IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–25. DOI:XXXXXXXX

One Fell Swoop: A Single-Trace Key-Recovery Attack on the Falcon Signing Algorithm

Kang Li[1], Shouran Ma[2], Haochen Dou[2] and Qian Guo[2*]

[1] Jinan University, Guangzhou, China
kanglee175@163.com
[2] Lund University, Lund, Sweden
shouran.ma@eit.lth.se, haochen.dou@eit.lth.se, qian.guo@eit.lth.se

Abstract. Falcon, a lattice-based signature scheme selected for NIST post-quantum standardization, is notable for its compact signature size alongside a complex signing procedure involving extensive floating-point arithmetic. Prior side-channel attacks on Falcon signing, while demonstrating vulnerabilities, have consistently required a large number of power traces for successful key recovery, which limits their real-world practicality. This paper presents a new single-trace attack on the Falcon signing procedure. We exploit side-channel leakage in the FP conversion and identify additional leakage in the FP multiplication at the first layer of the Fast Fourier Transform (FFT) during secret key expansion. By combining these leakages, we progressively partition the secret-key coefficients and establish a linear system to recover the entire key. Our attack is particularly devastating for the `sign_dyn` design—the memory-efficient implementation widely adopted in cryptographic libraries—which executes key expansion during every signature operation. We validate our attack on an ARM Cortex-M4 microcontroller, achieving a 100% key recovery success rate with just one power trace for Falcon-512 across the baseline `-O0`, size-optimized `-Os`, and highest `-O3` compilation levels of the PQClean implementation. Crucially, we also evaluate the Falcon team's highly optimized embedded assembly (ASM) implementation. Despite its reduced leakage, our adapted multi-trace attack successfully recovers 53 out of 100 test keys with only 10 traces. This marks the first successful single-trace attack at the `-O3` level and the first practical attack against the embedded ASM Falcon. Furthermore, our in-depth assembly analysis of these vulnerabilities provides critical guidelines for designing more secure, hardened embedded assembly implementations.

Keywords: NIST post-quantum cryptography standardization · Lattice-based cryptography · Side-channel attacks · Falcon

1 Introduction

Falcon [PFH+22] is a lattice-based post-quantum signature scheme built on the NTRU lattice and has been selected by NIST as one of the three signatures for standardization. Compared with the other two signatures (Dilithium [LDK+22], SPHINCS+ [HBD+22]), Falcon achieves the smallest combined size of public key and signature, making it especially attractive for embedded platforms. It also offers efficient signing and verification procedures.

As a product of many years of work, Falcon's theoretical foundations have undergone extensive scrutiny. However, as Falcon relies on the GPV hash-and-sign framework [GPV08] and is instantiated through fast Fourier sampling [DP16], the signing procedure becomes inherently complex and involves extensive floating-point (FP) operations. These

*Corresponding author.

Licensed under Creative Commons License CC-BY 4.0.

characteristics expose Falcon to side-channel attacks. Fouque et al. [FKT+20] first proposed a timing attack against the signing procedure, which was later addressed by the introduction of the isochronous Gaussian sampler [HPRR20]. Building upon this sampler, Pornin developed a constant-time implementation of Falcon [Por19]. Yet, even in its constant-time form, Falcon remains vulnerable to power or electromagnetic leakage.

Previously, side-channel evaluations of Falcon's signing procedure have primarily targeted two operations: pre-image computation and integer Gaussian sampling. Regarding the pre-image computation, Karabulut et al. [KA21] first exploited FP multiplications in the Fourier domain to launch a correlation power analysis (CPA) for key recovery targeting Falcon-512, and the CPA attack requires 9,000 – 10,000 electromagnetic traces to succeed. Later, Guerreau et al. [GMRR22] improved their method by significantly reducing the enumeration space of the mantissa multiplication. Their experiments show that 5,000 power traces are needed to recover the secret key. Chen et al. [CSM+25] optimized the attack and proposed the SERP-SCA attack framework, which achieves notable improvements in both success rate and efficiency. The required number of power traces is reduced to about 3,000. Most recently, Chen et al. [CCCY26] presented an improved CPA attack on the masked FP multiplication in the pre-image computation. The secret key can be recovered using 1,300 and 1,200 traces for two different simulated masking strategies, which demonstrates that only protecting the FP multiplication to defend against the CPA attack is not enough.

In the context of integer Gaussian sampling, Guerreau et al. [GMRR22] proposed the first attack targeting the so-called half Gaussian leakage. Their approach filters key-dependent signatures based on this leakage and recovers the secret key by adapting the hidden parallelepiped attack [NR06, DN12]. However, to maintain a practical post-processing time, this attack demands at least 1,000,000 power traces. Zhang et al. [ZLYW23] improved upon this by applying spectral decomposition to the covariance matrix of the filtered signatures to learn the secret key. By combining this improved method with both the half Gaussian leakage and a newly identified vulnerability termed sign leakage, they reduced the trace requirement to 45,000. Then, Lin et al. [LZY+25] further optimized the attack by exploiting the symmetry of the NTRU lattice structure alongside a more advanced decoding technique, bringing the minimum trace requirement down to just 6,500. Despite these significant improvements in trace efficiency, it is notable that the success rates of attacks in [ZLYW23] and [LZY+25] are approximately 25%.

Motivation As summarized in Table 1, the vast majority of prior side-channel analyses of Falcon require thousands of power or electromagnetic traces to succeed. While these studies demonstrate theoretical vulnerabilities, the requirement for such extensive data collection often renders them impractical against real-world targets where access may be limited or transient. In contrast, an attack requiring only a **single trace** represents the most severe threat model, as it can succeed with minimal measurements.

While single-trace attacks have been explored for various lattice-based schemes [PPM17, PP19a, KH18, EFGT17, QLZ+24, ACLZ20, KAA21], research on Falcon in this rigorous setting remains relatively scarce. Qiu et al. [QA25] proposed the first single-trace attack targeting the discrete Gaussian sampling process during the *key generation* phase. However, there are a few limitations: first, key generation is typically a "one-off" event performed in a controlled environment, making it difficult for an attacker to capture the specific trace; second, they develop their attack under the unoptimized `-O0` compilation level, but that level is rarely used in practice. Concurrent with our work, Kim et al. [KLL+26] exploit FP-operation leakage during key expansion, while Qiu et al. [QA26] exploit leakage in the Number Theoretic Transform (NTT). However, Kim et al.'s single-trace attack does not achieve reliable key recovery under higher optimization levels, whereas Qiu et al.'s practical attack restricts the secret-key coefficient range. We discuss these works and their

Table 1: Comparison of the number of traces required for key-recovery attacks on Falcon-512. Numbers with a superscript * indicate a trade-off between the number of traces and post-processing computation. We present numbers corresponding to a reasonable post-processing time. The compilation level is shown if it is reported in the corresponding work. Except for work [KA21], which collects electromagnetic traces, all other works collect power traces. Abbreviations: PIC = Pre-image computation, HGL = Half Gaussian leakage, SL = Sign leakage, DGS = Discrete Gaussian sampling, FPM = FP multiplication, FPCA = FP conversion and addition, FPCM = FP conversion and multiplication.

Work	Target	#Trace	Phase
[KA21] (`-O0`)	PIC	9,000 – 10,000	signing
[GMRR22]	PIC	5,000	signing
	HGL	1,000,000*	
[ZLYW23]	HGL	220,000*	signing
	SL	170,000*	
	HGL & SL	45,000*	
[LZY+25]	HGL	27,500*	signing
	SL	25,000*	
	HGL & SL	6,500*	
[CSM+25]	PIC	3,000	signing
[CCCY26] (`-Os`)	masked FPM in PIC	1,300 or 1,200	signing
[QA25] (`-O0`)	DGS	1	keygen
[QA26]	NTT	1	keygen & signing
[KLL+26](`-O0`,`-O1`,`-Os`)	FPCA	1	keygen & signing
This Work (`-O0`, `-Os`, `-O3`)	**FPCM**	**1**	**keygen & signing**
This Work (ASM Falcon)	**FPCM**	**10***	**signing**

relation to ours in detail below.

This paper addresses the critical need for a single-trace attack on the *signing* procedure even under the optimized compilation levels. Our motivation stems from the distinct implementation choices available in Falcon:

- `sign_tree`: A design optimized for **speed**, which precomputes and stores the expanded key.
- `sign_dyn`: A design optimized for **memory efficiency**, which recomputes the expanded key on-the-fly during every signature generation.

Because the `sign_dyn` design allows Falcon to fit on memory-constrained devices (where side-channel attacks are most relevant), it is the standard choice for embedded libraries. By identifying leakages in the FP conversions and FP multiplications at the beginning of the Fast Fourier Transform (FFT) that occur during this on-the-fly expansion, we turn the memory-saving feature of `sign_dyn` into a security liability. Unlike the `sign_tree` design, which happens once, `sign_dyn` performs these vulnerable expansion operations for *every* signature. This allows us to mount a single-trace attack on the signing process itself.

Contributions Our contributions are as follows:

- We identify side-channel leakage in the FP conversion, which is also independently identified by Kim et al. [KLL+26], and identify novel leakage in the FP multiplication. These leakages reveal the signs and value ranges of secret-key coefficients, enabling their candidate sets to be progressively refined and establishing a new attack path independent of previous attacks.
- We propose a single-trace attack that links FP leakage to complete key recovery through coefficient classification. Guided by this methodology, we adapt the profiling

and classification stages to the leakage changes introduced by compiler optimizations, enabling the attack to remain effective across different optimization levels.

- We validate the attack on the Falcon-512 PQClean reference implementation running on an ARM Cortex-M4. For both `sign_tree` and `sign_dyn`, we achieve 100% key recovery with a single power trace under `-O0`, `-Os`, and `-O3`. To the best of our knowledge, this is the first successful single-trace attack against Falcon compiled with `-O3`. This result significantly improves the state of the art, as shown in Table 1.
- We perform an in-depth assembly-level security analysis across `-O0`, `-Os`, `-O3`, and an embedded assembly implementation provided by the Falcon team[1], which we call *ASM Falcon*. Although ASM Falcon prevents our single-trace attack, repeated key expansion under `sign_dyn` enables recovery of 53 out of 100 Falcon-512 keys using 10 traces. Crucially, this stands as the first practical full key recovery on ASM Falcon. By comparing the assembly implementations and their leakage characteristics, we explain how compiler optimizations and human-audited assembly reshape, suppress, or preserve exploitable leakage, and provide concrete guidance for designing more leakage-resilient Falcon implementations.

The source code is available at https://github.com/OChicken/tches27_1_fpr_attack.

Independent and Concurrent Work Concurrently and independently, Kim et al. [KLL+26] presented a single-trace SPA attack on Falcon based on leakage from branchless conditional selection during FP operations. In the FP conversion, both works exploit leakage from the `fpr_scaled` function, including sign extraction, zero detection, and the conditional normalization implemented by `FPR_NORM64`; we therefore do not claim exclusive novelty for these leakage sources. Beyond this shared starting point, the attacks exploit different additional leakage and follow different recovery pipelines. Kim et al. further exploit `fpr_add` throughout the FFT, where the leakage reveals magnitude comparisons, sign matching, and exponent increments, and organize these observations according to the FFT structure to propagate and prune candidate intermediate states layer by layer. In contrast, in the FP multiplication, we additionally exploit `fpr_mul` at the first FFT layer, where coefficient-dependent mantissa multiplication and normalization further refine the magnitude ranges of the key coefficients. Rather than reconstructing and propagating FFT states, our attack determines sufficiently many coefficients of the secret keys and then recovers the remaining coefficients by solving a linear system derived from the public-key relation. We further adapt the leakage-extraction stage to compiler-induced instruction transformations and ASM Falcon: our attack achieves a single-trace attack under `-O3` for the reference implementation, whereas Kim et al. report that their leakage recovery degrades under `-O2/-O3` and is insufficient for reliable key recovery, and we additionally evaluate ASM Falcon.

Qiu et al. [QA26] proposed another single-trace attack targeting leakage from modular reduction in the NTT. This NTT appears both during public-key computation in key generation and when the signing algorithm is invoked through an external interface. However, their practical attack restricts each secret-key coefficient to $[-10, 10]$, reducing the actual key space and limiting the realism of the evaluated setting.

Organizations Section 2 introduces the threat model and Falcon signature background. Section 3 explores the identified leakages and details our attack methodology, which is then adapted to different compilation levels and ASM Falcon in Section 4. Section 5 presents our experimental validation. Section 6 discusses implications and countermeasures, followed by conclusions in Section 7.

[1] https://falcon-sign.info/

2 Preliminary

Notations. We use bold uppercase letters for matrices and bold lowercase letters for vectors. A complex number is represented as $a + i \cdot b$, where a is the real part and $i \cdot b$ is the imaginary part. For a complex number z, $\bar{z}$ denotes its conjugate. For a real number a, $|a|$ denotes its absolute value. Define the polynomial ring $\mathcal{R} := \mathbb{Z}[x]/(x^n + 1)$. For a polynomial $P \in \mathcal{R}$, it can be expressed either as $P(x) = P_0 + P_1 x + \cdots + P_{n-1}x^{n-1}$, or equivalently, as a vector of coefficients $(P_0, P_1, \ldots, P_{n-1})$. The Hermitian adjoint of P is denoted as $P^\star$. Let $\rho_{\sigma,c}(v) = \exp\left(-\frac{(v-c)^2}{2\sigma^2}\right)$ denote the Gaussian function over $\mathbb{R}$, and the parameter c is omitted when $c = 0$. Based on this, the discrete Gaussian distribution over an integer set $\mathbb{S}_1$ is defined as $D_{\mathbb{S}_1,\sigma,c}(v) = \frac{\rho_{\sigma,c}(v)}{\sum_{u\in\mathbb{S}_1}\rho_{\sigma,c}(u)}$ with $v \in \mathbb{S}_1$. For an integer set $\mathbb{S}_2 \subseteq \mathbb{S}_1$, the probability mass assigned to $\mathbb{S}_2$ under the discrete Gaussian distribution is defined as $D_{\mathbb{S}_1,\sigma,c}(\mathbb{S}_2) = \sum_{v\in\mathbb{S}_2} D_{\mathbb{S}_1,\sigma,c}(v)$.

2.1 The Falcon Signature Scheme

In this subsection, we provide a brief introduction to the Falcon signature scheme. Some parts are omitted as they are not relevant to our work; for a comprehensive description, we refer the reader to the NIST submission of Falcon [PFH+22].

Let $q \in \mathbb{N}^*$ and let n be a power of two. In Falcon, the modulus is fixed to $q = 12289$, while n is set to 512 or 1024. These choices define Falcon-512 and Falcon-1024, which target NIST security levels I and V, respectively. Here, $\bar{\sigma}$ denotes the standard deviation of the discrete Gaussian distribution used during signature generation and controls the spread of the resulting signature, while $\lfloor\beta^2\rfloor$ denotes the maximum allowed squared norm of a valid signature. The main parameters are summarized in Table 2.

Table 2: Selected parameters of Falcon-512 and Falcon-1024.

Parameter set	**NIST level**	n	$\bar{\sigma}$	$\lfloor\beta^2\rfloor$
Falcon-512	I	512	165.736 617 183	34 034 726
Falcon-1024	V	1024	168.388 571 447	70 265 242

The overall process for generating the key pair of Falcon is shown in Alg. 1. During key generation, the coefficients of f and g ($f, g \in \mathcal{R}$) are sampled from a discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ centered at 0 with standard deviation $\sigma = 1.17\sqrt{q/2n}$ and then restricted to small integer ranges, namely $[-31, 31]$ for $n = 512$ and $[-15, 15]$ for $n = 1024$. Subsequently, using the relation $fG - gF = q$, together with the algorithm proposed in [PP19b], the secret keys F and G can be computed efficiently. These operations are implemented in the `NTRUGen` subroutine. After generating these original keys and forming the secret key matrix $\mathbf{B}$, key expansion is performed. Specifically, $\mathbf{B}$ is transformed into its FFT representation $\hat{\mathbf{B}}$, based on which the Falcon Tree $\mathbf{T}$ is constructed using the LDL* decomposition. Finally, Alg. 1 outputs the expanded secret key $\mathbf{sk}$, which consists of $\hat{\mathbf{B}}$ and $\mathbf{T}$, and the public key $\mathbf{pk}$.

The signing procedure of Falcon is illustrated in Alg. 2. It takes as input a message $\mathbf{m}$, the expanded secret key $\mathbf{sk}$, and the parameter $\lfloor\beta^2\rfloor$. The algorithm begins by sampling a salt $\mathbf{r}$ uniformly at random and computing the hash c. After transforming c into the FFT domain, it is multiplied by the inverse of the secret key $\mathbf{B}$ in the FFT representation to obtain the preimage $\mathbf{t}$. Next, the subroutine `ffsampling` is invoked to execute fast Fourier sampling to produce the vector $\mathbf{z}$. Then, a short vector $\mathbf{s} = (s_1, s_2)$ is calculated such that $s_1 + s_2 h = c \pmod q$. Finally, if $\mathbf{s}$ is short enough, it is encoded and output together with $\mathbf{r}$ as the final signature; otherwise, it repeats the fast Fourier sampling to produce the short vector $\mathbf{s}$.

Algorithm 1 Keygen(ϕ, q) [PFH+22]

Input: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus q
Output: A secret key $\mathbf{sk}$, a public key $\mathbf{pk}$

$(f, g, F, G) \leftarrow \mathsf{NTRUGen}(\phi, q)$ ▷ Solving the NTRU equation
$\mathbf{B} \leftarrow \begin{bmatrix} g & -f \\ G & -F \end{bmatrix}$
$\hat{\mathbf{B}} \leftarrow \mathsf{FFT}(\mathbf{B})$ ▷ Compute the FFT for each of the 4 components $\{g, -f, G, -F\}$
$\mathbf{G} \leftarrow \hat{\mathbf{B}} \times \hat{\mathbf{B}}^{\star}$
$\mathbf{T} \leftarrow \mathsf{ffLDL}^{\star}(\mathbf{G})$ ▷ Computing the LDL* tree
for each leaf ℓ of $\mathbf{T}$ **do**
    $\ell.\text{value} \leftarrow \bar{\sigma}/\sqrt{\ell.\text{value}}$ ▷ Normalization step
$\mathbf{sk} \leftarrow (\hat{\mathbf{B}}, \mathbf{T})$
$h \leftarrow gf^{-1} \bmod q$
$\mathbf{pk} \leftarrow h$
return $(\mathbf{sk}, \mathbf{pk})$

Algorithm 2 Sign($\mathbf{m}$, $\mathbf{sk}$, $\lfloor \beta^2 \rfloor$) [PFH+22]

Input: A message $\mathbf{m}$, a secret key $\mathbf{sk}$, a bound $\lfloor \beta^2 \rfloor$
Output: A signature of $\mathbf{m}$

$\mathbf{r} \leftarrow \{0,1\}^{320}$ uniformly
$c \leftarrow \mathsf{HashToPoint}(\mathbf{r} \| \mathbf{m}, q, n)$
$\mathbf{t} \leftarrow \left(-\frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(F), \frac{1}{q}\mathsf{FFT}(c) \odot \mathsf{FFT}(f)\right)$ ▷ $\odot$ denotes the multiplication in the FFT domain
do
    do
        $\mathbf{z} \leftarrow \mathsf{ffSampling}_n(\mathbf{t}, \mathbf{T})$
        $\mathbf{s} \leftarrow (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$
    while $\|\mathbf{s}\|^2 > \lfloor \beta^2 \rfloor$
    $(s_1, s_2) \leftarrow \mathsf{invFFT}(\mathbf{s})$
    $\mathbf{s} \leftarrow \mathsf{Compress}(s_2, 8 \cdot \mathsf{sbytelen} - 328)$
while $s = \perp$
return $(\mathbf{r}, \mathbf{s})$

2.2 Two Signing Designs

The reference implementation of Falcon offers two signing designs, `sign_tree` and `sign_dyn`, which realize different ways of generating signatures. The `sign_tree` design strictly follows Algs 2, where the expanded key $\mathbf{sk}$ consists of $\hat{\mathbf{B}}$ and $\mathbf{T}$. In contrast, `sign_dyn` accepts the original key contained in $\mathbf{B}$ and computes $\hat{\mathbf{B}}$ and $\mathbf{T}$ on demand during signing. Both designs invoke a function called `smallints_to_fpr` to convert integer coefficients into FP numbers, followed by an FFT to obtain $\hat{\mathbf{B}}$ and construct $\mathbf{T}$. In the `sign_tree` design, these computations are performed once and the expanded key is stored, whereas `sign_dyn`, which stores only the original key, repeats them for every signing operation.

These two designs reflect a trade-off between signing speed and memory consumption. In `sign_tree`, since $\hat{\mathbf{B}}$ and $\mathbf{T}$ are precomputed, the signing process is faster. The `sign_tree` design is meant to be used when several signatures are to be computed with the same private key. According to the comments in the reference code [2], the amortized cost per signature is roughly halved when using the `sign_tree` design for short messages, although it depends

[2] https://falcon-sign.info/impl/falcon.h.html, see lines 174-186 in falcon.h.

on the underlying architecture and implementation choices. However, the expanded key size is much larger than that of the original key. For instance, in Falcon-512 (resp. Falcon-1024), the original key size is 1281 bytes (resp. 2305 bytes), while the expanded key reaches 57,352 bytes (resp. 122,888 bytes). Such large keys impose significant memory pressure on resource-constrained platforms. Specifically, the Falcon-1024 implementation based on the `sign_tree` design does not fit into 128KiB of RAM, rendering this implementation infeasible on Cortex-M4 microcontrollers [KRSS19]. Consequently, mainstream Falcon implementation repositories (e.g., PQClean) adopt `sign_dyn` for external interfaces.

2.3 Floating-Point Conversion and FFT in Falcon

In this subsection, we detail the FP conversion and the FFT in Falcon. As mentioned above, before applying the FFT to the secret key matrix $\mathbf{B}$, Falcon utilizes `smallints_to_fpr` to convert the coefficients of the key into the FP type, called `fpr`, which follows the IEEE-754 standard with 53 bits of precision. More precisely, Falcon encodes FP numbers using unsigned 64-bit integers. The 64 bits are divided into three fields: the most significant bit represents the sign s, the next 11 bits represent the exponent e, and the remaining 52 bits represent the mantissa m. These components (s, e, m) together define a FP number $x = (-1)^s \cdot 2^{e-1023} \cdot \left(1 + m \cdot 2^{-52}\right)$.

Falcon employs FFT to accelerate the polynomial multiplications. Although FFT is a well-known algorithm, certain implementation details in Falcon's reference code deserve special attention. Let ω denote a $2n$-th primitive root of unity, i.e., $\omega = \exp(i\pi/n)$. Define $\omega_j = \omega^{2j+1}$ for $j \in [0, n-1]$. These values are the roots of $x^n + 1$ over the complex field, and they satisfy the property $\omega_{n-1-j} = \overline{\omega_j}$. After applying FFT to f, the vector representation is given by $(f(\omega_0), f(\omega_1), \ldots, f(\omega_{n-1}))$. Since $f \in \mathcal{R}$ has real coefficients, it holds that $f(\omega_{n-1-j}) = \overline{f(\omega_j)}$. Therefore, the first point to note is that Falcon only needs to keep half of the complex values during FFT. Second, Falcon stores the real part of each complex value at index j of the original array holding the coefficients of f, while the corresponding imaginary part is stored at index $j + n/2$, thereby achieving an in-place representation. Finally, the FFT starts from the inputs whose form is $f_j + i \cdot f_{j+n/2}$, where $j \in [0, n/2 - 1]$.

2.4 Threat Model

In our threat model, we consider a passive adversary who is capable of measuring the power consumption of the target device while it executes cryptographic operations. The adversary is assumed to have knowledge of the software implementation and can approximately identify the locations of specific cryptographic computations. The attack follows a profiled attack setting, which requires building precise leakage models using template-based or machine-learning-based approaches.

During the profiling phase, the adversary is assumed to possess a clone of the target device, over which the adversary has full control of both the secret key and the input data. This enables the collection of corresponding power traces for the construction of accurate leakage models, which is performed in an offline manner. In the subsequent attack phase, however, the adversary can only acquire one or a few power traces from the actual target device, without access to the secret key. Based on the leakage models obtained in the profiling phase, the adversary aims to recover the secret key from the limited information available during the attack phase.

3 The Attack Methodology

In this section, we develop an interpretable attack methodology that connects leakage from Falcon's FP operations to secret-coefficient classification and ultimately full key recovery. We first analyze the leakage from the FP conversion and the FP multiplication at the first FFT layer, and show how it can be combined to determine sufficient secret-key coefficients. Based on these leakage semantics, we then build a complete attack pipeline covering both the single-trace attack and its multi-trace extension. Falcon-512 is used as the representative case throughout this section.

3.1 Core Principle of the Attack

In Falcon, the secret keys and the public key satisfy $h = gf^{-1} \pmod q$. We can rewrite it in the following form:

$$\begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{n-1} \end{bmatrix} = \begin{bmatrix} h_0 & -h_{n-1} & \cdots & -h_1 \\ h_1 & h_0 & \cdots & -h_2 \\ \cdots & \cdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & \cdots & h_0 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \end{bmatrix} \pmod q. \tag{1}$$

Once at least n coefficients of f and g are determined in total, the remaining coefficients can be recovered by solving the resulting system of equations. Since f and g follow the same distribution and are processed identically, recovering at least half of the coefficients of each allows the remaining coefficients to be recovered by solving a linear system of dimension at most $n/2$.

The remaining question is how to obtain sufficient leakage to determine these coefficients. This paper shows that several FP operations reveal substantial information about the secret key. In particular, when the coefficients of f and g are converted to FP representations and multiplied by twiddle factors at the first layer of FFT, information about their value ranges is leaked. In the following, we exploit these leakages, together with our proposed method, to determine at least half of the coefficients of f and g, ensuring that at least n secret-key coefficients are recovered in total.

3.2 Exploiting Leakages from the Floating-Point Conversion

`smallints_to_fpr` invokes the function `fpr_scaled` to convert every secret key coefficient into an FP number, which is illustrated in Listing 1[3]. Specifically, this function takes a signed 64-bit integer `i` and a scaling exponent `sc` as input, converting `i` into a `fpr`-type number that represents the value $\mathtt{i} \times 2^{\mathtt{sc}}$. When converting secret key coefficients, `sc` is simply set to 0. The entire flow of `fpr_scaled` can roughly be divided into four parts: extracting the absolute value, left-shifting the absolute value into a prescribed range, detecting the zero value, and packing the result into `fpr` for return.

By analyzing the source code, the absolute value extraction and zero detection introduce clear leakages. To make the source-level operations easier to interpret, we show the corresponding `-O0` assembly in Listing 2, since this compilation level follows the C semantics in a relatively direct form. In the absolute value extraction, `i ^= -(s64)s` generates a 64-bit mask `-s`, which is either all-zero or all-one because $\mathtt{s} \in \{0, 1\}$. In the assembly, this mask is explicitly produced by `rsbs` and `sbc.w`, causing a large Hamming weight (HW) difference. The zero detection follows a similar pattern: `-(u64)t` is generated by `negs` and `sbc.w`, also producing all-zero or all-one mask values with a large HW difference. The two 32-bit halves of the mask are then explicitly stored in memory, resulting in obvious

[3]We use `u64` to denote an unsigned 64-bit integer (C's `uint64_t`) and `s64` to denote a signed 64-bit integer (C's `int64_t`), etc., without further notice.

```
fpr fpr_scaled(s64 i, int sc) {
    ...

    /* absolute value extraction */
    s = (int)((u64)i >> 63);
    i ^= -(s64)s;
    i += s;

    m = (u64)i;
    e = 9 + sc;
    /* left-shifts the absolute value */
    FPR_NORM64(m, e);

    ...

    /* zero detection */
    t = (u32)((u64)(i | -i) >> 63);
    m &= -(u64)t;
    e &= -(int)t;

    /* packing the result into fpr type*/
    return FPR(s, e, m);
}
```

Listing 1: The C code of `fpr_scaled`.

```
@ i ^= -(s64)s
rsbs r4, r8, #0          @ low 32-bit mask -s
sbc.w r5, r3, r9         @ high 32-bit mask -s
ldrd  r2, r3, [r7, #328]
eor.w sl, r2, r4         @ i_low  ^= mask_low
eor.w fp, r3, r5         @ i_high ^= mask_high
strd  sl, fp, [r7, #328]

@ m &= -(u64)t
negs r2, r2              @ low 32-bit mask -t
str r2, [r7, #80]        @ store mask_low
sbc.w r3, r3, r2         @ high 32-bit mask -t
str r3, [r7, #84]        @ store mask_high
...
ands r4, r2              @ m_low  &= mask_low
ands r1, r3              @ m_high &= mask_high
strd  r3, r4, [r7, #352]

@ e &= -(int)t
negs r3, r3              @ 32-bit mask -t
...
ands r3, r2              @ e &= mask
str.w r3, [r7, #348]
```

Listing 2: Corresponding assembly of the marked C code in `fpr_scaled`.

additional leakage. The corresponding `eor.w` and `ands` instructions in these two parts also leak due to mask-dependent operands, but the main leakage is expected to come from generating and storing these masks.

The more subtle leakage comes from `FPR_NORM64`, shown in Listing 3. This function uses a binary search approach to left-shift the absolute value of the component `m` until its most significant bit becomes 1. It performs six similar tests, corresponding to left-shifts by $32, 16, 8, 4, 2$, and 1 bit(s). In each test, `nt` indicates whether the checked high bits of the current `m` are nonzero. If `nt`=0, the corresponding left shift is applied; otherwise, `m` remains unchanged. The main leakage in each test comes from the mask `((u64)nt - 1)`, which is either all-one or all-zero since `nt` $\in \{0, 1\}$. This mask is generated by `subs` and `adc.w` and then stored in memory by `str.w`, resulting in obvious leakage because of the large HW difference between the two mask values. The following `ands` also leak due to mask-dependent operands. Therefore, observing these leakages reveals whether each conditional shift is performed, allowing us to partition the possible absolute values of each coefficient.

Combining the above leakages in the FP conversion, we can deduce the sign of each coefficient of f (or g), and classify the absolute values of each coefficient into different sets. Recall that the absolute value range of coefficients is $[0, 31]$, each coefficient belongs to one of the following sets:

$$\{0\},\ \{1\},\ \{2, 3\},\ \{4, .., 7\},\ \{8, .., 15\},\ \{16, ..., 31\}. \tag{2}$$

This means we can determine coefficients with values of 0 and ± 1. According to the distribution of the coefficients of f (or g), the expected number of coefficients that are 0 and ± 1 is about $148 \approx D_{[-31,31],\sigma}(\{0, \pm 1\}) \times 512$ with $\sigma \approx 4.053$.

3.3 Exploiting Leakages from the Floating-Point Multiplication

Building on the partition of the coefficient range presented in (2), we find that leakage from the FP multiplication in the first layer of the FFT enables a finer-grained partition of the secret key coefficients, where secret key coefficients are directly multiplied by the fixed twiddle factor $(\frac{\sqrt{2}}{2} + i \cdot \frac{\sqrt{2}}{2})$. Based on the FFT structure, only the coefficients f_j with $j \in [128, 255] \cup [384, 511]$ are multiplied by the fixed twiddle factors.

```
#define FPR_NORM64(m, e) do {
    u32 nt;
    e -= 63;

    nt = (u32)(m >> 32);
    nt = (nt | -nt) >> 31;
    m ^= (m ^ (m << 32)) & ((u64)nt - 1);
    e += (int)(nt << 5);

    nt = (u32)(m >> 48);
    nt = (nt | -nt) >> 31;
    m ^= (m ^ (m << 16)) & ((u64)nt - 1);
    e += (int)(nt << 4);

    /* The same pattern is repeated
       for shifts by 8, 4, 2, and 1. */
    ...
} while (0)
```

Listing 3: The C code of `FPR_NORM64`.

```
@ m ^= (m ^ (m << 32)) & ((u64)nt - 1)
@ compute (m ^ (m << 32))
...

@ compute ((u64)nt - 1)
subs r3, #1                 @ low 32-bit mask
str.w r3, [r7, #288]        @ store mask_low
...
adc.w r3, r3, #0xffffffff   @ high 32-bit mask
str.w r3, [r7, #292]        @ store mask_high

@ apply mask and update m
...
ands r0, r5                 @ apply mask_low
ands r3, r2                 @ apply mask_high
...
@ The shift blocks of 16, 8, 4, 2, and 1
@ follow the same pattern.
```

Listing 4: Corresponding assembly of the marked conditional-shift statement.

Falcon uses `fpr_mul` to perform FP multiplication, as shown in Listing 5, with the corresponding assembly shown in Listing 6. In `fpr_mul`, two main tasks are performed. First, it multiplies the mantissas of the two FP numbers; second, it normalizes the product of the mantissas to the range $[2^{54}, 2^{55} - 1]$.

In the mantissa multiplication, the 53-bit mantissas are split into smaller limbs, and several `umull` instructions compute the partial products. Since the twiddle factor is fixed, these products depend on the secret coefficient and can be modeled by profiling attacks. For coefficient magnitudes in $[0, 31]$, the lower 25 bits of the coefficient mantissa remain zero. Therefore, `x0*y0` is not considered a leakage target, and we focus on the later products involving `x1`. As shown in Listing 6, their high and low words are explicitly stored in memory by `str.w`, providing additional exploitable leakage from the coefficient-dependent multiplication results.

The normalization procedure introduces another useful leakage. It checks whether the mantissa product `zu` is at least 2^{55}. Since the value range of `zu` is $[2^{54}, 2^{56} - 1]$ before the normalization procedure, `w`=1 if `zu` $\geq 2^{55}$, and `w`=0 otherwise. The following operation uses `-w` as an all-zero or all-one mask. As shown in Listing 6, this mask is generated by `negs` and `sbc.w` and then explicitly stored in memory by `str`, resulting in strong leakage due to the large HW difference between the two mask values. The following `ands` instructions may also leak through their mask-dependent operands. These leakages reveal whether `zu` crosses the threshold 2^{55}.

Since the absolute value range of the coefficient is $[0, 31]$, we can enumerate all possible values of the coefficient to calculate whether the results exceed 2^{55}. Taking f as an example, this will divide the absolute values (excluding the determined absolute values of 0 and 1) into 2 different sets:

$$\begin{aligned} \{|f_j| \mid \texttt{zu} < 2^{55}\} &= \{2, 4, 5, 8, 9, 10, 11, 16, 17, 18, 19, 20, 21, 22\}, \\ \{|f_j| \mid \texttt{zu} \geq 2^{55}\} &= \{3, 6, 7, 12, 13, 14, 15, 23, 24, 25, 26, 27, 28, 29, 30, 31\}. \end{aligned} \tag{3}$$

As a result, it can be determined which set $|f_j|$ belongs to based on the leakage from the normalization procedure. Combining the partition outlined in (2), coefficients in the set f_j ($j \in [128, 255] \cup [384, 511]$) with values of ± 2 and ± 3 can be determined. The remaining uncertain absolute values are further refined into

$$\{4, 5\},\ \{6, 7\},\ \{8, \ldots, 11\},\ \{12, \ldots, 15\},\ \{16, \ldots, 22\},\ \{23, \ldots, 31\}.$$

At this point, the expected number of coefficients determined as $0, \pm 1, \pm 2, \pm 3$ is approximately $148 + 256 \times D_{[-31,31],\sigma}(\{\pm 2, \pm 3\}) \approx 231$.

```
fpr fpr_mul(fpr x, fpr y) {
    ...
    xu = (x & (((u64)1 << 52) - 1)) | (1 << 52);
    yu = (y & (((u64)1 << 52) - 1)) | (1 << 52);
    /* mantissas multiplication */
    x0 = (u32)xu & 0x01FFFFFF;
    x1 = (u32)(xu >> 25);
    y0 = (u32)yu & 0x01FFFFFF;
    y1 = (u32)(yu >> 25);
    w = (u64)x0 * (u64)y0;
    ...
    w = (u64)x1 * (u64)y0;
    ...
    zu = (u64)x1 * (u64)y1;
    ...
    zu += z2;
    ...

    /* normalization procedure */
    zv = (zu >> 1) | (zu & 1);
    w = zu >> 55;
    zu ^= (zu ^ zv)  & -w;
    ...
    return FPR(s, e, zu);
}
```

Listing 5: The C code of `fpr_mul`.

```
@ w = (u64)x1 * (u64)y0
...
umull r2, r1, r2, r1      @ x1 * y0
str.w r1, [r7, #196]      @ store high word
str.w r2, [r7, #192]      @ store low word
...

@ zu = (u64)x1 * (u64)y1
...
umull r2, r1, r2, r1      @ x1 * y1
str.w r1, [r7, #188]      @ store high word
str.w r2, [r7, #184]      @ store low word
...

@ zu ^= (zu ^ zv) & -w
...
negs r0, r2               @ low 32-bit mask -w
str r0, [r7, #80]         @ store mask_low
sbc.w r3, r1, r3          @ high 32-bit mask -w
str r3, [r7, #84]         @ store mask_high
...
ands r0, r5               @ apply mask_low
...
ands r3, r2               @ apply mask_high
...
```

Listing 6: Corresponding assembly of the marked C code in `fpr_mul`.

The finer value range facilitates side-channel analysis on the mantissa multiplication to further partition the value range. However, it is not necessary to recover all of these remaining uncertain coefficients. Instead, a single binary classifier distinguishing 4 from 5 is sufficient for our purpose. Once the coefficients f_j with $j \in [128, 255] \cup [384, 511]$ that take values ± 4 and ± 5 are further determined, the expected total number of determined coefficients reaches approximately 285, exceeding half of the coefficients of the secret key f. Moreover, distinguishing 6 from 7 requires only one additional binary classifier and increases the expected number of determined coefficients to approximately 313.

3.4 The Pipeline of the Attack

Based on the leakage semantics and coefficient partitions established above, we integrate the three leakage sources into the complete attack pipeline shown in Fig. 1. The attack consists of a profiling phase and an attack phase. During profiling, classifiers are constructed for FP conversion, the normalization procedure, and mantissa multiplication. During the attack, these classifiers are applied sequentially to progressively refine the coefficient ranges. The resulting coefficient predictions are then processed by an ISD-style error-tolerant key recovery strategy to mitigate classification errors and recover the full secret key.

3.4.1 Profiling Phase

We build classifiers for three targeted stages; each is trained as both a Gaussian template [CRR03] and an MLP (multilayer perceptron), where the MLP generally achieves higher accuracy. First, an 11-class classifier is built for FP conversion to jointly predict the coefficient sign and its absolute value set in (2). Second, a binary classifier is built from the normalization procedure leakage to determine which set in (3) a coefficient belongs to, thereby further refining the coarse partition. Third, a binary classifier is constructed from the mantissa-multiplication leakage to distinguish 4 from 5. Optionally, another binary classifier can be built to distinguish 6 from 7, further increasing the number of determined coefficients.

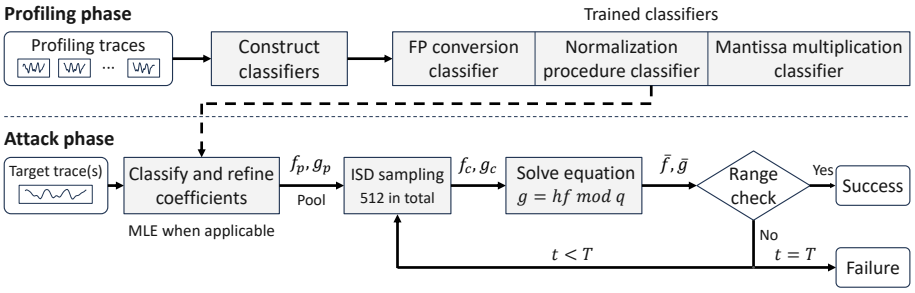


Figure 1: Workflow of the proposed attack. f_p, g_p are the predicted coefficient pools; f_c, g_c are the subsets selected in one ISD trial, totaling 512 constraints; and $\bar{f}, \bar{g}$ are the candidate recovered secret keys. MLE combines repeated observations when available. The same recovery pipeline applies to both single-trace and multi-trace attacks.

3.4.2 Attack Phase

During the attack phase, the adversary collects the target power trace(s) covering the identified leakages for both f and g, segments the identified leakage regions, and applies the profiled classifiers to progressively determine enough coefficients of f and g (at least 512). When repeated observations of the same coefficient are available, their class-probability distributions can be combined using maximum likelihood estimation (MLE) before making the final prediction. The resulting predicted coefficients form the candidate pools f_p and g_p shown in Fig. 1.

Since classification errors may exist in f_p and g_p, we employ an ISD-style (Information Set Decoding [Pra62]) strategy. The ISD-style approach is traditionally used in coding theory for solving the decoding problem of random linear codes. We adapt it to enable error-tolerant key recovery in our attack. In each trial, a total of 512 predicted coefficients are selected from the candidate pools, denoted by f_c and g_c, and used as constraints in the public relation $g = hf \bmod q$. Solving the resulting linear system yields candidate keys $\bar{f}$ and $\bar{g}$. If all recovered coefficients fall within the valid Falcon key ranges, the candidate is accepted; otherwise, a new subset is sampled until a valid key is obtained or the preset iteration threshold T is reached. For a candidate pool of N predicted coefficients containing m errors, a randomly sampled information set of 512 coefficients is error-free with probability $p = \binom{N-m}{512}/\binom{N}{512}$, so that T independent trials succeed with probability $1-(1-p)^T$. This requires $N - m \geq 512$; otherwise no error-free information set exists and recovery fails regardless of the trial budget. We note that the ISD-style strategy is practical. Even with an iteration threshold of 10^6, solving all 10^6 candidate linear systems in the worst case requires only about 4.7 core-hours for Falcon-512 on a standard personal computer using a C implementation of Gaussian elimination. Since the trials are independent, they can also be parallelized straightforwardly.

The same recovery pipeline applies to both the single-trace and multi-trace settings; the difference lies in how repeated observations are obtained. In a single-trace attack, all observations originate from one physical trace. For the leakage in the normalization procedure and mantissa multiplication, multiple observations of the same coefficient are already present within that trace (see the explanation in Section 5.2) and can therefore be combined by MLE. In the multi-trace setting, MLE additionally combines observations across multiple traces. The latter setting is applicable only to `sign_dyn`, where key expansion is repeatedly executed during signing; for `sign_tree`, key expansion is performed only once, and hence only the single-trace setting is considered.

4 Attack Adaptation to Optimized Implementations

Section 3 establishes an interpretable attack methodology by linking FP-operation leakages to coefficient classification and ultimately key recovery. The source code and the relatively straightforward `-O0` assembly are used to identify these leakage semantics and construct the attack pipeline. Higher optimization levels, however, may transform or weaken the corresponding leakage patterns. In addition, the Falcon team provides an embedded-assembly implementation optimized for Cortex-M4 processors, referred to as ASM Falcon. It implements FP operations with human-audited embedded assembly, and therefore its leakage patterns differ from those under `-O0`. In this section, we analyze the leakage changes under `-O3`, `-Os`, and ASM Falcon, and adapt the profiling and classification stages of the attack accordingly. Table 3 provides a qualitative overview of how the identified leakage evolves across these implementations.

Table 3: Qualitative comparison of the identified leakage across different Falcon implementations, with `-O0` serving as the reference. "Strong" denotes prominent and localized leakage, "Moderate" denotes reduced but still exploitable leakage, and "Mitigated" denotes substantially reduced residual leakage, which may limit classification capability in a single-trace attack. Abbreviation: Abs. = absolute-value, Norm. proc. = normalization procedure, Mant. mul. = mantissa multiplication.

Function	Leakage source	`-O0`	`-O3`	`-Os`	ASM Falcon
`fpr_scaled`	Abs. extraction	Strong	Moderate	Moderate	Moderate
	Zero detection	Strong	Moderate	Moderate	Moderate
	`FPR_NORM64`	Strong	Moderate	Moderate	Mitigated
`fpr_mul`	Norm. proc.	Strong	Mitigated	Mitigated	Moderate
	Mant. mul.	Strong	Moderate	Moderate	Moderate

4.1 Static Assembly Analysis

Assembly Analysis under `-O3` and `-Os` Under `-O0`, the main leakage sources arise from explicit mask generation, memory operations, and coefficient-dependent operations. Under `-O3` and `-Os`, compiler optimizations substantially change the assembly implementation of these operations, resulting in different leakage characteristics. Since the two optimization levels transform the mask-related operations in similar ways, we mainly use `-O3` to illustrate these assembly-level changes.

For the absolute value extraction and zero detection in `fpr_scaled`, `-O0` explicitly generates the all-zero/all-one masks `-s` and `-t`. Under `-O3`, the compiler avoids materializing these masks as standalone values. Instead, it uses instructions such as `eor.w r0, r0, r1, asr #31`; the inline barrel shifter performs the arithmetic right shift and transiently produces an all-zero or all-one value, which serves directly as the operand of the same instruction. This transformation reduces the leakage observed under `-O0`, while residual exploitable leakage remains mainly associated with the transient mask values produced within the inline barrel shifter.

For `FPR_NORM64`, `-O0` explicitly generates the mask `((u64)nt - 1)` and stores its two 32-bit halves in memory. Under `-O3`, the shift-control logic is implemented through conditional execution using `IT` blocks and conditional moves. This leakage source is therefore mitigated, but residual leakage remains in the register operations. When the condition holds, the conditional assignment causes the target register to transition from an all-one value to an all-zero value, leaving exploitable leakage.

A similar change occurs in the normalization procedure. Under `-O0`, `w` is converted into the 64-bit mask `-w`, which is stored in memory and then used for conditional data selection. Under `-O3`, the selection logic is implemented using a bit test (`tst.w`) to determine whether `zu` exceeds 2^{55}, followed by conditional data movement (`moveq`), without explicitly constructing the mask. Here, the conditional assignment only exhibits a small HW difference because `zv` is obtained by right-shifting `zu` by only 1 bit. Consequently, the prominent and localized leakage observed under `-O0` is mitigated.

For the mantissa multiplication, both `-O3` and `-Os` reduce memory accesses and apply interleaved instruction scheduling. The `-O3` implementation keeps more intermediate values in registers and applies more aggressive instruction scheduling, whereas `-Os` uses fewer registers to reduce code size. Although these optimizations reduce the leakage observed under `-O0`, coefficient-dependent leakage remains distributed across the mantissa multiplication region and is exploitable.

Overall, `-O3` and `-Os` reduce the prominent `-O0` leakage mainly by reducing explicit mask generation and memory stores, introducing conditional execution, and applying more aggressive instruction scheduling.

Assembly Analysis of ASM Falcon The Falcon team's embedded assembly is independent of the compiler optimization levels and can therefore be analyzed in isolation. For the absolute value extraction and zero detection in `fpr_scaled`, ASM Falcon substantially reduces memory operations, similarly to `-O3` and `-Os`. However, the absolute value extraction still explicitly generates an all-zero/all-one mask in a register, while the zero detection transiently produces the mask within the inline barrel shifter.

A more significant difference appears in `FPR_NORM64`. ASM Falcon uses the `clz` instruction to directly count the number of leading zeros and determine the required shift amount, thereby bypassing the mask-based shift-control logic. As a result, the leakage at this location is mitigated. By contrast, in the normalization procedure of `fpr_mul`, ASM Falcon still explicitly generates the all-zero/all-one mask instead of using a bit test (`tst.w`) and conditional data movement as in `-O3` and `-Os`. Therefore, this region still exhibits exploitable leakage. For the mantissa multiplication, ASM Falcon retains the same limb-based multiplication structure as the compiler-generated implementations, with the main differences lying in the organization of intermediate accumulation and sticky-bit processing, while still retaining exploitable leakage.

Overall, ASM Falcon further reduces the leakage in `FPR_NORM64`, while retaining exploitable leakage in the normalization procedure of `fpr_mul`.

Remark 1. Table 3 qualitatively summarizes how different implementations affect the prominent and localized leakage observed under `-O0`. Reduced or dispersed leakage may still be exploitable by profiling classifiers, so the labels in Table 3 are intended to guide attack adaptation rather than predict profiling accuracy.

4.2 Adjustment to the Attack Method

The preceding assembly analysis reveals leakage changes across the optimized and assembly implementations, as summarized in Table 3. These changes do not affect the overall leakage-to-key-recovery principle, but they require us to adjust how coefficient information is extracted from the traces. We therefore keep the same high-level attack strategy and adapt the trace windows and classification tasks to the leakage pattern.

A key adjustment concerns the coefficient-refinement stage after the 11-class FP-conversion classifier. In the original pipeline, the FP-conversion classifier first provides the coarse coefficient ranges in (2). The normalization-procedure classifier then refines these ranges, distinguishing 2 from 3 and separating $\{4, 5\}$ from $\{6, 7\}$. The mantissa-multiplication classifier is subsequently used for finer distinctions, such as separating 4 from

5. Under `-O3` and `-Os`, however, the leakage in the normalization procedure is mitigated, making this sequential refinement less reliable.

To address this issue, we jointly exploit the residual leakage from the mantissa multiplication and the immediately following normalization procedure. Since these two operations are consecutive, we merge their trace regions and use the combined leakage for coefficient refinement. After the 11-class FP-conversion classifier provides the coarse ranges, we construct a binary classifier to distinguish 2 from 3 and a four-class classifier to distinguish 4, 5, 6, and 7. This replaces the original sequential refinement based on separate normalization-procedure and mantissa-multiplication classifiers. The determined coefficients are then processed using the same candidate-pool construction and ISD-style key-recovery strategy described in Section 3. Thus, the overall leakage-to-key-recovery principle remains unchanged, while the profiling models and attack pipeline are adapted to the weaker and more temporally dispersed leakage under `-O3` and `-Os`.

The situation differs for ASM Falcon. Its main bottleneck occurs earlier, at the initial FP-conversion classification stage. The use of `clz` in `FPR_NORM64` mitigates the leakage used by the 11-class classifier to obtain the coarse coefficient-range partition. Since all later refinement stages depend on this partition, errors at this stage propagate through the entire attack pipeline. Meanwhile, the normalization procedure in ASM Falcon still exhibits exploitable leakage, so merging it with the mantissa-multiplication trace region is unnecessary and cannot address the earlier bottleneck of the 11-class classifier. For these reasons, we retain the original attack pipeline when evaluating ASM Falcon in the experiments. As shown by our results, the embedded assembly implementation successfully thwarts the single-trace recovery; however, we demonstrate in Section 5.3 that this limitation can be effectively overcome by mounting a multi-trace attack.

5 Experimental Validation

We demonstrate a single-trace attack on Falcon-512 across the `-O0`, `-O3`, and `-Os` optimization levels — both `-O3` and `-Os` are practical deployment settings. Additionally, we present a multi-trace attack against the Falcon team's ASM Falcon implementation.

5.1 Experimental Setup

We target the reference C implementation from PQClean[4], compiled at `-O0` (as a benchmark baseline), `-O3` and `-Os` (both commonly used in software releases), and the ASM Falcon implementation[5]. The compiler is `arm-none-eabi-gcc 14.2.0`. Our firmware targets the key expansion stage, which is in the key generation under `sign_tree` design while in the signing procedure under `sign_dyn`. Because the targeted FP conversion and FP multiplication are the same for `sign_tree` and `sign_dyn` designs, it is enough to run the functions `smallints_to_fpr` and the subsequent `PQCLEAN_FALCON512_CLEAN_FFT` as the harness. The full trace length covers `smallints_to_fpr` and the first layer of `PQCLEAN_FALCON512_CLEAN_FFT`, which is the interested operation that involved multiplication with $\frac{\sqrt{2}}{2} + i \cdot \frac{\sqrt{2}}{2}$.

Power traces are collected using a ChipWhisperer Husky with a CW313 board equipped with an STM32F415 microcontroller. GPIO triggers are inserted to mark the capture window. The MCU runs at 40 MHz for `-O0` and at 25 MHz for `-O3`, `-Os`, and ASM Falcon. The sampling rate is set to 4 points per clock cycle (PPC) for `-O0` and 8 PPC for the other implementations. The higher sampling rate provides finer temporal resolution for `-O3`/`-Os` and ASM Falcon, where the leakage is weaker and more temporally compressed than under `-O0`. After collection, we apply a 4th-order low-pass filter with a cutoff frequency set to

[4] https://github.com/PQClean/PQClean/tree/master/crypto_sign/falcon-512/clean
[5] https://falcon-sign.info/impl/fpr.c.html

half of the MCU frequency to reduce high-frequency noise. Throughout the analysis, all random seeds are fixed as 42 for reproducibility.

5.2 Profile

Profiling Keys Falcon-512 secret key coefficients are formally bounded within $[-31, 31]$, but values outside $[-22, 22]$ occur with negligible probability according to the prescribed distribution. We therefore use $[-22, 22]$ as the effective coefficient range in our experiments. For the profiling keys, we fabricate keys k_i in which all 512 coefficients equal i, for each $i \in \{-22, \ldots, 22\}$. For `-O0`, `-O3`, and `-Os`, we collect two traces per fabricated key, each of which covers all target leakage sources, and three such traces per fabricated key for ASM Falcon, giving $45 \times 2 = 90$ and $45 \times 3 = 135$ physical acquisitions, respectively. Collecting them takes approximately 47.7, 10.8, 13.2 and 14.4 minutes for `-O0`, `-O3`, `-Os` and ASM Falcon, respectively. These numbers reflect the trace-collection budget used for profiling in our experiments, rather than a measured minimum.

Trace Segmentation For each fabricated key, the corresponding traces are cropped into operation-specific regions for classifier training. The FP-conversion segment covers the entire execution of `fpr_scaled`, except for the final call to `FPR`, and is used to construct the 11-class classifier. Since `fpr_scaled` processes all 512 coefficients, for `-O0`, `-O3` and `-Os`, two traces per fabricated key provide $512 \times 2 = 1024$ segments for each coefficient value; for ASM Falcon, there are $512 \times 3 = 1536$ segments for each coefficient value.

For `fpr_mul`, the segments are taken from the coefficients multiplied by the twiddle factor at the beginning of the FFT. At this stage, the real and imaginary parts of the twiddle factor ($\frac{\sqrt{2}}{2} + i \cdot \frac{\sqrt{2}}{2}$) are equal, so each trace provides two observations for the normalization procedure and the mantissa multiplication. We also use this observation in the attack phase to improve classification accuracy. Since 256 coefficients are involved, for `-O0`, `-O3` and `-Os`, two traces per fabricated key provide $256 \times 2 \times 2 = 1024$ segments for each coefficient value; for ASM Falcon, there are $256 \times 3 \times 2 = 1536$ segments for each coefficient value. Notably, the `fpr_mul` crop strategy depends on the implementation as discussed in Section 4. Under `-O0`, we crop the normalization procedure and the mantissa-multiplication segments, respectively. For ASM Falcon, we use the same crop strategy as `-O0`. Under `-O3` and `-Os`, the leakage from the normalization procedure is mitigated, so we merge it with the mantissa-multiplication region.

Classifier Construction As described in Section 3.4.1, we train both a Gaussian template and an MLP classifier for each stage (for `-O0`, we just train the Gaussian template), using 75% of the segments for training and the remaining 25% for testing.

For the Gaussian template, we select points of interest (PoIs) based on F-statistics [MOW17, GO22] with $> 0.1\%$ threshold. For `-O0`, `-O3`, `-Os`, and ASM Falcon, training the Gaussian templates takes no more than 10 seconds. In particular, for the `-O0` implementation, the leakage patterns are sufficiently clear that the PoIs can even be selected manually. Figure 2 shows the F-statistics for the identified leakage regions under `-O0`. Taking leakages in `fpr_scaled` as an example, five distinct clusters of peaks can be observed, as highlighted by the red boxes. From left to right, they correspond to one peak related to the absolute value extraction, three peaks caused by the left shifts in `FPR_NORM64`[6], and the final cluster is related to zero detection. Selecting the PoIs corresponding to these peaks is sufficient to make the Gaussian template achieve 100% accuracy.

[6] Although `FPR_NORM64` contains six sequential snippets, only three peaks are visible because the absolute values of Falcon-512 coefficients lie in $[0, 31]$. Consequently, the upper 56 bits of the `u64` variable are always zero, and only the last three snippets exhibit data-dependent leakage.

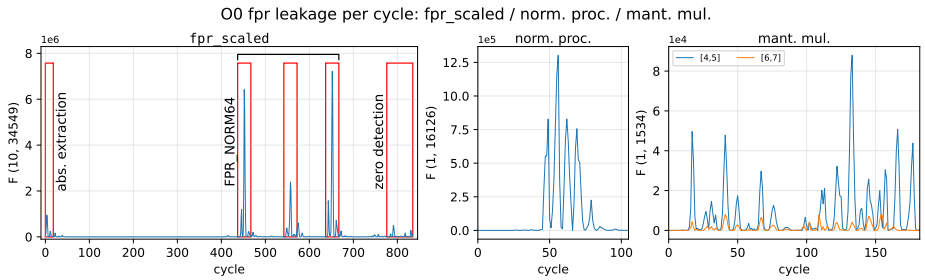


Figure 2: F-statistics of `-O0`'s targeted regions in `fpr_scaled`, normalization procedure (norm. proc.) and mantissa multiplication (mant. mul.) in `fpr_mul`. The y-axis label $F(d_1, d_2)$ denotes the F-statistics with degrees of freedom.

The MLP classifier takes the entire preprocessed trace segment as input, and consists of three fully connected layers with dimensions input $\to 64 \to 64 \to N$, where N is the number of target classes. Each hidden layer is followed by batch normalization and SELU activation and a final ReLU. The model is trained with cross-entropy loss; during inference, softmax is applied to the output to obtain class probability distributions. Owing to its small size, the MLP training takes approximately 3.9, 2.2 and 5.9 minutes for `-O3`, `-Os`, and ASM Falcon, respectively.

The accuracies of the two types of classifiers are summarized in Table 4. The values in parentheses are reported only for comparison and the corresponding classifiers are not used in the attack because their accuracy is insufficient for a single-trace attack. For the `-O0`, `-O3`, and `-Os` implementations, the classifiers used in the attack achieve very high accuracy. In particular, after adapting the attack method for `-O3` and `-Os`, the merged-window classifiers for distinguishing $\{2, 3\}$ and $\{4, 5, 6, 7\}$ achieve accuracies close to 1. Compared with the accuracies of the standalone normalization-procedure classifiers shown in parentheses, this confirms the effectiveness of the adjustment. For ASM Falcon, the clear accuracy degradation of the 11-class FP-conversion classifier indicates that the exploitable leakage is mitigated, consistent with the analysis in Section 4. This bottleneck prevents the single-trace attack from succeeding against ASM Falcon and motivates the additional multi-trace evaluation in the attack phase.

Table 4: Accuracy of the Gaussian template and MLP classifier for each targeted stage. Values in parentheses are reported only for comparison and are not used in the attack. "mul$\{a, \ldots, b\}$" denotes the mantissa-multiplication window for coefficients in $\{a, \ldots, b\}$. "merge$\{a, \ldots, b\}$" denotes the merged window covering the mantissa multiplication and the normalization procedure for distinguishing $a, \ldots, b$.

	`-O0`	`-O3`		`-Os`		ASM Falcon	
stage	Gaussian	Gaussian	MLP	Gaussian	MLP	Gaussian	MLP
FP conversion	1.0000	1.0000	1.0000	0.9999	1.0000	0.6479	0.7139
norm. proc.	1.0000	(0.9038)	(0.9596)	(0.7275)	(0.9141)	0.9913	0.9970
mul$\{4, 5\}$	1.0000	—	—	—	—	0.9863	0.9948
mul$\{6, 7\}$	1.0000	—	—	—	—	0.9329	0.9609
merge$\{2, 3\}$	—	0.9980	0.9980	0.9922	1.0000	—	—
merge$\{4, 5, 6, 7\}$	—	0.9844	0.9922	0.9961	0.9980	—	—

5.3 Attack

We randomly generate 100 genuine Falcon keys, none of which is used in profiling, to evaluate the success rate of the proposed attack. For each implementation, we first collect one power trace covering the identified leakage regions for both f and g and then segment it to validate the single-trace attack. Since the 11-class FP-conversion classifier for ASM Falcon does not achieve near-perfect accuracy, we further conduct a multi-trace attack.

Attack Results under `-O0`, `-O3`, and `-Os` For the `-O0` implementation, our single-trace attack achieves a 100% success rate using the Gaussian template even without using the ISD-style key-recovery strategy. This indicates that all classifiers used in the attack phase produce predictions with perfect accuracy for all tested keys.

For the `-O3` and `-Os` implementations, a small number of classification errors may occur during the attack. We therefore use the ISD-style key-recovery strategy. In addition, we use the trick that a single power trace provides two observations for both the normalization procedure and the mantissa multiplication to further improve the corresponding prediction accuracy by the MLE strategy. With the Gaussian template, an ISD-style sampling threshold of 100 is sufficient to recover 99 out of 100 keys for `-Os` and 98 for `-O3` from a single trace. Replacing the Gaussian template with the MLP classifier further pushes the success rate for both to 100%.

To assess how sensitive the attack is to the acquisition setting, we repeat the `-O3` experiment with the MCU at 40 MHz. At 5 PPC (200 MS/s, the maximum sampling rate of the Husky) the MLP-based single-trace attack still recovers 100 out of 100 keys, while at 4 PPC (160 MS/s, matching the `-O0` setting) it recovers 79 out of 100. The operating frequency thus has little effect on the attack; the sampling rate is the limiting factor.

Attack Results on ASM Falcon For ASM Falcon, the 11-class FP-conversion classifier in the first stage achieves only 71.39% accuracy even with MLP. As a result, our single-trace attack does not recover any of the 100 tested keys. This shows that ASM Falcon effectively prevents our current single-trace attack.

We then conduct a multi-trace attack. Recall that the multi-trace attack is only applicable to `sign_dyn` design. We collect multiple traces for the same key and use the MLE strategy to improve the prediction accuracy. This strategy is mainly used to compensate for the limited accuracy of the 11-class FP-conversion classifier. This differs from the MLE used above for the `-O3` and `-Os` single-trace attacks. In the single-trace case, MLE combines two observations already present within one trace, because the normalization procedure and the mantissa multiplication provide two trace segments related to the same coefficient. In the multi-trace case, MLE combines observations across multiple traces.

To further reduce the impact of errors from the 11-class FP-conversion classifier, we introduce a preprocessing step tailored to ASM Falcon. We retain the coefficient indices whose prediction probability is at least 0.999. After all classification stages are completed, the final determined coefficients are extracted only from these high-confidence indices. Equivalently, the subsequent range refinement is restricted to this filtered set. Within this set, coefficients predicted as 0 or ± 1 are treated as error-free. We then apply the ISD-style key-recovery strategy to these high-confidence coefficients. Let N denote the number of high-confidence coefficients predicted as 0 or ± 1. To form an information set of 512 coefficients, we repeatedly sample $512 - N$ additional coefficients from those predicted as $\{\pm 2, \pm 3, \pm 4, \pm 5, \pm 6, \pm 7\}$, and test whether the sampled set yields a correct key reconstruction. The maximum number of trials is set to 1,000,000.

Our experimental results show that, when 10 traces are collected for each key, the attack recovers 53 out of the 100 tested keys with the MLP classifier.

6 Discussion

6.1 On the Proposed Attack

Our results show that the current key expansion implementation exposes severe side-channel leakage. Although compiler optimizations reduce and disperse several leakage sources in the reference C implementation, our adapted attack still enables single-trace key recovery under `-O3` and `-Os`. For ASM Falcon, the remaining leakage supports an effective multi-trace attack, although single-trace recovery is not achieved. Our single-trace attack applies to both `sign_tree` and `sign_dyn`. The latter is more exposed because key expansion is performed during every signing operation, repeatedly revealing the same leakage and additionally enabling the multi-trace attack. Exploring implementation designs that retain Falcon's practicality while reducing the exposure of key expansion is therefore an important direction for future work.

Despite opening a new attack path against Falcon implementations, our attack still has limitations. In particular, it does not achieve single-trace key recovery against ASM Falcon. The main bottleneck lies in the initial FP-conversion classification stage, where ASM Falcon substantially reduces the leakage exploited by the 11-class classifier. This prevents reliable coarse coefficient-range classification and causes errors to propagate through the subsequent recovery pipeline. Future work may investigate more error-tolerant recovery strategies or Soft Analytical Side-Channel Attacks [VCGS14] (SASCA) as possible directions toward a single-trace attack against ASM Falcon.

6.2 Systematic Sensitivity Analysis of Falcon Implementations

To provide a broader view of Falcon's side-channel attack surface, we combine the leakage points identified in this work with those exploited by existing attacks and summarize the main sensitive data paths as a reference for future analysis. Inspired by the analysis of HQC implementations in [GSA+25], Fig. 3 presents the main sensitive data paths in Falcon key generation and signing. Sensitive variables and functions are marked in red, while identified leakage points in this work are highlighted in orange.

Key Generation The main sensitive surfaces in key generation include secret sampling, randomness generation, and secret-dependent NTT operations. The function `mkgauss` generates f and g, and leakage from this process has already been exploited in a single-trace attack [QA25]. The seed, `shake256` state, and expanded random stream are also sensitive because they determine the sampled secrets, although no Falcon-specific attack exploiting this path is currently known. In addition, secret-dependent modular reduction during NTT has been shown to leak key-related information [QA26].

Key Expansion Key expansion contains the main leakage sources studied in this work. The secret keys are converted into FP form and transformed by the FFT. Our attack exploits leakage in `fpr_scaled` and `fpr_mul`. The leakage in `fpr_scaled` is also independently exploited by Kim et al. [KLL+26], who additionally exploit leakage in `fpr_add`, whereas the `fpr_mul` leakage studied here provides our additional coefficient-refinement source. The subsequent `ffLDL_fft` processes secret-derived values and should therefore also be considered sensitive, although side-channel attacks against it remain unexplored. Under `sign_dyn`, key expansion is repeatedly executed during signing, increasing the exposure of secret-dependent intermediate values.

Signing During signing, there are leakage points in the pre-image computation, fast Fourier sampling, and randomness generation. When performing pre-image computation, intermediate products involving public $\hat{c}$ and the secret key in $\hat{\mathbf{B}}$ are natural CPA targets and

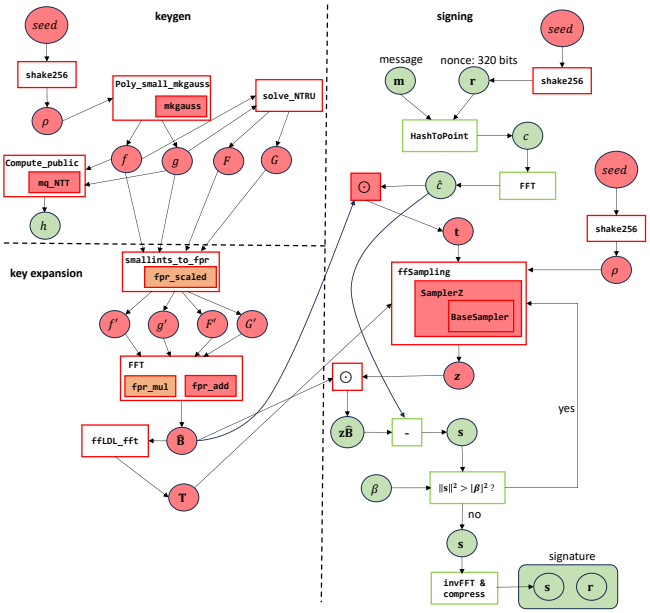


Figure 3: Sensitive data paths in Falcon key generation and signing. Red nodes and borders indicate sensitive variables and functions, while red-filled rectangles indicate functions already targeted by existing attacks and orange-filled rectangles highlight the leakage points identified in this work. Verification is omitted because it only processes public data.

have been exploited in prior work [KA21, GMRR22, CSM+25, CCCY26]. The `ffSampling` procedure and its internal `SamplerZ` and `BaseSampler` functions are also sensitive and have been attacked through half Gaussian and sign leakage [GMRR22, ZLYW23, LZY+25]. Its output $\mathbf{z}$ is sensitive because recovering it enables key recovery from the relation among $\mathbf{s}$, $\hat{c}$, and $\hat{\mathbf{B}}$. The randomness driving fast Fourier sampling should likewise be protected. By contrast, the signing nonce $\mathbf{r}$ is public but must remain unique, since nonce reuse may enable differential attacks [BDS23].

Overall, Falcon exposes sensitive paths across secret sampling, key expansion, FFT-domain processing, fast Fourier sampling, and randomness generation. Existing attacks already cover several of these components, while the FP conversion and multiplication leakages identified in this work further enlarge the known attack surface.

6.3 Countermeasures

Our attack exploits leakages from FP conversions and FP multiplications during the FFT and makes use of the distribution of the secret key. We focus only on implementation-level mitigations that reduce these exploitable leakages, as altering the key distribution would fundamentally change the Falcon design.

Lightweight Mitigation via Embedded Assembly While some leakage is unavoidable in unprotected FP conversions and FFTs, carefully designed embedded assembly can substantially mitigate it. This approach has already proven effective in defending against recent compiler-induced vulnerabilities in PQC KEMs [BBB+25, LZZ+25, DG26]. Compared to `-O3` and `-Os`, ASM Falcon effectively suppresses leakage in `FPR_NORM64` by using `clz` to count leading zeros. This degrades the initial coefficient-range classification and prevents our single-trace attack, making ASM Falcon preferable on Cortex-M4. However,

ASM Falcon could be further improved: the normalization procedure in its `fpr_mul` still generates an explicit mask with exploitable leakage. This could be fixed by adopting the `-O3`/`-Os` strategy of using `tst.w` and conditional `moveq` instructions. Finally, while assembly hardening increases attack complexity without performance overhead, it is a lightweight mitigation rather than a complete defense. An attacker can still bypass it using multiple traces via the MLE strategy; for instance, our multi-trace attack on ASM Falcon still achieves a 53/100 success rate with 10 traces, especially since the `sign_dyn` design repeatedly exposes key expansion.

The Ultimate Goal: Masking The masking countermeasure with provable security can effectively protect against our attack. Specifically, masking divides each secret coefficient into multiple random shares and makes the current attack infeasible. However, masking Falcon's FP arithmetic remains particularly challenging [CC24, CCCY26, KA24]. FP operations mix arithmetic computation with bit-level manipulation, requiring complex and secure conversions between masking domains.

In hardware, masked mantissa multiplication introduces substantial area and performance overheads and complicates glitch-resistant implementation [KA24]. In software, a first-order masking scheme increases the cycle count by approximately 23× for FP multiplication and 35× for FP addition [CC24]. Therefore, although masking offers stronger and more systematic protection, achieving efficient and comprehensive protection of Falcon's FP implementation remains an open challenge. This massive overhead highlights why exploring lighter, human-audited assembly mitigations remains highly relevant for resource-constrained platforms in the interim.

7 Concluding Remarks

We have presented the first single-trace side-channel attack on Falcon that succeeds even under the aggressive `-O3` compilation level. By exploiting leakage in FP conversion together with additional leakage from FP multiplication at the beginning of the FFT, we determine enough secret key coefficients to solve for the entire key via a system of linear equations. Our attack achieves 100% key recovery on an ARM Cortex-M4 with one power trace for Falcon-512 compiled at `-O0`, `-Os` and `-O3` under both `sign_tree` and `sign_dyn` designs. Compared with prior work, this approach not only drastically reduces the number of required traces but also extends the attack scope to highly optimized implementations.

These findings expose a critical vulnerability in Falcon's signing procedure that future standardization efforts must address. While the current ASM Falcon implementation resists our single-trace attacks and can be further hardened using compiler-inspired optimizations (`-O3`/`-Os`), this only serves as a lightweight mitigation rather than a true defense. It increases attack complexity but does not eliminate the vulnerability, because repeated key expansion under `sign_dyn` still enables multi-trace key recovery. Ultimately, securing Falcon's FP arithmetic requires efficient masking schemes, which remain a critical open challenge for future work due to their massive performance overhead on embedded devices.

Acknowledgment

Kang Li thanks the China Scholarship Council (CSC) for support under Grant No. 202406780039. This work was also supported in part by the Swedish Research Council (grant numbers 2021-04602 and 2023-03654), and the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.

References

[ACLZ20] Dorian Amiet, Andreas Curiger, Lukas Leuenberger, and Paul Zbinden. Defeating newhope with a single trace. In *International Conference on Post-Quantum Cryptography*, pages 189–205. Springer, 2020.

[BBB+25] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in kyber implementations. *IACR TCHES*, 2025(2):209–234, 2025.

[BDS23] Sven Bauer and Fabrizio De Santis. A differential fault attack against deterministic falcon signatures. In *International Conference on Smart Card Research and Advanced Applications*, pages 43–61. Springer, 2023.

[CC24] Keng-Yu Chen and Jiun-Peng Chen. Masking floating-point number multiplication and addition of falcon: First-and higher-order implementations and evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):276–303, 2024.

[CCCY26] Keng-Yu Chen, Ming Qing Ching, Jiun-Peng Chen, and Bo-Yin Yang. When masking multiplication isn't enough: Exploiting floating-point leakage in falcon's pre-image computation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):686–708, 2026.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[CSM+25] Shuyi Chen, Honglin Shao, Jingdian Ming, Yuejun Liu, Yiwen Gao, and Yongbin Zhou. Serp-sca: A strengthened side-channel attack framework for falcon signature. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

[DG26] Haiyue Dong and Qian Guo. Breaking optimized HQC: The first cache-timing full decryption oracle key-recovery attack in post-quantum cryptography. In *Advances in Cryptology – CRYPTO 2026*, Lecture Notes in Computer Science. Springer, 2026. In press.

[DN12] Léo Ducas and Phong Q Nguyen. Learning a zonotope and more: Cryptanalysis of ntrusign countermeasures. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 433–450. Springer, 2012.

[DP16] Léo Ducas and Thomas Prest. Fast fourier orthogonalization. In *Proceedings of the 2016 ACM International Symposium on Symbolic and Algebraic Computation*, pages 191–198, 2016.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on bliss lattice-based signatures: Exploiting branch tracing against strongswan and electromagnetic emanations in microcontrollers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1857–1874, 2017.

[FKT+20] Pierre-Alain Fouque, Paul Kirchner, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Key recovery from gram–schmidt norm leakage in hash-and-sign signatures over ntru lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 34–63. Springer, 2020.

[GMRR22] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on falcon. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 141–164, 2022.

[GO22] Si Gao and Elisabeth Oswald. A novel completeness test for leakage models and its application to side channel attacks and responsibly engineered simulators. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 254–283. Springer, Cham, May / June 2022.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the Fortieth Annual ACM symposium on Theory of Computing*, pages 197–206, 2008.

[GSA+25] Guillaume Goy, Maxime Spyropoulos, Nicolas Aragon, Philippe Gaborit, Renaud Pacalet, Fabrice Perion, Laurent Sauvage, and David Vigilant. Side-channel sensitivity analysis on hqc: Towards a fully masked implementation. *Cryptology ePrint Archive*, 2025.

[HBD+22] A. Hülsing, D. J. Bernstein, C. Dobraunig, M. Eichlseder, S. Fluhrer, S.-L. Gazdag, P. Kampanakis, S. Kölbl, T. Lange, M. M. Lauridsen, F. Mendel, R. Niederhagen, C. Rechberger, J. Rijneveld, P. Schwabe, J.-P. Aumasson, B. Westerbaan, and W. Beullens. Sphincs+. Technical report, National Institute of Standards and Technology, 2022. Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[HPRR20] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous gaussian sampling: from inception to implementation: With applications to the falcon signature scheme. In *International Conference on Post-Quantum Cryptography*, pages 53–71. Springer, 2020.

[KA21] Emre Karabulut and Aydin Aysu. Falcon down: Breaking falcon post-quantum signature scheme through side-channel attacks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 691–696. IEEE, 2021.

[KA24] Emre Karabulut and Aydin Aysu. Masking falcon's floating-point multiplication in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):483–508, 2024.

[KAA21] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-trace side-channel attacks on ω-small polynomial sampling: With applications to ntru, ntru prime, and crystals-dilithium. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 35–45. IEEE, 2021.

[KH18] Suhri Kim and Seokhie Hong. Single trace analysis on constant time cdt sampler and its countermeasure. *Applied Sciences*, 8(10):1809, 2018.

[KLL+26] GyuSang Kim, JeongHwan Lee, Myeonghoon Lee, Seokhie Hong, and HeeSeok Kim. Secret key recovery of falcon using simple power analysis in conditional calculator. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):953–975, 2026.

[KRSS19] Matthias J Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. 2019.

[LDK+22] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai. Crystals-dilithium. Technical report, National Institute of Standards and Technology, 2022. Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[LZY+25] Xiuhan Lin, Shiduo Zhang, Yang Yu, Weijia Wang, Qidi You, Ximing Xu, and Xiaoyun Wang. Thorough power analysis on falcon gaussian samplers and practical countermeasure. In *IACR International Conference on Public-Key Cryptography*, pages 229–258. Springer, 2025.

[LZZ+25] Zhenzhi Lai, Ruiyi Zhang, Zhiyuan Zhang, Julius Hermelink, Michael Schwarz, Van-Thuan Pham, and Udaya Parampalli. You only decapsulate once: Ciphertext-independent single-trace passive side-channel attacks on HQC. Cryptology ePrint Archive, Report 2025/2162, 2025.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 199–216. USENIX Association, August 2017.

[NR06] Phong Q Nguyen and Oded Regev. Learning a parallelepiped: Cryptanalysis of ggh and ntru signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 271–288. Springer, 2006.

[PFH+22] T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang. Falcon. Technical report, National Institute of Standards and Technology, 2022. Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[Por19] Thomas Pornin. New efficient, constant-time implementations of falcon. *Cryptology ePrint Archive*, 2019.

[PP19a] Peter Pessl and Robert Primas. More practical single-trace attacks on the number theoretic transform. In *International Conference on Cryptology and Information Security in Latin America*, pages 130–149. Springer, 2019.

[PP19b] Thomas Pornin and Thomas Prest. More efficient algorithms for the ntru key generation using the field norm. In *IACR International Workshop on Public Key Cryptography*, pages 504–533. Springer, 2019.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. Single-trace side-channel attacks on masked lattice-based encryption. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 513–533. Springer, 2017.

[Pra62] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.

[QA25] Jinyi Qiu and Aydin Aysu. Shift snare: Uncovering secret keys in falcon via single-trace analysis. *IACR Cryptol. ePrint Arch.*, 2025.

[QA26] Jinyi Qiu and Aydin Aysu. Ntt-peel: Bit shift side-channel in falcon's number theoretic transform. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):709–735, 2026.

[QLZ+24] Zehua Qiao, Yuejun Liu, Yongbin Zhou, Yuhan Zhao, and Shuyi Chen. Single trace is all it takes: Efficient side-channel attack on dilithium. *Cryptology ePrint Archive*, 2024.

[VCGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 282–296. Springer, 2014.

[ZLYW23] Shiduo Zhang, Xiuhan Lin, Yang Yu, and Weijia Wang. Improved power analysis attacks on falcon. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 565–595. Springer, 2023.