

Square Root of All Evil: The Dangers of Falcon's Superfluous Square Roots

Hiroto Kaihara

Kyoto University
Kyoto, Japan

kaihara.hiroto.v85@kyoto-u.jp

Mehdi Tibouchi

NTT Social Informatics Laboratories
Tokyo, Japan

mehdi.tibouchi@ntt.com

Calvin Abou Haidar

NTT Social Informatics Laboratories
Tokyo, Japan

calvin.haidar@ntt.com

Masayuki Abe

NTT Social Informatics Laboratories
Tokyo, Japan

msyk.abe@ntt.com

Abstract

FALCON is one of the 3 post-quantum signature schemes already selected by NIST for standardization (as FN-DSA). It is very compact and efficient, but also infamously difficult to implement correctly and securely. This is due in particular to its reliance of various floating point operations, the most complex and costly of which are square root computations.

In this paper, we first point out that those square root computations are in fact wholly unnecessary: the algorithm can be rewritten without them, resulting in a somewhat simpler implementation that is equally fast or even slightly faster.

We then observe that they also present security risks, in particular as a singularly sensitive target for physical attacks. We demonstrate this with a fault attack, supported by concrete experiments against an ARM Cortex-M4 microcontroller target. We show that injecting a single glitch in one square root computation, and then generating around a million signatures with the unperturbed signing algorithm, leads to full key recovery with 100% success rate and, moreover, faulty signatures are not easy to distinguish from validly generated ones. This makes this fault attack the most devastating against FALCON to date, in contrast with earlier attacks requiring hundreds of millions of signature samples, many injected faults, or resulting in signatures that are straightforward to distinguish from regular ones. In addition, we mention potential risks of the square root computations from the standpoint of dependency management and supply chain security.

CCS Concepts

• **Security and privacy** → **Cryptanalysis and other attacks**;
Digital signatures; **Side-channel analysis and countermeasures**;
Software security engineering.

Keywords

FALCON Signature Scheme, Post-Quantum Cryptography, Implementation Security, Fault Analysis, Software Supply Chain

ACM Reference Format:

Hiroto Kaihara, Calvin Abou Haidar, Mehdi Tibouchi, and Masayuki Abe. 2026. Square Root of All Evil: The Dangers of Falcon's Superfluous Square Roots. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3830454.3846751>

1 Introduction

FALCON in the post-quantum transition. Large quantum computers, should they become available, would break the number-theoretic assumptions underlying the RSA and elliptic-curve cryptosystems that are still widely deployed today. This has motivated a sustained standardization and engineering effort around post-quantum cryptography, most prominently through the NIST post-quantum cryptography process. That process has now selected a small number of algorithms for standardization, and deployment work has begun. As a result, it is critical to ensure that the selected algorithms admit implementations that are correct, portable, robust against physical attacks, and maintainable across the heterogeneous platforms on which cryptography is to be deployed.

FALCON, to be standardized by NIST as FN-DSA, is one of the selected post-quantum digital signature schemes [50]. It is a lattice-based hash-and-sign signature scheme, with an attractive efficiency profile: among the schemes selected by NIST so far, it offers the smallest combined public-key and signature sizes, very fast verification, and excellent signing speed on desktop and server platforms. These advantages, however, come at a cost. FALCON is widely regarded as the most delicate of the selected signature schemes to implement securely. In contrast with schemes whose implementations involve little more than modular integer and polynomial arithmetic, and the sampling of uniform distributions, FALCON relies on floating-point computations, fast Fourier transforms over cyclotomic rings, and a carefully designed lattice Gaussian sampler. Those ingredients are central to its compactness and speed, but they also lead to a significant implementation attack surface.

Hash-and-sign lattice-based signatures and their security. In hash-and-sign signatures over lattices such as FALCON, the key pair describes some lattice. The secret signing key is a “good” basis that allows the signer to solve an approximate closest-vector problem: to sign a message, one hashes it to a target point in the ambient space of the lattice, finds a short lattice point close to that target,



This work is licensed under a Creative Commons Attribution 4.0 International License. CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-6/2026/11

<https://doi.org/10.1145/3830454.3846751>

and outputs the difference as the signature. In contrast, the public verification key is a “bad” basis, that is sufficient to check lattice membership, and hence verify that the difference between the signature and the target is indeed in the lattice, but does not allow to find close vectors.

This idea appeared in early lattice signatures from the late 1990s such as GGH [20] and NTRUSign [24]. Those schemes were found to be insecure, however, because the procedures used to find close vectors (Babai’s round-off or nearest-plane algorithm) made the distribution of signatures depend in a simple way on the secret basis. Nguyen and Regev [38, 39] showed how this led to full key recovery from a few hundred to a few thousand signatures. Similar ideas, and refinements of them, later broke patched variants of NTRUSign and more recent hash-and-sign candidates whose output distributions did not sufficiently hide the secret basis, including DRS, PEREGRINE, and EHTv3 [11, 13, 31, 49].

Gentry, Peikert and Vaikuntanathan [18] (GPV) showed how to solve that issue by replacing the deterministic close vector algorithm used in schemes like GGH. Instead of using deterministic Babai round-off or nearest plane, the signer samples from a spherical discrete Gaussian distribution supported over the lattice. Since that distribution depends only on the lattice and not on the secret basis used by the signer, signatures do not reveal any information anymore on the key, and the resulting scheme is provably secure. FALCON is an efficient instantiation of the GPV framework, using NTRU lattices and the Ducas–Prest fast Fourier-based lattice Gaussian sampler [12] to achieve its compactness and efficiency.

Of course, it is critical for security that the Gaussian sampler over the lattice produces the exact intended distribution for signatures, up to negligible statistical deviations. Doing so correctly and securely from an implementation standpoint is not straightforward. For example, early FALCON implementations used a variable-time one-dimensional sampler and were hence shown to leak secret-key information through timing side-channels [16]. Then, the first proposed constant-time implementation of FALCON in 2019 [44] contained a bug that biased the sampler distribution [43], in a way that has recently been shown [2] to lead to complete key exposure; the bug was found and fixed within a couple of months, but showed how tricky it was to get the Gaussian sampler right. Subsequent work has also investigated side-channel attacks against FALCON’s sampler [21, 29, 52], the risks attached to floating-point arithmetic [32], and fault attacks against the scheme [22, 37]. Overall, FALCON provides a clear example of a post-quantum algorithm whose mathematical design is elegant and efficient, but whose real-world security depends on fine implementation details.

Statistical attack techniques. In a hash-and-sign lattice-based signature, signatures can generally be written in the form $\mathbf{x} \cdot \tilde{\mathbf{B}}$, where $\tilde{\mathbf{B}}$ is a fixed matrix related to the secret key (e.g., the secret basis itself, its Gram–Schmidt orthogonalization, or in the case of FALCON, the bit-reversed order Gram–Schmidt orthogonalization induced by the fast Fourier sampling) and where $\mathbf{x}$ is a random row vector following some well-defined distribution. For GGH and NTRUSign, the distribution of $\mathbf{x}$ is basically uniform over the centered unit hypercube. Hence, up to scaling by the singular values of $\tilde{\mathbf{B}}$, the signatures are uniformly distributed in a *rotated* hypercube, and finding that rotation is sufficient to reveal the matrix $\tilde{\mathbf{B}}$ and thus the

secret key. Nguyen and Regev showed that this could be done using Frieze et al.’s algorithm to “learn a linear transformation” [17], which “undoes” the scaling using Cholesky decomposition and recovers the rotation by minimizing a certain function that characterizes the distinguished directions of the target hypercube (the fourth moment function suffices). Most later statistical attacks are based on variants of this idea.

However, this approach can be suboptimal or not work at all if the distribution of $\mathbf{x}$ is too Gaussian-like: after undoing the scaling, one is left with a distribution that is close to rotation-invariant (basically a spherical Gaussian), and thus recovering the secret rotation is infeasible or at least requires an impractical number of signature samples. This was pointed out by the authors of [2, 22], who stressed that “undoing the scaling” throws away valuable information that can be used to break schemes for which the signature distribution is Gaussian-like but not spherical. The buggy 2019 implementation of FALCON and the faulty distribution arising from the attack of [22] are both of that kind. In such cases, they showed that one should apply principal component analysis (PCA), either as a direct method for key recovery that completely sidesteps Nguyen–Regev-style techniques, or as a dimension-reduction tool to make Nguyen–Regev techniques applicable with a more realistic number of signature samples and more practical computational resources.

Our contributions. The starting point of the present paper is a very concrete feature of FALCON’s implementation: its use of square-root computations. The floating-point operations required by FALCON are addition, subtraction, multiplication, division, and square roots, together with simpler conversion and truncation operations. Square roots occur only rarely, but they tend to be more costly and present qualitatively different characteristics. In software-emulated floating point, they are among the most complex and code-heavy operations. In native floating point, they are comparatively slow, and often not constant-time. They are also more idiosyncratic from an implementation standpoint: on some platforms they are provided by dedicated instructions, on others by compiler-generated sequences, by target-specific runtime routines, or by the system mathematical library.

Our first contribution is to show that the square root operation in FALCON is in fact superfluous: the implementation of FALCON can be modified to eliminate them altogether, at virtually no cost or even a slight performance gain. This is done by rewriting the subroutine within one-dimensional Gaussian sampling that samples a certain Bernoulli distribution: instead of comparing an input probability (whose value involves a square root) to a uniform distribution, one can square both sides of the comparison and achieve a square root-free sampler. While the idea is simple in principle, carrying it out at no loss of precision in the distribution and without using more randomness than the original code (which lazily samples the uniform randomness byte by byte) is trickier than it may seem.

Our second contribution is that FALCON’s square roots are highly sensitive targets for fault attacks. Several previous works have shown that perturbing the distribution of one-dimensional Gaussians generated as part of FALCON’s signature generation can lead to key-recovery through statistical attacks against the resulting biased signatures. What we demonstrate is that, on the one hand, injecting a fault in a single square root computation carried out during

FALCON “key expansion” introduces such a bias in all the signatures subsequently generated with that expanded key, and that, on the other hand, this bias is remarkably stark in the sense that it enables key recovery with far fewer signatures than previous attacks. When the fault is injected in the first square root computation, for instance, the covariance of the resulting perturbed distribution has a distinguished one-dimensional complex eigenspace generated by the secret vector. Similarly to the attack of [22], PCA on sufficiently many signatures therefore recovers an approximation of this secret subspace. Unlike in that paper, however, since the subspace dimension in our technique is so small, recovering the exact key (or an equivalent key) then reduces to a simple search on a grid in the complex plane, making it much more efficient than previously possible.

As a third contribution, we validate the fault attack experimentally against an ARM Cortex-M4 microcontroller target using clock glitches injected with ChipWhisperer. The square-root routines are particularly favorable to such faults because their algorithms compute the result progressively, often a few bits at a time. This leaves a wide range of exploitable faulty outputs: the result can be distorted enough to bias the sampler and reveal the key, while still remaining within a range that allows signature generation to continue and does not make faulty signatures trivially distinguishable. We test the glitch attack on all three implementations of FALCON’s square roots for the Cortex-M4, and obtain exploitable square-root faults with 100% success rate for all three. Moreover, while NIST has indicated that they would harden the one-dimensional sampler in the future FN-DSA standard by enforcing bound checks on input standard deviations [40, p. 5], we show that this doesn’t thwart our attack: depending on the square root implementation, we manage to successfully attack 30 to 70% of keys with the same faults, again because numerous different faulty values are attainable, and hence reaching the correct interval is often feasible.

Finally, we discuss a distinct risk created by square roots in platform-native floating-point implementations. On POSIX systems, a call to `sqrt` nominally introduces a dependency on `libm`. On mainstream architectures, however, compilers such as GCC or Clang will often replace such calls by compiler builtins or target instructions, so that the library function is not called at all. This creates an architecture-skewed testing gap: known-answer tests run on x86-64 or AArch64 may never exercise the `sqrt` implementation that is used on a rarer architecture, including modern ones like RISC-V and LoongArch which are geographically concentrated in a single region and may therefore attract nation-station level supply chain attacks. Introducing a slight misbehavior (possibly on a single value) in the `sqrt` function is then sufficient to skew the distribution of signatures on rare platforms in a way that is exploitable for key recovery in the same way as in the fault attack, but does not affect correctness or known-answer test validity on mainstream platforms, making the attack hard to detect. We illustrate this issue with a proof-of-concept based on linker wrapping of `sqrt` and software emulation of several targets with `qemu`.

Taken together, these results demonstrate that FALCON’s square roots are superfluous from an algorithmic standpoint, mildly undesirable from a performance and portability standpoint, and significantly increase the attack surface against the scheme from an

implementation security standpoint. Getting rid of them seems to be the way to go!

2 Background

2.1 Notation

We use bold lowercase letters to describe vectors, and the i -th coordinate of the vector $\mathbf{b}$ is denoted as b_i . The index i used here starts from 0. In other words, if $\mathbf{b} \in \mathbb{R}^n$, then $\mathbf{b} = (b_0, b_1, \dots, b_{n-1})$.

We assume that vectors are represented in row form, and denote the i -th row of a matrix $\mathbf{B}$ by $\mathbf{b}_i$. As an exception to the notation for representing elements of a vector, the j -th element of $\mathbf{b}_i$ is denoted by $b_{i,j}$, which means that $b_{i,j}$ corresponds to the element in the i -th row and the j -th column of $\mathbf{B}$.

In this paper, we deal with operations over the cyclotomic ring generated by a $(2N) - \text{th}$ root of unity for N some power of 2. We denote that ring by $\mathcal{R}_N = \mathbb{Z}[x]/(x^N + 1)$, and the ring obtained after base change from $\mathbb{Z}$ to the ring $\mathbb{Z}_q$ of integers modulo q by $\mathcal{R}_{N,q} = \mathbb{Z}_q[x]/(x^N + 1)$. For an element $f \in \mathcal{R}_N$, its coefficients over the $\mathbb{Z}$ -basis $(1, x, \dots, x^{N-1})$ is denoted by $f_0, \dots, f_{N-1}$, so that $f = \sum_i f_i x^i$. The adjoint f^* of f is defined by $f^* = f(x^{-1}) = f_0 - f_{N-1}x - \dots - f_2 x^{N-2} - f_1 x^{N-1}$.

The ring of $n \times m$ matrices over $\mathcal{R}_N$ is denoted by $\mathcal{R}_N^{n \times m}$. Matrices in general are denoted by bold uppercase letters.

2.2 The Klein–GPV Gaussian Sampler

As mentioned in the introduction, modern hash-and-sign lattice based signatures follow the GPV framework of Gentry, Peikert and Vaikuntanathan [18]: in that framework, the signer knows a “good” basis $\mathbf{B}$ (with relatively short and orthogonal vectors) of a lattice Λ . To sign a message, that message is first mapped via a hash function to a point $\mathbf{c}$ in the ambient space of Λ , and the signer then uses its good basis to sample a lattice point $\mathbf{v}$ close to $\mathbf{c}$ according to a spherical Gaussian distribution $D_{\Lambda, \sigma, \mathbf{c}}$ centered at $\mathbf{c}$ and supported the lattice Λ . The signature is then the short difference $\mathbf{c} - \mathbf{z}$.

Sampling from that lattice Gaussian can be carried out efficiently up to a negligible statistical distance provided that, essentially, the standard deviation σ is larger than the longest vectors of the Gram–Schmidt orthogonalization $\tilde{\mathbf{B}}$ of the basis $\mathbf{B}$. In that case, sampling can be done iteratively in the directions of the successive basis vectors, with coefficients that are distributed according to discrete Gaussian distributions over $\mathbb{Z}$ (much easier to sample from directly than the high-dimensional Gaussian over the lattice). The corresponding algorithm, sometimes called the Klein–GPV sampler in reference to [30], is shown as Algorithm 1.

2.3 Fast Fourier Orthogonalization

A practical instantiation of the GPV framework using NTRU lattices was first proposed by Ducas, Lyubashevsky and Prest in 2014 [10]. The resulting scheme, commonly referred to as DLP, had compact keys and signatures. However, its reliance on the Klein–GPV sampler of Algorithm 1 made it slow compared to lattice-based schemes in the competing “Fiat–Shamir with aborts” paradigm [34, 35], like BLISS [8]. This is because Algorithm 1 has quadratic complexity in the dimension $2N$ of the NTRU lattice, and the computation of the

Gram–Schmidt orthogonalization $\tilde{\mathbf{B}}$ required to run the sampling is even *cubic*.

To address that inefficiency, Ducas and Prest [12] demonstrated that the algebraic structure of lattices arising from cyclotomic rings can be exploited to accelerate the orthogonalization process, as well as algorithms like Babai’s nearest-plane to find close lattice vectors. Their approach leverages the ring’s structure to enable efficient decomposition of the corresponding lattice basis.

For any element $a \in \mathcal{R}_d$, one can define a corresponding matrix $M_d(a) \in \mathbb{Z}^{d \times d}$ such that arithmetic in $\mathcal{R}_d$ naturally corresponds to matrix operations over $\mathbb{Z}^{d \times d}$, in the sense that $M_d(ab) = M_d(a)M_d(b)$.

There are several possible forms of M_d that can achieve this. In this method, M_d is constructed by arranging the coefficients of the polynomial in bit-reversed order.

$$M_d(a) = \left((a \cdot x^{\text{bitrev}(i)})_{\text{bitrev}(j)} \right)_{i,j} \quad (1)$$

In (1), $\text{bitrev}(\cdot)$ denotes the integer obtained by reversing the order of the bits in the $\log_2 d$ -bit representation of its argument. For example, when $d = 2^4$, we have $\text{bitrev}(1) = 8$ and $\text{bitrev}(5) = 10$.

This arrangement allows for a smooth transformation from $\mathcal{R}_d$ to $\mathcal{R}_{d/2}^{2 \times 2}$, as described below, where $a_0, a_1 \in \mathcal{R}_{d/2}$ are defined to satisfy $a(x) = a_0(x^2) + xa_1(x^2)$.

$$M_d(a) = \begin{pmatrix} M_{d/2}(a_0) & M_{d/2}(a_1) \\ M_{d/2}(a_1 \cdot x) & M_{d/2}(a_0) \end{pmatrix} \quad (2)$$

In particular, when a is a self-adjoint polynomial, we have $a_1^* = a_1 \cdot x$. Therefore (2) can be rewritten as follows.

$$M_d(a) = \begin{pmatrix} M_{d/2}(a_0) & M_{d/2}(a_1) \\ M_{d/2}(a_1^*) & M_{d/2}(a_0) \end{pmatrix} \quad (3)$$

Here, we consider orthogonalizing the lattice basis matrix $\mathbf{B} = M_d(a)$. Let $\mathbf{B} = \mathbf{L} \cdot \tilde{\mathbf{B}}$, where $\mathbf{L}$ is a unit lower triangular matrix and the rows of $\tilde{\mathbf{B}}$ are pairwise orthogonal. Then $\mathbf{B}\mathbf{B}^\top = \mathbf{L} \cdot \tilde{\mathbf{B}}\tilde{\mathbf{B}}^\top \cdot \mathbf{L}^\top$ holds. Since $\tilde{\mathbf{B}}\tilde{\mathbf{B}}^\top$ is diagonal, this represents the LDL decomposition of $\mathbf{B}\mathbf{B}^\top$. In other words, the Gram–Schmidt orthogonalization of $\mathbf{B}$ is equivalent to performing an LDL decomposition on $\mathbf{B}\mathbf{B}^\top$. The matrix $\mathbf{B}\mathbf{B}^\top = M_d(aa^*)$ can be reinterpreted as $\begin{pmatrix} M_{d/2}(g_0) & M_{d/2}(g_1) \\ M_{d/2}(g_1^*) & M_{d/2}(g_0) \end{pmatrix}$ according to (3). Then, by performing

Algorithm 1 Gaussian_Sampler($\mathbf{B}, \sigma, \mathbf{c}$)

Require: Basis $\mathbf{B}$ of an n -dimensional lattice Λ , standard deviation $\sigma > 0$, center $\mathbf{c} \in \mathbb{Z}^n$

Ensure: $\mathbf{v}$ sampled in $\mathcal{D}_{\Lambda, \sigma, \mathbf{c}}$

```

1:  $\mathbf{v} \leftarrow \mathbf{0}$ 
2:  $\mathbf{c}' \leftarrow \mathbf{c}$ 
3: for  $i \leftarrow n - 1, \dots, 0$  do
4:    $\mathbf{c}' \leftarrow \langle \mathbf{c}', \tilde{\mathbf{b}}_i \rangle / \|\tilde{\mathbf{b}}_i\|^2$ 
5:    $\sigma'_i \leftarrow \sigma / \|\tilde{\mathbf{b}}_i\|$ 
6:    $z_i \leftarrow \text{SampleZ}(\sigma'_i, \mathbf{c}')$ 
7:    $\mathbf{c} \leftarrow \mathbf{c} - z_i \tilde{\mathbf{b}}_i$  and  $\mathbf{v} \leftarrow \mathbf{v} + z_i \tilde{\mathbf{b}}_i$ 
8: end for
9: return  $\mathbf{v}_0$ 
```

simple polynomial operations, we can obtain three polynomials L_{10}, D_{00} and D_{11} that satisfy

$$\begin{pmatrix} g_0 & g_1 \\ g_1^* & g_0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ L_{10} & 1 \end{pmatrix} \begin{pmatrix} D_{00} & 0 \\ 0 & D_{11} \end{pmatrix} \begin{pmatrix} 1 & L_{10}^* \\ 0 & 1 \end{pmatrix}. \quad (4)$$

Since d is assumed to be a power of 2, the matrix appearing on the diagonal can be further recast as a module over a polynomial ring of half the degree (here, $d/4$). This allows a similar LDL decomposition for 2×2 matrices. By repeating this process recursively until the dimension is reduced to one, the computation can be expanded in a binary-tree-like manner. This sequence of operations yields the decomposition of $\mathbf{L}$ shown in Fig. 1.

By storing the polynomials L_{10} obtained at each decomposition level, a hierarchical tree representation of polynomials can be constructed (see Fig. 2). At the i -th layer of this tree, there exist 2^i elements belonging to $\mathcal{R}_{d/2^i}$. Both the time and space complexity required for constructing this tree representation are $O(d \log d)$, provided that the ring operations are carried out in the FFT domain, representing a significant improvement over $O(d^3)$ complexity of the naive approach.

2.4 The FALCON Signature Scheme

FALCON [50] is a practical lattice-based hash-and-sign signature scheme combining the general structure of the DLP scheme [10] with the algorithmic improvements brought about by fast Fourier orthogonalization in order to achieve both compactness and high signing and verification performance. It has been selected by NIST for future standardization as FN-DSA, alongside the already published standards ML-DSA (formerly DILITHIUM [36]) and SLH-DSA (formerly SPHINCS⁺ [27]).

FALCON, like DLP, follows the GPV framework over an NTRU lattice, but replaces the quadratic Klein–GPV sampler of Algorithm 1 by a quasilinear fast Fourier Gaussian sampler `ffSampling`, obtained by randomizing the fast Fourier nearest-plane algorithm

$$\mathbf{L} = \begin{pmatrix} \boxed{1} & \boxed{0} \\ \boxed{L_{0,0}} & \boxed{1} \end{pmatrix} \begin{pmatrix} \boxed{1} & \boxed{0} & \boxed{0} \\ \boxed{L_{1,0}} & \boxed{1} & \boxed{0} \\ \boxed{0} & \boxed{L_{1,1}} & \boxed{1} \end{pmatrix} \begin{pmatrix} \boxed{1} & \boxed{0} & \boxed{0} \\ \boxed{0} & \boxed{1} & \boxed{0} \\ \boxed{0} & \boxed{0} & \boxed{1} \end{pmatrix} \dots$$

Figure 1: Decomposition of L into block matrices obtained at each level of the LDL decomposition over polynomial-ring modules.

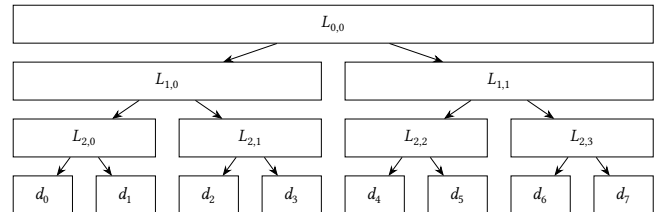


Figure 2: A visual representation of LDL decomposition tree.

ffNP of [12, Sec. 4] in the same way as the Klein–GPV sampler is obtained by randomizing Babai’s nearest-plane algorithm.

The secret basis is given by a 2×2 matrix over the ring:

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

satisfying the NTRU equation $fG - gF = q$, and sometimes stored in FFT form $\hat{\mathbf{B}}$ for efficiency:

$$\hat{\mathbf{B}} = \begin{pmatrix} \text{FFT}(g) & -\text{FFT}(f) \\ \text{FFT}(G) & -\text{FFT}(F) \end{pmatrix}. \quad (5)$$

We also denote by $\mathbf{B}$ the corresponding $2N \times 2N$ integer block matrix under the identification given by M_N .

In the ffSampling algorithm, the LDL decomposition tree of the Gram matrix¹ $\mathbf{G} = \mathbf{B}\mathbf{B}^*$ (of the form depicted in Fig. 2) plays a similar role as the Gram–Schmidt orthogonalization matrix $\hat{\mathbf{B}}$ in the Klein–GPV sampler. FALCON uses a slightly optimized representation of it called the *FALCONtree*, which differs in two ways from a straightforward representation. First, the polynomials stored at each node are represented, like $\hat{\mathbf{B}}$, in FFT form. Second, the leaf values are tweaked for efficient implementation. In a plain LDL tree, the i -th leaf element d_i is equal to the norm of the $(2i)$ -th and $(2i + 1)$ -st rows of the bit-reversed order GSO (those two norms are equal for reasons of symmetry). This is replaced by $\sigma/\sqrt{d_i}$ in the FALCON tree, where σ is the predetermined standard deviation of the signatures. Those modified values are the standard deviations of the one-dimensional Gaussian distributions generated as part of the ffSampling algorithm.

In the DLP scheme, there is a gap between the cubic complexity of Gram–Schmidt orthogonalization and the quadratic complexity of the Klein–GPV sampler than makes is basically necessary to precompute the GSO matrix and cache it across many calls to the signing algorithm. In contrast, there is no such gap in FALCON: both fast Fourier orthogonalization and fast Fourier sampling are quasi-linear. Therefore, it is possible to recompute the FALCON tree every time a signature is generated. Nevertheless, signature generation is substantially faster with a precomputed FALCON tree. For that reason, FALCON supports both approaches: *tree mode* in which the FALCON tree is precomputed (an operation called *key expansion* in FALCON) and cached across many signatures (faster, but requires some long-term memory); and *dynamic mode* in which the FALCON tree is recomputed every time in conjunction with the ffSampling algorithm (slower, but no caching necessary). Although the details of the future FN-DSA standard aren’t known yet, NIST has already announced that both modes are expected to be supported in the draft standard as well [40, p. 7].

3 FALCON Without Square Roots

In all FALCON implementations so far, and as recalled by FALCON’s main implementer T. Pornin on NIST’s pqc-forum mailing list [47], the following floating point operations are required (either in native or emulated form): addition, subtraction, multiplication, division and square roots (alongside simpler operations like truncation and integer conversion). Square roots are significantly more complex than the other four from several standpoints:

- in emulated floating point arithmetic, they are the slowest operation: even in Pornin’s latest hand-optimized assembly implementation for the Cortex-M4 [46, Tab. 2], which significantly sped up square root computations, the square root function still takes 33% more cycles than division, and over 6 times more than all the other operations;
- in native floating point arithmetic, they are again typically the slowest: for example, according to uops.info [1], the `sqrtsd` instruction on Alder Lake-P (resp. Zen 4) has a latency of about 17 uops (resp. 21 uops) compared to 13 uops for `divsd` and 2 to 4 uops for `addsd`, `subsd` and `mulsd`. Similarly, the throughput of `sqrtsd` is about 4.5 uops (resp. 8.4 uops), vs. 4 uops (resp. 5 uops) for `divsd` and 0.5 uops for `addsd`, `subsd` and `mulsd`;
- the native floating point instructions for square roots and division are typically not constant time, unlike those for addition, subtraction and multiplication. Pornin argues that this can only cause very limited key leakage in FALCON on modern x86_64 CPUs due to the fact that only very specific operands lead to faster operations [42, Sec. 3.2]. However, this is architecture-dependent, so relying on native floating point instructions platforms might conceivably be insecure on certain platforms;
- a unique drawback of square roots is that calling the C `sqrt` instruction introduces a dependency on the `libm` math library, regardless of whether compilers like GCC and clang replace calls to that library function by CPU intrinsics. The FALCON code tries to avoid this `libm` dependency on a handful of mainstream architectures by including architecture-specific inline assembly, but the native FPU code still has to pull the `libm` dependency on less common platforms.

For all these reasons, eliminating the use of square roots in FALCON is clearly a desirable goal. We now show how it is not only possible, but can in fact be achieved in a way that not only simplifies the code but also provides a slight performance improvement.

3.1 Use of Square Roots in FALCON

According to the specification [50], square roots are used at exactly one place in the FALCON algorithm: namely, when carrying out the normalization of the FALCON tree, which replaces each leaf value d_i by $\sigma' = \sigma/\sqrt{d_i}$ (steps 6–7 of [50, Alg. 4]).

In the code, this step occurs either during private key expansion (`Zf(expand_privkey)`) when calling the `ffLDL_binary_normalize` function, or when signing in dynamic mode (`Zf(sign_dyn)`) at the deepest level of the `ffSampling_fft_dyntrree` recursive function. In both cases, the final leaf value stored in the tree is actually the inverse of the value in the specification, namely `isigma` = $1/\sigma' = \sqrt{d_i}/\sigma$.

In turn, that value is used only when calling the one-dimensional Gaussian sampling procedure `Zf(sampler)`, where it intervenes in the computation of two values: $dss = \frac{1}{2(\sigma')^2} = \frac{1}{2} \cdot \text{isigma}^2$ and $ccs = \sigma_{\min}/\sigma' = \sigma_{\min} \cdot \text{isigma}$.

3.2 Avoiding Square Roots in FALCON

To be more precise, FALCON’s one-dimensional Gaussian sampler has the following pseudo-code:

¹ $\mathbf{B}^*$ is the conjugate transpose of $\mathbf{B}$ over the ring. Under the identification given by M_N , it simply corresponds to the $2N \times 2N$ transpose matrix $\mathbf{B}^T$ over $\mathbb{Z}$.

```

1:  $r \leftarrow \mu - \lfloor \mu \rfloor$ 
2:  $ccs \leftarrow \sigma_{\min} \cdot \text{isigma}; dss \leftarrow \text{isigma}^2 / 2$ 
3:  $z_0 \leftarrow \text{BaseSampler}()$ 
4:  $b \leftarrow \text{Unif}(\{0, 1\})$ 
5:  $z \leftarrow b + (2b - 1)z_0$ 
6:  $x \leftarrow dss \cdot (z - r)^2 - z_0^2 / (2\sigma_{\max}^2)$ 
7:  $u \leftarrow \text{Unif}([0, 1])$ 
8: if  $u < ccs \cdot \exp(-x)$  then
9:   return  $z + \lfloor \mu \rfloor$ 
10: else
11:   goto 3
12: end if

```

Clearly, the value dss above can be computed from $\text{isigma}^2 = d_i / \sigma^2$, which does not require taking a square root, so the only value for which the square root computation seems required is ccs . That value is used in the rejection sampling step of the sampler, in order to ensure that the rejection probability is independent of the input, which is critical to ensure that the implementation is constant time, as discussed in [26]. As shown in the pseudo-code, this is achieved by sampling a uniform real $u \in [0, 1)$ and accepting the sample z if and only if $u < p$, where the probability p is computed as $ccs \cdot \exp(-x)$.

Now, the same functionality can be achieved in a slightly different way in order to avoid the square root computation: instead of computing p , sampling u and checking whether $u < p$, one can equivalently compute $p^2 = ccs^2 \cdot \exp(-2x)$, sample u uniform in $[0, 1)$, and accept the sample z when $u^2 < p^2$. Indeed, since u and p are positive reals, the two conditions $u < p$ and $u^2 < p^2$ are equivalent. Moreover this approach does indeed avoid the square root since $ccs^2 = \sigma_{\min}^2 \cdot \text{isigma}^2$.

In order to implement this approach correctly and at no loss of performance, one needs to take care of two issues.

First, the precision with which the comparison between p^2 and u^2 is carried out matters. In the original algorithm, ignoring for a moment the lazy Bernoulli sampling optimization, u is sampled with 64 bits of precision, and the comparison is also carried out with a precision of 64 bits. In the square-root free approach, it is similarly sufficient to use around 64 bits for u itself, but in that case, u^2 becomes a 128-bit number. The comparison between p^2 and u^2 should therefore also be over 128 or so bits. This is to account for the fact that we can have $p < 2^{-32}$ with noticeable probability, say, in which case $p^2 < 2^{-64}$ but the comparison of p^2 and u^2 is nevertheless non trivial. Our implementation therefore uses effectively² 128 bits of precision to represent u^2 . As for the computation of $p^2 = ccs^2 \cdot \exp(-2x)$, it can simply use the same polynomial approximation of the exponential function as before. Since its input is in fact the reduction of $2x$ modulo $\log 2$, one technically loses a single ulp of precision on the input, but this is substantially less than the precision of the polynomial approximation itself, so it does *not* affect the theoretical security guarantees of the floating point precision in the sampler. In fact, this is a smaller precision loss than what the original sampler incurs by taking a square root in the computation of isigma and squaring it back when computing dss .

²Fortunately, since u is actually generated byte by byte, one never needs to store 128 bits of u^2 all at once, or to carry out 128-bit-wide multiplications: one can in fact get away with plain 64-bit arithmetic except in the very last iteration of the loop generating the bits of u , which is only executed with probability 2^{-56} .

Table 1: Performance comparison of the square root-free implementation of FALCON-512 with the original on an Intel Core i7-7820X machine. Timings in μs measured by FALCON’s speed tool: ek is for key expansion, sd for dynamic mode signing and st for tree mode signing.

Implementation	ek	sd	st
Baseline (FPemu)	1801	4390	1975
Square root-free (FPemu)	1747	4341	1973
Baseline (FPNATIVE)	90	321	206
Square root-free (FPNATIVE)	89	322	203
Baseline (AVX2)	91	251	126
Square root-free (AVX2)	87	243	130

Second, one of the costlier operations within this part of the algorithm is the sampling of the randomness u itself. To save on the cost of sampling, the original algorithm actually samples u byte by byte instead of 64 bits at a time, and carries out the comparison in a lazy fashion: more bytes only need to be generated when the prefix of u so far is equal to the corresponding prefix of p ; otherwise, the result of the comparison between p and u is known and early exit is possible. This does not lead to secret leakage because the probability of proceeding to the next iteration at each step is always equal to 2^{-8} regardless of the input. While not immediately obvious, it is possible to adapt this technique to the square root-free setting: it suffices to again generate u byte by byte, and proceed from the j -th iteration to the next only when p^2 is in the interval $[u^2, (u + 2^{-8})^2)$. This again leads to no leakage, because this is equivalent to saying that the prefix of p coincides with the prefix of u so far, and therefore always happens with probability 2^{-8} as well.

3.3 Performance Comparison

We have implemented the approach of Section 3.2 as a simple patch to the official FALCON implementation [41]. When the appropriate macro NOSQRT is defined, the `fpr_sqrt` function is commented out, and four functions (`Zf(sampler)`, `BerExp`, `ffLDL_binary_normalize` and `ffSampling_fft_dyntree`) as well as a few constants are modified to adjust to the square-root free algorithm.

We can then compare the efficiency of the resulting modified implementation to the original using the speed measurement tool provided along side the FALCON implementation. On our test workstation (Intel Core i7-7820X with GCC 11.4.0, using the default compiler flags of FALCON, namely `-Wall -Wextra -Wshadow -Wundef -O3`), we find that the square root-free implementation has a slight performance advantage over the original (around 3% faster key expansion, 1% faster dynamic signing and equal speed for tree signing) in floating point emulation mode, while the two implementations perform within measurement noise when using native floating point and AVX2 optimizations, as shown in Tab. 1 (we report numbers for FALCON-512, but the behavior is the same for FALCON-1024 and for the toy parameters FALCON-256).

We also performed cycle count measurements on a Cortex-M4 microcontroller (STM32F415RGT6) for the Cortex-M4 assembly implementation (ASM_CORTEXM4), and obtained similar results: on

average and with the precision of our measurements, both implementations carried out signing in dynamic mode (resp., tree mode) is 72 million (resp., 22 million) cycles. Moreover, the square root-free implementation saves just over 3400 bytes of compiled code, which is often non-negligible on this type of embedded platforms.

4 Fault Attack on FALCON's Square Roots

In this section, we discuss how modifying a specific leaf element of the FALCON tree affects the distribution of signatures in a way that leads to an efficient full key recovery attack. Furthermore, we show that this modification can be achieved using a single fault injection on the first square root computation during the computation of that FALCON tree (also known as “key expansion” in FALCON’s terminology).

Remark 1. Our analysis in this section assumes that FALCON’s signing algorithm is used in *tree mode*, where key expansion is performed once and the resulting FALCON tree is reused for many signatures. A single corrupted square-root output during expansion therefore contaminates the cached tree and affects all subsequent signatures produced with it. In contrast, in *dynamic mode*, the tree is recomputed for each signature. The same attack is still possible in that setting in principle, with key recovery requiring the same number of signatures if one manages to inject reliable faults, but the attacker then has to inject one fault into the first square root computation of *every* signature generation, so a total of a few million fault injections, which we regard as a considerably stronger attack model. In dynamic mode, again, a single fault once and for all during key expansion suffices.

4.1 The Distribution of Signatures in FALCON

As discussed in Section 2, FALCON’s signing algorithm uses the `ffSampling` lattice Gaussian sampler based on Ducas and Prest’s fast Fourier orthogonalization [12]. While it differs significantly from the Klein–GPV sampler given in Algorithm 1 from an algorithmic standpoint, it does functionally the same thing: both samplers take a center vector $\mathbf{c}$ as input, map it to a target $\mathbf{t} = \mathbf{c} \cdot \mathbf{B}^{-1}$ in $\mathbb{R}^{2N}$, and then sample an integer vector $\mathbf{z}$ according to an *elliptic* Gaussian distribution centered at $\mathbf{c}$ with suitably chosen covariance matrix Σ . Finally, the samplers output the lattice point $\mathbf{z} \cdot \mathbf{B}$, and the corresponding signature is the difference $\mathbf{s} = (\mathbf{t} - \mathbf{z}) \cdot \mathbf{B}$.

From that perspective, the only major difference between FALCON on the one hand, and its predecessor DLP [10] based on the Klein–GPV sampler on the other hand, is that the $2N \times 2N$ matrix $\mathbf{B}$ over $\mathbb{Z}$ associated with the NTRU signing key is obtained in the standard power basis, whereas for FALCON which uses `ffSampling`, it is in the bit reversed order basis.

In both cases, $\mathbf{x} = \mathbf{t} - \mathbf{z}$ follows a Gaussian distribution with mean 0 and correlation matrix Σ (this is a discrete Gaussian supported over a certain coset of $\mathbb{Z}^{2N}$, but that detail does not matter for our purposes), and therefore, $\mathbf{s} = \mathbf{x} \cdot \mathbf{B}$ is similarly Gaussian with mean 0 and correlation matrix $\mathbf{B}^{-\top} \Sigma \mathbf{B}$. One wants to choose Σ in such a way that $\mathbf{s}$ is spherical, so one picks:

$$\Sigma = \mathbf{L}^{-\top} \cdot \sigma^2 \mathbf{D}^{-1} \cdot \mathbf{L}^{-1},$$

where σ is the desired standard deviation, and $\mathbf{L}$ and $\mathbf{D}$ are the matrices obtained by LDL decomposition of the Gram matrix $\mathbf{G} = \mathbf{B}\mathbf{B}^\top$, i.e., $\mathbf{B}\mathbf{B}^\top = \mathbf{L}\mathbf{D}\mathbf{L}^\top$.

4.2 On Contaminating a Single Leaf of FALCON Tree

Let us examine how modifying the values of the leaf elements in the FALCON tree affects the distribution of signatures. Considering that the root and intermediate nodes of the FALCON tree correspond to $\mathbf{L}$ in the LDL decomposition of the Gram matrix $\mathbf{G}$, while the leaf elements correspond to $\mathbf{D}$, modifying the leaf values is equivalent to replacing the elements of $\mathbf{D}$.

It is also worth noting that a single leaf in the FALCON tree corresponds to two adjacent diagonal elements of $\mathbf{D}$. The elements of the diagonal matrix $\mathbf{D}$ correspond to the norms of the columns of the Gram-Schmidt matrix $\tilde{\mathbf{B}}$ of $\mathbf{B}$; due to its construction, the norms of the $2i$ -th and $(2i + 1)$ -th rows are equal. In the FALCON tree, this norm is associated solely with the i -th leaf element. In this way, N leaf elements fully represent the $2N \times 2N$ diagonal matrix $\mathbf{D}$.

Now, let us assume that the i -th leaf element of FALCON tree is replaced from $\sigma/\sqrt{d_i}$ to $\sigma'/\sqrt{d_i}$. Then the covariance matrix (denoted by Σ) of the discrete Gaussian distribution which $\mathbf{x}$ follows is expected to change into Σ' , which is described as:

$$\Sigma' = \mathbf{L}^{-\top} \cdot \sigma'^2 \mathbf{P} \mathbf{D}^{-1} \cdot \mathbf{L}^{-1} \quad (6)$$

where $\mathbf{P}$ is a diagonal matrix that satisfies

$$p_{k,k} = \begin{cases} (\sigma'/\sigma)^2 & (k \in \{2i, 2i + 1\}) \\ 1 & (\text{otherwise}) \end{cases} \quad (7)$$

Remark 2. Technically, the previous description is correct to the extent that the one-dimensional Gaussian sampler on input of the modified variance outputs a distribution that does indeed have that variance. In the case of FALCON, this is automatically satisfied if the variance falls in the “correctness interval” of the one-dimensional sampler `Zf(sampler)` or a bit beyond, but values further away behave differently. For faulty variances that are too small, the sampler’s acceptance rate becomes so low that it virtually turns into an infinite loop, so the attack fails. In contrast, for faulty variances that are very large, it almost always exits immediately, resulting in a fixed distribution that no longer depends on the input variance. The latter is true even for the case of “infinite variance” corresponding to a zero value of the inverse leaf isigma.

We ignore that issue in the remainder of this section, but the general case is amenable to a very similar analysis, provided that we replace the faulty input variance by the actual variance of `Zf(sampler)` on that input.

Since $\mathbf{s} = \mathbf{x}\mathbf{B}$, the covariance matrix of the signature $\mathbf{s}$ is given by $\mathbf{B}^\top \Sigma \mathbf{B}$. Now, by using the orthogonal matrix $\mathbf{Q}$, which is obtained by normalizing the row vectors of $\tilde{\mathbf{B}}$, $\mathbf{B}$ can be decomposed into $\mathbf{L}\mathbf{D}\mathbf{Q}$. Therefore,

$$\mathbf{B}^\top \Sigma \mathbf{B} = \mathbf{Q}^\top \mathbf{D}^{1/2} \mathbf{L}^\top \mathbf{L} \mathbf{D}^{1/2} \mathbf{Q} \quad (8)$$

$$= \mathbf{Q}^\top \mathbf{D}^{1/2} \cdot \sigma^2 \mathbf{P} \mathbf{D}^{-1} \cdot \mathbf{D}^{1/2} \mathbf{Q} \quad (9)$$

$$= \sigma^2 \mathbf{Q}^\top \mathbf{P} \mathbf{Q} \quad (10)$$

We now consider the eigenvalue equation in the row space,

$$\mathbf{w}(\sigma^2 \mathbf{Q}^\top \mathbf{P} \mathbf{Q}) = \lambda \mathbf{w}. \quad (11)$$

It follows that

$$\mathbf{w} \mathbf{Q}^\top (\sigma^2 \mathbf{P}) = \lambda \mathbf{w} \mathbf{Q}^\top \quad (12)$$

which implies that $\mathbf{w} \mathbf{Q}^\top$ is an eigenvector of $\sigma^2 \mathbf{P}$. Because $\mathbf{P}$ is diagonal, its eigenstructure is explicit: it possesses an eigenvalue σ^2 with multiplicity $2N-2$, corresponding to the eigenspace generated by the vectors

$$\mathbf{e}_0, \mathbf{e}_1, \dots, \mathbf{e}_{2i-1}, \mathbf{e}_{2i+2}, \dots, \mathbf{e}_{2N-1},$$

and an eigenvalue σ'^2 with multiplicity 2, corresponding to the eigenspace generated by $\mathbf{e}_{2i}$ and $\mathbf{e}_{2i+1}$, where $\mathbf{e}_j$ denotes the j -th standard basis vector in $\mathbb{R}^{2N}$.

Since $\mathbf{Q} = (\mathbf{q}_0, \mathbf{q}_1, \dots, \mathbf{q}_{2N-1})^\top$, the solution to $\mathbf{w} \mathbf{Q}^\top = \mathbf{e}_j$ is $\mathbf{w} = \mathbf{q}_j$. This leads to the conclusion that the covariance matrix of the signatures $(\mathbf{B}^\top \Sigma' \mathbf{B})$ possesses an eigenvalue σ^2 corresponding to the eigenspace generated by

$$\mathbf{q}_0, \mathbf{q}_1, \dots, \mathbf{q}_{2i-1}, \mathbf{q}_{2i+2}, \dots, \mathbf{q}_{2N-1},$$

and an eigenvalue σ'^2 corresponding to the eigenspace generated by $\mathbf{q}_{2i}, \mathbf{q}_{2i+1}$.

This fact indicates that the distribution of the signatures can be stretched or compressed along a specific direction by modifying a single element of FALCON tree. Furthermore, since each $\mathbf{q}_j$ is a scalar multiple of $\tilde{\mathbf{b}}_j$, analyzing the direction of the distortion in the distribution allows us to identify the subspace spanned by two particular row vectors of $\tilde{\mathbf{B}}$. When the leftmost leaf is modified (i.e., $i = 0$), in particular, the subspace spanned by $\tilde{\mathbf{b}}_0$ and $\tilde{\mathbf{b}}_1$ can be identified. Since $\tilde{\mathbf{b}}_0 = \mathbf{b}_0$ and $\tilde{\mathbf{b}}_1 = \mathbf{b}_1$, whose components correspond to the coefficients of the secret key polynomials f and g , this directly leads to a potential leakage of the secret key.

4.3 Full Key Recovery from the Normalized Eigenvectors

We have found that the covariance matrix of signatures generated from a FALCON tree whose leaf nodes have been overwritten admits, in theory, an eigenvector corresponding to the GSO basis. However, this eigenvector is obtained only in normalized form and retains two degrees of freedom. In this chapter, we describe a method for recovering $\tilde{\mathbf{b}}_0$, which consists of coefficients of f, g , by performing an exhaustive search over the scaling factor of the eigenvector. We also discuss the order of the number of signatures required for this attack to succeed.

4.3.1 Search over the Eigenspace. Assume that the experimentally derived eigenspace intersects the unit-volume hypercube centered at the correct lattice point $(g, -f)$. Under this assumption, there exist real coefficients $\alpha, \beta \in \mathbb{R}$ such that, after rounding each coordinate of $\alpha \mathbf{u}_0 + \beta \mathbf{u}_1$ to the nearest integer, the resulting vector in $\mathbb{Z}^{2N}$ coincides with $\tilde{\mathbf{b}}_0 = (g, -f)$.

According to FALCON specification, the norm of $(g, -f)$ is guaranteed not to exceed $1.17\sqrt{q}$; if a pair (f, g) violating this bound is generated, the coefficients will be resampled. Therefore, under the assumption that the unit eigenvectors $\mathbf{u}_0$ and $\mathbf{u}_1$ are orthogonal, it suffices to search over the region satisfying $\alpha^2 + \beta^2 < 1.17^2 \cdot q \approx 16682$.

We denote by S the circular domain in the (α, β) -plane defined by this inequality.

By examining each coordinate of $\tilde{\mathbf{b}}_0$, $\tilde{b}_{0,0} = d$ is inferred only when (α, β) lies within the region of S bounded by the two parallel lines:

$$u_{0,0}\alpha + u_{1,0}\beta = d - 1/2$$

and

$$u_{0,0}\alpha + u_{1,0}\beta = d + 1/2.$$

More precisely, the set S is partitioned by the family of lines

$$\left\{ u_{0,0}\alpha + u_{1,0}\beta = d + \frac{1}{2} \mid d \in \mathbb{Z} \right\} \quad (13)$$

into approximately $\lceil 2 \cdot 1.17\sqrt{q} \cdot \|(u_{0,0}, u_{1,0})\| \rceil$ regions, each of which corresponds to a different value of $\tilde{b}_{0,0}$. The same partitioning procedure is applied to every pair $(u_{0,i}, u_{1,i})$ for $i \in [2N]$. Every region of S produced by these intersecting lines corresponds to a distinct unit hypercube in $\mathbb{R}^{2N}$, and therefore to a different estimated value of $\tilde{\mathbf{b}}_0$.

Algorithm 2 presents the procedure that enumerates each partitioned region of S and identifies the correct $\tilde{\mathbf{b}}_0$.

Since the sum of the squared norms of $\|(u_{0,j}, u_{1,j})\|$ satisfies

$$\sum_{i=0}^{N-1} \|(u_{0,j}, u_{1,j})\|^2 = \|\mathbf{u}_0\|^2 + \|\mathbf{u}_1\|^2 = 2, \quad (14)$$

it follows that

$$\sum_{i=0}^{N-1} \|(u_{0,j}, u_{1,j})\| \leq N \cdot \sqrt{\frac{2}{N}} = \sqrt{2N}. \quad (15)$$

Algorithm 2 Search for the correct $\tilde{\mathbf{b}}_0$

Require: mutually orthogonal unit vectors $\mathbf{u}_0, \mathbf{u}_1, h \in \mathcal{R}_{N,q}$

Ensure: estimated $\tilde{\mathbf{b}}_0 = (g, -f)$ such that $fh = g \pmod q$

- 1: $d_i^{\text{lim}} \leftarrow \lceil 1.17\sqrt{q} \cdot \|(u_{0,i}, u_{1,i})\| \rceil$
- 2: **for** $i = 0$ to $2N - 1$, $d_i = -d_i^{\text{lim}}$ to $d_i^{\text{lim}} - 1$ **do**
- 3: **for** $j = i + 1$ to $2N - 1$, $d_j = -d_j^{\text{lim}}$ to $d_j^{\text{lim}} - 1$ **do**
- 4: Solve for (p_α, p_β) :

$$\begin{pmatrix} p_\alpha \\ p_\beta \end{pmatrix} \leftarrow \begin{pmatrix} u_{0,i} & u_{1,i} \\ u_{0,j} & u_{1,j} \end{pmatrix}^{-1} \begin{pmatrix} d_i + 1/2 \\ d_j + 1/2 \end{pmatrix}$$

- 5: **for** $s_i \in (0, 1)$, $s_j \in (0, 1)$ **do**
- 6: Compute $\tilde{\mathbf{b}}$ such that:

$$\tilde{b}_k \leftarrow \begin{cases} \lfloor u_{0,k} p_\alpha + u_{1,k} p_\beta \rfloor & (k \notin \{i, j\}) \\ d_i + s_i & (k = i) \\ d_j + s_j & (k = j) \end{cases}$$

- 7: $(\tilde{g}, -\tilde{f}) \leftarrow \tilde{\mathbf{b}}$
 - 8: **if** $\tilde{f} \cdot h = \tilde{g} \pmod q$ **then**
 - 9: **return** $\tilde{\mathbf{b}}$
 - 10: **end if**
 - 11: **end for**
 - 12: **end for**
 - 13: **end for**
 - 14: **return** $\perp$
-

Therefore, both the outer and inner loops iterate only $O(\sqrt{N})$ times, yielding a total of $O(N)$ iterations. Inside each loop, the bottleneck operation is the multiplication of f and h in $\mathcal{R}_{N,q}$, which requires at least the time $O(N \log N)$ achieved by using NTT. Consequently, the overall time complexity of Algorithm 2 is $O(N^2 \log N)$.

Although this algorithm counts regions of S with some redundancy (because each region is effectively enumerated once for every vertex of the polygon obtained by intersecting the partitioning lines), and because the actual key, since it is a high-dimensional Gaussian vector, actually has a norm very close to the upper bound $1.17\sqrt{q}$, there is still room for optimization. Nevertheless, even with those suboptimal choices, the resulting $O(N^2 \log N)$ complexity remains sufficiently efficient in practice.

4.3.2 Convergence of Eigenvectors under Distributional Bias. The authors of [22] provide insight into the number of signatures required to achieve full key recovery. By applying the Davis–Kahan theorem [7] (see [22, Th. 2]), they show that the distance between the leading eigenvector $\mathbf{v}_1$ of the true covariance matrix Σ and the top- k eigenspace $\tilde{V}_k$ of the perturbed covariance matrix $\tilde{\Sigma}$, with eigenvalues $\tilde{\lambda}_1 \geq \tilde{\lambda}_2 \geq \dots \geq \tilde{\lambda}_k \geq \dots \geq 0$, is inversely proportional to the eigenvalue gap $\tilde{\lambda}_1 - \tilde{\lambda}_{k+1}$.

Moreover, regarding the convergence rate of the empirical eigenvectors as the number of samples increases, we define ρ_k as the correlation between $\mathbf{v}_1$ and its projection onto the top- k eigenspace $\tilde{V}_k$. Abou Haidar et al. further suggest that ρ_k approaches 1 at a rate inversely proportional to the number of samples n , with proportionality constant $C \approx 1.7 \times 10^7$:

$$\rho_k \geq 1 - \frac{C}{n(\tilde{\lambda}_1 - \tilde{\lambda}_k)}. \quad (16)$$

In our setting, two eigenvalues of the distorted signature distribution are equally larger than the others, allowing us to extract the corresponding eigenspace. By setting $k = 2$, the coverage rate is therefore governed by the spectral gap $\tilde{\lambda}_1 - \tilde{\lambda}_3$. The third eigenvalue

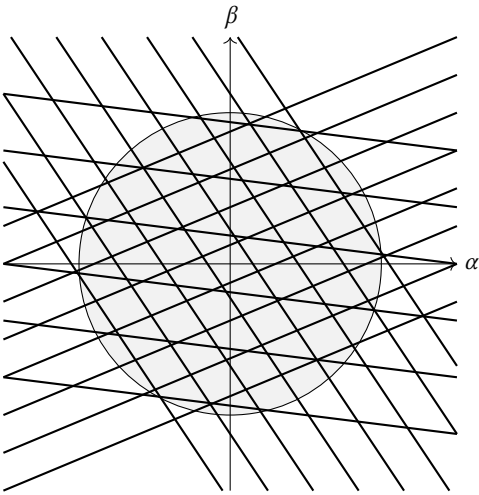


Figure 3: The disk-shaped region S of radius $1.17\sqrt{q}$, partitioned by numerous parallel lines whose normal vector is $(u_{0,i}, u_{1,i})$.

Algorithm 3 Outline of fpr_sqrt

```

1:  $xu \leftarrow (1 \ll 52) + x[51 : 0]$ 
2:  $ex \leftarrow x[62 : 52]$ 
3:  $e \leftarrow ex - 1023$ 
4: if  $e$  is odd then
5:    $e \leftarrow e - 1$ ;  $xu \leftarrow xu \ll 1$ 
6: end if
7:  $e \leftarrow e \gg 1$ 
8:  $xu \leftarrow \begin{cases} xu \ll 9 & (32\text{-bit ASM}) \\ xu \ll 1 & (\text{C impl.}) \end{cases}$ 
9:  $q \leftarrow \text{ISQRT}(xu)$ 
10: {other instructions}
11: return
```

$\tilde{\lambda}_3$ matches the variance of the nominal FALCON signature distribution, which is approximately $\sigma^2 \approx 2 \times 10^4$. Importantly, FALCON’s rejection sampling discards signatures whose Euclidian norm is too large, thereby slightly reducing the empirical variance compared to the ideal σ^2 . However, this reduction does not change its order of magnitude. The value of $\tilde{\lambda}_1$ depends on the magnitude of the fault introduced into FALCON tree leaves; assuming that the fault inflates the corresponding variance by a factor of roughly 1.5–2, we obtain $\tilde{\lambda}_1 - \tilde{\lambda}_3 \approx 10^4$. Consequently, achieving a correlation of $\rho_k = 0.999$ (a threshold indicative of reliable key recovery) requires on the order of $n \approx 2 \times 10^6$ signatures. We find that this order of magnitude is indeed well-verified in experiments.

4.4 Fault Attack Targets in the Floating-point Emulation Code

As we discussed, the values $d_i = \|\tilde{\mathbf{b}}_i\|^2$ appearing in the leaf elements of the LDL tree of Gram matrix $\mathbf{G}$ are normalized as $\sigma/\sqrt{d_i}$. To avoid unnecessary divisions, we have mentioned that the reference implementation instead stores its reciprocal $\text{isigma} = \sqrt{d_i}/\sigma$ (see lines 137 and 378 of `sign.c` in [41]).

When floating point emulation is enabled, either because the target platform lacks native FPU support or by deliberate choice³ at compile time (setting the `FALCON_FPEMU` macro to 1), square roots are computed using a hand-written `fpr_sqrt` function operating on 64-bit integers representing double precision IEEE 754 floating point values. Two versions of this function are provided in [41]: one in ARM Cortex-M4 assembly and one in pure C, that broadly follow the pseudocode provided in Algorithm 3 (since then, Pornin also proposed another assembly implementation with further optimizations [46]).

To illustrate the extent to which that code is sensitive to fault attack, we analyze how it may be possible to obtain faulty leaves of the FALCON tree leading to key recovery by skipping a well-chosen instruction in those algorithms. The analysis for now is at the pseudocode level, so real-world faults do not necessarily translate to instruction skips in that setting, but hopefully it gives an idea of the principles underlying the fault attack. Concrete experimental results will be described in the following section.

³Relying on the (well-defined, deterministic) emulated floating point implementation is sometimes needed to address issues related to the “weak determinism” of native FPUs. See for example the discussions in [9, 32].

If the operations in lines 1–3 of Algorithm 3 are skipped, the variables will not be assigned correctly, likely causing errors or producing values that are far from the expected ones. Some of those values are likely useful (for example, as we mentioned, the attack works well against current FALCON implementations if the fault results in a zero value for the output isigma), but they may be detected by further hardening, such as the bound validation that NIST plans to add to the one-dimensional sampler [40, p. 5].

Skipping the conditional operations in lines 4–6 allows the output to be multiplied by $1/\sqrt{2}$ when e is odd. The output does not change if e is even, but in the calculation of the FALCON tree leaves, the input value is highly likely to have $e = 13$. Therefore this could be an effective strategy. Skipping line 7 might seem like a reliable way to shift the value, but when $e = 13$, the output is multiplied by 2^6 . This can prevent the preimage sampling from terminating due to a too small standard deviation, making signature generation prohibitively slow. Skipping line 8 also produces a $1/\sqrt{2}$ change in the C implementation, which can be a useful attack. However, in the 32-bit ASM version, it causes a large change of $2^{4.5}$, likely leading to the same problem as on line 7.

In summary: under assembly emulation, the conditional operations in lines 4–6 are effective insertion points for a skipping fault; under C emulation, the left bit-shift on line 8 is also an effective target. For the assembly code, the relevant operations are written in `fpr.c` at lines 1050–1056 (in particular, skipping the assignments in either line 1055 or line 1056) will shift the output value. Skipping both of these lines (which may be achievable with a single glitch of sufficient width) shifts the output by a factor of $1/\sqrt{2}$. For the C emulation, lines 1228 and 1234 are effective attack targets: skipping both yields an output equal to half the correct value, while skipping only one yields an output equal to $1/\sqrt{2}$ times the correct value.

Furthermore, the routine that computes the square root of the mantissa can also be a viable fault-attack target. In [41], the square root is computed bit-by-bit using a loop (see Algorithm 4). Aborting that loop prematurely or skipping operations inside the loop can corrupt the computed result. The mantissa originally lies in the range $[1, 2)$, and even after the operation on line 5 it remains in $[1, 4)$. Its square root should therefore be in $[1, 2)$. Moreover, as long as the first iteration of the loop executes correctly the computed result will be at least 1, so no calculation error larger than a factor of 2 can appear. Consequently, such an error is unlikely to be large enough to be caught by validation or to noticeably slow down signature generation.

Algorithm 4 shows some more detailed pseudocode for the pure C version of the square root (up to small simplifications to improve readability). Note first that the input xu represents values in the range $[2^{53}, 2^{55})$, corresponding to the original range $[1, 4)$ (denoted by x) scaled by 2^{53} . The variable q stores the bits obtained from the preceding computations and is likewise scaled by 2^{53} relative to its original value (denoted by q).

Let us now verify that this routine outputs a correct approximation of the square root of x . To do so, we define a real number r as 2^{-53} times the value of the variable r . Suppose that at the beginning of the i -th loop, the variable q correctly represents the upper i bits of $\sqrt{x}$, and that the relation

$$xu = (x - q^2) \cdot r^{-1} \quad (17)$$

Algorithm 4 Bitwise Square Root Computation (pure C)

```

1:  $q \leftarrow 0$ 
2:  $s \leftarrow 0$ 
3:  $r \leftarrow 1 \ll 53$ 
4: for  $i = 0$  to 53 do
5:    $t \leftarrow s + r$ 
6:   if  $xu \geq t$  then
7:      $s \leftarrow s + (r \ll 1)$ 
8:      $xu \leftarrow xu - t$ 
9:      $q \leftarrow q + r$ 
10:  end if
11:   $xu \leftarrow xu \ll 1$ 
12:   $r \leftarrow r \gg 1$ 
13: end for

```

holds. The case $i = 0$ trivially satisfies this assumption.

Whether to set the bit corresponding to $r (= 2^{-i})$ in q during the current i -th iteration depends on whether the following condition holds:

$$x \geq (q + r)^2. \quad (18)$$

This inequality is equivalent to

$$(x - q^2) \cdot r^{-1} \geq 2q + r, \quad (19)$$

so we compare the values of the variables xu and $t = s + r$; if xu is greater, we set the bit corresponding to 2^{-i} in q .

When the bit is set, we add twice that bit value to s and subtract t from xu . The new value of xu then becomes

$$(x - q^2) \cdot r^{-1} - (2q + r) = (x - (q + r)^2) \cdot r^{-1}. \quad (20)$$

If we denote the updated values as q' and r' , this gives

$$xu = (x - q'^2) \cdot (2r')^{-1}. \quad (21)$$

After that, xu is multiplied by 2, restoring the invariant so that the initial assumption again holds at the start of the next iteration.

Therefore, by repeating this process, q will contain an increasingly accurate approximation of $\sqrt{x}$.

A close observation above reveals that the final output value is stored in the variable q . Therefore, by skipping line 9 of Algorithm 4 which directly modifies this variable, it is possible to manipulate the output value directly. Further more, since the digits of q are set sequentially from the most significant bit, aborting the loop at an earlier stage induces a larger computational error.

4.5 Experiments on a Cortex-M4 Target

We implemented the fault attack against a STM32F4 microcontroller using the CW120 ChipWhisperer Pro (depicted in Figure 4) to inject glitches in the clock signal. The different implementations of the square root functions were isolated in order to accurately identify the effects of the fault injection during execution. The ChipWhisperer can be fed some parameters to alter the clock signal of the chip in a specific way, allowing fine grain control and reproducibility of glitches. Our experiments reveal that most parameters having an effect on the output of the square root functions decrease the output value, sometimes bringing it very close to 0. Less commonly, the output can also be increased. Some parameters allowed to decrease the output value in a somewhat “continuous” fashion (for some

Table 2: Key-recovery rates out of a 100 keys. “In bound” refers to keys that were recovered using glitched values corresponding to a standard deviation within the $[\sigma_{\min}, \sigma_{\max}]$ range.

	Success (%)	In bound (%)
ASM25	100	65
ASM21	100	46
C21	100	34

input range), such that close inputs would result in close glitched outputs. Once we identified the glitch parameters of interest, we generated 100 keys and ran the glitch attack to inject the resulting faulty value into the first leaf of their FALCON tree. The key recovery attack of Section 4 was then run on these keys.

For completeness, we targeted 3 different square root implementations: the assembly and the pure C implementations of `fpr_sqrt` in the FALCON reference implementation [41] (referred to as ASM21 and C21 respectively), as well as the `fndsa_fpr_sqrt` function in Pornin’s newer code for the the Cortex-M4 [46]. We were able to confirm that a 100% of the keys were fully recovered in the attacks on all three implementations. The key recovery attack required 1–2 million signature samples in most cases, and never more than 10 million.

None of these implementations enforce a bound check on the input standard deviation of the one-dimensional Gaussian sampler. However, NIST has announced that such bound checks would be introduced in the future draft FN-DSA standard [40, p. 5], in a way that rejects inputs to the one-dimensional sampler outside of its validity range $[\sigma_{\min}, \sigma_{\max}]$. Since that range is fairly narrow, it imposes relatively stringent limits on successful glitches, and as a result, we get somewhat lower success rates with that bound check in place, with variations across all three implementations. Detailed numbers are provided in Table 2.



Figure 4: Experimental setup. A ChipWhisperer Pro connected to a STM32F415RGT6 target board.

As is clear from the tabulated results, our success rates relatively strong in all cases, and as such, it is difficult to argue that NIST’s upcoming bound check would constitute an effective countermeasure. Interestingly, this appears to be especially true for the newest optimized assembly implementation, which allows for full key recovery at a rate of around two-thirds. Given that it would be preferred over any generic C implementation for embedded devices, the attack appears to offer a comfortable margin of success to an attacker targeting such a device.

5 FALCON’s Square Roots and Supply Chain Security

The fault attack of the previous section targets the computation of a square root by physical means. In this section, we describe a distinct but closely related risk: the same kind of distributional perturbation can also be introduced at the software-supply-chain level. The purpose of the discussion is not to claim that compromising a mathematical library is the easiest attack against a particular deployment. Rather, it is to point out that square roots create a narrow and somewhat unusual trust boundary in FALCON’s platform-native floating-point code: depending on the target architecture and compiler options, the operation may be implemented by a target instruction, by compiler-specific inline code, by architecture-specific assembly in FALCON itself, or by a call to the system mathematical library. This makes square roots more exposed than the other floating-point operations used by the scheme, and creates testing gaps between common development platforms and less common deployment targets.

Why sqrt is a special dependency. In the native floating-point implementation of FALCON, addition, subtraction, multiplication and division are written directly as operations on C double values. A square root, however, is normally exposed in C as the library function `sqrt`. On mainstream platforms, the situation is less simple than that description suggests. Modern compilers recognize `sqrt` as a builtin and often lower it directly to a target instruction or intrinsic. Moreover, the FALCON implementation itself contains several architecture-specific cases intended to avoid an actual dependency on `libm`: on x86 with SSE mathematics it uses SSE2 intrinsics; on 32-bit x86 it uses the x87 `fsqrt` instruction; on PowerPC it uses the `fsqrt` instruction through inline assembly; and on ARM/AArch64 with hardware floating-point it uses the corresponding square-root instruction. Only when none of those cases applies does the native implementation fall back to a call to `sqrt`.

This design is reasonable from an engineering standpoint, but it has an important security consequence. Known-answer tests and ordinary continuous integration on x86-64, 32-bit x86, PowerPC, or AArch64 may never exercise the `libm` square-root function used by FALCON, even if the source code contains a call to `sqrt` in the generic fallback path. Conversely, a less common platform with hardware floating-point support, but without one of FALCON’s recognized assembly or intrinsic implementations and without a compiler builtin expansion, may execute the generic `sqrt` call. In that case, the security of FALCON depends on the target platform’s mathematical library, or on whatever runtime routine the compiler and libc use to provide `sqrt`. This produces an architecture-skewed testing gap: the code path used by the victim target is not necessarily the code

path used by the platforms on which the implementation is most often tested.

Threat model. We consider an attacker who can introduce a very small error into the `sqrt` implementation used by a rare target platform, or equivalently into a target-specific mathematical runtime routine. This could model a malicious library update, a compromised downstream package, a target-specific patch, or a malicious proprietary runtime. The attacker’s goal is not to make `sqrt` visibly incorrect in general. Instead, the malicious implementation agrees with the correct square root on essentially all inputs, but returns a perturbed value on a very narrow set of positive inputs that occur during FALCON key expansion. Signatures generated after such an expansion then follow a biased distribution of exactly the type analyzed in Section 4, and can be used for PCA-based key recovery.

In contrast with the fault attack of the previous section, this supply-chain attack works just as well in both tree mode and dynamic mode. Indeed, the error introduced by the faulty `sqrt` implementation is consistent⁴ throughout successive runs of the algorithm, so it affects the same leaves of the normalized FALCON tree even across multiple executions of dynamic-mode signature generation.

A targeted perturbation. The relevant square-root inputs are the leaf values d_i of the LDL tree, and the first leaf is the most valuable target because perturbing it reveals the eigenspace generated by the secret key vector. In practice, the value $d_0 = \|(g, -f)\|^2$ is concentrated in a relatively small interval. The reason is that the coefficients of f and g are sampled from a narrow Gaussian distribution, so d_0 is well approximated by a scaled chi-square random variable with $2N$ degrees of freedom, conditioned on the FALCON key-generation norm bound. Thus the fraction of affected keys for any interval I can be estimated as

$$\Pr[d_0 \in I] \approx \frac{F_{\chi_{2N}^2}(\sup I/s^2) - F_{\chi_{2N}^2}(\inf I/s^2)}{F_{\chi_{2N}^2}((1.17^2 q)/s^2)},$$

where s^2 is the variance scale of the key-generation distribution, and the denominator accounts for rejection of keys above the FALCON norm bound. Equivalently, the same probability can be estimated directly from generated keys. In our experiments, triggering on keys with $d_0 < 2^{14} = 16384$ already affects a substantial fraction of valid keys; triggering on a wider interval around the empirically observed range of d_0 would affect more keys, at the cost of making the malicious behavior easier to notice.

The concrete perturbation used in our proof of concept is deliberately simple. The malicious `sqrt` first computes the correct result. It then checks, using only bitwise operations, whether the input is a positive binary64 value in the exponent bucket corresponding to $[2^{13}, 2^{14})$, with a small tolerance on the least significant mantissa bits to account for the sticky-bit convention used by FALCON’s emulated floating-point code. When the predicate holds, the implementation clears the highest stored mantissa bit of the output. This leaves the result finite and of the correct order of magnitude (the corresponding standard deviations even remains in the $[\sigma_{\min}, \sigma_{\max}]$

tested by NIST for provided that $d_0 \geq 14400$, which is satisfied with overwhelming probability), but changes it enough to perturb the corresponding FALCON-tree leaf and create a clear spectral bias in the signature distribution, that leads to key recovery in around a million signature samples again. We emphasize that the specific bit manipulation is not meant to be unique or optimal. It is merely a compact way to realize, in software, the same kind of leaf perturbation exploited by the physical fault attack. Other rare-input modifications of `sqrt` that move the first leaf into the exploitable interval would have the same effect.

Linker-wrapping proof of concept. To demonstrate the mechanism without modifying any system library, we use the GNU linker option `-Wl,--wrap=sqrt`. With this option, calls to `sqrt` that remain as link-visible external references are redirected to a wrapper function `__wrap_sqrt`, while `__real_sqrt` denotes the original function. Our wrapper calls the real `sqrt` and then applies the targeted branchless perturbation described above. This models a compromised `libm` implementation: if FALCON reaches the generic `sqrt` call, the perturbation is applied; if FALCON uses inline assembly, intrinsics, or a compiler builtin instead, the wrapper is bypassed.

This experiment is useful precisely because it separates two issues that are often conflated. First, it confirms that a very small and rare perturbation of square-root outputs can produce signatures from which our key-recovery attack is efficient, for the affected keys. In particular, we verified that when the wrapper is actually used, keys satisfying $d_0 < 16384$ yield signatures that are readily exploitable by the PCA attack. Second, it shows which build configurations would actually cross the library-call boundary. On the mainstream platforms we tested—x86-64, 32-bit x86, PowerPC, and AArch64—the default FALCON code does not call the wrapped function, even when compiled with `-fno-builtin-sqrt`, because the implementation takes one of its architecture-specific intrinsic or inline-assembly paths. Thus those platforms are not vulnerable to this particular `libm`-level proof of concept. They remain relevant as negative controls: the same source code and the same known-answer tests can look perfectly normal there while a different target takes the generic fallback.

The platforms of concern are therefore not the usual desktop and server architectures, but native-FPU targets for which FALCON does not provide an architecture-specific square-root implementation and for which the compiler does not lower `sqrt` to a target instruction before linking. On such a platform, the generic fallback calls the system `sqrt`, and a corrupted mathematical library would affect FALCON key expansion. This category may include less common or newer architectures, experimental ports, and embedded or vendor-specific Linux environments. It is also precisely the kind of platform for which testing coverage is most likely to be thinner than for x86-64 and AArch64.

Stealth and detectability. The attack does not require `sqrt` to be broadly inaccurate. A trigger restricted to a single exponent bucket, or even to a sparse subset of mantissas within that bucket, is enough to compromise a non-negligible fraction of keys while being unlikely to be hit by generic numerical tests. Moreover, the corrupted output remains a finite, positive, plausible-looking floating-point value. This is quite different from replacing `sqrt` by an obviously wrong implementation, and it is also different from a fault that

⁴Unless the faulty implementation is somehow non-deterministic, but it seems difficult to pass off probabilistic code as a plausible implementation for such a basic floating point routine.

causes crashes or verification failures. The resulting signatures remain valid; the problem is that their distribution is subtly biased across many signatures from the same expanded key.

There are nevertheless limits to the stealth argument. A deliberately input-triggered error of this kind would be conspicuous if directly exposed in audited source code, and a rigorous correctly-rounded sqrt test suite that happened to include the trigger values would detect it immediately. The realistic concern is instead target-specific code that receives much less review: generated tables, architecture-specific fast paths, downstream patches, soft-float routines, or proprietary mathematical runtimes. In those settings, the defect can be rare enough to survive ordinary testing, while still frequent enough for FALCON because the scheme repeatedly evaluates square roots on values drawn from a narrow and predictable key-dependent range.

Implications. The supply-chain attack is not the primary practical attack in this paper; the physical fault attack is more direct and experimentally validated on a concrete microcontroller target. However, the supply-chain scenario highlights an additional reason why square roots are undesirable in FALCON. They create a dependency boundary that is absent from the square-root-free variant of Section 3. Eliminating square roots removes not only a slow and delicate operation, but also this architecture-dependent reliance on library or runtime code whose behavior may not be exercised by tests on mainstream platforms. In that sense, the square-root-free rewriting is a defense-in-depth measure against both fault injection and rare-target supply-chain failures.

6 Related Work

A number of works have been devoted to analyzing and improving the FALCON implementation, both from an efficiency and a security standpoint.

The current reference implementation was obtained [42] by applying the Pornin–Prest NTRU equation solver [48] and Howe et al.’s isochronous Gaussian sampling technique [26] to the previous code base (interestingly, the latter was initially done incorrectly in a way that leaked secret key information [2]). Subsequently, improvements to the key generation algorithm [45, 51], the signature compression [15], the floating point arithmetic (better floating point emulation [46], extended precision floating point [23], and very recently fixed point arithmetic [6]), or the choice of the polynomial approximation for the exponential function [25] have been proposed, though it remains unclear which of those improvements will eventually make it into the FN-DSA standard. No previous work has suggested eliminating the square root computations, which remain a challenge even in the very recent fixed point implementation of [6].

From an implementation security standpoint, a long series of works has been devoted to the side-channel security of the FALCON signature algorithm. Fouque et al. [16] proposed a timing attack against an earlier, non-isochronous version of the FALCON Gaussian sampler, and proposed a key recovery using idiosyncratic techniques based on the Gentry–Szydło algorithm [19]. Later attacks exploited standard power consumption or electromagnetic analysis leakage, particularly on the one-dimensional Gaussian sampler, and

used statistical learning techniques to then recover the key: Guerreau et al. [21] used the Nguyen–Regev gradient descent technique, Zhang et al. [52] and Lin et al. [33] used the sample covariance matrix (essentially relying on PCA), and very recently, Brinkmann et al. [5] reduced the key recovery problem in certain leakage scenarios to the halfspace learning problem [4]. A cold boot attack has also been proposed against FALCON, with key recovery relying on very different, error-correction-based techniques [28].

While the literature on side-channel against FALCON is fairly extensive, far less attention has been devoted to fault attacks. The first fault injection attack on FALCON signing, due to McCarthy et al. [37], adapts the loop-abort fault framework of Espitau et al. [14] to FALCON. However, due to the recursive rather than iterative nature of FALCON signing, key recovery requires multiple carefully synchronized fault injections to succeed, and hence a particularly powerful adversary. While not a fault attack per se, an attack of a similar flavor was presented by Lin et al. [32], who showed how signing the same message twice with different floating point variants of the FALCON signing algorithm could lead to key recovery in the presence of repeated salts; such repeated salts should not normally occur with the standard FALCON algorithm, but can appear due to random generator misuse or when using deterministic variants of FALCON. These deterministic variants have also been attacked by Bauer and De Santis [3] with actual fault injection techniques. Finally, Abou Haidar et al. [22] demonstrated a key recovery attack on FALCON proper using a single Rowhammer bit flip on the table in memory describing the base half-Gaussian distribution used by the one-dimensional sampler; the bit flip introduces bias in the signature distribution, which is exploited using a combination of PCA used as a dimension reduction technique with the Nguyen–Regev attack on a low-dimensional eigenspace.

In comparison, the fault attack described in the present paper is arguably the most devastating against FALCON to date, combining three advantages: a single fault suffices, unlike the multiple synchronized fault required by [29]; it applies to the regular FALCON algorithm, whereas [3] is restricted to non-standard deterministic variants where the salt is derived deterministically from the secret key and the message; and it recovers the full key from far fewer signatures (up to a few million) than an attack like [22], which needs tens to hundreds of millions of signatures from a single expanded key. Compared to [22], key recovery in our attack is also simpler, as the final key search space collapses to complex dimension 1, allowing recovery by straightforward enumeration rather than more elaborate gradient descent techniques.

7 Countermeasures

As we have seen, the square root algorithm is sensitive both to fault attacks and to certain supply chain attack scenario, and the planned NIST bound check on standard deviations in the one-dimensional sampler are insufficient to thwart either of those attacks. Moreover, while the fault attack requires the use of FALCON in tree mode (i.e., with a cached expanded key) to be effective (in the sense that it can be carried out with a single fault on key expansion, rather than individual faults on each signature), this is not the case for the supply chain scenario, where the square roots in dynamic signing are equally vulnerable. Moreover, in contrast with previous attacks

like [22], faulty signatures in our setting look very close to regular ones, until the point when they can be distinguished by statistical means (and that point is too late, since the attack becomes feasible!).

Our proposed square root-free implementation is, of course, a natural countermeasure. It removes the libm dependency issue, so eliminates that problem. Regarding the fault attack, it removes the specific target we described, but other steps could remain vulnerable: for example, the “normalization step” of the leaves that we attack still consists of a multiplication in our square root-free algorithm. Could it still be a fault target? We conducted experiments showing that this is very unlikely: the multiplication, as a simpler algorithm, has much fewer possible fault outcomes. We could only trigger fault where the leaf value remained basically unchanged, or where it became considerably larger, which turns signing into an infinite loop. So again, our approach seems to be an effective countermeasure.

Ethical Considerations

This work analyzes implementation-level attack surfaces in Falcon, which has been selected by NIST for future standardization as FN-DSA. Cryptanalytic results of this kind can raise dual-use concerns, since detailed attack descriptions may in principle help adversaries target vulnerable deployments. In the present case, we assess that risk as low. At the time of writing, the draft FN-DSA standard has not yet appeared, and we are not aware of substantial real-world deployments of Falcon that would be directly endangered by our results. In addition, the paper proposes a simple defensive change (eliminating square-root computations from the signing algorithm) which removes the attack surface studied here without changing the intended distribution of signatures or weakening the scheme. Finally, we have communicated with members of the Falcon team prior to publication, giving the parties most directly involved in standardization and implementation an opportunity to take the issue into account. We therefore believe that publication would rather provide useful guidance for robust FN-DSA implementations before large-scale deployment begins.

Open Science

Artifacts, available on Zenodo at <https://doi.org/10.5281/zenodo.12689188>, include the square root free implementation, the fault attack code and the supply chain attack proof of concept.

References

- [1] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS* (Providence, RI, USA) (ASPLOS '19). ACM, New York, NY, USA, 673–686. <https://doi.org/10.1145/3297858.3304062>
- [2] Calvin Abou Haidar and Mehdi Tibouchi. 2025. How Bad Was the Falcon Bug of 2019?. In *QRSEC 2025*, Ashish Kundu and Attila A. Yavuz (Eds.). ACM, 58–62. <https://doi.org/10.1145/3733820.3764935>
- [3] Sven Bauer and Fabrizio De Santis. 2023. A differential fault attack against deterministic Falcon signatures. In *CARDIS*, Shivam Bhasin and Thomas Roche (Eds.). Springer, 43–61.
- [4] Avrim Blum, Alan M. Frieze, Ravi Kannan, and Santosh Vempala. 1996. A Polynomial-Time Algorithm for Learning Noisy Linear Threshold Functions. In *37th FOCS*. IEEE Computer Society Press, 330–338. <https://doi.org/10.1109/SFCS.1996.548492>
- [5] Marcus Brinkmann, Nicolai Kraus, and Alexander May. 2026. Halfspace Learning for Lattice Signature Key Recovery from Signs. In *CRYPTO* (LNCS, Vol. 16802), Nadia Heninger and Mark Rosulek (Eds.). Springer, Cham, 359–390. https://doi.org/10.1007/978-3-032-35377-1_12
- [6] Bachir Lachguel Daniel De Almeida Braga, Pierre-Alain Fouque and Thomas Prest. 2026. Toward a Secure Fixed-Point Implementation of the Falcon Signature Scheme. In *CRYPTO* (LNCS, Vol. 16802), Nadia Heninger and Mark Rosulek (Eds.). Springer, Cham, 166–197. https://doi.org/10.1007/978-3-032-35377-1_6
- [7] Chandler Davis and W. M. Kahan. 1970. The Rotation of Eigenvectors by a Perturbation. III. *SIAM J. Numer. Anal.* 7, 1 (1970), 1–46.
- [8] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. 2013. Lattice Signatures and Bimodal Gaussians. In *CRYPTO 2013, Part I* (LNCS, Vol. 8042), Ran Canetti and Juan A. Garay (Eds.). Springer, Berlin, Heidelberg, 40–56. https://doi.org/10.1007/978-3-642-40041-4_3
- [9] Léo Ducas, Steven Galbraith, Thomas Prest, and Yang Yu. 2020. Integral Matrix Gram Root and Lattice Gaussian Sampling Without Floats. In *EUROCRYPT 2020, Part II* (LNCS, Vol. 12106), Anne Canteaut and Yuval Ishai (Eds.). Springer, Cham, 608–637. https://doi.org/10.1007/978-3-030-45724-2_21
- [10] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. 2014. Efficient Identity-Based Encryption over NTRU Lattices. In *ASIACRYPT 2014, Part II* (LNCS, Vol. 8874), Palash Sarkar and Tetsu Iwata (Eds.). Springer, Berlin, Heidelberg, 22–41. https://doi.org/10.1007/978-3-662-45608-8_2
- [11] Léo Ducas and Phong Q. Nguyen. 2012. Learning a Zonotope and More: Cryptanalysis of NTRUSign Countermeasures. In *ASIACRYPT 2012* (LNCS, Vol. 7658), Xiaoyun Wang and Kazuo Sako (Eds.). Springer, Berlin, Heidelberg, 433–450. https://doi.org/10.1007/978-3-642-34961-4_27
- [12] Léo Ducas and Thomas Prest. 2016. Fast Fourier Orthogonalization. In *ISSAC 2016*, Sergei A. Abramov, Eugene V. Zima, and Xiao-Shan Gao (Eds.). ACM, 191–198.
- [13] Léo Ducas and Yang Yu. 2021. Learning Strikes Again: The Case of the DRS Signature Scheme. *Journal of Cryptology* 34, 1 (Jan. 2021), 1. <https://doi.org/10.1007/s00145-020-09366-9>
- [14] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. 2016. Loop-Abort Faults on Lattice-Based Fiat-Shamir and Hash-and-Sign Signatures. In *SAC 2016* (LNCS, Vol. 10532), Roberto Avanzi and Howard M. Heys (Eds.). Springer, Cham, 140–158. https://doi.org/10.1007/978-3-319-69453-5_8
- [15] Thomas Espitau, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. 2022. Shorter Hash-and-Sign Lattice-Based Signatures. In *CRYPTO 2022, Part II* (LNCS, Vol. 13508), Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, Cham, 245–275. https://doi.org/10.1007/978-3-031-15979-4_9
- [16] Pierre-Alain Fouque, Paul Kirchner, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. 2020. Key Recovery from Gram-Schmidt Norm Leakage in Hash-and-Sign Signatures over NTRU Lattices. In *EUROCRYPT 2020, Part III* (LNCS, Vol. 12107), Anne Canteaut and Yuval Ishai (Eds.). Springer, Cham, 34–63. https://doi.org/10.1007/978-3-030-45727-3_2
- [17] Alan M. Frieze, Mark Jerrum, and Ravi Kannan. 1996. Learning Linear Transformations. In *37th FOCS*. IEEE Computer Society Press, 359–368. <https://doi.org/10.1109/SFCS.1996.548495>
- [18] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. 2008. Trapdoors for hard lattices and new cryptographic constructions. In *40th ACM STOC*, Richard E. Ladner and Cynthia Dwork (Eds.). ACM Press, 197–206. <https://doi.org/10.1145/1374376.1374407>
- [19] Craig Gentry and Michael Szydlo. 2002. Cryptanalysis of the Revised NTRU Signature Scheme. In *EUROCRYPT 2002* (LNCS, Vol. 2332), Lars R. Knudsen (Ed.). Springer, Berlin, Heidelberg, 299–320. https://doi.org/10.1007/3-540-46035-7_20
- [20] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. 1997. Public-Key Cryptosystems from Lattice Reduction Problems. In *CRYPTO '97* (LNCS, Vol. 1294), Burton S. Kaliski, Jr. (Ed.). Springer, Berlin, Heidelberg, 112–131. <https://doi.org/10.1007/BFb0052231>
- [21] Morgane Guereau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. 2022. The Hidden Parallelepiped Is Back Again: Power Analysis Attacks on Falcon. *IACR TCHES* 2022, 3 (2022), 141–164. <https://doi.org/10.46586/tches.v2022.i3.141-164>
- [22] Calvin Abou Haidar, Quentin Payet, and Mehdi Tibouchi. 2025. Crowhammer: Full Key Recovery Attack on Falcon with a Single Rowhammer Bit Flip. In *CRYPTO 2025, Part V* (LNCS, Vol. 16004), Yael Tauman Kalai and Seny F. Kamara (Eds.). Springer, Cham, 103–135. https://doi.org/10.1007/978-3-032-01901-1_4
- [23] Stef Halmarks, Christine van Vredendaal, Tobias Schneider, Frank Custers, and Tim Güneysu. 2026. TWFalcon: Triple-Word Arithmetic for Falcon. *IACR TCHES* 2026, 2 (2026), 347–371. <https://doi.org/10.46586/tches.v2026.i2.347-371>
- [24] Jeffrey Hoffstein, Nick Howgrave-Graham, Jill Pipher, Joseph H. Silverman, and William Whyte. 2003. NTRUSIGN: Digital Signatures Using the NTRU Lattice. In *CT-RSA 2003* (LNCS, Vol. 2612), Marc Joye (Ed.). Springer, Berlin, Heidelberg, 122–140. https://doi.org/10.1007/3-540-36563-X_9
- [25] Nicolas Houles and Thibaut Heckmann. 2026. Algorithmic Optimization of the Gaussian Sampler in the FN-DSA Post-Quantum Signature Scheme. *Cryptology ePrint Archive*, Report 2026/1610. <https://eprint.iacr.org/2026/1610>
- [26] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. 2020. Isochronous Gaussian Sampling: From Inception to Implementation. In *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, Jintai Ding and Jean-Pierre Tillich (Eds.). Springer, Cham, 53–71. https://doi.org/10.1007/978-3-030-44223-1_4
- [27] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange,

- Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. 2022. *SPHINCS*. Technical Report. National Institute of Standards and Technology. available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022>.
- [28] Hiroto Kaihara, Mehdi Tibouchi, and Masayuki Abe. 2026. FrozeNTRU: Cold Boot Attacks on NTRU-Based Hash-and-Sign Signatures. *IACR TCHES* 2026, 3 (2026), 1007–1031. <https://doi.org/10.46586/tches.v2026.i3.1007-1031>
- [29] Emre Karabulut and Aydin Aysu. 2021. Falcon Down: Breaking Falcon Post-Quantum Signature Scheme through Side-Channel Attacks. In *DAC* 2021.
- [30] Philip N. Klein. 2000. Finding the closest lattice vector when it's unusually close. In *11th SODA*, David B. Shmoys (Ed.). ACM-SIAM, 937–941.
- [31] Xiuhuan Lin, Moeto Suzuki, Shiduo Zhang, Thomas Espitau, Yang Yu, Mehdi Tibouchi, and Masayuki Abe. 2024. Cryptanalysis of the Peregrine Lattice-Based Signature Scheme. In *PKC 2024, Part I (LNCS, Vol. 14601)*, Qiang Tang and Vanessa Teague (Eds.). Springer, Cham, 387–412. https://doi.org/10.1007/978-3-031-57718-5_13
- [32] Xiuhuan Lin, Mehdi Tibouchi, Yang Yu, and Shiduo Zhang. 2025. Do Not Disturb a Sleeping Falcon - Floating-Point Error Sensitivity of the Falcon Sampler and Its Consequences. In *EUROCRYPT 2025, Part II (LNCS, Vol. 15602)*, Serge Fehr and Pierre-Alain Fouque (Eds.). Springer, Cham, 213–244. https://doi.org/10.1007/978-3-031-91124-8_8
- [33] Xiuhuan Lin, Shiduo Zhang, Yang Yu, Weijia Wang, Qidi You, Ximing Xu, and Xiaoyun Wang. 2025. Thorough Power Analysis on Falcon Gaussian Samplers and Practical Countermeasure. In *PKC 2025, Part I (LNCS, Vol. 15674)*, Tibor Jager and Jiaxin Pan (Eds.). Springer, Cham, 229–258. https://doi.org/10.1007/978-3-031-91820-9_8
- [34] Vadim Lyubashevsky. 2009. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In *ASIACRYPT 2009 (LNCS, Vol. 5912)*, Mitsuru Matsui (Ed.). Springer, Berlin, Heidelberg, 598–616. https://doi.org/10.1007/978-3-642-10366-7_35
- [35] Vadim Lyubashevsky. 2012. Lattice Signatures without Trapdoors. In *EUROCRYPT 2012 (LNCS, Vol. 7237)*, David Pointcheval and Thomas Johansson (Eds.). Springer, Berlin, Heidelberg, 738–755. https://doi.org/10.1007/978-3-642-29011-4_43
- [36] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. 2022. *CRYSTALS-DILITHIUM*. Technical Report. National Institute of Standards and Technology. available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022>.
- [37] Sarah McCarthy, James Howe, Neil Smyth, Séamus Brannigan, and Máire O'Neill. 2019. BEARZ Attack FALCON: Implementation Attacks with Countermeasures on the FALCON Signature Scheme. In *SECURITY*, Mohammad S. Obaidat and Pierangela Samarati (Eds.). SciTePress, 61–71. <https://doi.org/10.5220/0007834800610071>
- [38] Phong Q. Nguyen and Oded Regev. 2006. Learning a Parallelepiped: Cryptanalysis of GGH and NTRU Signatures. In *EUROCRYPT 2006 (LNCS, Vol. 4004)*, Serge Vaudenay (Ed.). Springer, Berlin, Heidelberg, 271–288. https://doi.org/10.1007/11761679_17
- [39] Phong Q. Nguyen and Oded Regev. 2009. Learning a Parallelepiped: Cryptanalysis of GGH and NTRU Signatures. *Journal of Cryptology* 22, 2 (April 2009), 139–160. <https://doi.org/10.1007/s00145-008-9031-0>
- [40] Ray Perlner. 2025. *FIPS 206: FN-DSA (Falcon) Status Update*. Technical Report. National Institute of Standards and Technology. Presented at the 6th PQC Standardization Conference. Available at <https://csrc.nist.gov/Presentations/2025/fips-206-fn-dsa-falcon>.
- [41] Thomas Pornin. 2010. Falcon reference implementation. <https://falcon-sign.info/Falcon-impl-20211101.zip>.
- [42] Thomas Pornin. 2019. New Efficient, Constant-Time Implementations of Falcon. Cryptology ePrint Archive, Report 2019/893. <https://eprint.iacr.org/2019/893>
- [43] Thomas Pornin. 2019. OFFICIAL COMMENT: Falcon (bug & fixes). pqc-forum official comment. <https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/7Z8x5AMXy8s/m/Spyv8VYyBQAJ>
- [44] Thomas Pornin. 2019. OFFICIAL COMMENT: Updated Falcon implementation. pqc-forum official comment. <https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/RuDwNXqO-20/m/kuDJQzoBCQAJ>
- [45] Thomas Pornin. 2023. Improved Key Pair Generation for Falcon, BAT and Hawk. Cryptology ePrint Archive, Report 2023/290. <https://eprint.iacr.org/2023/290>
- [46] Thomas Pornin. 2025. Falcon on ARM Cortex-M4: an Update. Cryptology ePrint Archive, Report 2025/123. <https://eprint.iacr.org/2025/123>
- [47] Thomas Pornin. 2025. Query on progress of FN-DSA. pqc-forum official comment. <https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/CsWPP35LJWw/m/75g1EhKiAQAJ>
- [48] Thomas Pornin and Thomas Prest. 2019. More Efficient Algorithms for the NTRU Key Generation Using the Field Norm. In *PKC 2019, Part II (LNCS, Vol. 11443)*, Dongdai Lin and Kazuo Sako (Eds.). Springer, Cham, 504–533. https://doi.org/10.1007/978-3-030-17259-6_17
- [49] Eamonn W. Postlethwaite and Wessel P. J. van Woerden. 2023. OFFICIAL COMMENT: EHTv3. pqc-forum official comment. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/mFL_5Rq6-RU
- [50] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2022. *FALCON*. Technical Report. National Institute of Standards and Technology. available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022>.
- [51] Chao Sun, Thomas Espitau, Junjie Song, Jinguang Han, and Mehdi Tibouchi. 2026. Generating Falcon Trapdoors via Gibbs Sampler. In *PQCrypto (LNCS, Vol. 16491)*, Magali Bardet and Ruben Niederhagen (Eds.). Springer, Cham, 225–251. https://doi.org/10.1007/978-3-032-22695-2_8
- [52] Shiduo Zhang, Xiuhuan Lin, Yang Yu, and Weijia Wang. 2023. Improved Power Analysis Attacks on Falcon. In *EUROCRYPT 2023, Part IV (LNCS, Vol. 14007)*, Carmit Hazay and Martijn Stam (Eds.). Springer, Cham, 565–595. https://doi.org/10.1007/978-3-031-30634-1_19