are all functions of **B**.

1. **Item (i):** guaranteed by construction. Furthermore, we obtain more aggressive upper-bounds by using rejection sampling in NTRUGen.
2. **Item (ii):** we use symplecticity (a form of symmetry) to convert upper-bounds into lower-bounds on G00.
3. Furthermore, we show that |L10| < 1

03

Signing

Bounding variables via Gaussian tails & inner products

ffSampling

Algorithm 8 $\mathsf{ffSampling}_k(\hat{\mathbf{t}}, \mathsf{T})$

Require: $\hat{\mathbf{t}} = (\hat{t}_0, \hat{t}_1) \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^k+1)\right)^2$, a FALCON tree T
Ensure: $\hat{\mathbf{z}} = (\hat{z}_0, \hat{z}_1) \in \mathsf{FFT}\left(\mathbb{Z}[x]/(x^k+1)\right)^2$
Format: All polynomials are in FFT representation.

1: **if** $k = 1$ **then**
2: $\quad \sigma' \leftarrow \mathsf{T.value}$ ▷ It holds that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$.
3: $\quad \hat{z}_0 \leftarrow \mathsf{SamplerZ}(\hat{t}_0, \sigma')$ ▷ For $k = 1$, $(\hat{t}_i, \hat{z}_i) = t_i, z_i$. Assume $z_0 \sim D_{\mathbb{Z},\sigma',t_0}$
4: $\quad \hat{z}_1 \leftarrow \mathsf{SamplerZ}(\hat{t}_1, \sigma')$ ▷ Assume $z_1 \sim D_{\mathbb{Z},\sigma',t_1}$
5: $\quad$ **return** $\hat{\mathbf{z}} = (\hat{z}_0, \hat{z}_1)$
6: $(\hat{\ell}, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T.value}, \mathsf{T.leftchild}, \mathsf{T.rightchild})$
7: $\hat{\mathbf{t}}_1 \leftarrow \mathsf{splitfft}(\hat{t}_1)$ ▷ $\hat{\mathbf{t}}_1 \in \mathsf{FFT}\left(\mathbb{Q}[x]/(x^{k/2}+1)\right)^2$, likewise $\hat{\mathbf{t}}_0$ below. ▷ `fadd`, `fcmul`.
8: $\hat{\mathbf{z}}_1 \leftarrow \mathsf{ffSampling}_{k/2}(\hat{\mathbf{t}}_1, \mathsf{T}_1)$ ▷ First recursive call to $\mathsf{ffSampling}_{k/2}$
9: $\hat{z}_1 \leftarrow \mathsf{mergefft}(\hat{\mathbf{z}}_1)$ ▷ $\hat{\mathbf{z}}_1 \in \mathsf{FFT}\left(\mathbb{Z}[x]/(x^{k/2}+1)\right)^2$, likewise $\hat{\mathbf{z}}_0$ below. ▷ `fadd`, `fcmul`.
10: $\hat{t}'_0 \leftarrow \hat{t}_0 + (\hat{t}_1 - \hat{z}_1) \odot \hat{\ell}$ ▷ `fadd`, `fmul`.
11: $\hat{\mathbf{t}}_0 \leftarrow \mathsf{splitfft}(\hat{t}'_0)$ ▷ `fadd`, `fcmul`.
12: $\hat{\mathbf{z}}_0 \leftarrow \mathsf{ffSampling}_{k/2}(\hat{\mathbf{t}}_0, \mathsf{T}_0)$ ▷ Second recursive call to $\mathsf{ffSampling}_{k/2}$
13: $\hat{z}_0 \leftarrow \mathsf{mergefft}(\hat{\mathbf{z}}_0)$ ▷ `fadd`, `fcmul`.
14: **return** $\hat{\mathbf{z}} = (\hat{z}_0, \hat{z}_1)$

Base case:
Two calls to SamplerZ

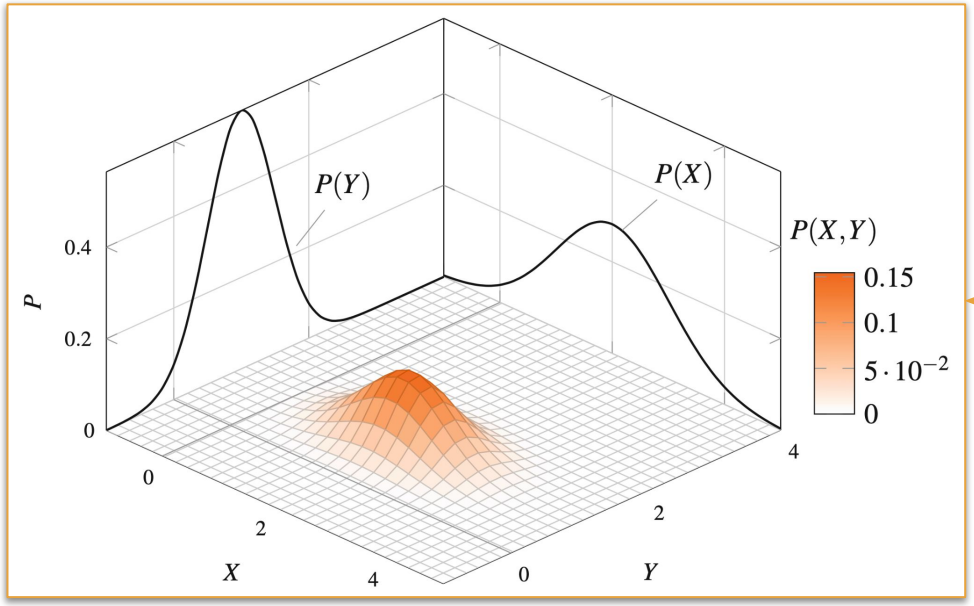
General case:
Two recursive self-calls

ffSampling

Algorithm 8 $\text{ffSampling}_k(\hat{\mathbf{t}}, \mathsf{T})$

Require: $\hat{\mathbf{t}} = (\hat{t}_0, \hat{t}_1) \in \text{FFT}\left(\mathbb{Q}[x]/(x^k+1)\right)^2$, a FALCON tree T
Ensure: $\hat{\mathbf{z}} = (\hat{z}_0, \hat{z}_1) \in \text{FFT}\left(\mathbb{Z}[x]/(x^k+1)\right)^2$
Format: All polynomials are in FFT representation.

1: **if** $k = 1$ **then**
2: $\quad \sigma' \leftarrow \mathsf{T}.\text{value}$ ▷ It holds that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$.
3: $\quad \hat{z}_0 \leftarrow \text{SamplerZ}(\hat{t}_0, \sigma')$ ▷ For $k = 1$, $(\hat{t}_i, \hat{z}_i) = t_i, z_i$. Assume $z_0 \sim D_{\mathbb{Z},\sigma',t_0}$
4: $\quad \hat{z}_1 \leftarrow \text{SamplerZ}(\hat{t}_1, \sigma')$ ▷ Assume $z_1 \sim D_{\mathbb{Z},\sigma',t_1}$
5: $\quad$ **return** $\hat{\mathbf{z}} = (\hat{z}_0, \hat{z}_1)$
6: $(\hat{\ell}, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T}.\text{value}, \mathsf{T}.\text{leftchild}, \mathsf{T}.\text{rightchild})$
7: $\hat{\mathbf{t}}_1 \leftarrow \text{splitfft}(\hat{t}_1)$ ▷ $\hat{\mathbf{t}}_1 \in \text{FFT}\left(\mathbb{Q}[x]/(x^{k/2}+1)\right)^2$, likewise $\hat{\mathbf{t}}_0$ below. ▷ `fadd`, `fcmul`.
8: $\hat{\mathbf{z}}_1 \leftarrow \text{ffSampling}_{k/2}(\hat{\mathbf{t}}_1, \mathsf{T}_1)$ ▷ First recursive call to $\text{ffSampling}_{k/2}$
9: $\hat{z}_1 \leftarrow \text{mergefft}(\hat{\mathbf{z}}_1)$ ▷ $\hat{\mathbf{z}}_1 \in \text{FFT}\left(\mathbb{Z}[x]/(x^{k/2}+1)\right)^2$, likewise $\hat{\mathbf{z}}_0$ below. ▷ `fadd`, `fcmul`.
10: $\hat{t}'_0 \leftarrow \hat{t}_0 + (\hat{t}_1 - \hat{z}_1) \odot \hat{\ell}$ ▷ `fadd`, `fmul`.
11: $\hat{\mathbf{t}}_0 \leftarrow \text{splitfft}(\hat{t}'_0)$ ▷ `fadd`, `fcmul`.
12: $\hat{\mathbf{z}}_0 \leftarrow \text{ffSampling}_{k/2}(\hat{\mathbf{t}}_0, \mathsf{T}_0)$ ▷ Second recursive call to $\text{ffSampling}_{k/2}$
13: $\hat{z}_0 \leftarrow \text{mergefft}(\hat{\mathbf{z}}_0)$ ▷ `fadd`, `fcmul`.
14: **return** $\hat{\mathbf{z}} = (\hat{z}_0, \hat{z}_1)$



Bounding variables in ffSampling via Gaussian tail bounds

We show that each variable in ffSampling can be expressed as the inner product <X, a> + b, for X a multivariate Gaussian, and (a, b) public.

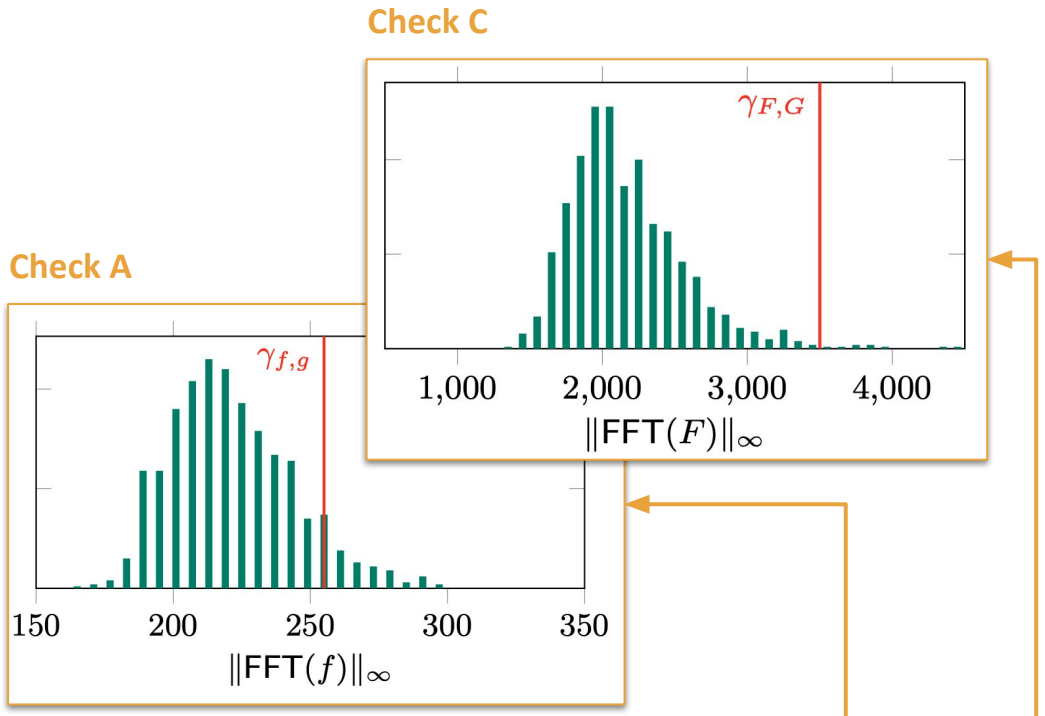
- All variables can then be bounded using tail bounds on projected Gaussians.
- Furthermore, tight upper-bounds on (a, b) are enabled by Check B & D as well as two additional Checks A & C.

ffSampling

Algorithm 8 $\text{ffSampling}_k(\hat{\mathbf{t}}, \mathsf{T})$

Require: $\hat{\mathbf{t}} = (\hat{t}_0, \hat{t}_1) \in \text{FFT}\left(\mathbb{Q}[x]/(x^k+1)\right)^2$, a FALCON tree T
Ensure: $\hat{\mathbf{z}} = (\hat{z}_0, \hat{z}_1) \in \text{FFT}\left(\mathbb{Z}[x]/(x^k+1)\right)^2$
Format: All polynomials are in FFT representation.

1: **if** $k = 1$ **then**
2: $\quad \sigma' \leftarrow \mathsf{T}.\text{value}$ ▷ It holds that $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$.
3: $\quad \hat{z}_0 \leftarrow \text{SamplerZ}(\hat{t}_0, \sigma')$ ▷ For $k = 1$, $(\hat{t}_i, \hat{z}_i) = t_i, z_i$. Assume $z_0 \sim D_{\mathbb{Z},\sigma',t_0}$
4: $\quad \hat{z}_1 \leftarrow \text{SamplerZ}(\hat{t}_1, \sigma')$ ▷ Assume $z_1 \sim D_{\mathbb{Z},\sigma',t_1}$
5: $\quad$ **return** $\hat{\mathbf{z}} = (\hat{z}_0, \hat{z}_1)$
6: $(\hat{\ell}, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T}.\text{value}, \mathsf{T}.\text{leftchild}, \mathsf{T}.\text{rightchild})$
7: $\hat{\mathbf{t}}_1 \leftarrow \text{splitfft}(\hat{t}_1)$ ▷ $\hat{\mathbf{t}}_1 \in \text{FFT}\left(\mathbb{Q}[x]/(x^{k/2}+1)\right)^2$, likewise $\hat{\mathbf{t}}_0$ below. ▷ `fadd`, `fcmul`.
8: $\hat{\mathbf{z}}_1 \leftarrow \text{ffSampling}_{k/2}(\hat{\mathbf{t}}_1, \mathsf{T}_1)$ ▷ First recursive call to $\text{ffSampling}_{k/2}$
9: $\hat{z}_1 \leftarrow \text{mergefft}(\hat{\mathbf{z}}_1)$ ▷ $\hat{\mathbf{z}}_1 \in \text{FFT}\left(\mathbb{Z}[x]/(x^{k/2}+1)\right)^2$, likewise $\hat{\mathbf{z}}_0$ below. ▷ `fadd`, `fcmul`.
10: $\hat{t}'_0 \leftarrow \hat{t}_0 + (\hat{t}_1 - \hat{z}_1) \odot \hat{\ell}$ ▷ `fadd`, `fmul`.
11: $\hat{\mathbf{t}}_0 \leftarrow \text{splitfft}(\hat{t}'_0)$ ▷ `fadd`, `fcmul`.
12: $\hat{\mathbf{z}}_0 \leftarrow \text{ffSampling}_{k/2}(\hat{\mathbf{t}}_0, \mathsf{T}_0)$ ▷ Second recursive call to $\text{ffSampling}_{k/2}$
13: $\hat{z}_0 \leftarrow \text{mergefft}(\hat{\mathbf{z}}_0)$ ▷ `fadd`, `fcmul`.
14: $\quad$ **return** $\hat{\mathbf{z}} = (\hat{z}_0, \hat{z}_1)$



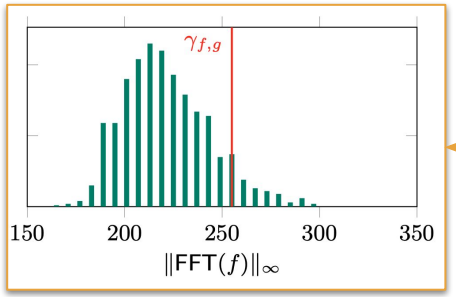
Bounding variables in ffSampling via Gaussian tail bounds

We show that each variable in ffSampling can be expressed as the inner product <X, a> + b, for X a multivariate Gaussian, and (a, b) public.

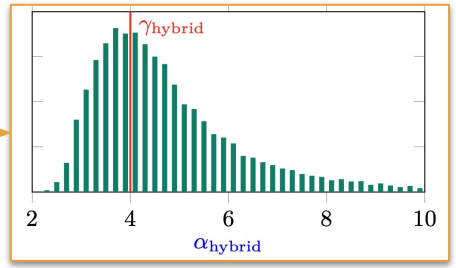
- All variables can then be bounded using tail bounds on projected Gaussians.
- Furthermore, tight upper-bounds on (a, b) are enabled by Check B & D as well as two additional Checks A & C.

Tweaking Keygen

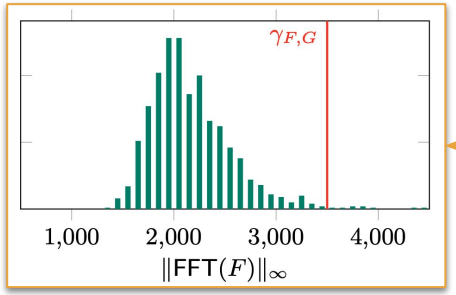
Check A



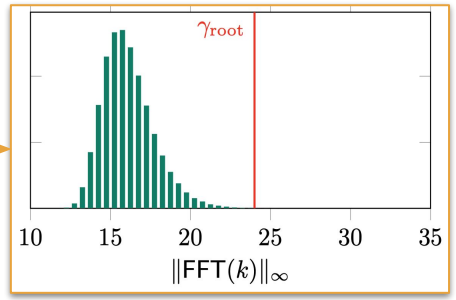
Check B



Check C



Check D



Algorithm 2 NTRUGen(ϕ, q)

Require: A monic polynomial $\phi \in \mathbb{Z}[x]$ of degree n, a modulus q
Ensure: Polynomials f, g, F, G

1: $\sigma_{\{f,g\}} \leftarrow 1.17\sqrt{q/2n}$ ▷ Precomputed
2: **for** i from 0 to $n-1$ **do**
3: $\quad f_i \leftarrow D_{\mathbb{Z},\sigma_{\{f,g\}},0}$ ▷ Table-based
4: $\quad g_i \leftarrow D_{\mathbb{Z},\sigma_{\{f,g\}},0}$ ▷ Table-based
5: $f \leftarrow \sum_i f_i x^i$ ▷ $f \in \mathbb{Z}[x]/(\phi)$
6: $g \leftarrow \sum_i g_i x^i$ ▷ $g \in \mathbb{Z}[x]/(\phi)$
7: **if** $\|\mathsf{FFT}(f), \mathsf{FFT}(g)\|_\infty > \gamma_{f,g}$ **then restart** ▷ `fadd`, `fcmul`, `fmul`
8: **if** $\alpha_{\text{hybrid}}(f,g) > \gamma_{\text{hybrid}}$ **then restart** ▷ `fadd`, `fcmul`, `fmul`
9: **if** $\alpha_{\text{GPV}}(f,g) > 1.17$ **then restart** ▷ `fadd`, `fcmul`, `fmul`, `fdiv`
10: **if** $\mathsf{NTT}(f)$ contains 0 **then restart** ▷ Check that f is invertible mod q
11: $F, G \leftarrow \mathsf{NTRUSolve}_{n,q}(f,g)$ ▷ Computing F, G such that $fG - gF = q \bmod \phi$
12: **if** $(F,G) = \perp$ **then restart**
13: **if** $\|\mathsf{FFT}(F), \mathsf{FFT}(G)\|_\infty > \gamma_{F,G}$ **then restart** ▷ `fadd`, `fcmul`, `fmul`
14: Compute $k = \frac{F f^\star + G g^\star}{f f^\star + g g^\star}$ ▷ `fadd`, `fcmul`, `fmul`, `fdiv`
15: **if** $\|\mathsf{FFT}(k)\|_\infty > \gamma_{\text{root}}$ **then restart** ▷ Bound root node of Falcon tree
16: **return** f, g, F, G

These four changes benefit **Keygen, ffLDL, LDL*, Sign and ffSampling**.
Less than **1/3** of admissible keys are rejected.

04

Experiments

Two Implementations

Both implementations are validated against Falcon's reference implementation from Pornin.

Python

https://github.com/fixed-point-fndsa/fxp-falcon-py

Type 2 FxP: each variable x has a compile-time scaling factor f(x).

- Efficient, but tedious to specify and implement.
- Format: 64-bit

Used for benchmarking precision (next slides).

C

[To be released]

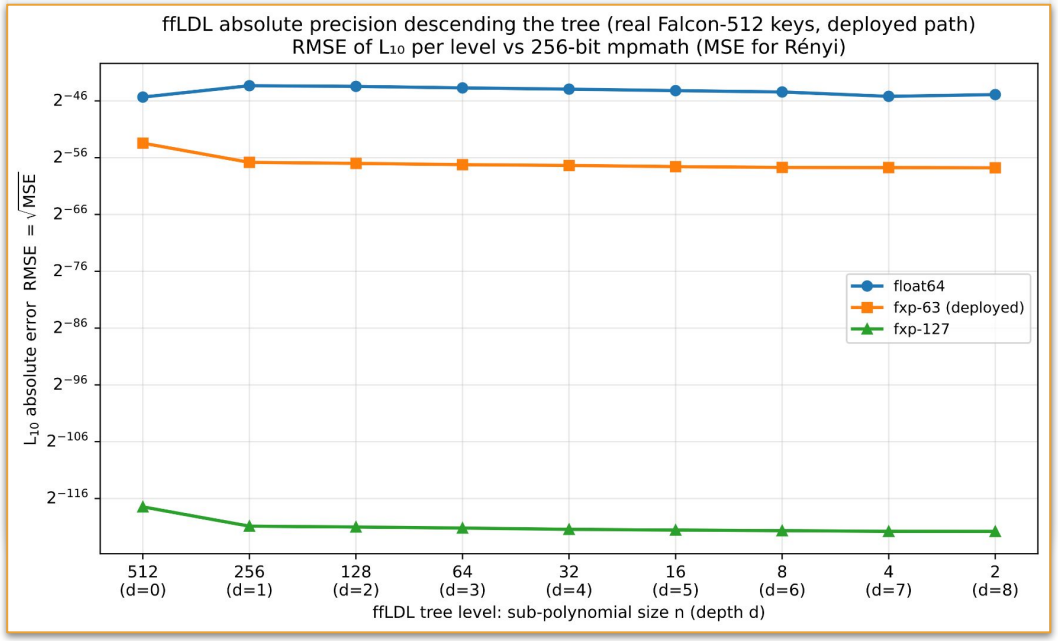
Type 1 FxP: all variables have the same scaling factor f.

- Simple to specify and implement, but slightly less efficient.
- Format: 64.64 (128-bit in total)

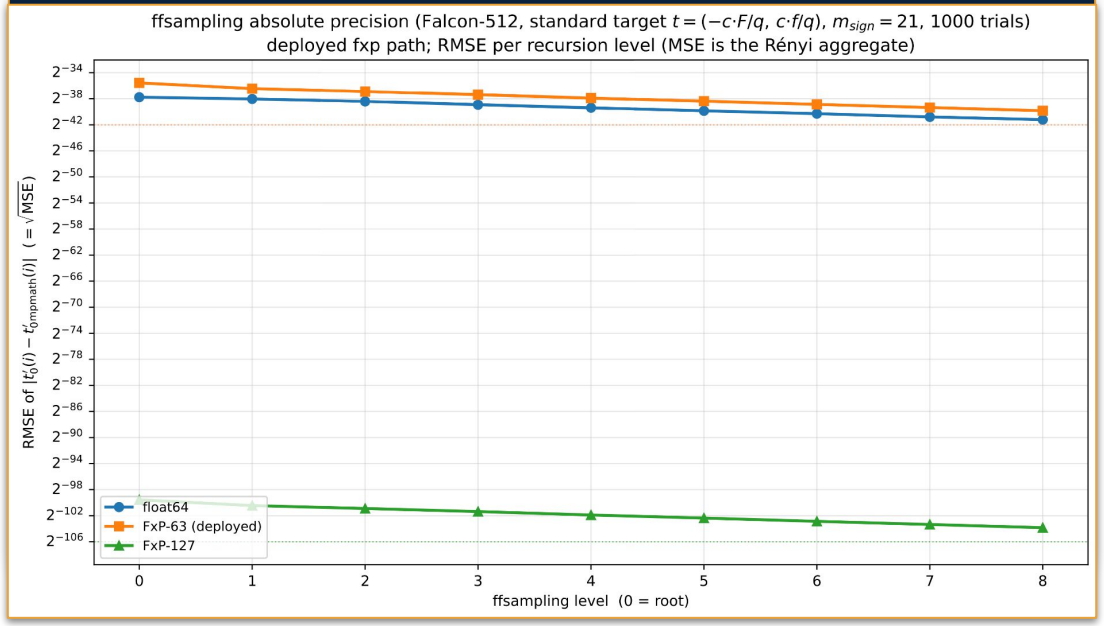
Benchmarking Precision in Python

The measured root mean-square error (RMSE) supports up to 2^{60} signing queries via a Rényi divergence argument.

Key Expansion (ffLDL)



Signing (ffSampling)



05

Conclusion and Next Steps

Provable precision, specification & masking

Conclusion and Next Steps

Paper accepted at IACR CRYPTO 2026, soon on ePrint.

100%

Integer arithmetic (no floating-point)

64

Bits of precision

2^{60}

Signing queries estimated
(Rényi divergence + benchmarks)

4

Minimal tweaks in NTRUGen

Provable Precision

In our paper, boundedness is proven but precision is *benchmarked*.
What can we *prove* about precision?

Specification

If we were to specify a fixed-point FN-DSA:

1. Which tweaks would we keep?
2. Which FxP format (Type 1 or Type 2)?

Masking

Can we mask fixed-point Falcon/FN-DSA?

Feel free to reach out to us!