Black-box validation of Falcon key generation under numerical instability

Maxime Bros[1,2], Christopher Celi[3], Pierre Ciadoux[1] and Ray Perlner[3]

[1] Associate, National Institute of Standards and Technology (NIST), Gaithersburg, USA
{maxime.bros,pierre.ciadoux}@nist.gov
[2] Contractor, Izum Inc., Potomac, Maryland, USA
[3] National Institute of Standards and Technology (NIST), Gaithersburg, USA
{christopher.celi,ray.perlner}@nist.gov

Abstract. FALCON is a lattice-based digital signature scheme offering excellent performance and key sizes, and it has been selected for standardization by the National Institute of Standards and Technology (NIST) as part of their post-quantum standardization project. However, the use of floating-point and/or fixed-point arithmetic in FALCON presents unique challenges. One such challenge is the lack of reproducibility, which can arise due to the inexact representation of fractional numbers. Traditionally, implementations are validated for correctness using Known Answer Tests (KATs), but this approach requires exact reproducibility.

We propose a novel alternate procedure for validating the correctness of FALCON's key generation. Our procedure never rejects correct implementations of FALCON that vary due to the numerical instability of floating and fixed-point arithmetic. It is still strict enough to guarantee that differences in generated keys due to these variations will not create security problems, although like all black-box testing methods, it does not rule out security problems due to other implementation characteristics, such as side channels. Combined with black-box conformance testing on signing and verification, this provides a path for validating the correctness of Falcon implementations on defined platforms. We further study FALCON's keys by defining and computing perfect keys that are generated using infinite precision. Last, we estimate the total number of valid keys that could be generated from a single seed. Our work is based on extensive experiments for which the code is available.

Keywords: Falcon Digital Signature · NIST Post-Quantum Standards · Cryptographic Validation · NIST FIPS 206

1 Introduction

Context. Public-key cryptography is a vital tool used to secure digital communications. Until recently, all deployed public-key cryptosystems relied on two distinct but connected problems: integer factorization and discrete logarithm. However, in 1994, Shor described a quantum algorithm [Sho99] that would enable one to break all factoring and discrete-log-based cryptosystems with a sufficiently powerful and stable quantum computer. Due to the many developments in building quantum computers, the National Institute of Standards and Technology (NIST) launched the NIST Post-Quantum Cryptography (PQC) Standardization Process to identify cryptosystems that would resist classical and quantum computers [NIS16]. Such cryptosystems are said to be quantum-resistant or post-quantum. The goal of this process is to select a set of post-quantum key-encapsulations mechanisms (KEMs) and digital signature schemes for standardization. Post-quantum cryptography includes five main categories: lattice-based, code-based, multivariate, symmetric-based, and

isogeny-based cryptography. In July 2022 and March 2025, NIST selected five algorithms as their first post-quantum standards [AAC+22, ABC+25], which are summarized in Table 1.

Table 1: The first set of NIST PQC standards (referred to as FIPS, which stands for "Federal Information Processing Standards"); KEM stands for "Key Encapsulation Mechanisms" and DSA for "Digital Signature Algorithms." A star * indicates that the final standard has not yet been released.

Candidate	Type	Standard	Category
CRYSTALS-Kyber	KEM	ML-KEM (NIST FIPS 203)	Lattice
CRYSTALS-Dilithium	Signature	ML-DSA (NIST FIPS 204)	Lattice
SPHINCS+	Signature	SLH-DSA (NIST FIPS 205)	Hash
Falcon	Signature	FN-DSA (NIST FIPS 206)*	Lattice
HQC	KEM	HQC-KEM (NIST FIPS 207)*	Code

Overview of FN-DSA — Falcon and NIST FIPS 206. Falcon [FHK+20] will be the basis of NIST's planned FN-DSA standard. It is a hash-and-sign signature scheme that applies the Gentry, Peikert, Vaikuntanathan (GPV) framework [GPV08] to NTRU lattices [HPS98, HHP+03] using a power-of-2 cyclotomic ring structure. That is, $\mathbb{Z}[X]/(X^n + 1)$, where n is a power of 2. This cyclotomic ring structure induces a *tower-of-rings* structure that enables a number of performance optimizations, including the recursive *NTRUSolve* algorithm used as a subroutine in Falcon's key generation, the recursive *fast-Fourier sampler* used in Falcon's signing algorithm, and efficient Fast Fourier Transform (FFT) and Number Theoretic Transform (NTT) implementations of polynomial multiplication.

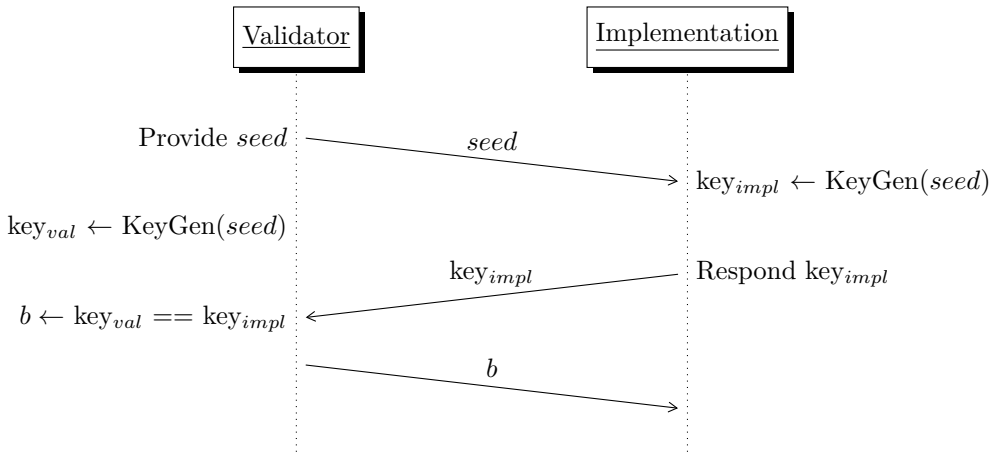

Figure 1: A standard method of performing black-box testing on a key-generation algorithm. The subscripts "*val*" and "*impl*" stand for validator and implementation, respectively.

Cryptographic algorithm validation. Ensuring correctness in a cryptographic algorithm implementation is an essential practice in development. There are many ways to validate that an implementation is correct (i.e., that it adheres to the published cryptographic standard). Test vectors for cryptographic algorithms may be published in the standard, such as part of a Request For Comments (RFC), or separately through a trusted party (often called a validator), such as Project Wycheproof[1] originally started by Google or the

[1]More information on Project Wycheproof can be found here: https://github.com/C2SP/wycheproof.

Cryptographic Algorithm Validation Program (CAVP)[2] from NIST. Test vectors, such as statically defined Known Answer Tests (KATs), in this form are considered black-box tests, where no knowledge of the implementation is required to run the test data. Black-box testing can obtain high assurance of the correctness of many existing cryptographic algorithms. To obtain an ultimate assurance, white-box testing may be required, where a tester has full access and control over the implementation. White-box testing may involve additional tools and analyzers to formally prove the correctness of an implementation relative to a cryptographic standard. Compared to black-box testing, this involved level of work for white-box testing causes it to be significantly more expensive.

Black-box testing relies on the numerical stability of the cryptographic algorithm. With specific inputs, the tests evaluate whether the implementation can arrive at the correct output, as described in Figure 1. In numerically unstable algorithms, the effectiveness of these tests collapses; over a series of black-box tests, a number of different results could be obtained even if the implementation is correct.

Any testing also relies on the execution platform for the implementation. A cryptographic implementation validated on one platform cannot be assumed to be validated on another without additional analysis.

Challenges raised by Falcon's use of fractional numbers. The use of floating-point arithmetic in Falcon has been a significant concern as discussed in several NIST PQC Forum posts [aut25, aut24]. One reason for this concern is that it may lead to a lack of reproducibility; from a given seed, different key pairs or signatures (for a fixed message and private key) could be generated, depending on the specific platform and implementation. This also makes it very challenging to validate Falcon implementations.

In July 2024, NIST announced that NIST FIPS 206 – FN-DSA would allow alternate representations for fractional numbers [Bro24]. Implementers will be able to use floating points, fixed points, or emulated floating points for key generation and either floating points or emulated floating points for the signature (Falcon's verification only requires integer arithmetic).

While there are some benefits to NIST's announced decision to allow more flexibility on the choice of the data structure for fractional numbers, this flexibility does not eliminate the numerical stability issues associated with the use of fractional numbers. In fact, as described in Section 3 below, there are many reasons why two distinct and valid implementations of Falcon might give different results, even on the same platform. Different platforms introduce an additional layer of complexity.

These numerical stability concerns have notably different implications for the three-component algorithms of the Falcon signature scheme:

- Key generation — For a variety of reasons, the keys generated from an input seed by one system may not match the keys generated by the same input seed on another system. Fractional number arithmetic is required for FFT operations used in Falcon's key generation. While FFT is used to speed up computations, one could theoretically make the computations without FFT but still need to represent polynomials with fractional coefficients. Implementations may use either floating or fixed-point representations for these operations, which might cause rounding differences due to the lack of precision. Even if the same representation is used, rounding differences may result from changing the order of operations. Implementations are also allowed to terminate loops after a fixed number of iterations to avoid timing side-channel attacks. Different implementations may use different limits to affect the runtime of the algorithm.

[2]More information on the CAVP at NIST can be found here: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program.

While it is desirable to allow implementations flexibility to maximize performance on specific platforms, key generation must not deviate from the standard in ways that could lower security. In particular, it is crucial to ensure that the method used to generate random values is cryptographically strong, that the private key is consistent with the public key, and that the private key can be used by the signing algorithm to securely generate signatures.

- Signing — While FALCON's signing algorithm requires floating-point arithmetic, a different approach will be required to validate the signing algorithm than the approach we apply to key generation. A standard variant of the FALCON signing algorithm will likely need to be specified in enough detail so that signatures from known inputs and randomness values can be exactly reproduced. This is because even a very small deviation from the ideal distribution of signatures can result in leakage of the private key. As such, it is very challenging to confirm that the distribution of signatures is sufficiently close to the ideal distribution unless the signatures can be matched exactly to known outputs of a signing algorithm known to have the correct behavior. NIST plans to publish such a standard variant of the signing algorithm in FIPS 206 [Per25]. Since this standard variant will be numerically stable, known answer tests can provide a high level of assurance that implementations are correct.

- Verification — Because the verification algorithm for FALCON can be implemented with only integer arithmetic, it does not have any numerical stability issues. Again, known answer tests can provide a high level of assurance that implementations are correct.

1.1 Our Contributions

In this paper, we evaluate and improve the ability to perform black-box testing on FALCON's key generation. Combined with black-box testing — via known answer tests — on signing and verification, this ensures that all algorithms in a FALCON implementation can be validated with a high level of assurance of correctness.

More precisely, we present the different data structures for representing fractional numbers (native and emulated floating points as well as fixed points), and we give detailed examples to illustrate how fractional number representations can create numerical instability. We also identify the ways, which we call *scenarios*, that this numerical instability may or may not manifest in the context of FALCON, and we analyze the frequency with which each scenario occurred in extensive experiments that we performed on different FALCON implementations.

We further propose a novel black-box validation procedure that can accommodate the aforementioned numerical instability, unlike the usual techniques for validation. Our proposed validation method always accepts correct implementations while being strict enough to reject implementations whose input-output behavior differs from the reference implementation in a way that could create security problems. In each scenario where a different key can arise from the seed, we explain why the variant key is equally secure to the expected key. Other important properties of cryptographic implementations such as constant time operations or side channel resistance require more advanced testing techniques. Such properties are out of scope for our procedure and should be performed separately.

Finally, after answering the above, we answer two additional questions regarding FALCON's key generation. First, between the two main representations of fractional numbers (floating and fixed point), which variation is "correct," if any, compared to what an infinite precision algorithm would produce? This question is answered using the concept of *perfect keys*. Second, since the same seed can produce distinct and valid keys, how many

such keys are there? We found that this question could not be answered using classic combinatorial techniques. We developed heuristic techniques to give an approximate answer and experimentally demonstrated the convergence of the resulting estimate, finding that the number of possible keys is surprisingly large. We further provide a visual illustration of our estimation technique, which makes it straightforward to see how the estimate can be adapted to count keys that pass our validation method.

1.2 Organization of the Paper

In Section 2, we present different ways of representing fractional numbers. We then provide some mathematical background and give a brief overview of the Falcon signature scheme with a focus on the key-generation algorithm. In Section 3, we define *numerical instability* and give examples of potential causes.

Section 4 is the core of the article. In this section, we present our extensive experiments and the different scenarios that render Falcon's key generation numerically unstable. We use these scenarios to derive a novel validation procedure.

Last, Section 5 answers the questions that arose while working on this article. The first one is essential since it discusses the security implications of the numerical instability. The last two parts are about *perfect keys* and estimating the total number of valid Falcon keys that one could get from a given seed.

2 Preliminaries

In this section, we give the preliminary background needed for the article. We start by describing the three fractional number representations permitted in Falcon's key generation, and we present the relevant parts of Falcon's algorithms needed for this article.

2.1 Fractional Number Representations

The terms "real numbers" and "fractional numbers" are often used interchangeably. For our purposes, we make the distinction between real numbers in the field $\mathbb{R}$ and fractional numbers that approximate them. More specifically, we use the term "fractional numbers" to refer to a subset of rational numbers $\mathbb{Q}$, where the information can be stored in a data type of a defined length. Such data types have a minimum and maximum value and are only able to represent some of the rational numbers in that interval. These representations have a defined precision due to the underlying data type used. Infinite precision is possible but will drastically affect performance. No rounding deviation will occur when using infinite precision fractional numbers.

2.1.1 Fixed Points

Fixed-point representation is an intuitive way of representing fractional numbers. It uses a defined number of bits to represent an integer and a fractional part. The integer part can be signed or unsigned. Fixed points can be used to emulate fractional values when no floating-point unit (FPU) is present or to enforce the same arithmetic behavior on different systems.

In this article, we refer to a specific fixed point: a signed 64-bit fixed point using 32 integer bits and 32 fractional bits. The most significant bit represents -2^{31} and is included in the integer bits. As stated in the PQC Forum [Bro24], fixed points will be allowed in Falcon key generation.

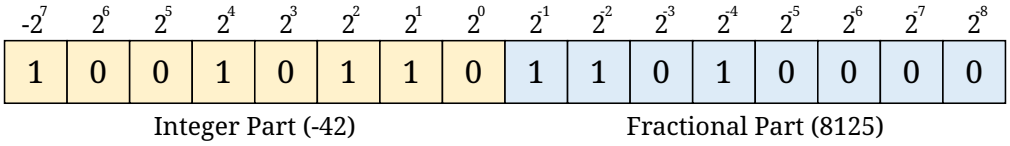

Figure 2: Representation of -42.8125 (in base 10) with a 16-bit signed fixed point

2.1.2 Floating Points

Floating-point representations contain a sign, an exponent, and a mantissa. Floating points are capable of storing very large or very small numbers because of the exponent. While there are multiple types of floating-point representations, we will exclusively use the 64-bit "binary64" floating-point standardized in IEEE754 [IEE19], commonly known as "double precision," unless noted otherwise in Section 3. This is the type defined in the FALCON specification [FHK+20] and the only floating-point representation allowed in FALCON. To perform fast arithmetic on floating points, hardware FPUs are used on most recent processors.

2.1.3 Emulated Floating Points

Emulated floating points can also be used to represent fractional numbers. They behave exactly the same as the "binary64" floating point, except for infinites, not-a-numbers (NaNs), or subnormals, which are not needed in FALCON. Emulated floating points do not require an FPU, though they are about 20 times slower in FALCON key generation compared to floating points [Por19].

2.2 Description of FN-DSA

2.2.1 Mathematical Preliminaries

This section gives some mathematical preliminaries that are useful for understanding the design of FALCON and the pseudocode reproduced in this work.

FALCON bases its security on the presumed difficulty of problems related to finding short vectors in lattices. A d-dimensional lattice is a discrete additive subgroup of $\mathbb{R}^d$ that is isomorphic to $\mathbb{Z}^d$. A lattice may also be described using a basis matrix $\mathbf{B} \in \mathbb{R}^d$, in which case the lattice $\Lambda(\mathbf{B})$ is defined to be the set of points that can be expressed as $\mathbf{vB}$ with $\mathbf{v} \in \mathbb{Z}^d$. FALCON involves approximate calculations in the number fields $Q(k) = \mathbb{Q}[X]/(X^k+1)$ and exact computations over their rings of integers $R(k) = \mathbb{Z}[X]/(X^k+1)$, where $1 \leq k \leq n$ is a power of 2 and n is a parameter that is either 512 or 1024, depending on the security level. FALCON additionally uses arithmetic in the ring $R_q(n) = \mathbb{Z}_q[X]/(X^k+1)$, where $q = 12289$. While we have defined $Q(k)$ and $R(k)$ as rings of polynomials, their elements may also be given an isomorphic representation as structured matrices in $\mathbb{Q}^{k\times k}$ and $\mathbb{Z}^{k\times k}$, respectively, with the isomorphism given by:

$$p \to \begin{bmatrix} p(X) \\ Xp(X) \\ \vdots \\ X^{k-1}p(X) \end{bmatrix} \to \begin{bmatrix} p_0 & p_1 & \cdots & p_{k-1} \\ -p_{k-1} & p_0 & \cdots & p_{k-2} \\ \vdots & \vdots & \ddots & \vdots \\ -p_1 & -p_2 & \cdots & p_0 \end{bmatrix} \triangleq M(p). \tag{1}$$

Finally, we note that polynomial multiplication in $Q(k)$ (and by inclusion, multiplication in $R(k) \subset Q(k)$) can be performed efficiently using FFT. For a power-of-2 $k \geq 2$, the FFT as implemented by FALCON embeds the field $Q(k) = \mathbb{Q}[X]/(X^k+1)$ into the direct product ring $\mathbb{C}^{k/2}$. In particular, let $\zeta = \exp(i\pi/k)$, and let $\zeta_j = \zeta^{2\mathrm{brv}(j)+1}$ for $0 \leq j \leq k/2-1$,

where brv modifies an integer (represented as a binary value of length $\log_2(k) - 1$) by reversing the order of its bits. If $p \in Q(k)$, then

$$FFT(p) = (p(\zeta_0), p(\zeta_1), p(\zeta_2), \ldots, p(\zeta_{k/2-1})) \in \mathbb{C}^{k/2}.$$

Since both FFT and its inverse FFT^{-1} can be efficiently computed, polynomial arithmetic over $Q(k)$ is most efficiently done by first computing the FFT, performing arithmetic component-wise in $\mathbb{C}^{k/2}$, and then performing FFT^{-1} on the result.

2.2.2 High-Level Description of Falcon

Falcon's private key (following NTRU [HPS98, HHP+03]) consists of four polynomials (f, g, F, G) with small Euclidean norms that live in the cyclotomic ring $R(n)$ and fulfill Equation (2), which is called the *NTRU equation*:

$$fG - gF = q \mod (X^N + 1). \tag{2}$$

The polynomials $(f, g) \in R(n)$ can be viewed as a lift to $R(n)$ of polynomials $(f, g) \in R_q(n)$, where $f \in R_q(n)$ is invertible. The associated public key is given by $h := f^{-1}g \in R_q(n)$. This polynomial is believed to be indistinguishable from a random polynomial in $R_q(n)$. As such, any lift of h to $R(n)$ is expected to have a relatively large Euclidean norm (i.e., coefficients of size similar to q).

For the remainder of Section 2.2.2, h will be treated as an element of $R(n)$. For concreteness, the reader may assume that $h \in R_q(n)$ is lifted to $R(n)$ so that its coefficients lie in the range $[-q/2+1, q/2]$, although this choice does not have any functional implications for the cryptosystem. The cyclotomic ring of polynomials in R is isomorphic to the ring of skew-circulant $n \times n$ matrices over $\mathbb{Z}$ with the isomorphism given by Equation 1.

Using this matrix representation of $R(n)$, the Falcon public key $\mathbf{h}$ defines a basis for the NTRU lattice $\Lambda(\mathbf{B}_{\text{pub}})$, which is spanned by the rows of the following block skew-circulant matrix (of size $2n \times 2n$), spanned by the rows of the secret matrix:

$$\mathbf{B}_{\text{pub}} = \begin{bmatrix} h & -I \\ qI & 0 \end{bmatrix}. \tag{3}$$

The Falcon private key gives an alternate basis for the same lattice $\Lambda(\mathbf{B}) = \Lambda(\mathbf{B}_{\text{pub}})$:

$$\mathbf{B} = \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}. \tag{4}$$

To sign a message M, the signer first hashes M to an element of $R_q(n)$, which is notated in the remainder of Section 2.2.2 as $H(M)$. This is used to define $\mathbf{c} = (H(M), 0)$. One then finds a vector $\mathbf{v}$ in $\Lambda(\mathbf{B})$ that is close to $\mathbf{c}$, which is easy knowing the trapdoor (i.e., the secret basis $\mathbf{B}$ of small norm). Finally, the signature consists of the vector $\mathbf{s} = \mathbf{c} - \mathbf{v}$, which has a small norm by construction.

In order for the above construction to be secure, $\mathbf{v}$ must be chosen by the signer from a distribution that does not leak information about the signer's private key. Falcon follows the GPV framework [GPV08] in that the distribution in question is designed to be computationally indistinguishable from a spherically symmetric discrete Gaussian distribution centered on $\mathbf{c}$. The sampling of $\mathbf{v}$ from this distribution uses the Fast Fourier Sampler to achieve good performance during signing [FHK+20].

2.2.3 Simplified Key Generation Algorithm

As this work focuses on the key generation of Falcon, we give a simplified version in Algorithm 1 based on [FHK+20]. Due to page limit constraints, we omit the code for several auxiliary algorithms, such as NTRUSolve, Reduce, and GramSchmidtNorm. These can be found in the Falcon specification [FHK+20]. The main steps of Algorithm 1 are:

- Deterministically generate (f, g) from the input seed.
- Compute the Gram-Schmidt norm of the basis $\mathbf{B}$, which can be computed directly from f and g, prior to computing (F, G). If the norm is too large, (f, g) will need to be resampled.
- Compute the public key h. This requires f to be invertible. If this is not the case, (f, g) will need to be resampled.
- Solve for (F, G) using NTRUSolve and Reduce. If the infinity norm of (F, G) is too large, (f, g) will need to be resampled.

Algorithm 1 SimplifiedKeygen($seed$)

Require: A 256-bit $seed$
Ensure: Key pair (pk, sk)

while `True` **do**
  $(f, g) \leftarrow$ Sample small $(f, g) \in \mathbb{Z}[X]/(X^n + 1)$
  if GramSchmidtNorm$(f, g) > 0.9999 \cdot 1.17 \cdot \sqrt{q}$ **then** ▷ 0.9999 as in [Per25]
    `continue`
  $h \leftarrow f^{-1} \cdot g \mod q$
  if $h == \perp$ **then**
    `continue`
  $(F, G) \leftarrow$ NTRUSolve(f, g) ▷ This calls Reduce internally
  if $(F, G) == \perp$ **then**
    `continue`
  if $||(F, G)||_\infty > 127$ **then**
    `continue`
  `break`
return $(pk = h, sk = (f, g, F, G))$

3 Numerical Instability

Depending on the fractional number representation used, even simple mathematical operations can produce varying results because not every fraction can be represented with a specific floating-point representation. We identify this as *numerical instability*. The C code in Example 1 shows how values can deviate because 0.1 cannot be exactly represented with single-precision floating points. This same concept extends to double-precision floating points and fixed points.

```c
int main(){
    float a = 2;
    for (int i = 0; i < 6; i++)
        a += 0.1;
    printf("%f\n", a); // output: 2.599999
}
```

Example 1: C Inexact Computation

Although inexact values can be computed as shown above, this is not an issue when comparing values between implementations because using the same fractional number representation would result in equivalent inaccuracies. It should only be observed when comparing a fixed-point implementation to a floating-point implementation. However, other implementation-dependent variations can occur because of the order of operations. This can be seen with as few as two addition operations in C.

```c
int main(){
    float a = 1.0000001f, b = 2.0000002f, c = 3.0000003f;
    float abc = a + b + c;
    float bca = b + c + a;
    printf("%.7f, %.7f\n", abc, bca); // output: 6.0000010, 6.0000005
}
```

Example 2: C Approximate Associativity

Compiler-induced variations also exist, meaning that even with the same hardware and implementation, a different compiler or different compiler flags can affect numerical stability. For example, fused instructions are CPU instructions that combine two (or more) operations into a single faster instruction. While it can save a few CPU cycles, it does so by removing intermediate rounding between two floating-point operations. Different results can be an issue for validation. We can illustrate the issue with a simple example, checking that $(a * b) - (a * b) = 0$ in C with values large enough that rounding occurs.

```c
int main(){
    double a = 33370374411415;
    double b = -185547331688695;

    double ab = a * b;        // FMUL (+rounding)
    double abMab = ab - ab;   // FSUB (+rounding)
    printf("2 lines: %f\n", abMab); // ab - ab == 0

    // FMUL (+rounding)
    // FNMSUB (multiply and subtract +rounding)
    abMab = (a * b) - (a * b);
    printf("1 line: %f\n", abMab); // ab - ab != 0
}
// output: 2 lines: 0.000000
// output: 1 line: 271906978383.000000
```

Example 3: C Fused Instructions

As seen in Example 3, separating the computation into two lines ensures that the multiplication is done separately from the subtraction, giving a correct answer. However, doing the computation in one line allows the compiler to change assembly instructions from two multiplications and a subtraction to using the fused instruction for a multiplication and subtraction, thus removing an intermediate rounding step on only one side. Although the intermediate value is closer to the real result, it creates a significant discrepancy. On recent compilers, fused instructions are sometimes enabled by default and can be disabled with flags such as "-ffp-contract=off." If Example 3 is compiled with the flag set to "off," both computations output a correct value of 0.

The above differences may not seem like much, but affecting the numerical stability will lead to differences in how cryptographic values are computed. A value computed by one system as 2.49999 but 2.50001 on another will lead to a rounding difference: $\text{Round}(2.49999) = 2$ but $\text{Round}(2.50001) = 3$. The implementations might then generate different values or take substantially different paths in the affected cryptographic algorithm.

4 Validating Falcon KeyGen Implementations

In this section, we describe how to validate the private key of Falcon, which consists of the four polynomials (f, g, F, G). The key-generation algorithm also produces the public key $h = (f^{-1} \cdot g) \mod q$, but this is derived from the private key through numerically stable operations. As such, the validation is performed on the private key first, and if the private key is found to be valid, the validator derives the public key from it and checks for equality. Below, polynomials (f, g, F, G) are from the validator, while polynomials (f', g', F', G') are from the implementation under test.

4.1 Experiments Identifying the Different Scenarios

As described in Section 2.2, several parts of Falcon's key-generation algorithm rely on the use of fractional numbers. Since representing fractional numbers induces numerical instability, as discussed in Section 3, this means that Falcon's key generation is numerically instable. In other words, two distinct implementations (or even the same implementation on different architectures) could lead to different keys when given the same seed. Note that the code for the experiments in this article is publicly available[3].

First, we experimentally tested the same reference implementation using native floating points with different hardware to find discrepancies. Keys were generated using the same seeds on four different platforms [4] :

- M2 processor from a laptop (ARMv8)
- BCM2837 processor from a Raspberry Pi 3B (ARMv8)
- Xeon E5-2650 processor from a server (X86_64)
- Xeon E5-2630 processor from a server (X86_64).

On each machine, 100,000 keys for both security levels were generated and then compared to each other. No difference was found in any key or signature. All platforms tested follow IEEE754, as required for Falcon, and the reference code[5] enforces the rounding rule in software. Moreover, compilation was done with recent compilers across machines with the same flags to reduce the risk of a different order of operations. This means that even if the numerical instability of Falcon's key generation could theoretically create discrepancies from a same implementation, they are extremely rare and thus not relevant for validation purposes. This is the reason why we focus hereafter on validating different implementations: floating point and fixed point.

In order to propose a validation procedure for Falcon, we need to understand precisely why some keys can be different and thoroughly study the kinds of differences that can happen between distinct implementations. This section is based on our extensive experiments as well as some theory explaining what we observed for the different scenarios.

Recall that Falcon's key generation proceeds by finding four polynomials of small Euclidean norm (f, g, F, G). This is done by picking small (f, g) and then solving for polynomials (F, G), fulfilling the NTRU equation given by Equation 2 (see more details in Section 2.2.2).

There are two core reasons why distinct implementations of Falcon could lead to distinct keys. First, for a given (f, g), there exist multiple (F, G)s that satisfy the requirements for a valid private key (see Section 5.3). Second, the use of fractional numbers can create discrepancies (see Section 3).

[3] https://github.com/usnistgov/falcon-validation

[4] Certain commercial equipment, instruments, or materials (or suppliers, or software, ...) are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

[5] Reference code available at https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Falcon-Round3.zip, using native floating points.

Table 2: Numbers and kinds of differences obtained experimentally over 10 million FALCON private keys (level 1) generated with both floating and fixed-point implementations.

Scenario	Reason	# instances	Percentage
Identical		9350051	93.50%
Different (f, g, F, G)'s	Gram-Schmidt norm fails	525787	5.258%
	NTRUSolve/Reduce failed	119657	1.197%
Only different (F, G)'s	Reduce differs	4505	0.045%

We generated 10 million FALCON keys for both security levels with each of the two implementations, one using floating points and one using fixed points. The statistics from these keys are summarized in Tables 2 and 3.

Table 3: Numbers and kinds of differences obtained experimentally over 10 million FALCON private keys (level 5) generated with both floating and fixed-point implementations.

Scenario	Reason	# instances	Percentage
Identical		7819995	78.20%
Different (f, g, F, G)'s	Gram-Schmidt norm fails	1960007	19.60%
	NTRUSolve/Reduce failed	211021	2.110%
Only different (F, G)'s	Reduce differs	8977	0.090%

Our experiments (summarized in Tables 2 and 3) helped identify the three scenarios in which two distinct implementations could disagree on a key when given the same seed. Upon further inspection of the key-generation algorithm, we concluded that there are no other possible scenarios.

Section 4.2 describes these three scenarios in more detail and suggests ways of validating an implementation despite their occurrence. Section 4.3 summarizes our proposed validation procedure. For the sake of simplicity, we will focus on level 1, but all results can be extended to level 5.

4.2 Details About the Three Scenarios and Suggested Validation Procedure

The three scenarios are as follows:

- *Scenario 1: Equality.* The private keys are identical between the two implementations. This is by far the most common case, representing 93.5% of generated keys.
- *Scenario 2: Completely Different Keys.* When $(f, g) \neq (f', g')$, it naturally leads to completely different (F, G) pairs. This is still fairly common, representing 5.3% of generated keys.
- *Scenario 3: Partially Different Keys (Only (F, G) Different).* Two distinct pairs (F, G) and (F', G') may also be derived from the same (f, g) pair. This is the rarest case, representing only 0.045% of keys.

4.2.1 Scenario 1: Equality

While Equality is a trivial case for validation, it happens often. Validating keys in this scenario requires checking that the two keys (coming from the validator and the implementation under test) are equal. This is naturally the first check to be performed.

4.2.2 Scenario 2: Completely Different Keys

In the event that the keys are completely different (i.e., $(f, g, F, G) \neq (f', g', F', G')$), a trivial check must be performed to ensure that (f', g', F', G') could possibly be a valid key. The infinity norm of F' and G' must be found to meet $||(F', G')||_\infty \leq 127$. If this is not found to be true, the key can be rejected. Otherwise, the key inequality indicates that a requirement during key generation has not been met on only one side, leading to a complete resampling of either pair (f, g) or (f', g'). Multiple requirements can fail in key generation, but only two conditions can differ due to numerical instability and implementation-specific optimizations.

Gram-Schmidt norm differences. The SimplifiedKeygen algorithm checks that the Gram-Schmidt norm of (f, g) fulfills the condition $\text{GramSchmidtNorm}(f, g) \leq 0.9999 \cdot 1.17 \cdot \sqrt{q}$. The computation of the Gram-Schmidt norm uses the FFT for efficiency, so the order of operations and the use of floating or fixed points will affect the result. This can sometimes change whether a value is below or above the bound, thus changing the execution path. The outcome is that some percentage of (f, g) pairs will be resampled in one implementation but not in the other when the norm is very close to the bound. In the case of our fixed and floating-point implementations, this happened for roughly 5% and 20% of (f, g) pairs for security levels 1 and 5, respectively (see Tables 2 and 3). When validating a different pair (f', g'), it is possible that the computation of the Gram-Schmidt norm results in a value below the bound for the implementer, when it was slightly above for the validator, due to differences in fractional values. Our validation technique allows the validation of Gram-Schmidt norms that are below $0.99995 \cdot 1.17 \cdot \sqrt{q}$, while still being a safe value.

Infinity norm failure due to insufficient reductions. Solving for (F, G) from (f, g) is done using the NTRUSolve procedure that recursively reduces (F, G) with Reduce. The termination condition of Reduce is that the factor k equals 0. It is possible for two implementations to terminate while their second to last value of $k > 0$ was different. This can lead to different (F, G) pairs, where one has a small enough infinity norm while the other does not. More precisely, once a candidate (F, G) is found by NTRUSolve, it will be discarded if $||(F, G)||_\infty > 127$. If only one implementation's (F, G) fails this test, it triggers that system to resample a new pair (f', g'), which also yields completely different keys.

Implementation-specific optimizations. Keys can also be different due to implementation-specific optimizations. These optimizations can be performed in ways that do not create security issues. One such optimization fixes the number of iterations in the Reduce loop to make it a constant-time operation. It assumes that enough reductions are performed with some margin because additional reductions do not affect values in most cases. While this method avoids leaking the number of reductions needed, it creates a trade-off between losing cycles with too many iterations or under-reducing values if not enough iterations are performed. For computational speed, a low iteration count may be selected, leading the implementation to reject some keys that would otherwise have been correct.

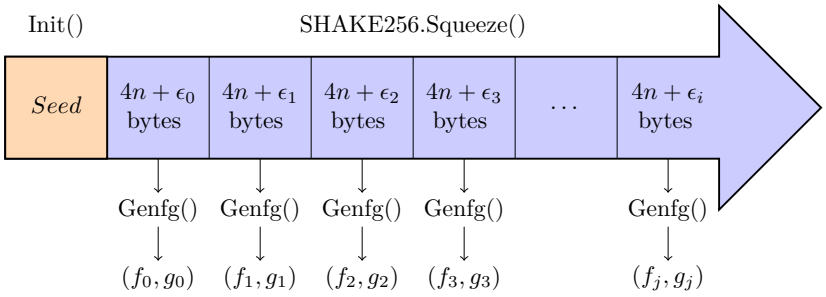

Figure 3: Stable generation of (f, g)

Validating Scenario 2 (completely different keys). If a completely different key is generated, validation will need to be performed in steps. First, the validator checks the NTRU equation, Gram-Schmidt norm, and infinity norm of (f', g', F', G'). This must be done with a small tolerance to account for rounding errors in the Gram-Schmidt norm calculation by the implementation under test. The validator then checks that (f', g') was correctly computed and finally checks the validity of (F', G'). With different $(f', g') \neq (f, g)$, validation relies on the fact that generation is deterministic from the *seed*. This is because all valid pairs (f_j, g_j) are generated by iterative sampling. This process is numerically stable, as shown in Figure 3. The validator expects a correct implementation to pick (f_j, g_j) that can be replicated within limit_1 steps or limit_5 steps, respectively, for security level 1 and 5. The multiple sampling technique can be implemented as described in MultipleSamplings.

Algorithm 2 MultipleSamplings$(seed, f', g')$

Require: Seed $seed$ and polynomials f', g' from the implementation under test.
Ensure: f'', g'' if exists or $\perp$

$limit \leftarrow limit_1$ **if** $n == 512$ **else** $limit_5$
for it **from** 0 **to** $limit$ **do**
    do
        $f'' \leftarrow \text{Genfg}(seed)$ ▷ Assumes Genfg keeps internal SHAKE256 state
        $\widehat{f''} \leftarrow NTT(f'')$
    while $\exists\, k \in [0, n-1]$ where $\widehat{f''}\,[k] == 0$
    $g'' \leftarrow \text{Genfg}(seed)$
    if $f'' == f'$ AND $g'' == g'$ **then**
        return (f'', g'')
return $\perp$

If no match is found, the key is deemed invalid because it did not follow the key-generation steps. If a match is found, the pair (f'', g'') is deemed valid, and the validator can solve for (F'', G'') using NTRUSolve on (f'', g'').

Once (F'', G'') is found, one checks for equality with (F', G'). This reduces to Scenario 1 (see Section 4.2.1). However, if a pair (F', G') is found but is different from (F, G), validation will proceed as defined in Scenario 3 (see Section 4.2.3).

4.2.3 Scenario 3: Partially Different Keys

Keys can share the same (f, g) but have a different $(F, G) \neq (F', G')$. This scenario happens because there are multiple solutions to the NTRU equation (see Equation 2) that are valid with $||(F, G)||_\infty \leq 127$.

A rounding difference in k. The Reduce procedure is a subroutine of NTRUSolve, which is an an important part of SimplifiedKeygen. In a few words, Reduce looks for two polynomials (F, G) from (f, g) that satisfy the NTRU equation (see Equation 2) such that $||(F, G)||_\infty \leq 127$. It uses a variant of Babai's Nearest Plane Algorithm [Bab86] to recursively reduce the size of (F, G) while ensuring that they still fulfill Equation 2. More precisely, the reduction is done by computing a value k to replace (F, G) with $(F - kf, G - kg)$ repeatedly, terminating when $k = 0$. The computation of k involves FFT, rounding large rational numbers to integers. This results in the need for fractional numbers, which makes it numerically unstable (see Section 3). Moreover, a scaling technique is needed to store the large integers with adequate sizes, since even a binary64 floating point may overflow or not have enough precision. This can be performed in multiple ways and may also create numerical instability. This can result in the fractional part of a coefficient of k being above 0.5 for one implementation and below 0.5 for another, yielding different values for k after rounding and thus distinct pairs $(F, G) \neq (F', G')$ that are both valid.

Implementation-specific optimizations. Similarly to Section 4.2.2, using a fixed number of iterations for the Reduce algorithm can sometimes output a pair (F', G') that is reduced enough to meet $||(F, G)||_\infty \leq 127$ but is not the most reduced value. This might happen if the iteration count is set to a low value, and the implementation attempts to use the last value, even if reduction did not reach $k = 0$. It should be noted that (F, G) pairs obtained before reaching the $k = 0$ termination condition are overwhelmingly likely to fail the infinity norm constraint. Nonetheless, if a pair passes the constraint and is used, it might result in a larger discrepancy between (F, G) and (F', G') than would occur in the case of a rounding difference in k. Since failure is overwhelmingly likely if the reduction process fails to reach $k = 0$, we think implementations should not attempt to use incompletely reduced (F, G) pairs. Therefore, our proposed validation process rejects discrepancies larger than would be expected in the case of a rounding difference. However, if a validation body were to determine it was reasonable to allow a larger discrepancy, the part of our validation that checks this (see constraints on Δ in the following section and on Figure 4) could be omitted without a detriment to security or interoperability.

Validating Scenario 3 (partially different keys). When the validator has the same (f, g) as the implementation but a different $(F, G) \neq (F', G')$, it means that a difference occurred in Reduce for the aforementioned reasons. After checking that the NTRU equation (2) is respected for (f, g, F', G') and that $||(F', G')||_\infty \leq 127$, the next step is to check the distance between them (i.e., $||F' - F||_2$, respectively $||G' - G||_2$). Experimentally, we noticed that this difference is a multiple of $||f||_2$, respectively $||g||_2$. This makes sense since the Reduce procedure recursively updates the pair (F, G) by smaller polynomials $(F - kf, G - kg)$. A difference in k yields a difference of a multiple of f. This distance can be measured using the quantity Δ defined as:

$$\Delta := \frac{F' - F}{f}. \tag{5}$$

This polynomial corresponds to the previous difference normalized by f. Since both keys are valid, they fulfill the NTRU equation. Thus, by solving the system of equations below for Δ, one gets that Δ can be computed in two ways:

$$\begin{cases} fG - gF = q \\ fG' - gF' = q \end{cases} \\ \implies f(G' - G) - g(F' - F) = 0 \\ \implies \quad \frac{G' - G}{g} = \frac{F' - F}{f} = \Delta \tag{6}$$

Experimentally, using "correct" implementations, Δ is almost always a polynomial with a unique term $\pm X^i$. When it has more than one term, its coefficients are never greater than 1 in absolute value (over millions of keys). This is to be expected for discrepancies due to rounding differences, as discussed earlier in this section. We therefore expect Δ to always be a sum of terms $\pm X^i$ (i.e., $||\Delta||_\infty \leq 1$), assuming that key generation is done according to the FALCON specification. To measure the distance between the validator's key and the implementation's key, and after having checked that $||\Delta||_\infty \leq 1$, we count the number of terms in the polynomial Δ. This is sometimes referred to as the pseudo-norm L^0. By abuse of notation, we will define $||\Delta||_0 := \#\{i|\Delta[i] \neq 0\}$, where Δ_i is the i-th coordinate of the polynomial Δ.

The last element to conclude the validation of Scenario 3 is to define the acceptable threshold for $||\Delta||_0$ (i.e., the number of non-zero terms in Δ). Experimentally, we observed that over 1 million keys — roughly 0.05% — belong to Scenario 3, and all of them had a Δ with a single term. Assuming that rounding errors in each coefficient of Δ are independent events, we extrapolated this result and expect that two coordinates of Δ would be affected in $\binom{n}{2}(0.05\%/n)^2$ of the cases, which would correspond to about 1 in 8 million keys. This was experimentally proven to be a good approximation since over 10 million keys (see Table 2), we found one with $||\Delta||_0 = 2$, and similar results happened for level 5. More precisely, for level 1, over the 4505 keys in Scenario 3 in Table 2, 4504 corresponded to Δ with 1 term and 1 to a Δ with 2 terms. For level 5, over the 8977 keys in Scenario 3 in Table 3, 8971 keys corresponded to Δ with 1 term and 6 to Δ with 2 terms. Since these experiments over millions of keys are consistent with a rough independence of the events, we could derive bounds for $||\Delta||_0$ such that the probability of a compliant implementation producing a Δ outside of bounds is less than 2^{-128}. Thus, solving $\binom{512}{\delta_1} \cdot ((4504 + 2 \cdot 1)/(10^7 \cdot 512))^{\delta_1} < 2^{-128}$ gives $\delta_1 = 10$ for level 1, and similarly solving $\binom{1024}{\delta_5} \cdot ((8971 + 2 \cdot 6)/(10^7 \cdot 1024))^{\delta_5} < 2^{-128}$ gives $\delta_5 = 11$ for level 5.

Overall, the validation of a pair of keys belonging to Scenario 3 requires first computing Δ, which is possible since it only uses polynomials that the validator has, (f, F, F'), and checking that $||\Delta||_\infty \leq 1$, where $||\Delta||_\infty = 0$ corresponds to equality, thus Scenario 1, and that $||\Delta||_0 \leq \delta_1$ for level 1 or $||\Delta||_0 \leq \delta_5$ for level 5.

4.3 Our Validation Procedure

We define a process in Figure 4 to validate FALCON keys generated by an implementation against a trusted validator's implementation. This is intended to be run over multiple iterations of key generation to determine whether an implementation is valid.

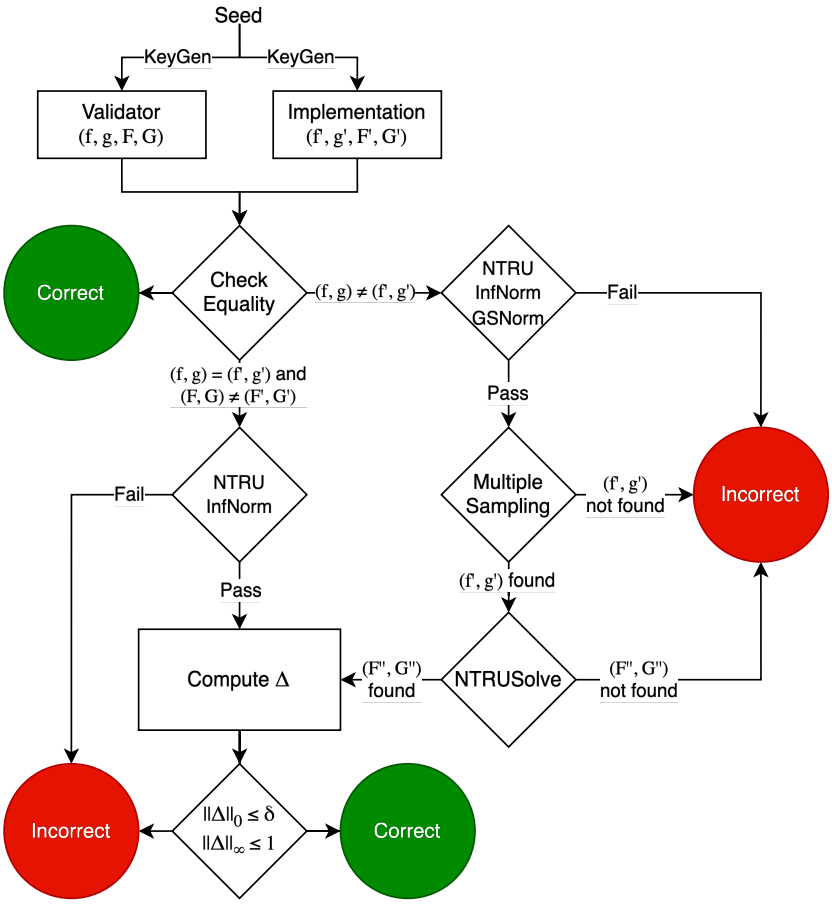

Figure 4: An overview of our proposed process to validate FALCON KeyGen implementations via a trusted validator. δ and the L0 norm are defined in Section 4.2.3.

The process starts by using the same seed in the key-generation function of the implementation and a trusted validator. The validator's implementation may use either floating or fixed points; it does not impact this procedure. This results in the two private key pairs (f, g, F, G) from the trusted validator and (f', g', F', G') from the implementation. If all values are equal, the key generation is correct.

If $(f, g) \neq (f', g')$, the validator checks the NTRU equation, Gram-Schmidt norm, and infinity norm for (f', g', F', G') and then checks whether (f', g') can be regenerated using MultipleSamplings, as in 4.2.2. The validator accepts (f', g') (or rejects the key generation as incorrect) and then computes $(F'', G'') \leftarrow \mathsf{NTRUSolve}(f', g')$ and checks whether $(F', G') = (F'', G'')$. If so, the key generation is accepted. Otherwise, the validator proceeds as below.

If $(f, g) = (f', g')$ but $(F, G) \neq (F', G')$, the validator checks the NTRU equation and infinity norm for (f', g', F', G'). If these checks fail, the key generation is incorrect. If the checks pass or if the validator proceeds from the previous step with (F'', G'') replacing (F, G), the validator determines whether a rounding difference occurred in (F', G'), as described in 4.2.3. If no rounding difference is detected, or the rounding difference polynomial $\Delta = (F' - F)/f$ has a small number of nonzero coefficients that are all equal to ± 1, the key generation can be declared correct. Otherwise, the key generation is incorrect.

5 Going Further in Understanding Falcon Keys

In this section, we answer three questions that arose as we worked on the validation procedure for Falcon key generation:

- Question 1 (Section 5.1): Since there are distinct keys that are valid for a given seed, is one better than another? Specifically, does the choice of one key versus another have any consequences for security?
- Question 2 (Section 5.2): Since floating-point and fixed-point implementations sometimes disagree when given the same seed, which of the two is more "correct"? To answer this question, we implemented an infinite precision version of Falcon's key generation.
- Question 3 (Section 5.3): Out of over 40 million executions of the different Falcon implementations, only a very small number (below 0.1%) corresponded to the case in which the keys were partially different (i.e., Scenario 3 where a pair (f, g) yields two distinct keys (f, g, F, G) and (f, g, F', G')). Given this, one might expect that the number of possible valid keys is quite small. However, we were able to successfully estimate the total number of valid keys (i.e., those satisfying Equation (2) and $||(F, G)||_\infty \leq 127$) that exist for a given (f, g) pair, and we demonstrate that it is very large indeed. While this result is not strictly required to understand the security of our proposed validation process, the novel techniques involved in developing our estimate might be of interest to the cryptographic research community.

5.1 Security Implications

5.1.1 Security Implications of the Choice of (f, g) (Scenario 2)

A number of possible variations in the way (f, g) are generated can have security implications:

- The pseudorandom sampling procedures used to generate (f, g) may differ from those specified in the standard. Our proposed procedure for validation explicitly tests this via MultipleSamplings.
- The key-generation procedure may incorrectly fail to reject an (f, g) pair that yields a too-large Gram-Schmidt norm or has a too-large infinity norm. A too-large Gram-Schmidt norm may lead to a private key where the generated signatures are likely to leak information about the private key. It may not be possible to byte-encode a private key with a too-large infinity norm in the way specified by Falcon. Our proposed validation procedure also tests for this.
- The key-generation procedure may reject some (f, g) pairs that other implementations would have accepted. Assuming that this does not happen with overwhelmingly high probability, and assuming that there are no other flaws in an implementation, this can have only a modest effect on security. In particular, if a seed that could be accepted by a correct implementation of Falcon is rejected with probability p by the implementation in question, the net effect is to restrict the key space by a factor of $1/(1-p)$. Any attack on keys generated by the implementation in question can be applied to the correct implementation with its probability of success reduced by a factor of $1/(1-p)$. Thus, the implementation in question can only be weaker than the correct implementation by at most $\log_2(1/(1-p))$ bits of security.

5.1.2 Security Implications of the Choice of (F, G) (Scenario 3)

Any Existential Unforgeability under Chosen Message Attack (EUF-CMA) on Falcon will rely on exploiting weaknesses in the public key $\mathbf{h}$ and/or in a transcript of signatures (potentially on adversarially chosen messages). The public key cannot have any weaknesses that depend on the choice of F and G, since the public key only depends on f and g. Thus, the security of Falcon is not sensitive to the choice of F and G as long as the signing algorithm does not leak information about the private key.

The distribution of Falcon signatures is analyzed by [HPRR20], which proves under random oracle assumptions on hashing and pseudorandomness that the signatures produced by Falcon will not leak information about the private key, nor will certain timing information. This proof crucially depends on the fact that the subroutine SamplerZ will never be called for an input standard deviation outside of the range $[\sigma_{\min}, \sigma_{\max}]$, where $\sigma_{\min}$ and $\sigma_{\max}$ are parameters defined by the Falcon specification. Appendix E of [HPRR20] gives sufficient conditions for this to hold:

1. The basis $\mathbf{B} = \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$ is symplectic (i.e. $\mathbf{B}^\top \mathbf{J} \mathbf{B} = q\mathbf{J}$ for $\mathbf{J} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$). This follows from the NTRU equation $fG - gF = q$.

2. The Gram-Schmidt norm of $\mathbf{B}$ is less than or equal to $0.9999 \cdot 1.17 \cdot \sqrt{q}$. Provided the NTRU equation holds, the Gram-Schmidt norm of $\mathbf{B}$ only depends on f and g.

Therefore, variability in F and G is acceptable as long as f and g are generated properly, the NTRU equation holds for (f, g, F, G), and the coefficients of F and G are small enough that they can be properly encoded.

5.2 Perfect Keys

The use of fractional numbers in Falcon is solely due to efficiency (e.g., use of FFT to multiply polynomials), but all computations can be done with infinite precision at the cost of a very inefficient execution. For the sake of solving the question of which implementation is "correct" between floating and fixed points, we implemented an infinite precision version of Falcon in the computer algebra language `Magma`. [6] Our infinite precision implementation replicates only NTRUSolve and Reduce, starting from a sampled (f, g) to compute (F, G). This is still meaningful as the call to NTRUSolve is taking almost all of the key-generation time, especially in the context of infinite-precision arithmetic. Not only is this code with infinite precision extremely slow, it also uses a lot of RAM, since the rational coefficients of the polynomials (coming from inverting polynomials in $\mathbb{Q}[x]/(x^n + 1)$ or the large integers) are not rounded and have to be kept in memory. In what follows, we refer to the output of our infinite-precision key generation as *perfect keys*. Table 4 summarizes the time needed to generate a Falcon key using floating point[7], fixed point[8], or our infinite-precision algorithm whose code is publicly available[9]. More precisely, getting a perfect key is roughly 200 times slower than using floating or fixed points for level 1 and roughly 300 times slower for level 5 (see Table 4).

[6] Certain commercial equipment, instruments, or materials (or suppliers, or software, ...) are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

[7] https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Falcon-Round3.zip

[8] https://github.com/pornin/c-fn-dsa.git

[9] https://github.com/usnistgov/falcon-validation

Table 4: Timing comparison between floating points, fixed points, and perfect precision for FALCON key generation with 1000 iterations. Tests ran using an Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz, 512GB DDR3 RAM.

Implementation	Average runtime (ms)	
	FALCON 512	FALCON 1024
Floating Point	7.2	23.6
Fixed Point	7.3	28.3
Our infinite precision algorithm	1510	8070

We used 4505 keys for level 1 and 8977 keys for level 5 from Scenario 3 in Section 4.1. These keys correspond to cases in which floating and fixed-point implementations produced two distinct keys $(f, g, F, G) \neq (f, g, F', G')$. For all of these keys, the floating-point implementation always exactly matched the *perfect key* (i.e., $(f, g, F, G) = (f', g', F', G')$). This suggests that rounding differences induced by fixed-point arithmetic are significantly larger than those induced by floating-point arithmetic in this context. Nonetheless, as noted in Section 5.1.2, all valid choices for (F, G) (i.e., those satisfying the NTRU equation with (f, g) and the infinity norm constraint) are equivalent from both a security and an efficiency perspective. Therefore, the rounding mismatch we see between the fixed-point and infinite-precision implementations of FALCON does not have any security implications.

5.3 Estimating the Number of Distinct Falcon Keys

In the particular experiments summarized in Tables 2 and 3, one needs to generate thousands of FALCON secret keys (f, g, F, G) with different implementations to find instances in which one (f, g) pair yields two distinct yet valid pairs $(F, G) \neq (F', G')$. This experimental observation and the fact that a pair (F, G) has to fulfill the NTRU equation and also have small enough coefficients to pass validation could lead one to think that the number of distinct pairs (F, G) associated with a given (f, g) would be "small." However, this section will prove that this intuition is incorrect by giving the first estimation of the number of distinct FALCON keys that share a valid (f, g) pair and satisfy the NTRU equation and infinity norm constraints on (F, G) procedures for a given (f, g) pair.

As far as we know, the usual combinatorial techniques cannot be used to count the number of valid keys. In fact, if $(f, g, F, G) \in R(n)^4$ is a valid FALCON private key, the (F', G') pairs that satisfy the NTRU equation with the same (f, g) are given by $(F', G') = (F, G) + \Delta \cdot (f, g)$, where Δ is any element of $R(n)$. This may alternatively be described as the space of all points in the $2n$-dimensional NTRU lattice on an n dimensional hyperplane. While L^2-norm constraint would further restrict the space of valid keys to lie inside a $2n$-dimensional hypersphere whose intersection with the relevant hyperplane is an n-dimensional sphere and whose hypervolume is relatively easy to compute and multiply by the density of points in the sublattice lying on the hyperplane, it is much more difficult to come up with a closed-form approximation for the number of valid keys subject to an infinity norm constraint, which corresponds geometrically to a hypercube rather than a hypersphere.

Instead, the first step of our new counting technique relies on tweaking our validation procedure to create many keys from one. More precisely, let $(f, g, F, G) \in \mathbb{Z}[X]/\phi$ be a valid FALCON key. One can reverse the procedure described in Section 4.2.3 to forge a new valid key, namely by picking a nonzero random polynomial $\Delta \in \mathbb{Z}[X]/\phi$. One then gets a new key (F', G') by computing $(F, G) + \Delta \cdot (f, g)$. It is trivial to check that this new key fulfills the NTRU equation. However, this new key might not necessarily be small enough to pass the validation process described above. To make sure that the L^∞ norm of (F, G) will not exceed 127, Δ needs to be picked with a small number of terms and small coefficients.

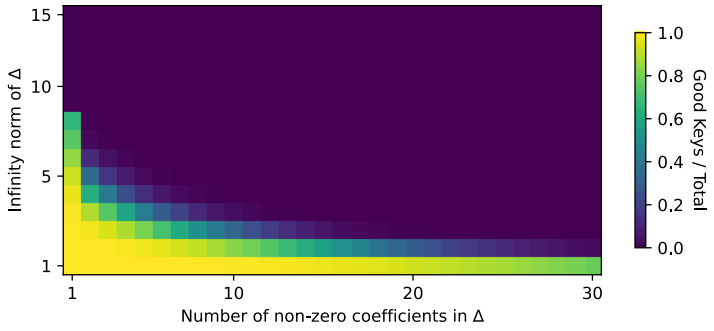

Figure 5: Heatmap of valid keys found when performing varying degrees of modifications on values in Δ using security level 1 with $n = 512$ over 10,000 total iterations. They are cropped to 30 on the x-axis for the sake of visibility, but there are nonzero coefficients after this arbitrary limit.

All possible keys come from this forgery procedure. If a key is valid, it fulfills the NTRU equations, so there exists a Δ such that this valid key can be forged from an existing one. The second step of our procedure relies on this and the fact that the two parameters of the size of Δ — its number of nonzero terms (pseudo-norm L^0) and the maximum value of its coefficients (L^∞ norm) — must have small values that one can exhaustively check. More precisely, we defined a partition of the space of possible Δ, which yields a partition of the space of all possible keys that we are trying to count. This is done through a 2-dimensional table H whose x-axis counts the number of nonzero coefficients and the y-axis the maximum value of these coefficients. The heatmap shown in Figure 5 gives the percentage of keys generated with this method that would pass validation. More precisely, to generate the $(i, j)-$th cell, one picks N_{total} random polynomials Δ with exactly i nonzero terms, and its coefficients are picked at random in $\{-j, -j+1, \ldots, j-1, j\}\backslash\{0\}$. The percentage of such keys that passed validation (i.e., $N_{\text{valid}}/N_{\text{total}}$) is then converted into a color.

The table H is a partition, since for a given value of Δ, there exists only one cell where it belongs. Additionally, the total number of distinct polynomials Δs that are in a cell (i, j) is easily computable using Equation (7):

$$(2j)^i - \binom{n}{i}(2j-2)^i. \tag{7}$$

Using some reasonable assumption about the uniform and independent distribution of the valid keys within each cell, the values in the table H can be used as frequencies that are simply multiplied by the total number of keys given by Equation (7). In other words, for a given (f, g) pair and given an instance of a table H, the total number of distinct valid keys (f, g, F, G) can be estimated by computing:

$$\sum_{x=1}^{\max_i}\sum_{y=1}^{\max_j} H(i,j)\left((2y)^x - \binom{n}{x}(2y-2)^x\right). \tag{8}$$

As previously mentioned, the values of $\max_i$ and $\max_j$ can be easily computed experimentally, since they correspond to the limits above which all cells are empty.

Finally, before applying our formula, we want to pick a key (f, g, F, G) such that the pair (F, G) found with the key generation is not too degenerate, such as a maximal L^∞-norm of 127. Otherwise, even a small Δ would cause the forged key to not pass validation. Thus, the estimate below was computed on a key selected to have norms

aligned with the averages we observed over the 10 million keys we generated (see Table 5). More precisely, the table H from which the heatmap (see Figure 5) was generated was computed from a pair (F, G) such that $||(F, G)||_\infty = 91$ and $||(F, G)||_2 = 857.0$ for security level 1, and $||(F, G)||_\infty = 95$ and $||(F, G)||_2 = 1218.7$ for security level 5 using a floating-point implementation.

Table 5: Statistics (mean and variance) for the L^∞ and L^2-norms of (F, G) over 10,000,000 keys generated with floating-point implementation.

SL	Mean L^∞-norm	Var. L^∞-norm	Mean L^2-norm	Var. L^2-norm
1	90.79184	6.95266	846.19094	20.19716
5	95.52139	6.56736	1194.76681	20.46013

To ensure that our procedure was consistent, we used a non-degenerate key and repeated the procedure for increasing values of N_{total}. By increasing the total number of polynomials generated per cell, some cells that were empty for a smaller value might contribute to the final sum with more iterations.

Our results are summarized in Table 6. The number N_{total} of repetition per cell does not have to be large to get a good estimation of the total number of keys, since the value quickly converges to a stable value. More precisely, the example given by the heatmap corresponds to a key (f, g) with roughly 2^{194} distinct pairs (F, G) for level 1, and a similar experiment gave 2^{217} for level 5. These values contradict the aforementioned intuition of a very small number of valid keys. Additionally, this large number of possible keys does not have any security implication (see Section 5.1.2 for more details).

Remark 1. The subset of good keys that pass our validation corresponds to a $1 \times \delta$-rectangle at the bottom left of the heatmap given by Figure 5. Thus, one could easily adapt the estimate to only count these.

Table 6: Estimation of the number of pairs (F, G) for a given pair (f, g) for security levels 1 and 5.

SL	Iterations	$\log_2(\#\text{ keys})$	SL	Iterations	$\log_2(\#\text{ keys})$
1	10	198.6487	5	10	217.79107
1	100	199.7379	5	100	217.25873
1	1000	200.7830	5	1000	217.24792
1	10000	200.4642	5	10000	217.32574

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John M. Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl A. Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the third round of the nist post-quantum cryptography standardization process. *National Institute of Standards and Technology: Gaithersburg, MD, USA*, 2022.

[ABC+25] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, et al. Status report on the fourth round of the nist post-quantum cryptography standardization process. *National Institute of Standards and Technology: Gaithersburg, MD, USA*, 2025.

[aut24] Multiple authors. [falcon official] keygen implementation. NIST PQC Forum, 2024. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/bjVkrZmI9VM/m/Tj9goDJCAAAJ.

[aut25] Multiple authors. Query on progress of fn-dsa. NIST PQC Forum, 2025. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/CsWPP35LJWw/m/86JLrfudCwAJ.

[Bab86] László Babai. On lovász'lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[Bro24] Maxime Bros. Selected algorithm 2022 official comment: Falcon. NIST PQC Forum, 2024. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/Zhwh95D0KII/m/hWmsRsTpAAAJ.

[FHK+20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over NTRU, specification v1.2 – 01/10/2020. Submission to the NIST's post-quantum cryptography standardization process, 2020. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[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. ACM Press, May 2008.

[HHP+03] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. NTRUSIGN: Digital signatures using the NTRU lattice. In Marc Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 122–140. Springer, Berlin, Heidelberg, April 2003.

[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. Springer, Cham, 2020.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. Ntru: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[IEE19] IEEE. 754-2019 - ieee standard for floating-point arithmetic, July 2019.

[NIS16] Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. National Institute of Standards and Technology: Gaithersburg, MD, USA, 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[Per25] Ray Perlner. FIPS 206 FN-DSA (falcon). Presented at the 6th PQC Standardization Conference, 2025. https://csrc.nist.gov/csrc/media/presentations/2025/fips-206-fn-dsa-(falcon)/images-media/fips_206-perlner_2.1.pdf.

[Por19] Thomas Pornin. New efficient, constant-time implementations of Falcon. Cryptology ePrint Archive, Report 2019/893, 2019.

[Sho99] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.