Vectorized Falcon-Sign Implementations using SSE2, AVX2, AVX-512F, NEON, and RVV

Jipeng Zhang, Jiaheng Zhang

National University of Singapore, Singapore
jp-zhang@outlook.com jhzhang@nus.edu.sg

Abstract. Falcon, a NTRU-based digital signature algorithm, has been selected by NIST as one of the post-quantum cryptography (PQC) standards. Compared to verification, the signature generation of Falcon is relatively slow. One of the core operations in signature generation is discrete Gaussian sampling, which involves a component known as the BaseSampler. The BaseSampler accounts for up to 30% of the time required for signature generation, making it a significant performance bottleneck. This work aims to address this bottleneck.

We design a vectorized version of the BaseSample and provide optimized implementations across six different instruction sets: SSE2, AVX2, AVX-512F, NEON, RISC-V Vector (RVV), and RV64IM. The AVX2 implementation, for instance, achieves an 8.4× speedup over prior work. Additionally, we optimize the FFT/iFFT operations using RVV and RV64D. For the RVV implementation, we introduce a new method using strided load/store instructions, with 4+4 and 4+5 layer merging strategies for Falcon-{512,1024}, respectively, resulting in a speedup of more than 4×.

Finally, we present the results of our optimized implementations across eight different instruction sets for signature generation of Falcon. For instance, our AVX2, AVX-512F, and RV64GCVB implementations achieve performance improvements of 23%, 36%, and 59%, respectively, for signature generation of Falcon-512.

Keywords: Falcon · FFT · SIMD · RISC-V · SSE2 · AVX2 · AVX-512 · NEON

1 Introduction

In July 2022, NIST announced that Falcon was selected as one of the post-quantum cryptographic digital signature standards [AAC+22], and it will be known as FN-DSA. In terms of performance, Falcon is characterized by its low bandwidth, which refers to the combined size of the public key and the signature. It offers fast verification; however, key generation and signature generation are relatively slow. As of this writing, the corresponding standard document, FIPS 206, has not yet been published. Therefore, we primarily refer to the Falcon specification [FHK+20] for this work.

1.1 Motivations and Contributions

Compared to signature verification, Falcon's signature generation is significantly slower. Therefore, we focus primarily on optimizing the implementation of signature generation.

Motivations. The most computationally intensive operation in signature generation is fast Fourier sampling, which primarily involves FFT-related computations (e.g., splitfft, mergefft) and discrete Gaussian sampling (i.e., SamplerZ). FFT-related subroutines often benefit from vectorization, as shown in the baseline C-FN-DSA project. However, strict compatibility with known answer tests (KAT) requires adhering to a predetermined sequence of pseudorandom number usage. This constraint prevents the time-consuming

BaseSampler subroutine within SamplerZ from being vectorized, as it is impossible to gather multiple BaseSampler instances for vectorized or batched execution while maintaining KAT compatibility.

Contributions. Our methodology comprises three key steps: (1) profiling the baseline C-FN-DSA project to identify performance bottlenecks, (2) optimizing the most time-consuming subroutines, and (3) reprofiling to guide future work. Our contributions are summarized as follows:

- **Performance profiling.**[1] Our profiling of the C-FN-DSA project on Intel i7-11700K and SpacemiT X60 reveals that BaseSampler accounts for more than 30% of the total signing time on both platforms. On the SpacemiT X60, FFT-related subroutines contribute nearly 38% of the execution time.

- **Vectorized BaseSampler.** To enable vectorization for BaseSampler, we deliberately relax compatibility with KAT and employ a modular design to ensure correctness. Importantly, this modification maintains full interoperability with signature verification. To quantify the benefits of this approach, we implement the vectorized BaseSampler across six different instruction sets (SSE2, AVX2, AVX-512F, NEON, RVV, and RV64IM), achieving significant performance improvements.

- **Vectorized FFT.** While prior work has explored FFT/iFFT optimization for other instruction sets, this work addresses the gap for RVV and RV64D. Leveraging vector strided load/store instructions, we propose a novel 4+5 layer merging strategy for Falcon-1024's FFT—a significant improvement over the 1+2+2+4 strategy previously used for NEON [NG23]. Our RVV-optimized FFT/iFFT achieves a 4.1–4.7× speedup, while the RV64D-optimized version delivers a 2.7–2.9× speedup.

- By incorporating existing optimized Keccak implementations alongside the above optimizations, our work achieves speedups of 13%, 23%, 36%, 17%, 37%, 36%, 58%, and 59% for Falcon-512's signature generation on SSE2, AVX2, AVX-512F, NEON, RV64GC, RV64GCB, RV64GCV, and RV64GCVB, respectively.

Our implementations and detailed raw experimental data are publicly available at https://github.com/Ji-Peng/VecFalcon under the MIT license.

1.2 Related Works

We focus on three main categories of related works: software implementations, hardware implementations, and HW/SW co-design approaches.

Software implementations. [KSS22, NG23] optimized the FFT/iFFT using NEON. The latter achieved better performance, employing layer merging strategies of 2+2+4 for Falcon-512 and 1+2+2+4 for Falcon-1024. On Cortex-A72, the FFT/iFFT implementations of [NG23] showed nearly a 2× improvement. Additionally, several works have studied optimizations for Cortex-M4, such as [OSHG19, CYS25].

Hardware implementations. [ASA+25, DTN+25] designed FFT/iFFT accelerators, while [ZASM25] developed an accelerator for the discrete Gaussian sampler. [OZZ+25] presented a full-hardware implementation for signature generation of Falcon.

HW/SW co-design. [KA24] offloaded parts of the discrete Gaussian sampling process—such as BaseSampler and BerExp—to hardware, achieving a 9.83× speedup compared to the reference software. [YSZ+24] introduced extended RVV instructions to accelerate modular arithmetic, SHA-3, and discrete Gaussian sampling.

[1] We tried profiling on Raspberry Pi 4B but got unstable results. Therefore, we rely on performance data collected from the other two platforms for our analysis.

In summary, these related works emphasize the importance of optimizing discrete Gaussian sampling and FFT/iFFT, which aligns with the focus of this paper. The implementations that can be directly compared with our work include [NG23] and the C-FN-DSA project, which will be discussed in Section 3.

2 Preliminaries

This paper focuses primarily on the optimized implementation of the signature generation of Falcon. Accordingly, this section will detail the signature generation while omitting in-depth discussions of key pair generation and signature verification. We emphasize technical aspects relevant to our optimized implementation, referring readers to the Falcon specification [FHK+20] for additional details. Section 2.1 introduces the notation used throughout this paper. Section 2.2 provides a detailed explanation of Falcon's signature generation process, along with a brief overview of key pair generation and signature verification. Section 2.3 discusses the Fast Fourier Transform (FFT), while Section 2.4 presents the three target platforms employed in this work. Section 2.5 describes the vector instructions utilized in this work.

2.1 Notation

We adopt the notation from [FHK+20]. Bold uppercase letters (e.g., $\mathbf{B}$) denote matrices, bold lowercase letters (e.g., $\mathbf{v}$) represent vectors, and scalars or polynomials are written in italics (e.g., s). The transpose of matrix $\mathbf{B}$ is denoted $\mathbf{B}^t$.

Falcon employs a prime integer modulus $q = 12289$, where $\mathbb{Z}_q$ denotes the quotient ring $\mathbb{Z}/q\mathbb{Z}$. The polynomial modulus is defined as $\phi = x^n + 1$, with $n = \{512, 1024\}$ for Falcon-$\{512,1024\}$, respectively.

For $\sigma, \mu \in \mathbb{R}$ where $\sigma > 0$, the Gaussian function is given by:

$$\rho_{\sigma,\mu}(x) = \exp\left(-\frac{|x-\mu|^2}{2\sigma^2}\right), \tag{1}$$

and the discrete Gaussian distribution $D_{\mathbb{Z},\sigma,\mu}$ over the integers is defined as:

$$D_{\mathbb{Z},\sigma,\mu}(x) = \frac{\rho_{\sigma,\mu}(x)}{\sum_{z\in\mathbb{Z}} \rho_{\sigma,\mu}(z)}. \tag{2}$$

The symbols $\gg$ and & denote bitwise right-shift and AND operations, respectively. The symbol $\odot$ denotes the multiplication in FFT domain. For a logical proposition P, the notation $[\![P]\!]$ evaluates to 1 if P is true and 0 otherwise; this operation is implemented in constant time to mitigate timing attacks. For any $k \in \mathbb{Z}^+$, $\mathsf{UniformBits}(k)$ samples z uniformly from $\{0, 1, \ldots, 2^k - 1\}$.

2.2 Falcon

Key pair generation of Falcon consists of two steps: (1) Compute polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$ satisfying the NTRU equation $fG - gF = q \mod \phi$, where $\phi = x^n + 1$; (2) Derive a Falcon tree T from the private key elements f, g, F, and G, which involves LDL* decomposition.

A Falcon private key sk comprises four polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$. In practice, G is omitted from sk since it can be efficiently recomputed from f, g, and F. Two additional components are derived: (1) A matrix defined as:

$$\hat{\mathbf{B}} = \left[\begin{array}{c|c} \mathsf{FFT}(g) & -\mathsf{FFT}(f) \\ \hline \mathsf{FFT}(G) & -\mathsf{FFT}(F) \end{array}\right], \tag{3}$$

Algorithm 1: Sign(m, sk, $\lfloor\beta^2\rfloor$)

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

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

Algorithm 2: $\mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$

Input : $\mathbf{t} = (t_0, t_1) \in \mathsf{FFT}(\mathbb{Q}[x]/(x^n+1))^2$; T
Output : $\mathbf{z} = (z_0, z_1) \in \mathsf{FFT}(\mathbb{Z}[x]/(x^n+1))^2$

if $n = 1$ **then**
  $\sigma' \leftarrow \mathsf{T.value}$
  $z_0 \leftarrow \mathsf{SamplerZ}(t_0, \sigma')$
  $z_1 \leftarrow \mathsf{SamplerZ}(t_1, \sigma')$
  return $\mathbf{z} = (z_0, z_1)$
$(\ell, \mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T.value}, \mathsf{T.leftchild}, \mathsf{T.rightchild})$
$\mathbf{t}_1 \leftarrow \mathsf{splitfft}(t_1)$
$\mathbf{z}_1 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_1, \mathsf{T}_1)$
$z_1 \leftarrow \mathsf{mergefft}(\mathbf{z}_1)$
$t_0' \leftarrow t_0 + (t_1 - z_1) \odot \ell$
$\mathbf{t}_0 \leftarrow \mathsf{splitfft}(t_0')$
$\mathbf{z}_0 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_0, \mathsf{T}_0)$
$z_0 \leftarrow \mathsf{mergefft}(\mathbf{z}_0)$
return $\mathbf{z} = (z_0, z_1)$

where FFT denotes the Fast Fourier Transform; (2) A FALCON tree T. The corresponding public key pk is the polynomial $h \in \mathbb{Z}_q[x]/(\phi)$, where $h = gf^{-1} \mod (\phi, q)$.

Signature generation (Algorithm 1) relies critically on fast Fourier sampling, i.e., ffSampling (Algorithm 2). The format of all polynomials in ffSampling is in FFT representation. The ffSampling is a recursive procedure where splitfft and mergefft represent a step of the inverse Fast Fourier Transform (iFFT) and Fast Fourier Transform (FFT), respectively. Specifically, for $f \in \mathbb{Q}/(\phi)$, splitfft decomposes $\mathsf{FFT}(f)$ into two smaller FFTs (i.e., $\mathsf{FFT}(f_e)$ and $\mathsf{FFT}(f_o)$ for $f_e, f_o \in \mathbb{Q}/(x^{n/2}+1)$), while mergefft combines two smaller FFTs into a larger FFT.

The core subroutine of fast Fourier sampling is SamplerZ (Algorithm 3), which securely samples z from the discrete Gaussian distribution $D_{\mathbb{Z},\sigma',\mu}$ for $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$, $1 \le \sigma_{\min} < \sigma_{\max}$, and $\mu \in \mathcal{R}$. SamplerZ utilizes techniques from [HPRR20, ZSS20], employing the BaseSampler (Algorithm 5) and BerExp (Algorithm 6) subroutines.

The BaseSampler samples non-negative integers $z_0 \in \{0, \ldots, 18\}$ from a distribution χ defined as:

$$\chi(i) = 2^{-72} \cdot \mathsf{pdt}[i], \quad \forall i \in \{0, \ldots, 18\}, \tag{4}$$

where pdt is a precomputed 19-element probability distribution table (scaled by 2^{72}), and its corresponding reverse cumulative distribution table is denoted as RCDT. Each entry in RCDT can be represented as a 72-bit integer. This distribution approximates a half-Gaussian $D_{\mathbb{Z}^+,\sigma_{\max}}$ with negligible Rényi divergence ($R_{513}(\chi\|D_{\mathbb{Z}^+,\sigma_{\max}}) \le 1 + 2^{-78}$).

BerExp (Algorithm 6), along with its subroutine ApproxExp (Algorithm 4), implements rejection sampling. Notably, the FALCON specification explicitly states that the loop within BerExp is not required to execute in constant-time (see [FHK+20, Alg 14]). Meanwhile, ApproxExp relies on a precomputed 13-element array C of 64-bit integers.

Signature verification procedure is significantly simpler than both key pair generation and signature generation, as referenced in [FHK+20, Alg 16].

Algorithm 3: SamplerZ(μ, σ')

Input : $\mu, \sigma' \in \mathcal{R}$; $\sigma' \in [\sigma_{\min}, \sigma_{\max}]$
Output : $z \in \mathbb{Z}$ close to $D_{\mathbb{Z},\mu,\sigma'}$

$r \leftarrow \mu - \lfloor \mu \rceil$
$ccs \leftarrow \sigma_{\min}/\sigma'$
while (1) **do**
  $z_0 \leftarrow$ BaseSampler()
  $b \leftarrow$ UniformBits(8) & 0x1
  $z \leftarrow b + (2 \cdot b - 1)z_0$
  $x \leftarrow \frac{(z-r)^2}{2\sigma'^2} - \frac{z_0^2}{2\sigma_{\max}^2}$
  if BerExp(x, ccs) $= 1$ **then**
    return $z + \lfloor \mu \rceil$

Algorithm 4: ApproxExp(x, ccs)

Input : $x \in [0, \ln(2)]$; $ccs \in [0, 1]$; precomputed array C
Output : An integer $\approx 2^{63} \cdot ccs \cdot \exp(-x)$

$y \leftarrow C[0]$
$z \leftarrow \lfloor 2^{63} \cdot x \rfloor$
for $i = 1$ **to** 12 **do**
  $y \leftarrow C[i] - (z \cdot y) \gg 63$
$z \leftarrow \lfloor 2^{63} \cdot ccs \rfloor$
$y \leftarrow (z \cdot y) \gg 63$
return y

Algorithm 5: BaseSampler()

Output : $z_0 \in \{0, \ldots, 18\}$; $z_0 \sim \chi$

$u \leftarrow$ UniformBits(72)
$z_0 \leftarrow 0$
for $i = 0$ **to** 17 **do**
  $z_0 \leftarrow z_0 + [\![u < \mathsf{RCDT}[i]]\!]$
return z_0

Algorithm 6: BerExp(x, ccs)

Input : Floating point values $x, ccs \geq 0$
Output : A single bit, equal to 1 with probability $\approx ccs \cdot \exp(-x)$

$s \leftarrow \lfloor x/\ln(2) \rfloor$
$r \leftarrow x - s \cdot \ln(2)$
$s \leftarrow \min(s, 63)$
$z \leftarrow (2 \cdot$ ApproxExp$(r, ccs) - 1) \gg s$
$i \leftarrow 64$
do
  $i \leftarrow i - 8$
  $w \leftarrow$ UniformBits(8) $-$ (($z \gg i$) & 0xFF)
while (($w = 0$) **and** ($i > 0$))
return $[\![w < 0]\!]$

2.3 FFT

We focus on the implementation aspects when $\phi = x^n + 1$, where $n = \{512, 1024\}$ for Falcon-$\{512,1024\}$, respectively. The FFT leverages a divide-and-conquer strategy. For a polynomial $f = f_e(x^2) + xf_o(x^2)$ and any root ζ of ϕ, we have $f(\zeta) = f_e(\zeta^2) + \zeta f_o(\zeta^2)$, where ζ^2 is a root of $x^{n/2} + 1$. This decomposition reduces an n-degree problem to two $n/2$-degree subproblems. The recursive application yields $\mathcal{O}(n \log n)$ time complexity.

Two fundamental operations enable efficient in-place computation: (1) CT (Cooley-Tukey) butterfly unit [CT65]: $(a, b) \leftarrow (a + \zeta b, a - \zeta b)$, which is commonly used for FFT computation with normal-order input and bit-reversed output. (2) GS (Gentleman-Sande) butterfly unit [GS66]: $(a, b) \leftarrow \left(a + b, (a - b)\overline{\zeta}\right)$, which is commonly used for inverse FFT (iFFT) with bit-reversed input and normal-order output.

The FFT leverages precomputed roots of unity, denoted as $\zeta_j = e^{j\pi i/n}$. Similarly, the iFFT utilizes ζ_j^{-1}. Notably, since $\zeta_j^{-1} = \overline{\zeta_j}$, the inverse roots can be obtained simply by negating the imaginary part of ζ_j.

Following the computational sequence, we refer to each divide-and-conquer step in the FFT as a layer. The initial step of the FFT is designated as layer 0, with subsequent layers numbered sequentially up to layer $\log_2 n - 1$. The layer 0 of FFT is computationally trivial. For instance, the pair $(f_0, f_{n/2})$ involves the root of $(x^2 + 1)$, which is i, resulting in $f_0 + if_{n/2}$. Since this operation requires no arithmetic computation, it incurs no cost. Subsequent layers, however, perform arithmetic of complex numbers. For example, in layer 1, one CT butterfly unit computes:

$$(f_0 + if_{n/2}) + \zeta(f_{n/4} + if_{n/4+n/2}) \text{ and } (f_0 + if_{n/2}) - \zeta(f_{n/4} + if_{n/4+n/2}), \tag{5}$$

where ζ is a root of ϕ.

Table 1: Latency and cycles per instruction (CPI) of commonly-used instructions on SpacemiT X60 core. For vector strided load and store operations (i.e., vlse64 and vsse64), the tested stride values in bytes are 8, 16, 32, 64, 128, and 256. SEW: selected element width. common: instructions related to arithmetic, logic, and comparison.

Instruction	Latency	CPI	Instruction	Latency	CPI
RV64IMB			**RV64V** with SEW=64		
common	1	0.5	vfadd/vfsub	4	1
lw/ld	3	1	vfsgnj	4	2
sw/sd	-	1/0.8	vfmul	5	1
mul	5	3	vfmacc	6	1
mulh	6	4	vfdiv	-	40∼48
rori/andn	1	0.5	vle64	3	3
RV64D			vse64	-	3.25
fadd/fsub	4	0.5	vlse64/vsse64	-	4
fsgnj	4	1	vrgather	5	4
fmul	5	0.75	vmerge	4	2
fmadd	6	1	**RV64V** with SEW=32		
fdiv	-	23∼24	vadd/vsub	4	1
fld	4	1.25	vsrl/vsll	4	2
			vmadc/vmsbc	4	2

2.4 Target Platforms and Benchmark Configurations

This work evaluates implementations across three target platforms with eight different instruction sets (SSE2, AVX2, AVX-512F, NEON, RV64GC, RV64GCB, RV64GCV, and RV64GCVB):

1. **x86-64**: The Intel i7-11700K CPU (Rocket Lake microarchitecture) operating at 3.6 GHz. We reference instruction latencies primarily from the uops.info study [AR19] and its accompanying website[2]. This platform serves to benchmark our optimized implementations for SSE2, AVX2, and AVX-512F instruction sets.

2. **ARMv8-A**: The Cortex-A72 processor in Raspberry Pi 4B running at 1.5 GHz. Instruction latencies are sourced from [ARM15]. This core tests our optimized implementation for NEON.

3. **RISC-V**: The SpacemiT X60 core[3] in Milk-V Jupiter operating at 2.0 GHz, supporting the RV64GCVB instruction set with vector extension v1.0 [RIS21b] (VLEN = 256 bits) and bit-manipulation extension v1.0.0 [RIS21a]. Here, G denotes the IMAFDZicsr_Zifencei extension combination. This core lacks the Zvkb extension [RIS21c]. Its instruction set configuration closely resembles the XuanTie C908 core used in [ZYHK25], differing primarily in VLEN (128 bits versus 256 bits). Due to limited processor documentation, we adapt the benchmarking methodology from [ZYHK25], with specific latency measurements detailed in Table 1. This core evaluates implementations targeted for four instruction sets: RV64GC, RV64GCB, RV64GCV, and RV64GCVB (collectively abbreviated as RV64GC{V}{B}).

All builds use -O2 optimization level. Since this work focuses on the SHAKE256X4 variant of FALCON (discussed in Section 3), we compile with -DFNDSA_SHAKE256X4=1. The performance profiling in Section 3 adopts gperftools[4]. We evaluate our implementation on target platforms with the following configurations:

[2] https://uops.info/table.html
[3] https://www.spacemit.com/en/spacemit-x60-core/
[4] https://github.com/gperftools/gperftools

Table 2: Vector operations and their corresponding instructions across different ISAs.

Operation	SSE2	AVX2/AVX-512F	RVV	NEON
VADD	paddd	vpaddd	vadd	add
VSUB	psubd	vpsubd	vsub	sub
VCMPGT	pcmpgtd	vpcmpgtd	vmsgt	cmgt
VSRLI	psrld	vpsrld	vsrl	ushr
VMULLO	pmullw	vpmulld	vmul	mul

1. **Intel i7-11700K**: Ubuntu 24.04 with GCC 13.3.0. Hyper-Threading and Turbo Boost are disabled. For benchmarking the SSE2 version, we explicitly disable unused instruction sets (e.g., -mno-avx -mno-avx2) to prevent compiler optimizations. Similar precautions are taken for the AVX2 and AVX-512F versions.
2. **Cortex-A72**: Ubuntu 20.04 with Clang 10.0.0.
3. **SpacemiT X60**: Bianbu 1.0.15 (Linux kernel 6.1.15) with GCC 13.2.0. We specify the target instruction set through -march flags: RV64GC (-march=rv64gc), RV64GCB (-march=rv64gc_zba_zbb), RV64GCV (-march=rv64gcv), and RV64GCVB (-march=rv64gcv_zba_zbb).

2.5 Vector Operations Across ISAs

To describe the vectorized BaseSampler proposed in Section 4.2, we express our core ideas using abstract vector operations: VADD, VSUB, VCMPGT, VSRLI, and VMULLO. These denote 32-bit integer vector addition, subtraction, comparison (greater-than), logical right-shift, and multiplication (returning the low 32 bits of the product), respectively. The corresponding instructions across various instruction set architectures (ISAs) are summarized in Table 2. Since SSE2 does not support the pmulld instruction (which is available in SSE4.1), we use pmullw instead.

3 Performance Profiling

This work builds upon the C-FN-DSA project[5] as our baseline. The project supports multiple CPU architectures: (1) x86-64 (with separate code paths for SSE2 and AVX2), (2) ARMv8-A (using NEON for floating-point operations), and (3) RISC-V (using the D extension for floating-point operations). It also incorporates features described in [Por19], such as constant-time signature generation and memory access patterns that do not depend on secret data. While the project also supports CPUs without floating-point units, this aspect falls outside the scope of our discussion.

One might question why the C-FN-DSA project maintains an SSE2 version rather than adopting AVX2 as the baseline. This design choice stems from Falcon's reliance on floating-point operations, where SSE2 represents the most widely compatible floating-point engine on x86-64 platforms (supported since the 2001 Pentium 4 processor).

Compared to the NIST submission implementation[6], the C-FN-DSA project exhibits several key differences:

- It uses SHAKE256 for pseudorandom number generation (i.e., UniformBits()) instead of ChaCha20, making it more compliant with NIST standards;
- It explicitly supports more processor architectures (versus only AVX2 and ARM Cortex-M4 in the NIST submission), demonstrating better compatibility;

[5] https://github.com/pornin/c-fn-dsa/tree/96e3b92
[6] https://falcon-sign.info/Falcon-impl-20211101.zip

Table 3: The performance profiling of Falcon-1024's signature generation (sign_core subroutine in C-FN-DSA), conducting 10,000 iterations using gperftools. BaseSampler and BerExp correspond to Algorithms 5 and 6. SHA-3 includes both hashing and extendable-output functions. Due to profiling errors, the percentages may not sum to exactly 100%.

Version	BaseSampler	BerExp	SHA-3	FFT-related
AVX2 on Intel i7-11700K	30.2%	31.2%	22.6%	15.1%
RV64GCVB on SpacemiT X60	30.3%	14.3%	20.6%	37.6%

- The NIST submission heavily relies on AVX2 optimizations (particularly for BaseSampler and FFT-related subroutines). C-FN-DSA's AVX2 version only employs AVX2 for the Keccak implementation in the SHAKE256X4 variant, using SSE2 for other floating-point and vectorized computations.
- Our benchmarks on Intel i7-11700K show that C-FN-DSA achieves faster signature generation despite its reduced AVX2 usage. We attribute this performance advantage to the AVX-SSE transition penalties [Kon11] incurred by the NIST submission, compounded by the fact that compilers typically generate SSE2 instructions for basic floating-point operations.

The C-FN-DSA project provides a signature generation variant called SHAKE256X4. Unlike the default version, this variant enables 4-way Keccak computations. This variant benefits AVX2 implementations (which natively support 4-way Keccak); SSE2, ARMv8-A, and RISC-V implementations must emulate it through multiple 2-way or 1-way Keccak. Although the SHAKE256X4 variant is not KAT-compatible with the default version, it maintains interoperability with signature verification. Unless otherwise specified, this work focuses on the SHAKE256X4 variant.

Our performance profiling results for the signature generation of the C-FN-DSA project are presented in Table 3. The data reveals two key observations: (1) BaseSampler accounts for more than 30% of execution time on both platforms; (2) FFT-related subroutines constitute 37.6% of execution time on the SpacemiT X60.

Based on these findings, we focus on BaseSampler and FFT-related operations. We note that BerExp also shows significant overhead (31.2%) in the AVX2 implementation, which we identify as promising future work and will revisit in Section 9.

4 Vectorized BaseSampler

Section 4.1 summarizes prior implementations. Section 4.2 details our vectorization approach. Sections 4.3 and 4.4 present our implementations across various vector instruction sets (SSE2, AVX2, AVX-512F, RVV, and NEON) as well as the RV64IM scalar instruction set. Section 4.5 describes the integration of our optimized BaseSampler into Falcon. Finally, Section 4.6 benchmarks the performance of different BaseSampler implementations.

For readers who prefer to avoid instruction-set-specific details, we recommend focusing on Section 4.2. The algorithms presented there (Algorithms 7 and 8) capture the core ideas of our approach without relying on architecture-specific instructions.

4.1 Previous Implementations

We begin by examining previous implementations of BaseSampler, known as gaussian0 subroutine in C-FN-DSA and gaussian0_sampler subroutine in the NIST submission.

Both implementations provide a reference version (denoted as ref version) that operates as follows: (1) A 72-bit integer is split into three 24-bit limbs; (2) The 72-bit integer

comparison is implemented using unsigned 32-bit subtraction with borrow. The core operation (Step 4 of Algorithm 5) is implemented as:

```
uint32_t cc;
cc = (v0 - RCDT[i][2]) >> 31;
cc = (v1 - RCDT[i][1] - cc) >> 31;
cc = (v2 - RCDT[i][0] - cc) >> 31;
z0 += cc;
```

where v0, v1, v2 correspond to the low, middle, and high 24-bit limbs, respectively.

The C-FN-DSA project exclusively employs the ref version across all code paths. The NIST submission implements a more sophisticated AVX2 version (denoted as AVX2-ref version) by partitioning 72-bit integers into 57+15 bits while completely unrolling all loops. Our microbenchmark results confirm the AVX2-ref version's performance advantage over the ref version. While C-FN-DSA does not document its rationale for excluding the AVX2-ref version, we hypothesize this aims to avoid AVX-SSE transition penalties.

[NG23] provides a NEON-based implementation[7]; our benchmarks demonstrate that it underperforms ref version (see Section 4.6) on Cortex-A72.

4.2 Vectorization Approach

The aforementioned AVX2-ref version proves suboptimal for two primary reasons. First, its implementation complexity is substantial, requiring not only comparison operations but also broadcast instructions, horizontal additions, and permutations—all of which exhibit higher latency than basic arithmetic and shift operations. For instance, on Haswell and Rocket Lake microarchitectures, the vpbroadcastw instruction demonstrates a latency of 3 cycles and a CPI of 1, compared to arithmetic instructions' latency of 1 cycle and CPI of 0.5. Second, a more efficient alternative exists: implementing an 8-way parallel version of the ref version using AVX2, where the core loop only involves comparison, addition/subtraction, and shift operations.

However, as evident in Algorithms 3, 5 and 6, the pseudorandom number usage (via UniformBits()) must follow a specific sequential order to maintain KAT compatibility. Consequently, vectorizing BaseSampler necessitates sacrificing KAT compatibility.

This compromise does not affect the interoperability of signature verification for two reasons: (1) our modification does not alter the specific Gaussian distribution followed by the output vector $\mathbf{s}$ generated in Line 7 of Algorithm 1; and (2) it also preserves the condition of the do-while loop in Line 8 of Algorithm 1—namely, that after Line 8, $\mathbf{s}^2 \leq \lfloor \beta^2 \rfloor$ remains satisfied. This condition is critical for signature acceptance during verification. Notably, the C-FN-DSA project already employs a similar trade-off in its SHAKE256X4 variant, which uses 4-way Keccak while abandoning KAT compatibility with the default version.

To ensure the correctness of our vectorized BaseSampler, we adopt a modular verification approach. We treat BaseSampler as an independent component and validate our implementation by comparing its output with the ref version when provided with identical pseudorandom number sequences. This approach can be understood as generating KATs at the subroutine level and comparing them to ensure correctness.

Steps 5 and 6 of Algorithm 3 transform the BaseSampler's output z_0 into a bimodal distribution via $z \leftarrow b + (2 \cdot b - 1)z_0$ where b is a random bit (actually retrieving 8 bits while discarding 7), followed by squaring z_0 in Step 7. Recognizing that both the bimodal transformation and squaring operations can be parallelized, we integrate these computations directly into our vectorized BaseSampler, which consequently outputs both z and z_0^2.

[7] https://github.com/GMUCERG/FALCON_NEON/blob/1d26700/falcon-armv8/neon/sampler.c#L40

Algorithm 7: Vectorized BaseSampler

Output: N independent pairs (z, z_0^2)

Constants:
  $N_s = 4$ for SSE2
  $N_s = 8$ for AVX2
  $N_s = 16$ for AVX-512F
  M is an integer such that $N = M \cdot N_s$
  RCDT_N_s: RCDT in vectorized form
Variable declarations:
  prn_24x3_N_s $prn[M]$
  ALIGNED_INT32(N) b
  $\mathbf{z}_0[0], \ldots, \mathbf{z}_0[M-1] \leftarrow 0$
`// Prepare random numbers`
for $j \leftarrow 0$ **to** $M-1$ **do**
  for $i \leftarrow 0$ **to** $N_s - 1$ **do**
    for $k \leftarrow 0$ **to** 2 **do**
      $prn[j].i32[k][i] \leftarrow$ UniformBits(24)
$bs \leftarrow$ UniformBits(N)
for $i \leftarrow 0$ **to** $N-1$ **do**
  $b.i32[i] \leftarrow (bs \gg i)\ \&\ 1$
`// Main computation loop`
for $i \leftarrow 0$ **to** 17 **do**
  $\mathbf{t}_l \leftarrow$ RCDT_$N_s[i][0]$ `// low 24-bit`
  $\mathbf{t}_m \leftarrow$ RCDT_$N_s[i][1]$ `// middle`
  $\mathbf{t}_h \leftarrow$ RCDT_$N_s[i][2]$ `// high`
  for $k \leftarrow 0$ **to** $M-1$ **do**
    $\mathbf{c} \leftarrow \mathsf{VSUB}(prn[k].\mathbf{v}[0], \mathbf{t}_l)$
    $\mathbf{c} \leftarrow \mathsf{VSRLI}(\mathbf{c}, 31)$
    $\mathbf{c} \leftarrow \mathsf{VSUB}(prn[k].\mathbf{v}[1], \mathbf{c})$
    $\mathbf{c} \leftarrow \mathsf{VSUB}(\mathbf{c}, \mathbf{t}_m)$
    $\mathbf{c} \leftarrow \mathsf{VSRLI}(\mathbf{c}, 31)$
    $\mathbf{c} \leftarrow \mathsf{VSUB}(prn[k].\mathbf{v}[2], \mathbf{c})$
    $\mathbf{c} \leftarrow \mathsf{VSUB}(\mathbf{c}, \mathbf{t}_h)$
    $\mathbf{c} \leftarrow \mathsf{VSRLI}(\mathbf{c}, 31)$
    $\mathbf{z}_0[k] \leftarrow \mathsf{VADD}(\mathbf{z}_0[k], \mathbf{c})$
`// Bimodal and squaring`
for $k \leftarrow 0$ **to** $M-1$ **do**
  $\mathbf{t}_b \leftarrow b.\mathbf{v}[k]$
  $\mathbf{t}_1 \leftarrow \mathsf{VADD}(\mathbf{t}_b, \mathbf{t}_b)$
  $\mathbf{t}_1 \leftarrow \mathsf{VSUB}(\mathbf{t}_1, 1)$
  $\mathbf{t}_2 \leftarrow \mathsf{VMULLO}(\mathbf{t}_1, \mathbf{z}_0[k])$
  $\mathbf{z}[k] \leftarrow \mathsf{VADD}(\mathbf{t}_2, \mathbf{t}_b)$
  $\mathbf{z}_0[k]^2 \leftarrow \mathsf{VMULLO}(\mathbf{z}_0[k], \mathbf{z}_0[k])$
return $(\mathbf{z}[k], \mathbf{z}_0[k]^2), k = 0, \ldots, M-1$

Algorithm 8: Vectorized BaseSampler

Output: N independent pairs (z, z_0^2)

Constants:
  $N_s = 4$ for NEON
  $N_s = 8$ for RVV with VLEN=256
  $N = M \cdot N_s$
  RCDT_N_s
Variable declarations:
  prn_24x3_N_s $prn[M]$
  ALIGNED_INT32(N) b
  $\mathbf{z}_0[0], \ldots, \mathbf{z}_0[M-1] \leftarrow 0$
`// Prepare random numbers`
for $j \leftarrow 0$ **to** $M-1$ **do**
  for $i \leftarrow 0$ **to** $N_s - 1$ **do**
    for $k \leftarrow 0$ **to** 2 **do**
      $prn[j].i32[k][i] \leftarrow$ UniformBits(24)
$bs \leftarrow$ UniformBits(N)
for $i \leftarrow 0$ **to** $N-1$ **do**
  $b.i32[i] \leftarrow (bs \gg i)\ \&\ 1$
`// Main computation loop`
for $k \leftarrow 0$ **to** $M-1$ **do**
  for $i \leftarrow 0$ **to** 17 **do**
    $\mathbf{t}_l \leftarrow$ RCDT_$N_s[i][0]$
    $\mathbf{t}_m \leftarrow$ RCDT_$N_s[i][1]$
    $\mathbf{t}_h \leftarrow$ RCDT_$N_s[i][2]$
    $\mathbf{c} \leftarrow \mathsf{VSUB}(prn[k].\mathbf{v}[0], \mathbf{t}_l)$
    $\mathbf{c} \leftarrow \mathsf{VSRLI}(\mathbf{c}, 31)$
    $\mathbf{c} \leftarrow \mathsf{VSUB}(prn[k].\mathbf{v}[1], \mathbf{c})$
    $\mathbf{c} \leftarrow \mathsf{VSUB}(\mathbf{c}, \mathbf{t}_m)$
    $\mathbf{c} \leftarrow \mathsf{VSRLI}(\mathbf{c}, 31)$
    $\mathbf{c} \leftarrow \mathsf{VSUB}(prn[k].\mathbf{v}[2], \mathbf{c})$
    $\mathbf{c} \leftarrow \mathsf{VSUB}(\mathbf{c}, \mathbf{t}_h)$
    $\mathbf{c} \leftarrow \mathsf{VSRLI}(\mathbf{c}, 31)$
    $\mathbf{z}_0[k] \leftarrow \mathsf{VADD}(\mathbf{z}_0[k], \mathbf{c})$
  `// Bimodal and squaring`
  $\mathbf{t}_b \leftarrow b.\mathbf{v}[k]$
  $\mathbf{t}_1 \leftarrow \mathsf{VADD}(\mathbf{t}_b, \mathbf{t}_b)$
  $\mathbf{t}_1 \leftarrow \mathsf{VSUB}(\mathbf{t}_1, 1)$
  $\mathbf{t}_2 \leftarrow \mathsf{VMULLO}(\mathbf{t}_1, \mathbf{z}_0[k])$
  $\mathbf{z}[k] \leftarrow \mathsf{VADD}(\mathbf{t}_2, \mathbf{t}_b)$
  $\mathbf{z}_0[k]^2 \leftarrow \mathsf{VMULLO}(\mathbf{z}_0[k], \mathbf{z}_0[k])$
return $(\mathbf{z}[k], \mathbf{z}_0[k]^2), k = 0, \ldots, M-1$

Our proposed vectorized BaseSampler is presented in Algorithms 7 and 8. Algorithm 7 targets SSE2, AVX2, and AVX-512F; Algorithm 8 is designed for NEON and RVV. Given their similar architectural approaches, we will use Algorithm 7 as the primary example for detailed explanation. The array notation and variable declarations follow a C-like style. Vector variables are denoted in bold (e.g., $\mathbf{c}$); $\mathbf{z}[]$ and $\mathbf{z_0}[]$ represent vector arrays. The underlying data structures, prn_24x3_N_s and ALIGNED_INT32, are defined in Listings 1 and 2, respectively; these listings illustrate the C-language implementations for SSE2, and the implementations for other instruction sets follow a similar pattern.

Algorithm 7 generates N independent pairs (z, z_0), where $N = M \cdot N_s$. Here, N_s denotes the parallelism granularity in 32-bit units, while M serves as a tunable parameter for performance optimization.

Unlike Step 5 of Algorithm 3, which discards 7 out of 8 pseudorandom bits during the bimodal transformation, our approach eliminates such waste. The main loop (Steps 19–33 of Algorithm 7) is a direct vectorization of Algorithm 5 and the specific optimizations for different instruction sets will be discussed in the following.

4.3 Implementations of Vectorized BaseSampler

The term *instruction interleaving* will be frequently used in the following. It refers to a technique for improving pipeline efficiency. As an example, consider the loop in Step 24 of Algorithm 7, which we will discuss later. By unrolling this loop, the bodies (Steps 25-32) of the first and second iterations are data-independent. This allows us to rearrange the instructions to reduce data dependencies, thereby enhancing pipeline performance.

4.3.1 SSE2, AVX2, and AVX-512F

Our implementations for SSE2, AVX2, and AVX-512F utilize Intel intrinsics. While maintaining the loop structures in Steps 12, 13, and 20 of Algorithm 7, we fully unroll the loops in Steps 24 and 35 to improve pipeline efficiency through instruction interleaving. The loop in Step 17 is partially unrolled. We attempted complete unrolling of Step 20's loop, but this resulted in performance degradation.

For {SSE2, AVX2} versions, our optimization leverages the {pcmpgtd, vpcmpgtd} instructions respectively, which compare packed signed 32-bit integers between two operands, producing -1 for greater-than conditions and 0 otherwise. On Rocket Lake processors, {v}paddd/{v}psubd[8], {v}psrld, and {v}pcmpgtd exhibit latency/CPI of 1/0.33, 1/0.5, and 1/0.5, respectively. Consequently, Steps 25–27 in Algorithm 7 are optimized into two instructions: $\mathbf{c} \leftarrow \mathsf{VCMPGT}(\mathbf{t}_l, prn[k].\mathbf{v}[0])$; $\mathbf{c} \leftarrow \mathsf{VADD}(\mathbf{c}, prn[k].\mathbf{v}[1])$.

The AVX-512F version shows different characteristics, with vpaddd/vpsubd, vpsrld, and vpcmpgtd having latency/CPI of 1/0.5, 1/1, and 3/1 respectively. Due to the higher latency of vpcmpgtd, which could cause pipeline stalls, we did not apply the aforementioned optimization to this version.

For the AVX2 version ($N = 16$, $M = 2$), our tests demonstrate that $M = 2$ outperforms $M = 1$ by better utilizing the 16 available registers and enabling manual loop unrolling in Step 24 for improved pipeline efficiency by instruction interleaving.

Similarly, the AVX-512F version ($N = 32$, $M = 2$) follows the same optimization approach. While $M = 4$ might theoretically better utilize the 32 available ZMM registers, we maintain $M = 2$ for consistency of loop structures with the AVX2 version.

In the SSE2 version, we set $N = 16$ and $M = 4$ for consistency of top-level interfaces with the AVX2 version. Limited to 8 registers, we unroll Step 24's loop of Algorithm 7 only twice, handling the remaining iterations outside Step 19's main loop.

[8]The VADD operation corresponds to the paddd instruction in SSE2 and vpaddd in AVX2. We uniformly represent this operation as {v}paddd, where the curly braces indicate the optional 'v' prefix in AVX2.

Compatibility on Haswell. As the first microarchitecture supporting AVX2 and one of the recommended benchmark platforms for the NIST PQC project, we analyze the applicability of our implementation to Haswell processors. Since Haswell does not support AVX-512, we focus our discussion on {SSE2, AVX2} versions. The latency/CPI of {v}paddd/{v}psubd, {v}psrld, and {v}pcmpgtd are 1/0.5, 1/1, and 1/0.5 respectively, making our optimization using the {v}pcmpgtd instruction also suitable for Haswell.

AVX-SSE transition penalties. As discussed in Section 3, compilers typically generate SSE2 instructions for floating-point operations. Frequent mixing of SSE and AVX instructions can lead to transition penalties [Kon11] that degrade performance. For our AVX2 and AVX-512F versions, we employ a batch processing approach where AVX instructions are used intensively to generate N samples consecutively. This design minimizes the frequency of AVX-SSE transitions, thereby reducing the performance penalties.

Algorithm 9: Batched BaseSampler

Output: N independent pairs (z, z_0^2)

Constants:
  N: batch size
  RCDT: in "64+8"-bit form
Variable declarations:
  uint64_t $prn[N][2]$
  int32_t $b[N]$
  $z_0[0], \ldots, z_0[N-1] \leftarrow 0$
// Prepare random numbers
for $j \leftarrow 0$ **to** $N-1$ **do**
  $prn[j].[0] \leftarrow \mathsf{UniformBits}(64)$
  $prn[j].[1] \leftarrow \mathsf{UniformBits}(8)$
$bs \leftarrow \mathsf{UniformBits}(N)$
for $j \leftarrow 0$ **to** $N-1$ **do**
  $b[j] \leftarrow (bs \gg j)\ \&\ 1$
// Main computation loop
for $j \leftarrow 0$ **to** N **do**
  for $i \leftarrow 0$ **to** 17 **do**
    // Set to 1 if less than
    $c \leftarrow \mathsf{SLTU}(prn[j].[0], \mathsf{RCDT}[i][0])$
    $c \leftarrow prn[j].[1] - c$
    $c \leftarrow c - \mathsf{RCDT}[i][1]$
    $c \leftarrow c \gg 63$
    $z_0[j] \leftarrow z_0[j] + c$
// Bimodal and squaring
for $j \leftarrow 0$ **to** $N-1$ **do**
  $t_b \leftarrow b[j]$
  $t_1 \leftarrow t_b + t_b$
  $t_1 \leftarrow t_1 - 1$
  $t_2 \leftarrow t_1 \cdot z_0[j]$
  $z[j] \leftarrow t_2 + t_b$
  $z_0[j]^2 \leftarrow z_0[j] \cdot z_0[j]$
return $(z[j], z_0[j]^2), j = 0, \ldots, N-1$

Algorithm 10: Implementation of Algorithm 8's Steps 24-32 using RVV with SEW=32

Input: vil, vim, vih: 72-bit integers in 3×24-bit format; vtl, vtm, vth: Required RCDT entries in vectorized form; vr: Accumulator
Output: vr: Updated accumulator

```
vmsbc.vv  v0, vil, vtl
vmsbc.vvm v0, vim, vtm, v0
vmsbc.vvm v0, vih, vth, v0
vadd.vi   vr, vr, 1, v0.t
return vr
```

Algorithm 11: Implementation of Algorithm 8's Steps 24-32 using NEON

Input: vil, vim, vih: 72-bit integers in 3×24-bit format; vtl, vtm, vth: Required RCDT entries in vectorized form; vr: Accumulator
Output: vr: Updated accumulator

```
cmgt vt0.4s, vtl.4s, vil.4s
sub  vt1.4s, vim.4s, vtm.4s
add  vt1.4s, vt1.4s, vt0.4s
ushr vt1.4s, vt1.4s, #31
sub  vt2.4s, vih.4s, vth.4s
sub  vt2.4s, vt2.4s, vt1.4s
usra vr.4s, vt2.4s, #31
return vr
```

```
typedef union
{
  int32_t i32[3][4];
  __m128i v[3];
} prn_24x3_4;
```

Listing 1: prn_24x3_4 on SSE2

```
#define ALIGNED_INT32(N)  \
  union {                 \
    int32_t coeffs[N];    \
    __m128i v[(N+3)/4];   \
  }
```

Listing 2: ALIGNED_INT32 on SSE2

4.3.2 RVV

The vectorized BaseSampler for RVV is presented in Algorithm 8. Unlike the SSE2 and AVX2 versions, we avoid comparison instructions (e.g., vmsgt.vv in RVV) because they produce mask outputs that cannot directly participate in arithmetic operations. Similar to the SSE2, AVX2, and AVX-512F versions, the RVV implementation partitions 72-bit integers into three 24-bit limbs. The loop structures in Steps 11, 12, and 19 of Algorithm 8 are maintained, while the loops in Steps 13 and 20 are completely unrolled. The loop in Step 16 is partially unrolled, with Steps 19–39 implemented in hand-optimized assembly.

Among the 54 total 24-bit integer segments (18 entries $\times$ 3 limbs) in the RCDT table, 12 segments are zero-valued, leaving 42 non-zero segments. We utilize 24 scalar registers[9] and 18 vector registers to store these segments. Following a *load-once-use-many* strategy, all RCDT entries are loaded before Step 19, enabling subsequent M iterations to access the values directly from registers without memory access.

For 72-bit integer comparison (in 3×24-bit format) in Steps 24–31 of Algorithm 8, RVV offers two methods: (1) Method 1: Direct implementation using vsub.vv and vsrl.vi instructions to instantiate the VSUB and VSRLI operations. (2) Method 2: Subtraction-with-borrow using vmsbc.vv and vmsbc.vvm instructions, as detailed in Algorithm 10.

The RVV mandates that mask registers must use v0. Exclusive use of Method 2 would cause pipeline stalls due to vmsbc's 4-cycle latency (as shown in Table 1) and the dependency chain through v0 that prevents instruction interleaving. To mitigate this, we completely unroll the Step 20 loop and package three iterations into one macro-operation: one using Method 2 and two using Method 1, with careful instruction interleaving to improve pipeline efficiency.

Our RVV version adopts $N = 64$, achieving 20% performance gain over $N = 16$ when excluding the overhead of pseudorandom number generation.

While the RV64IM version described in Section 4.4 leverages 12-bit signed immediates to accommodate RCDT entries, this approach is impractical for RVV due to two constraints: RVV instructions (e.g., vadd.vi) only permit 5-bit signed immediates, and the vmsbc instruction lacks an immediate variant.

Applicability for RVV with VLEN=128. The XuanTie C908 core [TH23], which is the target platform of [ZYHK25], has a vector length of 128 bits. The performance characteristics of the vadd/vsub, vsrl, and vmsbc instructions in XuanTie C908 are similar to those in SpacemiT X60 core used in this work. Therefore, the implementation method and register usage strategy we employed are also suitable for XuanTie C908.

4.3.3 NEON

The NEON version follows a similar approach to the RVV version, with two key distinctions: (1) the RCDT table access pattern, and (2) the instruction sequence used for Steps 24–32 of Algorithm 8. We elaborate on these differences below.

The RVV version leverages the vx instruction feature, which allows scalar registers to both cache RCDT entries and participate directly in vector arithmetic operations. Since

[9] RVV permits using scalar registers as a source operand (instructions with .vx suffix)

NEON lacks comparable functionality, we adopt an alternative strategy: persistently storing 20 table entries across 20 vector registers using the *load-once-use-many* approach, while loading the remaining entries on demand.

Algorithm 11 details the NEON version of Steps 24–32 in Algorithm 8. Similar to SSE2 and AVX2 versions, we employ the cmgt (compare greater than) instruction. The ushr represents unsigned shift right, and usra represents unsigned shift right and accumulate. According to [ARM15], the performance characteristics on Cortex-A72 are as follows: cmgt/add/sub (3 cycles, CPI 0.5); ushr (3 cycles, CPI 1); usra (4 cycles, CPI 1).

Similar to the RVV version, we completely unroll Step 20 of Algorithm 8 and package three iterations into a single assembly macro. This design enables extensive instruction interleaving to enhance pipeline efficiency. Our NEON version adopts $N = 64$, which demonstrates a modest performance improvement of approximately 3% compared to $N = 16$. The remaining implementation details closely mirror those of the RVV version.

4.4 Implementation of Batched BaseSampler on RV64IM

Our scalar implementation for 64-bit RISC-V, designated as the RV64IM version, utilizes exclusively the I and M instruction subsets. Note that the I instruction subset does not provide carry and borrow functionality, so we use subtraction followed by shifting to obtain the borrow. The fundamental approach is outlined in Algorithm 9, with the following optimizations.

Our implementation partitions 72-bit integers into the lower 64 bits and the upper 8 bits. This design capitalizes on immediate instructions such as addi, where the second source operand is a 12-bit signed immediate value, thereby enabling direct usage of the RCDT entries' high 8 bits without memory access. Among the 18 RCDT entries' lower 64 bits, two values are smaller than 2^{11}, permitting storage of remaining entries in just 16 registers. Given 30 available registers, we implement a *load-once-use-many* strategy for RCDT entries, which enhances performance through Algorithm 9's batch approach. Specifically, we complete loading 16 RCDT entries into registers before Step 16 of Algorithm 9. Consequently, during the N iterations in Step 16, the algorithm directly accesses the register-resident RCDT entries, eliminating redundant memory operations.

Our implementation uses $N = 64$, demonstrating approximately 6% performance improvement over $N = 16$ when excluding pseudorandom number generation overhead. We employ carefully hand-optimized assembly code to implement Steps 15–23 of Algorithm 9. The loop in Step 17 is completely unrolled to facilitate two key optimizations: identification of RCDT entries accessible via immediate instructions, and strategic instruction interleaving to improve pipeline efficiency. Steps 19 and 21 utilize slt{i}u and add{i} instructions respectively, where the i denotes immediate variants and add{i} achieves subtraction through negation of its second operand with no extra cost.

4.5 Integration of Optimized BaseSampler Implementation into Falcon

Our implementation of BaseSampler employs a batch processing strategy across all supported instruction sets (SSE2, AVX2, AVX-512F, RVV, NEON, and RV64IM). This approach, which generates multiple samples per subroutine call, follows established practice in the field. As demonstrated in [BBCT22, ZHZ+24] for batched key generation, we similarly implement a dedicated storage structure—GAUSSIAN0_STORE—to maintain batches produced by our BaseSampler.

In our implementation, GAUSSIAN0_STORE can hold up to 128 samples, occupying 1 KB of memory. In memory-constrained scenarios, implementers may reduce this size while also decreasing the parameter N. We will invoke our BaseSampler $128/N$ times to replenish it when the structure is depleted. From an implementation perspective, this modification primarily affects Steps 4–6 of Algorithm 3, where we now access GAUSSIAN0_STORE to

Table 4: Benchmark results of various BaseSampler implementations, where "ref" represents the reference implementation from the C-FN-DSA project, and "AVX2-ref" is discussed in Section 4.1. The "core" suffix indicates measurements that exclude pseudorandom number generation overhead. All our implementations employ the batch strategy, and the reported cycle and instruction counts are averaged per sample. Cycles: CPU cycles consumed. Ins: CPU instructions consumed. Results averaged over >160,000 runs.

Version	Cycles	Speedup	Ins.
SSE2			
ref	131	1.0×	641
Our	90	1.5×	423
ref core	59	1.0×	257
Our core	14	4.2×	50
AVX2			
ref	95	1.0×	396
AVX2-ref	66	1.4×	247
Our	49	1.9×	190
ref core	59	1.0×	223
AVX2-ref core	44	1.3×	71
Our core	7	8.4×	30
ARM Cortex-A72			
ref	175	1.0×	291
[NG23] NEON	180	0.97×	297
Our NEON	137	1.3×	243
ref core	54	1.0×	50
[NG23] core	59	0.92×	54
Our core	30	1.8×	35

Version	Cycles	Speedup	Ins.
AVX-512F			
ref	71	1.0×	302
AVX2-ref	42	1.7×	152
Our	26	2.7×	88
ref core	59	1.0×	223
AVX2-ref core	43	1.4×	69
Our core	6	9.8×	15
SpacemiT X60			
ref	395	1.0×	645
Our RV64IM	216	1.8×	398
Our RVV	196	2.0×	324
ref core	192	1.0×	287
Our RV64IM core	51	3.8×	98
Our RVV core	25	7.7×	19

obtain a pair (z, z_0^2). Here, z represents the result of applying the bimodal transformation to z_0.

A natural concern is the computational overhead introduced by accessing and managing GAUSSIAN0_STORE. Our measurements demonstrate that this structure adds minimal latency, requiring fewer than 5 clock cycles per sample access on average.

4.6 Benchmarks of Optimized BaseSampler

The target platforms and benchmark configurations are detailed in Section 2.4. The benchmarking results for BaseSampler are presented in Table 4. The RV64IM and RVV versions are compiled using the -march flags rv64gc_zba_zbb and rv64gcv_zba_zbb respectively. The inclusion of the B extension is due to our integration of optimized Keccak that leverages this extension.

For all performance comparisons, we ensure equivalent Keccak implementations are used. This means the reported improvements do not stem from any Keccak optimizations. The specific Keccak implementation employed will be detailed in Section 6. To ensure a fair comparison, we fine-tune the reference implementation to include the bimodal transformation and squaring operation.

For the AVX2 and AVX-512F implementations, we consider the ref version as the baseline rather than AVX2-ref, as the C-FN-DSA project uses the ref version. For a more accurate assessment of our advantages, we recommend focusing on the results that exclude pseudorandom number generation overhead (denoted by the "core" suffix).

Our approach demonstrates significant speedups across multiple instruction sets: 4.2×

for SSE2, 8.4× for AVX2, 9.8× for AVX-512F, 7.7× for RVV, and 3.8× for RV64IM, with a modest 1.8× improvement for NEON. The NEON implementation in [NG23] is even slower than the reference implementation on Cortex-A72.

An interesting observation is that on AVX-512F, the "AVX2-ref" and "AVX2-ref core" exhibit similar performance, suggesting that the Keccak computation might be "free". We attribute this to data dependencies between AVX2 instructions in the "AVX2-ref core", which cause pipeline stalls. The addition of Keccak-related instructions introduces new operations that do not depend on previous AVX2 instructions, allowing the out-of-order execution capability of the processor to improve pipeline efficiency.

5 Vectorized FFT

As shown in Table 3, FFT-related subroutines account for only 15.1% of execution time on Intel i7-11700K (AVX2 version), but contribute significantly (37.6%) on SpacemiT X60. This section consequently focuses on optimizing these subroutines specifically for RISC-V.

In the C-FN-DSA project, the most computationally intensive FFT-related subroutines on SpacemiT X60 (ordered by execution time) are: FFT, LDL_fft, split_selfadj_fft, split_fft, merge_fft, iFFT, mul_fft, and gram_fft. Among these, FFT demonstrates the highest computational overhead. Since iFFT shares similar optimization characteristics with FFT, our discussion will primarily focus on these two components.

As discussed in Section 2.3, layer 0 of FFT performs no arithmetic operations, instead combining pairs of floating-point numbers into complex representations $(f_0, f_{n/2}) \to f_0 + if_{n/2}$. Subsequent layers operate on these complex numbers, making layer 1 the starting point for our optimization analysis.

5.1 Previous Implementations

The C-FN-DSA project provides vectorized implementations of FFT-related subroutines using SSE2 and NEON, but only achieves 2-way parallelism without employing layer merging strategies. Layer merging, a well-established technique in NTT implementations, aims to reduce memory access overhead by loading multiple coefficients into registers and performing multiple computation layers without extra memory access.

[NG23] optimized Falcon's FFT using NEON with layer merging strategies: 2+2+4 and 1+2+2+4 for Falcon-{512,1024} respectively. A natural question arises regarding why we cannot implement 4+4 merging for Falcon-512 similar to Kyber NTT's 4+3 merging strategy on NEON [BHK+22]. There are two primary reasons: (1) Kyber operates on 16-bit integers, whereas Falcon's FFT uses 64-bit double-precision floating-point numbers; (2) In Falcon-512's FFT, the stride between butterfly unit inputs decreases from 128 coefficients (1024 bytes) in layer 1 to 16 coefficients (128 bytes) in layer 4. NEON lacks the necessary strided load instructions to load coefficients with a 128-byte stride, which limits the efficient implementation of layers 1–4 merging.

5.2 Optimized Implementations

5.2.1 RVV

RVV provides strided load/store instructions (e.g., vlse64.v and vsse64.v). As shown in Table 1, these instructions exhibit minimal performance overhead compared to contiguous accesses (CPI of 4 for vlse64.v versus 3 for vle64.v). This architectural feature enables us to efficiently implement layers 1–4 merging. We employ hand-written assembly to optimize FFT-related subroutines, with particular focus on FFT and iFFT.

Register allocation. Our implementation utilizes 16 vector registers to accommodate 32 complex numbers, with the remaining registers allocated for temporary values and

precomputed roots of ϕ. For layers requiring scalar roots (e.g., layers 1–2), we leverage the D extension's floating-point registers (f0–f31) to reduce pressure on vector registers. This optimization is enabled by RVV's support for vf-format instructions that accept a scalar floating-point operand.

Layer merging strategy. For FFT of Falcon-{512,1024}, we propose novel 4+4 and 4+5 layer merging strategies, respectively. To our knowledge, these approaches represent the first application of such deep merging for double-precision floating-point FFT implementations. This contrasts with the previously known optimal NEON strategies of 2+2+4 and 1+2+2+4 proposed by [NG23].

Layers 1–4 merging. The first four layers merging employ strided load/store instructions to directly construct the required coefficient arrangements in vector registers. After layers 2 and 3, we perform shuffling operations using vrgather and vmerge instructions to prepare the required coefficient arrangements of subsequent layers.

Layers 5–8 and 5–9 merging. The remaining layers merging (5–8 for Falcon-512, 5–9 for Falcon-1024) utilize contiguous memory operations (i.e., vle64.v and vse64.v) with similar shuffling patterns. The 5-layer merging is viable since 16 vector registers are sufficient to store 32 complex numbers (the minimum requirement for 5-layer merging), with the remaining registers available for temporary values and roots. This differs from NEON, where the 32 available registers are fully occupied by 32 complex numbers, preventing the use of additional registers for intermediate computations.

We encapsulate two independent CT or GS butterfly units (which operate on complex numbers) into a macro. This approach ensures that any set of four (and sometimes even eight) instructions in the core computation have no data dependencies between them, thereby improving pipeline efficiency.

While sharing the register allocation and layer merging strategies with FFT, iFFT exhibits two key distinctions: (1) it employs the GS butterfly unit rather than the CT approach, and (2) it requires a final multiplication by the factor $\frac{1}{2^n}$.

We intentionally avoid vector fused multiply-accumulate instructions due to rounding behavior differences from separate multiply+add operations, which could introduce numerical discrepancies from the reference implementation, as discussed in [NG23, Sec 5.6]. A similar reason applies to avoiding the fusion of multiplications with $\frac{1}{2^n}$ into precomputed roots.

In addition to FFT and iFFT, we also vectorize several FFT-related subroutines: LDL_fft, split_selfadj_fft, split_fft, merge_fft, and mul_fft.

Applicability for RVV with VLEN=128. Our layers 1–4 merging strategy remains viable (requiring exactly 16 registers for 16 complex numbers). However, 5-layer merging becomes impractical as it requires 32 complex numbers. For Falcon-1024, we recommend 4+1+4 or 4+4+1 strategies, which still outperform NEON's 1+2+2+4 approach in memory access efficiency.

Applicability for AVX-512F. AVX2's vgatherdpd instruction offers strided load functionality, and the corresponding strided store instruction vscatterdpd requires AVX-512F. Two factors limit potential performance gains when implementing layers 1–4 merging using AVX-512F: (1) significant instruction overhead (CPI of 5 for vgatherdpd versus 0.5 for vmovapd on Rocket Lake), and (2) AVX-SSE transition penalties.

5.2.2 RV64D

Our implementation, designated as the RV64D version, primarily utilizes the RISC-V D extension for double-precision floating-point operations. The D extension provides 32 available double-precision registers (f0–f31). We employ hand-written assembly to optimize FFT and iFFT subroutines.

Register allocation strategy. The implementation allocates 16 registers to store 8 complex numbers, while the remaining 16 registers hold temporary values and precom-

Table 5: Benchmark results of FFT/iFFT implementations on SpacemiT X60. The performance of the ref implementation on RVV and RV64D is the same, so it is only reported once. Cycles: CPU cycles consumed. Ins: CPU instructions consumed. Results averaged over 5,000 runs.

Version	Strategy	Cycles	Speedup	Ins.	Strategy	Cycles	Speedup	Ins.
	FALCON-512				FALCON-1024			
FFT ref	ref	35738	1.0×	26289	ref	80524	1.0×	58358
RV64D	3+3+2	12818	2.8×	16937	3+3+3	27115	3.0×	36245
RVV	4+4	8290	4.3×	5008	4+5	17181	4.7×	10793
iFFT ref	ref	34728	1.0×	28859	ref	76652	1.0×	63470
RV64D	2+3+3	12814	2.7×	17522	3+3+3	27074	2.8×	37474
RVV	4+4	8400	4.1×	5235	5+4	17974	4.3×	11060

puted roots of ϕ. To relieve register pressure, we temporarily borrow 4 integer registers to cache roots, transferring values to floating-point registers via fmv.d.x when needed.

Layer merging approach. For FFT of FALCON-{512,1024}, we adopt 3+3+2 and 3+3+3 layer merging strategies, respectively. The constraint of storing 8 complex numbers in 16 registers limits the maximum merged layers to 3.

Similar to our RVV implementation, the RV64D version encapsulates two independent butterfly units within a single macro to enhance pipeline efficiency. We deliberately avoid using the fused multiply-accumulate instructions, following the same rationale in the RVV implementation.

5.3 Benchmarks of Optimized FFT

The target platform is the SpacemiT X60, with detailed benchmark configurations provided in Section 2.4. Table 5 presents performance comparisons of various FFT and iFFT implementations. The RV64D and RVV versions are compiled using the -march flags rv64gc and rv64gcv respectively. While we vectorize several other FFT-related subroutines, we omit their performance results as they contribute minimally to the overall signature generation time.

Consider FALCON-1024 as an example, our RV64D version achieves 3.0× and 2.8× speedups for FFT and iFFT, respectively, compared to the reference implementation. The RVV version shows even greater enhancements, with more than 4× speedups.

6 Optimized Keccak

The C-FN-DSA project provides optimized Keccak implementations using SSE2 and AVX2, which we use without modification. For our AVX-512F version, we integrate the 8-way Keccak implementation from the XKCP project[10] with full round unrolling. Based on this, we develop the SHAKE256X8 variant exclusively for our AVX-512F version, while all other versions use the SHAKE256X4 variant.

We choose not to implement Keccak using RVV due to the SpacemiT X60's vector logic instruction characteristics. With a CPI of 2 for vector logic instructions and an effective CPI of 0.5 per 64-bit logic operation (equivalent to scalar operations), we directly adopt the RV64I and RV64IB implementations of Keccak from [ZYHK25]. Specifically, our RV64GC{V} versions use the RV64I implementation of Keccak, and our RV64GC{V}B versions use the RV64IB implementation of Keccak from [ZYHK25].

[10] https://github.com/XKCP/XKCP/ at commit id: 66069fa

Table 6: Benchmark results of Falcon-{512,1024}'s signature generation (sign_core subroutine) on three target platforms (8 distinct instruction set configurations). Cycles: CPU cycles consumed ($k = 1{,}000$). Results represent median values from: 10,000 runs on Intel i7-11700K and Cortex-A72, and 2,000 runs on SpacemiT X60.

Version	**Cycles**	**Speedup**	**Cycles**	**Speedup**	**Cycles**	**Speedup**	**Cycles**	**Speedup**
	Falcon-512		Falcon-1024		Falcon-512		Falcon-1024	
	SSE2				RV64GC			
ref	631k	1.00×	1266k	1.00×	2725k	1.00×	5587k	1.00×
Our	556k	1.13×	1101k	1.15×	1982k	1.37×	4127k	1.35×
	AVX2				RV64GCB			
ref	543k	1.00×	1104k	1.00×	2535k	1.00×	5262k	1.00×
Our	441k	1.23×	894k	1.23×	1867k	1.36×	3908k	1.35×
	AVX-512F				RV64GCV			
ref	536k	1.00×	1086k	1.00×	2730k	1.00×	5610k	1.00×
Our	393k	1.36×	839k	1.29×	1729k	1.58×	3527k	1.59×
	NEON				RV64GCVB			
ref	1230k	1.00×	2495k	1.00×	2530k	1.00×	5213k	1.00×
Our	1053k	1.17×	2183k	1.14×	1590k	1.59×	3292k	1.58×

For our NEON version, we incorporate the 4-way hybrid Keccak implementation (keccak_f1600_x4_hybrid_asm_v3p) from [BK22], which our testing shows to be 23% faster than the C implementation on Cortex-A72. Additionally, we integrate the optimized FFT/iFFT using NEON from [NG23].

7 Results and Comparisons

Table 6 presents the benchmarks of different implementations of Falcon-{512,1024}'s signature generation across three target platforms with eight distinct instruction sets. We use Falcon-512 as an example to illustrate our performance improvements.

For implementations using SSE2 and AVX2, the performance improvements are 13% and 23%, respectively, compared to the reference implementation. These gains are entirely attributed to our optimized BaseSampler implementation (Section 4.3.1).

For implementations using AVX-512F, the improvement reaches 36% over the reference implementation. This result is partially due to the 8-way Keccak (Section 6) and partially due to our optimized BaseSampler (Section 4.3.1). Integrating the vectorized BaseSampler alone—without additional optimizations—yields a 21% speedup over the reference implementation. The subsequent incorporation of the 8-way Keccak further amplifies performance. The intensive use of AVX-512F instructions in both components produces a synergistic effect that exceeds the sum of their individual contributions.

For implementations using NEON, the performance improvement is 17% compared to the reference implementation. If we exclude the 4-way hybrid Keccak (from [BK22]) and optimized FFT/iFFT (from [NG23]), the improvement reduces to 9%. Integrating our BaseSampler (Section 4.3.3) with the 4-way hybrid Keccak results in a 13% improvement over the reference implementation. We do not report results of [NG23] because its implementation uses ChaCha20 for pseudorandom number generation.

All four versions on RISC-V show significant improvements. Our RV64GC{B} versions benefit from the RV64IM-optimized BaseSampler (Section 4.4), the RV64D-optimized FFT/iFFT (Section 5.2.2), and the optimized Keccak (Section 6). Our RV64GCV{B} versions benefit from the RVV-optimized BaseSampler (Section 4.3.2), the RVV-optimized FFT-related subroutines (Section 5.2.1), and the optimized Keccak (Section 6). For all

four versions, integrating only the optimized BaseSampler yields a performance improvement of 10%∼12% compared to the reference implementation. When both the optimized BaseSampler and the optimized FFT-related subroutines are incorporated, the overall speedup increases to 22%∼41%.

Code size and memory footprint. As an example, our implementation using AVX2 increases the code size by approximately 2.7 KB compared to the reference implementation. The increase in memory footprint is mainly due to the GAUSSIAN0_STORE structure, which accounts for about 1 KB.

8 Scalability and Security

The scalability of our optimized BaseSampler for Haswell and RVV with VLEN=128 is discussed in Sections 4.3.1 and 4.3.2, respectively. For those aiming to implement our BaseSampler on RV32IM, we recommend using a 3×24-bit format. The register allocation strategy outlined in Section 4.4 and the batch strategy from Algorithm 9 can serve as useful references.

Regarding the applicability of our optimization approach to HAWK scheme [BBD+24], we note that the sampler in HAWK differs significantly from FALCON at the implementation level. Therefore, we consider this as future work.

The scalability of the FFT optimization strategy for AVX-512F and RVV with VLEN=128 is discussed in Section 5.2.1.

All instructions (including comparison) used in this work execute in constant time. Our optimized BaseSampler does not affect the rejection sampling behavior of the discrete Gaussian sampler (Algorithm 3), and therefore does not affect its constant-time promise, that is, the efficiency is independent of the Gaussian centre and standard deviation.

The replenishment of the GAUSSIAN0_STORE structure when its samples are exhausted, as well as the RCDT table access in our optimized BaseSampler implementation, follow fixed patterns that are independent of the algorithm's execution context. Thus, they do not introduce new attack surfaces. The samples stored in the GAUSSIAN0_STORE structure are kept in memory, and memory safety is ensured by the operating system.

Compared to the reference implementation, our optimized BaseSampler changes the order in which pseudorandom numbers are used. However, this does not affect the security of FALCON, as its security does not depend on the order of pseudorandom number usage. The consequence is that our signature generation is not KAT-compatible with the reference implementation, but this does not affect interoperability with verification (see Section 4.2).

9 Conclusion and Future Work

Conclusion. Our main contribution is the vectorized implementation of BaseSampler for FALCON signature generation. We provide implementations on various instruction sets, including SSE2, AVX2, AVX-512F, NEON, RVV, and RV64IM, demonstrating significant performance improvements that highlight the advantages of our method. Additionally, we optimize FFT/iFFT using RVV and RV64D, where the RVV implementation leverages a novel approach through strided load/store instructions with 4+4 and 4+5 layer merging strategies. Ultimately, our implementation of FALCON signature generation achieves performance improvements across eight different instruction sets.

Future Work. The C-FN-DSA project uses SSE2 for floating-point operations, which is a good choice from a compatibility perspective. However, when using AVX2 or AVX-512 to accelerate certain subroutines, an AVX-SSE transition penalty may occur. Therefore, we recommend that future implementers pay attention to this issue. Profiling our AVX-512F implementation revealed that BerExp (Algorithm 6) accounts for as much as 36%

of the signature generation. It lacks opportunities for vectorization and exhibits strict step-by-step dependencies, leading to suboptimal pipeline efficiency. We therefore suggest future research focus on alternative algorithms for BerExp, aiming to improve both vectorization potential and pipeline efficiency.

Acknowledgments

We would like to express our gratitude to Thomas Pornin, Hao Cheng, Junhao Huang, and the anonymous reviewers for their thoughtful discussions and constructive feedback.

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. Technical Report 8413, National Institute of Standards and Technology, July 2022. Accessed: July 13, 2025.

[AR19] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS '19*, pages 673–686, New York, NY, USA, 2019. ACM.

[ARM15] ARM. Cortex-A72 Software Optimization Guide, 2015.

[ASA+25] Ghada Alsuhli, Hani H. Saleh, Mahmoud Al-Qutayri, Baker Mohammad, and Thanos Stouraitis. Area and Power Efficient FFT/IFFT Processor for FALCON Post-Quantum Cryptography. *IEEE Trans. Emerg. Top. Comput.*, 13(2):423–437, 2025.

[BBCT22] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 845–862. USENIX Association, 2022.

[BBD+24] Joppe W. Bos, Olivier Bronchain, Léo Ducas, Serge Fehr, Yu-Hsuan Huang, Thomas Pornin, Eamonn W. Postlethwaite, Thomas Prest, Ludo N. Pulles, and Wessel van Woerden. HAWK version 1.0.2. https://hawk-sign.info, September 2024.

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):221–244, 2022.

[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS+ on AArch64. In Takanori Isobe and Santanu Sarkar, editors, *Progress in Cryptology – INDOCRYPT 2022*, pages 272–293, Cham, 2022. Springer International Publishing. https://eprint.iacr.org/2022/1243.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[CYS25] Junhyeok Choi, Seungyong Yoon, and Seog Chung Seo. Optimized Falcon Verify on Cortex-M4 for Post-Quantum secure UAV communications. *ICT Express*, 11(1):281–286, 2025.

[DTN+25] Duc-Thuan Dam, Thai-Ha Tran, Trong-Hung Nguyen, Trong-Thuc Hoang, and Cong-Kha Pham. Compact FALCON FFT/NTT Accelerator for Post-Quantum Cryptography. In *IEEE International Symposium on Circuits and Systems, ISCAS 2025, London, United Kingdom, May 25-28, 2025*, pages 1–5. IEEE, 2025.

[FHK+20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU (Specification v1.2 — 01/10/2020), 2020. https://falcon-sign.info/falcon.pdf.

[GS66] W. Morven Gentleman and G. Sande. Fast Fourier Transforms: for fun and profit. volume 29 of *AFIPS Conference Proceedings*, pages 563–578, 1966.

[HPRR20] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous Gaussian Sampling: From Inception to Implementation. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, volume 12100 of *Lecture Notes in Computer Science*, pages 53–71. Springer, 2020.

[KA24] Emre Karabulut and Aydin Aysu. A Hardware-Software Co-Design for the Discrete Gaussian Sampling of FALCON Digital Signature. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2024, Tysons Corner, VA, USA, May 6-9, 2024*, pages 90–100. IEEE, 2024.

[Kon11] Patrick Konsor. Avoiding AVX-SSE transition penalties, 2011.

[KSS22] YoungBeom Kim, Jingyo Song, and Seog Chung Seo. Accelerating Falcon on ARMv8. *IEEE Access*, 10:44446–44460, 2022.

[NG23] Duc Tri Nguyen and Kris Gaj. Fast Falcon signature generation and verification using ARMv8 NEON instructions. In *International Conference on Cryptology in Africa*, pages 417–441. Springer, 2023.

[OSHG19] Tobias Oder, Julian Speith, Kira Höltgen, and Tim Güneysu. Towards Practical Microcontroller Implementation of the Signature Scheme Falcon. In Jintai Ding and Rainer Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019, Chongqing, China, May 8-10, 2019 Revised Selected Papers*, volume 11505 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2019.

[OZZ+25] Yi Ouyang, Yihong Zhu, Wenping Zhu, Bohan Yang, Zirui Zhang, Hanning Wang, Qichao Tao, Min Zhu, Shaojun Wei, and Leibo Liu. FalconSign: An Efficient and High-Throughput Hardware Architecture for Falcon Signature Generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):203–226, 2025.

[Por19] Thomas Pornin. New Efficient, Constant-Time Implementations of Falcon. https://eprint.iacr.org/2019/893, 2019.

[RIS21a] RISC-V Foundation. RISC-V Bit-Manipulation ISA-extensions Version 1.0.0-2021-06-12. https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf, 2021.

[RIS21b] RISC-V Foundation. RISC-V V Vector Extension Version 1.0-rc1-20210608. https://github.com/riscv/riscv-v-spec/releases/download/v1.0-rc1/riscv-v-spec-1.0-rc1.pdf, 2021.

[RIS21c] RISC-V Foundation. RISC-V Zvkb - Vector Cryptography Bit-manipulation. https://github.com/riscv/riscv-crypto/blob/main/doc/vector/riscv-crypto-vector-zvkb.adoc, 2021. Commit: 3d9dff8.

[TH23] T-Head. XuanTie-C908-UserManual. https://www.xrvm.com/product/xuantie/C908, 2023. Accessed: 2023-10-01.

[YSZ+24] Xinglong Yu, Yi Sun, Yifan Zhao, Honglin Kuang, and Jun Han. RVCE-FAL: A RISC-V Scalar-Vector Custom Extension for Faster FALCON Digital Signature. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2024, Valencia, Spain, March 25-27, 2024*, pages 1–6. IEEE, 2024.

[ZASM25] Binke Zhao, Ghada Alsuhli, Hani H. Saleh, and Baker Mohammad. BiSamplerZ: A Hardware-Efficient Gaussian Sampler Architecture for Quantum-Resistant Falcon Signatures. *CoRR*, abs/2505.24509, 2025.

[ZHZ+24] Jipeng Zhang, Junhao Huang, Lirui Zhao, Donglong Chen, and Çetin Kaya Koç. ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

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

[ZYHK25] Jipeng Zhang, Yuxing Yan, Junhao Huang, and Çetin Kaya Koç. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):632–655, 2025.