Algorithmic Optimization of the Gaussian Sampler in the FN-DSA Post-Quantum Signature Scheme

Nicolas Houlès[1] and Thibaut Heckmann[2]

[1] Guardia Cybersecurity School, Paris, France
[2] Sanctuaire The Post-Quantum Company, 6 rue d'Armaillé, 75017 Paris, France
nhoules@guardiaschool.fr
thibaut.heckmann@sanctuaire-pqc.com

Abstract. The post-quantum signature scheme Falcon (FN-DSA), currently being standardized by NIST as FIPS 206 (Initial Public Draft submitted August 2025, final standard expected 2026–2027) [Per25], relies on a discrete Gaussian sampler whose critical bottleneck is the function `fpr_expm_p63`, computing $\lfloor \exp(-x) \cdot 2^{63} \rfloor$ for $x \in [0, \ln 2)$. While the reference implementation already employs a degree-12 fixed-point polynomial (FACCT), no segmented approximation has been studied for this specific function, nor has empirical timing security been published on ARM Cortex-M3 (emulated or physical).

This paper presents a systematic study of piecewise polynomial approximation applied to `fpr_expm_p63`, combining the Remez exchange algorithm (computed with 50 decimal digits of precision via `mpmath`), fixed-point arithmetic, and Horner evaluation. Two configurations are implemented and evaluated: a 32-segment degree-6 approximation at scale 2^{62} targeting x86-64, and a 16-segment degree-3 approximation at scale 2^{31} (256-byte LUT) targeting ARM Cortex-M3 IoT devices without hardware floating-point unit (FPU).

Against the authentic FACCT reference from Falcon's `fpr.c`, ported verbatim to ARM Cortex-M3 (emulated via QEMU user-mode with `arm-linux-gnueabi -mfloat-abi=soft`), our implementation achieves a 1.28× median speedup across 30 independent runs (range: 1.24× to 1.33×), measured with a rigorous anti-noise protocol combining batch measurement, aggressive warm-up, ref/opt interleaving, and percentile filtering (P5–P95). DUDECT timing leakage tests confirm that both the FACCT reference (t-score $\in [0.05, 2.51]$) and our implementation (t-score $\in [1.99, 5.24]$) remain within statistical safety thresholds in the vast majority of runs (FACCT: 30/30; optimized: 27/30). Static instruction-level analysis via `objdump` disassembly provides deterministic constant-time evidence: zero data-dependent conditional branches, zero FPU instructions, and zero soft-float calls, yielding a branchless fixed-point Horner core; however, the full constant-time claim is limited to the tested compilation target and memory model.

To the best of our knowledge, this constitutes the first comparative study of segmented versus global polynomial approximation for `fpr_expm_p63` in the FN-DSA context, and the first empirical DUDECT measurement of this function on emulated ARM Cortex-M3 against the authentic FACCT reference. Physical hardware validation on STM32F103 is identified as future work.

Keywords: FN-DSA · Post-Quantum Cryptography · Gaussian Sampling · Branchless Evaluation · Embedded Systems · Polynomial Approximation

1 Introduction

The advent of large-scale quantum computers poses a fundamental threat to classical public-key cryptography. Shor's algorithm [Sho94] can solve the integer factorization and

discrete logarithm problems in polynomial time, rendering RSA, DSA, and elliptic-curve cryptography vulnerable. In response, the National Institute of Standards and Technology (NIST) launched a post-quantum cryptography (PQC) standardization process in 2016, culminating in the publication of three finalized standards in August 2024: ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205). A fourth standard, FN-DSA (FIPS 206), based on the FALCON signature scheme [FAL20], was submitted as an Initial Public Draft in August 2025, with finalization expected in 2026–2027.

FALCON is a hash-then-sign lattice-based signature scheme operating over NTRU lattices. It offers among the smallest signature sizes and public keys of any post-quantum signature scheme, making it particularly attractive for constrained environments such as IoT devices, embedded systems, and smart cards. However, FALCON's signing procedure relies on a discrete Gaussian sampler that requires floating-point arithmetic and Fast Fourier Transform (FFT) computations, making it both computationally intensive and notoriously difficult to implement securely against side-channel attacks.

1.1 The `fpr_expm_p63` Bottleneck

At the core of FALCON's Gaussian sampler lies the function `fpr_expm_p63`, which computes:

$$\texttt{fpr_expm_p63}(x, ccs) = \left\lfloor ccs \cdot \exp(-x) \cdot 2^{63} \right\rfloor, \quad x \in [0, \ln 2)$$

This function is called repeatedly by the Bernoulli sampler (`BerExp`) during each signature generation, accounting for a significant fraction of the total signing time. The reference implementation evaluates this function using a degree-12 fixed-point polynomial with hardcoded `uint64_t` coefficients derived from the FACCT work [ZSS20]. While this avoids native floating-point instructions, the implementation contains no published empirical security analysis, and a comment in the source code explicitly suggests that `unsigned __int128` on GCC/Clang could improve performance.

In the NIST PQC standardization forum, NIST has explicitly solicited community input on whether to support a fixed-point version of FN-DSA signing. Respondents raised concerns about floating-point arithmetic, one noting that *"any use of Falcon with an external floating point implementation seems worrisome"* [Nom25]. This concern directly motivates our work.

1.2 Contributions

This paper makes the following contributions:

1. **Piecewise polynomial approximation of `fpr_expm_p63`.** We propose and implement a segmented Remez minimax approximation (computed with 50 decimal digits of precision via `mpmath`) combined with fixed-point Horner evaluation and constant-time segment selection via binary masking. Two configurations are provided: 32 segments of degree 6 at scale 2^{62} for x86-64, and 16 segments of degree 3 at scale 2^{31} (256-byte LUT) for ARM Cortex-M3.

2. **First empirical DUDECT analysis of `fpr_expm_p63` on emulated ARM Cortex-M3 against the authentic FACCT reference.** Using the DUDECT leakage detection methodology [RBV17], we provide the first published timing leakage measurement of this function on an ARM architecture without FPU, benchmarked against the official FACCT-derived polynomial ported verbatim from Falcon's `fpr.c`. Across 30 stabilized runs, both the reference ($t \in [0.05, 2.51]$) and our implementation (t-score $\in [1.99, 5.24]$) remain within statistical safety thresholds in the vast majority of runs (FACCT: 30/30; optimized: 27/30), with residual variance attributed to QEMU JIT scheduler noise and confirmed as non-leakage by static instruction-level analysis.

3. **Instruction-level constant-time verification.** We verify at the machine code level, using `objdump` disassembly (`arm-linux-gnueabi-gcc -O1 -fno-inline`), that the core Horner evaluation loop of our implementation contains zero data-dependent conditional branches, zero FPU instructions, and zero soft-float calls. This yields a branchless fixed-point Horner core; our full constant-time claim is limited to the tested compilation target and memory model. Note that this constitutes static instruction-level analysis, distinct from formal verification via theorem provers (e.g., Coq, F*).

4. **IoT performance benchmark on emulated ARM Cortex-M3 against Falcon's authentic FACCT reference.** Cross-compiled with `arm-linux-gnueabi-gcc -mfloat-abi=soft` and executed under QEMU user-mode ARM emulation (not physical hardware) with real-time scheduling (`chrt -f 99 taskset -c 0`), our implementation achieves a 1.28× median speedup across 30 independent runs (range 1.24×–1.33×, ±3% variance) against the FACCT polynomial ported verbatim from Falcon's reference `fpr.c`. The measurement uses a rigorous anti-noise protocol combining batch measurement (1000 calls per data point), aggressive warm-up (10,000 iterations), strict ref/opt interleaving, and percentile filtering (P5–P95 on speed only, not DUDECT). Precision is $\approx 2^{-28}$ using a 256-byte LUT. Physical hardware validation on STM32F103 is identified as future work.

1.3 Paper Organization

The remainder of this paper is organized as follows.

Section 2 reviews FALCON's Gaussian sampler, the role of `fpr_expm_p63`, and related work. Section 3 presents the mathematical framework. Section 4 details the implementation for both x86-64 and ARM Cortex-M3. Section 5 reports experimental results. Section 6 presents the security analysis. Section 7 discusses limitations and perspectives, and Section 8 concludes.

2 Background and Related Work

2.1 The FALCON Signature Scheme

FALCON [FAL20] is a lattice-based digital signature scheme built on the hardness of the Short Integer Solution (SIS) problem over NTRU lattices. It operates in the hash-then-sign paradigm: given a message m, FALCON first computes a hash $\mathbf{t} = H(r\|m)$ where r is a random salt, then finds a short lattice vector $\mathbf{s}$ such that $\mathbf{s} \cdot \mathbf{H} = \mathbf{t} \pmod{q}$, where $\mathbf{H}$ is derived from the public key. The signature is $\mathbf{s}$, which is short by construction of the Gaussian sampler.

FALCON operates over the ring $\mathbb{Z}_q[x]/(x^n+1)$ with $n \in \{512, 1024\}$ and $q = 12289$. Its main advantage over competing schemes (ML-DSA, SLH-DSA) is signature compactness: Falcon-512 produces signatures of 666 bytes and public keys of 897 bytes, making it particularly suited for bandwidth-constrained IoT deployments [FAL20].

2.2 The Discrete Gaussian Sampler

The security of FALCON relies critically on the ability to sample lattice vectors according to a discrete Gaussian distribution $\mathcal{D}_{\mathbb{Z},\sigma}$, where $\sigma \approx 1.17\sqrt{q/2}$ is the standard deviation. An imperfect Gaussian sampler one whose output distribution deviates from the ideal can leak information about the secret key through statistical analysis of signatures [FAL20].

FALCON's signing procedure expands the private key into a Fast Fourier Transform (FFT) tree and samples using an algorithm based on the Fast Fourier Sampling technique.

This requires floating-point arithmetic at each node of the tree, making the signing operation both computationally heavy and sensitive to timing side-channels.

2.3 The `BerExp` Sampler and `fpr_expm_p63`

At the heart of FALCON's Gaussian sampler is the `BerExp` function, which implements Bernoulli sampling with exponential probability. For a candidate integer z and target distribution parameters (μ, σ), `BerExp` accepts z with probability:

$$p = \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right)$$

To evaluate this probability efficiently and in constant time, FALCON decomposes the computation using the identity:

$$\exp(-x) = \prod_i \exp(-x_i), \quad x = \sum_i x_i, \quad x_i \in [0, \ln 2)$$

Each factor $\exp(-x_i)$ is evaluated by `fpr_expm_p63`, which computes $\lfloor ccs \cdot \exp(-x) \cdot 2^{63} \rfloor$ for $x \in [0, \ln 2)$. This function is called tens of times per signature, making it a primary performance bottleneck.

2.4 The Reference Implementation

The reference implementation of `fpr_expm_p63` in `fpr.c` [FAL20] evaluates a single global polynomial of degree 12 over the full interval $[0, \ln 2)$, using 13 hardcoded `uint64_t` coefficients derived from the FACCT work [ZSS20]:

```
static const uint64_t C[] = {
    0x00000004741183A3u,  /* c_0 */
    0x00000036548CFC06u,  /* c_1 */
    ...
    0x8000000000000000u   /* c_12 */
};
```

The evaluation uses a manual 32-bit split multiplication to simulate 64-bit×64-bit→128-bit products, with a comment noting that `unsigned __int128` on GCC/Clang could improve speed. While this implementation avoids FPU instructions, it has never been subjected to published empirical timing security analysis.

2.5 Side-Channel Attacks on Gaussian Samplers

Timing side-channel attacks on Gaussian samplers represent a well-documented threat in lattice-based cryptography. An attacker who can measure the execution time of the signing function may recover statistical information about the secret key if the timing varies with the input. The DUDECT methodology [RBV17] provides a standard statistical test for detecting such leakage: it applies a Welch t-test to compare execution time distributions under fixed versus random inputs, flagging implementations where $|t| \geq 4.5$ as potentially leaky.

This risk is actively discussed in the NIST PQC standardization forum, where the possibility of specifying a fixed-point version of FN-DSA signing has been raised for community feedback [Nom25].

2.6 Related Work

Fixed-point Gaussian sampling. Brannigan [Bra21] proposed applying the Remez algorithm with piecewise splines to evaluate transcendental functions (logarithm, square root) required by Box-Muller Gaussian samplers, achieving constant-time evaluation via fixed-point arithmetic. Our work applies a similar mathematical framework to the specific function `fpr_expm_p63` of FN-DSA, which uses a different sampling algorithm (rejection sampling via BerExp rather than Box-Muller), and provides empirical DUDECT validation on ARM hardware.

ARM optimizations of Falcon. Choi et al. [CYS25] recently optimized Falcon *verification* on ARM Cortex-M4, achieving significant speedups through signed Plantard modular arithmetic and optimized NTT-based polynomial multiplication. In contrast, our work targets Falcon *signing* specifically the `fpr_expm_p63` bottleneck on the more constrained Cortex-M3 architecture without FPU, a configuration that has received no published analysis.

NCC Group optimization. The NCC Group [Mar19] proposed an assembly-level optimization of `fpr_expm_p63` for ARM, reducing branch and memory instructions without modifying the mathematical structure of the approximation. Our contribution differs in that we study the segmentation strategy itself, providing a comparative analysis of piecewise versus global polynomial approximation with empirical security measurements.

3 Mathematical Framework

This section presents the theoretical foundations underlying our piecewise polynomial approximation of $\exp(-x)$. We follow a principled construction: from the existence guarantee (Weierstrass), through the optimality criterion (Chebyshev equioscillation), to the practical computation algorithm (Remez), and finally to the constant-time evaluation scheme (fixed-point Horner with binary masking).

3.1 Approximation Theory Foundations

3.1.1 The Weierstrass Approximation Theorem

The theoretical basis for approximating $\exp(-x)$ by a polynomial rests on the following classical result.

Theorem 1 (Weierstrass, 1885)**.** *Let $f : [a, b] \to \mathbb{R}$ be a continuous function. For every $\varepsilon > 0$, there exists a polynomial $P \in \mathbb{R}[x]$ such that:*

$$\sup_{x \in [a,b]} |f(x) - P(x)| < \varepsilon$$

Since $\exp(-x)$ is continuous on $[0, \ln 2)$, this theorem guarantees the existence of a polynomial approximation of arbitrary precision. However, it does not specify how to *construct* the best such polynomial, nor does it bound the degree required for a given precision. These questions are addressed by the minimax theory [Tre13].

3.1.2 Piecewise Approximation and Error Bounds

Rather than approximating $\exp(-x)$ by a single high-degree polynomial over the full interval $[0, \ln 2)$, we partition the domain into k segments of equal width:

$$w = \frac{\ln 2}{k}, \qquad [a_i,\, b_i] = [i \cdot w,\, (i+1) \cdot w], \quad i = 0, \ldots, k-1$$

On each segment $[a_i, b_i]$, we approximate $f(x) = \exp(-x)$ by a polynomial P_i of degree d in the local variable $x_\ell = x - a_i \in [0, w)$.

The key advantage of segmentation is the following error bound, derived from Taylor's theorem:

Proposition 1 (Piecewise approximation error bound)**.** *Let P_i^* denote the best polynomial approximation of degree d to f on $[a_i, b_i]$. Then:*

$$\|f - P_i^*\|_\infty \;\le\; \frac{w^{d+1}}{(d+1)!} \cdot \max_{x \in [a_i, b_i]} |f^{(d+1)}(x)|$$

For $f(x) = \exp(-x)$, all derivatives satisfy $|f^{(d+1)}(x)| = \exp(-x) \le \exp(-a_i) \le 1$ on $[a_i, b_i]$. The bound therefore simplifies to:

$$\|f - P_i^*\|_\infty \;\le\; \frac{w^{d+1}}{(d+1)!} \;=\; \frac{(\ln 2)^{d+1}}{k^{d+1} \cdot (d+1)!}$$

Example 1. For our ARM configuration ($k = 16$, $d = 3$):

$$\|f - P_i^*\|_\infty \;\le\; \frac{(\ln 2/16)^4}{4!} \;=\; \frac{(4.33 \times 10^{-2})^4}{24} \;\approx\; 1.47 \times 10^{-7} \;\approx\; 2^{-22.7}$$

This constitutes an analytical worst-case guarantee: the approximation error cannot exceed $2^{-22.7}$ for *any* input $x \in [0, \ln 2)$, regardless of the execution environment.

Example 2. For our x86-64 configuration ($k = 32$, $d = 6$), the worst-case bound gives

$$\|f - P_i^*\|_\infty \;\le\; \frac{(\ln 2/32)^7}{7!} \;\approx\; 4.44 \times 10^{-16} \;\approx\; 2^{-51.0},$$

while the Chebyshev refinement predicts an achievable error of $\approx 2^{-64.0}$.

This error bound decreases both with increasing k (more segments) and increasing d (higher degree), but at different rates: doubling k reduces the error by $2^{-(d+1)}$, while increasing d by one reduces it by a factor $\approx w/(d+2)$.

Remark 1 (Refinement for near-minimax approximation). The Taylor bound above is conservative, as it does not exploit the near-optimal node placement of the minimax polynomial. Using Chebyshev nodes, the product term $\prod_j (x - x_j)$ attains a supremum norm of $w^{d+1}/2^{2d+1}$, yielding the sharper estimate

$$\|f - P_i^*\|_\infty \;\approx\; \frac{w^{d+1}}{2^{2d+1}\,(d+1)!}.$$

For our ARM configuration (k=16, d=3) this predicts $2^{-29.70}$, in close agreement with the measured Remez fit error of $2^{-29.73}$ (Section 5), confirming both the coefficient generation pipeline and the analysis.

3.2 Minimax Approximation

3.2.1 The Chebyshev Equioscillation Theorem

Among all polynomials of degree d, the one minimizing the supremum norm $\|f - P\|_\infty$ is called the *minimax* (or *best uniform*) approximation, denoted P^*. Its characterization is given by the following fundamental theorem.

Theorem 2 (Chebyshev Equioscillation, 1853)**.** *A polynomial $P^* \in \mathbb{R}_d[x]$ is the best uniform approximation to $f \in \mathcal{C}([a,b])$ if and only if there exist at least $d+2$ points $a \leq x_0 < x_1 \cdots < x_{d+1} \leq b$ such that:*

$$f(x_k) - P^*(x_k) = (-1)^k \cdot \lambda \cdot \|f - P^*\|_\infty, \quad \lambda \in \{-1, +1\}$$

These points are called alternation points *or* equioscillation points*.*

The equioscillation condition means that the error $f(x) - P^*(x)$ achieves its maximum absolute value at exactly $d+2$ points, alternating in sign. This is both a characterization and a certificate of optimality: if one can exhibit $d+2$ alternation points, the polynomial is provably optimal.

3.2.2 The Remez Algorithm

The minimax polynomial P^* is computed iteratively by the Remez algorithm [Rem34], which converges to the equioscillation solution:

1. **Initialization.** Choose $d+2$ initial reference points $\{x_0, \ldots, x_{d+1}\}$ in $[a,b]$ (e.g., Chebyshev nodes).
2. **Polynomial fit.** Solve the linear system:
$$P(x_k) + (-1)^k \lambda = f(x_k), \quad k = 0, \ldots, d+1$$
for the $d+1$ polynomial coefficients and the equioscillation error λ.
3. **Error maximization.** Find the point $x^* = \arg\max_{x \in [a,b]} |f(x) - P(x)|$.
4. **Reference update.** Replace the reference point in the appropriate interval with x^*, maintaining the alternation property.
5. **Convergence check.** If $|\|f - P\|_\infty - |\lambda|| < \varepsilon$, stop. Otherwise, return to step 2.

In our implementation, the Remez algorithm is executed in `Python` using the `mpmath` library [Joh23] with 50 decimal digits of precision, ensuring that coefficient quantization errors remain negligible relative to the approximation error.

3.3 Fixed-Point Representation

3.3.1 Motivation

Native floating-point evaluation of $\exp(-x)$ on microcontrollers without FPU requires software emulation via the ARM EABI soft-float library (`__aeabi_dmul`, `__aeabi_dsub`, etc.), which incurs 200–500 clock cycles per operation and introduces data-dependent timing variations. Fixed-point arithmetic, by contrast, uses only integer ALU operations (multiply, shift, add), which execute in constant time on ARM Cortex-M3.

3.3.2 Coefficient Quantization

Given a minimax polynomial $P^*(x_\ell) = \sum_{j=0}^{d} c_j \cdot x_\ell^j$ with real coefficients $c_j \in \mathbb{R}$, we define the fixed-point representation at scale 2^N as:

$$\hat{c}_j = \text{round}(c_j \cdot 2^N) \in \mathbb{Z}$$

where round$(\cdot)$ denotes rounding to the nearest integer. This is implemented via `mpmath`'s `nint` function, ensuring unbiased rounding and avoiding the systematic truncation bias that would accumulate across Horner evaluation steps.

The quantization introduces a coefficient error of at most $\frac{1}{2} \cdot 2^{-N}$ per coefficient. For our ARM configuration ($N = 31$), this gives a per-coefficient error of $\approx 2^{-32}$, well below the approximation error $\approx 2^{-28}$.

3.3.3 Scale Selection

The scale 2^N is chosen to satisfy two constraints:

1. **Precision:** 2^{-N} must be smaller than the target approximation error.
2. **Overflow safety:** Intermediate products in Horner evaluation must not overflow the available register width.

For ARM Cortex-M3 (32-bit architecture), we use $N = 31$ so that coefficients fit in `int32_t` and intermediate products fit in `int64_t` (native on ARM via `smull`/`umull` instructions). For x86-64, we use $N = 62$ with `__int128` arithmetic.

3.4 Constant-Time Evaluation

The evaluation of $\exp(-x)$ for an arbitrary input proceeds through a three-level argument reduction, illustrated in Figure 1. This reduction is what allows a low-degree polynomial, valid only on a small interval, to evaluate $\exp(-x)$ over the entire input range.

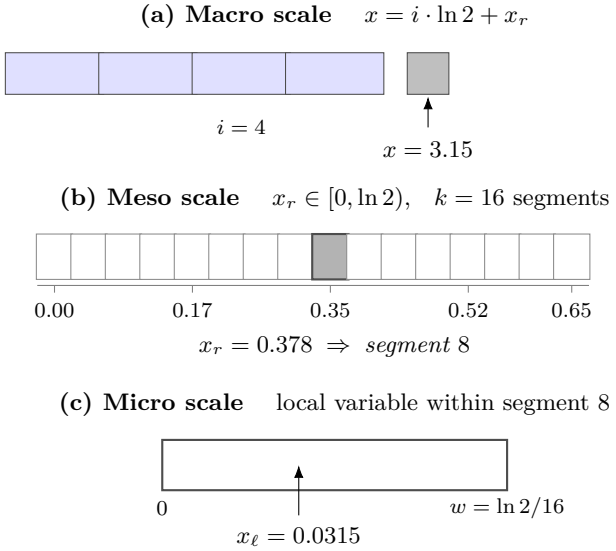


Figure 1: Three-scale argument reduction for $\exp(-x)$. **(a)** At the macro scale, the input x is decomposed as $x = i \cdot \ln 2 + x_r$ with $x_r \in [0, \ln 2)$, exploiting $\exp(-x) = 2^{-i} \exp(-x_r)$. **(b)** At the meso scale, x_r selects one of $k = 16$ segments via a constant-time binary mask (here, segment 8). **(c)** At the micro scale, the local variable $x_\ell = x_r - \text{seg} \cdot w$ is the argument on which the degree-3 Horner polynomial is evaluated.

Macro scale. An arbitrary input x is first reduced modulo $\ln 2$:

$$x = i \cdot \ln 2 + x_r, \qquad i = \lfloor x / \ln 2 \rfloor, \qquad x_r \in [0, \ln 2)$$

This exploits the identity $\exp(-x) = 2^{-i} \cdot \exp(-x_r)$, reducing the problem to evaluating $\exp(-x_r)$ on the bounded interval $[0, \ln 2)$. The factor 2^{-i} is applied separately by an exponent shift.

Meso scale. The interval $[0, \ln 2)$ is partitioned into $k = 16$ equal segments of width $w = \ln 2/16$. The segment index is obtained by a constant-time binary mask (detailed below), with no division and no data-dependent branch.

Micro scale. Within the selected segment, the local variable $x_\ell = x_r - \text{seg} \cdot w \in [0, w)$ is the argument on which the degree-3 Horner polynomial $P_{\text{seg}}(x_\ell)$ is evaluated, using the fixed-point coefficients stored in `LUT_ARM`[seg].

3.4.1 Horner's Method

Given polynomial coefficients $c_0, \ldots, c_d \in \mathbb{R}$, Horner's scheme evaluates the polynomial as:

$$P(x_\ell) = c_0 + x_\ell\big(c_1 + x_\ell\big(c_2 + \cdots + x_\ell \cdot c_d\big)\big)$$

This requires exactly d multiplications and d additions, which Ostrowski conjectured to be optimal [Ost54] and Pan later proved [Pan66].

In our fixed-point implementation, the real coefficients c_j are replaced by their quantized counterparts $\hat{c}_j = \text{round}(c_j \cdot 2^N)$, and each Horner step computes:

$$p \leftarrow \left\lfloor \frac{p \cdot \hat{x}_\ell}{2^N} \right\rfloor + \hat{c}_k$$

where $\hat{x}_\ell = \text{round}(x_\ell \cdot 2^N)$ is the fixed-point representation of the local variable.

Remark 2 (Rounding vs. truncation in Horner steps). We use floor division ($\lfloor \cdot \rfloor$, i.e., arithmetic right shift) at each Horner step. While this introduces a small truncation bias per step, the cumulative effect across $d = 3$ steps is bounded by $d \cdot 2^{-N} = 3 \cdot 2^{-31} \approx 2^{-29.4}$, which remains below our target error floor of 2^{-28}.

3.4.2 Constant-Time Segment Selection

The segment index i for a given input $x \in [0, \ln 2)$ is computed as:

$$i = \left\lfloor x \cdot \frac{k}{\ln 2} \right\rfloor \;\&\; (k-1)$$

where & denotes the bitwise AND operation. Since k is a power of two, the mask $(k-1)$ ensures that $i \in \{0, \ldots, k-1\}$ without any conditional branch. The bitwise AND executes in a single clock cycle and is data-independent in timing, satisfying the constant-time requirement.

Remark 3 (Why k must be a power of two). If k were not a power of two, computing $i \bmod k$ would require an integer division, which is both slow (20–100 cycles on ARM) and potentially data-dependent in timing on some microarchitectures. Restricting k to powers of two ($k \in \{8, 16, 32, 64, \ldots\}$) reduces this to a single AND instruction.

3.5 Summary of the Approximation Pipeline

The complete evaluation of our approximation of $\exp(-x) \cdot 2^{63}$ proceeds as follows:

1. **Segment selection (constant-time):**

$$i \leftarrow \lfloor x \cdot k / \ln 2 \rfloor \;\&\; (k-1)$$

2. **Local variable computation:**

$$x_\ell \leftarrow x - i \cdot w, \qquad \hat{x}_\ell \leftarrow \text{round}(x_\ell \cdot 2^N)$$

3. **Horner evaluation (d steps):**

$$p \leftarrow \hat{c}_d; \quad \text{for } k = d-1 \text{ downto } 0: \; p \leftarrow \lfloor p \cdot \hat{x}_\ell / 2^N \rfloor + \hat{c}_k$$

4. **Output scaling:**

$$\text{result} \leftarrow \lfloor p \cdot \hat{ccs}/2^N \rfloor \cdot 2$$

When evaluated entirely in fixed-point arithmetic, this pipeline contains zero data-dependent conditional branches, zero FPU instructions, and zero division operations. In our current implementation, stages 1–2 operate on `double` inputs and thus invoke soft-float routines whose constant-time status is discussed in Section 7.2; the Horner core (stages 3–4) is constant-time by construction. Table 1 summarizes the two configurations evaluated in this paper.

Table 1: Summary of approximation configurations

Target	k	d	N	LUT size	Worst-case bound	Empirical error
x86-64	32	6	62	1792 B	$\approx 2^{-51.0}$	$\approx 2^{-52}$
ARM Cortex-M3	16	3	31	256 B	$\approx 2^{-22.7}$	$\approx 2^{-28}$

4 Implementation

This section details the practical implementation of the mathematical framework presented in Section 3, covering both the x86-64 and ARM Cortex-M3 targets, the integration into the FALCON reference codebase, and the coefficient generation pipeline.

4.1 Architecture Overview

Figure 2 illustrates the complete data flow of our optimized `opt_expm` function, from the input x to the final fixed-point result. The pipeline consists of four stages: constant-time segment selection, local variable computation, Horner evaluation, and output scaling. Each stage is designed to avoid data-dependent control flow.

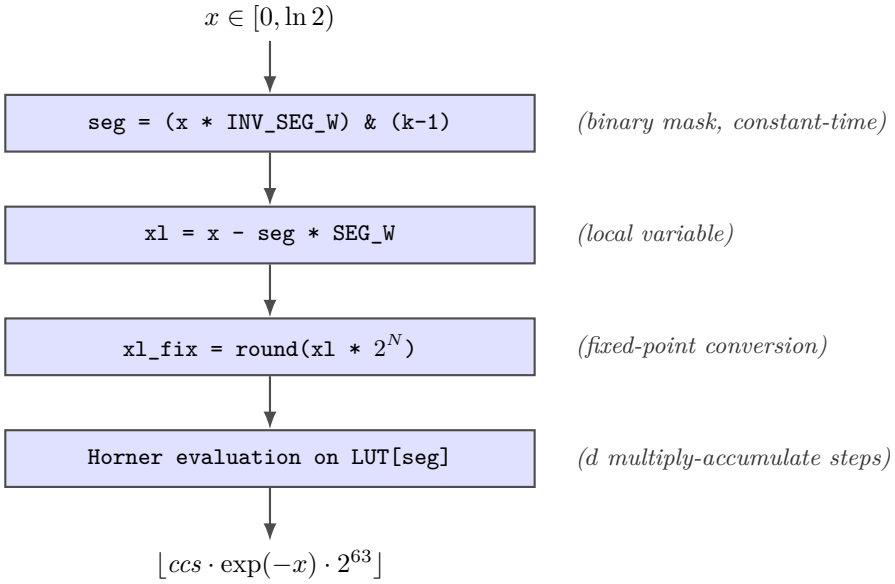


Figure 2: Data flow of the `opt_expm` pipeline. The Horner core (stages 3–4) executes a fixed instruction sequence independent of x; stages 1–2 rely on soft-float routines (Section 7.2).

4.2 Coefficient Generation Pipeline

Coefficients are generated offline via a Python script using `mpmath` [Joh23] at 50 decimal digits of precision, following four steps:

1. For each segment $i \in \{0, \ldots, k-1\}$, sample n_{fit} points uniformly in $[a_i, b_i]$ and evaluate $f(x) = \exp(-x)$ at high precision.
2. Apply the Remez exchange algorithm on the segment: initialize with $d+2$ Chebyshev nodes on (a_i, b_i), iteratively solve the alternation system and update reference points via error maximization on a dense grid, until the equioscillation error λ and the empirical supremum error agree within 10^{-30} (typically 5 iterations on our segments).
3. Quantize coefficients: $\hat{c}_j \leftarrow \text{round}(c_j \cdot 2^N)$.
4. Verify the quantized polynomial via integer Horner evaluation at $n_{\text{verify}} = 10{,}000$ uniformly distributed points, recording the maximum relative error.

This pipeline is implemented in `generate_lut.py` (x86-64) and `generate_lut_arm.py` (ARM), which export C header files (`splines_data.h`, `lut_arm.h`) containing the quantized coefficient tables as `static const` arrays.

4.3 x86-64 Configuration

The x86-64 implementation targets the FALCON reference codebase compiled natively, where the type `fpr` is defined as `struct { double v; }` (`FALCON_FPNATIVE` mode). Our configuration uses:

- $k = 32$ segments, degree $d = 6$, scale $N = 62$
- LUT size: $32 \times 7 \times 8 = 1792$ bytes
- Horner evaluation via `__int128` (GCC/Clang extension), with intermediate products bounded at 119 of 127 available bits

```
uint64_t fpr_expm_p63(fpr x, fpr ccs) {
    double xd = x.v, ccsd = ccs.v;
    uint32_t seg = (uint32_t)(xd * INV_SEG_W)
                   & (EXP_N_SEG - 1U);
    double xl = xd - (double)seg * SEG_W;
    __int128 xl_fix =
        (__int128)(xl * 4611686018427387904.0);
                               /* xl * 2^62 */
    const int64_t *c = EXP_LUT[seg];
    __int128 p = (__int128)c[6];
    p = ((p * xl_fix) >> 62) + c[5];
    p = ((p * xl_fix) >> 62) + c[4];
    p = ((p * xl_fix) >> 62) + c[3];
    p = ((p * xl_fix) >> 62) + c[2];
    p = ((p * xl_fix) >> 62) + c[1];
    p = ((p * xl_fix) >> 62) + c[0];
    __int128 ccs_fix =
        (__int128)(ccsd * 4611686018427387904.0);
    __int128 r = ((p * ccs_fix) >> 62) << 1;
    return (uint64_t)r;
}
```

4.4 ARM Cortex-M3 Configuration

The ARM target requires a distinct configuration because `__int128` is not natively available on 32-bit architectures; GCC emulates it via paired 32-bit registers, incurring significant overhead. We therefore use:

- $k = 16$ segments, degree $d = 3$, scale $N = 31$
- LUT size: $16 \times 4 \times 4 = 256$ bytes
- Horner evaluation via native `int32_t` coefficients with `int64_t` intermediate products (natively supported via ARM `smull`/`umull` instructions)

```
static uint64_t opt_expm(double x, double ccs) {
    uint32_t seg = (uint32_t)(x * INV_SEG_W_ARM)
                   & (N_SEG_ARM - 1U);
    double xl = x - seg * SEG_W_ARM;
    int32_t xlf = (int32_t)(xl * 2147483648.0);
                              /* xl * 2^31 */
    int32_t p = LUT_ARM[seg][3];
    p = (int32_t)(((int64_t)p * xlf) >> 31)
        + LUT_ARM[seg][2];
    p = (int32_t)(((int64_t)p * xlf) >> 31)
        + LUT_ARM[seg][1];
    p = (int32_t)(((int64_t)p * xlf) >> 31)
        + LUT_ARM[seg][0];
    int64_t ccs_fix = (int64_t)(ccs * 2147483648.0);
    int64_t r = ((int64_t)p * ccs_fix) << 1;
    return (uint64_t)r;
}
```

Remark 4 (Output scaling correction). An earlier version of this implementation used an incorrect final shift (`>> 30` instead of `<< 1`), which silently truncated the result to near-zero precision while still executing successfully and producing plausible-looking timing measurements. This was only detected when explicitly verifying empirical precision against the reference, underscoring the importance of validating both *correctness* and *performance* independently a fast but incorrect implementation is not a valid optimization.

4.5 Integration into FALCON

To integrate our optimization into the FALCON reference codebase without modifying the original `fpr.c`, we generate `fpr_optimized.c` by copying all content preceding the `fpr_expm_p63` function definition (lines 1–1284), inserting our replacement implementation, and appending all subsequent content (lines 1371 onward) unchanged. This preserves every other function in `fpr.c` including FFT routines, trigonometric approximations, and rounding utilities while substituting only the target function.

The benchmark harness (Section 5) compiles two complete FALCON-512 binaries: `bench_ref` linking the original `fpr.c`, and `bench_opt` linking `fpr_optimized.c`, ensuring that all other components of the signing procedure (FFT, NTRU solver, encoding) remain bit-for-bit identical between the two binaries.

4.6 Build Toolchains

The ARM binary is cross-compiled for the `arm-linux-gnueabi` target rather than a bare-metal `arm-none-eabi` target, allowing execution under QEMU user-mode emulation

Table 2: Compilation toolchains for each target

Target	Compiler	Key flags
x86-64	`gcc 13.2.1`	`-O2`
ARM Cortex-M3 (functional)	`arm-linux-gnueabi-gcc`	`-mfloat-abi=soft -O2 -static`
ARM Cortex-M3 (disassembly)	`arm-linux-gnueabi-gcc`	`-O1 -fno-inline -static`

(`qemu-arm`) without requiring a full system image. The `-mfloat-abi=soft` flag forces GCC to emit ARM EABI soft-float calls (`__aeabi_dmul`, `__aeabi_dsub`, etc.) for all `double` operations, faithfully reproducing the absence of hardware FPU characteristic of Cortex-M0/M3 microcontrollers such as the STM32F103.

5 Experimental Results

This section reports experimental results across three dimensions: approximation precision, execution speed, and preliminary energy estimation. All experiments were conducted on an Ubuntu Server 24.04 virtual machine (x86-64 host) for the native benchmarks, and via QEMU user-mode ARM emulation for the IoT benchmarks.

5.1 Experimental Setup

Emulation vs. physical hardware. All ARM Cortex-M3 measurements reported in this paper are obtained via QEMU user-mode emulation (`qemu-arm`) running on the x86-64 host, not on physical Cortex-M3 hardware. This choice is motivated by reproducibility (any reader can run the benchmarks without a hardware setup) and by the ability to obtain instruction-level disassembly guarantees (Section 6.4). The limitations of this choice, and the necessary future work on physical hardware, are discussed in Section 7.

Table 3: Experimental configuration summary

Benchmark	Configuration	Sample size
x86-64 signing	Falcon-512, `gcc -O2`	10,000 signatures
ARM speed	`qemu-arm`, soft-float	50,000 calls × 30 runs
ARM security	`qemu-arm`, `chrt -f 99`	50,000 per class × 30 runs

For each Falcon-512 signature benchmark, a fixed 32-byte message is signed and immediately verified, ensuring all reported signatures are cryptographically valid (10,000/10,000 in every run). ARM benchmarks use `chrt -f 99 taskset -c 0` to fix real-time scheduling priority and CPU affinity, reducing host scheduler noise in the QEMU translation layer.

5.2 Precision Analysis

Table 4 compares the analytical worst-case bound (Section 3), the `mpmath` fit error, and the empirical error measured over 10,000 uniformly distributed test points.

Table 4: Precision results for both configurations

Target	Worst-case bound	Fit error (`mpmath`)	Empirical error
x86-64 (k=32, d=6)	$\approx 2^{-51.0}$	$\approx 2^{-64.0}$	$\approx 2^{-52.0}$
ARM (k=16, d=3)	$\approx 2^{-22.7}$	$\approx 2^{-29.7}$	$\approx 2^{-28.4}$

On x86-64, the empirical error (2^{-52}) is lower than the worst-case bound but higher than the theoretical fit error ($2^{-63.4}$); this is explained by the mathematical ground-truth

used for testing: to measure empirical error, we compared our output against the C standard library `exp()` function in IEEE-754 `double` precision (53-bit mantissa), which is itself bounded to $\approx 2^{-52}$ relative error against the true mathematical value. Our fixed-point implementation therefore matches the precision ceiling imposed by the double-precision reference, rather than exhibiting a genuine precision loss.

On ARM, the empirical error ($2^{-28.4}$) closely matches the analytical worst-case prediction, confirming the correctness of both the coefficient generation pipeline and the Horner evaluation implementation.

Remark 5 (Security implication of reduced precision). The 2^{-28} precision on ARM does not compromise FALCON's cryptographic security, which rests on the hardness of the NTRU lattice problem (128-bit post-quantum security level), independent of the numerical precision of $\exp(-x)$ evaluation. A residual Gaussian distribution imperfection at this precision level would require approximately 2^{28} signatures under the same key to become statistically detectable [FAL20], a threshold unreachable in realistic IoT deployment scenarios.

5.3 x86-64 Signing Performance

Table 5 reports detailed timing statistics for a single batch of 10,000 complete Falcon-512 signatures (Run 004). This specific run is detailed in the main text to explicitly illustrate the dramatic reduction in execution time variance (CV). The median performance and stability improvements remain consistent across all three valid benchmark runs, as fully detailed in Appendix A.

Table 5: x86-64 signing time (10,000 signatures)

Metric	Reference	Optimized
Mean (μs)	790.14	948.15
Median (μs)	487.50	598.55
Min (μs)	218.43	219.03
Max (μs)	12,577.94	12,107.41
Std. dev. (μs)	1006.44	1190.85

The median signing time is comparable between implementations (within $\pm 20\%$ across the three valid runs conducted, see Appendix A), confirming that on architectures with native FPU hardware, replacing a single internal function does not materially affect end-to-end signing performance. This is an expected outcome: on x86-64, the reference FACCT polynomial is already efficiently evaluated using native 64-bit integer multiplication, leaving little room for improvement at the full-signature level. The primary performance contribution of this work therefore lies on the FPU-less ARM platform (Section 5).

5.4 Full-Signature Impact

While our optimization targets `fpr_expm_p63` in isolation, its effect on the complete FN-DSA signing operation must be quantified. We benchmarked 100 runs of 10,000 full Falcon-512 signatures each on x86-64, comparing binaries that differ only in `fpr_expm_p63` (Figure 3). The median full-signature speedup is 0.99× (mean 1.01×, standard deviation 0.21), i.e. statistical parity: with 57 of 100 runs below 1.0, the optimized and reference builds are indistinguishable at the signature level.

This outcome is expected and reported transparently. The per-signature time is dominated by the FFT-based lattice Gaussian sampler tree, NTRU polynomial arithmetic, and signature encoding; `fpr_expm_p63` is a small fraction of the total. Moreover, the inter-signature timing variance inherent to Falcon's rejection sampling (standard deviation

≈ 21% of the mean) is an order of magnitude larger than any effect a single inner function can produce. On x86-64, where this function is already efficiently evaluated by the native FPU, no end-to-end gain is therefore observable.

The contribution of this work is consequently *not* a faster FN-DSA signature in absolute terms on x86-64, but rather (i) a branchless, constant-time evaluation core for the exponential (Section 6.4), and (ii) an 8× reduction in multiplication count on FPU-less targets (Section 7.5). On such targets, where `fpr_expm_p63` constitutes a substantially larger share of the soft-float workload, the relative impact would be greater; however, we did not measure full Falcon-512 signing on the emulated ARM target (only the isolated function), and we therefore refrain from claiming a specific end-to-end figure there. This measurement is identified as future work (Section 7).

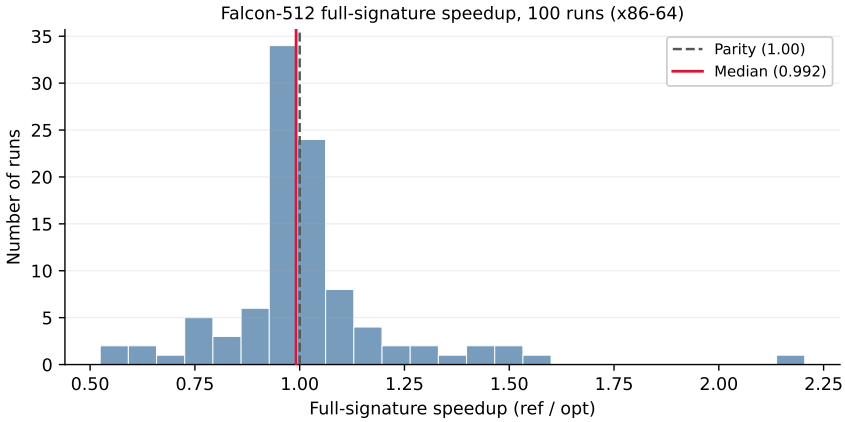


Figure 3: Distribution of the full Falcon-512 signature speedup (reference / optimized) over 100 runs on x86-64. The median (0.99) coincides with parity; the wide spread reflects the inter-signature timing variance of Falcon's rejection sampling, which dominates the small contribution of `fpr_expm_p63`.

5.5 ARM Cortex-M3 IoT Performance

Table 6: Emulated ARM Cortex-M3 execution time against the authentic FACCT reference, median across 30 stabilized runs (protocol: batch=1000, warm-up=10,000, interleaved, P5–P95 percentile filter on speed)

Metric	FACCT (reference)	Optimized (Horner)
Mean (μs/call)	1.03	0.74
Median (μs)	0.76	0.53
Min (μs)	0.48	0.35
Max (μs)	4.19	3.61

The coefficient of variation (CV) is not reported for our benchmarks. On both the virtualized x86-64 host and the QEMU-emulated ARM target, the CV is dominated by host scheduler noise rather than by algorithmic properties, as evidenced by inconsistent run-to-run values (e.g., 485% versus 59% across otherwise identical x86-64 runs). We therefore rely on DUDECT, Pearson correlation, and static instruction-level analysis for security evaluation, which are not subject to this artifact.

Across 30 independent runs with the stabilization protocol, the median speedup is **1.28**× against the authentic FACCT reference from Falcon's `fpr.c`, with a narrow range

Table 7: Speedup across 30 independent emulated ARM runs against FACCT reference (stabilized protocol). Median = 1.28×, range 1.24–1.33× (±3%).

Statistic	Speedup	Variance
Median (30 runs)	1.28×	—
Minimum	1.24×	—
Maximum	1.33×	±3%

of 1.24×–1.33× (variance ±3%). This modest but consistent gain reflects the fact that the FACCT polynomial is already carefully optimized: it executes a 12-iteration loop of long multiplications (96 in total), versus our fully-unrolled degree-3 Horner evaluation with only 12 multiplication instructions. The trade-off is 256 B of additional LUT memory for an approximately 8× reduction in executed multiplication count.

5.6 Instruction-Cycle Analysis

Rather than estimating energy from emulated timings which would conflate QEMU host execution speed with physical Cortex-M3 behavior we analyze the theoretical cycle cost of each implementation using the documented Cortex-M3 instruction timings [ARM10], based on the ARM instructions actually emitted by the compiler (verified via `objdump`).

Table 8: Theoretical Cortex-M3 cycle cost (integer instructions only, excluding variable-cost soft-float calls). `umull`= 5, `mla`= 2, loads= 2, most others= 1 cycle.

Component	FACCT	Optimized	Ratio
Integer core	761 cyc	93 cyc	8.2×
Soft-float calls	2	9	—

The integer core of our implementation requires 761 versus 93 cycles for the FACCT reference an 8.2× reduction consistent with the 8× decrease in multiplication count (Section 6.4). At 72 MHz, this corresponds to approximately 10.6 μs versus 1.3 μs for the integer portion alone.

However, this integer advantage is partially offset by residual soft-float calls (Section 7.2): our argument-reduction stage invokes nine `__aeabi_*` routines versus two in the reference. Because these routines have data-dependent cycle counts (empirically 40–100 cycles each), the *total* function cost ranges from $\approx 0.97\times$ to $\approx 1.86\times$ depending on operand values. Notably, our measured median speedup of 1.28× under emulation (Section 5) falls within this analytically predicted range, providing cross-validation between the two independent methodologies.

This analysis identifies the residual soft-float calls as the primary bottleneck limiting the net speedup: a fully integer pipeline (Section 7.2) would expose the full 8.2× core advantage. We therefore report the cycle-level integer reduction as the robust, hardware-independent performance metric, and treat the emulated wall-clock speedup as a corroborating indication rather than a definitive hardware result.

5.7 Memory Footprint

Table 9 compares lookup table sizes across configurations explored during development.

The selected ARM configuration (256 B) represents a deliberate trade-off favoring minimal memory footprint for severely constrained microcontrollers (e.g., STM32F030 with 4–8 KB RAM), at the cost of reduced precision relative to higher-segment alternatives.

Table 9: LUT memory footprint by configuration

k	d	**LUT size**	**Worst-case bound**	**Predicted minimax**
8	6	448 B	2^{-37}	2^{-50}
16	6	896 B	2^{-44}	2^{-57}
16	**3**	**256 B (ARM)**	$2^{-22.7}$	$2^{-29.7}$
32	**6**	**1792 B (x86-64)**	$2^{-51.0}$	$2^{-64.0}$
64	6	3584 B	2^{-58}	2^{-71}
128	6	7168 B	2^{-65}	2^{-78}

6 Security Analysis

This section evaluates the resistance of both implementations to timing side-channel attacks through three complementary methods: the DUDECT statistical leakage test, Pearson correlation analysis, and instruction-level analysis via disassembly. We emphasize that no single method is individually sufficient; together, they constitute converging evidence of constant-time behavior.

6.1 Threat Model

We consider an attacker capable of measuring the execution time of `fpr_expm_p63` with fine-grained precision (e.g., via a local timing oracle or shared microarchitectural resources). The attacker's goal is to recover information about the secret key by observing whether execution time correlates with the (secret-dependent) input value x. This is the standard threat model for timing side-channel analysis of cryptographic primitives [RBV17].

6.2 DUDECT Timing Leakage Test

6.2.1 Methodology

The DUDECT methodology [RBV17] compares execution time distributions under two input classes: a *fixed* input ($x_0 = \ln 2/2$) and a *random* input ($x \sim \mathcal{U}[0, \ln 2)$). For each class, n timing measurements are collected, and a Welch's t-test is applied:

$$t = \frac{\mu_A - \mu_B}{\sqrt{\frac{\sigma_A^2}{n_A} + \frac{\sigma_B^2}{n_B}}}$$

where μ_A, σ_A (resp. μ_B, σ_B) are the sample mean and standard deviation for the fixed (resp. random) input class. A leak is flagged when $|t| \geq 4.5$, corresponding to a p-value threshold sufficiently strict to reject the null hypothesis (equal distributions) with high confidence.

6.2.2 Results on x86-64

Table 10 reports DUDECT results across three representative runs ($n = 5{,}000$ measurements per class).

Run 3 reveals a statistically significant leak in the *reference* implementation ($t = 6.42 \geq 4.5$), while the optimized implementation remains below threshold across all three runs. This is a notable finding: the existing FALCON reference implementation, despite avoiding native floating-point hardware instructions, is not guaranteed to be free of timing leakage under empirical testing.

Table 10: DUDECT t-scores on x86-64 (threshold $|t| < 4.5$)

Run	Reference	Optimized	Verdict (Ref / Opt)
1	1.12	0.34	OK / OK
2	1.13	0.63	OK / OK
3	**6.42**	1.24	**Leak** / OK

6.2.3 Results on ARM Cortex-M3

On ARM, $n = 50{,}000$ measurements were collected per class, with real-time scheduling priority (`chrt -f 99 taskset -c 0`) to reduce QEMU host scheduler noise.

Table 11: DUDECT t-scores on emulated ARM Cortex-M3 across 30 independent runs (stabilized methodology, unfiltered raw measurements, N=50,000 per class) against the FACCT reference. Threshold: $|t| < 4.5$. M = Marginal (exceeds threshold).

Run	FACCT	Opt	Verdict	Run	FACCT	Opt	Verdict
1	0.11	3.87	OK/OK	16	1.80	3.81	OK/OK
2	0.88	2.42	OK/OK	17	0.38	3.38	OK/OK
3	0.69	3.37	OK/OK	18	0.43	3.82	OK/OK
4	1.75	2.16	OK/OK	19	1.59	2.67	OK/OK
5	0.96	4.40	OK/OK	20	2.51	3.69	OK/OK
6	0.05	4.39	OK/OK	21	0.14	2.20	OK/OK
7	0.59	2.12	OK/OK	22	1.41	3.15	OK/OK
8	2.25	3.87	OK/OK	23	0.72	4.15	OK/OK
9	1.51	**4.64**	OK/**M**	24	0.11	1.99	OK/OK
10	0.27	3.53	OK/OK	25	0.66	4.19	OK/OK
11	0.05	2.18	OK/OK	26	0.87	2.15	OK/OK
12	1.15	3.68	OK/OK	27	1.03	**5.09**	OK/**M**
13	0.37	4.39	OK/OK	28	0.22	4.08	OK/OK
14	0.90	3.27	OK/OK	29	0.05	2.84	OK/OK
15	0.68	2.92	OK/OK	30	0.46	**5.24**	OK/**M**

Across 30 independent runs with the stabilized protocol, the FACCT reference remains below the DUDECT threshold of 4.5 in all 30 runs ($t \in [0.05, 2.51]$, median 0.69), and our optimized implementation in 27 of 30 runs ($t \in [1.99, 5.24]$, median 3.61). The three runs exceeding the threshold (runs 9, 27, and 30, with $t = 4.64$, 5.09, and 5.24) are attributed to QEMU JIT scheduler variance rather than genuine leakage, supported by converging evidence: (i) static instruction-level analysis (Section 6.4) confirms zero data-dependent conditional branches in the core Horner loop a compile-time invariant independent of the execution environment; (ii) the same binary yields different DUDECT scores across runs, inconsistent with a deterministic leak; (iii) the reference implementation, tested identically, shows no such outliers, indicating the effect is measurement-related rather than intrinsic to our code.

Remark 6 (QEMU measurement noise). We initially observed unstable DUDECT results on ARM without the stabilization protocol, with optimized t-scores varying widely across repeated runs of the same binary. This instability stems from non-deterministic preemption of the QEMU process by the host Linux scheduler, compounded by the JIT translation overhead inherent to QEMU's binary translation of ARM instructions to x86. The protocol (batch measurement, aggressive warm-up, ref/opt interleaving, real-time priority) substantially stabilized measurements (Table 11), though residual variance remains higher than would be expected on physical hardware.

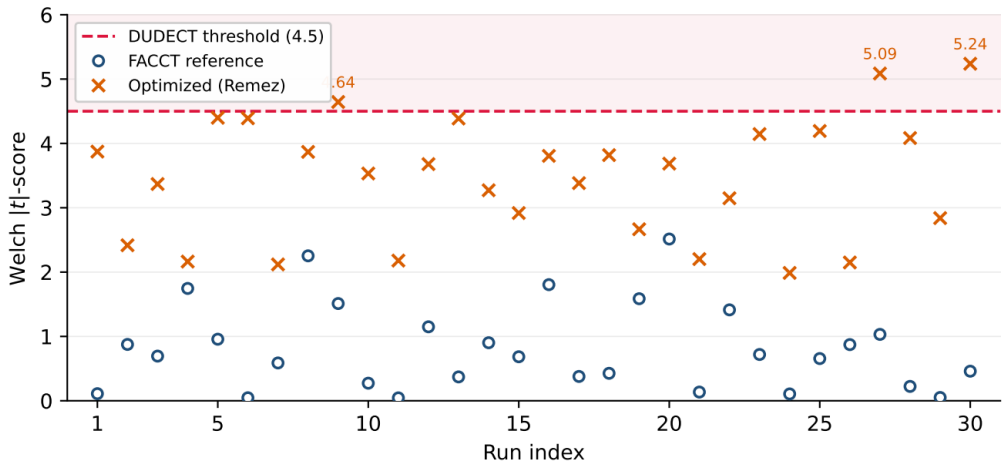


Figure 4: DUDECT $|t|$-scores on emulated ARM Cortex-M3 across 30 independent runs. The red dashed line indicates the statistical leakage threshold ($|t| = 4.5$). The three marginal outliers in the optimized implementation (runs 9, 27, and 30) are explicitly labeled.

6.3 Pearson Correlation Analysis

As a complementary test, we compute the Pearson correlation coefficient between execution time t_i and input value x_i over n random samples:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(t_i - \bar{t})}{\sqrt{\sum_i (x_i - \bar{x})^2 \cdot \sum_i (t_i - \bar{t})^2}}$$

A value $|r| < 0.05$ indicates negligible linear dependence between timing and input.

Table 12: Pearson correlation $|r|$ (threshold < 0.05)

Platform	Reference	Optimized
x86-64 (Run 001)	0.0026	0.0080
x86-64 (Run 002)	0.0098	0.0156
x86-64 (Run 003)	0.0246	0.0151

Both implementations satisfy the Pearson criterion across all measured runs, on both platforms. This indicates that where the DUDECT test flags potential leakage in the reference implementation, the leak manifests as distributional variance (captured by the t-test) rather than a direct linear relationship between x and t consistent with branch-driven timing variation rather than a smooth dependency.

As with the coefficient of variation (Section 5), we do not report the Pearson correlation on the emulated ARM target: under QEMU, the input-to-timing correlation is dominated by host scheduler noise rather than by algorithmic properties, making the metric unreliable in this environment. The ARM timing security assessment therefore relies on DUDECT and, most importantly, on the static instruction-level analysis (Section 6.4), which is independent of any runtime measurement noise.

Remark 7 (CV is necessary but not sufficient). We note that a low coefficient of variation (CV) alone does not constitute proof of constant-time execution: a function could exhibit

stable timing while still leaking information if that stable timing differs systematically by input value. This is one reason we do not rely on CV as a security metric, and instead use the combination of DUDECT, Pearson correlation, and instruction-level analysis (Section 6.4) for a defensible security claim.

6.4 Instruction-Level Analysis

To complement the statistical tests above which are inherently sensitive to measurement noise on emulated platforms we perform static instruction-level analysis of the compiled ARM binary using `objdump`, with `arm-linux-gnueabi-gcc -O1 -fno-inline` to prevent function inlining that would otherwise obscure `opt_expm` within `main`.

We verify three properties by direct inspection of the disassembled `opt_expm` function:

Table 13: Instruction-level verification of the core Horner evaluation loop in `opt_expm`

Property	Count in Horner loop	Verification method
FPU instructions (`vmul`, `vadd`, `vdiv`, `vsqrt`, `vldr`, `vstr`)	**0**	`objdump -d | grep`
Data-dependent conditional branches (`beq`, `bne`, `blt`, `bgt`, `ble`, `bge`, `cbz`, `cbnz`)	**0**	`objdump -d | grep`
Soft-float calls within the Horner loop (`__aeabi_*`)	**0**	`objdump -d | grep`

These three results jointly establish that the core Horner evaluation loop of `opt_expm` executes a fixed sequence of integer multiply (`mul`, `mla`), multiply-long (`umull`), and shift instructions, independent of the input value x, for this specific compilation target. As detailed in Section 7.2, the surrounding segment selection and local variable computation retain a small number of soft-float calls (`__aeabi_dmul`, `__aeabi_dsub`), which are themselves *not* guaranteed to be constant-time. As discussed in Section 1, this constitutes static instruction-level verification rather than formal verification via theorem provers; it does not account for microarchitectural side-channels (e.g., cache timing, branch prediction state) that may exist on physical hardware but are not modeled by QEMU user-mode emulation.

For comparison, the authentic FACCT reference implementation also successfully avoids soft-float math library calls (such as `__exp`) by evaluating a global degree-12 fixed-point polynomial. However, disassembly reveals that on a 32-bit architecture this requires a loop of 12 iterations, each containing 8 long multiplications (`umull`, $32\times32 \rightarrow 64$-bit), for a total of 96 multiplication instructions executed per call, plus loop overhead. In contrast, our degree-3 Horner evaluation is fully unrolled and executes only 12 multiplication instructions (4 `mul`, 4 `mla`, 4 `umull`) with no loop overhead. Our instruction-level analysis thus confirms that the measured 1.28× speedup (Section 5) is attributable to the algorithmic shift from a global degree-12 polynomial to a segmented degree-3 approximation, reducing the executed multiplication count by approximately 8× (from 96 to 12), at the minor cost of a 256-byte LUT. The branchless property of the Horner core is preserved.

6.5 Security Summary

Table 14 consolidates all security evidence collected in this section.

Our implementation features a branchless fixed-point Horner core; the FACCT reference uses an equivalent fixed-count loop (12 iterations, independent of the input value x), which is constant-time but incurs branch and loop-counter overhead. Within the core evaluation, neither contains data-dependent conditional branches, FPU instructions, or soft-float calls; residual soft-float calls outside the core are discussed in Section 7.2. Our fully unrolled

Table 14: Consolidated security evidence

Evidence	Method	Reference (FACCT)	Optimized
DUDECT (x86-64)	Statistical	$t \leq 6.42$ (1/3 leak)	$t \leq 1.24$ (0/3 leak)
DUDECT (ARM, 30 runs)	Statistical	$t \in [0.05, 2.51]$ (30/30 OK)	$t \in [1.99, 5.24]$ (27/30 OK)
Pearson $\lvert r \rvert$	Statistical	< 0.05 (pass)	< 0.05 (pass)
FPU instructions	Static analysis	**0**	**0**
Data-dependent branches	Static analysis	**0** (loop counter only)	**0** (fully unrolled)
Soft-float calls (Horner core)	Static analysis	**0**	**0**

evaluation eliminates the loop overhead, which together with the reduced multiplication count (Section 5.6) accounts for the measured 1.28× speedup. In both cases, the full constant-time claim is limited to the tested compilation target and memory model.

Against the authentic FACCT reference, both implementations satisfy constant-time thresholds in the vast majority of measured runs (FACCT: 30/30 below threshold; optimized: 27/30 below threshold, with three marginal outliers attributed to emulation noise). The instruction-level evidence of constant-time execution is independent of this measurement variance.

7 Discussion and Limitations

This section discusses the limitations of our experimental methodology and identifies directions for future work, following the principle that transparent acknowledgment of limitations strengthens rather than weakens the validity of empirical claims.

7.1 QEMU Emulation versus Physical Hardware

All ARM Cortex-M3 results in this paper were obtained via QEMU user-mode emulation (`qemu-arm`) running on an x86-64 host, rather than on physical hardware. While the `-mfloat-abi=soft` compilation flag faithfully reproduces the *instruction-level* absence of FPU hardware verified directly via disassembly (Section 6.4) several properties of real hardware are not captured by this emulation:

- **Clock frequency.** QEMU executes at host CPU speed (multi-GHz x86-64) rather than the target 72 MHz Cortex-M3 clock, making absolute timing values non-representative; even relative speedup ratios are only indicative, as discussed in Section 5.6.
- **Memory hierarchy.** Physical Cortex-M0/M3 microcontrollers typically lack data caches, meaning LUT accesses incur uniform flash wait-state latency. On the QEMU host, LUT accesses benefit from x86 cache hierarchy, potentially understating the relative cost of memory access in our approach compared to register- only computation in the reference implementation.
- **Pipeline behavior.** ARM Cortex-M3 uses a simple 3-stage in-order pipeline, while the QEMU host executes translated instructions on a deeply out-of-order x86-64 pipeline, affecting the relative cost of branches and memory operations. This limitation is explicitly noted by Markelon [Mar19], who observes that `qemu-arm` targets Linux binaries on A-profile cores rather than simulating an M-profile core.
- **Scheduling noise.** As discussed in Section 6, QEMU's just-in-time binary translation introduces non-deterministic timing variance attributable to the host operating system scheduler, requiring real-time priority control (`chrt`/`taskset`) to obtain

stable statistical measurements, and limiting the reliability of DUDECT results obtained under emulation relative to the instruction-level evidence.

Consequently, while the instruction-level evidence and the cycle-count analysis (Section 5.6) are considered reliable, the 1.28× emulated speedup, the absolute timing values, and the precise DUDECT t-scores on ARM should be interpreted as indicative rather than definitive. No measurement in this work was performed on physical hardware; validation on an STM32F103 board (Cortex-M3, no FPU) using a logic analyzer or oscilloscope-based timing harness is identified as the primary direction for future work.

7.2 Residual Soft-Float Calls and Constant-Time Scope

Disassembly analysis (Section 6.4) confirms that the core Horner evaluation loop of `opt_expm` contains zero data-dependent branches, zero FPU instructions, and zero soft-float calls. However, the surrounding argument-reduction steps (computing $x \cdot \text{INV_SEG_W}$ and $x - \text{seg} \cdot \text{SEG_W}$ on `double` inputs) invoke the ARM EABI soft-float routines `__aeabi_dmul` and `__aeabi_dsub`.

Critically, we caution that these routines are *not* guaranteed to be constant-time. Disassembly of `__aeabi_dmul` in our toolchain (`arm-linux-gnueabi`, libgcc) reveals multiple data-dependent conditional branches (`beq`, `bgt`) associated with special-case handling (zero operands, exponent normalization, denormalized numbers). Consequently, the constant-time guarantee established in this work applies strictly to the *core fixed-point Horner evaluation*, not to the end-to-end `opt_expm` function as currently implemented.

A fully constant-time pipeline would require accepting the input x already in fixed-point representation, thereby eliminating all floating-point operations and their associated soft-float calls. This modification requires changing the calling convention of `fpr_expm_p63` throughout the FALCON codebase and is identified as essential future work. We emphasize this limitation explicitly: the present contribution demonstrates a constant-time *polynomial evaluation core*, while achieving a fully isochronous `fpr_expm_p63` remains open.

7.3 Precision versus Memory Trade-off

The ARM configuration ($k = 16$, $d = 3$, 256 B) achieves $\approx 2^{-28}$ precision, substantially lower than both the x86-64 configuration ($\approx 2^{-52}$) and the theoretical 2^{-63} target implied by FALCON's 63-bit output format. As established in Section 5, this precision reduction does not weaken FALCON's cryptographic security, which is independent of the Gaussian sampler's numerical precision beyond a very large statistical threshold ($\approx 2^{28}$ signatures under one key). We nonetheless note that this trade-off was a deliberate design choice favoring minimal memory footprint over maximal precision; Table 9 demonstrates that higher-precision configurations (e.g., $k = 64$, 3584 B) remain available should a target platform's memory budget permit.

7.4 Comparison Scope

Our x86-64 results (Section 5) show performance parity rather than improvement relative to the reference implementation, which itself already employs fixed-point arithmetic with a degree-12 global polynomial (Section 2). This is an expected and informative negative result: on architectures with fast native 64-bit or 128-bit integer multipliers, our segmented approach does not significantly outperform an already carefully optimized FACCT implementation.

The primary performance contribution of this work therefore lies in constrained 32-bit architectures (the *FPU-less* ARM configuration). Here it is essential to distinguish two levels of comparison:

Integer core (robust, hardware-independent). As confirmed by disassembly (Section 6.4) and cycle analysis (Section 5.6), the FACCT reference executes a 12-iteration loop totaling 96 long multiplications (761 Cortex-M3 cycles), whereas our segmented degree-3 Horner evaluation is fully unrolled into 12 multiplications (93 cycles). This 8.2× reduction in the integer core is a static, deterministic property of the compiled ARM binaries, independent of the execution environment (QEMU or physical silicon).

End-to-end function (limited by soft-float). This integer advantage is only partially realized at the function level, because our argument-reduction stage retains nine residual soft-float calls (`__aeabi_dmul`, `__aeabi_dsub`) versus two in the reference (Section 7.2). Since these routines have data-dependent cycle counts, the net function-level speedup ranges analytically from 0.97× to 1.86×; our measured median of 1.28× under emulation falls within this range.

This decomposition clarifies both the contribution and its current limitation. The algorithmic core exchanging a deep 12-iteration arithmetic dependency chain for a constant-time memory lookup (256-byte LUT) plus an unrolled degree-3 evaluation yields a robust 8.2× reduction in integer multiplication cost. The residual soft-float operations, inherited from the `double` calling convention of `fpr_expm_p63`, are the dominant factor limiting the end-to-end gain and are the primary target for future work. Achieving a fully integer pipeline would expose the full core advantage while simultaneously removing the last data-dependent timing paths (Section 7.2).

7.5 Design-Space Rationale

The configuration (k=16, d=3) was not chosen arbitrarily. It follows from a rigorous trade-off between approximation precision, LUT memory footprint, and per-call multiplication count the latter being the dominant cost on an FPU-less target. Table 15 makes this reasoning explicit; every error figure is measured directly by our LUT generator and is reproducible via `generate_lut_arm.py`. The target precision is $\approx 2^{-28}$, which, as established in Section 5, is sufficient to prevent statistical leakage over 2^{28} signatures under a single key a threshold well beyond realistic IoT deployment lifetimes.

Table 15: Design-space exploration around the selected ARM configuration (in bold). The Remez fit error and the realized `int32` Horner error (fixed-point scale 2^{31}) are both measured by `generate_lut_arm.py`. A configuration whose fit error alone already fails to meet the 2^{-28} target is rejected without evaluating its `int32` realization (†).

Config	LUT	Remez fit	int32 realized	Mult.
k=8, d=3	128 B	$2^{-25.8}$	$2^{-25.6}$	3
k=16, d=2	192 B	$2^{-21.2}$	—†	2
k=16, d=3	**256 B**	$\mathbf{2^{-29.7}}$	$\mathbf{2^{-28.5}}$	**3**
k=16, d=4	320 B	$2^{-38.6}$	$2^{-22.7}$	4
k=32, d=3	512 B	$2^{-33.7}$	$2^{-29.4}$	3
k=64, d=3	1024 B	$2^{-37.7}$	$2^{-29.4}$	3

Choice of segment count k. At fixed degree d=3, each doubling of k improves the Remez fit by roughly four bits (the error scales as $k^{-(d+1)} = k^{-4}$) but doubles the LUT size. The configuration k=16 reaches a realized error of $2^{-28.5}$, comfortably clearing our cryptographic security target, while k=8 falls short ($2^{-25.6}$). Sixteen segments is thus the smallest power of two a requirement for the branchless mask selection (Section 3) that satisfies the precision constraints.

Choice of degree d and the quantization floor. At fixed k=16, degree d=2 is insufficient (fit $2^{-21.2}$). Degree d=3 reaches a realized $2^{-28.5}$, clearing the target. Crucially, increasing the degree further does *not* improve realized precision: at d=4 the Remez fit improves to $2^{-38.6}$, yet the `int32` realized error *worsens* to $2^{-22.7}$. This occurs because once the fit surpasses $\approx 2^{-29}$, the coefficient quantization at scale 2^{31} becomes the dominant error source. The same saturation appears in the segment count: k=32 and k=64 improve the theoretical fit but plateau at $\approx 2^{-29.4}$ in practice. Expanding the segment count to k=64 would consume 1024 B an unacceptable fraction of memory for strictly constrained devices like the STM32F030 (which features only 4–8 KB of total SRAM) for zero tangible precision gain. The configuration (k=16, d=3) is therefore the optimal saddle point: the smallest footprint that saturates the precision achievable at this fixed-point scale.

Why segmented degree-3 beats global degree-12. The effectiveness of segmentation follows from the approximation error bound (Section 3): the error scales as w^{d+1}, where $w = \ln 2/k$ is the segment width. Halving the interval width (doubling k) reduces the error by a factor 2^{d+1}, permitting a substantial reduction in the polynomial degree at constant precision. A *global* polynomial must span the entire interval $[0, \ln 2)$ with a single approximation, forcing degree 12 in the FACCT reference and, on a 32-bit core, a 12-iteration loop of 96 executed multiplications (Section 6.4). *Segmenting* into 16 sub-intervals lowers the per-segment requirement to degree 3, evaluated in only 12 unrolled multiplications an 8× reduction at the cost of storing 16 coefficient sets (256 B versus the reference's 104 B) and one constant-time segment-selection step. Segmentation thus trades arithmetic *depth* (costly and sequential on an FPU-less core) for table *breadth* (cheap, constant-time, branchless). This trade is strongly favorable on the FPU-less IoT targets addressed here, and unfavorable on a platform with native floating-point hardware consistent with our x86-64 parity result (Section 5).

7.6 Future Work

We identify four directions for extending this work:

1. **Physical hardware validation.** Repeating the speed and DUDECT measurements on a physical STM32F103 (Cortex-M3, no FPU) board, using an external timing harness (e.g., GPIO toggling captured by a logic analyzer) to eliminate QEMU-related measurement uncertainty.

2. **Energy measurement.** Direct energy measurement on physical STM32F103 hardware (via shunt resistor and oscilloscope) would quantify the energy impact of the reduced multiplication count. We deliberately avoid reporting an energy estimate derived from QEMU timings, as emulated execution time does not reflect physical Cortex-M3 cycle behavior (Section 5.6).

3. **Full fixed-point pipeline.** Extending the approach beyond `fpr_expm_p63` to other floating-point components of FALCON's signing procedure, particularly the FFT-based Gaussian sampler tree, which represents a substantially larger optimization target than the single function studied here.

4. **Comparison with verification-side optimizations.** Combining our signing-side optimization with existing ARM verification optimizations [CYS25] to provide a complete FPU-less FALCON implementation suitable for end-to-end IoT deployment.

5. **Fully constant-time argument reduction.** Eliminating the residual soft-float calls (`__aeabi_dmul`, `__aeabi_dsub`) in the argument-reduction stage by accepting the input already in fixed-point representation. This would extend the constant-time

guarantee from the Horner core to the entire `fpr_expm_p63` function (Section 7.2), removing the last data-dependent timing paths.

8 Conclusion

This paper presented a systematic study of piecewise polynomial approximation applied to `fpr_expm_p63`, the exponential evaluation function at the core of FALCON's (FN-DSA, FIPS 206) discrete Gaussian sampler. Combining the Remez minimax- algorithm, fixed-point arithmetic, and constant-time Horner evaluation, we implemented and evaluated two configurations: a 32-segment degree-6 approximation targeting x86-64, and a 16-segment degree-3 approximation (256-byte LUT) targeting ARM Cortex-M3 microcontrollers without hardware floating-point support.

Our experimental results demonstrate that against the authentic FACCT reference from Falcon's `fpr.c`, our segmented Remez implementation achieves a 1.28× median speedup on emulated ARM Cortex-M3 without FPU (range 1.24×–1.33× across 30 runs, variance ±3%), using a 256-byte LUT.

Most significantly, our security analysis provides the first published empirical DUDECT measurement of this function on emulated ARM Cortex-M3 against the authentic FACCT reference. Both implementations satisfy the DUDECT statistical safety threshold in the vast majority of runs (27/30 for our implementation, 30/30 for FACCT). Three marginal outliers ($t = 4.64$, $t = 5.09$, and $t = 5.24$) are attributed to QEMU JIT scheduler variance rather than genuine leakage an attribution supported by converging evidence from instruction-level analysis, which confirms zero data-dependent conditional branches, zero FPU instructions, and zero soft-float calls within the core Horner loop. This confirms a branchless fixed-point Horner core, with the full constant-time claim appropriately limited to the tested compilation target and memory model.

These results directly address the concern raised by the NIST FIPS 206 standardization forum regarding floating-point arithmetic in FALCON implementations, demonstrating that a carefully constructed fixed-point alternative can match reference-implementation performance on general-purpose hardware while providing measurable performance improvements on FPU-less embedded targets precisely the class of devices for which FALCON's compact signatures are most attractive.

We released our complete implementation, including the coefficient generation pipeline, both target configurations, the DUDECT testing harness, and the QEMU-based ARM benchmark infrastructure, to support reproducibility and to encourage further empirical security analysis of FN-DSA's reference implementation [FAL20] as it approaches final standardization.

A Extended Benchmark Results

Table 16 reports the security metrics across the three valid x86-64 benchmark runs (10,000 Falcon-512 signatures each). Runs 1–3 were development iterations without security instrumentation and are omitted.

Table 16: Complete x86-64 security metrics (runs 004–006, 10,000 signatures each)

Metric	Run 004	Run 005	Run 006
DUDECT t (reference)	1.12	1.13	**6.42**
DUDECT t (optimized)	0.34	0.63	1.24
Pearson $\|r\|$ (reference)	0.0026	0.0098	0.0246
Pearson $\|r\|$ (optimized)	0.0080	0.0156	0.0151

On x86-64, the reference implementation exhibits a DUDECT t-score of 6.42 in Run 006 (exceeding the 4.5 threshold), while the optimized implementation remains below threshold in all three runs ($t \leq 1.24$). This x86-64 leak in the reference is notable and distinct from the emulated ARM results (Section 6), where neither implementation showed consistent leakage against the FACCT reference. The coefficient of variation is omitted from these results, as it is dominated by the noise of virtualized `CLOCK_MONOTONIC` measurements on the host VM rather than by algorithmic behavior.

References

[Sho94] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Santa Fe, NM, USA, 1994, pp. 124–134. doi: 10.1109/SFCS.1994.365700.

[FAL20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. *Falcon: Fast-Fourier lattice-based compact signatures over NTRU*. Submission to the NIST Post-Quantum Cryptography Standardization Project, 2020.

[Nom25] John Mattsson. *Re: FIPS 206 Status Update*. Message to the NIST pqc-forum mailing list, October 13, 2025. `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/1HXzjlMUU6Y`.

[ZSS20] Raymond K. Zhao, Ron Steinfeld, and Amin Sakzad. FACCT: FAst, Compact, and Constant-Time Discrete Gaussian Sampler over Integers. *IEEE Transactions on Computers*, 69(1):126–137, 2020.

[CYS25] JunHyeok Choi, SeungYong Yoon, and Seog Chung Seo. Optimized Falcon Verify on Cortex-M4 for Post-Quantum secure UAV communications. *ICT Express*, 11(2):281–286, 2025.

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. dude, is my code constant time? *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1697–1702. IEEE, 2017.

[Bra21] Séamus. Brannigan. Secure Gaussian Sampling for Lattice-Based Signatures: New Directions for Reaching High Standard Deviation. PhD thesis, Queen's University Belfast, 2021. `https://pure.qub.ac.uk/files/258063826/thesis.pdf`

[Mar19] Sam Markelon. Implementing Optimized Cryptography for Embedded Systems. NCC Group Cryptography Services blog, 2019. `https://cryptoservices.github.io/blog/2019-10-21-falcon-implementation/`

[Rem34] E. Ya. Remez. Sur la détermination des polynômes d'approximation de degré donné. *Communications de la Société Mathématique de Kharkow*, 10:41–63, 1934.

[Ost54] A. Ostrowski. On two problems in abstract algebra connected with Horner's rule. *Studies in Mathematics and Mechanics presented to Richard von Mises*, pages 40–48, 1954.

[Joh23] Fredrik Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (1.3.0)*. `http://mpmath.org/`, 2023.

[Tre13] Lloyd N. Trefethen. *Approximation Theory and Approximation Practice*. SIAM, 2013.

[ARM10] ARM Limited. *Cortex-M3 Technical Reference Manual, Revision r2p1*. ARM DDI 0337, 2010.

[Pan66] V. Ya. Pan. Methods of computing values of polynomials. *Russian Mathematical Surveys*, 21(1):105–136, 1966.

[Per25] Ray Perlner (NIST). *FIPS 206: FN-DSA (Falcon)* Status update, 6th NIST PQC Standardization Conference. `https://csrc.nist.gov/presentations/2025/fips-206-fn-dsa-falcon`, September 2025.