# Do Not Disturb a Sleeping Falcon

## Floating-Point Error Sensitivity of the Falcon Sampler and Its Consequences

Xiuhan Lin[1], Mehdi Tibouchi[2], Yang Yu[3], and Shiduo Zhang[3]

[1] Shandong University, China
xhlin@mail.sdu.edu.cn
[2] NTT Social Informatics Laboratories, Japan
mehdi.tibouchi@ntt.com
[3] Tsinghua University, China
zsd@mail.tsinghua.edu.cn
yu-yang@mail.tsinghua.edu.cn

**Abstract.** Falcon is one of the three postquantum signature schemes already selected by NIST for standardization. It is the most compact among them, and offers excellent efficiency and security. However, it is based on a complex algorithm for lattice discrete Gaussian sampling which presents a number of implementation challenges. In particular, it relies on (possibly emulated) floating-point arithmetic, which is often regarded as a cause for concern, and has been leveraged in, e.g., side-channel analysis. The extent to which Falcon's use of floating point arithmetic can cause security issues has yet to be thoroughly explored in the literature.

In this paper, we contribute to filling this gap by identifying a way in which Falcon's lattice discrete Gaussian sampler, due to specific design choices, is singularly sensitive to floating-point errors. In the presence of small floating-point discrepancies (which can occur in various ways, including the use of the two *almost but not quite* equivalent signing procedures "dynamic" and "tree" exposed by the Falcon API), we find that, when called twice on the same input, the Falcon sampler has a small but significant chance (on the order of once in a few thousand calls) of outputting two different lattice points with a very structured difference, that immediately reveals the secret key. This is in contrast to other lattice Gaussian sampling algorithms like Peikert's sampler and Prest's hybrid sampler, that are stable with respect to small floating-point errors.

Correctly generated Falcon signatures include a salt that should in principle prevent the sampler to ever be called on the same input twice. In that sense, our observation has little impact on the security of Falcon signatures per se (beyond echoing warnings about the dangers of repeated randomness). On the other hand, it is critical for *derandomized* variants of Falcon, which have been proposed for use in numerous settings. One can mention in particular identity-based encryption, SNARK-friendly signatures, and sublinear signature aggregation. For all these settings, small floating point discrepancies have a chance of resulting in full private key exposure, even when using the slower, integer-based emulated floating-point arithmetic of Falcon's reference implementation.

**Keywords:** Falcon · Lattice-Based Cryptography · Floating-Point Arithmetic · Hash-and-Sign Signatures · NTRU

## 1 Introduction

Falcon [PFH+22] is a postquantum signature scheme based on structured lattices, which was one of the three signatures selected by NIST for standardization in 2022. It is a particularly efficient instantiation of the GPV lattice trapdoor framework [GPV08] over NTRU lattices. It was the most compact signature scheme in the third round of the NIST standardization process (some candidates

had shorter keys but much larger signatures, and vice versa), and has both fast signing and verification on architectures for which an optimized implementation is available.

In addition, since it is one of the most efficient available instantiations of a lattice trapdoor, FALCON is also an attractive building block for more advanced primitives and protocols, including identity-based encryption [DLP14, ZMS+24], trapdoor commitments [DOTT21] and ring signatures [LAZ19].

FALCON's main drawback, however, is its overall complexity. The signing algorithm samples lattice points according to a discrete Gaussian distribution, and a small deviation from the correct distribution may result in leakage of the entire private key. Moreover, the sampling algorithm relies fairly crucially on floating-point arithmetic. This makes FALCON challenging to implement correctly, and issues like side-channel resilience are difficult to address efficiently [FKT+20, KA21, GMRR22, ZLYW23].

FALCON's reliance on floating-point arithmetic, specifically, is often raised as a point of potential concern, but a thorough discussion of its security implications seems to be lacking aside from the context of side-channels.

### 1.1 Our contributions

This paper aims at filling this gap in the literature, with a discussion of FALCON's sensitivity to floating-point errors, and a discussion of its consequences. Our contributions are threefold:

- we first show that the discrete Gaussian sampler within FALCON behaves in such a way that small discrepancies introduced by floating-point arithmetic can result in very structured differences in its output;
- we then describe how this can yield to a complete key recovery when the sampler if called twice on the same input, but with different intermediate floating-point errors;
- we finally mention how such distinct intermediate floating-point errors can occur in certain contexts due to the API exposed by FALCON, demonstrate the key recovery attack in those contexts, and discuss the security impact on non-standard and advanced uses of FALCON.

### 1.2 Applicability

We stress that the sensitivity to floating-point errors that we identify only results in a vulnerability in contexts where the sampler can be called twice on the same input, but with different intermediate floating-point errors. For normal FALCON signatures, this should never happen, owing to the use of a salt that never repeats. It should be noted, however, that repeated randomness does occur in the real world, sometimes with catastrophic cryptographic consequences [HDWH12, BHH+14, HFH16].

More to the point, our observations are critical to the security of *derandomized* variants of FALCON, which are often necessary in advanced applications. For example, in an identity-based encryption scheme based on FALCON, key extraction (which corresponds to FALCON signing) should always output the same key on a given identity. Short of making the scheme stateful, this requires relying on a derandomized variant of FALCON signing, as noted in the FALCON specification document itself [PFH+22, §2.2.1]. Such a design is adopted, for instance, by the LATTE (H)IBE construction [ZMS+24], under consideration for NCSC and ETSI standardization.

Derandomization is also useful for other purposes. For example, Lazar and Peikert [LP21a] describe and fully implement a deterministic variant of FALCON in order to obtain SNARK-friendly signatures. The idea is that, when proving knowledge of a normal, salted FALCON signature on a given message, the digest (i.e., the center of the lattice discrete Gaussian distribution that signature generation samples from, and that is recomputed in signature verification) depends on the salt, which is part of the signature. As a result, the SNARK circuit has to include the entire digest computation (using the SHAKE expandable output function) which is very costly. In contrast, for deterministic signatures with no salt, or equivalently a fixed salt, the digest computation depends only on the message, and can thus be carried out "outside" the SNARK, resulting in much more efficient proofs.

Along those lines, the SNARK-based signature aggregation technique for FALCON signatures recently proposed by Aardal et al. [AAB+24] achieves substantially smaller (and asymptotically sublinear) aggregated signature size for deterministic FALCON compared to standard FALCON (for which, in particular, linear scaling is unavoidable since all salts need to be included in the aggregated signatures).

For these reasons, there is significant interest in derandomized versions of FALCON, and it is therefore important to understand their security, and how it is impacted by the floating-point error sensitivity of FALCON's sampler.

The impact is particularly dramatic in the IBE setting: in the presence of floating-point discrepancies, our attack shows that it suffices for a legitimate user to ask the master key authority for its own key twice to have a chance (on the order of once in a few thousands) to trigger the issue, and instantly recover the authority's master secret key! For deterministic and unsalted FALCON signatures, the situation is similar: each time a signer signs the same message twice in the presence of floating-point discrepancies, they have a chance of exposing their entire private key.

### 1.3 Technical overview

*Floating-point error sensitivity.* As mentioned above, we identify a particular property of the FALCON discrete Gaussian sampler that make it uniquely sensitive to small discrepancies introduced by floating-point computations, in a way that that other similar samplers, such as those of Mitaka [EFG+22] and Antrag [ENS+23] are not.

At a high level, this comes from the fact that FALCON's one-dimensional discrete Gaussian sampler SamplerZ, which samples from the discrete Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ over $\mathbb{Z}$ with given center $c$ and standard deviation $\sigma$, presents a discontinuity around integer values of the center $c$. For an integer $c$ and a small error $\varepsilon$, the distributions $D_{\mathbb{Z},\sigma,c+\varepsilon}$ and $D_{\mathbb{Z},\sigma,c-\varepsilon}$ are of course close, and SamplerZ samples correctly from them. However, *for a fixed set of random coins*, SamplerZ will output completely different results for the centers $c+\varepsilon$ and $c-\varepsilon$, due to those values rounding to different integers.

The second part of the observation is that, within the FALCON lattice Gaussian sampler, SamplerZ *can* in fact be called with an integer center up to floating point errors.[4] In fact, this happens with small but significant probability at one of exactly 4 positions during the traversal of the so-called FALCON tree, corresponding to the first two and the last two calls of SamplerZ (for reasons related to arithmetic properties of the FALCON keys).

*Key recovery from two outputs of the sampler.* By the preceding discussion, when the lattice Gaussian sampler of FALCON is called twice on the same input, but with different floating-point errors, it can happen that the SamplerZ outputs differ in one of the first two or the last two calls.

A difference in one of the *first* two calls affects the entire remainder of the sampling procedure, and hence one obtains two entirely different lattice points close to the chosen center of the lattice Gaussian. Their difference is therefore a fairly short vector in the NTRU lattice, which is somewhat concerning, but even obtaining many such vectors is not believed to enable a key recovery attack (they aren't small enough).

A difference in one of the *last* two calls to SamplerZ, however, is a totally different matter: it introduces a difference in just two components of (the Fourier domain representations of) the coefficient vectors of the outputs over the secret trapdoor basis. The difference between the two outputs is therefore highly structured, and recovering the entire private key from it is fairly straightforward (it involves at most a simple exhaustive search over a few thousand pairs of integers).

This means in particular that two signatures on the same message in a derandomized variant of FALCON have a chance of leaking the entire private key if a floating-point discrepancy occurs between them.

[4] This is precisely what *does not* happen for Mitaka and Antrag, except with negligible probability. Indeed, the implementations of those schemes also use FALCON's SamplerZ, but they call it with centers that are themselves distributed according to a continuous normal distribution (up to floating-point precision), and in particular, integer values only occur with negligible probability.

*Floating-point discrepancies in FALCON.* We then discuss some examples of the ways in which such floating-point discrepancies can appear.

A first way, which we do not explore further, is the weak determinism of floating-point arithmetic [DGPY20]. The exact same source code compiled with the same compiler with the same options and run on the same inputs can yield different floating-point results on architectures compatible with the IEEE-754 floating-point arithmetic standard [iee85] (e.g., because of the use of extended precision for intermediate results in registers, as is done on `x87` FPU [Mon08]).

A second way is through running different floating-point implementations of the same algorithm. When compiling FALCON with native floating-point arithmetic, the resulting compiled code can in principle depend on the target architecture, the compiler and the choice of compiler optimizations, and all those factors can result in different signatures for the same message with the same salt. In addition, the FALCON also includes optimized versions for the Intel AVX2 vector unit, with or without fused multiply-add (FMA) instructions, which can lead to discrepancies as well. This is a fairly well-understood problem: for example, due to these differences, the deterministic FALCON implementation of Lazar and Peikert [LP21b] actually includes a warning against the use of all floating-point arithmetic variants except the much slower, emulated implementation with 64-bit integers. Nevertheless, we do experimentally test for this issue. Interestingly, on our target platform, we find no signature discrepancies arising from the various floating-point variants exposed by the code base *except* for the FMA-optimized code. Signature pairs generated by the FMA code on the one hand and one of the other floating-point flavors on the other hand do present occasional discrepancies that allow for full key recovery (once every few thousand pairs).

A third and somewhat more surprising way is through the API exposed by the FALCON implementation (and its deterministic variants). Namely, FALCON includes two slightly different implementations of its signing procedure: the "dynamic" variant, which generates the FALCON tree on the fly, and the "tree" variant, which takes a precomputed FALCON tree as input. Those two variants carry out *almost* the same floating point computations on the same values in the same order, except at the bottom few levels of the tree traversal, where the "tree" variant uses some small shortcuts. It turns out that those minor differences are sufficient to induce exploitable discrepancies as well. The differences have in fact been pointed out in the literature before [PKKK24], albeit with no analysis of the reason why they occur or of their impact. In any case, we experimentally confirm that these discrepancies occur, including in the slow, emulated floating point version recommended by the deterministic FALCON authors, who do not warn against using those two variants concurrently. The discrepancies also give rise to a full key recovery at a similar rate as the ones from completely different floating-point instructions through the entire signing procedure.

### 1.4 Organization of the paper

Following some preliminary material in Section 2, Section 3 recalls the FALCON scheme along with its deterministic variant and different implementations. Section 4 demonstrates the floating-point error sensitivity of the integer Gaussian sampler of FALCON that could trigger discrepant signing executions for distinct implementations of FALCON. Section 5 then describes a key recovery attack exploiting close discrepant FALCON signatures for the same digest syndrome and discusses its feasibility. Section 6 presents our experiments validating that different signing modes of FALCON and the use of the FMA floating-point instructions can indeed cause discrepant signatures in practice. Finally, some countermeasures are proposed in Section 7.

## 2 Preliminaries

We use bold lowercase letters to represent (row) vectors, i.e., $\mathbf{b} = (b_0, \cdots, b_{n-1})$, where $b_i$ is coefficient $i$ of vector $\mathbf{b}$ (note that we use zero-based indexing throughout the paper). For $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, the coefficient-wise inner product is $\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=0}^{n-1} a_i b_i$. Given $\mathbf{b} \in \mathbb{R}^n$, its $\ell_2$-norm is denoted $\|\mathbf{b}\| = \sqrt{\langle \mathbf{b}, \mathbf{b} \rangle}$.

We also use bold uppercase letters to denote matrices, i.e., $\mathbf{B} = (\mathbf{b}_0, \cdots, \mathbf{b}_{n-1})$, where $\mathbf{b}_i$ is the row vector of index $i$ of $\mathbf{B}$. The inverse and conjugate transpose of $\mathbf{B}$ are denoted by $\mathbf{B}^{-1}$ and $\mathbf{B}^*$ respectively.

For $x \in \mathbb{R}$, $\lfloor x \rceil$ denotes rounding $x$ to the nearest integer and $\lfloor x \rfloor$ is the floor operation of $x$.

We abbreviate "floating-point representation" to fpr and define the fpr operations $\mathring{*} \in \{ \mathring{+}, \mathring{-}, \mathring{\times} \}$.

### 2.1 Linear algebra and lattices

Let $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}) \in \mathbb{R}^{n \times m}$ be a full-rank matrix. The Gram–Schmidt orthogonalization (GSO) of $\mathbf{B}$ is the unique matrix $\widetilde{\mathbf{B}} = (\widetilde{\mathbf{b}}_0, \cdots, \widetilde{\mathbf{b}}_{n-1})$ with pairwise orthogonal rows such that there exists a lower triangular matrix $\mathbf{L}$ with only 1's on its diagonal satisfying $\mathbf{B} = \mathbf{L}\widetilde{\mathbf{B}}$.

Any symmetric positive definite matrix $\mathbf{G}$ admits a unique decomposition of the form $\mathbf{LDL}^*$ where $\mathbf{L}$ is a lower triangular matrix with only 1's on its diagonal and $\mathbf{D}$ is a diagonal matrix. In particular, when $\mathbf{G}$ is the Gram matrix $\mathbf{BB}^*$ of the matrix $\mathbf{B}$ above, then the matrix $\mathbf{L}$ associated with its GSO coincides with the matrix $\mathbf{L}$ appearing in the LDL$^*$ decomposition of $\mathbf{G}$, and $\mathbf{D} = \widetilde{\mathbf{B}}\widetilde{\mathbf{B}}^*$.

A lattice $\mathcal{L}$ is a discrete additive subgroup of some finite dimensional vector space $\mathbb{R}^m$. It can always be written as the set of all integer linear combinations of some linearly independent vectors $\mathbf{b}_0, \ldots, \mathbf{b}_{n-1}$, i.e., $\mathcal{L} = \left\{ \sum_{i=0}^{n-1} x_i \mathbf{b}_i \mid (x_0, \ldots, x_{n-1}) \in \mathbb{Z}^n \right\}$. We call the matrix $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{n-1})$ a basis of the lattice $\mathcal{L}$, and $n$ the rank of $\mathcal{L}$ (it is independent of the choice of a basis). The lattice corresponding to $\mathbf{B}$ is denoted by $\mathcal{L}(\mathbf{B})$. When $m = n$, we say that it is full-rank.

### 2.2 Gaussian distributions

For a distribution $D$, we write $a \leftarrow D$ to mean that $a$ is a sample from the distribution $D$, and $y \sim D$ to say that the random variable $y$ is distributed according to $D$. We denote by $D(z)$ the probability that a random variable $y \sim D$ satisfies that $y = z$.

Given a standard deviation $\sigma > 0$ and a center $\mathbf{c} \in \mathbb{R}^n$, we define the Gaussian function as $\rho_{\sigma,\mathbf{c}}(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{c}\|^2}{2\sigma^2}\right)$. For some fixed $\sigma, \mathbf{c}$ and a lattice $\mathcal{L}$, we denote by $D_{\mathcal{L},\sigma,\mathbf{c}}$ the discrete Gaussian distribution over the lattice $\mathcal{L}$ given by:

$$D_{\mathcal{L},\sigma,\mathbf{c}}(\mathbf{u}) = \frac{\rho_{\sigma,\mathbf{c}}(\mathbf{u})}{\sum_{\mathbf{v} \in \mathcal{L}} \rho_{\sigma,\mathbf{c}}(\mathbf{v})}.$$

In the particular case when $\mathcal{L} = \mathbb{Z}$, we call $D_{\mathbb{Z},\sigma,c}$ the integer Gaussian distribution of parameters $\sigma, c$ and following the FALCON specification, denote by $D^+_{\mathbb{Z}^+,\sigma}$ the integer "half-Gaussian" distribution defined by $D^+_{\mathbb{Z}^+,\sigma}(u) = \frac{\rho_{\sigma,0}(u)}{\sum_{v \in \mathbb{Z}^+} \rho_{\sigma,c}(v)}$.

### 2.3 NTRU

Let $\mathcal{R} = \mathbb{Z}[x]/\phi$ where $\phi = x^n + 1$ with $n$ a power of 2 and $\mathcal{K} = \mathbb{Q}[x]/\phi$. In the NTRU scheme, the secret key consists of two short polynomials $f, g \in \mathcal{R}$ such that $f$ invertible modulo some prime number $q$, and the public key is $h = g/f \bmod q$. The NTRU lattice defined by $h \in \mathcal{R}$ is $\mathcal{L}_{NTRU} = \{(s_0, s_1) \in \mathcal{R}^2 \mid s_0 + s_1 h = 0 \bmod q\}$. The NTRU trapdoor basis is

$$\mathbf{B}_{f,g} = \begin{pmatrix} g & -f \\ G & -F \end{pmatrix}$$

where $(F, G) \in \mathcal{R}^2$ is such that $fG - gF = q$.

Algorithm 1: Sign

**Input:** A message msg and an NTRU lattice trapdoor basis $\mathbf{B}_{f,g}$
**Output:** A valid signature $(r, s)$

$r \xleftarrow{\$} \{0,1\}^{320}$, $c \leftarrow \mathsf{H}(r \| msg)$
$\mathbf{t} \leftarrow (\mathsf{FFT}(c), \mathsf{FFT}(0)) \cdot \mathsf{FFT}(\mathbf{B}_{f,g})^{-1}$ ▷ pre-image computation
**do**
  **do**
    $\mathbf{z} \leftarrow \mathsf{ffSampling}(\mathbf{t}, \mathsf{T})$ ▷ trapdoor sampling
    $\mathbf{s} = (\mathbf{t} - \mathbf{z}) \cdot \mathsf{FFT}(\mathbf{B}_{f,g})$ ▷ $\mathbf{s} \sim D_{(c,0)+\mathcal{L}(\mathbf{B}),\sigma_{\mathtt{sig}},0}$
  **while** $\|\mathbf{s}\| > \lfloor \beta^2 \rfloor$
  $(s_0, s_1) \leftarrow \mathsf{invFFT}(\mathbf{s})$
**while** $s = \perp$
**return** $(r, \mathbf{s})$

### 2.4 Fast Fourier transform

Let $\Omega_\phi$ a set of representatives of the complex roots of $\phi$ up to conjugation. We usually take:

$$\Omega_\phi = \left\{ \exp\left(\frac{i(2j+1)\pi}{2n}\right) \middle| 0 \leq j < n/2 \right\}.$$

The (fast, negacyclic) Fourier transform ($\mathsf{FFT}$) maps an element $f$ of $\mathcal{K}$ (or more generally of $\mathcal{K} \otimes_{\mathbb{Q}} \mathbb{R}$) to the vector of its evaluations at the roots $\zeta \in \Omega_\phi$, namely:

$$\mathsf{FFT}(f) = \big(f(\zeta)\big)_{\zeta \in \Omega_\phi}.$$

It induces an isomorphism of $\mathbb{R}$-algebras $\mathsf{FFT}\colon \mathcal{K} \otimes_{\mathbb{Q}} \mathbb{R} \to \mathbb{C}^{\Omega_\phi} = \mathbb{C}^{n/2}$. The inverse isomorphism is denoted by $\mathsf{invFFT}$. Both $\mathsf{FFT}$ and $\mathsf{invFFT}$ can be computed in $\mathcal{O}(n \log n)$ operations.

To make the computation of these maps cache-friendly, it is customary to represent the coefficients of the polynomials in $\mathcal{K} \otimes_{\mathbb{Q}} \mathbb{R}$ in bit reversed order, where the coefficient of index $i$ corresponds to the monomial of degree $\alpha_i$, where $\alpha_i$ is obtained from $i$ by reversing its representation as a number of $\log_2(n)$ bits. For example, the first four coefficients in bit reversed order correspond to the monomials of degree 0, $n/2$, $n/4$ and $3n/4$ in this order.

## 3 The Falcon Signature Scheme

This section covers some background about the Falcon signature scheme, including a description of its signing algorithm, an overview of one of its deterministic variants, and some details about its various implementations.

### 3.1 The Falcon signing procedure

Falcon is an instantiation over NTRU lattices of the GPV framework for hash-and-sign lattice based signatures [GPV08]. Using the notation of Section 2.3, it uses an NTRU trapdoor basis $\mathbf{B}_{f,g}$ as its secret key and $h = g/f \bmod q$ as the public key, where the coefficients of $(f, g)$ are sampled from $D_{\mathcal{R},\sigma_{\{f,g\}},0}$ with $\sigma_{\{f,g\}} = 1.17\sqrt{q/2n}$, ensuring nearly optimal parameters [DLP14].

*Parameters.* Falcon is defined over the power-of-two cyclotomic ring $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$, and uses the prime modulus $q = 12289$. It has two parameter sets, corresponding to $n = 512$ for NIST security level I (128-bit security), and to $n = 1024$ for NIST security level V (256-bit security). These parameter sets are usually called Falcon–512 and Falcon–1024 respectively. Note that the dimension over $\mathbb{Z}$ of the NTRU module lattice is $2n$ in both cases, so 1024 for Falcon–512 and 2048 for Falcon–1024.

Algorithm 2: $\mathsf{ffSampling}_n$

**Input:** A Target vector $\mathbf{t} = (t_0, t_1) \in \mathsf{FFT}(\mathcal{K})^2$, a FALCON tree $\mathsf{T}$
**Output:** $\mathbf{z} = (z_0, z_1) \in \mathsf{FFT}(\mathcal{R})^2$

**if** $n = 1$ **then**
  $\sigma \leftarrow \mathsf{T.value}$ ▷ $\sigma \in [\sigma_{\min}, \sigma_{\max}]$
  $z_0 \leftarrow \mathsf{SamplerZ}(t_0, \sigma)$, $z_1 \leftarrow \mathsf{SamplerZ}(t_1, \sigma)$
  **return** $\mathbf{z} = (z_0, z_1)$
**end if**
$(l, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T.value}, \mathsf{T.leftchild}, \mathsf{T.rightchild})$
$\mathbf{t}_1 \leftarrow \mathsf{split\_fft}(t_1)$
$\mathbf{z}_1 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_1, \mathsf{T}_1)$ ▷ `First recursive call`
$z_1 \leftarrow \mathsf{merge\_fft}(\mathbf{z}_1)$
$t'_0 \leftarrow t_0 + (t_1 - z_1) \odot l$
$\mathbf{t}_0 \leftarrow \mathsf{split\_fft}(t'_0)$
$\mathbf{z}_0 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_0, \mathsf{T}_0)$ ▷ `Second recursive call`
$z_0 \leftarrow \mathsf{merge\_fft}(\mathbf{z}_0)$
**return** $\mathbf{z} = (z_0, z_1)$

*Signing.* Algorithm 1 describes the signing procedure of FALCON, which essentially amounts to sampling a short signature vector $\mathbf{s} = (s_0, s_1) \sim D_{(c,0)+\mathcal{L}(\mathbf{B}),\sigma_{\mathrm{sig}},0}$ in a certain coset of the NTRU lattice, or equivalently, a lattice point $(c, 0) - \mathbf{s} \sim D_{\mathcal{L}(\mathbf{B}),\sigma_{\mathrm{sig}},(c,0)}$ close to $(c, 0)$. The center $(c, 0)$, also called the *syndrome*, is computed by the hash function $c = \mathsf{H}(r \| \mathsf{msg})$ where $r$ is a 320-bit random *salt* and $\mathsf{msg}$ is the message. The validly generated signature $(s_0, s_1)$ is short, and its acceptance bound is $\beta = 1.1 \cdot \sigma_{\mathrm{sig}}\sqrt{2n}$.

*Trapdoor sampling.* FALCON uses the fast Fourier sampler [DP16] ($\mathsf{ffSampling}$, Algorithm 2) as its trapdoor sampler. It takes as input the coordinates $\mathbf{t}$ over the trapdoor basis $\mathbf{B}_{f,g}$ of the syndrome, as well as the FALCON tree $\mathsf{T}$, which is in essence a compact representation of the GSO of the trapdoor basis $\mathbf{B}_{f,g}$ (or more properly, of the LDL$^*$ decomposition of its Gram matrix). All ring elements are represented in the $\mathsf{FFT}$ domain, and the sampling algorithm runs in quasilinear time overall.

In $\mathsf{ffSampling}$, the lattice Gaussian sampling is reduced to a series of calls to an integer Gaussian sampler $\mathsf{SamplerZ}$, which samples from the integer discrete Gaussian distribution $D_{\mathbb{Z},\sigma,c}$ with varying centers and standard deviations.

### 3.2 Reference and optimized implementations

FALCON provides one reference implementation and several optimized implementations in the NIST round 3 submission package [PFH+22].

The reference implementation only contains one pure portable C version in which all floating-point operations are emulated with integer arithmetic and bit fiddling. We refer to this implementation as $\mathsf{fpemu}$ throughout this paper. It stores IEEE-754 [iee85] double precision values as fixed-width unsigned integer elements of type `uint64_t`, and does not require the support of a hardware floating-point unit ($\mathsf{fpu}$). It is also fully constant time for all intermediate values in the context of FALCON. However, the overhead of floating point emulation makes it considerably slower than the optimized variants.

The following optimized implementations (summarized in Table 1) are also included in the FALCON code based:

- $\mathsf{fpnative}$: makes use of the native hardware $\mathsf{fpu}$ (SSE2 unit on `x86_64`) using floating point values of type `double`;
- $\mathsf{avx2}$: also uses the native hardware $\mathsf{fpu}$ with values of type `double`, and packs them into AVX2 registers for four-way vectorization;

**Table 1.** Different implementations of FALCON.

| Version | Reference | Optimized | $n = 512\&1024$ | sign_dyn&sign_tree | fpr |
|---|---|---|---|---|---|
| fpemu | ✔ | ✖ | ✔ | ✔ | uint64_t |
| fpnative | ✖ | ✔ | ✔ | ✔ | double |
| avx2 | ✖ | ✔ | ✔ | ✔ | double |
| avx2_fma | ✖ | ✔ | ✔ | ✔ | double |
| cxm4 | ✖ | ✔ | ✔ | ✔ | uint64_t |

– avx2_fma: further optimized by enabling "fused multiply-add" (FMA) intrinsics;
– cxm4: inline assembly code for ARM Cortex-M4 microcontrollers, particularly embedded in the fpr operations.

### 3.3 Deterministic Falcon

The signing procedure of the normal FALCON signature is probabilistic, since it samples the random salt $r$ as well as many random samples as part of the calls to the SamplerZ algorithm. As discussed in the introduction, however, there are numerous settings where derandomized variants may be desirable.

In [LP21a], Lazar and Peikert propose, specify and implement such a deterministic variant for SNARK-friendly applications, called "deterministic FALCON". In that scheme, signing the same message multiple times is supposed to result in the same signature every time. To this end, the signing algorithm uses a fixed salt value $r$, and the random tape used in all the random sampling procedures is obtained by expanding a seed randbytes derived from the secret key sk and the message msg as follows:

$$r = 0\|\ell\|\mathsf{FALCON\_DET}\|00\cdots00 \quad \text{and} \quad \mathsf{randbytes} = \ell\|\mathsf{sk}\|\mathsf{msg}$$

where the first zero byte is salt version with default value 0, $\ell = \log_2(n)$, the string FALCON_DET is in ASCII representation and the remaining part is padded by all zero bytes.

The specification of deterministic FALCON is accompanied by an implementation [LP21b] based on the most recent version of the standard FALCON implementation at the time of this writing, namely the implementation included in the NIST round 3 submission package [PFH+22]. The code base includes the both the reference implementations as well as the various optimized ones, although only the reference implementation is enabled by default, and the included `README` files warns against the use of the other ones due to risks from floating point discrepancies. We refer to those various implementations with the same names as the standard FALCON ones with a _det suffix to emphasize that they are deterministic FALCON implementations. The fpemu_det is thus the supported variant, whereas avx2_det, avx2_fma_det, etc., are unsupported.

## 4 Floating-Point Error Sensitivity of the Falcon Integer Gaussian Sampler

At the heart of the FALCON signing algorithm is the fast Fourier sampler ffSampling (Algorithm 2) which is an efficient variant of the Klein–GPV sampler [GPV08] over cyclotomic rings with smooth conductors. It can be seen as the Gaussian sampling version of the Ducas–Prest fast Fourier nearest plane algorithm [DP16]. As in the Klein–GPV sampler, the sampling procedure of ffSampling is decomposed into a sequence of integer Gaussian samplings. While a FALCON signature is fully determined by these integer samples, the sampling procedure involves extensive floating-point arithmetic.

In this section, we present our main observation: the integer Gaussian sampling algorithm SamplerZ of FALCON presents a marked sensitivity to floating point errors at very specific input points, namely, when sampling integer discrete Gaussians with *integer* centers. In addition, we show that such input points do have a (relatively small but nonetheless) significant at some well-defined locations within

Algorithm 3: SamplerZ

**Input:** A center $c$ and standard deviation $\sigma \in [\sigma_{\min}, \sigma_{\max}]$
**Output:** An integer $z$ derived from a distribution close to $D_{\mathbb{Z},\sigma,c}$

$r \leftarrow c - \lfloor c \rfloor$
$y_+ \leftarrow \mathsf{BaseSampler}()$
$b \xleftarrow{\$} \{0,1\}$
$y \leftarrow b + (2b-1)y_+$
$x \leftarrow \frac{(y-r)^2}{2\sigma^2} - \frac{y_+^2}{2\sigma_{\max}^2}$
**return** $z \leftarrow y + \lfloor c \rfloor$ with probability $\frac{\sigma_{\min}}{\sigma} \cdot \exp(-x)$, otherwise restart.

the execution of the ffSampling procedure (namely, at the positions corresponding to the first two and last two calls to SamplerZ), and are very rare otherwise.

This sensitivity to floating point errors gives rise to concrete attacks against various Falcon-based constructions, that will be discussed in later sections.

### 4.1 The Falcon integer Gaussian sampler

The integer Gaussian sampling algorithm of Falcon can be described as follows. To sample from $D_{\mathbb{Z},\sigma,c}$, the algorithm SamplerZ first shifts the center $c$ to its fractional part $r = c - \lfloor c \rfloor \in [0,1)$. Next it produces a base sample $y_+ \sim D^+_{\mathbb{Z}^+,\sigma_{\max}}$ and converts $y_+$ into a bimodal Gaussian $y = b + (2b-1)y_+$, where $b$ is uniformly random in $\{0,1\}$. Then rejection sampling is performed to ensure that $y$ follows the correct discrete Gaussian distribution of $y$ with center $r$ (and to guarantee isochronicity [HPRR20]), and the final output is $z = y + \lfloor c \rfloor$. A formal description of SamplerZ is given in Algorithm 3.

In the context of ffSampling, the execution of one call to SamplerZ has a direct impact on the outputs of the subsequent calls in two different ways. On the one hand, the output of SamplerZ affects the computation of the centers of later integer discrete Gaussians. On the other hand, since all integer samplings share the same random tape, the randomness consumption of each integer sampling also affects all subsequent results. Therefore, one can think of the algorithm SamplerZ as sending a triple $(\sigma, c, \mathsf{randbytes})$ to a pair $(z, \mathsf{randbytes})$, where randbytes is the state of the pseudorandom number generator. That pseudorandom number generator is used to generate the following random values:

- the generation of $y_+$ uses the first 9 bytes;
- the generation of $b$ uses the next byte;
- the rejection sampling uses a varying number of random bytes (a little over 1 byte on average) and if rejection happens, the previous samplings are of course repeated.

### 4.2 Sensitivity analysis

Significant efforts have been made in analyzing the numerical precision required on the inputs and in the computation of the integer Gaussian sampler [DN12, MW17, Pre17]. It is generally accepted that IEEE-754 double precision floating-point arithmetic is sufficient for most practical schemes including Falcon. It is worth noting that all these works focus on the *distribution* output by the sampler from a statistical standpoint.

In this work, we investigate how float-point errors on the center and the standard deviation affect the *execution* of SamplerZ rather than its distribution. To this end, we treat SamplerZ as a *function* with Gaussian parameters and randomness as input. We call the executions of two calls to $\mathsf{SamplerZ}(\sigma, c, \mathsf{randbytes})$ consistent if their all intermediate discrete samples $(y_+, b, y, z)$ are the same, and inconsistent otherwise.

Lemma 1 shows that for a nearly-integer center, SamplerZ can be sensitive to floating-point errors. Intuitively, a minor floating-point error is sufficient to trigger inconsistent outputs of the floor function around integers.

**Lemma 1 (Sensitivity of the centers).** *Let $c$ and $c'$ be two floating-point numbers such that $|c - c'| \le \epsilon < 1$. For $\sigma \in [\sigma_{\min}, \sigma_{\max}]$ and a PRNG state* randbytes*,*

1. *if $\lfloor c \rfloor = \lfloor c' \rfloor$, then* $\mathsf{SamplerZ}(\sigma, c, \mathsf{randbytes})$ *and* $\mathsf{SamplerZ}(\sigma, c', \mathsf{randbytes})$ *have the same execution with probability $\ge 1 - \frac{11\epsilon}{\sigma^2}$ over the randomness of* randbytes*;*
2. *if $\lfloor c \rfloor \ne \lfloor c' \rfloor$, then* $\mathsf{SamplerZ}(\sigma, c, \mathsf{randbytes})$ *and* $\mathsf{SamplerZ}(\sigma, c', \mathsf{randbytes})$ *have an inconsistent execution.*

*Proof.* Let $(r, y_+, b, y, x, z)$ and $(r', y'_+, b', y', x', z')$ denote the intermediate values in one repetition of $\mathsf{SamplerZ}(\sigma, c, \mathsf{randbytes})$ and $\mathsf{SamplerZ}(\sigma, c', \mathsf{randbytes})$ respectively. At each repetition, the computation before the rejection sampling consumes a fixed number of random bytes, which implies that $(y_+, b, y) = (y'_+, b', y')$ once the repetition in two executions begins with the same random tape.

When $\lfloor c \rfloor = \lfloor c' \rfloor$, the execution discrepancy must stem from rejection sampling that is implemented by lazy Bernoulli sampler. For uniformly random randbytes, the probability of inconsistent executions $\mathsf{SamplerZ}(\sigma, c, \mathsf{randbytes}) \ne \mathsf{SamplerZ}(\sigma, c', \mathsf{randbytes})$ is thus the probability of an inconsistency in rejection sampling, which is:

$$\frac{\sigma_{\min}}{\sigma}\,\mathbb{E}[|\exp(-x) - \exp(-x')|].$$

Now we can compute:

$$\begin{aligned}
&\mathbb{E}[|\exp(-x) - \exp(-x')|] \\
&= \sum_{y\in\mathbb{Z}} \left| \exp\Big(-\frac{(y-r)^2}{2\sigma^2} + \frac{y_+^2}{2\sigma_{\max}^2}\Big) - \exp\Big(-\frac{(y-r')^2}{2\sigma^2} + \frac{y_+^2}{2\sigma_{\max}^2}\Big)\right| \cdot \frac{\exp\big(-\frac{y_+^2}{2\sigma_{\max}^2}\big)}{2\rho_{\sigma_{max}}(\mathbb{Z}^+)} \\
&= \frac{1}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \sum_{y\in\mathbb{Z}} \left| \exp\Big(-\frac{(y-r)^2}{2\sigma^2}\Big) - \exp\Big(-\frac{(y-r')^2}{2\sigma^2}\Big)\right| \\
&= \frac{1}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \sum_{y\in\mathbb{Z}} \exp\Big(-\frac{(y-r)^2}{2\sigma^2}\Big)\left|1 - \exp\Big(\frac{(r'-r)y}{\sigma^2} + \frac{(r^2-r'^2)}{2\sigma^2}\Big)\right| \\
&\le \frac{1}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \sum_{y\in\mathbb{Z}} \exp\Big(-\frac{(y-r)^2}{2\sigma^2}\Big)\Big(1 - \exp\big(-\frac{19\epsilon}{\sigma^2}\big)\Big) \\
&\le \frac{\rho_\sigma(\mathbb{Z})}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \frac{19\epsilon}{\sigma^2} \;\approx \frac{\sigma\sqrt{2\pi}}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \frac{19\epsilon}{\sigma^2} \le \frac{8.5\epsilon}{\sigma}.
\end{aligned}$$

The first inequality is due to the fact that $|r - r'| = |c - c'| \le \epsilon$ and $|y| \le 18$. The second inequality follows from $\rho_{\sigma,r}(\mathbb{Z}) \le \rho_\sigma(\mathbb{Z})$. This concludes the proof of the first assertion.

Conversely, suppose that we have consistent executions $\mathsf{SamplerZ}(\sigma, c, \mathsf{randbytes})$ and $\mathsf{SamplerZ}(\sigma, c', \mathsf{randbytes})$. Then in particular, the intermediate values $y, z$ and $y', z'$ coincide, which implies that $z - y = \lfloor c \rfloor$ coincides with $z' - y' = \lfloor c' \rfloor$. Hence, if $\lfloor c \rfloor \ne \lfloor c' \rfloor$, the executions are necessarily inconsistent. □

Contrary to the integer Gaussian center, the standard deviation has a strong error tolerance.

**Lemma 2 (Non-sensitivity of the standard deviations).** *Let $\sigma, \sigma' \in [\sigma_{\min}, \sigma_{\max}]$ such that $|\sigma - \sigma'| \le \epsilon < 2^{-10}$. Then* $\mathsf{SamplerZ}(\sigma, c, \mathsf{randbytes})$ *and* $\mathsf{SamplerZ}(\sigma', c, \mathsf{randbytes})$ *have consistent executions with probability $\ge 1 - 160\epsilon$ over the randomness of* randbytes *for any $c$.*

*Proof.* We follow the notations in the proof of Lemma 1. By a similar argument, it suffices to investigate the difference within rejection sampling. The probability of inconsistent executions $\mathsf{SamplerZ}(\sigma, c, \mathsf{randbytes}) \ne$

$\mathsf{SamplerZ}(\sigma', c, \mathsf{randbytes})$ over the randomness of $\mathsf{randbytes}$ is again $\frac{\sigma_{\min}}{\sigma}\,\mathbb{E}[|\exp(-x) - \exp(-x')|]$ and we have:

$$
\begin{aligned}
&\mathbb{E}[|\exp(-x) - \exp(-x')|] \\
&= \sum_{y\in\mathbb{Z}} \left| \exp\Big(-\frac{(y-r)^2}{2\sigma^2} + \frac{y_+^2}{2\sigma_{\max}^2}\Big) - \exp\Big(-\frac{(y-r)^2}{2\sigma'^2} + \frac{y_+^2}{2\sigma_{\max}^2}\Big)\right| \cdot \frac{\exp\big(-\frac{y_+^2}{2\sigma_{\max}^2}\big)}{2\rho_{\sigma_{max}}(\mathbb{Z}^+)} \\
&= \frac{1}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \sum_{y\in\mathbb{Z}} \left| \exp\Big(-\frac{(y-r)^2}{2\sigma^2}\Big) - \exp\Big(-\frac{(y-r)^2}{2\sigma'^2}\Big)\right| \\
&= \frac{1}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \sum_{y\in\mathbb{Z}} \exp\Big(-\frac{(y-r)^2}{2\sigma^2}\Big)\left|1 - \exp\Big((y-r)^2 \cdot \frac{\sigma^2 - \sigma'^2}{2\sigma^2\sigma'^2}\Big)\right| \\
&\le \frac{1}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \sum_{y\in\mathbb{Z}} \exp\Big(-\frac{(y-r)^2}{2\sigma^2}\Big)\left|1 - \exp\Big(361 \cdot \frac{\sigma^2 - \sigma'^2}{2\sigma^2\sigma'^2}\Big)\right| \\
&\le \frac{1}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \sum_{y\in\mathbb{Z}} \exp\Big(-\frac{(y-r)^2}{2\sigma^2}\Big)\Big(\exp\big(361 \cdot \frac{\epsilon(\sigma+\sigma')}{2\sigma^2\sigma'^2}\big) - 1\Big) \\
&\le \frac{\rho_\sigma(\mathbb{Z})}{2\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot \frac{400\epsilon\cdot\sigma_{\max}}{\sigma^2\sigma'^2} \le \frac{\sigma_{\max}\sqrt{2\pi}}{\sigma_{\min}^3\rho_{\sigma_{\max}}(\mathbb{Z}^+)} \cdot 200\epsilon \le 160\epsilon.
\end{aligned}
$$

This completes the proof. □

### 4.3 Integer centers in Gaussian sampling

As indicated by Lemma 1, the integer Gaussian sampler of FALCON is sensitive to floating-point errors when the center is (within a small interval around) an integer value.

Now, while the centers in the successive calls to $\mathsf{SamplerZ}$ throughout the computation of a FALCON signature are all represented as floating point values, their exact theoretical values (if all computations were to be carried out with unlimited precision) are rational numbers, and some of the denominators involved are relatively small. As a result, certain centers have a non-negligible chance of having an integer their exact theoretical value, which gives rise to an *almost* integer floating point value in the actual computation, with high sensitivity to floating point errors.

We now argue that, in FALCON, this is the case for the first two and the last two centers sampled in the $\mathsf{ffSampling}$ procedure, whereas other coefficients are very unlikely to have an exact theoretical value equal to an integer.

Indeed, denote by $c_i$, $0 \le i \le 2n-1$ the exact theoretical value of the center output by the call of index $i$ to $\mathsf{SamplerZ}$ within the $\mathsf{ffSampling}$ algorithm. Then, by standard properties of the fast Fourier Gaussian sampling algorithm, $c_i$ has the following form:

$$c_i = \frac{\langle \mathbf{c}_i, \mathbf{b}^*_{2n-1-i}\rangle}{\|\mathbf{b}^*_{2n-1-i}\|^2},$$

where $\mathbf{b}^*_{2n-1-i}$ is one of the vectors of the Gram–Schmidt orthogonalization $\widetilde{\mathbf{B}} = (\mathbf{b}^*_0, \ldots, \mathbf{b}^*_{2n-1})$ of the NTRU trapdoor basis of FALCON in *bit reversed order*.

Several properties of this Gram–Schmidt basis $\widetilde{\mathbf{B}}$ are established in Theorem 1. In particular, let $m_k = \prod_{i=0}^{k-1} \|\mathbf{b}^*_{2i}\|^2$ for $0 \le k \le n$. Then, by assertions (c) and (e) of that theorem, $m_k$ is an integer of the same order of magnitude as $q^k$. Moreover, by assertions (g) and (h), the $(2k)$-th and $(2k+1)$-st centers:

$$c_{2k} = \frac{\langle \mathbf{c}_{2k}, \mathbf{b}^*_{2n-1-2k}\rangle}{\|\mathbf{b}^*_{2n-1-2k}\|^2} \quad \text{and} \quad c_{2k+1} = \frac{\langle \mathbf{c}_{2k+1}, \mathbf{b}^*_{2n-1-(2k+1)}\rangle}{\|\mathbf{b}^*_{2n-1-(2k+1)}\|^2}$$

are rational numbers with denominator dividing both $qm_k$ and $m_{n-k}$. By assertion (i) of the same theorem, $qm_k$ actually divides $m_{n-k}$ for $k < n/2$, and vice versa for $k \geq n/2$.

Denoting by $g_k$ the smaller of those two values, $g_k \cdot \mathbf{b}_j^*/\|\mathbf{b}_j^*\|^2$ for $j = 2n-1-2k$ and $j = 2n-1-(2k+1)$ is itself an integer vector, with coefficients that are not expected to satisfy any particular arithmetic relation, and therefore those coefficients are expected to be setwise coprime with overwhelming probability $\approx 1/\zeta(2n)$. As a result, since $\mathbf{c}_{2k}$ and $\mathbf{c}_{2k+1}$ are somewhat random, we heuristically expect the probability of $c_{2k}$ being an integer to be $1/g_k$, and similarly for $c_{2k+1}$. This is summarized as Heuristic 1.

**Heuristic 1.** *For $0 \leq k \leq n-1$, let:*

$$m_k = \prod_{i=0}^{k-1} \|\mathbf{b}_{2i}^*\|^2 \quad \textit{and} \quad g_k = \begin{cases} q \cdot m_k & \textit{if } k < n/2; \\ m_{n-k} & \textit{otherwise.} \end{cases}$$

*Then, the (exact theoretical values of the) centers $c_{2k}$ and $c_{2k+1}$ each have a probability $1/g_k$ of being integers.*

Under Heuristic 1, $c_0$ and $c_1$ each have a probability equal to $1/g_0 = 1/q$ to be integers, and for $c_{2n-2}$ and $c_{2n-1}$, the probability is $1/g_{n-1} = 1/m_1 = 1/\|\mathbf{b}_0^*\|^2 = 1/\|(g,-f)\|^2$. Both probabilities are between $1/10000$ and $1/20000$, so we should expect to observe integer centers at those positions once every few thousand signatures.

On the other hand, $g_k \approx q^{k+1}$ for $k < n/2$ and $q^{n-k}$ for $k \geq n/2$. One therefore concludes that all centers except the first and last two have a denominator $\gtrsim q^2$. This already yields a very low probability of getting an integer center in theory (occurring at best once every few tens of millions of signatures). Moreover, except for the centers corresponding to $k \leq 2$ or $k \geq n-3$ (i.e., the first six and the last six), the values $g_k$ even have a bit size exceeding the floating point precision, and thus even if integer centers happen to occur in theory, this won't be detectable in the double precision floating point computations.

*Remark 1.* The reason why Heuristic 1 is a heuristic and not a theorem is twofold.

On the one hand, there is a small chance that the coefficients of $\mathbf{u}_j = g_k \cdot \mathbf{b}_j^*/\|\mathbf{b}_j^*\|^2$ for $j = 2n-1-2k$ and $j = 2n-1-(2k+1)$ might not be setwise coprime, so that $\mathbf{b}_j^*/\|\mathbf{b}_j^*\|^2$ actually has a slightly smaller denominator than $g_k$. This should only happen with negligible probability around $1-\zeta(2n) \approx 2^{-2n}$, but making this estimate rigorous is tedious.

On the other hand, even if we can make the distributions of $\mathbf{c}_{2k}$ and $\mathbf{c}_{2k+1}$ more explicit than saying that they are "somewhat random" (they should in fact be uniform modulo $q$), it is not possible to get a probability of exactly $1/g_k$ of getting an integer due to counting reasons ($g_k$ does not usually divide $q^{2n}$). A rigorous way of establishing that the probability is *close* to $1/g_k$ involves applying the leftover hash lemma to the 2-universal hash $\mathbf{c} \mapsto \langle \mathbf{c}, \mathbf{u}_j \rangle \bmod g_k$, but we omit the details of that argument.

## 5 Exploiting Floating Point Discrepancies

As shown in Section 4, the integer Gaussian sampler of Falcon can have a different execution when some floating-point error is introduced on an integer center. Due to this sensitivity and the *weak determinism* of floating-point arithmetic, various Falcon implementations can generate distinct signatures for the *same* digest syndrome, as confirmed in Section 6 below.

This section demonstrates the insecurity of Falcon signatures for the same digest syndrome in the presence of floating point discrepancies. A key recovery attack can be mounted once close discrepant signatures are released. To clarify the impact of this attack, Section 5.2 showcases some Falcon-based schemes allowing the adversary to make two signing queries associated with the same syndrome.

### 5.1 Key recovery from signature discrepancies

In a GPV lattice signature scheme, the signing procedure proceeds in two steps. First, the signer hashes the message to a point $\mathbf{u} = \mathsf{Hash}(\mathsf{msg})$, the *syndrome*, in the ambient space of the underlying lattice $\mathcal{L}$. Then, the signer performs trapdoor sampling to compute a vector $\mathbf{v} \in \mathcal{L}$ close to the syndrome $\mathbf{u}$ and outputs $\mathbf{s} = \mathbf{v} - \mathbf{u}$ as the signature.

For a syndrome $\mathbf{u}$, the signing procedure of FALCON samples an integer vector $\mathbf{z} = (z_0, z_1) \in \mathcal{R}^2$ using ffSampling and outputs (a compressed form of):

$$\mathbf{s} = \mathbf{u} - \mathbf{z} \cdot \mathbf{B}_{f,g} = \mathbf{u} - \mathbf{z} \cdot \begin{pmatrix} g & -f \\ G & -F \end{pmatrix}$$

as a signature where $\mathbf{s} = (s_0, s_1) \sim D_{\mathcal{L}(\mathbf{B}_{f,g})+\mathbf{u},\sigma_{\mathrm{sig}}}$ is short. Let $\mathbf{s}, \mathbf{s}'$ be two distinct signatures for the same $\mathbf{u}$. Their difference is

$$\Delta\mathbf{s} = (s_0 - s_0', s_1 - s_1') = (\mathbf{z} - \mathbf{z}') \cdot \begin{pmatrix} g & -f \\ G & -F \end{pmatrix} = (z_0 - z_0', z_1 - z_1') \begin{pmatrix} g & -f \\ G & -F \end{pmatrix}$$

and then we have

$$\begin{aligned} \Delta s_0 &= s_0 - s_0' = (z_0 - z_0') \cdot g + (z_1 - z_1') \cdot G, \\ \Delta s_1 &= s_1 - s_1' = (z_0 - z_0') \cdot (-f) + (z_1 - z_1') \cdot (-F). \end{aligned}$$

The ffSampling (Algorithm 2) samples the vector $\mathbf{z}$ coefficient by coefficient in a tree-wise fashion, essentially from right to left in *bit reversed order*, each coefficient being the output of the SamplerZ algorithm discussed in the previous section.

If we denote by $z_{(i)}$, $0 \le i \le 2n-1$, the coefficient of $\mathbf{z}$ output by the $(i+1)$-st call to SamplerZ as part of the traversal of the FALCON tree, then the position of $z_{(i)}$ is depicted in Fig. 1 where the vector $\mathbf{z}$ has its components in bit reversed order for each of the two ring elements, and in Fig. 2 in the standard monomial order.

Now, floating point discrepancies arising for the instability of SamplerZ only have a good chance to occur in the first or the last two calls to SamplerZ, i.e., at coefficient $z_{(0)}$, $z_{(1)}$, $z_{(2n-2)}$ or $z_{(2n-1)}$.

When the discrepancy occurs at coefficient $z_{(0)}$ or $z_{(1)}$, the entire remainder of the ffSampling computation is affected, which yields a large, somewhat unstructured difference $\Delta\mathbf{s}$ between the two signatures output by the algorithm. This vector $\Delta\mathbf{s}$ is a short lattice vector, but it is not expected to be short enough to make key recovery feasible.

In contrast, when the discrepancy occurs at coefficient $z_{(2n-2)}$ or $z_{(2n-1)}$, *only* those two indices are affected, making the difference vector $\Delta\mathbf{s}$ highly structured. Indeed, as seen on Fig. 2, $z_{(2n-2)}$ corresponds to the element of degree 0 of the ring element $z_0$, and $z_{(2n-1)}$ to the element of degree $n/2$ of $z_0$. Therefore:

$$\Delta s_0 = (z_0 - z_0') \cdot g = \Delta z_0 \cdot g \quad \text{and} \quad \Delta s_1 = (z_0 - z_0') \cdot (-f) = \Delta z_0 \cdot (-f),$$

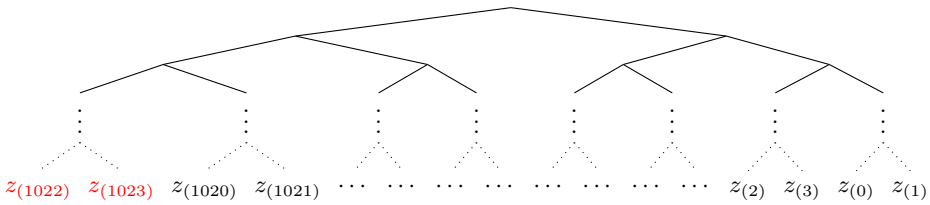


**Fig. 1.** Positions of the bit reversed order coefficients of $\mathbf{z}$ in the order of their generation as outputs of SamplerZ in ffSampling (case of FALCON–512, where $2n = 1024$).

where $\Delta z_0 = a + b \cdot x^{n/2}$ and $(a, b) = \left(z_{(2n-2)} - z'_{(2n-2)}, z_{(2n-1)} - z'_{(2n-1)}\right)$. Since $z_{(i)}$ and $z'_{(i)}$ are sampled by SamplerZ around almost the same centers, we know that $a, b \in \{-18, -17, \ldots, 0, \ldots, 18, 19\}$. A simple exhaustive search on the space of pairs $(a, b)$ of cardinality $38^2 < 2^{11}$ is thus sufficient for complete key recovery.

### 5.2 Cryptanalytic impact on Falcon and its variants

The attack described in this section is only possible to the extent that the adversary can make two signing queries giving rise to the same syndrome.

This normally does *not* happen in plain Falcon signatures, since the random salt in signatures should ensure that even multiple signatures on the same message are sampled with distinct syndromes. Care may still be warranted in that setting, however, as improperly repeated randomness does occasionally happen in the real world with catastrophic cryptographic consequences, as evidenced for example by the observation of pairs of RSA keys in the wild with nontrivial GCDs [HDWH12].

The attack is, however, much more directly relevant for *derandomized* variants of Falcon. This includes in particular the "deterministic Falcon" signature scheme specified and implemented by Lazar and Peikert [LP21a], with the goal of achieving SNARK-friendliness. Unsalted Falcon signatures are also considered for similar reasons by Aardal et al. [AAB+24] to achieve more efficient (and asymptotically sublinear) aggregation for Falcon signatures.

Derandomized variants of Falcon are also used in primitives other than signatures to provide efficient lattice trapdoors. As mentioned in the Falcon specification [PFH+22, §2.2.1], one can directly derive an identity-based encryption scheme from derandomized Falcon, similarly to [DLP14]: Falcon key generation becomes setup and signature generation becomes key extraction on a given identity. This exact design is used in the Latte (H)IBE [ZMS+24] considered for NCSC and ETSI standardization.

In those various settings, we argue that the attack is possible and invalidates the standard security definitions of the corresponding primitives.

*Deterministic Falcon.* The case of deterministic Falcon is straightforward: a chosen-message attack adversary can query the signer for two signatures on the same fixed, arbitrary message. In the presence of floating-point discrepancies (which can occur for all the reasons discussed in Section 6 below and more), the attack we have described shows that the two signatures will have a non-negligible probability (of one in a few thousands) of differing in a way that leads to full key recovery. This violates unbreakability under chosen-message attacks, and a fortiori existential unforgeability.

In all fairness, one should note that Lazar and Peikert's implementation of deterministic Falcon specifically warns against the use of any floating-point mode other than fpemu (though other modes remain included in the code base), and heeding this warning rules an entire class of possible sources of floating-point discrepancies. It *does not* however warn against the concurrent use of the "dynamic" and "tree" variants of the signing algorithm, which lead to similar issues. The suggestion of unsalted Falcon signatures in [AAB+24] also fails to include any particular security caveat.

Incidentally, we note that one can easily imagine real-world scenarios in which this vulnerability would have a substantial impact. Deterministic Falcon is particularly considered for blockchain-related applications; in such a context, an adversary can passively scan the blockchain, waiting for a discrepancy to appear, and skim off the corresponding private key when it happens (in a similar fashion as for Bitcoin signatures with repeated nonces [BHH+14, §4.4]).

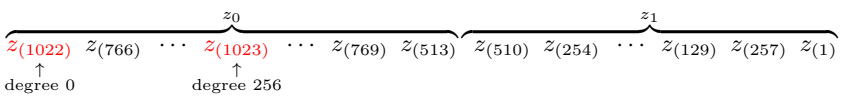


**Fig. 2.** Positions of the standard monomial order coefficients of $\mathbf{z}$ in the order of their generation as outputs of SamplerZ (case of Falcon–512).

*Falcon-based IBE.* Even in the weakest security definition for identity-based encryption, the adversary is allowed to query the master key authority for the secret key associated with a fixed identity, and do so twice. When doing so, in the presence of floating-point discrepancies, there is again a non-negligible probability that the two extracted keys (a.k.a. Falcon signatures) will differ in a way that fully exposes the authority's master secret key! This means that, when floating-point discrepancies can occur, a Falcon-based IBE will fail to even satisfy IND-sIDCPA (or even UBK-sIDCPA) security.

Moreover, the vulnerability is particularly severe in that it can manifest itself in the absence of any malicious party: indeed, a legitimate user can certainly query for the secret key on their *own* identity twice, and the authority's reply will have a chance of leaking the master secret key if they do. Deployments of Falcon-based IBE should therefore be especially careful to avoid all possible sources of floating-point discrepancies (and should also consider adopting some of the countermeasures we suggest at the end of this paper).

## 6 Sources of Floating-Point Discrepancies in Falcon

We have seen that, in the presence of floating-point discrepancies, two calls to the Falcon sampler on the same input can result in outputs that completely leak the private signing key. We now turn to the question of how such floating-point discrepancies can arise. We identify in particular two possible sources for such discrepancies, although there are undoubtedly more.

Firstly, the Falcon API exposes two variants of the signing procedure, called "dynamic" and "tree": in the dynamic mode, the Falcon tree is regenerated on the fly as part of the sampling algorithm, whereas the tree mode uses a precomputed Falcon tree. Interestingly, both modes carry out the same floating point operations on the same values in the same order, *except* on the last couple of levels of the tree, where the computations are done in a subtly different order. This introduces small discrepancies that we show are exploitable, and exist even when using the integer-based floating-point emulation of the Falcon reference implementation. We describe this issue in Section 6.1 below. We note that an observational mention of the existence of such discrepancies can be found in the literature [PKKK24], although that paper fails to give any explanation of how they arise.

Secondly, although it is less surprising, we verify in Section 6.2 that the use of fused multiply-add (FMA) floating-point instructions, which can be enabled in the Falcon code, can also cause signature discrepancies compared to the reference implementation.

In both two cases, discrepancies giving rise to a full key recovery appear between pairs of signatures at a frequency of once in a few thousands. In other words, collecting a few thousand pairs is sufficient to completely recover the key with high probability.

### 6.1 Discrepancies between two signing modes

The Falcon code offers two different signing modes: the "dynamic" mode sign_dyn and the "tree" mode sign_tree. In sign_dyn, the ffSampling algorithm dynamically computes the LDL$^*$ decomposition, i.e., the construction of the Falcon tree, as part of the lattice Gaussian sampling procedure. In sign_tree, the signing procedure uses takes the Falcon tree as input, and thus assumes that it has been precomputed and stored in memory. In summary, sign_tree supports faster signing while sign_dyn requires less RAM usage. As a result, both signing modes have their strong suits, and one may want to switch from one to the other and back depending on how frequently a given key is used, how much memory is available at a given point in time and whether spare cycles are available for precomputations.

*Discrepancies.* Let us have a close look at how the ffSampling algorithm (Algorithm 2) is concretely implemented in the two signing modes. The algorithm itself is recursive, and the implementations follow this recursive structure.

Consistently with the pseudo-code (Lines 1–5 of Algorithm 2), in sign_dyn, the deepest recursive layer, corresponding to $n = 1$, only contains the normalization of the leaf of the Falcon tree $\mathsf{T}$ and two calls to SamplerZ.

In contrast, in sign_tree, the deepest layer actually corresponds to $n = 4$, and effectively contains the last two recursion layers in inlined form. It includes 8 calls to SamplerZ, and inlined code equivalent to 2 split and 2 merge operations. Instead of recursively calling split_fft and merge_fft, it reimplements them directly. This could be done in a way that follows the same floating-point evaluation order, but the actual code introduces subtle differences in the order of the computations, which turn out to lead to floating-point discrepancies on the centers on the one-dimensional Gaussians. Note that only the centers are affected: the standard deviations, i.e., the FALCON tree leaves, have identical values in the two signing modes.

We exhibit the split (resp. merge) operation in sign_dyn and sign_tree via the code snippets in Listing 1 and Listing 3 (resp. Listing 2 and Listing 4). The discrepancies are marked with the color block (resp. ). The symbols of those code snippets are described in Fig. 3.

**Fig. 3.** Symbols in Listing 1, 3, 2 and 4. All involved operations are in fpr.

| Symbol | Description |
|---|---|
| a_re, a_im | the real and imaginary part of a complex number |
| FPC_ADD, FPC_SUB | the addition and subtraction of two complex numbers |
| FPC_MUL | the multiplication of two complex numbers |
| fpr_add, fpr_sub | the addition and subtraction of two numbers |
| fpr_mul | the multiplication of two numbers |
| fpr_half, fpr_neg | halve and negate the numbers |
| fpr_invsqrt2, fpr_invsqrt8 | the number of $1/\sqrt{2}$ and $1/\sqrt{8}$ |

```
for (u = 0; u < 1; u ++) {
    fpr a_re, a_im, b_re, b_im;
    fpr t_re, t_im;

    a_re = f[(u << 1) + 0];
    a_im = f[(u << 1) + 0 + 2];
    b_re = f[(u << 1) + 1];
    b_im = f[(u << 1) + 1 + 2];

    FPC_ADD(t_re, t_im, a_re, a_im, b_re, b_im);
    f0[u] = fpr_half(t_re);
    f0[u + 1] = fpr_half(t_im);

    FPC_SUB(t_re, t_im, a_re, a_im, b_re, b_im);
    FPC_MUL(t_re, t_im, t_re, t_im, fpr_invsqrt2,
    fpr_neg(fpr_invsqrt2));
    f1[u] = fpr_half(t_re);
    f1[u + 1] = fpr_half(t_im);
}
```

**Listing 1.** The for-loop of split_fft in the recursive layer $n = 4$ for the subroutine ffSampling of sign_dyn.

```
f[0] = f0[0];
f[2] = f1[0];
for (u = 0; u < 1; u ++) {
    fpr a_re, a_im, b_re, b_im;
    fpr t_re, t_im;

    a_re = f0[u];
    a_im = f0[u + 1];

    FPC_MUL(b_re, b_im, f1[u], f1[u + 1], fpr_invsqrt2,
    fpr_invsqrt2);


    FPC_ADD(t_re, t_im, a_re, a_im, b_re, b_im);
    f[(u << 1) + 0] = t_re;
    f[(u << 1) + 0 + 2] = t_im;
    FPC_SUB(t_re, t_im, a_re, a_im, b_re, b_im);
    f[(u << 1) + 1] = t_re;
    f[(u << 1) + 1 + 2] = t_im;
}
```

**Listing 2.** The for-loop of merge_fft in the recursive layer $n = 4$ for the subroutine ffSampling of sign_dyn.

```
a_re = t1[0];
a_im = t1[2];
b_re = t1[1];
b_im = t1[3];

c_re = fpr_add(a_re, b_re);
c_im = fpr_add(a_im, b_im);
w0 = fpr_half(c_re);
w1 = fpr_half(c_im);

c_re = fpr_sub(a_re, b_re);
c_im = fpr_sub(a_im, b_im);
w2 = fpr_mul(fpr_add(c_re, c_im), fpr_invsqrt8);
w3 = fpr_mul(fpr_sub(c_im, c_re), fpr_invsqrt8);
```

**Listing 3.** The first call reordered split operations in the recursive layer $n = 4$ for the subroutine ffSampling of sign_tree.

```
a_re = w0;
a_im = w1;
b_re = w2;
b_im = w3;


c_re = fpr_mul(fpr_sub(b_re, b_im), fpr_invsqrt2);
c_im = fpr_mul(fpr_add(b_re, b_im), fpr_invsqrt2);


z1[0] = w0 = fpr_add(a_re, c_re);
z1[2] = w2 = fpr_add(a_im, c_im);
z1[1] = w1 = fpr_sub(a_re, c_re);
z1[3] = w3 = fpr_sub(a_im, c_im);
```

**Listing 4.** The first call reordered merge operation in the recursive layer $n = 4$ for the subroutine ffSampling of sign_tree.

*Discussion.* To make entirely explicit how those source code differences yield floating-point discrepancies, let us write down the corresponding computations as formulas.

We first discuss the split operation split_fft. Given $t \in \mathsf{FFT}(\mathbb{Q}[x]/\phi)$, it can be uniquely split as $t = t_0(\zeta^2) + \zeta t_1(\zeta^2)$ in FFT domain, where $t_0, t_1 \in \mathsf{FFT}(\mathbb{Q}[x]/\phi')$ and $\phi' = x^{n/2} + 1$. In both sign_dyn and sign_tree, the split processes in ffSampling (Listings 1 and 3) at the recursive layer $n = 4$ are identical for the calculation of $t_0$, which is evaluated as:

$$t_0[0] = \frac{1}{2} \mathbin{\mathring{\times}} \Big(t[0] \mathbin{\mathring{+}} t[1]\Big) \qquad\qquad t_0[1] = \frac{1}{2} \mathbin{\mathring{\times}} \Big(t[2] \mathbin{\mathring{+}} t[3]\Big).$$

However, the calculations of $t_1$ proceed in different orders in sign_dyn and sign_tree. More concretely, the process in sign_dyn can be seen as the following expressions for $t_1$:

$$t_1[0] = \frac{1}{2} \mathbin{\mathring{\times}} \left(\frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} \Big(t[0] \mathbin{\mathring{-}} t[1]\Big) \mathbin{\mathring{-}} \Big(-\frac{1}{\sqrt{2}}\Big) \mathbin{\mathring{\times}} \Big(t[2] \mathbin{\mathring{-}} t[3]\Big)\right),$$
$$t_1[1] = \frac{1}{2} \mathbin{\mathring{\times}} \left(\Big(-\frac{1}{\sqrt{2}}\Big) \mathbin{\mathring{\times}} \Big(t[0] \mathbin{\mathring{-}} t[1]\Big) \mathbin{\mathring{+}} \frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} \Big(t[2] \mathbin{\mathring{-}} t[3]\Big)\right),$$

whereas in sign_tree, the following, slightly different equality holds:

$$t_1[0] = \frac{1}{2\sqrt{2}} \mathbin{\mathring{\times}} \Big(\Big(t[0] \mathbin{\mathring{-}} t[1]\Big) \mathbin{\mathring{+}} \Big(t[2] \mathbin{\mathring{-}} t[3]\Big)\Big),$$
$$t_1[1] = \frac{1}{2\sqrt{2}} \mathbin{\mathring{\times}} \Big(\Big(t[2] \mathbin{\mathring{-}} t[3]\Big) \mathbin{\mathring{-}} \Big(t[0] \mathbin{\mathring{-}} t[1]\Big)\Big).$$

Due to the fact that floating-point arithmetic does not satisfy the distributivity of multiplication over addition, $t_1$ ends up evaluating to a slightly different value in sign_dyn vs. sign_tree, and that value $t_1$ is one of the Gaussian centers passed to SamplerZ.

We next consider the merge operation merge_fft. Given $t_0, t_1 \in \mathsf{FFT}(\mathbb{Q}[x]/\phi')$, they can be merged into $t = t_0(\zeta^2) + \zeta t_1(\zeta^2) \in \mathsf{FFT}(\mathbb{Q}[x]/\phi)$. For the recursive layer $n = 4$, the merge operation in sign_dyn (Listing 2) is interpreted as following expression for $t$:

$$t[0] = t_0[0] \mathbin{\mathring{+}} \Big(\frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} t_1[0] \mathbin{\mathring{-}} \frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} t_1[1]\Big), \qquad t[2] = t_0[1] \mathbin{\mathring{+}} \Big(\frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} t_1[0] \mathbin{\mathring{+}} \frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} t_1[1]\Big),$$
$$t[1] = t_0[0] \mathbin{\mathring{-}} \Big(\frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} t_1[0] \mathbin{\mathring{-}} \frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} t_1[1]\Big), \qquad t[3] = t_0[1] \mathbin{\mathring{-}} \Big(\frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} t_1[0] \mathbin{\mathring{+}} \frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} t_1[1]\Big).$$

In sign_tree, the merge operation (Listing 4) can be written as the following expression instead:

$$t[0] = t_0[0] \mathbin{\mathring{+}} \frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} \Big(t_1[0] \mathbin{\mathring{-}} t_1[1]\Big), \qquad t[2] = t_0[1] \mathbin{\mathring{+}} \frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} \Big(t_1[0] \mathbin{\mathring{+}} t_1[1]\Big),$$
$$t[1] = t_0[0] \mathbin{\mathring{-}} \frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} \Big(t_1[0] \mathbin{\mathring{-}} t_1[1]\Big), \qquad t[3] = t_0[1] \mathbin{\mathring{-}} \frac{1}{\sqrt{2}} \mathbin{\mathring{\times}} \Big(t_1[0] \mathbin{\mathring{+}} t_1[1]\Big).$$

Again, since floating-point arithmetic is not distributive, $t_1$ may evaluate to different values in the two signing modes, which affects the centers of subsequent integer samplings.

*Experimental validation.* We experimentally verify the impact of reordered operations in the two signing modes of deterministic Falcon [LP21b]. We tested different implementations including the default fpemu_det and fpnative_det, avx2_det, avx2_fma_det for $n = 512$ and 1024. For each instance, 10 million signatures are generated using each of the two signing modes. All experiments are carried out on an Intel Xeon Gold 6338-based workstation, and all examples are compiled with GCC 9.4.0. Optimizations `-O3` are enabled, consistent with the `Makefile` of the provided implementation.

We observed that with 10 million signature queries, the two signing modes sign_dyn and sign_tree will generate a few hundred pairs of different signatures per instance. Detailed experimental results are provided in Table 2. Each table entry is of the form "A/B", where "A" represents the number of signatures with differences in just the last two integer samples, and "B" is the total number of different signature pairs.

Interestingly, for most instances, more than 70% of the discrepancies occur only in the last two calls to SamplerZ, which is the setting that allows for direct full key recovery as discussed earlier. This is somewhat unexpected. Indeed, by the results of Section 4.3, integer centers, which are sensitive to floating-point errors, occur slightly more frequently in the first two calls to SamplerZ than they do in the last two. However, for reasons that we do not fully understand, the probability of discrepancy on the last two calls *conditional* on having an integer center appears to be larger for the last two calls than the first two. This phenomenon is specific to the "dynamic" vs. "tree" discrepancy, and does not occur with, e.g., the FMA discrepancies discussed in the next section.

Exploiting those signature discrepancies, we mounted the key recovery attack shown in Section 5.1. Table 3 records the number of successful key recoveries for varying number of signatures. Around one in every 10,000 or so pairs of signatures appear to lead to full key recovery.

**Table 2.** Experimental results on deterministic Falcon.

| **Instance** | fpemu_det_512 | fpnative_det_512 | avx2_det_512 | avx2_fma_det_512 |
|---|---|---|---|---|
| 0 | 140 / 173 | 133 / 176 | 168 / 214 | 170 / 292 |
| 1 | 299 / 381 | 216 / 402 | 264 / 321 | 76 / 213 |
| 2 | 253 / 320 | 207 / 278 | 243 / 320 | 249 / 315 |
| 3 | 211 / 324 | 236 / 339 | 232 / 340 | 259 / 313 |
| 4 | 236 / 328 | 226 / 319 | 305 / 337 | 223 / 259 |
| 5 | 294 / 346 | 253 / 313 | 303 / 386 | 230 / 302 |
| 6 | 204 / 281 | 235 / 300 | 270 / 367 | 309 / 362 |
| 7 | 256 / 313 | 225 / 289 | 183 / 286 | 331 / 378 |
| 8 | 230 / 287 | 176 / 243 | 170 / 227 | 256 / 342 |
| 9 | 74 / 164 | 211 / 334 | 190 / 304 | 168 / 269 |
| **Instance** | fpemu_det_1024 | fpnative_det_1024 | avx2_det_1024 | avx2_fma_det_1024 |
| 0 | 234 / 297 | 244 / 313 | 255 / 303 | 214 / 277 |
| 1 | 259 / 339 | 281 / 361 | 223 / 260 | 191 / 281 |
| 2 | 224 / 290 | 244 / 346 | 206 / 299 | 282 / 348 |
| 3 | 232 / 303 | 226 / 283 | 251 / 300 | 224 / 282 |
| 4 | 242 / 317 | 264 / 341 | 260 / 326 | 276 / 355 |
| 5 | 267 / 328 | 219 / 266 | 161 / 265 | 279 / 316 |
| 6 | 221 / 282 | 214 / 321 | 204 / 268 | 217 / 306 |
| 7 | 244 / 304 | 185 / 277 | 232 / 299 | 209 / 283 |
| 8 | 253 / 332 | 216 / 276 | 315 / 409 | 241 / 323 |
| 9 | 276 / 306 | 203 / 297 | 253 / 324 | 37 / 110 |

### 6.2 Discrepancies caused by FMA floating-point instructions

For better efficiency, some optimized implementations of Falcon use the fused multiply-add (FMA) instructions that support the evaluation of $ab + c$ in a single instruction and with one floating-point rounding only. Compared to the regular "multiply then add" two-step computations, FMA instructions reduce the floating-point errors. On the flip side, they are also a source of floating-point discrepancies in Falcon signature generation.

**Table 3.** The number of successful key recovery with $N$ signature queries on deterministic FALCON.

| $N \times 10^{-3}$ | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|
| fpemu_det_512 | 1 | 4 | 6 | 6 | 6 | 7 | 8 | 8 | 8 | 8 |
| fpnative_det_512 | 2 | 5 | 7 | 7 | 8 | 8 | 10 | 10 | 10 | 10 |
| avx2_det_512 | 1 | 6 | 8 | 8 | 8 | 8 | 8 | 9 | 9 | 9 |
| avx2_fma_det_512 | 2 | 4 | 6 | 7 | 8 | 8 | 8 | 9 | 9 | 9 |
| fpemu_det_1024 | 5 | 6 | 6 | 6 | 7 | 7 | 7 | 8 | 8 | 9 |
| fpnative_det_1024 | 2 | 2 | 3 | 3 | 4 | 6 | 7 | 8 | 8 | 8 |
| avx2_det_1024 | 3 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 7 | 7 |
| avx2_fma_det_1024 | 1 | 3 | 4 | 7 | 8 | 9 | 9 | 9 | 10 | 10 |

*Experimental validation.* We ran experiments to compare the implementation using FMA of deterministic FALCON, avx2_fma_det with other implementations. Our experiments are performed in dynamic and tree signing modes respectively. For the same signing mode, we did not observe any discrepancies among fpemu_det, fpnative_det and avx2_det. Therefore, we only present the comparison between avx2_fma_det and the default implementation fpemu_det.

Table 4 shows the detailed experimental results measured over 10 million signatures per instance. Each table entry is again of the form "A/B", where "A" represents the number of signatures with differences in just the last two integer samples, and "B" is the total number of different signature pairs. We also present the experimental success rate of key recovery for this discrepancy in Table 5.

Compared to different signing modes (Table 2), the use of FMA instructions tends to cause a larger number of differing signature pairs This is because the FMA instructions are applied to a large amount of computations in both signing and key generation, which affects *almost all* floating-point intermediate values associated with ffSampling and has a more significant impact. In this setting, the discrepancies between avx2_fma_det and fpemu_det tend to occur in the first two calls of SamplerZ at a higher rate than in the last two calls, consistently with the results of Section 4.3, but unlike what happens for "dynamic" vs. "tree" discrepancies.

**Table 4.** Experimental results on avx2_fma_det and fpemu_det of deterministic FALCON.

| **Instance** | sign_dyn_512 | sign_tree_512 | sign_dyn_1024 | sign_tree_1024 |
|---|---|---|---|---|
| 0 | 274 / 748 | 344 / 852 | 483 / 991 | 432 / 987 |
| 1 | 316 / 819 | 340 / 720 | 380 / 828 | 446 / 1134 |
| 2 | 422 / 883 | 336 / 806 | 327 / 926 | 370 / 819 |
| 3 | 322 / 738 | 544 / 1205 | 328 / 763 | 396 / 1148 |
| 4 | 361 / 781 | 260 / 564 | 460 / 1138 | 409 / 803 |
| 5 | 448 / 1054 | 383 / 1011 | 292 / 630 | 452 / 1048 |
| 6 | 308 / 741 | 416 / 841 | 317 / 800 | 427 / 924 |
| 7 | 352 / 839 | 450 / 1066 | 438 / 782 | 358 / 726 |
| 8 | 292 / 767 | 341 / 858 | 377 / 784 | 534 / 1014 |
| 9 | 392 / 982 | 332 / 886 | 313 / 877 | 402 / 806 |

# 7 Countermeasures

## 7.1 Remedying the floating-point error sensitivity

As identified in Section 4, the one-dimensional discrete Gaussian sampler of FALCON is only sensitive to floating-point errors for a nearly-integer center. Unfortunately, (nearly) integer centers do occasionally

**Table 5.** The number of successful key recovery with $N$ signature queries on avx2_fma_det and fpemu_det of deterministic FALCON.

| $N \times 10^{-3}$ | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|
| sign_dyn_512 | 2 | 4 | 5 | 7 | 7 | 8 | 9 | 9 | 10 | 10 |
| sign_tree_512 | 4 | 6 | 6 | 8 | 8 | 8 | 8 | 9 | 9 | 9 |
| sign_dyn_1024 | 2 | 4 | 6 | 8 | 9 | 9 | 9 | 9 | 9 | 9 |
| sign_tree_1024 | 4 | 8 | 8 | 8 | 9 | 9 | 9 | 9 | 9 | 10 |

occur in FALCON signature generation. A natural solution to this issue could therefore be to shift the points at which the one-dimensional is numerically unstable to locations that *cannot* occur with significant probability within FALCON signing. This can be done as follows.

Algorithm 4: NewSamplerZ

**Input:** A center $c$ and standard deviation $\sigma \in [\sigma_{\min}, \sigma_{\max}]$
**Output:** An integer $z$ derived from a distribution close to $D_{\mathbb{Z},\sigma,c}$

$r \leftarrow c - \lfloor c \rceil$
$y_+ \leftarrow$ NewBaseSampler()
$b \xleftarrow{\$} \{0, 1\}$
$y \leftarrow (2b - 1)y_+$
$x \leftarrow \frac{(y-r)^2}{2\sigma^2} - \frac{y_+^2 - y_+}{2\sigma_{\max}^2}$
**return** $z \leftarrow y + \lfloor c \rceil$ with probability $\frac{\sigma_{\min}}{\sigma} \cdot \exp(-x)$, otherwise restart.

The reason why FALCON's discrete Gaussian sampler has sensitivity for integer centers is that the center received as input is first split into its integer and fractional parts, and computations are carried out on those two. It is not difficult, however, to modify the sampler to use rounding to the nearest integer instead of the floor operation. This results in Algorithm 4, where NewBaseSampler samples from the one-sided non-negative discrete Gaussian distribution centered at 1/2 of standard deviation $\sigma_{\max}$—or rather a truncation of it, such as the distribution $D$ given by:

$$D(i) \propto \begin{cases} \rho_{\sigma_{\max},\frac{1}{2}}(i) & \text{for } 1 \leq i \leq 18; \\ \frac{1}{2}\rho_{\sigma_{\max},\frac{1}{2}}(i) & \text{for } i = 0. \end{cases}$$

What this changes is that the algorithm now presents instability at *half-integers* (i.e., elements of $\mathbb{Z} + 1/2$) instead of integers, and preventing half-integer centers from appearing at all is feasible.

Indeed, let $m_k$ as in the statement of Theorem 1. As discussed in Section 4.3, under Heuristic 1, the centers $c_{2k}, c_{2k+1}$ are rational numbers of the form $n_k/g_k$ with $g_k = qm_k$ if $k < n/2$ and $g_k = m_{n-k}$ otherwise. Moreover, $m_k \approx q^k$, so for $3 \leq k < n-3$, $g_k$ exceeds the floating point precision, and hence even if the corresponding $c_{2k}$ or $c_{2k+1}$ happens to be half-integer for those $k$, this will not be detectable in the double precision floating point computations.

As a result, we can effectively avoid all half-integer centers if we ensure that the six denominators $g_0 = qm_0$, $g_1 = qm_1$, $g_2 = qm_2$, $g_{n-3} = m_3$, $g_{n-2} = m_2$ and $g_{n-1} = m_1$ are all odd. Since $q$ is itself an odd prime and $m_0 = 1$, this reduces to ensuring that $m_1$, $m_2$ and $m_3$. Now let:

$$t = \langle \mathbf{b}_0, \mathbf{b}_0 \rangle, \ u = \langle \mathbf{b}_0, \mathbf{b}_2 \rangle, \ v = \langle \mathbf{b}_0, \mathbf{b}_4 \rangle, \ w = \langle \mathbf{b}_0, \mathbf{b}_5 \rangle.$$

A tedious but straightforward computation shows that:

$$m_1 = t, \quad m_2 = t^2 - 2u^2, \quad m_3 = t^3 - 2t \cdot (u^2 + v^2 + w^2) + 2u(v - w)^2.$$

In particular, as long as $t = \|(g, -f)\|^2$ is odd, then so are $m_1$, $m_2$ and $m_3$.

Therefore, we conclude that the following steps constitute an effective countermeasure against the sensitivity of FALCON's sampler to floating-point errors:

1. replace the integer sampler by Algorithm 4, which is stable away from half-integer centers; and
2. restrict $\|(g, -f)\|^2$ to be an *odd* integer in key generation.

One issue with this countermeasure is that the reference C implementation of FALCON only generates keys with $\|(g, -f)\|^2$ *even*! Note that this is an idiosyncrasy of the C implementation itself, that is easily fixable, and does not reflect either the FALCON specification or alternate implementations like Prest's Python version, which *can* generate keys with odd $\|(g, -f)\|^2$.

This happens because the parity of $\|(g, -f)\|^2$ is simply the parity of the sum of all coefficients of $f$ and $g$, and the C implementation forces the sum of the coefficients of *both* $f$ and $g$ to be odd (so that their sum is even). This is unnecessary to solve the NTRU equation: one should ensure that *either* of the parity is odd (otherwise $f$ and $g$ are both divisible by the prime above 2 in the ring, and the NTRU equation is not solvable), but there is no reason to require *both* to be odd.[5] Modifying the FALCON code to lift this restriction (or rather, to impose instead that exactly one of $f$ and $g$ has an even coefficient sum) is simple and has no efficiency penalty—in fact, it is likely to lead to a slight speed-up, since it rejects half of the choices of $(g, -f)$ on parity grounds, whereas FALCON's C implementation rejects three quarters.

### 7.2 Eliminating discrepancies between dynamic and tree modes

As analyzed in Section 6, the basic reason why there exist floating point discrepancies between the sign_dyn and sign_tree modes (even when using IEEE-754 compliant floating point arithmetic) is the fact that the order of floating point operations differ between the two modes at the second-to-last level of recursion in the traversal of the FALCON tree by ffSampling. Therefore, the discrepancies should disappear provided that sign_tree is modified to follow the simpler ordering used by sign_dyn. We suggest two alternate approaches to achieve that goal.

*Re-ordering computations in* sign_tree. A first possible solution is to manually re-order the floating point computations in the bottom recursion layer of sign_tree (corresponding to $n = 4$) so that the inlined reimplementations of the split_fft and merge_fft that it contains match the floating point computations of the actual subroutines when called from sign_dyn. This boils down to modifying lines 13–14 of Listing 3 (corresponding to the split operation) and lines 6–7 of Listing 4 (corresponding to the merge operation) as shown below.

```
13  w2 = fpr_half(fpr_sub(fpr_mul(c_re, fpr_invsqrt2), fpr_mul(c_im, fpr_neg(fpr_invsqrt2))));
14  w3 = fpr_half(fpr_add(fpr_mul(c_re, fpr_neg(fpr_invsqrt2)), fpr_mul(c_im, fpr_invsqrt2)));
```

**Listing 5.** Countermeasure for Line 13 and 14 of Listing 3.

*Avoiding the re-ordered code path.* While the bottom recursion layer of sign_tree normally corresponds to $n = 4$, the code base actually contains, for testing purposes, another possible bottom layer for $n = 2$ that only gets called when FALCON is compiled for a base ring with extension degree 2 (instead of 512 or 1024 in proposed parameters).

[5] It appears that the FALCON C implementation does so in order to take some shortcuts in the computation of the extended GCD algorithm on the algebraic norms of $f$ and $g$. The underlying algorithm [Por20] does support the case of one operand being even, but by assuming that both operands are odd, the FALCON implementation possibly saves a parity test and a swap?

```
6 c_re = fpr_sub(fpr_mul(b_re, fpr_invsqrt2), fpr_mul(b_im, fpr_invsqrt2));
7 c_im = fpr_add(fpr_mul(b_re, fpr_invsqrt2), fpr_mul(b_im, fpr_invsqrt2));
```

**Listing 6.** Countermeasure for Line 6 and 7 of Listing 4.

This alternate recursion layer does not include any re-ordering like the $n = 4$ layer does, and instead follows the same floating point operations as sign_dyn. Therefore, a very simple countermeasure for sign_tree is to simply skip the code path corresponding to the bottom layer for $n = 2$ (e.g., by commenting out the `if (logn == 2)` conditional clause at the beginning of `ffSampling_fft`), and instead let the program flow fall over to the $n = 2$ bottom layer.

In fact, there is an even simpler $n = 1$ bottom recursion layer that exists in the code base in commented out form, and that can be used instead for the same purpose.

*Experimental validation.* We tested the effectiveness of those countermeasures by generating 10 million pairs of "dynamic" and "tree" signatures for deterministic Falcon, and as expected, did not observe any floating point discrepancies when either of the two countermeasures above are implemented. They did not appear to have measurable effect on performance either (although the second approach may be slightly slower when using AVX optimizations).

### 7.3 Avoiding native or optimized floating-point implementations

As illustrated by our findings regarding FMA instructions, a deterministic Falcon implementation should certainly avoid such floating point arithmetic optimizations known to break strict compliance with the IEEE-754 standard [iee85].

The use of native or vectorized floating-point arithmetic with FMA disabled was not found to produce discrepancies in our limited testing, but this may not be a robust observations across CPU architectures, compilers and choices of compiler options. It should be safe to rely on native floating-points if strict compliance with IEEE-754 is ensured, but this may be difficult to do in a portable way.

Failing that, deterministic Falcon implementations may prefer to heed the warning of Lazar and Peikert [LP21b] and completely avoid all floating-point implementations other than the integer-based emulated one. This is unfortunately a costly choice in terms of performance (and does not dispense from applying at least the countermeasure from the previous section if both dynamic and tree modes are exposed by the API).

## References


iee85. *IEEE standard for binary floating-point arithmetic.* IEEE, 1985.

AAB+24. Marius A. Aardal, Diego F. Aranha, Katharina Boudgoust, Sebastian Kolby, and Akira Takahashi. Aggregating falcon signatures with LaBRADOR. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part I*, volume 14920 of *LNCS*, pages 71–106, Santa Barbara, CA, USA, August 18–22, 2024. Springer, Cham, Switzerland.

BHH+14. Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 157–175, Christ Church, Barbados, March 3–7, 2014. Springer, Berlin, Heidelberg, Germany.

DOTT21. Ivan Damgård, Claudio Orlandi, Akira Takahashi, and Mehdi Tibouchi. Two-round n-out-of-n and multi-signatures and trapdoor commitment from lattices. In Juan Garay, editor, *PKC 2021, Part I*, volume 12710 of *LNCS*, pages 99–130, Virtual Event, May 10–13, 2021. Springer, Cham, Switzerland.

DGPY20. Léo Ducas, Steven Galbraith, Thomas Prest, and Yang Yu. Integral matrix gram root and lattice gaussian sampling without floats. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 608–637, Zagreb, Croatia, May 10–14, 2020. Springer, Cham, Switzerland.

DLP14. Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over NTRU lattices. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 22–41, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Berlin, Heidelberg, Germany.

DN12. Léo Ducas and Phong Q. Nguyen. Faster Gaussian lattice sampling using lazy floating-point arithmetic. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 415–432, Beijing, China, December 2–6, 2012. Springer, Berlin, Heidelberg, Germany.

DP16. Léo Ducas and Thomas Prest. Fast Fourier Orthogonalization. In *ISSAC 2016*, pages 191–198, 2016.

EFG+22. Thomas Espitau, Pierre-Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: A simpler, parallelizable, maskable variant of falcon. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 222–253, Trondheim, Norway, May 30 – June 3, 2022. Springer, Cham, Switzerland.

ENS+23. Thomas Espitau, Thi Thu Quyen Nguyen, Chao Sun, Mehdi Tibouchi, and Alexandre Wallet. Antrag: Annular NTRU trapdoor generation - making mitaka as secure as falcon. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VII*, volume 14444 of *LNCS*, pages 3–36, Guangzhou, China, December 4–8, 2023. Springer, Singapore, Singapore.

FKT+20. Pierre-Alain Fouque, Paul Kirchner, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Key recovery from Gram-Schmidt norm leakage in hash-and-sign signatures over NTRU lattices. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 34–63, Zagreb, Croatia, May 10–14, 2020. Springer, Cham, Switzerland.

Gal12. Steven D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012.

GHN06. Nicolas Gama, Nick Howgrave-Graham, and Phong Q. Nguyen. Symplectic lattice reduction and NTRU. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 233–253, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Berlin, Heidelberg, Germany.

GPV08. Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 197–206, Victoria, BC, Canada, May 17–20, 2008. ACM Press.

GMRR22. Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on falcon. *IACR TCHES*, 2022(3):141–164, 2022.

HFH16. Marcella Hastings, Joshua Fried, and Nadia Heninger. Weak keys remain widespread in network devices. In Phillipa Gill, John S. Heidemann, John W. Byers, and Ramesh Govindan, editors, *ACM IMC 2016*, pages 49–63. ACM, 2016.

HDWH12. Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security 2012*, pages 205–220, Bellevue, WA, USA, August 8–10, 2012. USENIX Association.

HPRR20. James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous gaussian sampling: From inception to implementation. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 53–71, Paris, France, April 15–17, 2020. Springer, Cham, Switzerland.

KA21. Emre Karabulut and Aydin Aysu. FALCON down: Breaking FALCON post-quantum signature scheme through side-channel attacks. In *DAC 2021*, pages 691–696. IEEE, 2021.

LP21a. David Lazar and Chris Peikert. Deterministic Falcon signatures. Technical report, Algorand, Inc., 2021. available at https://github.com/algorand/falcon/blob/main/falcon-det.pdf.

LP21b. David Lazar and Chris Peikert. Implementation of deterministic Falcon signatures. Technical report, Algorand, Inc., 2021. available at https://github.com/algorand/falcon.

LAZ19. Xingye Lu, Man Ho Au, and Zhenfei Zhang. Raptor: A practical lattice-based (linkable) ring signature. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19International Conference on Applied Cryptography and Network Security*, volume 11464 of *LNCS*, pages 110–130, Bogota, Colombia, June 5–7, 2019. Springer, Cham, Switzerland.

MW17. Daniele Micciancio and Michael Walter. Gaussian sampling over the integers: Efficient, generic, constant-time. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume

10402 of *LNCS*, pages 455–485, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Cham, Switzerland.

Mon08. David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, 2008.

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

PKKK24. Oleksandr Potii, Olena Kachko, Serhii Kandii, and Yevhenii Kaptol. Determining the effect of a floating point on the Falcon digital signature algorithm security. *Eastern-European Journal of Enterprise Technologies*, 1(9):52–59, 2024.

Pre17. Thomas Prest. Sharper bounds in lattice-based cryptography using the Rényi divergence. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 347–374, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland.

PFH+22. Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

ZLYW23. Shiduo Zhang, Xiuhan Lin, Yang Yu, and Weijia Wang. Improved power analysis attacks on falcon. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part IV*, volume 14007 of *LNCS*, pages 565–595, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland.

ZMS+24. Raymond K. Zhao, Sarah McCarthy, Ron Steinfeld, Amin Sakzad, and Máire O'Neill. Quantum-safe HIBE: does it cost a latte? *IEEE Trans. Inf. Forensics Secur.*, 19:2680–2695, 2024.

## A Properties of the Falcon Gram–Schmidt basis

Let $\mathbf{B}$ be the NTRU trapdoor basis of Falcon in bit reversed order, and $\tilde{\mathbf{B}}$ the associated Gram–Schmidt orthogonalized basis. The (row) vectors of $\mathbf{B}$ (resp. $\tilde{\mathbf{B}}$) are denoted by $\mathbf{b}_i$ (resp. $\mathbf{b}_i^*$), $0 \leq i \leq 2n-1$.

The following theorem collects a number of properties of the Gram–Schmidt basis, most of which are well-known, but others may be of independent interest.

**Theorem 1.** *The following properties hold.*

(a) *Let* $\omega\colon \mathbb{Z}^{2n} \to \mathbb{Z}^{2n}$ *be the isometry given by:*

$$\begin{aligned}&\omega(u_0, u_1, \ldots, u_{2k}, u_{2k+1}, \ldots, u_{2n-2}, u_{2n-1})\\ &= (-u_1, u_0, \ldots, -u_{2k+1}, u_{2k}, \ldots, -u_{2n-1}, u_{2n-2})\end{aligned}$$

*(i.e.,* $\omega$ *negates the second element in each pair of consecutive coefficients and swaps the pair). In the bit reversed order representation of the module lattice,* $\omega$ *corresponds to multiplication by* $x^{n/2} = \sqrt{-1}$ *on both ring elements, so that, e.g.,* $\mathbf{b}_0 = (g, -f)$ *is sent to* $\omega(\mathbf{b}_0) = (x^{n/2}g, -x^{n/2}f) = \mathbf{b}_1$.
*Then, for* $0 \leq i \leq n-1$*, we have:*

$$\omega(\mathbf{b}_{2i}) = \mathbf{b}_{2i+1} \quad \textit{and} \quad \omega(\mathbf{b}_{2i+1}) = -\mathbf{b}_{2i}.$$

*Moreover, the same relation holds for the Gram–Schmidt vectors:*

$$\omega(\mathbf{b}_{2i}^*) = \mathbf{b}_{2i+1}^* \quad \textit{and} \quad \omega(\mathbf{b}_{2i+1}^*) = -\mathbf{b}_{2i}^*.$$

*In particular,* $\|\mathbf{b}_{2i}^*\| = \|\mathbf{b}_{2i+1}^*\|$.

(b) *For all* $i$, $\|\mathbf{b}_i^*\| \cdot \|\mathbf{b}_{2n-1-i}^*\| = q$. *Moreover, we have:*

$$\mathbf{b}_{2n-1-i}^* = \frac{q}{\|\mathbf{b}_i^*\|^2}\mathbf{b}_i^*\mathbf{J}$$

*where* $\mathbf{J}$ *is the standard symplectic involution.*

(c) *For all $i$, $\frac{1}{1.17}\sqrt{q} \le \|\mathbf{b}_i^*\| \le 1.17\sqrt{q}$.*

(d) *For all $i$, $\mathbf{b}_i^*$ has rational coefficients, and in particular $\|\mathbf{b}_i^*\|^2$ is a rational number.*

(e) *For $0 \le k \le n$, let $m_k \coloneqq \prod_{i=0}^{2k-1} \|\mathbf{b}_i^*\|$ (in particular, $m_0 = 1$ by definition). Then $m_k$ is an integer for all $k$, and:*

$$m_k = \prod_{i=0}^{k-1} \|\mathbf{b}_{2i}^*\|^2 = \prod_{i=0}^{k-1} \|\mathbf{b}_{2i+1}^*\|^2.$$

(f) *For $0 \le k \le n$, $m_k \cdot \mathbf{b}_j^*$ has integer coefficients for $j = 2k$ and $j = 2k+1$.*

(g) *For $j = 2k$ and $j = 2k+1$, $m_{k+1} \cdot \mathbf{b}_j^* / \|\mathbf{b}_j^*\|^2$ is an integer vector. In particular, for any integer vector $\mathbf{c}$, the value*

$$c = \frac{\langle \mathbf{c}, \mathbf{b}_j^* \rangle}{\|\mathbf{b}_j^*\|^2}.$$

*satisfies that $m_{k+1} \cdot c$ is an integer.*

(h) *For $j = 2n-1-2k$ and $j = 2n-1-(2k+1)$, $qm_k \cdot \mathbf{b}_j^* / \|\mathbf{b}_j^*\|^2$ is an integer vector. In particular, for any integer vector $\mathbf{c}$, the value*

$$c = \frac{\langle \mathbf{c}, \mathbf{b}_j^* \rangle}{\|\mathbf{b}_j^*\|^2}.$$

*satisfies that $qm_k \cdot c$ is an integer.*

(i) *For $0 \le k \le n$, we have:*

$$m_{n-k} = q^{n-2k} m_k.$$

*Proof.* (a) For $0 \le i \le n-1$, let $\alpha_i$ denote the bit reversed representation of $i$. The vectors of $\mathbf{B}$ can be written as follows:

$$\mathbf{b}_i = (x^{\alpha_i} g, -x^{\alpha_i} f) \quad \text{and} \quad \mathbf{b}_{n+i} = (x^{\alpha_i} G, -x^{\alpha_i} F).$$

As a result, for $0 \le i < n/2$, we have:

$$\begin{aligned} \mathbf{b}_{2i+1} &= (x^{\alpha_{2i+1}} g, -x^{\alpha_{2i+1}} f) = (x^{\alpha_{2i}+n/2} g, -x^{\alpha_{2i}+n/2} f) \\ &= \omega(x^{\alpha_{2i}} g, -x^{\alpha_{2i}} f) = \omega(\mathbf{b}_{2i}) \end{aligned}$$

and similarly $\mathbf{b}_{n+2i+1} = \omega(\mathbf{b}_{n+2i})$. Moreover, we clearly have $\omega^2 = -1$. This all shows that for $0 \le i \le n-1$:

$$\omega(\mathbf{b}_{2i}) = \mathbf{b}_{2i+1} \quad \text{and} \quad \omega(\mathbf{b}_{2i+1}) = -\mathbf{b}_{2i}$$

as required.

Turning now to the Gram–Schmidt vectors, denote by $V^{(j)}$ the linear subspace spanned by $\mathbf{b}_k$ (or equivalently $\mathbf{b}_k^*$ for $k \le j$. By definition, $\mathbf{b}_{2i}^*$ is the unique vector in $V^{(2i+1)}$ orthogonal to $V^{(2i)}$. It follows that $\omega(\mathbf{b}_{2i}^*)$ is in $\omega\big(V^{(2i+1)}\big)$ and orthogonal to $\omega\big(V^{(2i)}\big)$. Now by the previous relations, we have:

$$\omega\big(V^{(2i)}\big) = V^{(2i)}$$

and

$$\omega\big(V^{(2i+1)}\big) = \omega\big(V^{(2i)} + \mathbb{R}\mathbf{b}_{2i}\big) = V^{(2i)} + \mathbb{R}\mathbf{b}_{2i+1} \subset V^{(2i+2)}.$$

Moreover, $\omega(\mathbf{b}_{2i}^*)$ is also orthogonal to $\mathbf{b}_{2i}^*$ since $\omega$ sends any vector to an orthogonal one. Hence, $\omega(\mathbf{b}_{2i}^*)$ is in $V^{(2i+2)}$ and orthogonal to $V^{(2i)} + \mathbb{R}\mathbf{b}_{2i}^* = V^{(2i+1)}$, so we have $\omega(\mathbf{b}_{2i}^*) = \mathbf{b}_{2i+1}^*$ as required. This concludes the proof of assertion (a).

(b) This is results from $q$-symplecticity of the NTRU basis [GHN06, Cor. 1].

(c) Falcon key generation imposes $\|\mathbf{b}_i^*\| \le 1.17\sqrt{q}$ for all $i$. Then, by relation (b), we also have $\|\mathbf{b}_i^*\| = q/\|\mathbf{b}_{2n-1-i}\| \ge \sqrt{q}/1.17$.

(d) This is a general fact about the rationality of the Gram–Schmidt orthogonalization process.

(e) First, the fact that:

$$m_k = \prod_{i=0}^{k-1} \|\mathbf{b}_{2i}^*\|^2 = \prod_{i=0}^{k-1} \|\mathbf{b}_{2i+1}^*\|^2$$

follows directly from (a). By (d), this shows moreoever that $m_k$ is rational for all $k$. Then, note that $m_k^2 = \prod_{i=0}^{2k-1} \|\mathbf{b}_i^*\|^2$ is the $2k$-th leading principal minor of the Gram matrix $\mathbf{G} = \mathbf{B}\mathbf{B}^*$, which is an integer matrix. Therefore, $m_k^2$ is an integer and also the square of a rational number. It must therefore be a perfect square, and hence $m_k$ is itself an integer.

(f) It is a general fact that the $i$-th Gram–Schmidt vector of any family of integer vectors becomes integral after multiplication by the $(i-1)$-st principal minor of the Gram matrix. A proof of that theorem is given, e.g., in [Gal12, Lemma 17.3.2]. In our case, this says that $m_k^2 \mathbf{b}_j^*$ has integer coefficients for $j = 2k$ and $j = 2k+1$.

To obtain our stronger claim that $m_k \mathbf{b}_j^*$ is already integral for those $j$, one can observe that assertion (a) says that $\mathbf{B}$ and $\tilde{\mathbf{B}}$ are in fact the Weil restrictions of $n \times n$ matrices over the ring of Gaussian integers $\mathbb{Z}[\sqrt{-1}]$ (resp. the quadratic field $\mathbb{Q}(\sqrt{-1})$), and apply the general fact, mutatis mutandis, to those matrices.

Concretely speaking, for $0 \leq i \leq n-1$, let $\mathbf{w}_i$ be the vector over $\mathbb{Z}[\sqrt{-1}]$ whose $j$-th coefficient is $b_{2i,2j} + b_{2i,2j+1}\sqrt{-1}$, and $\mathbf{w}_i^*$ the vector over $\mathbb{Q}(\sqrt{-1})$ whose $j$-th coefficient is $b^*_{2i,2j} + b^*_{2i,2j+1}\sqrt{-1}$. The corresponding matrices are denoted by $\mathbf{W}$ and $\tilde{\mathbf{W}}$. Then, for the standard Hermitian inner product on $\mathbb{Q}(\sqrt{-1})^n$ (antilinear on the right since we use row vectors), we claim that $\tilde{\mathbf{W}}$ is the Gram–Schmidt orthogonalization of $\mathbf{W}$. Indeed, for any $k, \ell$ we have:

$$\begin{aligned}
\langle \mathbf{w}_k^*, \mathbf{w}_\ell^* \rangle &= \sum_{j=0}^{n-1} \left(b^*_{2k,2j} + b^*_{2k,2j+1}\sqrt{-1}\right) \cdot \left(b^*_{2\ell,2j} - b^*_{2\ell,2j+1}\sqrt{-1}\right) \\
&= \sum_{j=0}^{n-1} \left(b^*_{2k,2j} b^*_{2\ell,2j} + b^*_{2k,2j+1} b^*_{2\ell,2j+1}\right) + \\
&\qquad \sum_{j=0}^{n-1} \left( - b^*_{2k,2j} b^*_{2\ell,2j+1} + b^*_{2k,2j+1} b^*_{2\ell,2j}\right)\sqrt{-1} \\
&= \langle \mathbf{b}_{2k}^*, \mathbf{b}_{2\ell}^* \rangle + \langle \mathbf{b}_{2k}^*, \omega(\mathbf{b}_{2\ell}^*) \rangle = \langle \mathbf{b}_{2k}^*, \mathbf{b}_{2\ell}^* \rangle + \langle \mathbf{b}_{2k}^*, \mathbf{b}_{2\ell+1}^* \rangle,
\end{aligned}$$

which shows in particular that $\langle \mathbf{w}_k^*, \mathbf{w}_\ell^* \rangle = 0$ for $\ell < k$ and $\langle \mathbf{w}_k^*, \mathbf{w}_k^* \rangle = \|\mathbf{b}_{2k}^*\|^2 = m_{k+1}/m_k$. Moreover, $\mathbf{w}_k^*$ is in the span of the vectors $\mathbf{w}_\ell$ for $\ell \leq k$. Indeed, write $\mathbf{b}_{2k}^*$ over the basis $\mathbf{B}$ as follows:

$$\mathbf{b}_{2k}^* = \mathbf{b}_{2k} - \sum_{i=0}^{k-1} a_{2k,2i}\mathbf{b}_{2i} + a_{2k,2i+1}\mathbf{b}_{2i+1}$$

From there and the fact that $\mathbf{b}_{2i+1} = \omega(\mathbf{b}_{2i})$, it follows that we have:

$$\mathbf{w}_k^* = \mathbf{w}_k - \sum_{i=0}^{k-1} v_{k,i}\mathbf{w}_i \quad \text{where} \quad v_{k,i} = a_{2k,2i} + a_{2k,2i+1}\sqrt{-1}.$$

This shows that $\tilde{\mathbf{W}}$ is indeed the Gram–Schmidt orthogonalization of $\mathbf{W}$.

To then obtain the desired integrality result, we take the inner product of the previous equation with all the $\mathbf{w}_\ell$ with $\ell < k$. Since $\mathbf{w}_k^*$ is orthogonal to all of these vectors, we get:

$$\langle \mathbf{w}_k, \mathbf{w}_\ell \rangle = \sum_{i=0}^{k-1} v_{k,i} \langle \mathbf{w}_i, \mathbf{w}_\ell \rangle$$

for all $\ell < k$. In matrix form, if we let $\mathbf{W}^{(k)}$ be the matrix consisting of the first $k$ rows of $\mathbf{W}$ (i.e., those of index $\ell < k$), this says:

$$\mathbf{w}_k\big(\mathbf{W}^{(k)}\big)^* = \mathbf{v}_k\mathbf{W}^{(k)}\big(\mathbf{W}^{(k)}\big)^* \quad \text{where} \quad \mathbf{v}_k = (v_{k,0},\dots,v_{k,k-1})$$

and the asterisk denotes the conjugate transpose as usual. Multiplying on both sides by the adjugate of $\mathbf{U}^{(k)} = \mathbf{W}^{(k)}\big(\mathbf{W}^{(k)}\big)^*$ shows that $\det(\mathbf{U}^{(k)})\cdot\mathbf{v}_k$ has coefficients in $\mathbb{Z}[\sqrt{-1}]$. Moreover, $\det(\mathbf{U}^{(k)}) = \prod_{\ell<k}\|\mathbf{w}_\ell^*\|^2 = m_k$. Therefore, the $m_k v_{k,i}$ are elements of $\mathbb{Z}[\sqrt{-1}]$ and hence $m_k a_{2k,i}$ is an integer for all $i < 2k$. This shows that $m_k\mathbf{b}_{2k}^*$ is an integer linear combination of the integer vectors $\mathbf{b}_i$, $i < 2k$, and the same is true for $m_k\mathbf{b}_{2k+1}^* = \omega(m_k\mathbf{b}_{2k}^*)$. This concludes the proof of assertion (f).

(g) To obtain the assertion for $j = 2k$, it suffices to write:

$$m_{k+1}\cdot\frac{\mathbf{b}_{2k}^*}{\|\mathbf{b}_{2k}^*\|^2} = m_k\mathbf{b}_{2k}^*$$

which is an integer vector by assertion (f). The result on $c$ follows immediately, and the case $j = 2k+1$ is treated similarly.

(h) Suppose $j = 2n-1-2k$. Applying assertion (b), we have

$$\mathbf{b}_j^* = \frac{q}{\|\mathbf{b}_{2k}^*\|}\cdot\mathbf{b}_{2k}^*\mathbf{J} \quad \text{and} \quad \|\mathbf{b}_j^*\|^2 = \frac{q^2}{\|\mathbf{b}_{2k}^*\|^2}.$$

Therefore:

$$qm_k\cdot\frac{\mathbf{b}_j^*}{\|\mathbf{b}_j^*\|^2} = qm_k\cdot\frac{q/\|\mathbf{b}_{2k}^*\|^2\cdot\mathbf{b}_{2k}^*\mathbf{J}}{q^2/\|\mathbf{b}_{2k}^*\|^2} = qm_k\cdot\frac{1}{q}\mathbf{b}_{2k}^*\mathbf{J} = m_k\mathbf{b}_{2k}^*\mathbf{J}$$

which is again an integer vector (as $\mathbf{J}$ obviously sends integer vectors to integer vectors). The result on $c$ again follows immediately, and the case $j = 2n-1-(2k+1)$ is treated in the same way.

(i) Applying assertions (b) and (e), we have:

$$\begin{aligned} m_{n-k} &= m_n\cdot\prod_{j=0}^{k-1}\frac{1}{\|\mathbf{b}_{2n-1-(2j+1)}^*\|^2} \\ &= \det(\mathbf{B})\cdot\prod_{j=0}^{k-1}\frac{\|\mathbf{b}_{2j+1}^*\|^2}{q^2} = q^n\cdot\frac{m_k}{q^{2k}} = q^{n-2k}m_k \end{aligned}$$

as required.