Falcon Verify on AVX-512: Speed Records

David Rubin and Emanuele Cesena

Jump Crypto
drubin@jumpcrypto.com,ecesena@jumpcrypto.com

July 27, 2026

Abstract.
We present a fast implementation of Falcon (FN-DSA) signature verification with AVX-512. On a modern AMD Zen5 core, it completes a Falcon-512 verification in 3.6μs, 2.6× faster than an already optimized baseline, with comparable gains on Zen4, and consistent results across clang 21 and gcc 15.
The speedup comes from rewriting the Number-Theoretic Transform (NTT) and from vectorising all other stages of the verification algorithm. The novelty is to use a 32-bit Barrett-style representation, instead of the reference 16-bit Montgomery, and adopt Shoup–Harvey precomputed multipliers for twiddle reduction.
With all optimizations applied, hash-to-point (and specifically Keccak) is the dominant cost. We therefore propose a non-standard Falcon variant that replaces `SHAKE256` with `KTP256`, an XOF based on KangarooTwelve with parallel squeeze. It cuts verification to 2.2μs on Zen5, yielding 4.2× over the baseline, and is of independent interest for any post-quantum scheme that uses a Keccak sponge to sample large amounts of data from a fixed seed. All code is open source.

Keywords: Post Quantum Cryptography, Falcon, FN-DSA, AVX-512

1 Introduction

Falcon [8] is a lattice-based digital signature algorithm selected by NIST as a standardized post-quantum cryptography algorithm. Known officially as FN-DSA [12], it provides two security levels: Falcon-512 (128-bit) and Falcon-1024 (256-bit).

Falcon is often considered hard to implement because its signing operation relies on floating-point Gaussian sampling, which is difficult to secure against side-channel attacks. Nevertheless, its small public keys and signatures make it the current top choice for bandwidth-sensitive applications like IoT, secure networking, and blockchains. We are specifically interested in efficient signature verification.

Related Work

The reference Falcon implementation is part of the NIST Round 3 submission [8]. Subsequent Falcon optimization work falls into the following categories.

Sign and key generation (software). Most reported Falcon implementation efforts target signing and key generation, which dominate the runtime cost of the algorithm. Zhang and Zhang [18] vectorise the signature generation pipeline (with a focus on the `BaseSampler` for discrete Gaussian sampling) across SSE2, AVX2, AVX-512F, NEON, and RVV. Pornin [16] reports further constant-time floating-point-free improvements to the key generation routines shared by Falcon, BAT, and Hawk.

Microcontrollers. Falcon on the ARM Cortex-M family has been extensively studied. The original constant-time Cortex-M4 port is due to Pornin [14]; a 2025 update [15] approximately halved the runtime cost. Becker et al. [1] target Cortex-A72 and Apple M1 with NEON optimizations of the NTT applicable to Dilithium, Kyber, and Saber. Kannwischer et al. [10] benchmark the NIST Round 1 additional post-quantum signature candidates inside the `pqm4` framework on Cortex-M4.

Hardware accelerators (FPGA, GPU). Closest to our work, Coulon, Xiong, and Xie [5] present `LEAF`, a lightweight FPGA accelerator dedicated to Falcon signature verification. On FPGA, Pendyala et al. [13] present a throughput-optimized `SamplerZ` accelerator for the Gaussian sampler used in signing. Coulon, Bao, and Xie [4] accelerate the large-integer XGCD step of key generation. On GPU, Li et al. [11] (`cuFalcon`) and Dai et al. [6] (`GOLF`) present full implementations targeting batched throughput.

Contribution

In this work, we focus on Falcon signature verification and present the fastest implementation targeting x86-64 with AVX-512 extensions. We work with Falcon-512 because it is the harder target to optimize: its smaller polynomials leave less work over which to amortize SIMD setup and twiddle loads, and the techniques carry over directly to Falcon-1024, which has more parallelism to spare.

In Section 3 we rewrite the Number-Theoretic Transform (NTT), which dominates verification. We widen $\mathbb{F}_q$ elements from the reference 16-bit Montgomery representation to 32-bit operands, we vectorise the NTT passes with adaptive steps, and we adopt the Shoup–Harvey multiplier [9] and a lazily reduced butterfly. We then vectorise the remaining stages of rejection sampling and decoding.

In Section 4 we evaluate on AMD Zen4 and Zen5 with both clang 21 and gcc 15, against a baseline derived from the reference implementation. Our verifier completes a Falcon-512 verification in 3.6μs on Zen5, 2.6× over the baseline, with comparable gains on Zen4, and we attribute the gains to their individual algorithmic components. With the NTT no longer the bottleneck, hash-to-point, and specifically Keccak, emerges as the dominant cost.

In Section 5 we propose a non-standard variant of Falcon that replaces `SHAKE256` with `KTP256`, a KangarooTwelve-style XOF with parallel squeeze, attacking the Keccak bottleneck directly: 12 permutation rounds instead of 24, and 8-way parallel squeezing on AVX-512. This brings verification down to 2.2μs on Zen5, 4.2× over the baseline. The construction is of broader interest, as most post-quantum schemes rely on Keccak-based XOFs and the choice of `SHAKE256` with 24 rounds and sequential squeezing is purely historical.

This work was developed as part of the Firedancer project [7], and all our code is open source and available at https://github.com/jumpcrypto/falcon.

2 Background

2.1 Notation

We follow Falcon's notation from the Round 3 NIST submission [8]. Falcon operates over the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with prime modulus $q = 12289$ and degree n, with $n = 512$ for Falcon-512 and $n = 1024$ for Falcon-1024.

The number-theoretic transform is well defined on R_q. We write NTT and NTT^{-1} for the forward and inverse transformations respectively. The product of two polynomials

$a, b \in R_q$ is computed as:

$$a \cdot b = \mathrm{NTT}^{-1}\big(\mathrm{NTT}(a) \otimes \mathrm{NTT}(b)\big) \ ,$$

where $a \otimes b$ is a coefficient-wise multiplication, or Hadamard product.

2.2 Signature Verification

Signature verification is described in Algorithm 1. It takes as input a public key pk, a message msg, and a signature $sig = (r, s)$, where r is a salt and s is the compressed signature polynomial, and produces as output either *valid* or *invalid*. The verifier proceeds in four stages:

1. *Hash-to-point*: compute $c \in R_q$ from $\texttt{SHAKE256}(r \,\|\, msg)$ with rejection sampling, i.e., extracting one coefficient at a time and rejecting those greater than q.
2. *Decode*: decode pk to recover $h \in R_q$, and decompress s to recover $s_2 \in R_q$. Coefficients of h use a fixed-size 14-bit encoding, while s encodes the coefficients of s_2 with a variable-size (compressed) encoding; an invalid encoding is rejected.
3. *Polynomial computation*: compute $s_1 = c - s_2 \cdot h$ in R_q via two forward NTTs, one Hadamard product, one NTT^{-1}, and one subtraction.
4. *Norm check*: accept if and only if $\|(s_1, s_2)\|^2 \leq \beta^2$, where β is a given acceptance bound.

Algorithm 1 Falcon verify (`Falcon-512`, $n = 512$, $q = 12289$, $\beta^2 = 34034726$).

Require: Public key pk, message msg, signature $sig = (r, s)$.
Ensure: *valid* or *invalid*.

$c \leftarrow \mathrm{HashToPoint}(r \,\|\, msg)$
$h \leftarrow \mathrm{Decode}(pk)$
$s_2 \leftarrow \mathrm{Decompress}(s)$
if $s_2 = \perp$ **then**
    return *invalid* ▷ reject invalid encoding
$s_1 \leftarrow c - \mathrm{NTT}^{-1}\big(\mathrm{NTT}(s_2) \otimes \mathrm{NTT}(h)\big)$ ▷ s_1 normalized in $(-q/2, q/2)$
if $\|(s_1,\, s_2)\|^2 \leq \beta^2$ **then**
    return *valid*
else
    return *invalid* ▷ reject invalid signature

3 Algorithmic Improvements

When the runtime of Falcon verify is broken down by stage, the ring multiplication dominates the computation. The rest is split between hash-to-point and decode. Our improvements follow this pattern: most of the speedup over the baseline comes from rewriting the NTT; the remainder comes from vectorising the auxiliary steps.

We make three sets of changes. In Section 3.1, we widen $\mathbb{F}_q$ elements from 16 to 32 bits, switch from Montgomery to Barrett-style reduction, and vectorise the NTT pass by pass, stepping the SIMD width down as the stride shrinks; together these account for most of the speedup. In Section 3.2 we adopt Shoup–Harvey [9] lazily reduced butterfly, and improve upon it by removing layers of conditional subtractions. In Section 3.3 we vectorise the remaining parts: hash-to-point rejection sampling, and decoding of public key and signature.

3.1 Field Arithmetic and SIMD NTT

We represent $\mathbb{F}_q$ elements as 32-bit integers so that we can lazily reduce the butterfly outputs (see Section 3.2 for the lazy reduction bound), and switch from the Montgomery representation used by the reference implementation to a Barrett-style reduction.

On top of this field representation, the NTT is specialized: we use 16 lanes while the stride $t \geq 16$, 8 lanes at $t = 8$, 4 lanes at $t = 4$, then two butterflies per `xmm` register at $t = 2$, and finally four butterflies per `xmm` register at $t = 1$. Algorithm 2 gives a high-level loop with this strategy, alongside the reference scalar NTT for comparison.

Algorithm 2 The reference scalar NTT (`ref`, left) vs. SIMD batched butterflies (`avx512`, right). The reference runs one scalar butterfly loop at every stride, while the SIMD path dispatches on t to the smallest register that fits.

`ref`

Require: $a[\,]$ `uint16`; twiddles ψ
Ensure: $a \leftarrow \texttt{NTT}(a)$, each $\in [0, q)$
$t \leftarrow n$; $m \leftarrow 1$
while $m < n$ **do**
  $t \leftarrow t/2$
  for $i = 0, \ldots, m{-}1$ **do**
    $s \leftarrow \psi_{m+i}$
    for $j = 2it, \ldots, 2it{+}t{-}1$ **do**
      $u \leftarrow a[j]$
      $v \leftarrow \textsc{MontMul}(a[j{+}t], s)$
      $a[j] \leftarrow \textsc{AddModQ}(u, v)$
      $a[j{+}t] \leftarrow \textsc{SubModQ}(u, v)$
  $m \leftarrow 2m$

`avx512`

Require: $a[\,]$ `uint32`; twiddles ψ
Ensure: $a \leftarrow \texttt{NTT}(a)$, each $\in [0, q)$
$t \leftarrow n$; $m \leftarrow 1$
while $m < n$ **do**
  $t \leftarrow t/2$
  if $t \geq 16$ **then** ▷ `zmm`: 16 lanes
    for $i = 0, \ldots, m-1$ **do**
      broadcast $\mathbf{s} \leftarrow \psi_{m+i}$
      for $j = 2it, \ldots, 2it + t - 1$ **step** 16 **do** ▷ $\textsc{Butterfly}_{16}$
        $\mathbf{u} \leftarrow a[j{:}j{+}16]$
        $\mathbf{v} \leftarrow \textsc{MulModQ}_{16}(a[j{+}t{:}j{+}t{+}16], \mathbf{s})$
        $a[j{:}j{+}16] \leftarrow \textsc{AddModQ}_{16}(\mathbf{u}, \mathbf{v})$
        $a[j{+}t{:}j{+}t{+}16] \leftarrow \textsc{SubModQ}_{16}(\mathbf{u}, \mathbf{v})$
  else if $t = 8$ **then** ▷ `ymm`: 8 lanes
    for $i = 0, \ldots, m-1$ **do**
      $\textsc{Butterfly}_8(2it,\ \psi_{m+i})$
  else if $t = 4$ **then** ▷ `xmm`: 4 lanes
    for $i = 0, \ldots, m-1$ **do**
      $\textsc{Butterfly}_4(2it,\ \psi_{m+i})$
  else if $t = 2$ **then** ▷ `xmm`, ×2 butterflies/reg
    for $i = 0, \ldots, m-1$ **step** 2 **do**
      $\textsc{Butterfly}^{\times 2}(2it,\ \psi_{m+i\,..\,m+i+1})$
  else ▷ t=1, `xmm`, ×4 butterflies/reg
    for $i = 0, \ldots, m-1$ **step** 4 **do**
      $\textsc{Butterfly}^{\times 4}(2it,\ \psi_{m+i\,..\,m+i+3})$
  $m \leftarrow 2m$
normalize each lane to $[0, q)$

3.2 Lazy Reduction and Shoup–Harvey

The butterfly requires one multiplication, addition and subtraction in $\mathbb{F}_q$. We follow [9] for an efficient multiplication, as well as for unreduced addition and subtraction.

However, we take further advantage of the 32-bit operands to remove even more reductions.

First, in the multiplication, [9] reduces the result to $[0, q)$ via a final conditional subtraction. We remove it, leaving the result in $[0, 2q)$. A product vs is computed as in [9]:

$$\hat{q} = \left\lfloor v \cdot s' / 2^{32} \right\rfloor$$
$$vs = v \cdot s - \hat{q} \cdot q$$

where $s' = \left\lfloor s \cdot 2^{32} / q \right\rfloor$ is precomputed, and $\cdot$ is a regular 32-bit multiplication. Algorithm 3 details the 16-lane multiplier using AVX-512 instructions, reduced in $[0, 2q)$. The algorithm is only 10 instructions long, and it highlights that 2-3 additional instructions for a conditional subtraction would make it 20–30% slower.

We can now analyze the impact of the multiplier on the NTT. The forward butterfly

becomes:

$$u' = u + vs$$
$$v' = u - vs + 2q$$

If we start with $u, v \in [0, q)$, after k lazily reduced passes, every element satisfies $0 \le x < (2k+1)q$. For $n = 512$, the bound after the $\log_2 n = 9$ passes is $19q < 2^{18}$, well within the range of a 32-bit integer.

Next, we note that the same "final conditional subtraction" also appears in other places: to reduce the final result of the NTT, and to reduce the Hadamard product. In our final implementation, we removed all these conditional subtractions, making the NTT result in $[0, 2q)$, and defining the Hadamard product as a map: $[0, 2q)^n \times [0, 2q)^n \to [0, 2q)^n$.

Putting all pieces together, the forward NTT is described in Algorithm 4. It uses Shoup–Harvey (Algorithm 3) as its inner multiplier and the lazy reductions described above. The inverse NTT is structurally similar and detailed in Appendix A, Algorithm 7.

Algorithm 3 Shoup–Harvey 16-lane lazy field multiplication.

Require: $\mathbf{v} \in [0, 2^{32})^{16}$, broadcasts $\mathbf{s}$, $\mathbf{s}' \in [0, q)^{16}$
Ensure: $\mathbf{r} \equiv \mathbf{v} \cdot s \pmod q$, with $\mathbf{r} \in [0, 2q)^{16}$

1: $\mathbf{w}_e \leftarrow$ `vpmuludq`$(\mathbf{v}, \mathbf{s}')$ ▷ $v \cdot s'$, even lanes
2: $\mathbf{v}_o \leftarrow$ `vpsrlq`$(\mathbf{v}, 32)$; $\mathbf{s}'_o \leftarrow$ `vpsrlq`$(\mathbf{s}', 32)$
3: $\mathbf{w}_o \leftarrow$ `vpmuludq`$(\mathbf{v}_o, \mathbf{s}'_o)$ ▷ $v \cdot s'$, odd lanes
4: $\hat{\mathbf{q}}_e \leftarrow$ `vpsrlq`$(\mathbf{w}_e, 32)$; $\hat{\mathbf{q}}_o \leftarrow$ `vpandnd`$(\texttt{0xFFFFFFFF}, \mathbf{w}_o)$
5: $\hat{\mathbf{q}} \leftarrow$ `vpord`$(\hat{\mathbf{q}}_e, \hat{\mathbf{q}}_o)$ ▷ $\hat{q} = \lfloor v \cdot s'/2^{32} \rfloor$
6: **return** `vpsubd`(`vpmulld`$(\mathbf{v}, \mathbf{s})$, `vpmulld`$(\hat{\mathbf{q}}, q)$) ▷ $v \cdot s - \hat{q} \cdot q \in [0, 2q)$

Algorithm 4 SIMD Forward NTT using 32-bit elements, per-stride dispatch, Shoup–Harvey multiplication, and lazy reductions.

Require: $a[\,]$ `uint32`; twiddles ψ, Shoup multipliers s'
Ensure: $a \leftarrow$ `NTT`(a), each $\in [0, 2q)$

$t \leftarrow n$; $m \leftarrow 1$
while $m < n$ **do**
  $t \leftarrow t/2$
  if $t \ge 16$ **then**
    for $i = 0, \ldots, m-1$ **do**
      broadcast $\mathbf{s} \leftarrow \psi_{m+i}$, $\mathbf{s}' \leftarrow s'_{m+i}$
      for $j = 2it, \ldots, 2it + t - 1$ **step** 16 **do**
        $\mathbf{u} \leftarrow a[j{:}j{+}16]$
        $\mathbf{v} \leftarrow \textsc{ShoupHarveyMul}_{16}(a[j{+}t{:}j{+}t{+}16], \mathbf{s}, \mathbf{s}')$ ▷ Algorithm 3
        $a[j{:}j{+}16] \leftarrow \mathbf{u} + \mathbf{v}$ ▷ unreduced
        $a[j{+}t{:}j{+}t{+}16] \leftarrow (\mathbf{u} - \mathbf{v}) + 2q$ ▷ unreduced
  else
    ...narrower widths, as in Algorithm 2...
reduce each lane from $[0, 19q)$ to $[0, 2q)$ ▷ Barrett, $M = \lfloor 2^{27}/q \rfloor$, no cond. sub

3.3 Vectorising Hash-to-Point and Decode

The remaining stages of the verifier are hash-to-point, public-key decoding, and signature decompression. Algorithms are in Appendix B and C. Individually, these are smaller wins than the NTT improvements we have described, but overall they are all necessary to obtain a performant SIMD implementation.

- Hash-to-point processes 16 candidates per iteration by using `vpcmpge` to reject elements above the bound and `vpcompressd` to shuffle valid candidates into a contiguous array.
- Public-key parsing handles 16 elements (28 packed bytes) per iteration via overlapping 32-bit loads and `vpcmpge` to reject invalid coefficients (Appendix C.1).
- Signature decompression uses a 64-bit window to amortize loading variable-length elements and recovers the unary high bits using `lzcnt` (Appendix C.2).

Within hash-to-point, the underlying Keccak permutation is sequential and untouched; producing the ∼1094 bytes that rejection sampling needs takes ∼9-10 permutations regardless of how fast the sampling loop runs.

As we will see in the next section, when everything is optimized, hash-to-point emerges as the slowest phase of the verification pipeline, which motivates the non-standard `KTP256` construction of Section 5.

4 Evaluation

4.1 Methodology

We benchmarked the implementations on two machines:

- **Zen4** - AMD EPYC 9754 at 3.0 GHz
- **Zen5** - AMD EPYC 9575F at 5.0 GHz

Each implementation is benchmarked with clang 21.1.8 and with gcc 15.2.1, both with options `-O3 -march=native`.

When benchmarking we compare all implementations against a *baseline*, which consists of the reference Falcon implementation [8] with the `SHAKE256` replaced by the XKCP library [2]. We did this because the Falcon reference ships with its own `SHAKE256` that is considerably slower than XKCP, and using it as a baseline would have unfairly amplified our results.

All results are easy to reproduce by compiling and running the `bench` tool available at https://github.com/jumpcrypto/falcon.

4.2 Experimental Results

Table 1 shows the results for six verifier variants:

- `ref`: Falcon Round 3 reference implementation from [8].
- `ref_xkcp`: Baseline implementation using `ref` code and replacing `SHAKE256` with the XKCP library.
- `avx512_mont`: Port of the reference implementation to AVX-512, using Montgomery representation.
- `x86`: Our implementation using Shoup–Harvey representation, without explicit AVX-512 code. Note that this is still compiled with AVX-512 instructions, auto-vectorised by the compiler.
- `avx512_barrett`: Our implementation with Barrett representation.
- `avx512`: Our final implementation with Shoup–Harvey representation.

All the implementations except `ref` use the XKCP library for `SHAKE256`.

The first remark is that the timings on Zen5 are ∼2× faster than those of Zen4, explained by the improved instruction latency and higher frequency. As mentioned, `ref` has a custom implementation of `SHAKE256` that makes it 10–20% slower than the baseline; it is included for completeness.

Our main result, `avx512`, achieves the best performance with clang: 3.6μs on Zen5 (2.6× over the baseline) and 6.8μs on Zen4 (2.5× over the baseline).

Table 1: Falcon-512 verification, ns/call and speedup over the `ref_xkcp` baseline.

	Zen4				Zen5			
	clang 21		gcc 15		clang 21		gcc 15	
Implementation	**ns**	×	**ns**	×	**ns**	×	**ns**	×
`ref`	18 791	0.9	23 950	0.8	9 901	0.9	12 717	0.9
`ref_xkcp` (baseline)	17 172	1.0	19 343	1.0	9 357	1.0	11 041	1.0
`avx512_mont`	17 113	1.0	13 420	1.4	9 408	1.0	7 293	1.5
`x86`	9 917	1.7	9 256	2.1	5 380	1.7	4 387	2.5
`avx512_barrett`	7 353	2.3	8 403	2.3	3 923	2.4	4 530	2.4
`avx512`	**6760**	**2.5**	**7554**	**2.6**	**3580**	**2.6**	**3994**	**2.8**

The remaining implementations are included to show a progression of results and to highlight that the wider Shoup–Harvey representation is more SIMD-friendly than the 16-bit Montgomery.

Since computing the product involves two independent NTTs, a natural question is whether they can be parallelized at the register level. Our experiments in this area were unsuccessful: gains at smaller strides were cancelled by worse memory access patterns, and the code grew more complex; we therefore abandoned this route.

5 Non-Standard Falcon

When the timing of Falcon verification is broken down by stage, the majority of the runtime is spent in hash-to-point—and specifically in Keccak. For example, on Zen5 with `avx512`, hash-to-point alone accounts for 57% of the verification cost (Table 3, row 3, column *h2p*). This may seem surprising as public-key cryptography is generally an order of magnitude slower than symmetric primitives such as hashing. Yet on platforms with high SIMD parallelism, that gap has closed for lattice-based schemes. We therefore propose a change that speeds up the hashing stage.

In Falcon's hash-to-point primitive, we replace `SHAKE256` with `KTP256`, a new XOF construction analogous to KangarooTwelve [17], but with parallel squeeze. The change has two effects: the underlying Keccak permutation runs at 12 rounds (the round count of KangarooTwelve and TurboSHAKE) instead of `SHAKE256`'s 24, and the squeeze step parallelizes; on AVX-512 we issue 8 permutations at once via XKCP's `KeccakP1600times8`.

The change requires a non-standard hash, but we believe it is significant beyond Falcon, since the same construction applies to any post-quantum scheme that relies on a Keccak XOF.

5.1 `KTP256`: KangarooTwelve with Parallel Squeeze

Following RFC 9861 [17], `KTP256` is defined in two layers: `TurboSHAKEPar256` (Algorithm 5) and `KTP256` (Algorithm 6). The former is `TurboSHAKE256`, where the squeeze phase is replaced by a CBC-like construction that can be parallelized. The latter is simply `KT256` where the two `TurboSHAKE256` calls are replaced by `TurboSHAKEPar256`.

Security reduces to the security of the underlying Keccak permutation, as detailed in the Farfalle framework [3].

Notation in the pseudocode follows RFC 9861 (Appendices A.3 and A.5). Versus the RFC, Algorithm 5 differs only on lines 10–15 (the squeeze phase), and Algorithm 6 differs only on lines 3, 13.

Algorithm 5 `TurboSHAKEPar256`: parallel-squeeze `TurboSHAKE256`. Identical to `TurboSHAKE256` (RFC 9861, Appendix A.3) except for the squeeze phase (lines 10–15), that can be parallelized.

Require: Message M, separation byte D, output length L.
Ensure: L-byte parallel-squeeze output.

1: $offset \leftarrow 0$; $state \leftarrow \texttt{00}^{200}$
2: $input \leftarrow M \,\|\, D$
3: **while** $offset < |input| - 136$ **do**
4: $\quad state \oplus= input[offset{:}offset{+}136] \,\|\, \texttt{00}^{64}$
5: $\quad state \leftarrow \mathrm{KP}(state)$ ▷ KP = Keccak-p[1600, 12]
6: $\quad offset \mathrel{+}= 136$
7: $LastBlockLength \leftarrow |input| - offset$
8: $state \oplus= input[offset{:}] \,\|\, \texttt{00}^{200-LastBlockLength}$
9: $state \oplus= \texttt{00}^{135} \,\|\, \texttt{80} \,\|\, \texttt{00}^{64}$
▷ Custom parallel squeeze replaces the standard squeeze
10: $K \leftarrow \lceil L/136 \rceil$
11: **for** $i = 0, \ldots, K-1$ **do**
12: $\quad state_i \leftarrow state \oplus \mathrm{lane}(i)$
13: $\quad state_i \leftarrow \mathrm{KP}(state_i)$
14: $\quad \mathrm{out}_i \leftarrow state_i[0{:}136]$
15: **return** $(\mathrm{out}_0 \,\|\, \mathrm{out}_1 \,\|\, \cdots \,\|\, \mathrm{out}_{K-1})[0{:}L]$

Each iteration in the final loop over K lanes (lines 10–15) is independent of the others, and the squeeze can be done in parallel, using an API like XKCP's `KeccakP1600times8`.

Algorithm 6 `KTP256`: XOF with parallel squeeze. Identical to `KT256` [17] (RFC 9861, Appendix A.5) except for two lines that swap the top-level `TurboSHAKE256` calls for `TurboSHAKEPar256` (Algorithm 5).

Require: Message M, customization string C, output length L.
Ensure: L-byte XOF output on input (M, C).

$S \leftarrow M \,\|\, C \,\|\, \texttt{length_encode}(|C|)$
if $|S| \leq 8192$ **then**
$\quad$ **return** $\texttt{TurboSHAKEPar256}(S,\ \texttt{07},\ L)$ ▷ in place of `TurboSHAKE256`
else
$\quad FinalNode \leftarrow S[0{:}8192] \,\|\, \texttt{03} \,\|\, \texttt{00}^{7}$
$\quad offset \leftarrow 8192$; $numBlock \leftarrow 0$
$\quad$ **while** $offset < |S|$ **do**
$\quad\quad b \leftarrow \min\big(|S| - offset,\ 8192\big)$
$\quad\quad CV \leftarrow \texttt{TurboSHAKE256}\big(S[offset{:}offset{+}b],\ \texttt{0B},\ 64\big)$
$\quad\quad FinalNode \leftarrow FinalNode \,\|\, CV$
$\quad\quad numBlock \mathrel{+}= 1$; $offset \mathrel{+}= b$
$\quad FinalNode \leftarrow FinalNode \,\|\, \texttt{length_encode}(numBlock) \,\|\, \texttt{FF FF}$
$\quad$ **return** $\texttt{TurboSHAKEPar256}(FinalNode,\ \texttt{06},\ L)$ ▷ in place of `TurboSHAKE256`

5.2 Experimental Results

Table 2 compares the speed of four variants of Falcon verify (`ref` vs. `avx512`, and `SHAKE256` vs. `KTP256`), in the same benchmarking environment as Table 1. The `KTP256` change alone buys ∼1.2× uniformly across compilers and machines (row `ref_ktp256`). Combined with

our AVX-512 improvements, `avx512_ktp256` reaches 4.4μs on Zen4 (3.9× over the baseline) and 2.2μs on Zen5 (4.2×).

Table 2: Comparison of standard vs. `KTP256`-based Falcon-512 verification, ns/call and per-column speedup over the baseline.

	Zen4				**Zen5**			
	clang 21		**gcc 15**		**clang 21**		**gcc 15**	
Implementation	**ns**	×	**ns**	×	**ns**	×	**ns**	×
`ref_xkcp` (baseline)	17 172	1.0	19 343	1.0	9 357	1.0	11 041	1.0
`ref_ktp256`	14 196	1.2	16 933	1.1	7 233	1.3	9 009	1.2
`avx512`	6 760	2.5	7 554	2.6	3 580	2.6	3 994	2.8
`avx512_ktp256`	**4 366**	**3.9**	**4 713**	**4.1**	**2 245**	**4.2**	**2 316**	**4.8**

Finally, Table 3 takes the timing of Zen5 + clang and breaks it into the main phases: hash-to-point (*h2p*), decoding (*dec*), and ring multiplication (*mul*). *mul* dominates the baseline; vectorising the NTT (`avx512`) flips the dominant component to *h2p*; applying `KTP256` (`avx512_ktp256`) flips it back to *mul*.

Table 3: Breakdown of Falcon-512 verification on Zen5 with clang, in ns/call and as a percentage of the per-row total. In bold is the column with the highest percentage.

Variant	**h2p**		**dec**		**mul**		**total**	
	ns	**%**	**ns**	**%**	**ns**	**%**	**ns**	×
`ref_xkcp` (baseline)	2831	30%	1051	11%	**5475**	**59%**	9357	1.0×
`ref_ktp256`	706	10%	1051	14%	**5475**	**76%**	7233	1.3×
`avx512`	**2041**	**57%**	723	20%	816	23%	3580	2.6×
`avx512_ktp256`	706	31%	723	32%	**816**	**36%**	2245	4.2×

6 Conclusion

We presented a fast AVX-512 implementation of Falcon-512 signature verification. Most of the speedup comes from rewriting the number-theoretic transform. We first widen $\mathbb{F}_q$ elements to SIMD-friendly 32-bit lanes and vectorise butterfly passes. We then defer modular reductions lazily across passes and reduce with Shoup–Harvey precomputed multipliers. Finally, we vectorise the remaining stages: hash-to-point, public-key and signature decoding. On clang, these changes bring verification to 3.6μs on Zen5, and 6.8μs on Zen4, ∼2.5× over the baseline.

After these optimizations, most of the verification time is spent hashing. We therefore introduce `KTP256`, a new XOF based on KangarooTwelve with parallel squeeze, and use it in place of `SHAKE256` inside hash-to-point. This lowers verification to 2.2μs on Zen5 (4.2× over the baseline), and 4.4μs on Zen4 (3.9× over the baseline).

The construction is of broader interest: most post-quantum schemes derive vectors, matrices, or other long streams of data by iteratively hashing a short seed. Adopting a construction like `KTP256` in place of the historical `SHAKE256` (with 24 rounds and sequential squeezing) would benefit the broader post-quantum ecosystem.

References

[1] Becker, H., Hwang, V., Kannwischer, M.J., Yang, B.Y., Yang, S.Y.: Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(1), 221–244 (2022). https://doi.org/10.46586/tches.v2022.i1.221-244

[2] Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., Van Keer, R.: The eXtended Keccak Code Package (XKCP), https://github.com/XKCP/XKCP

[3] Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., Van Keer, R.: Farfalle: Parallel permutation-based cryptography. IACR Transactions on Symmetric Cryptology **2017**(4), 1–38 (2017). https://doi.org/10.13154/tosc.v2017.i4.1-38

[4] Coulon, S., Bao, T., Xie, J.: FELIX (XGCD for FALCON): FPGA-based scalable and lightweight accelerator for large integer extended GCD. IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2024). https://doi.org/10.1109/TVLSI.2024.3417016

[5] Coulon, S., Xiong, J., Xie, J.: LEAF: Lightweight and efficient hardware accelerator for signature verification of FALCON. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2025) (2025). https://doi.org/10.1109/ICCAD66269.2025.11240686

[6] Dai, R., Dong, J., Qiu, M., Dong, Z., Xiao, F., Lin, J.: GOLF: Unleashing GPU-driven acceleration for FALCON post-quantum cryptography. Cryptology ePrint Archive, Report 2025/749 (2025), https://eprint.iacr.org/2025/749

[7] Firedancer Team: Firedancer: a high-performance Solana validator client, https://github.com/firedancer-io/firedancer

[8] Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier lattice-based compact signatures over NTRU. NIST PQC Submission, Round 3 (2020), https://falcon-sign.info/

[9] Harvey, D.: Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation **60**, 113–119 (2014). https://doi.org/10.1016/j.jsc.2013.09.002

[10] Kannwischer, M.J., Krausz, M., Petri, R., Yang, S.Y.: pqm4: Benchmarking NIST additional post-quantum signature schemes on microcontrollers. Cryptology ePrint Archive, Report 2024/112 (2024), https://eprint.iacr.org/2024/112

[11] Li, W., Wei, H., Shen, S., Yang, H., Dai, W., Zhao, Y.: cuFalcon: An adaptive parallel GPU implementation for high-performance Falcon acceleration. Cryptology ePrint Archive, Report 2025/249 (2025), https://eprint.iacr.org/2025/249

[12] NIST: FIPS 206, Fast-Fourier-Transform-based Digital Signature Algorithm (FN-DSA) (2026), https://nist.gov, in development.

[13] Pendyala, S., Magesh, R., Kavun, E.B., Aysu, A.: Accelerating FALCON: Speed records for FALCON's SamplerZ on Xilinx FPGAs. Cryptology ePrint Archive, Report 2025/1490 (2025), https://eprint.iacr.org/2025/1490

[14] Pornin, T.: New efficient, constant-time implementations of Falcon. Cryptology ePrint Archive, Report 2019/893 (2019), https://eprint.iacr.org/2019/893

[15] Pornin, T.: Falcon on ARM Cortex-M4: an update. Cryptology ePrint Archive, Report 2025/123 (2025), https://eprint.iacr.org/2025/123

[16] Pornin, T.: Improved (again) key pair generation for Falcon, BAT and Hawk. Cryptology ePrint Archive, Report 2025/1239 (2025), https://eprint.iacr.org/2025/1239

[17] Viguier, B., Wong, D., Van Assche, G., Dang, Q., Daemen, J.: TurboSHAKE and KangarooTwelve. RFC 9861 (2025). https://doi.org/10.17487/RFC9861

[18] Zhang, J., Zhang, J.: Vectorized Falcon-Sign implementations using SSE2, AVX2, AVX-512F, NEON, and RVV. IACR Transactions on Cryptographic Hardware and Embedded Systems (2026). https://doi.org/10.46586/tches.v2026.i1.302-324

A Inverse NTT

The inverse NTT (Algorithm 7) reuses the same ideas as the forward NTT: 32-bit operands, per-stride dispatch, Shoup–Harvey for every multiplication, and no reduction inside the passes. The differences between forward and inverse NTT are the obvious order of iteration, and the final pass where we fold the result in $[0, q)$.

Algorithm 7 The reference scalar NTT^{-1} (`ref`, left) vs. SIMD batched butterflies (`avx512`, right). The SIMD path dispatches on t to the smallest register that fits, as in Algorithm 2, and multiplies through Shoup–Harvey, as in Algorithm 4.

`ref`

Require: $a[\,]$ `uint16`; twiddles ψ^{-1}, Shoup multipliers s'
Ensure: $a \leftarrow \mathrm{NTT}^{-1}(a)$, each $\in [0, q)$
$t \leftarrow 1$; $m \leftarrow n$
while $m > 1$ **do**
  $h \leftarrow m/2$
  for $i = 0, \ldots, h{-}1$ **do**
    $s \leftarrow \psi^{-1}_{h+i}$
    for $j = 2it, \ldots, 2it{+}t{-}1$ **do**
      $u \leftarrow a[j]$; $v \leftarrow a[j{+}t]$
      $a[j] \leftarrow \textsc{AddModQ}(u, v)$
      $w \leftarrow \textsc{SubModQ}(u, v)$
      $a[j{+}t] \leftarrow \textsc{MontMul}(w, s)$
  $t \leftarrow 2t$; $m \leftarrow m/2$
multiply every $a[j]$ by N^{-1}

`avx512`

Require: $a[\,]$ `uint32`; twiddles ψ^{-1}, Shoup multipliers s'^{-1}
Ensure: $a \leftarrow \mathrm{NTT}^{-1}(a)$, each $\in [0, q)$
$t \leftarrow 1$; $m \leftarrow n$; $\mathit{off}_q \leftarrow 2$ ▷ inputs are lazy in $[0, 2q)$
while $m > 2$ **do**
  $h \leftarrow m/2$
  if $t \geq 16$ **then** ▷ `zmm`: 16 lanes
    for $i = 0, \ldots, h-1$ **do**
      broadcast $\mathbf{s} \leftarrow \psi^{-1}_{h+i}$, $\mathbf{s}' \leftarrow s'^{-1}_{h+i}$
      for $j = 2it, \ldots, 2it + t - 1$ **step** 16 **do**
        $\mathbf{u} \leftarrow a[j{:}j{+}16]$; $\mathbf{v} \leftarrow a[j{+}t{:}j{+}t{+}16]$
        $a[j{:}j{+}16] \leftarrow \mathbf{u} + \mathbf{v}$ ▷ unreduced
        $\mathbf{w} \leftarrow \mathbf{u} - \mathbf{v} + \mathit{off}$ ▷ $\mathit{off} = \mathit{off}_q \cdot q$ keeps $\mathbf{w} \geq 0$
        $a[j{+}t{:}j{+}t{+}16] \leftarrow \textsc{ShoupHarveyMul}_{16}(\mathbf{w}, \mathbf{s}, \mathbf{s}')$
  else if $t = 8$ **then** ▷ `ymm`: 8 lanes
    for $i = 0, \ldots, h-1$ **do**
      $\textsc{Butterfly}_8(2it,\ \psi^{-1}_{h+i})$
  else if $t = 4$ **then** ▷ `xmm`: 4 lanes
    for $i = 0, \ldots, h-1$ **do**
      $\textsc{Butterfly}_4(2it,\ \psi^{-1}_{h+i})$
  else if $t = 2$ **then** ▷ `xmm`, ×2 butterflies/reg
    for $i = 0, \ldots, h-1$ **step** 2 **do**
      $\textsc{Butterfly}^{\times 2}(2it,\ \psi^{-1}_{h+i\,..\,h+i+1})$
  else ▷ t=1, `xmm`, ×4 butterflies/reg
    for $i = 0, \ldots, h-1$ **step** 4 **do**
      $\textsc{Butterfly}^{\times 4}(2it,\ \psi^{-1}_{h+i\,..\,h+i+3})$
  $\mathit{off}_q \leftarrow 2\,\mathit{off}_q$; $t \leftarrow 2t$; $m \leftarrow m/2$
    ▷ last pass (m=2, t=n/2): N^{-1} fused, then fold
broadcast $\mathbf{s}_0 \leftarrow N^{-1}$, $\mathbf{s}_1 \leftarrow \psi_1^{-1} N^{-1}$
broadcast $\mathbf{s}'_0$, $\mathbf{s}'_1$ ▷ their Shoup multipliers
for $j = 0, \ldots, n/2 - 1$ **step** 16 **do**
  $\mathbf{u} \leftarrow a[j{:}j{+}16]$; $\mathbf{v} \leftarrow a[j{+}n/2{:}j{+}n/2{+}16]$
  $a[j{:}j{+}16] \leftarrow \textsc{Fold}_{16}\big(\textsc{ShoupHarveyMul}_{16}(\mathbf{u} + \mathbf{v}, \mathbf{s}_0, \mathbf{s}'_0)\big)$
  $\mathbf{w} \leftarrow \mathbf{u} - \mathbf{v} + \mathit{off}$; $k = j{+}n/2$
  $a[k{:}k{+}16] \leftarrow \textsc{Fold}_{16}\big(\textsc{ShoupHarveyMul}_{16}(\mathbf{w}, \mathbf{s}_1, \mathbf{s}'_1)\big)$

B Hash-to-Point

Hash-to-point produces n uniformly distributed elements of $\mathbb{F}_q$ from $\texttt{SHAKE256}(r \,\|\, msg)$ via rejection sampling. The `SHAKE256` state is squeezed in 128-byte chunks; each pair of bytes is read as a big-endian 16-bit candidate x; if $x < K \cdot q$, where $K = \lfloor 2^{16}/q \rfloor = 5$, the candidate is accepted and then reduced modulo q. We process 16 candidates (32 bytes of squeeze) per iteration, as shown in Algorithm 8.

1. Load 16 big-endian 16-bit integers into a `ymm` register, byte swap each lane in place, and zero-extend to a 16-lane `zmm` register.
2. Compare all 16 lanes against $K \cdot q$ to produce a 16-bit acceptance mask.
3. Pack the accepted elements to the front of the register with `vpcompressd`, tail values outside the mask are left undefined.
4. Perform a Barrett reduction modulo q on all 16 lanes. Compute $\hat{q} = (x \cdot K) \gg 16$ and $r = x - \hat{q} \cdot q$, with one conditional subtraction. A single Barrett round suffices because $K \cdot q \leq 2^{16}$ and $K \cdot q \cdot \left(\frac{1}{q} - \frac{K}{2^{16}}\right) \approx 0.312 < 1$, so $\hat{q} \in \{0, \ldots, K-1\}$.
5. Write all 16 lanes at the current output position and advance the position by `popcnt`(mask). Lanes beyond the acceptance mask are harmlessly stored and will be overwritten by later iterations. The output buffer must be over-allocated by 15 elements, as the final iteration may accept more elements than remain to be sampled.

The acceptance rate is $(K \cdot q)/2^{16} \approx 0.937$, so filling $n = 512$ outputs requires $\lceil 512/0.937 \rceil \approx 547$ candidates on average, consuming roughly 1094 bytes of `SHAKE256` output. Although the sponge rate is 136 bytes, we squeeze 128 bytes per call, a power-of-two chunk that aligns with the SIMD width, requiring approximately 9 squeezes on average. In standard Falcon, these squeezes are sequential. In our non-standard variant we compute 8 squeezes in parallel, using the AVX-512 implementation of Keccak available in the XKCP package.

C Decode

C.1 Public Key

The public key packs n elements as 14-bit integers, so every 7 input bytes contains 4 elements. Our AVX-512 implementation processes 16 elements (28 bytes) per iteration as shown in Algorithm 9, At a high level:

1. Load eight 32-bit words at byte offsets $0, 3, 7, 10, 14, 17, 21, 24$ and gather them into a 16-lane `zmm`, with each word duplicated into an adjacent lane. The overlapping loads are necessary because a 14-bit element may span up to three bytes; the two copies let neighbouring elements claim disjoint bit windows of the same source word.
2. A `vpshufb` byte-reverses each 32-bit lane so that the bit fields are oriented left-to-right within the lane.
3. A per-lane variable right shift (`vpsrlvd`) by the constant vector

$$\langle 18, 4, 14, 0,\ 18, 4, 14, 0,\ \ldots \rangle$$

aligns each element to bit 0 of its lane.

4. A bitwise AND with $2^{14}-1$ masks off the remaining high bits. A final `vpcmpud` against q flags any out-of-range element and aborts the decoder.

Algorithm 8 AVX-512 hash-to-point: 16-lane rejection sampling with Barrett reduction.

Require: flipped `SHAKE256` state sc, output $\mathtt{c}[0 \dots n{+}14]$
Ensure: $\mathtt{c}[0 \dots n{-}1]$ uniform in $[0, q)$

$\mathtt{sample}[128]$; $\mathit{offset} \leftarrow 128$; $i \leftarrow 0$
while $i < n$ **do**
  if $\mathit{offset} \geq 128$ **then**
    $\mathtt{shake_extract}(sc,\ \mathtt{sample},\ 128)$
    $\mathit{offset} \leftarrow 0$
  $\mathbf{x} \leftarrow \mathtt{vmovdqu}_{16\times\mathrm{u16}}(\mathtt{sample}{+}\mathit{offset})$
  $\mathbf{x} \leftarrow \mathtt{vpord}(\mathtt{vpsrlw}(\mathbf{x}, 8), \mathtt{vpsllw}(\mathbf{x}, 8))$ ▷ big→little endian per u16
  $\mathbf{y} \leftarrow \mathtt{vpmovzxwd}(\mathbf{x})$ ▷ zero-extend to 16-lane `zmm`
  $k_1 \leftarrow \mathtt{vpcmpud}(\mathbf{y},\ K \cdot q,\ \mathrm{LT})$
  $\mathbf{y} \leftarrow \mathtt{vpcompressd}(\mathbf{y},\ k_1)$ ▷ Pack accepted lanes to the front
  ▷ Barrett reduction mod q across all 16 lanes
  $\hat{\mathbf{q}} \leftarrow \mathtt{vpsrld}(\mathtt{vpmulld}(\mathbf{y},\ K),\ 16)$
  $\mathbf{r} \leftarrow \mathtt{vpsubd}(\mathbf{y},\ \mathtt{vpmulld}(\hat{\mathbf{q}},\ q))$
  $k_2 \leftarrow \mathtt{vpcmpud}(\mathbf{r},\ q,\ \mathrm{GE})$
  $\mathbf{r} \leftarrow \mathtt{vpsubd}_{k_2}(\mathbf{r},\ q)$ ▷ A masked conditional subtraction
  $\mathtt{vmovdqu32}(\mathtt{c}{+}i,\ \mathbf{r})$
  $\mathit{offset}\ {+}{=}\ 32$
  $i\ {+}{=}\ \mathtt{popcnt}(k_1)$

Algorithm 9 AVX-512 public-key decoding (16 elements/iter).

Require: byte stream $\mathtt{in}[0 \dots 14n/8{-}1]$
Ensure: 32-bit array $\mathtt{out}[0 \dots n{-}1]$, each in $[0, q)$

shifts $\leftarrow \langle 18, 4, 14, 0,\ 18, 4, 14, 0,\ \dots\rangle_{16}$
for each 28-byte chunk at offset k **do**
  $\mathbf{z} \leftarrow \mathtt{vinserti32x4}\big(\mathtt{in}{+}k,\ \langle 0,0,3,3,7,7,10,10,14,14,17,17,21,21,24,24\rangle\big)$
  $\mathbf{z} \leftarrow \mathtt{vpshufb}(\mathbf{z},\ [3,2,1,0]_{\times 16})$ ▷ big→little endian per lane
  $\mathbf{z} \leftarrow \mathtt{vpsrlvd}(\mathbf{z},\ \mathbf{shifts})$ ▷ align each element to bit 0
  $\mathbf{z} \leftarrow \mathtt{vpandd}(\mathbf{z},\ 2^{14}{-}1)$ ▷ isolate 14-bit element
  $k_1 \leftarrow \mathtt{vpcmpud}(\mathbf{z},\ q,\ \mathrm{GE})$
  if $\mathtt{kortestw}(k_1) \neq 0$ **then reject**
  $\mathtt{vmovdqu32}(\mathtt{out},\ \mathbf{z})$

C.2 Signature

Each s_2 coefficient is encoded as [sign:1][low:7][0^k][1], a sign bit, the low 7 bits of the magnitude, k unary zeroes giving the high bits, and a 1 terminator. The reference implementation scans the bitstream one bit at a time. We keep a 64-bit window aligned to the next coefficient and refill it lazily, as shown in Algorithm 10.

1. Perform a 64-bit load at the current bit position followed by a `bswap`, so that the bits arrive in left-to-right order in the register.

2. The top byte gives sign and low: sign = word $\gg 63$, low = (word $\gg 56$) & 0x7F.

3. Shift the window by 8 bits and perform a `lzcnt`, counting the leading zeroes and recovering k.

4. Advance the bit pointer by $9 + k$ (sign + low + k unary zeroes + terminator) and refill the window when fewer than 16 bits remain.

A long unary run that crosses the 64-bit window boundary is rare (k rarely exceeds $\sim$10). When this occurs, a fallback loop keeps shifting in fresh 64-bit chunks and accumulating leading-zero counts. After the bitstream is fully consumed, the signed coefficients are converted into $\mathbb{F}_q$ representation in a separate pass.

Algorithm 10 Compressed signature decoder with window

Require: zero-padded byte stream `buf`, bit length L
Ensure: 32-bit array `out`$[0 \dots n{-}1]$, each in $[0, q)$

$w \leftarrow$ `bswap64(loadu64(buf))`;
$avail \leftarrow 64$
$b \leftarrow 0$
for $i = 0, \dots, n-1$ **do**
  if $avail < 16$ **then**
    $s \leftarrow b \bmod 8$
    $w \leftarrow$ `bswap64(loadu64(buf+`$\lfloor b/8 \rfloor$`))` $\ll s$
    $avail \leftarrow 64 - s$
  $sign \leftarrow w \gg 63$
  $low \leftarrow (w \gg 56)$ & `0x7F`
  $tail \leftarrow w \ll 8$
  if $tail \neq 0$ **then** ▷ common path: unary fits in window
    $k \leftarrow$ `lzcnt(`$tail$`)`;
    $a \leftarrow 9 + k$
    $w \ll= a$
    $avail -= a$
    $b += a$
  else ▷ rare: long unary run, refill loop
    $k \leftarrow avail - 8$
    $b += 8$
    loop
      refill w from `buf+`$\lfloor b/8 \rfloor$, reset $avail$ to 0
      if $w \neq 0$ **then**
        $\ell \leftarrow$ `lzcnt(`w`)`
        $k += \ell$
        $a \leftarrow \ell + 1$
        $w \ll= a$
        $avail -= a$
        $b += a$
        break
      $k += avail$
      $b += avail$
  $mag \leftarrow (k \ll 7) \mid low$
  if $b > L$ or $mag \geq q$ **then reject**
  `tmp`$[i] \leftarrow sign\,?\,{-}mag : mag$
for $i = 0, \dots, n-1$ **step** 16 **do**
  $\mathbf{v} \leftarrow$ `vmovdqu32(tmp+`i`)`
  $\mathbf{m} \leftarrow$ `vpsrad(`$\mathbf{v}$`, 31)` ▷ 0 if ≥ 0, -1 if < 0
  `vmovdqu32(out+`i`, vpaddd(`$\mathbf{v}$`, vpandd(`q`, `$\mathbf{m}$`)))`